

Formal Dynamic Semantics of AVA 95

Michael K. Smith

Technical Report: 112

September 1995

Computational Logic Inc.
1717 W. 6th St. Suite 290
Austin, Texas 78703

(512) 322-9951

This work was sponsored in part by the Defense Advanced Research Projects Agency, ARPA Orders 6082 and 9151. The views, conclusions and modifications contained in this document are those of the authors and should not be interpreted as representing the official policies either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Chapter 1

INTRODUCTION

This report contains the operational semantics for the AVA subset of Ada 95. The informal description of this subset can be found in the Reference Manual [Smith 95].

1.1 Notation

Here is a summary of the syntax used in this document in terms of the official syntax of the Acl2 logic.

1. Variables. x, y, z , etc. are printed in italics.
2. Function application. For any function symbol for which special syntax is not given below, an application of the symbol is printed with the usual notation; e.g., the term $(fn\ x\ y\ z)$ is printed as $fn(x, y, z)$. Note that the function symbol is printed in Roman. In the special case that 'c' is a function symbol of no arguments, i.e., it is a constant, the term (c) is printed merely as c , in small caps, with no trailing parentheses. Because variables are printed in italics, there is no confusion between the printing of variables and constants.
3. Other constants. **t**, **f**, and **nil** are printed in bold. Quoted constants are printed in the ordinary syntax of the ACL2 logic, in a 'typewriter font.' For example, '(**a b c**)' is still printed just that way. **#b001** is printed as 001_2 , **#o765** is printed as 765_8 , and **#xa9** is printed as $a9_{16}$, representing binary, octal and hexadecimal, respectively.
4. $(if\ x\ y\ z)$ is printed as
if x then y else z fi.
5. $(cond\ (test1\ value1)\ (test2\ value2)\ (t\ value3))$ is printed as
if $test1$ then $value1$ elseif $test2$ then $value2$ else $value3$ fi.
6. $(case\ x\ (key1\ answer1)\ (key2\ answer2)\ (otherwise\ default))$ is printed as
case on x : case = $key1$ then $answer1$ case = $key2$ then $answer2$ otherwise $default$ endcase.
7. $(let\ ((var1\ val1)\ (var2\ val2))\ form)$ is printed as
let $var1$ be $val1$, $var2$ be $val2$ in $form$.
8. $(let*\ ((var1\ val1)\ (var2\ val2))\ form)$ is printed as
let* $var1$ be $val1$, $var2$ be $val2$ in $form$.
9. $(forall\ (x\ y)\ (p\ x))$ is printed as
 $\forall x, y: p(x)$.

10. `(exists (x y) (p x))` is printed as

$\exists x, y: p(x).$

11. `(not x)` is printed as

$\neg x.$

12. The remaining symbols that are printed specially are described in the following table.

ACL2 Syntax	Conventional Syntax
t	t
f	f
nil	nil
(lt x y)	$x <_n y$
(le x y)	$x \leq_n y$
(gt x y)	$x >_n y$
(ge x y)	$x \geq_n y$
(union-theories x y)	$x \cup y$
(set-difference-theories x y)	$x \text{ less } y$
(intersection-theories x y)	$x \cap y$
(congruent x y)	$x \equiv y$
(or x y)	$x \vee y$
(and x y)	$x \wedge y$
(* x y)	$x \times y$
(- x y)	$x - y$
(+ x y)	$x + y$
(union x y)	$x \cup y$
(remainder x y)	$x \bmod y$
(/ x y)	x / y
(iff x y)	$x \leftrightarrow y$
(implies x y)	$x \rightarrow y$
(append x y)	$x @ y$
(member x y)	$x \in y$
(>= x y)	$x \geq y$
(> x y)	$x > y$
(<= x y)	$x \leq y$
(< x y)	$x < y$
(lessp x y)	$x < y$
(greaterp x y)	$x > y$
(geq x y)	$x \geq y$
(leq x y)	$x \leq y$
(e0-ord-< x y)	$x <_{\epsilon} y$

ACL2 Syntax	Conventional Syntax
(equal x y)	$x = y$
(= x y)	$x =_n y$
(eql x y)	$x =_a y$
(eq x y)	$x =_{eq} y$
(not (member x y))	$x \notin y$
(not (equal x y))	$x \neq y$
(not (= x y))	$x \neq_n y$
(not (eql x y))	$x \neq_a y$
(not (eq x y))	$x \neq_{eq} y$
(minus x)	$-x$
(1+ x)	$1+x$
(zerop x)	$x \cong 0$
(numberp x)	$x \in \mathbf{N}$
(1- x)	$x-1$
(not (numberp x))	$x \notin \mathbf{N}$
(top-as x)	$x_{a[1]}$
(top-vs x)	$x_{v[1]}$
(top-es x)	$x_{e[1]}$
(clock x)	x_t
(as x)	x_a
(vs x)	x_v
(es x)	x_e
(length x)	$ x $
(len x)	$ x $
(abs x)	$ x $

A superscript "*" indicates repetition and may be used to indicate a list of components, $decl^*$, or a function that acts on a list of arguments, $I_e^*(l, env)$

Multiple values may be set or returned. We use $\langle x, \dots, z \rangle$ to indicate such cases.

Lists are composed using square brackets, e.g. $[x, 1, [y, z]]$.

Lists are composed using square brackets, e.g. $[x, 1, [y, z]]$. Literal symbols and lists are quoted with "'", e.g. $'*constraint-error*'$, $'[a, b, c]'$.

Most semantic operations have two components, an exception check and a modification to the environment. In the ACL2 logic we capture this notion using functions that return multiple values. We present a sequence of such forms by an ‘ilet’ indicated by { *form** }. An ‘ilet’ returns two values, an exception and an environment.

An ‘ilet’ has a procedural flavor, even though an *n* element ‘ilet’ simply expands into an *n* element-deep nested structure of **let**’s. The ‘ilet’ subforms are required to return a pair of values consisting of an exception and a new environment, which are bound to the variables, *exc* and *env*, respectively. The only exception to this is the ‘:=’ operation.

```

‘ilet’ == { form* }

form == ⟨a, b⟩ |
    exception(pred, [fail]) |
    check-assert(pred, [out], [in]) |
    check-asserts([out], [in]) |
    args := form; |
    expr
args == symbol | ⟨symbol*⟩

```

In an ‘ilet’ context, these forms are interpreted as follows.

A simple *expr* must return two values which are bound to ⟨*env*, *exc*⟩ before the next form is interpreted.

exception(*form*, *fail*) evaluates *form*. If not **t** in the current *env*, then we continue, otherwise we exit the ‘ilet’ with the multiple value, *fail*. *Fail* defaults to ⟨*form*, *env*⟩.

check-assert(*form*, [*instate*], [*outstate*]) interprets *form* with respect to the *instate* and *outstate* value stacks. If **t**, then we continue, otherwise we exit the ‘ilet’ with the exception, **logical-error**.

check-asserts([*instate*], [*outstate*]) checks the elements of the current assertion stack with respect to the *instate* and *outstate* value stacks. If all are **t** then we continue, otherwise we exit the ‘ilet’ with the exception, **logical-error**.

assert1(*form*) adds *form* to the assertion stack.

asserts(*forms*) appends *forms* to the assertion stack.

a := *form* binds *a* to the value returned by *form*.

⟨*a*, ... *z*⟩ := *form* does a multiple-value bind of ⟨*a*, ... *z*⟩ to the values returned by *form*.

If after the evaluation of any element of { *a* ... *b* }, the variable *exc* becomes non-false, we return ⟨*exc*, *env*⟩. For this reason you will sometimes see the exception component of a form bound to *exc2* so that we can handle it explicitly in the semantics.

1.2 Notes on the Implementation

Mutual recursion is difficult to reason about. So, we avoid it whenever possible.

The environment, *env*, consists of three stacks of stacks: the entry stack, the value stack, and the annotation stack.

The entry stack holds declarations (of objects, subprograms and packages). The value stack holds the results of expression evaluation. The annotation stack is intended to handle the accumulated requirements due to transition and invariant assertions.

Structure of the environment:

<i>env</i>	= [entry-stack value-stack assertion-stack]
entry-stack	= nil [local-estack . entry-stack]
local-estack	= nil [entry . local-estack]
value-stack	= nil [local-vstack . value-stack]
local-vstack	= nil [<i>value</i> . local-vstack]
assertion-stack	= nil [local-astack . assertion-stack]
local-astack	= nil [<i>lexpr</i> . local-astack]
entry	= [<i>id</i> ttype <i>value</i>] [<i>id decl</i>]

Chapter 2

OPERATIONAL DEFINITION

We begin with the top level definition of the interpreter.

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *ava-dynamic*.

INCLUDING the book: *predefined-packages*.

In 'interpret-eval' *form* is the form to be interpreted, with *flg* indicating what type of form it is. We use a single large recursive function because mutual recursion is difficult to reason about in ACL2.

flg = 'DECL for a declaration,
flg = 'DECLS for a list of declarations,
flg = 'STMT for a single statement,
flg = 'STMTS for a list of statements,
flg = 'EXP for an expression,
flg = 'EXPS for a list of expressions,

The *env* is composed of three parts:

- env_v is a stack of value stacks.
- env_a is a stack of annotation stacks.
- env_e is a stack of entry stacks, which includes at the top level the various predefined procedures and types.

The clock variable, *c*, in the interpreter is the maximum stack depth of subprogram calls. In addition to normal procedure and function recursion, each cycle of a loop is counted as a subprogram call.

DEFINITION:

$I_{flg}(form, env, c)$

=

if \neg pos-int-p(*c*) **then** hard-error-env(*env*, "Out of time", *c*)

elseif env-p(*env*)

then case on *flg*:

case = *stmts* **then**

if consp(*form*)

then { $I_s(car(form), env, c)$

$I_s^*(cdr(form), env, c)$ }

else $\langle nil, env \rangle$

fi

case = *exps* **then**


```

if consp (form)
  then {  $I_e$  (car (form), env, c)
           $I_e^*$  (cdr (form), env, c) }
  else ⟨nil, env⟩
fi
case = decls then
  if consp (form)
    then {  $I_d$  (car (form), env, c)
             $I_d^*$  (cdr (form), env, c) }
    else ⟨nil, env⟩
  fi
case = stmt then
  if consp (form) ∧ top-prefix-p (form, nil)
    then case on statement-opr (form):
      case = sl then  $I_s^*$  (arg* (form), env, c)
      case = constrained-st then
        { pre := env;
           $I_s$  (constrained-st-stmt (form), push-as ([constrained-st-relation (form)],
                                                    env), c)
          ⟨exc, pop-as (env)⟩
          check-asserts (enve, pree)
          }
      case = null then ⟨nil, env⟩
      case = assign then
        let* var be assign-var (form),
              value be assign-value (form),
              root be root* (var)
          in
        { pre := env;
          ⟨exc, env1⟩ :=  $I_e^*$  (actions (var), push-env, c - 1);
          actions := env1v[1];
          env2 := env;
           $I_e$  (var, env, c)
           $I_e$  (value, push-vs (nil, env2), c)
          value := top-value (env);
          env := pop-vs (env);
          assign-to-env (root, actions, value, env)
          check-asserts (enve, pree)
          }
      case = proc-call then
        let* proc-name be proc-call-id (form),
              proc be proc-lookup (proc-name, env),
              formals be arg* (procedure-params (proc)),
              actuals be arg* (proc-call-actuals (form)),
              spec be procedure-spec (proc)
                  ∨ ' (true)
          in
        { pre := env;
           $I_e^*$  (actuals, push-env (env), c)
           $I_d^*$  (reverse (formals), env, c - 1)
          instate := env;
          ⟨exc2, env⟩ :=  $I_s$  (procedure-body (proc), env, c - 1);
          exception (null (exc2),
                   hard-error-env (env,

```

```

                                "Procedure exit with NULL exc")
exception (exc2 ≠ *subprogram-return*,
          ⟨exc, pop-env (env)⟩)
exc := nil;
outstate := env;
check-assert (spec, outstate, instate)
Ie* (extract-ids (formals), env, c - 1)
values := reverse (envv[1]);
env3 := pop-env (env);
⟨exc2, env⟩ := Is* (assign-actuals (formals,
                                actuals,
                                values), pop-env (env), c - 1);
exception (exc2, ⟨exc2, env3⟩)
check-asserts (enve, pree)
}
case = return then
  if return-value (form)
  then { Ie (return-value (form), env, c)
        ⟨*subprogram-return*, env⟩ }
  else ⟨*subprogram-return*, env⟩
  fi
case = exit then ⟨*loop-exit*, env⟩
case = raise then ⟨*program-error*, env⟩
case = if-stmt then
  let ifarms be arg* (form)
  in
  if atom (ifarms) then ⟨nil, env⟩
  else let ifarm be car (ifarms)
  in
  { pre := env;
    Ie (ifarm-test (ifarm), env, c)
    if top-value (env)
      = true
    then Is (ifarm-statements (ifarm), pop-value (env), c)
    else Is (mk-if-stmt (cdr (ifarms)), pop-value (env), c)
    fi
    check-asserts (enve, pree)
  }
  fi
case = while-loop then
  { pre := env;
    Ie (while-loop-test (form), env, c)
    exception (top-value (env) = false,
              ⟨nil, pop-value (env)⟩)
    env := pop-value (env);
    ⟨exc2, env⟩ := Is* (arg* (while-loop-statements (form)), env, c);
    exception (exc2 = *loop-exit*, ⟨nil, env⟩)
    exception (exc2, ⟨exc2, env⟩)
    check-asserts (enve, pree)
    Is (form, env, c - 1)
  }
case = block then
  { pre := env;
    ⟨nil, push-es (nil, push-as (nil, env))⟩
    ⟨exc2, env⟩ := Id* (arg* (block-decls (form)), env, c);
  }

```

```

exception (exc2, ⟨exc2, pre⟩)
⟨exc2, env⟩ := Is* (arg* (block-body (form)), env, c);
exception (
  exc2
  ^ block-handler (form)
  ^ handler-error-p (exc2),
  Is* (arg* (block-handler (form)), env, c))
exception (exc2, ⟨exc2, pop-es (pop-as (env))⟩)
⟨nil, pop-es (pop-as (env))⟩
check-asserts (enve, pree)
}
otherwise hard-error-env (env,
  "Undefined statement type!",
  form)
endcase
else hard-error-env (env,
  "Unexpected atomic statement!",
  form)
fi
case = exp then
if literal-p (form) then ⟨nil, push-value (form, env)⟩
elseif id-p (form)
then { value := entry-value (variable-lookup (form, env));
  exception (¬ value,
    ⟨hard-error2 (nil,
      "Unbound variable ~p0",
      form), env⟩)
  ⟨nil, push-value (value, env)⟩ }
elseif indexed-component-p (form)
then { ⟨nil, push-env (env)⟩
  Ie (indexed-component-root (form), env, c)
  Ie (indexed-component-index (form), env, c)
  exception (get-array-elem-exc (nth-vse (1, env),
    nth-vse (0, env)))
  ⟨nil, push-value (get-t (nth-vse (1, env), nth-vse (0, env)),
    pop-env (env))⟩ }
elseif selected-component-p (form)
then { ⟨nil, push-env (env)⟩
  Ie (selected-component-root (form), env, c)
  ⟨nil, push-value (get-t (nth-vse (0, env),
    selected-component-field (form)),
    pop-env (env))⟩ }
elseif aggregate-p (form)
then hard-error-env (env,
  "Unqualified aggregate",
  form)
elseif aggregate-choice-p (form)
then hard-error-env (env,
  "Unqualified choice aggregate",
  form)
elseif aggregate-pos-p (form)
then Ie (aggregate-pos-value (form), env, c)
elseif qualified-p (form)
then if aggregate-p (qualified-value (form))
  then let agg be qualified-value (form),
    typ be qualified-type (form)
  in
  if pos-aggregate-p (agg)

```

```

then { <nil, push-env (env)>
  Ie* (extract-agg-values (arg* (agg)), env, c)
  labels := type-tree-labels (typ, nil);
  <nil, push-value (pairlis$ (labels,
                             reverse (envv[1])),
                             pop-env (env))> }
elseif choice-aggregate-p (agg)
then { <nil, push-env (env)>
  Ie* (extract-agg-values (arg* (agg)), env, c)
  labels := type-tree-labels (typ,
                             extract-agg-labels (arg* (agg)));
  <nil, push-value (pairlis$ (labels,
                             reverse (envv[1])),
                             pop-env (env))> }
else hard-error-env (env,
                      "Aggregate must be uniform",
                      form)
fi
else { Ie (qualified-value (form), env, c)
  exception (coerce-to-subtype-exc (top-value (env),
                                   qualified-type (form),
                                   nil))
  <nil, push-value (coerce-to-subtype-val (top-value (env),
                                         qualified-type (form)),
                  pop-value (env))> }
fi
elseif type-convert-p (form)
then { exception (type-convert-exc (type-convert-value (form),
                                                         type-convert-type (form)))
  <nil, push-value (type-convert-val (type-convert-value (form),
                                     type-convert-type (form)),
                  env)> }
elseif op-expr-p (form)
then let opr-name be id-root (op-expr-id (form)),
  operands be arg* (op-expr-actuals (form))
  in
  if opr-name =eq 'if
  then { <exc, env2> := Ie (operands0, push-vs (nil, env), c);
  if top-value (env2) = true
  then Ie (operands1, env, c)
  else Ie (operands2, env, c)
  fi }
  else { <exc, env2> := Ie* (operands, push-vs (nil, env), c);
  exception (eval-opr-exc (opr-name,
                          reverse (env2v[1])))
  env := push-value (eval-opr-val (opr-name,
                                  reverse (env2v[1])),
                    env); }
fi
elseif function-call-p (form)
then let* func-name be function-call-id (form),
  func be func-lookup (func-name, env),
  formals be arg* (function-params (func)),
  actuals be arg* (function-call-actuals (form)),
  spec be function-spec (func) ∨ ' (true),
  calling-env be env

```

```

    in
  { Ie*(actuals, push-env (env), c)
    Id*(reverse (formals), env, c - 1)
    instate := env;
    <exc2, env> := Is(function-body (func), env, c - 1);
    exception (null (exc2),
              hard-error-env (env,
                              "Function exit with NULL exc"))
    exception (exc2 ≠ *subprogram-return*,
              <exc2, calling-env>)
    outstate := env;
    check-assert (spec, instate, outstate)
    <nil, push-value (top-value (env), calling-env)>
  }
  else hard-error-env (env,
                      "Undefined expression type!",
                      form)
fi
case = decl then
  if consp (form) ∧ top-prefix-p (form, nil)
  then case on statement-opr (form):
    case = fp-spec then
      { pre := env;
        exc := coerce-to-subtype-exc (top-value (env),
                                     fp-spec-type (form),
                                     nil);
        <nil, push-ese (make-constrained-entry (fp-spec-id (form),
                                             constrain-range-if-necessary (
                                               fp-spec-type (form),
                                               top-value (env)),
                                             coerce-to-subtype-val (
                                               top-value (env),
                                               fp-spec-type (form))),
                                             pop-value (env)))>
        check-asserts (enve, pree)
      }
    case = assert1 then check-assert (arg1 (form))
    case = invariant then
      { check-assert (arg1 (form))
        <nil, push-ase (arg1 (form), env)>
      }
    case = object-decl then
      { pre := env;
        Ie(object-decl-body (form), env, c)
        exc := coerce-to-subtype-exc (top-value (env),
                                     object-decl-type (form),
                                     nil);
        <nil, push-ese (make-constrained-entry (object-decl-id (form),
                                             constrain-range-if-necessary (
                                               object-decl-type (form),
                                               top-value (env)),
                                             coerce-to-subtype-val (
                                               top-value (env),
                                               object-decl-type (form))),
                                             pop-value (env)))>
        check-asserts (enve, pree)
      }

```


fi

DEFINITION:

package-up (*comp-units*, *decls*)

=

if null (*comp-units*) **then** nil**elseif** \neg comp-unit-p (car (*comp-units*)) **then** nil**else** let *unit* be car (*comp-units*)**in****case on** car (arg1 (*unit*)):**case** = package-decl **then** package-up (cdr (*comp-units*),
cons (*unit*,
decls))**case** = package-body **then**cons (mk-comp-unit (merge-package (arg1 (*unit*),
cdr (*comp-units*)
@ decls),
arg2 (*unit*)),
package-up (cdr (*comp-units*), decls))**case** = procedure **then** cons (*unit*,
package-up (cdr (*comp-units*),
decls))**otherwise** nil**endcase****fi**

DEFINITION:

interpret-program (*main*, *library*)

=

if top-prefix-p (mk-compilation (*library*), nil)**then** $\langle exc, env \rangle := I_d^X$ (package-up (*library*, nil) @ *annex-a*, *initial-env*, *some-real-big-integer*);**if** null (*exc*)**then** I_s (['proc-call, *main*, nil], *env*, *some-real-big-integer*)**else** hard-error-env (*env*, "Initial library elaboration failure")**fi****else** hard-error-env (nil, "Library not a compilation")**fi**

2.1 Subsidiary Routines

SET CURRENT PACKAGE to be **ACL2**.INCLUDING the book: *macros*.INCLUDING the book: *ilet*.INCLUDING the book: *get-tree*.INCLUDING the book: *insert-sort*.INCLUDING the book: *subprefix-norm*.INCLUDING the book: *type-check-macros*.

Entity stack

DEFINITION:

es-p (*l*)

=

if \neg consp(l) **then** null(l)
else entry-p(car(l)) \wedge es-p(cdr(l))
fi

Value stack

DEFINITION:

vs-p(l)

=

if \neg consp(l) **then** null(l)
else literal-p(car(l)) \wedge vs-p(cdr(l))
fi

Assertion stack

DEFINITION:

as-p(l)

=

if \neg consp(l) **then** null(l)
else lexpr-p(car(l)) \wedge as-p(cdr(l))
fi

Stack of entity stack

DEFINITION:

es-p*(l)

=

if \neg consp(l) **then** null(l)
else es-p(car(l)) \wedge es-p*(cdr(l))
fi

Stack of value stack

DEFINITION:

vs-p*(l)

=

if \neg consp(l) **then** null(l)
else vs-p(car(l)) \wedge vs-p*(cdr(l))
fi

Stack of assertion stack

DEFINITION:

as-p*(l)

=

if \neg consp(l) **then** null(l)
else as-p(car(l)) \wedge as-p*(cdr(l))
fi

DEFINITION:

env-p(e)

=

true-listp(e) \wedge (($|e|$) = _{n} 3) \wedge es-p*(e_e) \wedge vs-p*(e_v) \wedge as-p*(e_a)

2.1.1 Constants

We would prefer to encapsulate ‘ava-min-int’ and ‘ava-max-int’ without providing a definition in order to reason about their effects generically. E.g

BEGIN ENCAPSULATION

CONSTRAIN the functions:

FUNCTION: ava-min-int

ACCORDING TO THE FOLLOWING EVENTS:

LOCAL DEFINITION:

ava-min-int = -32000

END ENCAPSULATION.

But if we do so, the interpreter is not executable.

So, in order to execute functions that depend on the integer bounds, we use the following, which gives ‘ava-min-int’ and ‘ava-max-int’ fixed values.

DEFINITION:

ava-min-int = -32000

DEFINITION:

ava-max-int = 32000

THEOREM: integerp-ava-min-int

integerp (ava-min-int) \wedge (ava-min-int < 0)

:type-prescription

THEOREM: integerp-ava-max-int

integerp (ava-max-int) \wedge (0 < ava-max-int)

:type-prescription

THEOREM: min-int-close-to-max-int

(ava-max-int + ava-min-int) \leq 1

MODIFY the current theory: Disable ‘ava-min-int’, ‘ava-max-int’, and their executable counterparts

TESTS and EXTRACTION

Basic extraction functions on prefix forms return the operator name, opr (*form*), a list of the arguments, arg* (*form*), and specific arguments, arg1 (*form*)..argn (*form*). Others extraction functions are defined according to the abstract syntax defined in "subprefix-norm.input". That file is used to generate the functions in "subprefix-norm.lisp", which are defined in .

2.1.2 Errors

Hard errors are errors in the semantics and always percolate all the way to the top. If we do everything right (and only pass in legal statically-checked programs), then we should never see a hard error. Someday we may want to prove that there is no hard error when interpreting a well-formed AVA program.

The simple AVA predefined exceptions are simply constants.

CONSTANT:

```
*program-error* = 'program-error
```

```
CONSTANT:
```

```
*constraint-error* = 'constraint-error
```

```
CONSTANT:
```

```
*storage-error* = 'storage-error
```

Defined only in AVA for purposes of tracking assertions:

```
CONSTANT:
```

```
*logical-error* = 'logical-error
```

The following are used to control the flow of execution.

```
CONSTANT:
```

```
*loop-exit* = 'loop-exit
```

```
CONSTANT:
```

```
*subprogram-return* = 'subprogram-return
```

```
MACRO:
```

```
logical-error(&REST args)
```

```
=
```

```
if args
```

```
  then `(list *logical-error*
             ,(car args)
             (list ,@(cdr args)))
```

```
  else `(list *logical-error*
             "Failed annotation")
```

```
fi
```

Added *logical expressions*, *lexprs*, which are just expressions in the ACL2 logic.

```
instate(lexpr)
```

```
outstate(lexpr)
```

qinstate and 'outstate' are macros that expand into

```
let env be x
```

```
  in
```

```
eval(lexpr)
```

, where *x* is the input or output state.

AVA predefined exception, `constraint_error`

Note that `constraint-error` exceptions have a little more structure than the others. This enables us to pass some debugging information out with what would otherwise be an uninformative exception. The semantics does not distinguish these otherwise.

```
DEFINITION:
```

```
handler-error-p(exc)
```

```
=
```

```
(listp(exc)  $\wedge$  (car(exc) = *constraint-error*))
```

```
 $\vee$  (exc = *program-error*)
```

```
 $\vee$  (exc = *storage-error*)
```

```
MACRO:
```

```

constraint-error(fmt-string, &REST args)
=
\ (list *constraint-error*
   ,fmt-string
   (list ,@args))

```

DEFINITION:

```

extend-constraint-error(exc, new-args)
=
if true-listp(exc)
  ^ (car(exc) =eq *constraint-error*)
  ^ stringp(cadr(exc)) then [car(exc), cadr(exc), caddr(exc) @ new-args]
else exc
fi

```

2.1.3 Ava Literals, Values and Types

A literal is either an integer, a character, a string, an array literal, or a record literal.

2.1.4 Operators

The elements of **ava-operators-alist** are the predefined operators and their arity.

CONSTANT:

```

*ava-operators-alist* = ' ((unary-plus . 1)
  (plus . 2)
  (minus . 2)
  (multiply . 2)
  (divide . 2)
  (mod . 2)
  (rem . 2)
  (abs . 1)
  (power . 2)
  (equal . 2)
  (ne . 2)
  (in . 2)
  (in-range . 2)
  (in-type . 2)
  (lt . 2)
  (array-< . 2)
  (not . 1)
  (and . 2)
  (or . 2)
  (xor . 2)
  (catenate . 2)
  (array-not . 1)
  (array-and . 2)
  (array-or . 2)
  (if . 3)
)

```

DEFINITION:

```

ava-operators-alist = *ava-operators-alist*

```

Let's state all the facts we'll need about this alist, and then disable it. This is an example of the utility of macros in ACL2.

DEFINITION:

```

ava-op-defthm-forms (op-alist)
=
if op-alist
then cons ( ` (defthm
              , (pack2
                'ava-op-
                (caar op-alist))
              (equal
                (assoc-eq
                  ' , (caar op-alist)
                  (ava-operators-alist))
                  ' , (car op-alist))),
            ava-op-defthm-forms (cdr (op-alist)))
else nil
fi

```

```

MACRO:
prove-ava-op-defthms
=
cons ( 'progn, ava-op-defthm-forms (*ava-operators-alist*))

```

```

EVENT:
(prove-ava-op-defthms)

```

THEOREM: symbol-alistp-ava-operators-alist *:type-prescription*
symbol-alistp (ava-operators-alist) = **t**

MODIFY the current theory:

Disable 'ava-operators-alist' and the executable counterpart of '(ava-operators-alist)'.

2.1.5 Expression evaluation

This is now handled with statement evaluation, by Interpret, which returns two values, an exception (or NIL) and an environment. The result of the evaluation is top-value (env), which is top (top (env_v)). For example,

$I_e(3 + 4, env) \Rightarrow \langle \mathbf{nil}, \text{push-value}(7, env) \rangle$

At one point we checked that all variables are bound, in order to say that we have an expression. But this requirement turned into an analogous requirement for statements, which in turn forced proof obligations that if the handler of a block is a statement-p with respect to a given variable stack, then it's still one even after we interpret the body of that block. Since we probably want to support the notion of unbound anyhow, we just allow variables to be unbound and check things dynamically.

The operator functions defined below all return a first value of nil (normal) unless the arguments require an exception to be raised.

```

DEFINITION:
fix-int (x)
=
if integerp (x) then x
else 0
fi

```

```

DEFINITION:
fix-bool (x)
=

```

```

if  $x$  then ' (true)
else ' (false)
fi

```

DEFINITION:

```
int-not-in-range ( $val$ ,  $lower$ ,  $upper$ )
```

=

```
( $val < lower$ )  $\vee$  ( $upper < val$ )
```

Guard: $\text{rationalp}(val) \wedge \text{rationalp}(lower) \wedge \text{rationalp}(upper)$

Let's define the AVA operators and then disable them all at once.

2.1.6 Numeric Operators

Notice that we don't need to allow different base types of integer.

LABEL: ava-op-fns-start

DEFINITION:

```
ava-plus-exc ( $x$ ,  $y$ )
```

=

```
if int-not-in-machine-range ( $x + y$ )
```

```
  then constraint-error("Integer overflow, (+ ~p0 ~p1).",  $x$ ,  $y$ )
```

```
  else nil
```

```
fi
```

DEFINITION:

```
ava-plus-val ( $x$ ,  $y$ ) =  $x + y$ 
```

DEFINITION:

```
ava-multiply-exc ( $x$ ,  $y$ )
```

=

```
if int-not-in-machine-range ( $x \times y$ )
```

```
  then constraint-error("Integer overflow, (* ~p0 ~p1).",  $x$ ,  $y$ )
```

```
  else nil
```

```
fi
```

DEFINITION:

```
ava-multiply-val ( $x$ ,  $y$ ) =  $x \times y$ 
```

DEFINITION:

```
ava-power-exc ( $x$ ,  $y$ )
```

=

```
if int-not-in-machine-range (expt ( $x$ ,  $y$ ))
```

```
  then constraint-error("Integer overflow, (* ~p0 ~p1).",  $x$ ,  $y$ )
```

```
  elseif  $y < 0$ 
```

```
    then constraint-error("Exponent underflow, (* ~p0 ~p1).",  $x$ ,  $y$ )
```

```
    else nil
```

```
fi
```

DEFINITION:

```
ava-power-val ( $x$ ,  $y$ ) = expt ( $x$ ,  $y$ )
```

Relations between division, mod and rem.

$$a = (a/b)*b + (a \text{ rem } b)$$

$$(a \text{ rem } b) \text{ has sign of } a \text{ and } \text{abs}(a \text{ rem } b) < \text{abs}(b)$$

$$(-a)/b = -(a/b) = a/(-b)$$

It should be a theorem that: $a = ((\text{truncate}(a, b) \times b) + \text{rem}(a, b))$

DEFINITION:
ava-divide-exc (x, y)
=

```

if  $y = 0$ 
  then constraint-error("Division by 0, (/ ~p0 ~p1).",  $x, y$ )
  elseif int-not-in-machine-range(truncate( $x, y$ ))
  then constraint-error("Integer overflow, (/ ~p0 ~p1).",  $x, y$ )
  else nil
fi

```

DEFINITION:
ava-divide-val (x, y) = truncate (x, y)

The possibility of the overflow exception is due to the case (*ava_min_int* rem -1).

DEFINITION:
ava-rem-exc (x, y)
=

```

if  $y = 0$ 
  then constraint-error("Zero divisor, ~p0 rem ~p1.",  $x, y$ )
  elseif int-not-in-machine-range(rem( $x, y$ ))
  then constraint-error("Integer overflow, (~p0 rem ~p1).",  $x, y$ )
  else nil
fi

```

DEFINITION:
ava-rem-val (x, y) = rem (x, y)

DEFINITION:
ava-mod-exc (x, y)
=

```

if  $y = 0$ 
  then constraint-error("Zero divisor, ~p0 mod ~p1.",  $x, y$ )
  elseif int-not-in-machine-range(mod( $x, y$ ))
  then constraint-error("Integer overflow, (~p0 mod ~p1).",  $x, y$ )
  else nil
fi

```

DEFINITION:
ava-mod-val (x, y) = mod (x, y)

DEFINITION:
ava-unary-minus-exc (x)
=

```

if int-not-in-machine-range(- $x$ )
  then constraint-error("Integer overflow, (- ~p0).",  $x$ )
  else nil
fi

```

DEFINITION:
ava-unary-minus-val (x) = - x

2.1.7 Boolean Operators

Important note: Booleans literals in ACL2 are **t** and **nil** (or non-**t**). Booleans literals in AVA are ‘true’ or ‘false’. This somewhat unfortunate circumstance was necessitated by the need to distinguish trees from leaves in array and record literals. E.g. we needed the following theorems:

THEOREM: tree-not-leaf
treep(x) \rightarrow (\neg leafp(x))

THEOREM: leaf-not-tree
 $\text{leafp}(x) \rightarrow (\neg \text{treep}(x))$

Regarding equality:

Array and record values are alists of the form: ((index . value)*). Two such values are equal if corresponding elements of their values are. We use a single equality function rather than generating the numerous type specific versions required by naive adherence to the manual. This makes it much more tractable to provide a library of predefined lemmas for reasoning about equality.

DEFINITION:
 $\text{minimum}(l)$
 $=$
if $\neg \text{consp}(l)$ **then** 0
elseif $\neg \text{rationalp}(\text{car}(l))$ **then** $\text{minimum}(\text{cdr}(l))$
elseif $\neg \text{consp}(\text{cdr}(l))$ **then** $\text{car}(l)$
else $\text{min}(\text{car}(l), \text{minimum}(\text{cdr}(l)))$
fi

DEFINITION:
 $\text{maximum}(l)$
 $=$
if $\neg \text{consp}(l)$ **then** -1
elseif $\neg \text{rationalp}(\text{car}(l))$ **then** $\text{maximum}(\text{cdr}(l))$
elseif $\neg \text{consp}(\text{cdr}(l))$ **then** $\text{car}(l)$
else $\text{max}(\text{car}(l), \text{maximum}(\text{cdr}(l)))$
fi

DEFINITION:
 $\text{array-literal-from}(literal) = \text{minimum}(\text{range}(literal))$

DEFINITION:
 $\text{array-literal-to}(literal) = \text{maximum}(\text{range}(literal))$

Note that the null record and array case are handled, since if x and y aren't equal, they cannot both be non-empty.

DEFINITION:
 $\text{ava-equal-val}(x, y)$
 $=$
if $x = y$ **then** true
elseif $(\neg \text{consp}(x)) \vee (\neg \text{consp}(y))$ **then** false
elseif $\text{array-literal-p}(x) \wedge \text{array-literal-p}(y)$
then $\text{fix-bool}(\text{set-equal}(x, y))$
elseif $\text{record-literal-p}(x) \wedge \text{record-literal-p}(y)$
then $\text{fix-bool}(\text{set-equal}(x, y))$
else false
fi

THEOREM: char-code-nonnegative-integerp-for-ada-char-p
 $\text{ada-char-p}(x) \rightarrow (\text{integerp}(\text{char-code}(x)) \wedge (0 \leq \text{char-code}(x)))$

:type-prescription

MODIFY the current theory:

Disable 'standard-char-p', 'char-code' and 'ada-char-p'.

DEFINITION:
 $x <_n y$
 $=$
if $\text{rationalp}(x)$
then if $\text{rationalp}(y)$ **then** $x < y$

```

    else nil
  fi
else nil
fi

```

DEFINITION:

$$x >_n y = \neg (x <_n y)$$

DEFINITION:

$$x \leq_n y = (x <_n y) \vee (x = y)$$

DEFINITION:

$$x \geq_n y = (x >_n y) \vee (x = y)$$

THEOREM: le-1

$$(x <_n y) \rightarrow (\neg (y \leq_n x))$$

THEOREM: le-2

$$(y \leq_n x) \rightarrow (\neg (x <_n y))$$

THEOREM: le-3

$$x \leq_n x$$

THEOREM: ge-le-eq

$$((x \leq_n y) \wedge (x \geq_n y)) \rightarrow (x = y)$$

DEFINITION:

$$\text{ava-<-val}(x, y)$$

=

```

fix-bool (if ada-char-p (x) & ada-char-p (y)
  then char-code (x) < char-code (y)
  elseif rationalp (x) & rationalp (y) then x <_n y
  else nil
fi)

```

Treat non-rational lower and upper as neg infinity and pos infinity, respectively

DEFINITION:

$$\text{between}(val, lower, upper)$$

=

```

if ¬ rationalp (val) then nil
else (lower ≤_n val) & (val ≤_n upper)
fi

```

DEFINITION:

$$\text{ava-in-range-val}(val, lower, upper)$$

=

```

fix-bool (if ada-char-p (val) & ada-char-p (lower) & ada-char-p (upper)
  then between (char-code (val), char-code (lower), char-code (upper))
  else between (val, lower, upper)
fi)

```

2.1.8 Unary Numeric Operators

DEFINITION:

$$\text{ava-abs-exc}(x)$$

=

```

let y be |ifix (x)|
in
if int-not-in-machine-range (y)
then constraint-error ("Integer overflow, (abs ~p0).", x)

```



```
else nil
fi
```

```
DEFINITION:
ava-abs-val (x) =|ifix (x)|
```

```
DEFINITION:
ava-not (x)
=
if true-p (x) then false
else true
fi
```

```
DEFINITION:
ava-and (x, y)
=
if true-p (x) then y
else false
fi
```

```
DEFINITION:
ava-or (x, y)
=
if true-p (x) then true
else y
fi
```

```
DEFINITION:
ava-xor (x, y)
=
if true-p (x)
then if false-p (y) then true
else false
fi
elseif true-p (y) then true
else false
fi
```

DEFINE the theory **ava-op-fns** to be

the current function theory *less* the function theory **ava-op-fns-start**.

MODIFY the current theory:

Disable ‘ava-op-fns’.

2.1.9 Application of Operators to Eevald Arguments

This function defines the behavior of the built-in functions. It is used in ‘interpret-eval’. Note that the arguments have already been evaluated without an exception. This evaluation is of course done in ‘interpret-eval’.

```
DEFINITION:
eval-opr-val (opr, args)
=
case on opr:
case = plus then ava-plus-val (args0, args1)
case = unary-minus then ava-unary-minus-val (args0)
case = minus then ava-plus-val (args0, - args1)
case = multiply then ava-multiply-val (args0, args1)
```

```

case = divide then ava-divide-val(args0, args1)
case = mod then  ava-mod-val(args0, args1)
case = rem then  ava-rem-val(args0, args1)
case = abs then  ava-abs-val(args0)
case = power then  ava-power-val(args0, args1)
case = equal then  ava-equal-val(args0, args1)
case = ne then  ava-not(ava-equal-val(args0, args1))
case = in-range then  ava-in-range-val(args0, args1, args2)
case = lt then  ava-<-val(args0, args1)
case = gt then  ava-<-val(args1, args0)
case = le then
  if false-p(ava-<-val(args0, args1))
  then  ava-equal-val(args0, args1)
  else true
fi
case = ge then
  if false-p(ava-<-val(args1, args0))
  then  ava-equal-val(args0, args1)
  else true
fi
case = not then  ava-not(args0)
case = and then  ava-and(args0, args1)
case = or then  ava-or(args0, args1)
case = xor then  ava-xor(args0, args1)
otherwise nil
endcase

```

VERIFY GUARDS for ‘top-prefix-p’

DEFINITION:

eval-opr-exc(*opr*, *args*)

=

```

case on opr:
  case = plus then  ava-plus-exc(args0, args1)
  case = unary-minus then  ava-unary-minus-exc(args0)
  case = minus then  ava-plus-exc(args0, -args1)
  case = multiply then  ava-multiply-exc(args0, args1)
  case = divide then  ava-divide-exc(args0, args1)
  case = mod then  ava-mod-exc(args0, args1)
  case = rem then  ava-rem-exc(args0, args1)
  case = abs then  ava-abs-exc(args0)
  case = power then  ava-power-exc(args0, args1)
  otherwise nil
endcase

```

MODIFY the current theory:

Disable ‘eval-opr-val’ and ‘eval-opr-exc’.

2.2 Arrays and Records

If the constraint isn't 'range-p', then it's 'unconstrained-p'.

```
DEFINITION:
satisfies-range-constraint (n, constraint)
=
if range-p (constraint)
  then between (n, range-from (constraint), range-to (constraint))
  else unconstrained-p (constraint)
fi
```

Let's save a lot of case splits.

MODIFY the current theory:

Disable 'nth'.

```
DEFINITION:
range-size (from, to)
=
if  $from \leq_n to$  then 1 + (to - from)
  else 0
fi
```

2.3 Types, including Conversion and Qualification

For the coercion functions below:

```
literal = empty | boolean-literal | numeric-literal | ada-char | array-literal | record-literal
type    = record-type | array-type | predefined-type | range
```

We assume when this is called that the base type of the literal equals the base type of the type. By "base type" we mean integer for range types, the corresponding unconstrained array type for array types, and the type itself otherwise.

```
DEFINITION:
adjust-array-literal (literal, delta)
=
if  $\neg$  consp (literal) then literal
  elseif  $\neg$  (consp (car (literal))  $\wedge$  integerp (caar (literal)))
  then literal
  else cons (cons (caar (literal) + delta, cdar (literal)),
             adjust-array-literal (cdr (literal), delta))
fi
```

```
DEFINITION:
constrain-array-val (literal, from)
=
let old be array-literal-from (literal)
  in
if old = from then literal
  else adjust-array-literal (literal, from - old)
fi
```

If \neg *strong-flg* just compare length, otherwise also check the equality of lower bounds.

```
DEFINITION:
constrain-array-exc (literal, from, to, strong-flg)
```

```

=
if (literal) = range-size (from, to)
  then if strong-flg
    then if from = array-literal-from (literal) then nil
      else constraint-error ("Literal ~p0 does not have base of ~p1.", literal)
    fi
  else nil
fi
else constraint-error ("Length of literal ~p0 is not equal to ~p1.",
  literal,
  range-size (from, to))
fi

DEFINITION:
coerce-to-subtype-from-lit-val (literal, old-lit)
=
if array-literal-p (literal)
  then constrain-array-val (literal, array-literal-from (old-lit))
  else literal
fi

DEFINITION:
coerce-to-subtype-from-lit-exc (literal, old-lit, typ)
=
if array-literal-p (literal)
  then constrain-array-exc (literal,
    array-literal-from (old-lit),
    array-literal-to (old-lit),
    nil)
  elseif range-p (typ)
  then if satisfies-range-constraint (literal, typ) then nil
    else constraint-error ("Literal ~p0 is out of range for type ~p1.", literal, typ)
  fi
  else nil
fi

DEFINITION:
coerce-to-subtype-val (literal, typ)
=
if array-type-p (typ)
  then let constraint be array-type-index (typ)
    in
    if range-p (constraint)
      then constrain-array-val (literal, range-from (constraint))
      else literal
    fi
  else literal
fi

DEFINITION:
coerce-to-subtype-exc (literal, typ, strong-flg)
=
if array-type-p (typ)
  then let constraint be array-type-index (typ)
    in
    if range-p (constraint)
      then constrain-array-exc (literal,
        range-from (constraint),
        range-to (constraint),
        strong-flg)
    else literal
  fi

```

```

    else nil
  fi
elseif range-p (typ)
then if satisfies-range-constraint (literal, typ) then nil
    else constraint-error ("Literal ~p0 is out of range for type ~p1.", literal, typ)
    fi
else nil
fi

```

Type-convert converts an expression to a base type, then checks that it satisfies subtype requirements.

Conversion checks:

- Numeric
 1. Base check is NOOP. Check range.
- Array
 1. Same dimensions.
 2. Index types the same or convertible.
 3. Component types the same.
 4. Constraints on component types the same.
 5. If type is unconstrained then bounds come from converting indices to base type of unconstrained index type. (Which is a noop, since the base type must be INTEGER.)

Raise constraint-error if numeric conversions fail to satisfy constraint.

DEFINITION:

```

type-convert-val (expr, typ)
=
if array-type-p (typ)
  ^ array-literal-p (expr)
  ^ range-p (array-type-index (typ))
then constrain-array-val (expr, range-from (array-type-index (typ)))
else expr
fi

```

Note that *expr* cannot be an aggregate or a string literal.

DEFINITION:

```

type-convert-exc (expr, typ)
=
if array-type-p (typ)
then let constraint be array-type-index (typ)
    in
    if array-literal-p (expr) ^ range-p (constraint)
    then if (expr)
        = range-size (range-from (constraint),
                      range-to (constraint)) then nil
    else constraint-error (
        "Convert: Range of literal ~p0 doesn't satisfy constraint ~p1.",
        expr,
        constraint)
    fi
else nil
fi
elseif range-p (typ)

```

```

then if satisfies-range-constraint(expr, typ) then nil
      else constraint-error("Convert: Value ~p0 does not satisfy subtype range ~p1.",
                           expr,
                           typ)
      fi
else nil
fi

```

2.4 More Expression Evaluation

```

array-literal := [[i . val] ...]
record-literal := [[f . val] ...]

```

```

array-literal-p(form) == treep(form) ∧ all-labels-integer(domain(form))
record-literal-p(form) == treep(form) ∧ all-labels-id(domain(form))

```

DEFINITION:

```

get-array-elem-exc(array-literal, int)
=
let from be array-literal-from(array-literal),
    to be array-literal-to(array-literal)
    in
if between(int, from, to) then nil
    else constraint-error("Array index out of bounds, index ~p0.",
                          int)
fi

```

Note that this does not logically guarantee that an element is present in an array-literal. Our predicates for arrays do not guarantee that all values between `array-literal-from(a)` and `array-literal-to(a)` are present in the alist that represents *a*. At the moment this is an implicit assumption that may need to be made explicit.

THEOREM: `get-array-elem-exc-null-array`

```
((¬ consp(x)) ∧ integerp(i)) → get-array-elem-exc(x, i)
```

AXIOM: `array-elem-exists`

```

(  array-literal-p(x)
  ∧ integerp(i)
  ∧ between(i,
            array-literal-from(array-literal),
            array-literal-to(array-literal)))
→ exists-t(x, i)

```

2.5 Statement Evaluation Support

DEFINITION:

```

variable-update-1(var, val, stack)
=
if consp(stack)
  then let entry be car(stack),
          rest be cdr(stack)
          in
          if basic-entry-p(entry) ∧ (entry-name(entry) = var)
            then cons(make-entry(var, entry-decl(entry), val), rest)
            else cons(entry, variable-update-1(var, val, rest))
          fi
  fi

```

else nil
fi

var is an 'id-p'

DEFINITION:

variable-update (*var*, *val*, *var-stacks*)

=

```

if consp (var-stacks)
  then if assoc-equal (var, car (var-stacks))
    then cons (variable-update-1 (var, val, car (var-stacks)),
              cdr (var-stacks))
    else cons (car (var-stacks),
              variable-update (var, val, cdr (var-stacks)))
  fi
else var-stacks
fi

```

2.5.1 Assignment

We are about to define the functions that break apart variable references, e.g. $v[i].j[k] \Rightarrow \langle v, [iv, j, kv] \rangle$ where *iv* and *kv* are the values obtained by evaluating *i* and *k*, respectively.

THEOREM: indexed-component-root-decrease

(consp (cdr (*x*)) \wedge indexed-component-p (*x*))

\rightarrow (acl2-count (indexed-component-root (*x*)) < acl2-count (*x*))

:linear

THEOREM: selected-component-root-decrease

(consp (cdr (*x*)) \wedge selected-component-p (*x*))

\rightarrow (acl2-count (selected-component-root (*x*)) < acl2-count (*x*))

:linear

MODIFY the current theory: Disable 'selected-component-root', 'selected-component-p', 'indexed-component-root' and 'indexed-component-p'.

DEFINITION:

root* (*x*)

=

```

if  $\neg$  (consp (x)  $\wedge$  consp (cdr (x))) then x
  elseif indexed-component-p (x) then root* (indexed-component-root (x))
  elseif selected-component-p (x) then root* (selected-component-root (x))
  else x
fi

```

we assume it's an id-p

DEFINITION:

actions (*x*)

=

```

if  $\neg$  (consp (x)  $\wedge$  consp (cdr (x))) then nil
  elseif indexed-component-p (x)
    then cons (indexed-component-index (x), actions (indexed-component-root (x)))
  elseif selected-component-p (x)
    then cons (selected-component-field (x),
              actions (selected-component-root (x)))
  else nil
fi

```

An exception will be raised if the index is not in range. To simplify, while it should not happen, an exception will also be raised if the variable is not in the value stack.

2.5.2 Procedure Call Support

DEFINITION:
 constrain-range-if-necessary (*typ*, *literal*)
 =
if array-type-p (*typ*) \wedge unconstrained-p (array-type-index (*typ*))
then mk-array-type (mk-range (array-literal-from (*literal*),
 array-literal-to (*literal*)),
 array-type-elements (*typ*))
else *typ*
fi

DEFINITION:
 make-constrained-entry (*id*, *typ*, *val*)
 =
 make-entry (*id*, constrain-range-if-necessary (*typ*, *val*), *val*)

THEOREM: len-0
 ((|x|) = 0) = atom (*x*)

2.6 Interpreter

In our statement interpreter, the clock is decremented just before subprogram calls and while-loop recursions. A clock of 0 means "out of time".

CONSTANT:
 out-of-time-msg = "Out of time!"

MACRO:
 pop-variable-stack (*n*, *env*)
 =
 `(nthcdr ,*n* ,*env*)

MACRO:
 interpret-stmt (*stmt*, *env*, *clock*)
 =
 ['interpret-eval, ' ' *stmt*, *env*, *clock*]

MACRO:
 interpret-stmts (*stmt-list*, *env*, *clock*)
 =
 ['interpret-eval, ' ' *stmts*, *env*, *clock*]

MACRO:
 interpret-exp (*exp*, *env*, *clock*)
 =
 ['interpret-eval, ' ' *exp*, *env*, *clock*]

MACRO:
 interpret-exps (*exp-list*, *env*, *clock*)
 =
 ['interpret-eval, ' ' *exps*, *env*, *clock*]

MACRO:
 interpret-decl (*decl*, *env*, *clock*)
 =
 ['interpret-eval, ' ' *decl*, *env*, *clock*]

MACRO:
 interpret-decls (*decl-list*, *env*, *clock*)

=
 ['interpret-eval, ' 'decls, decl-list, env, clock]

DEFINITION:
 statement-opr (stmt) = car (stmt)

THEOREM: statement-opr-non-nil-implies-consp-stmt
 statement-opr (stmt) \rightarrow consp (stmt)

:forward-chaining

DEFINITION:
 top-prefix-decls-p2 (l)
 =
if \neg consp (l) **then t**
elseif decl-p (car (l)) **then** top-prefix-decls-p2 (cdr (l))
else nil
fi

DEFINITION:
 top-prefix-decls-p (l)
 =
if \neg consp (l) **then t**
elseif top-prefix-decls-p2 (car (l)) **then** top-prefix-decls-p (cdr (l))
else nil
fi

DEFINITION:
 conjunction (l)
 =
if null (l) **then true**
elseif \neg consp (l) **then l**
else cons (' and, l)
fi

DEFINITION:
 extract-agg-values (l)
 =
if atom (l) **then nil**
elseif aggregate-pos-p (car (l))
then cons (aggregate-pos-value (car (l)), extract-agg-values (cdr (l)))
elseif aggregate-choice-p (car (l))
then cons (aggregate-choice-value (car (l)), extract-agg-values (cdr (l)))
else nil
fi

DEFINITION:
 extract-agg-labels1 (l, i)
 =
if atom (l) **then nil**
elseif aggregate-pos-p (car (l))
then cons (i, extract-agg-labels1 (cdr (l), i + 1))
elseif aggregate-choice-p (car (l))
then let choice **be** aggregate-choice-choice (car (l))
in
if others-p (choice) **then** [' others]
elseif id-p (choice)
then cons (id-root (choice),
 extract-agg-labels1 (cdr (l), i + 1))
else nil
fi
else nil
fi

DEFINITION:
 extract-agg-labels (l) = extract-agg-labels1 (l , 0)

DEFINITION:
 all-agg-pos (l)
 =
if atom (l) **then t**
else aggregate-pos-p (car (l)) \wedge all-agg-pos (cdr (l))
fi

DEFINITION:
 all-agg-choice (l)
 =
if atom (l) **then t**
else aggregate-choice-p (car (l)) \wedge all-agg-choice (cdr (l))
fi

DEFINITION:
 pos-aggregate-p (x)
 =
 aggregate-p (x) \wedge all-agg-pos (arg* (x))

DEFINITION:
 choice-aggregate-p (x)
 =
 aggregate-p (x) \wedge all-agg-choice (arg* (x))

Remember the form of ‘array-type-p’:

array-type == **array-type** index : *type-mark*, elements : *type*
type-mark == **type-mark** type : *id*, constraint : *constraint*
constraint == *subtype* | *unconstrained* | *range* | *attribute*

DEFINITION:
 create-inclusive-list ($from$, to)
 =
if (\neg integerp ($from$)) \vee (\neg integerp (to)) **then nil**
elseif $from > to$ **then nil**
elseif $from = to$ **then** [to]
else cons ($from$, create-inclusive-list ($from + 1$, to))
fi
Measure: ifix (max (0, $to - from$))

MODIFY the current theory:

Enable ‘nth’.

DEFINITION:
 compute-type-range-list (x)
 =
if type-mark-p (x) **then** compute-type-range-list (type-mark-constraint (x))
elseif id-p (x) **then** create-inclusive-list (ava-min-int, ava-max-int)
elseif range-p (x) **then** create-inclusive-list (range-from (x), range-to (x))
else nil
fi

DEFINITION:
 field-spec-ids ($fields$)
 =
if atom ($fields$) **then nil**
else cons (field-spec-id (car ($fields$)), field-spec-ids (cdr ($fields$)))

fi

DEFINITION:

```

type-tree-labels (typ, args)
=
if ( $\neg$  atom (args))  $\wedge$  symbolp (car (args)) then args
  elseif record-type-p (typ) then field-spec-ids (arg* (typ))
  elseif array-type-p (typ)
  then compute-type-range-list (array-type-index (typ))
  else nil
fi

```

DEFINITION:

```

extract-ids (formals)
=
if atom (formals) then nil
  else cons (fp-spec-id (car (formals)), extract-ids (cdr (formals)))
fi

```

pre and post are environments.

DEFINITION:

```

assign-actuals (formals, actuals, values)
=
if atom (formals) then nil
  elseif fp-spec-p (car (formals))
     $\wedge$  variable-p (fp-spec-mode (car (formals)))
  then cons (mk-assign (car (actuals), car (values)),
    assign-actuals (cdr (formals), cdr (actuals), cdr (values)))
  else assign-actuals (cdr (formals), cdr (actuals), cdr (values))
fi

```

DEFINITION:

```

extend-env-with-procedure (form, env)
=
push-ese (make-entry (procedure-id (form), form, nil), env)

```

DEFINITION:

```

extend-env-with-package (form, env)
=
push-ese (make-entry (package-id (form), form, nil), env)

```

Measure information for interpret-eval.

DEFINITION:

```

interpret-measure (stmt, clock)
=
cons (1 if pos-int-p (clock) then clock
  else 0
  fi,
  if stmt then acl2-count (stmt)
  else 0
  fi)

```

THEOREM: interpret-measure-facts

```

( pos-int-p (x)
   $\rightarrow$  (interpret-measure (stmt2,  $-1 + x$ )  $<_{\epsilon}$  interpret-measure (stmt1, x)))
 $\wedge$  (stmt2  $\rightarrow$  ( (interpret-measure (stmt1, c)  $<_{\epsilon}$  interpret-measure (stmt2, c))
  = (acl2-count (stmt1)  $<$  acl2-count (stmt2))))
 $\wedge$  e0-ordinalp (interpret-measure (stmt, n))

```

MODIFY the current theory:

Enable 'e0-ord-<'.

MODIFY the current theory:

Disable 'interpret-measure'.

MODIFY the current theory:

Disable 'top-prefix-p'.

THEOREM: count-actions :linear
 $\text{consp}(form) \rightarrow (\text{acl2-count}(\text{actions}(form)) < \text{acl2-count}(form))$

THEOREM: count-actions-cadr :linear
 $\text{consp}(\text{cadr}(form))$
 $\rightarrow (\text{acl2-count}(\text{actions}(\text{cadr}(form))) < \text{acl2-count}(\text{cadr}(form)))$

THEOREM: count-actions-cadr-2 :linear
 $(\text{consp}(form) \wedge \text{consp}(\text{cadr}(form)))$
 $\rightarrow (\text{acl2-count}(\text{actions}(\text{cadr}(form))) < \text{acl2-count}(form))$

THEOREM: acl2-count-extract-agg-values :linear
 $\text{acl2-count}(\text{extract-agg-values}(w)) \leq \text{acl2-count}(w)$

2.7 Input

User input, from parsed AVA files, comes in as follows:

ACL2 constants contain the declarations and bodies of compilation_units. User defined ACL2 functions, axioms and theorems, present in the annotated AVA, have been literally extracted.

CONSTANT:
pattern_scope-decl = ...

Extracted ACL2

DEFINITION:
 $\text{lowerp}(x) = ('a' \leq x) \wedge (x \leq 'z')$

THEOREM: upperp-not-lowerp
 $\text{upperp}(x) \rightarrow (\neg \text{lowerp}(x))$
 ...

CONSTANT:
pattern_scope-body = *pending*

The Library is defined.

CONSTANT:
library = *pending*

The main program is named.

CONSTANT:
main = '(id main 1)

Finally, a theorem is presented that states that there exists a permitted order of elaboration supporting the evaluation of **main**.

THEOREM: order-of-elaboration
some-order-exists(**main**, **library**)

Appendix A

Support

We depend on a number of ACL2 libraries that are part of the ACL2 V1.8 distribution. In particular:

```
public/sets
arithmetic/equalities
arithmetic/inequalities
arithmetic/rationals-with-axioms
```

The formal definition is then further built on the following.

A.1 Macros

INCLUDING the book: `/slocal/src/acl2/v1-8/books/public/sets`.

Note the important difference between the entity stack and the values stack. All entities, including variables will be contained in the entity stack. That is where we go to look up the value of a variable or constant, as well as where we get the body of a procedure or function.

```
MACRO:
mkenv (entities, values, assertions)
=
`(list ,entities ,values ,assertions)
```

An entry can be a constant, variable, type, function, procedure, or package. The types of constants and variables are fully expanded types.

```
MACRO:
make-entry (&REST entry) = `(list ,@entry)
```

```
MACRO:
entry-name (x) = `(car ,x)
```

```
MACRO:
entry-decl (x) = `(nth 1 ,x)
```

```
MACRO:
entry-value (x) = `(nth 2 ,x)
```

Get the stack*.

```
MACRO:
es (env) = `(nth 0 ,env)
```

entry stack

```
MACRO:
vs (env) = `(nth 1 ,env)
```

value stack

```
MACRO:
as (env) = `(nth 2 ,env)
```

assertion stack

Set the stack*.

```
MACRO:
set-es(x, env)
=
` (list
  ,x
  (vs ,env)
  (as ,env))
```

```
MACRO:
set-vs(x, env)
=
` (list
  (es ,env)
  ,x
  (as ,env))
```

```
MACRO:
set-as(x, env)
=
` (list
  (es ,env)
  (vs ,env)
  ,x)
```

Return the top stack of the stack*.

```
MACRO:
top-es(env) = `(car (es ,env))
```

```
MACRO:
top-vs(env) = `(car (vs ,env))
```

```
MACRO:
top-as(env) = `(car (as ,env))
```

Top value stack entry

```
MACRO:
top-vse(env) = `(car (top-vs ,env))
```

```
MACRO:
top-value(env) = `(car (top-vs ,env))
```

```
MACRO:
nth-vse(n, env)
=
`(nth ,n (top-vs ,env))
```

Pop a stack from the stack*.

```
MACRO:
pop-es(env)
=
` (list
  (cdr (es ,env))
  (vs ,env)
  (as ,env))
```

```
MACRO:
```



```
pop-vs(env)
=
` (list
  (es ,env)
  (cdr (vs ,env))
  (as ,env))
```

```
MACRO:
pop-as(env)
=
` (list
  (es ,env)
  (vs ,env)
  (cdr (as ,env)))
```

Push a stack onto the stack*.

```
MACRO:
push-es(x, env)
=
` (list
  (cons ,x (es ,env))
  (vs ,env)
  (as ,env))
```

```
MACRO:
push-vs(x, env)
=
` (list
  (es ,env)
  (cons ,x (vs ,env))
  (as ,env))
```

```
MACRO:
push-as(x, env)
=
` (list
  (es ,env)
  (vs ,env)
  (cons ,x (as ,env)))
```

Push an entry onto the the top stack of the stack*.

```
MACRO:
push-ese(x, env)
=
` (list
  (cons
    (cons ,x (top-es ,env))
    (cdr (es ,env)))
  (vs ,env)
  (as ,env))
```

```
MACRO:
push-vse(x, env)
=
` (list
  (es ,env)
  (cons
    (cons ,x (top-vs ,env))
```

```

      (cdr (vs ,env)))
    (as ,env))
MACRO:
push-ase(x, env)
=
`(list
  (es ,env)
  (vs ,env)
  (cons
    (cons ,x (top-as ,env))
    (cdr (as ,env))))
MACRO:
pop-ese(env)
=
`(list
  (cons
    (cdr (top-es ,env))
    (cdr (es ,env)))
  (vs ,env)
  (as ,env))
MACRO:
pop-vse(env)
=
`(list
  (es ,env)
  (cons
    (cdr (top-vs ,env))
    (cdr (vs ,env)))
  (as ,env))
MACRO:
pop-ase(env)
=
`(list
  (es ,env)
  (vs ,env)
  (cons
    (cdr (top-as ,env))
    (cdr (as ,env))))
MACRO:
append-vse(x, env)
=
`(list
  (es ,env)
  (cons
    (append ,x (top-vs ,env))
    (cdr (vs ,env)))
  (as ,env))
MACRO:
append-ase(x, env)
=
`(list
  (es ,env)
  (vs ,env)
  (cons
    (append ,x (top-as ,env))

```

```

      (cdr (as ,env)))
MACRO:
push-entry(x, env) = `(push-ese ,x ,env)
MACRO:
push-value(x, env) = `(push-vse ,x ,env)
MACRO:
push-assert(x, env) = `(push-ase ,x ,env)
MACRO:
pop-value(env) = `(pop-vse ,env)
MACRO:
push-env(&OPTIONAL env := 'env)
=
`(list
  (cons nil (es ,env))
  (cons nil (vs ,env))
  (cons nil (as ,env)))
MACRO:
pop-env(&OPTIONAL env := 'env)
=
`(list
  (cdr (es ,env))
  (cdr (vs ,env))
  (cdr (as ,env)))
DEFINITION:
update-v(x, val, va)
=
if atom(va) then va
elseif atom(car(va)) then cons(car(va), update-v(x, val, cdr(va)))
elseif x = caar(va) then cons(make-entry(x, cadr(va), val), cdr(va))
else cons(car(va), update-v(x, val, cdr(va)))
fi
DEFINITION:
set-vse*(x, val, vs)
=
if atom(vs) then vs
elseif assoc(x, car(vs)) then cons(update-v(x, val, car(vs)), cdr(vs))
else cons(car(vs), set-vse*(x, val, cdr(vs)))
fi
DEFINITION:
set-vse(x, val, env) = mkenv(enve, set-vse*(x, val, envv), enva)

```

Other, non-env related, macros.

```

MACRO:
int-not-in-machine-range(val)
=
`(int-not-in-range
  ,val
  (ava-min-int)
  (ava-max-int))

```

A.2 Ilet

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *leval-macros*.

MACRO:

```
assert1(x)
=
\ (mv nil (push-ase ,x env))
```

MACRO:

```
asserts(l)
=
\ (mv nil (append-ase ,l env))
```

MACRO:

```
check-assert(form,
              &OPTIONAL
              outstate := '(es env),
              instate := 'nil
              )
```

=

```
\ (check-annotations
    (list ,form)
    ,instate
    ,outstate
    env)
```

MACRO:

```
check-asserts(&OPTIONAL outstate := '(es env), instate := 'nil)
```

=

```
\ (check-annotations
    (top-as env)
    ,instate
    ,outstate
    env)
```

CONSTANT:

```
*logical-error* = 'logical-error
```

MACRO:

```
hard-error2(fmt-string, &REST args)
```

=

```
\ (list
    'hard-error
    ,fmt-string
    (list ,@args))
```

MACRO:

```
hard-error-env(env, fmt-string, &REST args)
```

=

```
\ (mv
    (list
     'hard-error
     ,fmt-string
     (list ,@args))
    ,env)
```

DEFINITION:

```
check-annotations(l, instate, outstate, env)
```

```

=
if null(l) then ⟨nil, env⟩
elseif leval(car(l), outstate, instate)
then check-annotations(cdr(l), instate, outstate, env)
else ⟨*logical-error*, env⟩
fi

DEFINITION:
do-mv-macro(l)
=
if ¬ consp(l)
then if null(l) then '(mv exc env)
else ['mv, 'exc, l]
fi
elseif ¬ consp(car(l))
then if cdr(l)
then hard-error2("Atomic non-terminal element of ilet (~s)",
                  l)
else `(mv nil ,(car l))
fi
elseif caar(l) = ' :=
then let* arg be car(l)1,
          step be car(l)2,
          fail be caddr(car(l) ^ car(l)3,
          rest be do-mv-macro(cdr(l))
in
if symbolp(arg)
then if arg = ' exc
then `(let
        ((exc ,step))
        (if exc
          ,(if fail fail
              '(mv exc env))
          ,rest))
elseif fail
then hard-error2("~%Cannot have FAIL branch without EXC (~s)",
                  l)
else `(let
        ((,arg ,step))
        ,rest)
fi
elseif consp(arg)
then if ' exc ∈ arg
then `(mv-let
        ,arg
        ,step
        (if exc
          ,(if fail fail
              '(mv exc env))
          ,rest))
elseif fail
then hard-error2("~%Cannot have FAIL branch without EXC (~s)",
                  l)
else `(mv-let
        ,arg
        ,step
        ,rest)
fi

```

```

    else hard-error2("%Arg to := must be symbol or list (~s)",
                    l)
  fi
elseif caar(l) = 'exception
then let* form be car(l)1,
      fail be if caddr(car(l)) then car(l)2
              else nil
              fi,
      rest be do-mv-macro(cdr(l))
  in
  `(let
    ((exc ,form))
    (if exc
      ,(if fail fail
        '(mv exc env))
      ,rest))
elseif null(cdr(l)) ^ (caar(l) = 'mv) then car(l)
else let form be car(l),
      rest be do-mv-macro(cdr(l))
  in
  `(mv-let
    (exc env)
    ,form
    (if exc
      (mv exc env)
      ,rest))
fi

MACRO:
ilet(&REST l)
=
do-mv-macro(l)

```

A.3 Trees for Array and Record Literals

In this version, labels can be integers, symbols or lists. In principle, there is no reason to limit them.

Leaves of trees can be integers, characters, strings, or the symbols TRUE or FALSE. We excluded other symbols because it was important to ensure that

```

treep(x) -> ¬ leafp(x) ^
leafp(x) -> ¬ treep(x)

```

We probably could just as well replace boolean with non-nil-symbol.

```

leaf == integer || character || boolean || string
tree == ((label . node)*)
node == leaf || tree
label == integer || symbol || consp

```

Note from previous versions :

1. We changed ADA-BOOLEANP so that NIL is not both leafp and treep.
2. We let LABELP include CONSP because we may want to leave record selectors as (id

field-name 23).

SET CURRENT PACKAGE to be **ACL2**.

DEFINITION:

$\text{ada-booleanp}(x) = (x = \text{'true}) \vee (x = \text{'false})$

DEFINITION:

$\text{leaf-p}(x)$

=

$\text{integerp}(x) \vee \text{characterp}(x) \vee \text{ada-booleanp}(x) \vee \text{stringp}(x)$

DEFINITION:

$\text{labelp}(x) = \text{integerp}(x) \vee \text{symbolp}(x) \vee \text{consp}(x)$

DEFINITION:

$\text{treep}(x)$

=

if $\text{consp}(x)$

then $\text{consp}(\text{car}(x))$

\wedge **let** $label$ **be** $\text{car}(\text{car}(x))$,

$node$ **be** $\text{cdr}(\text{car}(x))$

in

$\text{labelp}(label) \wedge (\text{leaf-p}(node) \vee \text{treep}(node))$

$\wedge \text{treep}(\text{cdr}(x))$

else $\text{null}(x)$

fi

DEFINITION:

$\text{labelp}^*(l)$

=

if $\neg \text{consp}(l)$ **then** $\text{null}(l)$

elseif $\text{labelp}(\text{car}(l))$ **then** $\text{labelp}^*(\text{cdr}(l))$

else nil

fi

Labelp theorems

THEOREM: $\text{labelp}^*\text{-cdr}$

$\text{labelp}^*(l) \rightarrow \text{labelp}^*(\text{cdr}(l))$

:forward-chaining, :rewrite

THEOREM: $\text{labelp}^*\text{-car}$

$\text{labelp}^*(l) \rightarrow \text{labelp}(\text{car}(l))$

THEOREM: $\text{labelp}^*\text{-singleton}$

$\text{labelp}(i) \rightarrow \text{labelp}^*([i])$

THEOREM: $\text{labelp}^*\text{-list-car-if-labelp}^*$

$(\text{labelp}^*(l) \wedge l) \rightarrow \text{labelp}^*(\text{[car}(l)])$

THEOREM: $\text{labelp}^*\text{-true-listp}$

$\text{labelp}^*(l) \rightarrow \text{true-listp}(l)$

:compound-recognizer

THEOREM: consp-labelp^*

$(\text{labelp}^*(l) \wedge l) \rightarrow \text{consp}(l)$

:forward-chaining

THEOREM: $\text{labelp}^*\text{-fwd}$

$\text{labelp}^*(\text{cons}(i, l)) \rightarrow (\text{labelp}^*(l) \wedge \text{labelp}(i))$

:forward-chaining

THEOREM: $\text{labelp}^*\text{-bkwd}$

$(\text{labelp}^*(l) \wedge \text{labelp}(i)) \rightarrow \text{labelp}^*(\text{cons}(i, l))$

MODIFY the current theory:

Disable 'labelp*' and 'labelp'.

These THEOREMS must precede put-1, get-1

THEOREM: treep-alistp *:forward-chaining*
 $\text{treep}(x) \rightarrow \text{alistp}(x)$

THEOREM: treep-cdr
 $(\text{treep}(x) \wedge x) \rightarrow \text{treep}(\text{cdr}(x))$

THEOREM: treep-nil
 $\text{treep}(\mathbf{nil})$

THEOREM: treep-cdr-fwd *:forward-chaining*
 $\text{treep}(\text{cons}(w, x)) \rightarrow \text{treep}(x)$

MODIFY the current theory:

Disable 'leaf-p' and 'treep'.

Valuep and theorems

DEFINITION:
 $\text{valuep}(x) = \text{leaf-p}(x) \vee \text{treep}(x)$

THEOREM: leaf-p-not-treep *:forward-chaining*
 $\text{leaf-p}(x) \rightarrow (\neg \text{treep}(x))$

THEOREM: treep-not-leaf-p *:forward-chaining*
 $\text{treep}(x) \rightarrow (\neg \text{leaf-p}(x))$

THEOREM: not-treep-imp-leaf-p
 $(\neg \text{treep}(x)) \rightarrow (\text{valuep}(x) \leftrightarrow \text{leaf-p}(x))$

THEOREM: not-leaf-p-imp-treep
 $(\neg \text{leaf-p}(x)) \rightarrow (\text{valuep}(x) \leftrightarrow \text{treep}(x))$

THEOREM: leaf-p-valuep *:forward-chaining*
 $\text{leaf-p}(x) \rightarrow \text{valuep}(x)$

THEOREM: treep-valuep *:forward-chaining*
 $\text{treep}(x) \rightarrow \text{valuep}(x)$

THEOREM: treep-cons
 $(\text{treep}(z) \wedge \text{labelp}(i) \wedge \text{valuep}(val))$
 $\rightarrow \text{treep}(\text{cons}(\text{cons}(i, val), z))$

MODIFY the current theory:

Disable 'valuep'.

Domain & range

The list of CARs of a tree.

DEFINITION:
 $\text{domain}(x) = \text{strip-cars}(x)$
Guard: $\text{treep}(x)$

The list of CDRs of a tree.

DEFINITION:

$\text{range}(x) = \text{strip-cdrs}(x)$

Guard: $\text{treep}(x)$

MODIFY the current theory:

Enable 'labelp', 'labelp*' and 'treep'.

A tree containing only those pairs in A whose CAR is in L.

DEFINITION:

$\text{domain-restrict}(l, a)$

=

if $(\neg \text{consp}(a)) \vee (\neg \text{consp}(\text{car}(a)))$ **then nil**

elseif $\text{member-equal}(\text{car}(\text{car}(a)), l)$

then $\text{cons}(\text{car}(a), \text{domain-restrict}(l, \text{cdr}(a)))$

else $\text{domain-restrict}(l, \text{cdr}(a))$

fi

Guard: $\text{labelp}^*(l) \wedge \text{treep}(a)$

MODIFY the current theory:

Disable 'labelp', 'labelp*' and 'treep'.

Atomic versions of GET and SET. Take a SINGLE label.

$(\text{get-t nil } i) = \text{nil}$ $(\text{get-t } x \text{ nil}) = \text{Treats NIL as symbol and looks for it.}$

MODIFY the current theory:

Enable 'treep'.

DEFINITION:

$\text{get-t}(tree, i)$

=

if $(\neg \text{consp}(tree)) \vee (\neg \text{consp}(\text{car}(tree)))$ **then nil**

elseif $i = \text{car}(\text{car}(tree))$ **then** $\text{cdr}(\text{car}(tree))$

else $\text{get-t}(\text{cdr}(tree), i)$

fi

Guard: $\text{labelp}(i) \wedge \text{treep}(tree)$

$(\text{put-t nil } i \text{ } v) = ((i . v))$ $(\text{put-t nil nil nil}) = ((\text{nil} . \text{nil}))$

DEFINITION:

$\text{put-t}(tree, i, val)$

=

if $(\neg \text{consp}(tree)) \vee (\neg \text{consp}(\text{car}(tree)))$ **then** $[\text{cons}(i, val)]$

elseif $i = \text{car}(\text{car}(tree))$ **then** $\text{cons}(\text{cons}(i, val), \text{cdr}(tree))$

else $\text{cons}(\text{car}(tree), \text{put-t}(\text{cdr}(tree), i, val))$

fi

Guard: $\text{treep}(tree) \wedge \text{labelp}(i) \wedge \text{valuep}(val)$

DEFINITION:

$\text{exists-t}(tree, i)$

=

if $(\neg \text{consp}(tree)) \vee (\neg \text{consp}(\text{car}(tree)))$ **then nil**

elseif $i = \text{car}(\text{car}(tree))$ **then t**

else $\text{exists-t}(\text{cdr}(tree), i)$

fi

Guard: $\text{treep}(tree) \wedge \text{labelp}(i)$

$(\text{defthm exists-in-domain (implies (and (treep tree) (labelp i)) (iff (exists-t tree i) (member-equal i$

(domain tree))) :hints (("Goal" :in-theory (disable labelp))))

MODIFY the current theory:

Disable 'treep'.

(in-theory (disable exists-in-domain))

Get-t openers

THEOREM: get-t-opener-1

(null (*tree*) \wedge force (labelp (*i*))) \rightarrow (get-t (*tree*, *i*) = nil)

THEOREM: get-t-opener-2

((*i* = car (car (*tree*)))
 \wedge *tree*
 \wedge force (labelp (*i*))
 \wedge force (treep (*tree*)))
 \rightarrow (get-t (*tree*, *i*) = cdar (*tree*))

THEOREM: get-t-opener-3

((*i* \neq car (car (*tree*)))
 \wedge *tree*
 \wedge force (labelp (*i*))
 \wedge force (treep (*tree*)))
 \rightarrow (get-t (*tree*, *i*) = get-t (cdr (*tree*), *i*))

THEOREM: get-t-nil-2

((\neg exists-t (*x*, *i*)) \wedge force (labelp (*i*)) \wedge force (treep (*x*)))
 \rightarrow (get-t (*x*, *i*) = nil)

PUT-T openers

THEOREM: put-t-opener-1

(null (*tree*)
 \wedge force (treep (*tree*))
 \wedge force (labelp (*i*))
 \wedge force (valuep (*val*)))
 \rightarrow (put-t (*tree*, *i*, *val*) = [cons (*i*, *val*)])

THEOREM: put-t-opener-2

(*tree*
 \wedge (*i* = car (car (*tree*)))
 \wedge force (treep (*tree*))
 \wedge force (labelp (*i*))
 \wedge force (valuep (*val*)))
 \rightarrow (put-t (*tree*, *i*, *val*) = cons (cons (*i*, *val*), cdr (*tree*)))

Acl2 count lemmas Count theorems about trees, needed to admit get-l and exists-l.

THEOREM: acl2-count-cadar-3

(car (*x*) \wedge treep (*x*) \wedge (0 \leq *w*) \wedge integerp (*w*))
 \rightarrow (acl2-count (cdar (*x*)) < (1 + acl2-count (car (*x*)) + *w*))

THEOREM: acl2-count-get-t-2

(exists-t (*x*, *i*) \wedge force (treep (*x*)) \wedge force (labelp (*i*)))
 \rightarrow (acl2-count (get-t (*x*, *i*)) < acl2-count (*x*))

TREEP theorems

THEOREM: treep-cons-car
 $(\text{treep}(y) \wedge y) \rightarrow \text{treep}([\text{car}(y)])$

THEOREM: treep-cons-cdr
 $(\text{treep}(x) \wedge \text{treep}(y) \wedge y) \rightarrow \text{treep}(\text{cons}(\text{car}(y), x))$

PUT-T theorems

THEOREM: treep-put-t
 $(\text{force}(\text{treep}(x)) \wedge \text{force}(\text{labelp}(i)) \wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow \text{treep}(\text{put-t}(x, i, val))$

THEOREM: get-t-put-t-identity
 $(\text{force}(\text{treep}(x)) \wedge \text{force}(\text{valuep}(foo)) \wedge \text{force}(\text{labelp}(i)))$
 $\rightarrow (\text{get-t}(\text{put-t}(x, i, foo), i) = foo)$

List versions of GET and SET.

$(\text{exists-l nil } i) = \text{nil} \quad (\text{get-l nil } l) = \text{nil} \quad (\text{get-l } x \text{ nil}) = x$

MODIFY the current theory:

Enable 'treep'.

MODIFY the current theory:

Disable 'leaf-p'.

THEOREM: exists-imp-not-leaf-p-get-treep-get
 $(\text{labelp}^*(\text{cons}(l1, l2))$
 $\wedge \text{treep}(x)$
 $\wedge \text{exists-t}(x, l1)$
 $\wedge (\neg \text{leaf-p}(\text{get-t}(x, l1)))$
 $\rightarrow \text{treep}(\text{get-t}(x, l1))$

DEFINITION:

$\text{get-l}(x, l)$
 $=$
if $\neg \text{consp}(l)$ **then** x
elseif $\text{leaf-p}(x)$ **then** nil
elseif $(\neg \text{treep}(x)) \vee (\neg \text{labelp}(\text{car}(l)))$ **then** nil
elseif $\text{exists-t}(x, \text{car}(l))$ **then** $\text{get-l}(\text{get-t}(x, \text{car}(l)), \text{cdr}(l))$
else nil
fi
Guard: $(\text{leaf-p}(x) \vee \text{treep}(x)) \wedge \text{labelp}^*(l)$

MODIFY the current theory:

Disable 'treep'.

$(\text{exists-l nil } i) = \text{nil} \quad (\text{put-l } x \text{ nil } v) = v$

DEFINITION:

$\text{put-l}(x, l, val)$
 $=$
if $\neg \text{consp}(l)$ **then** val
elseif $\neg \text{labelp}(\text{car}(l))$ **then** val
elseif $\neg \text{treep}(x)$ **then** $\text{put-t}(\text{nil}, \text{car}(l), \text{put-l}(\text{nil}, \text{cdr}(l), val))$
elseif $\text{null}(\text{cdr}(l))$ **then** $\text{put-t}(x, \text{car}(l), val)$
else $\text{put-t}(x, \text{car}(l), \text{put-l}(\text{get-l}(x, [\text{car}(l)]), \text{cdr}(l), val))$
fi

Guard: $(\text{leaf-p}(x) \vee \text{treep}(x)) \wedge \text{labelp}^*(l) \wedge \text{valuep}(val)$

$(\text{exists-l nil nil}) = \text{T}$ $(\text{exists-l x nil}) = \text{T}$ $(\text{exists-l nil (cons a b)}) = \text{F}$

DEFINITION:

$\text{exists-l}(tree, l)$

=

if $\neg \text{consp}(l)$ **then t**
elseif $\neg \text{labelp}(\text{car}(l))$ **then nil**
elseif $\neg \text{treep}(tree)$ **then nil**
elseif $\text{exists-t}(tree, \text{car}(l)) \wedge \text{treep}(\text{get-t}(tree, \text{car}(l)))$
then $\text{exists-l}(\text{get-t}(tree, \text{car}(l)), \text{cdr}(l))$
else nil
fi

Guard: $\text{treep}(tree) \wedge \text{labelp}^*(l)$

DEFINITION:

$\text{distinct}(i, j)$

=

if $(\neg \text{consp}(i)) \vee (\neg \text{consp}(j))$ **then nil**
elseif $\text{car}(i) \neq \text{car}(j)$ **then t**
else $\text{distinct}(\text{cdr}(i), \text{cdr}(j))$
fi

Guard: $\text{labelp}^*(i) \wedge \text{labelp}^*(j)$

Put-l, Get-l Theorems

THEOREM: valuep-get-t

$(\text{force}(\text{treep}(x)) \wedge \text{force}(\text{labelp}(i))) \rightarrow \text{valuep}(\text{get-t}(x, i))$

THEOREM: null-get-l-2

$((\neg \text{treep}(w)) \wedge \text{force}(\text{valuep}(w)) \wedge \text{force}(\text{labelp}^*(l)))$
 $\rightarrow (\text{get-l}(w, l)$
 = **if** $\text{null}(l)$ **then** w
else nil
fi)

THEOREM: valuep-get-l-1

$((\neg \text{treep}(\text{get-t}(x, i))) \wedge \text{labelp}(i) \wedge \text{treep}(x))$
 $\rightarrow \text{leaf-p}(\text{get-t}(x, i))$

THEOREM: valuep-get-l

$(\text{force}(\text{treep}(x)) \wedge \text{force}(\text{labelp}^*(l))) \rightarrow \text{valuep}(\text{get-l}(x, l))$

THEOREM: valuep-put-t

$(\text{force}(\text{labelp}(i)) \wedge \text{force}(\text{treep}(w)) \wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow \text{valuep}(\text{put-t}(w, i, val))$

MODIFY the current theory:

Enable 'valuep'.

THEOREM: $\text{valuep-get-l-list-i}$

$(\text{treep}(m) \wedge \text{labelp}(i)) \rightarrow \text{valuep}(\text{get-l}(m, [i]))$

THEOREM: valuep-put-l

$(\text{force}(\text{labelp}^*(l)) \wedge \text{force}(\text{valuep}(w)) \wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow \text{valuep}(\text{put-l}(w, l, val))$

THEOREM: treep-put-l

$(l2 \wedge \text{force}(\text{labelp}^*(l2)) \wedge \text{force}(\text{valuep}(w)) \wedge \text{force}(\text{valuep}(val)))$

→ treep (put-l (w, l2, val))

THEOREM: valuep-cons-put-l-nil

(valuep (val) ∧ labelp* (l) ∧ l)

→ valuep ([cons (car (l), put-l (nil, cdr (l), val))])

MODIFY the current theory:

Disable ‘valuep’, ‘leaf-p’ and ‘treep’.

VERIFY GUARDS for ‘put-l’

THEOREM: leaf-p-not-nil

leaf-p (val) → val

:forward-chaining

THEOREM: not-leaf-p-cons

→ leaf-p (cons (a, b))

THEOREM: put-l-leaf-p

(l

∧ (¬ treep (w))

∧ force (labelp* (l))

∧ force (valuep (val))

∧ force (valuep (w)))

→ (put-l (w, l, val) = put-l (nil, l, val))

THEOREM: exists-if-put-t

(force (labelp (i)) ∧ force (valuep (w)) ∧ force (treep (x)))

→ exists-t (put-t (x, i, w), i)

THEOREM: not-leaf-p-put-t

(labelp (i) ∧ valuep (val) ∧ treep (x))

→ (¬ leaf-p (put-t (x, i, val)))

THEOREM: get-put-l

(l ∧ force (treep (x)) ∧ force (valuep (val)) ∧ force (labelp* (l)))

→ (get-l (put-l (x, l, val), l) = val)

THEOREM: treep-cons-put-t

(x ∧ force (treep (x)) ∧ force (labelp (i)) ∧ force (valuep (w)))

→ treep (cons (car (x), put-t (cdr (x), i, w)))

THEOREM: get-t-put-t-ne

((i ≠ j)

∧ get-t (x, j)

∧ force (labelp (j))

∧ force (labelp (i))

∧ force (treep (x))

∧ force (valuep (w)))

→ (get-t (put-t (x, i, w), j) = get-t (x, j))

THEOREM: not-get-t-step

((¬ get-t (x, j))

∧ (i ≠ j)

∧ force (treep (x))

∧ force (valuep (w))

∧ force (labelp (i))

∧ force (labelp (j)))

→ (¬ get-t (put-t (x, i, w), j))

THEOREM: fail-get-l

$((\neg \text{get-t}(x, \text{car}(l))) \wedge l \wedge \text{force}(\text{treep}(x)) \wedge \text{force}(\text{labelp}^*(l)))$
 $\rightarrow (\text{get-l}(x, l) = \mathbf{nil})$

THEOREM: put-t-opener-3

$((i \neq \text{car}(\text{car}(tree)))$
 $\wedge \text{consp}(tree)$
 $\wedge \text{consp}(\text{car}(tree))$
 $\wedge \text{force}(\text{treep}(tree))$
 $\wedge \text{force}(\text{labelp}(i))$
 $\wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow (\text{put-t}(tree, i, val) = \text{cons}(\text{car}(tree), \text{put-t}(\text{cdr}(tree), i, val)))$

THEOREM: get-l-opener-1

$(\text{null}(l) \wedge (\text{leaf-p}(x) \vee \text{treep}(x)) \wedge \text{force}(\text{labelp}^*(l)))$
 $\rightarrow (\text{get-l}(x, l) = x)$

THEOREM: get-l-opener-3

$(l \wedge \text{exists-t}(x, \text{car}(l)) \wedge \text{treep}(x) \wedge \text{force}(\text{labelp}^*(l)))$
 $\rightarrow (\text{get-l}(x, l) = \text{get-l}(\text{get-t}(x, \text{car}(l)), \text{cdr}(l)))$

THEOREM: get-l-opener-4

$(l \wedge (\neg \text{exists-t}(x, \text{car}(l))) \wedge \text{treep}(x) \wedge \text{force}(\text{labelp}^*(l)))$
 $\rightarrow (\text{get-l}(x, l) = \mathbf{nil})$

THEOREM: put-l-opener-1

$(\text{null}(l)$
 $\wedge (\text{leaf-p}(x) \vee \text{treep}(x))$
 $\wedge \text{force}(\text{labelp}^*(l))$
 $\wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow (\text{put-l}(x, l, val) = val)$

THEOREM: put-l-opener-2

$(l \wedge \text{leaf-p}(x) \wedge \text{force}(\text{labelp}^*(l)) \wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow (\text{put-l}(x, l, val) = \text{put-t}(\mathbf{nil}, \text{car}(l), \text{put-l}(\mathbf{nil}, \text{cdr}(l), val)))$

THEOREM: put-l-opener-3

$(l$
 $\wedge \text{null}(\text{cdr}(l))$
 $\wedge \text{treep}(x)$
 $\wedge \text{force}(\text{labelp}^*(l))$
 $\wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow (\text{put-l}(x, l, val) = \text{put-t}(x, \text{car}(l), val))$

THEOREM: put-l-opener-4

$(l$
 $\wedge (\neg \text{null}(\text{cdr}(l)))$
 $\wedge \text{treep}(x)$
 $\wedge \text{force}(\text{labelp}^*(l))$
 $\wedge \text{force}(\text{valuep}(val)))$
 $\rightarrow (\text{put-l}(x, l, val)$
 $= \text{put-t}(x, \text{car}(l), \text{put-l}(\text{get-l}(x, [\text{car}(l)]), \text{cdr}(l), val)))$

THEOREM: exists-t-opener-1

$(\text{null}(tree) \wedge \text{force}(\text{treep}(tree)) \wedge \text{force}(\text{labelp}(i)))$
 $\rightarrow (\text{exists-t}(tree, i) = \mathbf{nil})$

THEOREM: exists-t-opener-2

$(\text{consp}(tree)$
 $\wedge \text{consp}(\text{car}(tree))$
 $\wedge (i = \text{car}(\text{car}(tree)))$
 $\wedge \text{force}(\text{treep}(tree))$

\wedge force (labelp (i))
 \rightarrow (exists-t ($tree$, i) = t)

THEOREM: exists-t-opener-3

(consp ($tree$)
 \wedge consp (car ($tree$))
 \wedge ($i \neq$ car (car ($tree$)))
 \wedge force (treep ($tree$))
 \wedge force (labelp (i))
 \rightarrow (exists-t ($tree$, i) = exists-t (cdr ($tree$), i))

MODIFY the current theory:

Disable 'get-t', 'get-l', 'put-t' and 'put-l'.

THEOREM: exists-t-exists-l

($l \wedge$ exists-l (x , l) \wedge force (treep (x)) \wedge force (labelp* (l)))
 \rightarrow exists-t (x , car (l))

THEOREM: exists-t-put-t

(exists-t (x , j)
 \wedge ($i \neq j$)
 \wedge force (treep (x))
 \wedge force (valuep (w))
 \wedge force (labelp (i))
 \wedge force (labelp (j))
 \rightarrow exists-t (put-t (x , i , w), j)

THEOREM: get-l-put-t-opener

(l
 \wedge ($i \neq$ car (l))
 \wedge force (treep (x))
 \wedge force (labelp* (l))
 \wedge force (valuep (w))
 \wedge force (labelp (i))
 \rightarrow (get-l (put-t (x , i , w), l)
 = **if** get-t (x , car (l)) **then** get-l (x , l)
else nil
fi)

DEFINITION:

induct3 (i , j , x)

=

if (\neg consp (i)) \vee (\neg consp (j)) **then nil**
elseif car (i) \neq car (j) **then nil**
elseif leaf-p (x) **then nil**
elseif null (cdr (j)) **then nil**
elseif exists-t (x , car (i)) **then** induct3 (cdr (i), cdr (j), get-t (x , car (i)))
else induct3 (cdr (i), cdr (j), get-t (x , car (i)))
fi

Guard: (leaf-p (x) \vee treep (x)) \wedge labelp* (i) \wedge labelp* (j)

THEOREM: not-distinct

(labelp* (i)
 \wedge labelp* (j)
 \wedge i
 \wedge j
 \wedge (car (i) = car (j))
 \wedge (\neg cdr (j))
 \rightarrow (\neg distinct (i , j))

THEOREM: get-l-put-l-opener-unequal-cars

```
(  labelp* (i)
  ^ labelp* (j)
  ^ i
  ^ j
  ^ (car (i) ≠ car (j))
  ^ treep (x)
  ^ valuep (val))
→ (get-l (put-l (x, i, val), j) = get-l (x, j))
```

THEOREM: get-l-put-l-nil

```
(  distinct (i, j)
  ^ force (labelp* (i))
  ^ force (labelp* (j))
  ^ force (valuep (val)))
→ (get-l (put-l (nil, i, val), j) = get-l (nil, j))
```

THEOREM: not-get-l-leaf

```
(l ∧ (¬ treep (w)) ∧ labelp* (l) ∧ force (valuep (w)))
→ (¬ get-l (w, l))
```

THEOREM: get-put-i-j

```
(  distinct (i, j)
  ^ force (treep (x))
  ^ force (valuep (val))
  ^ force (labelp* (i))
  ^ force (labelp* (j)))
→ (get-l (put-l (x, i, val), j) = get-l (x, j))
```

A.4 Support for Tree Equality Reasoning

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *get-tree*.

MODIFY the current theory: Disable 'leaf-p', 'labelp', their executable counterparts.

DEFINITION:

lessp-or-equal (x, y)

=

if x = y **then** t

elseif rationalp (x)

then if rationalp (y) **then** x < y

else t

fi

elseif rationalp (y) **then** nil

elseif characterp (x)

then if characterp (y) **then** char<(x, y)

else t

fi

elseif characterp (y) **then** nil

elseif symbolp (x)

then if symbolp (y) **then** symbol-<(x, y)

else t

fi

elseif symbolp (y) **then** nil

elseif stringp (x)

then if stringp (y) **then** string<(x, y)


```

    else t
  fi
elseif stringp(y) then nil
elseif ¬ consp(x) then t
elseif ¬ consp(y) then nil
elseif car(x) = car(y) then lessp-or-equal(cdr(x), cdr(y))
elseif lessp-or-equal(car(x), car(y)) then t
else lessp-or-equal(cdr(x), cdr(y))
fi

```

DEFINITION:

```

insert(a, l)
=
if ¬ consp(l) then cons(a, l)
elseif consp(car(l))
  ∧ consp(a)
  ∧ lessp-or-equal(car(car(l)), car(a))
then cons(car(l), insert(a, cdr(l)))
else cons(a, l)
fi

```

DEFINITION:

```

sortl(alist)
=
if ¬ consp(alist) then alist
elseif ¬ consp(cdr(alist)) then alist
else insert(car(alist), sortl(cdr(alist)))
fi

```

Needed for `ava-equal-1` admission below.

THEOREM: `insert-cons-count=`
 $\text{acl2-count}(\text{insert}(a, w)) = \text{acl2-count}(\text{cons}(a, w))$

THEOREM: `sortl-count=sup-1`
 $\text{acl2-count}(\text{insert}(x, cx)) = (1 + \text{acl2-count}(x) + \text{acl2-count}(cx))$

THEOREM: `sortl-count=sup-2`
 $(\text{acl2-count}(w) = m)$
 $\rightarrow (\text{acl2-count}(\text{insert}(a, w)) = (1 + \text{acl2-count}(a) + m))$

THEOREM: `true-listp-sortl`
 $\text{true-listp}(x) \rightarrow \text{true-listp}(\text{sortl}(x))$

THEOREM: `sortl-count=sup-3`
 $(\text{consp}(x)$
 $\wedge \text{consp}(\text{cdr}(x))$
 $\wedge \text{acl2-count}(\text{sortl}(\text{cdr}(x))) = \text{acl2-count}(\text{cdr}(x)))$
 $\wedge \text{true-listp}(\text{cdr}(x)))$
 $\rightarrow (\text{acl2-count}(\text{insert}(\text{car}(x), \text{sortl}(\text{cdr}(x))))$
 $= (1 + \text{acl2-count}(\text{car}(x)) + \text{acl2-count}(\text{cdr}(x))))$

THEOREM: `sortl-count=`
 $\text{acl2-count}(\text{sortl}(x)) = \text{acl2-count}(x)$

THEOREM: `strip-cdrs-count-le` *:linear*
 $\text{acl2-count}(\text{strip-cdrs}(w)) \leq \text{acl2-count}(w)$

THEOREM: `strip-cdrs-count-le-sup-1` *:linear*
 $(\text{consp}(w) \wedge \text{acl2-count}(w) \leq \text{acl2-count}(x))$
 $\rightarrow (\text{acl2-count}(\text{cdr}(w)) < \text{acl2-count}(x))$

THEOREM: strip-cdrs-count-le-sup-2 :linear
 true-listp (x)
 $\rightarrow (\text{acl2-count}(\text{strip-cdrs}(\text{sortl}(\text{cdr}(x)))) \leq \text{acl2-count}(x))$

THEOREM: cdr-strip-cdrs-count-le
 $(\text{true-listp}(x) \wedge \text{consp}(x))$
 $\rightarrow (\text{acl2-count}(\text{cdr}(\text{strip-cdrs}(\text{sortl}(\text{cdr}(x)))) < \text{acl2-count}(x))$

THEOREM: treep-imp-true-listp :forward-chaining
 $\text{treep}(x) \rightarrow \text{true-listp}(x)$

MODIFY the current theory:

Enable 'treep'.

THEOREM: simple-strip-count=1 :linear
 $(\text{consp}(w) \wedge \text{treep}(w))$
 $\rightarrow (\text{acl2-count}(\text{strip-cdrs}(w)) < \text{acl2-count}(w))$

THEOREM: treep-sortl=1
 $\text{consp}(x) \rightarrow \text{consp}(\text{sortl}(x))$

DEFINITION:
 branchp (a)
 =
 $\text{labelp}(\text{car}(a)) \wedge (\text{leaf-p}(\text{cdr}(a)) \vee \text{treep}(\text{cdr}(a)))$

THEOREM: treep-sortl=2-1
 $((a \in x) \wedge \text{treep}(x)) \rightarrow \text{branchp}(a)$

THEOREM: member-insert=1
 $\text{member-equal}(a, w) \rightarrow \text{member-equal}(a, \text{insert}(z, w))$

THEOREM: member-sortl=2
 $\text{member-equal}(a, x) \rightarrow \text{member-equal}(a, \text{sortl}(x))$

MODIFY the current theory:

Enable 'treep' and 'leaf-p'.

THEOREM: sort-strip-count=1-a
 $(\text{consp}(\text{sortl}(x)) \wedge \text{treep}(\text{sortl}(x)))$
 $\rightarrow (\text{acl2-count}(\text{strip-cdrs}(\text{sortl}(x))) < \text{acl2-count}(\text{sortl}(x)))$

THEOREM: treep-insert
 $(\text{consp}(a) \wedge \text{labelp}(\text{car}(a)) \wedge \text{treep}(\text{cdr}(a)) \wedge \text{treep}(w))$
 $\rightarrow \text{treep}(\text{insert}(a, w))$

THEOREM: treep-sortl
 $\text{treep}(x) \rightarrow \text{treep}(\text{sortl}(x))$

THEOREM: sort-strip-count=1
 $(\text{consp}(x) \wedge \text{treep}(x))$
 $\rightarrow (\text{acl2-count}(\text{strip-cdrs}(\text{sortl}(x))) < \text{acl2-count}(\text{sortl}(x)))$

THEOREM: strip-cdrs-sort-consp-count
 $(\text{consp}(x) \wedge \text{treep}(x))$
 $\rightarrow (\text{acl2-count}(\text{strip-cdrs}(\text{sortl}(x))) < \text{acl2-count}(x))$

THEOREM: treep-sortl=2
 $\text{treep}(x) \rightarrow \text{treep}(\text{sortl}(x))$

A.5 Normalized Abstract Prefix

Input to the interpreter satisfies the predicate, `top-prefix-p`. The following functions define recognizers, constructors, and extractors for the various elements of the abstract prefix.

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *kernel-extension*.

INCLUDING the book: *get-tree*.

INCLUDING the book: *subprefix-macros*.

DEFINITION:

`symbolsp (l)`

=

if `consp (l)`

then if `symbolp (car (l))` **then** `symbolsp (cdr (l))`

else nil

fi

else `l = nil`

fi

DEFINITION:

`arg1 (p)`

=

if `consp (p)` **then** `p1`

else nil

fi

DEFINITION:

`arg2 (p)`

=

if `consp (p)` **then** `p2`

else nil

fi

DEFINITION:

`arg3 (p)`

=

if `consp (p)` **then** `p3`

else nil

fi

DEFINITION:

`arg4 (p)`

=

if `consp (p)` **then** `p4`

else nil

fi

DEFINITION:

`arg5 (p)`

=

if `consp (p)` **then** `p5`

else nil

fi

DEFINITION:

`arg6 (p)`

=

if consp (p) **then** p_6
else nil
fi

DEFINITION:

arg7 (p)

=

if consp (p) **then** p_7
else nil
fi

DEFINITION:

arg8 (p)

=

if consp (p) **then** p_8
else nil
fi

DEFINITION:

arg9 (p)

=

if consp (p) **then** p_9
else nil
fi

DEFINITION:

ada-char-p ($form$)

=

characterp ($form$) \wedge standard-char-p ($form$)

DEFINITION:

acl2-boolean ($form$) = ($form =_{eq}$ ' τ) \vee ($form =_{eq}$ **nil**)

DEFINITION:

all-integer (l)

=

if null (l) **then** **t**
elseif \neg consp (l) **then** **nil**
elseif integerp (car (l)) **then** all-integer (cdr (l))
else nil
fi

DEFINITION:

all-id (l)

=

if null (l) **then** **t**
elseif \neg consp (l) **then** **nil**
elseif listp (car (l)) \wedge (caar (l) =_{eq} ' id) **then** all-id (cdr (l))
else nil
fi

DEFINITION:

all-labels-integer (l) = labelp* (l) \wedge all-integer (l)

DEFINITION:

all-labels-id (l) = labelp* (l) \wedge all-id (l)

DEFINITION:

pre-type (x , $kind$)

=

(listp (x) \wedge (car (x) = ' id)
 \wedge (cadr (x) = $kind$)

\wedge (caddr(x) = 0)

DEFINITION:

pre-integer(x) = pre-type(x , 'integer)

DEFINITION:

pre-positive(x) = pre-type(x , 'positive)

DEFINITION:

pre-natural(x) = pre-type(x , 'natural)

DEFINITION:

pre-character(x) = pre-type(x , 'character)

DEFINITION:

pre-string(x) = pre-type(x , 'string)

DEFINITION:

pre-boolean(x) = pre-type(x , 'boolean)

DEFINITION:

array-literal-p($form$)

=

$form \wedge$ treep($form$) \wedge all-labels-integer(domain($form$))

DEFINITION:

record-literal-p($form$)

=

$form \wedge$ treep($form$) \wedge all-labels-id(domain($form$))

DEFINITION:

error-p(x) = listp(x) \wedge (car(x) = 'error)

DEFINITION:

mk-error($form$, $message$) = cons('error, [$form$, $message$])

DEFINITION:

error-form($form$) = arg*($form$)₀

DEFINITION:

error-message($form$) = arg*($form$)₁

DEFINITION:

others-p(x) = consp(x) \wedge (car(x) = 'others)

DEFINITION:

mk-others = ['others]

DEFINITION:

unconstrained-p(x)

=

consp(x) \wedge (car(x) = 'unconstrained)

DEFINITION:

mk-unconstrained = ['unconstrained]

DEFINITION:

id-p(x) = listp(x) \wedge (car(x) = 'id)

DEFINITION:

mk-id($root$, uid) = cons('id, [$root$, uid])

DEFINITION:

id-root($form$) = arg*($form$)₀

DEFINITION:

$\text{id-uid}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{true-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'true})$

DEFINITION:

$\text{mk-true} = [\text{'true}]$

DEFINITION:

$\text{false-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'false})$

DEFINITION:

$\text{mk-false} = [\text{'false}]$

DEFINITION:

$\text{boolean-literal-p}(x) = \text{true-p}(x) \vee \text{false-p}(x)$

DEFINITION:

$\text{list-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'list})$

DEFINITION:

$\text{mk-list}(args) = \text{cons}(\text{'list}, args)$

DEFINITION:

$\text{literal-p}(x)$
 $=$
 $\text{boolean-literal-p}(x)$
 \vee ($\text{integerp}(x)$
 \vee ($\text{stringp}(x)$
 \vee ($\text{ada-char-p}(x)$
 \vee ($\text{array-literal-p}(x) \vee \text{record-literal-p}(x)$))))

DEFINITION:

$\text{lliteral-p}(x)$
 $=$
 $\text{literal-p}(x) \vee (\text{acl2-boolean}(x) \vee \text{list-p}(x))$

DEFINITION:

$\text{enumeration-literal-p}(x) = \text{id-p}(x) \vee \text{ada-char-p}(x)$

DEFINITION:

$\text{enumeration-p}(x)$
 $=$
 $\text{consp}(x) \wedge (\text{car}(x) = \text{'enumeration})$

DEFINITION:

$\text{mk-enumeration}(args) = \text{cons}(\text{'enumeration}, args)$

DEFINITION:

$\text{predefined-type-p}(x)$
 $=$
 $\text{pre-character}(x)$
 \vee ($\text{pre-integer}(x)$
 \vee ($\text{pre-positive}(x)$
 \vee ($\text{pre-natural}(x) \vee (\text{pre-string}(x) \vee \text{pre-boolean}(x))$))))

DEFINITION:

$\text{type-mark-p}(x)$
 $=$
 $\text{listp}(x) \wedge (\text{car}(x) = \text{'type-mark})$

DEFINITION:

$\text{mk-type-mark}(type, constraint)$

=
 $\text{cons}(' \text{type-mark}, [\text{type}, \text{constraint}])$

DEFINITION:
 $\text{type-mark-type}(\text{form}) = \text{arg}^*(\text{form})_0$

DEFINITION:
 $\text{type-mark-constraint}(\text{form}) = \text{arg}^*(\text{form})_1$

DEFINITION:
 $\text{subtype-p}(x) = \text{id-p}(x) \vee \text{type-mark-p}(x)$

DEFINITION:
 $\text{attribute-p}(x)$
 =
 $\text{listp}(x) \wedge (\text{car}(x) = ' \text{attribute})$

DEFINITION:
 $\text{mk-attribute}(\text{root}, \text{attr}) = \text{cons}(' \text{attribute}, [\text{root}, \text{attr}])$

DEFINITION:
 $\text{attribute-root}(\text{form}) = \text{arg}^*(\text{form})_0$

DEFINITION:
 $\text{attribute-attr}(\text{form}) = \text{arg}^*(\text{form})_1$

DEFINITION:
 $\text{range-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{range})$

DEFINITION:
 $\text{mk-range}(\text{from}, \text{to}) = \text{cons}(' \text{range}, [\text{from}, \text{to}])$

DEFINITION:
 $\text{range-from}(\text{form}) = \text{arg}^*(\text{form})_0$

DEFINITION:
 $\text{range-to}(\text{form}) = \text{arg}^*(\text{form})_1$

DEFINITION:
 $\text{constraint-p}(x)$
 =
 $\text{subtype-p}(x)$
 $\vee (\text{unconstrained-p}(x) \vee (\text{range-p}(x) \vee \text{attribute-p}(x)))$

DEFINITION:
 $\text{field-spec-p}(x)$
 =
 $\text{listp}(x) \wedge (\text{car}(x) = ' \text{field-spec})$

DEFINITION:
 $\text{mk-field-spec}(\text{id}, \text{decl}) = \text{cons}(' \text{field-spec}, [\text{id}, \text{decl}])$

DEFINITION:
 $\text{field-spec-id}(\text{form}) = \text{arg}^*(\text{form})_0$

DEFINITION:
 $\text{field-spec-decl}(\text{form}) = \text{arg}^*(\text{form})_1$

DEFINITION:
 $\text{record-type-p}(x)$
 =
 $\text{consp}(x) \wedge (\text{car}(x) = ' \text{record-type})$

DEFINITION:
 $\text{mk-record-type}(\text{args}) = \text{cons}(' \text{record-type}, \text{args})$

DEFINITION:

array-type-p(x)
 =
 listp(x) \wedge (car(x) = 'array-type)

DEFINITION:

mk-array-type($index$, $elements$)
 =
 cons('array-type, [$index$, $elements$])

DEFINITION:

array-type-index($form$) = arg*($form$)₀

DEFINITION:

array-type-elements($form$) = arg*($form$)₁

DEFINITION:

type-p(x)
 =
 record-type-p(x)
 \vee (array-type-p(x) \vee (id-p(x) \vee (range-p(x) \vee enumeration-p(x))))

DEFINITION:

type-decl-p(x)
 =
 listp(x) \wedge (car(x) = 'type-decl)

DEFINITION:

mk-type-decl(id , $decl$) = cons('type-decl, [id , $decl$])

DEFINITION:

type-decl-id($form$) = arg*($form$)₀

DEFINITION:

type-decl-decl($form$) = arg*($form$)₁

DEFINITION:

subtype-decl-p(x)
 =
 listp(x) \wedge (car(x) = 'subtype-decl)

DEFINITION:

mk-subtype-decl(id , $decl$) = cons('subtype-decl, [id , $decl$])

DEFINITION:

subtype-decl-id($form$) = arg*($form$)₀

DEFINITION:

subtype-decl-decl($form$) = arg*($form$)₁

DEFINITION:

qualified-p(x)
 =
 listp(x) \wedge (car(x) = 'qualified)

DEFINITION:

mk-qualified($type$, $value$) = cons('qualified, [$type$, $value$])

DEFINITION:

qualified-type($form$) = arg*($form$)₀

DEFINITION:

qualified-value($form$) = arg*($form$)₁

DEFINITION:

type-convert-p(x)
 =
 listp(x) \wedge (car(x) = 'type-convert)

DEFINITION:

mk-type-convert($type$, $value$)
 =
 cons('type-convert, [$type$, $value$])

DEFINITION:

type-convert-type($form$) = arg*($form$)₀

DEFINITION:

type-convert-value($form$) = arg*($form$)₁

DEFINITION:

aggregate-choice-p(x)
 =
 listp(x) \wedge (car(x) = 'aggregate-choice)

DEFINITION:

mk-aggregate-choice($choice$, $value$)
 =
 cons('aggregate-choice, [$choice$, $value$])

DEFINITION:

aggregate-choice-choice($form$) = arg*($form$)₀

DEFINITION:

aggregate-choice-value($form$) = arg*($form$)₁

DEFINITION:

aggregate-pos-p(x)
 =
 listp(x) \wedge (car(x) = 'aggregate-pos)

DEFINITION:

mk-aggregate-pos($value$) = cons('aggregate-pos, [$value$])

DEFINITION:

aggregate-pos-value($form$) = arg*($form$)₀

DEFINITION:

aggregate-arm-p(x)
 =
 aggregate-choice-p(x) \vee aggregate-pos-p(x)

DEFINITION:

aggregate-p(x)
 =
 consp(x) \wedge (car(x) = 'aggregate)

DEFINITION:

mk-aggregate($args$) = cons('aggregate, $args$)

DEFINITION:

indexed-component-p(x)
 =
 listp(x) \wedge (car(x) = 'indexed-component)

DEFINITION:

mk-indexed-component($root$, $index$)
 =

`cons('indexed-component, [root, index])`

DEFINITION:

`indexed-component-root (form) = arg* (form)0`

DEFINITION:

`indexed-component-index (form) = arg* (form)1`

DEFINITION:

`apply-1-p (x) = listp (x) ∧ (car (x) = 'apply-1)`

DEFINITION:

`mk-apply-1 (root, args) = cons ('apply-1, [root, args])`

DEFINITION:

`apply-1-root (form) = arg* (form)0`

DEFINITION:

`apply-1-args (form) = arg* (form)1`

DEFINITION:

`selected-component-p (x)`

`=`

`listp (x) ∧ (car (x) = 'selected-component)`

DEFINITION:

`mk-selected-component (root, field)`

`=`

`cons ('selected-component, [root, field])`

DEFINITION:

`selected-component-root (form) = arg* (form)0`

DEFINITION:

`selected-component-field (form) = arg* (form)1`

DEFINITION:

`designator-p (x)`

`=`

`consp (x) ∧ (car (x) = 'designator)`

DEFINITION:

`mk-designator (args) = cons ('designator, args)`

DEFINITION:

`dot-qual-1-p (x)`

`=`

`listp (x) ∧ (car (x) = 'dot-qual-1)`

DEFINITION:

`mk-dot-qual-1 (root, component)`

`=`

`cons ('dot-qual-1, [root, component])`

DEFINITION:

`dot-qual-1-root (form) = arg* (form)0`

DEFINITION:

`dot-qual-1-component (form) = arg* (form)1`

DEFINITION:

`defining-name-p (x) = id-p (x) ∨ designator-p (x)`

DEFINITION:

`name-p (x)`

=
 id-p(x)
 \vee (indexed-component-p(x)
 \vee (selected-component-p(x) \vee (apply-1-p(x) \vee dot-qual-1-p(x))))

DEFINITION:

apl-p(x) = consp(x) \wedge (car(x) = 'apl)

DEFINITION:

mk-apl(args) = cons('apl, args)

DEFINITION:

op-expr-p(x) = listp(x) \wedge (car(x) = 'op-expr)

DEFINITION:

mk-op-expr(id, actuals) = cons('op-expr, [id, actuals])

DEFINITION:

op-expr-id(form) = arg*(form)₀

DEFINITION:

op-expr-actuals(form) = arg*(form)₁

DEFINITION:

function-call-p(x)

=

listp(x) \wedge (car(x) = 'function-call)

DEFINITION:

mk-function-call(id, actuals)

=

cons('function-call, [id, actuals])

DEFINITION:

function-call-id(form) = arg*(form)₀

DEFINITION:

function-call-actuals(form) = arg*(form)₁

DEFINITION:

expr-p(x)

=

literal-p(x)
 \vee (aggregate-p(x)
 \vee (name-p(x)
 \vee (op-expr-p(x)
 \vee (function-call-p(x)
 \vee (type-convert-p(x) \vee qualified-p(x))))))

DEFINITION:

instate-p(x) = listp(x) \wedge (car(x) = 'instate)

DEFINITION:

mk-instate(expr) = cons('instate, [expr])

DEFINITION:

instate-expr(form) = arg*(form)₀

DEFINITION:

outstate-p(x) = listp(x) \wedge (car(x) = 'outstate)

DEFINITION:

mk-outstate(expr) = cons('outstate, [expr])

DEFINITION:

outstate-expr (*form*) = arg* (*form*)₀

DEFINITION:

assert-p (*x*) = listp (*x*) ∧ (car (*x*) = 'assert)

DEFINITION:

mk-assert (*relation*) = cons ('assert, [*relation*])

DEFINITION:

assert-relation (*form*) = arg* (*form*)₀

DEFINITION:

invariant-p (*x*)

=

listp (*x*) ∧ (car (*x*) = 'invariant)

DEFINITION:

mk-invariant (*relation*) = cons ('invariant, [*relation*])

DEFINITION:

invariant-relation (*form*) = arg* (*form*)₀

DEFINITION:

transition-p (*x*)

=

listp (*x*) ∧ (car (*x*) = 'transition)

DEFINITION:

mk-transition (*relation*) = cons ('transition, [*relation*])

DEFINITION:

transition-relation (*form*) = arg* (*form*)₀

DEFINITION:

return-relation-p (*x*)

=

listp (*x*) ∧ (car (*x*) = 'return-relation)

DEFINITION:

mk-return-relation (*var*, *relation*)

=

cons ('return-relation, [*var*, *relation*])

DEFINITION:

return-relation-var (*form*) = arg* (*form*)₀

DEFINITION:

return-relation-relation (*form*) = arg* (*form*)₁

DEFINITION:

return-value-p (*x*)

=

listp (*x*) ∧ (car (*x*) = 'return-value)

DEFINITION:

mk-return-value (*relation*)

=

cons ('return-value, [*relation*])

DEFINITION:

return-value-relation (*form*) = arg* (*form*)₀

DEFINITION:

$\text{defaxiom-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'defaxiom})$

DEFINITION:

$\text{mk-defaxiom}(id, relation) = \text{cons}(\text{'defaxiom}, [id, relation])$

DEFINITION:

$\text{defaxiom-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{defaxiom-relation}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{defthm-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'defthm})$

DEFINITION:

$\text{mk-defthm}(id, relation) = \text{cons}(\text{'defthm}, [id, relation])$

DEFINITION:

$\text{defthm-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{defthm-relation}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{defun-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'defun})$

DEFINITION:

$\text{mk-defun}(id, fpl, relation) = \text{cons}(\text{'defun}, [id, fpl, relation])$

DEFINITION:

$\text{defun-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{defun-fpl}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{defun-relation}(form) = \text{arg}^*(form)_2$

DEFINITION:

$\text{subprogram-annotation-p}(x)$
 $=$
 $\text{assert-p}(x) \vee (\text{return-value-p}(x) \vee \text{return-relation-p}(x))$

DEFINITION:

$\text{if-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'if})$

DEFINITION:

$\text{mk-if}(test, then, else) = \text{cons}(\text{'if}, [test, then, else])$

DEFINITION:

$\text{if-test}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{if-then}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{if-else}(form) = \text{arg}^*(form)_2$

DEFINITION:

$\text{set-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'set})$

DEFINITION:

$\text{mk-set}(id, index, value) = \text{cons}(\text{'set}, [id, index, value])$

DEFINITION:

$\text{set-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{set-index}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{set-value}(form) = \text{arg}^*(form)_2$

DEFINITION:

$\text{get-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'get})$

DEFINITION:

$\text{mk-get}(id, index) = \text{cons}(\text{'get}, [id, index])$

DEFINITION:

$\text{get-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{get-index}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{assoc-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'assoc})$

DEFINITION:

$\text{mk-assoc}(x, y) = \text{cons}(\text{'assoc}, [x, y])$

DEFINITION:

$\text{assoc-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{assoc-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{lookup-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'lookup})$

DEFINITION:

$\text{mk-lookup}(x, y) = \text{cons}(\text{'lookup}, [x, y])$

DEFINITION:

$\text{lookup-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{lookup-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{in-range-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'in-range})$

DEFINITION:

$\text{mk-in-range}(x, y) = \text{cons}(\text{'in-range}, [x, y])$

DEFINITION:

$\text{in-range-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{in-range-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{not-in-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'not-in})$

DEFINITION:

$\text{mk-not-in}(x, y) = \text{cons}(\text{'not-in}, [x, y])$

DEFINITION:

$\text{not-in-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{not-in-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{in-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{in})$

DEFINITION:

$\text{mk-in}(x, y) = \text{cons}(' \text{in}, [x, y])$

DEFINITION:

$\text{in-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{in-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{iff-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{iff})$

DEFINITION:

$\text{mk-iff}(x, y) = \text{cons}(' \text{iff}, [x, y])$

DEFINITION:

$\text{iff-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{iff-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{implies-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{implies})$

DEFINITION:

$\text{mk-implies}(x, y) = \text{cons}(' \text{implies}, [x, y])$

DEFINITION:

$\text{implies-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{implies-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{and-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{and})$

DEFINITION:

$\text{mk-and}(x, y) = \text{cons}(' \text{and}, [x, y])$

DEFINITION:

$\text{and-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{and-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{or-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{or})$

DEFINITION:

$\text{mk-or}(x, y) = \text{cons}(' \text{or}, [x, y])$

DEFINITION:

$\text{or-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{or-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{=p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{=})$

DEFINITION:

$\text{mk-}(x, y) = \text{cons}(' \text{=}, [x, y])$

DEFINITION:

$$=x(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$=y(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{ne-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = 'ne)$$

DEFINITION:

$$\text{mk-ne}(x, y) = \text{cons}('ne, [x, y])$$

DEFINITION:

$$\text{ne-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{ne-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{lt-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = 'lt)$$

DEFINITION:

$$\text{mk-lt}(x, y) = \text{cons}('lt, [x, y])$$

DEFINITION:

$$\text{lt-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{lt-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{le-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = 'le)$$

DEFINITION:

$$\text{mk-le}(x, y) = \text{cons}('le, [x, y])$$

DEFINITION:

$$\text{le-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{le-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{gt-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = 'gt)$$

DEFINITION:

$$\text{mk-gt}(x, y) = \text{cons}('gt, [x, y])$$

DEFINITION:

$$\text{gt-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{gt-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{ge-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = 'ge)$$

DEFINITION:

$$\text{mk-ge}(x, y) = \text{cons}('ge, [x, y])$$

DEFINITION:

$$\text{ge-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{ge-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$+-p(x) = \text{listp}(x) \wedge (\text{car}(x) = '+')$$

DEFINITION:

$$\text{mk-+}(x, y) = \text{cons}(' +, [x, y])$$

DEFINITION:

$$+-x(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$+-y(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$--p(x) = \text{listp}(x) \wedge (\text{car}(x) = '-')$$

DEFINITION:

$$\text{mk--}(x, y) = \text{cons}('-', [x, y])$$

DEFINITION:

$$--x(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$--y(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$*-p(x) = \text{listp}(x) \wedge (\text{car}(x) = '*')$$

DEFINITION:

$$\text{mk-*}(x, y) = \text{cons}('*', [x, y])$$

DEFINITION:

$$*-x(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$*-y(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$/-p(x) = \text{listp}(x) \wedge (\text{car}(x) = '/')$$

DEFINITION:

$$\text{mk-}/(x, y) = \text{cons}('/', [x, y])$$

DEFINITION:

$$/-x(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$/-y(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{mod-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{mod})$$

DEFINITION:

$$\text{mk-mod}(x, y) = \text{cons}(' \text{mod}, [x, y])$$

DEFINITION:

$$\text{mod-x}(form) = \text{arg}^*(form)_0$$

DEFINITION:

$$\text{mod-y}(form) = \text{arg}^*(form)_1$$

DEFINITION:

$$\text{rem-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{rem})$$

DEFINITION:

$$\text{mk-rem}(x, y) = \text{cons}(' \text{rem}, [x, y])$$

DEFINITION:

$\text{rem-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{rem-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{expt-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{expt})$

DEFINITION:

$\text{mk-expt}(x, y) = \text{cons}(' \text{expt}, [x, y])$

DEFINITION:

$\text{expt-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{expt-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{abs-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{abs})$

DEFINITION:

$\text{mk-abs}(x) = \text{cons}(' \text{abs}, [x])$

DEFINITION:

$\text{abs-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{not-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{not})$

DEFINITION:

$\text{mk-not}(x) = \text{cons}(' \text{not}, [x])$

DEFINITION:

$\text{not-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{minus-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{minus})$

DEFINITION:

$\text{mk-minus}(x) = \text{cons}(' \text{minus}, [x])$

DEFINITION:

$\text{minus-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{append-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{append})$

DEFINITION:

$\text{mk-append}(x, y) = \text{cons}(' \text{append}, [x, y])$

DEFINITION:

$\text{append-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{append-y}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{cons-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{cons})$

DEFINITION:

$\text{mk-cons}(x) = \text{cons}(' \text{cons}, [x])$

DEFINITION:

$\text{cons-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{car-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{car})$

DEFINITION:

$\text{mk-car}(x) = \text{cons}(' \text{car}, [x])$

DEFINITION:

$\text{car-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{cdr-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = ' \text{cdr})$

DEFINITION:

$\text{mk-cdr}(x) = \text{cons}(' \text{cdr}, [x])$

DEFINITION:

$\text{cdr-x}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{lexpr-p}(x)$

=

- symbolp(x)
- ∨ lliteral-p(x)
- ∨ list-p(x)
- ∨ in-p(x)
- ∨ not-in-p(x)
- ∨ iff-p(x)
- ∨ implies-p(x)
- ∨ and-p(x)
- ∨ or-p(x)
- ∨ not-p(x)
- ∨ =-p(x)
- ∨ range-p(x)
- ∨ ne-p(x)
- ∨ lt-p(x)
- ∨ le-p(x)
- ∨ gt-p(x)
- ∨ ge-p(x)
- ∨ +-p(x)
- ∨ --p(x)
- ∨ *-p(x)
- ∨ /-p(x)
- ∨ mod-p(x)
- ∨ rem-p(x)
- ∨ expt-p(x)
- ∨ abs-p(x)
- ∨ minus-p(x)
- ∨ assoc-p(x)
- ∨ lookup-p(x)
- ∨ in-range-p(x)
- ∨ append-p(x)
- ∨ cons-p(x)
- ∨ car-p(x)
- ∨ cdr-p(x)
- ∨ if-p(x)
- ∨ set-p(x)
- ∨ get-p(x)
- ∨ instate-p(x)
- ∨ outstate-p(x)

DEFINITION:

$\text{choice-p}(x) = \text{range-p}(x) \vee (\text{expr-p}(x) \vee \text{others-p}(x))$

DEFINITION:

$\text{choices-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'choices'})$

DEFINITION:

$\text{mk-choices}(args) = \text{cons}(\text{'choices'}, args)$

DEFINITION:

$\text{constant-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'constant'})$

DEFINITION:

$\text{mk-constant} = [\text{'constant'}]$

DEFINITION:

$\text{variable-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'variable'})$

DEFINITION:

$\text{mk-variable} = [\text{'variable'}]$

DEFINITION:

$\text{pmode-p}(x) = \text{constant-p}(x) \vee \text{variable-p}(x)$

DEFINITION:

$\text{fp-spec-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'fp-spec'})$

DEFINITION:

$\text{mk-fp-spec}(id, mode, type) = \text{cons}(\text{'fp-spec'}, [id, mode, type])$

DEFINITION:

$\text{fp-spec-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{fp-spec-mode}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{fp-spec-type}(form) = \text{arg}^*(form)_2$

DEFINITION:

$\text{fpl-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'fpl'})$

DEFINITION:

$\text{mk-fpl}(args) = \text{cons}(\text{'fpl'}, args)$

DEFINITION:

$\text{number-decl-p}(x)$

$=$

$\text{listp}(x) \wedge (\text{car}(x) = \text{'number-decl'})$

DEFINITION:

$\text{mk-number-decl}(id, mode, body)$

$=$

$\text{cons}(\text{'number-decl'}, [id, mode, body])$

DEFINITION:

$\text{number-decl-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{number-decl-mode}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{number-decl-body}(form) = \text{arg}^*(form)_2$

DEFINITION:

$\text{object-decl-p}(x)$
 $=$
 $\text{listp}(x) \wedge (\text{car}(x) = \text{'object-decl})$

DEFINITION:
 $\text{mk-object-decl}(id, mode, type, body)$
 $=$
 $\text{cons}(\text{'object-decl}, [id, mode, type, body])$

DEFINITION:
 $\text{object-decl-id}(form) = \text{arg}^*(form)_0$

DEFINITION:
 $\text{object-decl-mode}(form) = \text{arg}^*(form)_1$

DEFINITION:
 $\text{object-decl-type}(form) = \text{arg}^*(form)_2$

DEFINITION:
 $\text{object-decl-body}(form) = \text{arg}^*(form)_3$

DEFINITION:
 $\text{procedure-p}(x)$
 $=$
 $\text{listp}(x) \wedge (\text{car}(x) = \text{'procedure})$

DEFINITION:
 $\text{mk-procedure}(id, params, return, body, spec)$
 $=$
 $\text{cons}(\text{'procedure}, [id, params, return, body, spec])$

DEFINITION:
 $\text{procedure-id}(form) = \text{arg}^*(form)_0$

DEFINITION:
 $\text{procedure-params}(form) = \text{arg}^*(form)_1$

DEFINITION:
 $\text{procedure-return}(form) = \text{arg}^*(form)_2$

DEFINITION:
 $\text{procedure-body}(form) = \text{arg}^*(form)_3$

DEFINITION:
 $\text{procedure-spec}(form) = \text{arg}^*(form)_4$

DEFINITION:
 $\text{function-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'function})$

DEFINITION:
 $\text{mk-function}(id, params, return, body, spec)$
 $=$
 $\text{cons}(\text{'function}, [id, params, return, body, spec])$

DEFINITION:
 $\text{function-id}(form) = \text{arg}^*(form)_0$

DEFINITION:
 $\text{function-params}(form) = \text{arg}^*(form)_1$

DEFINITION:
 $\text{function-return}(form) = \text{arg}^*(form)_2$

DEFINITION:
 $\text{function-body}(form) = \text{arg}^*(form)_3$

DEFINITION:

$\text{function-spec}(form) = \text{arg}^*(form)_4$

DEFINITION:

$\text{exception-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'exception})$

DEFINITION:

$\text{mk-exception}(id) = \text{cons}(\text{'exception}, [id])$

DEFINITION:

$\text{exception-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{subprogram-p}(x) = \text{function-p}(x) \vee \text{procedure-p}(x)$

DEFINITION:

$\text{use-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'use})$

DEFINITION:

$\text{mk-use}(args) = \text{cons}(\text{'use}, args)$

DEFINITION:

$\text{inner-decl-p}(x)$

=

$\text{object-decl-p}(x) \vee (\text{number-decl-p}(x) \vee (\text{assert-p}(x) \vee \text{invariant-p}(x)))$

DEFINITION:

$\text{inner-decls-p}(x)$

=

$\text{consp}(x) \wedge (\text{car}(x) = \text{'inner-decls})$

DEFINITION:

$\text{mk-inner-decls}(args) = \text{cons}(\text{'inner-decls}, args)$

DEFINITION:

$\text{rename-pkg-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'rename-pkg})$

DEFINITION:

$\text{mk-rename-pkg}(new, old) = \text{cons}(\text{'rename-pkg}, [new, old])$

DEFINITION:

$\text{rename-pkg-new}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{rename-pkg-old}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{rename-sub-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'rename-sub})$

DEFINITION:

$\text{mk-rename-sub}(new, old) = \text{cons}(\text{'rename-sub}, [new, old])$

DEFINITION:

$\text{rename-sub-new}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{rename-sub-old}(form) = \text{arg}^*(form)_1$

DEFINITION:

rename-obj-p(x)
 =
 listp(x) \wedge (car(x) = 'rename-obj)

DEFINITION:

mk-rename-obj(new , $type$, old)
 =
 cons('rename-obj, [new , $type$, old])

DEFINITION:

rename-obj-new($form$) = arg*($form$)₀

DEFINITION:

rename-obj-type($form$) = arg*($form$)₁

DEFINITION:

rename-obj-old($form$) = arg*($form$)₂

DEFINITION:

rename-p(x)
 =
 rename-pkg-p(x) \vee (rename-sub-p(x) \vee rename-obj-p(x))

DEFINITION:

decl-p(x)
 =
 inner-decl-p(x)
 \vee (subprogram-p(x)
 \vee (type-decl-p(x)
 \vee (subtype-decl-p(x)
 \vee (rename-p(x)
 \vee (defun-p(x) \vee (defthm-p(x) \vee defaxiom-p(x))))))

DEFINITION:

decls-p(x) = consp(x) \wedge (car(x) = 'decls)

DEFINITION:

mk-decls($args$) = cons('decls, $args$)

DEFINITION:

raise-p(x) = consp(x) \wedge (car(x) = 'raise)

DEFINITION:

mk-raise = ['raise]

DEFINITION:

null-p(x) = consp(x) \wedge (car(x) = 'null)

DEFINITION:

mk-null = ['null]

DEFINITION:

assign-p(x) = listp(x) \wedge (car(x) = 'assign)

DEFINITION:

mk-assign(var , $value$) = cons('assign, [var , $value$])

DEFINITION:

assign-var($form$) = arg*($form$)₀

DEFINITION:

assign-value($form$) = arg*($form$)₁

DEFINITION:
proc-call-p(x)
= $\text{listp}(x) \wedge (\text{car}(x) = \text{'proc-call})$

DEFINITION:
mk-proc-call($id, actuals$) = $\text{cons}(\text{'proc-call}, [id, actuals])$

DEFINITION:
proc-call-id($form$) = $\text{arg}^*(form)_0$

DEFINITION:
proc-call-actuals($form$) = $\text{arg}^*(form)_1$

DEFINITION:
return-p(x) = $\text{listp}(x) \wedge (\text{car}(x) = \text{'return})$

DEFINITION:
mk-return($value$) = $\text{cons}(\text{'return}, [value])$

DEFINITION:
return-value($form$) = $\text{arg}^*(form)_0$

DEFINITION:
exit-p(x) = $\text{consp}(x) \wedge (\text{car}(x) = \text{'exit})$

DEFINITION:
mk-exit = $[\text{'exit}]$

DEFINITION:
sl-p(x) = $\text{consp}(x) \wedge (\text{car}(x) = \text{'sl})$

DEFINITION:
mk-sl($args$) = $\text{cons}(\text{'sl}, args)$

DEFINITION:
while-loop-p(x)
= $\text{listp}(x) \wedge (\text{car}(x) = \text{'while-loop})$

DEFINITION:
mk-while-loop($test, statements$)
= $\text{cons}(\text{'while-loop}, [test, statements])$

DEFINITION:
while-loop-test($form$) = $\text{arg}^*(form)_0$

DEFINITION:
while-loop-statements($form$) = $\text{arg}^*(form)_1$

DEFINITION:
loop-p(x) = $\text{listp}(x) \wedge (\text{car}(x) = \text{'loop})$

DEFINITION:
mk-loop($statements$) = $\text{cons}(\text{'loop}, [statements])$

DEFINITION:
loop-statements($form$) = $\text{arg}^*(form)_0$

DEFINITION:
for-loop-p(x) = $\text{listp}(x) \wedge (\text{car}(x) = \text{'for-loop})$

DEFINITION:
mk-for-loop($var, range, statements$)

=
`cons ('for-loop, [var, range, statements])`

DEFINITION:
`for-loop-var (form) = arg* (form)0`

DEFINITION:
`for-loop-range (form) = arg* (form)1`

DEFINITION:
`for-loop-statements (form) = arg* (form)2`

DEFINITION:
`reverse-for-loop-p (x)`
 =
`listp (x) ∧ (car (x) = 'reverse-for-loop)`

DEFINITION:
`mk-reverse-for-loop (var, range, statements)`
 =
`cons ('reverse-for-loop, [var, range, statements])`

DEFINITION:
`reverse-for-loop-var (form) = arg* (form)0`

DEFINITION:
`reverse-for-loop-range (form) = arg* (form)1`

DEFINITION:
`reverse-for-loop-statements (form) = arg* (form)2`

DEFINITION:
`loop-stmt-p (x)`
 =
`while-loop-p (x)`
 \vee `(loop-p (x) ∨ (for-loop-p (x) ∨ reverse-for-loop-p (x)))`

DEFINITION:
`block-p (x) = listp (x) ∧ (car (x) = 'block)`

DEFINITION:
`mk-block (decls, handler, body)`
 =
`cons ('block, [decls, handler, body])`

DEFINITION:
`block-decls (form) = arg* (form)0`

DEFINITION:
`block-handler (form) = arg* (form)1`

DEFINITION:
`block-body (form) = arg* (form)2`

DEFINITION:
`ifarm-p (x) = listp (x) ∧ (car (x) = 'ifarm)`

DEFINITION:
`mk-ifarm (test, statements) = cons ('ifarm, [test, statements])`

DEFINITION:
`ifarm-test (form) = arg* (form)0`

DEFINITION:
`ifarm-statements (form) = arg* (form)1`

DEFINITION:

$\text{if-stmt-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'if-stmt})$

DEFINITION:

$\text{mk-if-stmt}(args) = \text{cons}(\text{'if-stmt}, args)$

DEFINITION:

$\text{casearm-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'casearm})$

DEFINITION:

$\text{mk-casearm}(test, statements)$

=

$\text{cons}(\text{'casearm}, [test, statements])$

DEFINITION:

$\text{casearm-test}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{casearm-statements}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{casearms-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'casearms})$

DEFINITION:

$\text{mk-casearms}(args) = \text{cons}(\text{'casearms}, args)$

DEFINITION:

$\text{case-stmt-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'case-stmt})$

DEFINITION:

$\text{mk-case-stmt}(test, arms) = \text{cons}(\text{'case-stmt}, [test, arms])$

DEFINITION:

$\text{case-stmt-test}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{case-stmt-arms}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{st-compound-p}(x)$

=

$\text{if-stmt-p}(x) \vee \text{case-stmt-p}(x) \vee \text{loop-stmt-p}(x) \vee \text{block-p}(x))$

DEFINITION:

$\text{constrained-st-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'constrained-st})$

DEFINITION:

$\text{mk-constrained-st}(relation, stmt)$

=

$\text{cons}(\text{'constrained-st}, [relation, stmt])$

DEFINITION:

$\text{constrained-st-relation}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{constrained-st-stmt}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{st-simple-p}(x)$

=

$\text{null-p}(x)$
 $\vee (\text{assign-p}(x)$
 $\vee (\text{proc-call-p}(x) \vee (\text{return-p}(x) \vee (\text{exit-p}(x) \vee \text{raise-p}(x))))))$

DEFINITION:

$\text{ada-st-p}(x) = \text{st-simple-p}(x) \vee \text{st-compound-p}(x)$

DEFINITION:

$\text{st-p}(x)$

=

$\text{ada-st-p}(x) \vee (\text{constrained-st-p}(x) \vee \text{assert-p}(x))$

DEFINITION:

$\text{ids-p}(x) = \text{consp}(x) \wedge (\text{car}(x) = \text{'ids})$

DEFINITION:

$\text{mk-ids}(args) = \text{cons}(\text{'ids}, args)$

DEFINITION:

$\text{compilation-p}(x)$

=

$\text{consp}(x) \wedge (\text{car}(x) = \text{'compilation})$

DEFINITION:

$\text{mk-compilation}(args) = \text{cons}(\text{'compilation}, args)$

DEFINITION:

$\text{comp-unit-p}(x)$

=

$\text{listp}(x) \wedge (\text{car}(x) = \text{'comp-unit})$

DEFINITION:

$\text{mk-comp-unit}(unit, clause) = \text{cons}(\text{'comp-unit}, [unit, clause])$

DEFINITION:

$\text{comp-unit-unit}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{comp-unit-clause}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{context-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'context})$

DEFINITION:

$\text{mk-context}(with, use) = \text{cons}(\text{'context}, [with, use])$

DEFINITION:

$\text{context-with}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{context-use}(form) = \text{arg}^*(form)_1$

DEFINITION:

$\text{package-p}(x) = \text{listp}(x) \wedge (\text{car}(x) = \text{'package})$

DEFINITION:

$\text{mk-package}(id, outer, private, inner, body)$

=

$\text{cons}(\text{'package}, [id, outer, private, inner, body])$

DEFINITION:

$\text{package-id}(form) = \text{arg}^*(form)_0$

DEFINITION:

$\text{package-outer}(form) = \text{arg}^*(form)_1$

DEFINITION:

package-private (*form*) = arg* (*form*)₂

DEFINITION:

package-inner (*form*) = arg* (*form*)₃

DEFINITION:

package-body (*form*) = arg* (*form*)₄

DEFINITION:

library-unit-p (*x*) = subprogram-p (*x*) ∨ package-p (*x*)

DEFINITION:

compilation-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif comp-unit-p (car (*l*)) **then** compilation-args-p (cdr (*l*))
else nil
fi

DEFINITION:

ids-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif id-p (car (*l*)) **then** ids-args-p (cdr (*l*))
else nil
fi

DEFINITION:

casearms-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif casearm-p (car (*l*)) **then** casearms-args-p (cdr (*l*))
else nil
fi

DEFINITION:

if-stmt-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif ifarm-p (car (*l*)) **then** if-stmt-args-p (cdr (*l*))
else nil
fi

DEFINITION:

sl-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif st-p (car (*l*)) **then** sl-args-p (cdr (*l*))
else nil
fi

DEFINITION:

decls-args-p (*l*)

=

if ¬ consp (*l*) **then t**
elseif decl-p (car (*l*)) **then** decls-args-p (cdr (*l*))
else nil
fi

DEFINITION:

inner-decls-args-p (*l*)

```

=
if  $\neg$  consp(l) then t
  elseif inner-decl-p(car(l)) then inner-decls-args-p(cdr(l))
  else nil
fi

DEFINITION:
use-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif id-p(car(l)) then use-args-p(cdr(l))
  else nil
fi

DEFINITION:
fpl-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif fp-spec-p(car(l)) then fpl-args-p(cdr(l))
  else nil
fi

DEFINITION:
choices-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif choice-p(car(l)) then choices-args-p(cdr(l))
  else nil
fi

DEFINITION:
apl-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif expr-p(car(l)) then apl-args-p(cdr(l))
  else nil
fi

DEFINITION:
designator-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif id-p(car(l)) then designator-args-p(cdr(l))
  else nil
fi

DEFINITION:
aggregate-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif aggregate-arm-p(car(l)) then aggregate-args-p(cdr(l))
  else nil
fi

DEFINITION:
record-type-args-p(l)
=
if  $\neg$  consp(l) then t
  elseif field-spec-p(car(l)) then record-type-args-p(cdr(l))
  else nil
fi

```

DEFINITION:
enumeration-args-p (*l*)
=
if \neg consp (*l*) **then t**
elseif enumeration-literal-p (car (*l*)) **then** enumeration-args-p (cdr (*l*))
else nil
fi

DEFINITION:
list-args-p (*l*)
=
if \neg consp (*l*) **then t**
elseif expr-p (car (*l*)) **then** list-args-p (cdr (*l*))
else nil
fi

DEFINITION:
prefix-p-body (*form*)
=
let operator be opr (*form*)
in
case match operator:
case \equiv 'package **then**
id-p (package-id (*form*))
 \wedge (((package-outer (*form*) = **nil**)
 \vee decls-p (package-outer (*form*)))
 \wedge (((package-private (*form*) = **nil**)
 \vee decls-p (package-private (*form*)))
 \wedge (((package-inner (*form*) = **nil**)
 \vee decls-p (package-inner (*form*)))
 \wedge ((package-body (*form*) = **nil**)
 \vee sl-p (package-body (*form*))))))
case \equiv 'context **then**
((context-with (*form*) = **nil**) \vee ids-p (context-with (*form*)))
 \wedge ((context-use (*form*) = **nil**)
 \vee ids-p (context-use (*form*)))
case \equiv 'comp-unit **then**
library-unit-p (comp-unit-unit (*form*))
 \wedge context-p (comp-unit-clause (*form*))
case \equiv 'compilation **then** compilation-args-p (arg* (*form*))
case \equiv 'ids **then** ids-args-p (arg* (*form*))
case \equiv 'constrained-st **then**
transition-p (constrained-st-relation (*form*))
 \wedge st-compound-p (constrained-st-stmt (*form*))
case \equiv 'case-stmt **then**
expr-p (case-stmt-test (*form*))
 \wedge casearms-p (case-stmt-arms (*form*))
case \equiv 'casearms **then** casearms-args-p (arg* (*form*))
case \equiv 'casearm **then** expr-p (casearm-test (*form*))
 \wedge sl-p (casearm-statements (*form*))
case \equiv 'if-stmt **then** if-stmt-args-p (arg* (*form*))
case \equiv 'ifarm **then** expr-p (ifarm-test (*form*))
 \wedge sl-p (ifarm-statements (*form*))
case \equiv 'block **then**
((block-decls (*form*) = **nil**)
 \vee inner-decls-p (block-decls (*form*)))
 \wedge (((block-handler (*form*) = **nil**)
 \vee sl-p (block-handler (*form*)))
 \wedge sl-p (block-body (*form*)))

```

case ≡ 'reverse-for-loop then
  id-p(reverse-for-loop-var (form))
  ^ ( ( range-p(reverse-for-loop-range (form))
        ^ sl-p(reverse-for-loop-statements (form)))
case ≡ 'for-loop then
  id-p(for-loop-var (form))
  ^ ( ( range-p(for-loop-range (form))
        ^ sl-p(for-loop-statements (form)))
case ≡ 'loop then sl-p(loop-statements (form))
case ≡ 'while-loop then
  expr-p(while-loop-test (form))
  ^ sl-p(while-loop-statements (form))
case ≡ 'sl then sl-args-p(arg* (form))
case ≡ 'exit then t
case ≡ 'return then (return-value (form) = nil)
                    ∨ expr-p(return-value (form))
case ≡ 'proc-call then id-p(proc-call-id (form))
                    ^ apl-p(proc-call-actuals (form))
case ≡ 'assign then name-p(assign-var (form))
                    ^ expr-p(assign-value (form))
case ≡ 'null then t
case ≡ 'raise then t
case ≡ 'decls then decls-args-p(arg* (form))
case ≡ 'rename-obj then
  id-p(rename-obj-new (form))
  ^ ( ( type-p(rename-obj-type (form))
        ^ id-p(rename-obj-old (form)))
case ≡ 'rename-sub then
  subprogram-p(rename-sub-new (form))
  ^ id-p(rename-sub-old (form))
case ≡ 'rename-pkg then id-p(rename-pkg-new (form))
                    ^ id-p(rename-pkg-old (form))
case ≡ 'inner-decls then inner-decls-args-p(arg* (form))
case ≡ 'use then use-args-p(arg* (form))
case ≡ 'exception then id-p(exception-id (form))
case ≡ 'function then
  id-p(function-id (form))
  ^ ( ( fpl-p(function-params (form))
        ^ ( id-p(function-return (form))
            ^ ( ( (function-body (form) = nil)
                  ∨ block-p(function-body (form)))
            ^ ( (function-spec (form) = nil)
                  ∨ subprogram-annotation-p(function-spec (form))))))
case ≡ 'procedure then
  id-p(procedure-id (form))
  ^ ( ( fpl-p(procedure-params (form))
        ^ ( ( (procedure-return (form) = nil)
              ∨ id-p(procedure-return (form)))
        ^ ( ( (procedure-body (form) = nil)
              ∨ block-p(procedure-body (form)))
        ^ ( (procedure-spec (form) = nil)
              ∨ subprogram-annotation-p(procedure-spec (form))))))
case ≡ 'object-decl then
  id-p(object-decl-id (form))
  ^ ( ( pmode-p(object-decl-mode (form))
        ^ ( subtype-p(object-decl-type (form))
            ^ ( (object-decl-body (form) = nil)
                  ∨ expr-p(object-decl-body (form))))))

```

```

case ≡ 'number-decl then
  id-p (number-decl-id (form))
  ^ (  pmode-p (number-decl-mode (form))
      ^  expr-p (number-decl-body (form)))
case ≡ 'fpl then fpl-args-p (arg* (form))
case ≡ 'fp-spec then
  id-p (fp-spec-id (form))
  ^ (  pmode-p (fp-spec-mode (form))
      ^  type-p (fp-spec-type (form)))
case ≡ 'variable then t
case ≡ 'constant then t
case ≡ 'choices then choices-args-p (arg* (form))
case ≡ 'cdr then lexpr-p (cdr-x (form))
case ≡ 'car then lexpr-p (car-x (form))
case ≡ 'cons then lexpr-p (cons-x (form))
case ≡ 'append then lexpr-p (append-x (form))
  ^ lexpr-p (append-y (form))
case ≡ 'minus then lexpr-p (minus-x (form))
case ≡ 'not then lexpr-p (not-x (form))
case ≡ 'abs then lexpr-p (abs-x (form))
case ≡ 'expt then lexpr-p (expt-x (form))
  ^ lexpr-p (expt-y (form))
case ≡ 'rem then lexpr-p (rem-x (form))
  ^ lexpr-p (rem-y (form))
case ≡ 'mod then lexpr-p (mod-x (form))
  ^ lexpr-p (mod-y (form))
case ≡ '/ then lexpr-p (/x (form)) ^ lexpr-p (/y (form))
case ≡ '* then lexpr-p (*-x (form)) ^ lexpr-p (*-y (form))
case ≡ '- then lexpr-p (--x (form)) ^ lexpr-p (--y (form))
case ≡ '+ then lexpr-p (+x (form)) ^ lexpr-p (+y (form))
case ≡ 'ge then lexpr-p (ge-x (form))
  ^ lexpr-p (ge-y (form))
case ≡ 'gt then lexpr-p (gt-x (form))
  ^ lexpr-p (gt-y (form))
case ≡ 'le then lexpr-p (le-x (form))
  ^ lexpr-p (le-y (form))
case ≡ 'lt then lexpr-p (lt-x (form))
  ^ lexpr-p (lt-y (form))
case ≡ 'ne then lexpr-p (ne-x (form))
  ^ lexpr-p (ne-y (form))
case ≡ '= then lexpr-p (=x (form)) ^ lexpr-p (=y (form))
case ≡ 'or then lexpr-p (or-x (form))
  ^ lexpr-p (or-y (form))
case ≡ 'and then lexpr-p (and-x (form))
  ^ lexpr-p (and-y (form))
case ≡ 'implies then lexpr-p (implies-x (form))
  ^ lexpr-p (implies-y (form))
case ≡ 'iff then lexpr-p (iff-x (form))
  ^ lexpr-p (iff-y (form))
case ≡ 'in then lexpr-p (in-x (form))
  ^ lexpr-p (in-y (form))
case ≡ 'not-in then lexpr-p (not-in-x (form))
  ^ lexpr-p (not-in-y (form))
case ≡ 'in-range then lexpr-p (in-range-x (form))
  ^ lexpr-p (in-range-y (form))
case ≡ 'lookup then lexpr-p (lookup-x (form))
  ^ lexpr-p (lookup-y (form))
case ≡ 'assoc then lexpr-p (assoc-x (form))

```



```

                                ^ lexpr-p (assoc-y (form))
case ≡ 'get then    lexpr-p (get-id (form))
                                ^ lexpr-p (get-index (form))
case ≡ 'set then
    lexpr-p (set-id (form))
  ^ (lexpr-p (set-index (form)) ^ lexpr-p (set-value (form)))
case ≡ 'if then
    lexpr-p (if-test (form))
  ^ (lexpr-p (if-then (form)) ^ lexpr-p (if-else (form)))
case ≡ 'defun then
    symbolp (defun-id (form))
  ^ ( symbolsp (defun-fpl (form))
    ^ lexpr-p (defun-relation (form)))
case ≡ 'defthm then    symbolp (defthm-id (form))
                                ^ lexpr-p (defthm-relation (form))
case ≡ 'defaxiom then    symbolp (defaxiom-id (form))
                                ^ lexpr-p (defaxiom-relation (form))
case ≡ 'return-value then lexpr-p (return-value-relation (form))
case ≡ 'return-relation then
    symbolp (return-relation-var (form))
  ^ lexpr-p (return-relation-relation (form))
case ≡ 'transition then lexpr-p (transition-relation (form))
case ≡ 'invariant then lexpr-p (invariant-relation (form))
case ≡ 'assert then lexpr-p (assert-relation (form))
case ≡ 'outstate then lexpr-p (outstate-expr (form))
case ≡ 'instate then lexpr-p (instate-expr (form))
case ≡ 'function-call then
    id-p (function-call-id (form))
  ^ apl-p (function-call-actuals (form))
case ≡ 'op-expr then    id-p (op-expr-id (form))
                                ^ apl-p (op-expr-actuals (form))
case ≡ 'apl then apl-args-p (arg* (form))
case ≡ 'dot-qual-1 then
    name-p (dot-qual-1-root (form))
  ^ symbolp (dot-qual-1-component (form))
case ≡ 'designator then designator-args-p (arg* (form))
case ≡ 'selected-component then
    expr-p (selected-component-root (form))
  ^ symbolp (selected-component-field (form))
case ≡ 'apply-1 then    expr-p (apply-1-root (form))
                                ^ apl-p (apply-1-args (form))
case ≡ 'indexed-component then
    expr-p (indexed-component-root (form))
  ^ expr-p (indexed-component-index (form))
case ≡ 'aggregate then aggregate-args-p (arg* (form))
case ≡ 'aggregate-pos then expr-p (aggregate-pos-value (form))
case ≡ 'aggregate-choice then
    choices-p (aggregate-choice-choice (form))
  ^ expr-p (aggregate-choice-value (form))
case ≡ 'type-convert then
    type-p (type-convert-type (form))
  ^ expr-p (type-convert-value (form))
case ≡ 'qualified then    type-p (qualified-type (form))
                                ^ expr-p (qualified-value (form))
case ≡ 'subtype-decl then
    id-p (subtype-decl-id (form))
  ^ subtype-p (subtype-decl-decl (form))
case ≡ 'type-decl then

```

```

    id-p (type-decl-id (form))
  ^ ( (type-decl-decl (form) = nil)
    v type-p (type-decl-decl (form)))
case ≡ 'array-type then
  type-mark-p (array-type-index (form))
  ^ type-p (array-type-elements (form))
case ≡ 'record-type then record-type-args-p (arg* (form))
case ≡ 'field-spec then symbolp (field-spec-id (form))
  ^ type-p (field-spec-decl (form))
case ≡ 'range then expr-p (range-from (form))
  ^ expr-p (range-to (form))
case ≡ 'attribute then id-p (attribute-root (form))
  ^ id-p (attribute-attr (form))
case ≡ 'type-mark then
  id-p (type-mark-type (form))
  ^ constraint-p (type-mark-constraint (form))
case ≡ 'enumeration then enumeration-args-p (arg* (form))
case ≡ 'list then list-args-p (arg* (form))
case ≡ 'false then t
case ≡ 'true then t
case ≡ 'id then
  symbolp (id-root (form))
  ^ ((id-uid (form) = nil) v integerp (id-uid (form)))
case ≡ 'unconstrained then t
case ≡ 'others then t
case ≡ 'error then listp (error-form (form))
  ^ stringp (error-message (form))

otherwise nil
endcase

```

THEOREM: strip-cdrs-le :linear
 $\text{acl2-count}(\text{strip-cdrs}(x)) \leq \text{acl2-count}(x)$

THEOREM: strip-cdrs-lt :linear
 $\text{acl2-count}(\text{strip-cdrs}(x)) < (1 + \text{acl2-count}(x))$

THEOREM: top-literal-p-count :linear
 $\text{acl2-count}(\text{strip-cdrs}(\text{cdr}(\text{form})))$
 $< (1 + \text{acl2-count}(\text{car}(\text{form})) + \text{acl2-count}(\text{cdr}(\text{form})))$

DEFINITION:

top-literal-p (*form*, *flag*)

=

if *flag*

then if $\neg \text{consp}(\text{form})$ **then** *form* = nil
else top-literal-p (car (*form*), nil)
 ^ top-literal-p (cdr (*form*), t)

fi

elseif booleanp (*form*) **then** t
elseif integerp (*form*) **then** t
elseif characterp (*form*) **then** standard-char-p (*form*)
elseif array-literal-p (*form*) **then** top-literal-p (range (*form*), t)
elseif record-literal-p (*form*) **then** top-literal-p (range (*form*), t)
else nil

fi

Measure: acl2-count (*form*)

DEFINITION:

top-prefix-p (*form*, *flag*)

=

```

if flag
then if  $\neg$  consp (form) then form = nil
      else top-prefix-p (car (form), nil)
           $\wedge$  top-prefix-p (cdr (form), t)
      fi
elseif symbolp (form) then t
elseif integerp (form) then t
elseif characterp (form)  $\wedge$  standard-char-p (form) then t
elseif stringp (form) then t
elseif lexpr-p (form) then t
elseif ( $\neg$  consp (form))  $\vee$  ( $\neg$  listp (arg* (form))) then nil
elseif car (form)  $\in$  ' (defaxiom defun defthm) then t
elseif top-prefix-p (arg* (form), t) then prefix-p-body (form)
else nil
fi
Measure: acl2-count (form)

```

Disable forcing.

EVENT:

PROVE-AVA-PRIMITIVE-TYPE-DEFTHMS

```

( exit null raise
  variable constant false true unconstrained others
  package context comp-unit constrained-st case-stmt
  casearm ifarm block reverse-for-loop for-loop
  loop while-loop return proc-call assign
  rename-obj rename-sub rename-pkg exception function
  procedure object-decl number-decl fp-spec cdr
  car cons append minus not abs expt rem mod
  / * - + ge gt le lt ne = or and
  implies iff in not-in in-range lookup assoc
  get set if defun defthm defaxiom return-value
  return-relation transition invariant assert
  outstate instate function-call op-expr dot-qual-1
  selected-component apply-1 indexed-component
  aggregate-pos aggregate-choice type-convert qualified
  subtype-decl type-decl array-type field-spec range
  attribute type-mark id error)

```

CONSTANT:

ava-primitive-type-repeating =

```

' ( compilation ids
    casearms if-stmt sl decls
    inner-decls use fpl choices apl
    designator aggregate record-type
    enumeration list )

```

CONSTANT:

ava-primitive-type-equiv =

```

' ( library-unit st
    ada-st st-simple st-compound loop-stmt
    decl rename inner-decl subprogram
    pmode choice lexpr subprogram-annotation
    expr name defining-name aggregate-arm
    type constraint subtype predefined-type
    enumeration-literal lliteral literal
    boolean-literal )

```

DEFINE the theory **ava-primitive-type-fns-2** to be

```

suffix-fns(*ava-primitive-type-equiv*, '-p)
∪ suffix-fns(*ava-primitive-type-repeating*, '-args-p).

```

CONSTANT:

ava-primitive-type-mkfun =

```

' (mk-package mk-context
  mk-comp-unit mk-compilation mk-ids
  mk-constrained-st mk-case-stmt mk-casearms
  mk-casearm mk-if-stmt mk-ifarm mk-block
  mk-reverse-for-loop mk-for-loop mk-loop
  mk-while-loop mk-sl mk-exit mk-return mk-proc-call
  mk-assign mk-null mk-raise mk-decls mk-rename-obj
  mk-rename-sub mk-rename-pkg mk-inner-decls
  mk-use mk-exception mk-function
  mk-procedure mk-object-decl mk-number-decl mk-fpl
  mk-fp-spec mk-variable mk-constant mk-choices
  mk-cdr mk-car mk-cons mk-append
  mk-minus mk-not mk-abs mk-expt
  mk-rem mk-mod mk-/ mk-* mk--
  mk++ mk-ge mk-gt mk-le mk-lt
  mk-ne mk-= mk-or mk-and mk-implies
  mk-iff mk-in mk-not-in mk-in-range
  mk-lookup mk-assoc mk-get mk-set
  mk-if mk-defun mk-defthm mk-defaxiom
  mk-return-value mk-return-relation mk-transition
  mk-invariant mk-assert mk-outstate
  mk-instate mk-function-call mk-op-expr
  mk-apl mk-dot-qual-1 mk-designator
  mk-selected-component mk-apply-1
  mk-indexed-component mk-aggregate mk-aggregate-pos
  mk-aggregate-choice mk-type-convert mk-qualified
  mk-subtype-decl mk-type-decl mk-array-type
  mk-record-type mk-field-spec mk-range
  mk-attribute mk-type-mark mk-enumeration
  mk-list mk-false mk-true mk-id
  mk-unconstrained mk-others
  mk-error)

```

CONSTANT:

ava-primitive-type-argfun =

```

' (error-form error-message
  id-root id-uid type-mark-type
  type-mark-constraint attribute-root attribute-attr
  range-from range-to field-spec-id
  field-spec-decl array-type-index
  array-type-elements type-decl-id type-decl-decl
  subtype-decl-id subtype-decl-decl qualified-type
  qualified-value type-convert-type
  type-convert-value aggregate-choice-choice
  aggregate-choice-value aggregate-pos-value
  indexed-component-root indexed-component-index
  apply-1-root apply-1-args selected-component-root
  selected-component-field dot-qual-1-root
  dot-qual-1-component op-expr-id op-expr-actuals
  function-call-idfunction-call-actuals instate-expr
  outstate-expr assert-relation invariant-relation
  transition-relation return-relation-var
  return-relation-relation return-value-relation

```

```

defaxiom-id defaxiom-relation defthm-id
defthm-relation defun-id defun-fpl
defun-relation if-test if-then if-else
set-id set-index set-value get-id
get-index assoc-x assoc-y lookup-x
lookup-y in-range-x in-range-y not-in-x
not-in-y in-x in-y iff-x iff-y
implies-x implies-y and-x and-y or-x
or-y =-x =-y ne-x ne-y lt-x
lt-y le-x le-y gt-x gt-y ge-x
ge-y +-x +-y --x --y *-x
*-y /-x /-y mod-x mod-y rem-x
rem-y expt-x expt-y abs-x not-x
minus-x append-x append-y cons-x
car-x cdr-x fp-spec-id fp-spec-mode
fp-spec-type number-decl-id number-decl-mode
number-decl-body object-decl-id object-decl-mode
object-decl-type object-decl-body procedure-id
procedure-params procedure-return procedure-body
procedure-spec function-id function-params
function-return function-body function-spec
exception-id rename-pkg-new rename-pkg-old
rename-sub-new rename-sub-old rename-obj-new
rename-obj-type rename-obj-old assign-var
assign-value proc-call-id proc-call-actuals
return-value while-loop-test while-loop-statements
loop-statements for-loop-var for-loop-range
for-loop-statements reverse-for-loop-var
reverse-for-loop-range reverse-for-loop-statements
block-decls block-handler block-body
ifarm-test ifarm-statements casearm-test
casearm-statements case-stmt-test case-stmt-arms
constrained-st-relation constrained-st-stmt
comp-unit-unit comp-unit-clause context-with
context-use package-id package-outer
package-private package-inner
package-body)

```

```

DEFINE the theory ava-non-type-syntax-fns to be *ava-primitive-type-mkfun* @
*ava-primitive-type-argfun*.

```

A.6 Static Semantics Macros

SET CURRENT PACKAGE to be **ACL2**.

INCLUDING the book: *macros*.

INCLUDING the book: *subprefix-openers*.

Some type basics. See also *legality-overload.lisp*

```

CONSTANT:
*base-integer* = '(id integer 0)

```

Enable forcing.

```

MACRO:
add-variable-binding-to-vs (entry, vs)

```

```
=
\'(cons ,entry ,vs)
```

@subsection(More on Entries)

DEFINITION:

entry-mode (*x*)

```
=
if number-decl-p (entry-decl (x)) ∨ object-decl-p (entry-decl (x))
  then entry-decl (x)2
  else nil
fi
```

Does a package have a body? Yes, and its the 5th element.

DEFINITION:

entry-body (*x*)

```
=
if procedure-p (entry-decl (x))
  ∨ function-p (entry-decl (x))
  ∨ package-p (entry-decl (x)) then entry-decl (x)5
  else nil
fi
```

DEFINITION:

entry-p (*x*)

```
=
let name be entry-name (x),
  decl be entry-decl (x),
  value be entry-value (x)
  in
  id-p (name)
  ∧ (decl-p (decl) ∨ subtype-p (decl))
  ∧ (null (value) ∨ literal-p (value))
```

inner-decl = object-decl | number-decl | assert | invariant decl = inner-decl | subprogram | type-decl |
 subtype-decl | rename | defun | defthm | defaxiom

DEFINITION:

decl-id (*d*)

```
=
if object-decl-p (d)
  ∨ number-decl-p (d)
  ∨ type-decl-p (d)
  ∨ subtype-decl-p (d)
  ∨ subprogram-p (d)
  ∨ rename-pkg-p (d)
  ∨ rename-obj-p (d) then d1
  elseif rename-sub-p (d) then rename-sub-new (d)1
  else nil
fi
```

DEFINITION:

decl-kind (*d*)

```
=
if object-decl-p (d) then car (object-decl-mode (d))
  elseif number-decl-p (d) then 'constant
  elseif type-decl-p (d) then 'type
  elseif subtype-decl-p (d) then 'subtype
```

```

elseif procedure-p(d) then 'procedure
elseif function-p(d) then 'function
elseif rename-pkg-p(d) then 'rename
elseif rename-obj-p(d) then 'rename
elseif rename-sub-p(d) then 'rename
else nil
fi

```

MODIFY the current theory:

Disable 'nth'.

During the static semantics check we build an elaboration stack, which is the same thing as a variable stack except the value entry is ignored. The predicate BASIC-ENTRY-P is true of elaboration and value stacks, it does not say anything about what is the value part of this entry.

DEFINITION:

decl-type(*decl*)

=

case on opr(*decl*):

case = number-decl **then** *base-integer*

case = object-decl **then** object-decl-type(*decl*)

case = function **then** function-return(*decl*)

otherwise nil

endcase

DEFINITION:

entry-type(*x*) = decl-type(entry-body(*x*))

DEFINITION:

type-indication-p(*x*)

=

predefined-type-p(*x*) \vee id-p(*x*) \vee type-mark-p(*x*)

DEFINITION:

basic-entry-p(*entry*)

=

 entry-p(*entry*)

\wedge subtype-p(entry-decl(*entry*))

\wedge literal-p(entry-value(*entry*))

@subsection(Ava literals, values and types)

Moved definition of ADD-VARIABLE-BINDING-TO-VS from ava-dynamic. A. Flatau 4-May-1994

We have decided @i[not] to require all our values carry type information. Why overspecify? In the case of integers, we know what we want our operations to do, so why burden ourselves with useless type information.

For us, the "value" part of an object may be either an actual "raw" value or an expression. Or an apple, for that matter.

A elaboration-stack is a list of basic-entry-p's.

DEFINITION:

elaboration-stack-p(*alist*)

=

if consp(*alist*)

```

then basic-entry-p (car (alist))  $\wedge$  elaboration-stack-p (cdr (alist))
else alist = nil
fi

```

Later we may make requirements on the environment that say, for example, that every variable is assigned a value. For now we will make only trivial "type-theoretic" requirements.

MACRO:

```

lookup (x, env)
=
` (lookup2 ,x (es ,env) )

```

DEFINITION:

```

lookup3 (x, ea)
=
if  $\neg$  consp (ea) then nil
elseif x = entry-name (car (ea)) then car (ea)
else lookup3 (x, cdr (ea))
fi

```

DEFINITION:

```

lookup2 (x, es)
=
if  $\neg$  consp (es) then nil
elseif lookup3 (x, car (es)) then nil
else lookup2 (x, cdr (es))
fi

```

DEFINITION:

```

variable-lookup (x, env)
=
let entry be lookup (x, env)
in
if subtype-p (entry-decl (entry)) then entry
else nil
fi

```

DEFINITION:

```

proc-lookup (x, env)
=
let entry be lookup (x, env)
in
if procedure-p (entry-decl (entry)) then entry-decl (entry)
else nil
fi

```

DEFINITION:

```

proc-definedp (x, env)
=
let entry be lookup (x, env)
in
procedure-p (entry-decl (entry))

```

DEFINITION:

```

func-lookup (x, env)
=
let entry be lookup (x, env)
in
if function-p (entry-decl (entry)) then entry-decl (entry)
else nil

```


fi

DEFINITION:

func-definedp (x , env)

=

let $entry$ **be** lookup (x , env)

in

function-p (entry-decl ($entry$))

THEOREM: basic-entry-p-fact1

basic-entry-p ($entry$) \rightarrow entry-p ($entry$)

THEOREM: basic-entry-p-fact2

basic-entry-p ($entry$) \rightarrow id-p (entry-name ($entry$))

THEOREM: basic-entry-p-fact3

basic-entry-p ($entry$) \rightarrow subtype-p (entry-decl ($entry$))

THEOREM: basic-entry-p-fact4

basic-entry-p ($entry$) \rightarrow literal-p (entry-value ($entry$))

(defthm basic-entry-p-facts (and (implies (basic-entry-p entry) (entry-p entry)) (implies (basic-entry-p entry) (id-p (entry-name entry))) (implies (basic-entry-p entry) (member (entry-kind entry) '(constant variable))) (implies (basic-entry-p entry) (type-p (entry-body entry)))) :Hints (("Goal" :in-theory (disable expr-p type-p id-p))))

MODIFY the current theory:

Disable 'basic-entry-p'.

A.7 Predefined Packages

SET CURRENT PACKAGE to be **ACL2**.

CONSTANT:

standard =

```
\(package
  (id standard 0)
  (decls
    (type-decl (id boolean 0) (enumeration (false) (true)))
    (subtype-decl (id integer 0)
      (type-mark (id base-integer 0)
        (range (id ava_min_int 0) (id ava_max_int 0))))
    (subtype-decl (id natural 0)
      (type-mark (id integer 0) (range 0 (id ava_max_int 0))))
    (subtype-decl (id positive 0)
      (type-mark (id integer 0) (range 1 (id ava_max_int 0))))
    (function (id abs 0)
      (fpl (fp-spec (id left 0) (constant) (id integer 0))
        (id integer 0)
        nil nil))
    (function (id rem 0)
      (fpl (fp-spec (id left 0) (constant) (id integer 0))
        (fp-spec (id right 0) (constant) (id integer 0))
        (id integer 0)
        nil nil))
    (function (id mod 0)
      (fpl (fp-spec (id left 0) (constant) (id integer 0))
        (fp-spec (id right 0) (constant) (id integer 0))
        (id integer 0)
        nil nil))
    (type-decl (id character 0)
```



```

        (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
        nil nil nil)

(function (id mode 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
  (id file_mode 0)
  nil nil)
(function (id is_open 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
  (id boolean 0)
  nil nil)
(function (id end_of_file 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0)))
  (id boolean 0)
  nil nil)

(object-decl (id standard_output 0) (constant) (id file_type 0) nil)
(object-decl (id standard_input 0) (constant) (id file_type 0) nil)

(object-decl (id eol 0) (constant) (character) nil)

(procedure (id get 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
    (fp-spec (id item 0) (variable) (id character 0)))
  nil nil nil)
(procedure (id put 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
    (fp-spec (id item 0) (constant) (id character 0)))
  nil nil nil)

(procedure (id get_line 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
    (fp-spec (id item 0) (variable) (id string 0)))
  nil nil nil)

(procedure (id put_line 0)
  (fpl (fp-spec (id file 0) (constant) (id file_mode 0))
    (fp-spec (id item 0) (constant) (id string 0)))
  nil nil nil)))

(package (id io_exceptions 0)
  (decls (exception (id status_error 0))
    (exception (id mode_error 0))
    (exception (id name_error 0))
    (exception (id use_error 0))
    (exception (id device_error 0))
    (exception (id end_error 0))
    (exception (id data_error 0))
    (exception (id layout_error 0))))

(package (id system 0)
  (decls (type-decl (id name 0) nil)
    (object-decl (id ava_system_name 0) (constant) (id name 0))
    (object-decl (id ava_min_int 0) (constant) (id integer 0))
    (object-decl (id ava_max_int 0) (constant) (id integer 0))))))

```

CONSTANT:

annex-a = [**ada**, **standard**]

References

- [Smith 95] M. K. Smith.
AVA 95 Reference Manual.
Technical Report 114, Computational Logic, Inc., September, 1995.
Derived from ISO/IEC 8652:1995(E).

Index

- *-p 72, 74
- *-p, defined 72
- *-x 72, 87
- *-x, defined 72
- *-y 72, 87
- *-y, defined 72

- +p 72, 74
- +p, defined 71
- +x 72, 87
- +x, defined 72
- +y 72, 87
- +y, defined 72

- p 72, 74
- p, defined 72
- x 72, 87
- x, defined 72
- y 72, 87
- y, defined 72

- /-p 72, 74
- /-p, defined 72
- /-x 72, 87
- /-x, defined 72
- /-y 72, 87
- /-y, defined 72

- 1+ 4
- 1- 4

- := 5

- =p 70, 74
- =p, defined 70
- =x 71, 87
- =x, defined 70
- =y 71, 87
- =y, defined 71

- Abs 4, 23, 24
- Abs-p 73, 74
- Abs-p, defined 73
- Abs-x 73, 87
- Abs-x, defined 73
- Acl2, package used 7, 14, 43, 46, 55, 58, 92, 96
- Acl2-boolean 59, 61
- Acl2-boolean, defined 59
- Acl2-count 30, 35, 36, 49, 56, 57, 89, 90
- Acl2-count-cadar-3 49
- Acl2-count-extract-agg-values 36
- Acl2-count-get-t-2 49
- Actions 8, 30, 36
- Actions, defined 30
- Ada-booleanp 46
- Ada-booleanp, defined 46
- Ada-char-p 22, 23, 59, 61
- Ada-char-p, defined 59
- Ada-st-p 82
- Ada-st-p, defined 82
- Add-variable-binding-to-vs, defined 92
- Adjust-array-literal 26
- Adjust-array-literal, defined 26
- Aggregate-args-p 84, 88
- Aggregate-args-p, defined 84
- Aggregate-arm-p 64, 84
- Aggregate-arm-p, defined 64
- Aggregate-choice-choice 33, 64, 88
- Aggregate-choice-choice, defined 64
- Aggregate-choice-p 10, 33, 34, 64
- Aggregate-choice-p, defined 64
- Aggregate-choice-value 33, 64, 88
- Aggregate-choice-value, defined 64
- Aggregate-p 10, 34, 64, 66
- Aggregate-p, defined 64
- Aggregate-pos-p 10, 33, 34, 64
- Aggregate-pos-p, defined 64
- Aggregate-pos-value 10, 33, 64, 88
- Aggregate-pos-value, defined 64
- All-agg-choice 34
- All-agg-choice, defined 34
- All-agg-pos 34
- All-agg-pos, defined 34
- All-id 59
- All-id, defined 59
- All-integer 59
- All-integer, defined 59
- All-labels-id 29, 59, 60
- All-labels-id, defined 59
- All-labels-integer 29, 59, 60
- All-labels-integer, defined 59
- And-p 70, 74
- And-p, defined 70
- And-x 70, 87
- And-x, defined 70
- And-y 70, 87
- And-y, defined 70
- Apl-args-p 84, 88
- Apl-args-p, defined 84
- Apl-p 66, 86, 88
- Apl-p, defined 66
- Append-ase, defined 41
- Append-p 73, 74
- Append-p, defined 73
- Append-vse, defined 41
- Append-x 73, 87
- Append-x, defined 73
- Append-y 73, 87
- Append-y, defined 73
- Apply-1-args 65, 88
- Apply-1-args, defined 65
- Apply-1-p 65, 66
- Apply-1-p, defined 65
- Apply-1-root 65, 88
- Apply-1-root, defined 65
- Arg* 8, 9, 10, 11, 16, 31, 34, 35, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 85, 86, 87, 88, 89, 90
- Arg1 12, 13, 14, 16, 58
- Arg1, defined 58
- Arg2 13, 14, 58
- Arg2, defined 58
- Arg3 13, 58
- Arg3, defined 58
- Arg4 58
- Arg4, defined 58
- Arg5 58
- Arg5, defined 58
- Arg6 58
- Arg6, defined 58
- Arg7 59
- Arg7, defined 59
- Arg8 59
- Arg8, defined 59
- Arg9 59
- Arg9, defined 59
- Argn 16
- Array-elem-exists 29
- Array-literal-from 22, 26, 27, 29, 32
- Array-literal-from, defined 22
- Array-literal-p 22, 27, 28, 29, 31, 60, 61, 89
- Array-literal-p, defined 60
- Array-literal-to 22, 27, 29, 32
- Array-literal-to, defined 22
- Array-type-elements 31, 32, 63, 89
- Array-type-elements, defined 63
- Array-type-index 27, 28, 31, 32, 35, 63, 89
- Array-type-index, defined 63
- Array-type-p 27, 28, 32, 35, 63
- Array-type-p, defined 62
- As 4, 7, 15, 42
- As, defined 38
- As-p 15
- As-p* 15
- As-p*, defined 15
- As-p, defined 15
- Assert 5
- Assert-p 67, 68, 77, 82
- Assert-p, defined 67
- Assert-relation 67, 88
- Assert-relation, defined 67
- Assert1 5
- Assert1, defined 43
- Asserts 5
- Asserts, defined 43
- Assign-actuals 9, 35

- Assign-actuals, defined 35
- Assign-p 78, 82
- Assign-p, defined 78
- Assign-to-env 8, 31
- Assign-to-env, defined 31
- Assign-value 8, 78, 86
- Assign-value, defined 78
- Assign-var 8, 78, 86
- Assign-var, defined 78
- Assoc 42
- Assoc-equal 30
- Assoc-p 69, 74
- Assoc-p, defined 69
- Assoc-x 69, 87
- Assoc-x, defined 69
- Assoc-y 69, 87
- Assoc-y, defined 69
- Attribute-attr 62, 89
- Attribute-attr, defined 62
- Attribute-p 62
- Attribute-p, defined 62
- Attribute-root 62, 89
- Attribute-root, defined 62
- Ava-<-val 23, 25
- Ava-<-val, defined 23
- Ava-abs-exc 23, 25
- Ava-abs-exc, defined 23
- Ava-abs-val 24, 25
- Ava-abs-val, defined 24
- Ava-and 24, 25
- Ava-and, defined 24
- Ava-divide-exc 21, 25
- Ava-divide-exc, defined 20
- Ava-divide-val 21, 24
- Ava-divide-val, defined 21
- Ava-dynamic, included 7
- Ava-equal-val 22, 25
- Ava-equal-val, defined 22
- Ava-in-range-val 23, 25
- Ava-in-range-val, defined 23
- Ava-max-int 16, 34
- Ava-max-int, defined 16
- Ava-min-int 16, 34
- Ava-min-int, defined 16
- Ava-mod-exc 21, 25
- Ava-mod-exc, defined 21
- Ava-mod-val 21, 25
- Ava-mod-val, defined 21
- Ava-multiply-exc 20, 25
- Ava-multiply-exc, defined 20
- Ava-multiply-val 20, 24
- Ava-multiply-val, defined 20
- Ava-non-type-syntax-fns, theory 92
- Ava-not 24, 25
- Ava-not, defined 24
- Ava-op-defthm-forms 18, 19
- Ava-op-defthm-forms, defined 18
- Ava-op-fns, theory 24
- Ava-op-fns-start 20
- Ava-operators-alist 18, 19
- Ava-operators-alist, defined 18
- Ava-or 24, 25
- Ava-or, defined 24
- Ava-plus-exc 20, 25
- Ava-plus-exc, defined 20
- Ava-plus-val 20, 24
- Ava-plus-val, defined 20
- Ava-power-exc 20, 25
- Ava-power-exc, defined 20
- Ava-power-val 20, 25
- Ava-power-val, defined 20
- Ava-primitive-type-fns-2, theory 90
- Ava-rem-exc 21, 25
- Ava-rem-exc, defined 21
- Ava-rem-val 21, 25
- Ava-rem-val, defined 21
- Ava-unary-minus-exc 21, 25
- Ava-unary-minus-exc, defined 21
- Ava-unary-minus-val 21, 24
- Ava-unary-minus-val, defined 21
- Ava-xor 24, 25
- Ava-xor, defined 24
- Basic-entry-p 29, 94, 96
- Basic-entry-p, defined 94
- Basic-entry-p-fact1 96
- Basic-entry-p-fact2 96
- Basic-entry-p-fact3 96
- Basic-entry-p-fact4 96
- Between 23, 26, 29
- Between, defined 23
- Block-body 10, 80, 85
- Block-body, defined 80
- Block-decls 9, 80, 85
- Block-decls, defined 80
- Block-handler 10, 80, 85
- Block-handler, defined 80
- Block-p 80, 81, 86
- Block-p, defined 80
- Boolean-literal-p 61
- Boolean-literal-p, defined 61
- Booleantp 89
- Branchp 57
- Branchp, defined 57
- C 1
- Caar 26, 42, 44, 45, 59
- Car-p 74
- Car-p, defined 73
- Car-x 74, 87
- Car-x, defined 74
- Case-match 89
- Case-stmt-arms 81, 85
- Case-stmt-arms, defined 81
- Case-stmt-p 81
- Case-stmt-p, defined 81
- Case-stmt-test 81, 85
- Case-stmt-test, defined 81
- Casearm-p 81, 83
- Casearm-p, defined 81
- Casearm-statements 81, 85
- Casearm-statements, defined 81
- Casearm-test 81, 85
- Casearm-test, defined 81
- Casearms-args-p 83, 85
- Casearms-args-p, defined 83
- Casearms-p 81, 85
- Casearms-p, defined 81
- Cdar 26, 49
- Cdddr 44
- Cdr-p 74
- Cdr-p, defined 74
- Cdr-strip-cdrs-count-le 57
- Cdr-x 74, 87
- Cdr-x, defined 74
- Char-code 22, 23
- Char-code-nonnegative-integerp-for-ada-char-p 22
- Char< 55
- Characterp 46, 55, 59, 89, 90
- Check-annotations 43, 44
- Check-annotations, defined 43
- Check-assert 5, 9, 12
- Check-assert, defined 43
- Check-asserts 5, 8, 9, 10, 12, 13
- Check-asserts, defined 43
- Choice-aggregate-p 11, 34
- Choice-aggregate-p, defined 34
- Choice-p 75, 84
- Choice-p, defined 74
- Choices-args-p 84, 87
- Choices-args-p, defined 84
- Choices-p 75, 88
- Choices-p, defined 75
- Clock 4
- Coerce-to-subtype-exc 11, 12, 27
- Coerce-to-subtype-exc, defined 27
- Coerce-to-subtype-from-lit-exc 27, 31
- Coerce-to-subtype-from-lit-exc, defined 27
- Coerce-to-subtype-from-lit-val 27, 31
- Coerce-to-subtype-from-lit-val, defined 27
- Coerce-to-subtype-val 11, 12, 27
- Coerce-to-subtype-val, defined 27
- Comp-unit-clause 82, 85
- Comp-unit-clause, defined 82
- Comp-unit-p 14, 82, 83
- Comp-unit-p, defined 82
- Comp-unit-unit 82, 85
- Comp-unit-unit, defined 82
- Compilation-args-p 83, 85
- Compilation-args-p, defined 83
- Compilation-p 82
- Compilation-p, defined 82
- Compute-type-range-list 34, 35
- Compute-type-range-list, defined 34
- Congruent 3
- Conjunction 33
- Conjunction, defined 33
- Cons-p 73, 74
- Cons-p, defined 73
- Cons-x 73, 87
- Cons-x, defined 73
- Consp-labelp* 46
- Constant-p 75
- Constant-p, defined 75

- Constrain-array-exc 26, 27
- Constrain-array-exc, defined 26
- Constrain-array-val 26, 27, 28
- Constrain-array-val, defined 26
- Constrain-range-if-necessary 12, 32
- Constrain-range-if-necessary, defined 32
- Constrained-st-p 81, 82
- Constrained-st-p, defined 81
- Constrained-st-relation 8, 81, 85
- Constrained-st-relation, defined 81
- Constrained-st-stmt 8, 81, 85
- Constrained-st-stmt, defined 81
- Constraint-error 20, 21, 23, 27, 28, 29, 31
- Constraint-error, defined 17
- Constraint-p 62, 89
- Constraint-p, defined 62
- Context-p 82, 85
- Context-p, defined 82
- Context-use 82, 85
- Context-use, defined 82
- Context-with 82, 85
- Context-with, defined 82
- Count-actions 36
- Count-actions-cadr 36
- Count-actions-cadr-2 36
- Create-inclusive-list 34
- Create-inclusive-list, defined 34

- Decl-id 93
- Decl-id, defined 93
- Decl-kind 93
- Decl-kind, defined 93
- Decl-p 33, 78, 83, 93
- Decl-p, defined 78
- Decl-type 94
- Decl-type, defined 94
- Decls-args-p 83, 86
- Decls-args-p, defined 83
- Decls-p 78, 85
- Decls-p, defined 78
- Defaxiom-id 68, 88
- Defaxiom-id, defined 68
- Defaxiom-p 67, 78
- Defaxiom-p, defined 67
- Defaxiom-relation 68, 88
- Defaxiom-relation, defined 68
- Defining-name-p 65
- Defining-name-p, defined 65
- Defthm-id 68, 88
- Defthm-id, defined 68
- Defthm-p 68, 78
- Defthm-p, defined 68
- Defthm-relation 68, 88
- Defthm-relation, defined 68
- Defun-fpl 68, 88
- Defun-fpl, defined 68
- Defun-id 68, 88
- Defun-id, defined 68
- Defun-p 68, 78
- Defun-p, defined 68
- Defun-relation 68, 88
- Defun-relation, defined 68
- Designator-args-p 84, 88
- Designator-args-p, defined 84
- Designator-p 65
- Designator-p, defined 65
- Distinct 51, 54, 55
- Distinct, defined 51
- Do-mv-macro 44, 45
- Do-mv-macro, defined 44
- Domain 29, 47, 60
- Domain, defined 47
- Domain-restrict 48
- Domain-restrict, defined 48
- Dot-qual-1-component 65, 88
- Dot-qual-1-component, defined 65
- Dot-qual-1-p 65, 66
- Dot-qual-1-p, defined 65
- Dot-qual-1-root 65, 88
- Dot-qual-1-root, defined 65

- E0-ordinalp 35
- Elaboration-stack-p 94
- Elaboration-stack-p, defined 94
- Enable 34, 36, 48, 50, 51, 57
- Entry-body 93, 94
- Entry-body, defined 93
- Entry-decl 29, 31, 93, 94, 95, 96
- Entry-decl, defined 38
- Entry-mode 93
- Entry-mode, defined 93
- Entry-name 29, 93, 95, 96
- Entry-name, defined 38
- Entry-p 15, 93, 94, 96
- Entry-p, defined 93
- Entry-type 94
- Entry-type, defined 94
- Entry-value 10, 31, 93, 94, 96
- Entry-value, defined 38
- Enumeration-args-p 85, 89
- Enumeration-args-p, defined 84
- Enumeration-literal-p 61, 85
- Enumeration-literal-p, defined 61
- Enumeration-p 61, 63
- Enumeration-p, defined 61
- Env-p 7, 15
- Env-p, defined 15
- Error-form 60, 89
- Error-form, defined 60
- Error-message 60, 89
- Error-message, defined 60
- Error-p 60
- Error-p, defined 60
- Es 4, 7, 8, 9, 10, 12, 13, 15, 31, 42
- Es, defined 38
- Es-p 14, 15
- Es-p* 15
- Es-p*, defined 15
- Es-p, defined 14
- Eval 17
- Eval-opr-exc 11, 25
- Eval-opr-exc, defined 25
- Eval-opr-val 11, 24
- Eval-opr-val, defined 24
- Exception 8, 9, 10, 11, 12
- Exception-id 77, 86
- Exception-id, defined 77
- Exception-p 77
- Exception-p, defined 77
- Exists 2
- Exists-if-put-t 52
- Exists-imp-not-leaf-p-get-treep-get 50
- Exists-l 51, 54
- Exists-l, defined 51
- Exists-t 29, 48, 49, 50, 51, 52, 53, 54
- Exists-t, defined 48
- Exists-t-exists-l 54
- Exists-t-opener-1 53
- Exists-t-opener-2 53
- Exists-t-opener-3 54
- Exists-t-put-t 54
- Exit-p 79, 82
- Exit-p, defined 79
- Expr-p 66, 75, 84, 85, 86, 87, 88, 89
- Expr-p, defined 66
- Expt 20
- Expt-p 73, 74
- Expt-p, defined 73
- Expt-x 73, 87
- Expt-x, defined 73
- Expt-y 73, 87
- Expt-y, defined 73
- Extend-constraint-error 18, 31
- Extend-constraint-error, defined 18
- Extend-env-with-package 13, 35
- Extend-env-with-package, defined 35
- Extend-env-with-procedure 13, 35
- Extend-env-with-procedure, defined 35
- Extract-agg-labels 11, 34
- Extract-agg-labels, defined 33
- Extract-agg-labels1 33, 34
- Extract-agg-labels1, defined 33
- Extract-agg-values 11, 33, 36
- Extract-agg-values, defined 33
- Extract-ids 9, 35
- Extract-ids, defined 35

- Fail-get-l 52
- False 9, 21, 22, 24
- False-p 24, 25, 61
- False-p, defined 61
- Field-spec-decl 31, 62, 89
- Field-spec-decl, defined 62
- Field-spec-id 34, 62, 89
- Field-spec-id, defined 62
- Field-spec-ids 34, 35
- Field-spec-ids, defined 34
- Field-spec-p 62, 84
- Field-spec-p, defined 62
- Find-package-decl 13
- Find-package-decl, defined 13
- Fix-bool 19, 22, 23
- Fix-bool, defined 19
- Fix-int 19

- Fix-int, defined 19
- Fn 1
- For-loop-p 79, 80
- For-loop-p, defined 79
- For-loop-range 80, 86
- For-loop-range, defined 80
- For-loop-statements 80, 86
- For-loop-statements, defined 80
- For-loop-var 80, 86
- For-loop-var, defined 80
- Forall 1
- Fp-spec-id 12, 35, 75, 87
- Fp-spec-id, defined 75
- Fp-spec-mode 35, 75, 87
- Fp-spec-mode, defined 75
- Fp-spec-p 35, 75, 84
- Fp-spec-p, defined 75
- Fp-spec-type 12, 75, 87
- Fp-spec-type, defined 75
- Fpl-args-p 84, 87
- Fpl-args-p, defined 84
- Fpl-p 75, 86
- Fpl-p, defined 75
- Func-definedp 96
- Func-definedp, defined 96
- Func-lookup 11, 95
- Func-lookup, defined 95
- Function-body 12, 76, 86
- Function-body, defined 76
- Function-call-actuals 11, 66, 88
- Function-call-actuals, defined 66
- Function-call-id 11, 66, 88
- Function-call-id, defined 66
- Function-call-p 11, 66
- Function-call-p, defined 66
- Function-id 76, 86
- Function-id, defined 76
- Function-p 76, 77, 93, 94, 95, 96
- Function-p, defined 76
- Function-params 11, 76, 86
- Function-params, defined 76
- Function-return 76, 86, 94
- Function-return, defined 76
- Function-spec 11, 77, 86
- Function-spec, defined 76
- Function-theory 24

- Ge, defined 23
- Ge-le-eq 23
- Ge-p 71, 74
- Ge-p, defined 71
- Ge-x 71, 87
- Ge-x, defined 71
- Ge-y 71, 87
- Ge-y, defined 71
- Get-array-elem-exc 10, 29, 31
- Get-array-elem-exc, defined 29
- Get-array-elem-exc-null-array 29
- Get-id 69, 88
- Get-id, defined 69
- Get-index 69, 88
- Get-index, defined 69
- Get-l 50, 51, 52, 53, 54, 55
- Get-l, defined 50
- Get-l-opener-1 53
- Get-l-opener-3 53
- Get-l-opener-4 53
- Get-l-put-l-nil 55
- Get-l-put-l-opener-unequal-cars 54
- Get-l-put-t-opener 54
- Get-p 69, 74
- Get-p, defined 69
- Get-put-i-j 55
- Get-put-l 52
- Get-t 10, 31, 48, 49, 50, 51, 52, 53, 54
- Get-t, defined 48
- Get-t-nil-2 49
- Get-t-opener-1 49
- Get-t-opener-2 49
- Get-t-opener-3 49
- Get-t-put-t-identity 50
- Get-t-put-t-ne 52
- Get-tree, included 14, 55, 58
- Gt, defined 23
- Gt-p 71, 74
- Gt-p, defined 71
- Gt-x 71, 87
- Gt-x, defined 71
- Gt-y 71, 87
- Gt-y, defined 71

- Handler-error-p 10, 17
- Handler-error-p, defined 17
- Hard-error-env 7, 8, 10, 11, 12, 13, 14
- Hard-error-env, defined 43
- Hard-error2 10, 44, 45
- Hard-error2, defined 43

- Id-p 10, 33, 34, 60, 61, 62, 63, 65, 66, 83, 84, 85, 86, 87, 88, 89, 93, 94, 96
- Id-p, defined 60
- Id-root 11, 33, 60, 89
- Id-root, defined 60
- Id-uid 61, 89
- Id-uid, defined 60
- Ids-args-p 83, 85
- Ids-args-p, defined 83
- Ids-p 82, 85
- Ids-p, defined 82
- If-else 68, 88
- If-else, defined 68
- If-p 68, 74
- If-p, defined 68
- If-stmt-args-p 83, 85
- If-stmt-args-p, defined 83
- If-stmt-p 81
- If-stmt-p, defined 80
- If-test 68, 88
- If-test, defined 68
- If-then 68, 88
- If-then, defined 68
- Ifarm-p 80, 83
- Ifarm-p, defined 80
- Ifarm-statements 9, 80, 85
- Ifarm-statements, defined 80
- Ifarm-test 9, 80, 85
- Ifarm-test, defined 80
- Iff-p 70, 74
- Iff-p, defined 70
- Iff-x 70, 87
- Iff-x, defined 70
- Iff-y 70, 87
- Iff-y, defined 70
- Ifix 23, 24, 34
- Ilet, defined 45
- Ilet, included 14
- Implies-p 70, 74
- Implies-p, defined 70
- Implies-x 70, 87
- Implies-x, defined 70
- Implies-y 70, 87
- Implies-y, defined 70
- In-p 70, 74
- In-p, defined 69
- In-range-p 69, 74
- In-range-p, defined 69
- In-range-x 69, 87
- In-range-x, defined 69
- In-range-y 69, 87
- In-range-y, defined 69
- In-x 70, 87
- In-x, defined 70
- In-y 70, 87
- In-y, defined 70
- Indexed-component-index 10, 30, 65, 88
- Indexed-component-index, defined 65
- Indexed-component-p 10, 30, 64, 66
- Indexed-component-p, defined 64
- Indexed-component-root 10, 30, 65, 88
- Indexed-component-root, defined 65
- Indexed-component-root-decrease 30
- Induct3 54
- Induct3, defined 54
- Inner-decl-p 77, 78, 84
- Inner-decl-p, defined 77
- Inner-decls-args-p 83, 84, 86
- Inner-decls-args-p, defined 83
- Inner-decls-p 77, 85
- Inner-decls-p, defined 77
- Insert 56, 57
- Insert, defined 56
- Insert-cons-count= 56
- Insert-sort, included 14
- Instate 17
- Instate-expr 66, 88
- Instate-expr, defined 66
- Instate-p 66, 74
- Instate-p, defined 66
- Int-not-in-machine-range 20, 21, 23
- Int-not-in-machine-range, defined 42
- Int-not-in-range 20

- Int-not-in-range, defined 20
- Integerp-ava-max-int 16
- Integerp-ava-min-int 16
- Interpret-decl 8
- Interpret-decl, defined 32
- Interpret-decls 8, 9, 12
- Interpret-decls, defined 32
- Interpret-eval 7, 14
- Interpret-eval, defined 7
- Interpret-exp 8, 9, 10, 11, 12, 13, 19
- Interpret-exp, defined 32
- Interpret-exps 8, 9, 11, 12
- Interpret-exps, defined 32
- Interpret-measure 13, 35
- Interpret-measure, defined 35
- Interpret-measure-facts 35
- Interpret-program 14
- Interpret-program, defined 14
- Interpret-stmt 7, 8, 9, 12
- Interpret-stmt, defined 32
- Interpret-stmts 7, 8, 9, 10
- Interpret-stmts, defined 32
- Intersection-theories 3
- Invariant-p 67, 77
- Invariant-p, defined 67
- Invariant-relation 67, 88
- Invariant-relation, defined 67

- Kernel-extension, included 58

- Labelp 46, 47, 48, 49, 50, 51, 52, 53, 54, 57
- Labelp* 46, 48, 50, 51, 52, 53, 54, 55, 59
- Labelp*, defined 46
- Labelp*-bkwd 46
- Labelp*-cdr 46
- Labelp*-fwd 46
- Labelp*-list-car-if-labelp* 46
- Labelp*-singleton 46
- Labelp*-true-listp 46
- Labelp, defined 46
- Labelp-car 46
- Le, defined 23
- Le-1 23
- Le-2 23
- Le-3 23
- Le-p 71, 74
- Le-p, defined 71
- Le-x 71, 87
- Le-x, defined 71
- Le-y 71, 87
- Le-y, defined 71
- Leaf-not-tree 21
- Leaf-p 46, 47, 50, 51, 52, 53, 54, 57
- Leaf-p, defined 46
- Leaf-p-not-nil 52
- Leaf-p-not-treep 47
- Leaf-p-valuep 47
- Leafp 21, 22
- Len-0 32
- Lessp-or-equal 55, 56
- Lessp-or-equal, defined 55
- Leval 44
- Leval-macros, included 43
- Lexpr-p 15, 74, 87, 88, 90
- Lexpr-p, defined 74
- Library-unit-p 83, 85
- Library-unit-p, defined 83
- List-args-p 85, 89
- List-args-p, defined 85
- List-p 61, 74
- List-p, defined 61
- Listp 17, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 89, 90
- Literal-p 10, 15, 61, 66, 93, 94, 96
- Literal-p, defined 61
- Lliteral-p 61, 74
- Lliteral-p, defined 61
- Logical-error, defined 17
- Lookup 95, 96
- Lookup, defined 95
- Lookup-p 69, 74
- Lookup-p, defined 69
- Lookup-x 69, 87
- Lookup-x, defined 69
- Lookup-y 69, 87
- Lookup-y, defined 69
- Lookup2 95
- Lookup2, defined 95
- Lookup3 95
- Lookup3, defined 95
- Loop-p 79, 80
- Loop-p, defined 79
- Loop-statements 79, 86
- Loop-statements, defined 79
- Loop-stmt-p 80, 81
- Loop-stmt-p, defined 80
- Lowerp 36
- Lowerp, defined 36
- Lt, defined 22
- Lt-p 71, 74
- Lt-p, defined 71
- Lt-x 71, 87
- Lt-x, defined 71
- Lt-y 71, 87
- Lt-y, defined 71

- Macros, included 14, 92
- Make-constrained-entry 12, 32
- Make-constrained-entry, defined 32
- Make-entry 13, 29, 32, 35, 42
- Make-entry, defined 38
- Max 22, 34
- Maximum 22
- Maximum, defined 22
- Member-insert=1 57
- Member-sortl=2 57
- Merge-package 13, 14
- Merge-package, defined 13
- Min 22
- Min-int-close-to-max-int 16
- Minimum 22
- Minimum, defined 22
- Minus 4
- Minus-p 73, 74
- Minus-p, defined 73
- Minus-x 73, 87
- Minus-x, defined 73
- Mk-* 72
- Mk-*, defined 72
- Mk-+ 72
- Mk-+, defined 72
- Mk-- 72
- Mk--, defined 72
- Mk-/ 72
- Mk-/, defined 72
- Mk-= 70
- Mk=, defined 70
- Mk-abs 73
- Mk-abs, defined 73
- Mk-aggregate 64
- Mk-aggregate, defined 64
- Mk-aggregate-choice 64
- Mk-aggregate-choice, defined 64
- Mk-aggregate-pos 64
- Mk-aggregate-pos, defined 64
- Mk-and 70
- Mk-and, defined 70
- Mk-apl 66
- Mk-apl, defined 66
- Mk-append 73
- Mk-append, defined 73
- Mk-apply-1 65
- Mk-apply-1, defined 65
- Mk-array-type 32, 63
- Mk-array-type, defined 63
- Mk-assert 67
- Mk-assert, defined 67
- Mk-assign 35, 78
- Mk-assign, defined 78
- Mk-assoc 69
- Mk-assoc, defined 69
- Mk-attribute 62
- Mk-attribute, defined 62
- Mk-block 80
- Mk-block, defined 80
- Mk-car 74
- Mk-car, defined 74
- Mk-case-stmt 81
- Mk-case-stmt, defined 81
- Mk-casearm 81
- Mk-casearm, defined 81
- Mk-casearms 81
- Mk-casearms, defined 81
- Mk-cdr 74
- Mk-cdr, defined 74
- Mk-choices 75
- Mk-choices, defined 75
- Mk-comp-unit 14, 82
- Mk-comp-unit, defined 82
- Mk-compilation 14, 82
- Mk-compilation, defined 82
- Mk-cons 73
- Mk-cons, defined 73
- Mk-constant 75
- Mk-constant, defined 75

Mk-constrained-st 81
Mk-constrained-st, defined 81
Mk-context 82
Mk-context, defined 82
Mk-decls 78
Mk-decls, defined 78
Mk-defaxiom 68
Mk-defaxiom, defined 68
Mk-defthm 68
Mk-defthm, defined 68
Mk-defun 68
Mk-defun, defined 68
Mk-designator 65
Mk-designator, defined 65
Mk-dot-qual-1 65
Mk-dot-qual-1, defined 65
Mk-enumeration 61
Mk-enumeration, defined 61
Mk-error 60
Mk-error, defined 60
Mk-exception 77
Mk-exception, defined 77
Mk-exit 79
Mk-exit, defined 79
Mk-expt 73
Mk-expt, defined 73
Mk-false 61
Mk-false, defined 61
Mk-field-spec 62
Mk-field-spec, defined 62
Mk-for-loop 79
Mk-for-loop, defined 79
Mk-fp-spec 75
Mk-fp-spec, defined 75
Mk-fpl 75
Mk-fpl, defined 75
Mk-function 76
Mk-function, defined 76
Mk-function-call 66
Mk-function-call, defined 66
Mk-ge 71
Mk-ge, defined 71
Mk-get 69
Mk-get, defined 69
Mk-gt 71
Mk-gt, defined 71
Mk-id 60
Mk-id, defined 60
Mk-ids 82
Mk-ids, defined 82
Mk-if 68
Mk-if, defined 68
Mk-if-stmt 9, 81
Mk-if-stmt, defined 81
Mk-ifarm 80
Mk-ifarm, defined 80
Mk-iff 70
Mk-iff, defined 70
Mk-implies 70
Mk-implies, defined 70
Mk-in 70
Mk-in, defined 70
Mk-in-range 69
Mk-in-range, defined 69
Mk-indexed-component 64
Mk-indexed-component, defined 64
Mk-inner-decls 77
Mk-inner-decls, defined 77
Mk-instate 66
Mk-instate, defined 66
Mk-invariant 67
Mk-invariant, defined 67
Mk-le 71
Mk-le, defined 71
Mk-list 61
Mk-list, defined 61
Mk-lookup 69
Mk-lookup, defined 69
Mk-loop 79
Mk-loop, defined 79
Mk-lt 71
Mk-lt, defined 71
Mk-minus 73
Mk-minus, defined 73
Mk-mod 72
Mk-mod, defined 72
Mk-ne 71
Mk-ne, defined 71
Mk-not 73
Mk-not, defined 73
Mk-not-in 69
Mk-not-in, defined 69
Mk-null 78
Mk-null, defined 78
Mk-number-decl 75
Mk-number-decl, defined 75
Mk-object-decl 76
Mk-object-decl, defined 76
Mk-op-expr 66
Mk-op-expr, defined 66
Mk-or 70
Mk-or, defined 70
Mk-others 60
Mk-others, defined 60
Mk-outstate 66
Mk-outstate, defined 66
Mk-package 13, 82
Mk-package, defined 82
Mk-proc-call 79
Mk-proc-call, defined 79
Mk-procedure 76
Mk-procedure, defined 76
Mk-qualified 63
Mk-qualified, defined 63
Mk-raise 78
Mk-raise, defined 78
Mk-range 32, 62
Mk-range, defined 62
Mk-record-type 62
Mk-record-type, defined 62
Mk-rem 72
Mk-rem, defined 72
Mk-rename-obj 78
Mk-rename-obj, defined 78
Mk-rename-pkg 77
Mk-rename-pkg, defined 77
Mk-rename-sub 77
Mk-rename-sub, defined 77
Mk-return 79
Mk-return, defined 79
Mk-return-relation 67
Mk-return-relation, defined 67
Mk-return-value 67
Mk-return-value, defined 67
Mk-reverse-for-loop 80
Mk-reverse-for-loop, defined 80
Mk-selected-component 65
Mk-selected-component, defined 65
Mk-set 68
Mk-set, defined 68
Mk-sl 79
Mk-sl, defined 79
Mk-subtype-decl 63
Mk-subtype-decl, defined 63
Mk-transition 67
Mk-transition, defined 67
Mk-true 61
Mk-true, defined 61
Mk-type-convert 64
Mk-type-convert, defined 64
Mk-type-decl 63
Mk-type-decl, defined 63
Mk-type-mark 61
Mk-type-mark, defined 61
Mk-unconstrained 60
Mk-unconstrained, defined 60
Mk-use 77
Mk-use, defined 77
Mk-variable 75
Mk-variable, defined 75
Mk-while-loop 79
Mk-while-loop, defined 79
Mkenv 42
Mkenv, defined 38
Mod 21
Mod-p 72, 74
Mod-p, defined 72
Mod-x 72, 87
Mod-x, defined 72
Mod-y 72, 87
Mod-y, defined 72
Mv-let 14
Name-p 65, 66, 86, 88
Name-p, defined 65
Ne-p 71, 74
Ne-p, defined 71
Ne-x 71, 87
Ne-x, defined 71
Ne-y 71, 87
Ne-y, defined 71
Not-distinct 54
Not-get-l-leaf 55
Not-get-t-step 52
Not-in-p 69, 74
Not-in-p, defined 69
Not-in-x 69, 87
Not-in-x, defined 69
Not-in-y 69, 87
Not-in-y, defined 69

- Not-leaf-p-cons 52
- Not-leaf-p-imp-treep 47
- Not-leaf-p-put-t 52
- Not-p 73, 74
- Not-p, defined 73
- Not-treep-imp-leaf-p 47
- Not-x 73, 87
- Not-x, defined 73
- Nth 11, 24, 25, 31, 44, 45, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 93
- Nth-vse 10
- Nth-vse, defined 39
- Null-get-l-2 51
- Null-p 78, 81
- Null-p, defined 78
- Number-decl-body 13, 75, 87
- Number-decl-body, defined 75
- Number-decl-id 13, 75, 87
- Number-decl-id, defined 75
- Number-decl-mode 75, 87
- Number-decl-mode, defined 75
- Number-decl-p 75, 77, 93
- Number-decl-p, defined 75
- Numberp 4
- Object-decl-body 12, 76, 86
- Object-decl-body, defined 76
- Object-decl-id 12, 76, 86
- Object-decl-id, defined 76
- Object-decl-mode 76, 86, 93
- Object-decl-mode, defined 76
- Object-decl-p 75, 77, 93
- Object-decl-p, defined 75
- Object-decl-type 12, 76, 86, 94
- Object-decl-type, defined 76
- Op-expr-actuals 11, 66, 88
- Op-expr-actuals, defined 66
- Op-expr-id 11, 66, 88
- Op-expr-id, defined 66
- Op-expr-p 11, 66
- Op-expr-p, defined 66
- Opr 16, 85, 94
- Or-p 70, 74
- Or-p, defined 70
- Or-x 70, 87
- Or-x, defined 70
- Or-y 70, 87
- Or-y, defined 70
- Order-of-elaboration 37
- Others-p 33, 60, 75
- Others-p, defined 60
- Outstate 17
- Outstate-expr 67, 88
- Outstate-expr, defined 66
- Outstate-p 66, 74
- Outstate-p, defined 66
- P 1, 2
- Package-body 83, 85
- Package-body, defined 83
- Package-id 35, 82, 85
- Package-id, defined 82
- Package-inner 83, 85
- Package-inner, defined 83
- Package-outer 82, 85
- Package-outer, defined 82
- Package-p 82, 83, 93
- Package-p, defined 82
- Package-private 83, 85
- Package-private, defined 82
- Package-up 14
- Package-up, defined 14
- Pairlis\$ 11
- Pmode-p 75, 86, 87
- Pmode-p, defined 75
- Pop-as 8, 10
- Pop-as, defined 40
- Pop-ase, defined 41
- Pop-env 9, 10, 11
- Pop-env, defined 42
- Pop-es 10
- Pop-es, defined 39
- Pop-ese, defined 41
- Pop-value 9, 11, 12, 13
- Pop-value, defined 42
- Pop-variable-stack, defined 32
- Pop-vs 8
- Pop-vs, defined 39
- Pop-vse, defined 41
- Pos-aggregate-p 10, 34
- Pos-aggregate-p, defined 34
- Pos-int-p 7, 35
- Pre-boolean 60, 61
- Pre-boolean, defined 60
- Pre-character 60, 61
- Pre-character, defined 60
- Pre-integer 60, 61
- Pre-integer, defined 60
- Pre-natural 60, 61
- Pre-natural, defined 60
- Pre-positive 60, 61
- Pre-positive, defined 60
- Pre-string 60, 61
- Pre-string, defined 60
- Pre-type 59, 60
- Pre-type, defined 59
- Predefined-packages, included 7
- Predefined-type-p 61, 94
- Predefined-type-p, defined 61
- Prefix-p-body 85, 90
- Prefix-p-body, defined 85
- Proc-call-actuals 8, 79, 86
- Proc-call-actuals, defined 79
- Proc-call-id 8, 79, 86
- Proc-call-id, defined 79
- Proc-call-p 79, 82
- Proc-call-p, defined 78
- Proc-definedp 95
- Proc-definedp, defined 95
- Proc-lookup 8, 95
- Proc-lookup, defined 95
- Procedure-body 8, 76, 86
- Procedure-body, defined 76
- Procedure-id 35, 76, 86
- Procedure-id, defined 76
- Procedure-p 76, 77, 93, 95
- Procedure-p, defined 76
- Procedure-params 8, 76, 86
- Procedure-params, defined 76
- Procedure-return 76, 86
- Procedure-return, defined 76
- Procedure-spec 8, 76, 86
- Procedure-spec, defined 76
- Prove-ava-op-defthms, defined 19
- Push-as 8, 9
- Push-as, defined 40
- Push-ase 12
- Push-ase, defined 41
- Push-assert, defined 42
- Push-entry, defined 42
- Push-env 8, 10, 11, 12
- Push-env, defined 42
- Push-es 9
- Push-es, defined 40
- Push-ese 12, 13, 35
- Push-ese, defined 40
- Push-value 10, 11, 12, 19
- Push-value, defined 42
- Push-vs 8, 11
- Push-vs, defined 40
- Push-vse, defined 40
- Put-l 50, 51, 52, 53, 55
- Put-l, defined 50
- Put-l, guard verification 52
- Put-l-leaf-p 52
- Put-l-opener-1 53
- Put-l-opener-2 53
- Put-l-opener-3 53
- Put-l-opener-4 53
- Put-t 31, 48, 49, 50, 51, 52, 53, 54
- Put-t, defined 48
- Put-t-opener-1 49
- Put-t-opener-2 49
- Put-t-opener-3 53
- Qualified-p 10, 63, 66
- Qualified-p, defined 63
- Qualified-type 10, 11, 63, 88
- Qualified-type, defined 63
- Qualified-value 10, 11, 63, 88
- Qualified-value, defined 63
- Raise-p 78, 82
- Raise-p, defined 78
- Range 22, 47, 89
- Range, defined 47
- Range-from 26, 27, 28, 34, 62, 89
- Range-from, defined 62
- Range-p 26, 27, 28, 34, 62, 63, 74, 75, 86
- Range-p, defined 62
- Range-size 26, 27, 28
- Range-size, defined 26
- Range-to 26, 27, 28, 34, 62, 89
- Range-to, defined 62
- Record-literal-p 22, 29, 60, 61, 89
- Record-literal-p, defined 60
- Record-type-args-p 84, 89
- Record-type-args-p, defined 84

- Record-type-p 35, 62, 63
- Record-type-p, defined 62
- Rem 20, 21
- Rem-p 72, 74
- Rem-p, defined 72
- Rem-x 73, 87
- Rem-x, defined 72
- Rem-y 73, 87
- Rem-y, defined 73
- Rename-obj-new 78, 86
- Rename-obj-new, defined 78
- Rename-obj-old 78, 86
- Rename-obj-old, defined 78
- Rename-obj-p 78, 93, 94
- Rename-obj-p, defined 77
- Rename-obj-type 78, 86
- Rename-obj-type, defined 78
- Rename-p 78
- Rename-p, defined 78
- Rename-pkg-new 77, 86
- Rename-pkg-new, defined 77
- Rename-pkg-old 77, 86
- Rename-pkg-old, defined 77
- Rename-pkg-p 77, 78, 93, 94
- Rename-pkg-p, defined 77
- Rename-sub-new 77, 86, 93
- Rename-sub-new, defined 77
- Rename-sub-old 77, 86
- Rename-sub-old, defined 77
- Rename-sub-p 77, 78, 93, 94
- Rename-sub-p, defined 77
- Return-p 79, 82
- Return-p, defined 79
- Return-relation-p 67, 68
- Return-relation-p, defined 67
- Return-relation-relation 67, 88
- Return-relation-relation, defined 67
- Return-relation-var 67, 88
- Return-relation-var, defined 67
- Return-value 9, 79, 86
- Return-value, defined 79
- Return-value-p 67, 68
- Return-value-p, defined 67
- Return-value-relation 67, 88
- Return-value-relation, defined 67
- Reverse 8, 9, 11, 12
- Reverse-for-loop-p 80
- Reverse-for-loop-p, defined 80
- Reverse-for-loop-range 80, 86
- Reverse-for-loop-range, defined 80
- Reverse-for-loop-statements 80, 86
- Reverse-for-loop-statements, defined 80
- Reverse-for-loop-var 80, 86
- Reverse-for-loop-var, defined 80
- Root* 8, 30
- Root*, defined 30
- Satisfies-range-constraint 26, 27, 28, 31
- Satisfies-range-constraint, defined 26
- Selected-component-field 10, 30, 65, 88
- Selected-component-field, defined 65
- Selected-component-p 10, 30, 65, 66
- Selected-component-p, defined 65
- Selected-component-root 10, 30, 65, 88
- Selected-component-root, defined 65
- Selected-component-root-decrease 30
- Set-as, defined 39
- Set-difference-theories 3, 24
- Set-equal 22
- Set-es 31
- Set-es, defined 39
- Set-id 68, 88
- Set-id, defined 68
- Set-index 69, 88
- Set-index, defined 68
- Set-p 68, 74
- Set-p, defined 68
- Set-value 69, 88
- Set-value, defined 69
- Set-vs, defined 39
- Set-vse 42
- Set-vse* 42
- Set-vse*, defined 42
- Set-vse, defined 42
- Simple-strip-count=1 57
- Sl-args-p 83, 86
- Sl-args-p, defined 83
- Sl-p 79, 85, 86
- Sl-p, defined 79
- Some-order-exists 37
- Sort-strip-count=1 57
- Sort-strip-count=1-a 57
- Sortl 56, 57
- Sortl, defined 56
- Sortl-count= 56
- Sortl-count=sup-1 56
- Sortl-count=sup-2 56
- Sortl-count=sup-3 56
- St-compound-p 81, 82, 85
- St-compound-p, defined 81
- St-p 82, 83
- St-p, defined 82
- St-simple-p 81, 82
- St-simple-p, defined 81
- Standard-char-p 59, 89, 90
- Statement-opr 8, 12, 33
- Statement-opr, defined 33
- Statement-opr-non-nil-implies-consp-stmt 33
- String< 55
- Stringp 18, 46, 55, 56, 61, 89, 90
- Strip-cars 47
- Strip-cdrs 47, 56, 57, 89
- Strip-cdrs-count-le 56
- Strip-cdrs-count-le-sup-1 56
- Strip-cdrs-count-le-sup-2 56
- Strip-cdrs-le 89
- Strip-cdrs-lt 89
- Strip-cdrs-sort-consp-count 57
- Subprefix-macros, included 58
- Subprefix-norm, included 14
- Subprefix-openers, included 92
- Subprogram-annotation-p 68, 86
- Subprogram-annotation-p, defined 68
- Subprogram-p 77, 78, 83, 86, 93
- Subprogram-p, defined 77
- Subtype-decl-decl 63, 88
- Subtype-decl-decl, defined 63
- Subtype-decl-id 63, 88
- Subtype-decl-id, defined 63
- Subtype-decl-p 63, 78, 93
- Subtype-decl-p, defined 63
- Subtype-p 62, 86, 88, 93, 94, 95, 96
- Subtype-p, defined 62
- Suffix-fns 90, 91
- Symbol-< 55
- Symbol-alistp 19
- Symbol-alistp-ava-operators-alist 19
- Symbolsp 58, 88
- Symbolsp, defined 58
- Top 19
- Top-as 4
- Top-as, defined 39
- Top-es 4
- Top-es, defined 39
- Top-literal-p 89
- Top-literal-p, defined 89
- Top-literal-p-count 89
- Top-prefix-decls-p 33
- Top-prefix-decls-p, defined 33
- Top-prefix-decls-p2 33
- Top-prefix-decls-p2, defined 33
- Top-prefix-p 8, 12, 14, 89, 90
- Top-prefix-p, defined 89
- Top-prefix-p, guard verification 25
- Top-value 8, 9, 11, 12, 13, 19
- Top-value, defined 39
- Top-vs 4, 8, 9, 11
- Top-vs, defined 39
- Top-vse, defined 39
- Transition-p 67, 85
- Transition-p, defined 67
- Transition-relation 67, 88
- Transition-relation, defined 67
- Tree-not-leaf 21
- Treep 21, 22, 29, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 57, 60
- Treep, defined 46
- Treep-alistp 47
- Treep-cdr 47
- Treep-cdr-fwd 47
- Treep-cons 47
- Treep-cons-car 49
- Treep-cons-cdr 50
- Treep-cons-put-t 52
- Treep-imp-true-listp 57
- Treep-insert 57
- Treep-nil 47
- Treep-not-leaf-p 47

Treep-put-l 51
 Treep-put-t 50
 Treep-sortl 57
 Treep-sortl=1 57
 Treep-sortl=2 57
 Treep-sortl=2-1 57
 Treep-valuep 47
 True 9, 11, 21, 22, 24, 25, 33
 True-listp-sortl 56
 True-p 24, 61
 True-p, defined 61
 Truncate 20, 21
 Type-check-macros, included 14
 Type-convert-exc 11, 28
 Type-convert-exc, defined 28
 Type-convert-p 11, 64, 66
 Type-convert-p, defined 63
 Type-convert-type 11, 64, 88
 Type-convert-type, defined 64
 Type-convert-val 11, 28
 Type-convert-val, defined 28
 Type-convert-value 11, 64, 88
 Type-convert-value, defined 64
 Type-decl-decl 63, 89
 Type-decl-decl, defined 63
 Type-decl-id 63, 88
 Type-decl-id, defined 63
 Type-decl-p 63, 78, 93
 Type-decl-p, defined 63
 Type-indication-p 94
 Type-indication-p, defined 94
 Type-mark-constraint 34, 62, 89
 Type-mark-constraint, defined 62
 Type-mark-p 34, 61, 62, 89, 94
 Type-mark-p, defined 61
 Type-mark-type 62, 89
 Type-mark-type, defined 62
 Type-p 63, 86, 87, 88, 89
 Type-p, defined 63
 Type-tree-labels 11, 35
 Type-tree-labels, defined 35

Unconstrained-p 26, 32, 60, 62
 Unconstrained-p, defined 60
 Union-theories 3, 91
 Update-v 42
 Update-v, defined 42
 Update-value-exc 31
 Update-value-exc, defined 31
 Update-value-val 31
 Update-value-val, defined 31
 Upperp 36
 Upperp-not-lowerp 36
 Use-args-p 84, 86
 Use-args-p, defined 84
 Use-p 77
 Use-p, defined 77

Valuep 47, 48, 49, 50, 51, 52, 53,
 54, 55
 Valuep, defined 47
 Valuep-cons-put-l-nil 52
 Valuep-get-l 51
 Valuep-get-l-1 51
 Valuep-get-l-list-i 51
 Valuep-get-t 51
 Valuep-put-l 51
 Valuep-put-t 51
 Var 30
 Variable-lookup 10, 31, 95
 Variable-lookup, defined 95
 Variable-p 35, 75
 Variable-p, defined 75
 Variable-update 30, 31
 Variable-update, defined 30
 Variable-update-1 29, 30
 Variable-update-1, defined 29
 Vs 4, 7, 15, 19, 42
 Vs, defined 38
 Vs-p 15
 Vs-p* 15
 Vs-p*, defined 15
 Vs-p, defined 15

While-loop-p 79, 80
 While-loop-p, defined 79
 While-loop-statements 9, 79, 86
 While-loop-statements, defined 79
 While-loop-test 9, 79, 86
 While-loop-test, defined 79

Zerop 4

Table of Contents

Chapter 1. Introduction	1
1.1. Notation	1
1.2. Notes on the Implementation	6
Chapter 2. Operational Definition	7
2.1. Subsidiary Routines	14
2.1.1. Constants	16
2.1.2. Errors	16
2.1.3. Ava Literals, Values and Types	18
2.1.4. Operators	18
2.1.5. Expression evaluation	19
2.1.6. Numeric Operators	20
2.1.7. Boolean Operators	21
2.1.8. Unary Numeric Operators	23
2.1.9. Application of Operators to Evaluated Arguments	24
2.2. Arrays and Records	26
2.3. Types, including Conversion and Qualification	26
2.4. More Expression Evaluation	29
2.5. Statement Evaluation Support	29
2.5.1. Assignment	30
2.5.2. Procedure Call Support	32
2.6. Interpreter	32
2.7. Input	36
Appendix A. Support	38
A.1. Macros	38
A.2. Ilet	43
A.3. Trees for Array and Record Literals	45
A.4. Support for Tree Equality Reasoning	55
A.5. Normalized Abstract Prefix	58
A.6. Static Semantics Macros	92
A.7. Predefined Packages	96
Index	100

List of Figures

List of Tables