TOM Language Reference Manual

DRAFT version 0.99 *DRAFT* for TOM version 1.0
$Id: tlrm.texi,v 1.1 1998/01/12 22:45:52 tiggr Exp $

by Pieter J. Schoenmakers

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DE-

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along
with this program; if not, write to the Free Software Foundation, Inc.,
59 Temple Place - Suite 330, Boston, MA 02111-1307, USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands `show w` and `show c` should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than `show w` and `show c`; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Bugs

There are no known bugs in this manual, unless you count every missing feature as one. Bug and missing feature reports, suggestions, and other remarks concerning this document can be sent to the author, Pieter J. Schoenmakers `<tiggr@ics.ele.tue.nl>`.

Preface

This document describes the programming language TOM. It does not explain what programming is or what it is all about. It does not teach you how to program, not in any language nor in TOM. It does not provide extensive rationale for the whole design of TOM; only details are addressed. In short: this document is the language reference manual.

In addition to a definition of the language, some of the topics are concerned with the TOM implementation as written by yours truly. Examples of such topics are the garbage collector, the tools, and the C interface to TOM and the runtime.

While reading this document, knowledge of other programming languages, such as Objective-C, C++, C, Common Lisp, Smalltalk, and Eiffel will be helpful.

Pieter J. Schoenmakers `<tiggr@ics.ele.tue.nl>`

1 Introduction

TOM is an object oriented programming language. The following example shows the famous 'hello world' program, this time written in TOM.

```
 1  implementation class HelloWorld: stdio
 2
 3  int
 4    main Array arguments
 5  {
 6    [[out print "Hello, world!"] nl];
 7
 8    = 0;
 9  }
10
11  end;
12
13 implementation instance HelloWorld end;
```

In line 1, a class is defined with the name `HelloWorld`. It inherits from `stdio`, implying that anything defined by the `stdio` class is directly available.

Line 2 is empty. That's ok, as empty lines have no significance.

From line 3 to 9, a method is defined. This method is named `main`; it accepts one argument, named `arguments`, the type of which is `Array`; it returns an `int`.

The `main` method, when implemented by a class, is special as it is the method invoked when the program is started. The `arguments` argument is an array of strings holding the command line arguments to the program. `main` returns an `int` which will be the exit value of the program.

In line 6, two methods are invoked. Invoking a method is the same as sending a message. The first invocation send a message `print` to the object known as `out`. `out` is a variable inherited from `stdio`; it is the 'standard output' stream of the program, known as `stdout` in C. The `print` message has one argument, a string containing 'Hello, world!'.

By convention, every `print` method implemented by a stream returns the receiver, so that the invocation itself can be used as the receiver of another message to the stream. In this case, the argument-less message `nl` is sent. This causes a new line to be output to the standard output stream, and that stream to be flushed to make sure any buffered contents will be displayed on the user's terminal. `nl` also returns the receiver, but that value is ignored.

In line 8, the assignment of `0` with an empty left-hand-side sets the value to be returned from this method. It returns `0`, a value which indicates success. Note that this form of the return value assignment does not actually cause an immediate return from this method (it would if the `=` were replaced by `return`).

By the `end` in line 11, the definition of the `HelloWorld` class is finished. In line 13, the instances of the `HelloWorld` class are defined. Since we never allocate `HelloWorld` instances, this definition is rather empty.

So much for an easy introduction. The rest of this document will be more of a *reference* manual, with not that many elaborate examples.

1.1 Syntax

Where applicable, syntax of the construct under discussion is presented. The syntax of the rules is, written in itself:

```
rule:
          non_terminal ':' alternatives ';' ;
alternatives:
          alternative | alternative '|' alternatives ;
alternative:
          element* ;
element:
          non_terminal
        | terminal
        | element '*'
        | element '+'
        | '[' alternatives ']'
        | '(' alternatives ')'
        ;
non_terminal:
          non-space-characters ;
terminal:
          open-quote characters close-quote ;
```

The definitions of `non-space-characters`, `characters`, `open-quote`, and `close-quote` are left as an excercise to the reader.

2 Units

A unit is the TOM equivalent of a library or program. Every separate library is a unit, and every program is a unit. A unit consists of a number of source files, each containing one or more classes or extensions. A unit can depend on other units, like in C, where the Motif library depends on the X11 library, and like every C program depends on the C standard library.

As an example, the following unit file defines a unit `myprog`. It depends on the standard environment unit `tom` (see Section 8.1 [unit tom], page 61) and consists of a single source file `file1` which contains a class named `Hello` and an extension of the `OutputStream` class. Since all TOM source files have the extension `.t`, the actual file of this unit will be called `file1.t`.

```
unit myprog

uses tom;

file "file1"
{
  class Hello;
  extension OutputStream (myprogExtension);
}

end;
```

A unit is defined by its unit file. Normally, the unit file is created by the interface generator `gi` (see Section 9.2 [gi], page 64). For a unit `myprog`, the unit file will be `myprog.u`.

The TOM compiler uses the unit files to know where to look for definitions of a class. As a consequence, there is no need for TOM sources to contain `#include`, `import`, `use` or other manual dependency indications: If a TOM source uses a class `Foo`, the compiler will automatically read that class' definition.

Facts

- Unit dependency is a transitive relation and implicit dependencies need not be noted explicitly.
- Each unit defines a namespace. Class names in different units can be identical; they will not clash since they reside in different namespaces. The unit names of all units in a given project must be different.

- the class definition and the instance definition of an extension must must reside in the same file.

Syntax

The syntax defined in this chapter is the syntax of unit files. Unit file syntax is different from (and in fact unrelated to) the TOM syntax presented elsewhere throughout this document.

```
unit_file:
          unit
        ;

unit:
          'unit' unit_name unit_decl_entry* 'end' ';'
        ;

unit_name:
          identifier
        ;

unit_decl_entry:
          'uses' identifier_list ';'
        | unit_file_indication
          '{' (unit_plain_decl_entry ';')+ '}'
        | unit_decorated_decl_entry ';'
        ;
```

Each `unit_decl_entry` either declares the use of other units, defines the (partial) contents of a file or declares the existence of a class or extension.

```
unit_file_indication:
          'file' STRING_CST
        ;
```

The `STRING_CST` denotes the name of a file; such a filename excludes an extension; the default extension `.t` is always used.

```
unit_decorated_decl_entry:
          unit_plain_decl_entry [unit_file_indication]
        ;
```

A `unit_decorated_decl_entry` is a `unit_plain_decl_entry` optionally followed by an indication of the filename. If the filename is not provided, a suitable default is assumed. This is the name of the class for a class definition (the main extension thereof, really); it is the name of the extension for an extension definition.

```
unit_plain_decl_entry:
          'class' IDENTIFIER [unit_posing_decl]
        | 'extension' class_name '(' IDENTIFIER ')'
          [unit_posing_decl]
        ;
```

An entry either defines a class, or it defines an extension of another class, which could reside in the current unit.

[Note: This syntax is not consistent. Instanceless classes and classless instances should be possible. An extension is nothing more than a class with a non-empty extension name. End note.]

```
unit_posing_decl:
          'posing' class_name
        ;
```

The `class_name` is the name of an existing class, see Chapter 3 [Objects], page 19.

A `unit_posing_decl` declares that the definition of the enclosing class or extension declares a posing superclass. This information is necessary in the unit file, as the compiler deduces from the unit files which files need to be read to obtain the full declaration of a class, and for the full visible view of a class that is being posed, the posing subclasses must be read as well.

3 Objects, classes and instances

Objects are elementary building blocks. TOM's notion of objects is obviously inspired by Smalltalk, in that for every defined class, a *class object*, or factory object as it is sometimes called, exists. A class object describes the *state* (instance variables or member variables) and *behaviour* (instance methods or member functions) of its instances. Conceptually, each class object is the sole instance of its *meta class* object, the meta class object describing the state and behaviour of the class object. All meta class objects share the same meta-meta class object.

Classes and instances are distinct entities. A class of objects is only fully defined by both the definition of the class and the definition of the instances.

It is possible for instances not to carry any state; such instances can not actually be allocated, but their definition can be inherited by other objects, even class objects. A class with stateless instance is a *behavioural class*.

All classes with allocatable instances must inherit, directly or indirectly, from the `State` class: It is the `State` class that defines the instance and class variables necessary for the runtime system to function. Most notably, the `State` instance introduces the `isa` instance variable. For every instance, this variable points to the class object describing the instance.

A class or instance definition can declare methods to be *deferred*. A class having deferred methods, either because of its own definition or because of inheritance, is said to be a deferred class. Instances of a deferred class can not be allocated; only instances of a subclass which actually defines all deferred methods can be allocated.

A behavioural instance with only deferred methods (i.e. without providing any method implementation) resembles an Objective-C protocol, a Java interface, and a C++ signature. There are, however, elementary differences

- a behavioural instance can provide method implementations,
- a behavioural instance can be inherited by a class object,
- behavioural instances are inherent to the design of the language; they are not an afterthought extension.

A class can *pose* as another class. A posing class is inserted into the class hierarchy between the posed class and any subclass or client: all references to the posed class actually refer to the posing class (except for references from the posing class itself). Posing is especially useful in the context of library units over which the programmer has no source control.

With the semantics of extensions, posing might seem a bit too much like using a canon to shoot a mosquito. All posing really adds is the ability to override methods while still being able to refer to the original. The real advantage of posing is that referring to the original method implementation is as simple as messaging super.

Facts

- The class of a behavioural instance may not define or inherit non-static state.

  (All class objects implicitly inherit from the `tom.State` class object for its ability to describe instances. This implicit inheritance is excluded from the above fact.)
- The `tom.State` meta class object serves as the meta-meta class object.

Syntax

```
class_name:
          IDENTIFIER
        | unit_name '.' IDENTIFIER
        ;
```

A class name is either an identifier, or an identifier qualified by the name of the unit in which it is defined (see Chapter 2 [Units], page 15 for an explanation of units).

An unqualified class name must refer to the only class name visible or it is an error. A class `Foo` in some unit `myprog` makes classes by the same name, in units upon which the unit `myprog` depends, invisble.

The type of instances of a class is denoted as the `class_name`; Shifted in the direction of the `class` make it denote the class object.

```
/* my_foo references an instance of Foo
   or an instance of a subclass.  */
Foo my_foo;

/* foo_class references the Foo class
   or a subclass thereof.  */
class (Foo) foo_class;
```

3.1 Inheritance

A class can be similar to an already existing class, and it would be a waste to have to rewrite the behaviour already provided by that existing class. The new class can inherit the behaviour, and state, from an already existing class. The new class will then be a subclass of the other class, which will be the superclass of its subclass.

Inheritance is, by definition, public. This implies that any method declared in a superclass is available in exactly the same way to users of the inheriting class, as it is to users of the inherited class. Private inheritance, like it is available in C++, is not available in TOM, as the concept implies a contamination of the 'is-a' and 'part-of' hierarchies. Anything achievable through private inheritance can also be obtained by adding an instance variable of the proper type instead of privately inheriting that type. An advantage of the private inheritance is the availability of direct access to the inherited class' instance variables and the direct association of the instance part of the inherited class with the whole object of the subclass. On the other hand, the fact that a class can only be (privately) inherited once is only one the reasons private inheritance is considered a bad idea.

Inheritance is not restricted to a single superclass. A class may inherit any number of classes. If two inherited classes share a superclass, the state carried by that superclass will, by definition, appear in the new class exactly once. Thus, repeated inheritance is always shared inheritance.

Method clashes due to inheritance, when two distinct superclasses both implement the same method, must be resolved by the new class providing its own definition of the method, or by the new class indicating which inherited implementation is to be preferred.

[Note: Currently, the tools (compiler and resolver) do not yet provide the latter option. End note.]

If a class declares deferred methods, the class itself is said to be a deferred class. If a class is a direct subclass of a deferred class, and it does not provide an implementation for all inherited deferred methods, either explicitly or through inheritance, the new class is a deferred class too.

Syntax

A class or instance is defined or declared by an appropriate `object_def`.

```
object_def:
          ( 'implementation' | 'interface' )
          ( 'instance' | 'class' ) class_name [extension]
          [supers] [object_variables] method* 'end'
        ;
```

A class is only fully declared if the `interface` of both its `class` and its `instance` have been seen. A class is fully defined by the `implementation` of its `class` and `instance`. The implementation is what a programmer writes; the interface is usually generated by `gi` (see Section 9.2 [gi], page 64).

When compiling TOM code, the compiler reads only the implementation of the classes to be compiled (which, per run of the compiler, all reside in the same file). All other classes it needs to know about are declared by reading the interface files. Before actually compiling an implementation, the corresponding interface is read. Any discrepancy between the interface and the implementation is an error: state declared in the interface must exactly match state in the implementation; similarly, any method defined in the implementation must have been declared in the interface, and vice versa.

The `extension`, if present indicates that the current interface or implementation does not actually concern the class' main extension, but the extension named in the `extension` (see Section 3.3 [Extensions], page 24).

```
supers:
          ':' super_list
        ;

super_list:
          super_indication [',' super_list]
        ;

super_indication:
          ['posing'] class_name
        | 'instance' '(' class_name ')'
        ;
```

If the `super_indication` is a `class_name` the inheritance applies to both the class and the instance of the object currently being defined or declared. It is customary to indicate this kind of plain inheritance with the class, not the instance. If `posing` is present, the current class poses the indicated super class.

The inheritance of only an instance (the second case), applies only to the current object (either instance or class) being defined or declared. In this case, the instance being inherited must be a behavioural instance.

```
object_variables:
          ‘{’ object_variable+ ‘}’
        ;
```

The `object_variables` define the state introduced by a class or instance (or an extension thereof).

```
object_variable:
          [‘local’] [‘static’] object_var_qualifier*
          entity_type identifier_list ‘;’
        ;
```

An `object_variable` introduces the variables named in the `identifier_list`. Every variable has the type denoted by the `entity_type`.

The `static` may only be present in the definition of a class. It denotes that the variables introduced are shared between this class and all its subclasses. If omitted, every class will have its own copy of these variables, just like every instance always has its own copy of the instance variables defined for it.

The `local` qualifier may only be present on a `static` variable. It denotes that every thread in the running program will have its own value for this variable. An example of its use is the `current Thread` class variables, which is declared like this:

```
local static public instance (id) current;
```

```
object_var_qualifier:
          ‘redeclare’
        ;
```

An object variable can be indicated to be a redeclaration of an object variable inherited from a superclass. Redeclaration is only allowed to indicate that the variable will actually be a reference to an object whose class is a subclass of the class originally being the type of the variable. Thus, changing an object variable's type from, for instance, `char` to `long` is not allowed.

```
method:
          method_declaration
        | method_definition
        ;
```

A `method_declaration` is only allowed in an interface; an implementation will only contain method definitions. (Note that some identical expansions of a `method` are a declaration in the context of an interface a definition in an implementation. The reason for this is that interfaces are generated from implementations, and they must be generatable from *something*.)

3.2 Elementary classes

A few classes are elementary to TOM. Some of these are defined in the TOM unit (see Section 8.1 [unit tom], page 61), others are present in the unit built into the compiler, called `_builtin_`. (Note that it is impossible to refer to this unit directly, as TOM does not allow an identifier to start with an underscore. This is intentional.)

`_builtin_.Top`
: This is the implicit superclass of all defined classes. It contains the definitions of the builtin, statically bound, operators (see Section 5.6.6 [Operators], page 47).

`_builtin_.Any`
: This is the implicit subclass of all defined classes. If a method can return any object, it should declare the return type to be `Any`.

  In this context, all defined classes are those classes whose location in the inheritance graph is known, i.e. those classes of which the declaration has been read.

`tom.State`
: `State` is the superclass of all classes with allocatable instances; it is the state introduced by `State` that allows the dynamic binding and all the other services offered by the runtime to actually function.

`tom.All`
: The instance `All` defines methods needed for both classes and instances, such as `perform :`. It is inherited by both the `State` class and the `State` instance. It is considered good practice for any behavioural instance which does not inherit anything, to inherit `instance (All)`. Furthermore, if a method accepts all objects as the value for a particular argument, the type of the argument should be `All`.

3.3 Extensions

A class does not need to be defined by one monolithic definition; its definition can be split into multiple so-called extensions. A class definition always has at least one extension; this nameless extension is called the main extension. All other extensions have a name which must be unique within the class. Extensions are not limited by unit bounds: it is *legal* for a unit to define an extension of a class defined in any unit known to the compiler.

An extension can introduce new superclasses, behaviour and state, and it can replace methods previously defined in other extensions of its class

[Note: 'previously' implies an ordering of extensions. This ordering is undefined except that the main extension precedes all other extensions. End note.]

In the context of dynamic loading, the addition of state (either directly or by inheritance) is only allowed if no objects which are to carry that state have been allocated yet. Since all class objects are pre-allocated, the addition of non-static state to class objects is not possible.

Syntax

```
extension:
          'extension' identifier
        ;
```

3.4 Accessibility

An instance can directly access its own instance variables and the variables accessible to its class. A class can directly access its own class variables, which it either declared or inherited, and the `static` variables declared by any of its superclasses. All other direct variable accesses are not possible: such forms of state must be accessed through methods. Thus, state of objects other than `self` and `isa` (that is the `isa` of `self`, `self->isa` in C syntax) is not directly accessible.

3.5 Encapsulation

At compile time, a strict notion of protection and encapsulation is maintained. At runtime however, enough facilities exist to effectively void any protection. (This is intentional as it allows for very flexible dynamic behaviour.)

All methods and object variables are associated with a protection level.

```
protection_qualifier:
          'private'
        | 'protected'
        | 'public'
        ;
```

The protection level defines availability on a class basis; it does not discern between different extensions of the class. Given that an extension can be defined in a unit different from the main extension, protection is not hampered by unit boundaries. (This is intentional.)

`private` The item is only available to the current class.

`protected`
: The item is available to the current class and its subclasses. This is the default protection of instance and class variables.

`public` The item is available to any class. This is the default protection of methods.

An object variable declared `public` is not actually directly accessible from outside the current class (see Section 3.4 [Accessibility], page 25), but causes an accessor method to be defined for it. If the variable is declared to be `mutable`, usually in addition to the variable being `public`, a modifier method is defined for it.

For a `public mutable int foo_bar`, the following two methods are defined:

```
/* Accessor.  */
int
  foo_bar
{
  = foo_bar;
}

/* Modifier.  */
void
  set_foo_bar int value
{
  foo_bar = value;
}
```

4 Types

TOM has three kinds of types: basic types, objects and tuples. Furthermore, there is one special type, `void`, and there are two special type indications: `dynamic` indicates that the actual type will be dynamically checked; and `id` which, in denotes an *actual* object instead of the containing declaring object.

Syntax

```
entity_type:
          basic_type
        | tuple_type
        | object_type
        ;
```

An `entity_type` is the possible type of an entity, such as an object variable, method argument or local variable.

```
argument_type:
          'dynamic'
        | entity_type
        ;
```

In addition to the usual types, an argument can have the `dynamic` type. The type actually passed will be encoded in the selector of the method being invoked. It is the responsibility of that method to retrieve the correct types as indicated by the selector (see Section 10.3 [Selectors], page 70).

```
return_type:
          'void'
        | argument_type
        ;
```

The type of value returned by a method can be anything that can be the type of an argument, plus `void`, indicating that the method will not return any value.

4.1 Basic types

```
basic_type:
          'byte' | 'char' | 'int' | 'long' | 'float' | 'double'
```

```
        | ‘boolean’ | ‘pointer’ | ‘selector’
        ;
```

4.1.1 Numeric types

TOM has two kinds of numeric types: integer and floating point. The integer numeric types are:

`byte`	unsigned 8 bit integer number
`char`	unsigned 16 bit integer number
`int`	signed 32 bit integer number
`long`	signed 64 bit integer number

The floating point numeric types are:

`float`	single precision floating point number
`double`	double precision floating point number

In places where a numeric type is expected, a narrower numeric type of the same kind is accepted and implicitly converted. Thus, a byte is acceptable as a char is acceptable as an int is acceptable as a long; and a float is acceptable as a double.

The default value for the numeric types is zero.

4.1.2 The boolean type

The `boolean` type is used for truthness values. It is extensively used in conditional constructs (see Section 5.6.4 [Conditionals and loops], page 45). The default value for the boolean type is falseness. The `instance tom.All` defines the following constants, each with the `boolean` type.

```
const TRUE = !0;
const FALSE = !1;
const YES = TRUE;
const NO = FALSE;
```

4.1.3 The pointer type

The `pointer` type is a rather abstract type. Values of the `pointer` type can not be operated upon; they can only be passed around; their primary use is for the implementation of arrays and for debugging purposes. The default value for the pointer type is the invalid pointer (`NULL` in C). Note that, within TOM proper, it is impossible to refer to the constant null pointer.

4.1.4 The selector type

A selector is an abstract entity holding a name and typing information. A method invocation, also know as 'sending a message to an object', actually is the invocation of some behaviour identified by the object receiving the message, and the name of the message: the selector. The second part of a message are the arguments to the method. A selector is a name and typing information on the arguments carried by a message and return value expected by the sender of the message. Every method implementation has, as the second implicit argument the selector used to invoke the method's behaviour.

The default selector value denotes the non-existing selector. The only operations defined on `selector` typed values are equality comparisons. See Section 10.3 [Selectors], page 70 for information on selector names.

[Note: Currently, the compiler defines selector equality as equality of the pointers to the selector structs (in the output language). Obviously, in the context of dynamic loading, these are not the desired semantics. Before those can be implemented it has to be defined when two selectors are considered to be equal and when they are not. End note.]

4.2 The void type

The `void` type is a special type: it indicates the absence of a value. Its most profound use is in typing the return value from methods; another use is as an expression where an expression is not needed but also not allowed. `void` is the only one void-typed value; no operations can be performed on `void`; and there is no default value for the `void` type.

4.3 The dynamic type

A method which is implemented outside TOM (say, in C) can have a dynamic return type and dynamic argument types. A dynamic type implies anything can be passed and will be accepted. The type of the value actually passed to the method is encoded in the selector, which the method receives as the implicit second argument (after `self`). Similarly, the method implementation can deduce the expected return type from actual selector. The dynamic type is used by, for instance, the `perform` : method, which is defined as

```
extern dynamic
  perform selector sel
        : Array arguments = nil;
```

If `perform` : would be invoked as

```
int a, b, c;

(a, b, c) = [foo perform bar]
```

then it would be a (fatal) runtime error if the selector denoted by `bar` did not return a tuple of three integers.

[Note: Currently, the possibilities of the `dynamic` type are somewhat limited. For instance, the compiler insists on being able to deduce the actual type used for something declared dynamic. Thus, for example, the following doesn't work, even though it should, really:

```
[foo setValue [foo valueOfVariableNamed "bar"]
     ofVariableNamed "bar"]
```

End note.]

4.4 Object type

Objects are the only way in TOM to create an aggregate value—there is no such thing as a struct.

- All objects—classes and instances—share the same *type*. Obviously, the compiler can tell a difference between a `Number` and a `ByteArray`, so not all objects are of the same *kind*.
- A variable with an object type actually is a reference to an object. All objects are allocated from a heap.

- In the rest of this document, often the type of an expression or entity is referred to. In most cases, this means the type of the value, or the kind of object if the type is the object reference type.

The default value of an object-typed variable is the invalid reference, `nil`. The type of `nil` is `_builtin_.Any`.

```
object_type:
          class_name
        | 'id'
        | 'class' '(' object_type ')'
        | 'instance' '(' object_type ')'
        ;
```

The plain `object_type` is a `class_name`; this indicates the *instance* of the indicated class. `id` indicates the type of the actual receiver (see Section 4.5 [The id type], page 31). The variation with a 'class' or 'instance' shifts the meta level into the specified direction. Examples:

`Foo` A reference to an instance of `Foo` (or an instance of a subclass of `Foo`).

`class (Foo)`
: A reference to the `Foo` class object (or the class object of a subclass of `Foo`).

`instance (id)`
: A reference to an instance of the current receiver. Obviously, it is an error if the current receiver itself is an instance, since instances do not have instances.

`class (id)`
: A reference to the class object of the current receiver. For the current receiver being an instance, it denotes the class object; for a class it denotes the meta class object. For meta class objects, it denotes the meta class object of the `State` class.

As the receiver of a method invocation, a `class_name` denotes the class object (see Section 5.6.1 [Method invocations], page 43).

4.5 The id type

The `id` type is not an actual type. In the context of an object definition, `id` is identical to the current class or instance being defined, in contrast with a normal type that would indicate the class containing the declaration; in the context of an invocation of a method involving `id` typed arguments or return values, `id` denotes the actual receiver of the method.

For example, if a class `Foo` declares the following method:

```
id self;
```

then the type of the expression

```
{ Foo a; [a self] }
```

is `Foo`. If there exists a class `Bar` which inherits from `Foo`, then the type of the expression

```
{ Bar a; [a self] }
```

will be `Bar`, since it is a `Bar` that was the actual receiver of the `self` message (as far as the compiler knows).

The meta level of `id` can be shifted towards instances or classes, as can be seen in the following two method declarations from the `State` class.

```
<doc> Return a newly allocated instance
    of the receiving class.  </doc>
instance (id)
  alloc;

<doc> Return the class of the receiving object.  </doc>
class (id)
  class;
```

Thus, irrespective of the class of which the `alloc` method is invoked, the compiler knows that the object returned by `alloc` is an instance of the receiving class.

Similarly, the `isa` instance variable is declared by the `State` instance as

```
class (id) isa;
```

Thus, in the context of a subclass of `State`, the `isa` has the type of the actual class object, not just the `State` class.

4.6 Tuple types

A tuple is a hotch-botch of values. For example, `(123, 3.1415)` is a tuple, and its type is `(int, float)`.

The tuple type is not a first-class type: it is impossible to declare a variable with a tuple type. A single element tuple type actually is the type of the element. The default value of a tuple type is a tuple with as elements the default values of the tuple type's elements. Tuples can nest: `((1, 2), (3, 4))` is a valid tuple, the type of which is `((int, int), (int, int))`.

The `dynamic` type can not be an element of a tuple type. All other types are allowed.

The primary use of tuple types is in passing values to or from a method. The following example declares a method `divmod` which accepts one argument being a tuple of two integers and which returns another tuple of two integers:

```
<doc> Return (a / b, a % b).  </doc>
(int, int)
  divmod (int, int) (a, b);
```

Another use of tuples is in shorthands such as simultaneous assignments:

```
int a, b; ...; (a, b) = (b, a)
```

The evaluation of tuple elements is defined to be from left to right. Thus, the result of the following expression

```
{ int i = 0; (++i, ++i) }
```

is defined and equal to `(1, 2)`.

Syntax

```
tuple_type:
          `(' entity_type_list `)'
        ;

entity_type_list:
          entity_type [`,' entity_type_list]
        ;
```

The type of an element of a tuple can be neither `dynamic` nor `void`.

5 Structure

This chapter explains the contents of TOM interface and implementation files.

5.1 Comments

TOM provides three basic comment syntaxes. The most basic kind of comment uses C syntax.

```
/* This is a plain, C-style comment.  */
```

The second kind of comments uses the `//` syntax as used in C++.

```
// This is a C++-style comment.
```

Unlike the C-style comments, the C++-style comments do have a meaning to the compiler: it will issue a warning about the comment. The intended use of `//` comments is to mark places in the code which need revisiting soon, since something is very wrong with it and the comment writer did not have the time or capabilities to fix it. The `//` comment replaces the common use of `XXX`, `???`, and similar marks in regular comments used by a lot of programmers writing code, or people writing in general.

The third style of TOM comments are comments which can have a semantical meaning to programs manipulating TOM sources; they have no meaning to the compiler. These comments are called documentation comments, or documentation for short. Example:

```
<doc> This is a documentation comment.  </doc>
```

A documentation comment starts with a `<foo>` and ends with `</foo>`, where the `foo` can be any, even empty, combination of alphanumeric characters, including the underscore. Documentation comments with the same tag do not nest.

5.2 Method declarations

A method declaration declares the existence of a method for the current object being defined. Declarations are only possible in an `interface`. The `implementation` contains only method definitions (see Section 5.4 [Method definitions], page 39).

```
method_declaration:
          shared_method_part ';'
        ;

shared_method_part:
          method_decl [pre_condition] [post_condition]
        ;

method_decl:
          return_type method_qualifier* method_name_part
        | return_type method_qualifier* method_decl_part+
        ;
```

In the first rule of a `method_decl`, a method without any arguments is declared. In the second case, the method does have arguments.

```
method_name_part:
          [identifier | ...] [:]
        ;
```

A `method_name_part` can be an identifier or any of the TOM keywords, except `post` and `pre`. It is optionally suffixed by a colon. A colon itself also is a valid `method_name_part`. (Usually, by convention, any `method_name_part` containing a colon indicates an optional method part for which' arguments a default value is supplied in the declaration.)

```
method_qualifier:
          protection_qualifier
        | redo_qualifier
        | 'deferred'
        ;
```

If `deferred`, the method is not implemented by the current object. The protection qualifiers (see Section 3.5 [Encapsulation], page 25) define the availability of this method to the current class, its subclasses and the other classes.

```
redo_qualifier:
          'redeclare'
        | 'redefine'
        ;
```

If a class redefines a method implemented by a super class, it must declare the method as being `redefined`.

[Note: `redeclare` is meant to allow a subclass to indicate different object kinds for the arguments to a method implemented by a superclass. This is not yet implemented, its precise semantics have not yet been defined and its usefulness is being questioned. End note.]

```
method_decl_part:
          method_name_part argument_type argument_name
          [‘=’ expression]
        ;
```

If the `expression` is provided it denotes a default value for the arguments mentioned in the `argument_name`. Provision of such a default implies that this part of the method name is optional in the invocation of this method. Note that this is fully handled by the compiler: an invocation of a method specifying not all the arguments can not be discerned from an invocation specifying all default values for the arguments.

The `expression` must be evaluable in every context where it is used. It is, however, not enforced by the compiler that the expression is evaluable in every context where it can be used. [Note: This is a bug. End note.]

```
argument_name:
          identifier
        | ‘(’ identifier_list ‘)’
        ;
```

If the `argument_type` is not a tuple, the `argument_name` should be an identifier. For a tuple type, the `argument_name` should be a parenthesized identifier list, with the same number of elements as the tuple type.

[Note: This syntax rule does not allow an element of a tuple-typed argument to be a tuple. That is wrong, but currently how it is implemented. End note.]

Example

Given the following method declaration

```
int
   foo int a
  bar: (int, int) (b, c) = (0, 1);
```

which has one mandatory argument, `a`, and one optional argument, `(b, c)`. Given a receiver `rcv` of the proper type, this method can be invoked in different ways:

```
[rcv foo 1];

[rcv foo 2 bar: (3, 4)]
```

It is considered good practice to use a colon suffix for those nameparts of a method that are optional.

Since optional method parts are a purely compile time affair, if the object defining this `foo bar:` method is also to respond to a `foo` message, the object must explicitly define that method:

```
<doc> Respond to 'foo' in addition to 'foo bar:'.  </doc>
int
  foo int a
{
  = [self foo a bar: (0, 1)];
}
```

Obviously, since with one optional argument there are only two possibilities, the `foo bar:` method could just as well not have optional arguments, their omission being handled by the `foo` method. However, in the context of multiple optional arguments, the selectors to which an object responds do become an issue.

Similarly, if a library was previously distributed with the following method declared for some class handling streams on files

```
instance (id)
            with String name
          input: boolean input_p = FALSE
         output: boolean output_p = FALSE
  if-not-exists: int ne_action = 0;
```

and the new version of the library changes this method to be

```
instance (id)
            with String name
```

```
              input: boolean input_p = FALSE
             output: boolean output_p = FALSE
          if-exists: int e_action = 0
      if-not-exists: int ne_action = 0;
```

then it would be wise to incorporate a compatibility method in the new library, just to avoid having to recompile all code depending on the library, which is an issue especially in the case of shared libraries. The method below would then exist in the new library for the sole purpose of responding to the original selector.

```
instance (id)
            with String name
           input: boolean input_p = FALSE
          output: boolean output_p = FALSE
  if-not-exists: int ne_action = 0
{
  = [self with name input: input_p output: output_p
          if-exists: 0 if-not-exists: ne_action];
}
```

In a method invocation, if multiple methods match, methods with a longer name (i.e. with more (unused) optional name parts) are preferred over methods with a shorter name. This implies that the specification of `if-exists: 0` in the invocation in the example is unnecessary.

5.3 Pre- and postconditions

This section will explain pre- and postconditions.

5.4 Method definitions

```
method_definition:
          ['extern'] method_declaration
        | shared_method_part method_body
        ;
```

The first case is a valid definition if `extern` is given, which indicates that the actual implementation of this method is written in another language, or if the `method_declaration` includes a `redeclare` qualifier.

Every identifier in an argument_name in the shared_method_part introduces in the body of the method a local variable by that name, with the type as indicated in the declarations, which will be initialized to the actual value passed to an invocation of the method.

Every method has two implicit first arguments. First, there is self, the type of which is, by definition, id (see Section 4.5 [The id type], page 31). The second argument is the selector cmd, which is the selector that was used to invoke the current method. cmd isn't actually used in most method implementations; its precense is, however, an elementary part of the dynamic method binding and method forwarding mechanisms.

```
method_body:
          compound_expression
        ;
```

5.4.1 Compound expressions

```
compound_expression:
          '{' top_expression_list '}'
        ;
```

A compound_expression is a sequence of top expressions. The type and value of the compound_expression is the type and value of the last expression it contains. The value to be returned from a method is not the value of its method_body compound. A separate return_expression exists to assign the method return value (see Section 5.6.5 [Jumps and returns], page 46).

```
top_expression_list:
          top_expression ['; ' top_expression_list]
        ;
```

Note that the semicolon is a separator, not a terminator.

```
top_expression:
          local_var_decl
        | expression
        ;
```

A local_var_decl (see Section 5.5 [Local variables], page 41) is not an expression because that is unnecessary:

- The expression `int a = 1` can equally well be used without the `int a =` if the `a` is not used anywhere else, or, it can be braced: `{ int a = 1 }`. If it *is* used somewhere else, the declaration can be put in the enclosing compound and then, obviously, the plain assignment `a = 1` can be used as an expression.
- For the expression `int a = 1, b = a + 1`, the expression `{ int a = 1; a + 1}` can be used with exactly the same semantics. Remarks from the previous item are also valid for this case.
- The real reason for not letting a `local_var_decl` be a normal `expression` is because it would make typing a `tuple` unclear. For example, if `int a = 1, b = 2` were an `expression`, the type of the tuple

  ```
  (q = 1, 2, int a = 3, b = 4, 5)
  ```

  would depend on rules concerning the precedence of the comma in `tuple` and `local_var_decl`, i.e. the precedence of the comma would depend on its context. Since this is not considered a good idea and because of the limited usefulness of the `local_var_decl` being an `expression` (as shown in the preceding paragraphs), the `local_var_decl` is not an `expression`.

5.5 Local variables

Local variables can be introduced anywhere in a compound expression. The scope of the variable starts with its declarations and ends with the end of the enclosing compound (i.e. lexical scoping).

```
local_var_decl:
          entity_type local_var_list
        ;

local_var_list:
          local_var [‘,’ local_var_list]
        ;

local_var:
          identifier [ ‘=’ expression ]
        ;
```

If the `expression` is provided, it defines the initial value of the variable, named after the `identifier`, introduced by this `local_var`. If omitted, the variable is initialized to the default value of its type.

5.6 Expressions

Everything in a method body is an expression. In fact, the whole method body is an expression (a compound_expression, see Section 5.4.1 [Compound expressions], page 40).

```
expression:
          simple_expression
        | compound_expression
        | assign_expression
        | if_expression
        | loop_expression
        | return_expression
        | jump_expression
        | condition_expression
        ;
```

The condition_expression is explained later (see Chapter 7 [Conditions], page 57).

```
simple_expression:
          atom
        | array_reference
        | unary_operator expression
        | expression binary_operator expression
        | expression '?' expression ':' expression
        ;
```

For the definition of the semantics, precedence, and associativity of the unary operators, binary operators, and of ? : (the only ternary operator), see Section 5.6.6 [Operators], page 47.

```
atom:
          number
        | string_constant
        | identifier
        | tuple
        | method_invocation
        | type_conversion
        | 'void'
        | 'nil'
        ;
```

void is the only value of type void. nil is the invalid object reference; its type is tom.All.

5.6.1 Method invocations

```
method_invocation:
          '[' receiver method_name_part ']'
        | '[' receiver method_invocation_part+ ']'
        | method_name_part tuple
        ;
```

The first method invocation simply sends a message without arguments to the `receiver`. The second kind sends a message, together with associated arguments, and the third kind is a syntactic shorthand for

```
[isa <method_name_part> <tuple>]
```

A `method_invocation_part` is part of a method name plus and expression.

```
method_invocation_part:
          method_name_part expression
        ;
```

All the `method_name_part` elements of a `method_invocation` together define which message is sent to the `receiver`. The arguments to a method invocation, being the `expression` in each of the invocation's `method_invocation_part`, are evaluated from left to right. The receiver is evaluated before the arguments.

5.6.2 Tuples

A tuple is, as somebody once described, an 'unencapsulated ordered list of heterogeneous values'. The type of a tuple is the tuple type composed of the types of the elements (see Section 4.6 [Tuple types], page 33).

```
tuple:
          '(' tuple_elements ')'
        ;

tuple_elements:
          tuple_element [',' tuple_elements]*
        ;

tuple_element:
          [expression]
```

```
        ;
```

When used as an rhs expression, each `tuple_element` of a tuple must not be empty. Empty tuple elements are allowed when the tuple is used as the lhs in an assignment; every empty field indicates a value in the tuple to be ignored. For example, the following expression is an, albeith elaborate, way to assign the value 2 to the variable `a`.

```
(,, a,,,) = (0, 1, 2, 3, 4, 5)
```

5.6.3 Array indexing

```
array_reference:
          expression '[' tuple_elements ']'
        ;
```

The `array_reference` rule allows array indexing to be done in TOM in a way familiar to many (C) programmers. Actually, the expression `a[b]` is a short-hand for `[a at b]`; when used as the lhs of an assignment, an array reference results in the invocation of the method `set at`: `a[b] = c` is equal to `[a set c at b]`.

Note that the type of the index is not restricted to be integer; it can be anything. For example, a dictionary mapping objects to objects could implement a method

```
Any at All key;
```

allowing a lookup of the value for the key `HOME` in the dictionary `environment` to be written as

```
String home = environment["HOME"]
```

which obviously is syntactic, but very tasteful, sugar.

With the indexing expression being the elements of a tuple, multi dimensional arrays can be indexed using `a[x, y, z]`, which obviously is rather different from `a[x][y][y]`.

[Note: Multi-dimensional arrays are not provided by the standard environment, but there is nothing making them impossible.]

5.6.4 Conditionals and loops

Currently, the only conditional in TOM is the `if else` conditional. (Other languages provide things like `switch` statements; which haven't found their way to TOM yet.)

```
if_expression:
        'if' tuple expression ['else' expression]
        ;
```

The `tuple` must be a single-element tuple with a `boolean` type. If it evaluates to the truth, the value of the `if_expression` becomes the value of the first `expression`, otherwise, the value of the whole expression becomes the value of the `expression` in the `else` branch. If the `else` branch is omitted, its value is the default value for the type of the `if_expression`. If the types of the first and second `expression` do not match, the type of the whole `if_expression` is void.

```
loop_expression:
          while_expression
        | do_expression
        | for_expression
        ;
```

A `loop_expression` causes iterative evaluation of an expression. Each loop construct is a simple syntactical variation of the others.

```
while_expression:
        'while' tuple body
        ;

body:
        expression
        ;
```

The `tuple` must have a boolean type. If it evaluates to `TRUE` the `body` is evaluated; this is repeated until the `tuple` no longer evaluates to `TRUE`.

The type of the while loop is the type of the `body` expression. Its value is the value of the `body` the last time it was evaluated. If the body is never evaluated, the value of the loop is the default value for the type.

```
do_expression:
        'do' body 'while' tuple
```

```
        ;
```

This do-loop is similar to the while-loop except that the `tuple` is only evaluated *after* the `body` is evaluated, i.e. the `body` is guaranteed to be evaluated at least once. The type of the do loop is the type of the `body`. Its value is the value of the `body` the last time it was evaluated.

```
for_expression:
          'for' '(' [expression] ';' [expression] ';'
                    [expression] ')' body
        ;
```

The for loop is pure syntactic sugar. The loop `for (a; b; c) d` has exactly the same meaning as

```
{
  {
    a'
  }
  while (b')
    {
      typeof (d) d' = d;
      {
        c'
      }
      d'
    }
}
```

where the `b'` is `TRUE` if `b` is empty. Similarly, the `a'` and `c'` are `void` if `a` and `c` are empty, respectively. Note that the `typeof` type operator is not actually available in TOM.

5.6.5 Jumps and returns

```
return_expression:
          '=' [expression]
        | 'return' [expression]
        ;
```

A `return_expression` sets the current notion of the return value of the current method and, if the `return` variant is used, terminates execution of the method. The value of a `return_expression` is the value of the `expression`, or `void` if `expression` is omitted.

[Note: `return` without an `expression` should mean to terminate execution of the current method, without modifying the current notion of the return value of the method. End note.]

```
jump_expression:
          break_expression
        | continue_expression
        ;
```

(The kind of `return_expression` with the `return` keyword also jumps but is considered more of a `return_expression` than a `jump_expression`.)

```
break_expression:
          'break' [expression]
        ;
```

The `break_expression` terminates the enclosing loop. If the `expression` is provided, its value will be the value of the loop. If it is omitted, the value of the loop will be `void`.

[Note: It should be possible to break out of a loop different from the enclosing loop. Obviously, the indication of which loop should be static, i.e. something like 'computed break' should not be possible in the language; catch and throw can be misused for that purpose. End note.]

```
continue_expression:
          'continue' [expression]
        ;
```

The `continue_expression` skips the remainder of the body of the enclosing loop, continuing execution with the loop condition (preceded by the loop modifier expression in case of a for loop). The value of the loop, if it were to fail its condition, will be the value of the `expression`. If omitted, the value of the loop will be `void`.

5.6.6 Operators

TOM knows the following non-keyword operators, in increasing precedence.

`=` Assignment. The type of the rhs must match convertibly the lvalue lhs. All modifying assingments, `*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `>>>=`, `&=`, `|=`, `^=`, `&&=`, and `||=`, share the precedence of `=`.

Assignment is right associative. Thus, `a = b = 1` first assigns `1` to `b` and then assigns `b` to `a`.

`? :` Conditional expression with mandatory else branch. This really is syntactic sugar for `if else`. If the types of the if and else branches do not match, the type of the whole expression is `void`.

`->` Implication. `a -> b` is identical to `!a || b`.

`||` Short circuit boolean 'or' operation. Both arguments must have a boolean type.

`&&` Short circuit boolean 'and' operation. Both arguments must have a boolean type.

`<`
`<=`
`==`
`!=`
`>=`
`>` Comparisons. The ordering comparisons (`<`, `<=`, `=>`, and `>`) only operate on numeric-typed values. The equality comparisons (`==` and `!=`) operate on any type; they test the identicality of the arguments.

Testing the equality of of two tuple-typed values tests the equality of all the elements, i.e. *without* short-circuits.

`^` Boolean exclusive-or of boolean values or bitwise exclusive-or of integer numeric typed values.

`|` Boolean or of boolean values or bitwise or of integer numeric typed values.

`&` Boolean and of boolean values or bitwise and of integer numeric typed values.

`<<`
`>>`
`>>>` Bitwise shifting of integer numeric types. `<<` shifts left the number of bits specified by the rhs, `>>` shifts right, and `>>>` shifts right logically, as opposed to arithmetically. `>>>` is useful for 'unsigned' shifts on the signed `int` and `long` typed values.

`+`
`-` Numeric addition and subtraction.

`*`
`/`
`%` Numeric multiplication and division. `%` is integer modulo.

`~`
`!`
`-` `~` is the inversion of a boolean value, or the bitwise inversion of an integer numeric value. `!` extracts and then negates a truth value from any non-tuple typed value. It returns the boolean truth if the argument does has the default value for its type. `-` is the unary minus. It operates only on numeric types.

`[]` Square brackets are syntactical sugar for array indexing. See Section 5.6.3 [Array indexing], page 44.

Operations on numeric types are performed in the precision of the type of the result. Thus, adding two floats results in a float, adding a byte to an int results in an int, after the byte has implicitly been converted to an int.

Operators provide functionality which is statically bound.

[Note: They are actually special methods declared in the `class _builtin_.Top`. They are special since they can not be redeclared in any way, the syntax of TOM allowing neither a character like '`+`' in a method name nor a method with one namepart and two arguments. The operator methods are looked up in the class of the current object.

Thus, with a slightly bent syntax and ignoring the fact that not all objects have an `isa` as far as the compiler knows, `a + b` is actually a shorthand for `[isa + a, b]`. End note.]

6 Typing

This chapter explains tom's rules for typing, overloading, type conversion and type indication.

[Note: This chapter still needs a lot of work. End note.]

6.1 The type of constants

The type of integer constants, such as `0x12345` and `0` is `int`. The type of character constants, as in `'a'` is `byte`. The type of floating point constants, such as `1.0` is `float`. The type of floating point constants with a 'd' for an exponent indicator, such as in `1.23d4`, is `double`. The type of `"string constants"` is `ByteString`.

6.2 Resolving

Expressions, or parts thereof, are resolved in a context. If the context is empty, a set of possible types is returned; if the context is not empty, those types in the context are returned which the expression can match. For example, the expression `'a'` will return `{byte}` given an empty context; in the context `{State, float, int, short}` it will return `{int, short}`.

6.2.1 Resolving method invocations

This section describes the currently used algorithm for selecting which method to actually invoke for a given method invocation.

1. Resolve the receiver to be an object.
2. Get for the receiver a list of eligible methods. A method is eligible if
   a. it is defined for the receiver or inherited by the receiver,
   b. the name parts of the invocation fit the name parts of the method, i.e. the order of the name parts provided match the order of the corresponding name parts in the method, and no name part of a non-optional argument is omitted,
   c. its return type is either `dynamic` or it matches at least one type in the context (if that is non-empty),

   d. it is not a method which is overridden, in a subclass, by another eligible method, and
   e. if the receiver is not `super` and the types of the arguments and return values exactly match those of an already eligible method.
3. Build the possible argument types.
4. For each actual argument, resolve it in the context of its possible types. If it is unresolvable (i.e its set of possible types is empty), this invocation can not be resolved either.
5. If any of the possible argument types was restricted by the attempt to resolve, eliminate from the eligible method those methods which are no longer applicable. If at least one method was eliminated, rebuild the possible argument types and continue with the previous step.
6. If the context is a single type or all arguments have been fully resolved (i.e. typed) or all but one method have been eliminated, continue with the next step, otherwise return the list of possible types for this invocation.
7. If we're expected to fully resolve, i.e. the context is a single type, eliminate those methods from the eligible set which do not fit the expected type best.
8. Eliminate those methods from the eligible set which do not have the highest number of fully matching arguments.
9. Eliminate those methods from the eligible set which do not have the highest number of name parts. This favours methods with lots of optional parts over methods with less optional parts. The rationale behind this is that the latter is probably only provided to invoke the former with its default arguments while responding to an additional selector.
10. If any of the eligible methods involve dynamic typing while some do not, eliminate the former. This is needed, for instance, to have `[[stdio out] print 1234]` invoke the integer printing routine instead of the dynamic one which would just invoke the integer printing routine for this argument anyway.
11. If we're down to one method, fully type any remaining arguments which are yet untyped. The type of the invocation is the return type of the method, and the set returned is deduced from the context and the type in the usual way. If the return type is `dynamic`, the context must specify a single type.
12. Otherwise, resolving failed and, in the empty context, we return the set of possible return types of the remaining methods; in a non-empty context, return those types in the context which can be matched by at least one of the remaining methods.

An important aspect of this algorithm is that it correctly produces an error for undecidable invocations, such as the invocation of `foo (byte (3), byte (4))` while the available methods are `foo (int, byte)` and `foo (byte, int)`.

A disadvantage of the current setup in the compiler is that an attempt to invoke a method with slightly wrong argument types results in an error indicating that the expression could not be typed, without any positive hints about the possibly applicable methods.

6.2.2 Messages to super

A method invocation is a different name for sending a message. Such a message has a single receiver.

```
receiver:
          expression
        | 'super'
        | 'super' '(' class_name ')'
        ;
```

Normally, the receiver is an `expression`. This expression must resolve to an object type to denote the actual receiver of the message.

When a subclass redefines a method inherited from a superclass, it is often desirable to invoke the overridden method. A good and frequent example of this is provided by `init` methods: suppose a class `Foo` has as designated initializer `initWith int a`, and its subclass `Bar` has as designated initializer `initWith int a and int b`. Then the implementation of the latter could look like this:

```
<doc> The {Bar} designated initializer. </doc>
id
  initWith int a
       and int b
{
  my_b = b;

  = [super initWith a];
}
```

In this example, the value returned is what is returned by the invocation of the overridden method. It is of course also possible that the overridden method is called as the first expression, with the extra work performed after that; it all depends on the needs of the method. Example:

```
<doc> Another example of the {Bar} designated initializer. </doc>
id
  initWith int a
       and int b
{
  if (![super initWith a])
    return nil;

  /* Only foobar {b} if we're alive and
     super's initialization didn't fail. */
  my_b = foobar (b);
```

```
    = self;
  }
```

It is possible for an invocation of a method to `super` to be unresolvable because differing implementations of the method are provided by the superclasses of the current class. In this case, the last alternative of the `receiver` syntax should be used to direct the attention of the compiler in the direction of the right superclass. For example, if a class `A` is a direct subclass of both `B` and `C`, and these classes respond to `foo` differently, the invocation of `[super foo]` from `A` will be unresolvable. By replacing this expression by `[super (A) foo]` or `[super (B) foo]`, the expression becomes resolvable since the compiler is told which method to prefer.

Note the following details concerning sending messages to `super`:

- When the `super` receiver is classified this way by a class name, the class must be a direct superclass of the current class, even if that class only inherits the intended method implementation.
- An unclassified message send to `super` is implicitly classified by the compiler. This implies that, for the class `A` from the example, if only `B` implements the method `bar`, the invocation of `[super bar]` will remain to invoke `B`'s implementation even if, through dynamic loading, the class `C` suddenly implements that method too.

6.3 Implicit conversions

As hinted in the previous section, some implicit type conversions are performed in an attempt to match a type in the context of the expression. In short, every type exactly matches itself; narrow integer types match wider integer types; the narrow floating type matches the wider floating type; a subclass matches any of its super classes; and a tuple matches another tuple if they have the same number of elements, and corresponding elements match.

6.4 Explicit conversions

Explicit conversions exist for two reasons: to allow for making implicit conversions explicit, and to enable conversions which are not possible implicitly.

```
type_conversion:
        entity_type '(' expression ')'
      ;
```

Allowed explicit conversions are any conversion which is implicitly possible; conversion of an integer type to a narrower integer type; conversion of the wide floating point type to the narrow floating point type; conversion from integer to floating point and vice versa; conversion from a class to a subclass thereof; and conversion from `ByteString` to `selector` if the `ByteString` typed expression is a string constant.

6.5 Type indication

Type indication is primarily useful as a hint to the resolver, which would otherwise not be able to decide upon the type of an expression. TOM does not provide an explicit type indication syntax, since there are other, quite obvious, means to achieve the same effect.

Suppose the existence of two methods with the same argument types and which differ only in their return type, such as

```
int foo;
float foo;
```

Invoking `[a foo]` in an empty context will not be resolvable since it can not be decided which message to actually send. However, replacing this invocation by the following expression obviously solves the problem by providing an explicit, single type, context.

```
{float x = [a foo]}
```

7 Conditions

TOM's mechanism of handling exceptional events and other such conditions is based on the conditions mechanism as provided by Common Lisp [gls].

The creation and issueing of conditions is handled through objects defined in the TOM unit (see Section 8.1 [unit tom], page 61): `Condition` objects can be raised or signaled, and any object can be thrown.

Condition handling (in contrast to creation) is part of the language. For this purpose, TOM provides a few special expressions:

```
condition_expression:
          catch_expression
        | bind_expression
        | unwind_expression
        ;
```

[Note: Currently, each `Condition` instance is associated with a `ConditionClass` instance; the instances of `ConditionClass` defining a single inheritance hierarchy of condition classes. Since real conditions need to be subclasses from `Condition`, to carry the extra state possibly needed, it seems a good idea to implement the condition class hierarchy within the TOM class hierarchy. End note.]

7.1 catch

```
catch_expression:
          catch `(' tag `)' expression
        ;
```

`catch` establishes a point of return in the invocation stack for a *throw*. A throw is an invocation of the `throw` method of the tag. (`catch` actually has nothing to do with conditions, but its usefulness is most profound in condition handling, the reason for it being discussed in this chapter.)

```
tag:
          expression
        ;
```

The default implementation of throw (as provided by the instance tom.All) causes execution of the program to resume after the most recently established catch of the same tag object. Obviously, with throw being a method, an object can override it. Be aware however, that the semantics of the default implementation guarantee not to return from the invocation of throw.

Normally, the value returned from a catch is the value of the expression. If a catch is thrown, the value returned is the value of the argument to the throw. If the argument to the throw is void, the catch simply returns the default element for the type of the value it would otherwise return. For arguments to throw other than void, it is an error if the type thrown does not match the type caught.

7.2 bind

```
bind_expression:
        bind `(' handlers `)' expression
      ;

handlers:
        handler [`;' handlers]
      ;
```

A bind_expression sets one or more handlers for any conditions which are raised or signaled during the evaluation of the expression. The value returned from the bind is the value returned by the expression.

```
handler:
        `(' condition `,' expression `)'
      ;

condition:
        expression
      ;
```

Each handler is an expression which will be invoked if the condition, or a sub condition thereof, is raised (or signaled). The condition normally is an instance of tom.Condition.

Implicitly declared in the context of the handler expression is a local variable Condition condition; it is the condition being raised.

A useful thing to do for a handler is to perform a `throw` to force a non-local exit out of the exceptional state. Note, however, that the handler is invoked in the context of the condition raiser. This implies that in the following expression

```
catch (self)
  bind ((error, {[self throw void]; nil;}))
    {
      do_something_difficult;
    }
```

the `throw` need not immediately return from this `catch` since another catch of the same object `self` could have been set (and active) during the invocation of `do_something_difficult`.

A handler does not *need* to handle the condition by performing a non-local exit. If the handler does not wish to handle it, the value returned by the handler should be the `condition` object it was passed. If the handler returns any other value, which must always be an object, the handler considers the condition to have been handled. If the condition was signaled, the value returned by the handler is returned from the `signal`. If the condition was raised, the return value is ignored and the condition mechanism continues as if the handler had not handled the condition.

7.3 unwind

```
unwind_expression:
          unwind '(' protection ')' body
        ;

protection:
          expression
        ;
```

An `unwind` evaluates the `body`, followed by the evaluation of the `protection`. The value returned by the `unwind` is the value of its body.

If, during the evalutation of the body, a tag is thrown, prematurely exiting the context of the `unwind`, the protection expression is still evaluated. If the protection code does not raise or throw anything, then after its evaluation the original throw causing its invocation will continue to be thrown futher.

8 Standard environment

[Note The standard environment descriptions are currently rather empty, as I haven't written the documentation generator yet. For documentation, read the declarations plus anything between `<doc>` and `</doc>` in the sources. End note.]

8.1 unit tom

The 'tom' unit is the most elementary tom unit. Without the classes provided by the tom unit, the compiler can not compile and a TOM program can not run. The tom unit is closely related to the TOM runtime (see Chapter 10 [Runtime], page 69).

The tom unit provides abstractions and classes every TOM program will need or will probably need, such as the runtime, collections, streams, archiving, etc.

8.2 too

The 'too' unit is not as elementary as the tom unit, but it provides useful abstractions. TOO currently contains abstract network ports, TCP/IP ports, and a RunLoop (event dispatcher). In the future it will include Distributed Objects, and probably also Mach 'networking'.

8.3 C

The 'C' unit makes available to TOM programs those functions from the standard C library which are useful to TOM programs.

9 Tools

All TOM tools adhere to the following convention for filename extensions:

`.t` implementation file. This is the only kind of file to be created and edited by the programmer. All other files are generated.

`.j` interface file.

`.i` info file; this contains information on the corresponding implementation file for the resolver to use.

`.u` unit file. This describes a unit.

9.1 Common options

The following options are common to the TOM tools.

`-h` Output an invocation summary and exit.

`-I path` Add the elements of the colon-separated directories in the `path` to the list of directories searched for units.

When searching for a unit named `foo`, first the file `foo.u` is looked for in each directory of the search path. If it can not be found, the search is repeated for `foo/foo.u` relative to each directory in the path.

The default search path defaults to `/usr/local/tom`, but is actually `$prefix/tom` where `$prefix` was specified during installation, for example `/usr/gnu`, or `/usr`.

`-u unit` Name the unit to be handled. Default is the single unit in the current directory ('.'). If this option is not provided, it is an error if the current directory contains not exactly one file with the extension `.u`. If the option `-u foo` is provided, the file `foo.u` will be used as the unit file.

`-v` Increase verbosity.

`--version`

Output the version information to `stdout` and exit. With `-v` (preceding `--version`), the information is more verbose.

9.2 gi

`gi` is the interface generator. It can update the interface files from the implementation files in a unit, or generate a unit file for given implementation files.

Default, `gi` only writes interfaces for implementations seen in the implementation files it reads. When given the `-fdocumentation` option, all `<doc>` documentation comments will be written too. This option is meant for creating interface files to be distributed with a non-source distributed library.

[Note: `gi` should have options indicating which kinds of documentation comments are to be copied, instead of only `<doc>`. Similarly, it should have options to create, in addition to TOM interface files, other formats, such as texinfo and html files. End note.]

9.2.1 Invoking gi

`gi` understands the following options:

`-dgc number`
: Define the debugging verbosity of the garbage collector used within the gi. By default the garbage collection is turned off, which is equal to `-dgc-1`. To enable it without debugging information, supply `-dgc 0`. You normally do not enable gc.

`-F`
: Force writing of the applicable interface files irrespective of their modification time relative to the implementation files. Normally, an interface is only generated if it does not exist or if its modification time is less recent than that of the corresponding implementation.

`-fcomments`

`-fno-comments`
: Include (or not) top-level C-style comments in the output. Default is not to include comments.

`-fdocumentation`

`-fno-documentations`
: Include (or not) top-level documentation comments in the output. Default is not to include documentation comments. Currently, only `<doc>... </doc>` comments are included. [Note: This is a bug. End note.]

`-k`
: Continue with remaining files in case of an error. This is similar to the `-k` option to make(1).

`-U unit` When generating the unit file, specify that the unit under consideration depends on the `unit`.

9.3 otmc

`otmc` is the compiler proper. It digests an implementation file (with extension `.t`), reading any unit definition files and interface files it needs, and (currently) outputs a `.c` file to be compiled by the GNU C compiler. It also creates a `.i` file to be read by the resolver.

9.3.1 Invoking otmc

`otmc` understands the following options:

`-dgc number`
: Define the debugging verbosity of the garbage collector used within the compiler. By default the garbage collection is turned off, which is equal to `-dgc-1`. To enable it without debugging information, supply `-dgc 0`. You normally do not enable gc.

`-flookup-direct`
`-flookup-lookup`
`-flookup-send`
: Determine how method invocations are dispatched. The default dispatch mechanism depends on the machine otmc is running on, but for a mature port it probably is `-flookup-send`. See Section 10.4 [Message dispatching], page 71 for more information.

`-fsuper-direct`
`-fsuper-lookup`
`-fsuper-send`
: Determine how invocations to `super` are dispatched. The default is either `-fsuper-lookup` or `-fsuper-send`, again depending on the machine.

  `-fsuper-send` is not possible on machines which pass some of the arguments in registers.

`-fatomic-gc`
`-fincremental-gc`
: Define whether the code to be compiled can assume atomic garbage collection runs. With this assumption, an assignment to a class or instance variable incurs no overhead. An atomically garbage collected program consumes, on average, about half the memory

of an incrementally collected program, since when using incremental gc, a dead object is not reclaimde by the end of the *next* run that is completed. With atomic gc, it will be reclaimed by the end of this run.

The default is incremental gc. For units that are distributed in a binary (non-source) form, atomic gc should *never* be assumed.

[Note: Currently, the runtime does not actually take advantage of any atomicity in the garbage collection runs. End note.]

`-precious`
: Do not remove the output file, even in the case of errors.

`-Wempty`
: Warn about empty expressions. This is the default.

`-Wno-empty`
: Do not warn about empty expressions.

9.4 otmr

The resolver, `otmr`, can be used to create a C header file containing declarations for C sources to interface with a unit. Such declarations are declarations of selectors, classes, layout of an extension's object variables, etc. The resolver functions by reading the `.i` files created by the compiler.

When resolving a program `otmr` generates a C source file containing the data structures needed by the runtime. This implies that the startup time of a TOM program due to the runtime data structures needing to be built is zero. Resolving itself does take some time, and also GNU CC seems to be very slow at compiling static variable initializations.

9.4.1 Invoking otmr

`otmr` understands the following options:

`-n`
: Do not output the definition file, i.e. only output the declaration file.

`-v-deferred`
: Report the deferred selectors of each object. Works only if none of `-n`, `-y`, and `-Y` are given.

`-v-deferred-instances`
: Report the deferred instances. Works only if none of `-n`, `-y`, and `-Y` are given.

`-v-non-deferred-instances`
: Report the non-deferred instances. Works only if none of `-n`, `-y`, and `-Y` are given.

`-y unit`
: Emit information for dynamic loading the unit to be resolved into the program `unit`.

`-Y`
: Emit information for runtime resolution. This dramatically reduces the running time of the resolver and incurs an, albeith slight, startup overhead for the program.

9.5 gp

`gp` is the TOM parser generator. It generates a recursive descent parser given an LL(1) grammar. The current implementation of `gp` is a quick hack, especially with respect to error recovery (which is totally absent) en the lexer interface (which is a manual retrofit to flex, with all limitations implied by it). However, being written in TOM, `gp` serves as an excellent compiler and environment test case.

9.5.1 Invoking gp

GP does not share any of the options common to the compiler, resolver, and interface generator, since gp was written in TOM; it isn't a TOM tool strictly speaking as it involves transforming its input language into TOM.

```
gp [options] input-file
```

GP reads the `input-file` and emits its output to `stdout`.

In addition to the options provided by the standard environment, GP understands the following options:

`-d`
: Output a trace of the running gp parser to `[stdio err]`.

`-o file`
: Output to `file` instead of `[stdio out]`.

`-v`
: Be verbose. This reports the number of lines in the input file.

10 Runtime

The TOM runtime provides everything a TOM program needs, and more. This chapter explains the operation of the runtime.

In C, all functionality of TOM and the TOM runtime can be obtained by including `<tom/util.h>`. This will also include the header generated by the resolver for the tom unit.

10.1 Program startup

When a TOM program starts, the following sequence of actions is taken:

- If the program was not statically resolved, build the method dispatch tables and other runtime structures needed but not built by the resolver. When run, this installs what the GNU Objective-C runtime very nicely calls the 'premature' dispatch table for each object. Upon invocation of a method through said table, the actual dispatch table will be built and put in place.
- Collect the arguments to the program, excluding C's `argv[0]`, in `tom.ByteString` objects into `tom.Runtime.all_arguments`. Set the `tom.Runtime.program_name` and `tom.Runtime.long_program_name` from `argv[0]`.
- Initialize the `in`, `out`, and `err` streams of `tom.stdio`.
- Invoke the load imps, i.e. every class method matching `void load Array arguments`, with the arguments collected from the command line. A load imp is allowed to recognize options and remove them from the arguments. It is customary for such options to start with a colon (':') instead of a dash ('-').
- Store the array of remaining arguments in `tom.Runtime.arguments`.
- Invoke the following method of the `class tom.Runtime`:

  ```
  int
        start (All, selector) (object, sel)
    arguments Array arguments;
  ```

  where the `sel` is the selector for the main method to be invoked, which normally is `int main Array arguments`. The `object` is the receiver of this message, normally a class object.
- When the previous method invocation returns, all open streams are flushed and the value it returned it used as the exit code.

10.2 C names for TOM types

The following table lists the C type to use for which TOM types.

`void`	`void`
`tom_byte`	`byte`, `boolean`
`tom_char`	`char`
`tom_int`	`int`
`tom_long`	`long`
`tom_float`	`float`
`tom_double`	`double`
`void *`	`pointer`
`selector`	`selector`
`tom_object`	*any* object reference.

10.3 Selectors

In the runtime, not only in the C world, a selector is identifyable by a string which is called the selector's name. Such a name consists of the various name parts and an abbreviated form of the types involved. The abbreviated type names are:

```
v     void
o     boolean
b     byte
c     char
i     int
l     long
f     float
d     double
p     pointer
s     selector
r     object reference
x     dynamic
```

The following table shows examples: for several method declarations the name of the corresponding selector.

[Note: I should write down the grammar for this, but I'm too lazy. End note.]

`int value;`

`"i_value"`

`void setValue float d;`

`"v_setValue_d"`

`(Foo) bar (int, double) (a, b) with: int c = 0;`

`"r_bar_(id)_with:_i"`

In C, a selector is defined by a struct; The `struct selector` is the direct implementation of the TOM selectors.

```
typedef struct selector
{
  unsigned int sel_id;

  struct name name;

  struct trt_selector_args *in, *out;
} *selector;
```

`sel_id`
: This is the unique numeric identity of a selector. All selectors have such an identity in the closed enumeration of all selectors. Every two selectors with identical name, argument types and return types have the same identity. If no dynamic loading has taken place, the following is true for any two selectors `a` and `b`:

```
a->sel_id == b->sel_id <-> a == b.
```

`name`
: the name of the selector. This has two fields: `s` being the zero terminated string, and `len` being its length.

`in`
`out`
: the selector argument descriptions for the arguments to (`in`) and return value from (`out`) methods denoted by this selector.

10.4 Message dispatching

This section explains how messages can be dispatched, i.e. how the implementation of a method can be invoked, given the message.

A method is translated by the compiler to a C function with essentially the same arguments as the method, with two mandatory additions and some additions depending on the return type of the method. For the method

```
(boolean, Any) next
```

defined for the `instance tom.Enumerator`, the C function implementing this is

```
tom_byte
i_tom_Enumerator__or__next (tom_object self, selector cmd,
                            tom_object *ret1)
```

As can be seen, the first element of the tuple return type is the type returned by the implementation C function. Such a function always has two 'implicit' first arguments: `self` being the object receiving the message, and `cmd` being the message sent. Following these two are the 'normal' arguments to the method, which are none in this case. Finally, any of the remaining return values are to be returned in the pointer arguments supplied after the normal arguments. In the example, a pointer for returning the second tuple element is provided.

A message is dispatched by invoking the method implementation for the given `(self, cmd)` pair. The lookup of the implementation, i.e. the pointer to the C function, can be done in three different ways:

lookup
: Call the function `trt_lookup` with the receiver and the selector to be invoked (i.e. the values to be passed for `self` and `cmd`). `trt_lookup` returns a function pointer to the method implementation. Invoke that function with the arguments.

direct
: A direct lookup is equivalent to inlining the `trt_lookup` function. This is a rather unwise way of invoking a method as it considerably increases code size.

send
: When using `trt_send` to dispatch a message, the function `trt_send` is invoked with all the arguments to be passed to the method implementation. `trt_send` will perform the lookup and jump directly to the implementation.

These different dispatching mechanisms can be selected by an option to otmc. They are described here for explanatory purposes; never implement any of these directly in your C code; the next section explains how to do that portably.

Sending is the preferred way of dispatching messages, though possibly not present on all TOM implementations as it involves an assembly language routine. Also note that there are dependencies of the applicability of some dispatching mechanisms. For example, it is impossible to use sending on dynamically loaded code on `hppa-hpux` machines. This is not a TOM feature but due to the interspace stubs needed by the hpux shared library interspace calls, added to the fact that a callee stores the return program counter in the stack frame of the caller.

When doing profiling on a TOM program, all code should really use the lookup way of dispatching instead of sending to dispatch. Otherwise, all methods will be reported to only invoke `trt_send`, and `trt_send` will be reported as the culprit which invoked every method and thus effectively void the use of the call graph.

10.4.1 Messaging from C

To invoke a method of a TOM object, use the `TRT_SEND` macro. For example, to retrieve the length of an array `a`:

```
tom_object a = ...;

tom_int len = TRT_SEND (, a, SEL (i_length));
```

The second argument to `TRT_SEND` is the receiver of the message. The third argument is the selector to be sent. A selector is an invocation of the `SEL` macro, with as argument the selector's name (see Section 10.3 [Selectors], page 70) with any nasty characters replaced by an underscore.

A more elborate example shows almost all pitfalls when using `TRT_SEND`. In this example the method `(int, float) split float f` is invoked, which in TOM would be written as

```
SomeClass receiver = ...;
float fractional, number = ...;
int integer;

(integer, fractional) = [receiver split number]
```

is invoked from C as follows:

```
tom_object receiver = ...;
tom_float fractional, number = ...;
tom_int integer;

integer = TRT_SEND ((tom_int (*) (tom_object, selector,
                                  tom_float, tom_float *)),
                    receiver, SEL (_if__split_f), number, &fractional);
```

The first argument to `TRT_SEND` is a cast to the type of the function actually being invoked. This cast is mandatory if the return type of the function (implementing the method) invoked is not a `tom_int`. The argument prototypes in the cast are mandatory when needed to prevent the C compiler from doing undesirable type conversions due to it not having seen a full prototype of the function being invoked. For instance, usually, `tom_float` is simply a `float`, which the compiler will 'upgrade' to a `double` when passed as an argument for which the prototype has not been seen.

This example also shows that for a method returning a tuple, the first element of the tuple (or the first element thereof in case it is a tuple too (or...)) is actually returned from the C function implementation, and any remaining elements of the tuple returned are stored in variables the address' of which has been passed as 'invisible' trailing arguments.

10.5 More types

This section lists various types needed and/or provided by the runtime.

10.5.1 struct name

```
struct name
{
  char *s;
  int len;
};
```

The `s` points to the zero-terminated C byte-string holding the name. The `len` is the length of `s`.

10.5.2 trt_selector_args

```
struct trt_selector_args
{
  int num;

  enum trt_type_encoding args[0];
};
```

A `trt_selector_args` describes the arguments to or return value from a method. In this context, all values are de-tupled and concatenated. Thus, a selector accepting an `int` and a `float` has the same argument description as a selector accepting a tuple `(int, float)`. For the arguments, this excludes the implicit two first arguments, `self` and `cmd`.

`num` The number of entries in `args`.

`args` The description for each flat argument.

10.5.3 trt_type_encoding

The `enum trt_type_encoding` is used in the definition of argument and return types of selectors (see Section 10.3 [Selectors], page 70).

```
enum trt_type_encoding
{
  TRT_TE_VOID,
  TRT_TE_BOOLEAN,
  TRT_TE_BYTE,
  TRT_TE_CHAR,
  TRT_TE_INT,
  TRT_TE_LONG,
  TRT_TE_FLOAT,
  TRT_TE_DOUBLE,
  TRT_TE_POINTER,
  TRT_TE_SELECTOR,
  TRT_TE_REFERENCE,
  TRT_TE_DYNAMIC,
};
```

10.6 Functions

This section describes functions and macro's defined by the TOM runtime, or any of its C header files.

10.6.1 byte_string_with_c_string

```
tom_object
byte_string_with_c_string (const char *s);
```

Return a newly allocated instance of `tom.ByteString` holding the characters from the zero-terminated string `s`. Obviously, the trailing zero is not contained in the returned `ByteString`.

10.6.2 byte_string_with_string

```
tom_object
byte_string_with_string (const char *s, int len);
```

Return a newly allocated instance of `tom.ByteString` holding the first `len` characters pointed to by `s`.

10.6.3 trt_assign_local_var

```
TRT_INLINE void *
trt_assign_local_var (void *object)
```

This function must be invoked if the `object` has just been assigned to a local variable and it is to live over a method invocation. Note that **this is only necessary iff** the stack protection policy implemented by the garbage collector (configured when building the TOM tools) is `SP_PROTECT` as opposed to `SP_MARK` (see `config/default.h` and `config/target.h`), so, probably (ahem), it isn't necessary.

[Note: Obviously, the type of the argument should be `tom_object` instead of `void *`. End note.]

10.6.4 trt_assign_object_var

```
TRT_INLINE void *
trt_assign_object_var (void *object, void *value);
```

This function must be invoked if the object pointed to by `value` has just been assigned to an object variable of the `object`. This is needed in case the garbage collector performs non-atomic runs. Thus, if you're writing library code in C, you *must* use this function. The compiler outputs calls to this function if the flag `-fincremental-gc` is provided on the command line (see Section 9.3.1 [Invoking otmc], page 65).

[Note: Obviously, the type of the two arguments should be `tom_object` instead of `void *`. However, since debugging a TOM program currently means debugging the (not so unreadable, to the trained eye at least) C code output by the compiler, the compiler types each object in the output to its C struct (as far as the compiler can know the layout at compile time). Hence, `void *` is used in some places where really `tom_object` should be used, to avoid numerous casts or warnings. End note.]

10.6.5 trt_ext_address

```
void *
trt_ext_address (tom_object self, int extension_id);
```

`trt_ext_address` returns a pointer to the state information of the object `self` for the state introduced by the extension with the identity `extension_id`. **This is the only legitimate way to obtain a pointer to some state held by some object.** The only exception to this rule is the state information introduced by the State class (or instance), which, by definition, resides at offset 0 from `self`.

10.6.6 trt_selector_args_match

```
int
trt_selector_args_match (struct trt_selector_args *a,
                         struct trt_selector_args *b);
```

Return `1` if the number and types of the elements in `a` and `b` match, or `0` otherwise.

Normally, this test is very fast, since the resolver guarantees that for every pair of selector argument descriptions, `a` and `b`, if `trt_selector_args_match` returns `1`, the `a == b`. However, in the context of dynamic-loading, selector argument descriptions can guaranteed to be unique, thus making `trt_selector_args_match` slightly more expensive.

10.6.7 trt_selector_named

```
selector *
trt_selector_named (char *s, int len);
```

Return the selector whose name matches the name held in the first `len` bytes pointed to by `s`. Return `NULL` if such a selector does not exist.

10.6.8 trt_type_name

```
char *
trt_type_name (enum trt_type_encoding type);
```

Return a zero terminated C string holding the name of the type. If the type is not a valid value, the string "<unknown type %d>" is returned, with the numeric value of the type replacing the %d.

[Note: In the case of an unknown type being returned this function leaks memory, since the string returned is malloced. End note.]

10.6.9 xmalloc

```
void *xmalloc (unsigned int n);
void *xcalloc (unsigned int n, unsigned int m);
void *xrealloc (void *p, unsigned int n);
void xfree (void *p);
```

Use these allocation manipulation routines instead of the x-less counterparts they are wrapping, since the 'x' is a clue on some machines.

References

[gls] Guy L. Steele, *Common Lisp the language, 2nd edition*, Digital Press, Woburn, MA, 1994.

[bm] Bertrand Meyer, *Reusable software, the base object-oriented component libraries*, Prentice Hall International, Hemel Hempstead, UK, 1994.

Concept Index

R

S

T

U

V

W

X

Grammar Index

S

T

U

W

Table of Contents