O-Plan

Implementation Guide

Artificial Intelligence Applications Institute
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
United Kingdom

July 12, 1995

Version 2.3

Acknowledgements

The O-Plan project began in 1984. Since that time the following people have participated: Colin Bell, Ken Currie, Jeff Dalton, Roberto Desimone, Brian Drabble, Mark Drummond, Anja Haman, Ken Johnson, Richard Kirby, Glen Reece, Arthur Seaton, Judith Secker, Austin Tate and Richard Tobin.

Prior to 1984, work on Interplan (1972–4) and Nonlin (1975–6) was funded by the UK Science and Engineering Research Council and provided technical input to the design of O-Plan.

From 1984 to 1988, the O-Plan project was funded by the UK Science and Engineering Research Council on grant numbers GR/C/59178 and GR/D/58987 (UK Alvey Programme project number IKBS/151). The work was also supported by a fellowship from SD-Scicon for Austin Tate from 1984 to 1985.

From 1989 to 1992, the O-Plan project was supported by the US Air Force Rome Laboratory through the Air Force Office of Scientific Research (AFOSR) and their European Office of Aerospace Research and Development by contract number F49620-89-C-0081 (EOARD/88-0044) monitored by Northrup Fowler III at the USAF Rome Laboratory.

From 1992 to 1995, the O-Plan project was supported by the ARPA/Rome Laboratory Knowledge Based Planning and Scheduling Initiative through the US Air Force Rome Laboratory through the Air Force Office of Scientific Research (AFOSR) and their European Office of Aerospace Research and Development by contract number F49620-92-C-0042 (EOARD/92-0001) monitored by Northrup Fowler III at the USAF Rome Laboratory.

Additional resources for the O-Plan and O-Plan projects have been provided by the Artificial Intelligence Applications Institute through the EUROPA (Edinburgh University Research on Planning Architectures) institute development project.

From 1989 to 1993, research on scheduling applications of the O-Plan architecture was funded by Hitachi Europe Ltd. From 1989 to 1992, the UK Science and Engineering Research Council (grant number GR/F36545 – UK Information Engineering Directorate project number IED 4/1/1320) funded a collaborative project with ICL, Imperial College and other partners in which the O-Plan architecture was used to guide the design and development of a planner with a flexible temporal logic representation of the plan state. A number of other research and development contracts placed with AIAI have led to research progress on the O-Plan prototype.

Contact Information

The O-Plan project team can be contacted as follows:

O-Plan Team
Artificial Intelligence Applications Institute
The University of Edinburgh
80, South Bridge
Edinburgh EH1 1HN
United Kingdom

Tel: (+44) 131 650 2732
Fax: (+44) 131 650 6513
Email: oplan@ed.ac.uk

Created: 2nd December 1991 by Brian Drabble
Last Modified: July 7, 1994 (10:29) by Brian Drabble
Printed: July 12, 1995

IMPORTANT NOTE

This presents an overview of the eventual Implementation Guide which will be completed in the near future. It is meant to provide the user with an overview of the areas in which implementation support will be described.

In the Version 2.3. only two sections have been completed and these describe the Task Formalism Compiler and the Context Layering Mechanism.

Contents

1 Introduction

The aim of this document is to describe the major functionality and modules of the O-Plan planner agent. The description of each of these items will cover:

1. the design rational behind each of the major items.
2. a systems level description of the major functions within each item.

An O-Plan agent is based on an agenda driven architecture. In this architecture, knowledge sources[1] are executed to handle issues in a plan where an issue represents further processing which is required on the plan. The agenda entries hold the list of current issues to be processed (one flaw maps to agenda entry) and the agent moves through the list of agenda entries trying to reduce the number of issues to zero, i.e. to produce a complete and correct plan.

1.1 An implementational view of the O-Plan architecture

1.2 Process models

1.3 Connecting multiple agents

The O-Plan architecture has been designed for use in multi-agent command and control scenarios, e.g. spacecraft control, logistics planning, process flows, etc. In such scenarios we envisage there is a command structure in which tasks are accepted from a superior agent, decisions are made concerning the way to carry out a task and it is then delegated to a sub-ordinate agent for execution. In theory there could be any number of O-Plan style agents active in a scenario, but to clarify the research objectives the number of agents has been limited to *three* each of which has a specific role.

The relationship between each of the agents and their communication structure is described in Figure 1. The agents are as follows:

1. **Task Assignment Agent**:
   This agent accepts task descriptions from the user of the system and uses its *local* knowledge to elaborate the task description and to decide whether the planner agent could solve the given task. If the task is acceptable it is passed to the planner agent.
2. **Planner Agent**:
   In the current set of O-Plan scenarios the planner agent is by far the most important. The role of the planner agent is to use its *local* knowledge to expand the plan to a point at which is would be "understandable" to the sub-ordinate execution agent.

[1] these are pieces of code which carry out a specific part of the agent's task

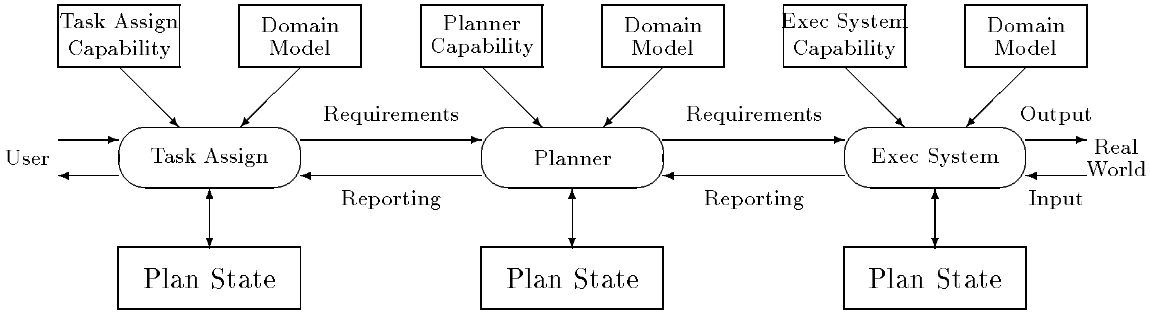

Figure 1: Three agent scenario for O-Plan

3. **Execution Agent**:
   At present the execution agent has a very limited number of capabilities which with its *local* knowledge allows it to schedule a partially ordered plan, monitor the execution of a plan in accordance with a policy specified by the user and to report on the success or failure of a plan. The execution agent is the only agent which can affect the real world directly.

Each of these agents could be built from a separate instance of the *generic* O-Plan architecture, tailored for its specific role and running on the same *or* separate machines. However, in the current implementation the task assignment agent is represented as a single process which provides a menu for user command selection and presents planner agent results. The Planner and Execution Agents are O-Plan architecture based.

The generic O-Plan architecture has been designed around *four* main bases:

1. to run under the UNIX operating system
2. coded at the application level in the LISP language
3. to communicate with external user defined viewers
4. to operate within the X WINDOWS environment

The O-Plan architecture has been designed to exploit parallel systems in the future. The agents can map to distributed computers, the agent components are suited to implementation on multi-process systems (such as a Sequent Symmetry which have a course grain level of parallelism) and the constraint management level is suited to fine grain parallel processing and constraint systems handling.

1.4 The components of an agent

This section describes in detail the main components of the O-Plan system. An overview of the O-Plan architecture is given in Figure 2.

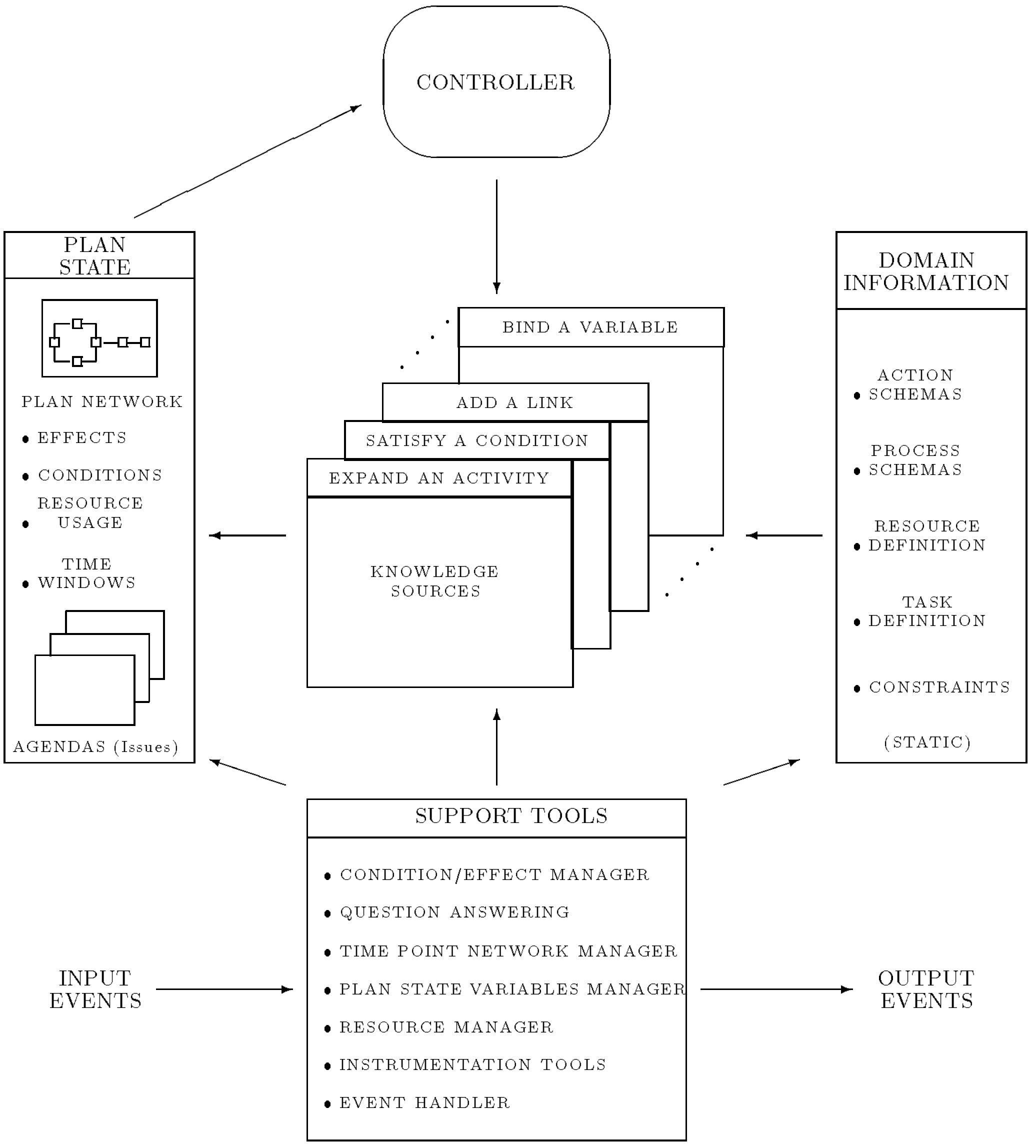

Figure 2: An Overview of the O-Plan Architecture

1.5 Agendas, contexts, and alternatives

1.6 Constraint managers

1.7 Processing cycle

2 Support

2.1 Pseudo-processes

2.2 IPC / message sending

2.3 Context-layering

2.3.1 Introduction

An access function, or *accessor*, is a function that can be applied to an object to obtain a value (another object), typically a value stored in a field or *slot*. For instance, `defstruct` defines an accessor for each slot in a structure.

In O-Plan, a *context* is an object that is used as an additional, implicit parameter to an accessor. An ordinary accessor is a mapping from objects to values; a context-layered or context-relative accessor is a mapping from objects *and the current context* to values. To make the presentation simpler, we will sometimes talk as if context-relative values were always stored in slots. However, layered accessors do not have to be based on slot accessors; they can be based on any mapping from objects to values.

A context has few properties of its own. It is simply an object with which the values considered by context-relative accessors can be associated. Contexts are arranged in a tree: each context has a single predecessor (parent) and a list of successors (children). When there is no value directly associated with a context, a value is inherited from the nearest ancestor context that does have a value associated with it. However, when a value is assigned in a context, it is always associated directly with that context; the associations between values and ancestor contexts are not changed. This makes it possible to protect the values associated with a context, while still being able to access them, by moving to a child context.

One context is always designated as the current context, and only values associated with that context or its ancestors are normally seen.

The O-Plan context-layering mechanism provides a way to define context-relative accessors based on ordinary access functions. If $\mathcal{F}$ is the name of an accessor such that

- ($\mathcal{F}$ x) accesses a slot of (or location in) objects x of a certain type, and
- (`setf` ($\mathcal{F}$ x) *value*) sets the same slot,

then a context-relative version of $\mathcal{F}$ named $\mathcal{C}$ can be defined by

(`define-context-accessor` $\mathcal{C}$ $\mathcal{F}$)

($\mathcal{C}$ x) will then return the slot value associated with the current context, directly or by inheritance, and `setf` can be used with $\mathcal{C}$ to directly associate a value with the current context.

Note that the slots of an object are normally context-layered independently. This is an advantage when only a few slots are layered or when the pattern of associations between slot values and contexts is not uniform. However, there are circumstances in which it would be better to layer entire objects instead. One way to handle this in O-Plan be to define a context-layered mapping from object ids (names) to objects and make a copy of an object (for the current context) when the first slot update is done.

2.3.2 Internals

Context-layered accessors are perfectly happy with ordinary slot values such as might be installed when an object is first created. However, when a context-relative value is first assigned (by using `setf` with a context-layered accessor), the ordinary value is replaced by a `cval` structure that contains a mapping from contexts to (ordinary) values.

Consequently, a layered accessor normally works by:

1. calling an ordinary accessor to obtain a `cval` structure, and
2. looking in the `cval` structure for the value associated with the current context or with the nearest ancestor that has a value.

Both steps can be performed by calling the function `accessincontext`. The second step, dereferencing a `cval`, can be performed by calling `derefincontext`.

The `cval` structure's mapping from contexts to values is an a-list. However, we don't want to search the complete a-list for the current context, and then search it again for each ancestor of the current context, until we finally find an entry that matches. Nor do we want to go through the a-list entries once, repeating a search along parent links from the current context each time (even if we could still make inheritance work correctly that way). Consequently, we combine the traversal of the a-list with that of the parent links in such a way that there's no repeated work.

An efficient search for the value associated with a context, or inherited from an ancestor, is accomplished by exploiting context numbers. Every context has a number which is assigned when it is created, and newer contexts have higher numbers than older ones. In particular, a context has a higher number than any of its ancestors. The a-list entries in `cval` structures are ordered by context number, with higher-numbered (newer) contexts first. The algorithm for finding the right value in a `cval` structure is therefore as follows:

1. Initialize *target* to the current context and *a-list* to the a-list in the `cval` structure.
2. Discard entries for contexts newer than *target* from the front of *a-list*. If *a-list* becomes null, return `:undef`.
3. At this point, the first of the remaining entries in *a-list* should be for *target* or for a context older than *target*. If it's for *target*, then we're done: return the associated value. Otherwise, set *target* to the parent of *target* and return to step 2.

In addition, a `cval` contains a one-context cache. If the current context is the same as the cached context, then the cached value is returned. Otherwise, the algorithm above is used to find a value, and that value and the current context are cached.

2.4 Externals

2.4.1 Defining forms

`(define-context-accessor` *name accessor-name*`)`

Defines *name* as a context-relative accessor based on *accessor-name*.

2.4.2 Variables

`*cucontext*`

A variable whose value is the currently active context. It's possible to make arbitrary moves in the context tree by assigning to or binding `*cucontext*`.

`*globalcontext*`

A variable whose value is a root context supplied by the system. When a location is first assigned a context-relative value by using `setf` with a context-relative accessor, any existing value is made the value in `*globalcontext*`.

2.4.3 Basic context operations

`(newcontext` *parent*`)` → *new child context*

Creates a new context with parent *parent*. *Parent* can be `nil` to create a new root context.

`(precontext` *context*`)` → *parent context or* `nil`

Returns the predecessor (parent) of a context.

`(succontexts` *context*`)` → *list of child contexts*

Returns a list of the successors (children) of a context.

`(contextnum` *context*`)` → *context-number*

Returns the number of a context.

If `(contextnum` *c*`)` returns *i*, then `(contextwithnum` *i*`)` returns *c*.

`(contextwithnum` *context-number*`)` → *context*

Returns the context that has the given number.

If `(contextwithnum` *i*`)` returns *c*, then `(contextnum` *c*`)` returns *i*.

`(pushcontext)` → *new child context*

Creates a new child of `*cucontext*` and then sets `*cucontext*` to be that child.

`(popcontext)` → *parent context or* `nil`

If `*cucontext*` is not a root context (i.e., if it has a non-null parent), `popcontext` sets `*cucontext*` to the parent of `*cucontext*`. Otherwise, `popcontext` simply returns `nil` without changing the value of `*cucontext*`.

`(incontext` *context function* `&rest` *arguments*`)` → *value*

Applies *function* to *arguments* with `*cucontext*` temporarily bound to *context*.

2.4.4 Value lookup and assignment

`(accessincontext` *accessor object*`)` → *value*

Applies *accessor* to *object* to obtain either an ordinary value or a `cval` structure. An ordinary value is simply returned. A `cval` structure is searched to find the value associated with `*cucontext*` or the nearest ancestor of `*cucontext*`. If such a value is found, it is returned; otherwise, `:undef` is returned.

`(updateincontext` *updater accessor object new-value*`)` → *new-value*

Establishes *new-value* as the value for `*cucontext*` so that `(accessincontext` *accessor object*`)` will find *new-value* as the value for `*cucontext*` directly rather than by inheritance from an ancestor of `*cucontext*`. The values associated with such ancestors are not changed.

Accessor is applied to *object* to obtain either an ordinary value or a `cval` structure. An ordinary value is replaced by a `cval` structure in which the ordinary value is associated with `*globalcontext*`. Otherwise, the existing `cval` structure is modified.

Updater should set the same field that *accessor* accesses so that after `(funcall` *updater object v*`)`, `(funcall` *accessor object*`)` will return *v*.

`(derefincontext` *object*`)` → *value*

If *object* is not a `cval` structure, then `derefincontext` returns *object*; otherwise, it finds a value for the current context in the same manner as `accessincontext`.

`(ctxt-symbol-value` *symbol*`)` → *value*

A context-layered version of `symbol-value` defined by:

```
(define-context-accessor ctxt-symbol-value symbol-value)
```

`(ctxt-gethash` *key hash-table*`)` → *value*, *true* or *false*
`(ctxt-puthash` *key hash-table value*`)` → *value*

A context-layered version of `gethash` and an associated update function. `ctxt-gethash` can also be used with `setf`.

2.4.5 Utilities

`(print-context-tree)`

Prints the context numbers of all contexts with children listed below their parent and indented further to the right.

3 Components

3.1 Who Owns What?

3.2 Agenda Protocols

3.3 How to Write a Knowledge Source

4 Data Structures and Constraint Managers

4.1 ADS: Nodes and Node-ends

4.2 QA and TOME and GoST Manager

4.3 Plan State Variable Manager

4.4 Time Point Network Manager

4.5 Resource Utilisation Manager

5 The TF compiler

The TF Compiler processes a domain description written in the Task Formalism language and builds or modifies a domain information structure. It is usually called by the database manager on behalf of a knowledge source, but it can also be called in syntax-checking mode by the user interface. In addition, it has been written so that it can be tested and debugged without the rest of O-Plan.

The compiler uses a fairly conventional recursive descent parser and a scanner (or "tokenizer") that employs the Common Lisp readtable mechanism. It performs a certain amount of compile-time analysis, especially of action and effect "levels". The interface to the compiler and the main components of the implementation are described below.

5.1 Interface

The TF Compiler is invoked by calling the function `compile-tf-stream`:

`(compile-tf-stream` *stream domain-information* [*check-only*]`)`
  → *revised-domain-information*,
    error-count,
    change-description

`Compile-tf-stream` takes three parameters and returns three results. The parameters are:

1. A character input stream containing TF forms followed by an end-of-file.
2. A domain information structure. This is a record structure that will be modified by the compiler (except when the compiler is only checking syntax). It also provides pre-compiled information to be used during compilation.
3. An optional flag that is *true* if the compiler should operate in syntax-checking mode and *false* (the default) if it should generate domain information.

The results are:

1. The modified domain information structure.
2. The number of errors found.
3. A description of what has been deleted, altered or added.

On the first call for a new domain, the domain information should be pre-initialized to an empty and defaulted structure before the TF Compiler is called. The domain information will not be modified if the compiler is operating as a syntax-checker or if any errors are found.

The *change-description* is simply a list of the pre-change versions of any schemas that have been changed.

5.2 Overall structure of the compiler

The TF compiler could be considered a three-pass compiler. The TF language does not require that many passes, but they provide a more modular structure for the code. The "passes" are as follows.

1. The parser

5.3 Scanner

5.3.1 Token generators

The TF scanner is implemented as a token generator: a function of no arguments that returns a new token each time it is called. If there are no more tokens, it returns the value of the variable `*end-token*`. From the point of view of the parser, any token generator can be used. For instance, when debugging the parser it may be useful to use a generator that returns successive elements from a list without doing any scanning at all.[2] The standard TF scanner is a generator that takes tokens from a character input stream and employs a syntax appropriate for the TF language. It is created by calling the function `tf-tokens`:

`(tf-tokens` *stream*`)` $\rightarrow$ *generator*

5.3.2 Tokens and character classes

The TF scanner tries to use a natural mapping from TF tokens to Lisp data structures. For instance $\langle number \rangle$s are returned as numbers and $\langle text_string \rangle$s as strings. Symbols are used for several types of tokens: for $\langle name \rangle$s; for *operators* such as `+`, `<=`, and `--->`; and for *punctuation* characters such as comma.

When forming symbols, the idea is to employ a natural grouping of characters. This is defined in terms of three disjoint *character classes*:

1. A set of *name characters* that make up normal identifiers, i.e., $\langle name \rangle$s.
2. A set of *operator characters* that make up special identifiers (operators) which may consist of more than one character.
3. A set of *punctuation characters* that result in single-character identifiers.

It's important that these sets are disjoint, so that the scanner can find the boundaries of tokens without requiring that they be separated by "white space". For characters in the name or operator classes, a maximal sequence of adjacent characters in the same set is combined into a

[2] For an example, see §5.4.6.

single token. Characters in the punctuation set are returned as single-character tokens. Thus in "`apple>=pie;`", the sequence of tokens is "`apple`", "`=>`", "`pie`", and "`;`".

It's usually clear whether a character should be an operator character or punctuation. For example, "`-`" and "`=`" have to be operator characters because they can occur in multi-character operators, and comma should be punctuation because it should always be treated as a single character. Characters such as "`+`" and "`*`" do not occur in any multi-character operators presently in TF, but for consistency it makes sense to put them in the same class as "`-`". Since both operators and punctuation are represented as symbols, however, moving a character from one class to the other changes only the grouping rule without changing the type of Lisp object that's returned.

This approach to operators has several advantages. Multi-character operators such as `>=` and `--->` can be returned as single tokens; the grouping rules are simple and do not require that the scanner perform a context-dependent analysis or be told what operators are in the language; and it's usually easy to move a character from one class to another.

However, it has the disadvantage that it's necessary to use white space or punctuation characters to delimit adjacent operators. Fortunately, adjacent operators are rare. In TF, the only place where they might occur is in arithmetic expressions. For instance, someone might write "`3*-4`" to multiply 3 by -4. Since "`*-`" will be returned as a single token, the user would have to write "`3 * -4`" or "`3*(-4)`" instead. This particular instance of the problem could be remedied by moving "`*`" to the punctuation class, but that would still leave cases such as "`3--4`".

5.3.3 Readtables

The TF scanner assigns meanings to characters by using a readtable. The next token is read from a stream by calling `read` with `*readtable*` bound to a "TF readtable". The TF scanner uses the TF readtable that is the value of `*tf-readtable*`.

The TF readtable allows names to be read as symbols in the usual way. Letters are converted to upper case, just as they are in normal Common Lisp input. As far as the readtable is concerned, the characters allowed in names are defined implicitly as the characters Common Lisp treats as "constituents" minus any characters defined as operator or punctuation characters or explicitly defined as illegal.

Tokens other than names are read by character macros. The character macros are defined according to character classes which include the classes described above plus some others such as *digit*. All macro characters in a class are given the same function as their macro definition, and this function is responsible for reading the entire token. For instance, all digits are defined to call a function that reads a number, and all operator characters are defined to call a function that reads a sequence of characters in the operator class and assembles them into a symbol. Some characters do not fit into any class and are given special macros of their own.

The most important special cases are "`;`" and "`?`".

Semicolon is a special case so that it can handle comments. A single semicolon behaves like a punctuation character and returns the symbol "`;`". However, the function that handles semicolons also looks ahead to see if the first semicolon is followed by two others. In that case, it reads the rest of the line as a comment.

Question mark is a special case so that `question-reader`, the OBase routine for reading ⟨*actor_restriction*⟩s, can also be used by the TF scanner. The macro function for "`?`" calls `question-reader` and whatever results is returned as a single token. The TF compiler then uses that token as-is.

However, `question-reader` expects some characters to have different meanings than they do in the TF readtable. For instance, it expects that "`{`" will be defined as a character macro that reads a list ending in "`}`", but in the TF readtable "`{`" is a punctuation character. Consequently, the scanner's macro for "`?`" establishes a different readtable before calling `question-reader`. This readtable is constructed by calling the function

`(make-actor-readtable` *base-readtable*`)` → *actor-readtable*

The base readtable can be any readtable, but the scanner uses the TF readtable as the base. `Make-actor-readtable` redefines characters in a copy of the base readtable to create a readtable suitable for use inside ⟨*actor-restriction*⟩s.

This example shows that it is possible to change the low-level syntax of the language within a well-defined context. However, it's important to bear in mind that the compiler has to be able to tell when it's time to switch readtables. In this case, the "`?`" character marks the start of the new context, and the syntax of ⟨*actor-restriction*⟩s is such that there is always a well-defined end. In other cases, the compiler might not be able to tell when to switch readtables, and we would have to change the TF language so that it could tell. For example, consider a `has`-actor such as `?{has length 3}`. The general syntax is as follows:

> `?{has` ⟨*function name*⟩ ⟨*argument*⟩ ...⟨*result*⟩}

If the readtable had to be changed between the last ⟨*argument*⟩ and the ⟨*result*⟩, we'd have to introduce some punctuation or other token to mark the end of the arguments. Otherwise, we couldn't tell the ⟨*result*⟩ was a ⟨*result*⟩, rather than another ⟨*argument*⟩, without reading it to see if it was followed by "}'. That is, in order to tell that the expression was a ⟨*result*⟩, we'd already have to know how to read a ⟨*result*⟩.

5.4 Parser

This section describes the parser and how it is derived from the grammar for the TF language. It also explains some of the coding conventions and briefly describes a set of auxiliary procedures that are used for such things as recognizing tokens, reporting errors, and error recovery.

Recursive descent parsers can be written almost directly from LL(1) grammars. A nonterminal symbol in the grammar is implemented as a procedure that consumes one or more tokens and then returns some result, such as a parse tree. For convenience, we will also refer to these procedures as "nonterminals".

For a grammar to be LL(1), there must not be any left-recursive productions, and it must always be possible to select between alternatives by looking only at the current token. The published grammar for the TF language is not quite LL(1), but it is still possible to write a parser by using a somewhat looser correspondence between grammar rules and procedures. It is also possible for a recursive descent parser to handle some cases that cannot be expressed in an LL(1) grammar at all.

5.4.1 Interface to the scanner

The scanner provides a token generator which is accessed from within the parser by calling the procedures `token` and `next-token`. `Token` returns the current token; `next-token` advances to and returns the next token by calling the token generator, making the result the new current token.

There is also a facility for looking ahead one or more tokens which is used at present only when a number is followed by a token that begins with "`.`". `(Push-token` *token*`)` makes the next call to `next-token` return *token*. When a number is followed by an operator such as "`..`", the scanner returns the number after calling `push-token` to save the operator. The next call to `next-token` will then return the operator.

It is important to have a convention for when the parser advances to the next token. In the TF parser, the following rules apply, and any departure from them is explicitly noted.

- When called, a nonterminal can assume that the current token has already been set to the first token it should consider.
- A nonterminal matches an initial segment of the input tokens. On exit, the current token should be the first one that is not part of the segment matched. That is, it should be the first token that the rest of the parser should consider.

5.4.2 Parsing utilities

In most cases, `token` and `next-token` are not called directly. Instead, one of the following procedures is used.

- `(token-is` *token*`)` → *true* or *false*

  `Token-is` compares its argument with the current token. If they are the same (`eql`), it advances to the next token by calling `next-token` and returns *true*. Otherwise it returns *false* without calling `next-token`.

- `(token-satisfies` *predicate*`)` → *token*

  `Token-satisfies` is used to determine whether the current token satisfies a predicate. If it does, `token-satisfies` advances to the next token by calling `next-token` then returns the token that was tested. Otherwise, `token-satisfies` returns *false* without calling `next-token`. The *predicate* is a function of one argument, a token, and returns *true* or *false*.

- `(must-be` *token*`)` → *token*

  `Must-be` is similar to `token-is` but requires that the current token be a particular token. If the current token is not acceptable, a syntax error will be reported.

- `(must-satisfy` *predicate description*`)` → *token*

  `Must-satisfy` is similar to `token-satisfies` but requires that the current token satisfy the predicate. If the current token is not acceptable, a syntax error will be reported. The *description* is a string that will be used in the error message. For example:

  ```
  (must-satisfy #'numberp "a number")
  ```

- `(one-or-more` *nonterminal* `&key :separator` *token* `:until` *end-token*`)` → *list*

  Nonterminal is a procedure that parses a nonterminal in the grammar. `One-or-more` parses a sequence of one or more occurrences of the nonterminal until a particular *end-token* appears. The *separator* is optional. `One-or-more` returns a list of the results returned by *nonterminal*.

- `(zero-or-more` *nonterminal* `&key :separator` *token* `:until` *end-token*`)` → *list*

  This function is similar to `one-or-more` but allows zero occurrences of the nonterminal.

5.4.3 Error recovery

The compiler uses a version of the "S-algol error recovery scheme" described in the book *Recursive Descent Compiling* by A. J. T. Davie and R. Morrison (Ellis Horwood, 1981), which in turn comes from a scheme developed by D. A. Turner for a SASL compiler. Some additional conventions are required by the error recovery scheme:

- If a syntax error is detected, it should be signalled by calling the `syntax-error` procedure. That call will (usually) return, after which parsing should continue as much as possible as if the syntax had been correct.

- At a point where a certain token must be present, require that it be present by calling the the procedure `must-be`. Something similar can be done for classes of tokens by calling `must-satisfy`. The "`must-`" procedures call `syntax-error` automatically.

The error recovery scheme is very simple, but it works reasonably well in a number of common cases. The basic idea is to take advantage of points where a particular token or class of tokens must occur. If the current token is not acceptable, the normal action would be to report an error. However, if the parser is already in an error state it can try to recover instead, by discarding tokens until an acceptable one appears. While the parser is waiting to recover from an error, further error messages are suppressed

5.4.4 Blocking tokens

Since the error recovery scheme can sometimes "eat" too much of the source text, it is possible to specify a list of *blocking tokens* that it will not pass. The `one-or-more` and `zero-or-more` procedures will also respect this list when they examine the current token to see whether their iteration should continue.[3]

A list of blocking tokens can be specified in two ways: by using the `:blocking-tokens` keyword parameter to `zero-` or `one-or-more`, or by binding `*blocking-tokens*`. The main use of blocking tokens in the TF compiler is when parsing a schema, when the blocking tokens are the symbols that can begin schema clauses (e.g. `expands`, `conditions`) or end schemas.

5.4.5 Translating grammar rules to Lisp code

To illustrate the coding conventions, we'll consider a subset of LL(1) grammars that can be translated directly to code and that is sufficient to cover most of the statement-level cases that arise.

A given nonterminal may be defined by more than one rule, or by one rule with several alternatives. If so, we need to be able to select among the alternatives without looking beyond the current token. It will clearly be possible to do this if all but at most one of the alternatives begins with a terminal symbol and the remaining alternative (if there is one) must begin in some other way. For instance, consider a definition of $\langle statement \rangle$ in which every alternative except the assignment statement begins with a reserved word. By looking at only the current token, we can determine what kind of statement we have. If the token isn't one of the reserved words, we have an assignment.

A rule of this form can be translated into a function definition as follows[4](or by using the equivalent `token-case` macro which inserts the calls to `next-token` automatically):

$\langle NT \rangle$ ::= t_1 $\langle NT_1 \rangle$ | t_2 $\langle NT_2 \rangle$ | ... | $\langle NT_n \rangle$

```
    ==> (defun ⟨NT⟩ ()
          (case (token)
            ((t1) (next-token) (⟨NT1⟩))
            ((t2) (next-token) (⟨NT2⟩))
            ...
            (otherwise ⟨NTn⟩)))
```

This can clearly be generalized to handle cases where the alternatives are more complex, and to cases where arbitrary predicates (not just `eql`) are applied to the current token. In other

[3]Note that this does not mean a blocking token will *never* be skipped inappropriately. Since they are not reserved words, a nonterminal might interpret them in other ways or mistake them for something else.

[4]When describing such translations, we'll present grammar rules *abstractly* and then show the corresponding Lisp. To do this, we need notations that stand for the terminal and nonterminal symbols that would actually appear in a rule, and here they are: upper-case italic names in angle-brackets such as $\langle NT \rangle$ stand for nonterminal symbols, and italic names without brackets such as t_1 or, later, *separator*, stand for terminal symbols.

cases, it will be necessary to (implicitly) transform the grammar to a form that can be handled directly.

Once we've selected an alternative, we're committed to it and should report an error if the input doesn't match what we expect. Some of the more common constructs that occur within alternatives can therefore be translated as follows:

$\langle NT \rangle$
  $\Longrightarrow \texttt{(}\langle NT \rangle\texttt{)}$

literal
  $\Longrightarrow \texttt{(must-be '}\mathit{literal}\texttt{)}$

$[\mathit{literal}\ \langle NT \rangle]$
  $\Longrightarrow \texttt{(when (token-is '}\mathit{literal}\texttt{) (}\langle NT \rangle\texttt{))}$

$\langle NT \rangle\ [\mathit{separator}\ \langle NT \rangle] \ldots\ \mathit{end}$
  $\Longrightarrow \texttt{(one-or-more \#'}\langle NT \rangle\ \texttt{:separator '}\mathit{separator}\ \texttt{:until '}\mathit{end}\texttt{)}$

Note that in the last case, both instances of $\langle NT \rangle$ must be replaced by the same nonterminal. As a concrete example, consider the following incomplete grammar for a fragment of the TF language:

$\langle \mathit{tf\ form} \rangle$	::=	`always` $\langle \mathit{always\ form} \rangle$
	|	`types` $\langle \mathit{types\ form} \rangle$

$\langle \mathit{always\ form} \rangle$	::=	$\langle \mathit{pattern\ assignment} \rangle$ [`,` $\langle \mathit{pattern\ assignment} \rangle$] ... `;`
$\langle \mathit{pattern\ assignment} \rangle$	::=	$\langle \mathit{pattern} \rangle$ [`=` $\langle \mathit{value} \rangle$]
$\langle \mathit{types\ form} \rangle$	::=	$\langle \mathit{type\ definition} \rangle$ [`,` $\langle \mathit{type\ definition} \rangle$] ... `;`
$\langle \mathit{type\ definition} \rangle$	::=	$\langle \mathit{name} \rangle$ `=` $\langle \mathit{name\ set} \rangle$

The corresponding procedures are shown below. They are written using the convention that the procedure for a nonterminal has a name in angle-brackets. Note that only the parsing code is shown, without anything that would construct a result.

```
(defun <tf-form> ()
  (token-case
    ((always) (<always-form>))
    ((types)  (<types-form>))
    (t (syntax-error "Unknown TF form."))))

(defun <always-form> ()
  (one-or-more #'<pattern-assignment> :separator '|,| :until '|;|))
```

```
(defun <pattern-assignment> ()
  (<pattern>)
  (when (token-is '=)
    (<value>)))

(defun <types-form> ()
  (one-or-more #'<type-definition> :separator '|,| :until '|;|))

(defun <type-definition> ()
  (<name>)
  (must-be '=)
  (<name-set>))
```

Of course, grammars are not always written in so accommodating a style. For instance, ⟨*types form*⟩ might have been defined as follows:

$$\begin{array}{lcl} \langle \textit{types form}\rangle & ::= & \langle \textit{types body}\rangle\texttt{;} \\ \langle \textit{types body}\rangle & ::= & \langle \textit{type definition}\rangle \\ & | & \langle \textit{types body}\rangle\texttt{,}\ \langle \textit{type definition}\rangle \end{array}$$

With this definition, some analysis is needed before it becomes clear that a ⟨*types form*⟩ is a sequence of ⟨*type definition*⟩s separated by commas and terminated by a semicolon.

This example also serves to illustrate a difficult case for us, because it involves left-recursion. Some analysis and recasting is therefore necessary. In other cases, it may not be necessary—because we can write out the recursive procedures rather than using `one-or-more`—but it may still be desirable. Indeed, most grammars not written specifically for use with a recursive descent parser will require some modification hare and there. Whether you handle this by explicitly rewriting the grammar or just by translating it more loosely into Lisp is up to you.

5.4.6 Testing

Several procedures are available for testing the parser or for parsing fragments of TF. The most important ones are:

- `(compile-tf-string` *nonterminal string*`)` → *object*, *error-count*

  This creates a TF scanner for a string input stream and calls *nonterminal*. The results are whatever object *nonterminal* returns and the number of errors that occurred.

- `(compile-tf` *nonterminal generator*`)` → *object*, *error-count*

  `Compile-tf` uses the *generator* as the scanner and calls *nonterminal*. The results are whatever object *nonterminal* returns and the number of errors that occurred.

- `(list-tokens` *list*`)` → *generator*

  `List-tokens` creates a generator that returns successive elements of a list followed by the value of `*end-token*`

Here's an example:

```
(compile-tf #'<pattern>
            (list-tokens '(|{| on a b |}|)))
```

returns the list `(on a b)`.

5.5 Compile-time analysis

5.5.1 Levels

The level algorithm assigns levels to actions, then to effects. Conditions are then checked against effects. Actions, effects, and conditions are represented by the first word of their respective patterns.

The level number of an effect is the max of the level numbers of all actions that have that effect.

The level number of an action is determined as follows:

1. Construct a directed graph G in which the vertices are actions and there's an edge from A to A' whenever A' is part of an expansion of A or can be brought in by an `achieve` / `only_use_for_effects` link.

2. Construct a graph, $SCC(G)$, of the strongly connected components of G. Two actions A and A' are in the same component if there's both a path from A to A' and one from A' to A. There's an edge in $SCC(G)$ from component C to component C' if there's an edge in G from an action in C to an action in C'.

3. Starting from actions that correspond to task schemas, determine the length of the longest path to each component in $SCC(G)$, giving each edge length 1. The length of the longest path to a component becomes the level number of all actions in that component.

6 The Plan and World Viewers

7 Executing Plans

7.1 The Execution Agent, Diary, and World

8 O-Plan as a Lisp Program

8.1 System Definitions

8.2 Code Organization

8.3 Testing and Debugging tools

8.4 Implementation-dependencies and Porting