The Sather 1.0 Specification

December 2, 1994

Steve Omohundro
David Stoutamire[1]

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

1. Direct email correspondence to David at `davids@icsi.berkeley.edu`

The Sather 1.0 Specification

INTRODUCTION

Sather is an object-oriented language that supports highly efficient computation, powerful abstractions for encapsulation and code reuse, a flexible interactive development environment, and constructs for improving code correctness. It has statically-checked strong typing, multiple inheritance, explicit subtyping which is independent of implementation inheritance, parameterized types, dynamic dispatch, iteration abstraction, higher-order routines and iters, garbage collection, exception handling, assertions, preconditions, postconditions, and class invariants. The development environment integrates an interpreter, a debugger, and a compiler. Sather code can be compiled into C code and can efficiently link with C object ("`.o`") files. This document is a terse but precise specification of Sather 1.0. "The Sather 1.0 Tutorial" presents more examples and motivations.

Data structures in Sather are constructed from *objects*, each of which has a unique *concrete type* that determines the operations that may be performed on it. The implementation of concrete types is defined by textual units called *classes*. *Abstract types* specify a set of operations without providing an implementation and correspond to sets of concrete types. Sather programs consist of classes and abstract type specifications. Each Sather *variable* has a declared type which determines the types of objects it may hold. Classes define the following *features*: *object attributes* which make up the internal state of objects, *shared* and *constant* attributes which are shared by all objects of a type, *routines* which perform operations on objects, and *iters* which encapsulate iteration. Features may be declared *private* to allow only the class in which they appear to access them. Accessor routines are automatically defined for reading object, shared, and constant attributes and for writing object and shared attributes. The set of non-private routines and iters in a class define the *interface* of the corresponding type. Abstract types directly specify their interfaces. Routine and iter definitions consist of *statements* and these are constructed from *expressions*. There are special *literal expressions* for boolean, character, string, integer, and floating point objects.

The following sections describe each of these constructs in turn. Most sections begin with an example of a syntactic construct followed by corresponding grammar rules. Multi-line examples are indented after the first line and three dots "..." indicate code that has been left out for clarity.

The grammar rules are expressed in a variant of Backus-Naur form. Non-terminal symbols are represented by strings of letters and underscores in an italic font and begin with a letter. The nonterminal symbol on the lefthand side of a grammar rule is followed by a double arrow "⇒ " and the right-hand side of the rule. The terminal symbols consist of Sather keywords and special symbols and are typeset in the Helvetica font. Vertical bars "*...|...* " separate alternatives, parentheses "*(...)*" are used for grouping, square brackets "*[...]*" enclose optional clauses and braces "*{...}*" enclose clauses which may be repeated zero or more times.

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

TYPES AND CLASSES

There are three kinds of objects in Sather: *value objects* (*e.g.* integers), *reference objects* (*e.g.* strings) and *bound objects* (Sather's version of closures). The corresponding types: *value*, *reference*, and *bound* types, are called *concrete types*. *Abstract types* represent sets of concrete types. *External types* specify interfaces to other languages.

The *type graph* for a program is a directed acyclic graph that is constructed from the program's source text. Its nodes are types and its edges represent the *subtype* relationship. If there is a path in this graph from a type t_1 to a type t_2, we say that t_2 is a *subtype* of t_1 and that t_1 is a *supertype* of t_2. Subtyping is reflexive; any type is a subtype of itself. Only abstract and bound types can be supertypes (see pages 8 and 27).

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

Operations are performed on objects by calling *routines* (page 13) and *iters* (page 14) on them. The *signature* of a routine consists of its name, the types of its arguments, if any, and its return type, if any. Iter signatures additionally specify that certain arguments are marked "!". This means that they will be re-evaluated on each iteration through a loop (page 14).

We say that the routine or iter signature *f* conflicts with *g* when

1. *f* and *g* have the same name,
2. *f* and *g* have the same number of arguments,
3. *f* and *g* either both return a value or neither does,
4. and each argument type in *f* is either equal to the corresponding argument type in *g* or one of the two types is either abstract or bound.

We say that the routine or iter signature *f* conforms to *g* when

1. *f* and *g* have the same name,
2. *f* and *g* have the same number of arguments,
3. the type of each argument of *g* is a subtype of the corresponding argument of *f* (*i.e.* contravariant conformance) and for corresponding arguments of iters, either both are marked "!" or neither is,
4. *f* and *g* either both return a value or neither does,
5. and if *f* and *g* return values, then the return type of *f* is a subtype of the return type of *g*.

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

Sather source files

Example:

```
type $COUNTRY is ... end; class USA < $COUNTRY is ... end;
```

Syntax:

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

Sather source files consist of semicolon separated lists of abstract type definitions and classes. Abstract types specify interfaces without implementation. Classes define types with implementions. Execution of a Sather program begins with a routine named "main" in a specified class (page 36).

Abstract type definitions

Example:

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

Syntax:

> *abstract_type_definition* ⇒ type *abstract_type_name*
> *[{* *parameter_declaration { , parameter_declaration }* } *]*
> *[subtyping_clause] [supertyping_clause]*
> is *[abstract_signature] { ; [abstract_signature] }* end
>
> *parameter_declaration* ⇒ *uppercase_identifier [* < *type_specifier]*
>
> *subtyping_clause* ⇒ < *type_specifier_list*
>
> *supertyping_clause* ⇒ > *type_specifier_list*
>
> *type_specifier_list* ⇒ *type_specifier {* , *type_specifier}*
>
> *abstract_signature* ⇒ *(identifier | iter_name)*
> *[* (*abstract_argument { ,* *abstract_argument }*) *] [* : *type_specifier]*
>
> *abstract_argument* ⇒ *identifier_list* : *type_specifier [* ! *]*

Abstract type definitions specify interfaces without implementations. Abstract type names must be entirely uppercase and must begin with a dollar sign "$" (page 32).

Abstract type definitions may be <u>*parameterized*</u> by one or more type parameters within enclosing braces. The scope of abstract type names is the entire program. Two abstract type definitions may define types with the same name only if they specify a different number of type parameters. Type parameter names are local to the abstract type 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 type 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.

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. An abstract type definition must satisfy all of the typing rules when its parameters are replaced by any subtype of their constraining types.

A subtyping clause adds to the type graph an edge from each type in the *type_specifier_list* to the type being defined. Each listed type must be abstract. Every type is automatically a subtype of $OB (page 37). 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 supertyping clause adds to the type graph an edge from the type being defined to each type in the *type_specifier_list*. These type specifiers may not be type parameters (though they may include type parameters as components) or external types. There must be no cycle of abstract classes such that each class appears in the supertype list of the next, ignoring the values of any type parameters but not their number. If both subtyping and supertyping clauses are present, then each type in the supertyping list must be a subtype of each type in the subtyping list using only edges introduced by subtyping clauses. This ensures that the subtype relationship can be tested by examining only definitions reachable from the two types in question.

The body of abstract type definitions consists of a semi-colon separated list of abstract signatures. Each specifies the signature of a routine or iter without providing an implementation. The argument names are for documentation purposes only and do not affect the semantics. The *abstract_signatures* of all types listed in the subtyping clause are included in the interface of the type being defined. Explicitly specified signatures override any conflicting signatures from the subtyping clause. If two types in the subtyping clause have conflicting signatures that are not equal, then the type definition must explictly specify a signature that overrides them. The interface of an abstract type consists of any explicitly specified signatures along with those introduced by the subtyping clause.

Classes

Examples:

```
class VIEWER{DATA < $VIEWER_DATA} is ... end
value class DOLPHIN < $MAMMAL, $SWIMMER is ... end
external class EXT is ... end
```

Syntax:

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

Classes define the types that have implementations: reference, value, and external types are defined by classes beginning with "class", "value class", and "external class", respectively. Class names must be entirely uppercase (page 32). Reference and value classes may be *parameterized* by one or more type parameters. The scope of class names is the entire program and two classes may have the same name only if they specify a different number of type parameters.

Class types may optionally declare one or more *type parameters* within enclosing braces. Type parameter names are local to the class definition in which they appear and they shadow non-parameterized types with the same name. Parameter names must be all uppercase, and they may be used within the class body as type specifiers. Whenever a parameterized type is referred to, its parameters are specified by type specifiers. The class behaves like a non-parameterized version whose body is a textual copy of the original class in which each parameter occurrence is replaced by its specified type.

If a parameter declaration is followed by a type constraint clause ("<" followed by a type specifier), then the parameter can only be replaced by subtypes of the constraining type. If a type constraint is not explicitly specified, then "< $OB" is taken as the constraint. A type constraint specifier may not refer to "SAME". The body of a parameterized class must be type-correct when the parameters are replaced by any subtype of their constraining types.

Subtyping clauses introduce edges into the type graph. The *type_specifier_list* must consist of only abstract types. There is an edge in the type graph from each type in the list to the type being defined. Every type is automatically a subtype of $OB (page 37).

Type specifiers

Examples:

```
A{B,C{$D}}
ROUT{A,B,C}:D
ITER{A,B!,C}
SAME
```

Syntax:

type_specifier ⇒ (*upercase_identifier* | *abstract_type_name*) [{ *type_specifier_list* }]
| (ROUT / ITER)
[{ *type_specifier* [!] { , *type_specifier* [!] } }]
[: *type_specifier*]
| SAME

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

- The name of a non-parameterized class or abstract type (*e.g.* "A" or "$A").
- The name of a parameterized class or abstract type followed by a list of parameter type specifiers in braces (*e.g.* "A{B,C}"). The parameter values must not cause the generation of an infinite number of types (*e.g.* FOO{FOO{T}} within the class FOO{T}).
- The name of a type parameter within the body of a parameterized class or abstract type definition (*e.g.* "T" in the body of "class B{T} is ... end").
- The keyword "ROUT" or "ITER" optionally followed by a list of argument types in braces, optionally followed by a colon and return type (*e.g.* "ROUT{A,B}:C"). Bound iter argument types may be followed by a "!" (page 27, *e.g.* "ITER{A!}:D").

- The special type specifier "SAME," which denotes the type of the class in which it appears. It may not appear in abstract type definitions.

CLASS ELEMENTS

Syntax:

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

The main body of each class is a semicolon separated list of *feature* definitions and include clauses. The possible features of a class are: *constant attributes*, *shared attributes*, *object attributes*, *routines* and *iters*. The semantics of a class is independent of the textual order of the class elements. All features are named and attributes may contribute a reader and a writer routine of the same name to the class interface. The scope of feature names is the class body and is separate from the class namespace. If a feature is *private*, then it may only be referred to from within the class and is not part of the class interface.

Constant attribute definitions

Examples:

```
const r:FLT:=45.6
private const a,b,c
```

Syntax:

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

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

1. a character, boolean, character, string, integer or floating point literal expression (page 34),
2. a void or void test expression (page 25),
3. an and or or expression (page 30), each of whose components is a constant expression,
4. an array creation expression (page 26), each of whose components is a constant expression,

5. a routine call applied to a constant expression, each of whose arguments is a constant expression (page 24),
6. or a reference to another constant in the same class or in another class using the "::" notation (page 24).

There must not be cyclic dependencies among constant initializers.

If a type specifier is not provided, then the construct defines one or more successive integer constants. The first identifier is assigned the value zero by default or its value may be specified by an integer expression. The remaining identifiers are assigned successive integer values.

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

Shared attribute definitions

Examples:

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

Syntax:

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

Shared attributes are variables that are directly accessible to all objects of a given type. When only a single shared attribute is defined by a clause, it may be provided with an initializing expression which must be a constant expression (page 11). If no initializing expression is provided, the shared is initialized the same as object attributes of that type would be (page 13).

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

Object attribute definitions

Examples:

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

Syntax:

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

An object's state consists of the *object attributes* defined in its class together with an optional array portion. The array portion appears if there is an include path (page 15) from the type to AREF for reference types or to AVAL for value types (page 37). Bound and reference objects must be explicitly allocated as described on pages 25 and 27. Variables have the value "void" until an object is assigned to them (page 25). There must be no cycle of value types where each type has an object attribute whose type is the next in the cycle. External classes may not define object attributes.

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

The writer routine takes different forms for reference and value types. For reference types, it takes a single argument whose type is the attribute's type and has no return value. Its effect is to modify the object by setting the value of the attribute. For value types, it takes a single argument whose type is the attribute's type, and returns a copy of the object with the attribute set to the specified new value, and whose type is the type of the object. This difference arises because it is not possible to modify value objects once they are constructed. Object attribute writer routines are private if and only if the corresponding attribute is declared either "private" or "readonly".

Routine definitions

Examples:

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

Syntax:

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

routine_argument ⇒ *identifier_list* : *type_specifier*

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

If a routine has arguments, the declaration list is enclosed in parentheses. The name and type of each argument is specified in this list. The types of consecutive arguments may be declared with a single type specifier. If a routine has a return value, it is declared by a colon and a specifier for the return type.

The optional "pre" construct contains a boolean expression which must evaluate to true whenever the routine is called; it is a fatal error if it evaluates to false. The expression may refer to self and to the routine's arguments.

The optional "post" construct contains a boolean expression which must evaluate to true whenever the routine returns; it is a fatal error if it evaluates to false. The expression may refer to self and to the routine's arguments. It may use "result" expressions (page 31) to refer to the value returned by the routine and "initial" expressions (page 31) to refer to values which are computed before the routine executes.

The body of a routine definition is a list of statements (page 16). The body is optional in external classes (page 38).

Iter definitions

Example:

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

Syntax:

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

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

Iters are similar to routines but encapsulate iteration abstractions. Their names end with an exclamation point "!" and they may only be called within loop statements (page 18). Iter argument type specifiers may be followed by a "!" to cause re-evaluation of that argument at each iteration.

The description of routine arguments and pre and post constructs also applies to iter definitions. Iters may contain yield (page 19) and quit (page 19) statements but may not contain return statements. The semantics of iter calls is described in the section on loop statements (page 18) . The pre clause must be true each time the iter is called and the post clause must be true each time it yields. The post clause is not evaluated when an iter quits.

include *clauses*

Examples:

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

Syntax:

> *include_clause* ⇒ [private] include *type_specifier*
> [*feature_modifier* { , *feature_modifier* }]
>
> *feature_modifier* ⇒ (*identifier* | *iter_name*) ->
> [[private / readonly] (*identifier* / *iter_name*)]

Implementation inheritance is defined by *include clauses*. These cause the incorporation of the implementation of the specified type, possibly undefining or renaming features with *feature_modifier* clauses. External classes may not have include clauses. The include clause may begin with the keyword "private", in which case any unmodified included feature is made private. We say that there is an *include path* from one type to another if there is a sequence of types between them such that each includes the next in the sequence.

The included type specified by the *type_specifier* must not be an external type, a bound type, or a type parameter (though type parameters may appear as components of the type specifier). There mustn't be include paths from reference types to AVAL or from value types to AREF (page 37). There must be no cycle of classes such that each class includes the next, ignoring the values of any type parameters but not their number.

Each *feature_modifier* clause specifies an identifier which must be the name of at least one feature in the included class. If no clause follows the "->" symbol, then the named features are not included in the class. If an identifier follows the "->" symbol, then it becomes the new name for the features. In this case, the listed features are included as part of the public interface unless they are specified as "private" or "readonly". Identifiers may only be renamed as identifiers and iter names may only be renamed by iter names.

A class may not explicitly define two routines or iters whose signatures conflict (page 6). A class may not define a routine whose signature conflicts with either the reader or the writer routine of any of its attributes (whether explicitly defined or included from other classes).

If a routine or iter is explicitly defined in a class, it overrides all conflicting routines or iters from included classes. The reader and writer routines of a class's attributes also override any included routines and must not conflict with each other. If an included routine or iter is not overridden, then it must not conflict with another included routine or iter. Feature modification clauses can be used to resolve any conflicts.

STATEMENTS

Syntax:

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

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

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

Declaration statements

Example:

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

Syntax:

declaration_statement ⇒ *identifier_list* : *type_specifier*

Declaration statements are used to declare the type of one or more local variables. Local variables may also be declared in assignment statements (page 17). The scope of a local variable declaration begins at the declaration and continues to the end of the statement list in which the declaration occurs. The scope of routine and iter arguments is the entire body of the routine or iter. Local variables shadow routines in the class which have the same name and no arguments. Within the scope of a local variable it is illegal to declare another local variable with the same name. Local variables are initialized to void (page 25) when the containing routine or iter is called.

Assignment statements

Examples:

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

Syntax:

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

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

1. *"identifier"*

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

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

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

3. *"[expression]* [*expression_list*] *"*

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

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

   This form both declares a new local variable and assigns to it (*e.g.* "g:INT:=5"). If a type specifier is not provided, then the declared type of the variable is the return type of the expression on the righthand side (*e.g.* "h::=5"). The scoping rules given on page 16 apply here as well. If a type is explicitly specified, the construct is syntactic sugar for a declaration statement followed by an assignment statement.

if *statements*

Example:

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

Syntax:

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

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

loop *statements*

Example:

```
loop ... end
```

Syntax:

loop_statement ⇒ loop *statement_list* end

Iteration is done with *loop statements*, used in conjunction with iter calls (page 24). An execution state is maintained for each textual iter call. When a loop is entered, the execution state of all enclosed iter calls is initialized. When an iter is first called in a loop, the expressions for self and for each argument without a "!" marking are evaluated left to right. Then the expressions for "!" arguments are evaluated left to right. On subsequent calls, only the expressions for "!" arguments are re-evaluated. self and any arguments not marked with a "!" retain their earlier values. The expressions for self and for arguments not marked "!" in an iter call may not themselves contain iter calls (such iters would only execute their first iteration).

When an iter 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 iter resume execution with the statement following the yield statement. If an iter 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 from the iter.

yield *statements*

Examples:

```
yield
yield x
```

Syntax:

yield_statement ⇒ yield *[expression]*

yield statements are used to return control to a loop and may appear only in iter definitions. The *expression* clause must be present if the iter has a return value and must be absent if it does not. If *expression* is present, then its type must be a subtype of the return type. Execution of a yield statement causes the expression to be evaluated and its value to be returned to the caller of the iter in which it appears.

quit *statements*

Example:

```
quit
```

Syntax:

quit_statement ⇒ quit

quit statements are used to terminate loops and may only appear in iter definitions. No value is returned from an iter when it quits. No statements may follow a quit statement in a statement list.

return *statements*

Examples:

```
return
return x
```

Syntax:

return_statement ⇒ return *[expression]*

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

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

case *statements*

Example:

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

Syntax:

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

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

typecase *statements*

Example:

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

Syntax:

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

An operation that depends on the runtime type of an object held by a variable of abstract type may be performed inside a *typecase statement*. The *identifier* must name a local variable or an argument of a routine or iter. If the typecase appears in an iter, then the *identifier* must not refer to a "!" argument, because 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 that 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.

assert *statements*

Example:

```
assert x>5
```

Syntax:

assert_statement ⇒ assert *expression*

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

protect *statements*

Example:

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

Syntax:

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

Sather uses *exceptions* to signal and recover from exceptional situations. Exceptions may be explicitly raised by a program (page 22) 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. If all exceptions which are raised are handled by other protect statements, then the statements in this list are executed to completion.

If an exception is raised which is not handled elsewhere, then the system finds the first type specifier listed in the "when" lists which is a supertype of the exception object type. The statement list following this specifier is executed and then control passes to the statement following the protect statement. An exception expression (page 30) 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 iter calls if they also contain the surrounding loop.

raise *statements*

Example:

```
raise x
```

Syntax:

raise_statement ⇒ raise *expression*

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

Expression statements

Example:

```
foo(1,2)
```

Syntax:

expression_statement ⇒ *expression*

A statement may consist of an expression (page 23) which doesn't return a value and is executed solely for its side-effects.

EXPRESSIONS

Syntax:

> *expression* ⇒ *self_expression* | *local_expression* | *call_expression* | *void_expression*
> | *void_test_expression* | *new_expression* | *create_expression* | *array_expression*
> | *bound_create_expression* | *sugar_expression* | *and_expression* | *or_expression*
> | *except_expression* | *initial_expression* | *result_expression* | *while!_expression*
> | *until!_expression* | *break!_expression* | *bool_literal_expression*
> | *char_literal_expression* | *str_literal_expression* | *int_literal_expression*
> | *flt_literal_expression*

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

self *expressions*

Example:

```
self
```

Syntax:

> *self_expression* ⇒ self

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

Local variable access expressions

Example:

```
a
```

Syntax:

> *local_expression* ⇒ *identifier*

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

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

Routine and iter call expressions

Examples:

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

Syntax:

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

expression_list ⇒ *expression* { , *expression* }

The most common expressions in Sather programs are *routine and iter calls*. The *identifier* names the routine or iter being called. The object to which the routine or iter is applied is determined by what precedes the *identifier*. If nothing precedes it, then the form is syntactic sugar for a call on self (*e.g.* "a(5,7)" is short for "self.a(5,7)"). If the *identifier* is preceded by an expression and a dot ".", then the routine or iter is called on the object returned by the expression. If *identifier* is preceded by a type specifier and a double colon "::", then the routine or iter is taken from the interface of the specified type with self initialized as described on page 13.

Routine calls are evaluated by first evaluating the expression to the left of the dot, if present, then evaluating any argument expressions from left to right and then calling the routine. The evaluation of iter calls is described on page 18.

Sather supports routine and iter *overloading*. In addition to the name, the number and types of arguments in a call and whether a return value is used contribute to the selection of the routine or iter. The *expression_list* portion of a call must supply an expression corresponding to each declared argument of the routine or iter. There must be a routine or iter with the specified name such that the type of each expression is a subtype of the declared type of the corresponding argument and it must be unique. If the routine or iter defines a return value, it must be used (*i.e.* the call may not be an *expression_statement*). Only non-private routines and iters may be called from outside a class, but all routines and iters may be called from inside a class.

Sather also supports *dynamic dispatch* on the type of self when the expression on which the call is made has an abstract declared return type. The routine or iter matching the call from the runtime type of the returned object is actually executed. Because of the subtyping rule (page 6), if the abstract type specifies a conforming routine or iter, so will the type of the returned object.

Direct calls of a type's routines or iters may be made using the double colon "::" syntax. The *type_specifier* must specify a reference, value, or external class. In such calls self has the default value described on page 13.

void *expressions*

Example:

```
void
```

Syntax:

void_expression ⇒ void

A *void expression* returns a value whose type is determined from context. void is the value that a variable of the type receives when it is declared but not explicitly initialized. The value of void for abstract, reference, and bound variables is a special value that represents the absence of a reference to an object. The value of void for boolean variables is false (page 34) and for other value types it is determined by recursively setting each attribute and array element to void. The built-in value types are defined in terms of arrays of BOOL and so have all their bits set to 0 by this rule.

void expressions may appear as the initializer for a constant or shared attribute, as the right hand side of an assignment statement, as the return value in a return or yield statement, as the value of one of the expressions in a case statement, as the exception object in a raise statement, or as an argument value in a routine or iter call or in a creation expression (page 26). In this last case, the argument is ignored in resolving overloading.

It is a fatal error to access object attributes of a void variable of reference type or to make any calls on a void variable of abstract type. It is not legal to dot into an explicit “void” expression.

void *test expressions*

Example:

```
void(a)
```

Syntax:

void_test_expression ⇒ void (*expression*)

Void test expressions evaluate their argument and return a boolean value which is true if the value is void (page 25).

new *expressions*

Examples:

```
new
new(17)
```

Syntax:

new_expression ⇒ new *[* (*expression*) *]*

<u>*new expressions*</u> are used to allocate space for reference objects and may only appear in reference classes. They return reference objects of type SAME. All attributes and array elements are initialized to void (page 25). If there is an include path from the type in which the new appears to AREF (page 37), then new must be provided with a non-negative INT argument which specifies the number of array elements in the returned object.

Creation expressions

Examples:

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

Syntax:

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

Value and reference object <u>*creation expressions*</u> are a convenient shorthand used for creating new objects and initializing their attributes. A creation expression is syntactic sugar for a call on a routine named "create" with the specified arguments. "self" is given the default value described on page 13 in this call. The type defining the "create" routine may be explicitly specified as a reference or value type. If the type is not explicitly specified, then it is taken to be the declared type of the context in which the call appears (and it must be a value or reference type). A type must be specified if the expression appears as the right-hand side of a "::=" assignment (page 17), as a routine or iter argument in which overloading resolution would otherwise be ambiguous, or as the object on which a call is made.

Array creation expressions

Examples:

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

Syntax:

array_expression ⇒ | *expression_list* |

Array creation expressions are used to create and directly specify the elements of an array object. The type is taken to be the declared type of the context in which it appears and it must be ARRAY{T} for some type T. An array creation expression may not appear as the righthand side of a "::=" assignment (page 17), as a routine or iter argument in which the overloading resolution is ambiguous, or as the object on which a call is made. The types of each expression in the *expression_list* must be subtypes of T. The size of the created array is equal to the number of specified expressions. The expressions are evaluated left to right and the results are assigned to successive array elements.

Bound routine and iter creation expressions

Examples:

```
#ROUT(2.plus(_))
#ITER(_:INT.upto!(5))
```

Syntax:

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

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

<u>*Bound routines and iters*</u> generalize the "function pointer" and "closure" constructs of other languages. They bind a reference to a routine or iter together with zero or more argument values (possibly including self).

The outer part of the expression is "#ROUT(...)" for bound routines and "#ITER(...)" for bound iters. These surround an ordinary routine or iter call in which any of the arguments or self may be replaced by the underscore character "_". Such unspecified arguments are <u>*unbound*</u>. Unbound arguments are specified when the bound routine or iter is eventually called. In forming a bound iter, all arguments marked "!" must be left unbound. Optional ":*type_specifier*" clauses are used to specify the types of underscore arguments or the return type and may be necessary to disambiguate overloaded routines or iters. If self is specified by an underscore without type information, the type is taken to be SAME.

The expressions in this construct are evaluated from left to right and the resulting values are stored as part of the bound routine or iter. Bound creation expressions return bound types. As described on page 10, the type specifiers for these types have the form:

bound_type_specifier ⇒ *(* ROUT | ITER *)*
[*{* *type_specifier* *[* ! *]* *{* , *type_specifier* *[* ! *]* *}* *}* *]*
[: *type_specifier* *]*

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

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

Bound types implicitly introduce edges into the type graph. There is an edge from each bound type *t1* to all bound types *t2* that satisfy the *contravariant* requirement that

1. Both *t1* and *t2* are routine types or both are iter types.
2. *t1* and *t2* have the same number of arguments, and either both have or both do not have a return value.
3. Each argument type in *t1*, if there are any, is a *subtype* of the corresponding argument type in *t2*. Also, in the case of iters, either both argument types are marked with "!" or both aren't (page 14).
4. The type of the return value, if any, in *t1* is a *supertype* of the corresponding return type in *t2*.

Syntactic sugar expressions

Examples:

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

Syntax:

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

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

As shown in the following table, several Sather constructs are simply *syntactic sugar* for corresponding routine calls. Each of these transformations is applied after the component expressions have themselves been transformed. The precedence ordering shown determines the grouping of these forms. Symbols of the same precedence associate left to right and parentheses may be used for explicit grouping.

Sugar form	*Translation*
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_leq(*expr2*)
expr1 = *expr2*	*expr1*.is_eq(*expr2*)
expr1 /= *expr2*	*expr1*.is_neq(*expr2*)
expr1 > *expr2*	*expr1*.is_gt(*expr2*)
expr1 >= *expr2*	*expr1*.is_geq(*expr2*)
- *expr*	*expr*.negate
~ *expr*	*expr*.not
[*expr_list*]	aget(*expression_list*)
expr1[*expression_list*]	*expr1*.aget(*expression_list*)
(*expression*)	*expression*

Table 1: Syntactic sugar expressions and their translations

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

Table 2: Precedence ordering of special symbols from strongest to weakest

and *expressions*

Example:

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

Syntax:

and_expression ⇒ *expression* and *expression*

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

or *expressions*

Example:

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

Syntax:

or_expression ⇒ *expression* or *expression*

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

exception *expressions*

Example:

```
exception
```

Syntax:

except_expression ⇒ exception

exception expressions may only appear within the statements of the then and else clauses in protect statements. They return the exception object that caused the when branch to be taken in the most tightly enclosing protect statement. The return type is the type specified in the corresponding when clause (page 21). In an else clause the return type is $OB.

initial *expressions*

Example:

```
initial(a)
```

Syntax:

initial_expression ⇒ initial (*expression*)

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

result *expressions*

Example:

```
result
```

Syntax:

result_expression ⇒ result

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

while! *expressions*

Example:

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

Syntax:

while!_expression ⇒ while!(*expression*)

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

until! *expressions*

Example:

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

Syntax:

until!_expression ⇒ until!(*expression*)

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

break! *expressions*

Example:

```
break!
```

Syntax:

break!_expression ⇒ break!

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

LEXICAL STRUCTURE

The character set used in Sather source files is implementation dependent, but it must include at least the characters which appear in the syntactic constructs in this specification. Many implementations will be based on ASCII, but this is not required. The case of characters in source files is significant. All syntactic constructs except identifiers and certain literals may be separated by an arbitrary number of *whitespace* characters and *comments*. The seven whitespace characters are space, tab, newline, vertical tab, backspace, carriage return, and form feed. Sather comments consist of two dashes "--" outside of a string (page 34) or character literal (page 34) and all following text until the end of the line.

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

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

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

abstract_type_name ⇒ $ *uppercase_letter {uppercase_letter | decimal_digit | _}*

iter_name ⇒ *[identifier]*!

letter ⇒ *lowercase_letter | uppercase_letter*

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

uppercase_letter ⇒ A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

decimal_digit ⇒ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Sather *keywords* are used to identify the fundamental syntactic constructs and may not be used as identifiers. The keywords are:

keyword ⇒ and | assert | attr | break! | case | class | const | else | elsif | end | exception | external | false | if | include | initial | is | ITER | loop | new | or | post | pre | private | protect | quit | raise | readonly | result | return | ROUT | SAME | self | shared | then | true | type | typecase | until! | value | void | when | while! | yield

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

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

In addition to the keywords "ROUT" and "ITER", there are several reserved names which may not be used to name user classes. Some of these are the names of built-in library classes known to the compiler, others are used in special situations as described on page 37.

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

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

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

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

special_featurenames ⇒ create | invariant | main

Finally, there are special lexical forms for literal expressions which define boolean, character, string, integer, and floating point values as described in the following sections.

Boolean literal expressions

Examples:

```
true
false
```

Syntax:

bool_literal_expression ⇒ true | false

BOOL objects represent boolean values (page 37). The two possible values are represented by the *boolean literal expressions*: "true" and "false".

Character literal expressions

Example:

```
'a'
```

Syntax:

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

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

CHAR objects represent characters (page 37). *Character literal expressions* begin and end with single quote marks. These may enclose either any single ISO-Latin-1 printing character except single quote or backslash or an escape code starting with a backslash.

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

String literal expressions

Examples:

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

Syntax:

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

STR objects represent strings (page 37). <u>*String literal expressions*</u> begin and end with double quote marks. The characters making up the string are specified in this construct from left to right. A backslash starts an escape sequence as with character literals. All successive octal digits following a backslash are taken to define a single character. Individual double-quote-bounded segments of string literals may not extend beyond a single line in the source text. However, successive quote bounded segments are concatenated together to form a single string and can be used to allow string literals to span more than one line of source code. They may also be used to force the end of an octal encoded character. For example: "\0367" is a one character string, while "\03""67" is a three character string. Such segments may be separated by comments and whitespace.

Integer literal expressions

Examples:

```
14
14i
-4532
39_832_983_298
0b101011
-0b_10111010_00101100_01010101
0o372363i
Ox_e98a_7c4d_65d7_6aa6_932d
```

Syntax:

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

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

binary_digit ⇒ 0 | 1

octal_int ⇒ 0o *{octal_digit* | *_}*

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

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

hex_int ⇒ 0x *{hex_digit* | *_}*

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

INT objects represent machine integers and INTI objects represent infinite precision integers (page 37). The literal form for INTI objects ends with a trailing "i". A leading "-" sign is used to denote a negative integer. Integer literals can be represented in four bases: binary is base 2, octal is base 8, decimal is base 10 and hexadecimal is base 16. These are indicated by the prefixes: "0b", "0o", nothing, and "0x" respectively. Underscores may be used within integer literals to improve readability and are ignored. INT literals are only legal if they are in the representable range.

Floating point literal expressions

Examples:

```
12.34
3.4e-8d
3.498_239_832_932_988_9e22x
2.2i
```

Syntax:

flt_literal_expression ⇒ *[-] decimal_int* . *decimal_int [*e *[-] decimal_int] [*d | x | dx | i*]*

FLT, FLTD, FLTX, and FLTDX objects represent floating point numbers according to the single, double, extended, and double extended representations defined by the IEEE-754-1985 standard and FLTI objects represent arbitrary precision floating point numbers (page 37). *Floating point literal expressions* of these types are indicated by the suffixes: nothing, "d", "x", "dx", and "i", respectively. 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.

SPECIAL FEATURES

This section describes several features of classes that are automatically defined or have special properties.

invariant

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

main

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

BUILT-IN CLASSES

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

- $OB is automatically a supertype of every type. Variables declared by this type may hold any object. It has no features.
- AREF{T} is a reference array class. Any reference class which includes it obtains an array of elements of type T in addition to any attributes it has defined. In such classes, new has a single integer argument that specifies the size of the array portion. It defines routines and iters named: asize, aget, aset, aclear, acopy, aelts!, aset_elts!, and ainds!. Array indices start at zero.
- AVAL{T} is the value class analog of AREF. Classes which include AVAL must define asize as an integer constant which determines the size of the array portion.
- ARRAY{T} defines general purpose array objects. They may be directly constructed by array creation expressions (page 26).
- TUP names a set of parameterized value types called tuples, one for each number of parameters. Each has as many attributes as parameters and they are named "t1", "t2", etc. Each is declared by the type of the corresponding parameter (*e.g.* "TUP{INT,FLT}" has attributes "t1:INT" and "t2:FLT"). It defines create with an argument corresponding to each attribute.
- BOOL defines value objects which represent boolean values. The initial value is false.
- CHAR defines value objects which represent characters. The number of bits in a CHAR object must be less than or equal to the number in an INT object. The initial value is '\0'.
- STR defines reference objects which represent strings.
- INT defines value objects which represent machine-dependent integers. The size is implementation dependent but must be at least 32 bits. The two's complement representation is used to represent negative values. Bit operations are supported in addition to numerical operations.
- INTI defines reference objects which represent infinite precision integers.
- FLT, FLTD, FLTX, and FLTDX define value objects which represent floating point values according to the single, double, extended, and double extended representations defined by the IEEE-754-1985 standard.
- FLTI defines reference objects which represent arbitrary precision floating point objects.
- EXT_OB is used to refer to "foreign pointers". These might be used, for example, to hold references to C structures. Such pointers are never followed by Sather and are treated essentially as integers which disallow arithmetic operations. They may be passed to external routines.

- SYS defines a number of routines for accessing system information. tp(ob:$OB):TYPE returns the type of an object. destroy(ob:$OB) explicitly deallocates an object (Sather is garbage collected and this is only done for efficiency reasons in special circumstances). id(ob:$OB):INT returns an integer associated with a particular object. ob_eq(o1,o2:$OB):BOOL is used to test two objects for equality.
- $REHASH defines the single routine rehash. Any class whose objects need to perform special operations when they are moved or copied should be a subtype of it. The rehash routine is called on such objects if the system changes their location during garbage collection.

INTERFACING WITH OTHER LANGUAGES

External classes are used to interface with code from other languages. Each external class is typically associated with an object file compiled from a language like C or Fortran. External classes do not support subtyping, implementation inheritance, or overloading. External class bodies consist of a list of routine definitions. Routines with no body specify the interface for Sather code to call external code. Routines with a body specify the interface for external code to call Sather code.

Each routine name without a body may only appear once in any external class and the corresponding external object file must provide a conforming function definition. Sather code may call these external routines using a class call expression of the form EXT_CLASS::ext_rout(5). External code may refer to an external routine with a body by concatenating the class name, an underscore, and the routine name (*e.g.* EXT_CLASS_sather_rout).

Only a restricted set of types are allowed for the arguments and return values of these calls. The built-in value types BOOL, CHAR, INT, FLT, FLTD, FLTX, and FLTDX are allowed anywhere and on each machine have the format supported by the C compiler used to compile Sather for that machine. The type "EXT_OB" is also allowed anywhere and is used to reference storage allocated by the external language. Sather cannot follow these pointers.

To enhance the efficiency of the interface, the arguments of external routines without bodies may also be declared by types which have include paths to AREF{CHAR}, AREF{INT}, AREF{FLT}, AREF{FLTD}, AREF{FLTX}, AREF{FLTDX}, or AREF{EXT_OB}. When a Sather program calls such a routine, the external routine is passed a pointer into just the array portion of the object; if void is passed, the C routine will receive a NULL pointer. 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. These restrictions help to ensure that the Sather type system and garbage collector will not be corrupted by external code while not sacrificing efficiency for the most important cases.

ACKNOWLEDGEMENTS

Sather has adopted ideas from a number of other languages. Its primary debt is to Eiffel, designed by Bertrand Meyer, but it has also been influenced by C, C++, Cecil, CLOS, CLU, Common Lisp, Dylan, ML, Modula-3, Oberon, Objective C, Pascal, SAIL, School, Self, and Smalltalk. Many people have been involved in the language design discussions including: Subutai Ahmad, Krste Asanovic, Jonathan Bachrach, David Bailey, Joachim Beer, Jeff Bilmes, Peter Blicher, John Boyland, Matthew Brand, Henry Cejtin, Richard Durbin, Jerry Feldman, Carl Feynman, Ben Gomes, Gerhard Goos, Robert Griesemer, Hermann Haertig, Ari Huttunen, Roberto Ierusalimschy, Phil Kohn, Franz Kurfess, Chu-Cheow Lim, Franco Mazzanti, Stephan Murer, Thomas Rauber, Steve Renals, Noemi de La Rocque Rodriguez, Hans Rohnert, Heinz Schmidt, Carlo Sequin, Andreas Stolcke, David Stoutamire, Clemens Szyperski, Martin Trapp and Bob Weiner.