Sather 1.1

August 7, 1995

DRAFT

David Stoutamire
International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, California 94704

Stephen Omohundro
NEC Research Institute, Inc.
4 Independence Way,
Princeton, NJ 08540

ABOUT SATHER

THE SATHER 1.1 SPECIFICATION

About Sather

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.

The Name

Sather was developed at the International Computer Science Institute, a research institute affliated with the computer science department of 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. Yet, at 307 feet it houses 50 tons of human, dinosaur and other animal bones mostly collected from the La Brea Tar Pits. Unseen by most visitors, the collection

covers six floors of the 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.

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

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 disasterously 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.

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 45). With garbage collection there is no need for destructors; however, explicit finalization is available when desired (page 60).

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 42).

In Sather, the redefinition of operators is orthogonal to the rest of the language. There is "syntactic sugar" (page 46) 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)'.

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.

Iterators

Early versions of Sather used a conventional 'until...loop...end' statement much like other languages. This made Sather suseptible 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 iters, whereas a traditional approach requires a different yet intimately coupled class for each kind of iteration over the major class. Cursor objects can also prevent optimizations on inner loops. 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 iters 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.

Bound Routines

Sather provides higher-order functions through *bound routines*, which are similar to closures and function pointers in other languages. These allow binding some or all arguments to arbitrary routines but deferring 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, bound routine objects follow static typing and behave with contravariant conformance.

Value and Reference Objects

Sather distinguishes between reference objects and value objects. Experienced C programmers immediately understand the difference when told about the internal representation the ICSI compiler uses: value 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 value objects never appear to be. Many of the built-in types (integers, characters, floating point) are value classes. There are a handful of other differences between reference and value types; for example, reference objects must be explicitly allocated, but value objects 'just are'.

Value objects never change once they are created. When one wishes to only modify a value object, one is compelled to create a whole new object with the modification.

Value types can have several performance advantages over reference types. Value 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 a value object is the overhead that some C compilers introduce in passing the entire object on the stack. This problem is worse in value classes with many attributes. Unfortunately the efficiency of a value 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.

Value 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 value class 'INT'. The standard string class 'STR' is also a reference type but behaves with value semantics. Explicitly declaring value classes allows the compiler to enforce value semantics and provides a hint for good code generation. Common value classes are defined in the standard libraries; defining a new value class is unusual.

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.

USING SATHER

At the time of this writing, the only compiler implementing the 1.1 language specification is available from ICSI. **DRAFT NOTE: this is not yet true; only the 1.0 is officially supported by ICSI at this time.** It is freely available, includes source for class libraries and the compiler, and compiles into ANSI C. It has been ported to a wide range of UNIX and PC operating systems.

Obtaining the compiler

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

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

These sites also mirror the Sather distribution:

ftp.sterling.com:	/programming/languages/sather
ftp.uni-muenster.de:	/pub/languages/sather
maekong.ohm.york.ac.uk:	/pub/csp
ftp.th-darmstadt.de:	/pub/programming/languages/sather
ftp.nuie.nagoya-u.ac.jp:	/languages/sather

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.

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.

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, due to be released September 1995. That compiler is 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

Acknowledgements

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, Matt Kennel, 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.

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. Also 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, C. Szyperski. "Sather Iters: Object Oriented Iteration Abstraction," TR-93-045, International Computer Science Institute, August 1993. Also available at the Sather WWW page. **DRAFT NOTE: will be replaced by TOPLAS reference.**

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

The Sather 1.1 Specification

INTRODUCTION

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.

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.

OBJECTS, 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[1] kinds of objects in Sather: *value* (*e.g.* integers), *reference* (*e.g.* strings), *bound* (Sather's version of closures), and *external* (used in represent entities in other languages). There are four corresponding concrete types: *value*, *reference*, *bound*, and *external* types. There are also abstract types, which represent sets of concrete types. Value, reference, external, and abstract types are defined textually by *value*, *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 and bound types can be supertypes (see pages 20 and 55); bound types can only be supertypes of other bound 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*).

1. The pSather extension adds a fifth: *spread* (page 80).

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. Abstract classes specify a set of *abstract signatures*, an interface without an implementation. Classes specify a set of *concrete signatures*; a type with an implemention. Execution of a Sather program begins with a routine named 'main' in a specified class (page 58), usually 'MAIN'.

Abstract class definitions

Example:

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

Syntax:

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

parameter_declaration ⇒ *uppercase_identifier* [< *type_specifier*]

subtyping_clause ⇒ < *type_specifier_list*

supertyping_clause ⇒ > *type_specifier_list*

type_specifier_list ⇒ *type_specifier* { , *type_specifier*}

abstract_signature ⇒ (*identifier* | *iter_name*)
[(*abstract_argument* { , *abstract_argument* })] [: *type_specifier*]

abstract_argument ⇒ *identifier_list* : *type_specifier* [!]

Abstract class definitions specify interfaces without implementations. Abstract class names must be entirely uppercase and must begin with a dollar sign '$' (page 56); 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 type definitions may have the same name only if they specify a different number of type parameters.

Abstract class definitions may be *parameterized* by one or more type parameters within enclosing braces; in the example, the type parameter is 'T'. Parameter names are local to the abstract class definition and they shadow non-parameterized types with the same name. Parameter names must be all uppercase, and they may be used within the abstract type definition as type specifiers. Whenever a parameterized type is referred to, its parameters are specified by type specifiers. The abstract class definition behaves like a non-parameterized version whose body is a textual copy of the original definition in which each parameter occurrence is replaced by its specified type. Parameterization may be thought of as a structured macro facility.

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 59). 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.

If a parameter declaration is followed by a type constraint clause ('<' followed by a type specifier), then the parameter can only be replaced by subtypes of the constraining type. If a type constraint is not explicitly specified (as in the example), then '< $OB' is taken as the constraint. An abstract type definition must satisfy all of the typing rules when its parameters are replaced by any subtype of their constraining types; this allows type-safe independent compilation.

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. 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.

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
```

Signatures

Operations are performed on objects by calling *methods* on them, which are either *routines* (page 29) or *iterators* (page 30). 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.

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. and each argument type in *f* is either equal to the corresponding argument type in *g* or one of the two types is either abstract or bound.

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.

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 explictly 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.

Conflict example

In this illegal abstract class, A and B do not conflict because their arguments are not abstract and are not the same type. However, because the argument of C is abstract, it conflicts with both A and B. INT and BOOL happen to be subtypes of $STR, but that does not affect whether C conflicts with A and B. D does not conflict with A, B or C because it has a different number of parameters.

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

Classes

Examples:

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

Syntax:

class ⇒ [value | partial | external *identifier*] class *uppercase_identifier*
[{ *parameter_declaration* { , *parameter_declaration* } }]
[*subtyping_clause*]
is [*class_element*] { ; [*class_element*] } end

There are three types that have implementations: reference, value, and external types. They are defined by classes beginning with 'class', 'value class', and 'external *language* class', respectively. Reference types may be aliased and usually are allocated on a dynamic heap. Value types (such as complex numbers) are immutable, 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 61. 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 into such a class are allowed, and no variables may be declared of such a type.

Class names must be entirely uppercase (page 56). 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.

Reference, value and partial classes may optionally be *parameterized* by one or more *type parameters* within enclosing braces. Type 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 body as type specifiers. Whenever a parameterized type is referred to, its parameters are specified by type specifiers. The class behaves like a non-parameterized version whose body is a textual copy of the original class in which each parameter occurrence is replaced by its specified type. Parameterization may be thought of as a structured macro facility.

If a parameter declaration is followed by a type constraint clause ('<' followed by a type specifier), then the parameter can 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 specifier may not refer to 'SAME'. The body of a parameterized class must be type-correct when the parameters are replaced by any subtype of their constraining types.

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 59).

Class 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)'. Parameterized typing like this can be resolved at compile time, leading to efficient code.

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

The complex number class from the standard library is a good example of a value class. Here we also see that complex numbers can be compared with other complex numbers ('$IS_EQ{CPX}') and can be converted to strings ('$STR'). The important thing about value classes is that they are immutable; see page 13. Because of this, value classes can be more efficient than regular classes. 'CPX' is a value class because it is small and behaves with a mathematical semantics.

```
value class CPX
        < $IS_EQ{CPX}, $STR
is
    ...
end
```

Type specifiers

Examples:

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

Syntax:

type_specifier ⇒ (*uppercase_identifier* | *abstract_class_name*) [{ *type_specifier_list* }]
| ROUT
[{ [*routine_mode*] *type_specifier* [, [*routine_mode*] *type_specifier*] }]
[: *type_specifier*]
| SAME

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' optionally followed by a list of argument types in braces, optionally followed by a colon and return type (*e.g.* 'ROUT{A,B}:C'). This is used for bound routine types (page 55).
- The special type specifier 'SAME,' which denotes the type of the class in which it appears. It may not appear in abstract type definitions.

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 61).

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*])
identifier_list ⇒ *identifier* { , *identifier* }

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 38),
2. a void or void test expression (page 43),
3. an and or or expression (page 46), each of whose components is a constant expression,
4. an array creation expression (page 45), 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.

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'.

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 26). If no initializing expression is provided, the shared is initialized to the value 'void' (page 43).

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'.

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 31) from the type to AREF for reference types or to AVAL for value types (page 59). Bound and reference objects must be explicitly allocated as described on pages 44 and 55. Variables have the value 'void' until an object is assigned to them (page 43). There must be no cycle of value 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 value types; this difference arises because value types are immutable (a new object is created instead of modifying in place). 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 value 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'.

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.

```
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
```

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

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 value 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 value type
```

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]

routine_argument ⇒ *routine_mode identifier_list* : *type_specifier*

routine_mode ⇒ [(out | inout)]

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 43). If a routine has a return value, it is declared by a colon and a specifier for the return type.

The 'pre' and 'post' clauses specify optional pre- and post-conditions, and are discussed further on page 53. The body of a routine definition is a list of statements (page 33).

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

iter_argument ⇒ *iter_mode identifier_list* : *type_specifier*

iter_mode ⇒ [(out | inout | once)]

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 48). Iterator arguments that are not marked 'once' are called *hot* and cause re-evaluation of that argument at each iteration (see also page 43).

The description of routine arguments and pre and post constructs also applies to iterator definitions. Iters may contain yield (page 35) and quit (page 49) statements but may not contain return statements (page 35). The semantics of iterator calls is described in the section on loop statements (page 48). 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 48.

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 bound 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 62) and C (page 63) classes may not be included. There mustn't be include paths from reference types to AVAL or from value types to AREF (page 59). 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 type of the class containing the clause (*i.e.*, as early 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.

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.

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.

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
```

It is possible to have objects of the 'AREF' class used above; it can stand alone. Sometimes one wishes to write partial classes which are never meant to be instantiated. In such classes stubs can be used to define methods meant to be filled in by an including class. It isn't possible to make an object of or call into the partial class 'MIXIN'.

```
partial class MIXIN is
    bar is
        -- Makes use of 'foo'
    end;

    -- 'foo' has no implementation here
    stub foo;
end
```

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.

Declaration statements

Example:

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

Syntax:

declaration_statement ⇒ *identifier_list* : *type_specifier*

Declaration statements are used to declare the type of one or more local variables. Local variables may also be declared in assignment statements (page 34). 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 43) when the containing method is called; they are not re-initialized when the declaration is encountered in the flow of control.

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*

Assignment statements 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 value and bound 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 33 apply here as well. If a type is explicitly specified, the construct is syntactic sugar for a declaration statement followed by an assignment statement.

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.

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 52) statement.

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. If one of these calls returns true, then the corresponding statement list is executed and control passes 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.

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.

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.

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
```

Boolean literal expressions

Examples:

```
true
false
```

Syntax:

bool_literal_expression ⇒ true | false

BOOL objects represent boolean values (page 59). The two possible values are represented by the *boolean literal expressions*: 'true' and 'false'.

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 59). *Character literal expressions* 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.

String literal expressions

Examples:

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

Syntax:

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

STR objects represent strings (page 59). *String literal expressions* 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.

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 59). 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 59).

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 59). 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.

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*
| *bound_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.

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.

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.

Method call expressions

Examples:

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

Syntax:

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

expression_list ⇒ *expression* { , *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 pre-

cedes 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 43.

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.
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.

Sather supports routine and iterator *overloading*. In addition to the name, the number and types of arguments in a call and whether a return value is used all contribute to the selection of the method. The *expression_list* portion of a call must supply an expression corresponding to each declared argument of the method. There must be a method with the specified name such that the type of each expression is a subtype of the declared type of the corresponding argument and it must be unique. 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.

Sather also supports *dynamic dispatch* on the type of self when the expression on which the call is made has an abstract declared return type. The method matching the call from the runtime type of the returned object is actually executed. Because of the fundamental subtyping rule (page 19), if the abstract type specifies a conforming method, so will the type of the returned object.

Direct calls of a type's routines or iters may be made using the double colon '::' syntax. The *type_specifier* must specify a reference, value, or external class. In such calls self has the void default value described on page 43.

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.

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
```

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 38) and for other value types it is determined by recursively setting each attribute and array element to void. The built-in value 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 59).

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 45). 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 41).

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 43).

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 43). If there is an include path from the type in

which the new appears to AREF (page 59), then new must be provided with a non-negative INT argument which specifies the number of array elements in the returned object.

Creation expressions

Examples:

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

Syntax:

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

Value and reference object *creation expressions* are a convenient shorthand used for creating new objects and initializing their attributes. A creation expression is syntactic sugar for a call on a routine named 'create' with the specified arguments. 'self' is given the default void value described on page 43 in this call. The type defining the 'create' routine may be explicitly specified as a reference or value 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 a value 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 34), as a method argument in which overloading resolution would otherwise be ambiguous, or as the left argument of the dot '.' operator (page 41).

Array creation expressions

Examples:

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

Syntax:

array_expression ⇒ | *expression_list* |

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 34), as a method argument in which the overloading resolution is ambiguous, or as the left argument of the dot '.' operator (page 41). 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.

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.

or *expressions*

Example:

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

Syntax:

or_expression ⇒ *expression* or *expression*

or expressions 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.

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 *syntactic sugar* for corresponding routine calls. Each of these transformations is applied after the component expressions have themselves been transformed. Note that 'and' and 'or' are not listed as

syntactic sugar for operations in 'BOOL'; this allows short-circuiting the evaluation of sub-expressions.

Sugar form	*Translation*	*Sugar form*	*Translation*
expr1 + *expr2*	*expr1*.plus(*expr2*)	*expr1* /= *expr2*	*expr1*.is_neq(*expr2*)
expr1 - *expr2*	*expr1*.minus(*expr2*)	*expr1* > *expr2*	*expr1*.is_gt(*expr2*)
expr1 * *expr2*	*expr1*.times(*expr2*)	*expr1* >= *expr2*	*expr1*.is_geq(*expr2*)
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*	*expr1*.is_leq(*expr2*)	(*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

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
```

LOOPS AND ITERATORS

Iterator definitions were described on page 30. 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.

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 41) 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 41.

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.

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 51). Yield causes assignment to out and inout arguments in the caller (page 41).

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 41). No statements may follow a quit statement in a statement list.

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.

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.

break! *expressions*

Example:

```
break!
```

Syntax:

break!_expression ⇒ break!

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

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
```

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.

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 52) 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 54) 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.

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.

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 tak-

en in the most tightly enclosing protect statement. The return type is the type specified in the corresponding when clause (page 51). In an else clause the return type is '$OB'.

SAFETY FEATURES

Methods definitions may include optional pre- and post-conditions (see page 29). Together with 'assert' (page 53), these features allow the earnest programmer to annotate the intention of code. Implementations of Sather must provide facilities for turning on or off the runtime checking these safety features imply.

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.

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 54) to refer to the value returned by the routine and 'initial' expressions (page 54) 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 58).

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.

initial *expressions*

Example:

```
initial(a)
```

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.

result *expressions*

Example:

```
result
```

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.

BOUND ROUTINES

Bound routine creation expressions

Examples:

```
#ROUT(2.plus(_))
#ROUT(_:INT.plus(2))
```

Syntax:

bound_create_expression ⇒ #ROUT
([*type_specifier* :: | *bound_argument* .] *identifier*
[(*bound_argument* { , *bound_argument* })] [: *type_specifier*])

bound_argument ⇒ *routine_mode expression* | _ [: *type_specifier*]

Bound routines are similar to the 'function pointer' and 'closure' constructs of other languages. They bind a reference to a routine together with zero or more argument values (possibly including 'self').

The outer part of the expression is '#ROUT(...)'. This surrounds an ordinary routine call in which any of the arguments or self may be replaced by the underscore character '_'. Such unspecified arguments are *unbound*. Unbound arguments are specified when the bound routine is eventually called. 'out' and 'inout' arguments must be left unbound. Optional ':*type_specifier*' clauses are used to specify the types of underscore arguments or the return type and may be necessary to disambiguate overloaded routines or iters. If self is specified by an underscore without type information, the type is taken to be SAME.

The expressions in this construct are evaluated from left to right and the resulting values are stored as part of the bound routine. Bound creation expressions return bound types. As previously described on page 25, the type specifiers for these types have the form:

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

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

Each bound routine defines a routine named 'call'. These have argument and return value types that correspond to the bound type descriptor. An invocation of this feature behaves like a call on the original routine with the arguments specified by a combination of the bound values and those provided to 'call'. The arguments to call match the underscores positionally from left to right (*e.g.* 'i::=#ROUT(2.plus(_)).call(3)' is equivalent to 'i::=2.plus(3)').

Bound types implicitly introduce edges into the type graph. There is an edge from each bound type g to all bound 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, or inout),
4. *contravariant conformance*:
   for any in 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.

Bound Routine Example

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

```
r ::= #ROUT(2.0.times(_));

loop
    a.set!(r.call(a.elt!))
end
```

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 39) or character literal (page 38) and all following text until the end of the line.

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 such as pSather (page 66). The keywords are:

keyword ⇒ and | any | assert | attr | break! | case | class | clusters | clusters! | cohort | const | else | elsif | end | exception | external | false | far | fork | guard | if | 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 | spread | sync | then | true | type | typecase | unlock | until! | value | void | when | while! | with | yield

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

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

In addition to the keywords 'ROUT', 'ITER', and 'SAME', there are several reserved names which may not be used to name user classes. Some of these are the names of built-in library classes known to the compiler, others are used in special situations as described on page 59, or by pSather (page 66).

special_classnames ⇒ $OB | ARRAY | AREF | AVAL | BOOL | CHAR | EXT_OB | FLT | FLTD | FLTX | FLTDX | FLTI | GATE | INT | INTI | $LOCK | MUTEX | $REHASH | SAME | STR | SYS

There are certain names in the feature namespace which are the translations of syntactic sugar expressions:

sugar_featurenames ⇒ aget | aset | div | is_eq | is_geq | is_gt | is_leq | is_lt | is_neq | minus | mod | negate | not | plus | pow | times

and there are feature names which have a special effect when they are defined in a class:

special_featurenames ⇒ create | invariant | main

SPECIAL FEATURES

This section describes features of classes that have special properties.

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.

main

A distinguished non-parameterized value 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.

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', 'aelts!', 'aset_elts!', and 'ainds!'. Array indices start at zero. 'AVAL{T}' is the value 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 45).
- 'TUP' names a set of parameterized value 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 37:

Type	Initial value	Description
BOOL	false	Value objects which represent boolean values.
CHAR	'\0'	Value 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	Value 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	Value objects which represent single precision floating point values as defined by the IEEE-754-1985 standard.
FLTD	0.0d	Value 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
type(ob:$OB):INT	Returns the concrete type of an object encoded as an 'INT'.
str_for_type(i:INT):STR	Returns a string representation associated the the integer. Useful for debugging in combination with 'type' above.
destroy(ob:$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).
id(ob:$OB):INT	Returns an integer associated with a particular object; it is unique for each existing object throughout its lifetime, although integers may be recycled after objects are garbage collected. It is a fatal error to call 'id' on value or external types.
ob_eq(o1,o2:$OB):BOOL	Tests two objects for equality. If the arguments are of different type, it returns 'false'. If both objects are value types, 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 'ob_eq' with void reference arguments or with external types.

- $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 at least 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 may occur more than once 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 when conventional coding will not suffice.

Extensions

INTRODUCTION

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.

INTERFACES TO OTHER LANGUAGES

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:

- C: interface to ANSI C, X3.159-1989
- FORTRAN: interface to ANSI Fortran 90, X3.198-1992

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. Routines with a body in an external class specify the interface for external code to call

Sather code. All routines and signatures in external classes must not conflict (page 22), and the corresponding external object file must provide a function that conforms according to the rules of the language interface. For C and Fortran, external code may refer to an external routine with a body by concatenating the class name, an underscore, and the routine name (*e.g.* 'EXT_CLASS_sather_rout'). It is an error if the external language namespace implementation does not permit the resulting name (*e.g.* due to length). The implementations or environments of other languages may impose other unavoidable constraints.

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*	*Sather array type*	*Fortran array type*
F_REAL	real	F_REAL_ARR	real x()
F_DOUBLE	double precision	F_DOUBLE_ARR	double precision x()
F_INTEGER	integer	F_INTEGER_ARR	integer j()
F_COMPLEX	complex	F_COMPLEX_ARR	complex z()
F_DOUBLE_COMPLEX	double precision complex	F_DOUBLE_COMPLEX_ARR	double precision complex z()
F_LOGICAL	logical	F_LOGICAL_ARR	logical b()
F_CHARACTER	character		

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 signatures defined in a Fortran external class. Routines defined in external Fortran classes that have bodies (are not abstract signatures) may use other types, but if they do, these routines are not visible to Fortran code.Fortran external classes may not contain any class elements other than routines and abstract signatures, and may not be parameterized.

Fortran implementations pass arguments by reference. The scalar types in the first column may only be passed to external routines as 'inout' arguments. The Sather implementation must assure that changes to 'inout' arguments passed to a Fortran routine are not observed until the Fortran routine returns.

Interfacing with ANSI C

An external class which interfaces to ANSI C is designated with the language identifier 'C'. External C classes may not be parameterized. The following C types are built into the extended library:

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 *		

These external types also define appropriate creation routines which may be used for convenient casting between Sather and C types. 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.

'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.

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.

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 be used to specify a C header file or other preprocessing directive that should be emitted at the beginning of any file in which the C declaration appears.

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:=
        "#include <widgets.h>\n";
end;
```

Attributes may be placed in external C classes; they are interpreted as fields of a C struct. Similarly shareds are interpreted as C global variables. Constants 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;
```

External C class attributes, shareds, and constants may only have external C types. The only exception is 'C_name' and 'C_header', which must be constant string literals when present.

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 argu-

ments, 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");
```

IEEE FLOATING-POINT SUPPORT

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 problem 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.

BOUND ITERATORS

In a Sather implementation supporting the bound iterator extension, iterators may be bound in the same way as routines. This is done with a '#ITER' expression, akin to bound routine creation expressions (page 55). Bound iterators are called with '.call!' in the same way that bound routines are called with '.call'. In, out, and inout arguments must be left unbound; only 'once' arguments may be bound. The ICSI implementation does not currently support bound iterators.

The Parallel Extension (pSather)

THREADS

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 69), parloop statement (page 70), or an attach (page 67). 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 58).

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.

Gates

Gates are powerful synchronization primitives which generalize fork/join, mailboxes, semaphores, and barrier synchronization. Gates have the following unnamed attributes:

- A locked status (*unlocked*, or *locked* by a particular thread);
- in the case of 'GATE{T}', a queue of values which must conform to 'T', or
- in the case of the unparameterized class 'GATE', an integer counter;
- a set of *attached* threads. Every pSather thread is attached to exactly one gate. Even the main routine is attached to an unnamed gate; a pSather program terminates when all threads have terminated. Attached threads may be thought of as producers that enqueue their return value (or increment the counter) when they terminate.

The attach *statement*

Example:

```
g :- deferred_computation
```

Syntax:

attach ⇒ *expression* :- *expression*

One way that threads can be created is by executing an *attach*. The left side must be of type 'GATE' or 'GATE{T}'. If the left side is of type 'GATE{T}', the return type of the right side must be a subtype of 'T'. If the left side is of type 'GATE', the right side must not return a value. There must be no iterators in the right side.

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

1. The left side is evaluated.
2. If the gate is locked by another thread, the executing thread is suspended until the gate 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 gate. The new thread receives a unique copy of every local variable; changes to this local by the originating thread are not observed by the new thread. Similarly, if 'out' and 'inout' arguments occur in the right side, changes to local variables will not be observed by the originating thread.The rules for memory consistency apply for other variables such as attributes of objects (see page 76).
5. When execution of the right side completes, the new thread terminates, detaches itself from the gate, and enqueues the return value or increments the counter.

In addition to having threads attached, gates support the operations listed in the following table. Some gate operations are *exclusive*: these lock the gate before proceeding and unlock

Signature	Description	Exclusive?
create:SAME	Make a new unlocked GATE{T} object with an empty queue and no attached threads.	N/A
size:INT	Returns number of elements in queue [GATE: returns counter].	No
has_thread:BOOL	Returns true if there exists a thread attached to the gate.	No
set(T) [GATE::set]	Replace head of queue with argument, or insert into queue if empty. [GATE: If counter is zero, set to one.]	Yes
get:T [GATE::get]	Return head of queue; do not remove from queue. Blocks until queue is not empty. [GATE: Blocks until counter is nonzero.]	Yes
enqueue(T) [GATE::enqueue]	Insert argument at tail of queue. [GATE: increment counter.]	Yes
dequeue:T [GATE::dequeue]	Block until queue is not empty, then remove and return head of queue. [GATE: Block until counter nonzero, then decrement.]	Yes

it when the operation is complete. Only one thread may lock a gate at a time. The exclusive operations also perform imports and exports significant to memory consistency (page 76). Gates support other operations listed on page 74.

Gate examples

Gates can be used to signal their threads about changes in the queue or attached threads. For example, a thread that wants to continue when a gate's queue has a value can use 'get' to wait without looping.

Using a gate as a *future*. The statement 'g :- compute' creates a new thread to do some computation; the current thread continues to execute. It is suspended at 'g.get' only if the result is needed and not available.

```
-- Create a gate with queue of FLTs
g: := #GATE{FLT};
g :- compute;
...
result := g.get;
```

Obtaining the first result from several competing searches. 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;
```

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 by an attach, but may also 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. 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 76).

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. Exceptions in a lock body (page 72) will not be raised outside the body until all associated locks have been released. Because par and fork bodies are executed as separate threads, an unhandled exception in their bodies is a fatal error.

cohort *expression*

Example:

```
cohort
```

Syntax:

expression ⇒ cohort

The thread executing a par statement implicitly creates a GATE object and forks a thread to execute the body. The gate may be accessed by the special expression cohort, which must be syntactically enclosed in a par statement.

The newly created thread as well as all threads created by fork statements syntactically in the par body are attached to this same gate. The thread executing a par statement blocks until there are no threads attached to the cohort gate. This ensures that all threads created by a fork have completed before execution continues past the par.

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
```

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

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
```

Sync

Example:

```
sync
```

Syntax:

sync_statement ⇒ sync

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

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.

LOCKS

Locks control the blocking and unblocking of threads. 'GATE', 'GATE{T}' and 'MUTEX' are special synchronization objects which provide a mutual exclusion lock. 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 conditions.

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 *lock_when* { *lock_when* } [else *statement_list*] end

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

Locks may be safely acquired with the *lock statement*. The type of all expressions following 'when' must be subtypes of $LOCK (page 74). 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*. Expressions following a 'when' are called *locking conditions*. The statements folloing the 'else' are called the *else branch*.

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

1. All 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 pSather 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 77).

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 not occur in a lock statement.

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.

$LOCK *Classes*

All synchronization objects subtype from $LOCK. The following primitive $LOCK classes are built-in:

- GATE{T} and GATE (page 67).
- MUTEX: a simple mutual exclusion lock. Two threads may not simultaneously lock a MUTEX. MUTEX is a subset of the functionality of GATE, and may require less overhead when a GATE isn't needed.

In addition to these 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.

- GATE and GATE{T} define the following methods which return $LOCK objects. These gate operations are not exclusive. Other gate operations are listed on page 68.

Operation	Description
empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is empty [for unparameterized GATE: counter zero]; then the gate is locked. Holding this lock does not prevent the holder from making the queue become not empty [counter become nonzero].
not_empty:$LOCK	Returns a lock which blocks until the gate is lockable and the gate's queue is not empty [GATE: counter nonzero]; then the gate is locked. Holding this lock does not prevent the holder from making the queue become empty [counter zero].
threads:$LOCK	Returns a lock which blocks until the gate is lockable and there is some thread attached to the gate; then the gate is locked. Holding this lock does not prevent the completion of attached threads.
no_threads:$LOCK	Returns a lock which blocks until the gate is lockable and there are no threads attached to the gate; then the gate is locked. Holding this lock does not prevent the attachment of threads by the holder.

- RW_LOCK is used to manage reader-writer synchronization, and defines two methods 'reader' and 'writer'. These return $LOCK objects in the same manner as the GATE operations above. 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 simulataneously hold 'rw.reader'. There is no guaranteed preference between readers and writers. Attempting to obtain a writer lock while holding the corresponding reader lock causes deadlock.

Locking examples

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
```

This code computes the maximum value in an array by using a thread to compute the maximum in each subrange of 1024 elements. After each subrange is computed, the global maximum is computed over all subranges.

```
-- Outside the par body so this is shared
global_max:FLT:=a[0];
parloop
   -- Step by 1024.
   -- Threads work on 1024 elements
   i::=0.upto!(a.size-1, 1024);
do
   -- 'm' is local to each body thread
   m:FLT:=a[i];
   loop
      -- 1024 elements starting at 'i'
      m:=m.max(a.elt!(i,1024))
   end
   -- Obtain mutual exclusion
   lock when cohort then
      global_max:=global_max.max(m)
   end
end
```

MEMORY CONSISTENCY MODEL

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 value objects 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. Exports and imports are 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 GATE operations (listed on page 68)	On entering exclusive GATE 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.

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 occuring 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'
```

The SYS *Class*

pSather extends the SYS class (page 60) 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 76).
export	Execute an export operation (page 76).

THE CLUSTER MODEL

This section introduces distributed constructs that allow the programmer to extend pSather code with explicit placement information for efficiency on distributed pSather implementations.

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*. Every thread also has a *fixed* or *unfixed* status. Unless modified explicitly, the locus of a fixed thread remains the same throughout the thread's execution. The locus of control of an unfixed thread may change at any time. When execution begins, the main routine (page 58) is unfixed[2]. The unfixed status or fixed locus of control of a child thread is the same as the status or locus of its parent at the time of the fork.

The '@' operator

Example:

```
start_work @ least_loaded;
```

Syntax:

expression ⇒ *expression* @ (*expression* | any)

fork_statement ⇒ fork @ (*expression* | any) ; *statement_list* end

parloop_statement ⇒
parloop *statement_list* do @ (*expression* | any) ; *statement_list* end

The locus of a thread may be explicitly fixed or unfixed for the duration of the evaluation of an expression. An expression following the '@' must evaluate to an INT, which specifies the cluster id of the locus of control the thread will be fixed at while it evaluates the preceeding expression. It is a fatal error for a cluster id to be less than zero or greater than or equal to clusters (see page 79). If any is given instead of a cluster id, the thread will be unfixed while it evaluates the expression. The '@' operator has lower precedence than any other operator (see page 47).

The '@' notation may also be used to explicitly fix or unfix body threads of fork and parloop statements. Although for these constructs the location expression may appear to be within the body, it is really part of the header. The location expression is executed before any threads are forked and is *not* part of the body.

Location expressions

All reference objects have a unique associated cluster id, the object's *location*, as well as a fixed or unfixed status. When a reference object is created by a fixed thread, its location will be the same as the locus of control when the new expression was executed. If the creating thread was unfixed, the object will be unfixed and its location may change at any time. 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*.

2. The fixed or unfixed status of the main routine and other characteristics of unfixed threads and objects may be changed by compiler options. It is a legitimate pSather implementation to have 'unfixed' threads and objects behave as though they were fixed at the point of creation.

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, a value type, or a spread type (page 80), it returns 'here'.
near(*expression*)	BOOL	true if the argument is on the same cluster as the executing thread. If the argument is void, a value type, or a spread 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, a value type, or a spread 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.

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 run-time. 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. Unfixed objects will not change location while they are held by near variables. 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.

Locality examples

This code creates an unfixed 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 @ any)
        @where(table);
```

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

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

or, equivalently:

```
fork @ table(where);
   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
```

Spread Objects

Example:

```
spread class BIG_ARRAY is ... end
```

Syntax:

class ⇒
spread class *uppercase_identifier*
[{ parameter_declaration {, parameter_declaration} }] [subtyping_clause]
is *[class_element] { ; [class_element] }* end

A *spread class* replicates object attributes and array elements across all clusters. An object of a spread class has a distinct instance of each attribute and array element on each cluster. Attribute and array accesses read or write only the instance on the cluster of the locus of execution. The instance on a particular cluster can be accessed with the idiom *'spread_var@location'*.

Like reference array classes, objects of spread classes have an array portion if there is an include path to AREF (page 59). There must not be an include path from spread types to AVAL. The 'new' expression in a spread array class is used just as in a reference class; it has

a single integer argument if the class has an array portion (page 44). An array of the given size is created on every cluster.

Spread examples

Objects of this class have a floating point attribute that exists on every cluster. The 'create' method initializes the value on every cluster. The 'sum' method adds these up but uses no parallelism.

```
spread class FOO_S is
    attr x_s:FLT;
    create(init:FLT):SAME is
        res ::= new;
        parloop c ::= clusters! do
            x_s @ c := init
        end
    end;
    sum:FLT is
        res ::= 0.0;
        loop res := res + x_s @ clusters! end;
        return res
    end
end
```

On a machine with one processor per cluster, spread might be used to implement a spread vector with subranges distributed across the clusters. Here an add on the distributed vector is implemented by adding all of the subranges in parallel, one on each cluster.

```
spread class SPREAD_VEC is
    attr subrange:VEC;
    -- There is one vector on each cluster;
    -- this is a pointer to it.
    ...
    plus(b:SAME):SAME is
        res: := new;
        -- Idiom recognized by compiler;
        -- implemented as a broadcast
        parloop do @ clusters!;
            res.subrange := subrange + b.subrange
        end;
        return res
    end
    ...
end
```

The previous implementation would not perform well on machines with more than one processor per cluster; a more portable class could be written this way. Here each cluster can have more than one subrange, so more than one processor could work at once.

The extended library provides classes to manipulate such distributed data structures by code reuse. **DRAFT: of course, not yet.**

```
spread class DIST_VEC is
    attr chunks_s:ARRAY{VEC};
    -- at least one chunk per processor
    ...
    plus(b:SAME):SAME is
        res: := new;
        res.chunks_s := #(size);
        -- Execute on each cluster
        parloop do @ clusters!;
            -- Fork for each chunk
            parloop i; := chunks_s.ind!; do
                res.chunks_s[i] :=
                    chunks_s[i] + b.chunks_s[i]
            end
        end;
        return res
    end
    ...
end
```