OPT Manual
Version 1.7.3 (Reflects Opt Version 1.6.11)
* DRAFT **

Drew McDermott

November 13, 2005

Change history:
V. 1.7 2004-07-19 Document web mode and namespace mode

Contents

1 Introduction and Overview

OPT stands for Ontology with Polymorphic Types. It is an attempt to create a general-purpose notation for creating *ontologies,* defined as formalized conceptual frameworks for domains about which programs are to reason. Its syntax is based on PDDL [**?**], but it has a more elaborate type system, which allows users to make use of higher-order constructs such as explicit λ-expressions. OPT is intended to be (almost) upwardly compatible with PDDL 2.1, the dialect used in the 2002 International Planning Competition. Some of the prose in this document is lifted is lifted straight from previous PDDL documentation (which in turn may have been lifted from the UNPOP language manual). Where OPT differs from PDDL 2.1, I will point out how it differs. Opt borrows much of its notation for processes and durative actions from [**?**].

It's my hope that PDDL will in fact evolve in the direction of OPT, but even if it doesn't, OPT will continue to include PDDL as a subset for obvious reasons. [[One difference between lang and PDDL is that less of an effort has been made in the design of lang to separate "advice" from "physics." Then again, the effort was a bit ill defined even for PDDL.]]

In OPT, as in PDDL, a formalized ontology is called a *domain*. Domains can be related by inheritance relations, so very general ontologies can be used over and over. Not all domains involve the same level of expressivity. For example, some domains may be based on the assumption that any variable-free atomic formula not known to be true may be assumed to be false; this is called a "closed-world assumption." In fact, this is the default in OPT, as in many AI applications, because closed worlds tend to be more tractable. However, there are domains where such an assumption is inappropriate, in which case verifying that a proposition is false requires a more substantial inference than failing to prove it true. To control these differing kinds of domains, we borrow from the UCPOP language [**?**] the idea of *requirements flags*. Any domain that departs from the simplest subset of OPT must declare the requirements that it is based on

```
(define (domain astronomy)
    (:requirements :open-world) ...)
```

In this case we say that `astronomy` *declares the requirement* `:open-world`.

Here is an example domain, a simple version of an ontology for an agent on the World-Wide Web:

```
(define (domain www-agents)
    (:extends knowing regression-planning commerce)
    (:requirements :existential-preconditions :conditional-effects)
    (:types Message - Obj Message-id - String•)

    (:type-fun (Key t) (Feature-type (keytype t)))
```

```
(:type-fun (Key-pair t) (Tup• (Key t) t))
(:functions (price-quote ?m - Money)
            (query-in-stock ?pid - Product-id)
            (reply-in-stock ?b - Boolean)
            - Message)
(:predicates (web-agent ?x - Agent)
             (reply-pending a - Agent id - Message-id msg - Message)
             (message-exchange ?interlocutor - Agent
                               ?sent ?received - Message
                               ?eff - Prop)
             (expected-reply a - Agent sent expect-back - Message))
(:facts
     (freevars (?agt - Agent ?msg-id - Message-id
                ?sent ?reply - Message)
          (<- (and (web-agent ?agt)
                   (reply-pending ?agt ?msg-id ?sent)
                   (expected-reply ?agt ?sent ?reply))
              (normal-value (receive ?agt ?msg-id)
                            ?reply))))
(:action (send ?agt - Agent ?sent - Message)
         - (?sid - Message-id)
         :precondition (web-agent ?agt)
         :effect (reply-pending ?agt ?sid ?sent))

(:action (receive ?agt - Agent ?sid - Message-id)
         - (?received - Message)
     :vars (?sent - Message ?eff - Prop)
     :precondition (and (web-agent ?agt)
                        (reply-pending ?agt ?sid ?sent))
     :effect (when (message-exchange ?agt ?sent ?received ?eff)
                   ?eff)))
```

OPT can be used purely as an ontology mechanism. We are working on using it as an internal notation for representation systems such as RDF [**?**, **?**] and DAML [**?**], for which it provides type checking and other services. But the Lisp implementation of the OPT system, which is under construction, is the basis for support of various inferential systems, including a Prolog-style theorem prover, a planner [**?**], and a plan checker. The description of these systems and their APIs is beyond the scope of this manual, but will appear in later documentation. A formal semantics of OPT is in the works.

In OPT, all types are capitalized, a style popularized by Java, and tending to create clarity in my opinion. In previous versions, PDDL has ignored case.[1] OPT tools that offer a PDDL2.1-compatibility mode must also ignore it.

[1] I think this was a bad design decision, even though I am largely responsible for it. However, in the future I would urge that this design decision be reversed, and that symbols which differ only in the case of some of their constituent characters be treated as different.

2 Technicalities

Those not interested in technicalities may skip this section on first reading. As the example above shows:

1. The syntax of OPT is like Lisp's.
2. Where types are declared, one does it by writing " `-` *type-expression*" after the thing being declared.

The details are fairly intricate and even interesting, but perhaps not right now.

One other point: The enriched type system of OPT makes it possible to define many symbols simply by giving their types and a short definition, making it unnecessary to incorporate every symbol in a syntactic equation, as is the style in the various PDDL manuals. All symbols defined in this way are collected in section 13.3 in alphabetical order for easy reference. Furthermore, from this point on, every occurrence of such a symbol is flagged with a superscript •. so you won't be baffled about where it is defined. Remember that upper-case symbols are types, and hence are documented in section 13.1; other symbols are documented in section 13.3. Requirements flags, which begin with a colon ("`:`"), are documented in section 13.2; because they are all built-in, their occurrences are not flagged. The only built-in constants that are not flagged are those with short, nonalphabetic names, such as `+`, `=`, `<`, and such, for which the annotation would be too distracting.

2.1 Syntactic Notation

Our notation is the Extended BNF (EBNF) of [**?**] with the following conventions:

- Each rule is of the form <*syntactic element*> `::=` *expansion*.
- Angle brackets delimit names of syntactic elements.
- Alternative expansions are written in two ways: by supplying two or more rules with the same left-hand side:

  ```
  <term>  ::= (<term> <term>*)
  <term>  ::= (is <type> <term>)
  ```

  or, if the alternative expansions are short, by writing one rule with the alternative expansions separated by "`|`":

  ```
  <term>      ::= <constant> | <variable>
  ```

- Square brackets (`[` and `]`) surround optional material. When a square bracket has a superscripted requirement flag, such as:

  ```
  ...[:expansion <action spec>]:action−expansions
  ```

it means that the material is includable only if the domain being defined has declared a requirement for that flag.

- Similarly, the symbol `::=` may be superscripted with a requirement flag, indicating that the expansion is possible only if the domain has declared that flag.

  `<structure-def>` $::=^{:action-expansions}$ `<method-def>`

- An asterisk (`*`) means "zero or more of"; a plus (`+`) means "one or more of."

- Some syntactic elements are parameterized. E.g., `<list (symbol)>` might denote a list of symbols, where there is an EBNF definition for `<list` x`>` and a definition for `<symbol>`. The former might look like

  `<list (`x`)> ::= (`x`*)`

  so that a list of symbols is just `(<symbol>*)`.

- Ordinary parenthesis are an essential part of the syntax we are defining; the only place they have a meaning in the EBNF meta language is inside angle brackets, where they wrap parameters as just explained.

Comments in OPT begin with a semicolon ("`;`") and end with the next newline. Any such string behaves like a single space.

Variables in OPT are always bound by explicit quantifiers or other devices. There is a tradition, in AI applications of logic, of flagging each occurrence of a variable somehow, by a prefix character (`?x` or `:y`, perhaps), or by a case convention (so that `x` is a variable and `Xerxes` is not, or vice versa). The original motivation for this convention was that variables were never declared, just implicitly universally quantified with scope equal to the largest formula they occurred in; so the occurrences had to be flagged. I find it odd that in formalisms with explicit declarations people still like those little reminders, which they cheerfully do without in programming languages and other contexts. In OPT, variables may be prefixed with question marks, but the question marks are almost always optional. The only exception is where the question mark is specifically for the purpose of avoiding an explicit declaration, as in the case of implicit type variables, discussed in section 2.2

2.2 The Type System

The type system is ubiquitous in OPT. Intuitively, the type of an expression is a constraint on all the values the expression may ever have. To make this definition precise, we have to distinguish the type of an object from the type of an expression. An expression such as `(+ x 3)` will denote the number thirty-six in situations `x` where `x` denotes the number thirty-three. We write the names of the numbers out to emphasize that an object is drawn from an abstract universe of some kind, whereas an expression is drawn from a universe of lexical entities usually defined by recursive rules. The types of thirty-three and thirty-six are examples of *object types*; it happens that they are of type integer. *Expression type* is a useful concept only if it is used to reason about the types of the objects an expression will denote, i.e., its *values*, before it actually denotes them. In languages like Lisp and Kif, there is minimal use of expression type. Instead, every time the value of an expression

is available in a context where checking the appropriateness of that value is feasible, the run-time system can and usually does do the check, raising a run-time exception if the object fails the test. A classic example is evaluating `(+ x 3)` in an environment where the value of `x` is a string such as `"oops"`, and getting an error message such as "Non-numerical argument `"oops"` to +."[2]

It is harder to come up with examples in a declarative language such as Kif, because in the normal course of events expressions aren't evaluated, but just used to infer further expressions. The consequence of a type error is likely to be a silent failure to draw an appropriate inference. OPT is also a declarative language. Long experience has taught us that detecting type errors before they occur saves a lot of debugging, and that requires figuring out the types of expressions, not objects.

We declare the types of symbols, when they are first bound, by writing

expression `-` *type*

where the expression contains (and usually just is) the symbol. For instance, in a universally quantified statement we might say

```
(forall (x - Vehicle) (has-gas-tank x))
```

to indicate that `x` ranges over `Vehicle`s in this context. We follow the Java/Haskell convention of capitalizing the names of types. We do not follow the Java convention of usingCapitalLetters to indicateWordBoundaries; hyphens are treated as ordinary alphabetic characters, so we can write `has-gas-tank` instead of `hasGasTank`.

The *expression* may be more complex. For instance, one construct for declaring a function looks, in part, like this example:

```
(gcd i j - Integer•) - Integer•
```

This declares the type of `gcd`, to wit, a function that takes two Integer arguments and produces an Integer result. Here we introduce the following terminological convention: In programming, if f is a function whose result type is y, we say f "returns" (an object of) type y. In logic, functions don't get called, so they don't actually "return" from anywhere. We will say instead that it *produces* type y. (Well, the truth is that for practical purposes we depart from this purity in "evaluation contexts"; see section 3.)

Every expression must be *typeable*, meaning that it is possible to infer a consistent assignment of types to all its subexpressions, including itself. OPT does most of these inferences itself. It is often convenient to speak of "the type" of an expression, but this phrase is technically often wrong. An expression may have more than one type; typeability just means it is possible to find *a* consistent assignment of types of expressions.

In the example, if `has-gas-tank` is a predicate on `Physical-object`s, and every `Vehicle` is a `Physical-object`, then the type of `(has-gas-tank x)` is `Boolean`[3]. As this example

[2]There is a philosophical problem here about the sense in which a computer can ever have access to the actual abstract universe of integers. The answer, of course, is that it can't. It can, however, have access to a canonical representation of an abstract domain, including computable manipulations of that representation that provably correspond to or approximate actual mathematical functions on the abstract domain. Evaluating an expression E means finding the expression E_C in the canonical representation that denotes the same object E does. A run-time type error corresponds to the detection of an E whose E_C is not in the right subset of the canonical representation.

[3]Actually, it's not really `Boolean`, as we will discuss shortly.

shows, one type can be a *subtype* of another, as `Vehicle` is a subtype of `Physical-object`. Any term with type `Vehicle` also has type `Physical-object`.

Types are not exactly sets; type theory is a competitor to set theory as a way of thinking about logic. In set theory, one starts with very general axioms about any kind of collection, and gradually infers theorems about various sorts of collection. In type theory, all the symbols (and compound expressions) must be labeled (or labelable) with their types. The labels must obey certain rules. For instance, if `f` is labeled as a function from `String`$^\bullet$s to `Integer`$^\bullet$s, then `(f x)` must be labeled `Integer`$^\bullet$ and all occurrences of `x` must be labeled `String`$^\bullet$. A formula that cannot be given a type is ill-formed. In a sense the type of an expression identifies a set all its values fall in, but these sets are not first-class citizens in the logic. We can't quantify over types; we can't take intersection of types; we can't infer that a type exists without knowing which type it is (i.e., knowing the standard name for it).

In return for these constraints, it becomes easier to avoid paradoxes. The most famous paradox of set theory is Russell's Paradox. Starting from the plausible idea that for any property there exists a set containing just the objects that satisfy that property, we quickly get a contradiction. For if the plausible idea is true, then from the property $x \notin x$, we can derive the term $S = \{x | x \notin x\}$. But then $S \in S \iff S \notin S$. Much ingenuity has been used in making this paradox go away. In type theory, it never arises, because terms $x \notin x$ and $x \in x$ are not typeable. The type of $\in$ is "predicate with two arguments, one an `Individual`, the other a `Set`." To oversimplify x can be of type `Individual` or `Set`, but not both, so there is no way to infer a correct type for $x \in x$.

We use a more Lisp-like notation. $x \in x$ would be written `(elt x x)`. The type of $\in$, or `elt`, is `(Fun Boolean <- (Individual Set))`. `(Fun` *range* `<-` *domain*`)` is the type of functions from the given *domain* to the given *range*. The notation `(Individual Set)` is actually short for `(Arg`$^\bullet$ `_ - Individual _ - Set)`, which describes an *argument tuple* with two anonymous elements (as indicated by the use of the name "`_`"). Argument tuples are those evanescent entities that exist only when a function is called in Lisp. If you write `(+ x y z)`, it is useful to analyze this as applying the function `+` to the argument tuple $\langle$`x y z`$\rangle$, although this entity never exists as an actual data object (for instance, you can't bind a variable to it).

The basic rule for typing a function application is: If $\langle a_1 \ldots a_k \rangle$ is an argument tuple with type A, and f has type `(Fun` r `<-` A`)`, then `(`f a_1 `...` a_k`)` has type r.

Function types are complicated by two phenomena: overloading and polymorphism. *Overloading* is the use of the same function symbol for different functions. The standard example is `+`, which can operate on `Integer`$^\bullet$s or `Float`$^\bullet$s. Fortunately, overloading is rare, and doesn't require a general-purpose, user-level notation. In fact, the arithmetic functions are the only overloaded functions in OPT.

Polymorphism is the use of a function symbol to denote a family of functions parameterized by one or more *type variables*. A standard example is the `reverse` function. If l is a list of objects, then `(reverse` l`)` is a list of the same elements, in the opposite order. It doesn't matter what the types of the elements are, and they don't all have to be the same type. One way to proceed is to declare `reverse` to be of type `(Fun`$^\bullet$ `(Lst`$^\bullet$ `Obj`$^\bullet$`) <- (Lst`$^\bullet$ `Obj`$^\bullet$`))`, where `Obj`$^\bullet$ is a general type that includes all objects, and `(Lst`$^\bullet$ y`)` is the type of lists of objects of type y. (We use the term *type function* for an entity such as `Lst`$^\bullet$ that resembles a function syntactically, but produces a complex type from other types.) The problem is that if `x` is known to be of type `(Lst`$^\bullet$ `Integer`$^\bullet$`)`, we should be able to infer that `(reverse x)` is of the same type; but all we can infer is that `(reverse x)` is a list of `Obj`$^\bullet$.

What we want to do is introduce a *type variable* u and give `reverse` the type `(Fun• (Lst• u) <- (Lst• u))`. To avoid ambiguity about the scoping of u, we will actually declare reverse thus:

```
reverse - (Fun• 1 (Fun• (Lst• u) <- (Lst• u))
                  <- (u - (T•)))
```

In other words, `reverse` is a function that takes u as an argument and produces a function type `(Fun (Lst• u) <- (Lst• u))`. The type of u is `(T)`, meaning type. For example, `(reverse Integer•)` is a function of type `(Fun (Lst• Integer•) <- (Lst• Integer•))`. So one might expect to write `((reverse Integer•) x)`.

We can't actually do things quite that way, for a variety of reasons, including the fact that the first function application is obviously operating with "type-level" entities, while the second is operating with "domain-level" entities. The "1" in the first `Fun` signals this. Domain-level entities are at level 0, whereas type-level entities are at level 1. (The only level-2 entity in sight is the type `(T)`, which is what type theorists call a *kind*, meaning that it is the type of a (level-1) type.) We indicate level-1 application with the symbol `!&`, so the proper way to write the example above is

```
((!& reverse Integer•) x)
```

But this is too clumsy for the average case. We normally want to write simply

```
(reverse x)
```

We arrange for that with the following convention: If a level-1 function f is used where a level-0 function is expected, say in the term `(`$f\ \ a_1\ \ \ldots a_k$`)`, the type system will try to infer level-1 entities $y_1 \ldots y_n$ such that `((!&` $f\ \ y_1\ \ \ldots y_n$`)` $a_1\ \ \ldots a_k$`)` is typeable. If `x` is of type `(Lst• Integer•)`, then the type system infers that `(reverse x)` means `((!& reverse Integer•) x)`, and is of type `(Lst• Integer•)`.

In addition, it is distracting to have to mention function levels in the usual case. Hence OPT allows some abbreviations. We can mush the two levels together

```
reverse - (Fun (Lst• u) <- ((Lst• u) !& u - (T)))
```

The label `!&` is used here to separate 0-level variables from 1-level variables.

An alternative notation is to flag level-1 variables with question marks, as in

```
reverse - (Fun (Lst• ?u) <- (Lst• ?u))
```

which is exactly equivalent to the original type declaration. This notation has the advantage of conciseness. It has the disadvantage that the scope of the variable `?u` is not always clear. The rule is that the scope of a question-marked type variable is the outermost `Fun` such that the variable occurs in the type of one of its parameters.

Using these notational devices, we can redo the Russell's Paradox example in a more elegant way. Declare `elt` as follows:

```
elt - (Fun Boolean <- (e - ?u s - (Set ?u)))
```

Here `?u` is a level-1 variable, which can be any type. For any type t, there is a type of sets of t's, which we denote by `(Set` t`)`. Now `(elt x x)` is not typeable because there is no type u such that $u =$ `(Set` u`)`.

Polymorphism has not been necessary for traditional planning applications, because the focus has been on manipulating simple objects. But for planning interactions with agents on the world-wide web, most actions include creating, and extracting data from, data structures. Here polymorphism will be essential. However, it is hard to get interesting polymorphism without data structures such as `(Lst• Integer•)`, and simple PDDL-style applications will not be able to handle them. Hence there is a requirement flag `:data-structures` that must be declared by any domain that uses them.

In addition to `Lst•`, we have one other built-in complex type constructor: `(Tup•` *–piece-declarations–*`)` is a `tuple` with the given elements. Example: `(Tup• i j - Integer• s - String•)` is a tuple with three components, `i`, `j`, and `s`, of types `Integer•`, `Integer•`, and `String•`.

Given a `Tup•` x, you can refer to the field named s by writing `(!_`s x`)`. If the fields are anonymous (named _, in other words), you can refer to them positionally. The second element of `u`, where `u - (Tup• _ _ - Integer•)` can be accessed by writing `(!_<2> u)`.

New type functions may be defined in terms of existing ones. See section 4.

To refer to the object of a given type with given arguments, write `(make` *type –arguments–* `)`. For example, `(make (Tup• i j - Integer•) 5 6)` denotes a two-integer tuple x such that `(!_i` x`)` is 5 and `(!_j` x`)` is 6. This notation is not very useful without the ability to define named types, like this:

In domain definition:

```
(:type Int2 (Tup• i j - Integer•))
```

In some domain axiom:

```
(list (make Int2 5 6) (make Int2 50 60))
```

The functions `(list•` a_1 `...`a_n`)` and `(tuple•` a_1 `...`a_n`)` are a simpler way to make lists and tuples. The OPT system will try to infer the most useful type it can (either a `Lst•` or `Tup•` type) for an occurrence of one of these expressions.

If $y_1, \ldots y_l$ are types, then `(Alt•` y_1 `...`y_l`)` is the "union" of those types. An object is of this type if and only if it is of type y_i for some i. For an expression to be of this type is for each of its possible values to be of type y_i for some i. (A synonymous expression is `(Either•` y_1 `...` y_l`)`.)

We now describe the type-declaration notation using EBNF:

```
<typed list (x)> ::= (<typed row (x)>)
    <typed row (x)>          ::= <x>*
    <typed row (x)>          ::= <x>+- <type> <typed row (x)>
    <type>                   ::= <variable>
                             ::= (<type function> <type>+)
                             ::= (Fun• [<level>]
                                       <val spec> <- <arg spec>)
    <type>                   ::= (<struct builder> <type row>)
    <type>                   ::= (<struct builder> <param row>)
```

```
<type function>          ::= <name>
<level>                  ::= 0 | 1
<struct builder>         ::= Tup•

<arg spec>               ::= <name arg spec> | <type arg spec>
<name arg spec>          ::= <name> | ([Arg] <named params spec>)
<type arg spec>          ::= <type> | ([Arg] <anon params spec>)
<named params spec>      ::= <param row> [!& <typed row (name)>]
<param row>              ::= <typed row (name)>
                                 [&optional <typed row (param)>]
                                 [<rest-or-key param row>]
<anon params spec>       ::= <type row> [!& <typed row (name)>]
<type row>               ::= <type>* [&optional <type>+]
                                     [<rest type row>]

<rest-or-key param row> ::= <rest param row> | <key param row>
<rest param row>         ::= &rest <variable> [- <type>]
<rest param row>         ::= &rest <variable> :- <type>
<rest param row>         ::= &rest (<variable>+) <variable> - <type>
<key param row>          ::= &key <typed row (keyparam)>
<keyparam>               ::= <dflt-param>
                             | ((<keyword> <variable>) <term>)

<rest type row>          ::= &rest <type>
<rest type row>          ::= &rest (= <type>)
<rest type row>          ::= &key (keytype)+
<keytype>                ::= <keyword> <type>

<dflt-param>             ::= <variable> | (<variable> <term>)
<val type spec>          ::= <type>
                             ([Val] <type>*)
                             ([Val] <typed list(name)>)
<keyword>                ::= :<name>
<variable>               ::= [?]<name>
<name>                   ::= <basicname>
```

An EBNF definition of <basicname> would be a bit clumsy; it's easier just to say that a name is a sequence of one or more alphanumeric characters starting with an alphabetic. We count characters -, _, +, -, *, /, <, and > as letters. (Note that colon is *not* allowed in a basic name. We will find a use for it in section 11.)

A typed list is used to declare the types of a list of entities; the types are preceded by a hyphen ("-"), and every other element of the list is declared to be of the first type that follows it; if there are no types that follow it, OPT is to infer the type. An example of a <typed var list> is

```
x y - Integer• ?l - (Lst• Integer•)
```

As explained above, the question marks in front of variables are optional.

`Fun•` and `Tup•` are not quite type functions, because of their idiosyncratic syntax. The argument position of a `Fun•` may be written `(Arg• ...)`, or the `Arg•` may be implicit. Similarly, the value position of a `Fun•` may be written with or without the explicit `Val•`. The syntax of `Arg•` and `Tup•` are identical; `Val` is somewhat simpler. They are all based on the syntax of Common Lisp [**?**], which exploits fully parenthesized syntax to allow for very flexible argument-passing conventions.

The easiest way to understand the `Arg•` syntax is to focus on the syntactic element `<named params spec>`; `<anon params spec>` is essentially the named version with the names removed. A `<named params spec>` specifies *required arguments* and *optional arguments* (if any), followed by either a `&rest` argument or a list of `&key` arguments. Every required argument is specified by a single named parameter. Optional arguments may have default values if omitted, in which case the parameter is written `(<name> <default value>)`. (The syntactic element `<term>` will be explained in section 2.3.) If the default is not specified, it is the standard "zero" value for the parameter's type (e.g., 0 for `Integer•`, `""` for `String•`, and so forth).

The `&rest` parameter row consists of a name followed by an optional type declaration. `&rest m - Integer•` indicates that the presence of zero or more `Integer•`s. In a `Tup•`, this notation means that the `m` slot (accessed by `!_m`) is of type `(Lst• Integer•)`. A double hyphen is a variant in which we give the type of the slot rather than the type of each element of the slot. So we could write `&rest m -- (Lst• Integer•)` with the same meaning.

Instead of a `&rest` parameter, one can write `&key` followed by a *keyword parameters*. In the simplest case parameters are declared as in this example:

```
ff - (Fun• String• <- (&key x y - Integer•))
```

after which we can use `ff` by writing, e.g., `(ff :x 3 :y 4)`, or `(ff :y 40 :x 30)`, or even `(ff :y 50)`. Keyword arguments may occur in any order, and some may be omitted. One can add a default value by writing a `<dflt-param>` instead of a `<variable>`. One can make the keyword differ from the slot name by writing `((<keyword> <variable>) <term>)`. If we use these conventions to define `ff`, we might end up with:

```
ff - (Fun• String• <- (&key (x -3) ((:y-arg y) 0) - Integer•))
```

So that `(ff :y-arg 4)` means the same as `(ff :y 4 :x -3)` in the previous version. If we declare a tuple thus:

```
t1 - (Tup• &key (x -3) ((:y-arg y) 0) - Integer•)
```

then if `t1` is bound to the value of `(tuple :y-arg 10)`, it will have two slots `(!_x t1)`, with value -3, and `(!_y t1)`, with value 10.

There is one semantic difference between `Tup•` on the one hand, and `Arg•` and `Val` on the other. The latter are "row types," meaning that an `Arg•` or a `Val` with one element is indistinguishable from that element, whereas a tuple or record with one element is not the same as that element alone. A function whose range type is `(Val n - Integer•)` is the same as one with range type `Integer•`, except that the former allows us to use the name `n` to stand for the value in appropriate contexts.

Type rows are essentially the same as parameter rows, but with all the names taken out. However, we have to have conventions for handling nameless `&rest` and `&key` arguments. We interpret

`&rest` t as `&rest _ -` t; to say `&rest _ --` t in parameterless style, write `&rest (=` t`)`. For `&key` parameters, you simply write the keywords and their types. So the type of the second version of our function `ff` might be written

```
(Fun• String• <- (&key (:x Integer•) (:y-arg Integer•)))
```

For a table of all the built-in types of OPT, see section 13.1.

2.3 The Logical Language

The basic syntax of the language is very simple. The primary syntactic element we define in this section is `<term>`. Most terms are simple applications, of the form `(`f $-args-$`)`. The term grammar is very general, and can generate lots of bogus terms, such as `(+ (\\ (x) (1 x)))`. These terms are weeded out by the typeability requirement of section 2.2. As new syntactic constructs are presented, I will sketch the typeability rules associated with them.

Let me emphasize again that the language I am describing is a *logical* language, not a programming language. To avoid misleading distractions I will eschew phrases like "the value of this term is ...," and instead say things like "the denotation of this term is ...." To repeat what I said above, nothing is "called," and nothing "returns a value" in a logical language.

```
<term>      ::= <literal> | <name> | <variable>
<term>      ::= (<term> <term>+)
<term>      ::= (!& <term> <term>+)
<term>      ::= (make <type> <term>+)
<term>      ::= (!_<name> <term>)
<term>      ::= (let-var <typed list (let-binding)>
                     <term>
                     [ :where <typed row (let-binding)>)]
<term>      ::= (\\ <function-def-innards>)
<term>      ::= (is <type> <term>)
<term>      ::= (be <type> <term>)
<term>      ::= (if <term> <term> <term>)
<term>      ::= (-> <term> <term>)
<term>      ::= (<- <term> <term>)
<term>      ::= (when <term> <term>)
<term>      ::= (and <term>*
<term>      ::= (or <term>*)
<term>      ::= (declare <typed list (variable)>
                     <term>)
<term>      ::= (<quantifier> (<typed row (variable)>
                                [<rest param row>])
                      <term>)
<let-binding>          ::= (<name> <term>)
<function-def-innards>::= [- <val  type spec>] <name arg spec>
                                  <term>
<literal>              ::= <boolean> | <number> | <string> | '<name>
```

```
<boolean>              ::= true | false
<quantifier>           ::= forall | exists | exists!  | freevars
```

Let's explain each line in turn. The first line is self-explanatory. Literals in OPT include the usual `Boolean`•s, numbers (`Integer`• and `Float`•) and `String`•s, plus `Symbol`•s, which are just quoted names. (That is, there is a literal format for every primitive type in the hierarchy of section 3.)

The second line,

```
<term> ::= (<term> <term>+)
```

is for function applications. A function application (*f –arg tuple–*) has type R if f has type (`Fun`• R `<-` A) and the *arg tuple* has type A. Note that the object in function position can be any term, not just a name, so long as it obeys this type rule. For instance, if `m` is of type

```
(Lst• (Fun• String• <- Integer•))
```

then `((car m) 3)` has type `Integer`•.

```
<term> ::= (!& <term> <term>+)
```

refers to level-1 function application. The first term must denote a level-1 function. The arguments (if any) usually denote types, but other sorts of argument are allowed, provided they are all defined when the type of the term is determined. The typing rule for level-1 function application is essentially the same as the rule for level-0 application.

```
<term> ::= (!_<name> <term>)
```

is for "slot access." For this to be typeable, `<term>` must be of a structured type (a `Tup`•) with a field named `<name>`. The term `(!_<name> <term>)` then refers to the contents of field `<name>`. The type of the term is then the type of that field. If the structured type has anonymous fields, they can be referred to as if they were named `<1>`, `<2>`, etc.

```
<term> ::= (let-var (<typed list (let-binding)>)
                <term>
                [ :where (<typed list (let-binding)>]
<let-binding>::=(<name> <term>)
```

The denotation of (`let-var` (*–var-bindings–*) e) is the denotation of e in an environment where each `<name>` is bound to the denotation of its corresponding `<term>`. If it's clearer to put bindings at the end of the expression, you may do so, if they're prefixed by keyword `:where`; the bindings at the front still have to be there, but can be `()`. Example:

```
(let-var ((s (set-of-all (\\ (x - Planet) (inhabited x)))))
         (and (elt earth s) (elt mars s)))
```

The type rule for (`let-var ((`v_1 a_1`) ...` (`(`v_k a_k`))` e) is that, if e has type t in an environment where each v_i has type t_i, where t_i is the type of a_i, then the entire `let`-expression has type t.

```
<term>                ::= (\\ <function-def-innards>)
<function-def-innards>
                      ::= [- <val type spec>] <typed list(name)> <term>
```

The \\ is our substitute for λ, which is not available on most keyboards.

`(\\ [- <type spec>] <name arg spec>` e`)`

denotes an anonymous function whose arguments are given by the `<name arg spec>`, and whose value, e, is of type `<type spec>`. The value type is optional because OPT can usually figure it out.

The typing rule for `(\\` a e`)` is that if e has type t in an environment where all the variables of a have their declared types, then the \\ expression has type `(Fun•` t `<-` a`)`.

`<term>` ::= `(is <type> <term>)`

`(is` t e`)` is true if and only if e is of the given type. This expression itself is of type `Boolean`.

`<term>` ::= `(be <type> <term>)`

is a declaration that the given `<term>` is of the given type. You can abbreviate `(!_`s `(be` t e`))` as `(!_(`t s`)` e`)`. Note that `be` is merely a hint to the reader or the type checker that `<term>` has the given type; it doesn't override anything, or give rise to a "run-time check," which is meaningless in a language with no run time.

`<term>` ::= `(if <term> <term> <term>)`

has the usual meaning: `(if` v a b`)` has the same denotation as a if the denotation of v is `true`, and the same denotation as b if the denotation of v is `false`•. `(if` v a b`)` has type u if both a and b have type u. `(if` v a `true)` may be abbreviated `(if` v a`)`. The variants `(->` v a`)` and `(<-` a v`)` are explained in section 8. The variant `(when` v e`)` is explained in section 5.2.

Some terms used in the *antecedent* (v-part) of an `if` have *type implications*. For instance, if we write

```
(if (null l) 0 (- (car l)))
```

in a context where `l` is declared to be of type `(Lst• Number•)`, the fact that `l` is not empty, as it is known to be in the false branch of the `if`, implies that `l` is of type `(Tup• Number• &rest Number•)`, and hence that `(car l)` is of type `Number•`. Similarly, in the true branch, `l` is known to be of type `(Tup•)`, the empty tuple. In general, we say that a term has *type implications* if knowing its type tells us something about the types of the variables that occur in it. `(if` w B_t B_f`)` then really means

`(if` w
`(declare` I_t B_t`)`
`(declare` I_f $B - f$`))`

where I_t are the type implications of knowing that w is of type `(Con true•)` and I_f are the type implications of knowing that w is of type `(Con false•)`. (See below for the explanation of `declare`.) Our example then translates to

```
(if (null l)
    (declare (l - (Tup•))
       0)
    (declare (l -  (Tup• Number• &rest Number•))
       (- (car l))))
```

The connectives `and` and `or` are defined in terms of `if` in the usual way: `(and)` means `true`; `(and` c_1 `...`c_k`)` means `(if` c_1 `(and ...`c_k`) false•)`. `(or)` means `false•`; `(or` c_1 `...`c_k`)` means `(if` c_1 `true (or ...`c_k`))`. Because of these equivalences, `and` and `or` are not commutative, due to the possible type implications of c_i on later c_j. For example,

```
(or (and (is Camel x) (two-hump x))
    (and (is Rhino x) (one-horn x)))
```

where `two-hump` and `one-horn` apply only to Camels and Rhinos, respectively, would not be typeable if the orders of the arguments to the `and`s were switched. The construct `(is` y x`)` has the obvious type implication, namely, that x is of type y.

```
<term> ::= (declare <typed list (variable)>
              <term>)
```

means the same thing as `<term>`, except that hints are provided to the type checker that the variables in `<typed list (variable)>` should be treated as though they were of the declared types. *Important:* These hints cannot be used to change the types of the variables arbitrarily. If the type checker cannot prove that the variables have the indicated type, it should signal an error. [[Actually, it's a bit more complicated than that. The type checker is allowed to constrain type variables in the course of the "proof," so it's more a license to discard some of the polymorphism. Example: if `l` is known to be of type "list of something," `(declare (l - (Lst• Eland)) ...)` restricts it to be a list of `Eland`s.]]

```
<term> ::= (<quantifier> (<typed row (variable)>
                          [<rest param row>])
              <term>)
```

There are four quantifiers, `forall`, `exists`, `exists!`, and `freevars`. `(forall` v $P[v]$`)` is true iff $P[a]$ is true for all properly typed values of the variables in v. `(exists` v $P[v]$`)` is true iff $P[a]$ is true for some value of the variables v. `exists!` is similar, but is true iff there is exactly one assignment to the variables v that makes P true. `(freevars` v $P[v]$`)` is a quantifier only "syntactically." It just means, Let variables v be free in $P[v]$, but it provides a way of declaring them. In most cases, `freevars` is synonymous with `forall`, as in Prolog.

Quantified expressions are of type `Prop`, for "proposition." The difference between a `Prop` and a `Boolean` is that the former can change value from situation to situation. For instance,

`(location Titanic Southampton)` is true in some situations, false, alas, in others. A `Situation•` is a state of affairs, a specification of the truth values of all ground atomic formulas. A `Fluent•` is a function from `Situation•`s to values; more precisely:

```
Fluent• = (Fun• u <- (Situation• !& u))
```

We can then define a `Prop` as a `(Fluent• Boolean)`.

In most contexts, an object of type y is acceptable whenever an object of type `(Fluent•` y`)` is required, and vice versa. Appropriate coercion rules are applied, as described in section 3.

For future reference in this manual, we introduce various subcategories of `term`, usually based on their types, but occasionally requiring other constraints. Here are some of those subcategories:

- An `<action term>` is a term with type `Action`
- A `<proposition>` is a term with type `Prop`
- A `<goal proposition>` is a `<proposition>` used to specify preconditions of actions, and other goals. See section 5.
- An `<effect proposition>` is a `<proposition>` used to specify effects of actions. See section 5.2.

Let us close this section with a further exhortation not to assume that the logic subset of OPT is a programming language. When we write `(if (> m n) m n)`, we do not mean, "Test whether `m` is greater than `n`, and if so…" There may not be any method for testing the inequality, and if there is the agent may not always know it. When we write

```
(exists (p - Planet)
    (and (orbits p sun) (not (= p earth)) (inhabited p)))
```

we do not mean "Loop through the planets that orbit the sun." Instead, the formula simply denotes `true` in universes where there is an inhabited planet in the solar system besides earth, and `false•` in all other universes. There may no easy way to tell which category our universe falls in.

OPT does include an imperative sublanguage, which will be described in section 5. However, its semantics are completely compatible with the semantics of the rest of the language. That is, a term in the imperative sublanguage *denotes* something for an agent to do, sometimes a complex structure of actions. Considered as a term, that's all it does. Of course, somewhere there has to be a plan executor that can take that term and perform the actions it describes.

3 Built-in Types: Primitives, Fluents, Expressions

The primitive types, with subtype relationships shown, are as follows

```
Obj
   Number
      Integer
      Float
   Char
```

```
String
Symbol
```

(These are all built in, and all described in section 13.3.)

Arithmetic and comparison functions are defined on type `Number•`. However, the type-inference system treats these functions as *overloaded*, so that, for instance, if `x` is of type `Float•`, the expression `(+ x 1)` is treated as meaning `(+ x 1.0)`, and as being of type `Float•`. That is, an arithmetic expression is of type `Integer•` only if all its arguments are; if it has any argument of type `Float•`, it is of type `Float•`; otherwise, it is of type `Number•`.

To make this work, OPT is provided with a modest *coercion* system that fixes type errors silently when it can using various ad-hoc rules. One of them is that if an `Integer•` is found where a `Float•` is wanted, the `Integer•` is treated as the `Float•` with the "same" value (in an inevitably implementation-defined sense).

In addition, in domains declaring the `:data-structures` requirement, we have the type `Sexp•`, or "S-expression," defined as

```
(Alt• Symbol• String• Char• Number• (Lst• Sexp•)))
```

The type constructor `(Fluent•` y`)` is built in to OPT, and defined as `(Fun•` y `<- Situation•)`, that is, a quantity of type y whose value changes from situation to situation. The type `Prop•`, for "proposition," is the type produced by all predicates, and is defined as `(Fluent• Boolean•)`.

If a domain declares requirements `:fluents`, then there are many more possible things one can do with fluents, which will be described in the appropriate sections below.

Arithmetic and comparison functions may, through the magic of coercion, be applied to fluents. Actually, the rule is more general. In any context where an expression of type y is wanted, if an expression e of type `(Fluent•` y`)` is found, it is treated as `(fl-v•` e`)`, which means "the value of e in the current situation." Similarly, if e is of type y, and `(Fluent•` y`)`, is required, e is treated as `(fl-^•` e`)`, which means "the fluent that has value e in all situations."

If a domain declares requirement `:expression-evaluation`, then it supports *computational equality substitution*, in which an expression such as `(+ ?x 5)` is replaced by 12 when `?x` has value 7. The built-in predicate `eval•` provides the canonical context for this kind of substitution: `(eval•` E V`)` is true if the computational value of expression E is V. E is an expression using the functions `+`, `-`, `*`, `/`, `min•`, `max•`, and `fl-v•`. The first argument to `e•val` is said to be an *evaluation context*. If E contains any unbound variables, then it has no computational value, and no conclusion can be drawn regarding the truth or falsity of `(eval•` E V`)`.

There are several other evaluation contexts in OPT, including both argument positions to the inequalities $>$, $=<$, etc. The proposition `(bounded-int•` I L H`)` is true if I is an integer in the interval $[L, H]$. L and H are evaluation contexts, as is I if it has no variables.

Note that `eval•` is a special case of equality, with a nudge to the deductive system that its first argument should be evaluated. Another version of equality is `(equation•` L R`)`, for which the "nudge" is as follows: If there is a single unbound deductive variable `?`v with one or more occurrences in L and R, then for every value of x of that variable that makes the values of L and R the same, the conclude `(=` $\{v = x\}(L)$ $\{v = x\}(R)$`)`, where $\{v = x\}(e)$ means e with all occurrences of `?`v replaced with x. E.g., if `?y` has been bound to 6, and `?x` is unbound, then `(equation (+ ?x 2) (- ?y 3))` is true, provided `?x` = 1. Exactly what equations are

solvable is implementation-dependent, but every implementation should at least handle the case where there is a single occurrence of an unbound variable, buried at most inside an expression of the form `(+/- ...)`.

If the OPT syntax checker (which is beyond the scope of this manual) cannot figure out a valid type for an expression, it will issue an error message. Its complaints are usually correct, but sometimes it has overlooked something. If you need to give it a hint about the type of an expression e, the usual way is to declare some of the variables occurring in e; an alternative is to replace e with `(be` y e`)` to declare that e should have type y. Then instead of trying to infer a type for e, it can work on the easier task of verifying the declared type.

4 Domains

An Opt file consists of a series of domain definitions, addendum definitions, and problem definitions, defined here, in section 9, and section 10.

The EBNF for defining a domain structure is:

```
<domain-def>     ::= (define (domain <name>)
                         [<extension-def>]
                         [<export-def>]
                         [<require-def>]
                         [<types-def>]
                         <complex-type-def>*
                         <type-function-def>*
                         [<objects-def>]
                         [<parameters-def>]
                         [<predicates-def>]
                         [<functions-def>]
                         [<facts-def>]
                         <structure-def>*)
<extension-def>  ::= (:extends <parent>+)
<parent>         ::= <name>
<parent>         ::= See section 11.
<export-def>     ::= See Section 11.
<require-def>    ::= (:requirements <requirement>+)
<requirement>    ::= See Section 13.2
<types-def>      ::= (:types <typed list(name)>)
<complex-type-def>
                 ::== (:type <name> <type-defn>)
<type-function-def>
                 ::== (:type-fun <name> <typed list(name)>
                 <type-defn>)
<objects-def>    ::= (:objects <object declarations>)
<parameters-def>::= (:parameters
                          <typed list(parameter-declaration)>)
<predicates-def>::= (:predicates <atomic formula skeleton>+)
```

```
<functions-def> ::= (:functions
                          <typed list (function skeleton)>)
<atomic formula skeleton>
                ::= (<name> <named params spec>)
<function skeleton>
                ::= (<name> [- <type>] <named params spec>)
<facts-def>     ::= (:facts <proposition>+)
<facts-def>     ::= (:axioms <proposition>+)
<structure-def> ::= <action-def>
<structure-def> ::=:domain−axioms <axiom-def>
<structure-def> ::=:action−expansions <method-def>
<structure-def> ::=:processes <process-def>
<structure-def> ::=:durative−actions <durative-def>
```

Although we have indicated the arguments in a particular order, they may come in any order, except for the `(domain ...)` itself.

If the `:extends` argument is present, then this domain inherits requirements, types, objects, facts, axioms, and actions from the named domains, which are called the *ancestors* of this domain. Inheritance is transitive.

The `:requirements` field is intended to formalize the fact that not all systems can handle all domains statable in the OPT notation. The listed requirements indicate the range of constructs that will be used in the domain being described. If a construct is used that requires a feature that the domain didn't declare, an OPT processor should complain. In general, a domain is taken to declare every requirement that any ancestor declares. A description of all built-in requirements is found in Section 13.2.

The `<types-def>` defines simple named types. It uses a `<typed list (name)>`, but with a different interpretation than normal.

```
(:types Elephant Ant - Animal Positive-integer - Integer•)
```

declares three new types, `Elephant`, `Ant`, and `Positive-integer`, and declares them to be subtypes of `Animal`, `Animal`, and `Integer`•, respectively. (It does *not* declare any object to be any of those types.)

A `:type` clause allows us to name an arbitrary type. Example:

```
(:type Message (Lst• (Tup• String• String•)))
```

This definition declares `Message` to be synonymous with `(Lst• (Tup• String• String•))`. By contrast, a `:types` clause declares a type to be an anonymous subtype of a given type.

A `:type-fun` clause defines a type function. Example:

```
(:type-fun (Mess a)
           (Lst• (Tup• a a)))
```

The `:parameters` declaration is described in Section 3.

The `:objects` field has the same syntax as the `:types` field, but the semantics is different. Now the names are taken as new constants in this domain, whose types are given as described above. E.g., the declaration

```
(:objects sahara - Theater
          division1 division2 - Division)
```

indicates that in this domain there are three distinguished constants, `sahara` denoting a `Theater` and two symbols denoting `Division`s.

The `:parameters` field defines parameters that vary from domain to domain. [[These were originally called `:domain-variables`, but I believe "parameter" is a better term.]] The syntax is

```
<parameters-def> ::= (:parameters
                            <typed list(parameter-declaration)>)
<parameter-declaration>
                  ::= <name> | (<name> <constant>)
```

Each such parameter is treated as its value in evaluation contexts.

E.g.:

```
(define (domain cat-in-the-hat)
    (:types Thing - Integer)
    (:parameters (numthings 2) - Integer•)
    ...
    (:axiom
         :vars (?i - Integer•)
         :context (bounded-int• ?i 1 numthings)
         :implies (is Thing ?i)))
```

The axiom (see section 8) states that "a `Thing` is an Integer between 1 and the value of `numthings` (in this domain, 2)."

The `:predicates` field consists of a list of declarations of predicates, once again using the typed-list syntax to declare the arguments of each one. Example:

```
(:predicates (loves ?x ?y - Person)
             (gender-of x - Person g - Gender))
```

The question marks before variables are optional in this context.

The `:functions` field is a list of declarations of functions. A function f is a constructor of terms, (*f* `--args--`). It is not a function in the sense used in functional programming. To describe a function, we have to describe the types of its arguments, but also the type it produces. There are two places to put the type. One is after the function symbol, as in:

```
(:functions (list-sum - Integer• (l - (Lst• Integer•)))
            (main-element - ?u (l - (Lst• ?u))))
```

indicating that `list-sum` produces type `Integer•`, and that `main-element` produces type `?u` when given a list of objects of type `?u`. The last question mark is significant; as explained in section 2.2, its presence declares `main-element` to be a level-1 function whose argument is a type `u`, and that produces a level-0 function that produces type `u`. Normally you may forget the details, unless your ontology fails to type-check.

If several of the functions in the list produce the same type, we can move the type up a layer of parens:

```
(:functions (plus &rest l - Number•)
            (times &rest l - Number•)
            - Number•
            (book-title b - Book)
            (book-author b - Book)
            - String•)
```

indicating that `plus` and `times` produce `Number•`, and `book-title` and `book-author` produce `String•`.[4]

A predicate is just the special case of a function that produces type `Prop`. There is no difference between the `:predicates` declaration above and

```
(:functions (loves ?x ?y - Person)
            (gender-of x - Person g - Gender)
            - Prop)
```

The `:facts` field consists of a list of propositions that are taken to be true in this domain. (`:axioms` is treated as a synonym for `:facts`.) It goes without saying that the symbols used in the facts must be declared either here or in an ancestor domain. (Built-in predicates such as "=" behave as if they were inherited from an ancestor domain, although whether they actually are implemented this way depends on the implementation.) See section 8.

A variable like this is scoped over the entire domain, and is inherited by domains that extend this one. If the variable is redeclared in an extending theory, it shadows the original binding.

The remaining fields define actions and rules in the domain, and will be described in their own sections.

5 Actions

5.1 Syntax of Actions

The EBNF for an action definition is:

```
<action-def>      ::= (:action (<name> <name arg spec>)
                               [ - <val type spec>]
                          <action-def body>)
<action-def body>
                                        :existential-preconditions
                                        :conditional-effects
            ::= [<vars-spec>]
                [:precondition <goal proposition>]
                [:effect <effect proposition>]
                [:expansion <action spec>]   :action-expansions
                [:only-in-expansions <boolean>]   :action-expansions
<vars-spec> ::= :vars <typed list (variable)>
<expansion spec> ::= :expansion <action spec>
```

[4] In PDDL2.1, functions all produce the type `(Fluent Number)`. This design decision will surely be overturned as PDDL evolves, but implementations with a "PDDL2.1 compatibility mode" should provide `(Fluent Number)` as a default return type.

```
<expansion spec> ::= :methods
```

The preferred form for declaring the arguments and values of actions in OPT is to use parentheses and hyphens in the same way that other functions are declared. However, it is also permissible to use the old-fashioned PDDL-style notation:

```
<action-def>     ::= (:action <action function>
                          :parameters <name arg spec>
                          [:value <action value spec>]
                          <action-def body>)
```

The `:vars` list are locally bound variables whose semantics are explained below.

The `:precondition` is an optional goal proposition that must be satisfied for the action to be feasible. If no preconditions are specified, then the action is always feasible. The `:effect` field lists the changes which the action imposes on the current state of the world.

A `<goal proposition>` is a `<proposition>` in which certain constructs are prohibited unless the domain declares certain requirements:

Construct	*Requirement*
`or`, `not`, `imply`	`:disjunctive-preconditions`
`exists`*	`:existential-preconditions`
`forall`*	`:universal-preconditions`

The asterisks next to `exists` and `forall` are to remind us that an `exists` in a negative context should be treated as a `forall`, and vice versa. The *polarity,* positive or negative, of an occurrence of a subexpression is defined thus: Moving from the subexpression occurrence outward, count the number of `not`s encountered, plus the number of `if`s that the occurrence occurs in the antecedent of. If the number is even, the subexpression is in a *positive* context; if odd, it's in a *negative* context. So, if a domain declares `:disjunctive-preconditions` but not `:universal-preconditions`, then `(not (exists (u v) ...))` is forbidden as a goal.

If the domain declares requirement `:action-expansions`, then it is legitimate to include an `:expansion` field for an action, which specifies all the ways the action may be carried out in terms of (presumably simpler) actions. See Section 7.

Free variables are not allowed. All variables in an action definition (i.e.,, in its precondition, expansion, or effects) must be included in the `:parameters`, `:vars`, or `:value` list, or explicitly introduced with a quantifier.

Variables appearing in the `:vars` field behave as if bound existentially in preconditions and universally in effects, except that it is an error if more than one instance satisfies the existential precondition. So, for example, in the following definition

```
(:action spray-paint
    :parameters (?c - color)
    :vars (?x - location)
```

```
:precondition (at robot ?x)
:effect (forall (?y - physob)
           (when (at ?y ?x)
                 (color ?y ?c))))
```

the robot must be in at most one place to avoid an error.

The optional argument `:only-in-expansions` is described in Section 7.

5.2 Effects and Values of Actions

The *effect* of an action is the way the world changes as a result of executing the action. An `<effect proposition>` has the syntax of a `<proposition>`, but describes how the world is to *change,* not how it *is*, which is the usual meaning of a proposition. Hence we do not talk about the "truth value" of an effect proposition; it does not "become true," but *is imposed.* There are certain pseudo-connectives allowed in this context that may not occur elsewhere, especially `(when•` p e`)`, which means "If p was true just before the action was executed, then e (another effect proposition) is imposed as a result of its being executed." In section 13.3, these pseudo-connectives, called *effect predicates,* are specially flagged. For the exact semantics of `when•` and its relatives, see section 6.

Another pseudo-connective is `not`; `(not` p`)` is imposed by deleting p.[5]

In addition to an effect, the action may have a *value,* which reflects information acquired or created as a result of carrying out the action. For example, turning on a light in a dark room changes the average illumination level; it also tells you if there's a hippopotamus in the room, which we can model by supposing the action returns an `Integer•` value:

```
(:action (turn-on-light rm - Room)
         - (h - Integer•)
         :effect (and (bright rm)
                      (not (dark rm))
                      (know-val-is (number-of-hippos rm)
                                   h)))
```

The effect of `(turn-on-light` r`)` is that `(bright` r`)` becomes true and `(dark` r`)` becomes false. These effects then persist until the next action. The predicate `(know-val-is` t c`)`, is supposed to be true if the agent knows that the value of term t is c, where c is a "computational" object (in this case, an `Integer•`).

The `:value` field of an action specifies the type of the value returned when the action is performed. The value can be used as an input to a subsequent action by using a *link,* described in section 7. Even if it is not transmitted to another step, it can be referred to in the `:effect` field by declaring its fields and using them as if they were variables. In the example, `h` means "the `!_h` field of the value of this execution of `(turn-on-light` r`)`."

Here's another example (using the PDDL syntax):

[5] It is surprisingly tricky to say what "deletion" is, but for domains not declaring the `:open-world` requirement, the idea is that every situation has a finite description as a list of atomic formulas; cf. section 10. The situation is obtained by assigning value true to formulas on the list, value false to all other atomic formulas, and then taking the deductive closure under all the axioms of the domain. Adding and deleting atomic formulas should be considered an operation on the finite description of the current situation; the resulting situation is then obtained by repeating the deductive-closure operation on the new finite description. For domains that do declare the `:open-world` requirement, the semantics of effects are murky and implementation-dependent.

```
(:action take-lunar-measurement
     :parameters ()
     :precondition (...)
     :value (?az ?elv - Float•)
     :effect (and (know-val (azimuth moon) ?az)
                  (know-val (elevation moon) ?elv)))
```

As usual, the question marks are optional.

An action definition defines a function; it takes arguments as specified in the `<val type spec>`, and produces an `Action`•. Actually, we can be more specific than that. If an action definition specifies no `:expansion`, then the action defined has type `(Fun (Skip`• r`) <-` a`)`, where a and r are the argument type and result type specified in the definition. An object of type `(Skip`• r`)` is an action that takes one "instant" of time (in a sense explained in section 6) and returns a value of type r. I will call these *skips*. If an action definition does specify an `:expansion`, then its name is declared to be of type `(Fun (Hop`• r`) <-` a`)`, as explained in more detail in sections 6.2 and 7. The type `Skip-action`• is defined to be `(Skip`• `(Val`• `&rest Obj`•`))`, that is, a skip that returns zero or more values.

5.3 Effects Involving Fluents

If a domain declares requirement `:fluents`, then it supports some special predicates, particularly some new effect predicates.

The effect predicate `(assign` F E`)` indicates that the value of fluent F changes to E. E is an evaluation context, and its value is computed with respect to the situation obtaining before the action (cf. `when`). The effect predicates `(increase` F E`)` and `(decrease` F E`)` are synonymous with `(assign` F `(+` F E`))` and `(assign` F `(-` F E`))`, respectively. Here is an example of how they are used:

```
(:action (pour ?source ?dest - container)
     :vars (?sfl ?dfl - (fluent number) ?dcap - number)
     :precondition (and (contents ?source ?sfl)
                        (contents ?dest ?dfl)
                        (capacity ?dest ?dcap)
                        (fluent-test (=< (+ ?sfl ?dfl) ?dcap)))
     :effect (and (assign ?sfl 0)
                  (increase ?dfl ?sfl)))
```

6 Time and Processes

6.1 Process Syntax and Semantics

If a domain declares requirement `:processes`, then it can have process definitions:

```
<process-def>     ::= (:process(<name> <name arg spec>)
                                [- <val type spec>]
```

```
                                  <process-def body>)
    <process-def body>
                                                :existential-preconditions
             ::= [<vars-spec>]                  :conditional-effects
                 [:condition <goal proposition>]
                 [:start-effect <effect proposition>]
                 [:effect <effect proposition>]
                 [:stop-effect <effect proposition>]
```

A process is like an action, except that it becomes active whenever its `:condition` is true, and remains active only for the time intervals over which the condition is true. The process is denoted by a constant term (*name* —*args*—), as described in its definition. The term names a unique process; there cannot be two distinct processes active at the same time named by the same term.

Processes have three effects. To explain them, I have to explain, informally, a bit of the formal semantics of OPT.

A *situation* is a snapshot of the state of the universe, i.e., a complete specification of the truth values of all logical formulas (which I will call "propositions" for brevity). The logical language of OPT is tenseless, so there are no formulas referring to the future or the past — and none referring to what time it is, although by using processes one can create clocks and hence get a time reading if you want.[6]

This semantic notion is distinct from the syntactic construct of section 10, although obviously the latter is meant to describe one of the former.[7]

In classical planning, situations occur in discrete chains, with instantaneous changes of the truth values of some propositions from one to the next. What happens in between is undefined, because there is no "in between." Processes, however, can describe continuous changes. For instance, we might describe the process of water leaking from a can thus:

```
(:process (leaking c - Can h - Hole)
     (:vars r - Float•)
     (:condition (and (hole-in h c)
                      (>= (level c) (height h))
                      (= (rate-constant c) r)))
     (:effect (deriv-comp• (level c)
                           (* r (- (level c) (height h))))))
```

In English: "If there is a hole `h` in can `c`, then so long as the water level stays above the height of `h`, the derivative of the level will have as one component an amount proportional to the difference between the water level and `h`'s height." We use `deriv-comp•` here instead of `derivative•` because there may be other holes, plus other flows into the can, and it's the job of the inference system, using a closed-world assumption, to infer what the net derivative is.

[6] Although my semantic ideas borrow heavily from [**?**], this concept of situation is different from Reiter's. He defines a situation in terms of the actions required to reach it. This is natural in some contexts, but in cases where a situation is not reachable by an agent's action, or where we don't care, and may not know, how a situation might have been reached, the analysis gets a bit strained.

[7] And, as a previous footnote discusses, the semantics of `not` in action effects depends on the idea that every situation that occurs as the result of a chain of actions have a finite description using this syntax.

Obviously, the semantics of the `leaking` process can't be described using discrete strings of situations. Instead we introduce the notion of a *continuum* of situations, in which an infinite number of situations can occur over a finite time period, during which quantities of type `Float•` can change continuously.

Continuous and discrete changes have to be able to coexist. For instance, if there is an action `puncture` that makes holes in cans, a planner may introduce occurrences of it in order to speed up the emptying of a can. The most natural approach is to make `puncture` be a garden-variety, instantaneous, classical action. To allow for this, we use an idea from nonstandard analysis [**?**], permitting there to be an indefinite number of successors to a situation within an "infinitesimal" period of time. Formally, we model a situation continuum as a mapping from $[0, \infty) \times N$ to situations. The domain of this mapping is the set of ordered pairs $\langle r, i \rangle$, where r is a real number ≥ 0 and i is an integer in the range $[0, n_r]$. The idea is that r is the date of the situation (measured in seconds from the initial situation), and i is the number of instantaneous planner actions that have occurred since r. The number n_r is a function of the continuum, and is the number of instantaneous actions that actually occur at r in this continuum. If the planning agent does nothing, then $n_r = 0$ for all r. At the points where it takes action, n_r is the number of actions it takes (in sequence) at r.

Following [**?**], we adopt the following constraint on action effects: In a given situation continuum, if an effect is imposed at point $\langle r, n_r \rangle$, then its changes must remain in place over an open interval in the continuum starting at r, that is, over an interval (r, r'), where $r < r'$.

We can then use the following semantic idea for `when•` and related constructs: `(when•` p e`)` is *triggered* at $\langle r, i \rangle$ if $i = 0$ and p is true over some open interval ending at r, or $i > 0$ and p is true at $\langle r, i - 1 \rangle$. If the `when•` is triggered, and $i = n_r$ then e is imposed over some open interval (closed on the left) starting at r; if $i < n_r$, then e is imposed at $\langle r, i + 1 \rangle$.

Real-world implementations of OPT can't calculate the exact times when processes become active or inactive. They must find some close approximation instead. The domain parameter `temporal-grain-size•` provides a hint as to what "close" means. The value of this parameter, G, is a number representing a time interval such that any change in continuous quantities over an interval shorter than G is to be treated as negligible. Time is represented in seconds, so the default value, 0.01, of `temporal-grain-size•` means that changes over an interval less than $\frac{1}{100}$ second are negligible.

The parameter `temporal-scope•` is a time interval such that any event that far in the future is irrelevant. The default value is 10^{10} seconds, or about 3171 years.

For every process P, there is a fluent `(elapsed•` P`)` that is the time (in seconds) that P has been active. In situations where P is not active, `(fl-v• (elapsed•` P`))` $= -1$. These fluents are updated automatically, and should not be altered by a user's process or action definitions.

Just as a skip-action name is of type `(Fun (Skip•` r`) <-` a`)` for some a and r, a process name is of type `(Fun (Slide•` r`) <-` a`)`. The type `Process•` is defined to be `(Slide• (Val• &rest Obj•))`.

It may seem odd to have processes "return" values, but there are cases, such as the autonomous action of an agent seeking information, where a return value makes sense. Such a value may be referred to in the `:stop-effect` field of a process definition just as an ordinary action's value may be referred to in its `:effect` field. In addition, there is an action `(wait-while•` p`)` that waits for process p to end, then returns the same value p does.

6.2 Durative Actions

Using the process formalism, it is possible to create models described by complex differential equations, which are beyond the powers of almost all existing planning algorithms. In setting up the AIPS 2002 Planning Competition, it was desirable to find ways to constrain the formalism to make for solvable problems. A good way might have been to require all derivatives to be piecewise constant. Unfortunately, the way that was chosen was the somewhat odd notion of a "durative action," an action that takes a certain amount of time. This idea may seem straightforward, when one thinks of examples like taking a plane from New York to Chicago, but even the straightforward examples start to break down if there is the slightest possibility that something might *interfere* with the action. In fact, in the cases where durative actions work, simple process models also work, so the advantages are not obvious.

Nonetheless, as a public service, OPT includes durative actions. Here is the grammar:

```
<durative-def>      ::= (:durative-action
                                (<name> <name arg spec>)
                                [ - <val type spec>]
                            <durative-def body>)
<durative-def body>::= :duration <duration constraint>
                       :condition <durative goal>
                       :effect <durative effect>
<duration constraint>
                    ::=[:duration-inequalities]
                    (and <simple duration constraint>+)
<duration constraint>
                    ::= <simple duration constraint>
<simple duration constraint>
                    ::= (<d-op> ?duration <numerical expression>)
<simple duration constraint>
                    ::= (at <time-specifier>
                            <simple duration constraint>)

<d-op>              ::= =
<d-op>              ::=[:duration-inequalities] =< | >=
<time specifier>    ::= at start | at end | over all
```

The syntactic categories `<durative goal>` and `<durative effect>` are the same as `<goal proposition>` and `<effect proposition>`, except that the following special construct may occur within them:

`(<time specifier>` p`)`,
where p itself contains no time specifiers.

The intended meaning is that p be true either at the beginning or end of the durative action, or that it be true throughout its execution.[8]

[8]The symbols `at` and `over` are not exactly reserved words of OPT. Users may use them as predicates if they wish, provided there is no possibility of taking an occurrence of one of them as the beginning of a time specifier.

In addition, in the `:effect` field of a durative action, the symbol `#t` means "elapsed time so far." So an effect like `(decrease (fuel-level ?p) (* #t (consumption-rate ?p)))` means that the fuel level of `?p` decreases continuously as a linear function of time (assuming that `(consumption-rate ?p)` is not itself changing).

We can explain how durative actions work by viewing them as "macros" that expand into an underlying process-based representation. (Cf. [**?**].) Suppose we have a durative definition

```
(:durative-action a - v
     :duration d
     :condition c
     :effect e)
```

First, introduce some new functions:

- `(dur-happening•` a`)`: The process corresponding to durative action a.
- `(dur-start•` a`)`: The action of starting the durative process.
- `(dur-stop•` a`)`: The action of stopping the durative process.
- `(dur-in-progress•` p $d_=$ $d_\geq$ $d_\leq$`)`: True if
  1. durative process p has started and not yet stopped;
  2. its duration is constrained to be $=$ to $d_=$;
  3. its duration is constrained to be $\geq d_\geq$;
  4. and its duration is constrained to be $\leq d_\leq$.
- `(dur-trace•` `[initial|throughout]` p q`)`: True if q was true when p started, and, in the `throughout` case, has been true ever since.

Second, classify the pieces of d, c, and e thus:

- *leqs*(d): The formula `(eval•` `(min•` l_1 ... l_k`)` `?Dle)` constructed by extracting l_i from each expression `(=<` `?duration` l_i`)` in d. (`?Dle` is a special logical variable that is assumed not to occur anywhere else.)
- *geqs*(d): The formula `(eval•` `(max•` g_1 ... g_k`)` `?Dge)` constructed by extracting g_i from each expression `(=<` `?duration` g_i`)` in d. (`?Dge` is a special logical variable that is assumed not to occur anywhere else.)
- *eqs*(d): The conjunction of `(eval•` e `?Deq)` for each expression `(=` `?duration` e`)` in d; if there are no such expressions, then *eqs*(d) is `(=` `?Deq` `?Dle)`. (`?Deq` is a special logical variable that is assumed not to occur anywhere else.)
- *stc*(c): The conjunction of all primitive subformulas p of c (see below) that either occur with an explicit `(at start` p`)`, or do not occur inside an `at` or `over`.
- *enc*(c): The conjunction of all primitive subformulas p of c that occur inside an explicit `(at end` p`)`.

- *oac*(c): The conjunction of all primitive subformulas p of c that occur inside an explicit `(over all` p`)`.
- *ste*(e): The conjunction of all primitive subformulas p of e that occur inside an explicit `(at start` p`)`.
- *dte*(e): The conjunction of primitive formulas p' obtained from all primitive subformula p of e of the form `([increase | decrease]` f `(* #t` d`))`, by replacing the `(increase ...)` or `(decrease ...)` with `(derivative` f d`)` or `(derivative` f `(-` d`))` respectively.
- *ene*(e): The conjunction of all primitive subformulas p of e that contain no occurrence of `#t`, and either occur with an explicit `(at end` p`)`, or do not occur inside an `at` or `over`.

The phrase "primitive subformula" used repeatedly in these definitions is defined as follows: Quantifiers and `when`s may occur in conditions and effects, in various combinations. A *primitive subformula* is an atomic formula or its negation, with the quantifiers and `when`s above it intact (after `not`s have been moved in as far as possible). For example, suppose the following occurs in an effect E:

```
(forall (u)
    (and (when (at start (P x y u))
               (at start (F x u)))
         (when (over all (Q y u))
               (not (G x u))))))
```

Here `x` and `y` are action parameters, while `u` is a locally quantified variable. Then *ste*(E) will include this conjunct:

```
(forall (u)
    (when (at start (P x y u))
        (F x u)))
```

and *ene*(E) will include this one:

```
(forall (u)
    (when (over all (Q y u))
        (not (G x u))))
```

We require one further transformation on these subformulas, which is to eliminate all occurrences of "`at start`" and "`over all`" from the `when`s. We do this with the `dur-trace`$^\bullet$ predicate introduced above. All occurrences of `(at start` q`)` are replaced by `(dur-trace`$^\bullet$ `initial` p q`)`, and all occurrences of `(over all` q`)` are replaced by `(dur-trace`$^\bullet$ `throughout` p q`)`, yielding the following as the final versions of the fragment of E we started with:

ase(E):

```
(forall (u)
    (when (dur-trace• initial (dur-happening• D) (P x y u))
        (F x u)))
```

ene(E):

```
(forall (u)
    (when (dur-trace• throughout (dur-happening• D) (Q y u))
        (not (G x u))))
```

where D is the term denoting the durative action.

Finally, we have to set up new effects to maintain the `dur-trace•`s. Define +*asene*(e) as a conjunction containing, for each `(at start` q`)` found, a conjunct of the form `(when` q `(dur-trace• initial (dur-happening•` D`)` q`))`. Define +*oaene*(e) in an analogous way for all the `(over all` q`)`'s, substituting `throughout` for `initial`. Finally, define -*oaene*(e) as a conjunction containing, for each `(over all` q`)` found, a conjunct of the form `(when (not` q`) (not (dur-trace• throughout (dur-happening•` D`)` q`)))`.

In our example, +*asene*(E) would contain

```
(when (P x y u) (dur-trace• (dur-happening• D) initial (P x y u)))
```

+*oaene*(E) would contain

```
(when (Q y u) (dur-trace • (dur-happening• D) initial (Q y u)))
```

and -*oaene*(E) would contain

```
(when (not (Q y u))
    (not (dur-trace• (dur-happening• D) throughout (Q y u))))
```

Now we can expand the given durative definition into two or three definitions, depending on whether *eqs*(d) is empty or not. If it's empty, then there is no set duration for the action, and the action ends only when explicitly stopped. The planner can decide to stop the action after any time interval that satisfies the given inequalities *geqs*(d) and *leqs*(d). But if *eqs*(d) is nonempty, then the duration is completely constrained by those equalities. Stopping would be redundant or contradictory, and so we do not define `dur-stop` at all.

Fortunately, the process of starting a durative action is the same in either case:

```
(:action (dur-start• a)
     :vars (Deq Dle Dge - Float•)
     :precondition (and stc(c) oac(c) eqs(d) leqs(d) geqs(d)
                        (=< Dge Dle) (=< Deq Dle))
     :effect (and (dur-in-progress• a Deq Dge Dle)
                  ste(e) +asene(e) +oaene(e)))
```

If *eqs*(d) is nonempty, then there is no definition for `dur-stop`, and the definition of `dur-happening` is as follows:

```
(:process (dur-happening• a)
     :vars (Deq Dle Dge - Float•)
     :condition (and (dur-in-progress• a Deq Dge Dle)
                     (< (elapsed• (dur-happening a))•
                        D=))
     :effect (and dte(d) -oaene(d))
     :stop-effect (and (when enc(c)
                             ene(e))
                       (not (dur-in-progress• a Deq Dge Dle))))
```

If $eqs(d)$ is empty, then the process definition is as follows:

```
(:process (dur-happening• a)
     :vars (Deq Dle Dge - Float•)
     :condition (and (dur-in-progress• a Deq Dge Dle)
                     (< (elapsed• (dur-happening• a))
                        D<))
     :effect (and dte(d) -oaene(d))
     :stop-effect (and (when (and enc(c)
                                  (dur-in-progress a Deq Dge Dle)))
                             ene(e))
                       (not (dur-in-progress a Deq Dge Dle))))
```

and `dur-stop•` is defined thus:

```
(:action (dur-stop• a) - v
     :vars (Deq Dle Dge - Float•)
     :precondition (and (dur-in-progress• a Deq Dge Dle)
                        (>= (elapsed• (dur-happening• a))
                            Dge)
                        (=< (elapsed• (dur-happening• a))
                            Dle))
     :effect (and ene(e)
                  (not (dur-in-progress• a Deq Dge Dle))))
```

Note that when $eqs(d)$ is empty, there are two ways for the durative action to end. Either the time limit `D<` can be reached, or the planner can execute the action `(dur-stop• a)`. The first case is handled by the `dur-happening•` definition, the second by the `dur-stop•` definition. Both definitions must be ready to impose the effect $ene(e)$, but at most one will be applicable for any given process instance.

Although this translation process sounds complex, most of its subtleties are never exercised, so its outputs are straightforward. For instance, here is a durative-action definition from the 2002 AIPS competition:

```
(:durative-action fly
```

```
 :parameters (?a - Aircraft ?c1 ?c2 - City)
 :duration (= ?duration (/ (distance ?c1 ?c2) (slow-speed ?a)))
 :condition (and (at start (at ?a ?c1))
                 (at start (>= (fuel ?a)
                         (* (distance ?c1 ?c2) (slow-burn ?a)))))
 :effect (and (at start (not (at ?a ?c1)))
              (at end (at ?a ?c2))
              (at end (increase total-fuel-used
                         (* (distance ?c1 ?c2) (slow-burn ?a))))
              (at end (decrease (fuel ?a)
                         (* (distance ?c1 ?c2) (slow-burn ?a))))))
```

Here is the translation:

```
(:process (dur-happening• (fly ?a - Aircraft ?c1 ?c2 - City))
     :vars (Deq Dle Dge - Float•)
     :condition (and (dur-in-progress• (fly ?a ?c1 ?c2) Deq Dge Dle)
                     (< (elapsed• (dur-happening• (fly ?a ?c1 ?c2)))
                        Deq))
     :stop-effect (and (at ?a ?c2)
                       (increase• total-fuel-used
                                  (* (distance ?c1 ?c2) (slow-burn ?a)))
                       (decrease• (fuel ?a)
                                  (* (distance ?c1 ?c2) (slow-burn ?a)))
                       (not (dur-in-progress• (fly ?a ?c1 ?c2) Deq Dge Dle))))

(:action (dur-start• (fly ?a - Aircraft ?c1 ?c2 - City))
     :vars (Deq - Float•)
     :precondition (and (at ?a ?c1)
                        (>= (fuel ?a)
                            (* (distance ?c1 ?c2) (slow-burn ?a)))
                        (eval (/ (distance ?c1 ?c2) (slow-speed ?a))
                              Deq))
     :effect (and (dur-in-progress (fly ?a ?c1 ?c2) Deq −∞ ∞)
                  (not (at ?a ?c1))))
```

I should point out that these "virtual" process and action definitions are not syntactically valid OPT, because of the presence of the non-variable term a (e.g., `(fly ?a ?c1 ?c2)`) as an argument. You could make them legal by introducing functions such as `dur-start-fly`, `dur-happening-fly`, and so forth, and discarding the "extra level of parentheses."

The type of a symbol defined using `:durative-action` with value r and arguments a is `(Fun (Hop` r`) <-` a`)`, the same as an action with expansions; see next section.

7 Action Expansions

An *expansion* of an action A is a structure of actions E such that successfully executing E is a way of executing A. For instance, to get a Windows machine to become healthy when it is sick, it suffices to reboot. To reboot, you must click the "Shut down" entry on the Start menu, then select "Restart," then wait until the computer is back to its normal operating state. These three actions are an expansion of the action "Reboot."

It is sometimes possible to avoid the use of expansions by introducing artificial propositions. For instance, one could say:

- To make a Windows machine m healthy, do the primitive action[9], "Wait until normal desktop appears," with precondition `(P2` m`)`.
- To achieve condition `(P2` m`)`, select "Restart" from the active menu, with precondition `(P1` m`)`.
- To achieve condition `(P1` m`)`, click the "Shut down" entry on the Start menu.

Here `(P1` m`)` means, in essence, "Shut down has been clicked and Restart has not been selected"; and `(P2` m`)` means, "Restart has been selected, but the normal dektop has not appeared yet." Putting things this way doesn't seem to provide much enlightenment. There are really only one or two ways to reboot a Windows machine, and concealing them beneath a pile of artificial preconditions seems pointless (except, of course, that many planners think only in terms of causes and effects, not expansions, so for them this tactic is a necessity). Furthermore, the causal explanation the `P1`-`P2` story provides for why clicking Shutdown and selecting Restart actually do what they're supposed to is obviously bogus. There is a correct causal story, but most of us don't know much about it. About all we know is: Click Shutdown, select Restart, wait.

The Windows-reboot expansion can be formalized thus:

```
(seq• (click shut-down) (select restart) (wait desktop-normal))
```

where `(seq•` a_1 `...`a_n`)` means, intuitively, "Do a_1, then a_2,..." and so forth.

If we augment `seq•` with `parallel•`, we can build large structures of actions. But the actions will have no relations with each other besides the basic sequencing mechanism. To go further, we need another building block, the *action link*, a sort of communication channel between steps.[10] A link gets a *value* when one action in an expansion sets it. The value is read by later actions.

For example, a plan to ask someone for a credit-card number and then send it to an authorization center might be written

```
(with-links (cclnk - (ccnum - String•))
    (seq• (link (ask-credit-card cust101) :output cclnk)
          (link (send (!_ccnum cclnk) auth-center)
                :input cclnk)))
```

[9] Waiting is not really a plausible primitive; it should really be modeled using autonomous processes; see section 6.

[10] This idea is the joint work of Mark Burstein and me. In some ways it is very similar to the idea of a *rendezvous* in the the Ada programming language [**?**], but not quite identical.

where `(ask-credit-card)` is defined as

```
(:action (ask-credit-card x - Customer)
    :value (Val• cc - String•)
    :effect (know-val-is (ccnum x) cc))
```
(although of course the real version of this action would have its own expansion)

The `link` specs explain how the values are transmitted: `:output` means that the link gets set, `:input` means that it gets read, by this action.

In other cases, links are used to express step-ordering relationships that go beyond what `seq`• and `parallel`• can express. See below.

OPT allows plans like these in the `:expansion` fields of action definitions, provided the domain declares requirement `:action-expansions`. Actions defined using `:expansion` fields are said to be *hops*; the others are *skips*. A skip occurs "instantaneously" (in that no other events occur between when it begins and when it ends). A hop is defined by an expansion, and occurs whenever a complex of subactions described by the expansion occurs. Two such actions A_1 and A_2 may well occur over overlapping subintervals, so that primitive actions that are part of A_2 occur during the interval spanned by the execution of A_1.

The presence or absence of the `:expansion` field in an action definition is an unambiguous indicator of whether it is primitive or composite. As we will discuss in section 9, an action may have extra expansions, called *methods*, defined after the action is. If all the expansions of the action are defined this way, then the `:expansion` field must still be present, but with the placeholder `:methods` instead of an action spec. (Even though the description of methods is postponed to section 9, a method may be defined in the same domain definition in which its action is defined.)

The `:expansion` field in an action definition has the syntax described below.

```
<action spec>      ::= <action-term>
<action spec>      ::= (with-links (<typed row (name)>)
                           <action spec>
                           <action constraint>*)
<action spec>      ::= (link <link rel>*
                             <action term>
                             <link rel>*)
<action spec>      ::= (forsome <typed list(variable)>
                           [<goal proposition>]
                           <action spec>)
<action spec>      ::=:foreach-expansions
                      (foreach <typed list(variable)>
                               <goal proposition>
                               <action spec>)
<link rel>         ::== <link mode> <name>
<link mode>        ::== :output | :then | :begin-then
                   | :input | :wait | :wait-end | :span
<action constraint>
                   ::= (:postcondition <link name> <proposition>)
<action constraint>
```

```
                    ::= (:maintain <link name> <proposition>)
```

If there is a choice of expansions, it is indicated using `choice`•. `(choice`• A_1 ... A_k`)` is executed whenever any of the A_i is executed. `(forsome` v $P[v]$ $A[v]$`)` is executed by executing any instance of $A[v]$ such that v satisfies $P[v]$ when execution begins. `(foreach` v $P[v]$ $A[v]$`)` is executed by executing all instances of $A[v]$ such that v satisfies $P[v]$ when execution begins. (The domain must declare requirement `:foreach-expansions` for a `:foreach` to be legal.)

A link declared as in this example

```
(with-links (ll - (u v - Integer•))
   ...body ...)
```

is treated, in *body*, as an object of type `(Lnk`• `u v - Integer`•`)` that is, a `Lnk`• that is to contain a value of type `(Val`• `u v - Integer`•`)`. A link is either *set* or *unset*. It can be set at most once. A link L is set when an action of the form `(link` A `:output` L`)` is finished executing, in which case L is set to the value of A (as defined in A's definition) when A is finished. The two keywords `:then` and `:begin-then` are similar to `:output`, except that they set L to `(values)`, i.e., to an empty row of values (of type `(Val`•`)`). `:begin-then` differs from the other two in that it sets L when A begins; if A is composite, then its beginning and its end can be different.

Once it is set, the link L has a value accessible as `(lv` L`)`. However, this value is a "row," not an object, so the only way to use it is to access one of its fields. In the example above, the `Integer`• `u` stored in link `ll` can be obtained as `(!_u (lv ll))`.

If a `link` action specifies an `:input` L or `:wait` L field, then the action must not begin execution until the link L is set. When the action begins execution, L is said to be *read*.[11] The difference between `:wait` and `:input` is that in `(order` A `:input` L`)`, all references to L in A are automatically coerced to `(lv` L`)`. Extending the previous example,

```
(with-links (ll - (u v - Integer•)
             mm - ())
   (parallel
       ...
       (link (proceed 3)
           :wait ll)
       (link (transmit (!_u ll) (!_v ll))
           :input ll)
           :wait mm)
       (link (deltrans (!_w nn) (!_v (lv ll))
                       ll rr)
           :wait ll
           :input nn
           :output rr)))
```

The `proceed` step just uses the link as a synchronization device; it does not execute until some previous step sets `ll`. The `transmit` step also waits, for `mm` and `ll` to be set, then uses the `!_u`

[11] More than one step may access a link; it is read on the first access.

and `!_v` slots of `(lv ll)`. Note that `mm` has no useful value, so its only purpose is to coordinate plan steps.

The `deltrans` step shows how these devices can be mixed and matched. It refers to three different links, `ll`, `nn`, and `rr`; the bindings of the last two are not shown. It waits for `ll` and `nn` to be set, then takes four values as arguments. The first two are the `!_w` slot of `(lv nn)` and the `!_v` slot of `(lv ll)`; the next two are the links `ll` and `rr` themselves. Presumably `deltrans` is defined by an expansion that uses the value of `ll` and sets the value of `rr`. The definition of `deltrans` might start like this:

```
(:action deltrans
    :parameters (z - Float• i - Integer•
                 in - (Lnk• u v - Integer•)
                 out - (Lnk• x - Float•))
    ...)
```

A link may also be used to define contextual conditions on plans, by using action constraints in a `with-link` action spec. The constraint `(:postcondition` L P`)` means that the first time L is read, P must be true. `(:maintain` L P`)` means that from the time L is set to the time it is first read, P must be true. If the link is set but never read, then P must be true until the entire `with-link` action is finished. The time from when a link is set to the time it is read or its defining `with-links` finishes is called its *wait interval*.

OPT also supplies `(link` A `:wait-end` L`)`, which reads L when A finishes. If L has not been set at that point, then the `link` action suspends until it is set, then reads it and finishes. As we will see below, it is often useful to have a link be set when an action begins, and read when it ends:

`(link` A `:begin-then` L `:wait-end` L`)`

This pattern can be abbreviated `(link` A `:span` L`)`.

One can abbreviate the common pattern

```
(seq•...
    (link a1 ...:output L ...)
    (link a2 ...:input L ...)
    ...)
```

as

```
(seq•...
    a1
    :link L
    a2
    ...)
```

The "extra" conditions declared using `:postcondition` and `:maintain` must not be conditions on the *feasibility* of the actions reading the links in question, unless the actions are defined using the flag `:only-in-expansions`. An action defined this way has no independent life, but serves as a convenient organizational linchpin for hierarchical plans. In this case every expansion containing an instance of the action must specify its preconditions as `:postconditions` on links it reads.

An ordinary action appearing in an expansion may also have a postcondition declared on a link it reads, but in this case it must be for a secondary precondition, i.e., a condition that is not required for the action to be feasible, but may be required for it to have one effect rather than another.

We illustrate `:wait-end` and `:maintain` with the definition of a plan to remove all children and pets from a lawn, then put pesticide on it:

```
(with-links (protec19) ;; the default type of a link is (Lnk•)
    (seq (link (foreach (x - Animal)
                   (and (in x ?area) (or (human-child x) (pet x)))
                   (remove x ?area))
               :then protec19)
         (link (apply-pesticide ?area)
               :wait-end protec19))
    (:maintain protec19
         (not (exists (x - Animal)
                  (and (in x ?area)
                       (or (human-child x) (pet x)))))))
```

The `parallel` construct imposes no constraints on the execution order of its constituents. It begins when the first of them begins, and ends when the last of them ends. So, to indicate that a condition be true from the end of `act1` until a set of actions performed in parallel with `act1` are finished, write

```
(with-links (protec39)
    (link (parallel (link act1 :then protec39)
                    (act2)
                    ...
                    (actN))
          :span protec39)
    (:maintain protec39
         (condition)))
```

If an action with expansions is declared to return a value of type R, then there must be a mechanism that allows the expansion to construct and return such a value. One part of the mechanism is a built-in link `result•` defined implicitly in every expansion, such that `:outputting` V to `result•` causes the expanded action to return value V. The other is a built-in action `(collect-value•` *−args−*`)` that does nothing but return a row of the *−args−* (which may be arbitrary expressions).

Here is an example of how these devices might work:

```
(:action (signal-attack-route enemy)
         - (i - Integer•)
   :expansion
      (seq• (climb-tower)
           (test• (see-direction enemy land-route)
                 (link (collect-value• 1)
```

```
        :output result•)
    (link (collect-value• 2)
        :output result•))))
```

This action returns 1 if the enemy is coming by land, 2 if by sea, as is traditional.

If a has expansions, then `(seq•` a b`)` means to do all of the actions of a before doing b. If you want finer control, so that after all the "substantive" steps of a b may proceed while some "cleanup" steps of a are done, write

```
(with-links (control)
    (parallel• (fa ...control)
               (link :input control
                     b)))
```

where a = `(`f_a `...)`. That is, add an argument to f_a for the `control` link, so that some step of a can set `control` and allow b to proceed.

An action name defined with `:expansions` is of type `(Fun• (Hop•` r`) <-` a`)`, where r is the result type and a is the argument type. The name "hop" seems to imply that hops always take longer than skips, but that's not quite right. A skip always takes exactly one infinitesimal time unit. In the notation of section 6.1, it lasts from $\langle r, i\rangle$ to $\langle r, i+1\rangle$. A hop can take place over any interval at all, from one with zero duration, to one that takes several infinitesimal steps, to one that lasts over a measurable time period.

An `Action•` is either a `Skip-action•` or a `Hop-action•`. The type `(Step•` r`)` is `(Alt• (Skip•` r`) (Hop•` r`))`.

Processes can be part of action expansions just by including actions to start and, if necessary, stop them. We must also provide actions to wait for them to complete. The hop action `(wait-while•` p`)` waits for p to complete. If p is of type `(Slide` r`)`, and returns value v (of type r), then `(wait-while•` p`)` returns v as well. If p is not active, but has already finished, then `(wait-while•` p`)` takes zero time and returns p's value. If p has never been active, `(wait-while•` p`)` waits for it to become active, then finish.

See Section 9 for a notation that allows cumbersome action expansions to be broken into more manageable pieces.

8 Facts and Axioms

The `:facts` field of a domain contains a list of propositions. These propositions are true in all situations in the domain, without exception.[12] We refer to these as *domain facts*. In section 10, we will allow propositions to be associated with defined situations, in the `:init` field. These we will call *situation facts*.

There are two variants of `if` that play a crucial role in facts:

`(->` a c`)`
`(<-` c a`)`

[12] They may be overridden in subdomains; see section 10.

Each of these means the same as `(if` a c`)`, but the former is used for forward chaining, the latter for backward chaining. Forward chaining operates as follows: the first time a and `(->` a c`)` are simultaneusly asserted, c will be concluded and asserted as well. The statement that p is *asserted* means that p is "explicitly present," i.e., stated as a domain fact or situation fact, introduced as the effect of an action, or deduced by forward chaining.[13] Of course, it is not really necessary that $a = b$ for a and `(->` b c`)` to interact via forward chaining, only that a and b have a unifying substitution θ, in which case $\theta(c)$ will be asserted.

Backward chaining through `(<-` c a`)` should occur when some program engages in proving g, and g that unifies with c yielding substitution θ. In that case the fact licenses the attempt to prove $\theta(a)$; for every successful instance $\phi(\theta(a))$, the program may conclude $\phi(\theta(g))$.

In addition to these applications, some planners may use `(if` p q`)` or its variants to reduce goals. That is, if a planner is trying to make q true, it may try to accomplish that by making p true. There are many other complex issues surrounding the interaction of facts and action definitions, but OPT is based on the assumption that *all effects of an action are explicitly stated in its definition*. For example, if a domain contains the facts

```
(forall (x - Physob r1 r2 - Room)
    (if (and (in x r1) (in x r2)) (not (= r1 r2))))
(forall (x - Physob p - Person r - Room)
    (if (and (holding p x) (in p r))
        (in x r)))
```

then if person `Fred` is holding object `ob23`, and is in `living-room52`, and the action `(goto-room Fred kitchen102)` occurs, the facts can remain true only if either `(holding Fred ob23)` or `(in ob23 living-room52)` becomes false. The definition of `goto-room` must state which it is. (It might say, "Forall objects the person is holding, if they are `portable` then they change room, otherwise they cease to be held.")

For historical reasons, and for compatibility with PDDL, OPT allows a domain to declare axioms using:

```
<axiom-def> ::= (:axiom
                    :vars (<typed list (variable)>)
                    :context <goal proposition>
                    :implies <literal(term)>)
```

The domain declaration

```
(:axiom
     :vars v
     :context a
     :implies c)
```

is exactly synonymous with putting `(forall` v `(<-` c a`))` in the `:facts` field of the domain definition.

For example, we might define the classical blocks-world predicates `above` and `clear` as follows:

[13] Hopefully the sly mutual recursion in these two definitions will go unnoticed.

```
(:axiom
     :vars (?x ?y - physob)
     :context (on ?x ?y)
     :implies (above ?x ?y))

(:axiom
      :vars (?x ?y - physob)
      :context (exists (?z - physob)
                  (and (on ?x ?z) (above ?z ?y)))
      :implies (above ?x ?y))

(:axiom
      :vars (?x - physob)
      :context (or (= ?x Table)
                   (not (exists (?b - block)
                           (on ?b ?x))))
      :implies (clear ?x))
```

Unless a domain declares requirement `:true-negation`, `not` is treated using the technique of "negation as failure" [**?**]. That means it makes no sense to *conclude* a negated formula; they should occur only as deductive goals, when `(not` g`)` succeeds if and only if g fails. (If g contains variables, the results are undefined.) Hence axioms are treated directionally, always used to conclude the `:implies` field, and never to conclude a formula from the `:context` field.

9 Adding Facts and Action Expansions Modularly

Although OPT allows a domain to be defined as one gigantic `define`, it is often more convenient to break the definition into pieces. The following notation allows adding axioms and action expansions to an existing domain:

```
<addendum-def> ::= (define (addendum <name>)
                           (:domain <name>)
                           <extra-def>*)
<extra-def>    ::= <facts-def>
<extra-def>    ::=:domain−axioms <axiom-def>
<extra-def>    ::=:action−expansions <method-def>
<extra-def>    ::=:safety−constraints <safety-def>
<method-def>   ::= (:method <action function>
                            [:name <name> ]
                            <action-def body>)
```

Inside a `(define (addendum ...) ...)` expression, `:actions` and `:axioms` behave as though they had been included in the original `(define (domain ...) ...)` expression for the domain. `:method` declarations specify further choice points for the expansion of

an already-declared action, almost as though the given `<action-def body>` included inside a `choice`$^\bullet$ in the original expansion of the action. (It doesn't work quite that neatly because the parameters may have new names, and because an `<action-def body>` is not exactly what's expected in a `choice`$^\bullet$.)

In a method definition, the `<action-def body>` may not have an `:effect` field or an `:only-in-expansions` field.

Method names are an aid in describing problem solutions as structures of instantiated action schemas. Each action has its own space of method names; there is no need to make them unique over a domain. If an action has a method supplied in its original definition, the name of that method is the same as the name of the action itself.

Example (from an old version of the UM-Translog domain):

```
(define (addendum carry-methods)
    :domain translog
    ...
    (:method carry-via-hub
         :name usual
         :parameters (?p - package ?tc1 ?tc2 - tcenter)
         :expansion (forsome (?hub - hub)
                        (exists (?city1 ?city2 - city
                                 ?reg1 ?reg2 - region)
                            (and (in-city ?tc1 ?city1)
                                 (in-city ?tc2 ?city2)
                                 (in-region ?city1 ?reg1)
                                 (in-region ?city2 ?reg2)
                                 (serves ?hub ?reg1)
                                 (serves ?hub ?reg2)
                                 (available ?hub)))
                        (seq (carry-direct ?p ?tc1 ?hub)
                             (carry-direct ?p ?hub ?tc2)))
         :precondition (not (hazardous ?p)))
         ...)
```

The reason to give addenda names is so the system will know when an addendum is being redefined instead of being added for the first time. When a `(define (addendum` N`) ...)` expression is evaluated, all the material previously associated with N is erased before the definitions are added. The name of an addendum is local to its domain, so different domains can have addenda with the same name.

10 Situations and Problems

As discussed in section 1, a situation is a particular set of propositions that may all hold at some time (or many times) in a particular domain. For instance, in a domain with just one predicate, `(light-on)`, and just one action, `(toggle-light)`, there are two situations, {`(light-on)`}

and {(not (light-on))}.[14] The time of a situation is not part of its description, so that literally the same situation can recur more than once.

A planner may produce descriptions of several different situations in the course of its search for a plan. What OPT supplies is a way of declaring *initial situations,* which provide the starting points for stating plan problems:

```
<situation-def> ::= (define (situation <initsit name>)
                        (:domain <name>)
                        [<objects def>]
                        [<facts def>]
                        [<init spec>]
                        [(:init <proposition>+)])
                    <init spec>
```

The `:objects` and `:facts` fields are just like the corresponding fields of a domain definition. The facts newly declared must be true in every situation reachable from this one. A proposition in the `:init` field of a situation is true in this situation, and persists until some action or event makes it false.

A problem is what a planner tries to solve. It is defined with respect to a domain. A problem specifies two things: an initial situation, and a goal to be achieved. Because many problems may share an initial situation, there is a facility for defining named initial situations.

```
<problem-def>          ::= (define (problem <name>)
                               (:domain <name>)
                               [<require-def>]
                               [<situation>]
                               [<objects def>]
                               [<facts def>]
                               [<init spec>]
                               <goal>+
                               [<metric spec>]
                               [<length-spec> ])
<situation>            ::= (:situation <name>)
<goal>                 ::= (:goal <goal proposition>)
<goal>                 ::=:action-expansions
                              (:expansion <action spec(action-term)>)
<metric spec>          ::= (:metric <min-or-max> <metric term>)
<min-or-max>           ::= minimize | maximize
```

A `problem` definition must specify some combination of an initial situation (by name), a list of facts, or a list of initial true literals. The situation `<name>` must be a name defined by either a prior

[14] Technicality: When I describe a situation by giving a finite list of propositions, I mean the deductive closure of that list.

`situation` definition or a prior `problem` definition. If a `<problem def>` specifies an initial situation, then the facts and inits are treated as effects (adds and deletes) to the named situation. The `:objects` field, if present, describes objects that exist in this problem or initial situation but are not declared in the `:constants` field of its domain or any superdomain. Objects do not need to be declared if they occur in the `:init` list in a way that makes their type unambiguous.

All atomic formulas which are not explicitly said to be true in the initial conditions are assumed by OPT to be false, unless the domain declares requirement `:open-world`.

The `<metric term>` appearing in a `:metric` field is a term of type `Number`$^\bullet$ that may (but need not) contain subterms `(total-time`$^\bullet$`)` and `(total-steps`$^\bullet$`)`. The values of these terms are the total time (in seconds) elapsed from the initial situation to the goal situation, and the total number of steps taken, respectively. [[Why they are parenthesized is not clear to me; I'm just following PDDL2.1 here.]]

For example,

```
(define (situation briefcase-init)
   (:domain briefcase-world)
   (:objects P D)
   (:init (place home) (place office)))

(define (problem get-paid)
    (:domain briefcase-world)
    (:situation briefcase-init)
    (:init (at B home) (at P home) (at D home) (in P))
    (:goal (and (at B office) (at D office) (at P home)))
    (:metric minimize (+ (* 3 (total-time)) total-steps)))
```

The `:goal` of a problem definition may include a goal description or (if the domain has declared the requirement `:action-expansions`) an expansion, or both. A solution to a problem is a series of actions such that (a) the action sequence is feasible starting in the given inital situation situation; (b) the `:goal`, if any, is true in the situation resulting from executing the action sequence; (c) the `:expansion`, if any, is satisfied by the series of actions ([[in a sense that I haven't made clear yet]]).

For instance, in the transportation domain, one might have the problem

```
(define (problem transport-beans)
   (:domain transport)
   (:situation standard-network)
   (:objects beans27 - Beans)
   (:init (at beans27 chicago))
   (:expansion (with-links (got-to-end)
                  (link (carry-in-train beans27 chicago newyork)
                     :span got-to-end)
                   (:postcondition got-to-end
                        (not (spoiled beans27))))))
```

The `:requirements` field of a problem definition is for the rare case in which the goal or

initial conditions specified in a problem require some kind of expressiveness that is not found in the problem's domain.

Unlike addendum names (see Section 9), problem names are global. Exactly how they are passed to a planner is implementation-dependent.

11 Web Mode and Namespaces

One important application of Opt is to planning and reasoning on the "Semantic Web," the envisioned future world-wide network of agents and information sources that have more explicit propositional content than WWW pages have today [**?**]. Opt fits into this vision by being a representation for ontologies and datasets. But there are a couple of other dimensions to address. A key design decision for RDF, the standard representation system for the Semantic Web[**?**], is that all names are URIs, Uniform Resource Identifiers. The rationale is that URIs are a familiar notation for web denizens, with enough degrees of freedom to ensure that the same name is unlikely to be used by two different people for different purposes.

There are several drawbacks. URIs are long and complicated, and don't look like names, especially for things like predicates. The standard notation for RDF is based on XML, and you can't put a URI just anywhere in XML. The measures taken to get around XML's limitations lead to a notation whose conventions are awkward, inconsistent, and hard to remember.

When Opt is used in "web mode," it adopts the same requirement that all names are URIs, but fits it into the notational base in a cleaner way. First, domains fetched from the web have URIs as names, instead of symbols. If a document has URL[15] `http://hostname.com/`p, and that document consists of a definition of domain d, then the URI `http://hostname.com/`p`?notation=opt&ontol=`d is the "web name" of that domain. If the last component of p is d`.`e, then the query `ontol=`d can be dropped; if the extension e is `opt`, then the query `notation=opt` can be dropped. These conventions allow a document to contain more than one domain definition, while giving the principal domain a concise URL. I write "`http://`," but the same idea works for other schemes, such as "`ftp://`." However, for scheme "`file://`," that is, for a local ontology, the name is assumed to be unambiguous.

Second, every domain implicitly defines a *namespace,* so that identifiers from different domains don't get confused. This namespace can be used in the traditional XML way. Here's an example of XML markup

```
<?xml version='1.0' encoding='ISO-8859-1'?>

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY zoo "http://cadmium.yale.edu/ontology/zoology.opt">
]>
   <rdf:RDF xmlns:zoo="&zoo;#">
      <rdf:description rdf:ID="Katie">
         <zoo:location rdf:resource="&zoo;#kangaroo-pen"/>
      </rdf:description>
   </rdf:RDF>
```

[15] A URL is a URI from which an actual document can be retrieved.

However, the XML namespace mechanism, whatever its virtues in general, has shortcomings as a device for name management among different ontologies. It allows at most one ontology to be the default namespace, so that if ontology A extends ontology B, and if the default namespace inside A is A itself, then every reference to a name from B must be preceded by a prefix. What we'd like to be able to do is import all the names from B, and, as long as there is no possibility of ambiguity, pretend that they live in A's namespace.

If there is such a possibility, we can introduce a prefix for the B symbols, as in this example:

```
(define (domain automobiles)
   (:extends (:domain
               "http://cadmium.yale.edu/ontology/zoology.opt"
               :prefix "zoo"))
   ...)
```

where, for concreteness, we've let

$$\begin{aligned} \text{A} &= \texttt{http://}\textit{host-A}\texttt{/automobiles.opt} \\ \textit{and}\ \text{B} &= \texttt{http://cadmium.yale.edu/ontology/zoology.opt"}. \end{aligned}$$

Now the symbol `kangaroo` in B can be referred to as `@zoo:kangaroo`.[16]

It looks like we've essentially duplicated the XML namespace machinery, but the details work out to be quite different. Suppose that `jaguar` is the only symbol declared in A that clashes with a symbol in B. Then we can write

```
(define (domain automobiles)
   (:extends (:domain
               "http://cadmium.yale.edu/ontology/zoology.opt"
               :prefix ("zoo" jaguar) :prefix ("" :remainder)))
   ...)
```

Now an occurrence of `@zoo:jaguar` in A means the B symbol `jaguar`, and an unprefixed `jaguar` means the A version. All other B symbols can be referred to with no prefix.

This is as good a place as any to put the definition of the syntactic extensions promised in section 4:

```
<parent>         ::= (:domain <domain-ref>
                          [:prefix <prefspec>]*)
<domain-ref>     ::= <name> | <url>
<url>            ::= String conforming to RFC [<>])
<prefspec>       ::= <string> | (<string> <import syms>)
<import syms>    ::= <name>*| :all | :all-but <name>*| :remainder
<export-def>     ::= (:exports <export syms>)
<export syms>    ::= <name>+| :all | :all-but <name>*
<name>           ::= @<basicname><more prefix>+:<basicname>
<more prefix> ::= :<basicname>
```

[16]The atsign is required because Opt is Lisp-based, and the colon character has a built-in significance to the Lisp reader, so that letting `zoo:kangaroo` be our notation without some major headaches is impossible. Using "`@`" in this way would be equivalent to the notation "`!`*space*`:`*sym*" of [**?**]; the use of atsign follows [**?**].

A <name> is *prefixed* if it contains one or more prefixes. In that case, its *root name* is the name given by the characters after the last colon; if it has no prefixes, the root name is the name itself.

If a domain has no :exports declaration, then all of its symbols are (potentially) imported into any domain that :extends it. I say "potentially" because, as suggested above, the :prefix clauses of an :extends clause can be used to block some of the importations that would otherwise occur.

In a :prefix clause, an empty <import syms> is synonymous with :remainder, which means "all symbols not mentioned to the left." The flags :all and :all-but refer to all symbols, or all but a finite list. A string s by itself is synoymous with s :remainder. So the :extends spec above could have been written

```
(define (domain automobiles)
   (:extends (:domain
                "http://cadmium.yale.edu/ontology/zoology.opt"
                :prefix ("" :all-but jaguar)
                :prefix ("zoo" jaguar)))
  ...)
```

or as

```
(define (domain automobiles)
   (:extends (:domain "http://cadmium.yale.edu/ontology/zoology.opt"
                :prefix ("zoo" jaguar) :prefix ""))
  ...)
```

If it's necessary to restrict the symbols exported from a domain, an explicit :exports clause can be used, in the obvious way. (:exports :all) is equivalent to no :exports declaration. (:exports *syms*) gives an explicit list of symbols to be exported. (:exports :all-but *syms*) causes all but the given symbols to be exported from the domain's namespace.

A key feature of this mechanism[17] is that once a prefix has guided Opt to a domain, the rest of the symbol designator can be any legal symbol specification, *including one with colons in it*. So if it happens that B imported the symbol foo from another ontology C with prefix "wow", one can refer to it in A as @wow:foo. If we had linked A and B by writing

```
(define (domain automobiles)
   (:extends (:domain "http://cadmium.yale.edu/ontology/zoology.opt"
                :prefix "zoo"))
  ...)
```

then this symbol would be referred to as @zoo:wow:foo. Of course, it is usually a better idea for A to inherit the domain where foo is declared directly. The point is that A has access to all the symbols exported by B, no matter where they originated. Such symbols, even if not referred to in the definition of A, may still pop up during runs of applications involving A. If so, the iterated-colon notation provides a way to refer to them.

The basic idea behind all this machinery is that it is possible, and desirable, to hide the URI machinery backing symbols. A <name> in Opt refers to the URI you get by following its prefixes

[17] Which owes more to programming-language technology [**?**] than to Web tradition.

back to a domain, which I'll call its *root namespace,* then pairing the domain's namespace with the root name as a fragment.[18] That is, if the prefixes take you back to root namespace `http://`h`/`p, and the root name is s, then the URI it designates is `http://`h`/`p`#`s.

There is no requirement flag for web mode, because the distinction between "local mode" and web mode is too fundamental. Explaining this point requires talking about implementation details that are beyond the scope of this manual; the bottom line is that "local Opt" and "web Opt" can be considered to be different systems. The former is suitable for applications in which a small number of domains are loaded in, a computation is performed, and the program quits. The paradigmatic example is using Opt as the basis for a planning system competing in a competition: you load a domain (which might extend one or two others), load a problem definition, run the planner, type out performance statistics, and quit. "Web Opt" is for applications in which domains are found incrementally by various web-oriented agents. Name clashes are entirely possible, and are controlled using namespaces. The emphasis is less on raw performance than on the ability to find and exploit resources dynamically.[19]

12 Scope of Names

Here is a table showing the different kinds of names and over what scope they are bound

Name type	*Scope*
Reserved word	OPT language
Domain name	Global
Type	Domain, inherited
Type-function	Domain, inherited
Type constructors	Domain, inherited
Constant	Domain, inherited
Parameter	Domain, inherited
Function	Domain, inherited
Predicate	Domain, inherited
Action function	Domain, inherited
Process function	Domain, inherited
Addendum	Domain, local
Situation name	Domain, inherited
Problem name	Global
Method	Per action function

[18]The algorithm to find the root namespace is nonobvious, because any colon in a prefixed name can stand for zero or more empty prefixes. The namespace system guarantees that if a path gets to two namespaces, each with a declaration of the same root name, an error will be signaled when the second path is created.

[19]Some implementation details for hackers: Opt is implemented in Lisp. In local mode, everything that looks like a Lisp symbol is implemented as a Lisp symbol, a standard tactic that is one of the reasons Lisp is such a terrific programming language. In web mode, we introduce a data type *canonized symbol*, to which most user-defined names resolve. (Built-in symbols continue to resolve to Lisp symbols.) Dealing with such symbols complicates the programmer's life, because the smooth interface between code and data structure that the Lisp `quote` mechanism provides is gone. One is reduced to transforming quoted structures into internal data structures in the awkward and painful way that Java and C programmers must live with. This distinction is so fundamental that one has to recompile the entire Opt system in namespace mode; it can't be switched at run time.

Global names (for domains and problems) are accessible "everywhere." For domain names, "everywhere" might be the entire World-Wide Web (or Semantic Web); see section 11. It is less obvious what it means for problem names to be global, because they can't be referred to from within Opt. In practice, referring to problems by their names lies outside Opt entirely, and is managed by the planning system that is given a problem to solve.

Names with scope "domain, inherited" are visible in a domain and all its descendants. Names with scope "domain, local" are visible within a domain but are not visible in descendant domains. Method names are a documentation convenience, and need have no scope except that of the function of which they are methods.

There is limited possibility of overloading names in OPT. The same name may be used for a global-scope entity (e.g., a problem) and a domain-scope entity (e.g., a predicate). But the same domain-scoped name cannot be used for two different kinds of entity. For instance, the same name cannot be used for a type and an action.

The rules for method names are looser, because they are not true names. The only restriction is that two distinct methods for the same action may not have the same name.

13 Built-in Symbols

13.1 Types

Table 1 gives the built-in types of OPT. If the middle column is not blank, it specifies a requirement that must be declared for the type to be available.

13.2 Requirement Flags

Table 2 is a table of all requirements in OPT. Some requirements imply others; some are abbreviations for common sets of requirements.

13.3 Built-in Constants

This section contains an alphabetical list of built-in constants of OPT. For each we give the type, plus the context in which the symbol may be used. This is usually a requirement flag, but may include other notes. The annotation *Effects* means that the symbol may be used only in an effect (section 5.2). If the context is "—," then the symbol may be used anywhere the type rules allow. If an argument position is an evaluation context (see section 3), it is flagged with a $\star$.

	Type:	*Context:*

=	`(Fun Boolean`
	`  <- (x y - u`
	`  !& u - (T)))`
		`:equality`

`(= x y)` if and only if x and y are the same object; the expression is well formed only if x and y are of the same type. Note that if f_1 and f_2 are fluents, `(= f1 f2)` tests whether they are the same

Type	*Requires*	*Meaning*
`Action`	—	A `Skip-action` or a `Hop-action`
`(Alt` y_1 `...`y_k`)`	—	The type of an object that has type y_1, type y_2, ..., or type y_k. `(Alt)` is a synonym for `Void`.
`(Arg ...)`	—	An "argument tuple," often implicit in the argument position of `Fun` types.
`Boolean`	—	`true` or `false`
`(Con` c_1 `...`c_k`)`	—	The type consisting of just the constants (literals) c_1 to c_k. The symbolic constants must be quoted, except for `false` and `true`. So `(Con true 'false)` is the (somewhat contrived) type that consists of true and the symbol `false` and no other objects.
`(Either` y_1 `...`y_k`)`	—	Synonymous with `(Alt` y_1 `...`y_k`)`.
`Float`	—	Floating-point number
`(Fluent` y`)`	—	`(Fun` y `<- Situation)`
`(Fun` r `<-` a`)`	—	Function from type a to type r
`(Hop` r`)`	`:action-expansions`	The type of an action that might take anywhere from zero time to a long time interval, producing a value of type r.
`Hop-action`	`:action-expansions`	An action of type `(Hop` r`)` for some r.
`Integer`	—	Integer (any number of digits)
`(Lnk ...)`	`:action-expansions`	See section 7
`(Lst` y`)`	`:data-structures`	A list of elements of type y.
`Number`	`:numbers`	A `Float` or `Rational`
`Obj`	—	The universal type; every object is of this type.
`Process`	`:processes`	An entity of type `(Slide` r`)` for some r.
`Prop`	—	`(Fluent Boolean)`
`(Skip` r`)`	—	The type of an action that takes exactly one infinitesimally long time interval and returns a value of type r.
`Skip-action`	—	An action of type `(Skip` r`)` for some r.
`Situation`	—	A world state
`(Slide` r`)`	`:processes`	The type of a process that returns a value of type r.
`(Step` r`)`	—	An action with value of type r
`String`	—	String of characters
`Symbol`	—	A Lisp-style symbol
`(T)`	—	A type
`(Tup ...)`	`:data-structures`	See section 2.2
`(Val ...)`	—	A "value tuple;" see section 2.2; these usually occur implicitly in `Fun` types.
`Void`	—	The empty type

Table 1: Built-in types

Requirement	*Description*
`:strips`	Basic STRIPS-style adds and deletes
`:disjunctive-preconditions`	Allow `or` in goal descriptions
`:equality`	Support `=` as built-in predicate
`:existential-preconditions`	Allow `exists` in goal descriptions
`:expression-evaluation`	Allow predicates some of whose arguments are evaluation contexts
`:numbers`	Allow numerical predicates and functions (Implies `:expression-evaluation`)
`:universal-preconditions`	Allow `forall` in goal descriptions
`:quantified-preconditions`	= `:existential-preconditions` + `:universal-preconditions`
`:conditional-effects`	Allow `when` in action effects
`:action-expansions`	Allow actions to have `:expansions`
`:foreach-expansions`	Allow actions expansions to use `foreach` (implies `:action-expansions`)
`:domain-axioms`	Allow domains to have `:axioms`
`:safety-constraints`	Allow `:safety` conditions for a domain
`:fluents`	Support type `(fluent` t`)`. Implies `:numbers`
`:open-world`	Don't make the "closed-world assumption" for all predicates — i.e., if an atomic formula is not known to be true, it is not necessarily assumed false
`:true-negation`	Don't handle `not` using negation as failure, but treat it as in first-order logic (implies `:open-world`)
`:adl`	= `:strips` + `:typing` + `:disjunctive-preconditions` + `:equality` + `:quantified-preconditions` + `:conditional-effects`
`:ucpop`	= `:adl` + `:domain-axioms` + `:safety-constraints`
`:data-structures`	Expect lists, tuples, etc.
`:processes`	Allow process definitions. Implies `:fluents`
`:durative-actions`	Allow durative-action definitions

Table 2: Requirement flags

fluent, not whether they have the same value. To test for that, write either `(= (fl-v` f_1`) (fl-v` f_2`))` or, for Floats, `(=~` f_1 f_2`)`, which OPT coerces to `(=~ (fl-v` f_1`) (fl-v` f_2`))`

`<, >, =<, <=, >=`

Type:	*Context:*
`(Fun Boolean` `  <- (x* y* - Number))`	`:numbers`

The usual inequalities. Although `(=<` x y`)` is the preferred way to express $x \leq y$, you can also use `<=`, in spite of the fact that it looks more like a left-pointing arrow than an inequality.

`=~, /=~`

Type:	*Context:*
`(Fun Boolean` `  <- (x* y* - Float))`	`:numbers`

`(=~` x y`)` is true if x and y are approximately equal. The definition of "approximately equal" is implementation-dependent, but at least must satisfy the following: if a situation is arrived just after `(wait-for (=~` q_1 q_2`)` p`)`, then in that situation q_1 and q_2 must be approximately equal. `(/=~` x y`)` is an abbreviation of `(not (=~` x y`))`.

`+, -, *, min, max`

Type:	*Context:*
`(Fun Number` `  <- (&rest l - Number))`	`:numbers`

Arithmetic functions that take an indefinite number of arguments. Each of these functions is overloaded, so it produces type `Number` only if at least one argument is of type `Number`; otherwise, it produces `Float` if at least one argument is of type `Float`; otherwise, it produces type `Integer`.

`/`

Type:	*Context:*
`(Fun Number` `  <- (x y - Number))`	`:numbers`

`(/` x y`)` is $\frac{x}{y}$.

`assign`

Type:	*Context:*
`(Fun Prop` `  <- ((Fluent Number) Number))`	`:fluents`, *Effects*

`(assign` q v`)` is the effect imposed by setting fluent q to value v.

`bounded-int`

Type:	*Context:*
`(Fun Prop` `  <- (i* j* k* - Integer))`	`:numbers`

`(bounded-int` i j k`)` is true if i lies in the interval $[j, k]$. In addition, if `(bounded-int ?v` j k`)` occurs as a deductive goal, it succeeds in $k - j + 1$ ways, in the mth of which `?v` is bound $j + m - 1$.

`choice`

Type:	*Context:*
`(Fun (Step Void)` `  <- (&rest (Step Void)))`	`:action-expansions`

`(choice` a_1 `...`a_k`)` is the action executed by executing one of the the a_i.

`collect-value`

Type:	*Context:*
`(Fun (Step` a_1 `...`a_n`)` `  <- (Arg` a_1 `...`a_n`))`	`:action-expansions`

`(collect-value` $-args-$`)` is the action executed by constructing and returning a row of the given arguments.

`current-value`

Type:	*Context:*
`(Fun Prop` `  <- (f - (Fluent ?u) v - ?u))`	`:fluents`

`(current-value` f v`)` is true in a situation if the value of f in that situation is v.

`decrease`

Type:	*Context:*

`(Fun Prop`
`  <- ((Fluent Number)`
`  Number*))`
	`:fluents`, *Effects*

`(decrease` v q`)` is the effect imposed by decreasing fluent v by amount q.

`derivative`

Type:	*Context:*
`(Fun Prop` `  <- (f df - (Fluent Float)))`	`:processes`, *Effects*

`(derivative` f df`)`, allowed only in the `:effect` fields of process definitions, is imposed by causing fluent f to vary at the rate df.

`deriv-comp`

Type:	*Context:*

`(Fun Prop`
`  <- (f df`
`  - (Fluent Float)))`
	`:processes`, *Effects*

`deriv-comp` differs from `derivative` in that it specifies a component of the derivative; the actual derivative is the sum of all known components.

`dur-happening`

Type:	*Context:*
`(Fun Process` `  <- (Hop ?r))`	`:durative-actions`

`(dur-happening` a`)` is the process that exists when durative action a is being executed

`dur-in-progress`

Type:	*Context:*
`(Fun Prop` `  <- (Step ?r))`	`:durative-actions`

`(dur-in-progress` a`)` is true when durative action a has begun and not yet ended.

`dur-start`

Type:	*Context:*
`(Fun (Step Void)` `  <- (Step ?r))`	`:durative-actions`

`(dur-start` a`)` is the skip action that consists of starting the durative action a.

`dur-stop`

Type:	*Context:*
`(Fun (Step Void)` `  <- (Step ?r))`	`:durative-actions`, when `?duration` is not constrained by equalities.

`(dur-stop` a`)` is the skip action that consists of stopping the durative action a, when the ability

to stop is under the planner's control.

elapsed	*Type:*	*Context:*
	`(Fun (Fluent Float) <- (Fun Process <- Situation))`	`:processes`

`(elapsed` p`)` is the time that the current instance process p has been active; if there is no current active instance of p, then `(elapsed` p`)` is undefined.

equation	*Type:*	*Context:*
	`(Fun Boolean <- (x y - Number))`	`:numbers`

`(equation` x y`)` is true if $x = y$, but it has the further pragmatic meaning that if there is a single unbound logic variable in a goal of this form, a planner should try to find a value for that variable that makes the equality true.

eval	*Type:*	*Context:*
	`(Fun Boolean <- (x* y - Obj))`	`:expression-evaluation`

`(eval` e v`)` is true if e evaluates to v, using `fl-v` and standard arithmetic functions (e.g., `+`).

eval-test	*Type:*	*Context:*
	`(Fun Boolean <- (x*))`	`:expression-evaluation`

`(eval-test` e`)` is true if e evaluates to " to v, using `fl-v` and standard arithmetic functions (e.g., `+`).

false	*Type:*	*Context:*
	`Boolean`	—

A constant with value false.

`fl+, fl-, fl*, fl/`

Type:	*Context:*
`(Fun (Fluent Number)` `  <- (x y - (Fluent Number)))`	`:fluents`

Each of these functions produces a fluent whose value in any situation is the sum, difference, product, or quotient of the values of the two argument fluents.

`fl-^`

Type:	*Context:*
`(Fun (Fluent ?u)` `  <- ?u)`	`:fluents`

`(fl-^` x`)` is the fluent that has value x in every situation. In practice, the ^ symbol is the carat character, typed "shift-6" on most American keyboards.

`fl-v`

Type:	*Context:*
`(Fun ?u` `  <- (Fluent ?u))`	`:fluents`

`(fl-v` f`)` is the value of fluent f in the current situation.

`increase`

Type:	*Context:*

`(Fun Prop`
`  <- ((Fluent Number)`
`  Number*))`
	`:fluents`, *Effects*

`(increase` v q`)` is the effect imposed by increasing fluent v by amount q.

`list`

Type:	*Context:*
`(Fun (Lst ?u)` `  <- (&rest l - ?u))`	`:data-structures`

`(list` x_1 `...`x_n`)` is the list consisting of x_1 to x_n in that order. If each x_i has type y, then the `list` expression has type `(Lst` y`)`

.

parallel	*Type:*	*Context:*
	`(Fun (Step Void) <- (&rest (Step Void)))`	`:action-expansions`

`(parallel` a_1 a_2 `...`a_k`)` is the action executed by executing all of $a_1, \ldots, a_k$, in no particular order. It starts when the first of the a_i starts, and finishes when the last of the a_i finishes.

result	*Type:*	*Context:*
	Context-dependent	`:action-expansions`

In an action expansion, a link of type r, where r is the type returned by the action being expanded. Setting this link to value V causes the expanded action to return V.

seq	*Type:*	*Context:*
	`(Fun (Step Void) <- (&rest (Step Void)))`	`:action-expansions`

`(seq` a_1 a_2 `...`a_k`)` is the action executed by executing $a_1, a_2, \ldots, a_k$ in that order.

temporal-grain-size	*Type:*	*Context:*
	`Number`	`:processes`

A time interval small enough that nothing important happens over any stretch of time shorter than this. This is a domain parameter, whose default value is 0.01, which may be overridden in subdomains.

temporal-scope	*Type:*	*Context:*
	`Number`	`:processes`

A time interval large enough that nothing relevant happens further in the future than this. This is a domain parameter, whose default value is 10.0`e`10, which may be overridden in subdomains.

test	*Type:*	*Context:*
	`(Fun (Hop Void) <- (s - Boolean iftrue iffalse - Action))`	`:action-expansions`

`(test` s at af`)` is equivalent to at if the value of expression s is `true`$^\bullet$, and to af if its value is `false`$^\bullet$.

`total-steps`

Type:	*Context:*
`(Fun Integer <- ())`	*The* `:metric` *field of a problem definition*

The number of steps (skips and hops) in the action sequence from the initial situation to a candidate goal situation.

`total-time`

Type:	*Context:*
`(Fun Float <- ())`	*The* `:metric` *field of a problem definition*

The total time elapsed from the initial situation to a candidate goal situation.

`true`

Type:	*Context:*
`Boolean`	—

A constant with value true.

`tuple`

Type:	*Context:*
`(Fun (Tup ?u`$_1$ `...?u`$_n$`)` `<- (?u`$_1$ `...?u`$_n$`))`	`:data-structures`

`(tuple` x_1 `...`x_n`)` is a tuple of type `(Tup` y_1 `...`y_n`)`, where y_i is the type of x_i. If you think this type specification is illegal, you're right; it takes us beyond the type language we described in this paper.

`wait-for`

Type:	*Context:*
`(Fun Hop-action` `<- (Prop Process))`	`:processes`

`(wait-for` q p`)`, where q is an inequality involving a quantity affected by process p, is the action of waiting until q becomes true.

`wait-while`

Type:	*Context:*
`(Fun (Hop ?r)` `<- (Slide ?r))`	`:processes`

`(wait-while` p`)` is the action of waiting for process p to finish. If p finishes, returning a value

of type r, then `(wait-while` p`)` finishes and returns the same value.