The rationale for the language Pascaline

Version 0.3

Contents

1 Introduction....4

2 The Pascal standards....4

3 Why Pascal (as a base language)?....6

4 Why extend Pascal at all?....7

5 Issues with ISO 7185 Pascal....7

6 Methodolgies for extending ISO 7185 Pascal....9

6.1 Upward compatibility....9
6.2 Downward compatability....10

7 Design ideas in Pascaline....10

8 What happened to level 0 and level 1 Pascal?....11

9 The Pascal-P6 compiler as a proving system for Pascaline....11

10 The good and bad of Pascaline....12

11 Things not included in Pascaline....12

12 Language extentions for Pascaline....13

12.1 Word-symbols....13
12.2 Special symbols....13
12.3 Comments....14
12.4 Identifiers....14
12.5 Labels....15
12.6 Numeric constants....15
12.7 Constant expressions....16
12.8 Boolean integer operations....16
12.8.1 Why not negative numbers?....16
12.9 View and out parameters....17
12.10 Extended case statements....17
12.11 Variant record case ranges....18
12.12 Array type shorthand....18
12.13 Container arrays....18
12.14 Parameterized variables....21
12.15 Extended write/writeln statements....22
12.16 Extended read/readln statements....23
12.17 Type converters/restrictors....23
12.18 Fixed objects....25
12.19 Extended file procedures and functions....26

12.20 Added program header standard bindings....26
12.21 Redeclaration of forwarded procedures and functions....27
12.22 Anonymous function result....27
12.23 Extended function result....28
12.24 Overloading of procedures and functions....28
12.25 Operator overloads....31
12.25.1 Justification....31
12.25.2 Implementation in Pascaline....31
12.25.3 Effect on modularity and joins/uses....32
12.25.4 Operators not implemented....32
12.26 Static procedures and functions....33
12.27 Relaxation of declaration order....33
12.28 Exception handling....33
12.29 Assert procedure....34
12.30 Extended range types....34
12.31 Character limit determination....35
12.32 Matrix mathematics....35
12.33 Saturated math operators....37
12.33.1 The trouble with multiplication....37
12.34 Properties....37
12.35 Modularity....38
12.36 Definition vs. implementation modules....40
12.37 Overrides....40
12.38 Parallel modules....41
12.39 Monitor signaling....43
12.40 Monitor analysis....43
12.41 Channels....45
12.42 Classes....46
12.42.1 Static objects....46
12.42.2 Dynamic objects....47
12.42.3 Classes as parameters....49
12.42.4 Class parameters....49
12.43 Inheritence....49
12.44 Overrides for objects....51
12.45 Self referencing....51
12.46 Constructors and destructors....52
12.47 Operator overloads in classes....53
12.48 Derivation vs. composition....53
12.49 Parallel classes....53

1 Introduction

The Pascaline rationale is an accompanying document to “The language Pascaline”, the standard document for Pascaline.

This document serves several purposes:

1. An explanation of the design decisions made in the languge.
2. An annotation for the standard document.
3. Implementors notes.
4. As a place for opinions and background on the language.

The language Pascaline is served by a series of fundamental documents and programs, including:

1. The standard for the language Pascaline.
2. The rationale (this document).
3. The Pascal-P6 implementation.

These documents and programs provide the essential knowledge and implementation basis for the language itself.

2 The Pascal standards

The original Pascal standards were released from ETH Zurich by Niklaus Wirth and his group of students. This was documented by “The Programming Language Pascal”. It progressed through several versions, the last one being the 1973 version, which documents the last version to come from Wirth’s group. Often referred to as the “J&W” standard (Jensen and Wirth) standard, it was republished several times, but the language received its final form under Wirth in that year. Further major changes document the ISO 7185 forms of the language.

In 1983, the ISO 7185 standard was published. This documented the original J&W language as fixed into international standard form. Several changes were made, but in the main, they were to cover insecurities and ambiguities in the original language, some of which were intentional and some were not (we go into why an ambiguity might be desirable in this document later). The standard also featured the famous (or infamous) “0 and 1 levels” of the language, which was the first and only extention that the ISO 7185 standard added to J&W, that of conformant array parameters.

In 1990, the ISO 10206 standard was published. This document introduced several extentions to the ISO 7185 standard, and incidentally changed the status of conformant array parameters to no longer be optional.

In 1978, an implementation of the Pascal-P compiler/interpreter created at ETH Zurich was created called “UCSD Pascal”. This implementation subsetted the J&W language and added several new constructs, notably strings and file handling. Although obsolete today, it deserves mention because several newer languages were based on the extentions created there.

In 1983 an implementation of Pascal was created at Borland Corporation using many of the extentions defined in USCD Pascal, and adding many new extentions. This implementation was notable because it was quite popular.

The standards process for Pascal occurred with reasonable speed. It was also unusual in that the first international standard for Pascal was designed, for the most part, to fully document the language and remove implementation problems without changing the language. However, even in this there was not total success. There were those who believed that the language was useless without the "conformant arrays"feature, and those who didn't want to add extentions for the language to the first standard, no matter how essential. This was why the ISO 7185 standard included "0 and 1 levels", and why the ANSI version of the standard left level 1 out.

The success of the ISO 7185 standard is certainly up for debate. It certainly had an effect on existing J&W compilers, but not much on dialects of the language that had already branched off (UCSD, Borland). Perhaps the thing that could be said with the most certainty is that the mainframe and minicomputer compilers of that era quickly rallied around the standard, but that the popular microcomputer implementations of the language, rallied around UCSD.

The UCSD rooted implementations formed a dialect of Pascal (defined in both programming languages and natural languages as a branch from the original without mutual understanding). There never was, and perhaps never will be, an official standard of the UCSD nor Borland languages. Here it depends on your definition. J&W was not a committee standard, but had several other hallmarks of a standard, including:

1. Being addressed to outside implementors of the language.
2. Only including general features of the language, and not local implementation features.

It is worth noting in retrospect that the subsetting of the language that occurred for UCSD, which Borland largely copied, was based on the Pascal-P implementation from Wirth's Zurich group. It can be (and was) stated that this was never the intention of the Pascal-P porting kit to be used as a language in and of itself, but the very description of the Pascal-P subset as a "minimal implementation of Pascal designed to port the language" must have seemed (at that time) as a perfect description of a minimal, microcomputer based implementation of the language[1].

Both UCSD and Borland proceeded to add back several of the "missing" features. However, standard compliance is a string that, once broken, is not easy to glue back together. Many of the features of J&W/ISO 7185 Pascal that were found wanting and later replaced were implemented in a manner incompatible with the original language. I'll have more to say on this effect later.

As for the success or failure of the standards, the answer is perhaps simple. All standards eventually fail if your criteria is lack of use. Some last longer than others. The standard 19 inch relay rack and a standard phono plug are examples of 100 year plus standards.

Some would point to the number of users or implementations as a criteria. If I were to venture an opinion on the success of the standards, I would point to other language standards. For example the ANSI C standard has both had a profound and lasting effect on the language. Its introduction caused the many incompatible dialects of C to be brought around to the standard (or be eliminated), and it has reigned as the operational standard for the language for many years, decades in fact. It also shares the ISO 7185 characteristic of making changes to the original language that were required for clarity and elimination of ambiguity. Based on that high bar, ISO 7185 Pascal is a failure, but then so are most other language standards in existence.

[1] While it is certainly unfair to criticize decisions made in a time when computer memory and power were limited, the lesson here is "don't create a subset implementation if you don't want that to be permanent".

3 Why Pascal (as a base language)?

At this writing, the Pascal langage is more than 40 years old. What makes this language compelling over the many other computer programming languages created before or since?

There are several main features in Pascal that make it an interesting target:

1. The language is relatively simple.
2. It uses a very readable format with depreciated use of special characters.
3. It is very syntactically simple and clear.
4. It is self-coherient (the parts of the language fit well together).
5. It is well documented and (still) widely implemented.
6. It is type safe.

The simplicity of the language is a feature championed by Niklaus Wirth himself, and he believed and believes that it is more important to gather up what was learned since the last language to come up with a new language of equal simplicity (or even less) and greater power. Wirth practices what he preaches, and since the design of the language Pascal designed a new language about once every 5 to 10 years. Thus, simplicity is paramount over (backward) compatibility. He may well be right. It is certainly more fun to design a new language every 10 years than it is to rewrite all of your existing code that often.

The readability of Pascal is based on the idea, from studies on the subject, that humans are more able to read the written equivalents than special symbols. That is, “one” takes less effort to read than “1”. The exceptions from this are notable, including APL and C, which heads into the direction that a short, special symbol is preferable to the spelled out equivalent. The reason is also clear in that case. C is not particularly designed to be read, but rather to be written. It is the “write only language” of legend.

The simple syntax of Pascal has led it to be used in many compiler design classes. For that, Pascal exists as the alternative to C. Certainly it would be more efficient to design elementary compilers in C, but C isn’t simple to parse by design. Rather it is based around the idea of terseness, with the idea that the implementor can work harder to devine what the programmer is writing in shorter form. This is in keeping with the theme of C.

For self consistency, I think all you can say is that most languages start out fairly self consistent, and become far less so over time. This is both a problem and a warning for the future.

For documentation and implementation, this factor can be said to be behind all modern successful languages, as well as behind problems as well. C# is an excellent language but is not widely implemented outside of one major corporation. Its inability to make further headway is constrained by that fact.

Wirth both documented the language and released support for its implementation. This series of small steps did wonders to spread the popularity of the language. Java repeated this and got the same results.

Finally there is type safety. When Pascal was created, strong typing was the rule, not the exception. After the advent of C, “escaping the type system” has become the norm, and and so entrenched as a programming method that often the first and strongest complaint from C programmers who use a type secure language is that they cannot perform real work without type escapes.

Today there is a widespread return to type secure languages, often based on the perception that it takes more development and debugging work to make an equivalent program in C than other languages. This has led to adoption of alternatives, such as Java, C#, and unfortunately use of interpreters such as Perl and Python. Unfortunate because this creates the widespread perception that interpreted slow speed of execution is the price that must be paid for type security.

What happened with Java and C# was essentially a rediscovery of Pascal typing principles, unfortunately without due credit to Niklaus Wirth. The newly coined term of “managed pointers” is nothing more than a rehash of the concept created by Niklaus Wirth and Tony Hoare that pointers can be made substantially more safe if they are bound both to a type and also to one dynamic object.

Today type security is more important than ever. With more programs moving into realms of multithreading and multiprocessing, type security forms the foundation of a secure thread and multiprocessor communications methodology as well. This was outlined by Per Brinch Hansen as well, in the 1970’s.

In short, type security is not *a thing* in Pascal. It is ***the thing*** in Pascal.

4 Why extend Pascal at all?

Occasionally the idea comes up that Pascal needs to be kept “pure” in its original form. I’d like to dispense with this first. Pascal never had an unextended form. Wirth’s early reports on the language separated the external standard from the internal extentions of the language, the CDC specific features.

Beyond implementation specific features, there is also the need to address new technologies, such as multitasking and multithreading.

One of the most useful aspects of any language is its ability to grow and modernize as time goes on. To thow that ability out seems to me to be to throw out one of the most useful characteristics of a language.

Finally, a thoroughly addressing backward compatibility means that persons who wish to stay with the original standard and not use new features can do so.

5 Issues with ISO 7185 Pascal

Over the years there have been several complaints and issues pointed out with Pascal. Many of these have been addressed over the years. For example, in the paper “ambiguities and insecurities in Pascal” [J. Welsh, W. J. Sneeringer, C. A. R. Hoare], many problems with J&W Pascal were pointed out, and many of these issues were addressed by the ISO 7185 standard.

Certianly I am not of the opinion that Pascal was the perfect language. In fact you’ll find a complaint section on Pascaline itself here later. I’ll cover here several of my issues with ISO 7185.

First and foremost is the need to have all array sizes known at compile time. N. Wirth backed away from this strict rule in later languages Modula and Oberon. This limitation has been well covered by others (Habermann, Kernighan). The problem centers around the idea of having a subrange as the basis for the declaration of indecies in an array type. This means that different arrays are incompatible even if they contain the same base elements.

Having all types essentially fixed in structure may be good for the compiler writer, but it is not good for the programmer, who is forced to deal with several array types even if they have the same base type. Thus, handling strings of characters means having different types for each size of string needed, and presumably translations between those types where required. Further, ISO 7185 Pascal is not, in fact, a completely fixed type language, but features variant records and methods to deal with the fact that they can take different forms at runtime.

The I/O model in ISO 7185 Pascal has several problems. The most pressing of these, the batch orientation of Pascal I/O and the fact that it effectively requires “look ahead” when reading files, was dealt with by the “lazy I/O” system so effectively that it is not a problem.[2]

However, other issues remain. The I/O model has no means to name or specify permanent files outside of the header files, which are fixed in number and leave the job of filename specification to the operating system. Outside of header files, all files are anonymous and temporary. This system virtually mandates that the first implementation specific extentions to ISO 7185 Pascal be proper file handling procedures and functions.

The model used in ISO 7185 Pascal of bonding header file variables to externally specified files is uncomfortably close to the oddities of the file system in the CDC SCOPE operating system. While it is certainly not practical in many cases to transcend the model your programs must run in, such a computer and operating system specific limitation needed something better than to be written in to the language as a standard form.

The goto limitation for labels of “apparent value” numbers is an odd rule that seems to make sense only in terms of punishing goto users[3]. In ISO 7185 Pascal gotos serve an essential purpose of deep nested error bailout, which had no other good alternative (a fact completely missed by the UCSD family tree of implementations)[4]. Today structured exceptions subsume this functionality in a neat and complete form.

The concept of return values in functions is to my mind flawed. Using the name of the function as an assignment target is semantically “cute”, but then led to several behaviors in real implementations that I can only describe as “bad”. ISO 7185 describes the behavior of return values as extending to other functions than the one executing! That is, you can specify a result not for the function you are in, but the surrounding function by specifying it by name. Further, the function name can be assigned to anywhere in the function (or its subprocedures/subfunctions), and this can be done repetitively.

All of this encourages the idea that the function result is a variable[5], and discourages the idea that a function result is what is formed and returned to callers at the end of the function, in my mind a clear violation of the idea of single entry, single exit functions with the result formed at the end.

Pascal in ISO 7185 form not only lacks separate compilation of files, but actively discourages it by introducing a strict tree structured form. Although it gives tolken service to the essential idea of externally created routines with the **external** directive, this is an incomplete service mainly aimed at system libraries. Modula corrected this lack in the proper way, which is to say that separate compilation needs to be central to programming, not an afterthought.

ISO 7185 lacks any sort of runtime constant definition capability. The const statement defines unstructured constants at compile time, which can indirectly specify constants at runtime, but this capability is insufficient to create constant data tables at runtime that are needed for practical programming projects. The only alternative is to construct such tables from variables, which is not only less clear than a constant definition facility, but it wastes program space, since it implies that the

[2] Unless you are worried about the overhead it adds.
[3] Actually the other reason is that it prevents needing to look ahead to determine that an assignment is not being parsed. A numeric label is distinct from a symbol, which could be a variable.
[4] In fact it got “reimplemented” in Borland Pascal in an incompatible form modeled after the same feature in C (longjump).
[5] And indeed, the ISO 10206 standard makes it into a variable.

constants needed will both be represented in constant form in the output code, as well as needing to be copied to variables and then take variable space.

6 Methodolgies for extending ISO 7185 Pascal

The goals of Pascaline are clearly stated in the introduction:

1. To be completely upward and downward compatible with ISO 7185 standard Pascal.
2. To be a "logical extension" of original Pascal. That is, to extend Pascal using the same working theories and means as the original language, and poses no element that does not interoperate completely with the original language.
3. To provide a reasonable upgrade to the language capability, that can be implemented using using an existing standard compiler with minor effort compared to the original implementation of the compiler.
4. To implement only features that could be implemented efficiently using existing computing hardware.

You'll note that goal (4) is borrowed from J&W.

6.1 Upward compatibility

Upward compatability means that every program designed to the requirements of ISO 7185 Pascal will also function in Pascaline.

The first order of business in extending ISO 7185 Pascal is the subject of keywords. Niklaus Wirth choose the reserved word method for Pascal as an aid to creating a reliable compiler and unambiguous source language. Thus, word-symbols like **program** cannot be redefined in the program (pun intended). This not only forces source to be more readable, but allows the compilers error recovery to be significantly improved. Every word-symbol gives the compiler the chance to "resyncronize" to the program source text after an erroroneous construct.

The problem with extending the language by defining new word-symbols is that it potentially invalidates existing programs. It is not the only means to do that. Pascaline Annex L, "character escapes" neatly breaks existing programs by borrowing the C language concept of backslashed escapes:

```
'This is my \ht string'
```

The ISO 7185 Pascal standard seems to come down on both sides of the argument, defining in 3.2 that:

3.2 Extension

A modification to clause 6 of the requirements of this International Standard that does not invalidate any program complying with this International Standard, as defined by 5.2, except by prohibiting the use of one or more particular spellings of identifiers (see 6.1.2 and 6.1.3).

While at the same time defining "forward" and "external" as directives, not word-symbols. In any case, ISO 10206 clearly comes down on the side of extentions as implemented via new word-symbols by definining several of them.

Thus, with these two features, new word-symbols and character escapes, Pascaline is technically not upward compatible with ISO 7185 Pascal in the sense that it is possible to create a ISO 7185 Pascal program that does not compile or run under Pascaline. The ISO 7185 standard allows for that, however, it is a real issue that should be minimized.

6.2 Downward compatability

In rule (1), we say that Pascaline should be “upward and downward” compatible with ISO 7185 Pascal. What does “downward compatible” mean in this context? Downward compatible means that we don’t define new features that require other new features to also be used to gain access to them. The best example of this, taken from real world compilers, is the addition of file handling procedures and functions that require the use of a new string type to function:

```
program test;

var s: packed array [1..10] of char;
    f: text;

begin

   assign(f, s);
   ...

end.
```

If the new feature assign requires a special string type, and cannot take standard ISO 7185 Pascal string types, the result is a downward incompatability. The new features do not interoperate with the old ISO 7185 Pascal features. Such downward incompatabilities work against programmers, requiring that they adopt whole sets of new features just to use a few of the new features in the extended language.

Another example of backward compatibility in Pascaline are “container arrays”, which interoperate with, and extend the functionality of, ISO 7185 Pascal fixed arrays.

7 Design ideas in Pascaline

The design ideas in Pascaline have roots with various places and people. A lot of the ideas in Pascaline come from C and C++, which perhaps may be amusing to some. However, from a functional standpoint, there are a lot of similarities between ISO 7185 Pascal and C, and one goal for Pascaline was that it would be able to accomplish everything that C could do with equal ease, of course excluding type escapes. I have found that the (unfortunately) large group of programmers who learned to program with type escapes remain quite resistant to the idea that programming is possible without type escapes, despite considerable evidence to the contrary (C#, Java). Thus perhaps creating a lost generation of programmers in the way of former Fortran advocates.

Several ideas and constructs were taken from Pascal dialects, including UCSD and Borland. For the class/object design, C++, C# and Java were extensively consulted. For parallel tasking, the work of Per Brinch Hansen was consulted extensively. The late professor Hansen was very much ahead of his time in his work.

I great deal of the work in Pascaline comes from the writings of professor Henry Ledgard, who not only performed quite a bit of research into language design, but produced a considerable body of work regarding it. From Professor Ledgard comes the basic idea in Pascaline that the goal of an extensible language is to circularly explain the objects of the language itself. That is, if the type extensions are sufficient to explain how the built in types were created, then the language has achieved a level of extensibility that will allow it to cover the majority of new features needed in the language.

8 What happened to level 0 and level 1 Pascal?

Level 1 ISO 7185 Pascal was the addition of conformant arrays. It's a necessary feature, but it qualifies more as an extention than a clarification of the J&W language. The reason that is important is that Pascal is very much a language that was designed to have harmony of its features (a fact mentioned in J&W).

Conformant arrays extend the definition of a parameter for a procedure or function to include the bounds of the array. For example:

```
procedure p(a: array [lb..ub: integer] of integer);

begin

   ...

end;
```

Conformant arrays break the J&W idea that parameters only have simple type identifiers in their declaration (recall that parameters are not the same syntax as variable declarations). In addition, the type of the bounds must be declared ***within the type declaration of the array itself***. Further, the identifiers of the bounds are themselves variables in the scope of the procedure or function, abet strangely read only. In fact, they are the only instance of such read only parameters. Thus we have no less than four J&W symmetries broken in the Level 1 version of the ISO 7185 standard..

The different syntax required by conformant arrays was required by the nature of the "fix" (if you will). Declaring the parameter in the usual way (type-identifier) would have required that types be extended to allow variable length arrays, but limited to procedure and function declarations only.In short, conformant arrays clearly are a one time fix with regards to J&W Pascal, not a feature organized into the language.

There was considerable controversy over this feature. The result was that it was included in ISO 7185 as not being required. Further, in ISO 10206 Extended Pascal, the feature is not regularized, but rather replaced, by schemas, telling volumes about what the Extended Pascal committee thought of it. It was adopted forward into the ISO 10206 standard, but serves only to cover the fact that ISO 7185 Pascal (fixed) arrays are not normally compatible with their schema counterparts, and thus conformance remains valuable for handling such arrays.

In Pascaline, the rule is that the language must qualify as if it were a local extension of the language ISO 7185 Pascal according to the rules of ISO 7185 Pascal extentions (which ISO 10206 also complies with). Thus, conformat arrays were left off in deference to the container array system. Thus Pascaline qualifies as a Level 0 implementation of ISO 7185 Pascal.

9 The Pascal-P6 compiler as a proving system for Pascaline

One of the reasons for the early success of J&W Pascal was the freely available porting kit, Pascal-P. It remains freely available to this day. This was in the line of BCPL, and was novel in that time. The language Java reused this concept and advanced it further with the “byte machine” (a pseudomachine formatted into byte opcodes and operands).

On the other hand, Pascal-P never actually implemented J&W Pascal, but rather a subset, and this was by design. This was “corrected” with the advent of the ISO 7185 standard with two important advances:

1. A test of tests for the ISO 7185 standard.
2. A “model compiler” that implemented a full ISO 7185 compiler/interpreter.

At the time, 1982, this was a landmark development. However, both of these products were restricted and charged for. Whether because of this, or in spite of this, the model compiler and the test suite for ISO 7185 Pascal never had much traction outside of mainframe and minicomputers.

To correct this, however belatedly, the Pascal-P5 compiler/interpreter was created, with the following features:

1. Full ISO 7185 compatibility.
2. A full test suite.

Ie., the model compiler and test suite originally created for ISO 7185 were replaced by a fully open system based entirely on the Pascal-P series compiler/interpreters. The only loss was the model compilers unique implementation as a mixed source code/document. For that purpose, the original model compiler source is available (in book form).

In addition, the Pascal-P5 compiler was created as an increment to the Pascal-P4 codebase, which means that the literature and methodologies created around Pascal-P4 are carried forward to Pascal-P5.

Pascal-P5 has, I believe, and important place alongside the definition of the language:

1. It serves as a working definition of the language.
2. It can be used to create real-world compilers and interpreters.
3. It serves as a go/no go test of implementations of the language.
4. It gives an example of one possible way to implement language features.

For these reasons the Pascal-P6 compiler was developed alongside the Pascaline language. Pascal-P6 is the increment to Pascal-P5 that allows the Pascaline extensions to be implemented. Thus, Pascal-P6 serves as an important part of the full Pascaline definition.

Pascal-P6 serves as a test ground for Pascaline constructs, and proves the methods of Pascaline. The completion of Pascal-P6 also marks the completion of the language standard Pascaline.

10 The good and bad of Pascaline

11 Things not included in Pascaline

Break and continue

Default function arguments

12 Language extentions for Pascaline

Here we will go over each language extention in turn. The contents of this section deliberately match the numbering and order of section 6 in the Pascaline standard.

12.1 Word-symbols

The first extention Pascaline (or any other Pascal extention) is going to feature is a set of added keywords, 63 to be exact.

It has been an opinion that no extention to Pascal should be presented with a new keyword at all. This would mean that instead of defining a new keyword, you would define it as an identifier with a "special meaning". The ISO 7185 standard strongly hints at this with the classification of "forward" and "external" as "directives" and not keywords.

While noting that the ISO 10206 standard apparently has no issue with introducing new keywords to the language (leading perhaps to the idea that addition of keywords is a special priveledge for standards comitees only), it seems to me that this goes back to the basic issue of "what is a keyword".

There have, in fact, been several languages that haven't defined keywords at all (or more properly "reserved words"), leading to the ability of the user to use what would appear to be keywords as identifiers. This leads to fun looking constructs such as:

```
for for := to to do do <statement>;
```

which would be perfectly valid if "for", "to" and "do" could be identifiers. We don't do this in modern languages because, besides being confusing for the programmer/program reader, it also increases the difficulty for the compiler to recover from errors, since it has no distinct tolkens to look to in order to resyncronise the parse after such an error.

Based on this, Pascaline marches forward not only with new, distinct keywords where necessary, but also changes the definition of "forward" and "external" to official keywords. In ISO 1785 mode, a Pascaline compiler does not define them as keywords, but in Pascaline mode it certainly does.

With the definition of new keywords, there is a price. There is always a possibility that an existing program will use such a keyword as an identifier in an ISO 7185 program. This is the first and only modal dependency of Pascaline.

That is to say, taking an existing ISO 1785 program and compiling it as a Pascaline program, this is the only thing that would cause it to not compile.

12.2 Special symbols

There is not much to say here. The special symbols '(.', '.)', and '@' remain as optional in Pascaline for the simple reason that they were really never needed. Much like the alternatives in C, it was designed to cover older keyboards and systems with out '[', ']', and '^', characters, but in a world where it is hard to find a non-IBM-PC keyboard, this is rare.

Before the standard, I used '@' for the introduction to octal values in Pascal. Thus perhaps, the older programmers will recognize '$', '@', and '%', as the Motorola assembly language convention for radix specification characters[6].

[6] Mainly because in 1979 when I wrote that code, I was sure the Motorola 68000 would soon become the dominant CPU.

12.3 Comments

Comments gain the famous or infamous single line comments. I think single line comments were suspect in N. Wirth's world because they acknowleged the line structure of a program, which isn't supposed to apply to Pascal. This convention started with Algol, and was a counter to the Fortran convention of not only being line oriented but having certain collumns in the text with special meaning, a convention that dates back to cards.

So here is the line oriented comment, a retrograde step. But programmers like it. Why not copy the '//' convention as many other languages have? Well, that convention happened basically because C/C++ had used up all of the single special characters in the ASCII character set. That exclamation point was unused in ISO 7185 Pascal, and seemed to be a natural fit. The convention comes from assembly language where I used it for the comment character. It never made sense to me why ';' would introduce a comment, and in fact I used that character to separate multiple statements on a line, a case of copying Pascal conventions back to assembly languge.

Perhaps the only remaining concern is about the comment character being a less distinct tolken than a double character tolken.

12.4 Identifiers

With identifiers, we enter into a long list of common extentions that everyone implements in one form or another. The standard "break" or "separator" method used in the literature for Pascal was something like:

```
MyNewVariable
```

or capitalization of each word in a composite identifier. Pascaline adds the "_" character as an official label character for use as a break. It is "significant". This means that:

```
my_new_variable
```

and

```
mynewvariable
```

are NOT the same identifier.

The only remaining issue of interest here is that there is a convention, apparently starting with C, that identifiers starting with "_", like:

```
_dangerous_system_symbol
```

Have special meanings. This is apparently for the idea that when writing assembly language support functions, it is best to choose a name that won't conflict with an identifer in the compiled program.

I didn't feel the need to include that convention in Pascaline, but it could well be a local implementation issue.

12.5 Labels

Labels have always been an interesting issue in Pascal. I'm convinced that Niklaus Wirth included numeric labels in J&W Pascal to punish their use[7], but that's my opinion. In any case, Pascaline, like so many other Pascal implementations, allows standard identifiers to be used instead of the (rather odd) "apparent value" label numbers.

12.6 Numeric constants

Pascaline features single character prefixes for alternate radixes, with "$" for hexadecimal, "&" for octal, and "%" for binary.

Despite resemblances to other Pascals, the convention (which by the way predates may other Pascals including Turbo Pascal) was stolen blatantly from the conventions used in Motorolas' assemblers. The "@" used by motorola for octal was changed to "&" in Pascaline because the ISO 7185 standard appropriated[8] it for use with systems that (apparently) didn't have a "^" character. So here we have a perfectly good convention that was ruined by machines and terminals that most certainly no longer exist.

There are several basic ways to implement alternate radixes, and I'll list some of them here:

$1234   - Single character prefix

16#1234 - Radix prefix

0x1234 - C and many other languages

1234h   - Postfix, copied from assembler notation

There are two reasons I prefer the single character prefix. First, it is simple, both to parse, and to write. Second, it tells the compiler what radix is being used when the number starts. The use of "1234h" postfix formats requires that the number have its characters stored and reread when the actual radix is found at the end. I don't find this format amusing even for assemblers, and I have never written an assembler that way.

The C standard "0x1234" format is borrowed from many compilers and assemblers that existed before C did (I have seen the convention used back into the 1960s). The idea is that it will be recognised as a simple decimal, and will have led the compiler completely into parsing a decimal "0" when the parser can notice that it is followed by a postfix "x", and then switch to parsing hex. The convention is fairly inexplicable and uses two characters to do what a single character can, but perhaps having the advantage of not using up a precious special character that the language might use. Important to C, but not to Pascal.

The prefix convention "16#1234", used in many languages, including ISO 10206 Pascal, is a good one for being able to represent all radixes. It is, of course, verbose. My chief issue with it is that, in my entire carreer, I have never been asked to use base 32 in a program.

A final mention is about the binary ("%") convention in Pascaline. This is not often useful, but when it is, such as the creation of a bit mask in an embedded product, it is greatly appreciated.

[7] Funny, but not true. Non-numeric labels like "skip:" prefixing a statement mean that the Pascal compiler has to look ahead to the next symbol to distinguish it from an assignment or procedure call, which is more complex.
[8] The reader will hopefully forgive my use of the word "appropriate".

12.7 Constant expressions

The disadvantage of original Pascals' scheme of introducing the declarations in sucessive levels, i.e., constants to types to variables and procedure/functions, is that it discards the menmonic power of expressions[9]. I suppose the point was that such calculations were more clearly left to the code blocks, but the effect was to cause programmers to perform such setup calculations with a pocket calculator, then place them in the program, perhaps placing the founding calculation in a comment to preserve the original idea.

For such constants, there is almost universal agreement amongst Pascal implementors that such constant expressions should not simply be standard expressions from the language, but rather a special set of rules applying to constants only. This avoids the idea of what to do if the programmer places a variable, function or worse in such an expression.

12.8 Boolean integer operations

I wonder if anyone noticed in original Pascal that the language was not dependent on if it was implemented on a binary or a decimal machine. D. E. Knuth would have been proud of such a development, as in his famous series "the art of programming" he described the use of assembly language that would not care what the base of numbers in the machine were.

To appreciate why this is interesting, you have to appreciate that as late as the 1960's, there were still two kinds of computers in use, the decimal based ones and the binary based ones. Even if you considered binary to be the "winner", there were machines that supported both types of math, including the famous IBM 360.

So in Pascaline, there is support for direct binary math on integers. The standard does not get into the issue of what that means on a signed integer. I think this is just, for the reason that Pascaline also supports the idea of unsigned types, and therefore removes any great need to perform binary math with signed integers. If you want to perform binary operations on integers, then it is only fair to use only unsigned types to do it.

The standard specifies that these operations probally won't work on a decimal machine. Don't laugh, there is still a good possibility that the language may be used on computer history simulators.

Note that besides being useful for common operations with binary computers, boolean integer operations are essential operations in polynomial math such as used to form CRCs.

12.8.1 Why not negative numbers?

The Pascaline standard specifically rules out Boolean operations on negative numbers. The reason for this is that this would create dependencies on 2’s complement binary representation. There are many other representations that can be commonly used. I, for one, used signed magnitude and even offset 0 numbers frequently for various reasons.

The standard negative number used for Boolean integer math is -1, which in 2’s complement is a mask with all bits set. The Pascaline equivalent of this is maxint, which is all bits set except the sign bit.

[9] Ok, I could see why anyone would take exception to that phrase, “the memnmonic power of expressions”. What was meant was something like 1000+1 where 1000 is the size of an array, and 1000+1 is one past the end.

12.9 View and out parameters

There isn't much controversy here for **view** or “protected” parameters[10], which are widely implemented in other languages. Everyone implements such parameters. In fact, an early version of N. Wirths' Pascal report had them. The import of a protected parameter is that any expression that can be allowed with a value parameter can be used, but the compiler is able to eliminate the copy operation involved with passing structured types into a value parameter. In practice view parameters use pass by value for small objects and pass by variable reference for large ones.

The only issue is the choice of keyword, "view", vs. "protected" or anything else. My only excuse is that "view" seemed to me to be more in keeping with the series of a value, variable or view parameter. In other words, "I pass by value", "I pass by variable", then "I pass by view" made more sense to me than "I pass by protected". It just rhymed better. I have the value for the parameter, I have the variable, I have a view on the parameter. Plus "view" is a nice short keyword.

12.10 Extended case statements

Of all of the extentions in Pascaline, I find the case statement extentions most distasteful. The problem with "else" (or "otherwise") and case ranges is that they really remove the best property of case statements, that of being a clear simple to understand construct that compiles efficiently to an underlying machine level construct, namely a table lookup jump.

The extentions encourage, as I have often seen in code examples, the idea of a case statement as a "general purpose" filter for values. It certainly isn't. If there are a lot of gaps between the sequences of case labels, the compiler isn't going to give you a neat jump table in the output code, but is going to be forced to recode the problem in another way. And that "way" may not have the performance you expect from a case statement. In fact, the compiler can just degenerate the case statement to a series of if-then-else comparison statements internally.

Of course, this can happen with ISO 7185 Pascal as well:

```
case x of

   1: ...;
   1000000: ...;

end;
```

Is a fairly ridiculous case statement, but perfectly valid under the rules of ISO 7185 Pascal. The result is sure to be something you won't like, such as a case jump table of massive length, but only two entries occupied, degeneration to if-then-else, etc.

The issue people seem to have with ISO 7185 Pascals standard case statement is that it "takes too much code" to get around the fact that the default for a case statement is an error:

```
if (x < 1) or (x > 3) then ...
else case x of

   1: ...;
   2: ...;
   3: ...;
```

[10] Aside from the name.

```
end;
```

I must admit that I don't share the idea of single if statement to guard a case statement as a hardship. A bit more difficult is collecting all of the gaps in a series into an if conditional on a large but slightly sparse case table.

In any case, the "straw that broke the camels' back" was using Pascaline to interface to C code, which is pretty much a requirement these days. Trying to efficiently dispatch a deliberately sparse enum statement from a C code module using a Pascal case statement is an excersize in frustration, so I gave up and gave in to the "dark side".

The use of the "else" keyword (vs. "otherwise") was based on else being a short keyword that already existed in the language and already had the intended meaning ("else do this"). It didn't hurt that Borland Pascal also had this convention.

The idea of ranges was introduced in the CDC 6000 series Pascal compiler, and used by Nicklaus Wirth himself in Oberon. It seems reasonable, and solves the issue where you are having to count off enumerations redundantly.

12.11 Variant record case ranges

Variant record case ranges are a twin of case statement ranges, and were implemented in the CDC 6000 compiler. It's a fairly low cost improvement to useability and readability.

12.12 Array type shorthand

The array type shorthand really is part of container arrays, in the next section, but also stands on its own. It acknowledges that the most common array form is integer indexed with base 1. For this, only the length of the array needs to be specified. This is a standard feature in Oberon and many other languages, and I noticed that the standard form of it:

```
array 10, 10 of char;
```

Was recognizable as syntactically distinct because of the lack of surrounding brackets. In fact, because the brackets completely enclose the index specification in declaration (unlike the specification of an index in a factor), the two types are distinct. Thus, Oberon and Pascaline specifications dovetail.

The lead in to container arrays is that we introduce the idea of integer indecies as the default type in Pascaline. Thus, when the index does not appear, it is integer.

How nice is it to be able to specify:

```
packed array 100 of char;
```

As a string?

12.13 Container arrays

Container arrays were the first extention I created for Pascal. I started work on the idea back in the late 1980's, and they were implemented in simulation in 1993. Introducing arrays of length defined at runtime is a difficult subject, but creating a scheme that was backwards compatible with ISO 7185 Pascal seemed impossible.

The problem is that if you implement such arrays in the obvious manner, by defining the layout of an array as

- Length
- Data

Or

- Low bound
- High bound
- Data

This makes the use and passing of such arrays self consistent. That is, you can pass a pointer to such a structure, copy the whole thing as a view parameter, any use, and that will work within the context of such arrays. However, it will not be backward compatible with ISO 7185 fixed length arrays.

This leads to the following "solutions" to the problem:

1. Define fixed ISO 7185 Pascal arrays to be a separate type from the newer, runtime defined arrays.
2. Require all arrays, ISO 7185 fixed or the new runtime defined arrays to be in the same, prefixed format.

I belived then as well as now that both methods were simply unsatisfactory.

The method choosen for Pascaline, was, believe it or not, suggested by the C language. I noticed that, given the ability to allocate arrays of arbitrary length, nobody complained much that C required you to keep track of the array length yourself. In fact, this "do it yourself length tracking" is considered one of the flexible points of C. Since the language does not presume to know the length of arrays, you can keep track of it in whatever way you find efficient, from storing the length of the array, or recognizing the length from the data in it (zero terminated strings).

I also noticed that the model in C was actually that you could allocate a fixed number of array cells at runtime, but anything more was up to you. There are no so called "dynamic" arrays. If you wish the array to vary in geometry at runtime, you can implement the strategy to do that yourself, from allocating larger than needed and keeping track of the size within that "container", to reallocating the array for a new, larger size request[11].

This formed the basic idea of Pascaline arrays. The only difference with C is that we hold on to the type safety of the array concept. Thus, the compiler takes care of allocating and deallocating the array, knows what size it is, and is ready to reject attempts to access outside of the bounds of that array. The only concession the language makes is that you can find out what the compiler thinks is the maximum size of the array using the function (appropriately enough named) **max**.

The essential rules are that only the compiler can size the runtime array, and it only does it once. To create a new sized array, it has to be disposed of an reallocated. The program can never change its length. Pascaline only tracks the information it must know, the exact size of the array, and where it keeps that information is up to the implementation[12].

[11] I have noticed that in both C#, my Pascaline code, and perhaps others, reallocation seems to be the rule. This convention is clean and easy to implement, even if it does keep the storage allocator busy. It also removes the need to add a length tracking variable.

[12] The far and away winning implementation, used in IP Pascal and Pascal-P6, was "pick up" templates that can be created and carried with the base pointer for the array. This neatly solves both initialization and backward

This last restriction, that the compiler only knows exactly where the length information is kept, enabled the implementation of backward compatible container arrays. For assignment compatibility between container and ISO 7185 compatible arrays, the hidden length fields simply need to be copied. The assignment of a fixed to container array copies from a constant, the length of the fixed array, and the assignment of a container to fixed array verifies that the length is the same, then simply discards the length. Similarly, when an array is passed as a parameter, the length is kept in a compiler specified area, usually a side register, and thus both a pointer to the array and the length of the array are passed together.

This then is the basis for "container arrays", meaning that the compile time specification of an array only specifies a "container" for the actual array at runtime. The actual array can either be a fixed, ISO 7185 array, a dynamically allocated array, or another container array. There is no fixed array implied to be allocated to the array. It cannot be used to directly specify an array based variable.

When I first implemented container arrays, I wondered if the concept would be to restrictive for actual use. For example, you could only assign between ISO 7185 and container arrays if they were the same length. Especially since the first implementation only specified single dimension arrays. In practice, however, scheme proved universal. It could be used to create dynamic, runtime varying arrays, dynamically specified arrays of pointers to other arrays, etc. It actually makes perfect sense. Container arrays are a building block. They can be used to form any higher level scheme you wish to implement.

The basis of C for the concept of container arrays had another, side bonus. Since container arrays enabled C style runtime allocated arrays, Pascaline could do whatever C could do by definition. With complete type safety.

In the first container array implementation, the idea that **view** parameters could allow a non-variable operand to be passed as an array parameter allowed, for the first time, a construct such as:

```
printstring('this is my string');
```

Where the string could be any length.

In the first implementation of container arrays, value parameters could not be container arrays, nor could container arrays have multiple indexes. I have actually done quite a few programs with multiple level arrays using the old system of single level container arrays. They are simply built from single level container arrays of pointers to other arrays. Thus, multilevel dynamic arrays could be built in the same way as is done in C.

In Pascaline, both of those restrictions are removed. This, for all practical purposes, requires "templates" to appear in Pascaline, a concept I admit I had a problem with for years. templates are essentially a runtime vector of array element lengths by indexing level:

```
array [1..x, 1..y, 1..z] of integer;
```

Has the template:

- Length for major index
- Length for middle index
- Length for minor index

compatibility issues.

Such templates are built when the array is created. I the first implementation of N-level indexed container arrays, I merged this with single level container arrays by ultilizing “fall over templates”. A fall over template means that you carry a pointer to the template and “walk” the pointer down the template entries as you process indexing in the array. When the end is reached, you are at single level indexing, and you pick up the length and use that to replace the array pointer.

Thus, the template directed access “falls over” from the template address method to the single level length method at the end of the template. N-level arrays retain all of the slicablilty of ISO 7185 arrays, and don’t need to even use the template when they reach the last level, also incidently guaranteeing they are compatible with single level container arrays.

You’ll note that the container array system does not keep track of the lower and upper bound of arrays. This saves the compiler from having to track two numbers for each array, when it only needs to track one, the length. In fact, the length is all that concerns the compliler in any case[13]. The compiler needs that to check for out-of-bounds access. The rest is up to the programmer.

In practice, needing only a single length figure dramatically aids the efficientcy of the compiler for container arrays. By one half, in fact. Further, the major use of container arrays is to pass arbitrary length arrays into procedures or functions, and usually these routines don’t need to know the exact indecies, only how long the array is. Thus, container arrays are allways integer indexed, and always start with index 1.

Further, container arrays are compatible with any other array regardless of their index offsets (array index ranges not starting with 1). This again fits with the general idea of container arrays, which is to hold other arrays. This can admittedly create odd code when combining multiply declared indecies, but in most code is very consistent.

The net impact of container arrays in Pascaline is:

1. They are as efficient as possible, using only the minimum overhead (the length).
2. They are completely cross compatible with ISO 7185 fixed array types.
3. Because they are completely compatible with fixed arrays, they allow the programmer to use container arrays when required and fixed arrays when not, meaning the overhead can often be spent only where abosolutely necessary.

All of the other schemes seen for Pascal, including ISO 10206, leave ISO 7185 fixed arrays and runtime defined length arrays as incompatible types. Pascaline container arrays not only dovetail with ISO 7185 array types, but are a logical extention of them, and both complement them and take them to higher levels.

12.14 Parameterized variables

Parameterized variables are also new with Pascaline. Although performing a calculated expression in the midst of a declaration is certainly odd, parameterized variables give the ability to treat container arrays effectively as common variables. It is possible now to create variables that track the size of parameters or other variables[14], say if you need to copy the contents of an input container array to a routine to a temporary array, then copy the result back.

Parameterized variables introduce two new concepts in the “theme” of Pascaline. The first is that all variables are conceptually program objects with a contructor, destructor, and parameters that define their “geometry”. In the “everything is an object” school, even an integer has a constructor, destructor

[13] This is one of the biggest differences between Pascaline and ISO 10206 Pascal.
[14] But with only a single evaluation at the start of the procedure or function.

and potentially a parameter list on creation[15]. You just cannot see it, because it is system defined. A contructor for an integer might create it from dynamic storage, and initialize it to a know value. The destructor would release it[16].

The second is that each object has a "geometry" or basic configuration, which is defined on its creation, and holds true for the life of the object.

Besides their use for runtime specified objects, parameterized variables allow a style of programming that dispenses with the idea that the size of an array is part of its type. Thus, an alternative to the definition:

```
type x = array [1..10] of char;
var y: x;
```

is:

```
type x = array of char;
var y(10): x;
```

Thus the type definition simply defines the general form of the program object to be created, and the variable declaration itself fixes the exact geometry of the object. The next effect is exacty the same as the ISO 7185 declaration.

Parameterized variables introduce some inconvienent semantic complexities, besides the idea of instroducing live expressions into delcarations. The parameter cannot use uninitialized variables in its expression, and the only way that such variables could possibly be initialized is either for them to be parameters of the enclosing routine, or to be initialized by subroutines declared before the parameterized variable is.

In a compiler that cannot reorder code, the code to evaluate parameter exressions is logically performed by a series of callable short strips of code that execute before the enclosing block is executed, can be interspersed with procedures and functions, and must be done in order. In a compiler with code reordering, this could be merged with the enclosing block. This is a high price to pay, but this issue occurs again with objects.

The other consequence of parameterized variables is that they may require disposal of the variables at the end of the enclosing routine[17].

12.15 Extended write/writeln statements

Extending the field width parameters in Pascaline is one of the features most obviously imported from C. The alternative to negative field arguments was a special syntax to fields that would indicate special actions. However, the idea of negative fields is that they indicate which side of the field the data is to be placed, so the convention is fairly memnonic. Positive values start from the right and end to the left, and negative values start from the left and end to the right.

Other candidates for special write/writeln parameters were radix values and formatting image strings. Both of these were left up to the libraries[18]. Negative field specifications are a very low cost enhancement to the language. In fact, they usually don't even require compiler modification.

[15] Pascal-P6 systematizes this. Even built in types can be used as the base of a class.
[16] Container arrays have several similarities with classes, and can be considered to be classes themselves.
[17] But are typically efficiently implemented by allocation on the stack.
[18] However P6 uses the radix characters $, & and % at the ends of numbers to do this.

An earlier version of Pascaline used the field value 0 to indicate padded string input[19]. In practice, I found it equally useful to have 0 length fields mean, well, 0 length, as in print nothing. The classic example is:

```
write(‘ ’:n);
```

Where n is an amount of space padding you want, which could well end up being zero.

12.16 Extended read/readln statements

The ability to specify fields in read/readln statements means that files that are generated by Pascaline can be read by Pascaline with equal code effort. The idea of fielded reads is fairly old, and it appeared in the PUG newsletters. The chief problem with fielded reads is that they can "clip" input values, that is:

```
read(i: 5);
```

Where the input is:

```
1234567 numbers in a row
```

Fielded reads were not meant to automatically adjust for input fields, but rather to accommodate fixed format input as created by the equivalent fielded writes. This encourages the representation of data files as human readable character files without extensive parsing.

The result is a fully symmetrical set up read and write routines that have complementary actions to each other. Just as in C[20], where the idea was clearly taken from, such fixed field read and write routines are no substitute for proper parsing of input. The ability of read/readln to match constant strings to the input shows this. The input either matches the string, or is an error. Clearly the programmer can get trickly with exception handling, but the system is designed to match fixed format input.

For both the write/writeln and read/readln procedures, quite a bit of room was taken up in the standard to show equivalent code to the new Pascaline method. The message here is not only to show that the procedures have ISO 7185 equivalents, but to show that Pascaline is not doing anything here that the program could not perform for itself.

12.17 Type converters/restrictors

Generalized type converters were introduced to some Pascal implementations to the detriment of type security. ISO 7185 has a set of limited type convertion with the functions **ord**[21] and **char**, and perhaps **round** and **trunc**. Notably absent is a way to transfer from enumerated types from integers, but this is relatively easy to arrive at using a table:

[19] P6 uses the special field character * to indicate padded string output.

[20] Although as it turns out that the C standard library is not quite symmetrical and can't actually read fielded in all cases.

[21] In fact, earlier versions of Pascal-P used **ord** for defacto arbitrary type conversion.

```
program enum;

var e:  (one, two, three, four, five, six, seven, eight, nine, ten);
    et: array [e] of integer;
    i:  e;

begin

   for i := one to ten do et[i] := ord(i);
   ...
   i := et[5]

end.
```

This recalls one of the fundamental tenents of Pascaline, which is that an operation that can be carried out using ISO 7185 Pascal can be considered a "reasonable" operation (one that obeys the basic type safety and rules of ISO 7185 Pascal) and thus reasonable for inclusion in Pascaline.

The main reason for not including a opposite to ord for enumerations is that no one fixed function can specify the desired result type of such a function. This is "solved" by many Pascals, including Borland, by the use of type identifiers in an "apparent" function:

```
program enum;

var e:  (one, two, three, four, five, six, seven, eight, nine, ten);
    i:  e;

begin

   ...
   i := e(5)

end.
```

Pascaline offers another feature using the same syntax, but with a completely different purpose in mind:

```
program p;

type a = 0..10;
     b = 0..20;

var c: a;
    d: b;

begin

   c := a(d*2);

end.
```

This paradygme is unique to Pascaline, and shows a "type convertion" between apparently compatible types, **a** and **b**. Why?

The answer is not simple. When dealing with expressions involving different integer types with different value ranges, there has to be a rule resolving what range of values will be calculated to find the result. In the above example, if **d** is, say, 7, the result is clearly 14. But this is out of range for type a, which is in fact where the result is going, since that is the underlying type of **c**.

In the programming language C for example, the rules involve the ranges of the integers involved. Combining two integer types, one with a large range and one with a smaller range of values, say a **short** and a **long**, yields a calculation using **long int** range to hold the intermediate results.

In Pascal the rule is simple. All integers expressions are promoted to integer value ranges. Period. This means that on a processor capable of, say, byte value calculations would have to perform all caluculations at integer size to comply with the rules of ISO 7185 Pascal.

The Pascaline's transfer between two otherwise compatible types is termed a type "restrictor" rather than a type convertion (in C this would be referred to as a "type cast". It is not strictly correct. It could be used also to specify a larger than integer precision for a particular operation.

What is important to understand is that the only "type convertion" that Pascaline allows that was not also present in ISO 7185 Pascal is conversion ***to*** an enumerated from an integer. Also, Pascaline is not specifying the silent truncation of any values when a larger to smaller range restrictor is specified. In fact, a restrictor on current implementations usually generates a range check and an error if the value does not fit into the destination type.

12.18 Fixed objects

Fixed objects in Pascaline are essentially "structured constants". On many extended Pascals, there was an attempt to fit this concept into standard **const** declarations. I found the creation of a new type, instead of explaining why some constants can be treated as variables in some cases, and other cases not. In ISO 7185, declared constants exist only at compile time, and have no direct presense at runtime. **fixed** declarations occupy space at runtime, can be treated equally as variables, and can have a defined format by the rules of the target compiler and machine.

Its important to understand what fixed objects are not, which is initialized variables. There is no runtime calculation associated with them. They are determed entirely at compile time, and included in a read only section of the code. They cannot be treated as variables in any way, including being "threatened" in the sense of a ISO 7185 **for** loop variable threat.

What fixed objects do is provide structured constants using array or record structures of unstructured constants. No other structuring methods are allowed, such as files of constants. No pointers are allowed.

Two more additions to Pascaline that were considered were the ability to use value-constructors inside code blocks, and the ability to have constant pointers, and thus specify meshed constant data. Both of these were rejected on simple complexity grounds, bringing to mind a basic tenant of programming language design: you can always add a feature later, but you can never remove a feature later, without creating user and codebase issues.

12.19 Extended file procedures and functions

Extended file procedures are perhaps the most common extension to ISO 7185 Pascal. There are two file handling procedures in early versions Pascaline that were originally included, but with the later advent of container arrays, could have been moved off to a library, probally **services**. In fact, I seriously considered creating an abstract file type of the kind:

```
procedure length(var f: file);
```

That is, a file with no base type[22]. That would have been sufficient to explain all of the rest of the extended file procedures and functions being present in a library, since the implementation didn't necessarily have to use standard Pascal rules.

There are several "themes" to the extended file procedures and functions. The first is that the elements of a file carry a 1..n enumeration. This matched the general array theme in ISO 1785 of having indexes using the range 1..n (which is not enforced anywhere, it just happens to be the default string type).

Another theme that may not be readily apparent is that Pascaline, like ISO 7185 Pascal, strictly obeys the rule that files have distinct modes. A file can be in read mode or write mode, but not both. This is an important rule in operating system design, and it is suprising to me that so many implementations feature file modes that are read/write (at the same time). Declaring a file to be in read and write mode might be fun and convienent, but it means that such a file has to be locked to all readers, since it could be modified. Having distinct read and write modes means that the file can be shared by readers.

This "modality" of files was a bit hard to maintain, while also allowing mid-file updates. I choose a set of update procedures **update** and **append**, to handle updating starting at the beginning and end of the file, respectively. The second is only really required in the case of a text file.

Of course **update** is basically a **reset** that places the file in write mode without clearing out the file. The inconvienence remains of having to position back to where you were if you are doing a read-update model. I found that cleaner than trying to explain that if the file were already open, you could leave the position alone, but that if the file were undefined, it was reset to 1.

Wirth avoided dealing with the "dirtiness" of filenames in ISO 7185. However, I have found that the convention of allowing the same names as identifiers to be sufficiently universal. Basically, having things lke devices and network paths in a filename is a great way to create portable code that is still system dependent. Pascaline perhaps punts this issue by giving you the tools to deal with filenames in a system independent way, but not requiring you to do so.

For the convention of "bonding" a file to an external name, one convention out there absolutely won hand down. That was the Borland **assign** method. Using an **assign** procedure to bond a file to an external name fits the ISO 7185 standard perfectly. You can only assign to a file if it is in the undefined state, and all it does is change the file from an anonymous temporary file to a named, semipermanent file. It does not require **reset** nor **rewrite** to change the way they are formatted, or how they work. Further, the name **assign** says exactly what is occurring. The name is being bonded into the file. This is yet another place in Pascaline where I was happy and proud to steal from another language.

12.20 Added program header standard bindings

The addition of **error** and **list** files as standard header file types is taken from C, and **list** from many other early language implementations. error does in Pascaline exactly what it does in C, that is, is a feature to aid filter building, that is, using the **input** and **output** to build filter pipelines. The convention makes sense, I am all for building filters, and it matched well with the ISO 7185 **input** and **output** header file concepts.

list, on the other hand, does absolutely nothing on many languages now. typically, it is shuffled off to the actual printer port (yes, it still exists on virtually all modern computers), where it relies on if the printer takes old fashioned characters or not. In the main body of Pascaline, list has no special powers

[22] It has been discussed in conjuction with "common objects", that is a class/type tree that explains all types.

either. However, in the libraries, list comes back to being what it was originally intended to be. A simple and clear paradigm of how to operate a printer.

Unstated in the main body of Pascaline is the bonding of header parameters to external parameters. This was not because I don't have an opinion on the matter, I do. Header files that don't match the standard names should be bonded to a file specified by the user in a clear fashion, and annex M clearly states that. However, I didn't think such a semantic action needed to be justified in the main standard.

12.21 Redeclaration of forwarded procedures and functions

The ability to define forwards in ISO 7185 Pascal is a well designed feature and wears well with time. However, I found myself carrying the parameter list down to the actual declaration of the function or procedure and then commenting the header out:

```
function search(l: list): boolean;
…

function search { (l: list): boolean; }
```

Because I don't want to flip back and forth to the original definition to reference the parameter list and their names. This artificial duplication of the parameter list does not allow the compiler to check it, so it is easy to fall behind with changes to the source code.

So Pascaline allows the parameter list to be duplicated, but also checks if the parameter list matches in type and form ("congruent", in the language of the ISO 7185 Pascal standard). It does not check if the same names are used, so that can still fall behind if you don't update any name changes. However, it makes the parameter lists easier to create and keep current with simple copy and paste.

12.22 Anonymous function result

Back in section 5, Issues with ISO 7185 Pascal, I outlined why I have a problem with the way function results in ISO 7185 Pascal are done. So with anonymous function results, Pascaline voliates the tenents expressed in the Pascaline standard introduction, that is it redesigns and duplicates a feature in ISO 7185 Pascal. It is a required feature, in that it is needed to explain how operator overloads get their result, although I suppose there could have been an alternative method.

Anonymous function results give, to me, a natural conclusion to the rule that blocks should have one entry and one exit. A block that gives a result specifys the result of the block at the exit:

```
function maxval(a, b: integer): integer;

begin

   if b > a then a := b;

   result a

end;
```

Its really all about code readablilty. If the result is always at the end of the function, looking there tells you what is going to happen with the result immediately. If the function result does need to be scattered around the body of the function, creating a variable to do that is appropriate:

```
function maxval(a, b: integer): integer;

var r: integer;

begin

   r := a;
   if b > a then r := b;

   result r

end;
```

So you see I don't have a problem with the ISO 10206 feature that if a function result looks like a variable, it should ***be*** a variable. I just think it makes more sense for the programmer to do that. Creating a strange pseudovariable with special powers in the name of the function is not, in my opinion, the way to go.

12.23 Extended function result

The original idea of having function results occur with a simple unstructured value matches the convention in the C language, which makes that all the more strange that it draws so many complaints. The real reason that extended function results are in Pascaline is to enable operator overloads to be extended to include structured results. This, in turn, enables the creation of matrix and complex types, among others. Of course, matrix types made it into Pascaline as a standard feature, so I have taken the fun away of creating those, but complex types are still there for the taking.

One of the nice side benefits of extended function results is they rationalize function result assignments with all other assignments, instead of being a restricted subset of assignments.

12.24 Overloading of procedures and functions

Overloading essentially bonds a series of procedures and functions under one identifier. Although classified as polymorphisim, it could well be argued that this is "faux" polymorphisim because it is all at compile time. The runtime configuration of the program is not changed.

The clearest use of Overloading is to allow the programming of "type flexible" routines without burdening the callers with having to memorize the different names for the different type forms. To take the classiscal example of a potential "print" statement:

```
program p;

procedure print_integer(i: integer); begin ... end;
procedure print_string(var s: string); begin ... end;

begin

   ...

end.
```

vs.

```
program p;

procedure print(i: integer); begin ... end;
overload procedure print(var s: string); begin ... end;

begin

   ...

end.
```

In this case, the type selected overload procedure is clearer than the identifier selected procedure because what is being expressed is the action of the procedure, to print, not the type of its operands.

In fact, ISO 7185 Pascal uses overloads in the implementation layer, in **write**, **sqr** and similar routines that adjust their behavior according to operand type.

Overloads are yet another abusable construct and can be used to create code that is highly modal. This can obfuscate code by using the same named procedure perform very different actions, which makes it hard to read and follow the code. What convinced me that they were a valuable addition to Pascaline was that, given that Pascal is a type secure language, here is a technique, overloading, that allows a programmer to implement constructs that other languages have implemented in a type unsafe way. For example, in C effectively "overloading" pointers to index several different types. Thus, here is a way for Pascaline to add such capabilities and yet retain type safety.

The first defining decision for Pascaline with respect to overloads was the concept of ambiguity. For example:

```
program p;

procedure print(i: integer); begin ... end;
overload procedure print(r: real); begin ... end;

begin

   ...

end.
```

There are very valid reasons for wanting to implement print differently for integers and reals. However, the result is clearly ambiguous. Does the call:

```
print(1);
```

Refer to the integer or the real version of **print**? Clearly it is an integer, a real value would be:

```
print(1.0);
```

However, attempting to classify the procedures this way would clearly violate the idea of value parameters, since integer 1 is assignment compatible with the parameters of both procedures in the print overload group.

A common way I have seen to handle ambiguity is simply to accept it. The compiler has “priority” rules about which form of the overload it will apply in which order. The result is that the programmer must be very familiar with the rules with which overloads get resolved, and the idea of finding out what, exactly, the program does is more complex.

Instead, Pascaline uses a series of simple rules to disallow ambiguity in overloads. Since ambiguous overload groups are rejected at compile time, there is never any need to find out how a particular overload will be resolved. It is always clear.

This capability seems like a problem at first (“you mean I cannot treat reals and integers with separate code”?), but I have found it less of a problem in real programs than expected.

The more complex rule in Pascaline was “convergence” of parameter lists. The reason this rule exists is because if two apparently compatible overloads exist when examining their parameters from left to right sourcewise, the parameters in a simple compiler (one that does not create an internal representation of the program) cannot be processed to final code if their modes differ. So the overloads:

```
program p;

procedure doit(var i: integer; b: char); begin ... end;
overload procedure doit(i: integer; b: bolean); begin ... end;

begin

   ...

end.
```

Cannot be reasonably processed because the code generated for the first parameter is different in the **var** parameter case vs. the value case. The convergence rule allows a simple compiler to generate code for parameter lists, then find the “defining parameter” to tell it what runtime overload to call with the parameter list as already constructed.

This rule admittedly is complex. In actual use, it is less of a barrier to writing code that it might seem.

Why is the overload word-symbol required in Pascaline? It is certainly possible just to assume that a duplicate function is a defacto overload (a la C++). This would have created a situation where the language was modal, and in ISO 7185 Pascal mode it would be an error, vs. Pascaline where it is not. It is reasonable for the programmer to declare intention here. The overload word-symbol effectively declares “this is not an error, it is deliberate”, and the behavior of ISO 7185 Pascal is brought forward to Pascaline.

12.25 Operator overloads

12.25.1 Justification

Operator overloads advance the idea of Pascaline being circularly defined. That is, each of the working principles of the language can be extended by the user.

Operator overloads have generated considerable, and in this author's mind justified, criticisim. The listed reasons are:

1. They hide a possibly very complex function behind a very simple operator.
2. They possibly redefine a system feature with expected standard meaning, and do so with little warning, being effectively a side effect of using or joining a module.

It is reasonable to ask if they are worth the issues. I included operator overloads in Pascaline Because:

1. They complete the idea that any power the complier has to implement types is extended to the user.
2. They allow user, and thus modular, implementation of features that would normally require a built in implementation.

For example the module "strings" would not be able to be implemented seamlessly without operator overloads.

12.25.2 Implementation in Pascaline

The rules of operator overloads are the same as function and procedure overloads, that the overload groups formed must not be ambiguous. The difference with operator overloads is that this includes system defined operators as well. So operator overloads include a power that function and procedure overloads do not, which is the ability to overload system definitions, since all operators are predefined in the system layer.

The first best use of operator overloads is extension of math operations to new types. Examples include matrix math (although Pascaline includes that elsewhere), and complex arithmetic.

The rules for operator overloads allow them to "overlay" operator definitions in outter layers, including the system layer. Procedure/function overloads allow outter layer definitions to be "obscured", as in made unavailable, but operator overload defintions only do that if the types of parameters are the same as an outter layer definition. This is an unfortunate rule, but it must be so. The best example of this is the assignment operator overload:

```
operator :=(out a: integer; b: integer);
```

There is essentially no limit, in the system layer, to the types of an assign, although the types may mismatch. The rules of overloads don't consider mismatched operands to be qualifying. Thus for a typical, and very valid, assignment operator overload:

```
operator :=(out a: pstring; b: pstring);
```

The assignment:

```
var a, b: pstring;
a := b;
```

The assignment has meaning both at the system level (pointer a gets pointer b), and at the string level (string a gets a copy of string b). Both levels can be applicable. The definition of the operator overload may well use the system definition internally. For this we allow:

```
operator :=(out a: pstring; b: string);
begin
   a inherited := b;
```

```
end;
```

Which means to promote the operator ":=" to its outter level definition.

12.25.3 Effect on modularity and joins/uses

Because operator overloads effectively evade name scoping requirements and specifically qualidents, they can be used to construct modules that, if joined, change the meaning of operators in the joining module without use of qualident. This means the programmer joining that module may get the operator redefinition without knowing it. On a used module, this kind of behavior is expected, since a used module is allowed to interact with the module that uses it. Indeed, that is the definition of a system oriented module, you expect that it can modify the behavior of your program intimately. With joined modules the user has a reasonable expectation that this is not so.

The solution to this is is to require that one or more of the types of an operator overload parameter be a named type of the joined module. An example would be:

```
type complex = record r, i: integer end;

operator +(a, b: complex): complex;
```

This means the operator overload in the module joined will not be invoked unless one of the named types of the joined module is referenced. This effectively ties the operator into use of a qualident for the joined module.

Note that this rule is only enforced on a **joins** module. Thus a **uses** module retains the power to take over common operators.

12.25.4 Operators not implemented

The "operators" [] (array index), . (record element offset) and ^ (pointer dereferencing) were not implemented because they are all variable constructors. Variable constructors always yield addresses of operands. This is required since they can be used on the left side of an assignment.

In Pascal, there is no ability to return the address of a general operand, which would introduce the ability in Pascal to arbitrarily address program objects, al la C. This is specifically avoided in Pascal (and in other languages, such as Java, C# and Go). Thus there is no operator overload function that could yield such an address.

Implementing such operator overloads would have changed the meaning of a variable reference to just the variable itself, and thus terminate the variable reference. Thus such operators would have to have very special rules regarding their use. I deemed this to be non-orthogonal with the rest of Pascal.

Thus these three "operators" were left out of the possible operator overloads.

12.26 Static procedures and functions

The **static** attribute could well be called the "8031/8051 attribute". There exist processors that make the creation of local procedure variables extrodinarily difficult, and indeed, even C language implementations on such processors commonly include extentions to address this.

The compiler can sometimes determine if a procedure or function can be **static**, that is,never recursively call itself. In order for this to be true, it cannot be global in a module or call other modules, because it is not possible, by the rules of separate compilation, to determine if the other module will call back to the present module and recurse the function or procedure.

For this, the static attribute is a low cost extension for such processors. The programmer knows if the function or procedure will only be used statically, and this attribute can be ignored on processors for which it has no net effect.

12.27 Relaxation of declaration order

The declaration order requirement of ISO 7185 Pascal was designed to enforce the strict “declare before use” ordering of the language. It does not accomplish that completely, since it is still possible to create circular references with respect to pointers. In Pascaline, there are more opportunities to create circular references by using constant expressions. Also the modular format occasionally requires that two declaration sections exist.

It’s a relatively simple change to the language. It does not change the rule that program objects must be declared before use.

12.28 Exception handling

ISO 7185 Pascal always had the concept of deep nested error aborts, that is, the ability to bail out of any depth of routine call with an exceptional condition. This is a practical requirement for any non-trivial program. For example, a compiler has to be able to bail out of any parsing routine and resume parsing the rest of the program on error.

UCSD was famous for removing this ability, just as Borland was famous for removing it, and then putting it back with an incompatible (to ISO 7185) syntax later. There are even ISO 7185 compatible compilers that implement it, but not across separately compiled modules, which voliates the fundamental rules of Pascaline modularity (see 12.35”Modularity”). Likely this can be attributed to the inherent difficulty of implementing a deep nested bailout. It didn’t appear in the Pascal-P series either (until P5).

What exception handling does is to establish at runtime what deep nested **goto**s did at compile time, and thus simplify and regularize the concept. When a program executes a **try** statement, it places sufficient information to recover from an exception on the stack, and the exception management system appears as a series of “frames” on the stack. When an exception is thrown, the program conceptually looks at each exception frame in turn and determines which one has declared it can handle the problem that caused the exception.

The exception handling mechanisim has a number of beneficial effects on Pascaline. It eliminates the need to circularly reference modules, a practice frowned upon but impossible to completely eliminate, since deep nested gotos between modules basically require it. It divorces the routines that throw exceptions from any need to know about the code that handles the exception, which makes exception handling a perfect match for a modular system. Finally it explains how system errors work. All of the system implementation modules establish “ultimate” exception handlers and then allow the program to catch exceptions as needed. The system implementation module itself will ultimately handle the exeception if the program does not. If the error is not suitable for recovery, then the error is simply “unrecoverable”, and is never thrown as an exception.

12.29 Assert procedure

The **assert** procedure became famous as a macro based add-on to the C language. It forms a simple and clear means of adding error checking during program development, which is to be encouraged. As such, it was a natural evolution for Pascaline.

12.30 Extended range types

Extended range types first appeared in UCSD Pascal. The idea was that, if **integer** represented the most efficient type for the target processor, typically the size of a register, that declaring a type range outside the limits of –**maxint**..**maxint** would signify that the compiler would generate a type that was not as efficient, but could be used to carry greater than normal precision. Most typically, this is done in the target processor by implementing double precision math, where the target uses to words at a time and performs common operations with those.

In several dialects of Pascal, this was done with the addition of predefined alternatives to **integer**. This is not substantially different in Pascaline, except that the predefined **linteger**, **cardinal**, and **lcardinal** types are predefined instances of extended range types, instead of the converse. The difference is that it is certainly possible for:

```
type x = 0..maxlint*2;
```

To be a valid declaration. In fact, there is nothing preventing a Pascaline implementation from implementing integers that have any given length of precision, as has been done in other Pascals (for example Pascal-XSC). The view in Pascaline is that integer ranges are essentially infinite, and the predefined types integer, linteger, cardinal and lcardinal are just convient placeholders on that scale.

The import of predefined overrange types is that it gives a common definition to two different features of many existing CPUs:

1. Many CPUs can operate on unsigned types.
2. Implementing double precision math is fairly easy, and supported in the instruction set (by features such as unsigned multiply).

The cardinal type gives access to unsigned arithmethic. lcardinal and linteger give types that are “more work than integer, but not inordinately so”. That is a perfect description of double precision arithmetic.

Finally, Pascaline gives the option that implementations can simply ignore overange arithmetic and implement all arithmetic as integer. The rule remains just as in ISO 7185 Pascal: if you know the range of the type you are going to use, declare it so. Leave it to the compiler do determine if it can comply with the requested type.

12.31 Character limit determination

There has been a tendancy in Pascal dialects to add constants for limits of all kinds to the language. Thus, minimum integer gets a limit (not just maxmum), as well as reals, etc. I didn’t see a great need for a list of limits to Pascal. How many programs will actually use a limit to real number math, and what would you use it for?

However, there have been many times when the lack of a clearly defined maximum character code was a problem. The chief difficulty with such a limit is that, on existing character sets, that value is not a printable character.

The remaining issue is what is the minimum character. maxint does not have a minimum, integers are defined to be in the range -maxint..maxint. This minimum was choosen, I suspect, for the signed magnitude representations of the CDC 6600 computer. However, it works for 2's complement notation as well since the most negative number ($8000_0000) is often invalid.

Fortunately, character sets are never negative, thus in all existing sets the value chr(0) is a valid, although usually unprintable, character. This is true of both 8 bit character sets as well as Unicode 16 and 32 bit character sets. Its even true of EBCDIC.

12.32 Matrix mathematics

There is certainly a case that this feature duplicates other features unnecessarily. Overloading of operators allowed matrix mathematics to be implemented as a library. However, there are so many compelling hardware direct implementations of vector and matrix mathematics that this is an important inclusion in the base language. This gives the compiler the opportunity to implement vector and matrix operations as efficient vector operations in the target computer.

The implementation of vector/matrix operations in Pascaline is along what I will call "computable" lines. Vectors and matricies are treated as a series computable unit as are integers, reals and sets. This means that:

var v(10): vector;

or a vector of 10 integers can be computed multiple times and carried from operator to operator in temporary form:

```
program test;

var a(10), b(10), c(10): vector;

begin

   a := b+c+1

end.
```

Means to set a to the vector sum of vector b and c, and the result of that will have 1 added to each element. This means that the intermediate value of the vector result of b+c will be carried as a temporary result to the operation +1, and then again to its destination a.

A Pascaline implementation does this by creating temps for each expression element, usually on a stack. The meaning of this is that the vector will be converted to standard form while on that stack, which is:

```
type vector = array of integer;
```

Pascal rightly has the rules that arrays (and matricies) are not convertible to different forms, thus:

```
type svector = array of 0..10;
```

is not compatible with type vector. This can be an implementation issue if, say, the target machine has hardware operations on byte vector values as well as integers.

Vectors and matricies are form equivalent in addition and subtraction, meaning that there is no implementation difference between:

```
var v(100): vector;
```
and

```
var m(10, 10): matrix;
```

Because each cell of the vector or matrix is treated individually. This is not true of matrix multiplication.

Finally, Pascaline features both real and integer vectors and matricies, and thus, for consistency, is able to perform:

```
program test;

var a(10): rvector;
    b(10): vector;

begin

   a := a+b

end.
```

Meaning that **b** is expected to be converted to an rvector before the addition. This is required to stay consistent with the rules of Pascal. However (as noted above):

```
program test;

var a(10): rvector;
    b(10): array of 1..10;

begin

   a := a+b

end.
```

Because **b** is not considered a vector, that is, operable by built in vector operations. To be fair, the users could well define vector operations for a such a vector themselves. They just won't be converted to native machine operations.

12.33 Saturated math operators

Saturated math operations originated with DSPs or Digital Signal processors. Such processors perform math on real world analog values, and often as arrays of values. For such values the standard processors handling of overflows and underflows of integer values is insufficient. The common example of this is a pixel screen display, which treats the display as an X-Y matrix of integer values of the intensity of each pixel, or even the intensity of a color value in each pixel.

To manipulate such values, say adding one to the entire picture to brighten it, or subtract one to dim it, there is no meaning for overflow (values greater than MAXINT) or underflow (values less than -MAXINT). Instead, the correct operation is to leave the value at its maximum or minimum value.

Saturated math operators are similar to the operators for vectors and matricies. In fact, saturated math and vectors or matricies go hand in hand, since analog values are often processed as arrays, matricies or groups of values.

12.33.1 The trouble with multiplication

One of the issues in Pascaline is that the definition of matrix multiply does not match the definition of matrix multiply for saturated math. In the matrix multiply operation, the rules of mathematical matrix multiplication are used, vs. in saturated math the elements of the matrix are treated as a series of individual values. This works because in saturated math context, such as the above mentioned array of pixels, treating the values as individual makes sense, and treating the matrix as a matrix in the rules of mathematics is not. I'll here call the mathematics definition of array multiplication "symmetric" and the individual value model "asymmetric".

Some languages use different operators for symmetric vs. asymmetric mathematics. I didn't want to implement this in Pascaline simply because this was "language symmetry" over function. Asymmetric matrix math has no use when multiplying using common integers, just as symmetric matrix math has no real use in saturated math operations. So unfortunately the dichotomy exists in Pascaline.

12.34 Properties

Procedure, function and operator overloads extend Pascaline for entire classes of variables according to type. Properties finish the extention paradigm by allowing the behavior of a variable to be completely specified for each variable.

Properties allow a new programming paradigm we will refer to as "property based programming". In procedural programming, procedures and functions are called to operate on data. In object based programming, methods are part of the data and operate specifically on that data. In property based programming, the read and write of data is the central paradigm, and any actions to be performed are a side effect or performing those reads and writes.

Although properties are one more tool in the programming toolchest, they have important applications in multitasking and multiprocessing in conjunction with channels and liasons, as will be covered later. Between tasks, processes or even separate CPUs, read and write becomes the central paradigm of the communications channel between the tasks, processes or separate CPUs. Procedures, functions and methods are unnatural to such communications channels and are usually broken down into read and write operations (which forms the basis of RPC or Remote Procedure Calls). Properties give a more natural construct to separate task, process and CPU communication as "actions resulting as a consequence" of channel reads and writes.

12.35 Modularity

Along with type security, modularity was a very important addition to Pascal dialects, and an underappreciated one. Modularity, as opposed to simple source inclusion (as in the C language), extends type security across separately compiled modular lines. Further, by specifically declaring what the module use order between modules as part of the module format, modules are self explanatory to the point where so called "automake" facilities can automatically form a tree of program dependencies from the source. This both helps program management, and also greatly aids the readablility of the program.

The primary goal in Pascaline was that the modular format be divisible with little consequence. That is, any existing ISO 7185 Pascal ("unmodular") program could be reorganized as a modular program with little effort and change to the existing program. It is this goal that shapes many of the characteristics of a module. Modules export their declarations by default, and everything in a module can be exported save one object, goto labels.

The module system in Pascaline is self consistent. Programs blocks are considered modules, and they export their definitions by default. Pascaline contains just one information hiding mechanisim, the **private** statement. private effectively divides the declarations into two sections, one public and one private. This means that the private section must follow the public section. There is no statement to restore the private section back to public again, and this is no accident. Although Pascaline supports information hiding, it does not support the idea of having public definitions that are built on private definitions for anything but procedures and functions. There are no hidden structures in public definitions that are not known.

The other advantage to having the private word-symbol delimit the end of the public section is that when the compiler reads definitions to be included in another module, the appearance of private ends the exportable definitions. There is no need to continue reading the source.

Another defining characteristic of Pascaline is that exportable objects cannot be individually qualified. They cannot be individually specified with a special symbol as exportable. They cannot be write protected on export. They cannot have their name changed or "aliased" to another name. Again, this is policy not accident. These abilities have been used in other languages to create a spaghetti of cross references between modules.

Of course, all of these features common in other languages and dialects are designed to address name collisions between modules, that is, importing a name along with a series of names that collides with one defined in the importing module.

The "normal" mode of importing a module is the **uses** statement, which was taken from the UCSD family. It means that all the definitions of a module will be exported, even if you may not want some of them imported. uses is a natural use of system defined modules such as the ones presented in the annexes of Pascaline. The names used in those modules are distinct and should not appear in the using module in any other way excepting their use or "shadowing" with other definitions. Such modules are a parallel to the "system" block that logically encloses a ISO 1785 Pascal program (see J&W).

However, Pascaline programs will use the modular concept to the point that name collisions are a real issue. Thus, another mode of import is possible, joins. The joins statement places the joined module "to the side" of the importer, but does not merge their name catalog. Pascaline uses the Oberon concept of a "qualident" or "qualified" identifier:

```
module.name
```

To distinctly specify the name to be used. Qualidents echo the same idea that exists in records and classes.

Joined modules and qualidents solve the problems that features like individually specifying exports, aliases and similar featuers were designed to solve, but in a cleaner and more structured way.

Earlier I mentioned that goto labels cannot be exported, but there is a way around this. An external module can reference a procedure or function that then executes the goto for it:

```
program p
```

```
label 99;

procedure abort; begin goto 99 end;

begin

   …
   99:

end.
```

Because **abort** is public, it can be referenced by an external module. This so effectively implements intermodule gotos that it can be used in all circumstances where a direct goto would have been used. The reason that intermodule gotos were not directly implemented in Pascaline is that only the defining module has easy access to the stack depth information needed to execute the goto in order to "cut" the stack frame and remove all stack frames below the target of the goto label. Calling a local routine places the flow of control of the runtime back into the module were that information exists.

Of course the disadvantage of such intermodule gotos is that it inherently creates module definition loops. This is because the most common use of intermodule gotos is to bail out of nested procedures and functions to a higher level module. This means that the lower level module must know of the exports from the higher level module (the routine that executes the goto).

There are various fixes for this issue, including declaring a "dummy" empty module to define the higher level export that can be included by the lower level module without creating a loop. But the exception processing system in Pascaline effectively solves this issue in a very permanent and clean way, and is to be preferred going forward.

The modular system defines a new block besides the main one, the "deinitializer" block. The main block, corresponding to the program block, is the "initializer" block. This is required because modules have no sense of permanent thread of execution (although there are exceptions). The initializer is executed when the module starts, and the deinitializer executes when it ends. Either the deinitializer or both the initializer and deinitializer can be left off. The deinitializer of a program module is always logically empty.

Although Pascaline does not specify the order that initializers and deintializers are executed, the system clearly lends itself to a herycarcal tree format in which lower level modules appear in the "stacking order" before the modules that they service. In the original implementation of Pascaline, a module executes its initializer, then performs a call to the next module physically appearing after it in the executable binary. When the call returns, the deinitializer is performed, then the module returns to its caller, presumably the last module in the chain. The system is clear, supports tree structured module organization and does not require a table of modules requiring initialization and/or deinitialization, nor any statement of "priority" of their calling as in C language "pragmas". The order is dictated by the stacking order when the binary is constructed.

It is possible for a Pascaline compiler to inforce the order of calls within modules. Specifically, calls to routines in a module can be trapped as errors if they are called before their initializer. This is more difficult to enforce with other exported declarations such as variables. However, careful design of module stacking order can prevent this issue completely.

Pascaline defines the **program** block as just another module that does not have a deinitializer. In Pascaline theory, modules are structures that form linkages at the "sides". The meaning of this is as opposed to the strict tree structure of ISO 7185 Pascal blocks. Each such block interfaces with other blocks only via the "top" and "bottom". The top is the list of parameters in the procedure or function. The bottom is the ability of any enclosed block to access the declarations of its parent.

Modules (and later classes) can interface with other modules via the side. The definitions in a module can be shared with other modules placed side by side. The tree structure of modules is really up to the user, since modules that are side by side and share mutual definition are inherently circularly referenced.

12.36 Definition vs. implementation modules

Modules in Pascaline represent themselves in intefaces. That is, there is no separate special "interface" module that serves to represent the module to other modules. The Pascaline processor can either directly read the source code for the module or consult a "digested" version of it (typically in the form of an intermediate file). The latter is just for speed. The effect of importing definitions from an external module is specified to be equal to the result of directly reading the source for the module.

Having modules represent themselves supports the idea of easy division of a program or module into submodules. Much less work is required to partition code into modules. Further, there is no need to separately maintain an interface and a content module, and no issues resulting from not keeping these two parallel modules up to date with each other.

The remaining complaint is that having a module represent itself is "unsafe" because it also contains the source code. Indeed many times the need arises to ship an interface module with a binary object file as a library. This is done by "stripping" the source module so that the definitions are empty. This can be done manually or by an automatic tool.

In fact, all of these advantages were detailed by Niklaus Wirth with respect to the language Oberon.

12.37 Overrides

An override procedure or function allows a newly defined module to replace the old definition in an existing module for all calls to that module, even from lower level modules. That is a simple description of an override, but in practice the consequences of overrides are quite complex.

Overrides allow the functions or procedures in a module to be extended to cover new capabilities. It is fundamental to the nature of the Pascaline libraries. The write/ln and read/ln statements, which start as simple serial output routines, are overridden to output to graphical output surfaces, or to communicate with Ethernet sockets, or any new functionality that might arise in a new operating system paradigm.

When a procedure or function is overridden, all further references to the original module by later modules are redirected to the new definition. These are called "down calls". However, overrides also affect modules that appear ***earlier*** in the stacking sequence than the overriding module. This seems imprudent, but this is how it must be. When the write/ln and read/ln calls of Pascal are overridden, all of the lower level libraries that also use basic Pascaline I/O must be redirected as well. This is why if the I/O is redirected, say to a network socket, all of the I/O statements in the **string** module also redirected.

The call of a lower level module to a higher level module is called an "up call". It seems counterintuitive. How can a low level module call a higher level module? The lower level module knows nothing of the higher level module. The reason it works is because the higher level module as called has its interface completely defined in terms of the lower level procedure or function. Anything additional the higher level module “knows” how to do is given in the context of that higher level module only.

Pascaline, unlike other dialects, does not allow the mixing of overrides and overloads. An overload of an override only has meaning at higher levels than the overriding module. It cannot redefine calls at a lower level. An overloaded procedure or function is a group of related procedures and functions selected by type. To override an overload, which of the overload group is to be overridden must also be selected by type, meaning that the overload group can have parts that are overridden, and parts that are not.

Overrides of overloads were excluded to keep the complexity of the procedure and function calls mechanisim reasonable, although it is already fairly complex in Pascaline.

Overriders nest in a natural way. Newer overriders override older overrides, and when/if the overrider calls the old definition via **inherited**, the next oldest overrider is called. This continues until the original definition is reached.

Some languages that implement overrides have the additional concept of “abstact” procedures or functions, which are procedures and functions without a natural definition. The original, defining module has no definition for the function or procedure, and it must be overridden to complete any reasonable function. This is a natural fit, for example to a device driver. The original module defines the general mechanics for the procedures and functions that all devices have, and the device implementation module fills out the actual functions and procedures based on the real device.

In the languages that implement abstracts, the original function or procedure generates and error if called. Pascaline can implement abstracts as well, I simply felt it was better left to the programmer how to handle the case of an abstract procedure or function that is directly called. Thus, it is a natural paradigm to have an abstract module that throws exceptions for all of its abstract procedures and functions.

12.38 Parallel modules

The system of modules in Pascaline extends itself very naturally to the implementation, in the language and not as an add-on, of parallel tasking features. Indeed, one of the biggest rewards for holding on to type safety in Pascal in general is that it sets the stage for Parallel tasking structures. A language that is type safe can be made to be multitask/multiprocess safe as well. There is no way to retrofit multitasking/multiprocessing security over a type unsafe language. C/C++ will never have a secure multitasking/multiprocessing system. Using the freedom to jump around the type system the multitasking/multiprocessing security can be jumped around as well.

Per Brinch Hansen understood this as well, and remarked that, sadly, some type safe languages also didn’t enforce multitasking/multiprocessing security as well, specifically in reference to Java. Type safety does not automatically lead to multitasking/multiprocessing safety as well, it has to be designed that way.

Pascaline addresses no less than three models of multitasking/multiprocessing:

1. Thread management.
2. Process management.
3. Multiprocessing.

A thread is started in a process module, which defines a new thread outside of the main thread that is assumed to run all of the other modules starting with the **program** module. Certainly this is a misnomer? A process could be defined as a thread plus a context in the form of variables, and a program to run. This describes the entire program, the variables, program code and any number of threads running in it. While the service module describes the creation and limited control of such separate, true processes, the **process** module describes the unition of one thread and one set of variables. Thus, a process module models a process in the context of the whole program. A **thread** does exist, but this is introduced later with classes.

Just as Per Brinch Hansen described, much of the security with Pascaline parallel modules is verified at compile time. process modules are excluded from calling other modules, which are, by default, under control of the main thread that is running the whole program. All processes, including the program module, can call a **monitor** module which comes directly from Per Brinch Hansens work and that of Dijikstra.

A monitor reads like an early definition of classes. It is a unification of data with the procedures and functions that access them. Any external access to the data contained is disallowed, and the procedures and functions of the monitor must be used to access the data for the client module. Further, the externally visible procedures and functions are all multitask "locked" such that only one thread at a time can enter the monitor.

The monitor is itself limited. It can only call other monitors or **share** modules. And it cannot call other modules with **var** parameters, even if they are monitors, since that effectively exports the data in the monitor.

The final class of module in the parallel tasking is the **share** module. The idea of a share module is simple. It has no data at all, just program code, constants and other fixed data and declarations. Since it has no data to corrupt, it can be called by all other modules. It is the "universal" module that can be called by all others, and it is the preferred way to represent pure declarations and code.

Monitors form the basic building block of the multitasking system. In fact, all of the lower level I/O mechanisims in typical Pascaline implementations are done with monitors.

Monitors in Pascaline give a single level system of multithread locking. Each monitor has its own lock, and it can only be held while the monitor is active. It could well be that there is only one lock actually in the system, since it is never held by more than one monitor at a time.

Situations certainly occur where a higher level of locking needs to exist. For example printing to the console device is certainly protected from multitasking failure, that is, the data will not be corrupted at a low level and cause a crash. But the normal I/O model freely mixes characters from multiple threads, resulting in a mishmash of output. In this case it is a matter of deciding what is wanted. If what is wanted is to present the data from each thread on a separate line, then a "secondary" lock is created that only allows one thread at a time to build up an output line, and only allows the other threads to present output when that line is finished.

An example of a secondary lock is given in the Pascaline standard. The point is that, even if monitors cannot handle all situations, they are building blocks that can form the basis of a higher level solution. This is very much in keeping with the theme of Pascaline.

12.39 Monitor signaling

Because monitors allow multiple threads in Pascaline to intersect, monitors are inherently communications mechanisims between such threads.To do that efficiently, monitors implement a system of semaphores, which are events, and procedures to signal such events and wait on them.

These procedures only make sense in the context of a monitor, and they are closely tied in with the underlying tasking mechanisim of the operating system. As professor Hansen showed, the act of waiting on a signal makes no sense unless the monitor lock is exited,allowing other tasks to run, then reentered when the signal appears. The actual mechanics of this are quite complicated and won't be covered here. Suffice to say that it is very easy to implement wrong, and very hard to implement correctly.

The reason why is that if the locking mechanisim allows tasks to "jump in front" of each other the signaling mechanisim is "unfair" and that is fatal for actual real world programs. This allows one thread to be locked out permanently because a more active thread is continually pushing it aside for the signal.

Semaphores look like variable definitions, and they can only be defined within a monitor. Likewise, **signal**, **signalone** and **wait** can only be used within a monitor, and can only reference a semaphore that is within their joint scope.

In the majority of cases, the **signalone** procedure is the appropriate means to flag an event. The broadcast nature of **signal** is sometimes useful, but more typically **signalone** is used to flag an event that only one thread at a time can handle. The Pascaline standard specifies that signalone may degenerate to signal, and this is used to justify always using a loop that rechecks if the event associated with the signal actually occurred:

```
while lockactive wait(mysignal);
```

Rather than just:

```
wait(mysignal);
```

However, I'll fully admit that this is a cheat. Implementing **signal**/**signalone**/**wait** in such a way that a signalone is not thus reliable strongly implies that fairlocking is not implemented. This is a good indication to run for it. Systems without fairlocking simply don't work in the real world, and thread lockout is a very real effect, difficult to avoid. While alas, there are several real world systems implemented without fairlocking, a reasonable Pascaline implementation is going to implement fairlocking on top of that unreliable, unfairlocked system.

Nevertheless, advising users to repeatedly check signaled events is good defensive programming, and I stand by it's inclusion in the standard.

12.40 Monitor analysis

Monitors and atoms are a key point in parallel tasking support because all calls that are shared between tasks are contained there. All ways in are locked using single level locks (I.E., not recursable). Monitors and atoms allow the constructor, destructor, and method code to use the public procedures, functions and methods as well, meaning that the monitor must create a different entry for external (to the monitor or atom) calls from that used internally.

Monitors and atoms form an exclusion zone. Only one task is allowed to enter the monitor or atom at a time, even if they call different routines. The monitor must protect not only the data within the monitor, but the data of callers as well. An example of this is:

```
! illegal example

monitor m;

procedure p(ps: pstring); forward;
```

```
var sp: pstring;

procedure p(ps: pstring);

begin

   sp := ps

end;

.
```

If a monitor were able to import pointers, it could store the pointer, and then operate on it during the call from another task. Voila! There is an escape from the exclusion zone. Thus Pascaline does not allow pointers to cross call boundaries in a parameter, even though a monitor or atom can only call other monitors or atoms.

What about var or view parameters?

```
! illegal example

monitor m;

procedure p(var i: integer); forward;

var x: integer;

procedure p(var i: integer);

begin

   x := i

end;

.
```

This example takes analysis. **var** or **ref** parameters do indeed import a pointer, but it exists in the locals stack only. There is no way to save it, and it only lasts as long as the call does. And the call is exclusive from the time it gains the lock at the entrance to the monitor, to the time it leaves. There are two ways out of a monitor while it holds the monitor lock:

1. An outbound call to another monitor.
2. A wait call task switch.

Both of these, of course, drop the monitor lock and regain it on reentry. And in both cases, the monitor must be written such that if another task enters the monitor during the time the monitor waits or is inside another monitor, it must account for that. This is one of the things that makes monitors tricky to use even though it is an exclusion zone[23].

[23] And indeed, outbound calls are depreciated for this reason.

Having said all that, there is no reason to prohibit **var** or **ref** prameters, either from inbound or outbound calls. The address is on the locals stack, and other tasks cannot access it. The original calling context (a program, process or task) is held in suspend while the call or wait is completed, and in fact this applies to any number of monitor or atom calls that it makes. A program that calls monitor A that calls monitor B holds two calls worth of parameter data on stack (monitor A and B) but that is on the local stack.

12.41 Channels

process, **monitor** and **share** modules are an answer to how to communicate and share data between threads, processes, and even multiple processors tied together using SMP architecture. However, what about disjoint communicating processors? In fact, that is a very common circumstance. Two machines connected by a network connection, or a slave processor on a peripheral, or even multiple processors connected in an array, but each with separate memory are examples.

What these situations all have in common is that they do not share all of their variables in common, but rather communicate a “block” of variables with the semantics of a reader and a writer. This model is the basis of two processors connected by a common “channel” and some amount of buffering on that channel.

Although it is possible to emulate procedure and function calls on such a channel, that is done by degenerating them down to read and write calls of data. Of late, a new model has emerged called “property based programming”. In this model, the read and write of variables are the key action, and procedures and functions are expressed as side effects of that read and write. In fact, this is a reasonable description of what happens in any variable access “underneath the covers”. A read or write is really implemented as a section of code that carries out the underlying action.

The result is a programming method that is well matched to communications between disjoint processors. It also so happens that the model is easily implemented between threads and processes as well, so property based programming can serve as a universal model of parallel programming.

The difference between a channel and a monitor is that the channel only exposes an external interface made of properties, and those properties are locked, and thus multitask hardened. The client of a channel and the server of a channel can be on different threads, or different CPUs,even connected by a network and geographically far from each other.

Further, the server could well be in hardware. Thus, a channel serves as a model for interface to hardware as well. This goes back to the IBM 360 concept of an I/O device interface as being an I/O “channel” to a hardware device.

12.42 Classes

Classes in Pascaline are implemented as modules whose data can be instantiated independently from the program code and other definitions that are static. This is the underlying model for Java and C#, and many other object oriented languages. It is in contrast to C++ and some dialects of Pascal, that treat classes as an extension of the record concept.

I started down the road of thinking of classes as an extension of the module concept with the reading of a paper on smalltalk which stated of the successor object oriented languages that “records (structures) were never the natural way to implement classes”. Indeed, classes are a program structure, not a data structure or record with extensions to handle object orientation.

In Pascaline, modules are conceptually static classes whose data and methods are joined a la C#. A class fits naturally in Pascaline herarchy under modules but over procedures, functions and methods. A module is the program text envelope for classes. Classes themselves are simply a model for the creation of objects. A class has no data directly associated with it. It becomes an object only when instantiated, and that can be as a static variable, or as a dynamic variable.

The representation of classes as a variant of modules both simplifies the language and extends the power of the resulting classes. Classes can contain any declaration. The declaration of a class contains all of its methods, and does not need to have its methods "explained" by name linkage later. A class has constructors and deconstructors just as modules do (in the form of initializers and deinitializers). Classes can have gotos. Within the methods of a class, members of the class can be accessed without qualified identifiers. A class can have private and public sections.

The relation to modules of classes underlies the basic style of Pascaline. In C++, the definition of a class must be included source-wise in the client code that uses the class, which effectively means a header file. Thus, there is incentive to break the methods of a class from its definition, and the various types, constants and associated data needed to build the class are not ***in*** the class. In Pascaline, program structures represent themselves, and having method code within the class is common, not just for short methods. Futher,the constants, types and other "building block" data associated with the class are more commonly in the class itself.

In fact, there is no reason a class cannot be void of variables, and thus be a wrapper for definitions. The qualified identifiers used to access members of a class acknowledge this. It "reference forgiveness" means that the name of the class itself can be used as a qualified identifier for compile time data such as constants, types and fixed. Alternately the qualified identifier can be the static or dynamic object reference itself. Neither generate runtime code.

12.42.1 Static objects

A static object is an image of the collection of runtime variables, virtual procedures and virtual functions that it contains. Constants, normal procedures and normal functions (methods) do not take space in the object.

The important thing about static objects is they are always exactly the image of the class that was used to construct them. Static objects cannot vary at runtime, although they can certainly be passed as a parameter to a compatible base class. In this case, just as for container arrays, a static class aquires a class template pointer "on the fly".

For a long time I thought that static objects would need to be declared as "instances" as follows:

```
var c: instance of ct;
```

The idea is that pointers to classes (references) need to be declared, and so should instances. What changed my mind is that it can be shown that classes are in fact a true type, and in fact, every type, including existing system types like integers, files, etc. For example and integer can be regarded as a class containing a member that contains its value, methods for setting it, adding it, etc.

Thus a declaration of a static object is equivalent to using that as a type.

12.42.2 Dynamic objects

Dynamic objects are allocated at runtime and resemble variant records and dynamic container arrays. Since a reference to an object (not a pointer) can reference any derived class from the base class specified.

So what is the difference between a pointer to a class and a reference to a class? For several years I, in fact, tried to unify the two, much as N. Wirth did in Oberon. I came to the conclusion that it was neither cleanly possible nor desirable. Why?

First, a pointer to a class would have a lot of special rules. A pointer in Pascal is bound to its base type. This is allowed to vary in structure, but that is according to a defined field in a variant record (the tagfield, even if missing). Also, the fields in a variant are defined at the time the base type of the pointer is defined. Pointers do not index type specifications that are incompletely defined at the time of their declaration.

Second, whereas the complete data collection a pointer points to is defined in Pascal, it is not with classes. Thus:

```
type r = record i: integer; c: char end;

var a, b: ^r;

begin

   a^ := b^;
```

Makes sense (assign what b is pointing at to what a is pointing at), the equivalent class does not:

```
program p;

type r = record i: integer; c: char end;

class c;

var i: integer;
    c: char;

begin end.

var a, b: ^c;
begin

   a^ := b^;

end.
```

(where ^c would stand in for a reference) does not. You cannot assign classes to one another via a reference, since by definition the length of those vary. Look at, for example, Oberon and the gymmastics that were required to rationalize direct assignment between objects.

Thus it made sense to me to have references be another, additional class of pointer, and that is how the situation is treated in other, similar langauges. In the above (incorrect) example for Pascaline:

```
program p;

type r = record i: integer; c: char end;
```

```
class c;

var i: integer;
    c: char;

begin end.

var a, b: reference to c;

begin

   a := b;
   a.i := b.i

end.
```

a := b is an assignment of the reference b to a, and a.i = b.i is assignment of one of the class members, i, from b to a. The dereferencing of a or b is not required, because a.i and b.i have only one possible meaning.

So what does a pointer to a class signify? It signifies a pointer to a static instance of the class:

```
program p;

class c;

var i: integer;
    c: char;

begin end.

var a, b: ^c;

begin

   a := b;
   a^.i := b^.i

end.
```

You told the compiler to create a static class instance, but then allocate that dynamically. We don't care that you have a pointer to a static class. It works, but it lost its special powers of inheritance, which only follow references. Note in the above example a := b would be invalid. Why? We forbid assignment of classes from one to another because their structure is not defined. The class c might look like a record, but it is not one. It has no specified format as a collection of types, as does a record.

This has precedence elsewhere in the language. Files cannot be assigned to each other. Further, you can actually assign two diffent objects to each other as long as you specify the definition of what it means to copy that object as an operator overload. This again matches other similar languages.

The final matter is why “reference to” and not a more compact (symbolic) representation? For this I followed the precendent of the declaration “file of” and “array of”. References are (as hopefully the above illustrates) pointers, and I wanted a declaration for them that was not similar.

12.42.3 Classes as parameters

When passing a class as a parameter, the main thing you need to be aware of is that classes cannot have their content copied. Thus, as with files, value or view parameters of classes have no meaning. Otherwise, passing classes as var or out parameters is perfectly reasonable. However, it, in fact, is rarely done in Pascaline. Why? Because as it passes an actual reference to the object, such a parameter loses the power of inheritance

The proper parameter mode to use is the **ref** parameter. Ref parameters are like “reference to” parameters but are not created dynamically. The power of a ref parameter is that it allows the parameter to be a class, or any of its derived classes. Thus it preserves inheritance as does a referenced object.

12.42.4 Class parameters

Classes, as with modules, can have both initializer code blocks as well as deinitializer code blocks (referred to as constructors and deconstructors in other similar languages). When a class has a parameter, it means that the initialization call of that instance can see those parameters. Also, unique to Pascaline, the deinitialization can see those as well, and they are the same parameter values. In fact, the value is calculated at the time the class instance is created, and a copy of that is held until the instance is deinitialized.

What is being expressed here is that the class parameters give per-instance characteristics that are true of the class instance throughout it’s lifetime. For example, a string class might have a fixed buffer length to store its string data throughout it’s lifetime. It would take a parameter of the length of that buffer, and maintain the value of that length until it is time to release the space occupied by the buffer.

Class parameters are by value only, since the duration of such parameters is unlimited, but the call to construct the class instance is not.

12.43 Inheritence

Inheritence in Pascaline is the ability to add new class members on top of old to obtain a new class. The correct term for this is “class extension”, and thus the extends keyword is the appropriate one for that. And that is the form in common use for newer languages.

Inheritence has meaning for both static and dynamic instances. In the case of static, it is simply the addition of fields to the base class.

Inheritence gets more interesting in the case of references. A reference can point at a class, or any of the base classes used to form that class. Further, it can be demonstrated that references can easily point to classes that are not defined ***anywhere*** in the base class sequence. The example here is of a list class:

```
program p;

class list;

var next: reference to list;

begin
```

```
end.

class data_list;

extends list;

var data: integer;

begin
end.

var l: reference to list;

begin
end.
```

Further uses of the list class to form more complex list entries can result in a list containing entries unknown to class data_list. Thus methods for that class must be able to determine if the reference given references a data_list object.

This is determined by the expression:

rp is data_list;

The difference between, say, C++ and Pascaline is that such a test is not optional. C++ puts it up to you to determine which class you are pointing to. There is a system called RTTI or "Run Time Type Identification" which can optionally be turned on to fix this. In Pascaline you could say this is always on. We don't do typeless here.

The result is that the **is** operator is really being executed anytime we need to determine if the reference indexes the class we are manipulating. This can be done by three steps:

1. Establish a globally unique ID for each class.
2. Carry a run time type identifier with each class.
3. Look up each class identifier in a table of class ids to determine if class we are referencing is the one we want.

Of course in real code we take advantage of several simplifications that make the system viable:

1. The globally unique ID for each class is formed by simply issuing addresses for each class, and each class has a table of addresses of the base classes that are used to construct it, in order, including for the class itself.
2. The is operator looks up the entry according to base class level. For example data_list is the second entry in the class table for the class data_list.
3. The address found is matched to the address of the class we want.

This makes the is operator very efficient, in fact. The drawback is that we cannot know how many classes total are in the system so we do not know how big each class table must be. There are two solutions to this issue:

1. Carry a table size for each class table.
2. Assume a global limit on the depth of classes.

I believe that #2 will be the most popular, since it reduces the **is** operator to a single index and compare operation.

Note also that the with statement evaluates a reference and holds that over an entire block:

```
program p;
class c;
var i: integer; r: real;
begin
end.
var x: reference to c;
begin
   with x do begin
      i := 1;
      r := 2.3
   end
end.
```

The test for if x refers to our class c, or another base class, is differed for all of the **with** controlled block. We know what x points to. No checks are required in the **with** controlled block.

12.44 Overrides for objects

Overrides for objects have a direct parallel in overrides for modules. They can even be implemented the same way, which is to save the virtual method vector in the object, then replace it with a new one. In C++ the idea was introduced that the resulting set of virtual vectors could be implemented as a fixed table, since all the methods would be defined at compile or link time. This remains an interesting future optimization.

12.45 Self referencing

One of the key features of the "class as module" plan (which other languages also use) is that the members of the class are accessable within methods without qualidents. Going back to the list class:

```
program p;

class list;

var next: reference to list;

procedure iterate;

begin

  self := next

end;

begin
end.

class data_list;
```

```
extends list;

var data: integer;

begin
end.

var l: reference to list;

begin
end.
```

For references, and references only, the keyword **self** stands for the object reference itself. Another way to say this is that all of the member references within a method are inherently based on **self** as a starting qualident. This is practically a list only feature, since that is the only real use for it.

What is the meaning of self for static instances? Well, you could use self in its other contexts:

self.data := 1;

But this typically is just redundant. Using self as an entire object is wrong according to copy rules:

self := other;

But you could well have defined a copy overload (assignment overload), so it could well be valid.

12.46 Constructors and destructors

The constructor and destructor arrangement in Pascaline is perhaps one of the features most different with other object oriented languages. However, it dovetails perfectly with the appearance of constructors and destructors as blocks of the class with blocks of a module, and that dovetails with the appearance of module constructor blocks as being the default block of any module.

Like module constructors and destructors, class constructors and destructors nest. When an object is instantiated, the constructors are called in order, starting with base class, through derived classes, and finally the instantiated class. For destructors it is the reverse. Constructors and destructors are tightly controlled. There is only one constructor per class, and one destructor. They cannot be overloaded, there is only one parameter list, and it is by value and lives for the duration of the instance. Both constructors and destructors, and indeed all method code, have the same parameters available. They can even change them, since they are value parameters, although it is hard to see why that would be required.

In fact, the Pascaline specification strongly implies that constructor parameters are nothing but class members that happen to be assigned when the class is instantiated.

Thus it is that Pascaline constructors and destructors have greater restructions than similar object oriented languages. I would argue that the reward is better and clearer control of object initialization and lifetime.

12.47 Operator overloads in classes

Operator overloads in classes typically involve one or more operands of the class type itself. However (see operator overloads in general), the rule is not enforced. You could well define operators in general, but that, again, is considered bad practice. Again, it is not Pascaline’s job to enforce good programming practice, but simply enforce type security and thus program integrity.

12.48 Derivation vs. composition

The “derivation vs.composition” is one of the few sections in Pascaline that can be considered a programming note. Inheritence based object programming as taken a few brickbats for having style issues, and I don’t consider it to be Pascaline’s issue to enforce or not enforce particular styles. Indeed, there is nowhere that Pascaline enforced object orientation in general, preferring multiple ways to do things if required. So composition has been, of late, touted as an alternative to inheritance, and so that is noted here.

12.49 Parallel classes

Classes in Pascaline essentially “make portable” the fixed module structure. Its worth having the discussion, “since classes are more powerful than modules, why not just have classes”. Indeed other languages have done exactly that, Java being a famous example. Typically in such languages you have the chicken vs. egg argument. That is, you must at least specify the main class to get started.

I found the C# design principle more compelling, which is to say that both static and dynamic classes are valuable, and static classes can contain dynamic classes and thus explain root classes completely. The fact that static classes (modules) and dynamic classes have a different look is perhaps an artifact of compatibility between Pascaline and Pascal, but it works well and is syntactically simple. Pascal-P6 uses the same code to create classes as modules.

The place in Pascaline where modules are most redundant with classes are in parallel classes. Parallel modules put a lot of work in just to create a tasking system that is inferior to parallel classes, and for that I should apologize. The excuse there is that parallel modules entered Pascaline long before parallel classes. Parallel modules are available if you don’t want to embrace the class model to get threading. The other excuse would be language symmetry, but that is a poor excuse indeed.

So in effect parallel classes give you a type safe toolset to create tasking. The example in the Pascaline specification is both instructive and fairly useless. The problem it solves is that, although the support modules for Pascaline I/O are multitask safe, they protect between multiple accesses within a single output statement and single characters, but not multiple characters on a line[24]. Thus it serializes each whole output line between tasks.

The thread class gives a new thread and context, a block of code to execute, and any number of routines to support that (as long as they are not called externally). As for the **program** module, the constructor for a thread is it’s on destructor. Realistically, it can be divided up into:

1. Startup code.
2. Run code.
3. Teardown code.

Just as for a **program** module. So what happens if the thread ends? It just ends, and its resources are reclaimed. With threads that can happen any number of times, with any number of threads.

Just as **process** or **program** modules can, a **thread** can use a **monitor**. However the atom class was specifically designed for this. Indeed the difference between a **monitor** and an **atom** is entirely the lifetime of its members. process/program controlled code can call atoms, just as **thread** code can call a **monitor**. Such mixing is controlled only by the lifetime of the member declarations. Its up to the programmer. I would personally pick a style, class or monitor, and stick with it.

[24] In fact the I/O library does, in fact, treat each whole writeln() or realn() call as whole for tasking purposes. Thus this is an artificial example.

Finally, classes have a version of the **channel** module known as a **stream**, which here could be literally defined as “little channel”. At last we arrive at a module class distinction that ***does*** matter, and matters a great deal.

Channels are a fundamental construct in Pascaline, as they are in the language **go**. For this you can go through the book “Communicating Sequential Processes” [C. A. R. Horare], although beware: that is a bit of a thick read. The basic idea is that threads or processes that pass messages can avoid a lot of the locking that is required when threads or processes attempt to share local variables.

However, that is just dipping a toe in the water. A thread or process connected by a channel can be an SMP thread in the same machine, or on another CPU core in the same machine, or on another machine on the other side of the world connected by the internet. It is a universal tasking contruct.

Channels also express neatly the border between software and hardware. Channels can describe the communication between software and the basic hardware devices of the system. For example a disk drive can be modeled as a channel, as well as a serial port, an Ethernet packet Network Interface card (NIC) or even a graphics card.

Thus the difference, in Pascaline, between a channel and a stream is that channels are used to express global hardware interfaces, and streams are used to express thread to thread communication, or perhaps very advanced hardware.