Practical Formalization of Programming Language Semantics

José E. Marchesi
jemarch@gnu.org

GNU Tools Cauldron - 2026

All my work is carefully elaborated by myself without the assistance of LLMs. This includes code, documentation, publications, communications, graphics, and everything else.

Outline

Prologue

Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

Prologue

Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

Welcome

I salute your bravery and apologize for this temerity.

Thanks for coming!

Prologue

Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

Programming Languages

- Languages are possibly infinite sets of **sentences**.
- Each sentence is a sequence of **symbols**.
- The sentences of a **programming languages** are the set of all correct programs.

Aspects of a language

- **Syntax** - structure or form of its sentences.
- **Semantics** - meaning of its sentences.
- **Pragmatics** - practical use of its sentences.

Language Specifications

- Describe some **subject language**.
- Use one or more **metalanguages**.
- **Good specifications** describe a language in a complete, consistent, precise, unambiguous, intelligible and useful way.
- *ab initio* vs. *a posteriori* specifications.

Informal descriptions

- Metalanguage is some **natural language**, like English.
- Sometimes augmented by a few mathematical devices.
- Can be very precise, but inherently **ambiguous**.
- Translations to other languages might be problematic.
- Will surely lead to multiple interpretations.

Formal descriptions

- Also known as formalisms.
- Require a formal **metalanguage**.
- Based on some **formal system**.
- The metalanguage may or may not be **grammatical**.

Why having language specifications

- For **language designers**.
  - Improved language design.
  - Depth insight into the nature of the language.
  - Expose non-obvious irregularities.
  - Standardization of programming languages.
- For **language implementers**.
  - Reference for implementers.
  - Proofs of implementations.
  - Automatic implementation.
- For **language users**.
  - Reference for users.
  - Proofs about programs.

Prologue

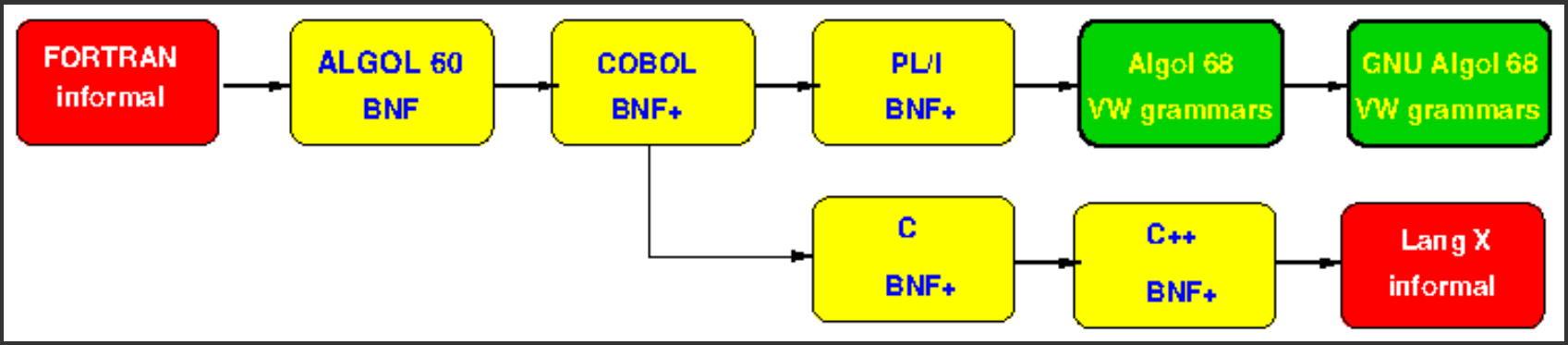
Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

The state of the art



FORTRAN

The FORTRAN Automatic Coding System (excerpt)

The FORTRAN language is most easily described by reviewing some examples.

Example 1: Compute:

$$root = \frac{-(\frac{B}{2})+\sqrt{(\frac{B}{2})^2-AC}}{A}$$

FORTRAN Program:

```
      ROOT = (-(B/2.0)+SQRTF((B/2.0)**2-A*C))/A.
```

Notice that the desired program is a single FORTRAN statement, an arithmetic formula. Its meaning is: ‘‘Evaluate the expression on the right of the = sign and make this the value of the variable on the left.’’ The symbol * denotes multiplication and ** denotes exponentiation[...]

Syntax, **Semantics** and **Pragmatics** are all expressed informally. Can you identify them?

ALGOL 60

ALGOL 60 Revised Report (excerpt)

```
2.4 IDENTIFIERS

2.4.1 Syntax

 <identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

2.4.2 Examples

 q Soup V17a a34kTMNs MARILYN

2.4.3 Semantics

 Identifiers have no inherent meaning, but serve for the
 identification of simple variables, arrays, labels, switches, and
 procedures.  They may be chosen freely (cf. however section 3.2.4
 STANDARD FUNCTIONS).

 The same identifier cannot be used to denote two different quantities
 except when these quantities have disjoint scopes as defined by the
 declarations of the program (cf. section 2.7. QUANTITIES, KINDS and
 SCOPES and section 5. DECLARATIONS).
```

Syntax expressed formally. **Semantics** and **Pragmatics** expressed informally.

BNF - Backus-Naur-Form

```
digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
conditional statement ::=
   if comparison then series fi
   | if comparison then series else series fi
```

- A BNF grammar is a set of **production rules**.
- The lhs of a rule consists in some non-terminal symbol.
- The rhs of a rule consists on one or more alternatives.
- Each rule **alternative** is a sequence of non-terminal symbols and symbols.
- The BNF grammar describes **both** the set of valid programs and also their structure.

BNF - Non-terminal symbols

- Marked with **<** and **>** in the metalanguage.
- Each non-terminal represents a **mini language**.
- Each non-terminals must appear at the lhs of some rule.
- Otherwise the BNF grammar is **not well formed**.

BNF - Terminal symbols

- Tokens in the subject language.
- Different languages conceive tokens in a different way.
- Terminals must be marked in the metalanguage.

BNF - Recursion

- **Recursion** is the device that allows the generation of an **infinite** language from a **finite** BNF description.
- Recursion may be **direct** or **indirect**.
- $L = \{a^{2n+1} | n \geq 0\}$
  - Right recursion: <a> ::= aa<a> | a
  - Left recursion: <a> ::= <a>aa | a
  - Center recursion: <a> ::= a<a>a | a
- Recursive rules are **well formed** as long as there is a way to terminate the recursion.
- Recursion in BNF plays a similar role than **unbounded loops** in algorithmic languages.
- Simple concatenation plays a similar role than **bounded loops** in algorithmic languages.

BNF - Example

Arithmetic expressions in BNF

```
adding operator ::= + | -
multiplying operator ::= * | /
primary ::= unsigned number | variable
                | function designator | (arithmetic expression)
factor ::= primary | factor ↑ primary
term ::= factor | term multiplying operator factor
simple arithmetic expression ::=
          term
        | adding operator term
        | simple arithmetic expression adding operator term
if clause ::= if boolean expression then
arithmetic expression ::=
          simple arithmetic expression
        | if clause simple arithmetic expression
          else arithmetic expression
```

Meanwhile, in Monkey Island...

Independent work by Noam Chomsky in the field of formal linguistics leads to **generative grammars**.

$G = (E, N, S, F)$ where

E is the alphabet of terminal symbols.
N is the alphabet of non-terminal symbols.
S is the start symbol.

F is a set of production rules (P, Q) such that P and Q are in $(E \cup N)*$ and P contains at least one symbol from N.

- Each grammar generates some language.
- Generative grammars can also act as **recognizers**.
- Finite descriptions of potentially infinite languages.

Chomsky's hierarchy of generative grammars

- Type 3: **regular** grammars.
- Type 2: **context free** grammars.
- Type 1: **monotonic** or **context-sentitive** grammars.
- Type 0: **phrase-structured** grammars.

Turns out BNF and type 2 **context-free** grammars have the same expressive power!

Hierarchy of languages

- Type 3: **regular** languages.

$$L1 = \{a^n | n \geq 0\}$$

- Type 2: **context free** languages.

$$L2 = \{a^n b^n | n \geq 0\}$$

- Type 1: **monotonic** or **context-sentitive** languages.

$$L3 = \{a^n b^n c^n | n \geq 0\}$$

- Type 0: **phrase-structured** languages.
  - It has been proven that there are **type 0** languages that are not type 1, but not a particular instance of such a language has been even found.

Hierarchy of abstract machines

Automaton capable of **recognizing** the corresponding class of languages.

- Type 3: **Finite-state machines**
- Type 2: **Push-down automatas**.
- Type 1: **Linear-bounded automatons**.
- Type 0: **Turing machines**.

COBOL 60

ISO 1989-1985 programming language - COBOL

6.3.1.5 Sign Condition

The sign condition determines whether or not the algebraic value of an arithmetic expression is less than, greater than, or equal to zero. The general format for a sign condition is as follows:

arithmetic-expression-1 IS [NOT] { POSITIVE / NEGATIVE / ZERO }

When used, NOT and the next key word specify one sign condition that defines the algebraic test to be executed for truth value; e.g., NOT ZERO is a truth test for a nonzero (positive or negative) value. An operand is positive, if its value is greater than zero, negative if its value is less than zero, and zero if its value is equal to zero. Arithmetic-expression-1 must contain at least one reference to a variable.

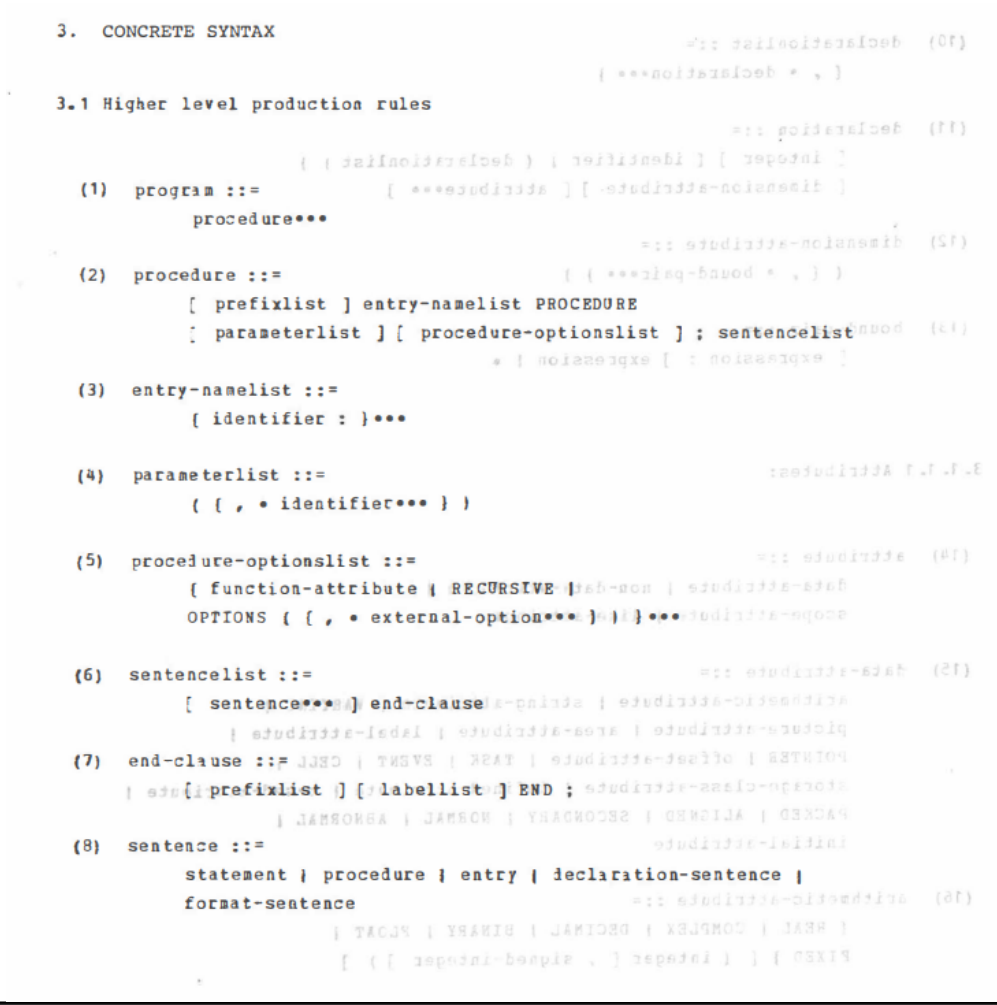
Syntax expressed formally in an extended BNF. **Semantics** and **Pragmatics** expressed informally.

COBOL extensions to BNF

- Brackets **[...]** denote options.
- Braces **{...}** denote grouping, via vertical stacking.
- Other convenient notational extensions ...

PL/I

Concrete Syntax of PL/I (1968)



3. CONCRETE SYNTAX

3.1 Higher level production rules

(1) program ::=
procedure•••

(2) procedure ::=
[prefixlist] entry-namelist PROCEDURE
[parameterlist] [procedure-optionslist] ; sentencelist

(3) entry-namelist ::=
{ identifier : }•••

(4) parameterlist ::=
({ , • identifier••• })

(5) procedure-optionslist ::=
{ function-attribute | RECURSIVE |
OPTIONS ({ , • external-option••• }) }•••

(6) sentencelist ::=
[sentence•••] end-clause

(7) end-clause ::=
[prefixlist] [labellist] END ;

(8) sentence ::=
statement | procedure | entry | declaration-sentence |
format-sentence

IBM’s attempt to produce a clear and readable *a posteriori* specification of PL/I using a context-free syntax.

EBNF - extended BNF

- Incorporates BNF and COBOL formalism, adding additional extensions.
- Factoring of strings.
- Optional items.
- Simple lists via recursion.
- Punctuated list via recursion.

EBNF - factoring of strings

'V ::= S_1 T_1 S_2 | S_1 T_2 S_2 | ... | S_1 T_n S_2'

may be replaced by

'V ::= S_1 { T_1 | T_2 | ... | T_n } S_2' and vice versa

(note, that this rule remains, valid also for the case n = 1)

Example: goto-statement ::= { GOTO | GO TO } reference ;

instead of: goto-statement ::= GOTO reference ; | GO TO reference ;

EBNF - optional items

'V ::= S_1 S_2 | S_1 T_1 S_2 | ... | S_1 T_n S_2'

may be replaced by

'V ::= S_1 [T_1 | ... | T_n] S_2' and vice versa.

Example: return-statement ::= RETURN [(expression)] ;

instead of:return-statement ::= RETURN ; | RETURN (expression) ;

EBNF - simple lists via recursion

'V ::= U | V U' or

'V ::= U | U V'

may be replaced by

'V ::= U•••' and vice versa.

Note : For the inversion it is to be regarded that if 'U•••' occurs in the grammar and no production rule has the form 'V ::= U•••', this missing rule is to be added to the grammar with a not yet used variable 'V' before the replacement can be performed.

Example: integer ::= digit•••

instead of:integer ::= digit | integer digit

EBNF - punctuated lists via recursion

'V ::= S_1 T_1 [{ T_2 T_1 } •••] S_2' may be replaced by

'V ::= S_1 { T_2 • T_1 ••• } S_2' and vice versa.

Note: Instead of 'S_1 [{ T_2 • T_1 ••• }] S_2' also

'S_1 [T_2 • T_1 •••] S_2' may be written.

Example: declarationlist ::= { , • declaration ••• }

instead of: declarationlist ::= declaration [{ , declaration } •••

Reconsidering semantics

```
2.4 IDENTIFIERS

[...]

2.4.3 Semantics

 Identifiers have no inherent meaning, but serve for the
 identification of simple variables, arrays, labels, switches, and
 procedures.  They may be chosen freely (cf. however section 3.2.4
 STANDARD FUNCTIONS).

 The same identifier cannot be used to denote two different quantities
 except when these quantities have disjoint scopes as defined by the
 declarations of the program (cf. section 2.7. QUANTITIES, KINDS and
 SCOPES and section 5. DECLARATIONS).
```

How much of the described *semantics* are in reality **syntax**?
How much of the syntax conveys meaning and is therefore **semantics**?

Static and dynamic semantics

- **Static semantics** cover meaning that can be entirely conveyed in the structure of the program text, as complicated as that structure might be. This meaning takes the form of **context conditions**.
- **Dynamic semantics** cover meaning that relies on the computation of some value that can only be determined at run-time.

How to formalize static semantics?

- Either you handle them like dynamic semantics, meaning using a method based on **abstract syntax**. This includes operational semantics, denotational semantics, axiomatic semantics, action semantics, and others techniques.
- Or you subsume the semantics in the formalization of the **concrete syntax**. In principle you would need to turn the grammar into a **context-sensitive grammar**.

The absolute misery of context-sensitive grammars

CS grammar for $L = \{a^n b^n c^n | n \geq 1\}$

```
G = (E,N,S,P)

where

  E = {a, b, c}
  N = {S, T, B, C, D}
  S = S

  P = { 1: S -> T
        2: T -> aTBD
        3: T -> abD
        4: DB -> CB
        5: CB -> CD
        6: CD -> BD
        7: bB -> bb
        8: D -> c }
```

But wait, there is hope ...

At univ we were taught that:

- Type 3 (regular) grammars cannot express nesting.
- Type 2 (context-free) grammars cannot express context sensitivity.

But **this is not exactly so**, because:

- Regular grammars can express a **finite** amount of nesting.
- Context-free grammars can express context sensitivity within a **finite** reach.

Infinite context-free grammars

Infinite CF grammar for $L = \{a^n b^n c^n | n \geq 1\}$

```
G = (E,N,S,P)

where

E = {a,b}
N = {S}
S = S

F = { 1. S -> abc
      2. S -> aabbcc
      3. S -> aaabbbccc
      4. S -> aaaabbbbcccc
      ... }
```

- Infinite CF grammars have **infinite non-terminals** and/or **infinite production rules**.
- But how can we formally denote infinite grammars (which are supposed to be finite descriptions of potentially infinite languages) in a finite way? Aadrian van Wijngaarden found the solution: by **thinking in two levels**.

The vision of Aad van Wijngaarden



- Champion of **generalization** and its corollary, **orthogonality**.
- Thinking on two levels: **VW grammars**.
- Unbreakable bond between the language and its description.
- Some publications:
  - **1963** - Generalized ALGOL.
  - **1965** - Orthogonal design and description of a formal language.
  - **1966** - Recursive Definition of Syntax and Semantics.
  - **1979** - Thinking on two levels.

Algol 68

Algol 68 Revised Report (excerpt)

5.4.4.1. Syntax

a) **strong MOID NEST jump{5A} : go to{b} option,**
label NEST applied identifier with TAG{48b}.

b) **go to{a} : STYLE go to{94f,-} token ;**
STYLE go{94f,-} token, STYLE to symbol{94g,-}.

{Examples:

a) ***goto** kootwijk • **go to** warsaw • zandvoort*
b) ***goto** • **go to** }*

5.4.4.2. Semantics

A **MOID-NEST-jump** J, in an environ E, is elaborated as follows:
• let the scene yielded in E by the **label-identifier** of J be composed of a **series** S2 and an environ E1;
Case A: '**MOID**' is not any '**procedure yielding MOID1**':
• let S1 be the **series** of the smallest {1.1.3.2.g} **serial-clause** containing S2;
• the elaboration of S1 in E1, or of any series in E1 elaborated in its place, is terminated {2.1.4.3.e};
• S2 in E1 is elaborated "in place of" S1 in E1;
Case B: '**MOID**' is some '**procedure yielding MOID1**':
• J in E {is completed and} yields the routine composed of
(i) a new **MOID-NEST-routine-text** whose **unit** is akin {1.1.3.2.k} to J,
(ii) E1.

Syntax and **Static semantics** expressed formally in a VW grammar. **Dynamic semantics** and **Pragmatics** expressed informally.

VW grammars

VW grammar for $L = \{a^n b^n c^n | n \geq 1\}$

```
A) LETTER :: letter a; letter b; letter c.
B) TALLY :: i ; TALLY i.

a) identifier : TALLY letter a, TALLY letter b, TALLY letter c.
b) TALLY i LETTER : TALLY LETTER, i LETTER.
c) i LETTER : LETTER symbol.
```

- Founding member of the family of **two-level grammars**.
- Capable of expressing context-sensitive languages in an **intelligible** way.
- Expresiveness power compared to a type 0 Chomsky grammar.

VW grammars

Behold!

- Protonotions.
- Notions.
- Symbols.
- Representation table.
- Metanotions.
- Metaproduction rules.
- Hyper-rules.
- Hypernotion.
- Consistent substitution.
- Paranotions.

Don’t panic, we go step by step :)

VW grammars - protonotions/notions/symbols

- A **protonotion** is a non-empty sequence of **small syntactic marks**, possibly interspaced with blanks.

```
foo, foo bar
```

- When a protonotion appears at the lhs of a rule, it becomes a **notion** and plays the role of a BNF non-terminal.

```
program : program token, strong integral new closed clause.
```

- When a protonotion ends in symbol it becomes a **symbol** and plays the role of a BNF terminal.

```
if symbol
letter a symbol
```

- A protonotion that is not a notion nor a symbol is a **blind alley**, which plays a very important role in VW grammars.

VW grammars - representation table

- Every VW grammar has a **representation table** that is required to provide a suitable representation for every symbol that appears in the grammar.

a letter symbol	a
bold a letter symbol	**a**
digit nine symbol	9
`if` symbol	if
`true` symbol	true
exp symbol	↑

- This makes it possible to abstract the metalanguage from representation aspects such as stropping. This keeps it simple and uniform.

VW grammars - production rules

```
multiplying operator : mul symbol ; div symbol.
term : factor ; term, multiplying operator, factor.
factor : primary ; factor, exp symbol, primary.
```

- A colon separates the lhs of the rule from its lhs.
- Semicolons separate **alternatives**.
- Commas separate **elements** in an alternative.

VW grammars - metanotions, metaproduction rules

```
A) LETTER :: letter x; letter y; letter z.
B) TALLY :: i ; TALLY i.
C) LEAP :: local ; heap ; primal.
```

- A **metanotion** is a non-empty sequence of **big syntactic marks**, with no intermixed spaces.
- A metanotion denotes a possibly infinite set of protonotions.
- This set is generated by a context-free grammar composed of **metaproduction rules**.

VW grammars - hypernotions, hyper-rules, hyperalternatives

- A **hypernotion** is a mix of protonotions and metanotions.

```
LETTER TAG unique
LETTER symbol
MOINE identity
```

- A rule that contains hypernotions is a **hyper-rule**.

```
a) reference to MODE NEST LEAP generator :
     LEAP token, actual MODE NEST declarer.
```

- Hyper-rules are **templates** of production rules, that are expanded following the rule of **consistent substitution**.

VW grammars - Example

VW grammar for $L = \{a^n b^n c^n | n \geq 1\}$

```
A) LETTER :: letter a; letter b; letter c.
B) TALLY :: i ; TALLY i.

a) identifier : TALLY letter a, TALLY letter b, TALLY letter c.
b) TALLY i LETTER : TALLY LETTER, i LETTER.
c) i LETTER : LETTER symbol.

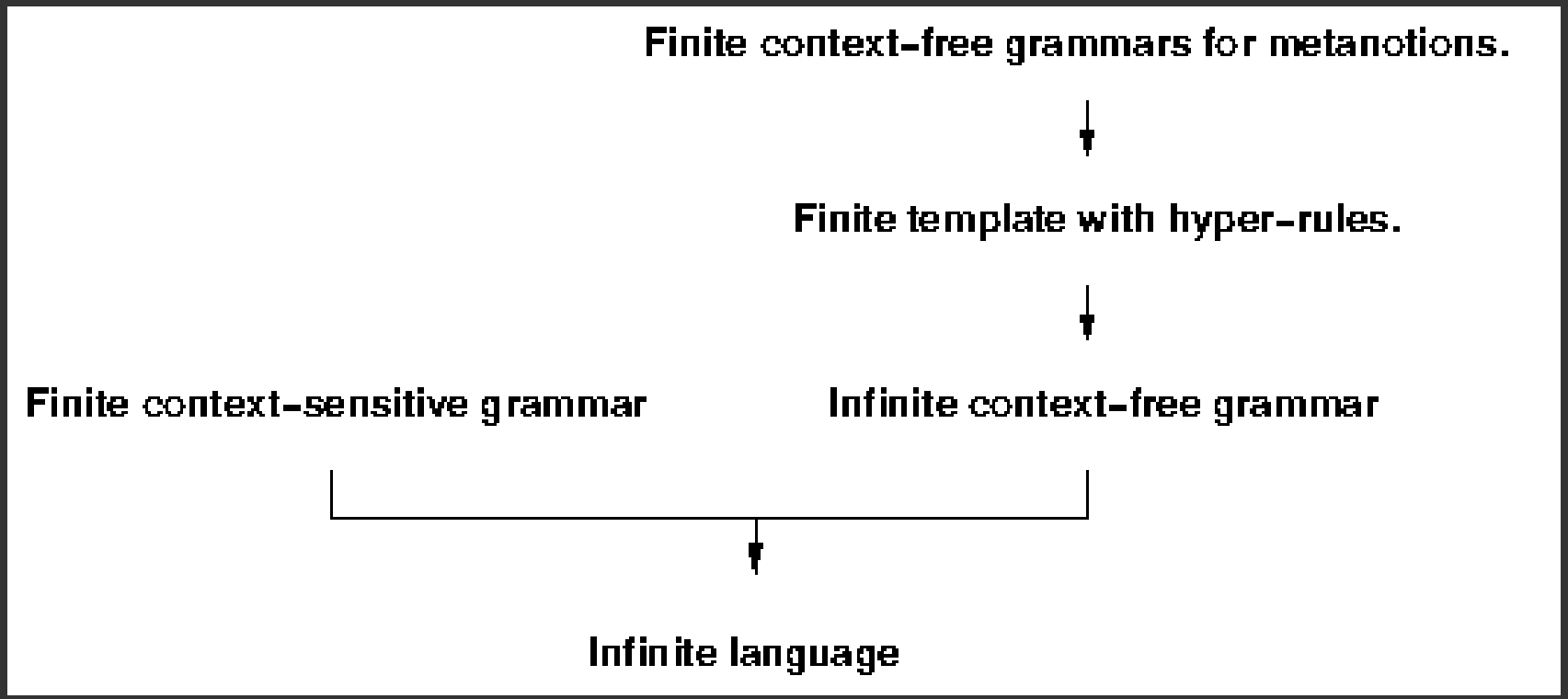
Expands to the infinite CF grammar:

identifier : i letter a, i letter b, i letter c.
identifier : ii letter a, ii letter b, ii letter c.w
...

ii letter a : i letter a, i letter a.
ii letter b : i letter b, i letter b.
ii letter c : i letter c, i letter c.
iii letter a : ii letter a, i letter a.
iii letter b : ii letter b, i letter b.
iii letter c : ii letter c, i letter c.
...

i letter a : letter a symbol.
i letter b : letter b symbol.
i letter c : letter c symbol.
```

VW grammars - the two levels



VW grammars - predicates

- Technique for expressing **context conditions**.
- A **predicate** is a hypernotion that leads to either an **invisible production**, in which case the predicate holds, or to a **blind alley**, in which case the predicate doesn't hold.
- Predicates are implemented as **a set of hyper-rules**.

VW grammars - predicates - example 1

VW grammar for double letters

```
A) ALPHA :: a ; b ; c ; d ; e ; f ; g ; h ; i
            j ; k ; l ; m ; n ; o ; p ; q ; r
            s ; t ; u ; v ; w ; x ; y ; z.
B) EMPTY :: .

a) s : letter ALPHA1 symbol,
       letter ALPHA2 symbol,
       where ALPHA1 is ALPHA2.

b) where ALPHA is ALPHA : EMPTY.
```

Note how ALPHA1 and ALPHA2 are distinct metanotion. By convention it is assumed

```
ALPHA1 :: ALPHA.
ALPHA2 :: ALPHA.
```

VW grammars - predicates - example 2

VW grammar for two ordered letters

```
A) ALPHA :: a ; b ; c ; d ; e ; f ; g ; h ; i
            j ; k ; l ; m ; n ; o ; p ; q ; r
            s ; t ; u ; v ; w ; x ; y ; z.
B) NOTETY :: EMPTY ; NOTETY ALPHA.
C) EMPTY :: .

a) s : letter ALPHA1 symbol,
       letter ALPHA2 symbol,
       where ALPHA1 precedes ALPHA2 in
          abcdefghijklmnopqrstuvwxyz.

b) where ALPHA1 precedes ALPHA2 in
      NOTETY1 ALPHA1 NOTETY2 ALPHA2 NOTETY3 : EMPTY.
```

Other two-level grammars

- Koster's Affix grammars.
- Extended Affix grammars (EAGs).
- Two-level grammars as programs: Dick Grüene's **Aleph**.

Attribute grammars

- Developed by Knuth, postdates VW grammars.
- **Not** a two-level grammar.
- Not complete: it needs an additional metalanguage.

Prologue

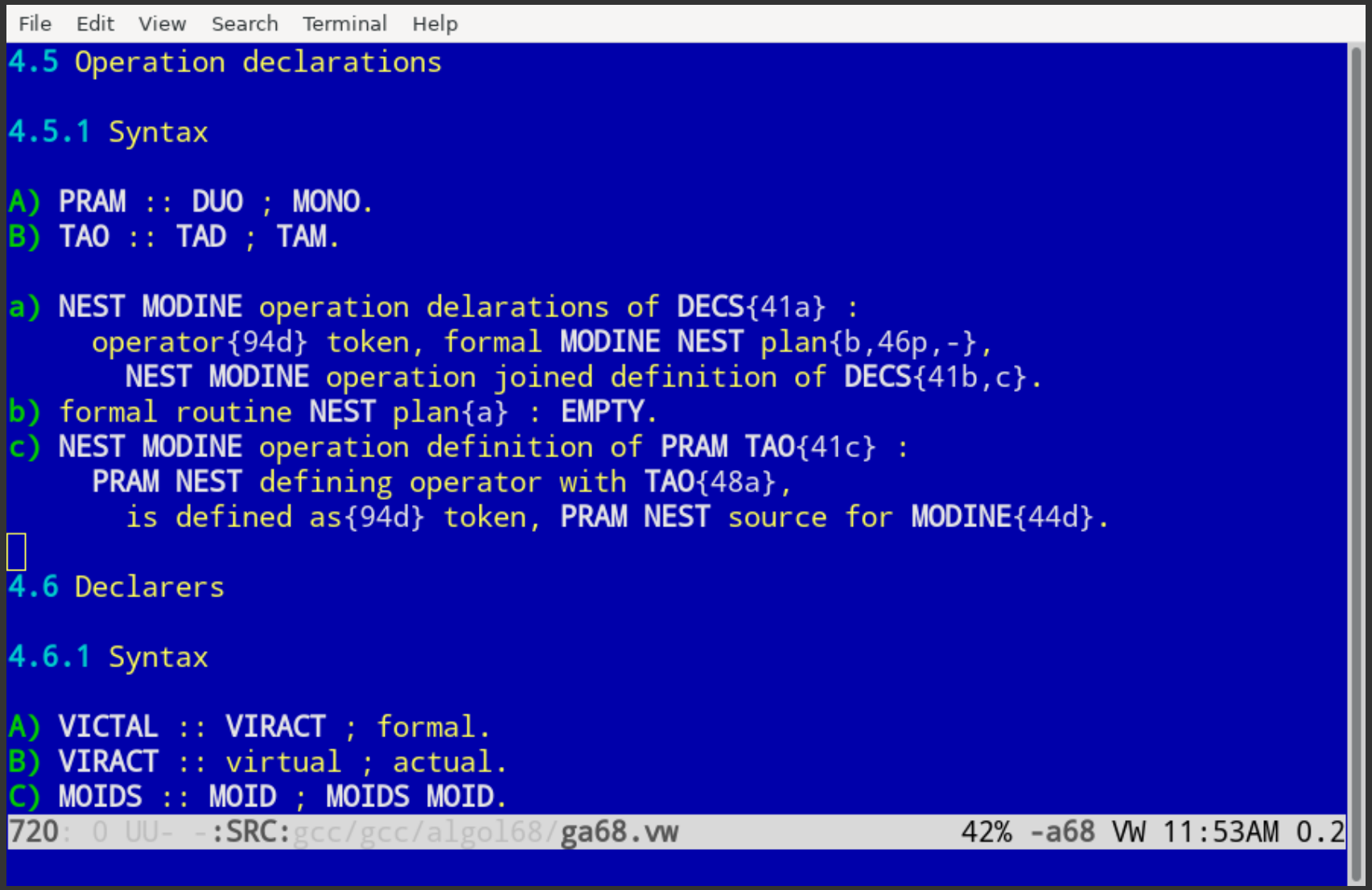
Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

The Emacs vw-mode



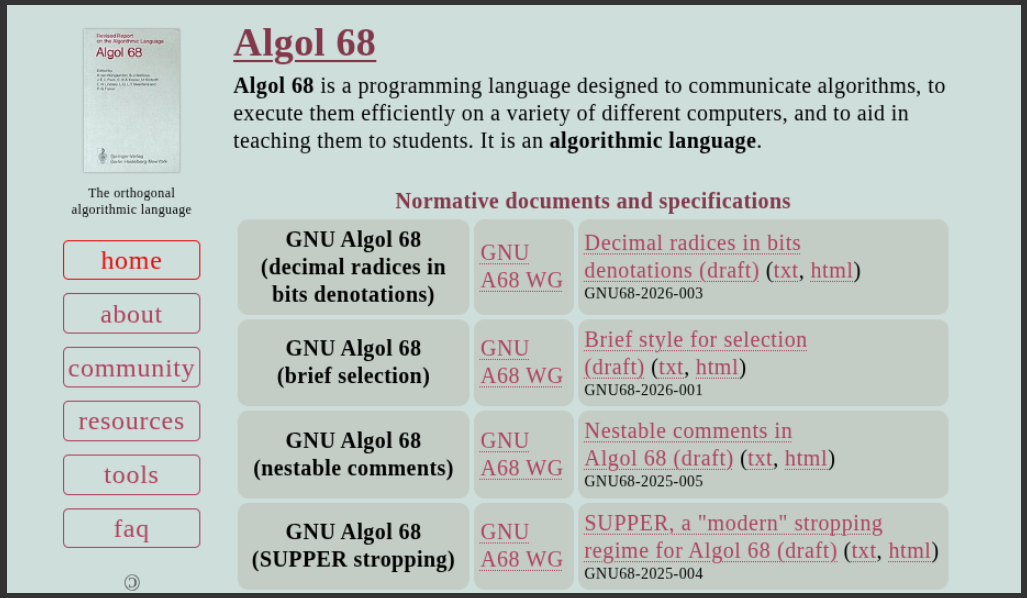
- Its purpose is to help writting language Reports.
- Available in GNU Elpa (M-xlist-packages).

The Algol 68 Revised Report

- Sectioning.
- Cross-references.
- Pragmatics.
- General metanotions.

GNU extensions to Algol 68

https://algol68-lang.org



- Each extension has a **formal description** and an **informal description**.
- Based on IFIP's TUM-10 process.

Prologue

Languages and their specification

Specifications and their formalisms

Static semantics formalization in practice

Epilogue

Bibliography

You will find a lot of this talk in the books and papers below.

- **Formal Specification of Programming Languages. A Panoramic Primer.** - Frank G. Pagan.
- **Grammars for Programming Languages** - J. Craig Cleaveland, robert C. Uzgalis.
- **Formal Syntax and Semantics of Programming Languages. A laboratory based approach.** - Kenneth Slonneger, Barry L. Kurtz.

GNU Algol 68

https://algol68-lang.org

Project with a mission:

- To bring **Algol 68** to the first line of programming and dispel the abundant myth and FUD surrounding it,
- to provide modern implementations of the language, well integrated in today's operating systems and computers,
- to produce documentation and other material to help people to learn the language, and
- to maintain, extend and evolve the language with the rigor and seriousness that it deserves and demands.

Not officially associated with the IFIP WG 2.1, which is still in charge of the language, at least theoretically. Contacts have been established.