(proclaim '(special \*CONSTANT-LIST\* \*TYPE-LIST\* \*PREDICATE-LIST\* \*PRINT-TYPE\*

\*OPERATORS\* \*INFERENCE-RULES\* \*OPS\* \*TYPE-ARRAY\*

\*CONNECTIONS\* \*GROUP-GRAPH\* \*TRANSFORMATIONS\* \*DROPPED\*

\*GROUP-INTERACTION\* \*VARIABLE-TYPING\* \*PRIMARY\*

\*ABSTRACTION-LEVEL\* \*ABS-PRINT-FLAG\* \*STATIC\*

\*TYPES\* \*TYPE-TABLE\* \*INSTANCE-TABLE\*

\*ARG-TYPE-TABLE\* \*PRIMARY-SCRULES\* \*VARIABLE-EXPANSION\*

\*SINGLE-HIERARCHY\* \*INS-TYPES\* \*ABS-OPERATORS\*

\*ABS-INFERENCE-RULES\* \*ABS-SCR-GOAL-SELECT-RULES\*

\*LEARNED-RULES-IN-SYS\* \*SCR-GOAL-SELECT-RULES\*

\*CREATE-SCRULES\*

))

(eval-when (compile)

(load-path \*PLANNER-PATH\* "data-types"))

;;;

;;; A formula is a ground level predicate that occurs as a precondition or effect

;;; of an operator.

;;;

(defstruct (formula (:print-function (lambda (formula stream level)

(declare (ignore stream level))

(format stream "#S~a" (formula-type formula)))))

instance ;;; e.g., (in-room block1 room5) possibly negated

type ;;; e.g., (in-room block room) possibly negated

preeffect ;;; one of 'precond, 'effect

status ;;; one of 'primary or 'secondary

operator) ;;; name of the operator

;;;

;;;

(setq \*PRINT-TYPE\* nil) ;;; Print out the type information as the algorithm runs

(setq \*ABS-PRINT-FLAG\* nil)

;;;

;;;

;;;

;;; Top level function for statically producing an hierarchy of non-interacting

;;; abstractions. (load-domain) has to be called first.

;;;

(defun create-abstraction ()

(setq \*ABSTRACTION-LEVEL\* nil) ; Used in the execution of an abstraction space.

(if (boundp '\*TYPE-LIST\*)

(mapcar #'(lambda (x)(setf (get x 'structure) nil)) \*TYPE-LIST\*))

(setq \*TYPE-LIST\* nil) ;;; List of the types in the domain

(setq \*PREDICATE-LIST\* nil);;; List of the predicates in the domain

;; Hash table that maps literals to types

(setq \*TYPE-TABLE\* (make-hash-table :test #'equal))

(setq \*INSTANCE-TABLE\* (make-hash-table :test #'equal))

(setq \*ARG-TYPE-TABLE\* (make-hash-table :test #'equal))

(mapcar #'eval \*VARIABLE-TYPING\*)

(if (null \*VARIABLE-TYPING\*)(setq \*VARIABLE-EXPANSION\* nil))

(set-variable-expansion \*VARIABLE-EXPANSION\*)

(create-scrules-if-needed)

(setq \*ABS-OPERATORS\* \*OPERATORS\*)

(setq \*ABS-INFERENCE-RULES\* \*INFERENCE-RULES\*)

(setq \*ABS-SCR-GOAL-SELECT-RULES\* \*SCR-GOAL-SELECT-RULES\*)

(load-domain) ; Must be after create-scrules and before process-domain.

(process-domain)

(assign-criticalities)

; (build-partial-order)

; (if \*SINGLE-HIERARCHY\* (build-hierarchy \*SINGLE-HIERARCHY\*))

(load-abstract-control)

(load-abstract-domain))

(defun create-scrules-if-needed ()

(if (and (or (not (boundp '\*CREATE-SCRULES\*))

(not (null \*CREATE-SCRULES\*)))

(or (not (boundp '\*PRIMARY-SCRULES\*))

(not (eq \*PRIMARY-SCRULES\* \*PRIMARY\*))))

(progn (create-scrules \*PRIMARY\*)

(setq \*PRIMARY-SCRULES\* \*PRIMARY\*))))

(defun load-abstract-domain ()

(dolist (op \*OPERATORS\*)

(load-abstract-operator op))

(dolist (inference-rule \*INFERENCE-RULES\*)

(load-abstract-inference inference-rule)))

(defun load-abstract-control ()

(dolist (rules '(\*SCR-NODE-SELECT-RULES\* \*SCR-GOAL-SELECT-RULES\*

\*SCR-OP-SELECT-RULES\* \*SCR-BINDINGS-SELECT-RULES\*

\*SCR-NODE-REJECT-RULES\* \*SCR-GOAL-REJECT-RULES\*

\*SCR-OP-REJECT-RULES\* \*SCR-BINDINGS-REJECT-RULES\*

\*SCR-NODE-PREFERENCE-RULES\* \*SCR-GOAL-PREFERENCE-RULES\*

\*SCR-OP-PREFERENCE-RULES\* \*SCR-BINDINGS-PREFERENCE-RULES\*

\*LEARNED-RULES-IN-SYS\*))

(dolist (rule (eval rules))

(if (listp rule)

(setf (scr-goals (car rule))(extract-goals (car rule)))

(setf (scr-goals rule)(extract-goals rule))))))

(defun load-abstract-operator (op)

(let ((name (car op)))

(setf (get name 'preconds)

(get name 'abs-preconds))

(setf (get name 'effects)

(get name 'abs-effects))

(setf (get name 'vars)(get name 'abs-vars))

(setf (get name 'all-vars)(get name 'abs-all-vars))

(setf (get name 'params)(get name 'abs-vars))

(make-add-and-delete-lists name (get name 'effects))

(add-lpreconds op)

))

(defun load-abstract-inference (inference-rule)

(let ((name (car inference-rule)))

(setf (get name 'preconds)

(get name 'abs-preconds))

(setf (get name 'effects)

(get name 'abs-effects))

(setf (get name 'vars)(get name 'abs-vars))

(setf (get name 'all-vars)(get name 'abs-all-vars))

(setf (get name 'params)(get name 'abs-vars))

(make-add-and-delete-lists name (get name 'effects))

(add-lpreconds inference-rule)

))

;;; The following code extracts all the problem space literals from

;;; a search control rule. These literals stored under scr-goals and

;;; used to determine if a control rule is relevant to an abstraction space.

(defun extract-goals (control-rule-name)

(let ((predicates (set-difference \*predicate-list\*

(mapcar #'(lambda (x)(car x)) \*static\*))))

(remove-duplicates (append (extract-abs-goals (scr-rhs control-rule-name)

predicates)

(extract-abs-goals (scr-lhs control-rule-name)

predicates)))))

(defun extract-abs-goals (exp predicates)

(cond ((null exp) nil)

((atom exp) nil)

((atom (car exp))

(cond ((member (car exp) predicates)

(list (extract exp)))

(t (extract-abs-goal-list (cdr exp) predicates))))

(t (append (extract-abs-goals (car exp) predicates)

(extract-abs-goals (cdr exp) predicates)))))

(defun extract-abs-goal-list (exp-list predicates)

(cond ((null exp-list) nil)

((atom (car exp-list))

(extract-abs-goal-list (cdr exp-list) predicates))

(t (append (extract-abs-goals (car exp-list) predicates)

(extract-abs-goal-list (cdr exp-list) predicates)))))

;;;

;;;

;;;

;;; This function constructs the partial order of abstraction spaces based

;;; on the monotonicity property.

;;;

(defun build-partial-order ()

(let\* ((ops (operator-names))

(type-array (group-predicates ops \*types\* \*ins-types\*)))

(order-groups \*ins-types\* type-array \*connections\*)

(create-context-array ops \*ins-types\*)))

;;;

;;;

;;;

;;; The domain is processed twice so that there are two data structures for each

;;; operator that are identical except that the variables are renamed in one.

;;; This is used for analysis on operators that loop on themselves.

;;;

(defun process-domain ()

(process-domain-definition nil)

(process-domain-definition t)

(if \*ABS-PRINT-FLAG\*

(and (format t "~%~%Predicates: ~a" \*PREDICATE-LIST\*)

(format t "~%~%Types: ~a" \*TYPE-LIST\*)

(format t "~%Static Preconditions: ~a" \*STATIC\*)))

)

;;;

;;;

;;;

;;; Processes a domain definition by creating the appropriate data structures

;;; and marking the primary effects and the static predicates of each operator.

;;;

(defun process-domain-definition (rename-vars)

(let ((ops (operator-names)))

(compute-type-information ops 0 rename-vars t)

(determine-types ops)

(determine-primary-effects ops)

(mark-all-static-predicates ops \*types\*)))

;;;

;;;

;;;==============================================================================

;;;

;;; Extract the names of all the opeators and inference rules

;;;

;;;

(defun operator-names ()

;;; (setq \*OPS\* (find-names \*OPERATORS\*)))

(setq \*OPS\* (append (find-names \*OPERATORS\*)(find-names \*INFERENCE-RULES\*))))

;;;

;;;

(defun find-names (ops)

(cond ((null ops) nil)

(t (cons (caar ops)(find-names (cdr ops))))))

;;;

;;;

;;;==============================================================================

;;;

;;;

;;; Extract the types, predicates, and constants from the operators.

;;;

;;; We still need to think about type hierarchies here.

;;;

;;; Recur through the list of operators and determine the types for preconditions

;;; and effects. This information is stored under the operators in 'typed-preconds

;;; and 'typed-effects. The previous values of these two properties are stored

;;; under 'unrenamed-typed-precond and 'unrenamed-typed-effects. This is a

;;; hack that essentially stores two versions of each operator with different

;;; variable names. This is used for operators that loop on themselves since

;;; the preconditions of an operator may need to be matched against the

;;; preconditions

;;; of the same operator and the matcher requires unique variable names in each

;;; expression. The rename-vars argument determines whether the variables are

;;; renamed.

;;;

(defun compute-type-information (ops num rename-vars augment)

(cond ((null ops) nil)

(t (setf (get (car ops) 'preconds) ; used by learning stuff

(setf (get (car ops) 'abs-preconds) ;used by planning stuff

(if augment

(augment-goal-exp (get (car ops) 'preconds))

(get (car ops) 'preconds))))

(setf (get (car ops) 'abs-effects)

(get (car ops) 'effects))

(setf (get (car ops) 'abs-all-vars)

(find-all-vars (list (get (car ops) 'abs-preconds)

(get (car ops) 'abs-effects))))

(setf (get (car ops) 'abs-vars)

(make-same-order-as-params

(car ops)

(remove-duplicates

(append (get-outside-existentials

(get (car ops) 'abs-preconds))

(lst-difference (get (car ops) 'all-vars-in-effects)

(get (car ops) 'wildcard-vars)))

:test #'equal)))

(setf (get (car ops) 'unrenamed-typed-preconds)

(get (car ops) 'typed-preconds))

(setf (get (car ops) 'unrenamed-typed-effects)

(get (car ops) 'typed-effects))

(if rename-vars (rename-variables (car ops) num))

(compute-type-info (car ops))

(compute-type-information (cdr ops) (1+ num) rename-vars t))))

;;;

;;;

;;;

;;; This function takes a list of typed literals and augments it with

;;; the negations of some of those literals. It determines which literals

;;; can be negated using several global variables from load-domain. The

;;; negable literals are those closed world literals that are either added

;;; or deleted by some operator. The purpose of this augmentation is so

;;; that you could specify negated or unnegated literals in a goal even if

;;; they can't be achieved by an operator. You might want to do this, for

;;; example, to prevent the plan from deleting some condition that cannot

;;; be reachieved.

;;;

;(defun augment-types (types)

; (augment-types-with-negations

; types

; (intersection \*closed-preds\*

; (union \*achievable-preds\* \*negatable-preds\*))))

;;;

;(defun augment-types-with-negations (types negatable-types)

; (cond ((null types) nil)

; ((member (car (first types)) negatable-types)

; (append (list (car types) (list '~ (car types)))

; (augment-types-with-negations (cdr types) negatable-types)))

; ((and (eq '~ (car (first types)))

; (member (caadr (first types)) negatable-types))

; (append (list (cadar types)(car types))

; (augment-types-with-negations (cdr types) negatable-types)))

; (t (cons (car types)

; (augment-types-with-negations (cdr types) negatable-types)))))

;;;

;;;

;;;

;;; This function types an operator and a number and appends the number to all

;;; of the variables in that operator.

;;;

(defun rename-variables (op num)

(let\* ((all-vars (remove-duplicates (append (extract-vars (get op 'preconds))

(extract-vars (get op 'effects)))))

(subst-vars (rename-vars all-vars num)))

(setf (get op 'renamed-vars)(mapcar #'cdr subst-vars))

(setf (get op 'preconds)(sublis subst-vars (get op 'preconds)))

(setf (get op 'effects)(sublis subst-vars (get op 'effects)))))

;;;

;;;

;;;

;;; Takes a tree and extracts all of the variables from that tree.

;;;

(defun extract-vars (tree)

(cond ((null tree) nil)

((listp tree)(append (extract-vars (car tree))

(extract-vars (cdr tree))))

((is-variable tree)(list tree))))

;;;

;;;

;;;

;;; Takes a list of variables and forms an association list with the old

;;; variable name and the new variable name that can be used by sublis.

;;;

(defun rename-vars (vars num)

(cond ((null vars) nil)

(t (cons (cons (car vars)

(intern (rename-var (string (car vars))

0

(princ-to-string num)

nil)))

(rename-vars (cdr vars) num)))))

;;;

;;;

;;;

;;; Takes a particular variable and constructs the new variable name. Note

;;; that the variable consists of two parts: the type and the name separated by

;;; a period. The name part is optional, but if the variable doesn't initially

;;; have a name, then the period must be explicitly added before the number.

;;;

(defun rename-var (var pos num period-flag)

(cond ((eq #\> (schar var pos))

(cond (period-flag (concatenate 'string num (string (schar var pos))))

(t (concatenate 'string "." num (string (schar var pos))))))

(t (concatenate 'string

(string (schar var pos))

(rename-var var

(1+ pos)

num

(or period-flag (eq #\. (schar var pos))))))))

;;;

;;;

;;;

;;;==============================================================================

;;;

;;;

;;;

;;;

;;; Computes the typing for the preconds and effects and stores them in the

;;; operator.

;;;

(defun compute-type-info (op)

(cond (\*PRINT-TYPE\* (format t "~%~%Computing type info for: ~a" op)

(format t "~%Preconds: ")))

(setf (get op 'typed-preconds)

(compute-typed-preconds op (simplify-exp (get op 'preconds))))

(setf (get op 'context-preconds)

(list-of-conjuncts (dnf (partially-simplify-exp (get op 'preconds)))))

(if \*PRINT-TYPE\* (format t "~%Effects: "))

(setf (get op 'typed-effects)(compute-typed-effects op (get op 'effects))))

;;;

;;;

;;;

;;; Given a dnf expression, returns a list of the conjuncts with the

;;; ands and ors.

;;;

(defun list-of-conjuncts (dnf-exp)

(cond ((null dnf-exp) nil)

((eq 'and (car dnf-exp))(list (cdr dnf-exp)))

((eq 'or (car dnf-exp))(list-of-conjuncts (cdr dnf-exp)))

((not (listp (car dnf-exp)))(list (list dnf-exp)))

(t (append (list-of-conjuncts (car dnf-exp))

(list-of-conjuncts (cdr dnf-exp))))))

;;;

;;; Takes an arbitrary expression and turns it into a disjunctive

;;; normal form expression.

(defun dnf (exp)

(cond ((null exp) nil)

((and (eq '~ (car exp))(or (eq 'or (caadr exp))(eq 'and (caadr exp))))

(dnf (move-negation (cadr exp))))

((eq 'or (car exp))

(cons 'or (check-disjuncts (cdr exp))))

((eq 'and (car exp))

(move-or-outside (cdr exp)))

(t exp)))

;;;

;;; Moves negations to the inside. I don't think we need to do this

;;; since prodigy doesn't allow negated ands and ors.

;;;

(defun move-negation (exp)

(cond ((null exp) nil)

((eq 'or (car exp))

(cons 'and (move-negation (cdr exp))))

((eq 'and (car exp))

(cons 'or (move-negation (cdr exp))))

((eq '~ (caar exp))

(cons (cadr exp)(move-negation (cdr exp))))

(t (cons (list '~ (car exp))

(move-negation (cdr exp))))))

;;;

;;;

;;; Move ors to the outside of the expression.

(defun move-or-outside (exp)

(cond ((null exp) nil)

(t (and-expressions (dnf (car exp))

(move-or-outside (cdr exp))))))

;;;

;;;

;;; Verify that each disjunct is in disjunctive normal form.

(defun check-disjuncts (exp)

(cond ((null exp) nil)

(t (cons (dnf (car exp))

(check-disjuncts (cdr exp))))))

;;;

;;; And two expressions together combining the ors.

;;;

(defun and-expressions (exp1 exp2)

(cond ((null exp2) exp1)

((and (eq 'or (car exp1))

(eq 'or (car exp2)))

(cons 'or (combine-or (cdr exp1)

(cdr exp2))))

((eq 'or (car exp1))

(cons 'or (combine-or (cdr exp1)

(list exp2))))

((eq 'or (car exp2))

(cons 'or (combine-or (list exp1)

(cdr exp2))))

((and (eq 'and (car exp1))

(eq 'and (car exp2)))

(cons 'and (append (cdr exp1)

(cdr exp2))))

((eq 'and (car exp1))

(cons 'and (append (cdr exp1)

(list exp2))))

((eq 'and (car exp2))

(cons 'and (cons exp1 (cdr exp2))))

(t (list 'and exp1 exp2))))

;;;

(defun combine-or (or-list1 or-list2)

(cond ((null or-list1) nil)

(t (append (combine-or-list (car or-list1)

or-list2)

(combine-or (cdr or-list1)

or-list2)))))

;;;

(defun combine-or-list (or1 or-list2)

(cond ((null or-list2) nil)

(t (cons (and-expressions or1 (car or-list2))

(combine-or-list or1 (cdr or-list2))))))

;;;

;;;

;;;

;;; This function takes a PDL expression and converts it to an expression

;;; consisting of conjunctions and disjunctions of literals.

;;;

(defun partially-simplify-exp (preconds)

(cond ((null preconds) nil)

((eq 'and (car preconds))

(cons 'and (partially-simplify-exp (cdr preconds))))

((eq 'or (car preconds))

(cons 'or (partially-simplify-exp (cdr preconds))))

((eq 'exists (car preconds))

(partially-simplify-exp (cddr preconds)))

((eq 'forall (car preconds)) nil) ;;; Might be an empty generator

((and (eq '~ (car preconds))

(eq 'exists (caadr preconds)))

nil) ;;; ~exists == forall with a possible empty generator

((eq '~ (car preconds)) preconds)

((not (listp (car preconds))) preconds)

(t (append (list (partially-simplify-exp (car preconds)))

(partially-simplify-exp (cdr preconds))))))

;;;

;;;

;;;

;;; This function types a PDL expression and converts it to a conjunction

;;; of literals.

;;;

(defun simplify-exp (preconds)

(cond ((null preconds) nil)

((eq 'and (car preconds))

(simplify-exp (cdr preconds)))

((eq 'or (car preconds))

(simplify-exp (cdr preconds)))

((eq 'exists (car preconds))

(simplify-exp (cddr preconds)))

((eq 'forall (car preconds))

(simplify-exp (cddr preconds)))

((and (eq '~ (car preconds))

(eq 'exists (caadr preconds)))

(simplify-exp (cadr preconds)))

((eq '~ (car preconds))(list preconds))

((not (listp (car preconds))) (list preconds))

(t (append (simplify-exp (car preconds))

(simplify-exp (cdr preconds))))))

;;;

;;;

;;; Recurs through the preconditions and creates a formula structure for each

;;; of the preconditions.

;;;

;;; Assumes only conjunctive preconditions right now.

;;;

(defun compute-typed-preconds (op preconds)

(cond ((null preconds) nil)

(t (if \*PRINT-TYPE\*

(format t "~% ~a ==> ~a"

(car preconds)(extract (car preconds))))

(cons (make-formula

:instance (car preconds)

:type (extract (car preconds))

:preeffect 'precond

:operator op)

(compute-typed-preconds op (cdr preconds))))))

;;;

;;;

;;;

;;; Recurs through the effects and determines the types. If it comes across a

;;; conditional effect it simply drops the conditional.

;;;

(defun compute-typed-effects (op effects)

(cond ((null effects) nil)

(t (cons (type-effect op

(remove-conditional (car effects))

(eq 'if (car (first effects))))

(compute-typed-effects op (cdr effects))))))

;;;

;;;

;;;

;;; Extracts the effect from a conditional effect.

;;;

(defun remove-conditional (effect)

(cond ((eq 'if (car effect))(caddr effect))

(t effect)))

;;;

;;;

;;;

;;; Creates a formula structure for an effect. If the secondary flag is set

;;; then the status is set to secondary, otherwise it is left unspecified.

;;;

(defun type-effect (op effect secondary)

(if \*PRINT-TYPE\*

(format t "~% ~a ==> ~a" (cadr effect)(extract (cadr effect))))

(make-formula

:instance (if (eq 'del (car effect))

(list '~ (cadr effect))

(cadr effect))

:type (if (eq 'del (car effect))

(list '~ (extract (cadr effect)))

(extract (cadr effect)))

:preeffect 'effect

:status (if secondary 'secondary)

:operator op))

;;;

;;;

;;;

;;; Given a particular instance if a formula, this function determines the types

;;; and and updates the list of predicates, constants, and types.

;;;

(defun extract (formula)

(cond ((gethash formula \*TYPE-TABLE\*))

((eq '~ (car formula))

(setf (gethash formula \*TYPE-TABLE\*)

(new-type (list '~ (extract (cadr formula))))))

(t (setf (gethash formula \*TYPE-TABLE\*)

(new-type (cons (add-predicate (car formula))

(extract-args (cdr formula))))))))

;;;

;;; This function essures that all identical types will be eq.

;;;

(defun new-type (type)

(cond ((gethash type \*TYPE-TABLE\*))

(t (setf (gethash type \*TYPE-TABLE\*) type)

type)))

;;;

;;;

;;;

;;; Recur through the argument list.

;;;

(defun extract-args (args)

(cond ((null args) nil)

(t (cons (arg-type (car args))

(extract-args (cdr args))))))

;;;

;;;

;;;

;;; Given a particular instance if a formula, this function determines the instance

;;; and and updates the list of predicates, constants, and types.

;;;

(defun extract-instance (formula)

(if (null \*VARIABLE-EXPANSION\*)

(extract formula)

(cond ((gethash formula \*INSTANCE-TABLE\*))

((eq '~ (car formula))

(setf (gethash formula \*INSTANCE-TABLE\*)

(list '~ (extract-instance (cadr formula)))))

(t (setf (gethash formula \*INSTANCE-TABLE\*)

(cons (add-predicate (car formula))

(extract-instance-args (cdr formula))))))))

;;;

;;;

;;;

;;; Recur through the argument list.

;;;

(defun extract-instance-args (args)

(cond ((null args) nil)

(t (cons (arg-instance-type (car args))

(extract-instance-args (cdr args))))))

;;;

;;;

;;;

;;; This function returns the types of the arguments. If typing

;;; is turned off, then it will always return 'type'. If

;;; type-expansion is turned off for the particular type it will

;;; return the type, otherwise it returns the original argument.

;;;

(defun arg-instance-type (arg)

(cond ((null \*VARIABLE-TYPING\*) 'type)

(t (let ((type (arg-type arg)))

(cond ((type-expansion (get type 'structure)) arg)

(t type))))))

;;;

;;;

;;; Meta predicate to check the argument type of a variable.

(defun is-arg-type (arg type)

(eq type (arg-type arg)))

;;;

;;;

;;;

;;; Determine the type of an argument. Arguments are of the form: foo.bar

;;; where foo is the type and the .bar is optional, but is used to distinguish

;;; multiple variables of the same type. If typing is turn off simply use

;;; type for every variable. Note that types are stored under the argument name

;;; to avoid redoing the typing.

;;;

(defun arg-type (arg)

(cond ((null \*VARIABLE-TYPING\*) 'type)

((gethash arg \*ARG-TYPE-TABLE\*))

((not (is-variable arg))

(cond ((null (gethash arg \*ARG-TYPE-TABLE\*))

(error "~%Need to declare type of: ~a. Use isa-instance." arg))

(t (gethash arg \*ARG-TYPE-TABLE\*))))

(t (setf (gethash arg \*ARG-TYPE-TABLE\*)

(check-arg-type arg (intern (argument-type (symbol-name arg) 0)))))))

;;;

;;;

;;;

;;; Determines the type of a argument by recurring through the string until

;;; either a '.' or a '>' is reached.

;;;

(defun argument-type (arg-string pos)

(cond ((or (= pos (array-dimension arg-string 0))

(eq #\. (schar arg-string pos))

(eq #\> (schar arg-string pos)))

nil)

((eq #\< (schar arg-string pos))

(argument-type arg-string (1+ pos)))

(t (concatenate 'string

(string (schar arg-string pos))

(argument-type arg-string (1+ pos))))))

;;;

;;;

;;;

;;; Maintain lists of predicates and types for informational purposes.

;;;

(defun add-predicate (predicate)

(cond ((member predicate \*PREDICATE-LIST\*) predicate)

(t (setq \*PREDICATE-LIST\* (cons predicate \*PREDICATE-LIST\*))

predicate)))

;;;

;;;

;;; Check that the given argument is a legal type.

;;;

(defun check-arg-type (arg type)

(cond ((member type \*TYPE-LIST\*) type)

(\*EBL-FLAG\* 'type)

(t (error "~%~%Invalid type: ~a for argument: ~a" type arg))))

;;;

;;; This is the defstruct for "type". There is a corresponding

;;; defstruct for each type in the system. We need to make this

;;; one explicit in order to compile this file.

;;;

(defstruct (type (:conc-name nil)

(:print-function

(lambda (struct stream level)

(declare (ignore level))

(format stream "#S(~a children ~a)"

(type-of struct)

(type-children struct))))

)

type-parent type-children type-instances type-expansion)

;;;

;;;

;;;

(defun set-variable-expansion (var-list)

(cond ((or (null var-list)(eq t var-list)))

(t (setf (type-expansion (get (car var-list) 'structure)) t)

(set-variable-expansion (cdr var-list)))))

;;;

;;;

;;; Incrementally create each type. This involves create a defstruct

;;; for each type, pointing the type to an instance of the defstruct

;;; under the property 'structure, and adding pointers to the correct

;;; parent and children.

;;;

(defun isa (child parent)

(let ((parent-struct (add-type parent))

(child-struct (add-type child)))

(setf (type-children parent-struct)

(cons child-struct (type-children parent-struct)))

(setf (type-parent child-struct) parent-struct)))

;;;

;;;

;;; Incrementally create each instance.

;;;

(defun isa-instance (instance type)

(let ((type-struct (get type 'structure)))

(cond ((not (member instance (type-instances type-struct)))

(setf (type-instances type-struct)

(cons instance (type-instances type-struct)))

(setf (gethash instance \*ARG-TYPE-TABLE\*) type)))))

;;;

;;;

;;; Add the type to \*TYPE-LIST\* and create the appropriate defstruct.

;;;

(defun add-type (type)

(cond ((member type \*TYPE-LIST\*) (get type 'structure))

(t (setq \*TYPE-LIST\* (cons type \*TYPE-LIST\*))

(setf (gethash type \*ARG-TYPE-TABLE\*) type)

(eval `(defstruct (,type (:conc-name nil)

(:print-function

(lambda (struct stream level)

(declare (ignore level))

(format stream "#S(~a children ~a)"

(type-of struct)

(type-children struct))))

)

type-parent type-children type-instances type-expansion))

(setf (get type 'structure)

(funcall (symbol-function

(intern (concatenate 'string

"MAKE-" (symbol-name type)))))))))

;;;

;;;

;;;

;;; This function is given the set of operators and determines all of

;;; the appropriate literal types. It does this as follows:

;;; First, find all of the types in the effects of operators, then

;;; call generate-descendents to expand these into all of the type

;;; descendents.

;;;

(defun determine-types (ops)

(setq \*types\* (delete-duplicates

(find-types (find-effects ops))

:test #'equal))

(if (null \*VARIABLE-EXPANSION\*)

(setq \*ins-types\* \*types\*)

(setq \*ins-types\* (mapcan #'(lambda (x)(generate-descendents x nil))

\*types\*))))

;;;

;;;

;;; Finds those types from the preconditions that are in all-types,

;;; but not in types.

;;;

;(defun find-additional-types (preconds types all-types)

; (cond ((null preconds) nil)

; ((and (not (member (formula-type (car preconds)) types :test #'equal))

; (member (formula-type (car preconds)) all-types :test #'equal))

; (cons (formula-type (car preconds))

; (find-additional-types (cdr preconds) types all-types)))

; (t (find-additional-types (cdr preconds) types all-types))))

;;;

;;;

;;; Take the given literal and generate all of the type descendents.

;;; The expand flag determines whether all of the descendents are

;;; generated or if it is constrained by the type-expansion bit.

;;;

(defun generate-descendents (literal expand)

(cond ((eq '~ (car literal))

(mapcar #'(lambda (x)(list '~ x))

(generate-descendents (cadr literal) expand)))

((null (cdr literal))(list literal))

(t (mapcar #'(lambda (x)(cons (car literal) x))

(generate-each-type (cdr literal) expand)))))

;;;

(defun generate-each-type (types expand)

(cond ((null types) nil)

((null (cdr types))

(mapcar #'(lambda (x)(list x))

(generate-type-descendents (car types) expand)))

(t (mapcan #'(lambda (x)

(mapcar #'(lambda (y) (cons x y))

(generate-each-type (cdr types) expand)))

(generate-type-descendents (car types) expand)))))

;;;

(defun generate-type-descendents (type expand)

(if (symbolp type)

(let ((type-struc (get type 'structure)))

(cond ((null type-struc) (list type))

((or expand (type-expansion type-struc))

(type-instances type-struc))

(t (list type))))

(list type)))