Constructing Compilers in a Persistent Environment

Alan Dearle

Computational Science Department
University of St.Andrews
North Haugh
St Andrews
KY16 9SS

Abstract

Traditionally compilers have been described as consisting of separate modules: the syntax analyser; lexical analyser; code generator etc. However, in practice modern compilers are rarely constructed in this manner. This may be because it is difficult to find the correct levels of abstraction necessary to build a compiler from components without losing efficiency.

In this paper a set of modules are described which may be combined in various ways to provide a whole family of compilers. The family of compilers includes: batch compilers; interactive compilers; file based compilers and compilers that operate entirely within the persistent environment.

The approach taken is that of plug-in components. The system facilitates language and architecture experiments by providing a framework in which modules may be interchanged and combined freely.

1. Introduction

Throughout the computing community the use of modules to divide a complex task into easier subtasks is recognised as *a good thing to do*. The task of compilation is an ideal application of modular decomposition. In this paper it will be demonstrated how a flexible compilation environment can be created using modular decomposition within a persistent environment.

The system described consists of a number of different modules each of which is specified by a type signature. Many different instances of particular modules may exist side by side in the persistent store. An instance of a compiler is constructed by composing instances of the different components. Compilers that appear to the user to be quite different, for example ones that compile different languages, may share much of the same code.

2. Why use a persistent environment?

The decomposition of a compiler into different sub-tasks has been well known for many years. Typically compilers are described to undergraduate students as consisting of a lexical analyser, a syntax analyser, a code generator and so on [rdc]. However, these theoretical methods of constructing a compiler are often not strictly followed in order to gain more performance from a system. It will be shown that using the persistent store no significant overhead is incurred in space or time in building a flexible, modular system.

Let us firstly examine the space arguments. Many programming languages [pascal,C] allow libraries of functions to be constructed. These are fragments of code that have been separately compiled and are thought of as being useful to a community of programmers. When an individual wishes to use one of these functions a binder is employed to **copy** the code in the library and bind it into a new program. Thus, if a compiler is constructed by building library functions every compiler constructed will have its own copy of the code. This is a crude form of software reuse. In such a system only copies of code are being shared and not instances of modules as in a persistent system.

In a persistent system such as PS-algol [PS-algol] procedures may be stored in a type secure manner. Programs may link dynamically or statically to code in the database simply by indexing a structure class. In this way different programs may share instances of code rather than merely own a copy of the code. This method of building large systems is much more persistent space efficient than building libraries of functions and using a linker to obtain copies of the code. Furthermore, as the usage of a library function increases the greater the benefit from using the persistent store. Clearly the persistent store subsumes the role of a conventional procedure library.

Let us now examine the time overheads. Since PS-algol supports first class procedures, closures may be stored in the persistent information space. The user finds a procedure by navigating the persistent graph from a root of persistence. Once a procedure has been found in the persistent store it is indistinguishable from one declared in the main program. In other words there is no difference between a closure retrieved from the database and one declared locally. This implies that by using the persistent store no penalty is paid for decomposing large programs into modules - provided that a functional interface is used.

The only time penalty that may be incurred in using the persistent store is the time taken to navigate the store to find an appropriate procedure. This operation is equivalent to linking in a conventional system. The navigation of the persistent graph may be performed at an earlier time than call time. Therefore the user does not necessarily have to pay this time penalty every time the code is used. The time at which binding is performed will be discussed later in this paper.

The previous paragraphs of this section compare the persistent store to conventional technology but other benefits may be gained by using a persistent information space. These have been discussed in many other papers [procs,app]. The fact that every data type is allowed the full range of persistence means that complex data structures may be created without concern about storage methods. A good example of this is in the symbol table package. The symbol tables model lexographical scope and contain both complex type information and initialising code for declarations. All this information automatically persists because it is reachable from the executable code as will be seen later. Such a system could not be contemplated using conventional technology.

3. Construction rules for persistent systems

During the construction of the compiler tool kit a set of rules has evolved. These rules should be used as a paradigm for constructing systems in a persistent environment. These rules are: I/O independence, plug compatibility, binding independence, information hiding and encapsulation. Each of these will now be examined in detail.

Adherence to the first rule of I/O independence is perhaps the most difficult. It says that no modules should directly perform I/O. Instead the input and output of information should go via a procedural interface. If this rule is followed, modules constructed will have a much higher degree of usability. To demonstrate this, imagine a lexical procedure to parse a real number. Such a procedure may be found in most compilers. If that procedure displays an error message when an error is detected; it cannot then be used in a window based system since it will destroy the display. Instead, if the procedure takes as a parameter an error displaying procedure the procedure will be of greater utility.

The second rule of plug compatibility says that each module should have a well defined interface and that modules with the same interface may be freely substituted for each other. It is this rule that allows us to create a whole family of compilers by specifying interfaces and by having a number of different instances which conform to those interfaces.

The third rule of system construction, binding independence, is not commonly practised by programmers in persistent systems. In order for us to understand this rule let us consider two procedures written in PS-algol.

```
let example1 = proc()
begin
        structure container( proc() somethingUseful )
        let aContainer = s.lookup( "usedByExample1","database" )
        let usedByA = aContainer( somethingUseful )
        usedByA()
end
```

In this example the procedure first declares a structure class definition. This introduces a class along with some selectors and a constructor In the second line of the procedure an object is looked up using the key "usedByExample1" from a database called "database". In the procedure we are assuming that aContainer now points to an object of class container. Next the procedure is retrieved from the object by indexing the structure class. Lastly the procedure is applied. Notice that this procedure dynamically looks up the database to retrieve the procedure every time it is called. Such an action may be required in a development system where the most recent version of a procedure from a library is required. This code is typical of code written by PS-algol programmers.

```
let example2 = proc( pntr aContainer )
begin
        structure container( proc() somethingUseful )
        let usedByA = aContainer( somethingUseful )
        usedByA()
end
```

Example2 is similar to example1, it also applies a procedure obtained by dereferencing a structure of class container. Let us now examine the differences between these two examples.

Example1 has the strings "usedByExample1" and "database" bound into it - leaving the user with no option but to use the procedure stored in the appropriate table. It also leaves the user of the procedure with no option other than to bind dynamically to the data every time it is executed. In the second procedure no information apart from the structure class has been bound into the procedures closure. This allows the procedure to be used in a number of different ways which we will now examine. If the semantics of the first example were required the user could now write,

```
let synthsiseExample1 = proc()
begin
        let aContainer = s.lookup( "usedByExample1","database" )
        example2( aContainer )
end
```

As in example1 this procedure looks up the database everytime thus retrieving the most recent version of the procedure.

A user may, however, wish a static system with which to experiment without having possible changes to other modules affect the experiment. In such a case a static bind would be required. This may also be written using example2 by producing a procedure with the function bound into its closure. This can be seen in the following example,

```
let staticExample =
begin
        let aContainer = s.lookup( "usedByExample1","database" )

        proc() ; example2( aContainer )                         ! this is the result
end
```

In this example the container is looked up only once. It is only the procedure in the final line that is exported from the block and assigned to staticExample. The object pointed at by aContainer is statically encapsulated in the scope of the procedure. This is an example of block retention.

The procedure synthsiseExample1 looks up the database every time - giving us a dynamic bind whereas the procedure staticExample has the structure instance aContainer in its closure giving us a static bind. In the compiler toolset all the modules have been written in the style of example2 giving the user the option of composing programs statically or dynamically.

The fourth rule, is that of information hiding. This has been known for many years and is commonly practiced by users of abstract data types [adt].

The rule that modules should be totally encapsulated is best understood by example. Consider the following program segment, once again in PS-algol. The procedure saves an integer and returns the last integer saved.

```
let saved := 0

let saver = proc( int this -> int )
begin
        let temp = saved
        saved := this
        temp
end
```

Since the procedure uses shared store it cannot be used by a number of different programs to save values. This may be overcome by wrapping the procedure in a generator like so,

```
let saverGen = proc( -> proc( int -> int ) )
begin
        let saved := 0                          ! this is encapsulated

        proc( int this -> int )                 ! this is the procedure returned
        begin
                let temp = saved
                saved := this
                temp
        end
end
```

By this mechanism every procedure wishing to use saver may do so safely by calling saverGen to obtain a saver with its own store. In this way procedures may share code without having to share state. Notice that every saver produced has its own copy of the variable saved. Although this technique has been well known to researchers in persistent languages for some time it is unfortunately not commonly practiced. In the compiler toolset all of the modules are encapsulated in a generator so that each instance of a module operates entirely within its piece of store.

The rules of I/O independence, plug compatibility, binding independence, information hiding and encapsulation have proved to be invaluable in constructing the compiler toolkit. In the following sections we will see how these rules have been applied, in practice.

4. Persistent Architecture Intermediate Language

The interface between the syntax analysis module and the code generation module is provided by the Persistent Architecture Intermediate Language, PAIL [pail]. PAIL code is a graph structure comprising of abstract syntax trees containing all the information gleaned from the syntax analysis phase of compilation. The provision of PAIL makes it possible to write language independent code generators. It also allows experiments in language and language implementation to be carried out independently and in parallel. By operating on PAIL code rather than on unstructured source code it also much easier to write source level optimisers. Other benefits of PAIL are that it provides: an aid to partial recompilation, better run time debugging, high level protection and portable information.

Partial recompilation is much easier due to the fact that the PAIL code allows much finer analysis of the program than either the source code or the executable code. Since PAIL is an abstract view of the program the provision of syntax directed editors is also a simple task.

Also, the PAIL code itself is persistent. It is not consumed by the code generation phase as in many systems. Instead pointers to it are placed in the executable code in the form of pointer literals. This means that when an error occurs during the execution of the code all the symbolic and address information contained in the PAIL trees is available. This allows meaningful error messages to be displayed and symbolic debuggers to display contextual information to the user.

In addition, the Napier abstract machine [napier-mc], currently being developed, does not contain the usual capability based protection found in most object based architectures [hydra,cap]. Instead this protection is provided at a much higher level by the PAIL code. By forcing all languages, that wish to use the persistent store, to compile into PAIL, illegal operations on the store may be detected at an early stage and thus save much overhead in the underlying architecture.

A final benefit of having PAIL is that it provides portable information. If code is transferred between two disparate machines the PAIL code contains all the information necessary to recreate the code on the new machine. In this way, PAIL is a useful tool in a distributed environment.

Since the system described is persistent all the PAIL graphs may also be persistent. PAIL trees are accessible from the executing code allowing better code optimisation. It has been suggested in [cgen] that this task may be performed lazily possibly automatically. This would, if possible, be very difficult in a conventional system. The building of such a system in a persistent environment makes the task both manageable and feasible

5. Compiler Components

The compiler toolset consists of many components. The interface of each component is well defined so that new versions of any of the components may be easily created and used in a compiler.

A type module is responsible for creating representations of programming language data types. Selectors and procedures for displaying types are also included in this module. All type checking is also contained in this module. Thus outside of this module no knowledge of the representation of the type system is required.

A single module is responsible for noting and reporting all compilation errors. This module interfaces with the output module which displays all output to the user. Combined with the input module the output module provides a file independent I/O interface.

Conventional compiler modules are provided to fulfill the tasks of lexical analysis, syntax analysis and code generation. The code generator does not output its results directly to the file system or persistent store. Instead, it passes its results to one of two modules. The first of these is a code planter which outputs the code to the file system. The other, called magic, turns code into a closure within the system. This is the fixed point in the system and has to be written in a lower level of language. This is the only place in the PS-algol system where the type system is broken.

6. Compiler Composition

Generators for the various compiler modules are placed in the persistent store independently. Typically, more than one instance of each module can be found in the persistent store. In order to construct a particular compiler, these components need to be joined together. A good analogy is having several jigsaws all cut using the same pattern. A new jigsaw may be constructed by selecting pieces from different jigsaws. Provided that the pieces are placed in the correct positions a jigsaw displaying a new picture may be created. This may be viewed graphically in figure1.

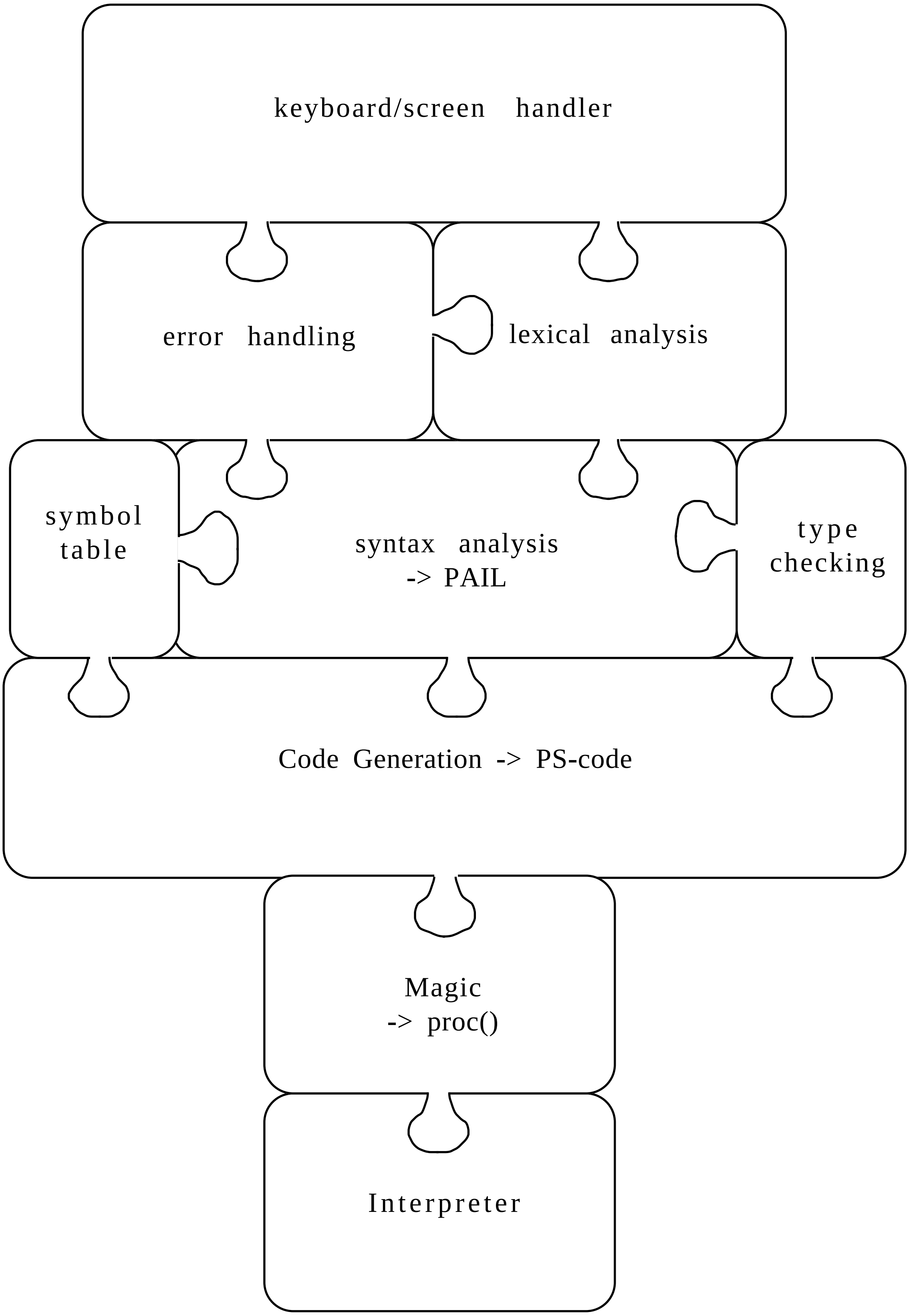
keyboard/screen handler
error handling
lexical analysis
symbol table
syntax analysis -> PAIL
type checking
Code Generation -> PS-code
Magic -> proc()
Interpreter

figure 1

In order to use a compiler built from the compiler toolset the user needs to write a small program to select the required modules. These linking programs are simple to write, very short and only a few of them have to be written. Most of the users of the current compiler toolset have never written such a program, relying on existing programs.

The compilation tools have been viewed as aids to the construction of a total compiler. Of course, they do not have to be used as such, being applicable in a number of other applications including spreadsheets (parsers and lexical analysers) and word processors (lexical analysers). Partial compilers may also be constructed, for example, merely as syntax checkers. Such tools have proved highly useful in the development of new language processors and type checkers.

7. Binding

Compiler components may be bound together statically or dynamically depending on the choice of the programmer. In some cases, for example when a new language system is still being developed, the user may want the most recent version of a particular module to be used. In this case, the programmer would use dynamic binding to combine the components. On the other hand, one could imagine a situation where a user would require a static system. This may be achieved by statically combining components into a compiler and then always using that version.

8. A few compilers

Let us now look at some of the useful tools that may be constructed from this technology. Perhaps the first tool that should be constructed is a conventional compiler that reads a source file compiles the code and puts the executable code into another file. This may be achieved simply by providing a module which performs I/O to the file system. Such a compiler is shown below.

```
!************** Get Modules from persistent store **************

! See appendix for a full type description of these modules.

let input = s.lookup( "input",comp.db )( Input.gen )( file.name )               ! an input.pack
let errors = s.lookup( "error",comp.db )( Error.gen )()                         ! an error.pack
let PS.types = s.lookup( "PS.types",comp.db )                                   ! a PS.types
let types = s.lookup( "types",comp.db )( Type.gen )( errors,PS.types )          ! type checker
let lex = s.lookup( "lex",comp.db )                                             ! a lex.gen
let sa = s.lookup( "sa",comp.db )( Sa.gen )(     options(),input,errors,
                                                 PS.types,types,lex )           ! proc( env -> PAIL )
let cgen = s.lookup( "cgen",comp.db )( Cgen.gen )( PS.types )
let planter = s.lookup( "planter",comp.db )( Planter.gen )()                    ! a code file planter

let global.env = s.lookup( "global.symbol.table",comp.db )                      ! global symbol table
let local.env = global.env( Create.scope )( global.env )                        ! a new symbol.table

let this.code = sa( local.env )                                                    ! do syntax analysis
if this.code is error.pack                                                         ! check errors
then write "**** Compilation fails ****'n",
        "No of errors = ",this.code( No.errors )(),"'n"
else
begin
          let c.file = cgen( this.code,global.env )                             ! do code generation
          planter( code.f.name( file.name ),c.file )                            ! put code out to a file
end
?
```

The work necessary to translate data in one format into data of another is a problem in any system. It is perhaps more acute in a persistent environment where programmers may use any data structure to hold persistent information. The ability to construct an arbitrary I/O module for the compiler avoids this problem entirely. For example an interactive compiler that reads and writes to the users console may be created by substituting the file handling module with one that reads and writes to the terminal.

A compiler which operates entirely within the persistent environment may be created by replacing the I/O module with one that reads from a piece of persistent data. If the module that places the compiled code in a file is replaced by one that turns the code into a closure, we create a compiler that is completely contained within the persistent environment. The benefits of having a compiler that is a procedure in the persistent store, known as a Callable Compiler, is well known to persistent programmers and is discussed elsewhere [dbs,browse].

A Callable Compiler which reads from the users console and immediately executes the code produced provides a compile and go environment which resembles a shell [sh,csh]. Of course, this shell is of much greater power since it has the backing a full programming language behind it.

9. Conclusion

We have seen that by combining instances of modules with slightly different functionality a rich set of tools may be provided. These modules may be bound together in different ways to provide an environment suitable for experimentation or production. The use of a high level intermediate language has also been explored and its benefits discussed.

The compiler toolset discussed in this paper has been implemented in PS-algol. The current tools available built from the components include callable compilers, batch compilers, interactive compilers and persistent information compilers. Currently syntax analysers exist for PS-algol and the language Napier [napier], currently struggling through its infancy. A compiler is also currently being developed to compile the applicative language Hope+ [hope+]. A compiler for an Object-Oriented language is also planned for the near future, as is the expansion of the compiler toolkit with tools that automatically generate some of the modules described in this paper from specifications.

10. Thanks for the fish...

I would like to thank Ron Morrison for his help during this work. Much of this work has been built from his original, beautifully simple compiler for S-algol [S-algol].

References

[rdc] Davie A.J.T. & Morrison R. "Recursive Descent Compiling",Ellis Horwood, (1981).

[pascal] Wirth N., "The Programming language Pascal", Acta Informatica 1, 1 (1971).

[C] Kernighan B.W. & Ritchie D.M., "The C programming language", Prentice-Hall, (1978).

[PS-algol] "The PS-algol Reference Manual fourth edition", Universities of Glasgow and St.Andrews PPRR-12 (1987).

[adt] Liskov B. & Zilles S.N., "Programming with abstract data types", ACM SIGPLAN Notices 9,4 (1974).

[procs] Atkinson M.P. and Morrison R., "Procedures as persistent data objects", ACM.TOPLAS 7,4 (1985).

[app] Atkinson M.P., Bailey P.J., Chisholm K.J., Cockshott W.P. and Morrison R., "An approach to persistent programming", Computer Journal 26,4 (1983).

[pail] Dearle A., "A Persistent Architecture Intermediate Language", Universities of Glasgow and St.Andrews PPRR-35 (1987).

[napier-mc] Dearle A. (ed), "The Napier Abstract Machine", Universities of Glasgow and St.Andrews PPRR, In preparation.

[hydra] Wulf W.A. et al., "Hydra: The Kernel of a Multiprocessor Operating System", CACM 17,6 (1974).

[cap] Needham R.M. & Walker R.D., "Protection and Process Management in the CAP Computer", International Workshop on Protection in Operating Systems, INRIA, Rocquencourt, (1974).

[cgen] Carrick R. & Munro D., "Execution Strategies in Persistent Systems", Proc Workshop on Persistent Object Systems: There Design Implementation and Use, Appin Scotland, (1987).

[dbs] Cooper R.L.,Atkinson M.P., Dearle A. & Abderrahamane A. "Constructing Database Systems in a Persistent Environment", Proc VLDB 1987, Brighton England, (1987).

[browse] Dearle A., Brown A.L., "Safe Browsing in a Strongly Typed Persistent Environment", to appear in The Computer Journal (1988).

[sh] Bourne S.R., "An Introduction to the Unix Shell", Bell Laboratories, (1978).

[csh] Joy W., "An Introduction to the C Shell", University of California, Berkeley, (1980).

[napier] Morrison R., "The Napier Reference Manual", Universities of Glasgow and St.Andrews PPRR, In preparation.

[hope+] Perry N. "Hope+", Imperial College Internal Report IC/FPR/LANG/2.5.1/7 (1987).

[S-algol] Morrison R.,"S-algol: a simple algol", Computer Bulletin II/31 (1982).

Appendix - Interfaces

This appendix describes the module interfaces in the PAIL system.

PS-algol Types

```
structure PS.types(      pntr  Global.INT,Global.CINT,Global.REAL,Global.CREAL,
                         Global.BOOL,Global.CBOOL,Global.PNTR,Global.CPNTR,
                         Global.STRING,Global.CSTRING,Global.FILE,Global.CFILE,
                         Global.PIXEL,Global.CPIXEL,Global.IMAGE,Global.CIMAGE,
                         Global.PIC,Global.CPIC,Global.VOID,Global.ANY,
                         Global.vector.type,Global.proc.type,Global.image.type,
                         Global.structure.type,Global.const.type,
                         Global.var.type,Global.field.type ;
                   proc( cpntr -> pntr )    Mk.vector.type,Mk.image.type,Mk.var.type,
                                            Mk.const.type ;
                   proc( pntr,pntr -> pntr)        Mk.proc.type,Mk.field.type ;
                   proc( string,pntr,pntr,int,int -> pntr)  Mk.structure.type ;
                   proc( pntr -> string ) Display.type,Reconstruct.source.type ;
                   proc( pntr,pntr -> bool ) Eq,Eq1,Eq2 ;
                   proc( pntr -> pntr) Elms,Args,Result,Decl.fields,Fields,Fieldof,Fieldt ;
                   proc( pntr -> string ) Tm ;
                   proc( pntr -> int) Total,Pntrs ;
                   proc( pntr -> pntr) Var.or.const,Strip.type ;
                   proc( pntr -> bool ) Pointer )
```

This structure contains all the base types for PS-algol. It also contains the necessary selectors and constructors needed for the higher order types. This module is completely applicative and therefore only contains the applicative type equality functions. Type comparing procedures which may cause errors are in another module which uses the equality functions contained in this one. Outside this module nothing need know the representations used for the types. Only one of these modules should exist in the PAIL system at a time.

Code generation

This module contains a code generator generator. The code generator in the initial version of the PAIL system generates PS-abstract machine code.

```
Cgen.gen = proc( PS.types -> proc( PAIL,symbol.table -> PS.Code ) )
```

```
structure PS.Code(    *int Code.vec ;
                      *string String.vec ;
                      *pntr Proc.vec ;
                      bool Valid.String.vec,Valid.Proc.vec ;
                      int Ms.size,Ps.size )
```

Magic

This module takes code from the code generator and manipulates it to form a PS-algol procedure.

```
Magic = proc( PS.code -> proc() )
```

Input

This module provides file independent input for the lexical analysis module.

```
input.gen = proc( string filename -> input.pack )

structure input.pack(   proc( -> int ) Readi.dev,Read.byte.dev ;
                        proc( -> string ) Read.dev,Peek.dev,Reads.dev,Read.a.line.dev ;
                        proc( -> bool ) Readb.dev,Eoi.dev ;
                        proc( -> real ) Readr.dev ;
                        proc( string -> string ) Read.name.dev )
```

where

each of the functions does the same as the PS-algol function of similar name.

Errors

This module is responsible for noting and reporting all compilation errors.

```
Error.gen = proc( -> error.pack )

structure error.pack( proc( string ) Inc.pos ;
                      proc( -> int ) No.errors ;
                      proc( -> bool ) Line.error ;
                      proc()Display.errors,Next.line,Reset.errors ;
                      proc( *string) Err.mess ;
                      proc( string) Err.mess1 )
```

where

Inc.pos increments the current line position with the input string.
No.errors reports the total number of errors since the last reset.
Line.error reports if there is an error on the current line.
Display.errors displays the current line and any errors on it.
Next.line instructs this module that the current input line has finished - this need only be called if Display.errors has not.
Reset.errors resets the module counters to zero.
Err.mess notes some error messages in the module.
Err.mess1 notes a single error message in the module.

Symbol Table

This module implements the symbol tables

```
structure symbol.table(     proc( pntr -> pntr ) Create.scope ;
                            proc( string,pntr ) Insert.entry ;
                            proc( string-> pntr ) Lookup.local,Lookup.rec ;
                            proc( -> pntr ) Enclosing.scope ;
                            proc( proc( string,pntr ) ) Scan.scope )
```

where

Create.scope returns a symbol.table
Insert.entry inserts an entry in the symbol table
Lookup.local looks up a name in the local symbol table
Lookup.local looks up a name recursively in the local symbol
Enclosing.scope returns the parent symbol table

Scan.scope applies the function to every entry in the table.

Lexical analysis

This module contains a lexical analyser generator.

```
structure lex.generator(    proc( *string,**string,pntr,pntr
                            -> proc( -> pntr ) ) Lex.gen )
```

```
Lex.gen = proc(     compiler.options
                    reserved.words.vector
                    input.pack
                    error.pack
                    -> Next.symbol )
```

lex.gen returns a next.symbol procedureof the following type:

```
next.symbol = proc( -> pntr )
```

and next.symbol returns one of these:

```
structure language.symbol( string the.symb )
structure literal.symbol( pntr the.literal )
structure identifier.symbol( string the.name )
```

Code planter

This module is responsible for putting code generated by the code generator into a file in the file system.

```
Planter = proc( filename,PS.code )
```

Syntax analysis

This module contains a syntax analyser generator.

```
Sa.gen = proc(  options,
                input.pack,
                error.pack
                PStypes,
                check.types,
                lex.generator
                -> proc( symbol.table -> PAIL or error.pack ) )
```

Type checking

This module does all the non-applicative type checking - that is type checking which may cause a type error to occur.

```
Type.gen = proc( error.pack,PStypes -> check.types )

structure check.types(  proc( pntr ) Bad.type ;
                        proc( pntr,pntr) Bad.types,Match ;
                        proc( pntr-> pntr) Int.real )
```

where

    Bad.type reports a type mis-match.
    Bad.types reports two types not matching.
    Match checks that two types are the same.
    Int.real checks that a type is a number.