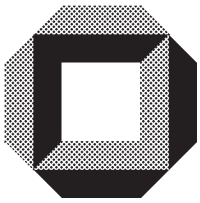
Sather-K

The Language

Gerhard Goos

Report 8/95



Universität Karlsruhe

Fakultät für Informatik

Gerhard Goos
Universität Karlsruhe, Fakultät für Informatik
und
Forschungszentrum Informatik Karlsruhe

Vincenz-Prießnitzstr. 3
D-76128 Karlsruhe, Deutschland
Netz: ggoos@informatik.uni-karlsruhe.de

Sather-K,
Version 1.3, May 1, 1995

Sather-K is a version of Sather, developed for teaching classical as well as object-oriented programming and design techniques, and as a basis for reusable class libraries and object-oriented frameworks. This report gives a precise language description for implementors.

Contents

1 Introduction

Sather-K is an object-oriented programming language based on the design of the language Sather (Omohundro 1991; Omohundro and Stoutamire 1994) developed by Stephen Omohundro in 1990/94 at the International Computer Science Institute at the University of California at Berkeley. This design in turn was based on the original design of the programming language Eiffel (Meyer 1988) for achieving higher efficiency of the generated code.

Sather-K has been designed for teaching classical as well as object-oriented programming and design techniques, and as a basis for reusable class libraries and object-oriented frameworks. This report gives a precise language description for implementors. It is not intended as a programmer's manual.

We give a traditional language description in the interpretive style. The syntax is described in extended Backus Naur form as described in the appendix of (Waite and Goos 1984). Rules marked with an asterisk are given for introducing terminology only; they are not used in derivations from the syntactic target program.

A program specifies a computation by describing a sequence of actions. A computation may be realized by any sequence of actions having the same effect as the one described here for the given computation. The meaning of constructs that do not satisfy the conditions given in this report is undefined. The meaning of constructs is also undefined if this report does not explicitly specify a meaning. Whether, and in what manner, a particular implementation assigns meaning to undefined constructs is outside the scope of this description.

Notes, hints, examples and problems are not part of the language description but describe motivations for certain decisions, implementation hints, examples and open questions which still need to be resolved.

Acknowledgements: This report owes a lot to contributions by Stephen Omohundro, and Markus Armbruster, Nigel Horspool, Peter Lockemann, Welf Löwe, Rainer Mauch, Martin Odersky, Erhard Plödereder, Arnd Pötsch-Heffter, Christian von Roques, Martin Trapp, Franz Weber, and Wolf Zimmermann.

2 Overview

Sather-K is an imperative, object-oriented language. A program text consists of class declarations which serve as possibly parametrized type templates for objects. Class declarations define *classes*, i.e. sets of features, i.e. attributes and methods. The values of the attributes make up the state of objects of the class. The methods define operations applicable to such objects and their attributes.

Sather-K considers all attributes to be private to the object to which they belong. Whenever an attribute is to be accessed from outside then this access is replaced by a method call for reading or writing the attribute value. The corresponding methods are automatically supplied by the implementation. Thus, a class defines an *interface* for its objects which consists of methods only.

Within the language the name space of classes is flat. Modules consisting of several class declarations may be formed by help of Sather-K's configuration manager, a tool which provides flexible information hiding also for class names.

The execution of a Sather-K-program therefore consists of creating a number of objects which then may call their own or other object's methods. The state of the computation at any time is given by the state of all the objects.

Class declarations are either parametrized or simple; in the latter case they correspond one-to-one to classes. Parametrized classes especially serve for constructing data structures in which the type of the

elements or other properties which are not of primary concern can be specified separately. This feature is one of the key issues when constructing reusable libraries.

Classes may be inherited, i.e. the declarations which make up the body of the class are literally copied into a *subclass*. Sather-K allows for multiple inheritance, i.e. there might be more than one class which is inherited by the same subclass. For simplifying matters we always call a class a subclass of itself.

In Sather-K inheritance is available in two forms: *Include inheritance* copies all or some of the declarations of a class into a subclass; it corresponds to simple code reuse. *Subtype inheritance* additionally requires that the subclass *conforms* to the superclass: It must be possible to use the features of a subclass and its objects in all contexts where the superclass would be admissible.

Subtyping is the basis for polymorphic feature access. There are two kinds of types available for attributes and local entities of methods: By normal or *monomorphic* types T we declare entities whose values always carry the same type. By *polymorphic* types $\$T$ we declare entities a whose values may carry any of the conforming subtypes of T including T itself; whenever a feature x of a is accessed then it is selected according to the type of the current value of a. This polymorphic access is also called *dynamic dispatch*.

Sather-K is statically type safe: It is possible to statically check that no access to an object and no method call will result in a typing error at run-time.

2.1 Values, Objects, Classes and Types

There are three kinds of objects: reference objects, value objects, and stream objects, and four kinds of values: simple values, composite values, references and bound methods. Each such object or value has a type which determines the operations applicable to it.

2.1.1 Values

Simple values are immutable abstract quantities upon which operations may be performed.

In Sather-K the simple values are integer and floating point numbers, characters, the Boolean values false and true which also serve as the simple values from which bit-sequences are built.

Additionally Sather-K has *reference values* which refer to reference objects and stream objects.

Composite values, e.g., strings, are values composed from other values, especially from simple ones. The value of such an object consists of the values of its attributes. Arrays have an indexed set of unnamed attributes.

Bound methods are values ⟨method, bound parameters⟩ which allow to determine the method to be called at "binding" time rather than at calling time. At binding time some or all of the arguments of the future calls may be specified as bound arguments; the remaining parameters are called *free parameters* (of the bound method).

2.1.2 Objects and Methods

A *reference* or a *value object* is a collection of *features*, i.e. attributes or methods. An attribute is an entity denoting a value. A method is a prescription for manipulating objects and their attributes. An object carries *identity* and *state*. The identity of an object is called its *reference*. The state consists of the values of its attributes.

There are two kinds of methods: *routines* are methods which upon call deliver a result or change the state of the computation. *Streams* are methods for sequentially accessing the elements of data structures, e.g. arrays or trees, in an order defined by the stream. To this end a call establishes a *stream object* carrying the state information for the stream. Thus, a stream object consists of a bound stream method and its state. The stream object may be explicitly established and assigned to a local entity, or it may be implicitly established during the first call of the stream. The latter is then called a *direct (stream) call* (cf. 6.7.2)

The reference of an object is a value created upon creation of the object. In general, it can be copied, assigned and compared with other values. An object is a *value object* if its references do not permit these operations.

For reference objects the basic operations assignment and comparison mean the assignment of (a copy of) the object reference and the comparison of such references respectively. For value objects assignment means moving a copy of the current value (state) of the object into the container representing the variable. Comparison of simple value objects yields true iff they have the same type and the same value; composite value objects are compared element by element.

Especially, all simple values are value objects with one unnamed attribute denoting its value and operations applicable to this value. An integer number such as 17 is a denotation for a (constant) object and at the same time denotes the sole attribute of this object.

2.1.3 Entities

An *entity* is a name denoting a *quantity*, i.e. a value, an object or a method.

Each entity has a type specifying the quantities which it may denote. This might be an class type, or a method type. It may additionally be a *polymorphic type* $\$C$, i.e. the entity may not only denote values or references of type C but also of any conforming subtype of C.

Except for methods an entity is usually a *variable*, i.e. the values which it may denote may be altered by help of an assignment, cf. 6.1. If the specification of the entity is prefixed with the keyword const then the entity is called a *(dynamic) constant*; only the assignment given by the initialization is allowed and further assignments are forbidden.

2.1.4 Types and Classes

Each reference or value object has a *class type* which specifies its features. A class type uniquely corresponds to a class and its class specifier. Hence the terms *type of a reference* or *value object*, *class* and *class specifier* may be used interchangeably.

Each method has a *signature* indicating the number, kinds and types of its parameters and result, if any. The *parameter kind* is in, out, inout or const. In and const parameters are transmitted through call-by-value. Out parameters are local entities of the method whose values are transmitted through call-by-result when the method call returns to the caller; the corresponding argument must be a variable. Inout parameters are in parameters as well as out parameters, i.e. they are transmitted by cal-by-value-and-result. Const parameters are permitted for stream methods only. A const parameter is assigned only once when the stream object is established and is not reassigned at later stream calls. The signature of a method is called a *method type*. A bound method has a method type in which only the free parameters are specified.

Each stream object has a *stream object type*. It is derived from the (bound) stream type used to create the stream object but additionaly indicates that the stream is in progress.

The value of an object and the reference to a stream or reference object carry the same type as the object. The class (type) of a reference or value object is called a *reference* or *value class* (*type*) respectively.

Classes and stream types for which objects can be created are called *concrete classes* and *types* respectively. For use in the context of inheritance there exist also *abstract classes* (*types*) which do not necessarily describe all the details of the features of objects. They may be used for forming polymorphic types. Additionally there are *external classes* for describing the interface between Sather-K and other parts of a software system. External classes can neither be abstract nor value nor reference classes.

Reference classes cannot be subtypes of value classes and vice versa. The predefined class OB is a superclass of all reference and value classes, and is exceptional in this respect.

Simple value classes are either predefined, cf. 8.2.2, or user defined. A user defined simple value class is a class that inherits a simple value class. A class must not transitively include more than one predefined simple simple value type and must not define attributes.

A numeric type is given by a simple value class which inherits one of the predefined numeric types, cf. 8.2.2.

For external classes see 9.1.2.

Problem: The existence of value and reference types besides each other, the fact that in the one case we have to create the objects explicitly in the other case not, and the differences in interpretation of the assignment and comparison operation constitute the most annoying properties of most imperative object-oriented languages. These properties are only justifiable from the efficiency point of view. ♦

2.1.5 Type Conformance

The *type dependence graph TDG* of a program is an (acyclic) graph whose nodes are the class types occurring in the program. There are two kinds of arrows in *TDG*: $T \rightarrow T'$ indicates that T' is a conforming subtype of T. $T \rightsquigarrow T'$ indicates that T' structurally conforms to T but is not necessarily a subtype, i.e. the subtype inheritance clause may be missing but otherwise the same requirements are fulfilled.

The precise definition of conformance and structural conformance is recursive:

A class type T' *conforms* to a class type T if

- T' and T are the same type, or if

- T is a polymorphic type, $T = \$\,\hat{T}$, and T' is a conforming subtype of $\hat{T}$.

A type T' is a *conforming subtype* of a type T, if

- T' and T are the same type, or if
- T', T are both class types, and
  1. class T' contains an inheritance clause subtype of T'' for a conforming subtype T'' of T, and
  2. for every method g visible at the interface of T there exists a method f visible at the interface of T' and f conforms to g.

A class type T' *structurally conforms* to a class type T if all the conformance conditions are met except that the inheritance clause subtype of T may be missing.

A method f *conforms* to a method g if

- both have the same name, and
- both define a routine or both define a stream, and
- both have the same number of parameters, and for each pair of parameters $x{:}\,S'$ of f, $y{:}\,S$ of g
  1. the kind of the parameter, i.e. in-, out, inout- or const-parameter, is the same, and
  2. if x is an in- or const-parameter then S conforms to S' (*contravariant conformance*),
  3. if x is an out-parameter then S' conforms to S (*covariant conformance*),
  4. if x is an inout-parameter then S, S' are the same type,
- if g has a result type R' then f has a result type R conforming to R' (*covariant conformance*).

A method type T' conforms to a method type T if any method of type T' conforms to a method of type T provided they have the same name.

A stream object type conforms to another one if the stream types from which these objects are established conform.

These definitions are also applicable to the methods generated from attribute declarations within classes, cf. 4.5.

3 Basic Symbols

```
1  *  basic_symbol ::= identifier | denotation | delimiter .
```

The representation of a Sather-K program is a sequence of basic symbols interspersed by newline and spacing characters.

3.1 Identifiers

```
2   identifier ::= letter ('_' | letter | digit)* .
3   letter ::= 'a' | 'A' | 'b' | 'B' | 'c' | 'C' | 'd' | 'D' |
4              'e' | 'E' | 'f' | 'F' | 'g' | 'G' | 'h' | 'H' |
5              'i' | 'I' | 'j' | 'J' | 'k' | 'K' | 'l' | 'L' |
6              'm' | 'M' | 'n' | 'N' | 'o' | 'O' | 'p' | 'P' |
7              'q' | 'Q' | 'r' | 'R' | 's' | 'S' | 't' | 'T' |
8              'u' | 'U' | 'v' | 'V' | 'w' | 'W' | 'x' | 'X' |
9              'y' | 'Y' | 'z' | 'Z' | special_letter .
10  digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' .
```

An identifier is a freely chosen representation for an *entity*, i.e. a class, a feature of a class or a local entity within a method. The meaning of an identifier within a program is prescribed by *defining occurrences* of the identifier. All other appearances of the identifier are called *applied occurrences*.

Example: MAIN_CLASS attribute_name method_name local ♦

Hint: Implementations may require that identifiers used as class names do not contain lower case letters. ♦

Hint: Within an external class the syntax may be adapted to suit the conventions of other programming languages. ♦

Hint: Which characters are considered special letters is implementation dependent. E.g., when using ISO 8859-1 encoding then the sedecimal values 0xC0-0xD6, 0xD8-0xF6, 0xF8-0xFF print as letters of certain European alphabets and are considered special letters. Special letters should not occur in portable classes. ♦

Letters are case sensitive: const is a reserved keyword whereas CONST is a freely chosen identifier.

3.2 Predefined Identifiers

```
11  * keyword ::= 'abstract' | 'and' | 'assert' | 'begin' | 'bind' | 'break' |
12              'case' | 'class' | 'const' | 'deferred' | 'div' |
13              'else' | 'elsif' | 'end' | 'except' | 'external' |
14              'false' | 'if' | 'include' | 'Inf' | 'is' |
15              'loop' | 'mod' | 'NaN' | 'not' | 'of' | 'or' |
16              'pragma' | 'private' | 'raise' |
17              'readonly' | 'resume' | 'return' | 'routine' |
18              'shared' | 'stream' | 'subtype' | 'then' | 'true' | 'typecase' |
19              'value' | 'void' | 'when' | 'while' .
20  * special_feature_names ::= 'aget' | 'aset' | 'arg' | 'copy' |
21                            'destroy' | 'downto' | 'exception' | 'invariant' |
22                            'main' | 'res' | 'self' | 'str' |
23                            'terminated' | 'type' | 'until' | 'upto' .
24  * special_class_names ::= 'EXCEPTION' | 'EXTOB' | 'OB' | 'ARR' | 'ARRAY' |
25                          'BITS' | 'BOOL' | 'BYTE' | 'CHAR' | 'FILE' |
26                          'FLT' | 'FLTD' | 'INT' |
27                          'INTINF' | 'LONG_INT' | 'LONG_UNSIGNED' |
28                          'REFERENCE' | 'ROW' |
29                          'SHORT_INT' | 'SHORT_UNSIGNED' |
30                          'STR' | 'STRING' | 'SYSTEM' | 'TEXT' 'TYPE' |
31                          'UNSIGNED' | 'VALUE' | 'SAME' .
32  * special_exception_names ::= 'FLOAT_OVERFLOW' | 'FLOAT_UNDERFLOW' |
33                              'INTEGER_OVERFLOW' | 'ZERO_DIVIDE' |
34                              'NUMERIC_ERROR' |
35                              'INDEX_ERROR' | 'REFERENCE_ERROR' |
36                              'INVARIANT_ERROR' | 'ASSERTION_ERROR' |
37                              'STREAM_TERMINATION' | 'EXIT' .
```

Keywords, special feature names, special class names and special exception names are identifiers with predefined meanings. Keywords are reserved and can only be used as prescribed in the syntax of the language. The other ones can be partially redefined by the programmer.

3.3 Denotations

```
38  denotation ::= integer_number | floating_number | Boolean_value |
39               character_constant | string_constant |
40               bit_constant | 'void' .
41  integer_number ::=  digit+ ( '_' digit+ )* .
42  floating_number ::= integer_number [ '.' integer_number]  scaling  |
43                      integer_number '.' integer_number | 'NaN' | 'Inf' .
44  scaling ::= ('E' | 'e') ['+' | '-' ] integer_number .
45  Boolean_value ::= 'true' | 'false' .
```

```
character_constant ::= ''' character ''' .
character ::= all printable characters except \ , ' and " |
              '\\' | '\'' | '\"' | '\e' | '\b' | '\t' | '\n' | '\v' | '\f' | '\r' |
              '\' octal_digit octal_digit octal_digit |
              '\' [ 'x' | 'X' ] sedecimal_digit sedecimal_digit .
string_constant ::= '"' character* '"' ('"' character* '"')
* .
bit_constant ::= octal_number | sedecimal_number .
octal_number ::= '0' ( 'o' | 'O' ) octal_digit+ ( '_' octal_digit+ )* .
octal_digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' .
sedecimal_number::= '0' ( 'x' | 'X' ) sedecimal_digit+ ( '_' sedecimal_digit+ )* .
sedecimal_digit ::= digit |
                    'a' | 'b' | 'c' | 'd' |'e' |'f' |
                    'A' | 'B'| 'C' | 'D' | 'E' | 'F' .
```

A '_' occurring in a number is ignored.

Integer and floating-point numbers have the usual meaning. They are of type UNIVERSAL_INT and UNIVERSAL_REAL respectively which are of finite but unbounded size and precision. These types can only be used for computing expressions which are constant at compile-time. Otherwise values of these types are coerced to the appropriate numeric type depending on context, cf. 7.2. The values NaN (Not a Number) and Inf have the meaning as defined in IEEE Standard 754-1985 (Binary Floating Point Arithmetic). NaN denotes an arbitrary signalling NaN; note that, according to the standard, the comparison NaN = NaN always yields false.

Character constants denote values of type CHAR consisting of one character only. As in the language C the character \ assigns to the subsequent character \ , ' or " its original meaning. \ followed by one of the characters a, b, t, n, v, f or r has the meaning of the control characters *bell* (or *alert*) (0x07), *backspace* (0x08), *tab* (0x09), *line feed* (0x0A), *vertical tab* (0x0B), *form feed* (0x0C), and *carriage return* (0x0D) respectively. '\' followed by three octal digits or by x and 2 sedecimal digits denotes the character whose integer encoding is given by the octal or sedecimal number.

String constants denote values of type STR which are sequences of characters. Empty strings and strings consisting of one character only are permitted.

Hint: Strings are implemented following the C conventions: a string in memory is terminated by the first occurrence of the NUL character. Hence a string of length *n* requires at least *n* + 1 bytes of storage. Sorry we know that this is dull but we cannot change the system conventions. ♦

Hint: Implementations may choose to provide multibyte characters. Hence programs cannot assume that a character is always encoded by one byte. ♦

Bit constants are written as octal or sedecimal numbers and represent values of type UNIVERSAL_BITS. Values of this type are coerced to the appropriate type depending on context, cf. 7.2.

Note: Universal types free the programmer from the burden to change the notation of constants when switching between different representations of integers or reals. Bit constants can be used for a variety of purposes; in some cases, e.g., when representing characters or strings in binary form, the meaning of bit constants is implementation dependent. ♦

The constant void denotes the *empty reference* and simultaneously belongs to all stream object, reference and polymorphic value types.

3.4 Delimiters

```
* delimiter ::= special | keyword | comment | pragma .
* special ::= '+' | '-' | '*' | '/' | '%' | '^' |'=' | '<' | '>' |
              '(' | ')' | '[' | ']' | '{' | '}' | ',' |
              '.' | ';' | ':' | '$' | '#' | '@' | '!' | '&' | '&&' |
              '/=' | '<=' | '>=' | '->' | '-<' | ':=' | '::' |
              other_special .
 other_special ::= '<<' | '>>' | '\' | '|' | '~' | '`' | '?' .
* comment ::= '- -' arbitrary characters up to end of line .
* pragma ::= 'pragma' arbitrary characters up to end of line .
```

Delimiters have the meaning ascribed to them by the syntax of the language, or they serve as operator symbols replacing certain routine calls, cf. 4.5.

The comment sign -- starts a character sequence which is ignored up to the following end of line. A character sequence starting with the keyword pragma up to the following end of line is semantically treated as a comment; it may, however, contain control information for an compiler or other tools in the programming environment; its meaning is thus implementation dependent.

Outside of character and string constants and comments other_specials can only be used as operators, cf. 4.5.

Outside of character and string constants the control characters HT (0x09), LF (0x0A), VT (0x0B), FF (0x0C), CR (0x0D) and the space character (0x20) may occur arbitrarily often and serve to separate basic symbols; otherwise they carry no meaning.

Hint: Note that in certain situations such separators must be used since implementations will follow the principle of the longest match. Hence if1 denotes the identifier if1 and not the keyword if followed by a number starting with the digit 1. ♦

4 Programs

```
69  program ::= [ class_declaration ] ( ';' [ class_declaration ] )* .
```

Example: class MAIN is . . . end; class TREE(T<ORDERED) is . . . end ♦

The text of a program in Sather-K is a collection of class declarations. It is specified by designating a class declaration K as the *main class*. K contains the main program which is a routine main. K must be a concrete class (cf. 2.1).

Hint: The classes belonging to a program may be partially stored within the same file or may be distributed across many files. The language does not determine how to find those files. ♦

4.1 Classes and Types

```
70  class_declaration ::= ( [ 'external' ] | [ 'abstract' ] [ 'value' ] )
71                        'class' parametrized_class 'is' class_body 'end' .
72  parametrized_class ::= class_identifier
73                         [ '[' class_bounds ']' ] [ '('class_parameters ')' ] .
74  class_identifier ::= identifier .
75  class_bounds ::= identifiers | '*' ( ',' '*' )* .
76  identifiers ::= identifier ( ',' identifier )* .
77  class_parameters ::= class_parameter ( ';' class_parameter )* .
78  class_parameter ::= identifier [ '<' type_bound ] .
79  type_bound ::= type_specifier .
80  type_specifier ::= class_specifier | 'SAME' | bound_method_type | stream_type |
81                     polymorphic_type .
82  polymorphic_type ::= '$' class_specifier | '$' SAME .
83  class_specifier ::= identifier [ '[' bounds ']' ] [ '(' class_arguments ')' ] .
84  bounds ::= bound ( ',' bound )* .
85  bound ::= '*' | expression .
86  class_arguments   ::= class_argument ( ',' class_argument )* .
87  class_argument    ::= type_specifier .
88  class_body        ::= [ inheritances ] [ feature_declarations ] .
89  inheritances      ::= inheritance ( ';' inheritance )* ';' .
90  inheritance       ::= subtype_inheritance | include_inheritance .
91  subtype_inheritance ::= 'subtype' 'of' class_specifier [ ':' modifications ] .
92  include_inheritance ::= [ 'private' ] 'include' class_specifier [(
93  ':' | '::') modifications ] .
94  modifications ::= modification ( ',' modification )* .
95  modification ::= feature_specification [ '->' [ identifier ] ] .
96  feature_specification ::= identifier [ method_specification ].
97  method_specification ::= [ '(' parameter_types ')' ] [ ':' result_type ] .
98  parameter_types ::= parameter_type ( ',' parameter_type )* .
```

```
 99  parameter_type ::= parameter_kind type_specifier .
100  parameter_kind ::= [ 'const' | '&' | '&&' ] .
101  result_type ::= type_specifier .
102  bound_method_type ::= ( 'routine' | 'stream' ) method_specification .
103  stream_type ::= 'stream' '!' method_specification .
```

For feature declarations see 5.

Examples:

```
abstract class STACK(T) is -- the interface of stack implementations
    ...
end;
value class COMPLEX is re,im: FLTD; ... end;
external class TCL_INTERFACE is ... end;
class ORDERED_SET(T<ORDERED) is ... end;
class TRIANGULAR_MATRIX[n](T) is ... end;
```

Class specifiers:

INT TCL_INTERFACE TRIANGULAR_MATRIX[n + 1](FLTD) SET(INT) T($SET(CHAR))

Type specifiers:

$SET(INT) SAME routine(INT, & INT) stream(const INT):INT ♦

A class declaration without class parameters and without inheritance clauses immediately defines a class. Other class declarations are transformed into classes as explained in 4.2. The prefixes external, abstract and value respectively determine the kind of the class. A class without the prefix external or abstract defines a concrete class. A class without the prefix external or value defines a reference class.

Class bounds are used for array classes, cf. 8.2.1. They determine the number and length of the dimensions of the array. Stars indicate that the names of the bounds are inherited from a superclass which then must also be an array class.

4.2 The General Processing Model

The execution of a program consists of three steps:

1. *Establish types and classes*: Collect all the class declarations which are directly or indirectly referred to from the main class K, transform them into classes and create the type dependence graph *TDG*.
2. *Initialize*: Allocate and initialize all shared attributes within those classes, cf. 5.2.
3. *Execute*: Call the routine main in the main class K as K::main.

Hint: Step 1 is usually part of program compilation whereas steps 2 and 3 constitute the program execution proper. ♦

The required class declarations are given by class specifiers.

Establishing types and classes consists of:

1. *Determining the program text*: Starting from a collection of class declarations and a distinguished declaration of the main class K, the transitive closure $\mathfrak{D}$ of K is inductively determined:
   (a) K belongs to $\mathfrak{D}$.
   (b) If a class specifier C or $\$C$ with or without arguments occurs in a class declaration belonging to $\mathfrak{D}$ then the class declaration for C also belongs to $\mathfrak{D}$. There must be exactly one such class declaration with the appropriated number of parameters in the given collection.
   Hint: Some of the class declarations may be predefined and a priori known to the language processor. There are language constructs which are only meaningful with respect to such predefined classes. ♦

   The classes belonging to $\mathfrak{D}$ together form the text of the program.
2. *Resolution of class parameters*: If any of the class declarations in $\mathfrak{D}$ is *generic*, i.e. its heading $C(P_1, \ldots, P_n)$ contains class parameters P_i with or without type bounds, then it is replaced in $\mathfrak{D}$ by a set of class declarations obtained as follows: For each class specifier $C(T_1, \ldots, T_n)$ occurring in any class declaration D in $\mathfrak{D}$ a copy of C is made in which all parameters P_i are consistently replaced by the corresponding T_i. $ symbols contained in any of the class specifiers T_i are retained.

The number of parameters in the class declaration and the number of arguments in the class specifier must agree. The replacements are made in textual order, T_1 is replaced first. If any P_i also occurs (as part of) a type bound then it is treated according to the same rules. The feature resolution restrictions given in 7.2 apply.

If initially a class specifier T_i is or contains a specifier which itself is a class parameter of the class declaration D then D must have been processed before C is considered. A program is illegal if it is not possible to find an order of the class declarations satisfying this requirement.

The identifiers of all the class parameters P_i must be pairwise distinct.

Note: According to these rules class parameters P_i are local to the class declaration D and shadow any class identifiers declared globally. ♦

3. *Resolution of inheritance*: The set $\mathfrak{D}$ of class declaration is transformed into a set $\mathfrak{C}$ of classes as follows:

   (a) The class declarations in $\mathfrak{D}$ are topologically sorted such that a class D which inherits a class C appears after class C. A program is illegal if such a topogical sort is not possible, i.e. if the inheritance relation $C \rightarrow D$ contains cycles.

   (b) Any class declaration in $\mathfrak{D}$ which does not contain inheritances belongs to $\mathfrak{C}$.

   (c) The remaining classes in $\mathfrak{D}$ are considered in an order consistent with the topological sort in step (a). If a class declaration D in $\mathfrak{D}$ inherits the classes $C_1, \ldots, C_k$ then all the C_i must already belong to $\mathfrak{C}$. Then a class D is added to $\mathfrak{C}$ which is derived from the class declaration D by textually replacing the inheritance clauses by possibly modified copies of the bodies of their corresponding classes. The potential modifications are described in 4.4.

   (d) If in step (c) a class C is inherited as a subtype then the type dependence graph *TDG* is augmented by a dependence $C \rightarrow D$.

4. *Checking of type conformance*: For every class argument T_i which in step 2 replaced a class parameter P and for which P was specified by $P < T$ with a type bound T, it is checked that T_i structurally conforms to T as explained in 2.1.5, where T is the type bound of P. A missing type bound means that the minimal constraints for T_i are inferred from the body of the generic class and that P must fulfill them all.

   For every dependence $C \rightarrow D$ in the type dependence graph it is checked that D conforms to C as explained in 2.1.5.

Hint: Steps 1-3 are textual substitutions which do not require any semantic processing except for identifying class declarations for a given class name. ♦

4.3 Class and Type Specifiers

Class specifiers specify object types. The corresponding class is determined according to the rules in 4.2.

A class specifier may contain (index-)bounds. This is required for array-classes including user-defined ones and is only useful for such classes, cf. 8.2.1. Each bound expression must yield an integer value specifying the length of the corresponding dimension of an array class. Stars may be used instead of explicit expressions when specifying array types except when objects are created.

Section 4.2 implies that the keyword SAME may be used to specify the type in which it occurs. SAME occurring in a class A changes its meaning if this class is inherited by another class B: Whereas any occurrence of SAME in A designates the class A, the inherited occurrence then designates class B. If SAME occurs in the signature of a polymorphic type $\$C$, it stands for $\$C$ itself.

For predefined class specifiers see 8.2.

Further type specifiers specify method types.

4.4 Modifications

An inheritance clause may be accompanied by modifications *id method type* $->$ or *id method type* $->$ *id'*. The first of these says that the feature *id* of the given method type is undefined within the subclass C, i.e. it is *not* inherited from the superclass D; the second says that *id* is renamed as *id'* before it is inserted into C. The method type may be omitted if no ambiguity arises.

An include inheritance clause may also have the form include D :: *id method type* or include D :: *id method type* $->$ *id'*. In this case *only* the feature *id* from D is reused (and possibly renamed). The

inheritance clause may be preceded by the keywort private; in this case all the included features are private, cf. 5.1.

4.5 Syntactic Replacements

A parameterless qualification $C :: a$ or $n.a$ in a read context is interpreted as a parameterless call $C :: a$ or $n.a$ respectively. Every non-private attribute a of type A implicitly defines such a *reader* method $a : A$; it is private, if the attribute is private. Qualifications $C :: a$ or $n.a$ in write context are replaced by calls $C :: a(v)$ or $n.a(v)$ where v is the new value to be written to a. Every non-const attribute a of type A implicitly defines such a *writer* method $a(v : A)$; it is private, if the attribute is private or readonly. These implicitly defined methods can only be renamed or overridden by renaming or overriding their attribute.

Note: By defining reader and writer methods $b : B$ and $b(v : B)$, $n.b$ and $C :: b$ may be used as if b was defined as an attribute. ♦

Furthermore the use of any monadic and dyadic operator τ except and and or in an expression is syntactically replaced by a function call according to table 4.2. For showing the priorities of operators the table also includes the built in operators or and and.

is_equal is automatically defined for all types but its definition may be overriden by the user.

It is asserted, that plus and times are left associative; is_equal and plus are commutative. pow is right associative. No assumptions are made on any of the other operators. If a user-defined function is_equal or plus is not commutative, the result is undefined. Commutativity is not asserted for times.

Furthermore bracketed expression lists for index expressions are replaced according to table 4.1.

written form	replacement
$x[e_1, \ldots, e_n] := e$	x.aset$(e_1, \ldots, e_n, e)$
$x[e_1, \ldots, e_n]$	x.aget$(e_1, \ldots, e_n)$

Table 4.1: Replacements for index expressions

x must evaluate to an array object for which the routines aset and aget are defined with the appropriate number of parameters of appropriate types. The first form is also used when retransmitting out parameters to the caller of a method. The second form is used when $x[e_1, \ldots, e_n]$ appears as an operand within an expression.

5 Feature and Local Declarations

```
104  feature_declarations::= feature_declaration ( ';' feature_declaration )* .
105  feature_declaration::= [ [ 'private' ] feature ].
106  feature ::= attribute | method .
107  local_declarations ::= ( local declaration )* .
108  local_declaration ::= attribute_declaration ';' | local_stream_declaration ';' .
109  local_stream_declaration ::= 'stream' identifier method_specification ':=' bound_method .
110  qualification ::= qualifier qualified_identifier .
111  qualifier ::= type_specifier '::' | primary '.' .
112  qualified_identifier ::= identifier .
```

For primaries see 7.

A declaration defines a defining occurrence of an identifier to designate a class or an entity.

Each such definition has a certain *scope*, i.e. that part of the program text in which an applied occurrence of an identifier identifies the given declaration or specification. The scope rules of 5.1 must deliver a unique defining occurrence for each applied occurrence.

The specifications of parameters of methods are replaced by initialized declarations during program execution as explained in 6.7. These declarations become part of the body of the method during execution.

priority	written form	replacement
1	$a >> b$	a.str_in(b)
1	$a << b$	a.str_out(b)
1	$>> b$	TEXT::sin.str_in(b)
1	$<< b$	TEXT::sout.str_out(b)
1	$a \setminus b$	a.backslash(b)
1	$a \mid b$	a.stroke(b)
1	$a \sim b$	a.tilde(b)
1	a ‘ b	a.grave(b)
1	a ? b	a.question(b)
2	a or b	built in
3	a and b	built in
4	$a = b$	a.is_equal (b)
4	$a /= b$	not (a.is_equal(b))
4	$a < b$	a.is_lt(b)
4	$a <= b$	a.is_leq(b)
4	$a > b$	a.is_gt(b)
4	$a >= b$	a.is_geq(b)
5	$a + b$	a.plus(b)
5	$a - b$	a.minus(b)
6	$a * b$	a.times(b)
6	a/b	a.quotient(b)
6	a % b	a.modulo(b)
6	a div b	a.divide(b)
6	a mod b	a.modulo(b)
7	a ^ b	a.pow(b)
8	$-a$	a.minus
8	not a	built in
8	$\setminus a$	a.backslash
8	$\mid a$	a.stroke
8	$\sim a$	a.tilde
8	‘a	a.grave
8	?a	a.question

Table 4.2: Replacements for unary and binary operators

5.1 Scope Rules

The scope of a class identifier consists of all class specifiers in the whole program except in those class declarations in which the identifier is specified as a class parameter.

The scope of a class parameter of a class declaration D consists of the class specifiers in that part of D following the defining occurrence of the class parameter. This rule implies that class parameters can also be used in type bounds for themselves or for subsequent class parameters.

The scope of an expression (cf. 7.3) specification consists of the largest aggregate in which it is contained. In this aggregate it shadows all unqualified occurrences of the given identifier.

The scope of a local declaration or a parameter specification consists of the block containing the declaration except for inner blocks or aggregates in which it is shadowed by another declaration or an expression specification of the same identifier. Parameter specifications are considered part of the block which is the body of the method.

The scope of a feature declaration consists of the body of the class declaration containing the declaration except for those parts of the class body in which the declaration is shadowed by a local declaration or an expression specification.

The scope of a feature declaration comprises all identifiers which are qualified by a class specifier for the given class or an object name for an object of the given class with the following exceptions:

- if the feature is private;
- if the qualification is given by a class specifier then only shared attributes and methods which only access shared attributes are accessible;

- if an attribute is declared readonly then write access is undefined;
- if any of the read and write routines for an attribute *a* are undefined by help of a modification then the read and write access respectively to the attribute *a* is not defined.

Implicitly defined reader and writter methods $a : A$ and $a(v : A)$ have the same scope as their attribute *a*.

A class body may contain several definitions of the same identifier as a feature, but if the result types are ignored, the signatures must be different. If there are several method declarations with the same signature in this sense, after resolution of inheritance any such declaration is discarded except for the last one. These rules also apply to the reader/writer routines generated from attribute declarations.

Note: Hence on inheritance local feature declarations of a class override inherited declarations. In case of multiple inheritance the order of the inherited classes is significant since the later declarations override the earlier ones. The rule also holds for applied occurrences preceding the ultimately resulting defining occurrence. ♦

If several feature declarations for the same identifier remain in a class body after applying the above rule then the feature identifier is called *overloaded*. Except for feature declarations, where ambiguities are resolved by taking the last definition and by overloading resolution, all definitions of an identifier must be unique within their scope. Overloading resolution is described in 7.2

5.2 Attributes

```
113  attribute ::= [ 'shared' ] [ 'readonly' ] attribute_declaration .
114  attribute_declaration::= attribute_specification |
115                           initialized_attribute_specification .
116  attribute_specification::= identifiers ':' type_specifier .
117  initialized_attribute_specification ::=
118              [ 'const' ] ( attribute_specification ':=' general_expression |
119                            identifiers ':' type_constructor ) .
```

For general expressions see 7.

An attribute is defined by an attribute declaration. It introduces the identifier, the type and the initial value of the attribute. Several attribute declarations may be combined into one in which case all attributes have the same type and the same initial value. The initial value is only computed once. If no initial value is specified then attributes of a reference type, of type $OB, or a polymorphic value type are automatically initialized by the value void. Attributes of monomorphic value type T are initialized with #T.

Shared attributes, declared with the keyword shared, simultaneously belong to all objects of the class.

For accessing an attribute the scope rules of section 5.1 apply. In particular any qualified access *name*.*a* will result in a routine call *name*.*a*(*new_value*) or *name*.*a* depending on whether there was a write or read access.

Access to shared attributes follows the same rules. Additionally a shared attribute *a*, declared in class *C* may be accessed by $C :: a$ without mentioning any object. This is particularly useful before any object of class *C* is created. It also allows access to shared attributes of external classes.

The declarations of shared attributes of a class are processed in textual order prior to calling the routine main in the main class. The initializing expressions for shared attributes can only contain constant values or other shared attributes as operands; they may only call routines which do not refer to non-shared attributes in their body.

If the initialization of a shared attribute *a* in class *C* refers to another shared attribute *b* or to a routine *r* of class C' then the declaration of *b* must either textually precede the given declaration in the same class or the declarations of the shared features of the class C' must already be processed before class *C* is processed. The implied order must have a common acyclic refinement and the classes are considered in the order of such a refinement.

All non-shared attribute declarations in a class *C* are processed in textual order whenever an object *o* of class *C* is created.

An attribute declaration comprising a type constructor (cf. 7.1) is an abbreviation for a declaration of an attribute of the type constructor initialized with the value resulting from the type constructor. In case of value types the declarations $a: \#T$ and $a: T$ are equivalent.

Note: In case of constant attributes only the quantity is constant. Thus if its value designates a reference or value object then nevertheless assignments to non-constant attributes of such an object are allowed. ♦

5.3 Processing Local Declarations

A local declaration within a method is processed during a call of this method whenever control reaches the place of the declaration. It is processed in the same way as a feature declaration of the same form except that the created quantity will be local within the block of the method call, not of the object to which the routine belongs.

5.4 The Extent of Objects and Entities

Each object, attribute or local entity of a method has a *lifetime* or *extent*, i.e. that part of the program execution during which it may be used.

The extent of an entity locally declared in a block of a method begins by executing its declaration and ends with the termination of the block. This rule also applies to method bodies including the declarations resulting from parameter transmissions.

The extent of all other quantities extends indefinitely from the time of their creation. They are created as follows:

- A shared attribute is created prior to the call of the routine main.
- A non-shared attribute of a class is created whenever an object belonging to this class is created.
- An object belonging to a class *C* is created whenever a type constructor # *C* or ## *a* is evaluated where *a* denotes an object of type *C*.

Note: Indefinite extent does not imply indefinite accessibility to an object. As soon as all the variables or constants which had a reference to a global object as their value cease to exist or have changed their value or have become inaccessible themselves, it is no longer possible to access such a global object or any of its features. Hence Sather-K should be implemented using a garbage collector since global objects are never automatically or explicitly deleted. ♦

5.5 Methods

```
120  method ::= [ 'stream' ] identifier [ '(' parameters ')' ] [ ':' result_type ]
121           'is' method_body .
122  parameters ::= parameter_specification ( ';' parameter_specification )* .
123  parameter_specification:= [ parameter_kind ] attribute_specification .
124  method_body ::= block 'end' | 'deferred' 'end'.
125  block ::= local_declarations statements .
```

For statements see section 6, for expressions see 7.

Example: stream elts : T is . . . end
divmod(const arg:INT; & div,mod:INT) is . . . end
combine(a,b:routine(INT):INT):routine(INT):INT is . . . end ♦

Methods are routines or stream methods. They are given by a block, the method body. They may be called by using their name in a statement or expression. The parameters of methods correspond to the arguments of method calls, cf. 2.1.4 and 6.7. Bound methods are established by a (partial) call and can then be used like a method.

Stream methods are (a limited form of) coroutines and mostly have the same form as routines. Additionally they may contain the keyword resume as a statement. Stream methods are used for establishing a bound stream or a stream object.

A method which is called in an expression must deliver a result. The method declaration must then specify a result type. In this case the method will automatically contain a local variable res of the result type. The result of a routine call is the value of res. Routines with results are also called *functions*. Other routines are called (proper) *procedures*. Similarly the value of res is transmitted to the caller whenever a resume-statement is executed in a stream call.

A method body deferred may only appear in an abstract or an external class.

6 Statements

```
126  statements ::= statement ( ';' statement )* .
127  statement ::= [ assignment | begin_block | conditional_statement |
128                  case_statement | type_case_statement | loop | 'break' |
129                  expression | 'resume' | 'return' |
130                  raise_statement | assertion_statement] .
```

6.1 Assignments

```
131  assignment ::= designator ':=' general_expression .
```

Example: var := op1 * op2; arr[4] := 5; obj.attr := 7 ♦

Certain assignments are transformed into routine calls before processing, cf. 4.5.

Any other assignment is processed by simultaneously evaluating the designator and the general expression. Then the entity identified by the designator is made to denote a copy of the value of the general expression.

The designator must evaluate to a non-constant attribute of the class in which the assignment occurs, a local variable, a shared attribute qualified by a class or an attribute qualified by an object; such attributes of other classes or objects must not be private, constant or readonly. Assignments to SELF are not allowed but SELF may occur as the qualifier of an designator. (For *self* see 6.7.)

In general, the type of the general expression must conform to the type of the designator. In case of value types it is also allowed that the type of the expression is coercible to the type of the designator, cf. 7.2.

Hint: "Simultaneous" evaluation of several expressions means: The evaluation of the expressions may be arbitrarily merged in time or may even be done in parallel; hence side-effects from evaluating one of the expression may lead to unpredictable results. ♦

6.2 Blocks and Exception Handling

```
132  begin_block ::= 'begin' block [ exceptions ] 'end' .
133  exceptions ::= 'except' exception_identifier
134                 ( 'when' exception_types 'then' block )*
135                   [ 'else' block ] .
136  exception_identifier ::= identifier .
137  exception_types ::= exception_type | exception_types ',' exception_type .
138  raise_statement ::= 'raise' [ exception_type ] .
139  exception_type ::= type_specifier .
140  assertion_statement ::= 'assert' expression .
```

Example: begin i:INT; i:=f(3); . . . end
begin a := b / c except e when ZERO_DIVIDE then a := 1000 end; ♦

A block is a sequence of statements possibly preceded by local declarations. Any locally declared identifier shadows non-local entities, cf. 5.1. The life-time of local variables and constants ends when the execution of the block terminates. Method bodies, parts of conditional case and typecase statements, the typecase statement as a whole and loop bodies all constitute blocks. begin-blocks may be used for further subdividing the scopes and life-times of entities and for exception handling.

A begin-block containing exceptions is called an *exception statement*. It is processed by first executing the initial declarations and statements. If during this processing no explicit or implicit raise-statement is executed the processing of the exception statement is finished.

If a raise-statement occurred it must have specified a class type C as its argument which is a conforming subtype of the predefined class EXCEPTION. A new object o of class C is created and control is transferred to the first exception clause which contains a supertype T of C in its exception_types. If such an exception clause is found then the corresponding block is prefixed by a local declaration $exception\_identifier : T := o$ and then executed. Then the processing of the exception statement is finished.

If in an exception statement no appropriate exception clause could be found or if the executed exception clause itself executed a raise-statement then the exception statement is terminated abnormally and the search for an exception clause is continued in the dynamically enclosing block.

If also in the enclosing block no conforming exception clause can be found or if a raise-statement occurs outside an exception statement then the processing of the routine is terminated abnormally and the search is continued at the place of the routine call.

The execution of the program terminates abnormally if the execution of the routine main terminates in this way.

A raise-statement must always specify an exception type except when it is used in an exception clause itself; in this case an omitted exception type indicates that the same exception object should be used again, i.e. the same exception as before is raised.

Exceptions can also be raised implicitly by the underlying implementation of Sather-K. Such implicit exceptions include FLOAT_OVERFLOW, FLOAT_UNDERFLOW, INTEGER_OVERFLOW, ZERO_DIVIDE, INDEX_ERROR, REFERENCE_ERROR, INVARIANT_ERROR, ASSERTION_ERROR, STREAM_TERMINATION, EXIT. The exception type NUMERIC_ERROR is a supertype of all numeric exceptions.

Note: Thus, an exception-type occurring in a raise statement must always be given by a class specifier whereas when handling the exceptions also polymorphic class types are allowed. ♦

The assertion statement evaluates its boolean expression. If the value is false, an ASSERTION_ERROR is raised. However, the user cannot rely on these checks actually being performed.

6.3 Conditional Statements

```
141  conditional_statement::= 'if' expression 'then' block [ alternative ] 'end' .
142  alternative ::= 'else' block | 'elsif' expression 'then' block [ alternative ] .
```

Example: if a > b then res := 1 elsif a = b then res := 0 else res := -1 end ♦

A conditional statement is processed by first evaluating its expression, which must hold a result of type BOOL. Depending on this result the block (true) or the alternative (false), if any, are processed.

If the alternative starts with elsif then it is processed as if it were a conditional statement on its own.

6.4 Case Statements

```
143  case_statement ::= 'case' expression
144                     ( 'when' values 'then' block )*
145                       [ 'else' block ] 'end' .
146  values ::= value ( ',' value )* .
```

Example: case neighbors when 1,5,6,7,8 then reset when 3 then toggle else set end ♦

A case statement is processed by evaluating the case expression. Then this value is compared to the values of the case labels. The case labels' values must be of boolean, character, or integer type. If the comparison with one of the values of a case label yields true then the subsequent block after this case label is processed. If none of the comparisons yields true and an else-clause was present, the block following else is processed.

The type of any case label must be a conforming subtype of the type of the case expression. A value must occur at most once as a case label.

6.5 Type Case Statements

```
147  type_case_statement::= 'typecase' ( local_declaration | identifier )
148                         ( 'when' types 'then' block)* [ 'else' block
149                         ] 'end' .
150  types ::= type_specifier ( ',' type_specifier )* .
```

Example: typecase o:=mail.next when BILL then o.pay when NOTE, $AD then o.order else drop end ♦

The local declaration of a type case statement must be an initialized declaration for an identifier *id*. The typecase statement is a block in which this local declaration is valid. Instead of this local declaration the identifier *id* of a local variable or a parameter of a routine may be used. All the types in the when-clauses must conform to the type of *id*.

The type case statement is processed as follows: First the initial local declaration is processed. Then the type T of the value v of the initializing expression is compared with the types T_i in the when-clauses in textual order. If a clause is found such that T conforms to T_i then the block of the first such clause is prefixed with a local declaration $id : T_i := v$ and then processed. If no such clause can be found and an else-clause is present then the block of the else-clause is prefixed with the initial declaration and processed.

The initializing expression is always evaluated only once.

6.6 Loops and the Break Statement

151 loop ::= ['while' expression] 'loop' block 'end' .

Example: loop i := 1.upto!(10) . . . end ♦

A loop statement while *e* loop *S* end is processed by repeating the evaluation of *e* and the execution of the block *S*, the body of the loop, as long as the expression *e* evaluates to true or until a statement terminating the loop is executed. *e* must be of type BOOL. loop *S* end has the same meaning as while true loop *S* end.

The statement break terminates the innermost loop containing it. It must be syntactically contained within a loop.

A loop body may contain one or more stream calls like until! or upto! which may influence the termination of the loop, cf. 6.7.2.

6.7 Method Calls

152 method_call ::= designator ['!'] ['(' arguments ')'] .
153 arguments ::= argument (',' argument)* .
154 argument ::= general_expression | '&' designator | '&&' designator .

Example: max(a,b); i1.divmod(i2,& div, & mod); obj.attr.elts!; o.f(„); ♦

Direct stream calls and calls of a stream are distinguished by the symbol ! after its identifier.

Methods whose bodies are deferred can only be called if they belong to an external class. It is then assumed that the body is supplied in another programming language, cf. 9.1.2.

Method calls can be object calls $a.m$ or class calls $C :: m$. If the type of a is a polymorphic type $\$C$ (*polymorphic method call*) then the declaration of m in class $\$C$ is used for checking the validity of the call and for inferring the type of arguments and the result.

A method call carries as its 0-th argument the reference to its object. In the method body this argument has the name self of type C. When the method is called by $C::m(\cdots)$ without reference to an object then the value of self is void. self is a constant local entity of the method.

A method call is valid only

- if the number of arguments and the number of parameters in the declaration of the method are the same, and
- if the types of all the in- and const-arguments conform to the types given in the declaration of the method, and
- if for all out-parameters the types given in the declaration of the method conform to the actual out-arguments, and
- if for all inout-parameters the types given in the declaration of the method and the types of the actual inout-arguments are the same, and
- if all out- and inout-arguments are entities to which assignments are allowed in the context of the method call.

6.7.1 Routine Calls

A valid routine call is processed as follows:

1. If the call has in- or inout-arguments then (a copy of) the block which is the routine body is prefixed by initialized local declarations *parameter specification* := *argument*.
2. If the call has out-parameters then the body is further prefixed by local declarations *parameter specification*.
3. If the routine specifies a result type then the body of the routine is prefixed with a local declaration res: *result type* declaring res as a local variable of the result type given in the declaration of the routine.

   The order in which these local declarations are added to the routine body is undefined although the arguments and the parameters correspond in the order in which they are written.

   Hint: The fact that the order of evaluating the arguments is not specified is especially important for inlining code. ♦

4. The local declarations specified in steps 1 to 3 are processed.
5. The body of the routine is processed. If the routine specifies a result type the processing of the body must contain at least one assignment to the variable res.
6. The processing of the body ends when the end of the body or the statement return is reached. Then the current value of res is the result of the routine call. If the routine contained out- or inout-parameters then the entities designated by the out-arguments are determined and the current values of the corresponding local variables created in step 2 are assigned to these entities.
7. If the class in which the routine is declared contains a feature named invariant then this must be a parameterless routine yielding a result of type BOOL. This routine is called. The original routine call is undefined if the result is false.
8. The local entities of the routine are discarded and the result is returned to the caller.

If a routine call occurs as a statement then its result, if any, is discarded. The semantics of a program must not depend on the evaluation of invariants.

6.7.2 Stream Objects, Stream Calls and the Resume Statement

A stream call is used for successively accessing the elements of data structures. This is achieved either by a direct stream call or by establishing an explicit stream object which then may be called.

A direct call of a stream method or the call of a stream object has the form $a!(\cdots)$ where a is the name of the stream or the stream object.

A direct stream call is executed in the same way as a routine call. But if during the execution of the body of the stream method a statement resume is met then the execution of the call is suspended, and the result, if any, and all out- and inout-parameters are returned to the caller (steps 7, 8 of 6.7.1).

Direct stream calls must be syntactically contained within a loop. On suspension of the stream call the execution of the loop is continued. Whenever the loop body is repeated then the same direct call is executed by continuing the stream method after the resume statement. At this time the in- and inout-arguments of the call are reevaluated and reassigned to the corresponding parameters.

The execution of a direct call terminates when a return statement or the end of the body of the stream method is reached. Termination of a direct stream call also terminates the execution of its innermost enclosing loop. At this time the result and the values of out- and inout-parameters, if any, are *not* transmitted to the caller; the class invariant is checked.

Note: There might be several stream calls within a loop; the first terminating one terminates the loop. All direct calls at syntactically different positions are different, even when they use the same stream method. ♦

Direct calls implicitly establish a stream object whose state between calls indicates how far the stream has progressed. An explicit stream object may be established by assigning a bound stream to a local quantity of the appropriate stream type. Establishing a stream object requires that all arguments corresponding to const parameters are given; arguments for out- and inout-parameters cannot be given. Then a bound method with the remaining free parameters together with the (uninitialized) stream state is created.

The stream object may be used instead of a stream method within a stream call with the same meaning as a direct call; especially, when such a call terminates within a loop then the loop is terminated. Additionally, such calls may also occur outside of loops and they always designate the same stream. If a call terminates outside of a loop or if an already terminated stream object is called then the exception STREAM_TERMINATION is raised.

Every explicit stream object has the automatically declared Boolean attribute terminated which has the value true after termination and false before.

For predefined stream methods see 8.1.

7 Expressions

```
155  general_expression::= expression | type_constructor | bound_method .
156  expression ::= ( unary_operator * primary
157               ( binary_operator ( unary_operator )* primary )* .
158  unary_operator ::=  see table 4.2 .
```

```
159  binary_operator ::= see table 4.2 .
160  primary ::= value | designator |
161            coercion | method_call | '(' expression ')' .
162  value ::= denotation .
163  designator ::= identifier | qualification | primary '[' expressions ']' ,
164  expressions ::= expression ( ',' expression )* .
165  coercion ::= type_specifier '-<' primary .
166  type_constructor ::= '#' class_specifier [ aggregate ] |
167                       aggregate | '#' '#' designator .
168  bound_method ::= 'bind' partial_call .
169  partial_call ::= designator ['(' partial_arguments ')' ] .
170  partial_arguments ::= [ general_expression ] ( ',' [ general_expression ])* .
```

All unary and binary operators carry the priorities defined in table 4.2. Except for or and and they are replaced by method calls. For qualified identifiers see 5, for aggregates see 7.3.

Example: a^2 * b^2, -a, true, FLT<-5, a[4], #LIST, ##a ♦

7.1 Evaluation of Expressions

Expressions are evaluated by simultaneously evaluating their constituent primaries and then applying the operators with the priorities specified by table 4.2. However, for the operators or and and only the first operand is initially evaluated. The second operand is evaluated only if the first operand does not determine the final result of the operation.

The semantics of all operator applications except for or and and is defined by syntactically replacing the operator application by routine calls as explained in 4.5. Symbols belonging to other_special's can only be used if the programmer has given an appropriate routine declaration. The operators or and and are only defined for the type BOOL and subtypes of BOOL. They designate the short circuit logical operations.

The (in-)equality operators are defined for operands of arbitrary types. '=' yields true only if the values of the operands are equal; when comparing two entities of a reference type only the pointers are compared!

If a designator occurs as a primary then it must either designate a parameterless routine or a variable. In the first case the method is called; the result of the call is the value of the primary. In the latter case the value of the primary is the current value of the variable. If the identifier occurs in the scope of a non-shared attribute, self must not be uninitialized.

If a primary evaluates into a parameterless bound routine of type *t*, then the routine is called and its result is treated likewise except when the primary is used as a general expression in a context where a value of type *t* is admissible.

If an identifier is qualified with a class specifier, $C :: x$, then x must designate a shared attribute or a method of class C. Within method calls $C :: m$ or $C :: m(\cdots)$ the variable self has the value void if C is a reference class. Otherwise self is uninitialized.

If an identifier is qualified with an object designator, $a.x$, then first the object a is determined. The qualification $a.x$ designates the attribute or method x of the designated object.

A primary followed by a bracketed expression list designates an array access as described in 4.5 and 8.2.1.

7.1.1 Type Constructors

A type constructor is evaluated by creating an object of the given type. If the type constructor has the form *##a* then an object of the same type as the current value of *a* is created. This current value must not be void. If the type constructor contains or is a aggregate then this aggregate is assigned to the new object as described in 7.3. If the new object defines a parameterless init method, cf. 5, this is called before attributes defined by initialized declarations are assigned their initial values. Aggregates and init methods are mutual exclusive.

7.1.2 Bound Methods

Bound methods are values of a bound method type. The type of the bound method is a method with the same result type as the original method from which the partial call is constructed; its parameter are those for which the partial call did not give arguments. The remaining parameters are replaced by the

arguments of the partial call computed at the time when the bound method is created. The new method belongs to the same object as the original method.

The 0-th parameter self is also bound by the partial call unless the method call is qualified in the form T :: with a polymorphic type T. In the latter case self remains unbound; when calling the bound method an object of a subtype of the given polymorphic type has to be passed as the first parameter of the call.

When a bound stream is used for establishing an explicit stream object then all const-parameters must be bound, cf. 6.7.2.

7.2 Coercions and Overloading Resolution

For uniquely indicating the type of an aggregate of a value type or the value of a primary it may be prefixed with the type in a coercion. Coercing a value v of type T to type T' is legal if any of the following holds:

- T is a subtype of T'.
- T and T' are simple value types that both (transitively) inherit the same predefined simple value type; T' must be monomorphic.
- T is a numeric type (including UNIVERSAL_INT and UNIVERSAL_REAL), and T' is a monomorphic numeric type.
- T is UNIVERSAL_BITS and T' is a numeric type, BYTE, CHAR, ROW[n](S), ARR[$*$](S), ARRAY[$*$](S) or a subtype of S, where S is again one of those types. The value is zero-extended on the left to the actual size of T'. Otherwise the value is only reinterpreted. When coercing to an ARR or ARRAY type, the size of the array is determined from the given value.
- T is a subtype of STR and T' is STRING.
- v can be coerced to some type S and the resulting value can be coerced to T'.

In general, a coercion is only reinterpreting the given value in the sense of its type. A value of a numeric type, including the universal numeric types is coerced to another numeric type by converting integer values to floating point values and adapting the length if necessary. Rounding from floating point numbers to lower precision or to integers must always be specified by explicit operations.

The resulting expression has exactly the type of the coercion. Hence, in overloading resolution no further conformance to supertypes will be checked.

A coercion is automatically inferred in the following cases:

- The value of an assignment's right hand side is coerced to the type of the designator.
- The same rule applies during transmission of in-, const-, inout- and out-parameters.
- A value of one of the universal types is automatically coerced as described below.

Applied occurrences of overloaded identifiers are resolved as follows:

1. If the applied occurrence is accompanied by arguments then the number, types and kinds of the arguments are used for determining the set of applicable feature definitions. If the feature belongs to a class C specified by a generic parameter T and T was declared with a typebound $T < D$ then only the features in D are considered.
2. If multiple features are applicable, then numeric arguments including universal types are coerced to the first type in the schemes SHORT_INT → INT → LONG_INT → INTINF, SHORT_UNSIGNED → UNSIGNED → LONG_UNSIGNED → INTINF, and FLTD → FLT that can represent the value and for which an applicable feature exists. The program is illegal if the resulting set of applicable features is empty or contains more than one feature.
3. An overloaded feature may not have private as well as non–private variants.

7.3 Aggregates

```
171  aggregate ::= [ expression_specification ] '{' named_expressions '}' .
172  expression_specification::= identifier '@' .
173  named_expressions ::= [ designator ':=' ] general_expression
174                        ( ',' [ designator ':=' ] general_expression )* .
```

Example: { { void, 3, bottom @{void, 4, void}, 1, {bottom,2,void} } ♦

An aggregate defines the values of the attributes of the type constructor to which it belongs. An aggregate yields as its result a value of the type constructor preceding it. In creating this value other objects to which elements of the aggregate refer may be created. Aggregates without a preceding type constructor may only occur as substructures of a larger aggregate; their type is given by the type of the attribute to which it is assigned; this type must not be polymorphic.

The expressions of an aggregate are evaluated simultaneously. The resulting values are assigned to the non-shared non-private variable attributes of a new object of the type of the aggregate in the order of the attribute declarations, thus overriding all existing initializiations of those attributes. This includes the attribute declarations which are inherited from other classes. In case assignments are given instead of expressions the designators of those assignments must be attribute names of the object; only the specified attributes are assigned to. Such named expressions must follow any unnamed expressions in the aggregate.

Within an aggregate the aggregate itself or any substructure can be prefixed by an identifier which, during evaluation of the aggregate will designate the reference of the object being created. This identifier may be used within the aggregate and its potential substructures instead of an expression and then evaluates to the object reference it designates.

Thus, for reference types not only tree-like object graphs but also DAGs and circular graphs can be constructed.

8 Predefined Features and Classes

8.1 Predefined Features

Each class automatically contains several predefined features or features with special properties. These comprise:

1. string: STRING

   string is a parameterless routine delivering an object of type STRING representing the current value of an object. In case of simple value types the string is a denotation. Otherwise it has the form of an aggregate $\{v_1, \ldots, v_n\}$ listing all the attributes which have simple values whereas the values of attributes of a reference type are replaced by a hash sign #. string may be redefined to provide more appropriate representations.

2. is_equal($OB): BOOL

   This function returns true, if the arguments is the same as self. That means identity for reference objects and same value for value objects. If the argument and self are of different type, is_equal returns false.

3. type: TYPE

   Every class type defines a constant type: TYPE with a value which is different for each class type. type must not be redefined. For an object *a* a string representation of its type may be obtained by *a*.type.string.

4. copy: SAME

   This routine delivers a new object of the same type containing copies of the values of all attributes of the current object. The type must be a reference type.

5. init

   If defined, the init method is called during object creation, before attributes defined by initialized declarations receive their initial values. Init must not be called explicitely. If an invariant method is supplied, this is not checked right after the call to init, but when all initializations are processed.

6. finalize

   This routine is called before an object is destructed. This will happen on return from methods for local value objects or when the garbage collector removes an object.

7. invariant: BOOL

   If the programmer supplies such a routine in a class then it is supposed to define the class invariant. If the call returns the result false then the exception INVARIANT_ERROR is raised.

The class OB defines a stream method

```
stream until(b: BOOL) is
  loop if b then return; end; resume; end;
end;
```

which may be used to define loops that are executed at least once.
Example: The typical usage of until is

```
loop ... until!(all_done); end;
```

The parameter *b* of until is not declared const. So *b* is reevaluated for every stream call and not only for the first call. ♦

The classes INT, SHORT_INT, LONG_INT, UNSIGNED, SHORT_UNSIGNED, LONG_UNSIGNED and INTINF define streams

```
stream upto(const to: SAME): SAME is
  res := self;
  loop if res > to then return; end; resume; res := res + 1; end;
end;

stream downto(const to: SAME): SAME is
  res := self;
  loop if res < to then return; end; resume; res := res - 1; end;
end;
```

that may be used for for-loops.
Example: Its typical usage is

```
loop 1.upto!(i); ... end;
```

The parameter *to* of upto is declared const. So *to* is only evauated once for the first call. Later changes of *i* will not affect the number of iterations. ♦

8.2 Predefined Classes

All implementations of Sather-K contain a number of predefined classes. In this section we give a rough description of these predefined classes. Further details are contained in the documentation of the Sather-K library.

The following abstract classes are used for structuring the type system and the type dependence graph:

OB: By definition every class conforms to OB.

REFERENCE: Every reference class is automatically a subtype of this class. It defines assignment and the operation is_equal for all reference classes.

VALUE: Every value class is automatically a subtype of this class. It defines assignment and the operation is_equal for all value classes.

EXCEPTION: Every exception class, including user defined exceptions, must be a a subtype of EXCEPTION. The class EXCEPTION is a reference class and contains the readonly attributes file_name:STRING and line_number:INT which describe the source position of the origin of the exception.

EXTOB: EXTOB is a simple value type which could be used to reference objects defined in other programming languages. It is guaranteed to be able to hold values of any pointer type of the target machine.

The following classes provide services needed from the operating system:

FILE(T): It defines sequential files of elements of type T.

TEXT: It defines the properties of text input and output. It defines the input/output routines str_in and str_out. str_in(*x*) is defined for arguments of all simple value types and of STR and STRING. str_out(*x*) is defined for values of types STR and STRING and for all other types T by str_out(*x*: T): TEXT is res := str_out(x.str) end.

There are three objects sin, sout and serr of type TEXT which designate standard input, standard output and standard error output. The simple value types and STR and STRING define str_in and str_out as parameterless routines for input and output on the standard input and output.

8.2.1 Array Classes

A class declaration $C[m, n, \ldots]$ with bounds declares an array class. The number of bounds is called the dimension of the array class. The dimension is part of the type defined by the array class. The bounds are transformed into readonly attribute declarations readonly *m*: INT; readonly *n*: INT;. . . . These attributes are initialized whenever a new object of the type is created by a type constructor $\#C[exp_0, exp_1, \ldots]$. In case of the predefined class ROW [asize](T) the attribute asize is constant and its value is part of the type.

An array class declaration *D* may inherit an array class *C* taking over *C*'s bounds including their names. This is indicated by supplying the symbol * in the inheritance as parameter for the index bounds to be taken over. The number of stars must match the dimension of *D*. The class must not inherit multiply in this way.

Every array class should define the routines aget and aset with the appropriate number and types of parameters.

The replacements of $a[i, j]$ by a.aget(i, j) or a.aset(i, j, v) described in 4.5 are made in any case, also if *a* does not belong to an array class. The correct number of indices in $a[i, j]$ is determined by the (method) types of aget and aset respectively.

A class specifier must give values to all its bounds when it is used as a type constructor or when it is of the form ROW[asize](T). In all other cases some bounds may be replaced by stars. In case of an assignment $a := b$ to a variable declared by $a : C[10, *]$ it is checked that the first bound of *b* has the value 10 whereas the second bound is not checked.

Hint: Since for value types `v:V` is equivalent to `v:#V` which is short for `v:V:=#V`, and since an explicit size must be given for ARR creation, a specification `a:ARR[*];` is legal for method parameters only. ♦

The built-in classes for handling arrays are:

- ROW[asize]($T <$ OB): Generic value class. Objects of type ROW[n](T) are fixed size one dimensional arrays of elements of type T indexed by $0, 1, \ldots$. The class supplies the routines aset(i : INT;v : T) and aget(i : INT) : T as explained in 4.5 for accessing the elements, and the constant asize giving its length. asize must be known to the compiler and is part of the row type, i.e., it is illegal to specify a row-type by ROW[$*$](T). No class may inherit from ROW.

- ARR[asize$_0, \ldots,$ asize$_{k-1}$]($T <$ OB): This generic value class is defined for every dimension $k > 0$. It supplies dynamic k-dimensional arrays of size asize$_0 \times \cdots \times$ asize$_{k-1}$. The bounds carry the names asize0, asize1, ... up to the dimension of the array. It defines the routines aget($i_0, \ldots, i_{k-1}$: INT) : T and aset($i_0, \ldots, i_{k-1}$: INT; v : T). The length is not part of the type. Whenever a type constructor # ARR[$exp_0, \ldots,$ exp$_{k-1}$](T) is evaluated then the exp_i will be evaluated and determine the length of the array which is then fixed. A type constructor # ARR[$*, \ldots, *$](T) for a k-dimensional array must be followed by an aggregate consisting of unnamed expressions. The number of these expressions determines asize$_0$. If $k > 1$ then the expressions must each be aggregates for $(k-1)$-dimensional arrays. Otherwise objects of an ARR-type have the same properties as rows.

- ARRAY [asize$_0, \ldots,$ asize$_{k-1}$]($T <$ OB): This generic reference class is defined for every dimension $k > 0$. It supplies k-dimensional arrays of size asize$_0 \times \cdots \times$ asize$_{k-1}$. The bounds carry the names asize0, asize1, ... up to the dimension of the array. It defines the routines aget($i_0, \ldots, i_{k-1}$: INT) : T and aset($i_0, \ldots, i_{k-1}$: INT; v : T). An aggregate for an array consists of a list of the values of the elements of the array or row. In case of multidimensional arrays the rows are given by substructures. ARR[$n_0, \ldots, n_{k-1}$](T) and ARRAY[$n_0, \ldots, n_{k-1}$](T) are not user-definable because it exists for any number k of dimensions.

  No class may transitiveley inherit from ARR more than once.

- STR[n]: This class is a subtype of ARR[n](CHAR). String denotations "abracadabra" are denotations for # STR{'a','b','r','a','c','a','d','a','b','r','a'}. When declaring a quantity of type STR with an initializing string denotation then the size of the string denotation may determine the size of the quantity. For convenience STR[*] may be abbreviated to STR.

- STRING[n]: This class is a subtype of ARRAY[n](CHAR). Otherwise it has the same properties as STR. For convenience STRING[*] may be abbreviated to STRING.

- BITS[n]: This class is defined as ROW[n](BOOL).

8.2.2 Simple Value Classes

- BOOL: This type represents Boolean values. Note, however, that for alignment reasons an implementation may allocate more than one bit for storing a Boolean value.

- BYTE: a subtype of BITS[8].
- CHAR: This may be a subtype of BITS[8] but other choices may be made, e.g. for accommodating characters according to the UNICODE standard.
- INT, SHORT_INT, LONG_INT, UNSIGNED, SHORT_UNSIGNED, LONG_UNSIGNED: These numeric types are subtypes of BITS[*n*] for appropriate values of *n*.
- FLT, FLTD: These numeric types are subtypes of BITS[*n*] and represent floating point values according to the IEEE standard 754 - 1985 in single and double precision.
- INTINF: This reference type implements integers of arbitrary size (and thus is an implementation of UNIVERSAL_INT).
- TYPE: This type is used for the values returned by the routine type defined in 8.1.

9 The Environment

9.1 Connecting a Program and the Environment

9.1.1 Program Execution

Compilers for Sather-K are usually invoked by

```
compilername options classname
```

where classname is the name of the main class. The compiler will find the necessary classes as specified in 4.2. The compiler will eventually produce an executable with the same name.

The routine main in the main class must not be overloaded. It may have a parameter of type ARRAY[*](STRING). The corresponding argument will be a sequence of strings on the command line from which the program was called. Note that the 0th component will be the programname.

The routine main in the main class may also specify a result of type INT. This result should be 0 when the program execution ended correctly. Results $\neq$ 0 indicate failure.

The compiler is not required to perform full semantic analysis on the complete program text but only on features reachable in the program.

9.1.2 The Foreign Language Interface

A class declaration which is specified as external constitutes an interface to other programming languages. An external class may define routines and shared attributes only. The attributes are replaced as described in section 4.5. Private declarations do not contribute to the interface. Routines may have a body or may be declared as deferred.

Routines that are not deferred are accessible from other programming languages. The implementations of deferred routines have to be available at link time and may be given in some foreign programming language. The foreign programming language must support the standard calling conventions of the target machine. *The necessary type checking is not done!*

The external identifier for an exported or imported routine is the concatenation of the identifier of the external class and the routine identifier. All external identifiers have to be unique at link time. Within an external class identifiers have to be unique; no overloading resolution applies.

There are no objects of an external class.

Literatur

Meyer, B. (1988): *Object-Oriented Software Construction.* Prentice Hall.

Omohundro, S. M. (1991): The Sather Language. Report, 3. Juni 1991, International Computer Science Institute, Berkeley.

Omohundro, S. M. and Stoutamire, D. (1994): The Sather 1.0 Specification. Working Paper, International Computer Science Institute, Berkeley, CA, USA.

Waite, W. M. and Goos, G. (1984): *Compiler Construction*. Springer Verlag. polnische Übersetzung Wydawnictwa Naukowo - Techniczna, Warschau, 1989.

Weber, F. (1992): Getting Class Correctness and System Correctness Equivalent. In *Proc. 8th International Conference TOOLS '92*, S. 199–213. Prentice Hall, New York.