Introduction

In 1990, the ISO/IEC 10206 standard for extended Pascal came out. As a Pascal compiler implementer, I admit I was dismayed at seeing the standard. It seemed to me to be overly complex, decorated with nonessential features, and unnecessarily confusing in certain aspects.

Time has not softened my opinion of the standard, but I have voiced my opinion in perhaps the only way that really matters: I have created an alternative extended language. It is on the event of publishing that standard that I write this. Also, I feel that the readers of the web site I maintain:

http://www.standardpascal.org

Have a certain right to know why I have not advanced the ISO 10206 standard with as much fervor as I have the ISO 7185 standard. In addition, it has been suggested that the ISO 7185 standard be phased out in favor of the ISO 10206 standard, and I have strongly protested.

For these reasons, I wish to publish a reasonably limited critique of the standard as my reason for not backing it.

Although I have taken great pains to provide only factual information here, and not to give offense, I am well aware that offense will be given none the less. Standards are hard work to write, and I am critiquing their work without having participated in that work. For that offense, I apologize.

The committee process and standards making

The first topic I will discuss is the committee standard making process itself. The ISO 7185 standard was an unusual standard in that it was convened under the idea that the Pascal language was not to be changed or extended, but to be codified and disambiguated. In large part, I believe the ISO 7185 group succeeded in this effort, and that is why I back the ISO 7185 standard.

I shared the ANSI group concern about so called “conformant arrays”, because this feature, while something like it I would agree was essential to making the language practical, was in fact the first extension created for the language. Conformant arrays had all the earmarks of a short term fix. They addressed only one issue, and did it with fairly irregular syntax compared to the rest of the language. Their declaration was not only an exception to the rule that parameter types are declared with type ***names***, but also the array type declaration used was also irregular.

I was clearly not alone in this concern, which was argued at great length and caused the ANSI group and the BSI/ISO group to part ways. Further, the ISO 10206 standard adopted the conformant array feature but clearly treats it as a sequestered feature. More on that later.

The ISO 10206 committee was formed and operated in clear contrast to the ISO 7185 group, even though many members were in both groups. The ISO 10206 group set out to design a new language based on Pascal. And this is the point of my first issue.

Committees are not good at design, especially technical design. Committees ***are*** good at making standards, and (unfortunately) this leads often to their crossing over into the design realm. Few good designs have come from committees. They are mostly laws, treaties and other acts where compromise among a large number of people is essential. Compromise is what committees ***are*** good at. Design is not. The most famous example is the 48 byte packet size of ATM (Asynchronous Transfer mode) networks. It was a political compromise between 32 byte and 64 byte factions in the committee, rather than a good technical decision.

Being as committees must make standards, I believe there are a few characteristics of a good design that goes through the committee standards process:

- An existing, good design is standardized
- For each proposed part of the standard, a good, working existing implementation exists or is made to exist before the standard is published
- If a feature must be performed during the standards process, a side group consisting of a few skilled designers makes the proposal, which is then taken to the general group.
- The standard is not considered complete until a working, satisfactory implementation is performed.

This is based on the idea that people simply work better in smaller numbers for design purposes. The most coherent designs come from 1 to at most 5 people who work closely together. Even at that, it is dependent on human factors such as how well the members of the team work together.

When design groups rise above 5, the coherence breaks down. There is more effort in trying to get consensus than good design. It is a simple fact of human existence that we tend to diversify across many design ideas in large groups. It is what makes us resilient as a species, but it does not make for good design.

The ISO 10206 group failed most of the above points:

- The design work was done in committee
- There was no working implementation before the standard
- There were no working implementations created before completion of the standard

ISO 10206 makes clear how the process of "designing" the extensions were done, which was as a roundup of proposed features wanted for Pascal, from both the working group and suggested externally, followed by "reconciling individual candidate extensions" [the standard's words]. At the risk of offending the standards committees, that is a great way to make a shopping list, but it is not a good design principle.

The result of ISO 10206 is that only one implementation exists for the full standard even 17 years after its introduction, and two others that claim "partial" implementation of the standard.

To comment on the technical aspects of the standard, I will use the same order as the ISO 10206 standard uses, in the Introduction, under Technical Development. If a section does not appear, it means I had no issue with it.

Modularity and Separate Compilation

The modularity of ISO 10206 appears to be mainly inspired by the language Modula, another Niklaus Wirth language. This makes sense, because Wirth designed the original Pascal language, and puts a great deal of thought into his work.

First, modularity is an important advance for Pascal, and virtually all extended Pascals have this feature. Also, many implementations put forth the idea that the type security of Pascal was to be extended across the multiple files that made up the modules. This single characteristic put Pascal ahead of such languages as C that made type verification across source files entirely up to the user.

The biggest problem I find with ISO 10206 module implementation is shared by Modula and similar implementations. It creates two different sections, an interface section and an implementation section, and these must match. Creating and maintaining these two "parallel" sections is, to me, unnecessary work. One section must be created, usually by copy and edit from the other. Then, both sections must be maintained with respect to the other. Although the compiler can check if they reflect each other and produce error messages, it creates a problem area in the source code none the less.

One can, of course, argue that an automated program (an IDE or standalone program) can generate one of these sections from the other, this, to me, stands as simple proof that the implementation is getting enough information from the source that it does not need these two, redundant sections.

A better plan appeared in Oberon, another Wirth language, in the language Pascal-XSC, and in the language Pascaline, where the module declarations simply represent themselves, and a (relatively) simple method is given to declare exactly which of the constructs are available externally to the module. Having modules represent their own interfaces reduces the amount of work to prepare a module.

The remaining feature of the separated interface and implementation sections is the ability to "publish" an interface without the requirement to also provide the source code. But this ability is easily provided to the "self interfaced" module compilers via a utility that prepares an interface module without the internal source code by stripping out all of the details of the implementation within the module, and leaving only the declaration prototypes.

Next, ISO 10206 provides for a way to protect exported variables from writes by external modules. I have found this to be bad policy in general. It not only is questionable for security (you can look, but not touch), but it creates two different domains, inside and outside of a module, where a variable behaves differently, even though it is typed and named the same.

Last, ISO 10206 provides a wealth of ways to qualify the import and export of program objects by name. A specific list of exports can be provided for each module, as well as having the importer module select a specific list of imports. Further, the names can be changed when traversing from one module to another!

This is a recipe for tangled connections between modules. It mainly appears to be aimed at avoiding name conflicts between modules. But ISO 10206 provides the feature of module qualified identification, which are names of the form:

module.identifier

Which is used in several languages, and is a good solution to the problem. The language Pascaline in particular, does not allow individual names to be marked for export, but requires that they be clearly separated by section marked off by a "private" keyword that makes it very clear which identifiers are exportable, and which are not, and Pascaline allows the use of qualified identifiers always, with unqualified imports reserved only for modules that specify it, and then only as a whole module. You either get the entire module unqualified, or the whole import module is qualified. This makes for far more clear naming in modules, and less modality (why is this identifier access different ***here***?).

Schemata

Schemas appear to offer the ability to define types that vary at runtime. In detail, however, they can only be used to define variable subranges, and by extension, arrays based on those subranges. The schema construct:

type a(b, c…: <type>) = <type>;

gives a type with a series of "discriminants", which can then be used to form the type on the right hand side. The actual point at which the discriminants can be plugged in is limited effectively to the subrange construct:

```
type a(b, c) = b..c;
     x       = array [a] of integer;
```

schemas are effectively a parameterized, or template type.

The first and most pressing need for schemas is to solve the age old problem of fixed array sizes in Pascal. ISO 7185 defined array indexes in terms of subranges, which were a type themselves novel with Pascal. This made the array size part of its type, and created difficulties in both assigning constants to, comparing, and passing as parameters, variables of array type.

The difficulty I have with schemas is that they are:

- Complex
- Have implications beyond what was actually needed for the language (the ability to parameterize all subranges)
- Do nothing to unify N length array handling in the language, and in fact contribute greatly to the disunification of it.

Consider, if you want to pass an array of packed characters (the ISO 1785 Pascal definition of a String, and the ISO 10206 definition of a Fixed String) to a procedure or function, you can do that job via no less than four different ways, many of them ***mutually incompatible***:

1. As a ISO 7185 String or ISO 10206 Fixed String.
2. As a ISO 7185/ISO 10206 conformant array parameter.
3. As a ISO 10206 Variable String.
4. As a ISO 10206 schema array.

The reason that many of these methods are mutually incompatible is that the rules of type compatibility and assignment compatibility in 6.4.5 and 6.4.6 make it clear that standard fixed Pascal arrays are not compatible with Schemas, except in the case of strings. Schema packed character arrays, which would be the direct schema equivalence to ISO 7185 strings, are not compatible with strings even though strings are said to be schemas in ISO 10206.

This appears to be by design. One would guess the goal here is that if you want advanced array behavior such as the ability to define and pass arbitrary length arrays, then you need to use a schema. However, this is very slippery ground. What it requires is that you must use schemas even if you don't require the abilities of a schema in a particular area of code, simply because you may use fixed or ISO 7185 Pascal arrays together with schema based arrays. The two domains are locked out from one another. Schema based arrays clearly have greater cost than fixed arrays, and forcing all arrays to be declared as schemas, whether or not they really require schema treatment, can increase the costs of the implementation, sometimes dramatically.

Further, the complexity of the standard was artificially increased by the wall between schemas and fixed ISO 7185 types. Schemas can take other schemas as actual parameters for a formal parameter, but not fixed array types. Had this not been so, the conformant array parameter scheme would have had no use, and could have been eliminated.

In the language Oberon, Niklaus Wirth introduced the concept of "open array" parameters, whose basic property is that they may take fixed arrays. In Pascaline, this concept is further elucidated as "container types", that is, an array type that does not directly specify a particular array, but can contain other arrays, either defined at compile time, or created dynamically at run time. This interoperation between fixed and runtime defined arrays was key to the language, since it meant that the cost of a particular array was up to the user to define, and standard ISO 7185 fixed arrays where admissible as parameters, and compatible with dynamic arrays, at any time in the program.

In Oberon and Pascaline, an array can appear as:

```
array of integer;
```

In Pascaline, this can also be a type, meaning that the requirement of parameters being declared by type name is not broken. And Pascaline allows pointers to be created to such types, meaning that arbitrary length arrays of any dimension can be created at runtime.

In short, the massive change that was schemata was not only more complex than the problem they were designed to solve required, but schemas also divide array types into several groups that are not mutually compatible (fixed, schema and string). It wasn't necessary.

String capabilities

The issue with strings is that they simply should not be included in the language. The standard states that they are a specific instance of a schema with the type identifier "string". If ISO 10206 can handle strings using existing methods with arrays, then string handling should be an appendix to the standard, and not in the standard, as is the common practice of including features that are suggested, but not required. However, the short answer is that strings are not actually a general case of schemas, since strings have an extra field hidden from the user, which is the length of the string. Strings must therefore be a built in type in ISO 10206.

Strings in several languages, such as C, C++ and C# are considered user implemented data for good reason. There are many ways to implement a string, and even a single program may use several of these methods. It is not up to the compiler to select the ideal form, and having picked one, it often ends up being changed or added to. The early implementations of Pascal with strings (UCSD, Borland) ended up being changed because of length limits on strings, and static vs. dynamic implementation differences.

ISO 7185 Pascal considered strings to be a special case of an array, which meant that aside from the simple act of writing a string to an output device, it was up to the user to decide on string implementation. This, of course, lead to difficulties with string assignments from constants, ability to pass variable length strings to procedures, and other similar issues, but these were all issues stemming from ISO 7185's difficulty with handling arbitrary length arrays. Schemas, and to a limited extent conformant array parameters, "fixed" the problem. So it follows that it should not be any more necessary to provide a compiler supported string type in ISO 10206 Pascal than it was in ISO 7185 Pascal.

The only logical reason I have heard for the inclusion of strings in ISO 10206 is that strings were commonly implemented in other extended Pascals, beginning with UCSD. However, this was done mainly to attract Basic language programmers to Pascal. I would argue that this feature actually hurt Pascal's image as a serious programming language.

Binding of variables

The binding of variables in ISO 10206 is a generalized principle with exactly one concrete use, which is to allow external filenames to be specified for a variable of file type. In most implementations of extended Pascal outside of the ISO 10206 standard this is taken care of by one or two procedures. Pascaline and Borland Turbo Pascal take care of this by using the assign(f, n) procedure, which has the advantage of treating the filename of the file variable type as a property, and thus being upward compatible with the file procedures and functions of ISO 7185. In specific, if the assign(f, n) procedure is not used on a file variable before reset(f) or rewrite(f) is applied, the ISO 7185 behavior is seen with the file, which is to have the file opened as an anonymous file.

The binding methods of ISO 10206 certainly accomplish the job. However, the bindability concept goes on and on through the ISO 10206 standard and greatly adds to the complexity.

Constant Expressions

I have backed the use of constant expressions in the language Pascaline. A small nit to pick however, is the ISO 10206 definition of constant-expressions as general expressions with a compile time known result (this is specified in 6.8.2 Constant-expressions).

In the language Pascaline, this was done by defining the syntax for constant expressions aside from normal expressions, so that variables are not syntactically a part of such expressions. The net result of this is that a ISO 10206 implementation is going to have to validate general expressions for use as constants, when it could be inherent to the syntax.

Function Result Variables

The addition of a function result variable, which the ISO 10206 text states helps with structured returns, has the net effect of promoting the function result to a variable for various uses within the function block. ISO 7185 already, in my opinion, takes liberties with the rule that a function has one entry, one exit, and delivers the result of the function at the exit. This common rule is designed to keep you from having to divine exactly where and how within the function that the result is formed.

In any case, the exact same outcome as a function result variable can be obtained by declaring a normal local variable for the function, then adding an assignment statement at the end of the function to indicate that as the result. That makes is clearer where and when the result is being delivered, does not require a new construct in the language, and moreover is the clearest way to write ISO 7185 compatible code as well.

Initial Variable State

The ability to provide an initial value to a variable is popular with users, and exists in several other languages (C, C#, Java). However, it fundamentally changes the nature of ISO 7185 Pascal at a stroke, and it encourages what is widely acknowledged to be a bad programmer habit, that of relying on the contents of variables before they are specifically assigned to. It was not, I am convinced, left out of ISO 7185 Pascal by accident. Wirth didn't include this feature in his later languages (Modula and Oberon).

Further, there are hidden implementation costs with this feature. In order to arrange for variables to end up with a certain value before code execution, the values must be placed into a section of the run time image, or else the compiler must generate code to perform the initialization in a way that is hidden to the user. The most common implementation as used in C, is to provide a separate section containing an image of all global variables that have initial state, and their values. If the program is to be rerun, the section cannot be co-resident with the variables themselves, but rather the section containing variable values must be copied to the variables themselves each time the program is started.

In ISO 10206 Pascal, even local variables can have initial states set, meaning that a section must be set aside for each local variable section, and copied to the newly created locals when the procedure or function is activated.

In short, there is a lot of space and effort to get this feature to work in practical compilers. Wirth clearly felt that it was better for the programmer to do this work in the source code, where it actually might be done more efficiently. The programmer would be forced to think about what did and did not actually require initialization, and thus optimize the code in ways that the compiler cannot.

Short circuit Boolean evaluation

The short circuit boolean operations in ISO 10206 Pascal appear to have been added to placate critics of Pascal for its strict interpretation of expression order evaluation (don't count on it), vs. languages such as C. It wasn't specified in ISO 1785 by accident, and despite the bad press, it is relatively easy to avoid expression order evaluation difficulties, most commonly by using stepped if statements, such as:

if a and b then …

Becomes:

if a then if b then …

What Wirth was saying for ISO 7185 Pascal is that it was best left to the compiler to sort out what the most efficient order for execution is, and I would argue that with today's superscalar processors, it is more important than ever.

I don't think it is a good idea to complicate the language to accommodate users and encourage bad habits such as relying on evaluation order. Simple programmer education adequately covers this issue.

I'll note here that Wirth appears to have changed his mind on this point, in the language Oberon, where boolean operators are specified has having short circuit order. However, Oberon does not also provide both methods (short circuit and non short circuit Boolean operators).

What was not commented on

There are a great number of points in the ISO 10206 standard I have no disagreement with:

- File handling procedures and functions
- The file extend procedure
- Constant expressions (except for the way they were implemented, as above)
- Relaxation of order of declarations
- Implementation characteristics
- Case statements and variant record enhancements
- Date and time handling
- Inverse ord
- Standard numeric input
- Non-decimal representation of numbers
- Underscore in identifiers
- Zero field widths
- Halt

- Protected parameters
- Tag fields of dynamic variables

I have not implemented all of the above, or have not implemented features in the same way, in the language Pascaline for two reasons:

1. I don't believe in the idea of "buffet" standards, or picking and choosing parts of the standard you agree with or don't agree with. Subsetting standards negates the basic purpose of a standard, that is compatibility, and confuses users.
2. Only the creator or creators of a given standard or implementation have an exact grasp of what the trade offs and the synergies are in it. This is the general problem with treating features as individual parts. Niklaus Wirth said of Pascal that a language designer must consider how the different parts will be used together.

Conclusion

Given any system such as a programming language, there are going to be things you agree with an disagree with. I hope I have provided clear reasons why I disagreed with the ISO 10206 language specification.

Perhaps more than any other feature of ISO 10206 Pascal, the schemata feature caused me to realize that I could not consider it as a path forward in Pascal. Looking at this feature from an implementer's point of view, it was very complex and would take a great deal of effort to implement. Further, it didn't implement my top requirement for any scheme to add arbitrary length support to Pascal arrays, which was the ability to be backwards compatible with fixed array types. Finally, it would not only increase the overhead on the handling of arrays, but the wall placed between fixed and non-fixed types in ISO 10206 would have encouraged many users to put all arrays into schema form to avoid incompatabilities.

The schema feature of ISO10206 was, for me, a "deal breaker". It was several steps too far.

Scott Moore

2009/02/21