The Sather 1.1 Specification

David Stoutamire[1]   Stephen Omohundro

TR-96-012

August 18, 1996

Abstract

This document is a concise specification of Sather 1.1. Sather is an object oriented language designed to be simple, efficient, safe, flexible and non-proprietary. Sather has parameterized classes, object-oriented dispatch, statically-checked strong (contravariant) typing, separate implementation and type inheritance, multiple inheritance, garbage collection, iteration abstraction, closures, exception handling, assertions, preconditions, post-conditions, and class invariants.

This 1.1 specification significantly polishes and improves the 1.0 language specification with an introduction, index, and examples. New constructs include `out' arguments, less restrictive overloading, and improved external language interfaces.

---

1. Direct email correspondence to the Sather group at `sather@icsi.berkeley.edu`

This page intentionally blank.

ABOUT SATHER

INTRODUCTION 9

IMPORTANT CONCEPTS 10

USING SATHER 14

HISTORY 15

THE SATHER 1.1 SPECIFICATION

INTRODUCTION 18

LEXICAL STRUCTURE 19

Basic Expressions 43

Loops and Iterators 51

Closures 54

Exceptions 57

Safety Features 58

Distributed Extension 83

About Sather

1.1 INTRODUCTION

Sather is an object oriented language designed to be simple, efficient, safe, and non-proprietary. It aims to meet the needs of modern research groups and to foster the development of a large, freely available, high-quality library of efficient well-written classes for a wide variety of computational tasks. It was originally based on Eiffel but now incorporates ideas and approaches from several languages. One way of placing it in the 'space of languages' is to say that it attempts to be as efficient as C, C++, or Fortran, as elegant but safer than Eiffel or CLU, and to support higher-order functions as well as Common Lisp, Scheme, or Smalltalk.

Sather has garbage collection, statically-checked strong (contravariant) typing, multiple inheritance, separate implementation and type inheritance, parameterized classes, dynamic dispatch, iteration abstraction, higher-order routines and iters, exception handling, assertions, preconditions, postconditions, and class invariants. Sather code can be compiled into C code and can efficiently link with object files of other languages. pSather, the parallel and distributed extension, presents a shared memory abstraction to the programmer while allowing explicit placement of data and threads.

Sather has a very unrestrictive license aimed at encouraging contribution to the public library without precluding the use of Sather for proprietary projects.

1.1.1 *The Name*

Sather was developed at the International Computer Science Institute, a research institute affiliated with the computer science department of the University of California at Berkeley. The Sather language gets its name from the Sather Tower (popularly known as the Campanile), the best-known landmark of the campus. A symbol of the city and the university, it is the Berkeley equivalent of the Golden Gate bridge across the bay. Erected in 1914, the tower is modeled after St. Mark's Campanile in Venice, Italy. It is smaller and a bit younger than the Eiffel tower. The way most people say the name of the language rhymes with 'bather'.

The name 'Sather' is a pun of sorts - Sather was originally envisioned as an efficient, cleaned-up alternative to the language Eiffel. However, since its conception the two languages have evolved to be quite distinct.

1.2 IMPORTANT CONCEPTS

This section briefly introduces some concepts important to Sather that the reader may not have been exposed to in C++ [2]. It isn't meant as a complete language tutorial. More information of a tutorial nature is available from the WWW page:

http://www.icsi.berkeley.edu/Sather

1.2.1 *Garbage Collection and Checking*

Like many object-oriented languages, Sather is *garbage collected*, so programmers never have to free memory explicitly. The runtime system does this automatically when it is safe to do so. Idiomatic Sather applications generate far less garbage than typical Smalltalk or Lisp programs, so the cost of collecting tends to be lower. Sather does allow the programmer to manually deallocate objects, letting the garbage collector handle the remainder. With checking compiled in, the system will catch dangling references from manual deallocation before any harm can be done.

More generally, when checking options have been turned on by compiler flags, the resulting program cannot crash disastrously or mysteriously. All sources of errors that cause crashes are either eliminated at compile-time or funneled into a few situations (such as accessing beyond array bounds) that are found at run-time precisely at the source of the error.

1.2.2 *No Implicit Calls*

Sather does as little as possible behind the user's back at runtime. There are no *implicitly* constructed temporary objects, and therefore no rules to learn or circumvent. This extends to class constructors: all calls that can construct an object are explicitly written by the programmer. In Sather, constructors are ordinary routines distinguished only by a convenient but optional calling syntax (page 47). With garbage collection there is no need for destructors; however, explicit finalization is available when desired (page 62).

Sather never converts types implicitly, such as from integer to character, integer to floating point, single to double precision, or subclass to superclass. With neither implicit construction nor conversion, Sather resolves routine overloading (choosing one of several similarly named operations based on argument types) much more clearly than C++. The programmer can easily deduce which routine will be called (page 45).

In Sather, the redefinition of operators is orthogonal to the rest of the language. There is ''syntactic sugar'' (page 49) for standard infix mathematical symbols such as '+' and '^' as calls to otherwise ordinary routines with names 'plus' and 'pow'. 'a+b' is just another way of writing 'a.plus(b)'. Similarly, 'a[i]' translates to 'a.aget(i)' when used in an expression. An assignment 'a[i] := expr' translates into 'a.aset(i,expr)'.

1.2.3 *Separation of Subtyping and Code Inclusion*

In many object-oriented languages, the term 'inheritance' is used to mean two things simultaneously. One is *subtyping*, which is the requirement that a class provide implementations for the abstract methods in a supertype. The other is code inheritance (called *code inclusion* in Sather parlance) which allows a class to reuse a portion of the implementation of another class. In many languages it is not possible to include code without subtyping or vice versa.

Sather provides separate mechanisms for these two concepts. *Abstract classes* represent interfaces: sets of signatures that subtypes of the abstract class must provide. Other kinds of classes provide implementation. Classes may include implementation from other classes using a special 'include' clause; this does not affect the subtyping relationship between classes. Separating these two concepts simplifies the language considerably and makes it easier to understand code. Because it is only possible to subtype from abstract classes, and abstract classes only specify an interface without code, sometimes in Sather one factors what would be a single class in C++ into two classes: an abstract class specifying the interface and a code class specifying code to be included. This often leads to cleaner designs.

Issues surrounding the decision to explicitly separate subtyping and code inclusion in Sather are discussed in the ICSI technical report TR 93-064: ''Engineering a Programming Language: The Type and Class System of Sather,'' also published as [7]. It is available at the Sather WWW page.

1.2.4 *Iterators*

Early versions of Sather used a conventional 'until...loop...end' statement much like other languages. This made Sather susceptible to bugs that afflict looping constructs. Code which controls loop iteration is known for tricky ''fencepost errors'' (incorrect initialization or termination). Traditional iteration constructs also require the internal implementation details of data structures to be exposed when iterating over their elements.

Simple looping constructs are more powerful when combined with heavy use of *cursor* objects (sometimes called 'iterators' in other languages, although Sather uses that term for something else entirely) to iterate through the contents of container objects. Cursor objects can be found in most C++ libraries, and they allow useful iteration abstraction. However, they have a number of problems. They must be explicitly initialized, incremented, and tested in the loop. Cursor objects require maintaining a parallel cursor object hierarchy alongside each container class hierarchy. Since creation is explicit, cursors aren't elegant for describing nested or recursive control structures. They can also prevent a number of important optimizations in inner loops.

An important language improvement in Sather 1.0 over earlier versions was the addition of *iterators*. Iterators are methods that encapsulate user defined looping control structures just as routines do for algorithms. Code using iterators is more concise, yet more readable than code using the cursor objects needed in C++. It is also safer, because the creation, increment, and termination check are bound together inviolably at one point. Each class may define many sorts of iterators, whereas a traditional approach requires a different yet intimately coupled class for each kind of iteration over the major class. Sather iterators are part of the class interface just like routines.

Iterators act as a lingua-franca for operating on collections of items. Matrices define iterators to yield rows and columns; tree classes have recursive iters to traverse the nodes in pre-order, in-order, and post-order; graph classes have iters to traverse vertices or edges breadth-first and depth-first. Other container classes such as hash tables, queues, etc. all provide iters to yield and sometimes to set elements. Arbitrary iterators may be used together in loops with other code.

The rationale of the Sather iterator construct and comparisons with related constructs in other languages can be found in the ICSI technical report TR 93-045: ''Sather Iters: Object-Oriented Iteration Abstraction,'' also published as [5]. It is available at the Sather WWW page.

1.2.5 *Closures*

Sather provides higher-order functions through *method closures*, which are similar to closures and function pointers in other languages. These allow binding some or all arguments to arbitrary routines and iterators but defer the remaining arguments and execution until a later time. They support writing code in an applicative style, although iterators eliminate much of the motivation for programming that way. They are also useful for building control structures at run-time, for example, registering call-backs with a windowing system. Like other Sather methods, method closures follow static typing and behave with contravariant conformance.

1.2.6 *Immutable and Reference Objects*

Sather distinguishes between reference objects and immutable objects. Imutable objects never change once they are created. When one wishes to modify an immutable object, one is compelled to create a whole new object that reflects the modification.

Experienced C programmers immediately understand the difference when told about the internal representation the ICSI compiler uses: immutable types are implemented with stack or register allocated C 'struct''s while reference types are pointers to the heap. Because of that difference, reference objects can be referred to from more than one variable (*aliased*), but immutable objects never appear to be. Many of the built-in types (integers, characters, floating point) are immutable classes. There are a handful of other differences between reference and immutable types; for example, reference objects must be explicitly allocated, but immutable objects 'just are'.

Immutable types can have several performance advantages over reference types. Immutable types have no heap management overhead, they don't reserve space to store a type tag, and the absence of aliasing makes more compiler optimizations possible. For a small class like 'CPX' (complex number), all these factors combine to give a significant win over a reference class implementation. Balanced against these positive factors in using an immutable object is the overhead that some C compilers introduce in passing the entire object on the stack. This problem is worse in immutable classes with many attributes. Unfortunately the efficiency of an immutable class is directly tied to how smart the C compiler is; at this time 'gcc' is not very bright in this respect, although other compilers are.

Immutable classes aren't strictly necessary; reference classes with immutable semantics work too. For example, the reference class 'INTI' implements immutable infinite precision integers and can be used like the built-in immutable class 'INT'. The standard string class 'STR' is also a reference type but behaves with immutable semantics. Explicitly declaring immutable classes allows the compiler to enforce immutable semantics and provides a hint for good code generation. Common immutable classes are defined in the standard libraries; defining a new immutable class is unusual.

1.2.7 *pSather*

Parallel Sather (pSather) is a parallel extension of the language, developed and in use at ICSI. It extends serial Sather with threads, synchronization, and data distribution.

pSather differs from concurrent object-oriented languages that try to unify the notions of objects and processes by following the *actors* model [1]. There can be a grave performance impact for the implicit synchronization this model imposes on threads even when they do not conflict. While allowing for actors, pSather treats object-orientation and parallelism as orthogonal concepts, explicitly exposing the synchronization with new language constructs.

pSather follows the Sather philosophy of shielding programmers from common sources of bugs. One of the great difficulties of parallel programming is avoiding bugs introduced by incorrect synchronization. Such bugs cause completely erroneous values to be silently propagated, threads to be starved out of computational time, or programs to deadlock. They can be especially troublesome because they may only manifest themselves under timing conditions that rarely occur (*race conditions*) and may be sensitive enough that they don't appear when a program is instrumented for debugging (*heisenbugs*). pSather makes it easier to write deadlock and starvation free code by providing structured facilities for synchronization. A *lock statement* automatically performs unlocking when its body exits, even if this occurs under exceptional conditions. It automatically avoids deadlocks when multiple locks are used together. It also guarantees reasonable properties of fairness when several threads are contending for the same lock.

pSather allows the programmer to direct data placement. Machines do not need to have large latencies to make data placement important. Because processor speeds are outpacing memory speeds, attention to locality can have a profound effect on the performance of even ordinary serial programs. Some existing languages can make life difficult for the performance-minded programmer because they do not allow much leeway in expressing placement. For example, extensions allowing the programmer to describe array layout as block-cyclic is helpful for matrix-oriented code but of no use for general data structures.

Because high performance appears to require explicit human-directed placement, pSather implements a shared memory abstraction using the most efficient facilities of the target platform available, while allowing the programmer to provide placement directives for control and data (without requiring them). This decouples the performance-related placement from code correctness, making it easy to develop and maintain code enjoying the language benefits available to serial code. Parallel programs can be developed on simulators running on serial machines. A powerful object-oriented approach is to write both serial and parallel machine versions of the fundamental classes in such a way that a user's code remains unchanged when moving between them.

1.3 USING SATHER

At the time of this writing, the only compiler implementing the 1.1 language specification is available from ICSI. It is freely available, includes source for class libraries and the compiler, and compiles into ANSI C. This compiler has been ported to a wide range of UNIX and PC operating systems.

1.3.1 *Obtaining the compiler*

The ICSI Sather 1.1 compiler can be obtained by anonymous ftp at

ftp.icsi.berkeley.edu: /pub/sather

Other sites also mirror the Sather distribution. The distribution includes installation instructions, 'man' pages, the standard libraries and source for the compiler (in Sather). Documentation, tutorials and up-to-date information are also available at the Sather WWW page:

http://www.icsi.berkeley.edu/~sather

ICSI also maintains a library of contributed Sather code at this page.

There is a newsgroup devoted to Sather:

comp.lang.sather

There is also a Sather mailing list if you wish to be informed of Sather releases; to subscribe, send email to:

sather-request@icsi.berkeley.edu

It is not necessary to be on the mailing list if you read the Sather newsgroup.

1.3.2 *How do I ask questions?*

If it appears to be a problem that others would have encountered (on platform 'X', I tried to install it but the it failed to link with the error 'Y'), then the newsgroup is a good place to ask. If you have problems with the compiler or questions that are not of general interest, mail to one of

sather-bugs@icsi.berkeley.edu
psather-bugs@icsi.berkeley.edu

This is also where you want to send bug reports and suggestions for improvements.

1.4 HISTORY

Sather is still growing rapidly. The initial Sather compiler (for 'Version 0' of the language) was written in Sather (bootstrapped by hand-translating to C) over the summer of 1990. ICSI made the language publicly available (version 0.1) June of 1991 [4]. The project has been snowballing since then, with language updates to 0.2 and 0.5, each compiler bootstrapped from the previous. These versions of the language are most indebted to Stephen

Omohundro, Chu-Cheow Lim, and Heinz Schmidt. pSather co-evolved with primary early contributions by Jerome Feldman, Chu-Cheow Lim, and Franco Mazzanti. The first pSather compiler [3] was implemented by Chu-cheow Lim on the Sequent Symmetry, workstations and the CM-5.

Sather 1.0 was a major language change, introducing bound routines, iterators, proper separation of typing and code inclusion, contravariant typing, strongly typed parameterization, exceptions, stronger optional runtime checks and a new library design [6]. The 1.0 compiler was a completely fresh effort by Stephen Omohundro and David Stoutamire. It was written in 0.5 with the 1.0 features introduced as they became functional. The 1.0 compiler was first released in the summer of 1994, and Stephen left the project shortly afterwards. The pSather 1.0 design was largely due to Stephan Murer and David Stoutamire.

This document describes Sather 1.1, released the summer of 1996. That compiler is primarily the work of David Stoutamire, Michael Philippsen, Claudio Fleiner and Boris Vaysman. Unlike previous specifications, pSather is now an extension that is part of the 1.1 specification.

A group at the University of Karlsruhe under the direction of Gerhard Goos created a compiler for Sather 0.1. The language their compiler supports, Sather-K, diverged from the ICSI specification when Sather 1.0 was released. Karlsruhe has created a large class library called Karla using Sather-K. More information about Sather-K can be found at:

http://i44www.info.uni-karlsruhe.de/~frick/SatherK

1.4.1 *Acknowledgments*

Sather has adopted ideas from a number of other languages. Its primary debt is to Eiffel, designed by Bertrand Meyer, but it has also been influenced by C, C++, Cecil, CLOS, CLU, Common Lisp, Dylan, ML, Modula-3, Oberon, Objective C, Pascal, SAIL, School, Self, and Smalltalk.

Steve Omohundro was the original driving force behind Sather, keeping the language specification from being pillaged by the unwashed hordes and serving as point man for the Sather community until he left in 1994. Chu-Cheow Lim bootstrapped the original compiler and was largely responsible for the original 0.x compiler and the first implementation of pSather. David Stoutamire took over as language tsar and compiler writer after Stephen left.

Sather has been very much a group effort; many, many other people have been involved in the language design discussions including: Subutai Ahmad, Krste Asanovic, Jonathan Bachrach, David Bailey, Joachim Beer, Jeff Bilmes, Chris Bitmead, Peter Blicher, John Boyland, Matthew Brand, Henry Cejtin, Alex Cozzi, Richard Durbin, Jerry Feldman, Carl Feynman, Claudio Fleiner, Ben Gomes, Gerhard Goos, Robert Griesemer, Hermann Häertig, John Hauser, Ari Huttunen, Roberto Ierusalimschy, Arno Jacobsen, Matt Kennel, Holger

Klawitter, Phil Kohn, Franz Kurfess, Franco Mazzanti, Stephan Murer, Michael Philippsen, Thomas Rauber, Steve Renals, Noemi de La Rocque Rodriguez, Hans Rohnert, Heinz Schmidt, Carlo Sequin, Andreas Stolcke, Clemens Szyperski, Martin Trapp, Boris Vaysman, and Bob Weiner. Countless others have assisted with practical matters such as porting the compiler and libraries.

1.4.2 *References*

[1] G. Agha, ''Actors: A Model of Concurrent Computation in Distributed Systems'', The MIT Press, Cambridge, Massachusetts, 1986.

[2] S. Burson, ''The Nightmare of C++'', Advanced Systems November 1994, pp. 57-62. Excerpted from *The UNIX-Hater's Handbook*, IDG Books, San Mateo, CA, 1994.

[3] C. Lim. "A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions," PhD thesis, University of California at Berkeley, October 1993. Available at the Sather WWW page.

[4] C. Lim, A. Stolcke. ''Sather language design and performance evaluation.'' TR-91-034, International Computer Science Institute, May 1991. Also available at the Sather WWW page.

[5] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, ''Iteration abstraction in Sather'', *Transactions on Programming Languages and Systems*, Vol. 18, No. 1, Jan 1996 p. 1-15. Available at the Sather WWW page.

[6] S. Omohundro. ''The Sather programming language.'' *Dr. Dobb's Journal*, 18 (11) pp. 42-48, October 1993. Available at the Sather WWW page.

[7] C. Szyperski, S. Omohundro, S. Murer. "Engineering a programming language: The type and class system of Sather," In Jurg Gutknecht, ed., *Programming Languages and System Architectures*, p. 208-227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993. Available at the Sather WWW page.

The Sather 1.1 Specification

2.1 INTRODUCTION

2.1.1 *About This Document*

When important terms are first defined, they are formatted like *this*. Most sections begin with an example of a syntactic construct followed by corresponding grammar rules. The grammar rules are expressed in a variant of Backus-Naur form. Nonterminal symbols begin with a letter and are represented by strings of letters and underscores in an italic font. The nonterminal symbol on the lefthand side of a grammar rule is followed by a double arrow '⇒' and the right-hand side of the rule. The terminal symbols consist of Sather keywords and special symbols and are typeset in the Helvetica font. Vertical bars '...|...' separate alternatives, parentheses '(...)' are used for grouping, square brackets '[...]' enclose optional clauses and braces '{...}' enclose clauses which may be repeated zero or more times. Multi-line examples are indented after the first line, and an ellipsis '...' indicates code that has been left out for clarity. Semicolons are used to separate examples only if, when taken together, the examples could be a legitimate section of Sather code. Trailing semicolons, which are optional, are not shown.

2.1.2 *Basic Concepts*

Data structures in Sather are constructed from *objects*, each of which has a specific *concrete type* that determines the operations that may be performed on it. *Abstract types* specify a set of operations without providing an implementation and correspond to sets of concrete types. The implementation of concrete types is defined by textual units called *classes*; abstract types are specified by textual units called *abstract classes*. Sather programs consist of classes and abstract class specifications. Each Sather *variable* has a *declared type* which determines the types of objects it may hold.

Classes define the following *features*: *attributes* which make up the internal state of objects, *shareds* and *constants* which are shared by all objects of a type, and *methods* which may be either *routines* or *iterators*. Any features are by default *public*, but may be declared *private* to allow only the class in which it appears access to it. An attribute or shared may instead be declared *readonly* to allow only the class in which it appears to modify it. Accessor routines are automatically defined for reading or writing attributes, shareds, and constants. The set of non-private methods in a class defines the *interface* of the corresponding type. Method definitions consist of *statements*; for their construction *expressions* are used. There are special *literal expressions* for boolean, character, string, integer, and floating point objects.

Certain conditions are described as *fatal errors*. These conditions should never occur in correct programs and all implementations of Sather must be able to detect them. For efficiency reasons, however, implementations may provide the option of disabling checking for certain conditions.

2.2 Lexical Structure

The character set used in source files is defined by the Sather implementation, but it must include at least the characters which appear in the syntactic constructs in this specification. Sather implementations may be based on ASCII, but this is not required. The case of characters in source files is significant. All syntactic constructs except identifiers and certain literals may be separated by an arbitrary number of *whitespace* characters and *comments*. The seven whitespace characters are space, tab, newline, vertical tab, backspace, carriage return, and form feed. Sather comments consist of two dashes '--' outside of a string (page 41) or character literal (page 40) and all following text until a newline.

Sather *identifiers* are used to name class features, method arguments, and local variables. Most consist of letters, decimal digits, and the underscore character, and begin with a letter. Iterator names additionally end with the '!' character. Abstract type names and class names are similar, but the letters must be uppercase and abstract type names begin with '$'. There are no restrictions on the lengths of Sather identifiers or class names. Identifiers, class names, and keywords must be followed by a character other than a letter, decimal digit, or underscore. This may force the use of white-space after an identifier.

identifier ⇒ *letter {letter* | *decimal_digit* | _*}*

uppercase_identifier ⇒ *uppercase_letter {uppercase_letter* | *decimal_digit* | _*}*

abstract_class_name ⇒ $ *uppercase_identifier*

iter_name ⇒ *[identifier]*!

letter ⇒ *lowercase_letter* | *uppercase_letter*

lowercase_letter ⇒ a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

uppercase_letter ⇒ A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z *decimal_digit* ⇒ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Sather *keywords* are used to identify the fundamental syntactic constructs and may not be used as identifiers. Some keywords are reserved for language extensions (page 67). The keywords are:

keyword ⇒ abstract | and | any | assert | attr | bind | break! | builtin | case | class
| clusters | clusters! | cohort | const | else | elsif | end | exception | external
| false | far | fork | guard | if | immutable | inout | include | initial | is | ITER
| lock | loop | near | new | once | or | out | par | parloop | post | pre | private | protect
| quit | raise | readonly | result | return | ROUT | SAME | self | shared | sync
| then | true | typecase | unlock | until! | void | when | while! | with | yield

The syntax also makes use of the following *special symbols*:

special_symbol ⇒ `( | ) | [ | ] | { | } | , | . | ; | : | $ | _ | + | - | * | / | = | < | > | # | ^ | % | ~ | | | !`
`| /= | <= | >= | := | :: | -> | @ | :-`

2.3 Types and Classes

Sather programs are textually made up of *classes*. Classes are used to define the code and storage that make up *types*. Each *object* is an instance of a type. Types can be thought of as representing sets of objects at runtime. Objects never change their type.

There are four kinds of objects in Sather: *immutable* (*e.g.* integers), *reference* (*e.g.* strings), *closures*, and *external* (used to represent entities in other languages). There are four corresponding concrete types: *immutable*, *reference*, *closure*, and *external* types. There are also abstract types, which represent sets of concrete types. Immutable, reference, external, and abstract types are defined textually by *immutable*, *reference*, *external*, and *abstract* classes. *Partial* classes define code that does not have corresponding objects or types, and may be included by other classes to obtain implementation.

The *type graph* for a program is a directed acyclic graph that is constructed from the program's source text. Its nodes are types and its edges represent the *subtype* relationship. If there is a path in this graph from a type t_1 to a type t_2, we say that t_2 is a subtype of t_1 and that t_1 is a *supertype* of t_2. Subtyping is reflexive; any type is a subtype of itself. Only abstract types and method closures can be supertypes (see pages 24 and 54); closure types can only be supertypes of other closure types.

Every Sather variable has a declared type. The fundamental typing rule is: *An object can only be held by a variable if the object's type is a subtype of the variable's type.* It is not possible for a program which compiles to violate this rule (*i.e.* Sather is *statically type-safe*).

2.3.1 *Type specifiers*

Examples:

```
INT
A{B,C{$D}}
$IS_EQ{T}
ROUT{A,B,C}:D
ITER{INT}:INT
SAME
```

Syntax:

type_specifier ⇒ (*uppercase_identifier* | *abstract_class_name*) [{ *type_specifier_list* }]
| *method_closure_type_specifier*
| SAME

method_closure_type_specifier ⇒ *routine_closure_type_specifier*
| *iter_closure_type_specifier*

routine_closure_type_specifier ⇒ ROUT
[{ *routine_mode type_specifier* { , *routine_mode type_specifier* } }]
[: *type_specifier*]

iter_closure_type_specifier ⇒ ITER
[{ *iter_mode type_specifier* { , *iter_mode type_specifier* } }]
[: *type_specifier*]

In source text, Sather types are specified by one of the following forms of *type specifier*:

- The name of a class or abstract class (*e.g.* 'A' or '$A'). This may be followed by a list of parameter type specifiers in braces (*e.g.* 'A{B,C}'). The parameter values must not cause the generation of an infinite number of types (*e.g.* 'FOO{FOO{T}}' within the class 'FOO{T}').
- The name of a type parameter within the body of a parameterized class or abstract type definition (*e.g.* 'T' in the body of 'class B{T} is ... end').
- The keyword 'ROUT' or 'ITER' optionally followed by a list of argument types in braces, optionally followed by a colon and return type (*e.g.* 'ROUT{A,B}:C', 'ITER{A,B}:C'). This is used for closure types (page 54).
- The special type specifier 'SAME,' which denotes the type of the class in which it occurs.

2.3.2 *Signatures*

Syntax:

> *abstract_signature* ⇒ *abstract_routine_signature* | *abstract_iter_signature*
>
> *abstract_routine_signature* ⇒ *identifier*
> [(*routine_argument* { , *routine_argument* })] [: *type_specifier*]
>
> *routine_argument* ⇒ *routine_mode identifier_list* : *type_specifier*
>
> *routine_mode* ⇒ [(out | inout)]
>
> *abstract_iter_signature* ⇒ *iter_name*
> [(*iter_argument* { , *iter_argument* })] [: *type_specifier*]
>
> *iter_argument* ⇒ *iter_mode identifier_list* : *type_specifier*
>
> *iter_mode* ⇒ [(out | inout | once)]
>
> *identifier_list* ⇒ *identifier* { , *identifier* }

Operations are performed on objects by calling *methods* on them, which are either *routines* (page 32) or *iterators* (page 32). All method arguments have a *mode*, which is one of: *in*, *out*, *inout*, or *once*. The *signature* of a method consists of its name, the modes and types of its arguments, if any, and its return type, if any. Abstract classes specify a set of *abstract signatures*, an interface without an implementation. Concrete classes specify a set of *concrete signatures* which do define an implementation.

We say that the method signature *f* *conflicts* with *g* when

1. *f* and *g* have the same name and number of arguments,
2. *f* and *g* either both return a value or neither does,
3. each argument mode in *f* is the same as the corresponding mode in *g*, or the mode in one is 'in' while the other is 'once',
4. and each argument type in *f* is neither a subtype nor a supertype of the corresponding argument type in *g*, unless both are concrete.

This rule for signature conflict defines which methods may be overloaded (page 45). Sather permits overloading based on the number, type and mode of arguments, as well as whether or not a return value is present. However, overloading is not permitted between 'in' and 'once' modes.

We say that the method signature f conforms to g when

1. f and g have the same name and number of arguments,
2. f and g either both return a value or neither does,
3. the mode of each argument is the same (in, out, inout or once),
4. *contravariant conformance*:
   for any 'in' or 'once' arguments, the type in g is a subtype of the type in f;
   for any 'inout' arguments, the type in f is the same type as in g;
   for any 'out' arguments, the type in f is a subtype of the type in g; and
   if it has one, the return type of f is a subtype of the return type of g.

The set of methods that may be called on a type is called the interface of that type. A type interface may not contain conflicting signatures. An interface I_1 conforms to an interface I_2 if for every method f_2 in I_2 there is a unique conforming method f_1 in I_1. The basic subtyping rule is: '*The interface of each type must conform to the interfaces of each of its supertypes.*' This ensures that calls made on a type can be handled by any of its subtypes.

2.3.3 *Sather source files*

Example:

```
abstract class $PLANET is ... end;
class GAS_GIANT < $PLANET is ... end;
```

Syntax:

source_file ⇒ *[abstract_class_definition | class] { ; [abstract_class_definition | class] }*

Sather source files consist of semicolon separated lists of classes. Execution of a Sather program begins with a routine named 'main' in a specified class (page 60), usually 'MAIN'.

2.3.4 *Abstract classes*

Example:

```
abstract class $SHIPPING_CRATE{T} < $CONTAINER{T}
is
   destination:$LOCATION;
   weight:FLT;
end
```

Syntax:

abstract_class_definition ⇒ abstract class *abstract_class_name [parameterization] [subtyping_clause] [supertyping_clause]* is *[abstract_signature] {* ; *[abstract_signature] }* end

subtyping_clause ⇒ < *type_specifier_list*

supertyping_clause ⇒ > *type_specifier_list*

type_specifier_list ⇒ *type_specifier {* , *type_specifier}*

Abstract class definitions specify interfaces without implementations. Abstract class names must be entirely uppercase and must begin with a dollar sign '$' (page 19); this makes it easy to distinguish abstract type specifications from other types, and may be thought of as a reminder that operations on objects of these types might be more expensive since they may involve dynamic dispatch. The scope of abstract type names is the entire program. Two abstract class definitions may be parameterized (see page 27) and may have the same name only if they specify a different number of type parameters.

A subtyping clause ('<' followed by type specifiers) adds to the type graph an edge from each type in the *type_specifier_list* to the type being defined. In the example, the subtyping clause is '< $CONTAINER{T}'. Each listed type must be abstract. Every type is automatically a subtype of $OB (page 61). There must be no cycle of abstract types such that each appears in the subtype list of the next, ignoring the values of any type parameters but not their number. A subtyping clause may not refer to 'SAME'.

A supertyping clause ('>' followed by type specifiers) adds to the type graph an edge from the type being defined to each type in the *type_specifier_list*. These type specifiers may not be type parameters (though they may include type parameters as components) or external types. There must be no cycle of abstract classes such that each class appears in the supertype list of the next, ignoring the values of any type parameters but not their number. A supertyping clause may not refer to 'SAME'.

If both subtyping and supertyping clauses are present, then each type in the supertyping list must be a subtype of each type in the subtyping list using only edges introduced by subtyping clauses. This ensures that the subtype relationship can be tested by examining only definitions reachable from the two types in question, and that errors of supertyping are localized.

The body of an abstract class definition consists of a semicolon separated list of abstract signatures. Each specifies the signature of a method without providing an implementation at that point. The argument names are required for documentation purposes only and are ignored. The abstract signatures of all types listed in the subtyping clause are included in the interface of the type being defined. Explicitly specified signatures override any conflicting signatures from the subtyping clause. If two types in the subtyping clause have conflicting signatures that are not equal, then the type definition must explicitly specify a signature that overrides them. The interface of an abstract type consists of any explicitly specified signatures along with those introduced by the subtyping clause.

2.3.4a **Abstract class examples**

Here's an example from the standard library. The abstract class '$STR' represents the set of types that have a way to construct a string suitable for output. All of the standard types such as 'INT', 'FLT', 'BOOL' and 'CPX' know how to do this, so they are subtypes of '$STR'. Attempting to subtype from '$STR' a concrete class that didn't provide a 'str' method would cause an error at compile time.

```
abstract class $STR is
   -- Subtypes of this define "str:STR".
   -- This should be a reasonable
   -- string representation of an object.

   str:STR;-- String form of object.
end
```

Here's another abstract class that subtypes from '$STR'. In addition to requiring the 'str' method, it adds a 'create' method for creating from the string representation.

```
abstract class $FROM_STR < $STR is
   -- Subtypes of this must define
   -- methods for going to and from
   -- the STR representation.

   create(s:STR):$FROM_STR;
end
```

In this illegal abstract class, A and B do not conflict because their arguments are concrete and are not the same type. However, because the argument of C is abstract and unrelated it conflicts with both A and B. D does not conflict with A, B or C because it has a different number of parameters.

```
abstract class $FOO is
   foo(arg:INT);        -- method A
   foo(arg:BOOL);       -- method B
   foo(arg:$FOO);       -- method C
   foo(a, b:INT)        -- method D
end
```

2.3.5 *Concrete classes*

Examples:

```
class VIEWER{DATA < $VIEWER_DATA} is ... end;
immutable class QUATERNION is ... end;
external FORTRAN class BLAS is ... end;
partial class MIXIN is ... end
```

Syntax:

class ⇒ *[* immutable | partial | external *identifier]*
class *uppercase_identifier [parameterization] [subtyping_clause]*
is *[class_element] {* ; *[class_element] }* end

There are three types that have implementations: reference, immutable, and external types. They are defined by classes beginning with 'class', 'immutable class', and 'external *language* class', respectively. Reference types may be aliased and usually are allocated on a dynamic heap. Immutable types (such as complex numbers) are immune to aliasing and usually do not require heap allocation (see page 13). External types are used to allow Sather variables to refer to entities of other languages, and are discussed further on page 67. Partial classes have no associated type and contain code that may only be included by other classes. Partial classes may not be instantiated: no routine calls from another class into a partial class are allowed, and no variables may be declared in another class of such a type.

Class names must be entirely uppercase (page 19). The scope of class names is the entire program and two classes may have the same name only if they specify a different number of parameters (page 27).

Subtyping clauses introduce edges into the type graph. Each type listed in the subtyping clause must be abstract. There is an edge in the type graph from each type in the list to the type being defined. Every type is automatically a subtype of $OB (page 61). A subtyping clause may not refer to 'SAME'. When a subtyping clause is used with a partial class, it does not introduce an edge into the type graph, but does enforce the basic subtyping rule (page 23) between the interface(s) of the abstract class(es) and the partial class. Only partial classes may have stubs (page 34).

2.3.5a Concrete class example

The complex number class from the standard library is a good example of a immutable class. 'CPX' is immutable (see page 13) because it is small and behaves with a mathematical semantics. Here we also see that complex numbers can be tested for equality with other objects ('$IS_EQ') and has a routine (str: STR) for conversion to a string ('$STR').

```
immutable class CPX < $IS_EQ, $STR
is
    is_eq(s: $OB): BOOL is ... end;
    str: STR is ..... end;
    ...
end
```

2.3.6 *Parameterization*

Syntax:

parameterization $\Rightarrow$ { *parameter_declaration* { , *parameter_declaration* } }

parameter_declaration $\Rightarrow$ *uppercase_identifier* [< *type_specifier*]

Class definitions may optionally have one or more *parameters* within enclosing braces. Parameters are placeholders for actual types that are filled in at a point of use. Whenever a parameterized class is referred to, its formal parameters are instantiated with type specifiers. Parameter names are local to the class definition in which they appear and they shadow non-parameterized types with the same name. Parameter names must be all uppercase, and they may be used within the class definition as type specifiers. Partial classes may not be used as parameters. There is no implicit type relationship between different parameterizations of a class.

If a parameter declaration is followed by a *type constraint* clause ('<' followed by a type specifier), then the parameter may only be replaced by subtypes of the constraining type. If a type constraint is not explicitly specified, then '< $OB' is taken as the constraint. A type constraint clause may not refer to 'SAME'. A class definition must satisfy all typing rules when its parameters are replaced by any potential subtype of their constraining type. This allows type-safe independent compilation: classes may be checked once for all parameterizations by type checking using prototypical unique subtypes of the type constraints.

An instantiated parameterized class definition is very similar to a non-parameterized copy of the original definition in which each formal parameter occurrence is replaced by the specified actual type. Parameterization may be thought of as a structured macro facility; however, it is not the same as simple textual replacement, because the resolution of overloading uses type constraints instead of actual parameter types (page 45). *Note: at this time the ICSI compiler efficiently but incorrectly resolves overloading based on fully instantiated types rather than type constraints.*

2.3.6a Parameterization examples

Most Sather code resides in ordinary reference classes. A frequently used class from the standard library is 'ARRAY{T}', which is a conventional array of the parameterized type 'T'. Here we show the method 'contains', which uses 'T'. For example, an 'ARRAY{INT}' would support the call 'contains(5)'.

```
class ARRAY{T} is
    ...
    contains(e:T):BOOL is
        ...
    end;
    ...
end
```

The priority queue abstraction requires parameter type constraints. Elements need to be comparable to each other, no matter what the parameter instantiation is. This is specified by the parameter type constraint '< $IS_LT{T}', which itself uses the parameter 'T'.

```
abstract class $PQ{T < $IS_LT{T}} < ... is
    top: T;
    pop: T;
    insert(e: T);
    clear;
    is_empty: BOOL;
end;
```

2.4 CLASS ELEMENTS

Syntax:

class_element ⇒ *const_definition* | *shared_definition* | *attr_definition*
| *routine_definition* | *iter_definition* | *include_clause* | *stub*

The main body of each class is a semicolon separated list of elements which define the features of the class. The semantics of a class is independent of the textual order of its class elements. All features are named. Some features may contribute a reader and a writer routine of the same name to the class interface. The scope of feature names is the class body and is separate from the class namespace. If a feature is private, then it may only be referred to from within the class and is not part of the class interface.

There are language-specific restrictions on the elements that may appear in external classes (page 67). Some names ('main' and 'invariant') are reserved for special purposes (page 60).

2.4.1 *Constant definitions*

Examples:

```
const r:FLT:=45.6;
private const a,b,c;
private const d:=4,e,f
```

Syntax:

const_definition ⇒ *[* private *]* const *identifier*
(: *type_specifier* := *expression* | *[* := *expression* *] [* , *identifier_list* *])*

Constants are accessible by all objects in a class and may not be assigned to. If a type is specified, then the construct defines a single constant attribute named *identifier* and it must be initialized by the expression *expression*. This must be a constant expression which means that it is:

1. a character, boolean, string, integer or floating point literal expression (page 40),
2. a void or void test expression (page 46),
3. an and or or expression (page 48), each of whose components is a constant expression,
4. an array creation expression (page 48), each of whose components is a constant expression,
5. a routine call applied to a constant expression, each of whose arguments is a constant expression other than void, or
6. a reference to another constant in the same class or in another class using the '::' notation.

There must not be cyclic dependencies among constant initializers. The libraries are designed so that no observable side-effects can occur during constant initialization.

If a type specifier is not provided, then the construct defines one or more successive integer constants. The first identifier is assigned the value zero by default; its value may also be specified by a constant expression of type 'INT'. The remaining identifiers are assigned successive integer values. This is the way to do enumeration types in Sather. It is an error if no type specifier is provided and there is an assignment that is not of type 'INT'.

Each constant definition causes the implicit definition of a reader routine with the same name. It takes no arguments and returns the value of the constant. Its return type is the constant's type. The routine is private if and only if the constant is declared 'private'.

2.4.2 *Shared attribute definitions*

Examples:

```
private shared i,j:INT;
shared s:STR:="name";
readonly shared c:CHAR:='x'
```

Syntax:

shared_definition ⇒ *[* private | readonly *]* shared
(*identifier* : *type_specifier* := *expression* | *identifier_list* : *type_specifier* *)*

Shared attributes are global variables that reside in a class namespace. When only a single shared attribute is defined, a constant initializing expression may be provided (page 29). If no initializing expression is provided, the shared is initialized to the value 'void' (page 46).

Each shared definition causes the definition of a reader routine and a writer routine, both with the same name. The reader routine takes no arguments and returns the value of the shared. Its return type is the shared's type. The reader routine is private if the shared is declared 'private'. The writer routine sets the value of the shared, taking a single argument whose type is the shared's type, and has no return value. The writer routine is private if the shared is declared either 'private' or 'readonly'.

2.4.3 *Attribute definitions*

Examples:

```
attr a,b,c:INT;
private attr c:CHAR;
readonly attr s1,s2:STR
```

Syntax:

attr_definition ⇒ *[* private | readonly *]* attr *identifier_list* : *type_specifier*

An object's state consists of the attributes defined in its class together with an optional array portion. The array portion appears if there is an include path (page 33) from the type to AREF for reference types or to AVAL for immutable types (page 61). Closure and reference objects must be explicitly allocated as described on pages 47 and 54. Variables have the value 'void' until an object is assigned to them (page 46). There must be no cycle of immutable types such that each type has an attribute whose type is in the cycle.

Each attribute definition causes the definition of a reader and a writer routine with the same name. The reader routine takes no arguments and returns the value of the attribute. Its declared return type is the attribute's type. It is private if the attribute is declared 'private'.

The writer routine takes different forms for reference and immutable types. For reference types, the writer routine takes a single argument whose type is the attribute's type and has no return value. Its effect is to modify the object by setting the value of the attribute. For immutable types, it takes a single argument whose type is the attribute's type, and returns a copy of the object with the attribute set to the specified new value, and whose type is the type of the object. Object attribute writer routines are private if the corresponding attribute is declared either 'private' or 'readonly'.

2.4.3a **Attribute, shared and constant examples**

Here's an example of a tree node class. Each node has attributes for storing child nodes as well as the data at that node. 'datum' and 'total_nodes_created' are marked readonly, so they may not be written by code in other classes. The 'total_nodes_created' field is presumably incremented in the create routine, and all nodes will see the same value.

The 'lchild' attribute implicitly defines two signatures: 'lchild:NODE{T}' for reading and 'lchild(NODE{T})' for writing.

```
class NODE{T} is

   attr lchild, rchild:SAME;
   readonly attr datum:T;

   readonly shared
      total_nodes_created:INT;

   const min_balanced_depth:INT:=5;

   ...
end
```

This example shows three different ways to modify an attribute. 'n' is a reference type, so the 'lchild' field can be modified by assignment, or by calling the implicit writer routine for the attribute. 'c' is an immutable type, so its implicit writer routine returns a new object instead of modifying the object in place.

```
n:NODE{T};  c:CPX;
...
n.lchild := x;      -- These two lines
n.lchild(x);        --    are equivalent.
...
c := c.re(1.0);     -- attr of immutable type
```

2.4.4 *Routine definitions*

Examples:

```
a(f:FLT):FLT
    pre f>1.2   post result<4.3
is ... end;
b is ... end;
private d:INT is ... end;
c(s1,s2,s3:STR) is ... end
```

Syntax:

> *routine_definition* ⇒ *[* private *]* *identifier*
> *[* (*routine_argument* *{* , *routine_argument* *}*) *]* *[* : *type_specifier* *]*
> *[* pre *expression* *]* *[* post *expression* *]* *[* is *statement_list* end *]*

A routine definition may begin with the keyword 'private' to indicate that the routine may be called from within the class but is not part of the class interface. The *identifier* specifies the name of the routine.

If a routine has arguments, the declaration list is enclosed in parentheses. The mode, name and type of each argument is specified in this list. The types of consecutive arguments may be declared with a single type specifier. Each argument's mode defaults to 'in' if neither 'out' nor 'inout' is specified (page 45). If a routine has a return value, it is declared by a colon and a specifier for the return type. SAME is permitted only for a return type or out arguments.

The 'pre' and 'post' clauses specify optional pre- and post-conditions, and are discussed further on page 58. The body of a routine definition is a list of statements (page 35).

2.4.5 *Iterator definitions*

Example:

```
elts!(once i:INT, x:FLT):T is ... end
```

Syntax:

> *iter_definition* ⇒ *[* private *]* *iter_name*
> *[* (*iter_argument* *{* , *iter_argument* *}*) *]* *[* : *type_specifier* *]*
> *[* pre *expression* *]* *[* post *expression* *]* is *statement_list* end

Iterators are similar to routines but encapsulate iteration abstractions. Their names end with an exclamation point '!' and they may only be called within loop statements (page 51). Iterator arguments that are not marked 'once' are called *hot* and cause re-evaluation of that argument at each iteration (see also page 45). As with routines, SAME is permitted only for a return type or out arguments.

The description of routine arguments and pre and post constructs also applies to iterator definitions. Iters may contain yield (page 38) and quit (page 52) statements but may not contain return statements (page 38). The semantics of iterator calls is described in the section on loop statements (page 51). The pre clause must be true each time the iterator is called and the post clause must be true each time it yields. The post clause is not evaluated when an iterator quits.

The semantics of iterators and loops are discussed in more detail on page 51.

2.4.6 *Code inclusion and* include *clauses*

Examples:

```
include A a->b, c->, d->private d;
private include D e->readonly f;
```

Syntax:

include_clause ⇒ [private] include *type_specifier*
 [*feature_modifier* { , *feature_modifier* }]

feature_modifier ⇒ (*identifier* | *iter_name*) ->
 [[private | readonly] (*identifier* | *iter_name*)]

Implementation inheritance is defined by *include clauses*. These cause the incorporation of the implementation of the specified type, possibly undefining or renaming features with *feature_modifier* clauses. The include clause may begin with the keyword 'private', in which case any unmodified included feature is made private. We say that there is an *include path* from one type to another if there is a sequence of types between them such that each includes the next in the sequence.

The included type specified by the *type_specifier* may not be a closure type or a type parameter (though type parameters may appear as components of the type specifier). Partial classes may be included. External classes may be included if the interface to the language permits this; external Fortran (page 68) and C (page 69) classes may not be included. There mustn't be include paths from reference types to AVAL or from immutable types to AREF (page 61). There must be no cycle of classes such that each class includes the next, ignoring the values of any type parameters but not their number. If SAME occurs in an include clause, it is interpreted as the eventual type of the class (as late as possible).

Each *feature_modifier* clause specifies an identifier which must be the name of at least one feature in the included class. If no clause follows the '->' symbol, then the named features are not included in the class. If an identifier follows the '->' symbol, then it becomes the new name for the features. In this case, the listed features are included as part of the public interface unless they are specified as 'private' or 'readonly'. Identifiers may only be renamed as identifiers and iterator names may only be renamed by iterator names. It is an error if there are no appropriate methods to rename in the included class, and both a reader and a writer method (page 30) must exist if 'readonly' is used.

A class may not explicitly define two methods whose signatures conflict (page 22). A class may not define a routine whose signature conflicts with either the reader or the writer routine of any of its attributes (whether explicitly defined or included from other classes). If a method is explicitly defined in a class, it overrides all conflicting methods from included classes. The implicit reader and writer routines of a class's attributes, shareds, and constants also override any included routines and must not conflict with each other. If an included method is not overridden, then it must not conflict with another included method; feature modification clauses can be used to resolve any conflicts.

2.4.7 *Stubs*

Example:

```
stub register_object(ob:FOO);
```

Syntax:

stub *abstract_signature*

A stub feature may only be present in a partial class. They have no body and are used to reserve a signature for redefinition by an including class. If code in a partial class contains calls to an unimplemented method, that method must be explicitly provided as a stub

2.4.7a **Code inclusion examples.**

The class 'ARRAY{T}' in the standard library is not a primitive data type. It is based on a built-in class 'AREF{T}' which provides objects with an array portion. 'ARRAY' obtains this functionality using an 'include', but chooses to modify the visibility of some of the methods. It also defines additional methods such a 'contains', 'sort', etc. The methods 'aget', 'aset' and 'asize' are defined as 'private' in 'AREF', but 'ARRAY' redefines them to be public.

```
class ARRAY{T} is
   private include AREF{T}
      -- Make these public.
      aget->aget,
      aset->aset,
      asize->asize;
   ...
   contains(e:T):BOOL is ... end
   ...
end
```

This code demonstrates the use of partial classes. Each MIXIN class provides a different way of prompting the user; each can be combined with COMPUTE to make a complete program. The stub in COMPUTE allows that class to be type checked without needing either mix-in class.

Only COMPUTE_A and COMPUTE_B may actually be instantiated.

This style of code reuse is very flexible because the stub routines can access private data in COMPUTE. Such flexibility requires extra care, because it bypasses ordinary class encapsulation.

```
partial class MIXIN_A is
    prompt_user is ... end;
end;

partial class MIXIN_B is
    prompt_user is ... end;
end;

partial class COMPUTE is
    main is ...prompt_user; ... end;
    stub prompt_user;
end;

class COMPUTE_A is
    include COMPUTE;
    include MIXIN_A;
end;

class COMPUTE_B is
    include COMPUTE;
    include MIXIN_B;
end;
```

2.5 Basic Statements

Syntax:

statement_list ⇒ [*statement*] { ; [*statement*] }

statement ⇒ *declaration_statement* | *assign_statement* | *if_statement*
| *return_statement* | *case_statement* | *typecase_statement*
| *expression_statement* | *loop_statement* | *yield_statement*
| *quit_statement* | *protect_statement* | *raise_statement*
| *assert_statement*

The body of a method is a semicolon separated list of statements. The statements in a statement list are executed sequentially unless a return, quit, yield, or raise statement is executed. In a routine with a return value, the final statement along each execution path must be either a return statement or a raise statement.

2.5.1 *Declaration statements*

Example:

```
i,j,k:INT
```

Syntax:

declaration_statement ⇒ *identifier_list* : *type_specifier*

<u>*Declaration statements*</u> are used to declare the type of one or more local variables. Local variables may also be declared in assignment statements (page 36). The scope of a local variable declaration begins at the declaration and continues to the end of the statement list in which the declaration occurs. The scope of method arguments is the entire body of the method. Local variables shadow routines in the class which have the same name and no arguments. Within the scope of a local variable it is illegal to declare another local variable with the same name. Local variables are initialized to void (page 46) when the containing method is called; they are not re-initialized when the declaration is encountered in the flow of control.

2.5.2 *Assignment statements*

Examples:

```
a:=5
b(7).c:=5
A::d:=5
[3]:=5
e[7,8]:=5
g:INT:=5
h::=5
```

Syntax:

assign_statement ⇒ (*expression* | *identifier* : [*type_specifier*]) := *expression*

<u>*Assignment statements*</u> are used to assign objects to locations and can also declare new local variables. The expression on the right hand side must have a return type which is a subtype of the declared type of the destination specified by the left hand side. When a reference object is assigned to a location, only a *reference* to the object is assigned. This means that later changes to the state of the object will be observable from the assigned location. Since immutable and closure objects cannot be modified once constructed, this issue is not relevant to them. We consider each of the allowed forms for the lefthand side of an assignment in turn:

1. *'identifier'*

    If the left hand side is a local variable or an argument of a method, then the assignment is directly performed (*e.g.* 'a:=5'). Otherwise the statement is syntactic sugar for a call of the routine named *identifier* with the right hand side of the assignment as the only argument (*e.g.* 'a(5)').

2. *'(expression* . | *type_specifier* :: *) identifier'*

These forms are syntactic sugar for calls of a routine named *identifier* with the right hand side as an argument: *(expression* . | *type_specifier* :: *) identifier* (*rhs*). For example, 'b(7).c:=5' is sugar for 'b(7).c(5)' and 'A::d:=5' is sugar for 'A::d(5)'.

3. *'[expression]* [*expression_list*] *'*

This form is syntactic sugar for a call of a routine named 'aset' with the array index expressions and the right hand side of the assignment as arguments: *[expression* . | *type_specifier* :: *]* aset(*expression_list* , *rhs*). For example, '[3]:=5' is sugar for 'aset(3,5)' and 'e[7,8]:=5' is sugar for 'e.aset(7,8,5)'.

4. *'identifier* : *[type_specifier]'*

This form both declares a new local variable and assigns to it (*e.g.* 'g:INT:=5'). If a type specifier is not provided, then the declared type of the variable is the return type of the expression on the righthand side (*e.g.* 'h::=5'). The scoping rules given on page 36 apply here as well. If a type is explicitly specified, the construct is syntactic sugar for a declaration statement followed by an assignment statement.

2.5.3 if *statements*

Example:

```
if a>5 then foo
elsif a>2 then bar
else error
end
```

Syntax:

if_statement ⇒ if *expression* then *statement_list*
{ elsif *expression* then *statement_list }*
[else *statement_list]* end

if statements are used to conditionally execute statement lists according to the value of a boolean expression. Each *expression* in the form must return a boolean value. The first expression is evaluated and if it is true, the following statement list is executed. If it is false, then the expressions of successive elsif clauses are evaluated in order. The statement list following the first of these to return true is executed. If none of the expressions return true and there is an else clause, then its statement list is executed.

2.5.4 return *statements*

Examples:

```
return
return x
```

Syntax:

return_statement ⇒ return *[expression]*

return statements are used to return from routine calls. No other statements may follow a return statement in a statement list because they could never be executed. If a routine doesn't have a return value then it may return either by executing a return statement without an *expression* portion or by executing the last statement in the routine body.

If a routine has a return value, then its return statements must specify expressions whose types are subtypes of the routine's declared return type. Execution of the return statement causes the expression to be evaluated and its value to be returned. It is a fatal error if the final statement executed in such a routine is not a return or raise (page 58) statement.

2.5.5 case *statements*

Example:

```
case i
when 5, 6 then ...
when j then ...
else ...
end
```

Syntax:

case_statement ⇒ case *expression*
when *expression { , expression }* then *statement_list*
{ when *expression { , expression }* then *statement_list }*
[else *statement_list]* end

Multi-way branches are implemented by *case statements*. There may be an arbitrary number of *when clauses* and an optional *else clause*. The initial *expression* construct is evaluated first and may have a return value of any type. This type must define one or more routines named 'is_eq' with a single argument and a boolean return value. The case statement is semantically syntactic sugar for (equivalent to) an if statement, each of whose branches tests a call of is_eq. The arguments to these calls are the expressions of successive when lists. The first one of these calls to returns true causes the corresponding statement list to be executed and control passed to the statement following the case statement. If none of the when expressions matches and an else clause is present, then the statement list following it is executed. It is a fatal error if no branch matches in the absence of an else clause.

2.5.6 typecase *statements*

Example:

```
typecase a
when INT then ...
when FLT then ...
when $A then ...
else ...
end
```

Syntax:

> *typecase_statement* ⇒ typecase *identifier*
> when *type_specifier* then *statement_list*
> *{* when *type_specifier* then *statement_list }*
> *[* else *statement_list]* end

An operation that depends on the runtime type of an object held by a variable of abstract type may be performed inside a *typecase statement*. The *identifier* must name a local variable or an argument of a method. If the typecase appears in an iterator, then the mode of the argument must be once; otherwise, the type of object that such an argument holds could change.

On execution, each successive *type_specifier* is tested for being a supertype of the type of the object held by the variable. The statement list following the first matching type specifier is executed and control passes to the statement following the typecase. Within each statement list, the type of the typecase variable is taken to be the type specified by the matching type specifier unless the variable's declared type is a subtype of it, in which case it retains its declared type. It is not legal to assign to the typecase variable within the statement lists. If the object's type is not a subtype of any of the type specifiers and an else clause is present, then the statement list following it is executed. It is a fatal error for no branch to match in the absence of an else clause. The declared type of the variable is not changed within the else statement list. If the value of the variable is void when the typecase is executed, then its type is taken to be the declared type of the variable.

2.5.7 *Expression statements*

Examples:

```
foo(1, x);
bar
```

Syntax:

> *expression_statement* ⇒ *expression*

A statement may consist of an expression that does not return a value and is executed solely for its side-effects.

2.6 Literal Expressions

Syntax:

expression ⇒ *bool_literal_expression* | *char_literal_expression* | *str_literal_expression* | *int_literal_expression* | *flt_literal_expression*

There are special lexical forms for literal expressions which define boolean, character, string, integer, and floating point values. These literal forms all have a concrete type derived from the syntax; typing of literals is not dependent on context. Sather does not do implicit type coercions (such as promoting an integer to floating point when used in a floating point context.) Types must instead be promoted explicitly by the programmer. This avoids a number of portability and precision issues (for example, when an integer can't be represented by the floating point representation.)

These two expressions are equivalent. In the first, the 'd' is a literal suffix denoting the type. In the second, '3.14' is the literal and '.fltd' is an explicit conversion.

```
3.14d      -- A double precision literal
3.14.fltd  -- Single, but converted
```

2.6.1 *Boolean literal expressions*

Examples:

```
true
false
```

Syntax:

bool_literal_expression ⇒ true | false

BOOL objects represent boolean values (page 61). The two possible values are represented by the *boolean literal expressions*: 'true' and 'false'.

2.6.2 *Character literal expressions*

Examples:

```
'a'
'\n'
'\016'
```

Syntax:

char_literal_expression ⇒ ' (*ISO_character* | \ *escape_seq*) '

escape_seq ⇒ a | b | f | n | r | t | v | \ | ' | " | *octal_digit* {*octal_digit*}

CHAR objects represent characters (page 61). <u>*Character literal expressions*</u> begin and end with single quote marks. These may enclose either any single ISO-Latin-1 printing character except single quote or backslash or an escape code starting with a backslash.

The escape codes are interpreted as follows: '\a' is an *alert* such as a bell, '\b' is the *backspace* character, '\f' is the *form feed* character, '\n' is the *newline* character, '\r' is the *carriage return* character, '\t' is the *horizontal tab* character, '\v' is the *vertical tab* character, '\\' is the *backslash* character, '\'' is the *single quote* character, and '\"' is the *double quote* character. A backslash followed by one or more octal digits represents the character whose octal representation is given. A backslash followed by any other character is that character. The mapping of escape codes to other characters is defined by the Sather implementation.

2.6.3 *String literal expressions*

Examples:

```
"a string literal"
"concat"  "enation"
```

Syntax:

str_literal_expression ⇒ "*{ISO_character}*" *{*"*{ISO_character}*"*}*

STR objects represent strings (page 61). <u>*String literal expressions*</u> begin and end with double quote marks. The characters making up the string are specified in this construct from left to right. A backslash starts an escape sequence as with character literals. All successive octal digits following a backslash are taken to define a single character. Individual double-quote-bounded segments of string literals may not extend beyond a single line in the source text. However, successive quote bounded segments are concatenated together to form a single string and can be used to allow string literals to span more than one line of source code. They may also be used to force the end of an octal encoded character. For example: "\0367" is a one character string, while "\03""67" is a three character string. Such segments may be separated by comments and whitespace.

2.6.4 *Integer literal expressions*

Examples:

```
14
14i
-4532
39_832_983_298
0b101011
-0b_10111010_00101100_01010101
0o372363i
0x_e98a_7c4d_65d7_6aa6_932d
```

Syntax:

int_literal_expression ⇒ *[-] (binary_int* | *octal_int* | *decimal_int* | *hex_int) [*i*]*

binary_int ⇒ 0b *{binary_digit* | *_}*

binary_digit ⇒ 0 | 1

octal_int ⇒ 0o *{octal_digit* | *_}*

octal_digit ⇒ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

decimal_int ⇒ *decimal_digit {decimal_digit* | *_}*

hex_int ⇒ 0x *{hex_digit* | *_}*

hex_digit ⇒ *decimal_digit* | a | b | c | d | e | f

INT objects represent machine integers and INTI objects represent infinite precision integers (page 61). The literal form for INTI objects ends with a trailing 'i'. A leading '-' sign is used to denote a negative integer. Integer literals can be represented in four bases: binary is base 2, octal is base 8, decimal is base 10 and hexadecimal is base 16. These are indicated by the prefixes: '0b', '0o', nothing, and '0x' respectively. Underscores may be used within integer literals to improve readability and are ignored. INT literals are only legal if they are in the representable range of the Sather implementation, which is at least 32 bits (page 61).

2.6.5 *Floating point literal expressions*

Examples:

```
12.34
3.498_239e-8d
```

Syntax:

flt_literal_expression ⇒ *[-] decimal_int* . *decimal_int [*e *[-] decimal_int] [* d *]*

FLT and FLTD objects represent floating point numbers according to the single and double representations defined by the IEEE-754-1985 standard (see also page 61). A floating point literal is of type FLT unless suffixed by 'd' designating a FLTD literal. The optional 'e' portion is used to specify a power of 10 by which to multiply the decimal value. Underscores may be used within floating point literals to improve readability and are ignored. Literal values are only legal if they are within the range specified by the IEEE standard.

2.7 BASIC EXPRESSIONS

Syntax:

expression ⇒ *self_expression* | *local_expression* | *call_expression* | *void_expression*
| *void_test_expression* | *new_expression* | *create_expression* | *array_expression*
| *and_expression* | *or_expression* | *sugar_expression*
| *while!_expression* | *until!_expression* | *break!_expression*
| *except_expression* | *initial_expression* | *result_expression*
| *closure_create_expression*

Sather *expressions* are used to compute values or to cause side-effects. If they return a value, then they have a *return type* that is either explicitly declared or inferred from context.

2.7.1 self *expressions*

Example:

```
self
```

Syntax:

self_expression ⇒ self

self expressions may appear in the bodies and in the pre and post clauses of routines and iters. They return the object on which the method was called. The return type is the type in which the method appears.

2.7.2 *Local variable access expressions*

Example:

```
a
```

Syntax:

local_expression ⇒ *identifier*

The name of an argument or local variable in a method is an expression which returns the value of that variable. The return type of such an expression is the declared type of the variable. Local variables may be accessed only within the body of a method. Arguments may additionally be accessed in method pre and post clauses.

All other expressions consisting of a single identifier are method calls on self as described in the next section.

2.7.3 *Method call expressions*

Examples:

```
a(5,7)
b.a(5,7)
A::a(5,7)
```

Syntax:

call_expression ⇒ [*expression* . | *type_specifier* ::]
(*identifier* | *iter_name*) [(*modal_list*)]

modal_list ⇒ *routine_mode expression* { , *routine_mode expression* }

The most common expressions in Sather programs are *method calls*. The *identifier* names the method being called. The object to which the method is applied is determined by what precedes the *identifier*. If nothing precedes it, then the form is syntactic sugar for a call on self (*e.g.* 'a(5,7)' is short for 'self.a(5,7)'). If the *identifier* is preceded by an expression and a dot '.', then the method is called on the object returned by the expression. If *identifier* is preceded by a type specifier and a double colon '::', then the method is taken from the interface of the specified type with self initialized to void as described on page 46.

When a method call occurs, the following takes place in strict order:

1. If it is an iterator call, and this call has not yet been evaluated since entering the enclosing loop, any 'once' arguments are evaluated, left to right.
2. 'in' and 'inout' arguments are evaluated, left to right. The object to which the method is applied is considered an 'in' argument.
3. The method call occurs. 'out' arguments are unassigned in the called method. It is a fatal error to use the value of an 'out' argument in the called method before it has been assigned. If the method terminates due to an uncaught exception, the following steps do not take place.
4. An assignment to each 'out' and 'inout' argument occurs in the caller, left to right. 'out' and 'inout' arguments behave according to the syntactic sugar rules that also apply to the left side of ':=' assignments.
5. The return value, if any, becomes available to the surrounding context.

If the method defines a return value, it must be used (*i.e.* the call may not be an *expression_statement*). Only non-private routines and iters may be called from outside a class, but all routines and iters may be called from inside a class.

Direct calls of a type's routines or iters may be made using the double colon '::' syntax. The *type_specifier* must specify a reference, immutable, or external class. In such calls self has the void default value described on page 46.

2.7.3a Modes

Method arguments each have a mode. Modes are specified by a keyword preceding argument names; if no keyword is given, the argument mode defaults to 'in'.

Mode	Description
in	All arguments are 'in' by default; there is no 'in' keyword. 'In' arguments pass a copy of the argument from the caller to the called method. With reference types, this is a copy of the reference to an object; the called method sees the same object as the caller.
out	An 'out' argument is passed from the called method to the caller when the called method returns. It is a fatal error for the called method to examine the value of the 'out' argument before assigning to it. The value of an 'out' argument may only be used after it has appeared on the left side of an assignment.
inout	An 'inout' argument is passed to the called method and then back to the caller when the method returns. It is not passed by reference; modifications by the called method are not observed until the method returns (value-result).
once	Only iterators may have 'once' arguments. Such arguments are evaluated exactly once, the first time the iterator is encountered in the containing loop. 'once' arguments otherwise behave as 'in' arguments.

2.7.3b Mode examples

This routine swaps the values of its arguments. If the arguments were not designated 'inout', calling the routine would have no effect.

```
swap(inout x, inout y:T) is
    temp::=x;
    x:=y;
    y:=temp
end
```

This iterator returns (head, tail) edges of a graph. 'out' arguments are convenient when one wants to return multiple values.

```
edges!(out head, out tail:V) is ... end
```

2.7.3c Overloading and dispatch

Sather supports routine and iterator _overloading_. In addition to the name, the number, types, and modes of arguments in a call and whether a return value is used all contribute to the selection of the method. The *modal_list* portion of a call must supply an expression corresponding to each declared argument of the method. There must exist a method with the specified name such that:

1. for each 'in' and 'once' argument, the type of each expression is a subtype of the declared type of the corresponding argument, and
2. for each 'out' argument, the type of each expression is a supertype of the corresponding argument, and
3. for each 'inout' argument, the type of each expression is the exact type of the corresponding argument.

If there is more than one such method, there must be a unique one which is *most specific*, conforming to all others. When argument expressions have the type of a class parameter, the type constraint of that parameter is used to select the most specific method, rather than the realized type of the parameter. Overloading may not occur solely by the type of out arguments or return type; there must be at least one non-out argument of differing type between the most specific method and any others.

Sather also supports dynamic *dispatch* on the type of self when the expression on which the call is made has an abstract declared type. The method matching the call from the runtime type is actually executed. Because of the fundamental subtyping rule (page 20), if the abstract type specifies a conforming method, so will the type of the returned object.

2.7.4 void *expressions*

Example:

```
void
```

Syntax:

void_expression ⇒ void

A *void expression* returns a value whose type is determined from context. void is the value that a variable of the type receives when it is declared but not explicitly initialized. The value of void for abstract, reference, and bound variables is a special value that represents the absence of a reference to an object. The value of void for boolean variables is false (page 40) and for other immutable types it is determined by recursively setting each attribute and array element to void. The built-in immutable types are defined in terms of arrays of BOOL and so have all their bits set to false by this rule. For numerical types, this results in the appropriate version of 'zero' (see page 61).

void expressions may appear as the initializer for a constant or shared attribute, as the right hand side of an assignment statement, as the return value in a return or yield statement, as the value of one of the expressions in a case statement, as the exception object in a raise statement, or as an argument value in a method call or in a creation expression (page 47). In this last case, the argument is ignored in resolving overloading.

It is a fatal error to access object attributes of a void variable of reference type or to make any calls on a void variable of abstract type. An explicit 'void' expression may not appear as the left argument of the dot '.' operator (page 44).

2.7.5 void *test expressions*

Example:

```
void(x)
```

Syntax:

void_test_expression ⇒ void (*expression*)

Void test expressions evaluate their argument and return a boolean value which is true if the value is void (page 46).

2.7.6 new *expressions*

Examples:

```
new
new(17)
```

Syntax:

new_expression ⇒ new [(*expression*)]

new expressions are used to allocate space for reference objects and may only appear in reference classes. They return reference objects of type SAME. All non-shared attributes and array elements are initialized to void (page 46). If there is an include path from the type in which the new appears to AREF (page 61), then new must be provided with a non-negative INT argument which specifies the number of array elements in the returned object.

2.7.7 *Creation expressions*

Examples:

```
#FOO(1,2,3)
#(1,2,3)
#FOO
#
```

Syntax:

create_expression ⇒ # [*type_specifier*] [(*modal_list*)]

Immutable and reference object *creation expressions* are a convenient shorthand used for creating new objects and initializing their attributes. A creation expression is a special syntactic sugar for a call on a routine named 'create' with the specified arguments. 'self' is given the default void value described on page 46 in this call. The type defining the 'create' routine may be explicitly specified as a reference or immutable type. If the type is not explicitly specified, then it is taken to be the declared type of the context in which the call appears (and it must be an immutable or reference type). A type must be specified when it cannot be inferred from context: if the expression appears as the right hand side of a '::=' assignment (page 36), as a method argument in which overloading resolution would otherwise be ambiguous, or as the left argument of the dot '.' operator (page 44).

2.7.8 *Array creation expressions*

Examples:

```
|2,4,6,8|
|"apple", "orange", "cherry", "kiwi"|
```

Syntax:

array_expression ⇒ | *expression_list* |

expression_list ⇒ *expression* { , *expression* }

Array creation expressions are used to create and directly specify the elements of an array object. The type is taken to be the declared type of the context in which it appears and it must be ARRAY{T} for some type T. An array creation expression may not appear as the right hand side of a '::=' assignment (page 36), as a method argument in which the overloading resolution is ambiguous, or as the left argument of the dot '.' operator (page 44). The types of each expression in the *expression_list* must be subtypes of T. The size of the created array is equal to the number of specified expressions. The expressions are evaluated left to right and the results are assigned to successive array elements.

2.7.9 and *expressions*

Example:

```
0<=x and x<10
```

Syntax:

and_expression ⇒ *expression* and *expression*

and expressions compute the conjunction of two boolean expressions and return boolean values. The first expression is evaluated and if false, false is immediately returned as the result. Otherwise, the second expression is evaluated and its value returned.

2.7.10 or *expressions*

Example:

```
x=2 or x=3
```

Syntax:

or_expression ⇒ *expression* or *expression*

<u>*or expressions*</u> compute the disjunction of two boolean expressions and return boolean values. The first expression is evaluated and if true, true is immediately returned as the result. Otherwise, the second expression is evaluated and its value returned.

2.7.11 *Syntactic sugar expressions*

Examples:

```
a+b
x<7
```

Syntax:

sugar_expression ⇒ *expression binary_op expression*
| - *expression*
| *[expression]* [*expression_list*]
| (*expression*)

binary_op ⇒ + | - | * | / | ^ | % | ~ | < | <= | = | /= | > | >=

As shown in the following table, several Sather constructs are simply <u>*syntactic sugar*</u> for corresponding routine calls. Each of these transformations is applied after the component expressions have themselves been transformed. 'out' and 'inout' modes may not be used with the syntactic sugar expressions. Note that 'and' and 'or' are not listed as syntactic sugar for operations in 'BOOL'; this allows short-circuiting the evaluation of subexpressions. The '<=' and '>' expressions do not reverse the original left to right order of argument evaluation. Ambiguity between unary minus and negative literals must be resolved with explicit parenthesis.

Sugar form	*Translation*	*Sugar form*	*Translation*
expr1 + *expr2*	*expr1*.plus(*expr2*)	*expr1* /= *expr2*	*expr1*.is_eq(*expr2*).not
expr1 - *expr2*	*expr1*.minus(*expr2*)	*expr1* > *expr2*	*expr2*.is_lt(*expr1*)
expr1 * *expr2*	*expr1*.times(*expr2*)	*expr1* >= *expr2*	*expr1*.is_lt(*expr2*).not
expr1 / *expr2*	*expr1*.div(*expr2*)	- *expr*	*expr*.negate
expr1 ^ *expr2*	*expr1*.pow(*expr2*)	~ *expr*	*expr*.not
expr1 % *expr2*	*expr1*.mod(*expr2*)	[*expr_list*]	aget(*expression_list*)
expr1 < *expr2*	*expr1*.is_lt(*expr2*)	*expr1*[*expression_list*]	*expr1*.aget(*expression_list*)
expr1 <= *expr2*	*expr2*.is_lt(*expr1*).not	(*expression*)	*expression*
expr1 = *expr2*	*expr1*.is_eq(*expr2*)		

The precedence ordering shown below determines the grouping of the syntactic sugar forms. Symbols of the same precedence associate left to right and parentheses may be used for explicit grouping. Evaluation order obeys explicit parenthesis in all cases.

Strongest	. :: [] ()
	^
	~ Unary -
	* / %
	+ Binary -
	< <= = /= >= >
Weakest	and or

2.7.11a **Syntactic sugar example**

Here's a formula written with syntactic sugar and the calls it is textually equivalent to. It doesn't matter what the types of the variables are; the sugar ignores types.

```
-- Written using syntactic sugar
r := (x^2 + y^2).sqrt;

-- Written without sugar
r := (x.pow(2).plus(y.pow(2))).sqrt
```

2.8 LOOPS AND ITERATORS

Iterator definitions were described on page 32. Iterators are used extensively in Sather to control loops. This section elaborates their semantics and describes the built-in iterators that correspond to statements such as 'while' and 'do' found in other languages.

2.8.1 loop *statements*

Example:

```
loop ... end
```

Syntax:

loop_statement ⇒ loop *statement_list* end

Iteration is done with *loop statements*, used in conjunction with iterator calls. An execution state is maintained for each textual iterator call. When a loop is entered, the execution state of all enclosed iterator calls is initialized. When an iterator is first called in a loop, the expressions for self and for each once argument are evaluated left to right. Then the expressions for arguments which are not once (in or inout before the call, out or inout after the call; see page 44) are evaluated left to right. On subsequent calls, only the expressions for arguments which are not once are re-evaluated. self and any once arguments retain their earlier values. The expressions for self and for once arguments may not themselves contain iterator calls (such iters would only execute their first iteration.) Method call semantics are detailed on page 44.

When an iterator is called, it executes the statements in its body in order. If it executes a yield statement, control is returned to the caller. Subsequent calls on the iterator resume execution with the statement following the yield statement. If an iterator executes quit or reaches the end of its body, control passes immediately to the end of the innermost enclosing loop statement in the caller and no value is returned.

2.8.2 yield *statements*

Examples:

```
yield
yield x
```

Syntax:

yield_statement ⇒ yield *[expression]*

yield statements are used to return control to a loop and may appear only in iterator definitions. The *expression* clause must be present if the iterator has a return value and must be absent if it does not. If *expression* is present, then its type must be a subtype of the return type of the iterator. Execution of a yield statement causes the expression to be evaluated and its value to be returned to the caller of the iterator in which it appears. Yield is not permitted within a protect statement (see page 57). Yield causes assignment to out and inout arguments in the caller (page 44).

2.8.3 quit *statements*

Example:

```
quit
```

Syntax:

quit_statement ⇒ quit

quit statements are used to terminate loops and may only appear in iterator definitions. No value is returned from an iterator when it quits, and no assignment takes place to out or inout arguments in the caller (page 44). No statements may follow a quit statement in a statement list.

2.8.4 while! *expressions*

Example:

```
while!(a<10)
```

Syntax:

while!_expression ⇒ while!(*expression*)

while! expressions are iterator calls which take a single boolean argument that is re-evaluated on each iteration. They yield when the argument is true and quit when it is false.

2.8.5 until! *expressions*

Example:

```
until!(a>10)
```

Syntax:

until!_expression ⇒ until!(*expression*)

until! expressions are iterator calls which take a single boolean argument that is re-evaluated on each iteration. They yield when the argument is false and quit when it is true.

2.8.6 break! *expressions*

Example:

```
break!
```

Syntax:

break!_expression ⇒ break!

break! expressions are iterator calls which immediately quit when they are called.

2.8.6a Iterator Examples

Because they are so useful, the 'while!', 'until!' and 'break!' iterators are built into the language. Here's how 'while!' could be written if it were not a primitive.

```
while!(pred:BOOL) is
   -- Yields as long as 'pred' is true
   loop
      if pred then yield
      else quit
      end
   end
end
```

The built-in class 'INT' defines some useful iterators. Here's the definition of 'upto!'. Unlike the argument 'pred' used above, 'i' here is declared to be 'once'; when 'upto!' is called, the argument is only evaluated once, the first time the iterator is called in the loop.

```
upto!(once i:SAME):SAME is
   -- Yield successive integers from
   -- self to `i' inclusive.
   r::=self;
   loop
      until!(r>i);
      yield r;
      r:=r+1
   end
end;
```

To add up the integers 1 through 10, one might say:

```
x::=0;
loop
   x:=x+1.upto!(10)
end
```

Or, using the library iterator 'sum!' like this. 'x' needs to be declared (but not initialized) outside the loop.

```
loop
   x:=INT::sum!(1.upto!(10))
end
```

Some of the most useful ways to use iters is with container objects. Arrays, lists, sets, trees, strings, and vectors can all be given iterators to yield all their elements. Here we print all the elements of some container 'c'.

```
loop
    #OUT + c.elt!.str + '\n'
end
```

This doubles the elements of array 'a'.

```
loop a.set!(a.elt! * 2) end
```

This computes the dot product of two vectors 'a' and 'b'. There is also a built-in method 'dot' to do this. 'x' needs to be declared (but not initialized) before the loop.

```
loop
    x:=sum!(a.elt! * b.elt!)
end
```

2.9 CLOSURES

Routine and iter <u>*closures*</u> are similar to the 'function pointer' and 'closure' constructs of other languages. They bind a reference to a method together with zero or more argument values (possibly including 'self').

2.9.1 *Closure creation expressions*

Examples:

```
bind(2.plus(_))
bind(filter!(bind(is_eq(a))))
```

Syntax:

> *method_closure_create_expression* ⇒ *routine_closure_create_expression* |
> *iter_closure_create_expression*
>
> *routine_closure_create_expression* ⇒ bind
> (*[type_specifier* :: | *routine_closure_argument* . *]* *identifier*
> *[* (*routine_closure_argument* *{* , *routine_closure_argument* *}*) *]*)
>
> *routine_closure_argument* ⇒ *routine_mode* (*expression* | _ *)*
>
> *iter_closure_create_expression* ⇒ bind
> (*[type_specifier* :: | *iter_closure_argument* . *]* *iter_name*
> *[* (*iter_closure_argument* *{* , *iter_closure_argument* *}*) *]*)
>
> *iter_closure_argument* ⇒ *iter_mode* (*expression* | _ *)*

The outer part of the expression is 'bind(...)'. This surrounds a routine or iterator call in which any of the arguments or self may have been replaced by the underscore character '_'. Such unspecified arguments are <u>*unbound*</u>. Unbound arguments are specified when the closure is eventually called. 'out' and 'inout' arguments must be left unbound. In case of ambiguity, the signature of the method specified in the 'bind(...)' expression is inferred from context. The same overloading resolution rules as for the method call expressions (page 45) apply to the closure creation expressions.

The expressions in this construct are evaluated from left to right and the resulting values are stored as part of the closure. Closure creation expressions return closure types. As previously described on page 21, the type specifiers for these types have the form:

method_closure_type_specifier ⇒ *routine_closure_type_specifier* |
iter_closure_type_specifier

routine_closure_type_specifier ⇒ ROUT
[{ *routine_mode type_specifier {* , *routine_mode type_specifier }* } *]*
[: *type_specifier]*

iter_closure_type_specifier ⇒ ITER
[{ *iter_mode type_specifier {* , *iter_mode type_specifier }* } *]*
[: *type_specifier]*

These specifiers begin with the keywords 'ROUT' or 'ITER' and are followed by the modes and types of the underscore arguments, if any, enclosed in braces (*e.g.* 'ROUT{A, out B, inout C}', 'ITER{once A, out B, C}'). These are followed by a colon and the return type, if there is one (*e.g.* 'ROUT{INT}:INT', 'ITER{once INT}:FLT').

2.9.2 *Closure calls*

Each routine closure defines a routine named 'call' and each iterator closure defines an iterator named 'call!'. These have argument and return types that correspond to the closure type specifiers. Invocations of these features behave like a call on the original routine or iterator with the arguments specified by a combination of the bound values and those provided to 'call' or 'call!'. The arguments to 'call' and 'call!' match the underscores positionally from left to right (*e.g.* 'i::=bind(2.plus(_)).call(3)' is equivalent to 'i::=2.plus(3)').

Closure types implicitly introduce edges into the type graph. There is an edge from each closure type g to all closure types f that satisfy the contravariant requirement that

1. f and g have the same name and number of arguments,
2. f and g either both return a value or neither does,
3. the mode of each argument is the same (in, out, inout, or once),

4. *contravariant conformance*:
for any in or once arguments, the type in *g* is a subtype of the type in *f*;
for any inout arguments, the type in *f* is the same type as in *g*;
for any out arguments, the type in *f* is a subtype of the type in *g*; and
if it has one, the return type of *f* is a subtype of the return type of *g*.

For example, 'ROUT{$OB}:INT' is a subtype of 'ROUT{INT}:$OB' and 'ITER{once $OB}' is a subtype of 'ITER{once INT}'.

2.9.2a **Closure Examples**

Here we double every element of an array by applying a routine closure 'r' to each element of an array 'a'.

```
r :ROUT{INT}:INT := bind(2.0.times(_));
loop
    a.set!(r.call(a.elt!))
end
```

This illustrates how 'self' may be left unbound. The type of self must be inferredfrom the type context (ROUT{INT}).

```
r :ROUT{INT} := bind(_.plus(3));
#OUT + r.call(5);        -- prints '8'
```

This creates an iterator closure that returns successive odd integers, and then prints the first ten.

```
odd_ints : ITER{INT}:INT;
odd_ints := bind(1.step!(_,2));
loop
    #OUT + odd_ints.call!(10);
end
```

An iterator closure is created that may be used to extract elements of a map that satisfy the selection criteria defined by 'select'.

```
select:ROUT{T}:BOOL;
select_elt:ITER{FMAP{K,T}}:T;
...
select_elt := bind(_.filter!(select));
```

By the conformance rule above, 'ROUT{$OB}:INT' is a subtype of 'ROUT{INT}:$OB'. *The ICSI compiler does not yet allow such contravariant closure assignments.*

```
a:ROUT{$OB}:INT;
b:ROUT{INT}:$OB;
b := a;     -- This is a legal assignment
a := b;     -- This is not.
```

2.10 EXCEPTIONS

Exceptions are used to escape from method calls under unusual circumstances. For example, a robust numerical application may wish to provide an alternate means of solving a problem under unusual circumstances such as ill conditioning. Exceptions bypass the ordinary way of returning from methods and may be used to skip over multiple callers until a suitable handler is found.

Exceptions may be thought of as implicit alternate return values for all methods. Exceptions can be significantly slower than ordinary routine calls, so they should be avoided except for truly exceptional (unexpected) cases.

2.10.1 protect *statements*

Example:

```
protect ...
when E then ...
when $F then ...
else ...
end
```

Syntax:

> *protect_statement* ⇒ protect *statement_list*
> *{* when *type_specifier* then *statement_list }*
> *[* else *statement_list]* end

Sather uses *exceptions* to signal and recover from exceptional situations. Exceptions may be explicitly raised by a program (page 58) or generated by the system. Each exception is represented by an *exception object* whose type is used to select a handler from a *protect statement*. Execution of a protect statement begins with the statement list following the 'protect' keyword. These statements are executed to completion unless an exception is raised which is not caught by some nested protect.

When there is an uncaught exception in a protect statement, the system finds the first type specifier listed in the 'when' lists which is a supertype of the exception object type. The statement list following this specifier is executed and then control passes to the statement following the protect statement. An exception expression (page 59) may be used to access the exception object in these handler statements. If none of the specified types are supertypes, then the statements in an 'else' clause are executed if it is present. If it is not present, the same exception object is raised to the next most recently entered protect statement which is still in progress. It is a fatal error to raise an exception which is not handled by some protect statement. protect statements may only contain iterator calls if they also contain the surrounding loop statement. protect statements without an else clause must have at least one when.

2.10.2 raise *statements*

Example:

```
raise x
```

Syntax:

raise_statement ⇒ raise *expression*

Exceptions are explicitly raised by *raise statements*. The *expression* is evaluated to obtain the exception object. No statements may follow a raise statement in a statement list because they can never be executed.

2.10.3 exception *expressions*

Example:

```
exception
```

Syntax:

except_expression ⇒ exception

exception expressions may only appear within the statements of the then and else clauses in protect statements. They return the exception object that caused the when branch to be taken in the most tightly enclosing protect statement. The return type is the type specified in the corresponding when clause (page 57). In an else clause the return type is '$OB'.

2.11 Safety Features

Methods definitions may include optional pre- and post-conditions (page 32). Together with 'assert' and 'invariant' (page 60), these features allow the earnest programmer to annotate the intention of code. Implementations of Sather must provide facilities for turning on and off the runtime checking that these features require. The behavior of code that fails to meet the stated safety assertions is undefined. This allows an optimizing compiler to exploit the stated assertions even if they are not checked.

2.11.1 *Pre- and post-conditions*

The optional 'pre' construct of method definitions contains a boolean expression which must evaluate to true whenever the method is called; it is a fatal error if it evaluates to false.

The expression may refer to self and to the routine's arguments. For iterators, pre and post conditions are checked before and after every invocation, not just once per loop.

The optional 'post' construct of method definitions contains a boolean expression which must evaluate to true whenever the method returns; it is a fatal error if it evaluates to false. The expression may refer to self and to the routine's arguments. It may use 'result' expressions (page 60) to refer to the value returned by the routine and 'initial' expressions (page 59) to refer to values which are computed before the routine executes.

Classes may also define 'invariant', which is a post condition that applies to all public methods (page 60).

2.11.2 assert *statements*

Example:

```
assert x>5
```

Syntax:

assert_statement ⇒ assert *expression*

assert statements specify a boolean expression that must evaluate to true; otherwise it is a fatal error.

2.11.3 initial *expressions*

Example:

```
add(a: INT):INT
    post initial(a) > result  is ...
```

Syntax:

initial_expression ⇒ initial (*expression*)

initial expressions may only appear in the post expressions of methods. The *expression* must have a return value and must not itself contain initial expressions. When a routine is called or an iterator resumes, it evaluates the *expression* of each initial expression from left to right. When the postcondition is checked at the end, each initial expression returns its pre-computed value.

2.11.4 result *expressions*

Example:

```
sum: INT
    post result > 5 is ...
```

Syntax:

result_expression ⇒ result

result expressions may only appear within the postconditions of methods that have return values and may not appear within initial expressions. They return the value returned by the routine or yielded by the iterator. Their type is the return type of the method in which they appear.

2.12 SPECIAL FEATURE NAMES

This section describes features of classes that have special properties.

2.12.1 invariant

If a routine with the signature 'invariant:BOOL', appears in a class, it defines a class invariant. It is a fatal error for it to evaluate to false after any public method of the class returns, yields, or quits.

2.12.2 main

A distinguished non-parameterized immutable or reference class is specified when a Sather program is compiled, usually 'MAIN'. This class must define a routine named 'main'. When the program executes, an object of the specified type is created and 'main' is called on it. If main is declared to have an argument of type ARRAY{STR}, it will be passed an array of any command line arguments provided by the environment when the program is called. If it is declared to have a return value of type INT, this will specify the exit code of the program when it finishes execution.

2.13 BUILT-IN CLASSES

This section provides a short description of classes that are a part of every Sather implementation and which may not be modified. The detailed semantics and precise interface are specified in the class library documentation.

- '$OB' is automatically a supertype of every type. Variables declared by this type may hold any object. It has no features.
- 'AREF{T}' is a reference array class. Any reference class which includes it obtains an array of elements of type T in addition to any attributes it has defined. In such classes, new has a single integer argument that specifies the size of the array portion. It defines routines and iters named: 'asize', 'aget', 'aset', 'aclear', 'acopy', 'aelt!', 'aset!', and 'aind!'. Array indices start at zero. 'AVAL{T}' is the immutable class analog of 'AREF'. Classes which include 'AVAL' must define asize as an integer constant which determines the size of the array portion. 'ARRAY{T}' includes from 'AREF' and defines general purpose array objects. They may be directly constructed by array creation expressions (page 48).
- 'TUP' names a set of parameterized immutable types called tuples, one for each number of parameters. Each has as many attributes as parameters and they are named 't1', 't2', etc. Each is declared by the type of the corresponding parameter (*e.g.* 'TUP{INT,FLT}' has attributes 't1:INT' and 't2:FLT'). It defines 'create' with an argument corresponding to each attribute.
- The literal form for a number of primitive types were introduced on page 40:

Type	Initial value	Description
BOOL	false	Immutable objects which represent boolean values.
CHAR	'\0'	Immutable objects which represent characters. The number of bits in a 'CHAR' object is less than or equal to the number in an 'INT' object.
STR	"" (void)	Reference objects which represent strings for characters. 'void' is a representation for the null string.
INT	0	Immutable objects which represent efficient integers. The size is defined by the Sather implementation but must be at least 32 bits. The two's complement representation is used to represent negative values. Bit operations are supported in addition to numerical operations.
INTI	0i	Reference objects which represent infinite precision integers.
FLT	0.0	Immutable objects which represent single precision floating point values as defined by the IEEE-754-1985 standard.
FLTD	0.0d	Immutable objects which represent double precision floating point values.
FLTI	0.0i	Reference objects which represent arbitrary precision floating point objects.

- SYS defines a number of routines for accessing system information:

Routine	Description
is_eq(ob1, ob2:$OB):BOOL	Tests two objects for equality. If the arguments are of different type, it returns 'false'. If both objects are immutable, this is a recursive test on the arguments' attributes. If they are reference types, it returns 'true' if the arguments are the same object. It is a fatal error to call with external, closure, or void reference arguments.
is_lt(ob1, ob2:$OB):BOOL	Defines an arbitrary total order on objects. This never returns true if 'is_eq' would return true with the same arguments. It is a fatal error to call with external, closure, or void reference arguments.
hash(ob:$OB):INT	Defines an arbitrary hash function. For reference arguments, this is a hash of the pointer; for immutable types, a recursive hash of all attributes. Hash values for two objects are guaranteed to be identical when 'is_eq' would return true, but the converse is not true.
type(ob:$OB):INT	Returns the concrete type of an object encoded as an 'INT'.
str_for_type(i:INT):STR	Returns a string representation associated with the integer. Useful for debugging in combination with 'type' above.
destroy(ob:$OB)	Explicitly deallocates an object. Sather is garbage collected and casual use of 'destroy' is discouraged. Sather implementations must provide a way of detecting accesses to destroyed objects (a fatal error).

- $FINALIZE defines the single routine finalize. Any class whose objects need to perform special operations before they are garbage collected should subtype from $FINALIZE. The finalize routine will be called once on such objects before the program terminates. This may happen at any time, even concurrently with other code, and no guarantee is made about the order of finalization of objects which refer to each other. Finalization will only occur once, even if new references are created to the object during finalization. Because few guarantees can be made about the environment in which finalization occurs, finalization is considered dangerous and should only be used in the rare cases that conventional coding will not suffice.

2.14 Conventions

This section presents conventions used throughout the standard Sather libraries. Some conventions regard naming, while others dictate subtyping from certain abstract classes in the base library. Adhering to these conventions allows code from independent developers to be used together and makes code easier to understand.

2.14.1 *Object Creation*

Sather provides a special syntactic sugar (page 49) for calls to the routine 'create', which nearly all classes define. It is often convenient to overload 'create' routines to do conversion between types as well.

This is the canonical 'create' routine, which simply returns an uninitialized (page 47) object.

```
create:SAME is
    return new
end
```

This is a 'create' routine for an object with an array portion. Such objects require an integer argument to 'new'. This example creates a new object with 10 array elements indexed zero through nine.

```
create:SAME is
    return new(10)
end
```

This 'create' routine converts a string to an object. This could be used in combination with still other create routines to convert from different types.

```
create(arg:STR):SAME is
    ...
end
```

All three create routines shown could be invoked with the '#' sugar (page 47).

2.14.2 *Naming*

In addition to 'create', there are a number of other naming conventions:

- Classes which are related should reflect this in their names. For example, there are many examples in the library of an abstraction, classes implementing the abstraction, and code testing implementations of the abstraction. For example, in the standard library the set abstraction is named $SET, H_SET is a hash table implementation, and the test code is TEST_SET.
- Some classes implement an immutable, 'mathematical' abstraction (*eg*. integers), and others implement mutable abstractions that can be modified in place (*eg*. arrays). Sometimes is is possible to have both immutable and mutable abstractions for the same concept. The former are usually easier to reason about and safer to program with, while the later can be more efficient. It is very important not to confuse the two: objects that at any time represent a set are not sets themselves. They have entirely different properties, such as observable aliasing and having an equality relation that changes over time.

Classes with immutable semantics are given their 'mathematical' names: STR, VEC, $SET. Mutable classes that represent a value that may change over time have the prefix 'OB': OBSTR, OBVEC, $OBSET. When both versions exist, the method 'value' is used to take a 'snapshot' of the object's state, converting from the mutable to the immutable form.

- Conversions from a type FOO to a type BAR occur in two ways: by defining an appropriate 'create' routine in BAR as seen above, or be defining a routine 'bar:BAR' in FOO. For example, in the standard library conversion of a FLT to a FLTD is done by calling the routine 'fltd:FLTD' defined in FLT.
- Methods which return a BOOL (*predicates*) should have the prefix 'is_'. For example, 'is_prime' tests integers for primality.
- Abstract classes that require a single method should be named after that method. For example, subtypes of $HASH define the method 'hash'.
- If there is a single iterator in a container class which returns all of the items, it should be named 'elt!'. If there is a single iterator which sets the items, it should be named 'set!'. In general, iterators should have singular ('elt!') rather than plural ('elts!') names if the choice is arbitrary.

2.14.3 *Object Identity*

Many languages provide built-in pointer and structural equality and comparison. To preserve encapsulation, in Sather these operations must go through the class interface like every method. The '=' symbol is syntactic sugar for a call to 'is_eq' (page 49). 'is_eq:BOOL' must be explicitly defined by the type of the left side for this syntax to be useful.

The SYS class (page 62) can be used to obtain equality based on pointer or structural notions of identity. This class also provides built-in mechanisms for comparison and hashing.

Classes which define their own notion of equality should subtype from $IS_EQ. This class is a common parameter bound in container classes.

```
-- In standard library
type $IS_EQ is
    is_eq(e:$OB): BOOL;
end
```

Many classes define a notion of equality which is different than pointer equality. For example, two STR strings may be equal although they are not unique.

```
class STR < $IS_EQ is
    ...
    is_eq(arg:$OB):BOOL is ... end;
    ...
end
```

Many container classes need to be able to compute hash values of their items. Just as with ‘is_eq’, classes may subtype from $HASH to indicate that they know how to compute their own hash value. ID also provides this built-in hash function.

```
-- In standard library
type $HASH is
    hash:INT;
end
```

To preserve class encapsulation, Sather does not provide a built-in way to copy objects. By convention, objects are copied by a class-defined routine ‘copy’, and classes which provide this should subtype from $COPY.

```
-- In standard library
type $COPY is
    copy:SAME;
end
```

2.14.4 *Nil and void*

Reference class variables can be declared without being allocated. Unassigned reference or abstract type variables have the void value, indicating the non-existence of an object (page 46). However, for immutable types this unassigned value is not distinguished from other legitimate values; for example, the void of type INT is the zero.

It is often algorithmically convenient to have a sentinel value which has a special interpretation. For example, hash tables often distinguish empty table entries without a separate bit indicating that an entry is empty. Because void is a legitimate value for immutable types, void can’t be used as this sentinel value. For this reason, classes may define a ‘nil’ value to be used to represent the non-existence of an immutable object. Such classes subtype from $NIL and define the routines ‘nil:SAME’ and ‘is_nil: BOOL’.

The ‘nil’ value is generally a rarely used or illegal value. For INT, it is the most negative representable integer. For floating point types, it is NaN. ‘is_nil’ is necessary because NaN is defined by IEEE to not be equal to itself.

```
type $NIL is  -- In standard library
    nil: SAME;
    is_nil: BOOL;
end
```

2.14.5 *IEEE Floating-Point*

Sather attempts to conform to the IEEE 754-1985 specification for its floating point types. Unfortunately, many platforms make it difficult to do so. For example, underflow is often improperly implemented to flush to zero rather than use IEEE’s gradual underflow. This happens because gradual underflow is a special case and can be quite slow if implemented using traps. When benchmarks include simulations which cause many underflows, marketing pressures make flush-to-zero the default.

There are many other problems. Microsoft's C and C++ compilers defeat the purpose of the invalid flag by using it exclusively to detect floating-point stack overflows, so programmers cannot use it. There is no portable C interface to IEEE exception flags and their behavior with respect to 'setjmp' is suspect. Threads packages often fail to address proper handling of IEEE exceptions and rounding modes.

Correct IEEE support from various platforms was the single worst porting headache of the Sather 1.0 compiler. In 1.1, we give up and make full IEEE compliance optional. Sather implementations are expected to conform to the *spirit*, if not the letter, of IEEE 754, although proper exceptions, extended types, underflow handling, and correct handling of positive and negative zero are specifically *not* required.

The Sather treatment of NaNs is particularly tricky; IEEE wants NaN to be neither equal nor unequal to anything else, including other NaNs. Because Sather defines 'x /= y' as 'x.is_eq(y).not' (page 49), to get the IEEE notion of unequal is necessary to write 'x=x and y=y and x/=y'. Other comparison operators present similar difficulties.

Sather 1.1 Extensions

All Sather 1.1 implementations must support the language kernel defined in the last chapter. This chapter defines language extensions which may not be meaningful on every platform or which can be very difficult to implement. For example, platforms without a Fortran compiler need not implement the Fortran language interface.

Although these extensions are optional, they should be considered part of the Sather specification. For example, Sather 1.1 implementations which interface to Fortran must provide the language extension described here. The ICSI compiler supports all extensions described in this chapter on one or more platforms.

The threaded and synchronization extensions enable parallel processing. The synchronization and distributed extensions are only of use with the threaded extension. Collectively, these three extensions are known as *pSather*, and the language without these extensions is *serial Sather*.

3.1 LANGUAGE INTERFACE EXTENSIONS

External classes are used to interface with code from other languages. Each external class is typically associated with an object file compiled from a language like C or Fortran. Each language identifier is associated with a Sather language extension. The extensions defined here are:

- FORTRAN: interface to a superset of ANSI Fortran 77, X3.9-1978
- C: interface to ANSI C, X3.159-1989

Interfaces to other languages, or alternate interfaces to C and Fortran, may be designated in the future using other identifiers.

Each external language extension may have its own restrictions on what may legally appear in an external class of that language and what the semantic interpretation of the external class contents is. The legality and semantics of subtyping and code inclusion are defined by the language extension. For the C and Fortran extensions, routines that have no body (abstract signatures) specify the interface for Sather code to call external code. Calls to such routines are compiled using the external language's name binding and parameter passing conventions. Routines with a body in an external class specify the interface for external code to call Sather code; such routines also use the external language's name binding and parameter passing conventions. No routines and signatures in external classes may conflict (page 22) , and the corresponding external object file must provide a function that conforms according to the rules of the language interface. There may be platform specific transformations of external routine names (*e.g.* prepended underscore or truncation); it is an error if the external language namespace implementation does not permit the resulting name. The implementations or environments of other languages may impose other unavoidable constraints.

3.1.1 *Interfacing with Fortran*

An external class which interfaces to Fortran is designated with the language identifier 'FORTRAN'. The following Fortran types are built into the extended library:

Sather type	*Fortran type*	< $F_SCALAR*?*
F_REAL	real	*Yes*
F_DOUBLE	double precision	*Yes*
F_INTEGER	integer	*Yes*
F_COMPLEX	complex	*Yes*
F_DOUBLE_COMPLEX	double precision complex	*Yes*
F_LOGICAL	logical	*Yes*
F_CHARACTER	character, character*1	*Yes*
F_STRING	character**n*	*No*
F_ARRAY{*T*} F_ARRAY*n*{*T*}	*array types:* T < $F_SCALAR *n=2, 3...*	*No*
F_ROUT{...}	*subroutine type*	*No*
F_HANDLER	*exception handler type*	*No*

These external types also define appropriate creation routines which may be used for convenient casting between Sather and Fortran types. Only the types listed above may be used in the abstract signatures defined in a Fortran external class. Methods defined in external Fortran classes that have bodies (are not abstract signatures) may have other types in their signature, but if they do, these routines are not visible to Fortran code. Fortran external classes may not contain abstract iterator signatures and may not be parameterized. Routines without bodies in Fortran external classes may not be overloaded.

Fortran implementations pass arguments by reference. The scalar types have conventional immutable semantics (*i.e.* there is no aliasing visible within Sather) and are subtypes of $F_SCALAR. The other types maintain reference semantics. The F_ROUT{...} types behave like Sather routine closures and are created with a similar syntax (page 54), but all argument types must be Fortran types and at creation all arguments must be left unbound. Fortran exception handlers may be passed by passing an F_HANDLER object, which is constructed from a Sather routine closure.

The Sather implementation must assure that changes to 'out' and 'inout' arguments passed to a Fortran routine are not observed until the Fortran routine returns. It is a fatal error for a Fortran routine to modify arguments with 'in' mode.

3.1.2 *Interfacing with ANSI C*

An external class which interfaces to ANSI C is designated with the language identifier 'C'. Unlike external Fortran classes, external C classes may be instantiated (may point to runtime objects). External C classes may not be parameterized. Types defined by external C classes are called *external C types*.

In external C classes, signatures without bodies must only use external C types. Routines with bodies (are not abstract signatures) defined in external C classes may use other types, but if they do, these routines are not visible to C code. Routines that could be called from C may not be overloaded. 'out' and 'inout' arguments are passed by a pointer to a local, which may be legally modified by the routine. The Sather implementation must guarantee that such modifications cannot be observed until the routine returns.

Attributes may be placed in external C classes; they are interpreted as fields of a C struct. Such attributes may only have external C types. Similarly, shareds are interpreted as C global variables. Constants of external C types are interpreted as C constants or macros.

For example, this C code:

```
typedef struct { int a,b; } *FOO;
FOO xxyyz;
```

can be accessed by users of this Sather class:

```
external C class FOO is
    const C_name:STR:="FOO";
    attr a, b:C_INT;
    shared xxyyz:FOO;
end;
```

There are two features of external C classes that have a special semantics. The STR constant 'C_name' may be used to force a particular C declaration to be used for an external C type. Similarly the STR constant 'C_header' may specify a space separated list of C header files that must be emitted at the beginning of any file in which the C declaration appears. 'C_name' and 'C_header' which must be constant STR when present.

For example, this creates a Sather type 'X_WIDGET' which may be used to declare variables, parameterize classes, and so forth. Furthermore, the C declaration used for variables of type 'X_WIDGET' will be 'struct XSomeWidget *'. Any generated C file containing any variable of this type will also include '<widgets.h>'.

```
external C class X_WIDGET is
    const C_name:STR:=
        "struct XSomeWidget *";
    const C_header:STR:=
        "<widgets.h>";
end;
```

Sometimes it isn't possible to decide at the time the external C class is written whether a routine will be implemented in the C code with a macro. This presents a portability problem, because the writer of the external class can't know ahead of time whether the routine will be obtained by linking or by a header file. Such petulant cases can be dealt with by the call 'SYS::inlined_C'. The argument must be a string literal, and is placed directly into the generated code, except that identifiers following '#' that correspond to locals and arguments are translated into the appropriate C name. An alternate form accepts two arguments, making it possible to specify an include file or macro required by the inlined code, which will be placed at the top of the generated file.

Here's an example from the UNIX headers:

```
SYS::inlined_C("#res = EPERM",
               "#include <errno.h>\n");
```

The following C types are built into the extended library; these external types also define appropriate creation routines which may be used for convenient casting between Sather and C types.

Sather type	ANSI C type	Sather type	ANSI C type
C_CHAR	char	C_UNSIGNED_CHAR_PTR	unsigned char *
C_UNSIGNED_CHAR	unsigned char	C_SIGNED_CHAR_PTR	signed char *
C_SIGNED_CHAR	signed char	C_SHORT_PTR	short *
C_SHORT	short	C_INT_PTR	int *
C_INT	int	C_LONG_PTR	long *
C_LONG	long	C_UNSIGNED_SHORT_PTR	unsigned short *
C_UNSIGNED_SHORT	unsigned short	C_UNSIGNED_INT_PTR	unsigned int *
C_UNSIGNED_INT	unsigned int	C_UNSIGNED_LONG_PTR	unsigned long *
C_UNSIGNED_LONG	signed long	C_FLOAT_PTR	float *
C_FLOAT	float	C_DOUBLE_PTR	double *
C_DOUBLE	double	C_LONG_DOUBLE_PTR	long double *
C_LONG_DOUBLE	long double	C_SIZE_T	size_t
C_PTR	void *	C_PTRDIFF_T	ptrdiff_t
C_CHAR_PTR	char *		

In addition, 'AREF{T}' defines a routine 'array_ptr:C_PTR' which may be used to obtain a pointer to the first item in the array portion of Sather objects. The external routine may modify the contents of this array portion, but must not store the pointer; there is no guarantee that the pointer will remain valid after the external routine returns. This restriction ensures that the Sather type system and garbage collector will not be corrupted by external code, while not sacrificing efficiency for the most important cases.

3.2 THREADED EXTENSION

In serial Sather there is only one thread of execution; in pSather there may be many. Multiple threads are similar to multiple serial Sather programs executing concurrently, but threads share variables of a single namespace. A new thread is created by executing a *fork*, which may be a par or fork statement (page 72), parloop statement (page 73), or an attach (page 77). The new thread is a *child* of the forking thread, which is the child's *parent*. pSather provides operations that can *block* a thread, making it unable to execute statements until some condition occurs. pSather threads that are not blocked will eventually run, but there is no other constraint on the order of execution of statements between threads that are not blocked. Threads no longer exist once they *terminate*. When a pSather program begins execution it has a single thread corresponding to the main routine (page 60).

Serial Sather defines a total order of execution of the program's statements; in contrast, pSather defines only a partial order. This partial order is defined by the union of the constraints implied by the consecutive execution order of statements within single threads and pSather synchronization operations between statements in different threads. As long as this partial order appears to be observed it is possible for a pSather implementation to overlap multiple operations in time, so a child thread may run concurrently with its parent and with other children. Using threads may render programs nondeterministic. Preconditions, postconditions, and class invariants (page 58) may not work as intended when originally serial code is used with multiple threads.

The threaded extension may be implemented without the synchronization extension. This is only useful with data parallel code, in which it is not possible for threads to affect each other through side effects. Platforms may interpret such data parallelism in different ways, such as an opportunity for vectorization, or by executing only one 'thread' at a time.

3.2.1 par *and* fork *statements*

Example:

```
par
    fork ... end
end
```

Syntax:

fork_statement ⇒ fork *statement_list* end

par_statement ⇒ par *statement_list* end

Threads may be created with the *fork statement*, which must be syntactically enclosed in a *par statement*, which also implicitly creates a thread. When a fork statement is executed, it forks a *body thread* to execute the statements in its body. Local variables that are declared outside the body of the innermost enclosing par statement are shared among all threads in the par body. All threads created by a fork must complete before execution continues past the par. The rules for memory consistency apply to body threads, so they may not see a consistent picture of the shared variables unless they employ explicit synchronization (page 81).

Each body thread receives a unique copy of every local declared in the innermost enclosing par body. When body threads begin, these copies have the value that the locals did at the time the fork statement was executed. Changes to a thread's copy of these variables are never observed by other threads. Iterators may not occur in a fork or par statement unless they are within an enclosed loop. 'quit', 'yield', and 'return' are not permitted in a par or fork body.

As a generalization of serial Sather, it is a fatal error if an exception occurs in a thread which is not handled within that thread by some protect statement. Because par and fork bodies are executed as separate threads, an unhandled exception in their bodies is a fatal error.

3.2.1a **Par and Fork Examples**

In this code A and B can execute concurrently. After both A and B complete, C and D can execute concurrently. E must wait for A, B, C, and D to terminate before executing.

```
par
    par
        fork A end;
        B
    end;
    fork C end;
    D
end;
E
```

In this code, 'outer' is declared outside the par, so this variable is shared by the forked thread. However, because 'inner' is inside the par, the fork body receives its own local copy at the time of the fork.

```
outer:INT;
par
    inner:INT;
    fork
        -- fork body
    end
end
```

3.2.2 parloop *statement*

Example:

```
parloop c::=clusters! do ... end
```

Syntax:

parloop_statement ⇒ parloop *statement_list* do *statement_list* end

The *parloop statement* is syntactic sugar to make convenient a common parallel programming idiom.

parloop *S1* do *S2* end

is syntactic sugar for:

par loop *S1* fork *S2* end end end

3.2.2a Parloop example

This code applies ‘frobnify’ using a separate thread for each element of an array.

```
par
    loop e::= a.elt!;
        fork e.frobnify end
    end
end
```

Using the parloop shorthand, the same code could also be written:

```
parloop e: := a.elt! do
    e.frobnify
end
```

3.3 SYNCHRONIZATION EXTENSION

The synchronization extension allows threads to block; this requires threading facilities not available on every platform. Programmers should not assume that synchronization is less expensive than thread creation; creating threads as required may be more efficient than attempting to manage a pool of threads that wait for things to do. Generally, minimizing synchronization provides the greatest throughput.

3.3.1 lock *statement*

Examples:

```
lock
    when m then ...
    else ...
end;
lock
    guard d.size > 0 when m then ...
    when rw.writer then ...
end
```

Syntax:

lock_statement ⇒
  lock *expression* { , *expression* } then *statement_list* [else *statement_list*] end
  lock *lock_when* { *lock_when* } [else *statement_list*] end

lock_when ⇒
  [guard *expression*] when *expression* { , *expression* } then *statement_list*

Locks are special built-in synchronization objects that control the blocking and unblocking of threads. A thread *acquires* a lock, then *holds* the lock until it *releases* it. A single thread may acquire a lock multiple times recursively; it will be held until a corresponding number of releases occur. Exclusive locks, such as 'MUTEX', may only be held by one thread at a time. In addition to these simple exclusive locks, it is possible to lock on other more complex synchronization types (on page 76).

Locks may be safely acquired with the *lock statement*. Expressions following a 'when' or 'lock' are called *locking conditions*, and must be subtypes of $LOCK (page 76). The statement list following the 'then' is called the *lock branch*. A lock statement guarantees that all listed locks are atomically acquired before a lock branch executes. Expressions following a 'guard' are called *guarding conditions*. The statements following the 'else' are called the *else branch*. The 'when' is dropped in the first form, convenient when there is only a single locking condition and no guard.

When a lock statement is entered the following occur in strict order:

1. Any guarding conditions are evaluated in textual order. If any evaluate to 'false', the corresponding when clause will not be considered further. when clauses without a guarding condition or for which the condition evaluates to 'true' are *accepted*.
2. If no when clauses are accepted, the else branch executes; it is a fatal error if there is no else clause in such a case.
3. For all accepted clauses, all locking conditions are evaluated, in textual order, left to right.
4. If the locking conditions of some when clause can be immediately satisfied, those locks are obtained, the corresponding lock branch executes, and execution concludes without considering other accepted when clauses.
5. If there is an 'else' clause and no when clauses have lock conditions that can immediately be satisfied, then the else branch executes. If there is no 'else' clause, the executing thread blocks until the locking conditions of some when clause can be satisfied. After the locking conditions are locked atomically, the corresponding lock branch executes.

Because all listed locks are acquired atomically, deadlock can never occur due to concurrent execution of two or more lock statements with multiple locks, although it is possible for deadlock to occur by dynamic nesting of lock statements or through other synchronization.

The implementation of lock statements also ensures that threads that can run will eventually do so; no thread will face starvation because of the operation of the locking and scheduling implementation. Similarly, no when clause will be repeatedly chosen over another such that a clause starves. However, it is frequently good practice to have threads whose programmer supplied enabling conditions are never met in a given run (exceptional cases) or are not met after some time (alternative methods). One thread in an infinite loop can prevent other threads from executing for an arbitrary time, unless it calls SYS::defer (page 82).

All locks acquired by the lock statement are released when the lock or else branch stops executing; this may occur due to finishing the branch, termination of a loop by an iterator, a return, a quit, or an exception. yield may occur in a lock statement, but locks are not released until the iterator quits. Exceptions in a lock body will not be raised outside the body until all associated locks have been released.

3.3.2 unlock *statement*

Example:

```
unlock g
```

Syntax:

unlock_statement ⇒ unlock *expression*

Locks may also be unlocked before exiting the lock branch by an unlock statement. An unlock statement must be syntactically within a lock branch; in a par or fork statement an unlock must be inside an enclosed lock branch. It is a fatal error if the expression does not evaluate to a $LOCK object which is locked by the enclosing lock statement.

3.3.3 $LOCK *classes*

All synchronization objects subtype from $LOCK. In addition to primitive $LOCK classes, some synchronization classes return $LOCK objects to allow different kinds of locking. The concrete type of the returned object is dependent on the pSather implementation.

- MUTEX is a simple mutual exclusion lock. Two threads may not simultaneously lock a MUTEX.
- RW_LOCK is used to manage reader-writer synchronization, and defines two methods 'reader' and 'writer'. These return $LOCK objects. If 'rw' is an object of type RW_LOCK, then a lock on 'rw.reader' or 'rw.writer' blocks until no thread is locking on 'rw.writer', although multiple threads can simultaneously hold 'rw.reader'. Readers are granted priority over writers. Attempting to obtain a writer lock while holding the corresponding reader lock causes deadlock.
- WR_LOCK and FRW_LOCK also manage reader-writer synchronization. WR_LOCK gives writes priority over reads, while FRW_LOCK grants readers and writers equal priority.
- Classes under $ATTACH and $ATTACH{T} (page 78) also subtype from $LOCK.

3.3.3a Locking example

This code implements five dining philosophers. The philosophers are seated at a round table and forced to share a single chopstick with each neighbor. They alternate between eating and thinking, but eating requires both chopsticks.

```
chopsticks ::= #ARRAY{MUTEX}(5);
loop chopsticks.set!(#MUTEX) end;
parloop
    i::= 0.upto!(4);
do
    loop
        think;
        lock
        when chopsticks[i], chopsticks[(i+1).mod(5)]
        then eat
        end
    end
end
```

3.3.4 *Attach statement*

Example:

```
g :- forked_computation
```

Syntax:

attach ⇒ *expression* :- *expression*

Threads can be created by executing an *attach*. The left side must be of type '$ATTACH' or '$ATTACH{T}'. If the left side is of type '$ATTACH{T}', the return type of the right side must be a subtype of 'T'. If the left side is of type '$ATTACH', the right side must not return a value. There must be no iterators on the right side.

When an attach is executed, the following takes place in strict order:

1. The left side is evaluated.
2. $ATTACH and $ATTACH{T} both subtype from $LOCK. If the synchronization object of the left side is locked by another thread, the executing thread is suspended until it becomes unlocked.
3. Any local variables on the right side are evaluated.
4. A new thread is created to execute the right side. This new thread is *attached* to the synchronization object of the left side. The new thread receives a unique copy of every local variable; changes to these locals by the originating thread are not observed by the new thread. Similarly, if 'out' and 'inout' arguments occur on the right side, changes to local variables are not be observed by the originating thread. The rules for memory consistency (page 81) apply for other variables such as object attributes.
5. When execution of the right side completes, the new thread terminates, *detaches* itself, and enqueues the return value or increments the counter, if appropriate.

Attached threads may be thought of as producers that enqueue their return value (or increment a counter) when they terminate.

Every pSather thread is attached to exactly one $ATTACH object; even the main routine is attached to an unnamed object. The thread executing a par statement implicitly creates an $ATTACH object and forks a thread to execute the body. The newly created thread, as well as all threads created by fork statements syntactically in the par body, are attached to this same unnamed object. The thread executing a par statement blocks until there are no threads attached to the object. This ensures that all threads created by a fork have completed before execution continues past the par.

3.3.5 $ATTACH *classes*

There are four built-in $ATTACH classes; all subtype from $LOCK. These classes all have an implicit locked status (unlocked, or locked by a particular thread) and a set of attached threads.

- FUTURE{T} provides a handle to the result of a computation. It is an error to attach more than one thread to a future at a time.
- ATTACH allows multiple threads to be attached, but does not allow return values.
- *Gates* are powerful synchronization primitives which generalize fork/join, mailboxes, semaphores, and barrier synchronization. There is a typed GATE{T} that has a queue of values which must conform to 'T', and an unparameterized class GATE with only an integer counter.

In addition to thread attachment, these classes support the operations listed in the following tables 1, 2, 3, and 4. Some operations are *exclusive*: these lock the gate before proceeding and unlock it when the operation is complete. The exclusive operations also perform imports and exports significant to memory consistency (page 81).

Signature	Description	Exclusive?
create:SAME	Make a new unlocked synchronization object with an empty queue or zero counter and no attached threads.	N/A
has_thread:BOOL	Returns true if there is an attached thread.	No
threads:$LOCK	Returns a lock which blocks until lockable and there is some thread attached; then it is locked. Holding this lock does not prevent the completion of attached threads.	No
no_threads:$LOCK	Returns a lock which blocks until lockable and there are no threads attached; then it is locked. Holding this lock does not prevent the attachment of threads by the holder.	No

Table 1: Operations supported by ATTACH, FUTURE{T}, GATE, and GATE{T}

Signature	Description	Exclusive?
get:T	Return head of queue without removing. Blocks until queue is not empty.	Yes
empty:$LOCK	Returns a lock which blocks until lockable and the queue is empty; then it is locked. Holding this lock does not prevent the holder from making the queue become not empty.	No
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is not empty; then the gate is locked. Holding this lock does not prevent the holder from making the queue become empty.	No

Table 2: Operations supported by FUTURE{T} and GATE{T}

Signature	Description	Exclusive?
size:INT	Returns number of elements in queue.	No
set(T)	Replace head of queue with argument, or insert into queue if empty.	Yes
enqueue(T)	Insert argument at tail of queue.	Yes
dequeue:T	Block until queue is not empty, then remove and return head of queue.	Yes

Table 3: Operations supported only by GATE{T}

Signature	Description	Exclusive?
size:INT	Returns counter.	No
get	Blocks until counter is nonzero.	Yes
set	If counter is zero, set to one.	Yes
enqueue	Increment counter.	Yes
dequeue	Block until counter nonzero, then decrement.	Yes
empty:$LOCK	Returns a lock which blocks until lockable and the counter is zero; then it is locked. Holding this lock does not prevent the holder from making the counter become nonzero.	No
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's counter is nonzero; then the gate is locked. Holding this lock does not prevent the holder from making the counter become zero.	No

Table 4: Operations supported only by GATE

3.3.5a Attach examples

Using a *future*. The statement 'f :- compute' creates a new thread to do some computation; the current thread continues to execute. It blocks at 'f.get' if the result is not yet available.

```
-- Create a future of FLT
f: := #FUTURE{FLT};
f :- compute;
...
result := f.get;
```

Obtaining the first result from several competing searches. Unlike a future, a gate may enqueue multiple values. When one of the threads succeeds, its result is enqueued in 'g'. If the results of the other two threads are not needed, additional code would be needed to prematurely halt the other threads.

```
g :- search(strategy1);
g :- search(strategy2);
g :- search(strategy3);
...
result := g.dequeue;
```

3.3.6 sync *statement*

Example:

```
sync
```

Syntax:

sync_statement ⇒ sync

The sync statement allows barrier synchronization between threads attached to the same synchronization object. A thread executing a sync blocks until all threads attached to the same object are also blocking on sync (or have terminated).

3.3.6a Sync example

This code applies 'phase1' and 'phase2' to each element of an array, waiting for all 'phase1' before beginning 'phase2':

```
parloop e::= a.elt! do e.phase1 end;
parloop e::= a.elt! do e.phase2 end
```

This code does the same thing without iterating over the elements for each phase. A single thread is forked for each element. Each thread executes 'phase1', the sync, and 'phase2'. The thread executing the par waits for all threads to terminate before proceeding.

```
parloop e::= a.elt! do
    e.phase1;
    sync;
    e.phase2
end
```

Because local variables declared in the parloop become unique to each thread, the explicit sync can be useful to allow convenient passing of state from one phase to another through the thread's local variables, instead of using an intermediate array with one element for each thread.

3.3.7 *Memory consistency*

Threads may communicate by writing and then reading variables or attributes of objects. All assignments are atomic (the result of a read is guaranteed to be the value of some previous write); assignments to variables of immutable type atomically modify all attributes. Writes are always observed by the thread itself. Writes are not guaranteed to be observed by other threads until an *export* is executed by the writer and a subsequent *import* is executed by the reader, even if the writes were previously observed by the reading thread. Exports and imports may be written explicitly (page 82) and are also implicitly associated with certain operations:

An import occurs:	An export occurs:
In a newly created thread	In parent thread when a child thread is forked
On exiting a par statement (children have terminated)	By a thread on termination
On entering one of the branches of a lock statement	On entering an unlock, or exiting a lock
On exiting exclusive operations (page 78)	On entering exclusive operations
On completion of a sync statement	On initiation of a sync statement

This model has the property that it guarantees sequential consistency to programs without data races.

3.3.7a Memory consistency examples

This incorrect code may loop forever waiting for flag, print 'i is 1', or print 'i is 0'. The code fails because it is trying to use flag to signal completion of 'i:=1', but there is no appropriate synchronization occurring between the forked thread and the thread executing the par body. Even though the forked thread terminates, the modification of 'flag' may not be observed because there is no import in the body thread. Even if the modification to flag is observed, there is no guarantee that a modification to 'i' will be observed before this, if at all.

```
-- These variables are shared
i:INT;
flag:BOOL;
par
    fork
        i := 1;
        flag := true;
    end;
    -- Attempt to loop until change
    -- in 'flag' is observed
    loop until!(flag) end
    #OUT + 'i is' + i + '\n'
end
```

This code will always print 'i is 1' because there is no race condition (unlike the previous example). An export occurs when the forked thread terminates, and an import occurs when par completes. Therefore the change to 'i' must be observed.

```
i:INT; -- This is a shared variable
par
    fork i:=1 end;
end
#OUT + 'i is' + i + '\n'
```

3.3.8 SYS *class*

pSather extends the SYS class (page 62) with the following routines:

Routine	Description
defer	Inform scheduler that this is a good time to preempt this thread.
import	Execute an import operation (page 81).
export	Execute an export operation (page 81).

3.4 Distributed Extension

This section introduces distributed constructs that allow the programmer to extend pSather code with explicit placement information for efficiency on distributed pSather implementations. Explicitly placing objects and threads does not affect the semantics of the original code, but it is also possible to deliberately change the original flow of control (ie. using with-near on page 84).

The memory performance model of pSather has two levels. The basic unit of location in pSather is the *cluster*. The programmer may assume that reading or writing memory on the same cluster is significantly faster than on a remote cluster. A cluster corresponds to an efficient group in the memory hierarchy, and may have more than one processor. For example, on a network of workstations a cluster would correspond to one workstation, although that workstation may have multiple processors sharing a common bus. This model is appropriate for any machine for which local cached access is significantly faster than general access.

At any time a thread has an associated *cluster id* (an INT), its *locus of control*. Until modified explicitly, the locus of thread remains the same throughout the thread's execution. When execution begins, the main routine (page 60) is at cluster zero. The locus of control of a child thread is the same as the locus of its parent at the time of the fork.

3.4.1 *The '@' operator*

Example:

```
start_work @ least_loaded;
```

Syntax:

expression ⇒ *expression* @ *expression*

fork_statement ⇒ fork @ *expression* ; *statement_list* end

parloop_statement ⇒ parloop *statement_list* do @ *expression* ; *statement_list* end

The locus of a thread may be explicitly moved for the duration of the evaluation of a method call. An expression following the '@' must evaluate to an INT, which specifies the cluster id of the locus of control the thread will be at while it evaluates the preceding method. Subexpressions of the left side are evaluated at the current locus of execution and are not relocated. It is a fatal error for a cluster id to be less than zero or greater than or equal to clusters (see page 84). The '@' operator has lower precedence than any other operator (see page 50). When iterator calls are on the left side, each iterator evaluation may be placed differently on successive iterations.

The '@' notation may also be used to explicitly place forked body threads of fork and par-loop statements. Although for these constructs the location expression may appear to be within the body, the location expression is executed before threads are forked and is *not* part of the body.

3.4.2 *Location expressions*

All reference objects have a unique associated cluster id, the object's *location*. When a reference object is created by a thread, its location will be the same as the locus of control when the new expression was executed. A reference object is *near* to a thread if its current location is the same as the thread's locus of control, otherwise it is *far*.

There are several built-in expressions for location:

Expression	Type	Description
here	INT	The cluster id of the locus of control of the thread.
where(*expression*)	INT	The location of the argument. If the argument is void or an immutable type, it returns 'here'.
near(*expression*)	BOOL	true if the argument is on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
far(*expression*)	BOOL	true if the argument is not on the same cluster as the executing thread. If the argument is void or an immutable type, it returns false.
clusters	INT	Number of clusters. Although a constant, may not be available at compile time.
clusters!	INT	Iterator which returns all cluster ids in order, 0 through clusters-1.

3.4.3 with-near *statement*

Example:

```
with able, baker near ... end
```

Syntax:

with_near_statement ⇒
with *ident_or_self_list* near *statement_list* [else *statement_list*] end

ident_or_self_list ⇒ *identifier* | self { , *identifier* | self }

The *with-near statement* asserts that particular reference objects must remain near at runtime. The *ident_or_self_list* may contain local variables, arguments, and self; these are called _near variables_. When the with statement begins execution, the identifiers are checked to ensure that all of them hold either objects that are near or void. If this is true then the statements following near are executed, and it is a fatal error if the identifiers stop holding either near objects or void at any time. It is a fatal error if some identifiers hold neither near objects nor void and there is no else. Otherwise, the statements following the else are executed.

3.4.3a **Locality examples**

This code creates a object and then inserts it into a table, taking care that the insertion code runs at the same cluster as the table.

```
table.insert(#FOO)
        @where(table);
```

To make sure the object is at the same cluster as the table, one could write

```
loc: := where(table);
table.insert(#FOO @ loc) @ loc;
```

or, equivalently:

```
fork @ where(table);
   table.insert(#FOO)
end
```

This code recursively copies only that portion of a binary tree which is near. Notice that 'near' returns false if its argument is void.

```
near_copy:NODE is
   if near(self) then return
      #NODE(lchild.near_copy,
             rchild.near_copy)
   else return self
   end
end
```

Index

A

B

C

Index

S

T

U