SAPPEUR Language Manual

Frank Gerlach

April 16, 2021

Language Version: 10 Manual Revision: 3

Chapter 1

Rationale For SAPPEUR

Contemporary software which must execute efficiently is almost always written in C/C++ or FORTRAN. Examples are large scale statistical analysis, large scale simulations, database servers, operating systems, office software packages, signal processing software, 3D video games or image/video processing software. Java(R) or the .Net(R) languages cannot deliver this efficiency, because they have not been designed with the same efficiency imperatives as C++. Performance issues are a) the inability to allocate objects or arrays on the stack, b) garbage collection and memory management, c) inability to create value arrays. Performance issues have relegated Java(R) into application domains which are not processing-intensive, mostly database-based business/transaction processing applications.

On the other hand, C++ does not provide a safe execution environment, as I) array bounds are not checked, II) unsafe casts are possible, III) the language has no notion of concurrency. Many C++ programmers had horrible experiences tracking down bugs in multithreaded C++ programs. Some bugs are impossible to find, as they destroy the integrity of the run-time environment (such as heap management data structures) and cause more or less random and non-deterministic errors. Tools like Purify or Valgrind do not provide relief either, as they slow down execution by a factor of 10 to 100, thereby making the nasty, infrequent and timing-dependent errors improbable.

SAPPEUR is intended to address the weaknesses of those "managed" languages (.Net(R) and Java(R)) and the weaknesses of C++. The goal of SAPPEUR is to provide both a safe execution environment and efficiency of execution. If Multi-core CPUs are to be used efficiently, the programming language must support concurrency.

Chapter 2

Differences To Java(R) and C++

2.1 Bounds Checking

All Arrays are bounds-checked at run-time in SAPPEUR. This is not the case in C++.

2.2 Stack Depth Checking

C++ does not check for stack overflows at runtime. This can be a source of memory corruption. The Sappeur assurances require stack depth checking at runtime (in the general case of recursive calls).

Note that users of Sappeur executables must use operating-system-specific mechanism to commit more stack memory than Sappeur allows to safely execute the program. On Unix systems, this is done with the `ulimit -s <STACKSIZE>` mechanism.

2.3 Only Safe Casts

SAPPEUR does only allow casts that will never endanger the integrity of the execution environment. In C++, the programmer can cast anything into everything, thereby creating a host of potential safety issues.

2.4 Multithreading in the Type System

SAPPEUR is "multithreading-aware". The type system is aware of objects and methods which might be called simultaneously in different threads. Such objects/methods are protected by locks. Objects/methods which are thread-local (or local to a multithreaded object) don't have this locking overhead. The type system will also ensure that these local objects are not passed to a different thread or a different multithreaded object.

2.5 No Operator Overloading

Operator Overloading is considered "syntactic sugar" and therefore not provided in SAPPEUR.

2.6 No Multiple Inheritance

Multiple Inheritance is very difficult to properly understand for most programmers, so it is not provided in SAPPEUR. You can use interfaces, like it is the case in Java(R) and .Net(R).

2.7 Value Arrays

SAPPEUR does allow the programmer to specify arrays of pointers and arrays of values, whereas Java(R) does only allow for arrays of pointers/references. Obviously, Java(R) wastes processor cycles and memory cells (for the pointers).

2.8 Objects On the Stack

SAPPEUR allows the programmer to allocate almost all variables on the stack (sometimes called "automatic" variables). Stack allocation is the most efficient way of creating an object, as the needed memory locations are in most cases already in the CPU cache. When the object is no longer needed, it will be recycled immediately upon leaving the current code block/method. When the next code block/method is executed, the same memory location can be reused for another object. Java(R) does only provide for the stack allocation of pointers/references and primitive data types (int, char, float etc).

2.9 Arrays on the Stack

Arrays of almost all types can be allocated on the stack with SAPPEUR. Java(R) does not allow for the allocation of any arrays on the stack. The same efficiency argument as with "Objects On the Stack" applies.

2.10 Reference Counting Instead of Garbage Collection

The SAPPEUR language contains a built-in mechanism for memory management, namely reference counting. This is virtually identical to the "Smart Pointer" concept used by many C++ programmers. The difference is that the SAPPEUR user (programmer writing code in SAPPEUR) will never "see" the reference counting mechanism, as it is done (and enforced) by the code generated by the SAPPEUR compiler. This means that all objects which are no longer needed will have their memory reclaimed immediately. With Java(R), memory reclaiming is more or less non-deterministic and may take any time between a few microseconds to days (or even infinity). Note that reference counting in SAPPEUR means that the programmer must break any pointer cycles, which is not necessary in Java(R).

2.11 Generics

SAPPEUR provides a simple Generics concept which may be used to create generic container classes and other constructs. This is similar to the template and the macro concept of C++. At this time, the C++ template functionality is (at least in theory) more powerful, though.

Developers are encouraged to generate generic code using proper Macro Languages like m4 (http://www.gnu.org/software/m4/). Most generic code such as containers can be nicley implemented this way. The whole build/debug process will be more efficient than a template-based process.

Example

2.12 Integrity of Pointers

All pointers in SAPPEUR are guaranteed to be either NULL or pointing to a valid object. This is the same assurance as in Java(R) or .Net(R). There are no such things as I) pointer arithmetics II) dangling pointers III) uninitialized Pointers IV) unsafe pointer casts. The C++ language does provide all of those four features.

2.13 Destructors

SAPPEUR provides the Destructor mechanism in virtually the same manner as C++ does. The only difference is that Destructors will be also called if a heap-allocated object's reference count goes to zero. Besides efficient use of memory, Destructors also allow for very elegant programming, which should be considered more than just syntactic sugar.

Destructors cannot use the `thisreturn` modifier. The rationale for this integrity rule is to assure the `this` pointer will not be stored in a variable which will be alive after the destructor has been run. If that would be possible the latter variable would point to a "dead" object or point to memory which hads been assigned to a new object of potentially different type. This would be a first-rate security and integrity risk and must not be allowed to happen.

Consider the following example:

```
int Banking::doSomethingWithTheDatabase()
{
      DatabaseConnection db("rdbms.bank.de","wolfgang","geheim");
      ResultSet rs=db.query("select * from accounts......");
      if(!rs.hasMoreRows()){ return -1; } //failure
      while(rs.hasMoreRows())
      {
           Row r=rs.getNextRow();
           //process row
      }
      return 1;//success
}
```

The interesting point here is that the destructor of the database connection will be called at both return statements implicitly and will free the resources related to the database connection (or put them back into a pool). With Java, one would have to manually call db.disconnect() before each return statement. Additionally, a finally() block would be needed in Java(R), to handle exceptions.

An important semantic rule for destructors is that the `this` pointer of the object to be destructed must not be assigned to any variable in the destructor or in any methods called by the destructor.

Chapter 3

Conventions, Abbreviations

1. EOF End Of File

Chapter 4

Build Process Overview

4.1 Phase 1

The SAPPEUR Compiler will at first parse all declaration (*.ad) files in a project directory, in the order defined by the file `compileorder`. The declaration files must be separated by a comma (,).

Example content of file ”compileorder”:

```
String.ad, System.ad, auto.ad,  ContainerClass.ad,
LightTruck.ad,RC4Cipher.ad,Scanner.ad,UnitTest.ad,
ThreadedMatrixMultiplier.ad,  Matrix.ad, Lotto.ad
```

A user-defined type will be in the symbol table as soon as its declaration has been parsed completely. From that point on, the type will be available for use as a superclass (or interface in the case of interfaces). Pointer types can be incomplete at the point of definition of the pointer, but they must be defined before Phase 1 ends. Super classes, interfaces and the types of non-pointer class members must always be fully defined before they can be used.

4.2 Phase 2

The SAPPEUR Compiler will parse all implementation (*.ai) files in a project directory. Due to the structure of the SAPPEUR language, the order of compilation does not matter. Methods of a class can be scattered throughout all files. SAPPEUR programmers are encouraged to have a system for the location of methods and source files, though.

For each declaration and each implementation file, a correspondingly named C++ file will be generated. Generated C++ classes and methods will be in their "natural" place, as defined by the SAPPEUR programmer.

Chapter 5

Scanner

In the following, extended regular expressions (EREs) as defined by POSIX are used to specify the tokens of the SAPPEUR language. If one were to try these EREs out (using `egrep`) on a shell like `bash`, some ERE characters would have to be escaped using the backslash (\) character. The token specifications were tested using GNU grep version 2.5.3. The SAPPEUR Scanner itself is (roughly speaking) a C++ `switch` construct.

The SAPPEUR language defines the following token Types:

```
name of token type | extended regular expression      | example


literal char 1     | {

literal char 2     | }

literal char 3     | ~

literal char 4     | \*

literal char 5     | %

literal char 6     | ^

literal char 7     | ;

literal char 8     | \]

literal char 9     | \[
literal char 10    | (
literal char 11    | )
literal char 12    | ,
literal char 13    | .
literal char 14    | =
literal char 15    | +
literal char 16    | -
literal char 17    | :
literal char 18    | &
literal char 19    | |
literal char 20    | <
literal char 21    | >
operator 1         | ==
operator 2         | ++
operator 3         | ->
operator 4         | ::
operator 5         | ->
operator 6         | !=
operator 7         | !
operator 8         | !!
```

`operator 9`	`&&`	
`operator 10`	`<<`	
`operator 11`	`>>`	
`operator 12`	`>=`	
`operator 13`	`<=`	
`STRINGLITERAL`	`"[^"]*"`	`"wolfram"`
`INTEGERLITERAL`	`[0-9]+`	`176`
`CHARLITERAL`	`'[a-z]' or '\n' or '\0' or '\\'`	`'x'`
`IDENTIFIER`	`[a-zA-Z]{1,1}[a-zA-Z0-9_]*`	`vector3`

5.1 Comments

Comments can either be single-line or multi-line, as in C++ or Java(R). Single-line comments start with "//" and extend to the line break. Multi-line comments start with "/*" and run to the closing "*/". Nesting of multi-line comments is not possible.

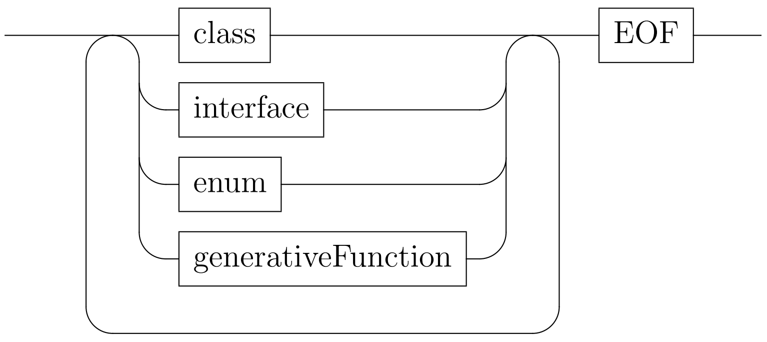
Chapter 6

Declaration File

A declaration file may contain classes, interfaces, enums or functions in any order and cardinality. The compiler must not assume any functional relationship of the file name and the things defined in the file.

6.1 Declaration File Grammar

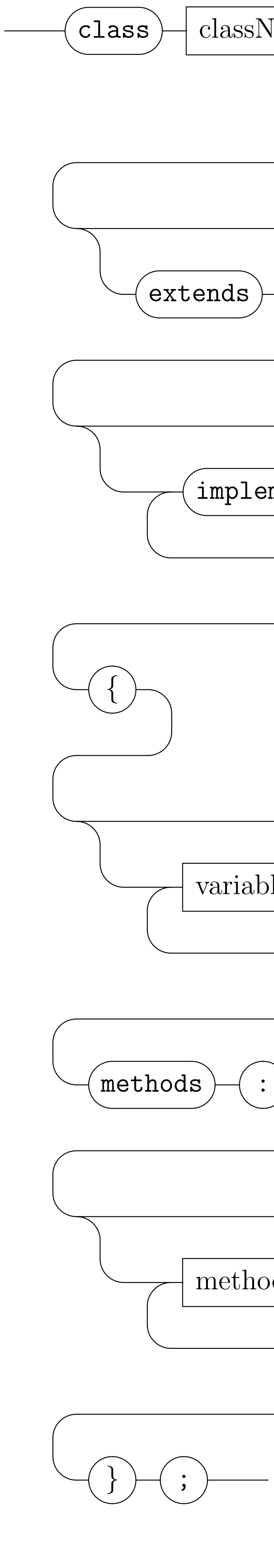
declarationFile



”EOF” denotes the end of the file

6.2 Class Definition

class



As the diagram makes clear, classes can only be derived from a single superclass. A class may implement zero or more interfaces. The generated C++ class is derived from class `SPRMTObject` if it is `multithreaded` or from `SPRObject` if not. A `multithreaded` class can be accessed from different threads of execution and can be passed as an argument to the `createThread()` construct. This kind of classes contain a Mutex, which will be locked each time an `external` Method is being called. This is similar to the Java(R) Monitor concept. The Mutex assures the integrity of the runtime system even when a data structure is manipulated by multiple threads. Also see chapter "Multithreading".

Examples:

A)

```
class ThreeDVector
{
     int x;int y;int z; methods: int
     length();
};
```

B)

```
class Person
{
   String _firstname;
   String _middle;
   String _lastname;
   String _placeOfBirth;
   String _dateOfBirth;
  methods:
   Person(&String first,&String last,
          &String placeOfBirth,&String date);
   void setStreet(&String street);
};
```

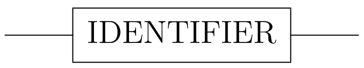
C)

```
class Officer extends Person
{
   String _rank;
   String _branch;
  methods:
   Officer(&String first,&String last,&String place,
           &String date, &String branch);
   void setRank(&String rank);
   void promote();
   void degradeTo(&String newRank);
};
```

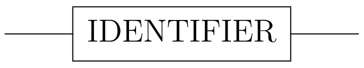
D)

```
class EngineeringOfficer extends Officer implements Engineer
{
           String _specialty;
          methods:
           EngineeringOfficer(&String first,&String last,
                              &String place,&String dateString,
                              &String branch);
           void setSpecialty(&String specialty);
};
```

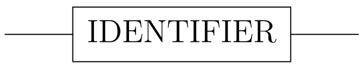
className



superClassName

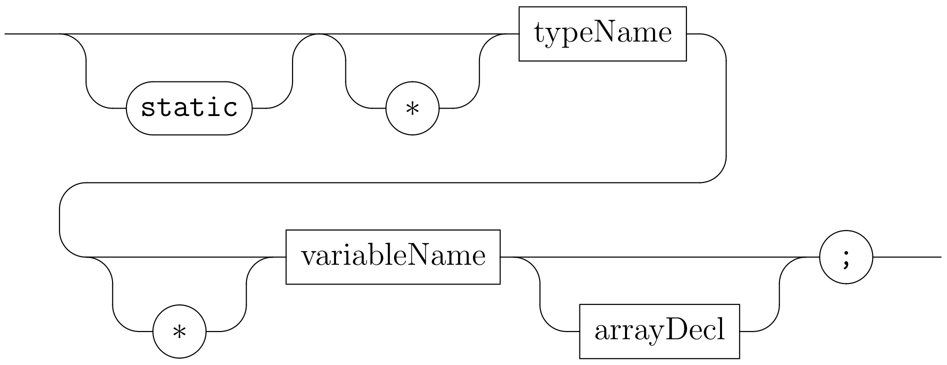


interfaceName



6.3 Member Variable Declaration

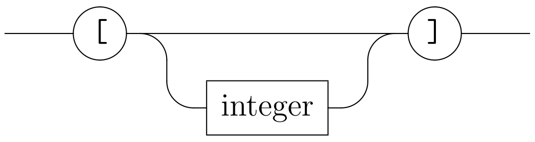
variableDeclaration



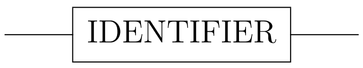
Description of production `variableDeclaration`:

1. `static`: indicates that this variable exists only once in the program. Variable must be of a `multithreaded` type.
2. first asterisk: indicates an array pointer
3. `typeName`: type of the variable
4. second asterisk: indicates that this variable contains a pointer
5. `variableName`: name of variable
6. `arrayDecl` : indicates that this variable is an array. If the variable is not a pointer to an array, the array size must be specified.

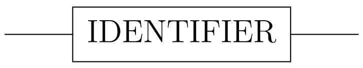
arrayDecl



typeName



variableName



Examples:

A)

```
class VerySimple
{
    int _x;
   methods:
};
```

B)

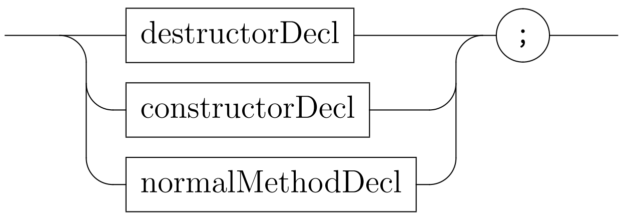
```
class ABitMoreComplex
{
    int _array[10];
   methods:
};
```

C)

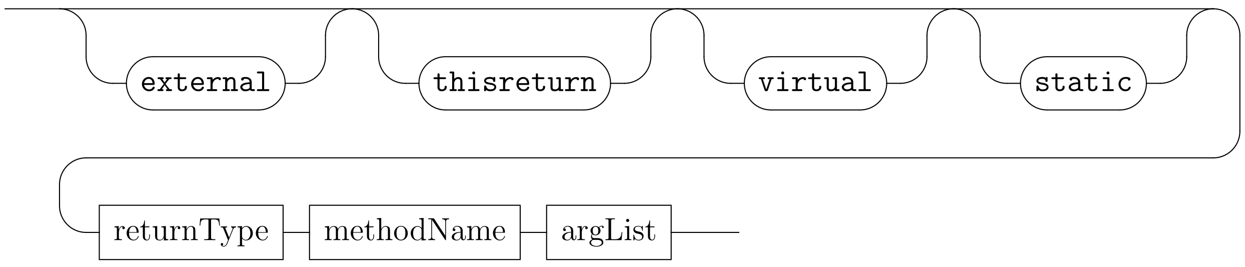
```
class EvenMore
{
    int    _x[10];
    String _dictionary[2000];
   methods:
};
```

6.4 Method Declaration

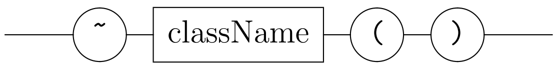
methodDeclaration



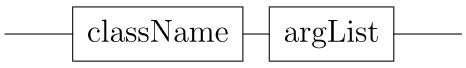
normalMethodDecl



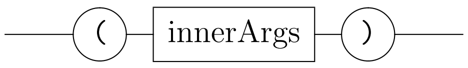
destructorDecl



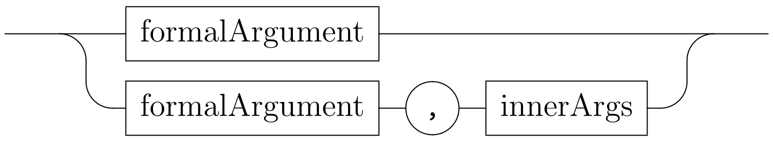
constructorDecl



argList



innerArgs



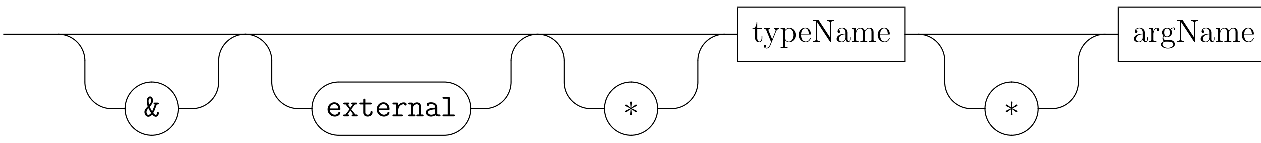
1. `external` can only be used with `multithreaded` classes. It indicates that this method may be called from the context of other objects and from multiple threads.

2. `thisreturn` indicates that this method can return a pointer to the object itself. This implies that the current method may not be called on a stack-allocated object.

3. `virtual` indicates that calling this method includes dereferencing a function pointer, depending on the current sub-type. This is the same mechanism as in C++.

4. `static` indicates that this method can only access static members of this class. This is the same mechanism as in C++.

formalArgument



1. '&' indicates that this argument is a reference. That means it is not copied upon calling the method.

2. `external`: in case this class is `multithreaded`, this indicates that the argument cannot be stored in a member variable of the class. (only applies to pointers)

3. first asterisk: indicates that this argument is a pointer to an array

4. typeName specifies the type of the variable

5. argName specifies the name of the argument, which is used to refer to the variable inside the method body

6. second asterisk: indicates that the argument is of a pointer type. If there is also the first asterisk specified, the variable is a pointer to an array of pointers

7. arraySpec specifies that the variable is an array.

A formal argument may be a reference, which is indicated by the ampersand. The first asterisk indicates that the variable is a pointer to an array. The second asterisk indicates that the variable does contain pointers instead of values. Finally, the array specification indicates an array of values or pointers.

Examples:

A) `int fac(int x);`//A method having an integer input and returning an integer

pointer to a Customer as input and returning a pointer to an invoice object

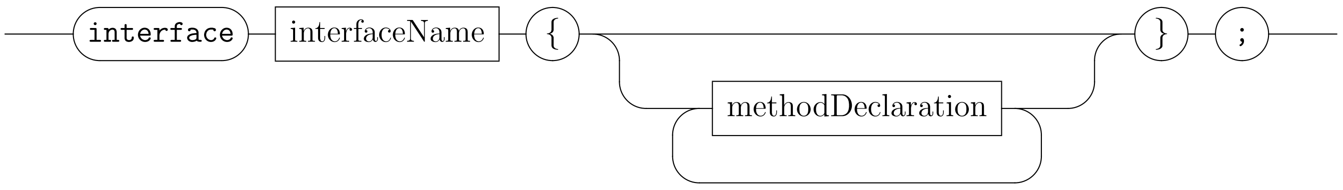
C) `void incrementInt(int& x);`//A method modifying an integer

D) `void createFleet(&*Ship* fleet[]);`//A method setting a pointer to an array of Ship pointers.

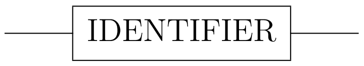
E) `void createFleet(&*Ship* fleet[], int numberOfShips);`//Another method setting a pointer to an array of Ship pointers.

6.5 Interfaces

interface

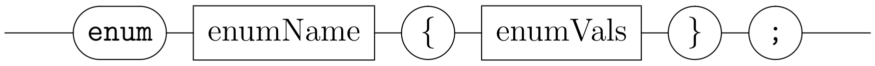


interfaceName

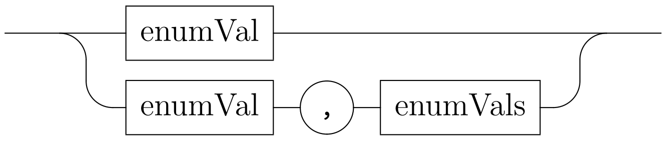


6.6 Enumerations

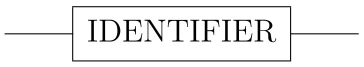
enum



enumVals



enumVal

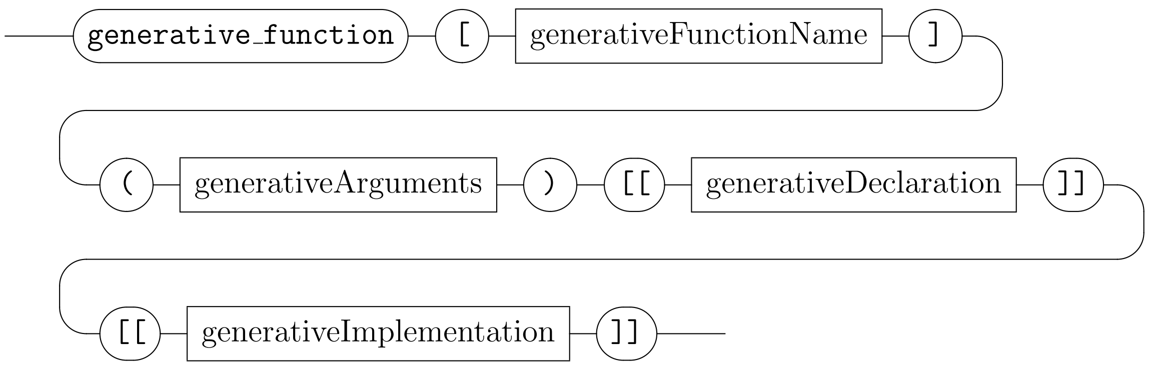


Examples:
A) `enum favouriteMammals{man,dolphin,dog,cat,mouse,tiger};`
B)

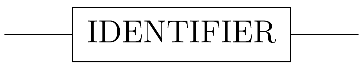
```
enum meansOfTransport\{aircraft,auto,bicycle,motorbike,lorry,
                      helicopter,segway,train,gyrocopter,
                      ship,hovercraft,uboot};
```

6.7 Generative Functions

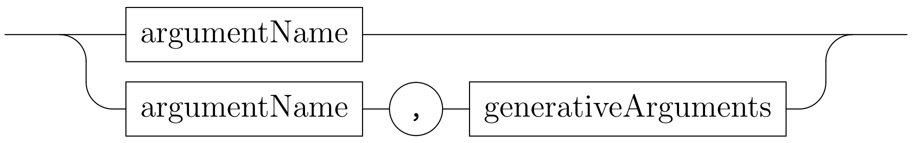
generativeFunction



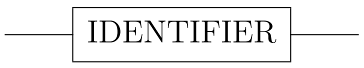
generativeFunctionName



generativeArguments



argumentName



Generative functions are similar to the macro mechanism of the C++ preprocessor. There is a crucial difference which makes SAPPEUR generative functions significantly more powerful (the separation of declaration and implementation, namely). Major usage scenarios of this feature include generic container classes and strings with a configurable pre-allocated buffer.

Generative functions are based on a very simple concept of string replacement and invocation of an SAPPEUR compiler instance upon "instantiation" of a generative function. A generative function is instantiated using the `generate` expression. The following process takes place while instantiating a generative function:

1. All variable occurrences (the `@variable@` syntax) will be replaced by the actual argument of the instantiation. This happens both in the generative declaration (all the SAPPEUR code inside the first [[..]] construct) and in the generative implementation (all SAPPEUR code inside the second [[..]]

construct). The replacement is done in a very literal manner - as soon as an '@' is encountered, the use of a generative Variable is assumed. The end of a generative Variable name is indicated by a second '@'. '\' can be used to escape the next character.

2. The result of this literal replacement inside of the generative declaration code will be fed to a new instance of the declaration parser. This means that after this literal replacement of generative variables, the result must conform to the grammar specified by production `declarationFile` (see above). This will normally result in a new type or enum being added to the list of types.

3. The same literal replacement of generative functions takes place in the generative implementation code specified in the second construct. The only difference to step 2. is that the resulting code must conform to the `implementationFile` production (see above), as it is passed to an instance of the implementation parser. This will normally result in the creation of C++ methods. **As the implementation code will be compiled separately from the declaration code, cyclic dependencies are possible. This is more powerful than the C++ #define amacro() ... approach**

Example In this example, a very simple generic hash-table will be shown. To keep this sample short, the key is assumed to be an integer.

```
//in an *.ad file
generative_function[SimpleHashtable](type, capacity)
[[
    class SimpleHashtable_@type@_@capacity@
    {
          @type@* _array[@capacity@];
         methods:
          void insert(@type@* object);
          @type@* get(int key);
    };
]]

[[
    //note: for purpose of illustration, hash collisions are not
    //      handled here
    void SimpleHashtable_@type@_@capacity@::insert(@type@* object)
    {
         var int hash=object.key % _array.sz;
         _array[hash]=object;
    }

    //note: for purpose of illustration, hash collisions are not
    //      handled here
    @type@* SimpleHashtable_@type@_@capacity@::get(int key)
    {
          var int hash=key % _array.sz;
          var @type@* obj=_array[hash];
          if( obj != NULL)
          {
              if(obj.key==key){return obj;}
          }
          return NULL;
    }

]]

class SimpleHashable
{
    char _payload[50];
    int key;
   methods:
};

class xyz{
   methods:
    generate SimpleHashtable(SimpleHashable,1000);
   void doSomething();
};

//in an *.ai file

void xyz::doSomething()
{
   var SimpleHashtable_SimpleHashable_1000 table;
   var SimpleHashable* sh=new SimpleHashable;
   sh.key=77;
   table.insert(sh);
}

//The internally generated SAPPEUR Code during instantiation:
```

```
class SimpleHashtable_SimpleHashable_1000
{
      SimpleHashable* _array[1000];
     methods:
      void insert(SimpleHashable* object);
      SimpleHashable* get(int key);
};


void SimpleHashtable_SimpleHashable_1000::insert(SimpleHashable* object)
{
     var int hash=object.key % _array.sz;
     _array[hash]=object;
}

SimpleHashable* SimpleHashtable_SimpleHashable_1000::get(int key)
{
      var int hash=key % _array.sz;
      var SimpleHashable* obj=_array[hash];
      if( obj != NULL)
      {
          if(obj.key==key){return obj;}
      }
      return NULL;
}
```

6.8 Built in Data Types

Sappeur has the following builtin data types:

1. Boolean
2. character (”char”)
3. short (see chapter ”Builtin Numeric Data Types”)
4. integer (see chapter ”Builtin Numeric Data Types”)
5. long long integer (see chapter ”Builtin Numeric Data Types”)

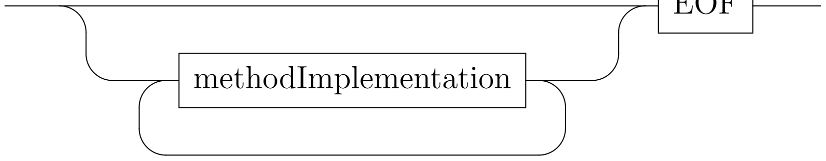
6. float, double (see chapter "Builtin Numeric Data Types")

Chapter 7

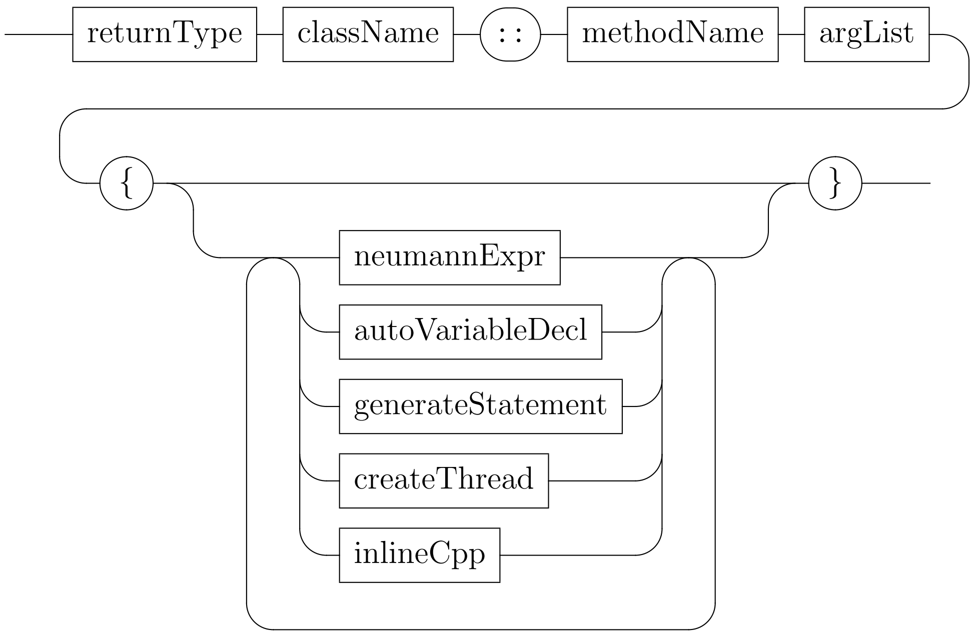
Implementation File

Implementation files must be named *.ai. They can contain an any number of method implementations of arbitrary classes. All classes must have been defined in a declaration file.

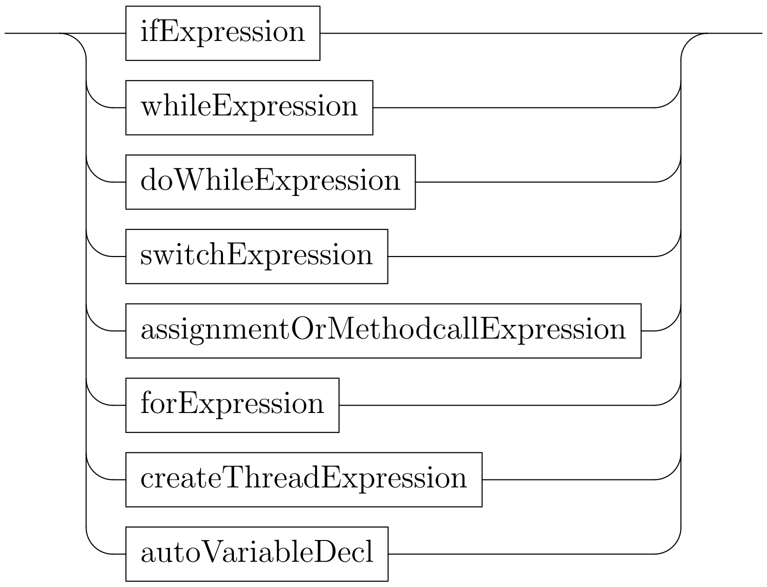
implementationFile



methodImplementation



neumannExpr



7.1 Block of SAPPEUR Code

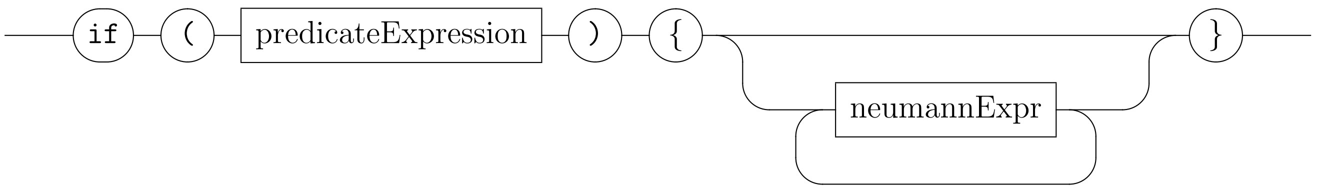
A "block" of SAPPEUR code normally starts with opening braces ({) and ends with closing braces (}). A block may contain sub-blocks. The semantic meaning of a block is very similar to the equivalent C++ and Java construct. Blocks control the scoping and life-cycle of automatic and method argument variables. Whenever the thread of execution leaves a block, automatic variables and method arguments will be reclaimed. This means that

1. variables of a basic data type are just ”forgotten”
2. objects allocated on the stack will have their destructor called
3. pointers will call the release() method on the object they point to. The latter may cause this object to call its destructor (if reference count is zero). If a pointer is NULL, the pointer is just ”forgotten”. In C++, this would be called a ”Smart Pointer”. This behavior must of course be thread-safe for `multithreaded` objects.
4. arrays will perform the same operations on their elements

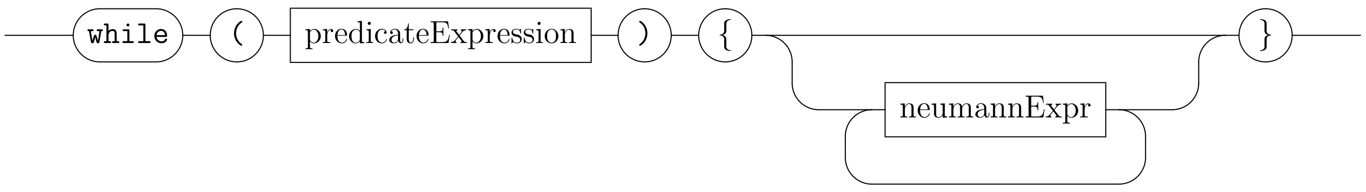
7.2 Program Flow Control Expressions

Program flow control expressions are nearly identical to those available in C++ or Java(R).

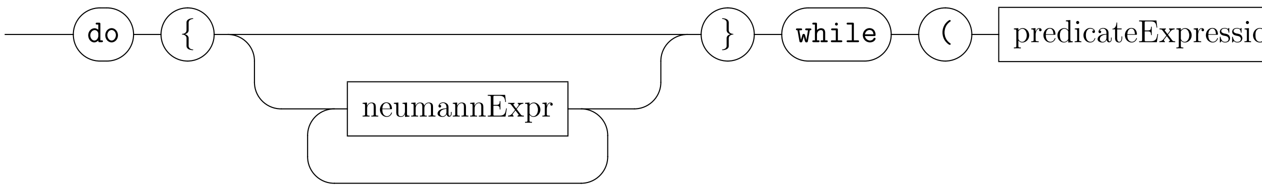
ifExpression



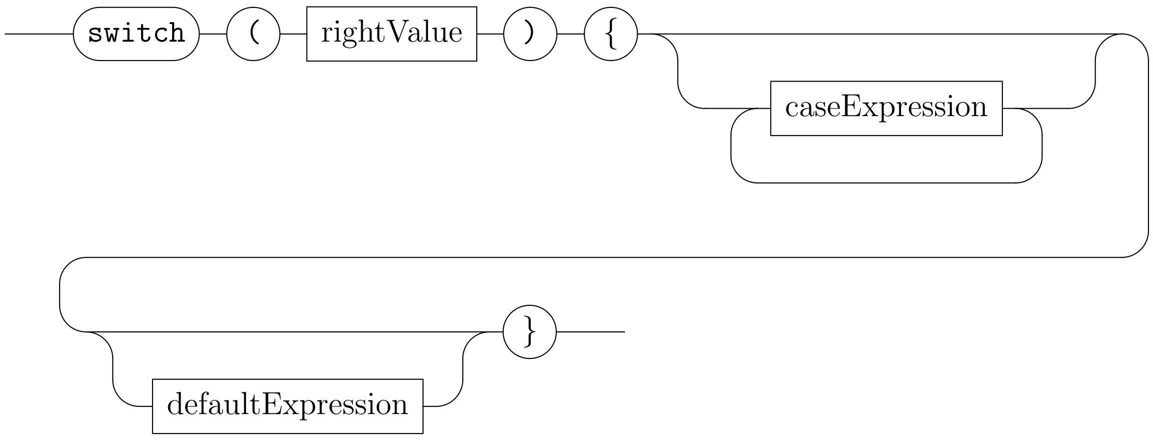
whileExpression



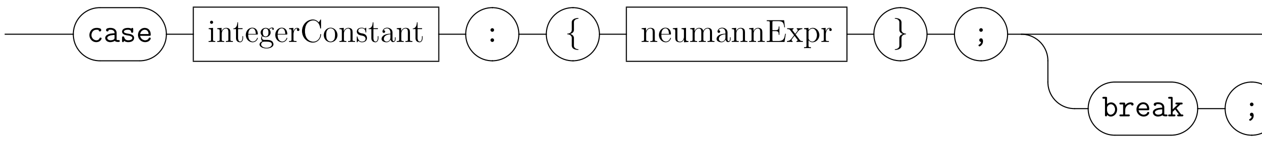
doWhileExpression



switchExpression

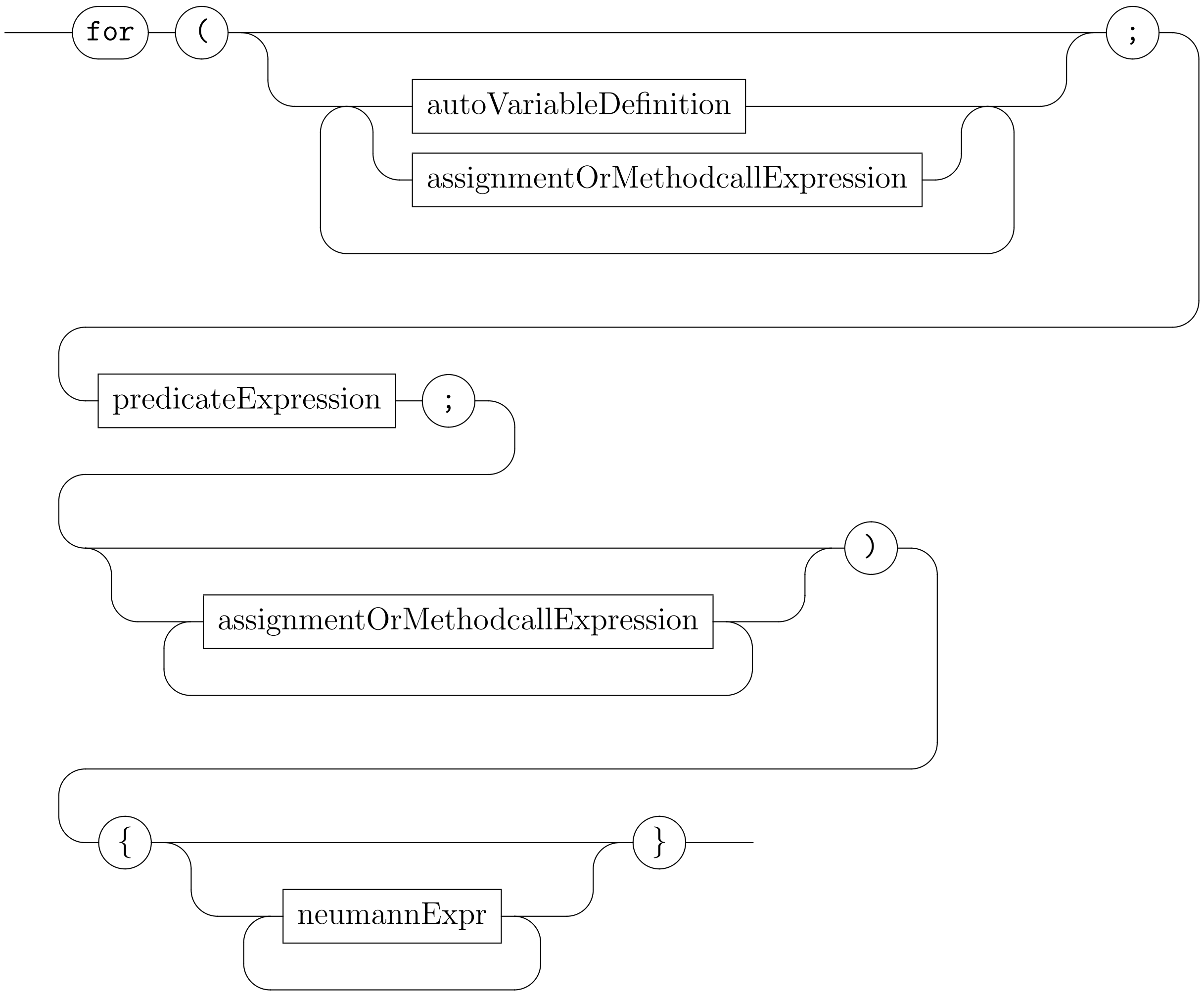


caseExpression



7.3 For Loops

forExpression


for
(
autoVariableDefinition
assignmentOrMethodcallExpression
;
predicateExpression
;
assignmentOrMethodcallExpression
)
{
neumannExpr
}

SAPPEUR `for` loops behave very similar to the `for` loops of C++ and Java. There is one difference, though: Variables defined in a for expression are scoped to exist only in the for expression and its sub-expressions.

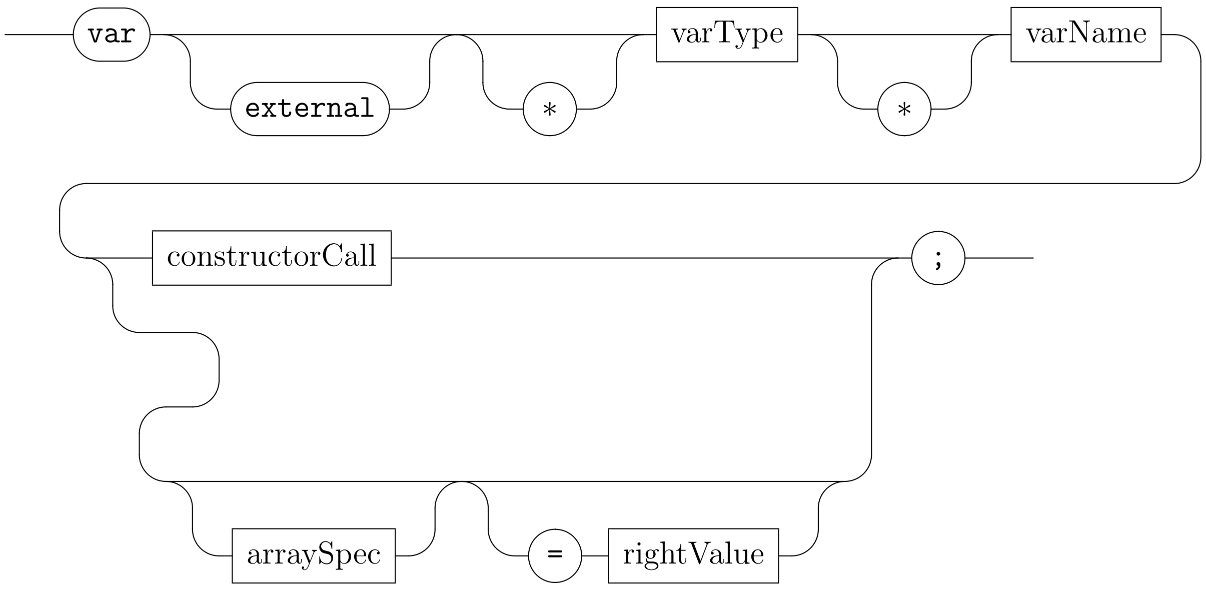
7.4 Automatic Variables

Automatic Variables allow for the allocation of variables on the stack, which is very efficient. The memory needed for the variables most often is already in the microprocessor's cache and does seldomly cause costly main memory interactions. After leaving a context, automatic variables have their memory implicitly reclaimed simply by the stack pointer arithmetics (for basic types at least).

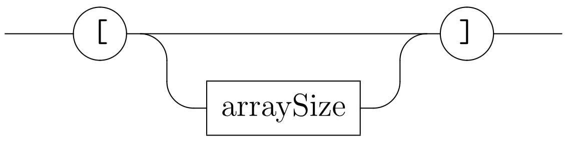
SAPPEUR Programmers should aim to use automatic variables whenever possible and resort to creating variables on the heap only when absolutely necessary.

Upon leaving the current block, the destructor of automatic variables will be called. This is the same behaviour as in C++. Basic types do not have destructors.

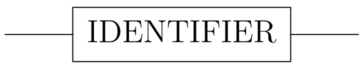
autoVariableDecl



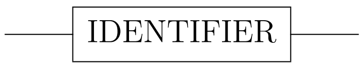
arraySpec



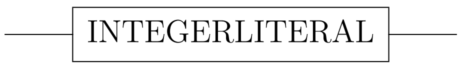
varType



varName



arraySize



Description of the autoVariableDecl production from left to right:

1. `var`: indicates the beginning of the declaration of an automatic variable.

2. `external`: indicates that the variable may point to an object E which is not local to the current `multithreaded` object C

3. varType: the type of the variable

4. first asterisk: indicates that this variable is a pointer to an array

5. varName: name of variable

6. second asterisk: indicates that the variable is of a pointer type

7. constructorCall: arguments for calling a constructor of a user-defined type

8. arraySpec: indicates that this variable is an array. Arrays of user-defined types and pointers will be initialized by calling their default constructor or NULL, respectively. Arrays that are automatic variables and are not pointers must be of a defined size. Pointers to arrays **cannot** have a defined size.

9. rightValue: Automatic variables of type `char` and `int` can be initialized by a `rightValue` expression.

Scoping of Variables: The scope of an automatic variable starts from its definition and extends to the closing brace of the current block. A block normally starts with an opening brace and ends with a closing brace. In an enclosed block, variable names must not be reused in order to avoid any ambiguity. This is different to C++, Java(R) and C#(R).

Examples:

```
A) var int x;

B) var int y=17+x;
```

```
   C) var char firstname[30];

   D) var *char lastname[];

   E) var *Ship* fleet[];
```

Example:

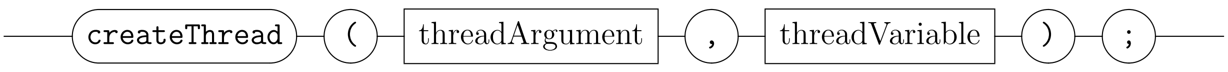
```
if(i==7)
{
   anObject.foo(x);//error:x not yet defined
   var int x=6;
   foo(x);//ok

   if(x==i)
   {
      var int x=6;//error: redefinition of x
   }
}
```

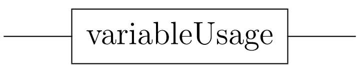
7.5 Create Thread Expression

The Create Thread Expression enables the SAPPEUR programmer to start a new thread of execution. Semantic rules governing this mechanism guarantee that the integrity of the run-time system is assured. The first argument of the `createThread` clause is the Thread Argument. This object must be `multithreaded` and have an `external` method `void threadMain();`. This method will be called upon starting the thread and is the logical equivalent of the function pointer that must be passed to `pthread_create()`. The second argument is the Thread Variable, which must be of Type `ThreadInfo*`. This variable will hold a handle to the created Thread.

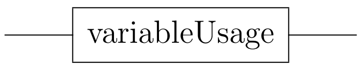
createThreadExpression



threadArgument



threadVariable



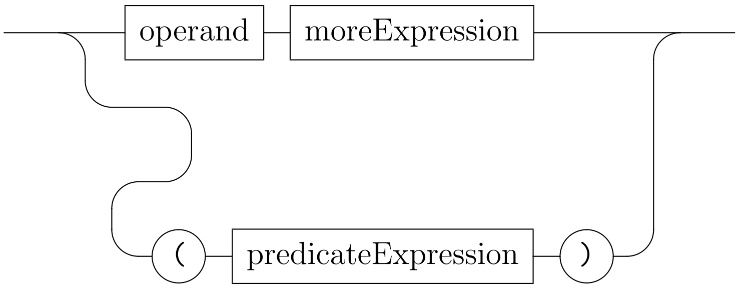
Examples:

A) Start three threads that each print the numbers from 0 to 1000000.

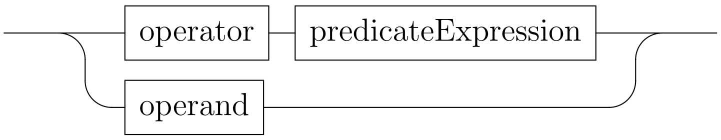
```
//in an *.ad file:
class NumberPrinter multithreaded
{
    methods:
      external void threadMain();
};
<other file>
//in an *.ai file:
void NumberPrinter::threadMain()
{
    for(var int i=0;i<=1000000;i++)
    {
       inline_cpp[[printf("%i\\n",i);]]
    }
}
<snip>
//in an existing thread of execution:
for(var int i=0;i<3;i++)
{
    var ThreadInfo* ti;
    var NumberPrinter* np=new NumberPrinter;
    createThread(np,ti);
}
```

7.6 Boolean Expression

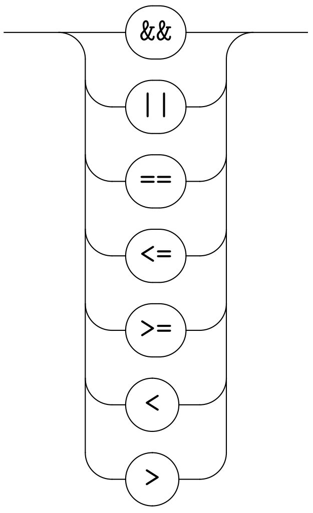
predicateExpression



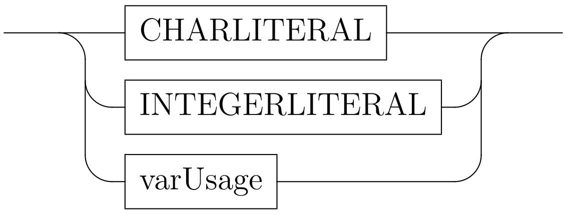
moreExpression



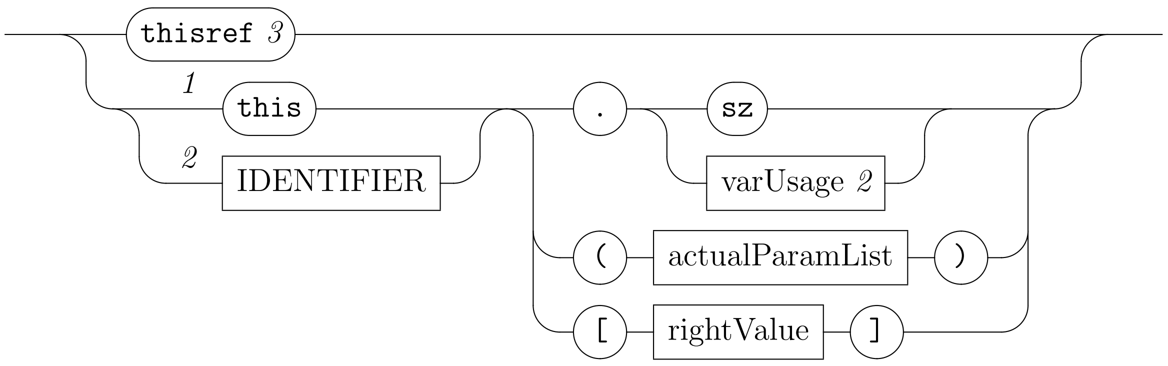
operator



operand



varUsage



Description

1. `thisref`: This keyword is equivalent to `*this` in c++, which means that it returns a reference to the object.

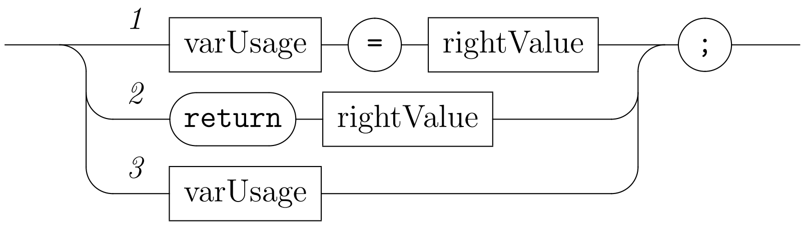
2. `this`: This keyword is equivalent to `this` in c++, which means that it returns a pointer to the object.

3. `sz`: This keyword may be used to determine the size of an array.

Examples:
A) `var int x=3; ...  ; if(x==7){doSomething();}`
B) `var Customer x; ...  if(x.networth>1000000){processFatCat(x);}`
C) `var Customer x; ...  while(x.isCreditWorthy()){sellSomething(x);}`
D) `var int x=3,z=77; ...  ; if( (x==7) && (z==100) ){doSomething();}`
Note: All operators on the same level in the imagined parse tree must be the same.

7.7 Assignment, Method Call or Return Statement

assignmentOrMethodcallExpression



Variant 1 does not permit calling a method in the leftmost varUsage clause.

Examples:

A) `x=17;`

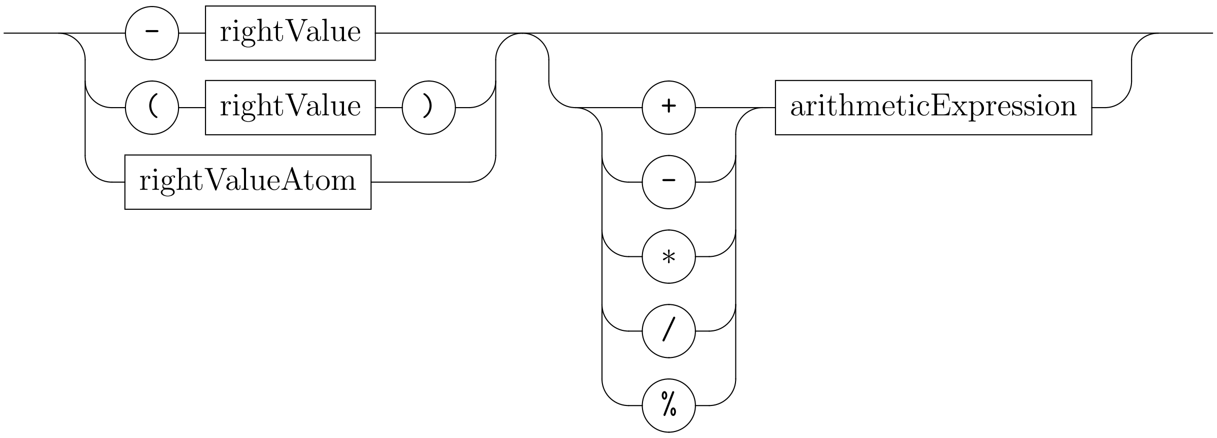
B) `x=17+y*6;`

C) `var ADC adc; var int n; ...  n=adc.getBitcount();`

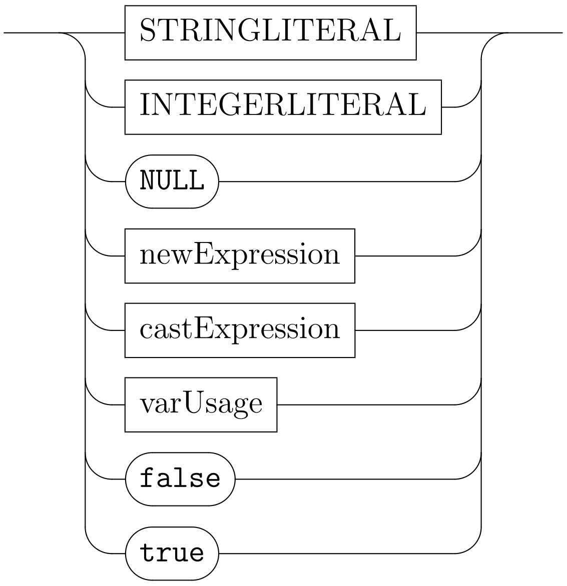
D) `var ADC adc; ...  adc.startSampling();`

Note: Regardless of whether a variable is a value or a pointer, members are accessed using the dot (".") operator. There is no "-¿" operator in SAPPEUR.

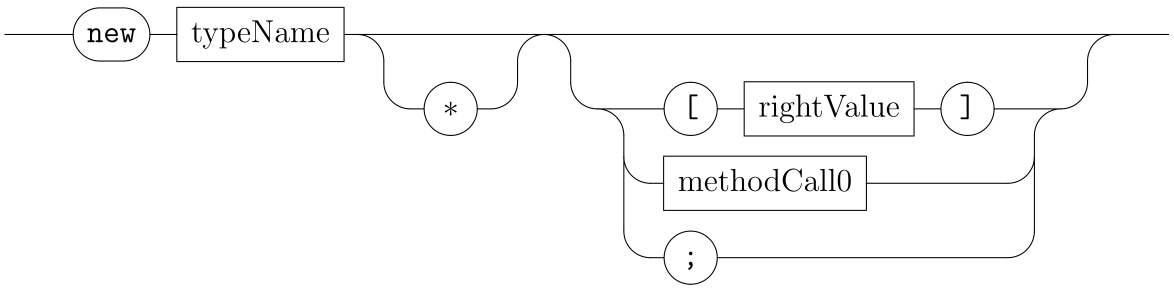
rightValue



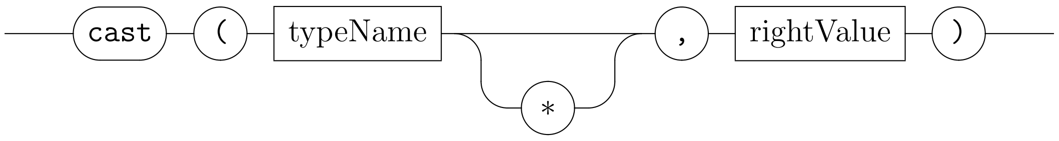
rightValueAtom



newExpression



castExpression



There are no automatic casting conventions in SAPPEUR. All casting must be done explicitly using the `cast` clause. The first argument of the cast clause is the type to be converted to, whilst the second argument is an expression to be casted.

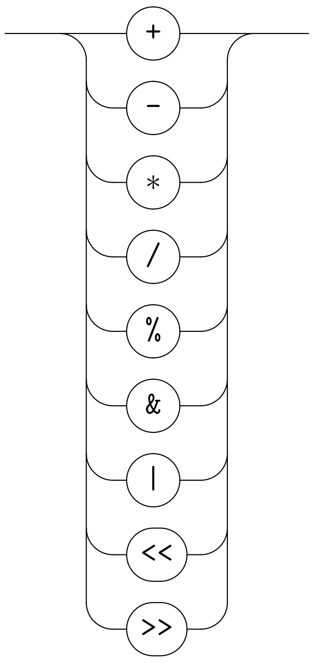
Examples:

A) `var char c; var int y; ...  c=cast(char,y+7);`

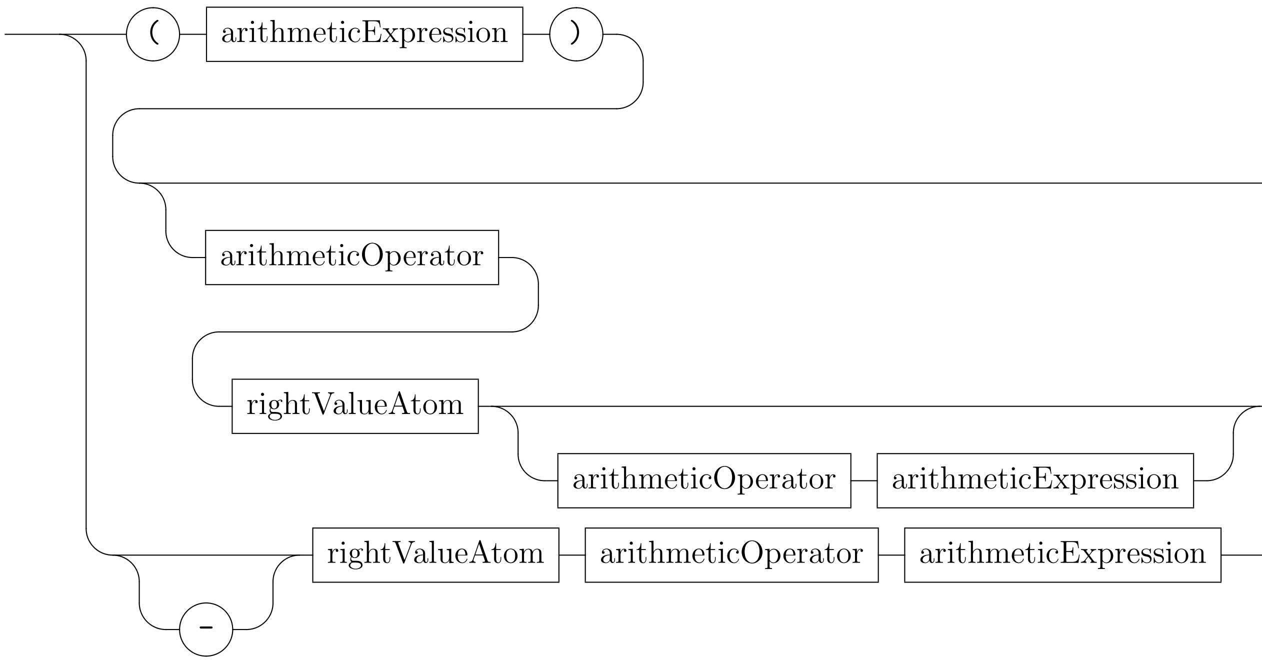
7.7.1 Arithmetic Expressions

Arithmetic Expressions in Sappeur are handled exactly in the same manner as C++ arithmetic expressions. This is due to the fact that they are more or less copied into the generated C++ code. Users are encouraged to use braces liberally, as the precedence rules in C++ arithmetic expressions aren't simple.

arithmeticOperator



arithmeticExpression



Examples:
A) `var int x; ...  x=17*i+a.fac(i+3)*(j[k]+i);`

B) `var int x[10]; ...  x[i]=17*i+a.fac(i+3)*(j[k+r[m]]+i);`

Chapter 8

Builtin Numeric Data Types (”Numbers”)

8.1 General Remarks on Numbers

Sappeur numbers are very similar to numbers in C++. The mapping is more or less one-to-one. Sappeur does not support unsigned numbers.

8.2 int

Integers are signed integers and mapped to the C++ type ”int”. On most architectures/compilers this means that it is a signed 32 bit numbers (twos-complement), also on 64 bit systems.

8.3 longlong

long long Integers are signed integers and mapped to the C++ type ”long long”. On most architectures/compilers this means that it is a signed 64 bit numbers (twos-complement).

8.4 double

A double floating-point number is mapped to the C++ type of the same name. This normally is a 64-bit IEEE754 floating-point number. Note that the x86 CPU internally has a storage resolution of 80 bits and will therefore produce slightly different results than most other IEE754-based CPU architectures. Sappeur is not different to C++ in this respect.

8.5 float

A "normal" floating-point number is mapped to the C++ type of the same name ("float"). This normally is a 32-bit IEEE754 floating-point number. Also see section "double".

8.6 casting numbers

The general policy of Sappeur is to require explicit type casts, even if an automatic (C-style) cast is conceivable and could be implemented reasonably. In some cases (expressions), numbers will be automatically casted into a type of the following list:

```
(int, long,longlong,float,double)
```

It is only possible to cast an expression from the left to the right direction in the list above. In general, use the cast Expression liberally to express your intentions.

Example:

```
var int x=77; var float y; y=cast(float,x);
```

Chapter 9

Multithreading

The rationale for creating SAPPEUR was the current trend to multi-core microprocessors. At least for the foreseeable future, progress in the semiconductor industry will not manifest itself in a much higher number of clock cycles per second. Better process technologies will allow for much higher transistor and wire counts on high-volume chips, though. Industry standard PC microprocessor chips from INTEL(R) and AMD(R) already contain two or for "cores" (logical processors). Other companies (e.g. SUN(R)) are selling CPUs with even higher numbers of cores on a single chip.

The typical development process of a C++ developer includes compiling, debugging and stress testing of her code. This works reasonably well for single-threaded programs, but it fails for multithreaded software, because the number of different execution traces quickly grows astronomical. By stress testing alone, a developer simply cannot gain much confidence about a multithreaded C++ program. Strictly speaking, this is true for any programming language, but it is especially critical for C++ programs, because memory can be corrupted so easily. As these errors do often manifest themselves in a totally correct piece of code, finding the root cause of a bug is often virtually impossible or at least too expensive.

9.1 The SAPPEUR approach to Multithreading

The creator of SAPPEUR does not pretend that there is a magic bullet solution to creating efficient and robust programs. Real-world software engineering often is as intellectually challenging as the creation of a new airplane design or the development of a novel drug. Only through hard work, good organization, experience, creativity and the right tools we can achieve great results. Sometimes the objectives are too ambitious with the given people, processes, technology, financial resources and time budget. Failure must be accepted as a business risk as it is accepted in other large scale engineering and construction projects, even though the political implications of that may be severe.

Instead of proposing a totally new approach to programming, the creator of SAPPEUR started out with the observation that single-threaded performance is very important for a multi-threaded program. SAPPEUR harnesses the high efficiency of C/C++ and adds robust multithreading functionality. SAPPEUR borrows a lot from ADA's communicating-sequential-process (CSP) metaphor.

SAPPEUR's basic idea could be called "sequential processes safely sharing memory". The CSP metaphor can of course be emulated efficiently with shared memory. SAPPEUR's creator does not support the notion that automatic parallelizers of any kind can speed up a broad spectrum of algorithms/systems. Instead, the SAPPEUR philosophy relies on the skills and hard work of an engineer to devise a strong concept of multithreading for the problem at hand. SAPPEUR Programmers must understand the economics (time and space costs) of using contemporary hardware. Without a basic appreciation of registers, caches, cache lines, main memory, process switch costs and network transmission costs, one should not even start out to create a high-performance software system.

9.2 Multithreading in the Type System

The SAPPEUR type system makes multithreading explicit, as all classes to be accessed by more than one thread at a time must be marked as `multithreaded` in the definition of the class. Each object of a `multithreaded` class does contain a Mutex (Mutual Exclusion facility), which is protecting this instance. The following rules exist for objects of a `multithreaded` class:

1. upon calling an `external` method, the object's Mutex will be locked. Recursive locks are possible.

2. by default, members of an object can only be accessed by the methods of the same object.

3. methods declared as `external` can be accessed from other objects

4. non-`multithreaded` data members cannot be passed out of an `external` method call. This means that such data members cannot A)be returned or B)assigned to a pointer reference.

5. regular methods (those without a `external` modifier must only be called from a context that belongs to the same object

6. parameters of an `external` method are implicitly `external`

7. parameters of an `external` method can only be assigned to a data member, if the parameter is `multithreaded`

8. in the context of a method, automatic variables may be declared as `external` and can in this case be assigned a method argument which is `external`

9. all classes that are derived from a `multithreaded` class are implicitly `multithreaded`, too

10. all classes that are **not** derived from a `multithreaded` class are implicitly not `multithreaded`

11. accessing data members of a class directly (using the '.' operator) is not possible

12. a variable used in a return `return` statement of an `external` method must be of one of the folloing types: A) a value type (like `bool`, `int` or `char`), B) of a `mulithreaded` type or C) an `external` variable.

13. value-type variables like `bool`, `int` or `char` can be freely exchanged through `external` and non-`external` method calls. The `external` specifier has no meaning for these variables.

14. only classes can be `multithreaded`

15. reference counting of `multithreaded` objects must be thread-safe

16. the thread parameter of a `createThread()` statement must be `multithreaded` and have a method `void threadMain();`

9.3 Example

The following example will perform a multithreaded Matrix multiplication of two 300x300 matrices of integers.

In an SAPPEUR definition file:

```
//This is not intended to be a benchmark algorithm,
//but a demonstration of SAPPEUR multithreading capabilities.
//There is plenty of scope for improving performance;
//different storage strategies for the two operand matrices, par example.

class ThreadedMatrixMultiplier multithreaded
{
    ThreadedMatrixMultiplier* _multi;
    MTMatrix* _left;
    MTMatrix* _right;
    MTMatrix* _result;
    int _rowIdx;
    int _rows;
methods:
    ThreadedMatrixMultiplier(MTMatrix* left,
                             MTMatrix* right,
                             MTMatrix* result,
                             int rowIdx,
                             int rows);

    external void threadMain();
    external ~ThreadedMatrixMultiplier();

    //test methods:
    static void runTest();
    static void initMTMatrix(MTMatrix* m);

};

class MTMatrix multithreaded
{
  *int _data[];
  int _size;
 methods:
    MTMatrix(int size);
    external void setAt(int x,int y,int val);
    external int  getAt(int x,int y);
    external void getRow(int y,&*int row[]);
    external void getColumn(int x,&*int column[]);
    external void threadMain();
    external int getSize();
    external void print();
    ~MTMatrix();
};
```

Somewhere in an SAPPEUR implementation file:

```
void ThreadedMatrixMultiplier::threadMain()
{
       var external *int rowData[]=new int[_left.getSize()];
       var external *int columnData[]=new int[_right.getSize()];
       var int summe;
       var int size=_left.getSize();
       var int maxRow=_rowIdx+_rows;
       if(maxRow > size){maxRow=size;}

       //core multiplication code
       //each threads "knows" its starting row and the number of
       //rows it has to process
       for(var int ri=_rowIdx;ri<maxRow;ri++)
       {
           _left.getRow(ri,rowData);
           for(var int x=0;x<size;x++)
           {
              summe=0;
              _right.getColumn(x,columnData);
              for(var int x2=0;x2<size;x2++)
              {
                    summe=summe+rowData[x2]*columnData[x2];
              }
              _result.setAt(x,ri,summe);
           }
       }
}


ThreadedMatrixMultiplier::ThreadedMatrixMultiplier(
                                          MTMatrix* left,
                                          MTMatrix* right,
                                          MTMatrix* result,
                                          int rowIdx,//starting row of this
                                                     //worker thread
                                          int rows)  //number of rows to proce
{
   _left=left;
   _right=right;
   _result=result;
   _rowIdx=rowIdx;
   _rows=rows;
}


int MTMatrix::getSize()
{
   return _size;
}

void MTMatrix::setAt(int x,int y,int val)
{
   _data[y*_size+x]=val;
}

int MTMatrix::getAt(int x,int y)
{
   return _data[y*_size+x];
}

void MTMatrix::getRow(int y,&*int row[])
{
    if(row.sz != _size){return;}
    for(var int i=0;i<_size;i++)
    {
        row[i]=_data[y*_size+i];
    }
}

void MTMatrix::getColumn(int x,&*int column[])
{
    if(column.sz != _size){return;}
    for(var int i=0;i<_size;i++)
    {
        column[i]=_data[i*_size+x];
    }
}

//this is a dummy
void MTMatrix::threadMain(){}

MTMatrix::MTMatrix(int size)
{
   _size=size;
   _data=new int[_size*_size];
}


void MTMatrix::print()
{
   for(var int y=0;y<_size;y++)
   {
       for(var int x=0;x<_size;x++)
       {
           //using the inline mechanism for printing.
           //This should be encapsulated in a class for production programs
           inline_cpp[[printf("%i ",this->getAt(x,y));]]
       }
       inline_cpp[[printf("\\n");]]
   }
}

MTMatrix::~MTMatrix()
{
       inline_cpp[[printf("MTMatrix::~MTMatrix() %x\\n",
                  (SPRMTObject*)this);]]
}

ThreadedMatrixMultiplier::~ThreadedMatrixMultiplier()
{
    inline_cpp[[printf("~ThreadedMatrixMultiplier()"
                      "%x\\n",
                      (SPRMTObject*)this);]]
}
```

```
void ThreadedMatrixMultiplier::runTest()
{
   //multiply two 300x300 Matrices
   var int size=300;
   var MTMatrix* m1=new MTMatrix(size);
   var MTMatrix* m2=new MTMatrix(size);
   var MTMatrix* mRes=new MTMatrix(size);

   ThreadedMatrixMultiplier::initMTMatrix(m1);
   ThreadedMatrixMultiplier::initMTMatrix(m2);

   var *ThreadedMatrixMultiplier* args[]=
        new ThreadedMatrixMultiplier*[5];
   var *ThreadInfo* threads[]=new ThreadInfo*[5];
   var int rowsPerCore=m1.getSize()/4;
   var int i;
   for(var int row=0,i=0;row<size;row=row+rowsPerCore,i++)
   {
      args[i]=new ThreadedMatrixMultiplier(m1,m2,mRes,row,rowsPerCore);
      createThread(args[i],threads[i]);
   }

    //wait for the threads to finish their work
    for(var int ti=0;ti<i;ti++)
    {
       threads[ti].join();
       threads[ti]=NULL;
       args[ti]=NULL;
    }

    inline_cpp[[printf("m1: \\n");]]
    m1.print();
    inline_cpp[[printf("m2: \\n");]]
    m2.print();
    inline_cpp[[printf("Result: \\n");]]
    mRes.print();
}
```

Chapter 10

Input/Output

10.1 PrintfClass

This class may be used to print data formatted onto a console/standard output.

Methods of the class:

1. `&PrintfClass fstr(&char str[]);` set the formatting string; arguments are specified using the dollar ($) sign. This method must be called before arguments are being added.

2. `&PrintfClass sa(&char str[]);` set a string argument

3. `&PrintfClass sa(int num);` set an integer argument

4. `void pr();` actually print the string to the console/STDOUT

Example:

```
//example 1
var PrintfClass pfc;
pfc.fstr("the answer is $").sa(42).pr();

//example 2
var PrintfClass printfc2;
printfc2.fstr("$ $ $ $ $ $ $ $ $ $\n");
for(var int j=0;j<5;j++)
{
   printfc2.sa(j);
   printfc2.sa("x");
}
printfc2.pr();
```

10.2 File Class

This class can be used to read and write files on mass storage media.

1. `File(&char path[]);` open a file specified by path
2. `int systemHandle();` get the system file handle. A negative value normally indicates that the file could not be created/openend
3. `int read(int pos,int count,&char buffer[]);` read `count` bytes from position `pos` in the file

   returns the number of bytes actually read into the buffer
4. `int write(int pos,int count,&char buffer[]);` write `count` bytes at position `pos` into `buffer`

   returns the number of bytes actually written
5. `void close();` close the file. This is implicitly done upon the destruction of this object
6. `int size();` get the current size of the file

Example:

```
void Bar::foo()
{
  var PrintfClass pfc;
  var File f("c:\\temp\\SAPPEUR_test_file.txt");//file is opened
  f.write(0,10,"0123456789");
}//file is being closed
```

Chapter 11

The Name

A Sappeur or Sappeur-Pompier (french name for a member of the fire brigade) is an expert in the rapid construction and destruction of physical things. In the English language Sappers are combat demolition experts.

One of the most important features of the Sappeur language are destructors and the ability to rapidly destroy unneeded data structures. This differentiates this language from Garbage-collected languages.

Chapter 12

SAPPEUR License

Sappeur is distribued under the terms of the Sappeur license. (see license.txt which accompanies this software package).

Chapter 13

Reports And Comments

Please report any Bugs related to SAPPEUR technology (compiler, documentation, etc) to frankgerlach22@gmx.de .

13.1 About the Author

Frank Michael Gerlach was born in Württemberg/Germany. He studied Computer Science at Berufsakademie Stuttgart and HP Boeblingen from 1993 to 1997 and was awarded a "Diplom-Ingenieur" in "Informationstechnik". He has worked as a software development engineer on Document Management, Internet Banking, CAD/CAM, Flight Reservation, Stock Trading and Database Systems since.

Mr Gerlach invented the Sappeur language and the compiler and is the lead developer.

Chapter 14

Acknowledgments

1. .Net(R) is a trademark of Microsoft Corporation
2. Java(R) is a trademark of SUN Microsystems
3. Visual C++(R) is a trademark of Microsoft Corporation
4. INTEL(R) is a trademark of Intel Corporation
5. AMD(R) is a trademark of Advanced Micro Devices Corporation

Chapter 15

Rules Pertaining thisreturn

1. Objects which are allocated on the stack must not call `thisreturn` methods
2. Methods which are not `thisreturn` must not use `this` as a parameter for a method call
3. Only methods which are `thisreturn` may use `this` as a right-value