Sather - A Language Tutorial

Ben Gomes, David Stoutamire, Boris Vaysman, Holger Klawitter

Edited by

Norbert Nemec

Sather - A Language Tutorial

by Ben Gomes, David Stoutamire, Boris Vaysman, Holger Klawitter Norbert Nemec

Table of Contents

Chapter 1. Introduction

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

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

Sather and the GNU Sather compiler are licensed under a GPL/LGPL-combination aimed at encouraging contribution to the public library without precluding the use of Sather for proprietary projects.

This chapter will provide a basic introduction for new users, pointing to sources of information about the language and the compiler. It also contains a summary of Sather features - for those familiar with another object-oriented language, this section provides an overview of the key features of Sather.

1.1. Acknowledgements

This text has its roots in the Sather 1.1 specification, the Eclectic tutorial and Holger's iterator tutorial. This document also contains several organizational ideas and some text from S. Omohundro's originally planned Sather book.

This text has benefitted from corrections, comments and suggestions from several people including Cary D. Renzema, Jerome Feldman, Claudio Fleiner and Arno Jacobsen. Particular thanks to Cary, Arno and Feldman for detailed error reports. Arno also made several suggestions regarding terminology and examples that have been incorporated.

In the course of the transition from ICSI Sather to GNU Sather in Summer 1999, the text has been converted to SGML and heavily cleaned up by Norbert "Nobbi" Nemec.

1.2. How to read this Document

This document is meant to be a complete description of Sather, and is intended as an introduction to the language for a person with some programming background. It is more expository in nature than the specification and contains sections that motivate particular aspects of the language, such as the overloading rules. In addition, it deals with some more abstract design issues that arise when programming in Sather (such as the effect of the contra-variant subtyping rule).

1.3. Sources of Information

This section briefly introduces some concepts important to Sather that the reader may not have been exposed to in C++[1]. It isn't meant as a complete language tutorial. More information of a tutorial nature is available from the WWW page:

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

1.4. Obtaining the Compiler

The GNU Sather compiler can be obtained by anonymous ftp at

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

There is also a library of contributed Sather code to be found at this page.

There is a newsgroup devoted to Sather:

`comp.lang.sather`

The newsgroup is mirrored in both ways by the mailing list

`<info-sather@gnu.org>`

It is meant for any public discussion about the using and developing Sather. To subscribe, send email to:

`<info-sather-request@gnu.org>`

with subject "subscribe".

1.4.1. 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/mailinglist 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

`<bug-sather@gnu.org>`

This is also where you want to send bug reports.

1.5. Summary of Features

This section provides a summary of Sather's features, with particular attention to features that are not found in the most common object oriented languages.

1.5.1. 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.

1.5.2. Garbage Collection and Checking

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

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

1.5.3. 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 107). With garbage collection there is no need for destructors; however, explicit finalization is available when desired (page 143).

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. (See unnamedlink for details)

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

1.5.4. 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,"[2]. It is available at the Sather WWW page.

1.5.5. Iterators

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

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

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

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

The rationale of the Sather iterator construct and comparisons with related constructs in other languages can be found in the ICSI technical report TR 93-045: "Sather Iters: Object-Oriented Iteration

Abstraction,"[3]. It is available at the Sather WWW page.

1.5.6. Closures

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

1.5.7. Immutable and Reference Objects

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

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

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

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

1.5.8. IEEE Floating-Point

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

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

Correct IEEE support from various platforms was the single worst porting 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.

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

1.5.9. 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[4]. 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.

Data placement

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

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

1.6. 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[5]. 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 contributions by Jerome Feldman, Chu-Cheow Lim, Franco Mazzanti and Stephan Murer. The first pSather compiler [6] 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[7]. The 1.0 compiler was a completely fresh effort by Stephen Omohundro, David Stoutamire and Robert Greisemer. 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 Jerome Feldman, Stephan Murer and David Stoutamire.

This document describes Sather 1.1, released the summer of 1996. The compiler was originally designed and implemented by S. Omohundro, D. Stoutamire and (later) Robert Griesemer. Boris Vaysman is the current Sather czar and feature implementor. Claudio Fleiner implemented most of the common optimizations, a lot of debugging support, the pSather runtime and back-end support for pSather. Michael Philippsen implmented the front/middle support for pSather. Holger Klawitter implemented type checking of parametrized classes. Arno Jacobsen worked on bound iterators. Illya Varnasky implemented inlining support and Trevor Paring implemented an early version of common subexpression elimination.

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. It seems that development of Sather-K has ceased silently sometimes in the past, although no detailed information could gathered about that. More information about Sather-K can be found at:

At the end of 1998, the development of Sather at the ICSI came to a halt. The last release there was the beta-version 1.2b. Short time later, this version was taken up into the GNU project by Norbert Nemec. The ICSI agreed to change the license to GPL/LGPL to make that step possible. From here on, GNU Sather will be developed in a open style, i.e. everybody is welcome to join the team and help improving the compiler, the library, the documentation and the tools that make up the Sather distribution.

As a intermediate step, the development of GNU Sather is being halted until a group at Waikato, New Zealand, led by Keith Hopper, have finished their complete rework of the Sather library, which is expected any time during fall 1999. After thorough testing of the library, it will replace the library we have right now.

1.6.1. The Name

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

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

1.6.2. Sather's Antecedents

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. That position was, in turn, taken over by Boris Vaysman in late 1995.

Sather has been very much a group effort; many, many people have been involved in the language design discussions including: Subutai Ahmad, Krste Asanovic, Jonathan Bachrach, David Bailey, Joachim Beer, Jeff Bilmes, Chris Bitmead, Peter Blicher, John Boyland, Matthew Brand, Henry Cejtin, Alex Cozzi, Richard Durbin, Jerry Feldman, Carl Feynman, Claudio Fleiner, Ben Gomes, Gerhard Goos, Robert Griesemer, Hermann Häertig, John Hauser, Ari Huttunen, Roberto Ierusalimschy, Arno Jacobsen, Matt Kennel, Holger Klawitter, Phil Kohn, Franz Kurfess, Franco Mazzanti, Stephan Murer, Michael Philippsen, Thomas Rauber, Steve Renals, Noemi de La Rocque Rodriguez, Hans Rohnert, Heinz Schmidt, Carlo Sequin, Andreas Stolcke, Clemens Szyperski, Martin Trapp, Boris Vaysman, and Bob Weiner. Countless others have assisted with practical matters such as porting the compiler and libraries.

Notes

1. 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.
2. C. Szyperski, S. Omohundro, S. Murer. "Engineering a programming language: The type and class system of Sather," In Jurg Gutknecht, ed., *Programming Languages and System Architectures*, p. 208-227. Springer Verlag, Lecture Notes in Computer Science 782, November 1993. Available at the Sather WWW page.
3. S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, "Iteration abstraction in Sather", *Transactions on Programming Languages and Systems*, Vol. 18, No. 1, Jan 1996 p. 1-15. Available at the Sather WWW page.
4. G. Agha, "Actors: A Model of Concurrent Computation in Distributed Systems", The MIT Press, Cambridge, Massachusetts, 1986.
5. 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.
6. C. Lim. "A Parallel Object-Oriented System for Realizing Reusable and Efficient Data Abstractions," PhD thesis, University of California at Berkeley, October 1993. Available at the Sather WWW page.

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

Chapter 2. Classes and Objects

All entities in Sather are objects, and objects are defined by classes. Even the basic entities in Sather, such as integers and floating point values are objects in Sather. Sather has several different kinds of classes - reference classes, abstract classes, immutable classes, partial classes and external classes. The important kinds of classes are reference classes and abstract classes - the rest are used in restricted circumstances. There are also some special objects (closures) which are not directly defined by classes, but we will defer their discussion till later.

Each Sather object has an associated type which indicates the class that was used to create the object. A variable in Sather also has a type, which indicates the kinds of objects it can be assigned to.

This chapter will focus on the most common kind of classes, reference classes, and the standard constructs used to create classes. Though iterators are an essential component of Sather code, their discussion has been deferred to the next chapter, since they are a relatively novel language feature.

2.1. Preliminaries

To make it easier to present examples in the following sections, we will start by introducting a few basic classes - integers, floating point numbers and strings. We will also describe how to print out data and to use the compiler

2.1.1. Some basic classes - INT, FLT and STR

Though basic numbers and strings enjoy some special language support (such as a means to initialize them to values like 5 or "foo"), they are defined as regular classes, and are a part of the standard library.The FLT class represents floating point numbers, while the INT class represents integers and the STR class represents strings. Variables may be declared to be of any of these classes and assigned when they are declared

```
a:FLT := 3.0;
b:INT := 5;
c:STR := "foo";
```

It is also possible to perform the usual operations on these classes, such as addition of numbers and concatenation of strings (represented by the "+" operator):

```
a:STR := "foo";
b:STR := "bar";         - + concatentates strings
c:STR := a + b;         - c is "foobar"
e:INT := 5;
```

```
f:INT := 7;
g:INT := e+f;            - g is 12
compare:BOOL := e > f;   - compare is false
#OUT + compare;          - Prints out 'false'
```

Comments in Sather start with a – and extend to the end of the line. Note that all variables have a default initial `void` value. For the present, `void` may be thought of as either the NULL pointer for reference objects, `0` for integers, `0.0` for floats and `false` for booleans.

2.1.2. Printing output

You can print data of various types in Sather using the command `#OUT+`

```
a:INT := 10;
#OUT + "hello world " + a;  - Prints out "hello world 10"
```

Treat `'#OUT+'` as an idiom for now; it is equivalent to the standard output routines in other languages.

2.1.3. Sather source files

Sather source files consist of lists of classes. In addition to the source files that a user specifies on the command line to the compiler, the standard library files are always implicitly examined. Definitions of the basic classes such as integers and strings as well as containers of all kinds are to be found in the standard library.

Execution of a Sather program begins with a routine named `'main'` in a specified class, (a class called `'MAIN'` is used by default). If main is declared to have a return value of type `INT`, this will specify the exit code of the program when it finishes execution.

2.1.4. Hello World

The hello world program is show below:

```
class HELLO_WORLD is
   main is
      #OUT+"Hello World\n";
   end;
end;
```

As we mentioned earlier, printing to standard output is obtained by calling `#OUT+`.

If the above code is stored in the file hw.sa, it can be compiled (using the ICSI Sather compiler) by:

```
sacomp -main HELLO_WORLD -o hw hw.sa
```

The '-main' option simply indicates to the compiler that the main routine will be found in class `HELLO_WORLD`. The resulting executable, `'hw'` can be run as follows

```
prompt> hw
Hello World
prompt>
```

2.2. Defining Classes and Creating Objects

Objects are usually models of conceptual or real-world entities; they consist of a combination of data, which models the state of the entity and operations which model the behavior of the entity. The body of a Sather class consists of a list of features which define the data and behavior of the class. A class defines a new type and may be used to create object instances of that type.[1]

We will start by describing the data elements and then move on to the operations. In subsequent sections, we will describe the definition of object behavior in the form of routines. We will then point out that Sather provides a level of abstraction, which permits the state and behavior of the object to be treated in a uniform manner. Finally, we will describe the somewhat unusual meaning of assignment in Sather that makes this uniformity possible.

2.2.1. Defining Simple Classes

The state of a class is defined by attributes, which are have the prefix `attr`:

```
class POINT is
   attr x:INT;
   attr y:INT;
end;
```

The `POINT` class above defines an `'x'` and a `'y'` attribute both of which are integers. This class is useless, as it stands, since it provides no way to create instances of itself.

Object Creation: `create` and `new`

To make objects of the `POINT` class, we have to introduce a `create` routine

```
class POINT is
   attr x, y:INT;

   create(xvalue,yvalue:INT):POINT is
```

```
      res:POINT := new;
      res.x := xvalue;
      res.y := yvalue;
      return res;
   end;
end;
```

The `create` routine first calls the special expression `'new'`. `'new'` creates a new uninitialized instance of the `POINT` class and returns it. All the attributes in the new instance have default 'void' values. It then assigns the `'x'` and `'y'` attributes of this new instance to `xvalue` and `yvalue` respectively. Instances of the `POINT` class can then be created as shown below

```
p:POINT := POINT::create(3,5);
```

Since creation is such a common operation, Sather provides a special shorthand for calls to the routine `'create'`. The `'create'` routine shown could be invoked with the # sign as shown below

```
point:POINT := #POINT(3,5);
```

Expressions using the `#` sign are referred to as *creation expressions*, and are a convenient shorthand used for creating new objects and initializing their attributes.

Attribute access

When an object of the class `POINT` is created, the `'x'` and `'y'` attributes may be accessed by 'dotting' into the object.

```
a:POINT := #POINT(3,5);
- Create a new point
#OUT + a.x ;
- Prints out the value of 'x', which is 3
a.x := 5;
- Sets the value of the 'x' attribute to 5
```

Points to note

- The semantics of a class is independent of the textual order of its class elements. In particular, the actual attribute layout used by a Sather implementation is invisible to a programmer.
- The scope of feature names is the class body
- Feature names may be either lower or upper case.

- Class names must be all upper case letters (underscores and digits are permitted except as the first character).
- The feature namespace is separate from the class namespace.
- The scope of class names is the entire program; no two classes can have the same name (unless they have different number of parameters, which will be explained in the chapter on class parametrization).
- You have to explicitly call `'new'` in the create routine. The following code exhibits a common error:

```
class POINT is
   attr x,y:INT;

   create(xval, yval:INT):POINT is
      x := xval;
      - Run time error! We have no object as yet!
      y := yval;
   end;
end;
```

2.2.2. Checking whether an object has been created

Before a variable is assigned to an object, the variable has the void value. The expression `'void'` may be used to determine whether a value is void or not.

The following example will print out the string "a is void!" since a `POINT` is a reference class and `'a'` has not been created.

```
a:POINT;

if void(a) then
   #OUT + "a is void!";
end;
```

In the second version, the string "a is not void!" will be printed since an object has been assigned to the variable `'a'`:

```
a:POINT := #POINT(3,5);

if void(a) then
   #OUT + "a is void!"
else
   #OUT + "a is not void!"
end;
```

Note that the above test will not work in the same way for some of the built-in classes such as integers and booleans[2].

2.2.3. Types Introduced

Each Sather variable and object has an associated type. The type of the object indicates the class that was used to create the object. In the following example, both 'a' and 'b' have the type POINT, indicating that they are associated with instances of the POINT class.

```
a:POINT := #POINT(2,3);
b:POINT := #POINT(4,5);
```

In this example, the type of the variable 'a' is the same as the type of the object to which it is assigned. This is always the case with the reference classes we have seen so far.

When we introduce abstract classes (unnamedlink), we will see that some Sather variables can hold objects of many different types. In this case, it is useful to distinguish between the type of the variable (called the declared type) and the type of the object that it holds (called the actual type or the concrete type).

2.2.4. Hiding features: private and readonly

A fundamental feature of object oriented programming languages is that they permit an object to hide certain features which are for internal use only. Attributes may be completely hidden by marking them `private`. Routines may likewise be marked private, meaning that they cannot be accessed outside the original class. Attributes can also be hidden so that they can be read but not modified from outside the class, by marking them `readonly`.

```
class POINT2 is
   private attr x:INT;
   - x cannot be seen from outside
   readonly attr y:INT;
   - y cannot be changed from outside
   create(xvalue,yvalue:INT):POINT is
      res:POINT := new;
      res.x := xvalue;
      res.y := yvalue;
      return res
   end;
end;
```

This restricts external access to the attributes in the object

```
foo is
   - some other piece of code
   a:POINT2 := #POINT2(3,5);      - Create a new POINT2
   #OUT + a.y;                    - Print out '5'
   - Illegal: #OUT + a.x;
```

```
   - Illegal: a.y := 10;
end;
```

Points to note

- Privacy is on a per-class basis, rather than on a per-object basis. Thus, an object can access the private features of other objects of the same class. We actually use this fact in the `create` routine of the class `POINT2` above. Assignments to the attributes of `res` are being done outside the object being returned.

2.3. Class Data: shared and const

In addition to object attributes, a class definition may also contain 'shared' data, which is shared by all the objects of that class.

2.3.1. Shared Attributes - Restricted global variables

Shared attributes are similar to object attributes, but are shared between all the instances of a class. They are essentially global variables that reside within a class namespace. They can be accessed and modified by any instance of the class. Shareds can have the same private and readonly restrictions that regular attributes have

```
private shared i,j:INT;
readonly shared c:CHAR := 'x'
```

Unlike regular attributes, when only a single shared attribute is defined, a constant initializing expression may be provided.

```
shared s:STR := "name";
- ILLEGAL shared s,p:STR := "name";
- cannot use initializing expression if two shareds are
- declared at the same time
```

If no initializing expression is provided, the shared is initialized to the value `'void'`.

2.3.2. Class Constants

Constants are accessible by all objects in a class and may not be assigned to - they must have an initializing expression from which their value is determined at compile time (there is an exception when no type is specified, as descrbed in the next subsection). If a type is specified, then the construct defines a single constant attribute which must be initialized to a constant expression. Constant expressions are recursively composed out of a combination of literals, function calls on literals, and references to other constants. More precisely, legal assignments are to

- a character, boolean, string, integer or floating point literal
- a `void` or `void` test expression
- an `and` or `or` expression, each of whose components is a constant expression
- an array literal, each of whose components is a constant expression
- a routine call applied to a constant expression, each of whose arguments is a constant expression *other than void*. This caveat is imporant, since `create` routines are called on void. Thus the following is illegal:[3]

  ```
  – ILLEGAL const  a:POINT := #POINT(3,3);
  const a:POINT := void;
  – The only legal kind of constant POINT is void
  ```

- a reference to another constant in the same class or in another class using the `'::'` notation.

  ```
  const r:FLT:=45.6;
  – Reader routine is private r:FLT;
  private const a,b,c;
  private const d:=4,e,f
  const bar:BOOL := r > 10;
  – Function call on constants
  const foo:ARRAY{INT} := |1,2,4,5,6|;
  – Sather arrays are explained later
  const  baz ::= BAR::foz ;
  – foz must be a constant expression in foz
  ```

Integer constants and Enumerated Types

If a type specifier is not provided, then no initializing expression is required and 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 is specified and there is an assignment that is not of type `INT`.

```
const a;
– a is of type INT and gets the value 0
const c,d;
– c gets 0 and d gets 1
const e := 3;
```

```
- e is also of type INT
```

Points to note

- There must not be cyclic dependencies among constant initializers.

  ```
  class FOO is
     const b:INT := BAR::c;

  class BAR is
     const c:INT := BAZ::d;

  class BAZ is
  - ILLEGAL!  const d:INT := FOO::b;
  - Introduces a cycle between b, c and d
  ```

- Since constant initialization involves permits operations on the built-in types, the operations on the built-in types are designed so that no observable side-effects can occur during constant initialization.
- The prefix readonly cannot be applied to constants, since constants cannot be modified in any case.
- Due to their definition, constants are only useful for the basic classes such as numbers, booleans and characters. All other constants can only be assigned to be void!

  ```
  class FOO is
     const a:BAR := void;
     - only legal value
  ```

2.3.3. Accessing Class Data - the :: notation

It is possible to directly access the class data or features using the :: notation.

```
class FOO is
   const a:INT := 3;
   private const b:INT := 5;
   readonly shared c:INT := 6;
   shared d:INT := 7;
   attr f:INT;

   create(i:INT):SAME is
      res:SAME := new;
      res.f := i;
      return res;
   end;
```

```
   method1:INT is
      return d+a;
   end;

   method2:INT is
      return f+a;
   end;
end;
```

The shared and const class data can then be accessed using the :: notation

```
#OUT+ FOO::a+"\n";
FOO::d := 3;
```

When a method is called using the `'::'` notation, it is equivalent to calling the method on a void object. Calling a method on a void object makes sense if the feature only makes use of shared data and local state. If the method makes use of object data, a run-time error will result.

```
#OUT+FOO::method1;
- Prints out d+a = 10
#OUT+FOO::method2;
- Tries to print out self.f+a
- However, self (the object) is void, so trying to access 'f'
- results in a run-time error - Attribute access of void
```

- The usual privacy and modification restrictions are maintained

  ```
  a_copy:INT := FOO::a;
  - ILLEGAL FOO::c :=3; - c is readonly
  - FOO::a := 7;  - a is a constant
  ```

2.4. Routine definitions

The behavior of a class is specified by routines in the class body. Routines may take arguments and may return a value.

```
class CALCULATOR is
   attr running_sum:INT;

   create:CALCULATOR is
      res:CALCULATOR := new;
      res.running_sum := 0;
      return res;
```

```
   end;

   add(x:INT):INT is
      res:INT := running_sum + x;
      return res;
   end;
end;
```

A routine definition may begin with the keyword 'private' to indicate that the routine may be called from within the class but is not visible from outside the class. The methods that are visible from outside the class are referred to as the class interface.

The body of a routine is a list of statements, separated by semicolons. In a routine with a return value, the final statement along each execution path must be a return statement. Thus, the following is not legal:

```
scale_x(x:INT):INT is
   - Illegal routine - the else clause has no return value
   if x > 0 then
      return 15;
   else
      #OUT + "Error!";
      - last statement on this branch is not return
   end;
end;
```

A raise statement raises an exception, and can be used wherever a return statement might be required. (See unnamedlink for more details) For now, we merely note that the following version of the routine 'scale_x' does not return a value in the second branch of the if statement, but raises an exception instead, which is perfectly legal.

```
scale_x(x:INT):INT is
   if x > 0 then
      return 15;
   else
      raise "An error occurred!";
   end;
end;
```

Using the return value

Note that, unlike most other languages, Sather forces you to make use of the return value. This may be considered an extension of strong typing - the presence or absence of a return value is a part of the signature that should not be ignored.

```
new_x:INT := scale_x(15);
```

```
- Legal, the return value used
scale_x(15);
- ILLEGAL! Return value unused
```

The return value can also be used as part of an expression.

```
a := scale_x(15) + 3;
```

2.4.1. Routine Arguments and Modes

The arguments to a routine are specified as a comma-separated list. Each argument must provide a name and type. The types of consecutive arguments may be declared with a single type specifier.

```
create(x,y:INT):POINT ...
```

The scope of method arguments is the entire body of the method, and also shadows methods and attributes in the class. If a routine has a return value, it is declared by a colon and a specifier for the return type. You can get around this restriction by using the `self` expression explicitly

```
class POINT is
   attr x,y:INT;
   add_x(x:INT) is
      self.x := self.x + x;
   end;
end;
```

Each argument also has a *mode* which determines how that argument is treated when the routine is called. If no mode is explicitly stated, the argument mode is `in`. That means it is simply a value sent into the routine. The other possible modes are `out`, `inout` and `once` (which will be described in the section on iterators).

Multiple return values and `out` arguments

An out argument is really like an extra return value. An `out` argument is not set when the routine is called; rather, it is filled in by the routine itself. Consider an integer division function that returns both the divident and remainder of the two integer arguments

```
divide(x,y, out dividend, out remainder:INT) is
   - Note that the 'INT' type specifier applies to  multiple
   - arguments while the mode qualifiers apply to only one
   - argument.
   dividend := x/y;
   - Integer division result
   remainder := x - y*(x/y);
```

```
   - Remainder after the division.
   - Could also use x.mod(y)
end;
```

The divide routine may be used as shown below:

```
a:INT := 15;
b:INT := 10;
div, rem:INT;
- These are defined but not assigned
divide(a,b,out div, out rem);
#OUT + "Divident=" + div + " Remainder=" + rem + "\n";
- Prints out "Divident=1 Remainder=5"
```

Note that the `out` argument has to be marked both where the method is defined (i.e. as a marker of the formal parameter) and at the point of call, or the compiler will complain (`once` and `in` arguments need not be mentioned at the point of call)

`inout` arguments

`inout` arguments are a combination of `in` and `out` arguments. They take a value into the function and return a value out of the function. We can thus write the swap function compactly as:

```
swap(inout x, inout y:INT) is
   tmp:INT := x;
   x := y;
   y := tmp;
end;

a:INT := 5;
b:INT := 10;
- a and b have an initial value
swap(inout a,inout b);
#OUT + "a=" + a + " b=" + b;
- Prints "a=10 b=5"
```

The list below describes the argument modes in more detail:

`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

Once parameters are discussed in detail in the chapter on Loops and Iterators (See unnamedlink). 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, and are not marked at the point of call.

2.4.2. Local Variables - Scoping and Shadowing

Declaration Statements are used to declare the type of one or more local variables. 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. Local variables shadow routines (including the accessor routines of attributes) in the class which have the same name and no arguments.

```
... in the POINT class ...
swap_x_y is
   temp:INT;
   temp := x;
   x := y;
   y := temp;
end;
```

Within the scope of a local variable it is illegal to declare another local variable with the same name.

Points to note

- Local variables are initialized to void when the containing method is called.
- Local variables are *not re-initialized* when the declaration is encountered in the flow of control. This is particularly relevant in loop statements, which are discussed in the next chapter. The integer 'a' is

initialized to zero when the function `'compute'` is entered. It is not initialized every time through the loop.

```
compute is
   loop 3.times!;
      a:INT;
      a := a + 3;
      #OUT + a + "\n"; - Prints out successively 3, 6, 9
   end;
end;
```

- Note that explicit initialization (in this case `'a:=15'`) is performed every time it is encountered

```
compute is
   loop 3.times!;
      a:INT := 15
      a := a + 3;
      #OUT + a + "\n"; - Prints out successively 18, 18, 18
   end;
end;
```

2.4.3. Routine calls

The most common expressions in Sather programs are method calls[4]. A routine call usually takes the form of a 'dotted' expression such as `a.foo(b).` The object on which the routine is being called (`'a'` in this example) is determined by what precedes the dot. If no object name precedes the 'dot', the `self` object i.e. the current object, is assumed. We use the following definition of the POINT class to

```
class POINT is

   attr x,y:INT;

   create(x,y:INT):POINT is
      res:POINT := new; res.x := x;  res.y := y;  return res;
   end;

   add(xval,yval:INT):POINT is
      xsum:INT := x + xval;
      ysum:INT := y + yval;
      res:POINT := #POINT(xsum, ysum);
      return res;
   end;

   offset_by(val:INT):POINT is
      return add(val,val);    - short for 'return self.add(val,val);'
```

```
   end;
end;
```

illustrate different kinds of routine calls

- If nothing precedes the method name, then the form is syntactic sugar for a call on `self` If the method name is preceded by an expression and a dot '`.`', then the method is called on the object returned by the expression. In the following example, pair (3,7) is first added to `p1` and the pair (4,9) is added to that result. Note that the intermediate point that is created after the first 3,7 is added is not accessible from any variable and will be garbage collected.

  ```
  p1:POINT := #POINT(3,5);
  p2:POINT := p1.add(3,7).add(4,9);
  ```

- If the method name is preceded by a type specifier and a double colon '`::`' it is presumed to be a call on a `void` object of the specified class (`POINT` in the case below)

  ```
  a:POINT := POINT::create(3,5);
  ```

This works for the `create` routine, since it creates a new object, `res,` and then makes use of it. However, this will not work for a call on, say, `add`

```
res:POINT := POINT::add(4,7);
- Runtime Error!
```

Since `xsum := x + xval;` is actually equivalent to saying `xsum := self.x + xval;` the routine accesses `self`, which is void and cannot be accessed.

2.4.4. Simple Overloading - Selecting a routine to call

Sather supports routine *overloading*. We will present a simplified version of the overloading here, as it applies to the simple reference classes we have discussed. The full overloading rule will be described in more detail lateron (See unnamedlink).

Two routines in a class may have the same name provided they differ in at least one of the following aspect:

- the number of arguments
- the presence or absence of a return value
- the type of one of the arguments (provided the types are not abstract).

Here are some examples of properly overloaded routines

```
foo(a:INT, b:INT);
foo(a:INT);            - Different number of arguments
```

```
foo(a:INT,b:INT):INT;  - Has a return value
```

All of the above routines could co-exist in a single class interface. The right one would be selected at the point of call. The following two routines, however cannot co-exist in the same interface

```
foo(a:INT,b:INT):INT;
- foo(a:INT,b:INT):BOOL
- differs only in return type, cannot overload 'foo'
```

2.5. Conditional Execution

Sather supports the standard constructs for conditional execution - if statements and multi-way case statements

2.5.1. *if* statements

if statements are used to conditionally execute statement lists according to the value of a boolean expression. In this form, the *if* keyword is followed by a boolean expression, the keyword *then*, a list of statements and the final keyword *end*. When the statement is executed, the boolean expression is evaluated and if the result is *true* the statements in the statement list are executed. If it is *false*,then control passes directly to the end of the *if* statement.

```
i:INT := -15;
if i < 0 then
   i:=-i;
end;
#OUT + i;                 - Prints out 15
j:INT := 15;
if j < 0 then
   j:=-j;
end;
#OUT + j;                 - Prints out 15
```

It often happens that one wishes to perform a sequence of tests, executing only the statements which correspond to the first test in the sequence which evaluates to `true`. For example, we may want to produce a integer value `'y'` from an integer value `'x'` which has the shape of a triangular bump. It should be zero when `'x<0'`, equal to `'x'` when `'0<=x<100'`, equal to `'200-x'` when '`100 <= x<200'`, and equal to `'0'` when `'x>=200'`. This can be accomplished with a nested series of *if* statements:

```
if x < 0 then
   y := 0;
```

```
else
   if x < 100 then
      y := x;
   else
      if x < 200 then
         y := 200 - x;
      else
         y := 0;
      end;
   end;
end;
```

Because this kind of construct is so common and the deeply nested `if` statements can get confusing, Sather provides a special form for it. A series of `elsif` clauses may appear after the statements following the `then` keyword:

```
if x < 0 then
   y := 0;
elsif x < 100 then
   y := x;
elsif x < 200 then
   y := 200 - x;
else
   y := 0;
end;
```

There may be an arbitrary number of such *elsif* clauses. Each is evaluated in turn until one returns true. The statement list following this clause is evaluated and the statement finishes. If none of the expressions is true, the statements following the final `else` clause are evaluated.

2.5.2. *case* statements

Multi-way branches are implemented by *case statements*. There may be an arbitrary number of *when* clauses and an optional *else* clause. The initial construct is evaluated first and may have a return value of any type.

```
i:INT := 7;
switch i
when 1,2,3 then
   j := 3;
when 4,5,6 then
   j := 4;
when 7,8,9 then
   j := 5;
```

```
else
   j := 10;
end;
#OUT + j;            -- Prints out 5
```

This type must define one or more routines named 'is_eq' with a single argument and a boolean return value.

```
class POINT is
   attr x,y:INT;

   create(x,y:INT):POINT is
      res:POINT := new; res.x := x; res.y := y; return res;
   end;

   is_eq(point2:POINT):BOOL is
      -- In Sather,= is short hand for a call on 'is_eq'
      return x = point2.x and y = point2.y;
   end;

   str:STR is return "X=" + x + " Y=" + y; end
end;
```

Points can then be used in a case statement as shown below

```
p:POINT := #POINT(3,4);
zero_point:POINT := #POINT(0,0);

case p
when zero_point then
   #OUT + "Zero point\n";
when #POINT(1,1), #POINT(1,-1),#POINT(-1,-1), #POINT(-1,1) then
   #OUT + "Unit point: " + p.str + "\n";
else
   #OUT + "Some other point\n";
end;
```

Note that the equal sign is really short hand for the routine `is_eq`. The case statement is equivalent to an `if` statement, each of whose branches tests a call of `is_eq`. Thus the above case is equvalent to

```
if p = zero_point then
   #OUT + "Zero point\n";
elsif p = #POINT(1,1) or p = #POINT(1,-1) or ... etc. then
   #OUT + "Unit point:" + p.str + "\n";
else
   #OUT + "Some other point\n";
```

```
end;
```

The expressions tested in the branches of the `if` statement are the expressions of successive `when` lists. The first one of these calls to returns `true` causes the corresponding statement list to be executed and control passed to the statement following the *case* statement. If none of the `when` expressions matches and an *else* clause is present, then the statement list following the *else* clause is executed

There is one difference between the *case* statement and the equivalent *if* statement. If none of the branches of an *if* statement match and no *else* clause is present, then execution just continues onto the next statement after the *if* statement. However, if none of the branches of the *case* statement matches and there is no *else* clause, then a fatal run-time error will result.

Points to note

- It is a fatal error if no branch matches and there is no *else* clause for *case* statements but not for *if* statements.

2.5.3. Short circuit boolean expressions: *and* and *or*

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* 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.

Consider the code

```
p: POINT;
if p.x > 3 then
   #OUT + p.x;
end;
 - Runtime error if p is void
```

The above block of code will work if `p` is not void. If it is void, however, the test `p.x >3` will result in a runtime error, since it is attempting to dot into a void reference type. We can catch this problem by using the following piece of code, and the semantics of the short-circuit and

```
if ~void(p) and p.x > 3 then
   - The ~ symbol indicates logical negation
   #OUT + p.x;
end;
```

The above piece of code will not generate an error, even if `p` is void. The first part of the and expression tests for whether `p` is void. If it is void, then the void test returns true and the *not* turns this into a false. The *and* therefore fails before trying to evaluate the dotted expression `p.x`.

A similar behavior can be seen with the short-circuit *or* statement, where the second expression is not examine if the first expression evaluates to true

```
a:INT := 15;
p:POINT;
if a>10 or p.x < 10 then
   - Since a>10 is true, the second expression is not evaluated
```

- Note that booleans also define an `and_rout` routine, which does not have the same short-circuit behavior:

  ```
  if ~void(p).and_rout(p.x > 3) then
  - May generate a run-time error, when 'p' is void
  - The argument to the 'and_rout' routine (p.x) is evaluated
  - even when the first condition, ~void(p) fails.
  - Hence, if 'p' is void, p.x is still evaluated and generates a
  - run-time error (attribute access of void)
  ```

2.6. Attribute Accessor Routines

The distinction between data and behavior is not as strong as has been described above. In fact, it is possible to implement a feature such that outside the class it is impossible to tell whether it is a feature or a pair of functions This section describes how this level of uniformity is achieved.

Each attribute definition adds a field to the object's state and 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 attribute. Its return type is the attribute's type. The reader routine is private if the attribute is declared '`private`'. The writer routine sets the value of the attribute, taking a single argument whose type is the attribute's type, and has no return value. The writer routine is private if the attribute is declared either `private` or `readonly`.

```
class INTERVAL is
   attr start:FLT;
   - Defines the public reader start:FLT
   - and the public writer     start(FLT)
   attr finish; INT;

   create(st,fin:INT):SAME is
      - Create a new interval
```

```
      res:SAME := new;
      res.start(st);
      - Equivalent to res.start := st;
      res.finish(fin);
      - Equivalent to res.finish := fin;
   end;
end;
```

Thus, the levels of privacy are defined by whether the reader and writer routines are public or private

```
private attr a:FLT;   - Defines the reader, private a:FLT
                      - and the writer  private a(FLT);
readonly attr b:FLT;  - Defines the public reader, b:FLT
                      - and the private writer b(FLT)
```

The same holds true for shared attribubtes. 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.

```
class FOO is
   shared a:INT := 3;     - Defines a:INT and a(arg:INT);
   readonly shared b:INT; - Defines a:INT and private a(arg:INT);
...

#OUT + FOO::a;    - Prints out 3
FOO::a(4);        - 'a' is set to 4, same as FOO::a := 4;
#OUT + FOO::a;    - Prints out 4;
FOO::a := 7;      - 'a' is set to '7' , equivalent to FOO::a(7);
FOO::b(3);        - ILLEGAL! The writer routine is private
```

Constants do not define a writer routine. 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`'.

```
const r:FLT:=45.6;
   - Reader routine is r:FLT;
private const a,b,c;
   - Reader routine is private a:INT;
private const d:=4,e,f
const bar:BOOL := r > 10;
   - Function call on constants
```

2.6.1. Attribute assignment

In order to achieve the unification of attribute assignment and routine calls, for attributes, assignment has to be given a meaning in terms of function calls.

By default, the assignment is syntactic sugar for a call of the routine with the same name as the attribute with the right hand side of the assignment as the only argument

```
p:POINT := #POINT(3,5);
p.x := 3; - Is syntactic sugar for  p.x(3);
```

In the above example, the assignment to 'x' is the same as calling the routine 'x' with a single argument.

Replacing an attribute by a routine

The beauty of this treatment of assignment is that an attribute in a class can later be substituted by a pair of routines. Consider a class to represent integer intervals, where we store the first and last value in the interval

```
class I_INTERVAL is
   - Integer intervals
   attr start:INT;     - Defines start:INT and start(INT)
   attr finish:INT;    - Defines finish:INT and finish(INT)

   create(start,finish:INT):SAME is
      res:SAME := new;
      res.start := start;       - Equivalent to res.start(start);
      res.finish := finish;     - Equivalent to res.finish(finish);
      return res;
   end;

   size:INT is return finish - start + 1; end;
        - Returns the number of integers in the interval
end;
```

We can make calls on this class

```
i:I_INTERVAL := #I_INTERVAL(3,10);
i.finish := 11;
- Equivalent to a call i.finish(11);
#OUT + i.finish;
- Prints out 11
i.start := 15;
- Equivalent to the call i.start(15);
```

Suppose we then realize that we usually want to know the size of the interval, and rarely need to know the end point. It would then be cheaper to store the size directly, rather than computing it. The class can be changed so that we store the `first` and `size` and compute `finish`.

```
class I_INTERVAL is
   - Integer intervals
   attr start:INT;          - Defines start:INT and start(INT)
   readonly attr size:INT;  - Defines  size:INT and private size(INT)
   - size is readonly, since we only need size:INT in the interface

   create(start,finish:INT):SAME is
      res:SAME := new;
      res.start := start;          - Equivalent to res.start(start);
      res.size :=  finish-start+1; - Store the result in res.size
      return res;
   end;

   finish:INT is return start+size-1 end;
      - Replacement for the reader routine for 'finish'
      - Compute finish using 'start' and 'size'

   finish(new_finish:INT) is  size:=new_finish-start+1  end;
      - Replacement for the writer routine for 'finish'

end;
```

All the calls described above will continue to work as before. The assignment to `finish` in particular will now be a call on the user-defined finish routine, instead of a call to the implicit writer routine for the attribute `finish`.

2.7. Static Type Inference

For the sake of convenience, Sather provides a mechanism for statically inferring the type of a variable from the context. This type-inference takes place in different situations, where the type is available from the context.

2.7.1. Creation Expressions

In a creation expression, it is tedious to have to repeat the type of a class on both sides of a creation expression and assignment. Hence, the # symbol may infer its type from the context.

```
a:POINT;
a := #(3,4);
- Equivalent to a := #POINT(3,4);
```

2.7.2. Assignments and ::=

Type inference can also take place in a declaration, if it is combined with an assignment. Since the declared type of the right hand side of the assignment is known, its type is used as the type of the variable. This combination of declaration and assignment is extremely common in Sather code.

```
a ::= 3;                     - Equivalent to a:INT := 3;
p1:POINT := #POINT(3,5);
p2:POINT := #POINT(4,5);
p3 ::= p1.add(p2);           - 'p3' is of type POINT.
- Assumes the function 'add' in POINT i.e. POINT::add(POINT,POINT);
```

When an assignment is associated with a creation, we can make use of either form of type inference

```
a ::= #POINT(3,4);  - Equivalent to a:POINT := #POINT(3,4);
a:POINT := #(3,4);  - Means the same
```

2.7.3. Arguments to a function call

The type of the arguments to a function call are also known and can be used to infer the type of a creation expression in a call to the function.

```
foo(a:POINT) is  ...

foo(#(3,5));
- The create expression infers its type
- from the type of the argument that 'foo' is expecting
```

This form of type inference may be used for closure creation expressions as well, which will be discussed in the chapter on Closures

```
apply(arg:ROUT{INT}:INT) is ....
apply(bind(3.plus(_));
```

If the plus routine in the INT class is overloaded, then the appropriate routine is chosen based on the declared type of the argument to 'apply' i.e. ROUT{INT}:INT. Note that if both the 'apply' routine and the 'plus' routine are overloaded, type inference may not be able to determine the type and it might be necessary to create a temporary variable with the right type

```
r:ROUT{INT}:INT := bind(3.plus(_));
apply(r);
```

In any case, we strongly recommend that static type inference not be used in cases where confusion might result; the extra typing is usually worthwhile!

2.8. Class Parameters

We will briefly describe simple parametrized classes here so that they may be used in examples through the rest of the text. For a full description of parametrized classes, please see the chapter on Parametrized Classes.

A Sather class may have various type parameters, which are basically place holders for types which are specified when the class is actually used. This allows us to write code that is *generic* and can be used with a different types. By convention, these type parameters are given names like `T` or `TP`. We show below a class `TUP`, which holds pairs of objects. Since we would like to be able to hold objects of any types, we just specify type parameters, `T1` and `T2`. These parameters are place-holders, which must be set to actual honest-to-goodness concrete classes when the `TUP` is actually used

```
class TUP{T1,T2} is
   - Simple version of the library tuple class
   attr t1:T1;
   attr t2:T2;
   create(t1:T1, t2:T2): SAME is
      -  Standard create routine. Arguments use the type parameters
      res ::= new;   - Using static type inference - new returns SAME
      res.t1 := t1;  - The types of res.t1 and the argument t1
                     - are both T1 so the assignment is legal
      res.t2 := t2;
      return res;
   end;
end;
```

We can now create a tuple object that holds, for instance, a pair consisting of a string and an integer:

```
t ::= #TUP{INT,STR}(5,"this");
- Create a new tuple.
- Uses ::= to determine the type of 't'
#OUT + t.t1 + "\n";
```

2.8.1. Arrays

A standard parametrized class is the array class, `ARRAY{T}`. (See unnamedlink for more details). When an array is actually used to hold objects, the type parameter must be instantiated to indicate the kind of objects being held.

```
a:ARRAY{INT} := |2,5,7|;
- Special syntax for initializing an array with values 2,5,7
#OUT+a[1];
- Return the second element of the array
```

For example, arrays are used to pass in the arguments to a program into the main procedure.

```
main(args:ARRAY{STR}) is
   #OUT + args[0];
   - On unix, args[0] is the name of the program
end;
```

We can hold a collection of points using an array, as follows

```
a:ARRAY{POINT} := #(3);
a[0] := #POINT(0.0,0.0);
a[1] := #POINT(0.0,1.0);
a[2] := #POINT(2.0,2.0);
```

2.9. A Running Example: Employees

We will illustrate the points made above by using a simple example, which will be something of a running example to be extended in later chapters. We will start here by defining a class `'EMPLOYEE'`. Please bear in mind that this example is used to illustrate various language features, not object-oriented design.

EMPLOYEE definition

The class is composed of several attributes which hold the employee information. Various degrees of privacy are illustrated

```
class EMPLOYEE is
   private attr wage:INT;
   readonly attr name:STR;
   attr id:INT;
   const high_salary:INT := 40000;

   create(a_name:STR, a_id:INT, a_wage:INT):SAME is
      res ::= new;
      res.id := a_id;
      res.name := a_name;
      res.wage := a_wage;
      return(res);
   end;

   highly_paid:BOOL is
      return wage >= high_salary;
   end;
```

```
end;
```

Note the use of the special type `SAME` as the return type of the `create` routine, which denotes the current class name. `SAME` changes to mean the including class when it is included, as will be explained in the next chapter on code inclusion.

TESTEMP definition

The employee class may be exercised using the following main class.

```
class TESTEMP is
   main is
      john:EMPLOYEE := #EMPLOYEE("John",100,10000);
      peter:EMPLOYEE := #EMPLOYEE("Peter",3,10000);
      john.id := 100;           - Set the attr "id" in john to 100
      #OUT + john.name + "\n";  - Prints "John"
      #OUT + peter.id + "\n";   - Prints "3"
   end;
end;
```

Note that the following calls would be illegal:

```
#OUT + john.wage + "\n";  - ILLEGAL! "wage" is private
john.name := "martha";    - ILLEGAL! "name" is readonly
```

A distinguished class must be specified when a Sather program is compiled (the default is to look for a class called `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.

Running the example

To run the above example - type the code into a file emp.sa and then run the executable 'emp'

```
sacomp emp.sa -main TESTEMP -o emp
```

This generates the executable "emp", using the "main" routine in `TESTEMP` as its starting point. You can browse the resulting code by calling

```
sabrowse emp.sa -main TESTEMP
```

Notes

1. This is only true for reference, immutable and some kinds of external classes. Abstract a, partial and most external classes cannot have instances.

2. The void test returns true for all integers with a value of 0 and booleans with a value of false. In general, the void test is not useful for immutable classes.
3. Implementation Note: The compiler currently does not always detect this illegal case.
4. We use the term 'method' here to indicate that the same description is applicable to both iterators, which have not yet been introduced, and routines.

Chapter 3. Loops and Iterators

Until now we have only made use of the simple loop constructs, while! and until!. These are similar to the loop constructs found in other languages, aside from the terminal '!'. However, that terminal '!' hides something very special about these loop constructs in Sather; a programmer can define such looping constructs as easily as he or she could define a standard routine. Once defined, they may be used as conveniently as the while! and until! iterators.

To a first approximation, iterators are like streams that can "yield" different values on successive loop iterations. When an iterator has no more values to yield, it "quit"s. This, in turn, terminates the enclosing loop.

Iterators are defined as class features, just like routines, but iterator names must terminate with an '!'. 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. In this, the iterator is similar to a coroutine whose state remains persistent over multiple calls. 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.

3.1. Using iterators

3.1.1. *loop* statements

Iteration is done with *loop statements*, used in conjunction with iterator calls. In the absence of iterator calls, a loop statement simply executes an infinite loop. The difference between an iterator call and a routine call is that the iterator call "remembers" its state after it yields a value and, on subsequent calls, it simply resumes execution. The "lifetime" of an iterator usually includes several calls within a particular loop. Hence, an execution state is maintained for each iterator call textually enclosed within a loop - this execution state will be used to "remember" the state of the iterator between invocations. When a loop is entered, the execution state of all enclosed iterator calls is initialized. When an iterator is encountered, control is transferred to the iterator until the iterator "yields" control. Just as a routine may provide a value when it returns, so too an iterator may provide a value when it yields.

```
sum: INT := 0;
loop sum := sum + 1.upto!(10); end;
#OUT + sum + '\n';              - Prints sum of integers from 1 to 10
```

Instead of yielding control back to the enclosing loop, an iterator may also terminate or quit, which terminates the enclosing loop.

Note that each loop may contain more than one iterator call, thus providing much more flexibility than conventional languages. When any of the iterators terminates, the whole loop terminates, and execution continues at the next statement after the loop.

3.1.2. Built-in iterators

The `until!`, `while!` and `break!` iterators are built-in. They have the standard definitions of until, while and break in other languages and may occur anywhere in the loop body. `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 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 are iterator calls which immediately quit when they are called.

```
sum:INT := 0;
i:INT := 0;
loop while!(i < 5);
   sum := sum + i;
   i := i + 1;
end;
#OUT+ "Sum=" + sum + '\n';          - Prints out Sum=10
```

The `break!` iterator can be used to terminate a loop at any time. We illustrate this with the bubble sort routine show below, which terminates the first time a pass through the data occurs with no order change.

```
bubble_sort(a:ARRAY{INT}) is
   loop
      done: BOOL := true;
      i:INT := 0;   - Loop until the "break!" is encountered
      loop until!(i = (a.size-2));
         if a[i] > a[i+1] then
            done := false
            swap(inout a[i], inout a[i+1]);
         end;
         i := i + 1;
      end;
      if done then
         break!;
      end;
   end;
end;
```

The `'swap'` routine is as we have described earlier.

```
swap(inout x:INT, inout y:INT) is
```

```
   tmp:INT := x;
   x := y;
   y := tmp;
end;
```

Note that the above `'bubblesort'` routine could easily be rewritten to only use `until!`.

In addition to the built-in iterators, there are many commonly used iterators in the INT class that provide for interation over a range of values. For instance, the iterator `upto!` yileds successive integer values.

```
sum:INT:= 0;
loop
   sum := sum + 10.upto!(20);
end;
```

The `upto!` iterator returns successive integers from 10 upto 20, inclusive. Below, are examples of a few other common iterators

```
i:INT := 10;
sum:INT := 0;
loop
   11.times!;
   sum := sum + i;
   i := i + 1;
end;
```

The `'times'` iterator yields a certain number of times. For iterating over a range with a certain stride, use the `'step'` iterator

```
sum:INT := 0;
loop
   sum := sum + 18.step!(11,2);
end;
- The first argument is the number of iterations, 11 in this case
- the second argument is the stride
```

The following example counts 11 even numbers starting at 18

The `'step_upto!'` iterator is similar, but instead of specifying a number of iterations, it specifies the maxium value to be reached

```
sum:INT := 0;
loop
   sum := sum + 18.step_upto!(40,2);
end;
```

The following loop is equivalent to the preceeding one.

3.2. Defining Iterators

3.2.1. *yield* statements

Iterator definitions are similar to routine definitions, except that we need to indicate when control should be transferred back to the calling point. In a routine, this transfer of control is indicated by a `return` statement, which terminates the routine. An iterator, however, can return control in two different ways. It can either

- Temporarily yield control to the callig point, ready to continue the next time it is encountered. This yield of control is done by a `yield` statement
- Permanently yield control to the calling loop, terminating the loop in the process. This termination of the enclosing loop is achieved by a `quit` statemen or by reaching the end of the iteratort.

The `yield` must return a value (of the appropriate type), if the iterator has a return value.

```
range!(min, max:INT):INT is
   i:INT := min;
   loop until!(i > max);
      yield i;
      i := i + 1;
   end;
end;
```

This iterator can be used to add up all the numbers in a particular integer range

```
sum: INT := 0;
loop
   sum := sum + range!(1,10);
end;
```

3.2.2. Explicitly leaving an iterator using *quit*

When an iterator has yielded as many times as needed, it can either reach the end of it's statement list or explicitly call a quit statement. *quit statements* are used to terminate loops and may only appear in iterator definitions. No value is returned from an iterator when it quits. No statements may follow a `quit` statement in a statement list. The following definition of 'range!' is equivalent to the preceeding definition:

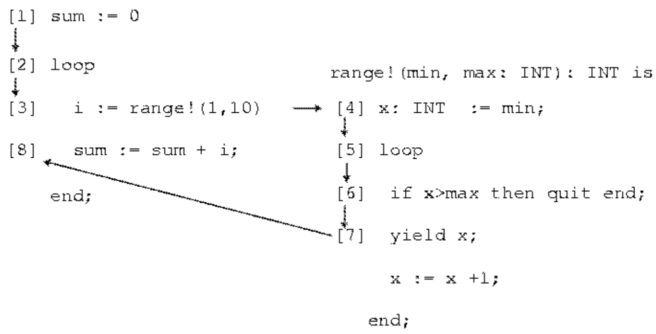
```
range!(min, max:INT):INT is
   x:INT := min;
   loop
```

```
      if x > max then
         quit;
      end;
      yield x;
      x := x + 1;
   end;
end;
```

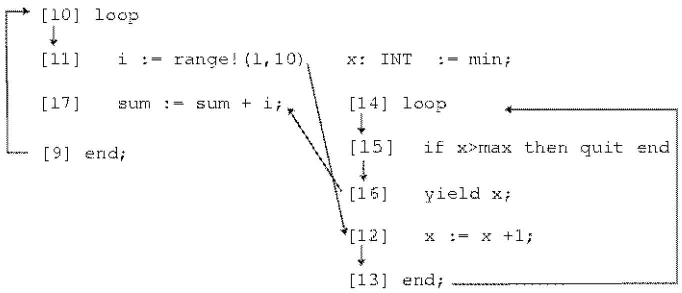
3.2.3. Control flow within an iterator

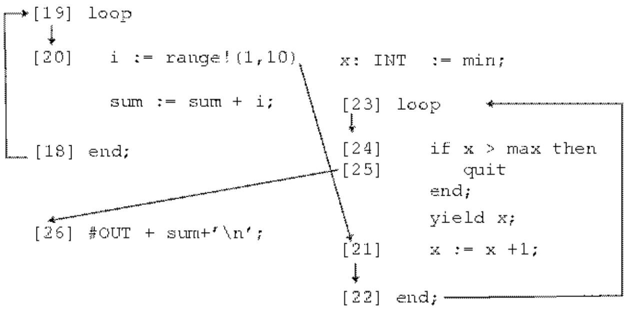
The following figures illustrate the control flow between an interator and its calling loop.

When the iterator is first called, control goes into the iterator and then returns to the outer loop, when the iterator yields in step [7]



After the first yield, control continues in the outer loop until the iterator is encountered again in step [11] and control is again transferred to the iterator, right after the point of the yield, in step [12]





The above sequence will continue until the `if` statement is true and the `quit` statement is encountered in the iterator. Control is then transferred to the end of the enclosing loop. The iterator calling context keeps track of the internal state of the iterator from the last yield.

3.2.4. The *once* argument mode

One problem with the above definition of `'range!'` is that the arguments to the function will be evaluated each time through the loop. Consider the following loop

```
sum:INT := 0;
max:INT := 5;
loop
   sum := sum + range!(3,max);
   max := max+2;
end;
```

This, somewhat silly, example will go into an infinite loop, since the argument `'max'` increases each time through the loop.

Iterator argument are *hot* by default. This means that the arguments will be re-evaluated and passed to the iterator each time through the loop. When the arguments to the iterator are constant, it is not important whether they are re evaluated or not. However, in many cases it is important to ensure that the argument is only evaluated the first time through the loop.

This happens to `once`-arguments. Arguments which are marked with the mode '`once` are only evaluated the first time they are encountered during a loop execution. Thus, the correct definition of the 'range' iterator is:

```
range!(once min, once max:INT):INT is
   i:INT := min;
   loop
      until!(i > max);
      yield i
```

```
      i := i + 1;
   end;
end;
```

Note that 'once' arguments are only marked at the point of definition, not at the point of call. Thus, invoking the loop will look the same as before

```
sum:INT := 0;
loop
   sum := sum + range!(3,5);
end;
```

The 'self' parameter (i.e. the object on which the iterator is being called) is *always* a once parameter.

```
i:INT := 5;
loop
   #OUT + i.upto!(11)! + ' ';
   i := 1;
end;
- The above loop prints out 5 6 7 8 9 10 11
```

In the above example, though the value of 'i' changes the second time through the loop, the change is ignored - the first value of 'i' is used.

The following more complex example will sum up some of the elements of the first row although the variable `row` will contain different rows in consecutive loop iterations.

```
loop  - Sum up some of the elements of the first row!
   row := matrix.row!;
   sum := sum + row.elt!;
   - row is only evaluated at the first iteration!
end;
```

3.2.5. *out* and *inout* argument modes

Yield causes assignment to `out` and `inout` rguments in the caller i.e. these arguments are assigned each time when the iterator yields..

```
range!(once min, once max:INT, out val:INT) is
   i:INT := min;
   loop until!(i > max);
      val := i;
      yield;
      i := i + 1;
   end;
end;
```

Which may be used by:

```
sum:INT := 0;
loop
   res:INT;
   range2!(3,5, out res);
   sum := sum + res;
end;
#OUT + sum + '\n'; - Prints out 12
```

Note that no assignment to `out` and `inout` arguments takes place when an iterator quits.

3.2.6. *pre* and *post* conditions in iterators

The behavior of pre- and post- conditions in iterator definitions is a natural extension of their behavior in routine definitions. 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.

3.2.7. Argument evaluation in iterators

At a more technical level, 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) 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.

3.2.8. Points to note

Iterator usage

- Iterators may only be called within `loop` statements.
- `once` mode arguments are only marked at the point of definition, not at the point of call, unlike `out` and `inout` arguments. `out` and `inout` arguments cannot be `once` arguments as well.
- Each textual instance of an iterator in a loop is distinct. The following loop prints out [2,2] [3,3] [4,4] (and not [2,3])

  ```
  loop
     a:INT := range!(2,4);
     b:INT := range!(2,4);
  ```

```
   #OUT + "[" + a + "," + b + "]";
end;
```

- Not all iterators reach their end or quit - execution may be terminated because some other iterator in the same loop quit. See the next point.
- Iterator instances in a conditional statement are evaluated every time they are encountered. The following loop prints out [2,2] [3,2] [4,3] and then is terminated when the first iterator quits, even though the second iterator is not yet complete

```
b: INT := 0;
loop
   a:INT := range!(2,4);
   if a.is_even then
      b := range!(2,4);
   end;
   #OUT + "[" + a + "," + b + "]";
end;
```

- The expressions for `self` and for `once` arguments may not themselves contain iterator calls. (Such iter calls would not be useful anyway, since they would only execute their first iteration.) Thus, the following code is illegal, even though the 'times!' iterator is a perfectly valid iterator on integers.

```
loop
   a:INT := range!(3,4).times!;
end;
```

- Iterators may call themselves recursively as routines do. As iterators are normally supposed to yield more than once, one should not forget to define a loop within the iterator to catch all of these results

```
class BINARY_TREE is
   attr left,right:SAME; - subtrees
   attr data:INT;
   elt!:INT is
      if void(self) then
         quit;
      end;
      yield data;
      loop
         yield left.elt!;
      end; - yield data in the left subtree.
      loop
         yield right.elt!;
      end;
   end; - elt!
end; - class BINARY_TREE
```

- If an iterator in complex expression quits, the surrounding expression might not be fully evaluated.

```
loop
   #OUT + "(" + c.elt! + ")\n"
end;
```

When the iterator elt! terminates the surrounding loop, an opening bracket has already been printed. The expression producing the matching closing bracket will not be evaluated, hence the algorithm will always print a bogus closing bracket in the end. The standard solution looks as follows:

```
loop
   #OUT + ( "(" + c.elt! + ")\n" );
end;
```

The extra paratheses force the whole line to be evaluated first. As this evaluation will be aborted by the quit of the iterator the printing evaluation will not happen for the last iterator call.

Iterator definitions

- Iterator names always end with an exclamation mark '`!`'.
- Yield is not permitted within a protect statement (see the Chapter on Exceptions)
- Iterators enjoy the same access options as routines. Just as with routine definitions, iterator definintions may be marked private.
- Iterator overloading and conformance rules are the same as those for routines.
- An iter argument may have only one mode. Thus, it is neither possible nor meaningful to have '`once inout`' or '`once out`' arguments.

3.3. Iterator Examples

Some of the following examples make use of arrays, which will be introduced in the chapter on Parametrized Classes

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

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

Or, using the library iterator 'sum!' like this. 'x' needs to be declared (but not initialized) outside the loop, so its value is available after the loop terminates.

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

Some of the most common uses of iters are with container objects. Arrays, lists, sets, trees, strings, and vectors all have iterators to yield all their elements. Here we print all the elements of some container 'a'

```
a:ARRAY{INT} := |1,2,7|;
loop
   #OUT + a.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;
```

Separating elements of a list

When printing out the elements of a container, or other kinds of lists, it is usually appropriate to insert a separator between all the elements (but, of course, not after the last element). There is a convenient iterator in the string class that does exactly this:.

```
a:ARRAY{INT} := |1,2,3|;
loop
   #OUT + ",".separate!(a.elt!.str);
end;
- Prints out 1,2,3
```

The `separate!` iterator is called on the string which you wish to use to separate components of the list. In this case, the list elements will be separated by a comma. The definition of this iterator is as shown below

```
class STR is
   ...
   separate!(s: STR): STR is
      - On the first iteration just outputs 's', on later iterations
      - it outputs self followed by 's'.
      yield s.str;
      loop
         yield self + s.str;
      end;
   end;
   ...
end;
```

Note that the argument to the iterator is not a `once` argument, and will be evaluated each time the iterator is called.

Chapter 4. Code Inclusion and Partial Classes

Object oriented languages usually support the derivation of new classes by inheriting from existing classes and modifying them. In Sather, the notion of inheritance is split into two separate concepts - type relations between classes and code relations between classes. In this chapter we will deal with the latter (and simpler) concept, that of reusing the code of one class in another. We refer to this as implementation inheritance or code inclusion.

4.1. Include Clauses

The re-use of code from one class in another class is defined by *include clauses*. These cause the incorporation of the implementation of the specified class, 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.

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

Code inclusion permits the re-use of code from a parent concrete class in child concrete class . Including code is a purely syntactic operation in Sather. To help illustrate the following examples, we repeat the interface of `EMPLOYEE`.

```
class EMPLOYEE is
   private attr wage:INT;
   readonly attr name:STR;
   attr id:INT;
   const high_salary:INT := 40000;

   create(a_name:STR, a_id:INT, a_wage:INT):SAME is ...

   highly_paid:BOOL is ...
end;
```

Routines that are redefined in the child class over-ride the corresponding routines in the included class. For instance suppose we define a new kind of `EMPLOYEE` - a `MANAGER`, who has a number of subordinates.

```
class MANAGER is
   include EMPLOYEE
      create->private oldcreate;
   - Include employee code and rename create to 'oldcreate'
```

```
   readonly attr numsubordinates:INT; - Public attribute

   create(aname:STR, aid:INT,awage:INT,nsubs:INT):SAME is
      - Create a new manager with the name 'aname'
      - and the id 'aid' and number of subordinates = 'nsubs'
      res ::= oldcreate(aname,aid,awage);
      res.numsubordinates := nsubs;
      return res;
   end;
end;
```

See unnamedlink for the EMPLOYEE definition. The create routine of the `MANAGER` class extends the `EMPLOYEE` create routine, which has been renamed to oldcreate (renaming is explained below) and is called by the new create routine.

Points to Note

- The order of inclusion is not significant and cannot affect conflicts.
- External classes may be included if the interface to the language permits this; external Fortran and C classes may not be included.
- Immutable (see unnamedlink) and reference classes cannot be mixed into a single class during inclusion. In the case of arrays (see unnamedlink), there cannot be include paths from reference types to `AVAL` or from immutable types to `AREF` i.e. reference types cannot include an immutable array portion and immutable classes cannot include a reference array portion.
- There must be no cycle of classes such that each class includes the next, ignoring the values of any type parameters.

  ```
  class A is include B; ...
  class B is include C; ...
  class C is include A; ...
  ```

- If `SAME` occurs in included code it is interpreted as the eventual type of the class (as late as possible). We make use of this fact every time we include a create routine that returns `SAME`

  ```
  class  FOO is
     create:SAME is  return new; end;
  end;

  class SON_OF_FOO is
     include FOO;
     - Since create returns SAME, we have create:SON_OF_FOO;
  end;
  ```

```
class GRANDSON_OF_FOO is
   include SON_OF_FOO;   - Now we have create:GRANDSON_OF_FOO;
end;

a ::= #GRANDSON_OF_FOO; - Calls GRANDSON_OF_FOO::create:SAME,
       - which returns a GRANDSON_OF_FOO.
```

4.1.1. Renaming

The include clause may selectively rename some of the included features. It is also possible to include a class and make all routines private, or some selectively public

```
class MANAGER is
   private include EMPLOYEE;
   - All included features are made private

class MANAGER is
   private include EMPLOYEE  id->id;
   - Makes the "id" routine public and others stay private
```

If no clause follows the '->' symbol, then the named features are not included in the class. This is equivalent to 'undefining' the routine or attribute.

```
class MANAGER is
   include EMPLOYEE id->;  - Undefine the "id" routine
   attr id:MANAGER_ID;     - This ' id' has a different type
```

Points to note

- All overloaded features must be renamed at the same time - there is no way to specify them individually.
- A public routine can be made private by either a private include or by renaming the individual routine to be private

  ```
  class MANAGER is
     include EMPLOYEE id->private id;
     - Renames both reader and writer routines of the attribute 'id'
  ```

- In a private include, renaming a particular feature has the effect of making just that one feature public. For instance

  ```
  class MANAGER is
     private include EMPLOYEE
  ```

```
      name->name;           - only 'name' is made public
```

- Iterator names may only be renamed as iterator names.
- It is an error if there are no appropriate methods to rename in the included class.
- Both a reader and a writer method must exist if 'readonly' is used in a renaming clause.

```
class I_INTERVAL is
   private attr first, size:INT;

   finish:INT is
      return first + size - 1;
   end;

   finish(fin:INT) is
      size := fin - first + 1;
   end;
...

class LINE_SEGMENT is
   include I_INTERVAL
      finish->readonly finish;
   - makes private finish(fin:INT)
   - and leaves public finish:INT;
```

4.1.2. Multiple Inclusion

Sather permits inclusion of code from multiple source classes. The order of inclusion does not matter, but all conflicts between classes must be resolved by renaming. The example below shows a common idiom that is used in create routines to permit an including class to call the attribute initialization routines (by convention, this is frequently called 'init') of parent classes.

```
class PARENT1 is
   attr a:INT;

   create:SAME is
      return new.init;
   end;

   private init:SAME is
      a := 42;
      return self;
   end;
end;
```

In the above class, the attributes are initialized in the `init` routine. The use of such initialization routines is a good practice to avoid the problem of assigning attriutes to the "self" object in the create routine (which is void)

The other parent is similarly defined

```
class PARENT2 is
   attr c:INT;

   create:SAME is
      return new.init;
   end;

   private init:SAME is
      c := 72;
   end;
end;
```

In the child class, both parents are initialized by calling the initialization routines in the included classes

```
class DERIVED is
   include PARENT1 init-> PARENT_init;
   include PARENT2 init-> PARENT2_init; - Rename init

   attr b:INT;

   create:SAME is
       - a gets the value 42, b the value 99 and c the value 72
      return new.PARENT1_init.PARENT2_init.init
   end;

   private init:SAME is b := 99; return self;
end; - class DERIVED
```

4.1.3. Resolving conflicts

Two methods which are included from different classes may not be able to coexist in the same interface. They are said to conflict with each other. For a full discussion of resolving conflicts, please see unnamedlink. We have to first present the general overloading rule, before discussing when included signatures will conflict and what can then be done about it.

For now, we simply note that if we have signatures with the same name in two included classes, we can simply rename one of them away i.e.

```
class FOO is
```

```
   include BAR bar->;      - eliminate this 'bar' routine
   include BAR2;           - Use the 'bar' routine from BAR2
```

4.2. Partial Classes and Stub routines

Partial classes have no associated type and contain code that may only be included by other classes. Partial classes may not be instantiated: no routine calls from another class into a partial class are allowed, and no variables may be declared in another class of such a type.

A stub feature may only be present in a partial class. They have no body and are used to reserve a signature for redefinition by an including class. If code in a partial class contains calls to an unimplemented method, that method must be explicitly provided as a stub. The following class is a stub debugging class which checks on the value of a boolean and then prints out a debugging message (preceeding by the class name of 'self')

```
partial class DEBUG_MSG is
   stub debug:BOOL;

   debug_msg(msg:STR) is
   - Prints out the type of "self" and a debugging message
      if not debug then
         - Don't print out the message if the debug flag is false
         return;
      end;
      type_str:STR;
      - Declared here since used in both branches of the if
      if ~void(self) then
         type_id:INT := SYS::tp(self);
         - SYS::tp will not work if self is void!
         type_str:STR := SYS::str_for_tp(type_id);
      else
         type_str := "VOID SELF";
      end;
      #OUT + "Debug in class:" + type_str + " " + msg + "\n";
   end;
end;
```

This class can be used by some other class - for instance, a main routine that wants to print out all the arguments to main. The stub routine 'debug' must be filled in using either an attribute (a constant, in this case) or a routine.

```
class MAIN is
   include DEBUG_MSG;
```

```
   const debug:BOOL := true;           - Fill in the stub.

   main(args:ARRAY{STR}) is
      loop
         arg:STR := args.elt!
         debug_msg("Argument:"+arg);    - Print out the argument
      end;
   end;
end;
```

Points to note

- Partial classes cannot be used to instantiate parameters of a parametrized class. For example, 'ARRAY{DEBUG_MSG}' would not be legal.
- Create cannot be called on a partial class, nor can a partial class occur as the type of a variable or attribute.

4.2.1. Mixins: A Prompt Example

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

Now suppose that we have a 'COMPUTE' class that requires a prompt for some input data. It can leave the prompt routine as a stub, which will later be filled in by some prompt class

```
partial class COMPUTE is
   stub prompt_user:STR;

   main is
      res ::= prompt_user;
      - Convert it to an integer and do something with it
      i:INT := res.cursor.get_int;
      #OUT + "I'm going to compute with this number, now:" + i + "\n";
      ...
   end;
end; - partial class COMPUTE
```

We can now create different computation classes by mixing an arbitrary prompt style with the main computation partial class.

```
class COMPUTE_A is
   include COMPUTE;
   include PROMPT_STYLE_A;
end; – class COMPUTE_A

class COMPUTE_B is
   include COMPUTE;
   include PROMPT_STYLE_B;
end; – class COMPUTE_B
```

Chapter 5. Abstract Classes and Subtyping

Abstract class definitions specify interfaces without implementations. Abstract class names must be entirely uppercase and must begin with a dollar sign `'$'` ; 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. In order to motivate the notion of abstract classes, we will start by considering different implementations of a data structure.

5.1. Abstracting over Implementations

We will illustrate the need for abstraction by considering the implementation of a classic data structure, the stack. Objects are removed from a stack such that the last object to be inserted is the first to be removed (Last In First Out). For the sake of simplcity, we will define our stack to hold integers.

5.1.1. Implementing a Stack using an Array

The obvious implementation of a stack is using an array and a pointer to the top of the stack. When the stack outgrows the original array we allocate, we double the size of the array and copy the old elements over. This technique is known as amortized doubling and is an efficient way to allocate space for a datastructure whose size is not known when it is created.

```
class ARR_STACK is
   private attr elems:ARRAY{INT};
   private attr index:INT; - Points to the next location to insert

   create:SAME is
      res ::= new;
      res.elems := #ARRAY{INT}(5);
      res.index := 0;
      return res;
   end;

   push(e:INT) is
      if index > elems.size then
         new_elems:ARRAY{INT} := #ARRAY{INT}(index * 2);
         - copy over the old elements
         loop
            new_elems.set!(elems.elt!);
         end;
         elems := new_elems;
```

```
      end;
      elems[index] := e;
      index := index + 1;
   end;

   pop:INT is
      index := index - 1;
      return elems[index];
   end;

   is_empty:INT is
      return index = 0;
   end;
end;
```

It would be appropriate to also shrink the array when elements are popped from the stack, but we ignore this complexity for now.

5.1.2. A Stack Calculator

The stack class we defined can now be used in various applications. For instance, suppose we wish to create an calculator using the stack

```
class RPN_CALCULATOR is
   private attr stack:ARR_STACK;

   create:SAME is
      res ::=new;
      res.stack := #ARR_STACK;
      return res;
   end;

   push(e:INT) is
      stack.push(e);
   end;

   add:INT is
      - Add the two top two eleemnts
      if stack.is_empty then
         empty_err;
         return 0;
      end;
      arg1:INT := stack.pop;
      if stack.is_empty then
```

```
         empty_err;
         return 0;
      end;
      arg2:INT := stack.pop
      return arg1 + arg2;
   end;

   private empty_err is
      #ERR + "No operands available!"
   end;
end;
```

This corresponds to a H-P style reverse polish notation calculator (RPN) where you first enter operands and then an operator.

5.1.3. A Linked List Representation of a Stack

An alternative implementation of a stack might make use of a chain of elements i.e. a linked list representation. Each link in the chain has a pointer to the next element

```
class STACK_ELEM_HOLDER is
   readonly attr data:INT;
   attr next:INT_STACK_ELEM;

   create(data:INT):SAME is
      res ::= new;
      res.data := data;
      res.next := void;
      return res;
   end;
end;
```

The whole stack is then constructed using a chain of element holders.

```
class LINK_STACK is
   private attr head:STACK_ELEM_HOLDER;
   create:SAME is  res ::= new; return res; end;

   push(e:INT) is
      elem_holder ::= #STACK_ELEM_HOLDER(e);
      elem_holder.ext := head;
      head := elem_holder;
   end;

   pop:INT is
```

```
      res:INT := head.data;
      head := head.next;
   end;

   is_empty:BOOL is
      return void(head);
   end;
end;
```

5.1.4. Switching Representations:Polymorphism

Each of these stack implementations has advantages and disadvantages (the trade-offs are not very significant in our example, but can be quite considerable in other cases). Either of these stacks could be used in our calculator. To use the linked list stack we would need to replace `ARR_STACK` by by `LINK_STACK`. wherever it is used.

It would be nice to be able to write code such that we could transparently replace one kind of stack by the other. If we are to do this, we would need to be able to refer to them indirectly, through some interface which hides which particular implementation we are using. Interfaces of this sort are described by abstract classes in Sather. An abstract class that describes the stack abstraction is

```
abstract class $STACK is
   create:SAME;
   push(e:INT);
   pop:INT;
   is_empty:BOOL;
end;
```

Note that the interface just specifies the operations on the stack, and says nothing about how they are implemented. We have to then specify how our two implementations conform to this abstraction. This is indicated in the definition of our implementations. More details on this will follow in the sections below.

```
class ARR_STACK < $STACK is ... same definition as before ...

class LINK_STACK < $STACK is ... same definition as before ...
```

The calculator class can then be written as follows

```
class RPN_CALCULATOR is
   private attr stack:$STACK;
   create(s:$STACK):SAME is
      res ::= new;
      res.stack := s;
      return res;
   end;
```

```
   ... 'add' and 'push' behave the same
end;
```

In this modified calculator, we provide a stack of our choice when creating the calculator. Any implementation that conforms to our stack abstraction my be used in place of the array based stackt

```
s:LINK_STACK := #LINK_STACK;
calc:RPN_CALCULATOR := #RPN_CAlCULATOR(s);
calc.push(3);
calc.push(5);
#OUT + calc.add;          - Prints out 8
```

5.2. Abstract Class Definitions

The body of an abstract class definition consists of a semicolon separated list of 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.

```
abstract class $SHIPPING_CRATE is
   destination:$LOCATION;
   weight:FLT;
end; - abstract class $SHIPPING_CRATE
```

Due to the rules of subtyping, which will be introduced lateron (See unnamedlink), there is one restriction on the signatures - `SAME` is permitted only for a return type or out arguments in an abstract class signature.

Abstract types can never be created! Unlike concrete classes, they merely specify an interface to an object, not an object itself. All you can do with an abstract type is to declare a variable to be of that type. Such a variable can point to any actual object which is a subtype of that abstract class. How we determine what objects such an abstract variable can point to is the subject of the next section.

Note that we can, of course, provide a create routine in the abstract class

```
abstract class $SHIPPING_CRATE is
   create:SAME; ...
```

However, we can never call this creation routine on a void abstract class i.e. the following is prohibited

```
crate: $SHIPPING_CRATE := #$SHIPPING_CRATE; - ILLEGAL
```

In fact, all class calls (:: calls) are prohibited on abstract classes

```
f:FLT := $SHIPPING_CRATE::weight;  - ILLEGAL
```

Since abstract classes do not define objects, and do not contain shared attributes or constants, such calls on the class are not meaningful.

Example: An abstract employee

`$EMPLOYEE` illustrates an abstract type. `EMPLOYEE` and `MANAGER` are subtypes. Abstract type definitions specify interfaces without implementations.. Below, we will illustrate how the abstract type may be used.

```
abstract class $EMPLOYEE is
   - Definition of an abstract type.  Any concrete class that
   - subtypes from this abstract class must provide these routines.
   name:STR;
   id:INT;
end;
```

This abstract type definition merely states that any employee must have a name and an id.

More 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
   - Ensures that subtypes have a 'str' routine
   str:STR;   - Return a string form of the object
end;
```

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

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

5.3. Subtyping

As promised, here is the other half of inheritance, subtyping. A subtyping clause ('<' followed by type specifiers) indicates that the abstract signatures of all types listed in the subtyping clause are included in the interface of the type being defined. In the example, the subtyping clause is

```
abstract class $SHIPPING_CRATE < $CRATE is ...
```

The interface of an abstract type consists of any explicitly specified signatures along with those introduced by the subtyping clause.

Points to note about subtyping:

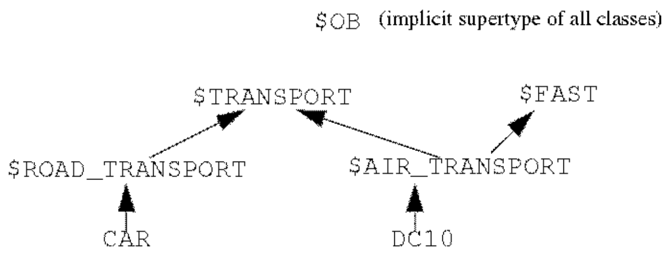
- Every type is automatically a subtype of `$OB`
- Only abstract types can be mentioned in the subtyping clause
- When a subtyping clause is used in a partial class, it enforces the basic subtyping rule using the stub routine.
- There must be no cycle of abstract types such that each appears in the subtype list of the next, ignoring the values of any type parameters but not their number.
- A subtyping clause may not refer to `SAME`.
- `SAME` is only permitted as a return type or on `out` arguments in abstract class signatures.

5.3.1. The Type Graph

We frequently refer to the Sather type graph, which is a graph whose nodes represent Sather types and whose edges represent subtyping relationships between sather types. Subtyping clauses introduce edges into the type graph. There is an edge in the type graph from each type in the subtyping clause to the type being defined. The type graph is acyclic, and may be viewed as a tree with cross edges (the root of the tree is `$OB`, which is an implicit supertype of all other types).

```
abstract class $TRANSPORT is ...
abstract class $FAST is ...
abstract class $ROAD_TRANSPORT < $TRANSPORT is ...
abstract class $AIR_TRANSPORT < $TRANSPORT, $FAST is ...
class CAR < $ROAD_TRANSPORT is ...
class DC10 < $AIR_TRANSPORT is ...
```

Since it is never possible to subtype from a concrete class (a reference, immutable or external class), these classes, CAR and DC10 form the leaf nodes of the type graph.



5.3.2. Dynamic Dispatch and Subtyping

Once we have introduced a typing relationship between a parent and a child class, we can use a variable of the type of the parent class to hold an object with the type of the child. Sather supports dynamic dispatch - when a function is called on a variable of an abstract type, it will be dispatched to the type of the object actually held by the variable. Thus, subtyping provides polymorphism.

An example: Generalizing Employees

To illustrate the use of dispatching, let us consider a system in which variables denote abstract employees which can be either `MANAGER` or `EMPLOYEE` objects. Recall the defintions of manager and employee

```
class EMPLOYEE < $EMPLOYEE is ...
   - Employee, as defined earlier

class MANAGER < $EMPLOYEE is ...
   - Manager as defined earlier
```

For the definition of these classes, see unnamedlink

The above defintions can then be used to write code that deals with any employee, regardless of whether it is a manager or not

```
class TESTEMPLOYEE is
   main is
      employees:ARRAY{$EMPLOYEE} := #ARRAY{$EMPLOYEE}(3);
      - employees is a 3 element array of employees
      i:INT := 0;
      wage:INT := 0;
      loop
         until!(i = employees.size);
         emp:$EMPLOYEE := employees[i];
         emp_wage:INT := emp.wage;
         - emp.wage is a dispatched call on "'age'
```

```
            wage := wage + emp_wage;
         end;
         #OUT + wage + "\n";
      end;
   end;
```

The main program shows that we can create an array that holds either regular employees or managers. We can then perform any action on this array that is applicable to both types of employees. The `wage` routine is said to be `dispatched`. At compile time, we don't know which wage routine will be called. At run time, the actual class of the object held by the `emp` variable is determined and the `wage` routine in that class is called.

5.4. Supertyping

Unlike most other object oriented languages, Sather also allows the programmer to introduce types above an existing class. A supertyping clause (`'>'` followed by type specifiers) adds to the type graph an edge from the type being defined to each type in the supertyping clause. These type specifiers may not be type parameters (though they may include type parameters as components) or external types. There must be no cycle of abstract classes such that each class appears in the supertype list of the next, ignoring the values of any type parameters but not their number. A supertyping clause may not refer to `SAME`.

If both subtyping and supertyping clauses are present, then each type in the supertyping list must be a subtype of each type in the subtyping list using only edges introduced by subtyping clauses. This ensures that the subtype relationship can be tested by examining only definitions reachable from the two types in question, and that errors of supertyping are localized.You define supertypes of already existing types. The supertype can only contain routines that are found in the subtype i.e. it cannot extend the interface of the subtype.

```
abstract class $IS_EMPTY > $LIST, $SET is
   is_empty:BOOL;
end;
```

5.4.1. Using supertyping

The main use of supertyping arises in defining appropriate type bounds for parametrized classes, and will be discussed in the next chapter (see unnamedlink).

5.5. Type Conformance

In order for a child class to legally subtype from a parent abstract class, we have to determine whether the signatures in the child class are consistent with the signatures in the parent class. The consistency check must ensure that in any code, if the parent class is replaced by the child class, the code would continue to work. This guarantee of substuitability which is guaranteed to be safe at compile time is at the heart of the Sather guarantee of type-safety.

5.5.1. Contravariant conformance

The type-safe rule for determining whether a signature in a child class is consistent with the definition of the signature in the parent class is referred to as the conformance rule[1]. The rule is quite simple, but counter-intuitive at first. Assume the simple abstract classes which we will use for argument types

```
abstract class $UPPER is ...
abstract class $MIDDLE < $UPPER is...
abstract class $LOWER < $MMIDDLE is ...
```

If we now have an abstract class with a signature

```
abstract class $SUPER is
   foo(a1:$MIDDLE, out a2:$MIDDLE, inout a3:$MIDDLE):$MIDDLE;
end;
```

What are the arguments types of `foo` in a subytpe of `$SUPER`? The rule says that in the subtype definition of foo

- Normal arguments (with the mode *in*) must have the same type or a supertypes
- `out` arguments and return values must have the same type or a subtype
- `inout` arguments must have the same type

Thus, a valid subtype of `$SUPER` is

```
abstract class $SUPER is
   foo(a1:$MIDDLE, out a2:$MIDDLE, inout a3:$MIDDLE):$MIDDLE;
end;
```

We will explain this rule and its ramifications using an extended example.

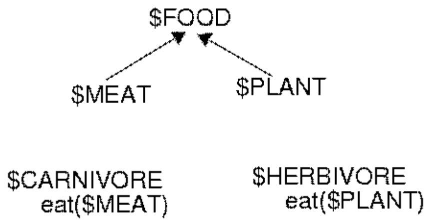
Suppose we start with herbivores and carnivores, each of which are capable of eating

```
abstract class $HERBIVORE is
   eat(food:$PLANT);
...

abstract class $CARNIVORE is
```

```
   eat(food:$MEAT);
...

abstract class $FOOD is ...
abstract class $PLANT < $FOOD is...
abstract class $MEAT < $FOOD is...
```



What does not work

It would appear that both herbivores and carnivores could be subtypes of omnivores.

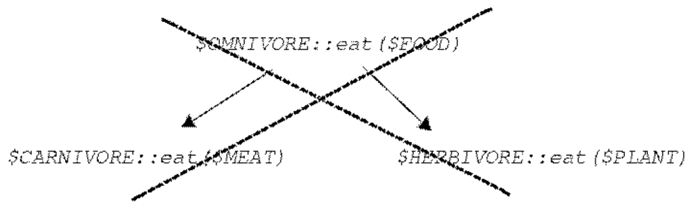
```
abstract class $OMNIVORE is eat(food:$FOOD);
abstract class $CARNIVORE < $OMNIVORE is ...
abstract class $HERBIVORE < $OMNIVORE is ...
```

However, *subtyping conformance will not permit this!* The argument to eat in `$HERBIVORE` is `$PLANT` which is not the same as or a supertype of `$FOOD`, the argument to eat in `$OMNIVORE`.

To illustrate this, consider a variable of type `$OMNIVORE,` which holds a herbivore.

```
cow:$HERBIVORE := #COW; - assigned to a COW object
animal:$OMNIVORE := cow;
meat:$MEAT;
animal.eat(meat);
```

This last call would try to feed the animal meat, which is quite legal according to the signature of `$OMNIVORE::eat($FOOD)`, since `$MEAT` is a subtype of `$FOOD`. However, the animal happens to be a cow, which is a herbivore and cannot eat meat.



What does work

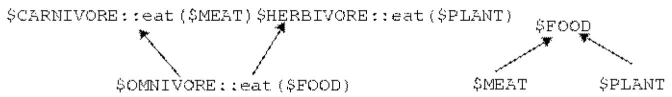
When contravariance does not permit a subtyping relationship this is usually an indication of an exceptional case or an error in our conceptual understanding. In this case, we note that omnivores are creatures that can eat anything. But a herbivore really is not an omnivore, since it cannot eat anything. More importantly, a herbivore could not be substuted for an omnivore. It is, however, true that an omnivore can act as both a carnivore and a herbivore.

```
abstract class $CARNIVORE is
   eat(food:$MEAT);
...

abstract class $HERBIVORE is
   eat(food:$PLANT);
...

abstract class $OMNIVORE < $HERBIVORE, $CARNIVORE is
   eat(food:$FOOD);
...
```

The argument of eat in the omnivore is `$FOOD`, which is a supertype of `$MEAT`, the argument of eat in `$CARNIVORE`. `$FOOD` is also a supertype of `$PLANT` which is the argument of eat in `$HERBIVORE`.



5.5.2. Subtyping = substitutability

A key distinction is that between is-a and as-a relationships. When a class, say `$OMNIVORE` subtypes from another class such as `$CARNIVORE`, it means that an omnivore can be used in any code which deals with carnivores i.e. an omnivore can substitute for a carnivore. In order for this to work properly, the child class omnivore must be able to behave *as-a* carnivore. In many cases, an is-a relationship does not satisfy the constraints required by the as-a relationship. The contravariant conformance rule captures the necessary as-a relationship between a subtype and a supertype.

5.6. The `typecase` statement

It is sometimes necessary to bypass the abstraction and make use of information about the actual type of the object to perform a particular action. Given a variable of an abstract type, we might like to make use of the actual type of the object it refers to, in order to determine whether it either has a particular implementation or supports other abstractions.

The `typecase` statement provides us with the ability to make use of the actual type of an object held by a variable of an abstract type.

```
a:$OB := 5;
... some other code...
res:STR;

typecase a
when INT then                 - 'a' is of type INT in this branch
   #OUT + "Integer result: " + a;
when FLT then                 - 'a' is of type FLT in this branch
   #OUT + "Real result: " + a;
when $STR then                - 'a' is $STR and supports '.str'
   #OUT + "Other printable result: " + a.str;
else
   #OUT + "Non printable result";
end;
```

The typecase must act on a local variable or an argument of a method.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`.

Points to note

- 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.
- 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. In the above example, the declared type of `a` is `$OB` , which does not match any of the branches, so the `else` clause is taken
- The variable of the typecase must be a local variable or a method argument.
- 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.
- The typecase does not search for the branch with the tightest match - it goes down the first branch that matches.

- After a branch has been selected, the typecase tries to cast the variable to as narrow a type as possible - if the declared type of the variable is actually stronger than (a subtype of) the chosen branch, then the variable will keep the stronger, declared type. For instance

  ```
  a:$SET{INT};
  typecase a
  when INT then ... - a will never get here, INT is not < $SET{INT}
  when $OB then ...
     - a has the type of $SET{INT} which is stronger than $OB
  ```

Typecase Example

For instance, suppose we want to know the total number of subordinates in an array of general employees.

```
peter ::= #EMPLOYEE("Peter",1);      - Name = "Peter", id = 1
paul  ::= #MANAGER("Paul",12,10);    - id = 12,10 subordinates
mary  ::= #MANAGER("Mary",15,11);    - id = 15,11 subordinates
employees: ARRAY{$EMPLOYEE} := |peter,paul,mary|;
totalsubs: INT := 0;
loop employee:$EMPLOYEE := employees.elt!; - yields array elements
   typecase employee
   when MANAGER then
      totalsubs := totalsubs + employee.numsubordinates;
   else
   end;
end;
#OUT + "Number of subordinates: " + totalsubs + "\n";
```

Within each branch of the typecase, the variable has the type of that branch (or a more restrictive type, if the declared type of the variable is a subtype of the type of that branch).

5.7. The Overloading Rule

We mentioned an abridged form of the overloading rule in the chapter on Classes and Objects. That simple overloading rule was very limited - it only permitted overloading based on the number of arguments and the presence or absence of a return value. Here, it is generalized.

As a preliminary warning:the overloading are flexible, but are intended to support the coexistance of multiple functions that *have the same meaning, but differ in some implementation detail.* Calling functions that do different things by the same name is wrong, unwholesome and severely frowned upon! Hence, using the function name times with different number of arguments to mean

5.7.1. Extending Overloading

Overloading based on Concrete Argument Types

However, we often want to overload a function based on the actual type of the arguments. For instance, it is common to want to define addition routines (`plus`) that work for different types of values. In the `INT` class, we could define

```
plus(a:INT):INT is ...
plus(a:FLT):INT is ...
```

We can clearly overload based on a the type of the argument if it is a non-abstract class - at the point of the call, the argument can match only one of the overloaded signatures.

Overloading based on Abstract Argument Types

Extending the rule to handle abstract types is not quite as simple. To illustrate the problem, let us first introduce the `$STR` abstract class

```
abstract class $STR is
   str:STR;
end;
```

The `$STR` absraction indicates that subtypes provide a routine that renders a string version of themselves. Thus, all the common basic types such as `INT`, `BOOL` etc. are subtypes of `$STR` and provide a `str: STR` routine that returns a string representation of themselves.

Now consider the interface to the `FILE` class. In the file class we would like to have a general purpose routine that appends any old `$STR` object, by calling the str routine on it and then appending the resulting string. This allows us to append any subtype of `$STR` to a file at the cost of a run-time dispatch. We also want to define more efficient, special case routines (that avoid the dispatched call to the str routine) for common classes, such as integers

```
class FILE is
   - Standard output class
   plus(s:$STR) is ...   - (1)
   plus(s:INT) is ...    - (2)
end;
```

The problem arises at the point of call

```
f:FILE := FILE::open_for_read("myfile");
a:INT := 3;
f+a;
```

Now which plus routine should we invoke? Clearly, both routines are valid, since `INT` is a subtype of `$STR`. We want the *strongest* or *most specific* among the matching methods, (2) in the example above. Though the notion of the most specific routine may be clear in this case, it can easily get murky when there are more arguments and the type graph is more complex.

The Demon of Ambiguity

It is not difficult to construct cases where there is no single most specific routine. The following example is hypotheical and not from the current Sather library, but illustrates the point. Suppose we had an abstraction for classes that can render a binary versions of themselves. This might be useful, for instance, for the floating point classes, where a binary representation may be more compact and reliable than a decimal string version

```
abstract class $BINARY_PRINTABLE is
   - Subtypes can provide a binary version of themselves
   binary_str:STR;
end;
```

Now suppose we have the following interface to the `FILE` class

```
class FILE is
   plus(s:$STR) is ...         - (1)
   plus(s:$BINARY_STR) is ... - (2)
   plus(s:INT) is ...         - (3)
end;
```

Now certain classes, such as `FLT` could subtype from `$BINARY_STR` instead of from `$STR`. Thus, in the following example, second plus routine would be seletected

```
f:FILE;
f + 3.0;
```

Everything is still fine, but suppose we now consider

```
class FLTD < $BINARY_STR, $STR is
   binary_str:STR is ... - binary version
   str:STR is ...        - decimal version
```

The plus routine in `FILE` cannot be unambiguously called with an argument of type `FLTD` i.e. a call like ’`f+3.0d`’ is ambiguous. None of the ’plus’ routines match exactly; (1) and (2) both match equally well.

The above problem arises because neither (1) nor (2) is more specific than the other - the problem could be solved if we could always impose some ordering on the overloaded methods, such that there is a most specific method for any call.

We could resolve the above problem by ruling the `FILE` class to be illegal, since there is a common subtype to both `$STR` and `$BINARY_STR`, namely `FLTD`. Thus, a possible rule would be that overloading based on abstract arguments is permitted, provided that the abstract types involved have no subtypes in common.

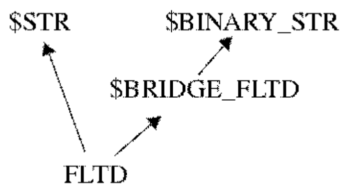
However, the problem is somewhat worse than this in Sather, since both subtyping and supertyping edges can be introduced after the fact. Thus, if we have the following definition of `FLTD`

```
class FLTD < $BINARY_STR is
   binary_str:STR is ...
   str:STR is ...
```

the file class will work. However, at a later point, a user can introduce new edges that cause the same ambiguity described above to reappear!

```
abstract class $BRIDGE_FLTD < $STR > FLTD is
end;
```

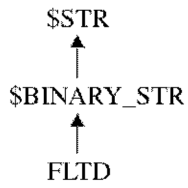
Adding this new class introduces an additional edge into the type graph and breaks existing code.



The essense of the full-fledged overloading rule avoids this problem by requiring that the type of the argument in one of the routines must be known to be more specific than the type of the argument in the corresponding position in the other routine. Insisting that a subtyping relationship between corresponding arguments must exist, effectively ensures that one of the methods will be more specific in any given context. Most importantly, this specificity cannot be affected by the addition of new edges to the type graph. Thus, the following definition of `$BINARY_STR` would permit the overloading in the `FILE` class to work properly

```
abstract class $BINARY_STR < $STR is
   binary_str:STR;
end;
```

When the `'plus'` routine is called with a `FLTD`, the routine `'plus($BINARY_STR)'` is unambiguously more specific than `'plus($STR)'`.



5.7.2. Permissible overloading

Two signatures (of routines or iterators) can overload, if they can be distinguised in some manner- thus, they must differ in one of the following ways

Overload 1.: The presence/absence of a return value

Overload 2.: The number of arguments

Overload 3.: In at least one case corresponding arguments must have different marked modes (in and once modes are not marked at the point of call and are treated as being the same from the point of view of overloading).

Overload 4.: In at least one of the `in, once` or `inout` argument positions: (*a*) both types are concrete and different or (*b*) there is a subtyping relationship between the corresponding arguments i.e. one must be more specific than the other. Note that this subtyping ordering between the two arguments cannot be changed by other additions to the type graph, so that working libraries cannot be broken by adding new code.

Note that this definition of permissible permissible coexistance is the converse of the definition of conflict in the specification. That is, if two signatures cannot coexist, they conflict and vice-versa.

```
abstract class $VEC is ...
abstract class $SPARSE_VEC < $VEC is ...
abstract class $DENSE_VEC < $VEC is...

class DENSE_VEC < $DENSE_VEC is ...
class SPARSE_VEC < $SPARSE_VEC is ....
```

Given the above definitions of vectors, we can define a multiply and add routine in the matrix class

```
abstract class $MATRIX is
- (1)
   mul_add(by1:$VEC, add1:$SPARSE_VEC);

- (2)
   mul_add(by2:$DENSE_VEC,  add2:$VEC);
   - (1) and (2) can overload, since the arg types can be ordered
   - by2:$DENSE_VEC < by1:$VEC,
```

```
   - add2:$VEC       > add1:$SPARSE_VEC

- (3)
   mul_add(by3:DENSE_VEC, add3:SPARSE_VEC);
   - (3) does not conflict with the (1) and (2) because there
   -     is a subtyping relation between corresponding arguments.
   - (vs 1) by3:DENSE_VEC    < by1:$VEC ,
   -        add3:SPARSE_VEC < add1:$SPARSE_VEC
   - (vs 2) by3:DENSE_VEC    < by2:$DENSE_VEC ,
   -        add3:SPARSE_VEC < add2:$VEC
end;
```

While any of the above conditions ensures that a pair of routines can co-exist in an interface, it still does not describe which one will be chosen during a call.

Finding matching signatures

When the time comes to make a call, some of the coexisting routines will match - these are the routines whose arguments are supertypes of the argument types in the call. Among these matching signatures, there must be a single most specific signature. In the example below, we will abuse sather notation slightly to demonstrate the types directly, rather than using variables of those types in the arguments

```
f:$MATRIX;
f.mul_add(DENSE_VEC, SPARSE_VEC);      - Matches (1), (2) and (3)
f.mul_add($DENSE_VEC, $SPARSE_VEC);   - Matches (1) and (2)
f.mul_add($DENSE_VEC, $DENSE_VEC);    - Matches (2)
f.mul_add($SPARSE_VEC, SPARSE_VEC);   - Matches (1)
```

Finding a most specific matching signature

For the method call to work, the call must now find an unique signature which is most specific in each argument position

```
f:$MATRIX;
f.mul_add(DENSE_VEC, SPARSE_VEC)      - (3) is most specific
f.mul_add($DENSE_VEC, $DENSE_VEC);    - Only one match
f.mul_add($SPARSE_VEC, $SPARSE_VEC);  - Only one match
```

The method call `'f.mul_add($DENSE_VEC, $SPARSE_VEC)'` is illegal, since both (1) and (2) match, but neither is more specific.

More examples

Let us illustrate overloading with some more examples. Consider `'foo(a:A, out b:B);'`

All the following can co-exist with the above signature

```
foo(a:A, out b:B):INT    - Presence return value (Overload 1)
foo(a:A)                 - Number of arguments (Overload 2)
foo(a:A, b:B)            - Mode of second argument (Overload 3)
foo(a:B, out b:B)        - Different concrete types in
         - the first argument (Overload 4a)
```

The following cannot be overloaded with `foo(a:A,out b:B):INT;`

```
foo(a:A,b:B):BOOL;  - Same number, types of arguments,
                    - both have a return type.
- Difference in actual return type cannot be used to overload
```

For another example, this time using abstract classes, consider the mathematical abstraction of a ring over numbers and integers. The following can be overloaded with the `'plus'` function in a class which describes the mathematical notion of rings

```
abstract class $RING is
   plus(arg:$RING):$RING;
...

abstract class $INT  < $RING is
   plus(arg:$INT):$RING;
   -  By Overload 4 since he type of  arg:$INT  < arg:$RING
...

abstract class $CPX < $RING is
   plus(arg:$CPX):$RING;
   - By Overload 4b, since the type of arg:$CPX < arg:$RING
...
```

The overloading works because there is a subtyping relationship between the arguments 'arg' to 'plus' The following overloading also works

```
abstract class $RING is
   mul_add(ring_arg1:$RING, ring_arg2:$RING);
...

abstract class $INT < $RING is
   mul_add(int_arg1:$INT, int_arg2:$INT);
   - int_arg1:$INT  < ring_arg:$INT and
   - int_arg2:$INT < ring_arg2:$INT
```

```
...
```

Now there is a subtyping relationship between `$INT::mul_add` and `$RING::mul_add` for both `'arg1'` and `'arg2'`, but there is no subtyping

This somewhat complex rule permits interesting kinds of overloading that are needed to implement a kind of statically resolved, type-safe co-variance which is useful in the libraries, while not sacrificing compositionality. Externally introducing subtyping or supertyping edges into the typegraph cannot suddenly break overloading in a library.

5.7.3. Overloading as Statically resolved Multi-Methods

For the curious reader, we would like to point out a connection to the issue of co and contra-variance. It was this connection that actually motivated our overloading rules. The first point to note is that overloading is essentially like statically resolved multi-methods i.e. methods that can dispatch on more than one argument. Overloaded methods are far more restricted than multi-methods since the declared type must be used to perform the resolution. The second point to note is that multi-methods can permit safe 'covariance' of argument types. For instance, consider the following abstractions

```
abstract class $FIELD_ELEMENT is
   add(f:$FIELD_ELEMENT):$FIELD_ELEMENT;
...

abstract class $NUMBER < $FIELD_ELEMENT is
   add(f:$NUMBER):$NUMBER
...

abstract class $INTEGER < $NUMBER is
   add(f:$INTEGER):$INGEGER
...
```

Note that all the above definitions of the `'plus'` routines safely overload each other. As a consequence, it is possible to provide more specific versions of functions in sub-types.

5.7.4. Conflicts when subtyping

When we described subtyping earlier, we said that the interface of the abstract class being defined is augmented by all the signatures of the types in the subtyping clause. But what if some of these supertypes contain conflicting signatures?

It is important to note that a conflict occurs when two signatures are so similar that they cannot co-exist by the over-loading rules. This happens when there is not even one argument where there is a sub- or supertyping relationship or where both arguments are concrete. As a consequence, you can always construct a signature that is more general than the conflicting signatures

```
abstract class $ANIMAL is ...
abstract class $PIG < $ANIMAL is ...
abstract class $COW < $ANIMAL is ...
abstract class $COW_FARM is
   has(a:$COW);
...

abstract class $PIG_FARM is
   has(a:$PIG);
...

abstract class $ANIMAL_FARM < $COW_FARM, $PIG_FARM is
   - The signatures for has(a:$COW) and has(a:$PIG) must
   - be generalized
   has(a:$ANIMAL);
   - $ANIMAL is a supertype of $COW and $PIG, so this 'has'
   - conforms to both the supertype 'has' signatures
...
```

In the above example, when we create a more general farm, we must provide a signature that conforms to all the conflicting signatures by generalizing the in arguments. If the arguments in the parent used the `out` mode, we would have to use a subtype in the child. A problem is exposed if the mode of the arguments in the parents is `inout`

```
abstract class $COW_FARM is
   processes(inout a:$COW);
end;

abstract class $PIG_FARM is
   processes(inout a:$PIG);
end;

- ILLEGAL! abstract class $ANIMAL_FARM < $COW_FARM, $PIG_FARM is
- No signature can conform to both the 'processes' signatures
- in the $COW_FARM and $PIG_FARM
```

5.7.5. Conflicts during code inclusion

Since Sather permits inclusion from mulitple classes, conflicts can easily arise between methods from different classes. The resolution of inclusion conflicts is slightly different for attributes than it is for methods, so let us consider them separately.

Conflicting Methods

1. First, let us consider the resolution method for routines. Conflicts can occur between methods in different classes that have been included and must be resolved by renaming the offending feature in all but one of the included classes:

```
class PARENT1 is
   foo(INT):INT;
...

class PARENT2 is
   foo(INT):BOOL; - conflicts with PARENT1::foo
...

class PARENT3 is
   foo(INT):FLT;  - would similarly conflict
...

class CHILD is
   include PARENT1 foo -> parent1_foo;
   - Include and rename away the routine 'foo'
   include PARENT2 foo -> parent2_foo;
   - Include and rename away the routine 'foo'
   include PARENT3;
   - Use the routine from this class
...
```

2. The other way to resolve method conflicts is to explicitly define a method in the child class that will then over-ride all the parent methods.

```
class CHILD is
   include PARENT1;
   include PARENT2;
   include PARENT3;

   foo(INT):BOOL is
   - over-rides all the included, conflicting routines.
end;
```

Conflicting Attributes

With conflicting attributes (including shareds and consts), the offending attributes must be renamed away, even if they are going to be replaced by other attributes i.e. Method 2 described above is not allowed for attributes:

```
class PARENT is
   attr foo:INT;
...

class CHILD is
   foo:BOOL;   - ILLEGAL!
 - Conflicts with the included reader for 'foo' i.e. foo:INT
...
```

Also the implicit reader and writer routines of attributes defined in the child must not conflict with routines in a parent

```
class PARENT is
   foo(arg:INT);
...

class CHILD is
   include PARENT;
   - ILLEGAL! attr foo:INT;
   - the writer routine foo(INT) conflicts
   - with the writer for the include attribute foo(INT)
...
```

In other words, as far as attributes are concerned, they must always be explicitly renamed away - they are never silently over-ridden.

5.7.6. Points to note

- It is not possible to overload based based solely on `out` or `inout` arguments (by the pre-condition for applying the overload rule 4a and 4b)
- When a class explicitly defines a signature and includes a conflicting signature from another class, the included signature is over-ridden. This might lead to included signatures unexpectedly disappearing, instead of overloading.
- In certain special cases, subtyping from two classes with conflicting signatures that use `out` or `inout` arguments might not be possible, since the conflict cannot be resolved.

5.7.7. Overloading in Parametrized Classes

For details on the overloading rule for parametrized classes, see unnamedlink.

5.7.8. Why not use the return type to resolve conflicts?

According to the current overloading rules, the type of the return value and `out` arguments cannot be used to differentiate between methods in the interface. There is no theoretical reason to disallow this possibility. However permitting overloading based on such return values involves significant implementation work and was not needed for the usages we envisaged. Thus, overloading is not permitted based on differences in the return type (or `out` arguments, which are equivalent to return types) of a method

5.8. When Covariance Ails You

In some cases, however, one type can substitute for the other type but with a few exceptions. There are several ways to deal with this problem when it occurs.

[This section attempts to provide some insight into dealing with covariance. It is not essential to understanding the language, but might help in the design of your type hierarchy.]

5.8.1. But don't animals eat food?

We will consider the definition of an animal class, where both herbivores and carnivores are animals.

```
abstract class $ANIMAL is
   eat(food:$FOOD);
...

abstract class $HERBIVORE < $ANIMAL is ...
abstract class $CARNIVORE < $ANIMAL is ...
```

The problem is similar to that in the previous section, but is different in certain ways that lead to the need for different solutions

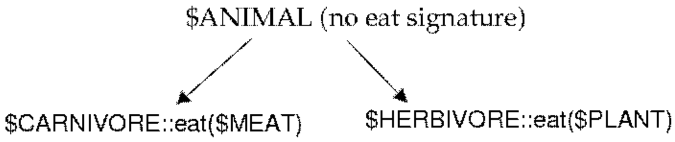
5.8.2. Solution 1: Refactor the type hierarchy

The ideal solution would be to do what we did in the previous section - realize the conceptual problem and rearrange the type hierarchy to be more accurate. There is a difference in this case, though. When considering omnivores, the 'eat' operation was central to the definition of the subtyping relationship. In the case of animals, the eat operation is not nearly as central - the subtyping relationship is determined by many other features, completely unrelated to eating. It would be unreasonable to force animals to be subtypes of carnivores or herbivores.

5.8.3. Solution 2: Eliminate the offending method

A simple solution would be to determine whether we really need the ’eat’ routine in the animal class. In human categories, it appears that higher level categories often contain features that are present, but vary greatly in the sub-categories. The feature in the higher level category is not "operational" in the sense that it is never used directly with the higher level category. It merely denotes the presence of the feature in all sub-categories.

Since we do not know the kind of food a general animal can eat, it may be reasonable to just omit the ’eat’ signature from the definition of $ANIMAL. We would thus have:



5.8.4. Solution 3: Dynamically Determine the Type

Another solution, that should be adopted with care, is to permit the ’eat($FOOD)’ routine in the animal class, and define the subclasses to also eat any food. However, each subclass dynamically determines whether it wants to eat a particular kind of food.

```
abstract class $ANIMAL is
   eat(arg:$FOOD);
...

abstract class $HERBIVORE < $ANIMAL is ... - supports eat(f:$FOOD);

class COW < $HERBIVORE is
   eat(arg:$FOOD) is
      typecase arg
      when $PLANT then ... - eat it!
      else raise "Cows only eat plants!"; end;
   end;
end;
```

The ’eat’ routine in the COW class accepts all food, but then dynamically determines whether the food is appropriate i.e. whether it is a plant.

This approach carries the danger that if a cow is fed some non-plant food, the error may only be discovered at run-time, when the routine is actually called. Furthermore, such errors may be discovered after an arbitrarily long time, when the incorrect call to the 'eat' routine actually occurs during execution.

This loss of static type-safety is inherent in languages that support co-variance, such as Eiffel. The problem can be somewhat ameliorated though the use of type-inference, but there will always be cases where type-inference cannot prove that a certain call is type-safe.

Sather permits the user to break type-safety, but only through the use of a typecase on the arguments. Such case of type un-safety uses are clearly visible in the code and are far from the default in user code.

5.8.5. Solution 4: Parametrize by the Argument Type

Another typesafe solution is to parametrize the animal abstraction by the kind of food the animal eats.

Notes

1. Frequently called the contravariant conformance rule to distinguish it from the more restrictive C++ rule of invariance and the unsafe Eiffel rule (of covariance in the argument types). Hence, the co- vs. contra variance debate just refers to the behavior of the argument types.

Chapter 6. Parametrized Classes and Arrays

All Sather classes may be parametrized by one or more type parameters. Type parameters are essentially placeholders for actual types; the actual type is only known when the class is actually used. The array class, which we have already seen, is an example of a parametrized class.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.

6.1. Parametrized concrete types

As an example of a parametrized class, consider the class PAIR, which can hold two objects of arbitrary types. We will refer to the types as T1 and T2:

```
class PAIR{T1,T2} is
   readonly attr first:T1;
   readonly attr second:T2;

   create(a_first:T1, a_second:T2):SAME is
      res ::= new;
      res.first := a_first;
      res.second := a_second;
      return res;
   end;
end;
```

We can use this class to hold a pair of integers or a pair of an integer and a real etc.

```
c ::= #PAIR{INT,INT}(5,5);      - Holds a  pair of integers
d ::= #PAIR{INT,FLT}(5,5.0);    - Holds an integer and a FLT
e ::= #PAIR{STR,INT}("this",5);- A string and an integer
f:INT := e.second;
g:FLT := d.second;
```

Thus, instead of defining a new class for each different type of pair, we can just parametrize the PAIR class with different parameters.

6.1.1. Why Parametrize?

Parametrization is normally presented as a mechanism for achieving efficiency by specializing code to use particular types. However, parametrization plays an even more important conceptual role in a language with strong typing like Sather.

For instance, we could define a pair to hold $OBs

```
class OB_PAIR is
   readonly attr first,second:$OB;

   create(a_first, a_second:$OB):SAME is
      res ::= new;
      res.first := a_first;
      res.second := a_second;
      return res;
   end;
end; - class OB_PAIR
```

There is no problem with defining OB_PAIR objects; in fact, it looks a little simpler.

```
c ::= #OB_PAIR(5,5);      - Holds a  pair of integers
d ::= #OB_PAIR(5,5.0);    - Holds an integer and a FLT
```

However, when the time comes to extract the components of the pair, we are in trouble:

```
- f:INT := e.second; ILLEGAL! second is declared to be a $OB
```

We can typecase on the return value:

```
f_ob:$OB := e.second;
f:INT;
typecase f_ob
when INT then f := f_ob;
end;
```

The above code has the desired effect, but is extremely cumbersome. Imagine if you had to do this every time you removed an INT from an ARRAY{INT}! Note that the above code would raise an error if the branch in the typecase does not match.

The parametrized version of the pair container gets around all these problems by essentially annotating the type of the container with the types of the objects it contains; the types of the contained objects are the type parameter.

6.2. Support for Arrays

Arrays (and, in fact, most container classes) are realized using parametrized classes in Sather. There is language support for the main array class `ARRAY{T}` in the form of a literal expressions of the form

```
a:ARRAY{INT} := |1,2,3|;
```

In addition to the standard accessing function, arrays provide many operations, ranging from trivial routines that return the size of the array to routines that will sort arbitrary arrays. See the array class in the container library for more details. There are several aspects to supporting arrays:

- Support for accessing array elements
- Support for objects which represent arrays
- Support for initializing these arrays using literals

6.2.1. Array Access

The form '`a[4]:=..`' 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. In the class TRIO below we have three elements which can be accessed using array notation

```
class TRIO is
   private attr a,b,c:FLT;

   create:SAME is
      return new
   end;

   aget(i:INT):FLT is
      case i
      when 0 then return a
      when 1 then return b
      when 2 then return c
      else raise "Bad array index!\n";
      end;
   end;

   aset(i:INT, val:FLT) is
      case i
      when 0 then a := val;
      when 1 then b := val;
      when 2 then c := val;
      end;
   end;
```

```
end;
```

The array notation can then be used with objects of type TRIO

```
trio:TRIO := #TRIO;  - Calls TRIO::create
trio[2] := 1;
#OUT + trio[2];        - Prints out 1
```

See unnamedlink for more details on the section on operator redefinition.

6.2.2. Array Classes: Including *AREF* and calling *new()*;

Sather permits the user to define array classes which support an array portion whose size is determined when the array is created. An object can have an array portion by including `AREF{T}`.

```
class POLYGON is
   private include AREF{POINT}
      aget->private old_aget, aset->private old_aset;
         - Rename aget and aset

   create(n_points:INT):SAME  is
      - Create a new polygon with a 'n_points' points
      res:SAME := new(n_points);  - Note that the new takes
      - as argument of the size of the array
   end;

   aget(i:INT):POINT is
      if i > asize then raise "Not enough polygon points!" end;
      return old_aget(i);
   end;

   aset(i:INT, val:POINT) is
      if i > asize then raise "Not enough polygon points!" end;
      old_aset(i,val);
   end;
end;
```

Since `AREF{T}` already defines `'aget'` and `'aset'` to do the right thing, we can provide wrappers around these routines to, for instance, provide an additional warning message. The above example makes use of the `POINT` class from unnamedlink. We could have also used the `PAIR` class defined on unnamedlink. The following example uses the polygon class to define a triangle.

```
poly:POLYGON := #POLYGON(3);
poly[0] := #POINT(3,4);
poly[1] := #POINT(5,6);
```

```
poly[2] := #POINT(0,0);
```

AREF defines several useful routines:

```
asize:INT                       - Returns the size of the array
aelt!:T;                        - Yields successive array elements
aelt!(once beg:INT):T;          - Yields elements from index 'beg'
aelt!(once beg,once num:INT):T;  - Yields 'num' elts from index 'beg'
aelt!(once beg,once num,once step:INT):T;
   - Yields 'num' elements, starting at index 'beg' with a 'step'
... Analgous versions of aset! ..
acopy(src:SAME);                 - Copy what fits from 'src' to self
acopy(beg:INT,src:SAME);         - Start copying into index 'beg'
acopy(beg:INT,num:INT,src:SAME);
   - Copy 'num' elements into self starting at index 'beg' of self
aind!:INT;                        - Yields successive array indices
```

When possible, use the above iterators since they are built-in and can be more efficient than other iterators.

6.2.3. Standard Arrays: ARRAY{T}

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;
```

The array portion appears if there is an include path from the type to AREF for reference types or to AVAL for immutable types.

Array Literals

Sather provides support for directly creating arrays from literal expressions

```
a:ARRAY{INT} := |2,4,6,8|;
b:ARRAY{STR} := |"apple","orange"|;
```

.

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
- as a method argument in which the overloading resolution is ambiguous
- as the left argument of the dot '`.`' operator.

  ```
  a:INT := |1,2,3|.size - ILLEGAL
  ```

The types of each expression in the array literal must be subtypes of `T`. The size of the created array is equal to the number of specified expressions. The expressions in the literal are evaluated left to right and the results are assigned to successive array elements.

6.2.4. Multi-dimensional Arrays

Special support is neither present nor needed for multi-dimensional arrays. The `'aget'` and `'aset'` routines can take multiple arguments, thus permitting multiple indices. The library does provide `ARRAY2` and `ARRAY3` classes, which provide the necesary index computation. All standard array classes are addressed in row-major order. However, the MAT class is addressed in column major order for compatibility with external FORTRAN routines[1]. Multi-dimensonal array literals may be expressed by nesting of standard array literals

```
a:ARRAY{ARRAY{INT}} := ||1,2,3|,|3,4,5|,|5,6,7||;
```

6.3. Type Bounds

When writing more complex parametrized classes, it is frequently useful to be able to perform operations on variables which are of the type of the parameter. For instance, in writing a sorting algorithm for arrays, you might want to make use of the "less than" operator on the array elements.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 this allows type-safe independent compilation.

For our example, we will return to employees and managers. Recall that the employee abstraction was defined as:

```
abstract class $EMPLOYEE is
   name:STR;
   id:INT;
end;
```

We can now build a container class that holds employees. The container class makes use of a standard library class, a LIST, which is also parametrized over the types of things being held.

```
class EMPLOYEE_REGISTER{ETP < $EMPLOYEE} is
   private attr emps:LIST{ETP};

   create:SAME is
      res ::= new;
      res.emps := #;
      return res;
   end;

   add_employee(e:ETP) is
      emps.append(e);
   end;

   n_employees:INT is
      return emps.size;
   end;

   longest_name:INT is
    - Return the length of the longest employee name
      i:INT := 0;
      cur_longest:INT := 0;
      loop
         until!(i=n_employees);
         employee:ETP := emps[i];
         name:STR := employee.name;
         - The type-bound has the ".name" routine
         if name.size > cur_longest then
            cur_longest := name.size;
         end;
      end;
      return cur_longest;
   end;
```

```
end;
```

The routine of interest is "longest_name". The use of this routine is not important, but we can imagine that such a routine might be useful in formatting some printout of employee data. In this routine we go through all employees in the list, and for each employee we look at the "name". With the typebound on `ETP`, we know that `ETP` must be a subtype of `$EMPLOYEE`. Hence, it must have a routine "name" which returns a `STR`.

If we did not have the typebound (there is an implicit typebound of `$OB`), we could not do anything with the resulting "employee"; all we could assume is that it was a `$OB`, which is not very useful.

6.3.1. Why have typebounds?

The purpose of the type bound is to permit type checking of a parametrized class over all possible instantiations. Note that the current compiler does not do this, thus permitting some possibly illegal code to go unchecked until an instantiation is attempted.

6.3.2. Supertyping and Type Bounds

The need for supertyping clauses arises from our defintition of type-bounds in parametrized types. The parameters can only be instantiated by subtypes of their type bounds.

You may, however, wish to create a parametrized type which is instantiated with classes from an existing library which are not under the typebound you require. For instance, suppose you want to create a class PRINTABLE_ SET, whose parameters must support both hash and the standard string printing routine `str`. The library contains the following abstract classes.

```
abstract class $HASH < $IS_EQ is
   hash:INT;
end;

abstract class $STR is
   str:STR;
end;
```

However, our PRINTABLE_SET{T} must take all kinds of objects that support both $HASH and $STR, such as integers, floating point numbers etc. How do we support this, without modifying the distributed library?

```
abstract class $HASH_AND_STR > INT, FLT, STR is
   hash:INT;
   str:STR;
end;
```

```
class PRINTABLE_SET{T < $HASH_AND_STR} is
   - Set whose elements can be printed

   str:STR is
      res:STR := "";
      loop
         res := res + ",".separate!(elt!.str);
      end;
      return res;
   end;
end;
```

The PRINTABLE_SET class can now be instantiated using integers, floating point numbers and strings. Thus, supertyping provides a way of creating supertypes without modifying the original classes (which is not possible if the original types are in a different library).

Note that this is only useful if the original classes cannot be modified. In general, it is usually far simpler and easier to understand if standard subtyping is used.

A more complicated example arises if we want to create a sorted set, whose elements must be hashable and comparable. From the library we have.

```
abstract class $HASH < $IS_EQ is
   hash:INT;
end;

abstract class $IS_LT{T} < $IS_EQ is  - comparable values
   is_lt(elt:T):BOOL;
end;
```

However, our SORTABLE_SET{T} must only take objects that support both $HASH and $IS_LT{T}

```
abstract class $ORDERED_HASH{T} < $HASH, $IS_LT{T} is
end;

class ORDERED_SET{T < $ORDERED_HASH{T}} is
   - Set whose elements can be sorted

   sort is
      - ... uses the < routine on elements which are of type T
   end;
```

The above definition works in a straightforward way for user classes. For instance, a POINT class as defined below, can be used in a ORDERED_SET{POINT}

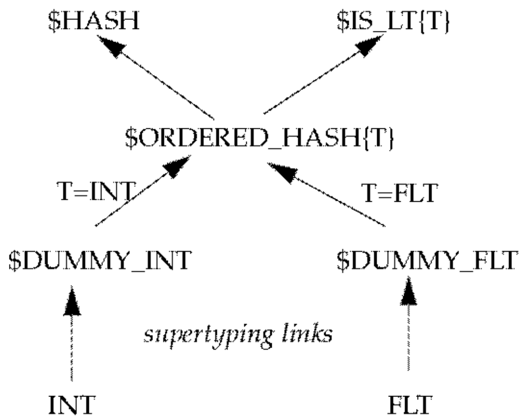
```
class POINT < $ORDERED_HASH{POINT} is ...
```

```
- define hash:INT and is_lt(POINT):BOOL
```

But how can you create an ordered set of integers, for instance? The solution is somewhat laborious. You have to create dummy classes that specify the subtyping link for each different parametrization of $ORDERED_HASH

```
abstract class $DUMMY_INT > INT < $ORDERED_HASH{INT} is end;
abstract class $DUMMY_STR > STR < $ORDERED_HASH{STR} is end;
abstract class $DUMMY_FLT > FLT < $ORDERED_HASH{FLT} is end;
```

Note that the above classes are only needed because we are not directly modifying INT and FLT to subtype from $ORDRED_HASH{T}. In the following diagram , recall that since there is no relationship between different class parametrizations, it is necessary to think of them as separate types.



6.4. Parametrized Abstract Classes

Abstract class definitions may be *parameterized* by one or more type parameters within enclosing braces; in the example, the type parameter is 'T'. There is no implicit type relationship between different parametrizations of an abstract class. 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

Sather abstract classes may be similarly parametrized by any number of type parameters. Each type parameter may have an optional type bound; this forces any actual parameter to be a subtype of the corresponding type bound. Given the following definitions,

```
abstract class $A{T < $BAR} is
   foo(b:T):T;
end; - abstract class $A{T}

abstract $BAR is
end;

class BAR < $BAR is
end;
```

we may then instantiate an abstract variable a:$A{BAR}. BAR instantiates the parameter T and hence must be under the type bound for T, namely$BAR. If a type-bound is not specified then a type bound of $OB is assumed.

How are different parametrizations related?

It is sometimes natural to want a `$LIST{MY_FOO} < $LIST{$MY_FOO}`. Sather, however, specifies no subtyping relationship between various parametrizations. Permitting such implicit subtyping relationships between different parametrizations of a class can lead to type safety violations.

6.5. Overloading

There are two aspects to the use of overloading in a parametrized class - one aspect is the behavior of the interface of the parametrized class itself, and the other aspect is calls within the parametrized class where one or more arguments have the type of one of the type parameters, or is related to the type parameters through static type inference (see .

6.5.1. Overloading In the Parametrized Class Interface

Argument with the type of a class parameter cannot be used to resolve overloading (such an argument is similar to an ’out’ argument or a return type in this respect).

```
class FOO{T1<$STR ,T2<$ELT} is
   bar(a:T1);  - (1)
   bar(a:T2);  - (2)
```

Even though the type bounds for T1 and T2 are distinct and one is more specific than the other, this is not a sufficient constraint on the actual instantiation of the parameter. In a class such as

FOO{ARRAY{INT}, ARRAY{INT}}

for instance, the two versions of 'bar' will essentially be identical.

6.5.2. Overloading Resolution within the Parametrized Class

Note: The current ICSI compiler does not yet have this behaviour implemented. In the current compiler, overloading resolution is based on the actual instantiated class.

For all calls within the parametrized class, the resolution of overloading is done with respect to the type bounds of the parameters. Consider a class that makes use of output streams

```
abstract class $OSTREAM is
   plus(s:$STR);
end;
```

A parametrized class can then write to any output stream

```
class FOO{S < $OSTREAM} is
   attr x,y:INT;

   describe(s:S) is
      s + "Self is:";
      s + x;
      s + ",";
      s + y;
   end;
end;
```

Now, suppose we instantiate the class `FOO` with a `FILE`

```
class FILE < $OSTREAM is
   plus(s:$STR) is ... - (1)
   plus(s:INT) is ...  - (2)

a:FOO{FILE} := ..
f:FILE := FILE::open_for_read("myfile");
a.describe(f)
```

Only `'(1) plus($STR)'` will be called in `FOO{FILE}`, even though the more specific `'(2) plus(INT)'` is available in `FILE`.

The reason for this behavior is to preseve the ability to analyze a stand alone class, which is needed for separate compilation of parametrized classes - this requires that the behavior of the parametrized class be completely determined by the typebounds and not based on the existance of specialized overloaded routines in particular instantiations.

Notes

1. Efficiency in converting to FORTRAN was more important for mathematical entitites which will be used with existing mathematical libraries such as BLAS and LAPACK, most of which are in FORTRAN.

Chapter 7. Operator Redefinition

7.1. Method Names for Operators

It is possible to define operators such as + and * to work with objects of arbitrary classes. These operators are transformed into standard routine calls in the class. Thus, if a class defines the routine 'plus' you can then apply the + operator to objects from that class. For instance, the complex number class POINT could define the plus routine to mean pairwise addition

```
class POINT is
   readonly attr x,y:INT;
   create(x,y:INT):SAME is ... - same as before
   plus(s:POINT):POINT is
      return #POINT(x + s.x, y + s.y);
   end;
```

we can now use the plus routine on two points

```
p1:POINT := #POINT(3,5);
p2:POINT := #POINT(4,6);
p3:POINT := p1 + p2;  -  p3 is set to the point 7,11
```

Most of the standard operators may be redefined; in some cases, redefining one operator such as the < operator implicitly redefines the associated >, >= and <= operators. These operators are meant to be used together in a consistent manner to indicate the mathematical notion of complete or partial ordering. They are not intended to be used as a convenient short-hand for other purposes.

7.2. Operator expressions

The following table shows how the standard operators are directly converted into routine calls.

```
Operator          Routine
------------------------
expr1 + expr2     expr1.plus(expr2)
expr1 - expr2     expr1.minus(expr2)
expr1 * expr2     expr1.times(expr2)
expr1 / expr2     expr1.div(expr2)
expr1 ^ expr2     expr1.pow(expr2)
expr1 % expr2     expr1.mod(expr2)
expr1 < expr2     expr1.is_lt(expr2)
expr1 = expr2     expr1.is_eq(expr2)
```

```
- expr            expr.negate
~ expr            expr.not
```

In addition to the unary and binary operators, there are additional operators that are defined in terms of a combination of the unary and binary operators

```
Operator         Translation
--------------------------
expr1 <= expr2   expr2.is_lt(expr1).not
expr1 >= expr2   expr1.is_lt(expr2).not
expr1 /= expr2   expr1.is_eq(expr2).not
expr1 > expr2    expr2.is_lt(expr1)
```

1

The form `'[expression list]'` is translated into a call on the routine `aget.` For instance,

```
a := [3,5];   - Equivalent to a := aget(3,5); Used in the array class
f := arr[2];  - Equivalent to f := arr.aget(2); Used outside the array
```

This is described in more detail later.

Grouping

In addition to the above mentioned operators, it is possible to group expressions using plain parentheses, which have the highest precedence.

7.2.1. Operator precedence

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
```

Points to note

- The >, >= and /= operators are not directly translated into their own routine. Rather, they are defined in terms of is_lt and is_eq.
- Each of these transformations is applied after the component expressions have themselves been transformed.
- 'out' and 'inout' modes may not be used with the syntactic sugar expressions.
- The '<=' and '>' expressions do not reverse the original left to right order of argument evaluation.
- and' and or' are not listed as syntactic sugar for operations in BOOL'; this allows short-circuiting the evaluation of subexpression.
- The aget and aset routines are meant to support array like indexed access and require at least one index argument.

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
```

7.3. Array Access Routines

Sather supports the standard array access syntax of square brackets. For instance:

```
a:ARRAY{INT} := |1,2,3|;
a[2] := 5;             - Sets the third element of the array to 5
#OUT + a[0];           - Prints out '1'
c:ARRAY2{INT} := ||1,2,3|,|4,5,6|,|7,8,9||;
#OUT + c[2,2];         - Prints out '9'
```

However, the array bracket notation is not built into the array class. It is just a short hand for the routines aget and aset

```
a[2] := 5;   - equivalent to a.aset(2,5);
#OUT + a[1]; - equivalent to #OUT+a.aget(1);
```

Thus, classes which are not arrays can make use of the array notation as they please:

```
class INT is
   - The standard integer class
   aget(i:INT):BOOL is - returns the 'i'th bit of the integer
end;
```

In order for a class to actually have an array portion, it must inherit from `AREF{T}` (if it is a reference class) or `AVAL{T}` if it is a value class. The array setting notation is not as useful for immutable classes, since any modification of an immutable class must return a whole new object.

Notes

1. Earlier versions of Sather 1.0 defined separate routines for each of these operators.

Chapter 8. Immutable Classes

Sather has special support for classes that define immutable objects. Such objects cannot be modified after they have been created, and are said to have value semantics. Many of the basic types such as integers and floating point numbers (the INT and FLT classes) are implemented using immutable classes. This chapter illustrates how immutable classes may be defined, and highlights the peculiarities in their usage that may trip up a beginning user.

At a fundamental level: immutable classes define objects which, once created, never change their value. A variable of an immutable type may only be changed by re-assigning to that variable. When we wish to only modify some portion of an immutable class, we are compelled to reassign the whole object. For experienced C programmers the difference between immutable and reference classes is similar to the difference between structs (immutable types) and pointers to structs (reference types). Because of that difference, reference objects can be referred to from more than one variable (aliased), while immutable objects cannot.

This section illustrates the definition of immutable types using a simple version of the complex number class, CPX. We also describe the benefits of immutable classes and when they should be used. Finally, we close with a description of a how to transparently replace in immutable class by a standard reference class which implements value semantics.

8.1. Defining Immutable Classes

In most ways, defining and using immutable classes is similar to defining and using reference classes. Immutable classes consist of a collection of attributes and functions that can operate on the attributes. Since we have already described reference classes in considerable detail, we will describe immutable classes in terms of their differences from reference classes.

8.1.1. Immutable Class Example

We illustrate the use of immutable classes through the example of the complex class CPX. The version shown here is a much simplified version of the library class. The key point to note is the manner in which attribute values are set in the create routine.

```
immutable class CPX is
   readonly attr real,imag:FLT;

   create(re,im:FLT):SAME is
      - Returns a complex number with real and imaginary parts set
      res:SAME;
```

```
      res := res.real(re);
      res := res.im(im);
      return res;
   end;

   plus(c:SAME):SAME is
      - Return a complex number, the sum of 'self' and c'.
      return #SAME(real+c.real,imag+c.imag);
   end;
end; - immutable class CPX
```

The complex class may then be used in the following manner.

```
b:CPX := #(2.0,3.0);
d:CPX := #(4.0,5.0);
c:CPX := b+d;
```

8.1.2. Creating a new object

Unlike reference classes, instances of an immutable class are not explicitly allocated using the 'new' expression. A variable of an immutable class always has a value associated with it, from the point of declaration. In the example above, the return variable of the create routine ,'res' simply has to be declared.

8.1.3. Initial value of immutable objects

The initial value of an immutable object is defined to have all its fields set to the 'void' value and this is defined to be the 'void' value of the immutable object. Note that this 'void' value means something different than it does for a reference class. It does not mean that the object does not exist, but rather that all its fields have the 'void' value.

Void value of the basic classes:

```
Class      Initial Value
----------------
INT        0
CHAR       '\0'
FLT        0.0
FLTD       0.0d
BOOL       false
```

The initial values for the built-in immutable classes are defined above. These values will return true for the ’void’ test.

8.1.4. Attribute access routines

Since an immutable object cannot change its value, what does assigning to an attribute mean? Sather’s immutable classes define attribute assignment to create a copy of the original object, with the attribute modified. Thus the attribute declaration ’attr re:FLT ’ of the CPX class has an implicit attribute setting routine with the signature:

```
re(new_re_part:FLT):SAME;
```

which returns a copy of the original CPX object in which the attribute ’re’ has the new value ’new_re_value’. Contrast this with a reference class, in which the setting routine would have the signature

```
re(new_re_part:FLT);
```

The syntax of the setting routines of immutable classes is a common source of confusion.

8.1.5. Points to note

- There must be no cycle of immutable types such that each type has an attribute whose type is in the cycle. In the following example, the class PAIR has a FIRST_PART that contains a PAIR - leading to an infinite loop and an infinite size structure.

  ```
  immutable class PAIR is
     attr first:FIRST_PART;
     attr second:SECOND_PART;
  ...

  immutable class FIRST_PART is
     attr begin:PAIR;
  ...
  ```

- Accessing an attribute of a void immutable object will always work. Accessing an attribute of a void reference object results in a fatal error
- The ’void’ value for the basic classes are useful values - false is the ’void’ value for the BOOL class, and 0 for the number classes.

8.2. Using Immutable Classes

Immutable classes behave differently from reference classes both in terms of their abstract behaviour (value semantics) and in terms of their implementation.

To begin with, immutable classes cannot suffer from aliasing problems, since they are immutable. You can get the same effect with reference classes by not providing any modifying operations in the interface - any operation that would modify the object, returns a new object instead. For example, take a look at the STR class

Immutable classes may have several efficiency advantages over reference classes in certain circumstances. Since they are usually stored on the stack, they have no heap management overhead and need not be garbage collected. They also don't use space to store a tag, and the absence of aliasing makes more C compiler optimizations possible. For a small class like CPX, all these factors combine to give a significant win over a reference class implementation. On the other hand, copying large immutable objects onto the stack can incur significant overhead. Unfortunately the efficiency of an immutable class appears directly tied to how smart the C compiler is; "gcc" is not very bright in this respect.

Note that when an immutable class is passed as an argument to a function which is expecting an abstract type, the compiler boxes it i.e. it is given a temporary reference class wrapper with a type-tag. Thus, immutable objects behaves exactly like an immutable reference objects in this situation.

Rules of Thumb

So, when should you use an immutable class? Here are a few rules of thumb.

- You want the class to have immutable semantics. You could still consider an immutable reference class.
- The class is small - the exact speed trade-offs have not been investigated, but immutable classes have so far been used when there are a fewer than a handful of attributes.
- There are going to be a large number of objects of that class. This goes along with the previous point. For instance, if you are going to have large arrays of complex numbers, then the space that would be required for an object pointer and an object tag may be considerable.

Chapter 9. Closures

Routine and iter *closures* are similar to the 'function pointer' and 'closure' constructs of other languages. They bind a reference to a method together with zero or more argument values (possibly including `self`). The type of a closure begins with the keywords `ROUT` or `ITER` and followed by the modes and types of the underscore arguments, if any, enclosed in braces (*e.g.* '`ROUT{A, out B, inout C}`', '`ITER{once A, out B, C}`'). These are followed by a colon and the return type, if there is one (*e.g.* '`ROUT{INT}:INT`', '`ITER{once INT}:FLT`').

9.1. Creating and Calling Closures

9.1.1. Creating a closure

A closure is created by an expression that binds a routine or an iterator, along with some of its arguments. The outer part of the expression is '`bind(...)`'. This surrounds a routine or iterator call in which any of the arguments or `self` may have been replaced by the underscore character '_'. Such unspecified arguments are *unbound*. Unbound arguments are specified when the closure is eventually called.

```
a:ROUT{INT}:INT := bind(3.plus(_))
b:ITER:INT := bind(3.times!);
```

Out and inout arguments must be specified in the closure type. If the routine has `inout` or `out` arguments

```
swap(inout x, inout y:INT) is
   tmp ::= x;
   x := y;
   y := tmp;
end;
```

as show below, they are mentioned in the type of the closure:

The routine '`swap`' swaps the values of the two arguments, '`x`' and '`y`'. '`r`' is a closure for binding the '`swap`' routine.

```
r:ROUT{inout INT, inout INT} := bind(swap(_,_));
```

9.1.2. Calling a closure

Each routine closure defines a routine named '`call`' and each iterator closure defines an iterator named '`call!`'. These have argument and return types that correspond to the closure type specifiers.

Invocations of these features behave like a call on the original routine or iterator with the arguments specified by a combination of the bound values and those provided to `call` or `call!`. The arguments to `call` and `call!` match the underscores positionally from left to right .

The previously defined closures are invoked as shown

```
#OUT + a.call(4);  - Prints out 7, where a is bind(3.plus(_)
sum:INT := 0;
loop
   sum := sum + b.call!;
end;
#OUT + sum;        - Prints out 3 (0+1+2)
```

In the following example, we define a bound routine that takes an `INT` as an argument and returns an `INT`.

```
br:ROUT{INT}:INT := bind(1.plus(_));
#OUT + br.call(9);      - Prints out '10'
```

The variable br is typed as a bound routine which takes an integer as argument and returns an integer. The routine 1.plus, which is of the appropriate type, is then assigned to br. The routine associated with br may then be invoked by the built in function `call`. Just as we would when calling the routine `INT::plus(INT)`, we must supply the integer argument to the bound routine.

9.1.3. Binding overloaded routines

When binding a routine which is overloaded, there might be some ambiguity about which routine is meant to be bound

```
class FLT is
   plus(f:FLT):FLT  - add self and 'i' and return the result
   plus(i:INT):FLT; - add self and 'f' (after converting 'i' to FLT)
end;
```

When binding the `plus` routine, it might not be obvious which routine is intended

```
b ::= bind(_.plus(_));
```

In case of ambiguity, the right method must be determined by the context in which the binding takes place.

Binding in an assignment

If there is ambiguity about which method is to be bound, the type of the variable must be explicitly specified

```
b:ROUT{FLT,FLT}:FLT := bind(_.plus(_)); - Selects the first 'plus'
```

Binding in a call

A method may also be bound at the time a call is made. The type of the closure is determined by the type of the argument in the call.

```
reduce(a:ARRAY{FLT}, br:ROUT{FLT,FLT}:FLT):FLT is
   res:FLT := 0.0;
   loop
      el:FLT := a.elt!;
      res := br.call(res,el);
   end;
   return res;
end;
```

We can call the reduction function as follows:

```
a:ARRAY{FLT} := |1.0,7.0,3.0|;
#OUT + reduce(a,bind(_.plus(_)));
- Prints '11.0', the sum of the elements of 'a'
```

The second argument to the function `reduce` expects a `ROUT{FLT,FLT}:FLT` and this type was used to select which `plus` routine should be bound. When there could be doubt about which routine is actually being bound, it is very good practice to specify the type explicitly

```
r:ROUT{FLT,FLT}:FLT := bind(_.plus(_));
#OUT + reduce(a,r);
```

9.1.4. Points to note

- `out` and `inout` arguments must be left unbound. This is a reasonable restriction, since such arguments must return a value to the calling context. If such an argument were bound, when the closure is invoked, variables that existed at the point of closure binding would be affected. Such variables might not even be alive at the point where the closure is actually invoked.

9.1.5. Binding some arguments

When a routine closure is created, it can preset some of the values of the arguments.

```
class MAIN is
   foo(a:INT, b:INT):INT is
```

```
      return(a+b+10);
   end;

   main is
      br1:ROUT{INT,INT}:INT := bind(foo(_,_));
      br2:ROUT{INT}:INT := bind(foo(10,_));
      #OUT + br1.call(4,3) + "," + br2.call(9);
            - Should print 17 and  29
   end;
end;
```

In the example above, `br2` binds the first argument of `foo` to 10 and the second argument is left unbound. This second argument will have to be supplied by the caller of the bound routine. `br1` binds neither argument and hence when it is called, it must supply both arguments.

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

```
r :ROUT{INT}:INT := bind(2.times(_));
loop
   a.set!(r.call(a.elt!))
end
```

9.1.6. Leaving *self* unbound

bound routines are often used to apply a function to arbitrary objects of a particular class. For this usage, we need the self argument to be unbound. This illustrates how self may be left unbound. The type of self must be inferred from the type context (`ROUT{INT}`).

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

In the following example we will make use of the `plus` routine from the `INT` class.

```
... from the INT class
plus(arg:INT):INT is
... definition of plus

main is
   plusbr1:ROUT{INT,INT}:INT := bind(_.plus(_)); - self and arg unbound
   br1res:INT := plusbr1.call(9,10);             - Returns 19
   plusbr2:ROUT{INT}:INT := bind(3.plus(_));     - Binding self only
   br2res:INT := plusbr2.call(15);               - Returns 18
   plusbr3:ROUT{INT}:INT := bind(_.plus(9));     - Binding arg only
   br3res:INT := plusbr3.call(11);               - Returns 20
   #OUT + br1res + "," + br2res + "," + br3res;
```

```
              - 19,18,20
end;
```

In the above example, `plusbr1` leaves both self and the argument to `plus` unbound. Note that we must specify the type of self when creating the bound routine, otherwise the compiler cannot know which class the routine belongs to (the type could also be an abstract type that defines that feature in its interface). `plusbr2` binds self to `3`, so that the only argument that need be supplied at call time is the argument to the plus. `plusbr3` binds the argument of `plus` to `15`, so that the only argument that need be supplied at call time is self for the routine.

9.2. Further Examples of Closures

Just as is the case with C function pointers, there will be programmers who find closures indispensible and others who will hardly ever touch them. Since Sather's closures are strongly typed, much of the insecurity associated with function pointers in C disappears.

9.2.1. Closures for Applicative Programming

Closures are useful when you want to write Lisp-like "apply" routines in a class which contains other data . Routines that use routine closures in this way may be found in the class ARRAY{T}. Some examples of which are shown below.

```
every(test:ROUT{T}:BOOL):BOOL is
   - True if every element of self satisfies 'test'.
   loop
      e ::= elt!;   - Iterate through the array elements
      if ~test.call(e) then
         return false;
      end
      - If e fails the test, return false immediately
   end;
   return true
end;
```

The following routine which takes a routine closure as an argument and uses it to select an element from a list

```
select(e:ARRAY{INT}, r:ROUT{INT}:BOOL):INT is
   - Return the index of the first element in the array 'e' that
   - satisfies the predicate 'r'.
   - Return -1 if no element of 'e' satisfies the predicate.
   loop i:INT := e.ind!;
```

```
      if r.call(e[i]) then
         return i;
      end;
   end;
   return -1;
end;
```

The selection routine may be used as shown below:

```
a:ARRAY{INT} := |1,2,3,7|;
br:ROUT{INT}:BOOL := bind(_.is_eq(3));
#OUT + select(a,br);  - Prints the index of the first element of 'a'
                      - that is equal to '3'. The index printed is '2'
```

9.2.2. Menu Structures

Another common use of function pointers is in the construction of an abstraction for a set of choices. The `MENU` class shown below maintains a mapping between strings and routine closures associated with the strings.

```
class MENU is
   private attr menu_actions:MAP{STR,ROUT};
        - Hash table from strings to closures
   private attr default_action:ROUT{STR};

   create(default_act:ROUT{STR}):SAME is
      res:SAME := new;
      res.menu_actions := #MAP{STR,ROUT};
      res.default_action := default_act;
      return(res)
   end;

   add_item(name:STR, func:ROUT) is menu_actions[name] := func end;
     - Add a menu item to the hash table, indexed by 'name'

   run is
      loop
         #OUT + ">";
         command: STR := IN::get_str;  - Gets the next line of  input
         if command = "done" then
            break!;
         elsif menu_actions.has_ind(command) then
            menu_actions[command].call;
```

```
         else
            default_action.call(command);
         end;
      end;
   end;
end;
```

We use this opportunity to create a textual interface for the calculator described earlier (See unnamedlink):

```
class CALCULATOR is
   private attr stack:A_STACK{INT};
   private attr menu:MENU;

   create:SAME is
      res ::= new;
      res.init;
      return res;
   end;

   private init is   - Initialize the calculator attributes
      stack := #;
      menu := #MENU(bind(push(_)));
      menu.add_menu_item("add",bind(add));
      menu.add_menu_item("times",bind(times));
   end;

   run is menu.run; end;

...

-- Now, the main routines of the calculator computation are:

   push(s:STR) is
      - Convert the value 's' into an INT and push it onto the stack
      - Do nothing if the string is not a valid integer
      c: STR_CURSOR := s.cursor;
      i: INT := c.int;
      if c.has_error then
         #ERR + "Bad integer value:" + s;
      else
         stack.push(i);
      end;
   end;
```

```
   add is  - Add the two top stack values and push/print the result
      sum:INT := stack.pop + stack.pop;
      #OUT + sum+"\n";
      stack.push(sum);
   end;

   times is  - Multiply the top stack values and push/print the result
      product:INT := stack.pop * stack.pop;
      #OUT + product + "\n";
      stack.push(product);
   end;
end;  - class CALCULATOR
```

This calculator can be started by a simple main routine

```
class MAIN is
   main is
      c:CALCULATOR := #;
      c.run;
   end;
end;
```

:

After compiling the program, we can then run the resulting executable

```
prompt> a.out
>3
>4
>add
7
>10
>11
>times
110
>done
prompt>
```

9.2.3. Iterator closures

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

```
select: ROUT{T}:BOOL;
select_elt: ITER{MAP{E,T}}:T;
```

```
...
select_elt := bind(_.filter!(select));
```

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

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

Chapter 10. Exceptions

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

10.1. Throwing and Catching Exceptions

There are two aspects to indicating errors using exceptions - how the error is indicated at the point where it occurs. This is usually referred to as throwing the exception. The other aspect of exceptions is how the error message is handled, which is referred to as catching the exception.

10.1.1. Throwing Exceptions with *raise*

Exceptions are explicitly raised by *raise statements*. The raise statement specifies an expression, which is evaluated to obtain the exception object.

```
add_if_positive(i:INT) is
   if i < 0 then
      raise "Negative value:" + i + "\n";
   end;
end;
```

In the example above, the object happens to be a string that indicates the problem. In general, the exception object must provide enough information for the error handling mechanism. Since the error handling mechanism can discriminate between different objects of different types, it is standard practice to use the type of the exception object to indicate the type of the error that occurred.

10.1.2. Catching Exceptions with *protect*

Exceptions are passed to higher contexts until a handler is found and the exception is caught. Exceptions are caught using protect statements. The protect statement surrounds a piece of code, and provides an appropriate method of handling any exceptions that might occur when executing that piece of code.

```
protect
   foo;
when $STR then
   #ERR + "An error in foo!:" + exception.str;
```

```
when INT then
   #ERR + "INT error=" + exception; - 'exception' of type INT
else
   - Some other error handling
end;
```

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.

In the protect clause, the exception raised may be referred to by the built in expression 'exception'[1], which refers to the exception object. The type of the exception object can be used to categorize the exception and to discriminate between exceptions when they are actually caught. In fact, the when clauses may be viewed as a typecase (see unnamedlink) on the exception object.

Points to note

- No statements may follow a raise statement in a statement list because they can never be executed.
- If there is no else clause in a protect statement, and none of the types in the when branches matches the type of the exception object, then the exception is passed to the next higher protect statement

10.1.3. Usage to avoid

Exceptions can be significantly slower than ordinary routine calls, so they should be avoided except for truly exceptional (unexpected) cases. Using exceptions to implement normal control flow may be tempting, but should be avoided. For instance, in the STR_CURSOR class, we can make use of exceptions for parsing. It might be tempting to write code like the following

```
test_bool:BOOL is
   protect
      current_state ::= save_state;
      b ::= get_bool;
      restore_state(current_state);
   when STR_CURSOR_EX then
      return(false);
   end;
   return(true);
end;
```

The above code determines whether a boolean is present in the string by trying to read one and treating an error state as evidence that there is no boolean. While it is perfectly correct code, this is an example of what you should not do. The implementation of a function should not rely on exceptions for its normal functioning. Doing so is extremely inefficient and can result in an unnecessarily complicated flow of control.

10.1.4. Alternatives to Exceptions

The alternative to using exceptions is to use a sticky error flag in the class, as is done by IEEE exceptions and the current `FILE` classes. This has problems such as the fact that the outermost error is logged, not the most immediate one, and it is very easy to forget to test for the error. However, this method has a much lower overhead and is suitable in certain cases.

10.2. A more elaborate example

Consider the following routine, which tries to read a boolean value from a string:

```
get_bool(file_name:STR):BOOL is
   f:FILE := FILE::open_for_read(file_name);
   if f.error then
      raise #FILE_OPEN_EXC(file_name);
   end;
   s:STR := f.str;  - Read the file into a string
   f.close;         - Close the file
   res:BOOL;
   bool ::= "";
   i:INT := 0;
   loop
      until!(~(s[i].is_alpha) or (s[i].is_space) or i >= s.size);
      bool := bool + s[i];
      i := i + 1;
   end;
   case bool
   when "true","t","True","T","TRUE" then
      return true;
   when "false","f","False","F","FALSE" then
      return false;
   else
      raise #PARSE_BAD_BOOL_EXC(s);
   end;
end;
```

In the above routine there are two possible errors - either the file could not be opened or it does not contain a valid boolean. The two cases can be distinguised at the point when the exception is caught

```
protect
   file_name:STR; ... set to a value
   b:BOOL := get_bool(s);
when FILE_OPEN_EXC then
   #ERR + "Could not open:" + exception.file_name + "\n";
when PARSE_BAD_BOOL_EXC then
   #ERR + "Error in reading boolean:" + exception.str + "\n";
end;
```

The classes that implement these exceptions can be fairly simple

```
class FILE_OPEN_EXC is
   readonly attr str:STR:
   create(file_name:STR):SAME is
      res ::= new;
      res.str := file_name;
      return res;
   end;
end;
```

The other exception class is very similar.

Notes

1. In fact, you can look at the tail half of the protect as a typecase on the exception object.

Chapter 11. Safety Features

Methods definitions may include optional pre- and post-conditions. Together with 'assert' and these features allow the earnest programmer to annotate the intention of code. The Sather compiler provides facilities for turning on or off the runtime checking these safety features imply. Classes may also define a routine named 'invariant', which is a post condition that applies to all public methods.

These safety features are associated with the notion of programming contracts. The precondition of a method is the contract that the method requires the caller to fulfill. It is a statement of the condition of the world that the method needs to find, in order to work correctly. The postcondition is a contract that the method guarantees, if its precondition has been met. It is a statement of the state the method will leave the world in, when it is finished executing. These programming contracts are very important in the creation of robust, reusable code.

In addition to providing a level of checking, these safety features are also an invaluable form of documentation. Since preconditions and postconditions must actually execute, they can be trusted to be accurate and up-to-date, unlike method comments which may easily fall out of sync with the code.

11.1. Preconditions

A precondition states the assumptions that a method makes. It is the contract that the caller must fullfil in order for the routine to work properly. Preconditions frequently include checks that an argument is non-zero or non-void.

The optional 'pre' construct of method definitions contains a boolean expression which must evaluate to true whenever the method is called; it is a fatal error if it evaluates to false.The expression may refer to self and to the routine's arguments. For iterators, pre and post conditions are checked before and after every invocation of the iterator (not just the first or last time the iterator is called).

```
class POSITIVE_INTERVAL is
   readonly attr start, finish:INT;
   create(start, finish:INT)
    - Ensure that the interval is positive on positive numbers
   pre start > 0 and finish > 0 and finish-start >= 0
   is
      res ::= new;
      res.start := start;
      res.finish := finish;
      return res;
   end;
end; - class POSITIVE_INTERVAL
```

Note that it is usually not appropriate to place conditions on the internal state in the precondition. This is an inappropriate conduct, since it may be impossible for the caller to determine whether the conduct can be properly fulfilled.

```
move_by(i:INT)
pre start > 0
is ...
```

The test on 'start' is actually verifying something about the internal state of the object, and has nothing to do with the caller of the routine. Tests such as the one above are more appropriately placed in assertions.

11.2. Postconditions

Post conditions state what a method guarantees to the caller. It is the method's end of the contract. Post conditions are also stated as an optional initial construct in a method.

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 method's arguments.

```
class VECTOR is
   ...
   norm:FLT; - norm of the vector

   normalize
   post norm = 1.0
   is ...
   - Normalize the vector. The norm of the result must be 1.0
```

It is frequently useful to refer to the values of the arguments before the call, as well as the result of the method call. A problem arises because the initial argument values are no longer known by the time the method terminates, since they may have been arbitrarily modified. Also, since the post condition is outside the scope of the method body, it cannot easily refer to values which are computed before the method executes. The solution to this problem consists of using `result` expressions which provide the return value of the method and `initial` expressions which are evaluated at the time the method is invoked.

11.2.1. `initial` expressions

initial expressions may only appear in the `post` expressions of methods.

```
add(a:INT):INT
```

```
post initial(a)>result
is ...
```

The argument to the initial expression must be an expression with a return value and must not itself contain `initial` expressions. When a routine is called or an iterator resumes, it evaluates each `initial` expression from left to right. When the postcondition is checked at the end, each `initial` expression returns its pre-computed value.

11.2.2. *result* expressions

Result expressions are essentially a way to refer to the return value of a method in a postcondition (the post condition is outside the scope of the routine and hence cannot access variables in the routine).

```
sum:INT
post result > 5
is ...
- Means that the value return must be > 5
```

Result expressions may only appear within the postconditions of methods that have return values and may not appear within `initial` expressions. A `result` expression returns the value returned by the routine or yielded by the iterator. The type of a `result` expression is the return type of the method in which it appears (`INT`, in the above example).

11.2.3. Example

The above routine maintains an (always positive) running sum in 'sum'. Only positive numbers are added to the sum, and the result must always be bigger than the argument.

```
class CALCULATOR is
   readonly attr sum:INT;  - Always kept positive

   add_positive(x:INT):INT
   pre x > 0 post result >= initial(x)
   is
      return sum + x;
   end;
end;
```

11.3. Assertions

Assertions are not part of the interface to a routine. Rather, they are an internal consistency check within a piece of code, to ensure that the computation is proceeding as expected.

11.3.1. *assert* statements

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

```
private attr arr:ARRAY{INT};
...
sum_of_elts is
   sum:INT := 0;
   loop e ::= arr.elt!;
      assert e > 0;
      sum := sum + e;
   end;
   return sum;
end;
```

In the above piece of code, we expect the class to only be storing postive values in the array 'arr' . To double check this, when adding the elements together, we check whether each element is positive.

11.4. Invariants

A class invariant is a condition that should never be violated in any object, after it has been created. Invariants have not proven to be as widely used as pre- and post- conditions, which are quite ubiquitous in Sather code.

11.4.1. The *invariant* routine

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.

Consider a class with a list (we use the library class A_LIST) whose size must always be at least 1. Such a situtation could arise if the array usually contains the same sort of elements and we want to use the first element of the array as a prototypical element)

```
class PROTO_LIST is
   private attr l:A_LIST{FOO};

   create(first_elt:FOO):SAME is
      res ::= new;
```

```
      res.l := #;
      res.l.append(first_elt);
      return res;
   end;

   invariant:BOOL is
      return l.size > 0
   end;

   delete_last:FOO is
      return l.delete_elt(l.size-1);
   end;
end;
```

If the 'delete_last' operation is called on the last element, then the assertion will be violated and an error will result.

```
proto:FOO := #;   - Some FOO object
a:PROTO_LIST := #(FOO);
last:FOO := a.delete_last;
   - At runtime, an invariant violation will occur
   - for trying to remove the last element.
```

The invariant is checked at the end of every public method. However, the invariant is not checked after a private routine. If we have the additional routines

```
delete_and_add is
   res ::= internal_delete_last;
   l.append(res);
   return res;
end;

private internal_delete_last:FOO is
   return l.delete_elt(l.size-1);
end;
```

Now we can call 'delete_and_add'

```
proto:FOO := #;
a:PROTO_LIST := #(FOO);
last:FOO := a.delete_and_add;  - does not violate the class invariant
```

The private call to 'internal_delete_last' does violate the invariant, but it is not checked, since it is a private routine.

Chapter 12. 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.

12.1. Fundamental Classes

There are a handful of classes that are specially recognized by the compiler and are implicitly and explicitly used in most Sather code.

12.1.1. *$OB*

'`$OB`' is automatically a supertype of every type. Variables declared to be of this type may hold any object. It has no features.

12.1.2. Array support

Sather objects may have an array portion, which is specified by including either the primitive reference of value array

- '`AREF{T}`' is a reference array class. Any reference class which includes it obtains an array of elements of type `T` in addition to any attributes it has defined. In such classes, `new` has a single integer argument that specifies the size of the array portion. It defines routines and iters named: '`asize`', '`aget`', '`aset`', '`aclear`', '`acopy`', '`aelt!`', '`aset!`', and '`aind!`'. Array indices start at zero.
- '`ARRAY{T}`' includes from '`AREF`' and defines general purpose array objects. They may be directly constructed by array creation expressions.
- '`AVAL{T}`' is the immutable class analog of '`AREF`'. Classes which include '`AVAL`' must define `asize` as an integer constant which determines the size of the array portion.

12.2. Tuples

Tuples are not really a fundamental class, but are commonly used for a very fundamental purpose - multiple return values.

'`TUP`' names a set of parameterized immutable types called tuples, one for each number of parameters. Each has as many attributes as parameters and they are named '`t1`', '`t2`', etc. Each is declared by the type of the corresponding parameter (*e.g.* '`TUP{INT,FLT}`' has attributes '`t1:INT`' and '`t2:FLT`'). It defines '`create`' with an argument corresponding to each attribute.

12.3. The `SYS` Class

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

Operation in the `SYS` class

Routine

Description

`is_eq(ob1, ob2:$OB):BOOL`

Tests two objects for equality. If the arguments are of different type, it returns '`false`'. If both objects are immutable, this is a recursive test on the arguments' attributes. If they are reference types, it returns '`true`' if the arguments are the same object. It is a fatal error to call with external, closure, or void reference arguments.

`is_lt(ob1, ob2:$OB):BOOL`

Defines an arbitrary total order on objects. This never returns true if '`is_eq`' would return true with the same arguments. It is a fatal error to call with external, closure, or void reference arguments.

`hash(ob:$OB):INT`

Defines an arbitrary hash function. For reference arguments, this is a hash of the pointer; for immutable types, a recursive hash of all attributes. Hash values for two objects are guaranteed to be identical when 'is_eq' would return true, but the converse is not true.

`type(ob:$OB):INT`

Returns the concrete type of an object encoded as an '`INT`'.

`str_for_type(i:INT):STR`

Returns a string representation associated with the integer. Useful for debugging in combination with '`type`' above.

`destroy(ob:$OB)`

Explicitly deallocates an object. Sather is garbage collected and casual use of 'destroy' is discouraged. Sather implementations provide a way of detecting accesses to destroyed objects (a fatal error).

12.4. Object Finalization: $FINALIZE

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

12.5. Basic Classes and Literal Forms

The basic Sather classes such as integers and booleans are not treated specially by the compiler. However, they do have language support in the form of convenient literal forms that permit easy specification of values. These literal forms all have a concrete type derived from the syntax; typing of literals is not dependent on context. Each of these basic classes also has a default void initial value.

Type

Description

BOOL

Initial Value: false - Immutable objects which represent boolean values.

CHAR

Initial Value: '\0' - Immutable objects which represent characters. The number of bits in a 'CHAR' object is less than or equal to the number in an 'INT' object.

STR

Initial Value: "" (= void) - Reference objects which represent strings for characters. 'void' is a representation for the null string.

INT

Initial Value: `0` - Immutable objects which represent efficient integers. The size is defined by the implementation but must be at least 32 bits. Bit operations are supported in addition to numerical operations.

`INTI`

Initial Value: `0i` - Reference objects which represent infinite precision integers.

`FLT`

Initial Value: `0.0` - Immutable objects which represent single precision floating point values as defined by the IEEE-754-1985 standard.

`FLTD`

Initial Value: `0.0d` - Immutable objects for double precision floating point values.

12.5.1. Booleans and the BOOL class

Examples:

```
a:BOOL := true;
b ::= false;
c:BOOL := a.and(b);
if a.and(b).or(d) then
end;
```

`BOOL` objects represent boolean values (See also unnamedlink). The two possible values are represented by the *boolean literal expressions*: '`true`' and '`false`'. Boolean objects support the standard logical operations. Note that these logical operations are evaluated in the standard way, and not short-circuited. The Sather expressions "and" and "or" provide a short circuit logical operations.

```
if b.has_value and b.get_value > 3 then
   - The short circuit and will only evaluate b.get_value
   - if b.has_value is true
end;
```

12.5.2. Characters and the CHAR class

Examples:

```
c:CHAR := 'a';
new_line:CHAR := '\n';
code_16:CHAR := '\016';
```

`CHAR` objects represent characters. *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.

- `'\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
- `'\"'` 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.

12.5.3. The string class, STR

Examples:

```
s:STR := "a string literal"
d:STR := "concat" "enation\015"
         - d  is  "concatenation\015"
```

`STR` objects represent strings. *String literal expressions* begin and end with double quote marks. 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 string literals may not extend beyond a single line, but successive string literals are concated together. Thus, a break in a string literal can 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.

12.5.4. Integers and the INT class

Examples:

```
a:INT := 14;
b:INT := -4532;
c:INT := 39_8322_983_298;
binary:INT := 0b101011;
bin:INT := -0b_101010_1010;
octal:ITN := 0o37323;
hex_num:INT:= 0x_ea_75_67;
```

`INT` objects represent machine integers. 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.

Underscores may be used to separate the digits of an integer to improve readability - this may be particularly useful for very long binary numbers.

12.5.5. Infinite precision integers and the INTI class

Examples:

```
b:INTI := -4532i;
infinite_hex:INTI := 0o373254i;
```

Infinite precision integers are are implemetned by the INTI class and supported by a literal form which is essentially the same as that of integers, but with a trailing 'i'. All the standard arithmetic operations are defined on infinite precision integers.

12.5.6. Floating point numbers: the FLT and FLTD classes

Examples:

```
f:FLT := 12.34;
fd:FLTD := 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). 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.

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.

The following 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
```

12.6. Library Conventions

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

- Classes which are related should reflect this in their names. For example, there are many examples in the library of an abstraction, classes implementing the abstraction, and code testing implementations of the abstraction. For example, in the standard library the set abstraction is named `$SET, H_SET` is a hash table implementation, and the test code is `TEST_SET`.
- Some classes implement an immutable, 'mathematical' abstraction (*eg.* integers), and others implement mutable "object" abstractions that can be modified in place (*eg.* arrays). For most objects, the mutable, object semantics are natural and efficient. However, for classes such as sets, the semantics may be different from those of the traditional mathematical set entities.
- Classes with immutable semantics are given their 'mathematical' names: STR, `VEC`, `$SET`. When separate abstractions exist to handle value and reference semantics, the method `value` will be provided in the reference version to provide an immutable snapshop of the reference class.
- Conversions from a type `FOO` to a type `BAR` occur in two ways: by defining an appropriate '`create(f:FOO):BAR`' routine in `BAR` as seen above, or be defining a routine '`bar:BAR`' in `FOO`. For example, in the standard library conversion of a `FLT` to a `FLTD` is done by calling the routine '`fltd:FLTD`' defined in `FLT`.
- Methods which return a `BOOL` (called *predicates*), usually have the prefix '`is_`'. For example, '`is_prime`' tests integers for primality.
- Abstract classes that require a single method should be named after that method. For example, subtypes of `$HASH` define the method '`hash`'.

- If there is a single iterator in a container class which returns all of the items, it should be named '`elt!`'. If there is a single iterator which sets the items, it should be named '`set!`'. In general, iterators should have singular ('`elt!`') rather than plural ('`elts!`') names if the choice is arbitrary.

12.6.1. Object Identity

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

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

IS_EQ

Classes which define their own notion of equality should subtype from `$IS_EQ`. This class is a common parameter bound in container classes. In the standard library, we have

```
abstract class $IS_EQ is
   is_eq(e:$OB):BOOL;
end;
```

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

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

Programmer defined hash functions and $HASH

Many container classes need to be able to compute hash values of their items. Just as with '`is_eq`', classes may subtype from `$HASH` to indicate that they know how to compute their own hash value. $HASH is defined in the library to be

```
abstract class $HASH is
   hash:INT;
end;
```

Objects that can be copied and $COPY

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

```
abstract class $COPY is
   copy:SAME;
end;
```

12.6.2. Nil and void

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

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

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

```
abstract class $NIL is
   nil:SAME;
   is_nil:BOOL;
end; – anstract class $NIL
```

Chapter 13. Interfacing with Fortran

Providing a type-safe Sather interface to Fortran 77 is desirable for several reasons. There is a large body of well debugged and well tested high performance Fortran source code for various kinds of numerical computations. Many vendors provide versions of low level numerical Fortran libraries tuned for particular hardware platforms. Fortran 77 BLAS have become a de facto standard for the elementary vector and matrix operations. The external Fortran interface provides a standard mechanism for Fortran procedures and data to be accessed from Sather and vice versa. It enables a Sather programmer to exploit the wealth of available numerical software in a type safe and portable manner.

Several important issues need to be resolved to provide interoperability between Sather and Fortran. The issues are:

- name binding
- datatype mapping
- parameter passing

unnamedlink introduces the Sather/Fortran interface and provides a few illustrative examples. unnamedlink talks about binding Sather entities to corresponding Fortran entities. Section unnamedlink provides a mapping of "basic" Sather types to Fortran types. Section unnamedlink explains how arguments in a Sather call are passed to a Fortran procedure or function that implements the feature. Finally, section unnamedlink talks about various portability issues.

13.1. Overview

Sather 1.1 provides an interface to a superset of Fortran 77 (ANSI X3.9-1978). The interoperability with Fortran code is achieved with the help of external Fortran classes. External Fortran classes are used to implement a strongly typed bidirectional Sather/Fortran interface. The extended library provides a set of built-in classes corresponding to all Fortran 77 types. Signatures of all inter-language calls must contain only these built-in classes as argument or return types.

13.1.1. External Fortran Call Example

The keywords `'external'` and `'FORTRAN'` preceding `'class'` indicate that some class features may be implemented externally in Fortran and some other features are compiled in a way that makes it possible to call them from Fortran. An example of a simple call to a Fortran function is given below

```
external FORTRAN class FOO is
   foo(a:F_INTEGER,b:F_INTEGER):F_INTEGER;
```

```
   - a feature with a missing body is implemented externally
   - in Fortran.
   - Fortran definition:
   - INTEGER FUNCTION foo(A,B)
   - INTEGER A
   - INTEGER B
   - ...
end;

- a call to an externally defined Fortran function
i:F_INTEGER := FOO::foo(#F_INTEGER(1), #F_INTEGER(2));
- #F_INTEGER(1) creates a variable of Fortran type F_INTEGER and
-  initializes it to 1,
- #F_INTEGER(2) does a similar job,but initializes a new variable to 2
```

`F_INTEGER` is a built-in type representing Fortran integers. A full list of builtin-in Fortran types will be given in unnamedlink. Standard libraries provide a set of constructors and conversion routines for conversion from Sather to Fortran types and vice versa. The definition of feature `'foo'` in external class FOO looks similar to abstract signatures in abstract classes. The implementation of external classes methods without bodies is assumed to be given in a corresponding language (Fortran in the case of 'foo'.) Such abstract signatures specify the interface from Sather code to Fortran code.

13.1.2. Overall Organization

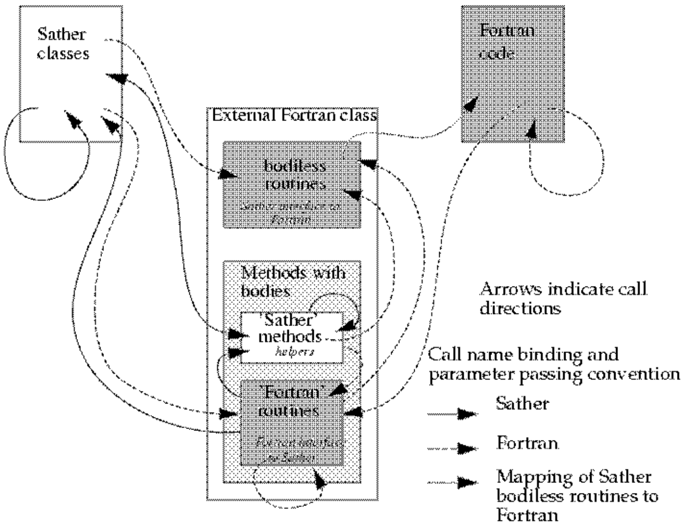
External Fortran classes are used to provide both Sather/Fortran and Fortran/Sather interfaces. External Fortran classes can contain methods of two kinds: bodyless routines indicating externally implemented features and methods with code bodies some of which could be called from Fortran code. External Fortran classes cannot be instantiated and exist only to provide a bidirectional interface from Sather to Fortran.

Only routines may have no body in the external Fortran classes (not iterators). Bodyess routines specify the interface for Sather code (both in external and "regular" Sather classes) to call Fortran code. They have Sather signatures corresponding to the Fortran functions and subroutines implementing these features. Calls to such routines are compiled using the Fortran style name binding and parameter passing convention. The full correspondence between Fortran 77 types and Sather built-in Fortran classes is given in unnamedlink.

Methods with bodies in external Fortran classes serve a dual purpose. Methods whose arguments and return types are a combination of Sather and external Fortran types are merely helper routines and iterators whose semantics is the same as that of regular routines and iterators. They could be called from any Sather or external classes and such calls support the Sather name binding and parameter passing convention. Code for such methods can contain all sorts of calls without restrictions.

If all argument types and a return type, if any, in a routine with a body are built-in Fortran types (e.g. F_INTEGER, F_REAL, etc.) , such routines are meant to be callable from Fortran. They are compiled using the Fortran name binding and parameter passing convention. In fact, they could be freely substituted for Fortran 77 subroutines and functions that perform the same functions. Such routines could be also called from Sather code, but these calls will also support the Fortran parameter passing convention which is often less efficient relative to regular Sather calls. There are no restrictions on the implementation of these function: they can freely use internally any methods implemented either in Sather or Fortran. Routines which are meant to be called from Fortran cannot be overloaded.

In the diagram, arrows indicate the direction of calls. For example, an arrow connecting Fortran classes with bodyless routines in External classes indicate calls in the regular Sather code to abstract routines in the external Fortran classes. The type of the arrow demonstrates that such calls are compiled using the Fortran style call name binding and parameter passing convention. On the other hand, calls from routines with bodies in the external Fortran classes into regular Sather classes are represented by a solid arrow which denotes the Sather call name binding and parameter passing convention.




```
- This is a Fortran definition for FOO
INTEGER FUNCTION FOO(I,A,C)
INTEGER I
REAL A
CHARACTER C
....
END

external FORTRAN class FOO is
```

```
   - this routine is implemented externally in Fortran and could
   - be called in Sather like this: tmp::=FOO::foo(i,a,c)
   foo(i:F_INTEGER,a:F_REAL,c:F_CHARACTER):F_INTEGER;

   - this routine could be called from both Sather and Fortran
   - all calls to bar (either from Sather or Fortran) use the
   - Fortran 77 parameter passing convention
   bar(i:F_INTEGER,a:F_REAL) is
      ...
   end;

   - this routine can only be called in Sather since
   - argument size has a Sather type
   helper(arr:F_ARRAY{F_INTEGER}, size:INT) is
      ...
      t::=foo(i,a,c);  -call uses Fortran parameter passing convention
      bar(t,a);        -Fortran convention, but implemented in Sather
   end;
end;
```

In this example, a Fortran function implementing 'foo' is called in Sather code as if it were a regular Sather routine: FOO::foo(i,a,c). However, the call is generated using the Fortran name binding and parameter passing convention. Calls to 'bar' are compiled in a similar fashion; however, it could be called from both Sather or external Fortran code. Finally, 'helper' has both Sather and external Fortran types as arguments and therefore could be called from Sather code only.

Points to note

- External Fortran class routines without bodies (abstract signatures) provide Sather/Fortran interface.
- Routines with bodies could be called from Fortran if and only if their signatures contain only built-in Fortran types. Such routines can be also called from Sather. However, regardless of the way they are called, they always support the Fortran style name binding and parameter passing convention.
- Methods with bodies whose signatures have non-Fortran types are regular Sather methods. They could be used as helper methods for the interface classes. They always support Sather style name binding and parameter passing.

13.2. Name Binding

Symbols for Sather calls to Fortran code need to be generated in exactly the same way as a Fortran 77 compile would. This is also necessary for the names of routines intended to be called from Fortran. This is difficult to ensure in a portable way since neither Sather nor Fortran 77 language specification prescribes any symbol binding convention and the name mangling strategy is usually very sensitive to particular Fortran platforms. Sather 1.1 attempts to solve the name biding problem in an easy to use, but sufficiently general manner.

13.2.1. Difficulties

Various naming issues have to be resolved to provide seamless platform independent interoperability between Sather and Fortran. Neither Sather nor Fortran specifies a way to mangle symbols generated for the linking stage. Moreover, various Fortran compilers adopt vastly different naming strategies and, in general, it is impossible to link together object files generated by different Fortran compilers. Unfortunately, this is the case even for relatively mainstream platforms: for instance, AT&T f77 compiler name mangling is very different from that of Sun's f77 compiler.

This is an incomplete list of various Fortran 77 naming practices

- long names may be truncated at various lengths
- Fortran names may have one (most common) or two underscores appended
- Fortran names are usually forced to lower case
- external names (external procedures and common blocks) may be mangled differently from internal names (various number of trailing underscores, etc.)

The Sather symbols may be generated using quite different naming conventions. For instance, the ICSI Sather 1.1 compiler generates symbols for Sather routine and iterator names by concatenating a class name (including class parameters) with a routine name, truncating the resulting name to a length specified at the compiler configuration/installation step and appending a number at the end to make the name unique. Other Sather implementations are free to choose any name binding convention.

The set of problems we have to deal with is the same set of problems that needs to be resolved to provide interoperability between Fortran and such an "old" language as C. To this day, there is no standard or even a concrete proposal to resolve F77/C, HPF/C or F95/C name binding issues in a platform independent fashion.

The Sather 1.1 implementation deals with the naming issues in a more fundamental fashion, in some respects, than any of the mentioned external interface proposals.

13.2.2. Implementation

The name mangling strategy for external Fortran names generated by Sather is set at the compiler configuration time. Thus, to move a mixed language program or library from one platform to another, it is only necessary to reconfigure the Sather compiler at the compiler installation time to inform it about the naming convention of the Fortran compiler on the new platform. All user and library code will continue working as is.

There are at least three potential ways to insure the portability of name binding. The simplest (conceptually, not practically!) way is to keep a list of all known Fortran compilers and used name mangling conventions. The Sather compiler should be able to implement any of the possible name binding strategies. This solution was adopted (not implemented!) by the HPFF proposal to provide HPPF/C interoperability. Problems with this approach:

- works only with the existing compilers for other languages. A new compiler cannot be support unless major modifications to the existing tools are performed.
- adds lots of complexity to the Sather compiler as it must know many things about common Fortran compilers
- moving to a new Fortran platform may potentially require large modifications to the Sather compiler internals as the mangling decisions for special cases are hardcoded in the Sather compiler

Another solution that tries to simplify Sather compiler complexity is to add a "Fortran name bind" directive to the Sather language. This directive would specify an actual binding name for each Fortran routine meant to be called from Sather and each Sather routine callable from Fortran. F95/C interoperability proposal partially adopts this approach. This solution, however tedious it may be for the user, may be unavoidable for Fortran to interface other languages since Fortran names are always converted to lowercase and to call an external routine whose name in the symbol table has at least a single uppercase letter a new language construct needs to be added to Fortran. This particular problem, however, may be avoided for Sather. Nevertheless, there are some serious problems with this approach:

- the burden is entirely on the user's shoulders. He/she needs to be aware about too many low-level name binding details
- the "name bind" directive pollutes the code with things that are irrelevant for the program semantics
- it is a pain to port a program to a different Fortran platform: name binding will need to change accordingly

Finally, a completely general solution is to provide a Sather compiler at configuration time with a stand alone function that would take the Sather name as an input and generate a biding Fortran name as output that conforms to all conventions of the current Fortran platform. A library of such functions for most common platforms could be distributed with the compiler, and to port the Sather compiler to an exotic Fortran platform, only a single function will need to be written (or modified given a valid Fortran platform with a similar functionality.) This approach was considered as superior in the F95/C Interoperability Technical Report (ISO/IEC JTC1/SC22/WG5 N1147), but it was not accepted because of the F95 compiler implementation difficulties.

Sather 1.1 tries to shield the user completely from the horrors of low-level mangling details. It adopts the third and most general strategy. In addition, it also provides simple hooks for most common Fortran mangling conventions.

Most Fortran compilers simply append an underscore as a prefix or suffix to the textual name (modulo necessary truncation) . The same behavior for external names could be achieved by setting either one or both configuration variables in the CONFIG file for a particular platform at installation time:

```
FORTRAN_APPEND_UNDERSCORE: true;
FORTRAN_PREFIX_UNDERSCORE: false;
```

In this example, the Fortran binding name is generated from the routine name used in the external Fortran class by appending '_'.

If this is not sufficient, a general Fortran name mangling function can be specified at installation time:

```
FORTRAN_BIND_FUNC:  true;
```

When FORTRAN_BIND_FUNC configuration variable is set to true, a general name binding function BIND_FORTRAN::bind_name(name:STR):STR is invoked whenever Fortran symbols are generated. It, in turn, can call any user supplied mangling function capturing the peculiarity of a particular Fortran platform. BIND_FORTRAN class contains most common binding functions. To port the system to an exotic Fortran environment, a single name binding routine needs to be added to BIND_FORTRAN.

Class BIND_FORTRAN resides in the Fortran library. The following Fortran name binding function simply appends an underscore to the textual name:

```
class BIND_FORTRAN is
   - contains various functions binding Fortran names for exotic
   - architectures. "bind_name" should always call the appropriate
   - function and FORTRAN_FUNC_BIND in CONFIG should be set to true
   bind_name(name:STR):STR is
      res:STR;
      - various Fortran mangling routines should be plugged in here
      res := append_underscore(name);
      return res;
   end;

   append_underscore(s:STR):STR is
      return s + "_";
   end;
end;
```

13.3. Datatype Mapping

The extended Sather 1.1 library provides a set of built-in classes interfacing to Fortran. These types are "binary" compatible with their Fortran 77 counterparts. Only these built-in classes may be used in signatures of routines implemented in Fortran or Sather routines called from Fortran. Fortran scalar types can be used alone or as parametrizations for built-in Fortran array classes. Sather also provides a convenient way for packaging Sather routines and passing them to Fortran functions or subroutines that expect externally defined subroutines as arguments.

Built-in Scalar Types

Fortran 77 -> Sather class

Features

`integer` -> `F_INTEGER`

binary compatible with Fortran 77 integers and can be used whenever Fortran integer type is expected. Supports arithmetic and relational operations, construction from and convention to INT

`real` -> `F_REAL`

represents Fortran 77 reals and can be used whenever Fortran real type is expected. Supports arithmetic and relational operations, construction from and convention to FLT

`logical` -> `F_LOGICAL`

binary compatible with Fortran 77 logical. Supports logical operations and constructors from Sather BOOL type.

`double precision` -> `F_DOUBLE`

binary compatible with Fortran 77 double precision type. Supports a set of features simialr to F_REAL

`complex` -> `F_COMPLEX`

binary compatible with Fortran 77 complex type. Supports arithmetic operations and creation from Sather CPX type (although the binary representation is quite different from CPX)

`double complex` -> `F_DOUBLE_COMPLEX`

binary compatible with Fortran 77 double complex type. Supports a set of features similar to F_COMPLEX, but uses double precision arithmetic.

`character,character*1` -> `F_CHARACTER`

binary compatible with both Fortran 77 character and character*1 types. As an optimizations, inside Sather space it is represented by a single byte and is, therefore, more efficient than corresponding Fortran 77 types.

`character*n` -> `F_STRING`

binary compatible with Fortran 77 character*n type (including character*1). Intra Sather calls are slightly more efficient than corresponding Fortran/Fortran, Sather/Fortran or Fortran/Sather calls.

Array Types

Fortran 77 -> Sather Types

Features

Various array types -> `F_ARRAYn{T<$F_SCALAR}` where n = 1,2,..

Can be parametrized by any scalar Fortran types, binary compatible with the corresponding Fortran 77 arrays: use the same layout. Can be constructed using Sather arrays, matrix and vector classes. arr:F_ARRAY{F_INTEGER} corresponds to INTEGER arrr(*) in Fortran.

Fortran Routine and Exception Handler Types

Fortran 77 -> Sather Type

Features

External subroutines passed as arguments -> `F_ROUT{}`

Used to bind Fortran routines, strongly type checked. Can be passed as arguments to external Fortran routines that expect externally defined subroutines as parameters.

Alternate returns (exception handling) -> `F_HANDLER`

Implements Fortran exception handling in Sather. Can be passed as an arguments to Fortran subroutines with alternate returns (Fortran's way to handle exceptional or abnormal conditions.)

There is also a facility for Sather to provide exception handlers for Fortran subroutines with alternate returns (Fortran's way to handle exceptional or abnormal conditions).

13.3.1. Scalar Types

There are eight built-in scalar types: F_INTEGER, F_REAL, F_LOGICAL, F_DOUBLE, F_COMPLEX, F_DOUBLE_COMPLEX, F_CHARACTER, and F_STRING. They correspond to Fortran 77 types as shown in the table. All scalar Fortran types are subtypes of $F_SCALAR ($F_SCALAR is used as a bound for array parametrizations to ensure that arrays are parameterized with scalar types only).

It is important to distinguish between external Fortran interface types and "regular" Sather types with similar semantics. For example, Sather type INT is different from Fortran F_INTEGER, although both abstract the meaning of integers. There is no sub- or super-typing relationship between INT and F_INTEGER and these types cannot be used interchangeably. No assumption could be made about the relative amounts of memory the Sather and Fortran types need. This is defined differently by Sather and Fortran 77 language specifications. For instance, the only relevant Fortran 77 rule guarantees that integer, logical, and real Fortran types occupy the same amount of memory, and double precision and complex types occupy twice as much (the language does not specify the absolute amounts). Sather, on the other hand, does not specifically support these assumptions.

F_INTEGER

F_INTEGER is a Sather 1.1 class representing Fortran 77 integer type. It can be used whenever a Fortran 77 integer is expected: calls to routines implemented in Fortran, Fortran array parametrizations, etc. The Sather 1.1 library defines the following features for F_INTEGER

```
class F_INTEGER is
   create(x:INT):F_INTEGER is ...  - construct from INT
   int:INT is ...                  - INT version of self
   str:STR is ...                  - string representation
   zero:SAME is ...                - zero and
   nil:SAME is ...                 - nil values
   is_nil:BOOL is ...              - true if self is nil
   plus(i:SAME):SAME is ...
   minus(i:SAME):SAME is ...
   times(i:SAME):SAME is ...
   div(i:SAME):SAME is ...
   is_eq(i:SAME):BOOL is ...
   is_lt(i:SAME):BOOL is ...
end;
```

F_INTEGER could be created using a Sather INT type. An existing F_INTEGER could also yield a corresponding Sather INT value. Although the intended use for F_INTEGER variables is to be passed as arguments to and from external Fortran routines, some simple operations on F_INTEGER variables are built-in and could be performed in Sather directly without going through Fortran. Such operations are the regular arithmetic operations (+ -* /) and logical operations. Syntactic sugar and operator precedence rules are same as those for Sather types.

This example uses an external function defined in Fortran to implement a factorial function missing in the F_INTEGER interface:

```
*      A Fortran function that implements factorial of N
       INTEGER FUNCTION FACTORIAL(N)
       INTEGER N
       FACTORIAL = 1
       DO 10, I=1,N
          FACTORIAL = FACTORIAL * I
 10    CONTINUE
       END

external FORTRAN class USEFUL_FUNCTIONS is
   factorial(i:F_INTEGER):F_INTEGER;
   - a function implemented in Fortran that returns factorial of i
end;

class MAIN is
   main is
      i:F_INTEGER := #(4);
      a:F_INTEGER := USEFUL_FUNCTIONS::factorial(i);
      #OUT + "This " + a.str + " should be 24\n";
   end;
end;
```

F_REAL

F_INTEGER, F_REAL represents Fortran 77 real type.

```
class F_REAL is
   create(x:FLT):F_REAL is ...  - construct from FLT
   flt:INT is ...               - FLT version of self
   str:STR is ...               - string representation
   zero:SAME is ...             - zero and
   nil:SAME is ...              - nil values
   is_nil:BOOL is ...           - true if self is nil
   plus(i:SAME):SAME is ...
   minus(i:SAME):SAME is ...
   times(i:SAME):SAME is ...
   div(i:SAME):SAME is ...
   is_eq(i:SAME):BOOL is ...
```

```
   is_lt(i:SAME):BOOL is ...
end;
```

Sather syntactic sugar for arithmetic and relational operations and operator precedence rules apply to F_REAL. Now, we can extend USEFUL_FUNCTIONS class with a power routine for F_REAL:

```
external FORTRAN class USEFUL_FUNCTIONS is
   - external Fortran function that raises x to power y
   power(x:F_REAL,y:F_REAL):F_REAL;
end;
```

F_DOUBLE

F_DOUBLE represents Fortran 77 double type.

```
class F_REAL is
   create(x:FLTD):F_REAL is ...       - construct from FLTD
   fltd:INT is ...                    - FLTD version of self
   str:STR is ...                     - string representation
   zero:SAME is ...                   - zero and
   nil:SAME is ...                    - nil values
   is_nil:BOOL is ...                 - true if self is nil
   plus(i:SAME):SAME is ...
   minus(i:SAME):SAME is ...
   times(i:SAME):SAME is ...
   div(i:SAME):SAME is ...
   is_eq(i:SAME):BOOL is ...
   is_lt(i:SAME):BOOL is ...
end;
```

Sather syntactic sugar for arithmetic and relational operations and operator precedence rules apply to F_DOUBLE.

F_LOGICAL

F_LOGICAL is a Sather class representing Fortran 77 logical type. It is "binary" compatible with Fortran's "logical" type (Sather BOOL has a vastly different representation in ICSI 1.1 compiler). In particular, F_LOGICAL occupies the same amount of space as Fortran integer and real types to conform to Fortran 77 rules.

```
class F_LOGICAL is
   create(x:BOOL):F_LOGICAL is ...  - construct from INT
   bool:BOOL is ...                 - INT version of self
   str:STR is ...                   - string representation
```

```
   not:SAME is ...
   is_eq(B:SAME):BOOL is ...
   f_or(b:SAME):SAME is ...
   f_and(b:SAME):SAME is ...
end;
```

Logical operations are called f_or and f_and to avoid name collisions with short-circuited Sather operators `'and'` and `'or'`. The following function implementing exclusive or can be added to USEFUL_FUNCTIONS

```
xor(a:F_LOGICAL,b:F_LOGICAL):F_LOGICAL is
   return (a.not.f_and(b)).f_or(a.f_and(b.not));
end;
```

F_COMPLEX

F_COMPLEX is a Sather class binary compatible with Fortran 77 COMPLEX type. Although F_COMPLEX provides a constructor that accepts a variable of Sather CPX type, F_COMPLEX has a binary representation quite different from that of CPX. F_COMPLEX provides a set of features for setting and returning the values of the real and imaginary parts. It also provides useful constructors and supports a set of arithmetic operations.

```
class F_COMPLEX is
   re:F_REAL is ...             - return real part
   re(x:F_REAL) is ...          - set real part
   im:F_REAL is ...             - return imaginary part
   im(x:F_REAL) is ...          - set imaginary part
   create(c:CPX):SAME is ...    - create new and
                                - initialize to value of c
   create(re:F_REAL,im:F_REAL):SAME is ...
   create(re:FLT,im:FLT):SAME is ...
   create(fc:F_COMPLEX):SAME is ...
   cpx:CPX is ...               - Sather comlplex type
   str:STR is ...               - string representation
   zero:SAME is ...             - zero and
   nil:SAME is ...              - nil value
   is_nil:BOOL is ...           - true if self is nil
   plus(c:SAME):SAME is ...
   minus(c:SAME):SAME is ...
   times(c:SAME):SAME is ...
   div(c:SAME):SAME is ...
   is_eq(c:SAME):BOOL is ...
end;
```

This is a possible implementation of addition of F_COMPLEX numbers:

```
plus(c:F_COMPLEX):F_COMPLEX is
   return #F_COMPLEX(re+c.re,im+c.im);
end;
```

F_DOUBLE_COMPLEX

Similar to F_COMPLEX, F_DOUBLE_COMPLEX is a Sather class binary compatible with the Fortran double complex type. Double complex type is an extension to Fortran 77 supported by many F77 compiler. F_DOUBLE_COMPLEX class provides functionality similar to F_COMPLEX, but works with double precision floating point representations.

```
class F_DOUBLE_COMPLEX is
   re:F_DOUBLE is ...           - return real part
   re(x:F_DOUBLE) is ...        - set real part
   im:F_DOUBLE is ...           - return imagianry part
   im(x:F_DOUBLE) is ...        - set imaginary part
   create(c:CPXD):SAME is ...   - create new and
                                - initialize to value of c
   create(re:F_DOUBLE,im:F_DOUBLE):SAME is ...
   create(re:FLTD,im:FLTD):SAME is ...
   create(fc:F_DOUBLE_COMPLEX):SAME is ...
   cpxd:CPXD is ...             - CPXD version of self
   str:STR is ...               - string representation
   zero:SAME is ...             - zero and
   nil:SAME is ...              - nil value
   is_nil:BOOL is ...           - true if self is nil
   plus(c:SAME):SAME is ...
   minus(c:SAME):SAME is ...
   times(c:SAME):SAME is ...
   div(c:SAME):SAME is ...
   is_eq(c:SAME):BOOL is ...
end;
```

F_CHARACTER

F_CHARACTER is binary compatible with Fortran 77 types character and character*1. Fortran 77 character and character*1 types are, in fact, instances of character*n types with n set to 1. In Sather terms, they are strings with size always set to one. For parameter passing purposes, Fortran character and character*1 variables behave exactly as generic character*n types (the length of the string which is always one is passed as an extra parameter for each character or character*1 argument). Since the goal for F_CHARACTER is binary compatibility with Fortran, this is how F_CHARACTER class behave when a call crosses the language boundary. However, as long as F_CHARACTER variables stay within the Sather space, they are represented and passed to routines more efficiently, as a single byte. As a result, simple character operations performed on F_CHARACTER class in Sather are more efficient than their Fortran versions!

```
class F_CHARACTER is
   create(c:CHAR):SAME is ...  - create new and
                               - initialize to value of c
   char:CHAR is ...            - CHAR version of self
   str:STR is ...              - STR version of self
   zero:SAME is ...            - zero
   is_eq(c:SAME):BOOL is ...
   is_lt(c:SAME):BOOL is ...
end;
```

F_STRING

F_STRING is binary compatible with Fortran 77 character*n types. Note, that both F_CHARACTER and F_STRING can be used to interface with Fortran character*1 type, but F_CHARACTER yields better performance for computations performed in Sather.

F_STRING is internally represented by a tuple: the first field points to the string itself, and the second records the string length. An inter-language call requires that both be passed as separate arguments. In unnamedlink you find more information on this. Inside Sather however (calls using the Sather parameter passing convention), F_STRING is passed as a whole, which is slightly more efficient than the Fortran calls.

```
class F_STRING is
   create(s:STR):SAME is ...   - create new and
                               - initialize to value of s
   create(n:INT):SAME is ...   - new of size n
   create(c:CHAR):SAME is ...  - create from c
   address:C_CHAR_PTR is ...   - the "string" part
   size:INT is ...             - string length
   str:STR is ...              - STR version of self
end;
```

13.3.2. Fortran Array Classes

Providing a convenient array interface is an important goal for Sather/Fortran interoperability. A set of parametrized classes F_ARRAY{T<$F_SCALAR}, and F_ARRAYn{T<$F_SCALAR}, where n=2,3...

are used for this purpose. Array classes can be parametrized by any of the scalar types. For example, F_ARRAY{F_INTEGER} corresponds to a Fortran 77 integer array type. Similarly, F_ARRAY2{F_REAL} represents a Fortran 77 two-dimensional array of real numbers.

F_ARRAY classes must be binary compatible with the Fortran 77 arrays and therefore they conform to the Fortran array layouts. For instance, this requires that in a two dimensional arrays successive elements of a column are in a contiguous memory locations (i.e. column major layout.) Note that regular Sather arrays (ARRAY{}, ARRAY2{}, etc.) support C-like row-major layout. Thus, creation of Fortran arrays based on Sather arrays may require a layout change. On the other hand, matrix classes provided by the Sather Math library have the same layout as Fortran arrays. F_ARRAY2 classes provide constructors from MAT classes that have reference semantics - thus the creation procedure is fairly inexpensive.

Combining materials from this chapter, and using Fortran array types, we can construct a simple Sather interface to standard Fortran BLAS single precision matrix multiplication routine as follows:

```
    SUBROUTINE SGEMM (TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)
    CHARACTER*1        TRANSA, TRANSB
    INTEGER            M, N, K, LDA, LDB, LDC
    REAL               ALPHA, BETA
    REAL               A( LDA, * ), B( LDB, * ), C( LDC, * )

external FORTRAN class BLAS is
   sgemm(transa:F_CHARACTER, transb:F_CHARACTER, m,n,k:F_INTEGER,
         alpha:F_REAL, a:F_ARRAY2{F_REAL}, lda:F_INTEGER,
         b:F_ARRAY2{F_REAL}, ldb:F_INTEGER,beta:F_REAL,
         C:F_ARRAY2{F_REAL},ldc:F_INTEGER);
   - this corresponds to the fortran BLAS signature
end;

class TEST_BLAS is
   main is
      fa,fb,fc:F_ARRAY2{F_REAL};
      sa,sb,sc:MAT;
      initialize(sa,sb,sc);
      fa := #(sa);  - these creations has "reference" semantics
      fb := #(sb);
      fc := #(sc);
      dim:F_INTEGER := #(fa.size);
      TEST_BLAS::sgemm(#('N'),#('N'),#(sa.nr),#(sb.nc),#(sa.nc),#(1.0),
                       fa,#(sa.size1),fb,#(sb.size1),#(0.0),fc,#(sc.size1));
      - at this point, both fc and sc have a multiplication result
   end;

   initialize(sa:MAT,sb:MAT,sc:MAT) is
       - initialization code ...
```

```
   end;
end;
```

We can go one step father and hide the details of Fortran implementation of sgemm entirely from the user:

```
class MAT is
   ...
   various methods from MAT class
   ...
   times(m:SAME):SAME is
      - multiply self by m and return the resulting matrix
      - For efficiency, uses high-performance Fortran 77 BLAS sgemm
      res:MAT := #(nr,m.nc); - storage for result
      fa,fb,fc:F_ARRAY2{F_REAL};
      fa := #(self);
      fb := #(m);
      fc := #(res);
      - now, call the Fortran BLAS sgemm
      TEST_BLAS::sgemm(#('N'),#('N'),#(nr),#(m.nc),#(nc),#(1.0),
      fa,#(size1),fb,#(m.size1),#(0.0),fc,#(res.size1));
      - at this point, both fc and res have a multiplication result
      return res;
   end;
end;

- now it is really easy to multiply matrices!
a,b,c:MAT;
c := a*b;
```

This code shows that using high-performance Fortran BLAS in Sather is, in fact, much easier than in Fortran! The internal workings of BLAS could be buried in the libraries. As a result, matrix multiplication is expressed as easily as "a*b" in the example. If the code is compiled with compiler optimizations on, the Sather inlining stage eliminates an extra routine call, and the end result will be as efficient as calling "sgemm" from Fortran directly. However, we get away with not specifying about a dozen parameters in the most general case.

In the given example, the space for the multiplication result '`fc`' needs to be allocated in Sather (Fortran 77 has no means for a dynamic memory allocation). This is also necessary even when Fortran arrays are returned by functions.

Points to note

- Fortran arrays have a different layout from standard Sather arrays. In particular, in F_ARRAY2, consecutive elements in array columns occupy consecutive storage, while ARRAY2 has a row-major layout.
- MAT classes have the same layout as Fortran arrays, and conversion from MAT to F_ARRAY2 is very light-weight (reference semantics)

13.3.3. F_ROUT and F_HANDLER Types

Passing Routines as Arguments, F_ROUT{}

Fortran 77 supports passing procedures as arguments to subroutines and functions. It is desirable to be able to package a Sather routine and pass it as an argument to Fortran code. It may prove necessary for example, when Fortran numerical code expects a differentiation or integration function to be passed as an argument. Since we would like to exploit Sather flexibility and development speed whenever possible, a natural thing to do is to write such integration routines in Sather and pass them to numerical Fortran code.

Sather 1.1 provides a way to bundle any routine in the External class that supports the Fortran parameter passing convention and pass it as a functional argument to Fortran code that expects external procedures as parameters. A Fortran routine type F_ROUT{} serves this purpose. In many ways, F_ROUTs are similar to Sather routine closures. Just as routine closures, they are strongly typed and provide similar creation facilities. However, unlike routine closures, all arguments in the Fortran routine used for creation must be left unbound. This is necessary to adhere to Fortran semantics and for performance considerations.

'#F_ROUT(...)' is a creation expression that surrounds a Fortran calls with all arguments replaced by the underscore character '_'. For example, this code may be used to compute a distance between two points on the plane whose coordinates are represented by Fortran complex numbers:

```
external FORTRAN class STAT is
   distance(point1:F_COMPLEX, point2:F_COMPLEX,res:F_REAL) is
      - this routine is compiled using the Fortran parameter
      - passing convention and name binding. It could be called
      - from either Sather or Fortran
      x1:FLT := point1.re.flt; y1:FLT := point1.im.flt;
      x2:FLT := point2.re.flt; y2:FLT := point2.im.flt;
      res := #F_REAL(((x1-x2).square + (y1-y2).square).sqrt);
   end;
```

```
   - this routine is implemented externally in Fortran
   process_points(array1:F_ARRAY{F_COMPLEX}, array2:F_ARRAY{F_COMPLEX},
                  func:F_ROUT{F_COMPLEX,F_COMPLEX,F_REAL),size:F_INTEGER);
end;
```

In the above example, an externally implemented Fortran subroutine process_points expects two arrays of complex numbers and a function that will be applied to corresponding elements in the arrays:

```
    SUBROUTINE PROCESS_POINTS(ARRAY1,ARRAY2,FUNC,SIZE)
    COMPLEX ARRAY1(*), ARRAY2(*)
    EXTERNAL FUNC
    INTEGER SIZE

    REAL RES
    DO 10 I=1,SIZE
      CALL FUNC(ARRAY1(I),ARRAY2(I),RES)
      PRINT *, RES
10  CONTINUE
    END
```

We can pass a routine defined in Sather to Fortran subroutine process_points the following way:

```
- This code appears in some STAT feature
array1, array2:F_ARRAY{F_COMPLEX}
- some code to initialize array1 and array2

rout:F_ROUT{F_COMPLEX,F_COMPLEX,F_REAL} := #F_ROUT(distance(_,_,_));
process_points(array1,array2,rout); - call Fortran code
```

Variables of F_ROUT type behave similarly to ROUT variables. It is possible to assign to such variables, pass them as parameters, etc.:

```
rout:F_ROUT{F_COMPLEX,F_COMPLEX,F_REAL} := #F_ROUT(distance(_,_,_));

rout1:F_ROUT{F_COMPLEX,F_COMPLEX,F_REAL);
rout1 := rout;  - F_ROUT assignment: lhs and rhs types are the same
```

Points to note

- F_ROUT type specifiers are very similar to routine closure type specifiers, but all types inside F_ROUT{} must be Fortran types.
- All call arguments in #F_ROUT() must remain unbound (Fortran 77 does not support binding subroutine or function arguments)

- F_ROUT variables could be assigned to just like other Sather variables. The types of the right-hand side and the left-hand sides of such assignments are strongly checked the same way as for other assignments.
- Unlike Sather routine closures, there may be no subtyping relationship between different F_ROUT types (this is because all Fortran types are concrete). Assignments or calls are possible only when the types are exactly the same.
- #F_ROUT() could be used just as well to bind externally defined Fortran routines (routines without bodies). These could be passed back to Fortran or used in Sather without restrictions.
- Type inferencing for F_ROUT creations works exactly as that for closure creation expressions.
- F_ROUT arguments cannot be passed as `"out"` or `"inout"` arguments.

Exceptional Condition Handling, F_HANDLER

It is possible in Fortran to anticipate exceptional conditions and have different flow paths depending on whether the called subroutine has terminated properly, or has detected abnormal circumstances. This is achieved using the alternate RETURN facility.

```
*       A call to a subroutine with "alternate returns"
*       This is a Fortran's way to handle exceptional conditions
*       If, for some reason, FOO detects an abnormality
*       it can choose to return to exception handlers
*       (passed as labels 100 and 200), rather than to the caller
        CALL FOO(I,J,*100,*200)
1       ....

*       Handle exceptions
*       Exception Handler 1
100     ....
        GO TO 1
200     Exception Handler 2
        ....
        GO TO 1

*       A subroutine with alternate returns
*       Two exception handlers are passed in (marked by *)
*       RETURN 1 transfers control to the first handler, and
*       RETURN 2 transfers control to the second handler
*       "Normal" RETURN transfers control to the caller
        SUBROUTINE FOO(I,J,*,*)
        ...
*       Detect abnormal conditions and transfer control to
```

```
        the appropriate exception handlers
        IF (I.EQ.0) RETURN 1
        IF (J.EQ.0) RETURN 2
        END
```

In the given example, the argument list of the call to subroutine FOO includes 2 labels corresponding to the exception handler entries. If an exceptional condition of some sort arises, FOO will transfer control to the appropriate exception handler (passed as an argument) rather than the caller. For example, if the value of argument I is 0, the control is transferred to exception handler 1, if J is 0, exception handler 2 handles the exception. The exception handlers are indicated by the dummy asterisk arguments in the subroutine argument list. Only subroutines are allowed to have such arguments.

Since alternate returns are a part of Fortran, they may be present in the interfaces provided by the Fortran libraries. It is, therefore, desirable to call such subroutines from Sather and provide exception handlers written in Sather for such calls.

The F_HANDLER class captures the essence of the Fortran exception handlers and could be passed in as an argument to a subroutine with alternate returns. F_HANDLER provides a single constructor create(rout:ROUT):SAME. The argument is a bound routine with no arguments since Fortran handlers do not have any arguments. Now, we will call the Fortran subroutine FOO, but supply Sather exception handlers at the moment of the call.

```
class HANDLERS is
   h(i:INT) is
      #OUT + "Sather handler for Fortran exception " + i.str + "\n";
   end;

   create:SAME is
      return new;
   end;
end;

external FORTRAN class FOO is
   foo(i:F_INTEGER,j:F_INTEGER,handler1:F_HANDLER, handler2:F_HANDLER);
   - note that foo can't have a return value - this is a Fortran
   - restriction on subroutine with alternate returns
end;

- code that calls Fortran FOO
handlers:HANDLERS := #;
handler1:F_HANDLER := #(bind(handlers.h(1))); - create first handler
handler2:F_HANDLER := #(bind(handlers.h(2))); - create second handler
FOO::foo(#(1),#(0),handler1,handler2);
```

When this code is executed, it prints: "Sather handler for Fortran exception 2".

F_HANDLER mechanism allows to integrate Fortran and Sather exceptions even more closely. For example, we can use Sather exception handlers that catch Fortran exceptions to raise standard Sather exceptions that are caught by the Sather protect mechanism. Essentially, this turns Fortran exception into regular Sather exceptions:

```
class HANDLERS is
   r_h(i:INT) is
      raise "FORTRAN->Sather exception redirected by handler #" + i.str;
   end;
   create:SAME is return new; end;
end;

external FORTRAN class FOO is
   foo(i:F_INTEGER,j:F_INTEGER,handler1:F_HANDLER, handler2:F_HANDLER);
   - note that foo can't have a return value - this is a Fortran
   - restriction on subroutine with alternate returns
end;

- code that calls Fortran FOO
handlers:HANDLERS := #;
redirect_handler1:F_HANDLER := #(bind(handlers.r_h(1)));
redirect_handler2:F_HANDLER := #(bind(handlers.r_h(2)));
protect
   FOO::foo(#(1),#(0),redirect_handler1,redirect_handler2);
when STR then
   #OUT + "Sather exception for "+exception+\n";
end
```

This code produces: "Sather exception for FORTRAN->Sather exception redirected by handler 2"

Points to note

- Only routines that have no return value can have F_HANDLER arguments. This is a Fortran restriction: only subroutines (not functions) can have alternate returns.
- F_HANDLER can be created from a standard closure with no arguments or return value: ROUT. An attempt to use closures of other types (like ROUT{INT}) is reported as an error. This restriction is also necessitated by the semantics of alternate returns in Fortran. Fortran exception handlers do not permit arguments.
- F_HANDLER types cannot be passed as "`out`" or "`inout`" arguments.

13.4. Parameter Passing

Some routines and calls in external Fortran classes are compiled using the Fortran parameter passing convention. This section describes how this is achieved. Routines without bodies in external Fortran classes and Fortran routines (routines whose return types and all arguments are Fortran types) are compiled as described below. The explanation is done in terms of mapping the original Sather signatures to C prototypes. All Fortran types are assumed to have corresponding C types defined. For example, F_INTEGER class maps onto F_INTEGER C type. See unnamedlink for details on how this could be achieved in a portable fashion. The examples are used to illustrate parameter passing only - the actual binding of function names is irrelevant for this purpose.

13.4.1. Return Types

Routines that return F_INTEGER, F_REAL, F_LOGICAL, and F_DOUBLE map to C functions that return corresponding C types. A routine that returns F_COMPLEX or F_DOUBLE_COMPLEX is equivalent to a C routine with an extra initial arguments preceding other arguments in the argument list. This initial argument points to the storage for the return value.

```
F_COMPLEX foo(i:F_INTEGER,a:F_REAL);
- this Sather signature is equivalent to
void foo(F_COMPLEX* ret_val, F_INTEGER* i_address, F_REAL* a_address)
```

A routine that returns F_CHARACTER is mapped to a C routine with two additional arguments: a pointer to the data, and a string size, always set to 1 in the case of F_CHARACTER.

```
F_CHARACTER foo(i:F_INTEGER, a:F_REAL);
- this Sather signature maps to
void foo(F_CHARACTER* address, F_LENGTH size,
         F_INTEGER* i_address, F_REAL* a_address);
```

Similarly, a routine returning F_STRING is equivalent to a C routine with two additional initial arguments, a data pointer and a string length[1]

```
F_STRING foo(i:F_INTEGER, a:F_REAL);
- this Sather signature maps to
void foo(F_CHARACTER* address, F_LENGTH size,
         F_INTEGER* i, F_REAL* a);
```

13.4.2. Argument Types

All Fortran arguments are passed by reference. In addition, for each argument of type F_CHARACTER or F_STRING, an extra parameter whose value is the length of the string is appended to the end of the argument list.

```
foo(i:F_INTEGER,c:F_CHARACTER,a:F_REAL):F_INTEGER
- this is mapped to
F_INTEGER foo(F_INTEGER* i_address, F_CHARACTER*c_address,
              F_REAL* a_address, F_LENGTH c_length);
- all calls have c_length set to 1

foo(i:F_INTEGER,s:F_STRING,a:F_REAL):F_INTEGER
- this is mapped to
F_INTEGER foo(F_INTEGER* i_address, F_CHARACTER* s_address,
              F_REAL* a_address, F_LENGTH s_length);
- propoer s_length is supplied by the caller
```

Additional string length arguments are passed by value. If there are more than one F_CHARACTER or F_STRING arguments, the lengths are appended to the end of the list in the textual order of string arguments:

```
foo(s1:F_STRING,i:F_INTEGER,s2:F_STRING,a:F_REAL);
- this is mapped to
void foo(F_CHARACTER* s1_address, F_INTEGER* i_address,
         F_CHARACTER* s2_address, F_REAL a_address,
 F_LENGTH s1_length, F_LENGTH s2_length);
```

Sather signatures that have F_HANDLER arguments correspond to C integer functions whose return value represents the alternate return to take. The actual handlers are not passed to the Fortran code. Instead, code to do the branching based on the return value is emitted by the Sather compiler to conform to the alternate return semantics.

Arguments of type F_ROUT are passed as function pointers.

Thus, the entire C argument list including additional arguments consists of:

- one additional argument due to F_COMPLEX or F_DOUBLE_COMPLEX return type, or two additional arguments due to F_CHARACTER or F_STRING return type
- references to "normal" arguments corresponding to a Sather signature argument list
- additional arguments for each F_CHARACTER or F_STRING argument in the Sather signature

The following example combines all rules

```
foo(s1:F_STRING, i:F_INTEGER, a:F_REAL, c:F_CHARACTER):F_COMPLEX
- is mapped to
void foo(F_COMPLEX* ret_address, F_CHARACTER* s1_address,
```

```
        F_INTEGER* i_address, F_REAL* a_address,
        F_CHARACTER* c_address, F_LENGTH s1_length,
        F_LENGTH c_length);
-- all Sather calls have c_length set to 1
```

13.4.3. OUT and INOUT Arguments

Sather 1.1 provides the extra flexibility of `'out'` and `'inout'` argument modes for Fortran calls. The Sather compiler ensures that the semantics of `'out'` and `'inout'` is preserved even when calls cross the Sather language boundaries. In particular, the changes to such arguments are not observed until the call is complete - thus the interlanguage calls have the same semantics as regular Sather calls.

This additional mechanism makes the semantics of some arguments visually explicit and consequently helps catch some bugs caused by the modification of `'in'` arguments (all Fortran arguments are passed by reference, and Fortran code can potentially modify all arguments without restrictions.) A special compiler option may enable checking the invariance of Fortran `'in'` arguments[2].

In the case of calling Fortran code, the Sather compiler ensures that the value/result semantics is preserved by the caller - the Sather compiler has no control over external Fortran code. This may involve copying `'inout'` arguments to temporaries and passing references to these temporaries to Fortran. In the case of Sather routines that are called from Fortran, the Sather compiler emits a special prologue for such routines to ensure the value/result semantics for the Fortran caller. In summary, the value/result semantics for external calls to Fortran is ensured by the caller, and for Sather routines that are meant to be called by Fortran it is implemented by the callee.

This example suggests how a signature for a routine that swaps two integers:

```
SUBROUTINE SWAP(A,B)
INTEGER A,B

-- a Sather signature may look like
swap(inout a:F_INTEGER, inout b:F_INTEGER);
```

Note that using argument modes in this example makes the semantics of the routine more obvious.

In the following example, compiling the program with all checks on may reveal a bug due to the incorrect modification of the vector sizes:

```
SUBROUTINE ADD_VECTORS(A,B,RES,size)
REAL A(*),B(*),RES(*)
INTEGER SIZE

-- Sather signature
add_vectors(a,b,res:F_ARRAY{F_REAL}, size:F_INTEGER)
-- size is an 'in' parameter and cannot be modified by Fortran code
```

In addition to extra debugging capabilities, `'in'` arguments are passed slightly more efficiently than `'out'` and `'inout'` arguments.

Points to note

- F_ROUT and F_HANDLER types cannot be `"out"` or `"inout"` arguments.

13.5. Portability Issues

This section discusses the portability of the Sather/Fortran interface. See unnamedlink about Various name binding portability issues. Issues relevant to code portability are addressed here.

13.5.1. Portability of the Interface Implementation Code

It is important to distinguish between portability of the Sather compiler module that implements the Sather/Fortran interface and the portability of the code it generates. The Fortran 77 interface module is written entirely in Sather and is integrated with the ICSI Sather compiler. The Fortran interface should be available on all platforms where the ICSI Sather compiler is available. In particular, it is available on most UNIX platforms.

13.5.2. Portability of the Generated Code

The Fortran 77 standards says that all Fortran 77 types except for COMPLEX, DOUBLE PRECISION, and CHARACTER of any flavor occupy a single "unit" of storage space. COMPLEX and DOUBLE PRECISION types take two "units" of storage. This may need to be adjusted accordingly when porting the Sather compiler to a different platform. A modification to "System/Common/fortran.h" may be necessary. "System/Common/fortran.h" contains a set of definitions for Fortran storage types used by the Sather/Fortran interface:

```
typedef long int    F_INTEGER;
typedef long int    F_LOGICAL;
typedef float       F_REAL;
typedef double      F_DOUBLE;
typedef char        F_CHARACTER;
typedef long int    F_LENGTH;
typedef struct {
   F_REAL re, im;
} F_COMPLEX_struct;
```

```
typedef F_COMPLEX_struct F_COMPLEX;
...
```

This proves to be adequate for most UNIX platforms. On the Cray, however, both float and double types occupy the same storage, and to conform to Fortran 77 specification, fortran.h needs to be edited to define F_DOUBLE as "long double". For the Macintosh, however, it should be defined as "short double."

This is a full set of C types that are used by the interface as return and argument types:

```
F_INTEGER
F_LOGICAL
F_REAL
F_DOUBLE
F_CHARACTER
F_STRING
F_LENGTH
F_COMPLEX
F_HANDLER
F_ROUT
integer or integer*4
logical
real
double
character or character*1
character*n
string length (same as F_INTEGER)
complex
call argument for a subroutine with alternate returns
a routine passed as argument
```

Array types are represented as pointer to corresponding scalar types.

Notes

1. The current Sather 1.1 implementation disallows returning Fortran strings of size greater than 32 bytes. This restriction may be lifted in the future releases.
2. The ICSI Sather 1.1 compiler currently does not implement this functionality.

Chapter 14. Interfacing with ANSI C

This chapter describes interfacing with ANSI C , X3.159-1989. See unnamedlink for a short overview of the C interface functionality. See unnamedlink for a introducion on built-in C types provided by the extended Sather library. In unnamedlink you find information about user defined external C types, constants, attributes, and shared elements. unnamedlink covers parameter passing issues, and finally unnamedlink describes the inline C facility.

14.1. Overall Organization

An external class which interfaces to ANSI C is designated with the language identifier '`C`'. Types defined by external C classes are called *external C types*. Similar to external Fortran types, signatures without bodies (abstract signatures) are allowed in external C types. Such signatures must contain only built-in or user defined C types and they are implemented externally in ANSI C. Abstract iterator signatures are not allowed in external C classes. Routines with bodies whose signatures contain only C types may be called from C. Routines with bodies whose signatures use types other than C types are regular Sather routines and are not accessible from C. External C routines cannot be overloaded.

In contrast with the external Fortran classes, external C classes may have attributes and objects of external C types may exist. All attributes must also be of C types. The C interface provides a naming facility that allows interoperability with the existing C header files.

Global C variables can be accessed as shared attributes of external C classes.

C symbols are generated by applying a platform specific C name binding convention to the textual external C routine names. It is also possible to explicitly specify name binding for external C classes.

Finally, it is possible to inline ANSI C code into Sather sources. This allows for even greater flexibility in achieving Sather/C interoperability.

14.2. Built-in C classes

The following C types are built into the extended library:

```
Sather Class          ANSI C type
-------------------------
C_CHAR                char
C_CHAR_PTR            char *
C_UNSIGNED_CHAR       unsigned char
C_UNSIGNED_CHAR_PTR   unsigned char *
C_SIGNED_CHAR         signed char
```

```
C_SIGNED_CHAR_PTR     signed char *
C_SHORT               short
C_SHORT_PTR           short *
C_INT                 int
C_INT_PTR             int *
C_LONG                long
C_LONG_PTR            long *
C_UNSIGNED_SHORT      unsigned short
C_UNSIGNED_SHORT_PTR  unsigned short *
C_UNSIGNED_INT        unsigned int
C_UNSIGNED_INT_PTR    unsigned int *
C_UNSIGNED_LONG       signed long
C_UNSIGNED_LONG_PTR   unsigned long *
C_FLOAT               float
C_FLOAT_PTR           float *
C_DOUBLE              double
C_DOUBLE_PTR          double *
C_LONG_DOUBLE         long double
C_LONG_DOUBLE_PTR     long double *
C_PTR                 void *
C_PTRDIFF_T           ptrdiff_t
C_SIZE_T              size_t
```

Variable of the built-in types are binary compatible with the corresponding C types. These classes define appropriate creation routines which may be used for convenient casting between Sather and C types. Also, many basic operations on the built-in C types are provided by the library. For example, it is not necessary to call external C code to add two C_INT variables. All operations on built-in C types defined by the library have the ANSI C semantics. Syntactic sugar for the built-in C types is defined exactly as for "regular" Sather classes.

```
- "basic" operations may be done in Sather
a:C_LONG := #(10);
b:C_LONG := #(5);
c::= a + b;
#OUT + c.str + " should be 15\n";
```

'`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.

The following example shows how a Sather array could be passed to external C functions:

```
/* ANSI C prototypes for functions called from Sather */
```

```
void clear(void* p, int size);
void better_clear(int *, int size);

external C class PROCESS_ARRAYS is
   - routines implemented externally in C that zero
   - all elements in an integer array of a specified size
   clear(p:C_PTR, size:C_INT);
   better_clear(p:C_INT_PTR, size:C_INT);
end;

- This code demonstrates how to call external C routines
a:ARRAY{INT} := #(10);
- this call just passes an array portion and avoids typecheking
- This is not recommended ("a" could be of type ARRAY{CHAR} and the
- call would still compile resulting in a runtime error)
PROCESS_ARRAYS::clear(a.arr_ptr, #(a.size);

- this is a better sequence achieving the same result
- if "a" is not an array of integers, an error is reported
PROCESS_ARRAYS::better_clear(#C_INT_PTR(a), #(a.size));
```

The second call is type-safe. It exploits the constructor for C_INT_PTR that allows creation from ARRAY{INT}.

14.3. User-defined External C types

User-defined external C classes are used for multiple purposes. C routines in external C classes implement Sather/C and C/Sather call interfaces. In addition, objects of external C types could be created and passed to or received from C. C global variables are accessed from Sather as shared attributes of external C classes.

14.3.1. Constants and C binding names

Constants are allowed in external C classes. The rules for constant initialization are the same as for constants in "regular" Sather classes.

There are two constant features of external C classes that have a special semantics. If present, the `STR` constant '`C_name`' may be used to force a particular C declaration for an external C type. Similarly the `STR` constant '`C_header`' may be used to specify a list of C header files that should be included in each file in which the C declaration appears.

The STR constant 'C_name' provides a C binding name for the type in which it occurs. The STR constant 'C_header' must be initialized to a space separated list of header files (the standard C notation <foo.h> is allowed). Note that if constants C_name and C_header are absent, the Sather compiler generates layouts for the external C objects. If they are present, no layouts are generated and the necessary types must be defined in the specified header files. In this case, it is the responsibility of the programmer to ensure that attribute names are exactly as the structure filed names provided by the header files.

Examples

```
external C class BAR is
   attr bar_attr_int:C_INT;
   attr bar_attr_float:C_FLOAT;

   - the constructor is defined in C
   create_bar:BAR;
   - this routine that does some processing of bar is also
   - defined in   C
   process_bar(bar:BAR);
end;

- create an object of type BAR by calling an external
- C constructor
bar:BAR := BAR::create_bar;

- now pass "bar" back to C from processing
BAR::process_bar(bar);
```

In this example, the Sather compiler generates the layout for the external objec BARt. The corresponding C layout and prototypes of C functions that are called from Sather are below:

```
typedef struct {
   int integer_field;
   float float_field;
} *C_BAR;
/* Note that C names for the type and struct fields could be
   different from the corresponding names in Sather */

C_BAR create_bar();
void process_bar(C_BAR bar);
```

This is a similar example, but an existing C header file is used with Sather code:

```
external C class BAR is
```

```
   const C_name:STR := "C_BAR";   - C binding name for the type
   const C_header:STR := "bar.h <stdlib.h>";

   attr integer_field:C_INT;
   attr float_field:C_FLOAT;  - the constructor is defined in C
   create_bar:BAR;
     - this routine that does some processing of bar is also
     - defined in C
   process_bar(bar:BAR);
end;

- code that creates an object of type BAR by calling an external
- C constructor and then passes the object back to C
   bar:BAR := BAR::create_bar;

      - now pass "bar" back to C from processing
   BAR::process_bar(bar);
```

The C header "bar.h" contains the following:

```
typedef struct {
   int integer_field;
   float float_field;
} *C_BAR;
/* Note that C names for the type must be exactly as the binding C
   name specified by the C_name attribute.
   also, struct field names must be exactly the same as attribute
   names in the external C class*/

C_BAR create_bar();
void process_bar(C_BAR bar);
```

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

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

14.3.2. Attributes and C structs

Attributes and C structs

Attributes may be placed in external C classes; they are interpreted as fields of a C struct. If the layout of the class is generated by Sather (C_name and C_header symbolic constants are absent), then attributes can have any names. If a C layout from a header file specified by C_header is used, attribute textual names must be exactly the same as a struct filed names from a corresponding C type. It is the responsibility of the programmer to ensure this correspondence.

Points to note

- External C class attributes may only have built-in or user-defined external C types.
- Class constants do not contribute anything to the class layouts; all attributes do.

14.3.3. Shared Attributes and C globals

Global C variables may be accessed from Sather as shared attributes of external C classes. Such shared attributes must have names corresponding to those of C globals. Similar to constants, shared attributes do not contribute to the storage needed to layout the class objects.

```
external C class FOO is
   C_name:STR := "FOO";
   C_header:STR := "foo.h";

   shared foo:FOO:
   attr val:C_INT;

   - this is implemented in C
   create_foo:FOO;
end;

- accessing a global C variable
FOO::foo := FOO::create_foo;
FOO::foo.val := #(10);
```

```
#ifndef _FOO_H_
#define _FOO_H_

typedef struct {
   int val;
} *FOO;
```

```
FOO create_foo();
#endif  _FOO_H_

/* in some C file */
FOO foo;
```

14.4. Parameter Passing

The ANSI C standard prescribes that a copy is made of each call argument and all argument-passing is done strictly by value. To conform to ANSI C, all "in" arguments are passed by value. In the case of the built-in C types, a copy of a variable is passed. In the case of user defined external C types, a pointer to the object is copied and passed by value.

In addition, for extra flexibility, Sather supports `"out"` and `"inout"` argument modes for external C routines. `"out"` and "`inout"` arguments are passed by a pointer to a local, which may be legally modified by the called routine. The Sather implementation guarantee that such modifications cannot be observed until the routine returns. For C routines called from Sather this is guaranteed by emitting special code for the caller. For Sather routines that may be called from C, this is guaranteed by emitting special function prologues for the callee.

14.5. Inlining C Code

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 names.

Chapter 15. Statement and Expression Catalogue

This chapter presents a catalogue of statements and expressions in Sather and descriptions of them that originated in the specification. In some cases, these definitions are duplicated elsewhwere in the text. However, they have been included here, sometimes with more elaborate examples, as a convenient reference.

15.1. Statements

15.1.1. Assignment statements

Examples:

```
a := 5;
b(7).c := 5;
A::d := 5;
[3] := 4;
e[7,8] := 5;
g:INT := 5;
h ::= 5;
```

Assignment statements are used to assign objects to variables or attributes. The expression on the right hand side must have a return type which is a subtype of the declared type of the destination specified by the left hand side. When a reference object is assigned to a location, only a reference to the object is assigned. This means that later changes to the state of the object will be observable from the assigned location. Since immutable and closure objects cannot be modified once constructed, this issue is not relevant to them.

An assignment can also declare new local variables using the `::=` syntax.

The operation of assignment statements on attributes is described in the section on Attribute Accessor Routines. They are often syntactic sugar for function calls with one argument, which is the right hand side.

- Type inference in assignment statements: unnamedlink
- Attribute assignment sugar: unnamedlink
- Array element assignment: unnamedlink

- Immutable class attribute assignment on unnamedlink

15.1.2. `case` statements

Example:

```
case i
when 5,6 then ...
when j then ...
else ...
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 expressions tested in the branches of the if statement are the expressions of successive when lists. The first one of these calls to returns true causes the corresponding statement list to be executed and control passed to the statement following the case statement. If none of the when expressions matches and an else clause is present, then the statement list following the else clause is executed

There is one difference between the `case` statement and the equivalent `if` statement. If none of the branches of an `if` statement match and no `else` clause is present, then execution just continues onto the next statement after the `if` statement. However, if none of the branches of the `case` statement matches and there is no `else` clause, then a fatal run-time error will result.

Points to note

- It is a fatal error if no branch matches and there is no `else` clause.

See unnamedlink for a statement description.

15.1.3. `if` statements

Example:

```
if a>5 then
   foo;
elsif a>2 then
   bar;
else
```

```
   error;
end;
```

if statements are used to conditionally execute statement lists according to the value of a boolean expression. Each expression that is tested 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. Note that the else clause is not compulsory.

See unnamedlink for a statement description.

15.1.4. `protect` statements

Example:

```
protect
   < some statements >
when $STR  then
   #ERR + exception.str;
when FOO then
   #ERR + exception.foobar;
else
   < some statements >
end;
```

Exceptions may be explicitly raised by a program 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 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`.

See unnamedlink for more information.

15.1.5. `loop` statements

Example:

```
f: INT:=4; - Compute b factorial
res: INT := 1;
i :INT := 1;
loop until!(i > f);
   res := res * i;
   i := i + 1;
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; 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.) .

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.

See also: unnamedlink

15.1.6. `return` statements

Examples:

```
foo(a: INT): INT is
   return a*10;
end;
```

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 (see the chapter on Abstract Classes and Subtyping). 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 a routine with a return type is not a `return` or `raise` statement.

15.1.7. `typecase` statements

Example:

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

For a description of the typecase statement see unnamedlink

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 variable in the typecase ('a' in the above example) 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.

15.1.8. `yield` statements

Examples:

```
odd_upto!(n: INT): INT is
   i: INT := 0;
   loop until!(i = n);
      if i.is_odd then
         yield i;
      end;
```

```
      i := i + 1;
   end;
end;
```

`yield` statements are used to return control to a loop and may appear only in iterator definitions. The yield statement must be followed by a value if the iterator has a return value and must be absent if it does not. The value yielded 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. Yield causes assignment to `out` and `inout` arguments in the caller

In the example above the iterator yields odd numbers upto the specified value, "n".

See unnamedlink for more details.

15.1.9. `quit` statements

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

See unnamedlink for more details.

15.2. Expressions

We describe below a few special expressions used in Sather - `void`, `void()` and the short circuit boolean operations `or` and `and`.

15.2.1. `void` expressions

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 objects (except for immutable objects) is a special value that indicates the absence of an object - it is essentially the NULL pointer. Immutable objects are described in their own chapter, but for the sake of reference:

```
Class     Initial Value
---------------
INT       0
CHAR      '\0'
FLT       0.0
```

```
FLTD      0.0d
BOOL      false
```

For other immutable types the void value is determined by recursively setting each attribute and array element to `void`[1]. For numerical types, this results in the appropriate version of 'zero'.

`void` expressions may appear

- as the initializer for a constant or shared attribute. In fact, for most built-in classes, the only legal constant value is the void value e.g.

```
const a: POINT := void;
```

- 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 (see the chapter on Exceptions)
- as an argument value in a method call
- in a creation expression. In this last case, the argument is ignored in resolving overloading.

`void` expressions may not appear:

- as the left argument of the dot '`.`' operator.

  ```
  a: POINT := #POINT(3,3); - ILLEGAL (and silly) a.void
  ```

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. Calls on a void variable of an immutable type are, however, quite legal (otherwise you would not be able to dot into a false boolean or a zero valued integer!)

15.2.2. `void` test expressions

Example:

```
void(x)
```

Void test expressions evaluate their argument and return a boolean value which is true if the value is `void`.

```
p: POINT;
#OUT + void(p);   - Prints out true
p := #POINT(3,5);
```

```
#OUT + void(p);   - Prints out false
p := void;
#OUT + void(p);   - Prints out true;
b: BOOL;
#OUT + void(b);   - Prints out true
b := false;
#OUT + void(b);          - Prints out true!
- Even though b has been assigned, it has the void value
```

15.2.3. Short circuit boolean expressions: `and` and `or`

Example:

```
if (3>a and b>6) or (c = "Goo") then
   #OUT + "Success!"
end;
```

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 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.

For an expression description see unnamedlink

15.2.4. `exception` expressions

Example:

```
protect
   ... some code
when STR then
   #OUT + exception.str;
when ...
else ...
end;
```

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

Notes

1. The other built-in basic types are defined as arrays of BOOL and all have their values set to void by this rule.