Gardens Point Component Pascal — Release Notes

John Gough

September 17, 2004

> **This document applies to GPCP version 1.3 for .NET**
> **(Microsoft Common Language Runtime)**

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 other *.NET* languages *N2cps*. There will be other utilities added later.

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 Microsoft.NET 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 either three or four 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`", "`Hello.il`", and either "`Hello.dll`" or "`Hello.exe`".

---

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

The "`*.cps`" file is the symbol file which contains the metadata that describes the facilities exported from the module. The "`*.il`" file contains the Common Intermediate Language (*CIL*) representation of the program. The program executable will be "`*.exe`" if the program contains an entry point (i.e. if the module imports *CPmain*), otherwise the compilation will create a dynamic link library "`*.dll`". All of these files are created in the current directory. If a listing file is created it will have filename extension "`.lst`".

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`".

It is possible to invoke the compiler so as to produce just the intermediate language file, and to then manually invoke the intermediate language assembler "`ilasm`". The assembler may then be used to produce any of its possible output formats.

2.2 Invoking the compiler

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

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

The options are given in Figure 1.

UNIX-style options "–" may also be used. In the *JVM* versions the "–"form is the expected default. Any number of files may be added in a white-space separated list.

2.3 Target choice

The compiler may choose its output language at runtime. The default output when running on the .*NET* platform is .*NET* assembler (*CIL*). The recognized options are —

> `/target=net`   *this is the default* CIL *format*
> `/target=jvm`   *this causes* Java *byte codes 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.

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 one *CIL* assembly language file is produced for each input file. No executable 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 executable (*PE*) file (either .*DLL* or .*EXE*) is produced for each input file. If the default `/debug` flag is in effect then a textual *CIL* file (extension .il) is produced and the *PE*-file is created by the `ilasm` tool. If the `/nodebug`

`/bindir=X`	place *PE* (and pdb) files in directory "X"
`/copyright`	display the copyright notice
`/debug`	emit debugging symbols (default)
`/nodebug`	do not emit debugging symbols
`/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)
`/ilasm`	use ilasm even with /nodebug
`/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
`/nocheck`	produce code without arithmetic overflow checks
`/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"
`/unsafe`	allow import of *SYSTEM* functions
`/vX.X`	*.NET* framework (v1.0 | v1.1 | v1.2)
`/verbose`	chatter on about progress during compilation
`/version`	emit version information
`/vserror`	errors are in Visual Studio format
`/warn-`	suppress warning messages from the console
`/nowarn`	same as /warn-
`/whidbey`	target code for "Whidbey Preview" (same as v1.2)
`/xmlerror`	errors are in *XML* format

Figure 1: *gpcp* options

flag is in effect then a *PE*-file is directly produced using the *PEAPI* component. This behavior may be overridden by use of the `/ilasm` flag, which causes textual *CIL* to be produced, even in the presence of the `/nodebug` flag.

Output files with "`/target=jvm`" option

If the compiler is run with the `/target=jvm` flag, the input files are parsed and type-checked, and a symbol file and one or more class files will be produced. These class files are written directly, and do not require the installation of *Jasmin*.

   If, in addition, the `/nocode` flag is used, then *Jasmin* assembly language (`*.j`) files will be produced, but *Jasmin* will not be invoked.

   If, instead of /nocode the `/jasmin` flag is added, *Jasmin* assembly language files are produced for each input file. Following this, the *Jasmin* assembler will be automatically invoked to create the corresponding class files. Because a separate process needs to be created for each invocation of *Jasmin* this is quite slow.

2.4   Overflow checking

Ordinarily the compiler produces code that performs arithmetic overflow checks on all operations. Narrowing assignments (such as assigning a long value to an integer variable) are also range checked. Compiling with the `/nocheck` option removes these checks. 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 4.13.

2.5   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

2.6   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: 855 entries in hashtable of size 8209
#gpcp: import time      63mSec
#gpcp: source time     110mSec
#gpcp: parse time      202mSec
#gpcp: analysis time    47mSec
#gpcp: symWrite time    16mSec
#gpcp: asmWrite time   219mSec
#gpcp: assemble time   281mSec
#gpcp: total time      938mSec
```

Figure 3: Compile statistics example

* AsmWrite time is the time to write out the assembly language (*CIL*) output.

* Assemble time is the time taken to spawn a new process and run `ilasm`. Assemble time is always zero if the direct to *PE*-file output path is selected by `/nodebug`.

2.7   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.

2.8   Choosing the Output Directories

By default all output files are created in the current directory. This behavior may be overridden with the options `/bindir` 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 specified directory using the `/bindir=`*target-directory* option.

If the *JVM* target has been chosen then the `/symdir` option still applies, but `/bindir` option does not. Instead, the root of the output class file hierarchy may be specified using a syntactically similar `/clsdir` option.

2.9 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 —

> `$> 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, 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* 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.

2.10 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 —

> `$> 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*

2.11 Symbol File Generator N2CPS

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

$> N2cps [*options*] *assemblyName*

where current options are /v for "verbose" and /w for "warnings". Each specified assembly will produce one or more symbol "*.cps" files. For example, the main system library assembly *mscorlib* will produce *mscorlib_System, mscorlib_System_Reflection*

and so on. It is prudent to ensure that you have the *NetSystem* symbol files that exactly correspond to the version of *.NET* that you have installed.

To synchronize, go to the `\gpcp\source\N2CPS` directory and run first the *NetClean* script and then the *NetMake* script.

> **Warning**
> The current release version of *N2CPS* cannot consume the metadata for generic types in the Whidbey preview. Do not attempt to run *N2CPS* on assemblies such as mscorlib that contain generics.

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
REGISTER	attaches a procedure to a (*.NET*) multicast delegate
DEREGISTER	detaches a procedure from a multicast delegate
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 Common Language Specification names

Fully qualified names in the Common Language Specification (*CLS*) comprise four parts.

* Assembly name – the assembly in which the class will be found

* Namespace name – this specifies the namespace of the class

* Class name – the class name

* Feature name – the field or method name.

An example might be –

```
[mscorlib]System.Exception::ToString
```

where *mscorlib* is the assembly name, *System* is the namespace, *Exception* is the class name, and *ToString* is a method name.

In this version of *gpcp*, the compiler produces one assembly per module, and one namespace per module. Both the assembly and the namespace names are the same as the module name. Thus a type-bound procedure called *isString*() bound to the type *UnaryX* in module *ExprDesc* would have the *CLS* name —

```
[ExprDesc]ExprDesc.UnaryX::isString
```

Procedures and variables at the module level are declared in the *CLS* 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 *CLS* name —

```
[Foo]Foo.Foo::xId
```

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

If you do browse the assembler output of the compiler, you will notice that almost all names are escaped with single quotes like `'this'`. This is done to avoid clashes with the many names that are reserved in the assembler.

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 (low-line 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 DLLs and EXEs

The compiler can produce either stand-alone executables (.exe files) or dynamic link libraries (.dll files). Executable files must have an entry point known to the runtime. The entry point method optionally takes an array of native-strings as parameters. Any such command line arguments are accessed through the library *ProgArgs*.

If the source file contains the import of the special module name *CPmain*, then an executable file is produced as output. In this case the module body is named "`.CPmain`", 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, instead, the source file contains the import of the special module name *WinMain*, then again an executable file is produced as output. However, in this case the *PE*-file produced is a Windows executable, and the module body is named "`.WinMain`". Windows executables do not start a command window when launched.

If the source file does not import either *CPmain* or *WinMain* then the module body becomes the "class constructor" which is executed at the time that the dynamic link library is loaded on demand.

If the compiler is invoked with the `/nocode` option, then only the assembler (*CIL*) file is created. In this case the assembler `ilasm` may be invoked so as to create either a .dll or an .exe file using the command "`ilasm /DLL`" or "`ilasm /EXE`". Of course, it is an error to try to create an executable file if the source does not contain an entry point.

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 (restricted)

* Uplevel access to reference parameters (inexact semantics)

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 restricted in the current release. Arbitrary procedures of matching type may be assigned to procedure variables, and called in the usual way. However assignment of procedure variables is only permitted if the two sides of the assignment have the same type. That is, assignment of procedure values other than literal procedures requires *name compatibility*, rather than the *structural compatibility* specified in the language Report. This restriction will probably be removed in the next major release.

Non-local variable access is permitted in an unrestricted way since release 1.1.6. However, in the case of reference (*VAR*) parameters of unboxed[2] type that are accessed from within nested procedures the semantics of parameter passing is modified because the actual parameters are passed by *copying* rather than by *reference*. The compiler gives an explicit warning in these unusual circumstances.

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 —

$$i \; DIV \; j = \lfloor i/j \rfloor$$

$$(i \; DIV \; j) \times j + (i \; MOD \; j) = i$$

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.

$$i \; DIV0 \; j = RTZ(i/j)$$

$$(i \; DIV0 \; j) \times j + (i \; MOD0 \; j) = i$$

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 —

---

[2] *Component Pascal* types that are unboxed in the *.NET* implementation are scalar values and record types that are not extensible, do not extend another type, and are not defined as the anonymous bound type of a pointer type.

```
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 —

<div align="center">

Type   ::-   *...*   - - *other type constructors*
 | "VECTOR" "OF" *Type*.

</div>

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 *.NET CLS*. Types which extend *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* } "]" .

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 EVENT types

Event types are declared in *gpcp* with the same syntax as procedure types, but with the keyword *PROCEDURE* replaced by *EVENT*. Events are implemented as multicast delegate types in the *.NET* framework. If variables are declared to be of some event type, then it is possible to use the new built-in procedures *REGISTER* and *DEREGISTER* to register or deregister callbacks on the multicast delegate.

The usage for registering a callback is —

---

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

```
REGISTER(target-variable, callback-method);
```

The target variable is the designator of the object, which must be of some event type. The denotation of the callback method has two forms. If a *static procedure* is to be registered, then the simple procedure name is used. If the callback is intended to invoke a particular type-bound method on some particular object, then the syntax "*object.method*" is used. This works for any type-bound procedure in *Component Pascal*. The usage for deregistering a callback is syntactically identical, but using the non-standard built-in procedure *DEREGISTER* rather than *REGISTER*. A callback may be registered multiple times. The delegated calls are made in order of registration.

4.8   Unsigned Byte Type

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*().

4.9   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 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 `System.Object::GetType()`.

4.10   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. There is a special case version also, for *CLS* value classes. The signatures are —

```
PROCEDURE BOX(s : CP-type) : POINTER TO CP-type;
PROCEDURE BOX(s : CLS-ValCls) : System.Object;
```

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^);".

The special case of *BOX* applies to arguments that belong to *CLS* value classes as used in the *.NET* base class libraries. In this case, for compatability with the libraries, the function returns a "boxed" copy of the value class datum. Such boxed values are treated by the system as having type *System.Object*. Boxed values may be unboxed by using the standard type-check syntax. Here is an example —

```
VAR datTim : Sys.DateTime; (* A CLS value type *)
    objRef : Sys.Object;    (* Native object type *)
  ...
objRef := BOX(Sys.DateTime.get_Now()); (* Boxing *)
  ...
datTim := objRef(Sys.DateTime);         (* UnBoxing *)
```

Note that in the final statement of the fragment the type-check unboxes the object to create a reference, and the assignment performs a *value copy* as would be expected for a value type.

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.

4.11   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.

4.12   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 test-verify the output of compilation by running the stand-alone program executable verifier "`peverify`" over the file. Figure 5 shows the result of running the verifier over an example module.

```
D:\gpcp-CLR\work>peverify Browse.exe
Microsoft (R) .NET Framework PE Verifier  Version 1.0.3705.0
Copyright (C) Microsoft Corporation 1998-2001.

All Classes and Methods in Browse.exe Verified
D:\gpcp-CLR\work>_
```

Figure 5: Running `peverify` over an example *PE*-file

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 N2cps.

4.13   Unchecked arithmetic

By default, all arithmetic is overflow-checked, and all narrowing assignments are range checked. Sometimes it is necessary to turn off this behaviour. There are two means to do this. One of these is a custom attribute that is applied on a per-procedure basis. Checks may also be turned off from the command line for all compilations in that invocation.

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 —

$$Body \quad ::- \quad \text{“BEGIN”} \, [\, \text{“[UNCHECKED\_ARITHMETIC]”} \,]$$
$$StatementSequence \, \text{“END”} \, identifier \, .$$

An example of the use of this construct, from the source of the compiler itself, is the identifier hash function shown in Figure 6. This function performs a rotate-and-add 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.

```
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 6: Code of the hash function

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, as described on page 11.

---

[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 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.

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 *java.lang.Exception* type, with the given string as embedded information. If

> **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*.

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.

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

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 7. 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. *SYSTEM* and *FOREIGN* has slightly different semantics on the *.NET* platform, but are synonyms on the *JVM* version.

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*

```
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 *)
```
<div align="right">RTS continues ...</div>

Figure 7: 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 8: Source of the *RTS* pseudo-module, continued

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 9.

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.

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 10.

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

```
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 9: Interface of the RealStr library

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 (.NET only)

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 —

```
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 10: 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 11: Interface to the *StringLib* library

```
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);
```

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 12. In the first release the predicate function *More* always returns the *TRUE*

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

Figure 12: Source of the *StdIn* pseudo-module

value. The team will restore the functionality when we figure out a way of making the behaviour the same on the two execution platforms.

7 Foreign Language Interface

7.1 Accessing the underlying native types

As seen in Figure 7 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 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["[blah]namespace"];
```

declares that this module will be found in *.NET* assembly "`blah`" within namespace "`namespace`". 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 10.

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 "`.ctor`". 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=jvm`" version, the magic name is "`<init>`".

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.

7.4   Accessing Nested Classes

The *CLS* allows for class declarations to be nested within other classes. In *Component Pascal* such classes have names of the form

> *EnclosingClassName*$*NestedClassName*

This compound name is a single identifier, as far as *gpcp* is concerned, and must not have any embedded spaces. The *Browse tool* uses the same convention. As an example, in the foreign module "System_Windows_Forms_" there is a class that browse displays as —

```
Control$ControlCollection* =
    POINTER TO EXTENSIBLE RECORD (mscorlib_System.Object)
    STATIC
      PROCEDURE init*(p0 :  Control) :
                      Control$ControlCollection,CONSTRUCTOR;
    END;
```

In this example the nested class *ControlCollection* is enclosed by the class *Control*. Figure 13 is an example program that accesses the nested class, and creates an instance of the class. Note that the outer, enclosing object is constructed first, using the no-arg constructor, and then is passed as an argument to the explicit constructor for the nested class object.

```
MODULE Nested;
  IMPORT SWF := System_Windows_Forms_;
  VAR ct : SWF.Control;
      cc : SWF.Control$ControlCollection;
BEGIN
  NEW(ct);(* Create outer object and pass to inner constructor *)
  cc := SWF.Control$ControlCollection.init(ct);
 ...
END Nested.
```

Figure 13: Using a nested class

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 14. 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 14: 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 14 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 15 is the complete folder hierarchy of the installed compiler. The six first-level subdirectories of the distribution are

* **bin** — the binary files of the compiler

* **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*, and the environment variable *CPSYM* must point to the "`libs`" directory. Typical commands to set these variables are —

```
set CPSYM=.;C:\gpcp\libs;C:\gpcp\libs\NetSystem
set PATH=%PATH%;C:\gpcp\bin
```

Preferably these should be set in the system window of the control panel. If you use the installer version, the paths should be set automatically during installation.

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.

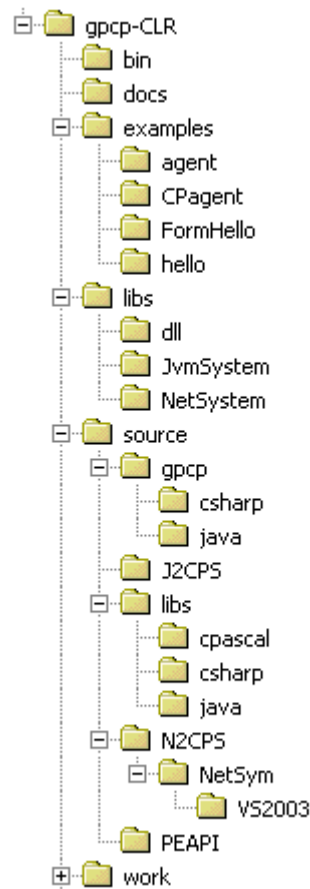Updates are announced and available from `http://www.citi.qut.edu.au/ research/plas/projects/cp_files`

Figure 15: Distribution File Tree

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 *System.String* to compare with a *Component Pascal* literal string.

```
MODULE StringCompare;
  IMPORT Sys := mscorlib_System, CPmain;

  VAR type :  Sys.Type;
      name :  Sys.String;
      ltNm :  Sys.String;
      sObj :  Sys.Object;
BEGIN
  name := type.get_Name();
 (*
  * This attempt does not work, since literal strings may have several automatic
  * coercions that match different overloads of the Equals( ) method
  *)
  IF name.Equals("Blah") THEN END;
 (*
  * Type assertions only apply to variables. This attempt caused a compiler
  *  exception in 1.1.4!
  *)
  IF name.Equals((Sys.String)"Blah") THEN END;
 (*
  * This next try is invalid syntax, the cast construct is a type-check, not a conversion
  *)
  IF name.Equals("Blah"(Sys.String)) 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 System.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 different overloads.
  *)
  sObj := "Blah"; (* gpcp automatically converts the string to System.Object *)
  IF name.Equals(sObj) THEN END;
  IF name.Equals(sObj(Sys.String)) THEN END;
END StringCompare.
```

Curiously, these problems do not arise for the *JVM* version, since in the *Java* libraries the "`equal`" predicate for the *string* type overrides the predicate from *object*. In the *JVM* case there is no overloading.

12   Appendix: Overriding the Default Naming

The default naming scheme for the *.NET* version of gpcp uses the module name as the stem name for the output files, the *CLR* assembly name, the namespace 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 file will be "`ModId.dll`", the name of the dummy static class will be "`[ModId]ModId.ModId`", and the name of the class that represents the record type will be "`[ModId]ModId.ClsId`".

It is allowed to follow the module name with a bracketed string that specifies either or both of the assembly name and the namespace name. A typical string would be —

```
MODULE ModId ["[AsmNm]SpcNm"]; (* both *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`AsmNm.dll`", the name of the dummy static class will be "`[AsmNm]SpcNm.ModId`", and the name of the class that represents the record type will be "`[AsmNm]SpcNm.ClsId`".

In the case that only the assembly name is specified, there is no namespace defined.

```
MODULE ModId ["[AsmNm]"]; (* assembly name only *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`AsmNm.dll`", the name of the dummy static class will be "`[AsmNm]ModId`", and the name of the class that represents the record type will be "`[AsmNm]ClsId`".

Conversely, if the namespace name is specified, but no assembly name, then the assembly name is taken from the module identifier, as in the default case.

```
MODULE ModId ["SpcNm"]; (* namespace only *)
  TYPE ClsId* = RECORD ...  END;
END ModId;
```

In this case the name of the output file will be "`ModId.dll`", the name of the dummy static class will be "`[ModId]SpcNm.ModId`", and the name of the class that represents the record type will be "`[ModId]SpcNm.ClsId`".

There is just one special case remaining. In all of the previous cases the name of the dummy static class is taken from the module identifier, with the symbol (metadata) file using the same stem name. If the default name of the static dummy class clashes with the name of an explicit class then the dummy static class will be renamed, as described in Section 8.6.

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

In this example the name of the output file will be still be "ClsId.dll" but name
of the dummy static class will be renamed to "[ClsId]ClsId._ _ClsId", and the
name of the class that represents the record type will be "[ClsId]ClsId.ClsId".
The symbol file will have the name "_ _ClsId.cps" and, as noted earlier, will be
automatically found by the compiler if the module name appears in an import list.