Higher-Order

Report on the Tycoon-2 Programming Language
Version 1.0

Draft, Do not Copy or Distribute

Andreas Gawecki, Axel Wienberg

© Higher-Order GmbH, Hamburg.

All rights reserved.

February 6, 1998

Contents

1 Tycoon Language Rationale **3**

2 Lexical Entities **4**

2.1 Literal Constants 4

2.1.1 Numbers 4

2.1.2 Char 5

2.1.3 String 6

2.1.4 Symbol 7

2.2 Identifiers 7

2.3 Reserved Words 7

2.4 Documentation Strings 7

2.5 Single Line Comments 7

2.6 Other Tokens 8

3 Classes **8**

3.1 Method Definitions 9

3.2 Slot Definitions 9

3.3 Private Slots and Methods 9

4 Types **10**

5 Values **10**

5.1 Messages 10

5.1.1 Standard Dot Notation 10

5.1.2 Assignment Messages 13

5.1.3 Binary Messages 13

5.1.4 Lazy Binary Messages 14

5.1.5 Unary Prefix Messages 15

5.1.6 Mapping Messages 15

5.2 Local Variables 16

5.3 Sequences 16

5.4 Function Objects 16

5.5 Array Constructors 18

Higher-Order

6 Proposed Extensions **18**

6.1 Pre- and Postconditions 18

6.2 Packages 19

6.3 Inner Classes 19

7 System Library Library Examples **20**

7.1 What about nil, true and false? 21

7.2 Control Structures 21

7.2.1 Object Protocol 21

7.2.2 Boolean Protocol 22

7.2.3 Loops and Conditionals 24

7.2.4 Safe Downcast 25

7.2.5 Exception Handling 26

7.3 Arithmetic 27

7.4 Strings and Symbols 29

7.5 Output 30

8 EBNF Syntax **32**

9 Changes from Tycoon Version 0.9 to Version 1.0 **36**

9.1 Syntactic Changes 36

9.2 Semantic Changes 37

9.3 Standard Library Changes 37

1 Tycoon Language Rationale

Tycoon is a programming environment for the development of persistent distributed systems which provide customer-oriented information services in open environments.

Tycoon is a pure object-oriented language with classes and inductively defined subtyping rules. Tycoon supports the classical object model where objects are viewed as abstract data types encapsulating both state and behaviour. Method dispatch is based solely on the receiver's class. Tycoon

- is equipped with powerful type abstraction mechanisms like bounded and F-bounded type parameterization and run-time type-discrimination,
- has rich subtyping rules (covariant Self typing, higher-order subtyping)
- supports multiple inheritance as a way to easily modify the behaviour of objects using small packages of behaviour (called mixins in some other object-oriented languages),
- is higher-order with statically-scoped functions as first-class objects,
- provides orthogonal persistence for data, code and threads,
- provides orthogonal mobility across platforms for data, code and threads
- minimizes built-in language functionality in favor of flexible system add-ons.

To take full advantage of object-oriented software construction, Tycoon is a pure object-oriented language in the sense that every data entity is an object and all kinds of computations are expressed as (strongly typed) patterns of passing messages:

- Primitive control structures like conditionals (if), loops (while, for) and exception handling (try, catch) are not built into the language, but are implemented by classes and methods in the standard system library.
- Even low-level operations like integer arithmetic, instance variable access, and array indexing involve message sends. There is a subtype or 'is a' relationship between built-in value objects (e.g. Int, Long, Char, Real, Bool) and their abstract superclasses (e.g. Integer, Number, Object).
- (Higher-order) functions are objects which understand 'evaluate-yourself' messages.
- Classes are also first class objects that understand messages, e.g.to create and initialize their instances. This naturally leads to the concept of metaclasses.

The pure object-oriented language model leads to lean language semantics and inspires a highly object-oriented programming style which makes frequent use of fine-grained abstractions. It should be noted that modern compiler technology eliminates most of the overhead traditionally associated with this approach. Thus, Tycoon encourages programmers to factor code into small, reusable components.

The Tycoon language is strongly and statically typed in the sense that no operation will ever be invoked on an object that does not support it, i.e. errors like 'message not understood' cannot occur at run time, provided the Tycoon type checker is consulted before program

execution (a notable exception is a message sent with the builtin `perform` method which allows messages to be sent with a dynamically computed message selector). Tycoon programs may be executed without prior type checking if rapid prototyping is desired.

Several conventional object models couple the implementation of an object with its type by identifying classes with types. In these models, an object of a certain class cannot be used in a context where another class is required if there is no inheritance relationship between these classes, even though both classes may implement the same interface. Tycoon has adopted a more expressive type system based on conformance. Intuitively, an object conforms to a type when it supports at least the operations required by the type. That is, Tycoon views types as (unordered) sets of method signatures. The additional flexibility of conformance-based typing becomes especially useful when integrating external services in distributed systems.

Similar to CLOS, Tycoon supports multiple inheritance by an ordered specification of multiple superclasses within a class declaration. Possible conflicts are resolved by a linearization of the inheritance tree performed by a topological sort on the superclass lattice. Inheriting from the same class more than once has no effect: Tycoon has no repeated inheritance as in Eiffel or C++.

Tycoon is a reflective and bootstrapped system, i.e. its language processors and development tools are accessible (in a controlled and type-safe manner) to Tycoon applications and these core components are implemented in Tycoon itself.

2 Lexical Entities

Tycoon source code consists of characters in the ISO Latin-1 character set. The source is parsed as a sequence of tokens. At each point, the longest matching token is accepted (so the input `hello` is interpreted as one token, even though every prefix of `hello` is also a valid token). Except for separating tokens, intervening whitespace and single line comments are ignored.

2.1 Literal Constants

Several kinds of immutable objects can be specified as literal constants in the source text. These are numbers, characters, strings and symbols. The syntax used to denote these constants is described in the following sections.

2.1.1 Numbers

Numbers can be distinguished into integers, either 32 or 64 bit wide, called Int and Long respectively, and floating point numbers called Real. All numbers are optionally preceded by a negative sign. To avoid confusion with the binary subtraction operator (minus), it may be necessary to enclose negative number literals in parenthesis.

Int

Instances of the class Int represent 32 bit two's complement integer numbers. Constants can be written in decimal or hexadecimal notation. Decimal numbers consist of a sequence of

decimal digits. Hexadecimal numbers are introduced by the sequence 0x and may consist of decimal digits and the letters a to f in upper or lower case. Constants that cannot be represented as an Int result in a compiler error.

```
 123
0xff
0x5C5C5C5C
```

Long

An instance of the class Long represents a 64 bit two's complement integer number. Long literals have an upper case L appended, otherwise their syntax is identical to Int literals.

```
123L
0xffeeL
0x5C5C5C5C5C5C5C5CL
```

Real

An instance of the class Real represents an IEEE double precision floating point number. One or more digits are optionally followed by a dot and one or more further digits, optionally followed by the exponent. The exponent consists of the letter E in upper or lower case, followed by an optional sign and one or more digits. Either the dot part or the exponent must be present, lest the number be interpreted as an integer. Note that at least one digit is required before and after the dot.

In EBNF, the syntax is as follows:

Digit `::= '0'...'9' ;`
Digits `::=` *Digit* {*Digit*};
Real `::=` *Digits* `'.'` *Digits*
`|` *Digits* `[ '.'` *Digits* `] ('E'|'e') ['+'|'-']` *Digits*

Examples:

```
0.0
1E20
123.456e+77
```

Counter-examples:

```
0.    ; parsed as integer 0 followed by dot
.0    ; parsed as dot followed by integer zero
1.e10 ; parsed as integer 1, dot, identifier 'e10': a message send
```

2.1.2 Char

Tycoon interprets 8 bit character values according to the ISO Latin-1 character set and encoding. Therefore, the Latin-1 mapping of character symbols to 8 bit unsigned integer values

Escape	ASCII Code	Character
\t	7	Horizontal Tab
\n	10	Linefeed (Newline)
\f	11	Form Feed
\r	13	Carriage Return
\'	39	Single Quote
\"	34	Double Quote
\\	92	Backslash
ddd	*ddd* decimal	
\x*hh*	*hh* hexadecimal	

Table 1: Figure 2.1.2: Character Escape Codes

determines the semantics of character comparisons using the '<', '>', '<=', '>=' binary messages (see section 5.1.3, Binary Messages). Future Tycoon versions are expected to support the Unicode character set and encoding which is a superset of ISO Latin-1.

Figure 2.1.2 shows all recognized escape sequences. Any other characters following a backslash result in a syntax error.

Note: In difference to C, no fewer than three decimal digits or two hexadecimal digits are allowed.

Examples:

```
'a'         glyph
'\069'      decimal
'\0xfe'     hex
'\n'        escape character: newline
'\''        escape character: single quote
```

2.1.3 String

A String is an immutable sequence of characters. A string literal is enclosed in double quotes. String literals use the same escape sequences as character literals. The order defined on Strings is induced by the order on characters by lexicographic extension.

String literals may not exceed a single source line. An embedded escape sequence ('\n') may be used for multi-line string literals:

```
"Murkel"
"Dear Mr. Foobar,\n\n\tWelcome to the world of \"Tycoon\""
"\x00\x41\x78\x65\x6c"
```

Note: There are no special terminator characters or size limits.

Since the class String does not export any mutator methods, string literals are immutable and cannot be modified accidentally. Implementations are allowed to share string literal objects from different classes and methods.

2.1.4 Symbol

A Symbol is similar to a String in that it is an immutable sequence of characters, but in each store there is at most one object of class Symbol for each distinct sequence of characters. Thus, if two symbols represent the same character sequence, they are actually the same object. This makes comparing symbols very efficient.

Symbols are denoted as an identifier or a String literal preceded by a hash mark (`'#'`).

```
#murkel
#"murkel II"
```

2.2 Identifiers

The set of allowed characters within an identifier includes the upper case letters `'A'-'Z'`, the lower case letters `'a'-'z'`, the digits `'0'-'9'`, and the underscore character `'_'`. The first character of an identifier must not be a digit. Identifiers are case sensitive.

2.3 Reserved Words

The following words serve a special syntactic purpose and are not interpreted as identifiers:

```
builtin class deferred ensure extern fun Fun import interface meta
old package private require self Self super
```

The words `import`, `interface`, and `package` are reserved for future extensions (see section 6, Proposed Extensions).

2.4 Documentation Strings

Documentation strings are used to attribute explanatory text to class and method definitions. Documentation strings start with `'(*'` and end with `'*)'`. Their content is not transformed in any way. Documentation strings can not be nested.

2.5 Single Line Comments

Within method bodies, the Tycoon code may be annotated with single line comments. Single line comments are treated as whitespace by the grammar. A single line comment starts with a semicolon and ends at the next end of line:

```
m(a :A, b :B) :C
{
  blib  ; we do this because...
  blurb ; we do that because...
  ;; we are done.
}
```

2.6 Other Tokens

The following character sequences are recognized by the Tycoon lexical analyzer as a single token, respectively:

```
{ } [ ] ( ) < <= > >= != !== = == : := + - * / % << >> & && ^ | || ! ^ <: . , #(
```

3 Classes

Class declaration start with the reserved word class, followed by the class name, optionally a list of formal type parameters of the class with their corresponding type bounds, a number of optional declarations, a set of public slot and method definitions, and optionally a set of private slot and method definitions:

```
class C(T1 <:B1, T2 <:B2)
super S1(T2), S2, S3(T1, T2)
(* This is a documentation string
   describing faithfully the whole
   purpose of this class.
*)
meta CClass(C(T1, T2))
Self <: T1
{
  s1 :T1
  m1(t :T1) :T2
    (* This is a documentation string
       describing what this method is
       supposed to do.
    *)
    require t.isGood
    ensure !t.isGood
  {
    s1 := t
    s2
  }
  s2 :T2
  m2 :Int deferred
private
  _m3(i :Int) :Void { nil }
  _s3 :B1
}
```

For a class C with formal type parameters T1 and T2, the following optional declarations are permitted:

- The superclass(es), preceded by the reserved word `super`. In this example: `S1(T2), S2, S3(T1, T2)`. Default is no superclasses.
- The metaclass, preceded by the word meta. Abstract classes (i.e. classes that cannot be instantiated) are usually denoted by the metaclass `AbstractClass`. In this example: `CClass(C(T1, T2))`. Default is `SimpleConcreteClass(C(T1, T2))`.

- ▷ The constraint on the `Self` type. In this example: `Self <: T1`. Default is `Self <: C(T1, T2)`.

The reserved word `Self` denotes the type of instances of the class, including instances of any subclasses. That is, the Self type denotes the type of any potential receiver of the methods specified in this class definition. In general, the Self type will be a subtype of the specified class. This is reflected by the default Self type constraint. However, a different Self type constraint may be specified explicitly. If actual subclassing occurs, the Tycoon type checker verifies that the constraint is satisfied. Examples of explicit Self constraints can be found in the standard system library within classes Ordered and Number.

Method and Slot definitions are surrounded by braces.

3.1 Method Definitions

A method definition begins with the method's signature, optionally followed by a documentation string, a precondition and a postcondition. A precondition describes the circumstances under which the method may be called, i.e. the obligations of the caller. The postcondition describes the obligations of the callee, i.e. the implemented method. See also section 6.1, Pre- and Postconditions, for more.

The body of a method consists of a possibly empty sequence of expressions surrounded by braces (see section 5.3, Sequences). The return value is given by the last element in the sequence. If the sequence is empty, the body is of type `Void`, and the value `nil` is returned. This is legal only for methods having Void as the declared return type.

Within the body, the reserved word `self` denotes the receiver of the message. The type of the receiver `self` is given by `Self` (also a reserved word).

3.2 Slot Definitions

For each slot s of type T, the Tycoon compiler generates a pair of getter and setter methods with the following signatures:

```
s() :T
"s:="(value :T) :Void
```

These getter and setter methods are the only means to access or update the slot. Consequently, any slot getter or setter method may be overridden in subclasses. However, the usual slot access and update notation described in section 5.1.1, Standard Dot Notation, and section 5.1.2, Assignment Messages, is supported:

```
s      ; same as self."s"()
s := v ; same as self."s:="(v)
```

3.3 Private Slots and Methods

The reserved word `private` indicates that the following slots and methods are private to the class C and any subclasses of C. This means that only instances of class C or instances

of subclasses of C are allowed to use these slots and methods, or more accurately: The receiver of messages invoking private methods (including the getter and setter methods of private slots) must be the reserved word `self` or `super` (private message rule). Otherwise, a runtime exception will be raised. The Tycoon type checker is able to detect such errors at compile time.

There is one exception to this private message rule: a metaclasses is allowed to send a message with the selector of a private method to an instance of its instance. In the above example, a method in class `CClass`, the metaclass of class `C`, may access the private slot `_s3` of an instance of class `C`. This is useful for initialization purposes, for example: A `new` method in class `CClass` will want to write its arguments into slots of the newly created instance of `C`, so that `C`'s `_init` method can work with these values.

Any other attempts to send messages with selectors of private methods results in an `DoesNotUnderstand` exception being raised at runtime (if we would distinguish between ordinary `DoesNotUnderstand` exceptions and `PrivateMethodAccess` exceptions, say, the existence of private methods would become observable to other objects, violating encapsulation). The Tycoon type checker detects such errors at compile time.

4 Types

A type is written in one of the following forms:

- An identifier denoting a class or formal type parameter.

  T

- A type operator application, consisting of a type identifier (the operator) followed by a (possibly empty) comma-separated list of type arguments in parenthesis. Any legal type can be used as a type argument.

  Op`(`*T1*`,` *T2*`,` *T3*`)`

  No whitespace is allowed between the operator and the opening parenthesis.

- A function type, consisting of the reserved word Fun followed by a formal parameter list in parentheses, followed by a colon and a return type. See section 5.4, Function Objects, for more information.

  `Fun(`*n1*`:` *T1*`,` *n2*`:` *T2*`):`*T*

5 Values

5.1 Messages

5.1.1 Standard Dot Notation

In general, messages are written in dot notation, i.e. receiver expression and message name (the selector) are separated by a single dot:

o."*m*"

This denotes a message to the object *o* (the receiver) with selector *m*. If the selector is a legal Tycoon identifier (see section 2.2, Identifiers), the surrounding quotes may be omitted, i.e. the above message may also be written as

o.*m*

Furthermore, if the selector is a legal Tycoon identifier, the receiver is also optional. The implied receiver is `self`, i.e. the receiver of the method in which the message expression occurs:

m

is equivalent to

`self`.*m*

provided *m* is not locally bound (see section 4.2, Local Variables). The shortcut is not implied if the selector is enclosed in double quotes:

"*m*"

is NOT interpreted as self.m, but as an ordinary string literal.

A message may take an arbitrary number of arguments. These must be surrounded by parentheses, and the opening parenthesis must follow the message selector without any intervening whitespace:

o.*m*(*a1*, ..., *an*)

Tycoon distinguishes between type and value arguments. Type arguments are preceded by a colon and may be omitted. If omitted, the Tycoon type checker tries to guess the type argument. This process is called type inference. Type arguments are never passed around at runtime. The following message passes two type parameters (`Int`, `Char`) and three value parameters (`1`, `2`, `'c'`):

```
x.foo(:Int, 1, 2, :Char, 'c')
```

Arguments are divided into positional and keyword arguments:

- Positional arguments must be separated by comma, and the corresponding formal parameter of the invoked method is found by counting the number of preceding value arguments, and for type arguments, the number of type arguments given since the last value argument (or from the beginning if there are no preceding value arguments). Positional arguments can be either value or type arguments. Positional value arguments are always required, i.e. the number of arguments must match the number of formal parameters.

- ▷ Following the possibly empty list of positional arguments, any number of keyword arguments may be given. Keyword arguments are introduced by a keyword, consisting of an identifier followed by a colon, and give a value to be passed for that keyword. The corresponding formal parameter is found by searching for a matching keyword identifier in the formal type parameter list. Keyword arguments are always optional. No keyword may appear more than once in the argument list.

The same division applies to the formal parameter list. Positional parameter declarations are separated by commas, and are optionally followed by keyword parameters. The keyword parameters are separated by their respective keywords, so no commas are used. Each keyword is followed by the corresponding parameter name. The parameter's type is introduced by another colon. If the parameter name is omitted, the parameter is anonymous and therefore inaccessible in the method body. An expression computing some default value may be provided for each keyword parameter, otherwise the parameter will default to nil. If the method is not implemented in Tycoon (it is deferred or external), the default value expression only serves documentation purposes.

A method with the signature

```
foo(p1 :P1, p2 :P2
    k1: ka1 :K1
    k2: ka2 :K2) :T
```

may be implemented as

```
foo(p1 :P1, p2 :P2 k1: ka1 :K1 := expr1
                   k2: ka2 :K2) :T
{
  ; may use p1, p2, ka1, and ka2.
  ; ka1 defaults to expr1 and ka2 to nil.
}
```

Such a method may be called with the expression

```
x.foo(v1, v2 k2: v4 k1: v3)
```

The number of positional value arguments supplied by the sender must match the number of positional value parameters specified in the method signature. All keywords given in the message must be expected by the receiver. If either rule is violated, a `WrongSignature` exception is raised at runtime. The Tycoon type checker detects such errors at compile time.

The default value expression of a keyword parameter is evaluated if no matching keyword argument was supplied by the sender. The scope of the default value expression includes the receiver denoted by self, and any preceding type and value parameters. Therefore, the following method declaration is permitted:

```
m(p1 :Int
  k1: ka1 :Int := p1
  k2: ka2 :Int := p1+ka1) :Int {
  ka2
}
```

5.1.2 Assignment Messages

Messages performing some kind of 'assignment' (although this term implies nothing about the actual implementation of the message) may be written in the form

o.*m* := *arg*

Such a message is interpreted as

o."*m*:="(*arg*)

As in standard dot notation, the receiver is implied as self, i.e.

m := *arg*

is equivalent to

self.*m* := *arg*

provided *m* is not locally bound (see section 5.2, Local Variables). By convention, assignment methods have the following signature and postcondition:

```
"m:="(arg :T) :Void
ensure self.m = arg
```

where *T* is the type of the value being assigned.

5.1.3 Binary Messages

Some frequently used messages performing some kind of 'binary operation' (although this term implies nothing about the actual implementation of the message) may be abbreviated in the following form:

o m arg

where *m* is a binary operator, one of:

```
* / % + - << >> < <= > >= = == != !== & ^ |
```

That is, the expression

```
x + y
```

is interpreted as the message

```
x."+"(y)
```

Operator	Standard Library Interpretation
`* / %`	multiplication, division, modulus
`+ -`	addition / concatenation, subtraction
`<< >>`	left shift / output, right shift
`< <= > >=`	comparison
`= == != !==`	equal, identical, nonequal, nonidentical
`&`	logical/binary and
`^`	logical/binary xor (exclusive or)
`|`	logical/binary or
`&&`	lazy and
`||`	lazy or
`:=`	assignment

Table 2: Figure 5.1.3: Operator Precedences

This can be interpreted as a request to the object denoted by x to answer the sum of itself and the argument object denoted by y.

Figure 5.1.3 summarizes the precedence rules for binary messages, lazy binary messages (see section 5.1.4, Lazy Binary Messages), and assignment (see section 5.1.2, Assignment Messages). Operators are listed in decreasing precedence.

Messages on the same line have equal precedence. All operators are left-associative. Note that precedences are the same as in C++/Java. With the exception of the equality/identity operators, the same is true for the standard library interpretation (see section 6, Standard Library Examples). Because of the precedence and associativity rules,

```
1 + 2 * 3 - 4
```

is parsed as

```
(1 + (2 * 3)) - 4
```

which in turn is normalized to

```
1."+"(2."*"(3))."-"(4)
```

5.1.4 Lazy Binary Messages

Lazy binary messages are binary messages that do not always evaluate their second operand. They are normalized in the following way:

p m q

becomes

p`."`*m*`"({ `*q*` })`

Where *m* is one of: `&& ||`

`{ q }` denotes a function object (see section 5.4, Function Objects). The syntax for lazy binary messages facilitates short circuit boolean evaluation.

Operator	Standard Library Interpretation
!	not (logical negation)
~	bitwise negation

Table 3: Figure 5.1.5: Unary Operators

5.1.5 Unary Prefix Messages

There are two unary prefix message, the exclamation mark and the tilde. Unary messages have higher precedence than binary messages. Normalization resolves

```
!p
~i
```

to

```
p."!"
i."~"
```

Again, the standard library interpretation resembles that of C++/Java (see section 6, Standard Library Examples).

5.1.6 Mapping Messages

Messages performing some kind of 'mapping' (although this term implies nothing about the actual implementation of the message) may be written in the form

```
o[arg1, ..., argn]
```

which is equivalent to

```
o."[]"(arg1, ..., argn)
```

Examples of such messages include function application and array indexing. Note that an Array object with element type T may be passed as an argument where a function object (see section 5.4), Function Objects) with a single parameter of type Int and a result type T is expected. This corresponds to the common interpretation of arrays as a finite mapping of integer values to some range type T. Since array objects support many more operations than function objects (e.g. iteration), the reverse is not true.

If the receiving object supports an update of the underlying 'mapping', the form

```
o[arg1, ..., argn] := value
```

is equivalent to

```
o."[]:="(arg1, ..., argn, value)
```

Examples of such messages include MutableArray and MutableString updates.

5.2 Local Variables

Local variables may be declared within the body of a method or function. A variable *x* with type *T* and initial value *v* may be declared as

```
x :T := v
```

If the type *T* is omitted as in

```
x ::= v
```

the type of *x* is inferred as the type of the initial value *v*. If the initial value is omitted, e.g. with

```
x :T
```

the variable *x* is initialized to `nil`.

The value of a local variable *x* may subsequently be changed to a new value *w* using an assignment expression of the form

```
x := w
```

5.3 Sequences

In Tycoon, it is possible to group a sequence of expressions and local bindings into a single value expression by enclosing it in parenthesis. The scope of a local binding extends from the element following the binding to the last element of the sequence. The sequence evaluates to the result of the last expression or binding, and also takes its type. Empty sequences evaluate to `nil` and have type `Void`.

Note that sequences with only one element serve the traditional purpose of resolving precedences between binary operators.

Explicit sequence expressions are most useful with lazy binary messages (see section 5.1.4, Lazy Binary Messages). Most of the time their use is implicit in function expressions (see section 5.4, Function Objects) and method definitions (see section 3.1, Method Definitions).

Examples:

```
a * (b + c)            ;; precedence resolution
a + (x ::= compute()   ;; limit scope of x
     x * x)
!seq.isEmpty && ( el ::= seq[0]   ;; defer evaluation of seq[0]
                  predicate[el] )
```

5.4 Function Objects

Function objects are denoted by the reserved word `fun`, followed by a list of positional parameters enclosed in parantheses, an optional return type declaration, and a body. Like a method body, a function body must be enclosed in braces:

```
fun(base :Int, c :Char) :Int { base + c.asInt }
fun(T <: Object, x :T) { x }
```

If the function return type is omitted, it is inferred from the type of the body. The body implicitly forms a sequence (see section 5.3, Sequences), so the return value and type is given by the last element of the body. A function without declared return type and an empty body has an inferred return type `Void`, and returns the value `nil` when called.

The keyword `fun` and the parameter list may be omitted if the function takes no arguments:

```
{ blub }
```

is equivalent to

```
fun(){ blub }
```

Function objects support an 'evaluate-yourself' method with the selector `"[]"`. This method must be supplied with the actual parameters, and it will return the function result. For example, the above function objects have methods with the following signatures and bodies:

```
"[]"(base :Int, c :Char) :Int { base + c.asInt }
"[]"(T <: Object, x :T) :T { x }
```

Assuming the above functions are denoted by the variables `f1` and `f2`, these functions may be invoked by the following code sequence:

```
f1[13, 'a']
f2[i]
```

Recall that this code sequence is equivalent to the following:

```
f1."[]"(13, 'a')
f2."[]"(i)
```

The type of a function object may be denoted with the reserved word `Fun`, followed by a list of formal parameters enclosed in parenthesis, and a colon followed by the return type. The example functions above have the following types, respectively:

```
Fun(base :Int, c :Char):Int
Fun(T <: Object, x :T):T
```

These notations represent types equivalent to the type defined by a direct subclass of Object with the single public method

```
"[]"(base :Int, c :Char):Int deferred
```

or

```
"[]"(T <: Object, x :T):T deferred
```

respectively.

5.5 Array Constructors

There is a built-in Array constructor. The expression

`#(`*el1*`, ..., `*eln*`)`

creates an Array object with elements obtained by evaluating *el1* to *eln* in order. Since the class Array does not support any update methods, the resulting array object cannot be modified.

The array constructor facilitates aggregation without having to count the number of items to aggregate in the source text, which is often a cumbersome and error-prone task.

6 Proposed Extensions

6.1 Pre- and Postconditions

In Tycoon, each method has an optional precondition and an optional postcondition.

The precondition is an ordinary value expression. It is in scope of the method's arguments, and is evaluated when the method is called, after default values for keyword parameters have been computed. If it evaluates to false, a `PreconditionFailed` exception is raised.

The postcondition is a value expression that may contain `old` expressions. These serve to compare the state before and after execution of the method. Syntactically, `old` behaves like a unary prefix operator. The operands of all `old` expressions occurring in a method's postcondition are evaluated at method entry, after the precondition has been checked. They are in the same scope as the precondition. When checking the postcondition, the obtained results are inserted in place of the `old` expressions. The postcondition is in the same scope as the precondition, with the addition of an immutable identifier `result` of the method's result type, denoting the method's return value. The postcondition is checked after the method body has been executed. If the postcondition evaluates to false, a `PostconditionFailed` exception is raised.

Runtime checking of pre- and postconditions can be disabled using a compiler switch. The conditions will still be typechecked.

Note that since pre- and postconditions are ordinary Tycoon expressions, they can have side effects, and do not necessarily terminate. So switching debug mode on or off may alter the behaviour of your program.

Note also that an `old` expression may not refer to identifiers bound locally inside the post-condition, because it is evaluated long before the postcondition proper is reached. Phrased differently, `old` expressions are scoped statically, but fall outside block scoping.

Just like a method's signature and documentation string, pre- and postconditions have to be written anew at each redefinition of the method. As with documentation strings, the system neither enforces nor supports a relation to any inherited definitions. However, the programmer should strive to make the implementation substitutable for all superclasses' definitions, in that the method's precondition must be equal to or weaker than those of the superclasses, and the postcondition must be equal or stronger.

6.2 Packages

Packages are a mechanism to structure the global name space (containing interfaces and classes) hierarchically. They allow developers to name their classes without worrying about possible name conflicts with classes in other packages.

In each store, there is a global tree of packages, with classes as leaves. A class is assigned to a package by preceding the class definition with a `package` *fullyQualifiedPackageName* statement. A fully qualified package name is a dot-separated list of package names, which defines a path from the root of the package tree.

The package statement may be followed by an import statement `import` *importList*, where the import list is a comma-separated list of fully qualified package names to be searched when resolving identifiers. The current package is implicitly added at the beginning of the list.

A type identifier is defined to be a dot-separated list of identifiers. The first component of this list is searched for in the import list. If there is more than one match, the compiler gives an error. The following components of the type identifier denote subpackages, and the last component denotes the class whose interface is denoted by the type identifier. The compiler gives an error if any intermediate component does not exist or is not a package, or if the final component is not a class.

Note that packages can not be used as types (what type should that be?) Note also that following components are not used to disambiguate the identifier, in symmetry with the value case below, and in order not to introduce too many dependencies

Value identifiers (or syntactically equivalent, implicit self sends without arguments) are extended in a similar way. If the value identifier is not bound locally, it is searched for in the import list as described above. If it is found to denote a package, the identifier has to be the receiver in a zero-argument message send, the selector of which is interpreted as the name of a subpackage or of a class contained in the package. In the case of a subpackage, the process is repeated, otherwise the identifier is resolved as referring to the class object.

If the identifier is neither bound locally nor found in the import list, it is interpreted as a zero-argument message to self.

In effect, value identifiers referring to classes are defined as dot-separated lists of components, starting at some point in the import list and ending at a class, just like type identifiers.

Note: The process of identifer resolution will have to be redone if any path component of the resolved name is removed, or if a class or package with the name of the first component is inserted somewhere else in the package hierarchy, or whenever the import list is changed

6.3 Inner Classes

An inner class is a class declared inside the scope of another class. Inner classes have the following advantages:

- they are allowed to access the private methods (and therefore, also the private slots) of their lexically enclosing class(es). In most cases, this makes a language mechanism similar to the C++ friend mechanism obsolete.

- ▷ they bridge the gap between function objects and instances of ordinary (top-level) classes. A `fun` expression is simply a syntactic abbreviation for an anonymous inner class declaration and a subsequent new message, just as a `Fun` type is equivalent to an anonymous interface type.

Example: A Hypothetical Implementation of Sequence::select

```
select(pred :Fun(e:E):Bool) :Reader(E)
{
  outer ::= self    ; we need a way to access the outer self in inner classes!
  class Local
  super Reader(E)
  meta SimpleConcreteClass(_)
  {
    isEmpty :Bool
    { pos = outer.size }
    read :E
    { let e = outer[pos]
      findNext
      e }
  private
    pos :Int
    findNext {
      pos := pos + 1
      if( pos < outer.size then: {
        if( !pred[outer[pos]] then: {   ; block scoping for pred
          findNext
        })
      })
    }
    _init {
      super._init
      pos := -1
      findNext
    }
  }
  Local.new
}
```

7 System Library Library Examples

The following examples should give a general impression of the look and feel of Tycoon code. Section 7.2, Control Structures, shows the interaction of keyword parameters and function objects to produce "add-on" syntax for control structures. Section 7.3, Arithmetic, shows the use of F-bounded parametric polymorphism and the Self type for typing binary methods. Section 7.4, Strings and Symbols, gives applications of the various techniques: optional arguments, typecase, control structures and tail recursive loops. Section 7.5, Output, shows the use of dynamic dispatch to implement a C++-like << operator without static overloading.

7.1 What about nil, true and false?

The 'constants' `nil`, `true` and `false` are not built into the language, but are available in the standard system library as private, builtin methods in class Object:

```
class Object
meta AbstractClass
{
...
private

  true :Bool  builtin
  false :Bool builtin
  nil :Nil builtin
}
```

This way, in every method, you may simply use the identifiers `nil`, `true` and `false`. These identifiers are interpreted as a message to self according to section 5.1.1, Standard Dot Notation. Thus, the corresponding methods in class `Object` are called. Under the hood, they query the Tycoon system about the `nil`, `true` and `false` objects and simply return them to the caller. Concrete implementations may restrict the redefinition of these methods in order to prevent too much confusion.

7.2 Control Structures

7.2.1 Object Protocol

Every object understands the following messages:

```
class Object
meta AbstractClass
{
"=="(other :Object) :Bool
  (* object identity test *)
  builtin

"!=="(other :Object) :Bool
  (* object identity test *)
{
  !(self == other)
}
"="(other :Object) :Bool
  (* Object equality. Default implementation uses object identity *)
{
  self == other
}
"!="(other :Object) :Bool
  (* object identity test *)
{ !(self = other) }

clazz :Class
  (* answer the receiver's class object *)
   builtin
```

```
isNil :Bool
  {
  false
  }

isNotNil :Bool
  {
  true
  }

perform(selector :Symbol, arguments :Array(Object)) :Object
  builtin
}
```

The `isNil` and `isNotNil` methods are overridden in class `Nil` (the class of `nil`) to answer `true` and `false`, respectively.

7.2.2 Boolean Protocol

Interface and implementation:

```
class Bool
super Object
meta AbstractClass
{
  case(T <: Void
     true: :Fun():T
     false: :Fun():T) :T
    (* used to implement if *)
    deferred
  "!" :Bool deferred
  "&"(aBool :Bool) :Bool deferred
  "^"(aBool :Bool) :Bool deferred
  "|"(aBool :Bool) :Bool deferred
  "&&"(ifTrue :Fun():Bool) :Bool deferred
  "||"(ifFalse :Fun():Bool) :Bool deferred
}

class True
super Bool
meta OddballClass
{
  case(T <: Void
     true: ifTrue :Fun():T := { nil }
     false: :Fun():T) :T {
    ifTrue[]
  }
  "!" :Bool {
    false
  }
  "&"(aBool :Bool) :Bool {
    aBool
  }
  "^"(aBool :Bool) :Bool {
```

```
    !aBool
  }
  "|"(aBool :Bool) :Bool {
    true
  }
  "&&"(ifTrue :Fun():Bool) :Bool {
    ifTrue[]
  }
  "||"(ifFalse :Fun():Bool) :Bool {
    true
  }
}

class False
super Bool
meta OddballClass
{
  case(T <: Void
     true: :Fun():T
     false: ifFalse :Fun():T := { nil }) :T {
    ifFalse[]
  }
  "!" :Bool {
    true
  }
  "&"(aBool :Bool) :Bool {
    false
  }
  "^"(aBool :Bool) :Bool {
    aBool
  }
  "|"(aBool :Bool) :Bool {
    aBool
  }
  "&&"(ifTrue :Fun():Bool) :Bool {
    false
  }
  "||"(ifFalse :Fun():Bool) :Bool {
    ifFalse[]
  }
}
```

Due to the special message syntax, logical tests can be written in a style familiar to C++/Java programmers, even though they can be explained using message passing semantics. An example:

```
!x.isNil && (x > bound || aSet.includes(x))
```

Note that the case message defined here is a variant of the Visitor pattern. The pattern can be used in similar situations to define a 'poor man's typecase':

```
class AbstractXYZ
meta AbstractClass
{
  case(T <:Object
    ifX: :Fun(x :X):T
```

```
    ifY: :Fun(y :Y):T
    ifZ: :Fun(z :Z):T
  ) :T deferred
}

class X
super AbstractXYZ
{
  case(T <:Object
      ifX: :Fun(x :X):T := fun(:X){nil}
      ifY: :Fun(y :Y):T := nil
      ifZ: :Fun(z :Z):T := nil
    ) :T
  {
  ifX[self]
  }
}

;; etc.
```

7.2.3 Loops and Conditionals

Usage:

```
if(i.isOdd then: {
  out << "odd stuff!"
})

if(i < 10 then: {
  out << "less"
} else: {
  out << "greater or equal"
})

i::=0
while({ i<10 } do: {
  i := i+1
})
repeat({
  i := i-1
} until: { i = 0 })
for(0 to: 9 do: fun(i:Int) {
  ...
})
for(0.0 to: 0.9 step: 0.1 do: fun(r:Real) {
  ...
})
```

Interface and Implementation:

```
class Object
meta AbstractClass
{
...
private
```

```
  if(T <: Void, condition :Bool
     then: then :Fun():T := { nil }
     else: else :Fun():T := { nil }) :T
  {
    condition.case(true: then false: else)
  }

  while(condition :Fun():Bool
        do: body :Fun():Void := { }) :Void
  {
    if( condition[] then: {
      body[]
      while(condition do: body)
    })
  }

  repeat(statement :Fun():Void := { }
         until: terminate :Fun():Bool := { false }) :Void
  {
    statement[]
    if( !terminate[] then: {
      repeat(statement until: terminate)
    })
  }

  for(N <:Number(N), lower :N
        to:   upper :N := lower
        step: step  :N := lower.one
        do:   body  :Fun(i:Int):Void := fun(:Int){}) :Void
  {
    if( lower <= upper then: {
      body[lower]
      for(lower+step to: upper step: step do: body)
    })
  }
}
```

Note that the `if` message is just syntactic sugar for the `case` message defined in class Boolean.

7.2.4 Safe Downcast

Usage:

```
typecase(x
  case: fun(x :T1) { x.t1 }
  else: {
    typecase(x
       case: fun(x :T2) { x.t2 }
       else: {
           ...
       })
   })
```

Interface:

```
class Object
meta AbstractClass
{
...
private
  ...
  typecase(T <:Object, x :Object
             case: caseHandler :Fun(x :Nil):T := nil
             else: defaultHandler :Fun():T := { nil}
           ) :T
    {
    ...
    }
}
```

Note that any function with a single parameter may be passed to the `typecase` method (provided it has the right return type) because of contravariant subtyping of method (and, therefore, function) parameters.

The implementation in the body of the `typecase` method (code not shown) utilizes the reflective capabilities of Tycoon by requesting the argument type from its first actual function argument (the `case:` keyword parameter) and performing a dynamic subtype test with the value parameter `x`. If the subtype test succeeds, the `case:` function is invoked. Otherwise, the `else:` function is invoked. The precise semantics of dynamic subtype tests will be described in a separate document.

7.2.5 Exception Handling

Usage:

```
try(expr
  catch: fun(e :IOError) {
    e.msg.print
  }
  else: fun(e :Exception) {
    e.clazz.name.print
 })
```

Interface and implementation:

```
class Exception
super Object
meta AbstractClass
{
raise :Nil
  (* Nil is the bottom type of Tycoon *)
  builtin
}

class Object
meta AbstractClass
```

```
{
...
private

  try(T <:Object, E <:Exception, expr :Fun():T
        catch: catchHandler :Fun(e :E):T := nil
        else:  defaultHandler :Fun(e :Exception):T
                              := fun(e :Exception) { e.raise }
      ) :T
  {
    ...
  }
}
```

Concrete exception classes must be subclasses of Exception (otherwise they do not inherit the raise primitive and, consequently, cannot be thrown at runtime!).

The implementation of the try method (code not shown) uses another private primitive (builtin) method `_try` to catch any exceptions which might occur while evaluating `expr`. A typecase message is then sent to check whether the type of the exception is a subtype of the argument of the `catchHandler` function object. If it is, the `catch:` function is invoked with the exception object (by sending it the `"[]"` message). Otherwise, the `defaultHandler` function is invoked with the exception object. If the `defaultHandler` function is not provided by the sender, the default implementation re-raises the exception by sending it another `raise` message.

7.3 Arithmetic

The numeric class hierarchy uses F-Bounded Self typing to support binary methods in a type-safe manner:

```
class Number(F <:Number(F))
super Ordered(F)
Self <: F
meta AbstractClass
{
  "+"(x :F) :F  deferred
  "-"(x :F) :F  deferred
  "*"(x :F) :F  deferred
  "/"(x :F) :F  deferred
  ...
  zero :Self  deferred
  one  :Self  deferred
  negated :F {
    zero - self
  }
  ...
}

class Integer(F <:Integer(F))
super Number(F)
Self <:F
meta AbstractClass
{
```

```
  "&"(x :F) :F
    (* bitwise and *)
    deferred
  "|"(x :F) :F
    (* bitwise or *)
    deferred
  "^"(x :F) :F
    (* bitwise exclusive or *)
    deferred

  "~" :F
    (* bitwise inversion *)
  {
    ;; assumes two's complement representation
    negated - one
  }

  even :Bool {
   (self & one) = zero
  }

  odd :Bool {
    !even
  }
  ...
}

class Int
super Integer(Int)
Self = Int
{
  _brand_Int :Void
    (* prevent unintentional subtyping with Long *)
    { }

  "+"(x :Int) :Int  builtin
  "-"(x :Int) :Int  builtin
  "*"(x :Int) :Int  builtin
  ...
}

class Long
super Integer(Long)
Self = Long
{
  _brand_Long :Void
    (* prevent unintentional subtyping with Int *)
    { }

  "+"(x :Long) :Long  builtin
  "-"(x :Long) :Long  builtin
  "*"(x :Long) :Long  builtin
  ...
}
```

7.4 Strings and Symbols

```
class String
super Array(Char), OrderedSequence(Char, String)
meta StringClass
{
  "+"(aString :String) :String
    (* String concatenation *)
  { ... }

  "="(x :Object) :Bool {
    self == x
     || typecase(x, fun(s :String) {
           ;; subclass of string, i.e. String, MutableString or Symbol
           stringEqual(s)
       } else: {
           false
       })
  }

  stringEqual(s :String) :Bool { ... }

  locateChar(ch :Char startingAt: i:Int := 0 before: limit :Int := size) :Int
    (* for the indices at which the receiver contains a character
       equal to 'ch', answer the lowest index which
       is greater than or equal to 'startingAt'.
       answer nil if no such index exists.
     *)
  {
    if( i >= limit then:{ nil } else: {
      if( _elementEqual(self[i], ch) then: {
        i
      } else: {
        locateChar(ch startingAt: i+1 before: limit)
      })
    })
  }

  locateLastChar(ch :Char
                 startingAt: i :Int := 0
                 before: limit :Int := size ) :Int
    (* for the indices at which the receiver contains a character
     equal to 'ch', answer the highest index which
     is less than 'before'.
     answer nil if no such index exists.
    *)
  {
    ...
  }

  startsWith(other :String) :Bool
    (* answer true if 'other' is a prefix of the receiver,
       else answer false
    *)
  {
    size >= other.size
     && subStringEqual(n: other.size at: 0 equals: other)
```

```
  }
  endsWith(other :String) :Bool { ... }

  subStringEqual(n: n :Int
                 at: at :Int
                 equals: other :String
                 startingAt: otherAt :Int := 0) :Bool
    (* answer true if the n character sequence at..at+n-1 of
       the receiver is equal to the n character sequence
       otherAt..otherAt+n-1 of 'other',
       else answer false
       arguments for n:, at: and equals: have to be supplied.
    *)
    require n.isNotNil & at.isNotNil & equals.isNotNil
  {
    n <= 0
     || (self[at] == other[otherAt]
          && subStringEqual(n: n-1 at: at+1
                  equals: other startingAt: otherAt+1))
  }
}


class Symbol
super String
meta SymbolClass
{
  _brand_Symbol :Void { }

  "="(x :Object) :Bool {
    self == x
     || typecase(x case: fun(sym :Symbol) {
          ;; no need to check since equality is identity
          false
        } else: {
          typecase(x case: fun(s :String) {
            ;; subtype of string, i.e. String or MutableString
            stringEqual(s)
          } else: {
            false
          })
        })
  }
  ...
}
```

7.5 Output

Usage:

```
tycoon.stdout << "Dear Mr. "<<name<<",\n"
              << "  We are very "<<howAreWe<<" to hear about your "
              << eventType<< ".\n"
```

Interface and implementation:

```
class Output
super Writer(Char)
meta AbstractClass
{
  ...
  "<<"(x :Object) :Output
  {
    x.writeOn(self)
    self
  }

  writeBuffer(s :String start: start :Int := 0
                        n: n :Int := s.size-start)
    deferred
}


class Writer(E <: Object)
super Object
meta AbstractClass
{
  write(e :E)
    (* write the given object into the receiver *)
    deferred

  writeAll(aCollection :Collection(E))
    (* write all objects in aCollection  into the receiver.
       Concrete subclasses may give a more efficient implementation
       using double dispatch or typecase. *)
  {
    aCollection.do(fun(e :E){
      write(e)
    })
  }

  flush { }
  close { }
}

class Object
meta AbstractClass
{
  writeOn(out :Output)
    (* append the pure 'value' or 'contents' of the receiver to the given
       output stream. this method is called by Output:"<<".
       the default implementation uses printOn(), subclasses should override
       it appropriately. This is done, for example, by String and Char in order
       to strip the quotes.
     *)
  {
    printOn(out)
  }

  printOn(out :Output)
    (* append to the given output stream a textual representation that
       describes the receiver. default implementation, subclasses should
       override it appropriately.
    *)
```

```
  {
    className :String := self.clazz.name
    out << if(className[0].isVowel then: {"an "} else: {"a "}) << className
  }

  printString :String
    (* To obtain a string from the printed representation of the receiver,
       print it on a string and answer the printed subsequence. *)
  {
    buffer ::= StringBuilderOutput.new(capacity: 10)
    printOn(buffer)
    buffer.contents
  }

  print
  {
    printOn(tycoon.stdout)
  }
}

class String ...
{
  ...
  writeOn(out :Output) {
    out.writeBuffer(self)
  }

  printOn(out :Output) {
    out.write('\"')
    do(fun(ch :Char) {
      ch.printEscapeSequence(out)
    })
    out.write('\"')
  }
}
```

8 EBNF Syntax

For this grammar, we extend the Extended Backus Naur Form further:

$$\{X,\}$$

is a shortcut for

$$[X\ \{\text{','}\ X\}]$$

```
(* Values *)

Value ::=
'self'
| string | char | Number | symbol
| '(' Sequence ')'
| Value BinOp Value
```

```
| Value LazyBinOp Value
| UnaryOp Value
| 'old' Value (* ### only allowed in postconditions *)
| (Value|'super') '.' Selector [ Arguments ]
| identifier [ Arguments ]
| (Value|'super') '.' Selector ':=' Value
| identifier ':=' Value
| Value '[' PositionalArguments ']'
| Value '[' PositionalArguments ']' ':=' Value
| [ 'fun' '(' Signatures ')' [ ':' Type ] ] '{' Sequence '}'
(* implementation of fun parameters currently restricted to
ValueSignatures *)
| '#(' {Value,} ')'
;
Sequence ::=
{Value|Binding}
;
Number ::=
['-'] (int | long | real)
;
Selector ::=
identifier | string
;
Arguments ::=
'(' PositionalArguments KeywordArguments ')'
;
PositionalArguments ::=
{PositionalArgument,}
;
PositionalArgument ::=
Value | ':' Type
;
KeywordArguments ::=
{ Keyword Value }
;
Keyword ::=
identifier ':'
;
Binding ::=
identifier ':' [Type] ':=' Value (* -> ValueSignature *)
| identifier ':' Type
;
LazyBinOp ::=
'&&' | '||'
;
BinOp ::=
'=' | '==' | '!=' | '!=='
| '>' | '<' | '>>'
| '>=' | '<=' | '<<'
| '^' | '&' | '|'
```

```
| '+' | '-' | '*' | '/' | '%'
;
UnaryOp ::=
'!' | '~'
;

(* Types *)

Type ::=
'Self'
| identifier [ '(' { Type,} ')' ]
| 'Fun' '(' Signatures ')' ':' Type
(* implementation currently restricted to ValueSignatures *)
;

(* Signatures *)
Signature ::=
ValueSignature
| TypeSignature
;
ValueSignature ::=
[identifier] ':' Type
;
TypeSignature ::=
identifier '<:' Type
| identifier '=' Type
;
Signatures ::=
{Signature,}
;
MethodSignature ::=
Selector [ '(' Signatures KeywordSignatures ')' ] ':' Type
;
KeywordSignatures ::=
{ Keyword ValueSignature }
;

(* Classes and Methods *)
Class ::=
'class' ClassNameDomain
[ 'super' {Type,} ]
[ documentationString ]
[ SelfSignature ]
[ 'meta' Type ]
'{' { MethodDefinition | SlotDefinition }
[ 'private'
{ MethodDefinition | SlotDefinition } ] '}'
;
ClassNameDomain ::=
identifier
| identifier '(' {Type,} ')'
;
```

SelfSignature ::=
'Self' '<:' *Type*
| 'Self' '=' *Type*
;
SlotDefinition ::=
Selector ':' *Type*
[documentationString]
;
MethodDefinition ::=
Selector ['(' *Signatures KeywordParameters* ')'] ':' *Type*
[documentationString]
['require' *Value*]
['ensure' *Value*]
MethodBody
;
KeywordParameters ::=
{ *Keyword ValueSignature* [':=' *Value*]}
;
MethodBody ::=
'{' *Sequence* '}'
| 'builtin' ['{' *Sequence* '}']
| 'deferred'
| 'extern' *ForeignName*
;
ForeignName ::=
string
;

When ambiguities between the sign '-' or '+' and the binary operator '-' or '+' arise, the parser decides in favor of the binary operator. Thus, to return a value of -1 as the last element of a sequence, you have to put the minus one in parentheses:

```
{
  foo(mumble bar: humble friend: goo)
  (-1)
}
```

Otherwise, the expression will be interpreted as a subtraction and will presumably result in a type error.

No whitespace is allowed between a method selector and the opening parenthesis of the argument list. This is to distinguish a message with arguments from a sequence of a message without arguments and an expression in parenthesis:

```
{
   foo(-1)    ; message foo with argument -1
   foo (-1)   ; message foo without argument, followed by return value -1
}
```

No whitespace is allowed between a type operator and the opening parenthesis of the argument type list. This is to disambiguate cases like the following:

```
{
```

```
  x :T
  (Array)
}
```


9 Changes from Tycoon Version 0.9 to Version 1.0

9.1 Syntactic Changes

The surface syntax of Tycoon has been completely revised. Changes:

- ▷ introduction of `Fun` types
- ▷ removal of whitespace between operators and argument lists
- ▷ reserved word `metaclass` changed to `meta`
- ▷ removed class invariants
- ▷ missing method bodies disallowed
- ▷ comments organized into documentation strings and single-line comments
- ▷ changed syntax for grouping slots and methods
- ▷ removed commata between expressions in a sequence
- ▷ changed syntax for conditional (`if` or `?:`)
- ▷ removed special syntax for assertions (`assert`)
- ▷ changed syntax for local bindings (removed keyword `let`)
- ▷ removed special syntax for implications (`=>`, `=>=>`)
- ▷ eager conjunction and disjunction (`&`, `|`) have higher precendece than lazy conjunction and disjunction (`&&`, `||`), as in C++/Java

Syntactic features newly introduced in version 1.0:

- ▷ keyword arguments
- ▷ lazy binary operators
- ▷ added C++/Java operators $\sim$ and `^`

A translation tool which automatically converts Tycoon programs from the 0.9 syntax to 1.0 syntax is available at Higher-Order. This tool also converts standard control structures (see section 9.3, Standard Library Changes). All remaining changes will be reported by the Tycoon type checker.

9.2 Semantic Changes

Assignments no longer return the value they assigned, but return the value `nil` with static type `Void` (actually meaning 'no return value'). This change allows the clean separation of covariant accessor interfaces from contravariant mutator interfaces.

The introduction of keyword arguments extends the syntax as well as the semantics of Tycoon.

9.3 Standard Library Changes

The looping constructs and conditionals have been rewritten using keyword syntax.

The _init Method in Object is no longer declared with return type Self, but with a Void return type.