The Oasis Programming Language Reference Manual

(Draft Version 1.0, incomplete, to be revised)

Fah-Chun Cheong
Department of Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122
(313) 763-2153
fcc@eecs.umich.edu

March 27, 1992

1 Lexical Conventions

This section describes the lexical convention of the Oasis programming language. It defines tokens in an Oasis program and describes comments, identifiers, keywords, literals (integers, floating-points, characters, strings and internet addresses).

1.1 Tokens

There are five kinds of tokens: identifiers, keywords, operators, separators and literals. Blanks, tabs and newlines are collectively called the white space and are ignored except as they separate tokens. Some white space is needed to separate otherwise adjacent identifiers, keywords and literals. If the input program stream has been separated into tokens up to a certain character, the next token is the longest string of character that could constitute a token.

1.2 Comments

The two-character sequence `/*` begins a block comment that could span multiple lines, and which is terminated by the character sequence `*/`. Block comments do not nest. Single-line comments are introduced by the hash mark character `#` which turns the rest of the line into a comment. The block comment delimiters `/*` and `*/` have no special meanings within a line comment and are treated just like other characters. Similarly, the character sequence `/*` and the hash mark `#` has no special meanings inside a block comment. The comment delimiters have no special meanings when they appear quoted within character literals or string literals.

1.3 Identifiers

Identifiers are used to name semantic entities in the program. There are three kinds of identifiers, each of which has a different micro-syntax. *ID* begins with a lowercase letter and is used for identifying object classes, constants, class attributes, method arguments and built-in mathematical functions. *CID* starts with an uppercase letter and is used for naming agent classes and local variables. Finally, *$ID* begins with a dollar sign `$` and is used for identifying condition variables and generic type variables. Although there are altogether nine distinct name spaces in Oasis, the three kinds of identifiers mentioned above suffice to distinguish them as the Oasis parser uses contextual information for disambiguation. The micro-syntax of the three kinds of identifiers are as follows:

ID ::= {`a-z`}{`a-z,A-Z,0-9,_`}*

CID	::=	{A-Z,_}{a-z,A-Z,0-9,_}*
$ID	::=	${a-z,A-Z,0-9,_}*

1.4 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

`attribute`	`char`	`class`	`constant`
`int`	`method`	`nil`	`private`
`protected`	`public`	`real`	`self`

1.5 Operators and Separators

The following characters or character sequences are used as operators or for punctuations:

```
!    "    #    $    %    '    (    )    *    +    ,    -    .
/    :    ;    <    =    >    ?    @    [    ]    {    |    }
?-   :-   |-   ::   ==   <>   <=   >=
```

Each is a single token.

1.6 Literals

There are five kinds of literals: integers, floating-points, characters, strings, and internet addresses.

An integer literal is a sequence of digits interpreted as decimals. For example, the number twelve is `12`.

A floating-point literal consists of an integer whole number part, followed by an optional fraction part with a preceding decimal point, and followed by an optional integer exponent, optionally signed and prefixed by `e` or `E`. Either the fraction part or the exponent part may be missing, but not both (otherwise it may be interpreted as an integer). For example, `3.14`, `272e-2` and `6.02E18` are floating-point numbers.

A character literal is either a single character or one of the following escape sequences enclosed in a pair of single quotes `'`. The value of a character literal is the numeric value of the character in the machine's character set at program binding time.

newline	NL	`\n`
horizontal tab	HT	`\t`
backspace	BS	`\b`
carriage return	CR	`\r`
formfeed	FF	`\f`
decimal character code	*xxx*	*xxx*
other printable character	*c*	*c*

A string literal is a sequence of characters surrounded by a pair of double quotes, as in (`"..."`). A string is typed as an array of characters and is basically a short-hand for enumerating elements in the corresponding character array instance. The same escape sequences for character literals apply to characters within strings as well. In addition, within a string the double quote character " must be preceded by a backslash, as in \\", so as not to be mistaken as the end of a string.

Finally, an internet address literal can be expressed in either the internet domain name convention or the internet dot notation for specifying 32-bit quantities. For example, both internet address literals `z.eecs.umich.edu` and `141.212.99.7` refers to the same internet address.

INTEGER	::=	{0-9}$^+$
FLOAT	::=	{0-9}$^+$[.{0-9}$^+$][{e,E}[-]{0-9}$^+$]
CHARACTER	::=	'{a-z,A-Z,0-9,_,*ASCII*}'
STRING	::=	"{a-z,A-Z,0-9,_,*ASCII*}*"
INTERNET	::=	{a-z,A-Z,0-9,_}$^+$ {.{a-z,A-Z,0-9,_}$^+$}$^+$
	|	{0-9}$^+${.{0-9}$^+$}$^+$
ASCII	::=	{\n,\t,\b,\r,\f,\\{0-9}$^+$,\ascii,ascii}

2 Program Organization

An Oasis program is organized as a collection of class *definitions.* A class definition is the basic unit of compilation. Multiple class definitions may thus be spread over several files for separate compilation as well as for easier maintenance. Each class definition denotes an abstract compile-time semantic entity called a *class.* Intuitively, classes could be viewed as some kind of templates for dynamically creating *objects* and *agents*, which are also abstract semantic entities, as instances of their representative classes.

A user can create and interact with objects and agents through an Oasis shell which accepts commands from the user in the form of *goal clauses.* The Oasis shell compiles each goal clause into native code of the local machine prior to dynamically linking and loading the object code into the Oasis shell itself for execution. Upon whose completion, an answer is promptly displayed on the screen and the Oasis shell is ready to accept the next goal clause from the user.

A class definition has two parts: a *specification* and an *implementation.* The class specification declares the interface of a class as it relates to other classes. The class implementation describes the runtime behavior of instances of that particular class. Class specifications are textually decoupled from their corresponding class implementations and they usually reside in separate header files for sharing purposes. At the top-level, an Oasis program looks like:

$$\begin{array}{lcl} \textit{program} & ::= & \textit{sections goal} \\ \textit{sections} & ::= & \textit{sections section} \\ & | & \textit{section} \\ \textit{section} & ::= & \textit{specification} \\ & | & \textit{implementation} \end{array}$$

A class specification consists of a header part followed by a list of *declarations.* The header provides a name to the class and establishes it as either the *root* of a family of yet-to-be defined classes or the child of some previously specified parent class. In other words, the class header describes the lineage of the corresponding class in some *inheritance hierarchy*, to be further described in a following section. *Constants*, *attributes* and *methods* are abstract semantic entities associated with an Oasis class. The declarations part describes the types of constants and attributes of the class and also provides them with default values. Furthermore, the call interface to methods of the class is also specified in the declarations. These declarations are used mainly for compile-time type-checking, but they are also useful at runtime for other purposes like garbage collection and the automatic marshaling/unmarshaling of remote procedure call parameters. An Oasis class specification looks like:

$$\begin{array}{lcl} \textit{specification} & ::= & \textit{header} \ \{ \ \textit{declarations}_0 \ \} \\ \textit{declarations}_0 & ::= & \textit{constant-lists}_0 \ \textit{attribute-lists}_0 \ \textit{methods}_0 \end{array}$$

A class implementation completes the definition of a class by providing the actual code which describes how methods previously declared in the class specification are to be carried out when properly invoked at runtime. A method is defined by an ordered list of textually adjacent *clauses.* The clauses which define a particular method are identified by the same method name appearing in the *head* of each clause. The implicit textual boundaries of lists of clauses defining different methods of a class can thus be clearly demarcated without resorting to syntax. A class implementation is a collection of such clauses, tagged with a name identifying the class itself. An object or agent class implementation looks like:

$$\begin{array}{lcl} \textit{implementation} & ::= & \textit{object-id} \ \{ \ \textit{clauses}_0 \ \} \\ & | & \textit{agent-id} \ \{ \ \textit{goal}_0 \ \textit{clauses}_0 \ \} \end{array}$$

An agent class implementation may optionally prepend a *goal clause* to the list of clauses. Intuitively, the goal clause summarizes in a nutshell the *purpose in life* of an agent. When an agent becomes instantiated from its representative class at runtime, it executes the goal clause and upon whose completion the agent voluntarily terminates its existence. Oasis agents with a goal clause are said to be *active.* In the absence of a goal clause in its representative class, an instantiated agent lives forever to serve other agents and never dies. These agents are said to be *reactive.* All objects are *passive.*

3 Inheritance

A class specification header describes the inheritance relationship of a class. It either anchors a class firmly in some existing inheritance hierarchy or establishes the current class as the root of a new hierarchy. In other words, an Oasis class could either be descended from some parent class or be a root class itself in which case it has no parent. To be semantically meaningful, an object class that is not a root can only be descended from some previously specified object class. Similarly, a non-root agent class can only have another previously specified agent class as its parent. The header of a root class specification is introduced with the `class` keyword. The implication is that there could be multiple root classes in an Oasis program, each of which leads a homogeneous family of agent or object classes with shared commonalities. This is summarized in the following:

```
header          ::=  object-parent :: object-name
                  |  agent-parent :: agent-name
                  |  class object-name
                  |  class agent-name
```

An object class is *generic* in the sense that it could be parameterized with *type variables* representing *generic types*. The *scope* of a variable definition is the area of program text within which an occurence of the name refers to that definition. Type variables have a scope that extends beyond its defining class to include all of its descendent classes uniformly. For reasons of type consistency, a root object class defines the set of type variables once and for all for each of its descendent classes. In other words, the descendent classes are not allowed to override or extend the set of inherited type variables in any way. For readability purposes, each descendent object class specification is required to carry verbatim the full list of type variables as part of its object name in the specification header.

```
object-name     ::=  object-id < generics >
                  |  object-id
object-parent   ::=  object-id
object-id       ::=  ID
generics        ::=  generic , generics
                  |  generic
generic         ::=  generic-id
generic-id      ::=  $ID
```

Conditions specify names of condition variables used for internal synchronizations between multiple thread within an agent. Like type variables, conditions have a scope that extends beyond its defining class to include all of its descendent classes uniformly. For simplicity, a root object class defines the set of conditions once and for all for each of its descendent classes. In other words, the descendent classes are not allowed to extend the set of inherited conditions.

```
agent-name      ::=  agent-id [ conditions ]
                  |  agent-id
agent-parent    ::=  agent-id
agent-id        ::=  CID
conditions      ::=  condition conditions
                  |  condition
condition       ::=  condition-id
condition-id    ::=  $ID
```

4 Protection

Each attribute and method of a class can be independently assigned a protection. There are three levels of protection: *public*, *protected* and *private*. They are used for both visibility and access control. The Oasis compiler statically enforces the scoping rules dictated by these protection levels without incurring runtime overhead. Since Oasis does not differentiate between read and write accesses for attributes, an attribute is considered accessible if it is updatable. A method is accessible if it is callable. The protections are:

protection$_0$	::=	`public`
	|	`protected`
	|	`private`
	|	ε

The scoping rules of the Oasis language distinguishes between three different scopes: *class*, *descendents* and *world*, one nested within the next and in that order. If an attribute or a method lies within the scope of class, it is accessible only from within areas of text within the class definition. If an attribute or a method lies within the scope of descendents, then it is accessible from all of its descendent classes as well as from the defining class itself. Finally, if an attribute or a method lies within the scope of world, then it is accessible from all the classes. The following protection matrix illustrates the relationship between protection levels and scoping rules for Oasis as they relate to attributes and methods of objects and agents:

	object attribute	object method	agent attribute	agent method
public	world	world	descendents	world
protected	descendents	descendents	descendents	descendents
private	class	class	class	class

It should be noted that the scope assigned to a public agent attribute is descendents and not world as would be expected. This is a deliberate design decision. For purposes of mutual exclusions and synchronizations, agents are viewed as monitors. If public agent attributes were meant to be accessible by the world, then other agents will have a chance to violate the data integrity afforded by a monitor.

5 Constants

Constants have the scope of descendents and are visible from within its defining class as well as from all of its decendent classes. Constant lists look like:

constant-lists$_0$	::=	`constant` `:` *constant-lists*
	|	ε
constant-lists	::=	*constant-list constant-lists*
	|	*constant-list*
constant-list	::=	*type constants* `;`
constants	::=	*constant* `,` *constants*
	|	*constant*
constant	::=	*constant-id* `=` *expression*
	|	*constant-id*
constant-id	::=	*ID*

6 Attributes

Each attribute has a protection and a type associated with it. The protection, if not explicitly specified, defaults to that currently in effect. The scope of a protection specifier includes the remaining text region of the class specification plus those of its descendents unless overridden by another protection specifier. The protection is initialized to `protected` at the start of the root class before it is overridden. Attributes can have explicitly specified initial values, otherwise they assume an implicit default value. In the case of integers and reals, the default is zero; in the case of characters, the default is the null character. All others, i.e. arrays, lists, objects and agents, default to `nil`'s. Attribute lists look like:

attribute-lists$_0$	::=	`attribute` : *attribute-lists*
	|	ε
attribute-lists	::=	*attribute-list attribute-lists*
	|	*attribute-list*
attribute-list	::=	*protection*$_0$ *type attributes* ;
attributes	::=	*attribute* , *attributes*
	|	*attribute*
attribute	::=	*attribute-id* = *rvalue*
	|	*attribute-id*
attribute-id	::=	*ID*

7 Methods

Each method has a protection and a list of arguments associated with it. The number of arguments, or arity, of a method is fixed. An argument is specified to operate in one of two modes: as an input argument or as an output argument. An argument identifier by itself denotes an input argument. With a preceding question mark, an argument identifer denotes an output argument. For each method specified, there is a corresponding list of clauses in the implementation section that describes the sequence of actions associated with an invocation of the method.

methods$_0$	::=	`method` : *methods*
	|	ε
methods	::=	*method methods*
	|	*method*
method	::=	*protection*$_0$ *method-id* (*argument-lists*$_0$) .
argument-lists$_0$	::=	*argument-lists*
	|	ε
argument-lists	::=	*argument-list* ; *argument-lists*
	|	*argument-list*
argument-list	::=	*type arguments*
arguments	::=	*argument* , *arguments*
	|	*argument*
argument	::=	*argument-out*
	|	*argument-in*
argument-out	::=	? *argument-id*
argument-in	::=	*argument-id*
argument-id	::=	*CID*

8 Types

The type universe is made up of: (1) basic types, consisting of integers, reals and characters, (2) structured types, consisting of lists and arrays, and (3) user-defined types, consisting of classes of objects and agents.

types	::=	*type* , *types*
	|	*type*
type	::=	*simple-type*
	|	*complex-type*
	|	*generic-type*
simple-type	::=	`int`
	|	`real`
	|	`char`
complex-type	::=	*type* `*`
	|	*type* `[` *dimensions* `]`

```
                 |  object-id < types >
                 |  object-id
                 |  agent-id
generic-type    ::= generic-id
dimensions      ::= dimension , dimensions
                 |  dimension
dimension       ::= INTEGER
                 |  _
```

9 Clauses

Every clause has a head, an optional body and an optional tail. A dotted turnstile (`:-`) precedes the body and separates it from the head. A solid turnstile (`|-`) precedes the tail and separates it from the body or, in case the body is missing, from the head. All the clauses with the same method identifier in the clause head are said to be in the same clause group. Each clause group implements a method.

Clause heads serve two purposes: as message templates for matching input parameters of a call, and for constructing return parameters upon successful completion of the call. The parameter matching proceeds in textual order, from top-to-bottom and from left-to-right. The body or tail of a clause consists of clusters of one or more messages. If there are more than one messages within each cluster, then they are meant to be sent as concurrent remote procedure calls to other agents. All the messages within a cluster must have successfully received replies from their respective calls before execution can proceed. If there is some method call within a cluster that has returned with a failure, then the program control flow will depend on whether the cluster is part of a clause body or a clause tail. In the case that the failed cluster is part of a clause tail, the method call returns with failure, even if there are other clauses within the same clause group that have not been tried. If the failed cluster is part of a clause body, then the next clause in the clause group, if it exists, is tried. Otherwise, there is no more clauses in the clause group and the method call returns with failure.

```
clauses_0       ::= clauses
                 |  ε
clauses         ::= clause clauses
                 |  clause
clause          ::= head :- body |- tail .
                 |  head :- body .
                 |  head |- tail .
                 |  head .
goal_0          ::= goal
                 |  ε
goal            ::= ?- body .
head            ::= method-id ( parameters_0 )
body            ::= messages ; body
                 |  messages
tail            ::= messages ; tail
                 |  messages
```

10 Messages

A method call can be either dynamically or statically bound. Dynamic method calls are marked by the presence of an exclamation mark (`!`) which is interposed between the method call and its destination agent or object. A simple method call is always statically bound; as it is directed to itself by default. In case there are more than one implementation of a method as a result of overriding, a method call is allowed to specify the version to use by preceding the method call with the name of an agent or object class followed by a double colon (::). Each method call activates a number of clauses in the same clause group in the order of their textual occurrences.

```
messages        ::=  message , messages
                 |   message
                 |   invocation
                 |   matching
                 |   comparison
message         ::=  destination ! send
invocation      ::=  object-id :: send
                 |   agent-id :: send
                 |   send
matching        ::=  ( type ) value = rvalue
                 |   global = rvalue
comparison      ::=  expression == expression
                 |   expression <> expression
                 |   expression <  expression
                 |   expression <= expression
                 |   expression >  expression
                 |   expression >= expression
send            ::=  method-id ( parameters_0 )
```

11 Parameters

```
parameters_0    ::=  parameters
                 |   ε
parameters      ::=  parameter , parameters
                 |   parameter
parameter       ::=  value
destination     ::=  condition
                 |   reference
                 |   handle
                 |   self
```

12 Values

```
value           ::=  lvalue
                 |   rvalue
                 |   _
lvalue          ::=  reference' : value
                 |   reference'
rvalue          ::=  instance
                 |   expression
                 |   handle
instance        ::=  list
                 |   array
                 |   object
                 |   agent
```

13 Lists

```
list            ::=  [ elements | rest ]
                 |   [ elements ]
```

```
elements        ::=  element , elements
                 |   element
element         ::=  value
rest            ::=  value
```

14 Arrays

```
array           ::=  $ [ sizes ] { items0 }
                 |   $ [ sizes ]
                 |   STRING
sizes           ::=  size , sizes
                 |   size
size            ::=  value
items0          ::=  items
                 |   ε
items           ::=  item , items
                 |   item
item            ::=  { items0 }
                 |   value
```

15 Objects and Agents

```
object          ::=  object-id { properties0 }
agent           ::=  agent-id { properties0 } @ site
                 |   agent-id { properties0 }
handle          ::=  agent-type @ site
properties0     ::=  properties
                 |   ε
properties      ::=  property , properties
                 |   property
property        ::=  value
site            ::=  machine : port
                 |   machine
machine         ::=  INTERNET
port            ::=  INTEGER
```

16 Expressions

The arithmetic operators `+`, `-`, `*`, `/`, and `%` group left-to-right. The multiplicative operators `*`, `/`, and `%` have higher precedence than the additive operators `+` and `-`. Automatic type coercions are performed when the two operands are of unequal types.

```
expression      ::=  expression + term
                 |   expression - term
                 |   term
term            ::=  term * unary
                 |   term / unary
                 |   term % unary
                 |   unary
unary           ::=  - factor
                 |   factor
factor          ::=  function-id ( expression )
                 |   ( expression )
```

```
                    |   reference
                    |   literal
function-id        ::=  ID
literal            ::=  INTEGER
                    |   FLOAT
                    |   CHARACTER
                    |   [ ]
                    |   nil
```

17 References

```
reference          ::=  global
                    |   local
                    |   self
global             ::=  attribute-id accessors
                    |   attribute-id
                    |   local-id accessors
local              ::=  local-id
local-id           ::=  CID
accessors          ::=  accessor accessors
                    |   accessor
accessor           ::=  .attribute-id
                    |   [ indices ]
indices            ::=  index , indices
                    |   index
index              ::=  expression
```