Extended BNF — A generic base standard

R. S. Scowen
9 Birchwood Grove, Hampton, Middlesex, Great Britain TW12 3DU

Abstract

Generic base standards, that is, ones defining these fundamental concepts of information technology, offer a way of improving standardization in IT by enabling greater commonality.

This paper looks briefly at the fundamental base standards of terminology and character sets. It then considers in more depth two other generic base concepts: syntactic metalanguages (for example, Backus Naur Form) and numeric floating-point constants. Case studies illustrate typical unnecessary variations in existing standards and demonstrate that even the simplest ideas are treated in widely different ways.

1 Background

Information Technology (IT) is one of the world's largest industries. Standardization is beneficial but complex; there are hundreds of subcommittees and working groups standardizing different topics in IT. The standardization process should be performed in accordance with the ISO/IEC Directives [4] which state:

> "Uniformity of structure, of style and of terminology shall be maintained not only within each standard, but also within a series of associated standards. The structure of associated standards and the numbering of their clauses shall, as far as possible, be identical. Analogous wording shall be used to express analogous provisions; identical wording shall be used to express identical provisions.
>
> The same term shall be used throughout each standard or series of standards to designate a given concept. The use of an alternative term (synonym) for a concept already defined shall be avoided. As far as possible ... only one meaning shall be attributed to each term chosen.
>
> These requirements are particularly important not only to ensure comprehension of the standard but also to derive the maximum benefit available through automated text processing techniques and computer-aided translation."

But these directives are not being followed. JTC1 subcommittees produce information technology standards after several years deliberation, drafting, and argument. Some are descriptive, defining a version of what already exists in multiple incompatible dialects. Others are prescriptive, defining something new which will fill a gap or replace some proprietary product. Almost all the members of the committees and working groups, despite the official liaisons, know little about most existing standards nor of the work which is going on in other subcommittees.

The inevitable result is that many standards, despite the requirements prescribed in the Directives, redefine the same concepts. Naturally, they do it more or less differently in content, notation and terminology. The standards are therefore unnecessarily large and incompatible with each other. They also take considerably longer to complete. Note that the common concepts are often the simplest and most easily understood, they thereby attract the most discussion and arguments.

Generic base standards define common basic elements that can be invoked as required for specific standards and applications. Such base standards provide the benefit of promoting commonality across the various invoking standards while simultaneously reducing the effort needed to develop those standards. In IT they would define the fundamental concepts of information technology, offer a way of improving standardization in IT by enabling greater commonality and less work by working groups reinventing wheels. Several such standards could be applicable, not just in different working groups of a JTC1 subcommittee, but even in several different subcommittees of JTC1 and Technical Committees of ISO. If they do not exist, or the experts defining standards do not know about them, then each new standard will redefine and re-express the relevant concepts: the result is wasted

effort and unnecessary diversity and incompatibility among different standards.

Generic base standards

1. Reduce the effort needed to define standards because the working group avoids arguing about the basic concepts,
2. Facilitate the interworking of standards because they are based on common foundations,
3. Prevent needless diversity by avoiding different notations for the same concepts,
4. Enable standards to be shorter and simpler, because the concepts can be defined by reference to an existing standard, rather than being redefined,
5. Encourage the development of tools to check the internal consistency of standards,
6. Make standards easier to understand, because the basic concepts and terminology are common to several different standards.

2 Syntactic metalanguages

A syntactic metalanguage is a notation for defining the syntax of a language by a number of rules. The concepts are well known, but many slightly different notations are in use.

A metalanguage brings order to the formal definition of a syntax and is useful not just for the definition of programming languages, but for many other formal definitions, for example the format for references in papers submitted to a journal, or the instructions for performing a complicated task.

Since the definition of the programming language Algol 60 [12] the custom has been to define the syntax of a programming language formally. Algol 60 was defined with a notation now known as BNF. This notation has proved a suitable basis for subsequent languages but has frequently been extended or slightly altered. The many different notations are confusing and have prevented the advantages of formal unambiguous definitions from being widely appreciated.

2.1 What is a syntactic metalanguage?

A syntactic metalanguage is defines the syntax of a language by a set of rules. Each rule names part of the language (called a non-terminal symbol of the language) and then defines its possible forms. A terminal symbol of the language is an atom that cannot be split into smaller components of the language.

A formal syntax definition has three distinct uses:

1. it names the various syntactic parts (i.e. non-terminal symbols) of the language;
2. it shows which sequences of symbols are syntactically valid sentences of the language;
3. it shows the syntactic structure of any sentence of the language.

Note that the language being defined must be linear, i.e. the symbols in a sentence of the language can be placed in an ordered sequence. For example knitting patterns and recipes in cooking are linear languages, but electronic circuit diagrams are not.

2.2 The need for a standard syntactic metalanguage

Without a standard syntactic metalanguage every programming language starts by specifying the metalanguage used to define its syntax. This causes various problems:

Many different notations It is rare for two different programming languages to use the same metalanguage. Thus human readers are handicapped by having to learn a new metalanguage before they can study a new language.

Concepts not widely understood The lack of a standard notation hinders the use of rigorous unambiguous definitions.

Imperfect notations Because a metalanguage needs to be defined for every programming language, the metalanguage often contains defects — examples are given below in sections 5.5 and 5.7.

Special purpose notations A metalanguage defined for a particular programming language is often simplified by taking advantage of special features in the language to be defined. However, the metalanguage is then unsuitable for other programming languages.

Few general syntax processors The multiplicity of syntactic metalanguages has limited the availability of computer programs to analyse and process syntaxes, for example to check the consistency of a syntax, to list a syntax neatly, to make an index of the symbols used in a syntax, to produce a syntax-checker for programs written in the language.

In practice experienced readers have little difficulty in picking up and learning a new notation, but even so the differences obscure mutual understanding and hinder communication. A standard metalanguage enables more people to crystallize vague ideas into an unambiguous definition. It is also useful because people needing to provide formal definitions no longer need to reinvent similar concepts.

2.3 ISO/IEC 14977 *Extended BNF*

ISO/IEC 14977 *Extended BNF* [2] defines a standard syntactic metalanguage based on BNF. It includes the most widely adopted extensions together with additional features which experience has shown are often required when providing a formal definition:

Terminal symbols Terminal symbols of the language can be denoted by enclosing them in either double or single quotes, for example `"x"` or `'x'`. This enables any character to be a terminal symbol of the language.

Definitions for an explicit number of items Fortran contains a rule that a label field contains exactly five characters; an identifier in PL/I or COBOL has up to 32 characters. Rules such as these can be expressed only with difficulty in BNF, but in *Extended BNF*:

```
Fortran label = 5 * character ;
```

Definitions specifying the exceptional cases An Algol comment ends at the first semicolon. A rule like this cannot be expressed concisely or clearly in BNF, but in *Extended BNF*:

```
comment character = character - ";" ;
```

Comments Programming languages and other structures with a complicated syntax need many rules to define them. A syntax will be clearer if explanations and cross-references can be provided; accordingly the standard metalanguage contains a comment facility so that ordinary text can be added to a syntax for the benefit of a human reader without affecting the formal meaning of the syntax.

Multi-word meta-identifiers A meta-identifier (the name of a non-terminal symbol in the language) need not be a single word or enclosed in brackets because there is an explicit concatenate symbol. This also ensures that the layout of a syntax (except in a terminal symbol) does not affect the language being defined.

Extensibility A user may wish to extend the syntactic metalanguage. A special-sequence is provided for this purpose, the format and meaning are not defined in the standard except to ensure that the start and end of an extension can always be seen easily.

Extended BNF is summarized in table 1. The middle column of the table indicates, when appropriate, whether the metalanguage symbol is a prefix operator, or an infix operator, or a postfix operator.

Table 1: *Extended BNF*

Extended BNF	Operator	Meaning
`unquoted words`		Non-terminal symbol
`" ... "`		Terminal symbol
`' ... '`		Terminal symbol
`( ... )`		Brackets
`[ ... ]`		Optional symbols
`{ ... }`		Symbols repeated zero or more times
`{ ... }-`		Symbols repeated one or more times
`=`	in	Defining symbol
`;`	post	Rule terminator
`|`	in	Alternative
`,`	in	Concatenation
`-`	in	Except
`*`	in	Occurrences of
`(* ... *)`		Comment
`? ... ?`		Special sequence

The metalanguage symbols '`=`' '`|`' '`,`' '`-`' '`*`' are infix operators which bind increasingly tightly.

Extended BNF has already been used for the syntax of several language standards, including Modula-2 [8], VDM-SL [11, 3], and Prolog [10].

3 Floating-point constants

People normally count in decimal, many computers calculate efficiently only in binary. The two systems are not compatible, and arbitrary decimal numbers cannot be represented exactly internally in the computers, instead an approximation to the specified value must be stored.

Every scientific programming language therefore needs to define how to represent an arbitrary numerical constant, and what approximation to the value of this constant will be stored in the program.

4 Case studies — Syntactic metalanguages and floating-point constants

The following case studies examine four scientific programming languages: Algol 60, Extended Pascal, Minimal BASIC, and C. Algol 60 was for long regarded as a model language definition, the others are all International Standards. Two examples, syntactic metalanguages and unsigned floating-point constants illustrate typical unnecessary variations in these languages.

For each language, the syntactic language used to define the language's syntax is first summarized, and then illustrated by reproducing the definitions of the forms of decimal numbers which can be written in programs. The definition is then rewritten in *Extended BNF*.

4.1 Algol 60

The syntactic metalanguage in Algol 60 [12], see table 2, is BNF (Backus Naur Form) — Peter Naur's adaptation of John Backus's notation (Note that the Algol 60 report never actually refers to BNF which was at first an abbreviation for 'Backus Normal Form', and only later changed to stand for 'Backus Naur Form' in recognition of Naur's improvements. It is rare for someone who claims to use BNF as a syntactic metalanguage actually to do so; more often they use a dialect or extension instead).

Table 2: BNF syntactic metalanguage

BNF	Operator	Meaning
⟨ unquoted words ⟩		Non-terminal symbol
unquoted characters		Terminal symbol
::=	in	Defining symbol
|	in	Alternative

An Algol 60 floating-point constant is called an 'unsigned number' and defined by the rules (clauses 2.2.1, 2.5.1 of [12]):

⟨unsigned integer⟩ ::= ⟨digit⟩
  | ⟨unsigned integer⟩ ⟨digit⟩
⟨integer⟩ ::= ⟨unsigned integer⟩ |
  + ⟨unsigned integer⟩ | − ⟨unsigned integer⟩
⟨decimal fraction⟩ ::= . ⟨unsigned integer⟩
⟨exponent part⟩ ::= $_{10}$ ⟨integer⟩
⟨decimal number⟩ ::= ⟨unsigned integer⟩ |
  ⟨decimal fraction⟩ |
  ⟨unsigned integer⟩ ⟨decimal fraction⟩
⟨unsigned number⟩ ::= ⟨decimal number⟩ |
  ⟨exponent part⟩ |
  ⟨decimal number⟩ ⟨exponent part⟩
⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

These rules expressed in *Extended BNF* are:

```
unsigned integer = digit
  | unsigned integer, digit ;

integer = unsigned integer
  | "+", unsigned integer
  | "-", unsigned integer ;

decimal fraction = ".", unsigned integer ;

exponent part = "10", integer ;

decimal number = unsigned integer
  | decimal fraction
  | unsigned integer, decimal fraction ;

unsigned number = decimal number
  | exponent part
  | decimal number, exponent part ;

digit = "0" | "1" | "2" | "3" | "4"
  | "5" | "6" | "7" | "8" | "9" ;
```

Algol 60 defines the value of a 'decimal number' in clauses:

> "2.5.3 Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10."
>
> "3.3.3 ... The actual numerical value of a primary is obvious in the case of numbers. ..."
>
> "3.3.6. Numbers ... of type **real** must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a

finite accuracy. ... No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. ..."

4.2 Extended Pascal

Table 3: Extended Pascal syntactic metalanguage

Pascal	Operator	Meaning
meta-identifier		Non-terminal symbol
' ... '		Terminal symbol
[...]		Optional symbols
{ ... }		Symbols repeated zero or more times
= or >	in	Defining symbol
.	post	Rule terminator
|	in	Alternative

The metalanguage used in Extended Pascal (and Pascal) is defined in clause 4 of [7], and summarized in table 3. A Pascal floating-point constant is called an 'unsigned-real' and defined by the rules (clauses 6.1.1, 6.1.7 of [7]):

unsigned-real = digit-sequence '.' fractional-part
['e' scale-factor]
| digit-sequence 'e' scale-factor .

sign = '+' | '–' .

fractional-part = digit-sequence .

scale-factor = [sign] digit-sequence .

digit-sequence = digit { digit } .

digit = '0' | '1' | '2' | '3' | '4'
| '5' | '6' | '7' | '8' | '9' .

These rules expressed in *Extended BNF* are:

```
unsigned real = digit sequence, ".",
   fractional part, [ "e", scale factor ]
 | digit sequence, "e", scale factor ;

sign = "+" | "-" ;

fractional part = digit sequence ;

scale factor = [ sign ], digit sequence ;

digit sequence = digit, { digit } ;

digit = "0" | "1" | "2" | "3" | "4"
 | "5" | "6" | "7" | "8" | "9" ;
```

Extended Pascal defines the value of an 'unsigned-real' in clauses:

"6.1.7: ... An unsigned-real shall denote in decimal notation a value of real-type (see 6.4.2.2). ..."

"6.4.2.2: ... b) real-type. ... The values shall be implementation-defined approximations to an implementation-defined subset of the real numbers, denoted as specified in 6.1.7 by signed-real."

4.3 Minimal BASIC

Minimal BASIC [5] is a standard which defines a language which is approximately the common intersection of all BASIC implementations. The metalanguage is defined only in an informative annex (annex B of [5]). It is summarized in table 4.

Table 4: Minimal BASIC syntactic metalanguage

BASIC	Operator	Meaning
italics		Non-terminal symbol
ITALICS		Terminal symbol
=	in	Defining symbol
/	in	Alternative
?	post	Optional symbols
*	post	Symbols repeated zero or more times

There is only one numeric type in BASIC and a 'numeric-rep' is defined by the rules (clauses 5.2, 7.2 of [5]):

digit = *0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9*

sign = *plus-sign / minus-sign*

numeric-rep = *significand exrad?*

significand = *integer full-stop? / integer? fraction*

integer = *digit digit**

fraction = *full-stop digit digit**

exrad = *E sign? integer*

These rules expressed in *Extended BNF* are:

```
digit
 = "0" | "1" | "2" | "3" | "4"
 | "5" | "6" | "7" | "8" | "9" ;

sign = plus sign | minus sign ;

numeric rep = significand, [ exrad ].

significand
 = integer, [ full stop ]
 | [ integer ], fraction ;

integer = digit, { digit } ;

fraction = full stop, digit, { digit } ;

exrad = "E", [ sign ], integer ;
```

Minimal BASIC defines the value of a 'numeric-rep' in clause:

> "7.4 Semantics
>
> The value of a *numeric constant* is the number represented by that constant. 'E' stands for 'times ten to the power'; if no *sign* follows the symbol *E*, then a *plus-sign* is understood. *Spaces* shall not occur in *numeric-constants*.
>
> A *program* may contain numeric representations which have an arbitrary number of *digits*, though implementations may round the value of such representations to an implementation-defined precision of not less than six significant decimal digits.
>
> *Numeric constants* may also have an arbitrary number of *digits* in the *exrad*, though non-zero constants whose magnitude is outside an implementation-defined range may be treated as exceptions. It is recommended that the implementation-defined range for *numeric constants* be approximately 1E–38 to 1E+38 or larger. Constants whose magnitudes are less than machine infinitesimal shall be replaced by zero, while constants whose magnitudes are larger than machine infinity shall be reported as causing an overflow."

4.4 C

The metalanguage used in C [1] is defined very briefly (clause 3 of [1]), and summarized in table 5.

Table 5: C syntactic metalanguage

C	Operator	Meaning
italics		Non-terminal symbol
bold		Terminal symbol
:	in	Defining symbol
"one of" or on a separate line	pre	Alternative
opt	post	Optional symbols

A C floating-point constant is called a 'floating constant' and defined by the rules (clauses 3.1.2, 3.1.3.1 of [1]):

floating-constant:
  fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
  digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
  digit-sequence$_{opt}$ **.** *digit-sequence*
  digit-sequence **.**

exponent-part:
  e *sign*$_{opt}$ *digit-sequence*
  E *sign*$_{opt}$ *digit-sequence*

sign: one of
  + −

digit-sequence:
  digit
  digit-sequence digit

floating-suffix: one of
  f l F L

digit: one of
  0 1 2 3 4 5 6 7 8 9

These rules expressed in *Extended BNF* are:

```
floating constant
```

```
 = fractional constant, [ exponent part ],
   [ floating suffix ]
 | digit sequence, exponent part,
   [ floating suffix ] ;

fractional constant
 = [ digit sequence ], ".", digit sequence
 | digit sequence, "." ;

exponent part
 = ( "e" | "E" ), [ sign ], digit sequence ;

sign = "+" | "-" ;

digit sequence = digit
 | digit sequence, digit ;

floating suffix = "f" | "l" | "F" | "L" ;

digit = "0" | "1" | "2" | "3" | "4"
 | "5" | "6" | "7" | "8" | "9" ;
```

C defines the value a 'floating constant' in clause:

> "3.1.3.1: ... **Semantics**
>
> The significand part is interpreted as a decimal rational number; the digit sequence in the exponent part is interpreted as a decimal number. The exponent indicates the power of 10 by which the significand part is to be scaled. If the scaled value is in the range of representable values (for its type) the result is either the nearest representable value, or the larger or smaller representable value immediately adjacent to the nearest representable value, chosen in an implementation-defined manner."

5 The metalanguages reviewed

Each language has not only adopted a different metalanguage in order to define its syntax, it has adopted a different terminology. *Extended BNF* and Extended Pascal refer to "terminal symbols and non-terminal symbols", but Minimal BASIC refers to "terminal metanames and metanames", and C refers to "syntactic categories (nonterminals) and literal words and character set members (terminals)".

More seriously they all contain at least one of the following undesirable features.

5.1 Layout is significant

The meaning of a syntax depends on the layout of the rules, for example 1) a space is necessary to indicate concatenation of successive symbols in Pascal, BASIC, and C, and 2) a new line is necessary to separate two successive syntax rules in Algol 60, and a blank line is necessary in C.

5.2 No comment facilities

No comment facilities are provided in any of the metalanguages: with a large syntax, it is very helpful to a reader if syntax rules include cross references to the clauses where non-terminal symbols are defined.

5.3 Recursion needed to express repetition

A recursive definition is necessary to represent repetition of syntactic elements in Algol 60 and C.

5.4 Special fonts or characters are required

The syntax relies on different fonts or special characters to distinguish different syntactic elements in Algol 60, Pascal and C; this makes it necessary for the syntax to be printed rather than being expressed in ASCII, and thus makes it more difficult to read the syntax as data for a program.

BNF has metalinguistic symbols ⟨ ⟩ which differ from < > (greater than, less than symbols). Those who express BNF in ASCII (which came later) are unable to preserve this distinction because it is impossible to distinguish between the terminal character for 'greater than' and the opening angle bracket of a non-terminal symbol.

Pascal quotes terminal symbols with ' and ' because these symbols are not characters which can occur in a Pascal program.

5.5 Careless definition of the metalanguage

The definition of the syntactic metalanguage is frequently carelessly expressed. For example, C fails to state that a non-terminal symbol is one or more words separated by hyphens. And since **bold** type indicates a terminal symbol and *italics* indicates a non-terminal symbol, it is necessary in C to know whether '.' in a syntax rule is in bold or italic.

Both Pascal and C fail to define the relative precedence of the metalanguage operators.

Extended Pascal defines (table 1 of [7]) the metalanguage character `>` to have the meaning "shall have as an alternative definition". This could (but shouldn't) be taken to mean that the two production rules for `formal-parameter-section` (clauses 6.7.3.1, 6.7.3.7.1 of [7]) are alternatives; in fact the second rule is an additional rule which is applicable in some circumstances.

```
formal-parameter-section
 > value-parameter-specification
 | variable-parameter-specification
 | procedural-parameter-specification
 | functional-parameter-specification .
formal-parameter-section
 > conformant-array-parameter-specification .
```

5.6 Careless use of the metalanguage

One of Peter Naur's innovations in BNF was that the names of non-terminal symbols should reflect the semantics of the language. In both Pascal and C `digit sequence` represents both an integer and the digits of a decimal fraction. It would be better to replace the relevant rules by:

```
pascal fractional part = digit, { digit } ;

c fractional constant
 = [ digit sequence ], ".", digit, { digit }
 | digit sequence, "." ;
```

In Pascal, BASIC, and C the definition and use of `sign` fails to indicate that `+` and the absence of a sign are semantically the same: it would be better to define a Pascal scale factor as

```
sign = [ "+" ] | "-" ;

scale factor = sign, digit sequence ;
```

5.7 Limited use of the metalanguage

BNF does not distinguish between the symbols of the metalanguage, and the terminal characters of the language being defined. It is therefore impossible to specify any language which has | as a terminal symbol.

In C, syntax rules are limited by the length of the printed line.

6 Numerical constants

All four languages refer to a floating-point constant by a different name. Perhaps this is deliberate because they define different entities, perhaps not. Table 6 shows which numbers are valid in the four languages.

Table 6: Numerical constants

Constant	Algol	BASIC	Pascal	C
12	√	√		
.34	√	√		√
12.		√		√
12.34	√	√	√	√
$_{10}5$	√			
$12_{10}5$	√	√	√	
$.34_{10}5$	√	√		√
$12._{10}5$		√		√
$12.34_{10}5$	√	√	√	√
$_{10}$	$_{10}$	E	e or E	e or E

There are rational reasons for this diversity. Thus Pascal uses '.' for many different purposes and it would at least complicate and more probably lead to an ambiguous syntax if a constant could begin or end with '.'. And languages which use a letter to indicate the exponent indicator $_{10}$ often need to distinguish between an identifier and numerical constant.

The four languages all recognize that the value of the constant will need to be an approximation of the decimal value. But only Minimal BASIC requires a minimum level of precision, and no language specifies a range of values that must be accepted (although Minimal BASIC recommends a minimum one).

The accuracy of the value stored in the computer compared with the numeric value of the constant is also treated differently. In Extended Pascal it is implementation-defined (with no minimum accuracy required), in Minimal BASIC at least "six significant decimal digits", and in C "the nearest representable value" or one adjacent to it.

7 Conclusions

Technically there are no great difficulties defining generic base standards; there are however other problems:

1. Can they be defined sufficiently generally?
2. How can their existence be made known to working groups? Many generic base standards are not limited in applicability to a single subcommittee.

The full benefits will be realized only if they are known and used when appropriate by the whole of JTC1.

3. Can they be defined in time for use by working groups?

4. How can agreement be reached on such standards? It is often the case that the simpler the topic, the more difficult it is to reach agreement.

5. How can the definition and dissemination of these standards be financed? ISO receives the bulk of its income from the sale of standards. ISO cannot therefore afford to give standards away free. But generic base standards are only of benefit if they are made easily available to working groups, and then adopted by them.

A few solutions which would improve the development and effectiveness of IT standards are:

1. JTC1 to insist on the adoption of existing generic base standards.

2. JTC1 to study generic base standards: (1) to identify suitable areas of standardization, (2) to consider ways to ensure their adoption.

3. JTC1 or DISC to educate new (and existing) WG conveners and project editors.

4. JTC1 to adopt such technology as it becomes available.

5. ISO/IEC Secretariat to publish standards in computer-readable form so that they can analysed easily and so that particular instantiations and bindings can be incorporated easily into new standards.

JTC1 acts only at the bequest of its members so BSI (prompted by DISC) will require to take the initiative.

Many of these solutions would cost money. This could be raised in various ways, for example, by requiring (1) a contribution to the cost of standardization by every National Body voting in support of a new work items, or (2) an annual subscription by each member of a Working Group. The latter suggestion would be new in ISO and JTC1, but is already followed in USA where membership is open, but requires a fee to cover the cost of circulating papers, etc.

8 Short paper

8.1 Introduction

8.1.1 BNF uses of

8.1.2 Background

8.1.3 Introduction

8.2 Extended BNF

8.2.1 BNF

8.2.2 Extensions

8.2.3 Examples

8.3 Extensions

8.3.1 Characters set

8.3.2 End-of-line comment

8.3.3 Simultaneous keys

8.3.4 Attribute grammars

Acknowledgements

This paper is bn updated and shortened version of a paper presented at SESS'93 (Software Engineering Standards Symposium 1993) which was itself based on a report [13] prepared for and supported by UK Department of Trade and Industry, Information and Manufacturing Technologies Division.

References

[1] ANSI X3.159-1989. *American National Standard for information systems — Programming Language C.* American National Standards Institute, New York, USA. 1989.

[2] BSI (British Standards Institution). *BS 6154:1981 Method of defining — syntactic metalanguage.* 1981. ISBN 0-580-12530-0.

[3] J. Dawes. *The VDM–SL Reference Guide*, Pitman Publishing, London. 1991. ISBN 0-273-03151-1.

[4] ISO/IEC Directives — Part 3. *Drafting and presentation of International Standards.* International Organization for Standardization, Geneva, 1989.

[5] ISO 6373:1984. *Data processing — Programming languages — Minimal BASIC.* ISO Copyright Office, Geneva. 1984.

[6] ISO 8859-1:1987, *Information Processing — 8-bit single-byte coded graphic character sets — Part 1: ISO Latin Alphabet No. 1 (ISO Latin-1).* ISO/IEC Copyright Office, Geneva. 1987.

[7] ISO/IEC 10206:1991. *Information technology — Programming languages — Extended Pascal.* ISO/IEC Copyright Office, Geneva. 1991.

[8] ISO/IEC 10514-1:1996(E). *Information technology — Programming languages — Modula-2 — Part 1: Base language.* International Organization for Standardization, Geneva, 1996.

[9] ISO/IEC TR 12382:1992. *Permuted index of the vocabulary of information processing.* ISO/IEC Copyright Office, Geneva. 1992.

[10] ISO/IEC 13211-1:1995(E). *Information technology — Programming languages — Prolog — Part 1: General core.* International Organization for Standardization, Geneva, 1995.

[11] ISO/IEC DIS 13817-1:1995(E). *Information technology — Programming languages, their environments and system software interfaces — Vienna Development Method - Specification Language — Part 1: Base language.* International Organization for Standardization, Geneva, 1995.

[12] P. Naur (editor), J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, M. Woodger. "Report on the Algorithmic Language ALGOL 60," *Comm. ACM*, Vol. 3, pp. 299-314, 1960.
Also published by Regnecentralen (Copenhagen, 1960) and elsewhere.

[13] R. S. Scowen. *Generic base standards — Final report.* SEG C1 N10 (DITC Software Engineering Group), National Physical Laboratory, Teddington, Middlesex, UK. January 1993.

9 INFORMATIVE ANNEX A — Document history

9.1 Changes

16 September: A copy sent to Markus Kuhn.

18 September 1996: The file is copied from r246.tex which is a paper, based on the deliverable final report for NPL SEG 1991-92 work package C1, was submitted and accepted for SESS'93 (Software Engineering Standards Symposium 1993).

NOTE — This annex is not part of the paper.

9.2 Document history

Author(s): R S Scowen.

Date: September 17, 1998

Files: /usr/users/rss/r368.tex

Status: Draft

Document type: Paper for SIGPLAN Notices, or Computers and standardization.

Circulation: ??

Action: To complete.

9.3 Plans

1. ??