Gardens Point Component Pascal — Release Notes

John Gough

September 17, 2004

This document applies to GPCP version 1.3 for JVM (Java Virtual Machine)

1 Introduction

Gardens Point Component Pascal (*gpcp*) is an implementation of the *Component Pascal* Language, as defined in the Component Pascal Report[1] from Oberon Microsystems. It is intended that this be a faithful implementation of the Report, except for those changes that are explicitly detailed here. Any other differences in detail should be reported as potential bugs.

The distribution consists of four programs, and a number of libraries. The programs are the compiler *gpcp*, the make utility *CPMake*, a module interface browser tool *Browse*, and a tool for extracting public symbol metadata from assemblies written in the Java[2] language, *J2CPS*.

The compiler produces either *.NET* Common Intermediate Language (*CIL*) or *Java* byte-codes as output. The compiler can be bootstrapped on either platform. These release notes refer to the JVM platform.

There are a number of syntactic extensions to the *Component Pascal* language accepted by the compiler which are introduced to allow interworking with the native libraries of the underlying platform. The guiding philosophy in such cases is to not significantly extend the semantics of the constructs that form part of Component Pascal, but rather to provide syntax for accessing features of other languages, which have no direct counterpart in *Component Pascal*.

2 Overall Structure

2.1 Input and Output files

In normal usage the compiler creates two or more output files for every source file. If the file “`Hello.cp`” contains the module *Hello*, and is compiled, then the output files will be “`Hello.cps`” and “`Hello.class`”.

---

[1] The defining document is simply referred to throughout this document as *the Report*.

[2] Java is a registered trademark of Sun Microsystems.

In general for a module *MName*, the “`MName.cps`” file is the symbol file which contains the metadata that describes the facilities exported from the module. The program executable will be file “`MName.class`” in the package “`MName`”. If the module *defines* any record types, then there will be an additional class file for each such type, also defined in package “`MName`”. If a listing file is created it will have filename name “`MName.lst`”. The “`MName.cps`” and “`MName.lst`” files will be created in the current directory, while all of the class files will be found in the directory “`CP/MName`”.

Be aware that the stem name of the output files comes from the *module* name, and not from the source-file name. Thus if module *Foo* is in source file “`Hello.cp`” then all of the output files will have stem name “`Foo`”.

By default the compiler writes class files directly. However it is possible to force *gpcp* to produce an output text file in the *Jasmin* byte code assembly language. These files have filename extension “`*.j`”. The corresponding class files may then be produced by manually invoking `Jasmin` but this is not recommended as, Jasmin does not handle floating poing literal correctly. Nevertheless, it may be instructive to view the Jasmin output, in order to understand how programs are encoded for this exectution platform.

2.2 Invoking the compiler

The compiler is invoked from the command line using the following command line syntax —

`$> cprun gpcp` [*gpcp-options*] *files*

The gpcp-options are given in Figure 1.

In the *.NET* versions “/”is the option prefix, but “–” is recognized also. Any number of files may be added in a white-space separated list.

2.3 The cprun script

cprun is a script or batch file that lives in the “`gpcp/bin`” directory. A corresponding script **cpint** invokes the java system without the just in time interpreter. However, the Java Runtime System may be invoked directly without using the script, using the syntax —

`$> java` [*java-options*] `CP.gpcp.gpcp` [*gpcp-options*] *files*

If this format is used then the available *Java* options allow for a choice of *JIT* compiler, output directories, or to pass property values to the *Java* runtime.

2.4 Target choice

The compiler may choose its output language at runtime. The default output when running on the *JVM* platform is *Java* class files. The recognized options are —

`-target=net`	*this is the* .NET CIL *format*
`-target=jvm`	*this causes* Java *class files to be emitted*
`-target=dcf`	*this chooses the Gardens Point “d-code” form*

The *Java* output option produces either *JVM* class files directly, or produces assembly language files for the *Jasmin* byte code assembler.

The “`dcf`” format is not yet available, but is intended to access the Gardens Point native code generators on all the platforms for which Gardens Point Modula-2 (*gpm*) implemented.

`-clsdir=`*X*	set class file tree root to directory X
`-copyright`	display the copyright notice
`-dostats`	emit timing and other statistics
`-extras`	enable experimental compiler features
`-help`	emit this usage prompt
`-hsize=`*N*	set hashtable size, with N (0 .. 65000)
`-jasmin`	create asm files and invoke Jasmin automatically
`-list`	create an output listing if there are errors (default)
`-list+`	always create an output listing
`-list-`	never create an output listing
`-noasm`	produce a symbol file, but no il
`-nocode`	create il output, but do not assemble
`-nosym`	produce no output files, not even a symbol file
`-strict`	disallow non-standard language constructs
`-special`	used for creating symbol files for foreign interfaces
`-symdir=`*X*	place symbol files in directory “X”
`-target=`*X*	emit assembler output for platform “X”
`-verbose`	chatter on about progress during compilation
`-version`	emit version information
`-warn-`	suppress warning messages from the console
`-nowarn`	same as /warn-
`-xmlerror`	errors are in *XML* format

Figure 1: *gpcp* options

Output files

Running the compiler with the `-nosym` flag causes the input files to be parsed and type-checked, but no output files are created except possibly a listing file.

If the compiler is run with the `-noasm` flag, the input files are parsed and type-checked, and a symbol file is produced for each input file. No assembly language or program executable file output is produced however.

If the compiler is run with the `-nocode` flag, the input files are parsed and type-checked, and a symbol file and *Jasmin* assembly language files are produced for each input file. No class files are produced in this case.

If the compiler is run without any flags, the input files are parsed and type-checked, and a symbol file, and a program class files are produced for each input file.

Output files with “`-target=net`” option

If the compiler is run with the `-target=net` flag, the input files are parsed and type-checked, and a symbol file and an assembly language file with extension “`*.il`” will be produced. There are additional program options available in this case. The compiler can directly produce program executable files, or Common Intermediate Language (*CIL*) assembly language files. *CIL* is always produced if the “`-nocode`” option is given.

2.5 Runtime checking

On the *Java* platform there is no facility for efficiently performing arithmetic overflow tests. On the *.NET* platform such checks are performed, but there is a very small speed gain if checks are turned off. Checks may also be turned off on a per-procedure basis, as described in Section 4.12, when it is logically necessary to do so. It is good practice to do this in the source code, even when writing for the *JVM* platform in order to ensure that source code is portable between targets.

2.6 Listing output

The compiler, by default, produces a listing file only if there are compile-time errors or warnings. It is possible to force the compiler to produce a listing, using the “`/list+`” option. Equally, it is possible to prevent the creation of a listing file even if there are errors, by using the “`/list-`” option.

The listing file contains the complete listing of the program, with four digit line numbers prepended. Errors are reported in the format shown in Figure 2

```
   1 MODULE BarMod;
   2  IMPORT FooMod;
   3  TYPE
   4   Bar* = POINTER TO ABSTRACT RECORD (FooMod.Foo)
****   ^ Only ABSTRACT basetypes can have abstract extensions
   5         i,j,k : INTEGER
   6       END;
   7 END BarMod.
```

Figure 2: Example error message

2.7 Statistics output

If the compiler is invoked with option `/dostats` then compile time statistics are produced. Figure 3 is an example, compiling the program *Browse*.

The meaning of the values written to the console is as follows.

- The compiler imports symbol files in dependency order, if necessary. The maximum recursion depth for this example turned out to be 3.
- The size of the hash-table, and the number of entries used is shown
- Import time is the time to read and process metainformation for all imports. In this example module *Browse* imports much of the compiler data structures.
- Source time is the time to read the source file into the internal buffer.
- Parse time is the time to parse the buffer, create the syntax tree and resolve all identifiers.
- Analysis time is the time to do type checking, and dataflow analysis.
- SymWrite time is the time to write out metatdata to the symbol file.

```
E:\gpcp-CLR\work> gpcp /dostats Browse.cp
#gpcp: created Browse.exe
#gpcp: <Browse> No errors
#gpcp: net version 1.2.x   of June 2004+
#gpcp: 2281 source lines
#gpcp: import recursion depth 3
#gpcp: 853 entries in hashtable of size 8209
#gpcp: import time      93mSec
#gpcp: source time     125mSec
#gpcp: parse time      157mSec
#gpcp: analysis time    31mSec
#gpcp: symWrite time     0mSec
#gpcp: asmWrite time   500mSec
#gpcp: assemble time     0mSec
#gpcp: total time      906mSec
```

Figure 3: Compile statistics example

- AsmWrite time is the time to write out the jasmin or class-file output. For this example in Figure 3 33 class files are written.
- Assemble time is the time taken to spawn a new process and run `jasmin`. Assemble time is always zero if *Jasmin* is not invoked.

2.8 Setting the hash table size

The compiler uses closed hashing internally, with a default number of identifiers of 8209 in the current version. It is possible to increase the number of entries by means of the `-hsize=`*NUMBER* option. Numbers up to 66000 are meaningful to the program.

If the hash table overflows, the compiler gives an error message, with a hint to increase the size. There is a example program with the distribution that creates a program that will break the compiler, so that users may test this feature. The compilation fails with “`-hsize=4000`”, but succeeds with the default table size.

2.9 Choosing the Output Directories

By default all output files are created in the current directory or in the “`./CP`” directory tree. This behavior may be overridden with the options `-clsdir` and `-symdir`. The symbol file is placed in the directory specified by the option `-symdir=`*target-directory*. Note carefully that if a target directory is chosen that is not on the *CPSYM* path then *gpcp* will not be able to find the symbol files automatically.

Program executable directories, and debug files in the case that debugging symbols are being created may be placed in a directory tree the root of which is specified by the `-clsdir=`*target-directory* option.

If the *.NET* target has been chosen then the `-symdir` option still applies, but `-clsdir` option does not. Instead, the binary output files may be place in a directory specified by a syntactically similar `-bindir` option.

2.10 The Make utility

The compilation process with *Component Pascal* guarantees type safety across separately compiled module boundaries. Since interface meta-information resides in the symbol files which *gpcp* creates, modules must be compiled in an order that respects the partial order induced by the global importation graph. For complex programs, this may be difficult to determine manually.

The utility *CPMake* reads symbol files, and if necessary source files, in order to determine a valid order of compilation. The syntax for invocation is —

`$> cprun CPMake` [*options*] *moduleName*

The module name may be given with or without a file-extension, but must be the name of a module which imports module *CPMain*, that is, it must be a *base module*. The module name given to *CPMake* is case sensitive.

In general, when source files of a program have been modified only a subset of the modules have to be recompiled. *CPMake* is able to work out which modules must be recompiled by checking the date stamps on the files, and also checking the module hash-keys ("magic numbers") in the symbol files. If a module has been edited, but the public interface of the module has not changed a recompilation should compute a new magic number that is the same as that expected by any previously compiled, dependent modules. In this case *CPMake* detects that the dependent modules are still consistent and do not require recompilation. This "domino-stopping" feature of the program ensures that a conservative minimum of modules are recompiled.

The options accepted by the program are exactly the options accepted by *gpcp*, except that an additional option `-all` forces compilation of **all** modules in the local directory irrespective of date stamps and magic numbers.

> Hint:
> If you use *CPMake* to bootstrap the compiler on the *.NET* platform, be aware that output file-creation will fail if the output would overwrite any file of a loaded assembly. This means that you cannot bootstrap *gpcp.NET* using an instance of the compiler from the same directory, unless you use the "`-nocode`" option and then invoke `ilasm` manually, or use the "`-bindir=directory`" option.

2.11 Module Interface Browser

The program *Browse* reads the symbol file of a module and displays the public interface. This public interface is shown in a form similar to a *Component Pascal* module. This "module" shows all the types, variables and procedures that are exported from the specified module. Only the exported fields of record types are shown. Any exported procedures are shown as procedure headers only. The output from *Browse* is not a proper *Component Pascal* module and will not compile using *gpcp*. It simply shows all of the identifiers that may be imported and used by a client module.

This program is invoked with the command —

`$> cprun Browse` [*options*] *moduleName*

The symbol file extension "`.cps`" may optionally be included in *moduleName*. As with *gpcp*, any number of files may be added in a white-space separated list. The *Browse* program sends its output to the console by default, and has the following options:

`-all`	browse this and all imported modules
`-full`	display full foreign names
`-file`	write output to the file <*moduleName*>`.bro`
`-html`	write html output to the file <*moduleName*>`.html`

The `-all` option produces output for all of the modules on the global imports graph of the specified module. The `-full` option is only meaningful for *FOREIGN* modules where the output from *Browse* will include the full external names for all procedures. The default for *Browse* is to only display the internal (*Component Pascal*) names. See Section 7 for more on Foreign Language Interfaces. The `-file` option sends the output to the file <*moduleName*>`.bro` instead of to the console. The `-html` option produces hyperlinked html text in the file <*moduleName*>`.html`. In the html output defining occurrences of identifiers are red and are anchored, while module names and external types are blue and hyperlinked. Figure 4 is the html output from the command "`Browse -html ClassMaker`".

```
MODULE ClassMaker;
IMPORT
    RTS,
    GPCPcopyright,
    Console,
    IdDesc;
 TYPE
  Assembler* = POINTER TO ABSTRACT RECORD
               END;

  Assembler* = POINTER TO ABSTRACT RECORD
                 mod* :  IdDesc.BlkId;
               END;

PROCEDURE (self:Assembler) Assemble*(),NEW,EMPTY;
PROCEDURE (self:ClassEmitter) Init*(),NEW,EMPTY;
PROCEDURE (self:ClassEmitter) Emit*(),NEW,ABSTRACT;
END ClassMaker.
```

Figure 4: Browse output from *gpcp* source file *ClassMaker.cp*

2.12 Symbol File Generator J2CPS

This program generates symbols files corresponding to *JVM* packages. Taken together with the *Browse tool*, this makes the libraries of the *Java* framework accessible to *Component Pascal* users. Usage is —

`$> java J2CPS` [*options*] *JavaPackageName*

To run *J2CPS* —

- Unpack the class files if they are in a “`jar`” archive file. The runtime system files are in an archive named “`rt.jar`” in the “`jre/lib`” directory of your *Java Development Kit*.
- Put the directory containing the class files on your path. You may need to edit the “`j2cps`” shell or batch file. Add the class file directory to the path under the “`-classfile`” option.
- You should now be able to invoke the “`j2cps`” script

3 Lexical Issues

3.1 Non-standard Keywords

In order to provide facilities for the foreign language interface there are a total of six new keywords defined. These are all upper case names and cannot be used as program identifiers.

DIV0	an additional arithmetic operator (C integer division)
REM0	an additional arithmetic operator (C integer remainder)
EVENT	used to declare multicast delegate type for *.NET* events
RESCUE	used to mark a procedure-level exception catch block
ENUM	used in dummy foreign modules in the *.NET* system
INTERFACE	used in dummy foreign modules for defining interfaces
STATIC	used to declare static features in dummy foreign modules

Only *DIV0*, *REM0*, *EVENT* and *RESCUE* may be used in normal programs, the remainder are used in dummy foreign definition modules.

The following new predefined identifiers have been added. These can be redefined, but not at the outer lexical level. Definitions for these built-in identifiers are given below.

UBYTE	an unsigned 8-bit integer type
MKSTR	function to convert a *CP* “string” to the native string type
BOX	make a dynamically allocated copy of record or array
TYPEOF	fetch the runtime type descriptor, for reflection
USHORT	convert a value to unsiged byte, with range-check
THROW	procedure that (re)throws a native exception object
APPEND	appends a new element to an extensible array (vector)
CUT	shortens an extensible array to the given length

There are some other predefined identifiers used in the extended syntax, but these are “*context sensitive markers*” and do not prevent the same names being used for program identifiers.

Warning

Remember, if you use any of these non-standard keywords or built-in identifiers, your program source will not be portable to other implementations of *Component Pascal*.

3.2 Java Package and Class Names

Fully qualified names in the Java virtual machine (*JVM*) comprise three parts.

- Package name – this defines the directory in which the class files are found. The package name may be a “dotted name”.
- Class name – the class name
- Feature name – the field or method name.

An example might be –

```
java.lang.Excecption.ToString
```

where *java.lang* is the package name, *Exception* is the class name, and *ToString* is a method name.

In this version of *gpcp*, the compiler produces one package per module, The package is the same as the module name. Thus a type-bound procedure called *isString*() bound to the type *UnaryX* in module *ExprDesc* would have the *JVM* name —

```
CP.ExprDesc.ExprName_UnaryX.isString
```

where *CP.ExprDesc* is the package name, *ExprDesc_UnaryX* is the class name, and *isString* is the method name.

Procedures and variables at the module level are declared in the *JVM* as belonging to a synthetic “class” that contains only static data and code. This *implicit static class* has the same name as the module. Thus variable “`xId`” in module *Foo* will have the somewhat boring *JVM* name —

```
CP.Foo.Foo::xId
```

Users of the compiler should almost never have to deal with explicit *JVM* names.

All aspects of the default naming scheme may be overridden, if required. Such a necessity might arise if the *Component Pascal* code must interface with a framework that has particular naming patterns hardwired in. The details of the mechanisms for overriding are given in Appendix 12.

3.3 Identifier syntax

The identifier syntax for *Component Pascal* allows arbitrary use of the underscore (lowline character). There is a further extension that is specific to the foreign language interface of gpcp.

Occasionally, names that are imported from foreign modules will happen to clash with CP reserved words. In this case, we may escape the reserve word detection by starting the identifier with the back-quote character, “‘”. Thus, if an imported module has (say) a class with a field named “`IF`”, then the field may be referenced as “`‘IF`” in the source of your program. You may not *define* identifiers using this escape mechanism, except in foreign definition modules. You may however *refer* to imported identifiers using this mechanism.

It may be important to know that the back-quote is stripped at the time that the program is scanned. The presence of the escape simply suppresses the usual check for reserved identifiers that normally follows identifier scanning. Thus the back-quote is not used during any name matching of identifiers. A curious result of this strategy is that if a program escapes an identifier that does not need it, the escaped and non-escaped identifiers will refer to the same name.

4 Semantic Issues

4.1 Class files and entry points

The compiler produces one or more class files from each module which it compiles. Classes may be dynamically loaded, or may contain an entry point with the *Java* language signature —

```
public static void main(java.lang.string[] args)
```

This entry point method takes a possibly empty array of native-strings as argument. Any command line arguments are accessed through the library *ProgArgs*.

If the source file contains the import of the special module name *CPmain*, then an class file with an entry point is produced as output. In this case the module body becomes the method “`main`”, and begins with a hidden call which saves any command line arguments so that they may be later accessed by calls to the *ProgArgs* library.

If the source file does not import *CPmain* then the module body becomes the “class constructor” which is executed at the time that class is loaded on demand.

4.2 Unimplemented constructs

There are a small number of constructs that are unimplemented or restricted in this release of the compiler. These are —

* Module finalizers (unimplemented)
* Procedure variables (not implemented on the *JVM*)
* Passing of reference parameters (inexact semantics), see sidebox page 16

All of these features were implemented in a prototype version of the compiler.

Module finalizers are intended to be run prior to unloading the module code. There is no facility for doing this on either of the *gpcp* target platforms.

Procedure variables are not permitted on the *JVM* plaform. This restriction will probably be removed in the next major release.

On the *JVM* platform argument of certain types are passed by copying rather than the semantically specified reference semantics. See the sidebox on page 16.

4.3 Additional Arithmetic Operators

The usual arithmetic operators *DIV* and *MOD* in Pascal-family languages have well defined semantics that are different to the division and remainder operators of implementations of C-family languages. In *Component Pascal* the operators *DIV* and *MOD* are defined as follows —

$$i\ \mathit{DIV}\ j = \lfloor i/j \rfloor$$

$$(i\ \mathit{DIV}\ j) \times j + (i\ \mathit{MOD}\ j) = i$$

where i, j are integers, i/j denotes real division, and $\lfloor\ .\ \rfloor$ is the *floor* function.

Notice that *DIV* always rounds toward negative infinity unlike most C-language implementations (which normally round toward zero). The Pascal operators are mathematically preferred, but in case the alternative semantics are required for compatibility reasons, *gpcp* introduces alternatives. *DIV0* denotes integer division with rounding toward zero, while *REM0* denotes the corresponding remainder operation.

$$i \mathit{DIV0}\, j = \mathit{RTZ}(i/j)$$

$$(i \mathit{DIV0}\, j) \times j + (i \mathit{MOD0}\, j) = i$$

where i, j are integers, i/j denotes real division, and *RTZ*(.) is the *Round-to-Zero* function.

Warning

Remember, if you use any of these non-standard operators your program source will not be portable to other implementations of *Component Pascal*.

4.4 Semantics of the WITH statement

The semantics of the *WITH* statement have been slightly modified so as to strengthen the guarantees on the properties of the selected variable. In the code —

```
WITH x : TypeTi DO
  ...  (* guarded region *)
| x : TypeTj DO
  ...  (* guarded region *)
END;
```

the variable x is asserted to have the specified type throughout the so-called *guarded region*. The base language guarantees that the type of the selected variable cannot be “widened” in the guarded region, but might possibly be narrowed. In *gpcp* the selected variable is treated as a constant, and neither the type nor the value can be modified either directly or indirectly. Any attempt to do so attracts a compile-time error message.

4.5 Extensible arrays: the vector types

From version 1.3 there is direct support for extensible array types. Values of these *vector* types are dynamically allocated, and automatically extend their capacity when an append operation is performed on an array that is already full. Vectors may be declared to have any element type, and extend their length using *amortized doubling*.

In most circumstances when a linked list would otherwise have been used the vector types are faster, more memory efficient, and allow memory-safe indexing. Elements of vectors may be accessed using the familiar index syntax, with index values checked against the *active length* of the array, rather than the array *capacity*.

Declaring vector types

Vectors are declared using the new syntax —

Type ::- ... - - *other type constructors*
| “`VECTOR`” “`OF`” *Type*.

Variables of vector type are not automatically allocated. They must be explicitly allocated using a variant of the built-in *NEW* procedure which specifies the initial capacity. Here is an example –

```
TYPE IntVec = VECTOR OF INTEGER;
VAR  iVec   : IntVec;
  ...
NEW(iVec, 16); (* Allocate vector with initial capacity 16 *)
```

Built-in procedures

There are two new procedures defined on the vector types. The first of these appends a new value of the declared element type to an existing vector. The signature of the procedure is —

```
PROCEDURE APPEND(v : VectorOfEType, e : EType);
```

As noted above, vectors are reference types, so that the first argument may be passed by value. The vector will double its length if there is no further space left in the array.

There is another built-in procedure which allows for the *active length* of the vector to be reduced. This has the effect of truncating the array at the given length. The signature is —

```
PROCEDURE CUT(v : VectorOfEType, i : INTEGER);
```

It is a runtime error if the requested new length of the vector is less than zero, or is greater than the current active length.

A new version of the standard built-in function *LEN* returns the active length of the vector. There is no way of querying the current capacity of a vector datum.

As noted above, a new version of the standard built-in procedure *NEW* allocates vectors of the specified initial capacity.

Assignment semantics

Vector values are references, so that an assignment of a vector value creates an alias to the original r-value. If you really do have to make a value copy, here is a coding pattern —

```
VAR  a,b : SomeVecType;
  ...
NEW(b, LEN(a)); (* b is barely big enough *)
FOR i := 0 TO LEN(a)-1 DO APPEND(b, a[i]) END;
```

Note that in this case the value copy b will extend at the very next append operation, since its initial *capacity* is the same as the *active length* of a. The active length of a may have been as little as one half of its capacity.

4.6 Implementing foreign interfaces

Component Pascal types may extend classes from the underlying execution platform. Types which extend *JVM* or *.NET CLS* classes may also declare that they implement interfaces[3] from the *CLS*. The syntax extension to access this feature has *BNF* —

RecordDecl ::- “`RECORD`” [*BaseType*] [*Fields*] “`END`” “`;`” .
BaseType ::- “`[`” *QualifiedIdent* { “`+`” *QualifiedIdent* } “`]`” .

---

[3] By “interface” in this context, we mean *fully abstract class*.

The first qualified identifier, as in the Report, is the class that is extended by the type being defined. Any additional qualified identifiers are the names of interfaces that the type promises to implement. The compiler checks that this contract is honored. In the case that interfaces are implemented, the base type may be left blank, or may be explicitly set to *ANYREC*.

The semantics of type-assertions are also relaxed whenever a reference is asserted to be of some interface type. For non-interface types many erroneous type-checks can be detected at compile time. However, there are almost no cases where an assertion that a dynamically typed object belongs to some interface type can be rejected at compile time.

Thus, interface types may be *used* in *Component Pascal*. However, it is not possible to *define* interface types using *gpcp*.

4.7 Unsigned byte type on .NET platform

The 8-bit type used in the *.NET* Common Language Specification (*CLS*) is an unsigned type. If *Component Pascal* is to be a full consumer of *CLS* libraries then it must be possible to declare variables and fields of such types in *Component Pascal* programs. In order to facilitate this a new built-in type *UBYTE* has been introduced in version 1.2 of *gpcp*. Values of this type may be assigned to variables of larger integral types as required. However, if values of this type are assigned to locations of the signed 8-bit type *BYTE* a runtime range-check is required. Similarly if values of any signed type are assigned to a location of unsigned byte type an explicit narrowing cast is required, using the new built-in function *USHORT*().

4.8 Runtime type descriptors

A new function since version 1.2 returns runtime type descriptors. This allows easy access to the facilities of the *system reflection* libraries. The function is overloaded, and has the following signatures —

```
PROCEDURE TYPEOF(typename): RTS.NativeType;
PROCEDURE TYPEOF(IN s : anytype) : RTS.NativeType;
```

If the target is *.NET*, then *NativeType* is an alias for *System.Type* on the underlying runtime. If the target is the *JVM*, then the return value type will be *java.lang.Class*.

The procedure with the first signature takes any type name as actual parameter. The procedure with the second signature takes an actual parameter that is any variable designator. If the type of the designator is statically known (perhaps because it denotes an object of an inextensible type) then the compiler resolves the reference and no call is needed to the runtime function `java.lang.Object.getClass()`.

4.9 Additional built-in functions

There are four additional built-in functions added to the implementation. One allows convenient access to the underlying native string object type. The signature is —

```
PROCEDURE MKSTR(IN s : ARRAY OF CHAR) : RTS.NativeString;
```

Note that it is never necessary to use MKSTR when passing a *literal* string to a formal parameter of native string type. In the literal case the compiler does the conversion for the programmer automatically.

Another handy function takes a record or array type, and makes a value copy onto the heap, returning a pointer to the copy. The signature is —

```
PROCEDURE BOX(s : CP-type) : POINTER TO CP-type;
```

Here, *CP-type* is a *Component Pascal*-defined record, array or string type. The function copies the value so that modification of the boxed value does not affect the original value. The function is particularly convenient for programs that manipulate character data implemented as dynamically allocated arrays. Thus “`BOX("hello")`” returns a pointer to an array of characters of length 6, while “`BOX(ptr1^ + ptr2^)`” performs a string concatenation and allocates a destination array of the required length. If the function is applied to an array of fixed length the return value is an open array of the same length. In the case of character arrays the use of the array “stringifier” mark “`$`” on the argument of *BOX* boxes a copy of the array which is truncated at the position of the “`nul`” character. Here is an example program fragment —

```
VAR str : ARRAY 16 OF CHAR;
    ptr : POINTER TO ARRAY OF CHAR;
  ...
str := "Hello";
ptr := BOX(str); (* ptr points to an array of length 16 *)
ptr := BOX(str$);(* ptr points to an array of length 6 *)
```

Without the *BOX* function, the construction of a value copy of an open array would require the following tedious construction —

```
VAR a,b : POINTER TO ARRAY OF CHAR;
  ...
NEW(b, LEN(a));
FOR i := 0 TO LEN(a) DO b[i] := a[i] END;
```

Using the *BOX* function, the same effect is achieved by “`b := BOX(a^);`”.

As of version 1.2 a new built-in unsigned byte type has been introduced, for conformance with the *.NET CLS*. In order to coerce values of signed type to the new type a new function *USHORT*(), analogous to the standard *SHORT*() function is also introduced. This function has the signature —

```
PROCEDURE USHORT(s : AnyNumericType) :  UBYTE;
```

It is a runtime error if the value of the parameter is not within the unsigned byte range.

The fourth new built-in function, *TYPEOF*, allows programs to access the reflection facilities of the underlying platform. The function was described in the previous section.

4.10 Deprecated features and warnings

The use of procedure variables and of super-calls are deprecated. Both attract compile-time warning messages. Warnings are also issued in the case of procedures that are not exported, and are not called (or assigned as procedure variables) within their defining module. This situation is usually an error arising from failure to mark the procedure for export.

4.11 Program executable verification

Component Pascal is a type-safe language. Every correct program is type-safe in the same sense that is guaranteed by the *.NET* virtual object system’s verifier. In principle therefore, all output of *gpcp* should be verifiable.

You may force the *Java* runtime to invoke the verifier by running programs using the —

```
java -verify ...
```

option, together with any other options required for the program.

Output might fail to verify if a manually constructed interface to a library does not correspond to the internal metadata of the imported assembly. This potential problem has largely gone away with the use of *J2CPS*.

4.12 Unchecked arithmetic

The *JVM* version of *Component Pascal* does not perform overflow-checking, but this is the default on the *.NET* target. If you wish to write code that is portable between the versions, you should explicitly turn off overflow checking for those procedures that require this for semantic correctness. Overflow checking is turned off on a per-procedure basis using a custom attribute.

The syntax of the custom attribute is a context sensitive marker that appears immediately after the keyword *BEGIN* in a procedure or module body. The syntax is —

Body ::- “`BEGIN`” [“`[UNCHECKED_ARITHMETIC]`”]
StatementSequence “`END`” *identifier* .

An example of the use of this construct, from the source of the compiler itself, is the identifier hash function shown in Figure 5. This function performs a rotate-and-add

```
PROCEDURE hashStr(IN str : ARRAY OF CHAR) : INTEGER;
  VAR tot : INTEGER;
      idx : INTEGER;
      len : INTEGER;
BEGIN [UNCHECKED_ARITHMETIC] (* Turn off overflow checks *)
  len := LEN(str$);
  tot := 0;
  FOR idx := 0 TO len-1 DO
    INC(tot, tot);
    IF tot < 0 THEN INC(tot) END;
    INC(tot, ORD(str[idx]));
  END;
  RETURN tot MOD size;
END hashStr;
```

Figure 5: Code of the hash function

computation, in which bits are carried out of the sign bit back into the least significant bit of the variable “`tot`”. Overflow checking must be turned off, in order to prevent very long identifiers from crashing the compiler.

5 Exception Handling

Component Pascal does not define exception handling, but it is necessary to deal with foreign libraries that may throw exceptions. There is one new keyword and one new built-in procedure introduced to facilitate this.

Important note on parameter passing semantics for the *JVM*

The *JVM* version of *gpcp* takes liberties with the precise semantics of parameter passing almost everywhere. Actual parameters of unboxed[a] value type that are passed to reference formals are passed by copying. In the case of formal parameters of *VAR* mode, actual values of unboxed value type are copied in **and** copied out. In the case of formal parameters of *OUT* mode the value is only copied out. The current implementation method is necessary in order to obtain reasonable performance on the *JVM*. The change will not affect the results of your program unless you access the actual of a reference formal along two paths (either by having two reference formals sharing the same actual argument value, or accessing a static variable directly and through a parameter). You should not write programs that do this! You might also care to know that with this change, the performance of code is good if you have only one such copied parameter, but becomes poor if you have more than one in any frequently called procedure.

In contrast, on the *.NET* platform unboxed reference parameters are only passed inexactly if they are non-locally accessed from within a nested procedure.

---

[a] Unboxed value types on the *JVM* platform are the built-in standard types such as *CHAR* and *INTEGER*, together with the pointer types. Structures and arrays are always boxed at runtime in the *JVM*, and are not affected by this semantic inexactness.

5.1 The RESCUE clause

Procedures, but not modules may include exactly one *RESCUE* clause, at the end of the procedure body. This has syntax —

ProcBody ::- “`BEGIN`” *Statements*
[“`RESCUE`” “`(`” *ident* “`)`” *Statements*]
“`END`” *ident*.

The identifier introduced in the parentheses is of type *RTS.NativeException*, and must have a name that is distinct from every other identifier in the local scope.

If any exception is thrown in the body of the procedure, or if any exception is unhandled in a procedure called from this procedure, then the rescue clause is entered with the exception object in the named local variable. This variable is read-only within the rescue clause, and is not known in the rest of the procedure body.

If the program has imported or defined any extensions of the native exception type, filtering may be performed by using the usual type-test syntaxes. The compiler will check that the rescue clause fulfills any contracts implied by the procedure signature. For example, in the case of function procedures the rescue clause must explicitly return a type-correct value, or explicitly throw another exception.

5.2 The THROW statement

Code may throw an exception by using the built-in procedure THROW. This procedure has two signatures —

```
PROCEDURE THROW(x : RTS.NativeException);
PROCEDURE THROW(x : RTS.NativeString);
```

These may be used anywhere in the program. The first is useful for rethrowing an exception from within a rescue clause. The second of these may be passed a literal string, without requiring a call of *MKSTR*() since the the compiler will automatically coerce literal strings to formals of native string type. This call will throw an exception object of *System.Exception* type, with the given string as embedded information. If you want to create an exception object to abort program execution with a meaningful string, you may also use the library function

```
RTS.Throw(msg : ARRAY OF CHAR);
```

Exceptions thrown by this library function can be caught by a *RESCUE* clause.

> **Warning**
> Remember, if you use any of these non-standard facilties for exception handling your program source will not be portable to other implementations of *Component Pascal*.

6 Facilities of the CP Runtime System

6.1 Supplied libraries

This release has a small number of libraries supplied. These are —

- *Console* writes strings and numbers to the console
- *StdIn* reads characters and whole lines from the console
- *Error* this library writes strings and number to the error stream
- *ProgArgs* provides access to the command line arguments, if any
- *GPText* a basic library for handling text formatting
- *GPFiles* defines the supertype of *GPBinfFiles.FILE* and *GPTextFiles.FILE*
- *GPBinFiles* reading and writing binary files
- *GPTextFiles* reading and writing text files
- *RealStr* formatting real numbers: based on the *ISO-Modula-2* library
- *RTS* access to the facilities of the runtime system
- *StringLib* string library, based on the *ISO-Modula-2* library
- *SYSTEM* some unsafe, low-level facilites.

For the most part these libraries are the ones that were required to bootstrap the compiler. More will come later.

6.2 The runtime system (RTS)

The runtime system provides a variety of low-level access facilities. The source file for this module, “`RTS.cp`”, is not really the source. This file is a dummy, as is denoted by the context-sensitive mark *SYSTEM* appearing before the keyword *MODULE*. All such “modules” are actually implemented in the *C#* file named “`RTS.cs`”, and at runtime are found in the assembly “`RTS.dll`”.

The “source” of *RTS* is shown in Figure 6. The four character *defaultTarget* string will hold “`net`” when running on the *.NET* platform, and “`jvm`” when running under the Java Runtime Environment. The word *SYSTEM* in the first line of the definition is a context sensitive mark, rather than a reserved word. This means that the word may be used as an identifier elsewhere in the program. The mark simply indicates that the resources of this module are actually found in the assembly “`RTS.dll`”. *Console*, *Error* and *ProgArgs* are also *SYSTEM* modules.

6.3 The ProgArgs library

The *ProgArgs* library provides access to the command line argument, if any. From *gpcp* release 1.3 it also provides access to the process environment. This is a system library, with the following public interface —

```
SYSTEM MODULE ProgArgs;
  PROCEDURE ArgNumber*() : INTEGER;
  PROCEDURE GetArg*(num : INTEGER; OUT arg : ARRAY OF CHAR);
  PROCEDURE GetEnvVar*(IN str : ARRAY OF CHAR;
                      OUT val : ARRAY OF CHAR);
END ProgArgs.
```

Note carefully that on the *.NET* platform *GetEnvVar* fetches an environment variable, or an empty string. On the *JVM* platform the use of environment variables is deprecated, and the procedure fetches the corresponding *Property String*. Such property strings are passed to the underlying *Java* process at startup, using options of the form —

`-D`*name*=*value*

6.4 The RealStr library

The RealStr library is a port to *Component Pascal* of the *ISO-Modula-2* real number formatting library. The interface to the library is shown in Figure 8.

The library contains procedures to transform real number values into fixed format strings, floating format strings and the so-called “engineering” format in which exponents are always a multiple of three. For the string parser, *StrToReal*, the recognized format is given by the regular expression —

Number ::- [“`+`” | “`-`”] *dig* {*dig*} [“`.`” {*dig*}] [“`E`” [“`+`” | “`-`”] *dig* {*dig*}] .

where *dig* denotes a decimal digit.

The *RealStr* library will exactly round trip numbers via *RealToFloat* and *StrToReal*, provided a full 17 significant figures are specified for *RealToFloat*. So far as possible the results of using module *RealStr* should be identical on the two platforms.

```
SYSTEM MODULE RTS;
  VAR defaultTarget- : ARRAY 4 OF CHAR;

  TYPE  CharOpen* = POINTER TO ARRAY OF CHAR;

  TYPE  NativeType*      = POINTER TO RECORD END;
        NativeObject*    = POINTER TO RECORD END;
        NativeString*    = POINTER TO RECORD END;
        NativeException* = POINTER TO RECORD END;

  PROCEDURE getStr(x : NativeException) : CharOpen;
  (* Get error message from Exception x *)

  PROCEDURE StrToReal*(IN  s  : ARRAY OF CHAR;
                       OUT r  : REAL;
                       OUT ok : BOOLEAN);
  (* Parse array into an IEEE double REAL *)

  PROCEDURE StrToInt*(IN  s  : ARRAY OF CHAR;
                      OUT i  : INTEGER;
                      OUT ok : BOOLEAN);
  (* Parse an array into a CP INTEGER *)

  PROCEDURE StrToLong*(IN  s  : ARRAY OF CHAR;
                       OUT i  : LONGINT;
                       OUT ok : BOOLEAN);
  (* Parse an array into a CP LONGINT *)

  PROCEDURE RealToStr*(r : REAL;
                   OUT s : ARRAY OF CHAR);
  (* Decode a CP REAL into an array *)

  PROCEDURE IntToStr*(i : INTEGER;
                  OUT s : ARRAY OF CHAR);
  (* Decode a CP INTEGER into an array *)

  PROCEDURE LongToStr*(i : LONGINT;
         OUT s : ARRAY OF CHAR);
  (* Decode a CP INTEGER into an array *)

  PROCEDURE realToLongBits*(r : REAL) : LONGINT;
  (* Convert IEEE double to longint with same bit pattern *)

  PROCEDURE longBitsToReal*(l : LONGINT) : REAL;
  (* Convert IEEE double to a longint with same bit pattern *)
                                                        RTS continues ...
```

Figure 6: Source of the *RTS* pseudo-module

```
RTS continuation ...
  PROCEDURE hiInt*(l : LONGINT) : INTEGER;
  (* Get hi-significant word of long integer *)

  PROCEDURE loInt*(l : LONGINT) : INTEGER;
  (* Get lo-significant word of long integer *)

  PROCEDURE Throw*(IN s : ARRAY OF CHAR);(* Abort execution *)

  PROCEDURE GetMillis*() : LONGINT;(* Get time in milliseconds *)

  PROCEDURE ClassMarker*(o : ANYPTR);(* Write class name *)

  PROCEDURE GetDateString*(OUT str : ARRAY OF CHAR);
  (* Get a date string in some native format *)
END RTS.
```

Figure 7: Source of the *RTS* pseudo-module, continued

6.5 The StringLib library

The *StringLib* library reproduces the functionality of the *ISO Modula-2* string library, although the implementation has little similarity. The publicly accessible interface to the library is shown in Figure 9.

The library contains the expected procedures for assigning, extracting, replacing, deleting, concatenating and searching strings. As well, each of the procedures that mutates a string value has a corresponding predicate function that tests if the operation can be carried out exactly. This allows a guarded style of coding.

None of these routines raises program exceptions, but have sensible behaviour in the case that the incoming arguments do not allow correct completion. For example, in the case of the *Assign* procedure, if the source string is too long for the supplied destination the result is truncated to fit. Similarly, for the *Extract* procedure the length of the extracted string is the least of: (i) the requested character count, (ii) the number of characters left in the source string, and (iii) the capacity of the destination array.

6.6 The SYSTEM facilities

The *SYSTEM* module consists of three procedures. It must be explicitly imported, and programs that import it will only compile if the command line argument “`/unsafe`” is in effect and the target is *.NET*. Programs which use any of these facilities will be unverifiable. Furthermore, the careless use of these facilities may compromise the correctness of the garbage collector. The module is useful for diagnostic testing, but should never be used in deployed code.

The procedures are —

```
PROCEDURE ADR(IN obj : any type) : INTEGER;
PROCEDURE GET(IN adr : INTEGER; OUT dst : any basic type);
PROCEDURE PUT(IN adr : INTEGER; IN val : any basic type);
```

```
MODULE RealStr;

(* Ignores any leading spaces in str. If the subsequent characters in str are in the     *)
(* format of a signed real number, assigns a corresponding value to real. Argument      *)
(* res reports whether conversion was successful.                                       *)
  PROCEDURE StrToReal*(str      : ARRAY OF CHAR;
                       OUT real : REAL;
                       OUT res  : BOOLEAN);

(* Converts the value of real to floating-point string form, with sigFigs significant   *)
(* digits and copies the possibly truncated result to str.                              *)
  PROCEDURE RealToFloat*(real    : REAL;
                         sigFigs : INTEGER;
                         OUT str : ARRAY OF CHAR);

(* Converts the value of real to floating-point string form, with sigFigs significant   *)
(* digits, and copies the possibly truncated result to str. The number is scaled with one *)
(* to three whole-number digits and an exponent that is a multiple of three.            *)
  PROCEDURE RealToEng*(real    : REAL;
                       sigFigs : INTEGER;
                       OUT str : ARRAY OF CHAR);

(* Converts the value of real to fixed-point string form, rounded to the given place    *)
(* relative to the decimal point, and copies the result to str.                         *)
  PROCEDURE RealToFixed*(real  : REAL;
                         place : INTEGER;(* num. of frac. places *)
                     OUT str   : ARRAY OF CHAR);

(* Converts the value of real as RealToFixed if the sign and magnitude can be shown     *)
(* within the capacity of str, or otherwise as RealToFloat, and copies the possibly     *)
(* truncated result to str. The format is implementation-defined.                       *)
  PROCEDURE RealToStr*(real: REAL; OUT str: ARRAY OF CHAR);
END RealStr.
```

Figure 8: Interface of the RealStr library

There is a demonstration program named `\examples\hello\testadr.cp` This example demonstrates some of the capabilities of the library. Study the results, you may find them surprising. Note, for example, that *ADR*(*arr*) is not equal to *ADR*(*arr*[0]).

6.7 The StdIn library

In version 1.3 a new library is supplied that provides primitives for reading single characters and whole lines from the standard input stream. This stream is connected by default to the machine console, but may be redirected using the facilities of the underlying platform libraries.

This library has very simple functionality, described by the foreign module shown in Figure 11. In the first release the predicate function *More* always returns the *TRUE* value. The team will restore the functionality when we figure out a way of making the behaviour the same on the two execution platforms.

```
MODULE StringLib;(* from GPM module StdStrings.mod *)

  PROCEDURE CanAssignAll*(sLen : INTEGER;
                       IN dest : ARRAY OF CHAR) : BOOLEAN;
(* Check if an assignment is possible without truncation.                   *)

  PROCEDURE Assign*   (IN  src : ARRAY OF CHAR;
                       OUT dst : ARRAY OF CHAR);
(* Assign as much as possible of src to dst, with terminating nul           *)

  PROCEDURE CanExtractAll*(len : INTEGER;
                           sIx : INTEGER;
                           num : INTEGER;
                       OUT dst : ARRAY OF CHAR) : BOOLEAN;
(* Check if extraction of "num" chars starting at index sIx is possible.    *)

  PROCEDURE Extract*  (IN  src : ARRAY OF CHAR;
                           sIx : INTEGER;
                           num : INTEGER;
                       OUT dst : ARRAY OF CHAR);
(* Extract num characters starting from sIx. Result is truncated if there   *)
(* are fewer characters left, or the destination is too short.              *)

  PROCEDURE CanDeleteAll*(len,sIx,num : INTEGER) : BOOLEAN;
(* Check if num chars may be deleted starting from sIx. len is the source length *)

  PROCEDURE Delete*(VAR str : ARRAY OF CHAR;
                        sIx : INTEGER;
                        num : INTEGER);
(* Delete num chars starting from sIx. Less are deleted if there are less num after sIx. *)

  PROCEDURE CanInsertAll*(sLen : INTEGER;
                          sIdx : INTEGER;
                      VAR dest : ARRAY OF CHAR) : BOOLEAN;
(* Check if sLen chars may be inserted into dest starting from sIdx.        *)

  PROCEDURE Insert*   (IN  src : ARRAY OF CHAR;
                           sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR);
(* Insert src string into dst starting from sIx. Less chars are inserted if there is *)
(* insufficient space in dst. dst is unchanged if sIx is beyond the end of dst. *)

  PROCEDURE CanReplaceAll*(len : INTEGER;
                           sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR) : BOOLEAN;
(* Check if len chars may be replaced in dst starting from sIx.             *)
                                                      StringLib continues ...
```

Figure 9: Interface to the *StringLib* library

StringLib continuation ...

```
  PROCEDURE Replace*  (IN  src : ARRAY OF CHAR;
                           sIx : INTEGER;
                       VAR dst : ARRAY OF CHAR);
(* Insert the characters of src into dst starting from sIx. Less chars are replaced if the *)
(* initial length of dst is insufficient. The string length of dst is unchanged. *)

  PROCEDURE CanAppendAll*(len : INTEGER;
                      VAR dst : ARRAY OF CHAR) : BOOLEAN;
(* Check if len characters may be appended to dst *)

  PROCEDURE Append*(src : ARRAY OF CHAR;
                VAR dst : ARRAY OF CHAR);
(* Append the chars of src string onto dst. Less characters are appended if the *)
(* length of the destination string is insufficient. *)

  PROCEDURE Capitalize*(VAR str : ARRAY OF CHAR);

  PROCEDURE FindNext*    (IN  pat : ARRAY OF CHAR;
                          IN  str : ARRAY OF CHAR;
                              bIx : INTEGER;(* Begin index *)
                          OUT fnd : BOOLEAN;
                          OUT pos : INTEGER);
(* Find the first occurrence of the pattern pat in str starting the search from bIx *)
(* If no match is found fnd is false and pos is bIx. Empty patterns match everywhere. *)

  PROCEDURE FindPrev*(IN  pat : ARRAY OF CHAR;
                      IN  str : ARRAY OF CHAR;
                          bIx : INTEGER;(* Begin index *)
                      OUT fnd : BOOLEAN;
                      OUT pos : INTEGER);
(* Find the previous occurrence of the pattern pat in str starting the search from bIx. *)
(* If no match is found fnd is false and pos is bIx. Empty patterns match everywhere. *)

  PROCEDURE FindDiff*   (IN  str1 : ARRAY OF CHAR;
                         IN  str2 : ARRAY OF CHAR;
                         OUT diff : BOOLEAN;
                         OUT dPos : INTEGER);
(* Find the index of the first char of difference between the two input strings. *)
(* If the strings are identical diff is false, and dPos is zero. *)

END StringLib.
```

Figure 10: Interface to the *StringLib* library

7 Foreign Language Interface

7.1 Accessing the underlying native types

As seen in Figure 6 the *RTS* module defines four type aliases. The binding of these types to the native platform types is determined dynamically, at compile time. Thus, the

```
SYSTEM MODULE StdIn;
  (* Read a line of text, discarding new-line *)
  PROCEDURE ReadLn*(OUT arr : ARRAY OF CHAR);
  PROCEDURE SkipLn*();(* Discard remainer of line *)
  PROCEDURE Read*(OUT ch : CHAR);(* Fetch next character *)
  PROCEDURE More*() : BOOLEAN;(* Return TRUE in gpcp v1.3! *)
END StdIn.
```

Figure 11: Source of the *StdIn* pseudo-module

underlying types are accessible without any other import other than *RTS*. At compiler-runtime the compiler queries the target flag, or takes the default target value if there is no target command option.

If the target is "`net`" then *NativeObject*, *NativeString* and *NativeException* will be the *CLR* types *System.Object*, *System.String* and *System.Exception* respectively.

If the target is "`jvm`" then *NativeObject*, *NativeString* and *NativeException* will be the *Java* types *java.lang.Object*, *java.lang.String* and *java.lang.Exception* respectively.

In any case, literal strings may be implicitly coerced to either the native string type, or to the native object type. This saves a lot of clutter in code that interfaces to foreign libraries. However, if the value of a charater array *variable* needs to be transformed to a native string, the non-standard built-in function —

```
PROCEDURE MKSTR(IN s : ARRAY OF CHAR) : RTS.NativeString;
```

must be used. See the appendix for an extended example of using these facilities for working with native string types.

7.2 Compiling dummy definition modules

As a convenience during bootstrapping, the compiler has been enhanced so as to allow the construction of metainformation files for foreign language libraries. Such modules must be compiled with the "`-special`" option.

Foreign language interfaces are denoted by the context sensitive marks *FOREIGN* or *SYSTEM* preceding the keyword *MODULE* at the start of the file. Such "dummy" modules do not contain the code of the foreign language facilities, but simply define the interface to those facilities. Such modules must be compiled with the "`-special`" option. The system marker has special meaning in the *.NET* platform, but has the same semantics as foreign in the *JVM* platform.

When a dummy definition module is compiled there are a small number of syntactic extensions and changes.

- Modules can be given an explicit external name
- Procedures can be given an explicit external name
- Features with "protected" scope may be defined
- Static features of classes may be defined
- Escaped identifiers may be defined
- Interface types may be defined

`*` Overloaded names may be given aliases

`*` Constructors may be given an alias

A module declaration of the form —

```
MODULE Foo["PackageName"];
```

declares that this module will be found in *JVM* assembly "`CP`.*PackageName*" where the it PackageName is a possibly dotted name of the form "`a.b ...`" or "`a/b ...`". It is not necessary to use this mechanism if you write the foreign module so that it has the default name as described in Section 3.2.

A procedure declaration of the form —

```
PROCEDURE (x : T)BarII*["Bar"](i,j : INTEGER);
```

declares that this type-bound procedure has the external name "`Bar`" and the internal (CP) name "`BarII`". This mechanism allows overloaded names in the *CLS* to be given non-overloaded aliases in CP.

The mark "`!`" is used to declare that a foreign name has protected scope. The mark is placed in the same position in a declaration as the standard export markers "`*`" and "`-`".

If a name clashes with a *Component Pascal* keyword, it should be defined using the back-quote escape, as described on page 9.

Here is an example of the syntax that is required to define a foreign interface type.

```
TYPE Foo* = POINTER TO INTERFACE RECORD (* always empty *) END;
```

The keyword *INTERFACE* is reserved. Such types cannot declare any instance fields in the record, nor can they define type-bound procedures which are not declared *ABSTRACT*.

Finally, constructors must be declared with the special name "`<init>`". Declaring a constructor is not necessary if only the no-arg constructor is required, since *NEW*(*obj*) works in this case as for all other types in *Component Pascal* (see Section 8.4 for more detail). If access to constructors with arguments is required, then these may be given a *Component Pascal* alias, and are marked as constructors by using the magic explicit name. For the "`-target=net`" version, the magic name is "`.ctor`".

7.3 Accessing Static Features of Foreign Classes

If a class has been imported from a foreign definition, and the class has static members, these may be accessed by means of a semantic extension to the designator grammar.

Normally, the syntactic construct —

QualifiedIdent {*Selector*}

is in error if the qualified identifier resolves to a type-identifier. However there are two exceptional cases where this is legal in *gpcp*. If a designator begins —

TypeIdentifier "`.`" *Identifier ...*

and the following is true —

The type identifier resolves to an imported, foreign type, **and either**
  the identifier is a static field or constant of the type, **or**
  the identifier is a static method of the named type

then this is a legal reference to the named static feature of the type.

In order to define such constructs in the syntax of dummy definitions the following productions are added to the record syntax. Note that these extensions are only recognised if the module is compiled with the "`-special`" command-line option.

Record ::- “`RECORD`” [“`(`” *TypeId* “`)`”] {*FieldList*}
[“`STATIC`” {*StatFeature*}] “`END`”.
StatFeature ::- *ProcHeading* | *StatConst* | *StatField* .
StatConst ::- *identifier* “=” *ConstExpression* .
StatField ::- *identifier* “`:`” *TypeId* .

All undefined syntactic categories in the fragment have the same meaning as in the unmodified *Component Pascal* syntax. In particular, procedure headings have the same syntax as elsewhere in the language.

8 Creating and Using Foreign Definition Modules

This Section is only of relevance if you plan to write your own foreign definition modules. For most users the information in the previous section on the usage of these facilities will be sufficient.

Hint:
This section is included for mainly historical reasons. The need to write foreign definition modules has significantly decreased with the availablity of the *N2CPS* and *J2CPS* tools. It is usually easier to write the foreign language code, use the tool to produce the symbol file, and *Browse* to produce a human-readable version.
An exception occurs when the same module is required for both platforms. In that case it may still be simpler to write a foreign module, and then separately implement the code in *Java* and *C#* to match the shared definition.

8.1 Syntax of Foreign Definitions

The syntax of foreign definition is shown in Figure 12. Unless otherwise defined here, the meanings of syntactic-category symbols is the same as in the Component Pascal Report.

The syntax begins with the context sensitive mark *FOREIGN* or *SYSTEM*. On the *.NET* platform the system marker indicates that the code will be found in the runtime system assembly. In the *JVM*, where each class file contains a single class, the marker has the same semantic effect as the foreign marker.

8.2 Explicit package or namespace names

The way in which runtime names are generated from module names was described in Section 3.2. In the case of the *JVM* we have the following correspondence —

```
GPModule       ::-  Module | ForeignMod .
ForeignMod     ::-  ( "FOREIGN" | "SYSTEM" ) "MODULE" ident [ string ] ";"
                    ImportList DeclSeq "END" ident "." .
DeclSeq        ::-  {   "CONST" {ConstDecl ";"}
                      | "TYPE" {TypeDecl ";"}
                      | "VAR" {VarDecl ";"}}
                    { ProcHeading ";" | MethodHeading ";" }
ProcHeading    ::-  "PROCEDURE" IdentDef ["[" string "]"] [FormalPars] .
MethodHeading  ::-  "PROCEDURE" Receiver IdentDef ["[" string "]"]
                    [FormalPars] ["," "NEW"]
                    ["," ("ABSTRACT" | "EMPTY" | "EXTENSIBLE")] .
TypeDecl       ::-  IdentDef "=" Type .
Type           ::-  ["POINTER" "TO"] [Attributes] "RECORD" [Supers]
                    FieldList {";" FieldList }
                    [ "STATIC" StaticDecl {";" StaticDecl} ] "END"
                 |  - - Other types as in the Report .
StaticDecl     ::-  IdList ":" Type | IdentDef "=" ConstExpr | ProcHeading .
Attributes     ::-  "ABSTRACT" | "EXTENSIBLE" | "INTERFACE" .
Supers         ::-  "(" [Qualident] {"+"Qualident}")" .
```

Figure 12: Syntax of foreign modules

Component Pascal Name	**JVM Name**
`MODULE ModNm;`	`CP.ModNm` *// package name*
`TYPE Cls = RECORD...END;`	`CP.ModNm.ModNm_Cls`
`VAR varNm : Cls;`	`CP.ModNm.ModNm.varNm`
`PROCEDURE ProcNm();`	`CP.ModNm.ModNm.ProcNm()`
`PROCEDURE (t:Cls)MthNm();`	`CP.ModNm.Cls.MthNm()`
`END ModNm.`	

Notice that in the JVM there are no features that are defined outside of classes, so that the static features *varNm* and *ProcNm* are considered at runtime to belong to an implicit static class with the same name as the module name. However, so far as an importing *Component Pascal* program is concerned, these features will be accessed by the familiar *ModuleName.memberName* syntax.

Component Pascal Name	**.NET CLS Name**
`MODULE ModNm;`	`[ModNm]ModNm` *// namespace name*
`TYPE Cls = RECORD...END;`	`[ModNm]ModNm.Cls`
`VAR varNm :  Cls;`	`[ModNm]ModNm.ModNm::varNm`
`PROCEDURE ProcNm();`	`[ModNm]ModNm.ModNm::ProcNm()`
`PROCEDURE (t:Cls)MthNm();`	`[ModNm]ModNm.Cls::MthNm()`
`END ModNm.`	

In the virtual object system of *.NET* the situation is similar, with an implicit static class being defined with the same name as the module.

If, as a user, you are writing a foreign definition and plan to implement the library yourself in either *Java* or in *C#* (say), then you may define the foreign module in this way and write the foreign code so as to match the default "name mangling" scheme. In this case you may even use the same foreign definition for both versions of *gpcp*, and implement a foreign module on each underlying platform. If on the other hand you are planning to match a foreign definition to an existing library written in *Java* or *C#*, then you must override this default naming scheme.

The syntax —

“`FOREIGN`” “`MODULE`” *ident* “`[`” *string* “`]`” “`;`”

allows an arbitrary package or namespace name to be defined. For example, in order to access the facilities of the package `java.lang.Reflect` a foreign module might begin

```
FOREIGN MODULE java_lang_Reflect["java.lang.Reflect"];
```

Similarly, in order to access the facilities of the namespace *System.Reflect* in the assembly *mscorlib* a foreign module might begin

```
FOREIGN MODULE mscorlib_System_Reflect
                             ["[mscorlib]System.Reflect"];
```

Note that the form of the literal string is different on the two platforms, and thus any such foreign modules will be specific to a particular platform. Notice also that there is no mechanism to explicitly give a name to an implicit static class.

8.3 Dealing with overloaded names

Each of the underlying platforms allows name overloading for methods. This feature is deliberately not permitted in *Component Pascal*. Nevertheless, it is necessary to gain access to library methods that have overloaded names. The option of using explicit external method names facilitates this. Suppose we have two methods, both of which are named *Add*`()`, one with a single integer parameter, and the other with two. We might define these as follows in a foreign definition.

```
PROCEDURE (this : Cls)AddI*["Add"](I : INTEGER),NEW;
PROCEDURE (this : Cls)AddII*["Add"](I,J : INTEGER),NEW;
```

Within the importing *Component Pascal* program the two names are distinct, but the program executable will correctly refer to the underlying overloaded methods. This manually specified name-mangling is rather awkward, particularly in the case of parameters of object types.

Since *gpcp* release 1.1 users are able to access the unmangled names of overloaded foreign methods directly. The *N2CPS* and *J2CPS* tools create symbol files that have overloaded names, and the compiler will match calls to the intended method. Because this is a language extension, the compiler is strict about matching calls to methods in the presence of automatic type coercions. If more than one method matches when taking into account all legal coercions, gpcp will reject the program and require the user to specify the intended coercions of the actual parameters.

8.4 Interfacing to constructors

If a foreign class has a “no-arg” constructor, then this will be implicitly called whenever an object is created by the use of the standard procedure *NEW*. However if it is necessary to access constructors with arguments, then it is possible to define an alias for the constructor in a foreign module. In every case the constructor will be accessed by means of a static, value returning function that returns an object of the constructed class. The fact that this is a constructor *must* be made known to *gpcp* since the way in which these methods are called differs from other methods. On each underlying platform there is a "magic" name that is used for calling a constructor. On the *JVM* the name is "`<init>`", while on *.NET* the name is "`.ctor`". These two strings are used as the explicit string that defines such a procedure in the foreign definition. An example of an interface to a constructor with arguments, in the syntax used by the *Browse* tool, might be —

```
PROCEDURE Init*(width,height : INTEGER) : Rect,CONSTRUCTOR;
```

The identifier "`CONSTRUCTOR`" is not a reserved word, but a context sensitive mark that may be used as an ordinary identifier elsewhere in the program.

Note that this declaration would normally appear in the static part of the record defining the class *Rect*. Calls to this procedure in a *Component Pascal* program, such as —

```
rec1 := F.Rect.Init(25,17);
```

would, depending on the target platform, translate into a call to one or the other of —

```
namespaceName.Rect::.ctor(int32,int32)
packageName.Rect.<init>(II)
```

Of course, if you extend a foreign class that does not have a public no-arg constructor, then you will not be able to construct values of your own type using *NEW*, since this implicitly calls the no-arg constructor of its super-type. In this case, it is necessary to define a new constructor signature for your extended type. From *gpcp* release 1.2 there are two ways to do this. If the desired constructor has the same signature as the constructor of the supertype, then the first method may be used. In the case of the example above, the required syntax is shown in the following fragment —

```
TYPE MyRect* = POINTER TO RECORD (Mod.Rect) ... END;
  ...
PROCEDURE Init*(w,h : INTEGER) : MyRect,CONSTRUCTOR;
```

The constructor does not define a code body, and simply passes its arguments to the super-type constructor with matching signature.

The new syntax in *gpcp* version 1.2 is considerably more flexible. The *Component Pascal* constructor is not required to have the same signature as the constructor of the super-type. An example of the syntax defining another constructor for the extended type defined above is —

```
PROCEDURE MkMyRect*(Formals) : MyRect,BASE(actuals);
  (* Local-declarations *)
BEGIN
  (* Constructor body code *)
  RETURN SELF;
END MkMyRect;
```

in the code the formal and actual parameter lists have been left un-elaborated.

The identifier "`BASE`" is a not a reserved word, but is a context sensitive mark. Of all publicly available constructors for the super-type it specifies a call of the one with signature matching the types of the "*actuals*" argument list. This super-type constructor will be called as the first action of the constructor, before the new fields of the derived object are initialized. Within the body of the constructor the object under construction is denoted by the identifier "`SELF`". The constructor *must* return this object along every terminating path of the body. It is an error if the actual parameter expression types in the *BASE* super-call do not choose a unique super-type constructor.

8.5 Declaring static features of classes

Classes in foreign modules may be declared either as records or as pointers to records. However, it is recommended that on the *JVM* platform the pointer form be always used, as a helpful reminder to the user that at runtime the objects will be dynamically allocated. On the *.NET* platform value classes should be declared as plain records, with no explicit base type. On both platforms array types should be declared as pointers to arrays, again reminding the user that all arrays are dynamically (and explicitly) allocated.

In order to access static features of foreign classes, the syntax extension of records given in Figure 12 must be used. In the optional static section of a record declaration we may define constants, static fields and static (i.e. non type-bound) procedures.

We may consider the following example —

```
CP Foreign Definition                Component Pascal Usage
FOREIGN MODULE ModNm;
  TYPE Cls =                         ModNm.Cls        (* class name *)
      POINTER TO RECORD
      STATIC
        statVar* :  CHAR;            ModNm.Cls.statVar
        PROCEDURE StatProc();        ModNm.Cls.StatProc()
      END;
END ModNm.
```

In this example we select the static member by qualifying the designator by the type-name of the class.

Type-bound methods will be defined lexically outside of the record declaration in the normal *Component Pascal* way, remembering that only the heading is required. On the *.NET* platform the distinction between virtual and instance methods is made automatically. Instance methods are *NEW* but not *EXTENSIBLE*. On the *JVM* platform the possibility of optimizing the calls to such methods is left to the *JIT* to determine.

Note that the foreign modules which arise from *C#* on the *.NET* platform or are written in *Java* can never have static features outside of classes. If you are writing the foreign module yourself you may use the default class naming scheme described in Section 3.2. However if you are matching an existing package, you will need to use the explicit name override described earlier in this Section. This allows you to control the package name, but does not allow you to name an implicit static class for static features. Therefore you will need to use the mechanisms of this sub-section if the package contains any static features.

8.6 Automatic module renaming

Programs written in *C#* that contain a single class definition only are often created in files that take their name from the name of the class. If you try to match this same structure in *Component Pascal*, you run into a small difficulty on the *.NET* platform. Suppose you want to export a class *Rename* from a module named *Rename*. In this case the external class name in *.NET* will be “`[Rename]Rename.Rename`”, and this name will clash with the name of the “synthetic static class”. In this circumstance *gpcp* will automatically rename the static class, by pre-pending two underscore characters. If the module with the renamed class is imported, *gpcp* will find the renamed symbol file. In both contexts *gpcp* will issue a warning that the renaming is taking place —

```
C:\gpcp\work> gpcp Rename.cp UseRename.cp
   1 MODULE Rename;
**** -------^ Warning: Default static class has name clash
**** Renaming static class to <__Rename>
#gpcp: <Rename> No errors, and one warning
   2   IMPORT Rename, CPmain;
**** ---------^ Warning: Looking for a auto-renamed module
**** ---------^ Looking for module "Rename" in <__Rename.cps>
#gpcp: <UseRename> No errors, and one warning
```

9 Installing and Trying the Compiler

9.1 Installation

The compiler is packaged in a single installer file “`setup.exe`”. If you use the installer version (from version 1.1.4) you should not need to do anything other than make responses to the installer’s queries. Complete instructions for installing and trying out the compiler are in the separate document “*Getting Started with GPCP*”.

Figure 13 is the complete folder hierarchy of the installed compiler. The six first-level subdirectories of the distribution are

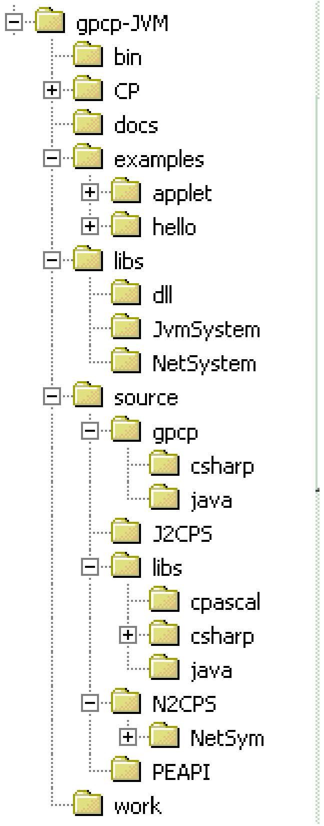

Figure 13: Distribution File Tree

- **bin** — the binary files of the compiler
- **CP** — the class file tree of the tools and libraries
- **docs** — the documentation, including this file
- **examples** — some example programs
- **libs** — contains the simple library files
- **source** — the source files
- **work** — a working directory to play around with

The bin directory needs to be on your *PATH*. Typical commands to set this variables are —

```
set PATH=%PATH%;C:\gpcp\bin
```

On *UNIX* systems the environment variables would typically be set using commands such as —

```
PATH=$PATH:$CPROOT/gpcp/bin
```

Where *CPROOT* is the root of the *gpcp* distribution. The command file “`cprun`” will pass the environment variable *CPSYM* to the program, and will also set the class path.

The “`CP`” directory is the root of the class-file tree. This directory contains a subdirectory for each module of the system. There are almost 250 class files in the tree, in the initial distribution. However, when you run programs class files in the *local* class file directory take precedence over those in the *CPROOT* directory.

The “`libs`” directory contains the symbol files for the Component Pascal libraries. There are three subdirectories under “`libs`”. The first of these is empty in the *JVM* version. The “`JvmSystem`” directory is for the symbols files to interface to the Java runtime. The “`NetSystem`” directory contains the symbol files that allow Component Pascal programs to access the base classes of the *.NET* system.

10 Future Releases

Release 1.2 still has a very limited range of libraries packaged with it, essentially only those needed to bootstrap the compiler. The distribution is sufficient to try out the compiler, and is being updated on a frequent basis. We expect new releases to contain new tools and new libraries.

Updates are announced and available from `http://www.citi.qut.edu.au/research/plas/projects/cp_files`

10.1 Change summary

Changes from 1.2.0

The following changes and corrections are included in the 1.2.x release.

- Support for boxing and unboxing of *CLS* value types is included.
- The vector types have been included.

- The parser now allows return types and formal parameters to be anonymous constructed types. The compiler gives a warning when the type so defined will be inaccessible and hence useless.
- A string library *StringLib* has been included.
- Some corrections have been made to the *RealStr* library.
- The “`WinMain`” pseudo-module introduced to mark base modules for windows executables that do not start a console when launched.
- Unsafe facilities in module “`SYSTEM`” introduced.
- Enhanced compatability between native strings, string literals and character literals.
- Correction to the semantics of subset inclusion tests, both versions.

Changes from 1.1.6

The following changes and corrections are included in the 1.2.0 release.

- The semantics of “super-calls” were incorrect in the case that the immediate super-type did not define the method being overridden. In version 1.2 the notation “`Foo^()`” denotes the overridden method no matter how distant it is in the inheritance hierarchy.
- New options have been implemented for output directories.
- The default behavior for the “`/nodebug`” option is to use the direct *PE*-file writer. This is significantly faster than going through `ilasm`. Unfortunately, this new file-writer does not produce debug symbols at this stage. There is separate documentation for the *PEAPI* component included with this release.
- The permitted semantics for constructors with arguments is significantly enhanced. This is of some importance when deriving from types that do not have public no-arg constructors.

Changes from 1.1.4

The following changes and corrections are included in the 1.1.6 release.

- Uplevel addressing of reference parameters is now permitted in the *.NET* release, although this has inexact semantics in some cases.
- A number of corrections to the *JVM* code-emitter have been added.
- The new built-in function *BOX* has been added.
- Trapping of types that attempt to indirectly include themselves is improved.
- An automatic renaming scheme is implemented for modules that attempt to export types with the same name as the module on the *.NET* platform.

Changes from 1.1.3

The following changes and corrections are included in the 1.1.4 release.

- The copyright notice has been revised. *gpcp* is still open source, but now has a "FreeBSD-like" licence agreement.
- A correction to the *Java* class-file emitter now puts correct visibility markers on package-public members. Appletviewer didn't care, but most browsers objected!
- It is now permitted to export type-bound procedures of non-exported types, provided the procedure overrides an exported method of a super-type.
- More line-markers are emitted to *IL* in *.NET*. This makes it possible to place a breakpoint on the predicate of a conditional statement, and have the debugger stop on the predicate rather than the next executable statement.
- The type-resolution code of "`SymFileRW.cp`" has been radically revised. It is believed that the code is now immune to certain problems caused by importing foreign libraries with circular dependencies.

11 Appendix: Working with Native Strings

There are some subtleties in converting to native strings. The following example demonstrates several strategies. The example tries to call the *equals*() method of *java.lang.String* to compare with a *Component Pascal* literal string.

```
MODULE StringCompare;
  IMPORT JL := java_lang, CPmain;

  VAR type :  JL.Class;
       name :  JL.String;
       ltNm :  JL.String;
       sObj :  JL.Object;
BEGIN
  name := type.getName();
 (*
  * This attempt works because String.equals() is not overloaded
  * This binds to the procedure matching
  * PROCEDURE (s :  JL.String)equals*(JL.Object) :  BOOLEAN
  *)
  IF name.equals("Blah") THEN END;
 (*
  * Conversions use built-in functions. Here is a non-standard one that converts
  * char-arrays to native strings. This works ...
  *)
  IF name.equals(MKSTR("Blah")) THEN END;
 (*
  * In the case of assigments (or non-overloaded method calls), the compiler can
  * work it out by itself without the MKSTR. Literal char arrays can be assigned to
  * objects or strings. This works.
  *)
  ltNm := "Blah";(* gpcp automatically converts the string to JL.String *)
  IF name.equals(ltNm) THEN END;
 (*
  * In the case of reference variables the type-assertion / cast syntax does work –
  * the following two calls bind to the same method.
  *)
  sObj := "Blah"; (* gpcp automatically converts the string to JL.Object *)
  IF name.equals(sObj) THEN END;
  IF name.equals(sObj(JL.String)) THEN END;
END StringCompare.
```

The eqivalent example using the libraries of the *.NET* platform is much more complicated, because the *Equals* method of the native string type has several overloads. The release notes for the *.NET* version treat the example in some detail.

12 Appendix: Overriding the Default Naming

The default naming scheme for the *JVM* version of gpcp uses the module name as the stem name for the output files, the *JVM* package name and the dummy static class name. All of these defaults may be overridden as described here. This may be necessary if another component expects a particular naming pattern.

Consider the following short program —

```
MODULE ModId; (* default naming will be used *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output class files will be “`CP/ModId/ModId.class`” and “`CP/ModId/ModId_ClsId.class`”. The name of the dummy static class will be *“CP.ModId.ModId”*, and the name of the class that represents the record type will be *“CP.ModId.ModId_ClsId”*.

It is allowed to follow the module name with a bracketed string that specifies the complete package name of the resulting classes. A typical string would be —

```
MODULE ModId ["CP.Foo"]; (* explicit package name *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the base class file will be “`CP/Foo/ModId.class`”, and the name of the dummy static class will be *“CP.Foo.ModId”*. The name of the class that represents the record type will be *“CP.Foo.ModId_ClsId”*, which will be found in file “`CP/Foo/ModId_ClsId.class`”.

The only special case is that of an empty package name, signified by an explicit empty string.

```
MODULE ModId [""]; (* empty package name *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the base class file will be “`ModId.class`”, and the name of the dummy static class will be *“ModId”*. The name of the class that represents the record type will be *“ModId_ClsId”*, which will be found in file “`ModId_ClsId.class`”.

For the *.NET* target there is a special case that arises if an explicit class has the same name as the module. On that platform an automatic renaming of the symbol file and dummy static class is required. On the *JVM* platform the case is innocuous.

```
MODULE ClsId; (* module name clashes with class id *)
  TYPE ClsId* = RECORD ...  END;
END ClsId;
```

In this case the name of the base class file will be “`CP/ClsId/ClsId.class`”, and the name of the dummy static class will be *“CP.ClsId.ClsId”*. The name of the class that represents the record type will be *“CP.Foo.ClsId_ClsId”*, which will be found in file “`CP/Foo/ClsId_ClsId.class`”.