

Backus–Naur form

CSC 302, “Programming language concepts”

January 30, 2009

The *syntax* of a programming language specifies the structure and arrangement of the elements in the text of programs written in that language. The designers and implementers of FORTRAN, the first high-level programming language, described its syntax in English prose, supplemented by examples.

Early FORTRAN programs were written on punched cards, and the syntax of the language reflects this. A FORTRAN program consists of more or less independent *statements*, normally in one-to-one correspondence with cards. Each kind of statement has an idiosyncratic syntax, which is either completely fixed or has only a finite number of variants. FORTRAN statements are not nested.

For example, FORTRAN has an analogue of the `for`-statement as we know it from languages like Java and C, but in FORTRAN the body of the loop is not considered to be part of the `for`-statement; it’s just a sequence of statements, like any other section of a FORTRAN program. To arrange for the repetition, the FORTRAN programmer adds a label to the last statement in the sequence and prepends a special `DO`-statement at the beginning. The `DO`-statement specifies four things: the loop-control variable, its initial value, its final value, and the label of the last statement. The FORTRAN compiler replaces the `DO`-statement with the code for initializing the loop-control variable and adds, after the labelled statement, code for updating and testing the loop-control variable and for making the jump back to the top of the loop.

The statements making up the sequence to be repeated were not necessarily perceived as “belonging” to the loop, and in fact it was not uncommon for programmers to direct control out of this region and back in again if they needed to execute a few additional statements that it was convenient to write elsewhere.

This conception of the program as a flat list of independent statements turned out to be too limiting in some respects and too flexible in others.

The purpose and function of the notation

The syntactic constituents of a program form a hierarchy, with the elements (such as identifiers, operator symbols, keywords, and literal constants) at the bottom and progressively larger constituents at higher levels. These constituents are thought of as belonging *syntactic categories*, with members of the same category having similar syntactic roles and functions. One can represent the structure of a program as a *syntax tree* in which the leaf nodes are syntactic elements and each internal node is labelled with the syntactic category of the constituent formed by the leaves that descend from it.

The Backus–Naur formalism provides a concise way of describing possible modes of combination of constituents. It has an additional advantage for implementers of languages: It shows us exactly how to build the recursive data structures for syntax trees.

The fundamental idea

The fundamental idea is to describe the syntax of a programming language as a set of *rules* (sometimes called *productions*). Each rule describes one possible way of constructing a constituent belonging to a particular syntactic category, which is named at the beginning of the rule. We'll write the names of syntactic categories within angle brackets, thus: $\langle \text{program} \rangle$. The right-hand side of the rule then lists, in order, the components of the constituents; these may include syntactic elements, other syntactic categories, or both. Any number of components can appear in the right-hand side of a rule. The syntactic category at the beginning of the rule is separated from the right-hand side by the (more or less arbitrary) symbol $::=$.

Two or more rules beginning with the same syntactic categories may be combined into one by concatenating their right-hand sides, separating the alternative constructions with vertical bars, thus:

```
 $\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{lambda-expression} \rangle \mid \langle \text{procedure-call} \rangle$ 
```

The reader must recognize that the vertical bar in such cases, like the $::=$ symbol, is part of the *meta*-notation, not part of the programming language that is being described. Indeed, all of these extensions involve additions to the meta-notation; when they are all deployed at once, it can sometimes be troublesome to distinguish the BNF symbols from syntactic elements of the object language. This is one reason why some authors prefer to enclose syntactic elements in quotation marks.

There are many trivial variants of this notation. Sometimes syntactic elements are enclosed in quotation marks and the names of syntactic categories are written without the angle brackets. Sometimes an arrow of some sort is used in place of the ::= symbol.

Examples

We've already seen some examples of BNF in our textbook. For instance, here's the concrete syntax of expressions of the λ -calculus:

```
<lc-expression> ::= <lc-identifier>
                  | ( lambda ( <lc-identifier> ) <lc-expression> )
                  | ( <lc-expression> <lc-expression> )
```

Dropping all of the fixed symbols gives us the corresponding abstract syntax:

```
<lc-expression> ::= <lc-identifier>
                  | <lc-identifier> <lc-expression>
                  | <lc-expression> <lc-expression>
```

When we build our own syntax trees in Scheme, we'll write a data type definition for each non-trivial syntactic category, with a variant for each rule. The right-hand side of the BNF rule tells us what the fields of the corresponding variant should be—each field corresponds to one of the non-fixed constituents.

It is theoretically possible for the right-hand side of a BNF rule to be empty:

```
<empty> ::=
```

In practice, however, this possibility is seldom used. (For one thing, it messes up the correspondence with syntax trees, since a node labelled with a syntactic category like `<empty>` is technically a leaf node but contributes no syntactic element to the visible text of the program.)

Extensions

Plain vanilla BNF is somewhat cumbersome. For practical use, the notation is generally extended in various ways. Here are the ones that are used in our textbooks:

It often happens that two rules for the same syntactic category differ only in the presence or absence of some optional component, as in this example from Java:

```
<break-statement> ::= break ;  
                    | break <identifier> ;
```

Such a pair of rules can be combined by enclosing the optional component in braces and writing a question mark after it, thus:

```
<break-statement> ::= break {<identifier>}? ;
```

Again, it often happens that a constituent may contain any number of successive components of the same category. We'll specify this in our extended Backus–Naur formalism by enclosing the repeatable component in braces and writing a star after it:

```
<procedure-call> ::= ( <expression> {<expression>}* )
```

In unextended BNF, one would get roughly the same effect by providing a separate category for arbitrary-length sequences of the repeatable component and analyzing it thus:

```
<procedure-call> ::= ( <expression> <expression-sequence> )  
<expression-sequence> ::= <empty>  
                        | <expression> <expression-sequence>
```

This is much like the recursive definition of a list as either `null` or a pair consisting of one element (the `car`) and a list (the `cdr`). Note that the syntax trees constructed according to the unextended BNF rule look different from those constructed according to the star rule—none of the `<expression>` nodes for the operands will be children of the `<procedure-call>` node if the unextended BNF rule is used.

The star notation allows for the possibility that the repeatable component does not appear at all. (In our example, a procedure call may contain no operands.) If we want to specify a repeatable component that must occur at least once, we'll use a plus sign in place of the star, as in this example from the Icon programming language:

```
<program> ::= {<declaration>}+
```

In other words, an Icon program consists of one or more declarations. Again, we could get a similar effect in unextended BNF:

```

<program> ::= <declaration-sequence>
<declaration-sequence> ::= <declaration>
                           | <declaration> <declaration-sequence>

```

Finally, it often happens that a construction can include any number of successive occurrences of some component, except that if there are two or more of them, any two that are adjacent must be separated by some fixed symbol. For instance, the compound statement in Algol 60 normally contains several statements separated by semicolons:

```

begin
  temp := first;
  first := second;
  second := temp
end

```

Notice that the semicolon is a separator, not (as in Java) a terminator. It would be an error to put a semicolon between the second occurrence of `temp` and the keyword `end`. A compound statement in Algol 60 can contain just one statement in its body:

```

begin
  temp := first
end

```

And it may contain none at all:

```

begin
end

```

No semicolon is needed in these cases, since there are no two adjacent component statements to be kept separate.

We'll express this separated-repetition rule using a variant of the star notation, placing the separator symbol in braces after the star:

```

<compound-statement> ::= begin {<statement>}*{;} end

```

Data type definitions for extensions

All of the extensions just described can be handled within the framework provided by `define-datatypes` and `cases`.

For the question-mark extension, it's usually simpler to expand it into two variants:

```
(define-datatype break-expression break-expression?
  (simple-break ())
  (targeted-break (target identifier?)))
```

When there is more than one optional component in the same rule, so that separating all the variants would lead to a combinatorial explosion, one can provide a single variant but allow some kind of an absence marker to fill any or all of the optional positions:

```
<for-stmt> ::= for ( {<expr>}? ; {<expr>}? ; {<expr>}? ) <stmt>
```

```
(define-datatype for-stmt for-stmt?
  (a-for-stmt (initializer (maybe expr?))
              (entry-test (maybe expr?))
              (updater (maybe expr?))
              (for-body stmt?)))
```

```
;;; Given a unary predicate, the maybe procedure constructs
;;; and returns a new predicate that is satisfied by everything
;;; that satisfies the given predicate, but also by the symbol
;;; absent (conventionally signalling an absent optional component
;;; of a syntactic construction).
```

```
;; maybe : (SchemeVal -> Bool) -> (SchemeVal -> Bool)
(define maybe
  (lambda (predicate)
    (lambda (something)
      (or (predicate something)
          (eqv? something 'absent)))))
```

As our textbook notes (in exercise 2.29), when a Kleene star or plus is used in the concrete syntax, it is usual for the data type definition to provide a single field containing a list of the components belonging to the iterated category:

```
;; Scheme procedure calls
(define-datatype procedure-call procedure-call?
  (a-procedure-call (operator expression?)
                    (operands (listof expression?))))
```

```
;; Icon programs
(define-datatype program program?
  (a-program (declarations (listof declaration?))))
```

```
; ; Algol 60 compound statements
(define-datatype compound-statement compound-statement?
  (a-compound-statement (body (listof statement?))))
```

In the last example, note that the semicolons between statements in a compound statement do not appear at all in abstract syntax trees. Like other fixed components, they are not represented in data type definitions.

The name of the notation

Backus–Naur form is named for John Backus (1924–2007) and Peter Naur (b. 1928). Backus, who headed the team that developed the first high-level programming language (FORTRAN), devised the formalism in 1959. Naur helped to popularize it by adapting and using it in the definition of the Algol 60 programming language.

Exercises

Exercise A.3 [★] Describe the Algol 60 compound statement (see page 5) in *unextended* BNF.

Exercise A.4 [★] The basic form of Java `for`-statement is the one shown in the example on page 6. Recent versions of Java also support an “enhanced” form of the `for`-statement, with the concrete syntax

```
<for-stmt> ::= for ( <type> <identifier> : <expr> ) <stmt>
```

Revise the data type definition for `for-stmt` on page 6 so that it supports both basic and enhanced forms (as variants).

Exercise A.5 [★★] The variable-declaration part of a Pascal program, procedure definition, or function definition consists of the keyword `var` followed by one or more declarations, each consisting of one or more identifiers (separated by commas), a colon, and either an identifier or a constructed type. Each declaration is terminated by a semicolon.

Translate this description into one or more BNF rules. (You can assume that the category `<constructed-type>` is defined elsewhere.) Then write a data type definition for it.

Exercise A.6 [★★★] A grammar for the Lua programming language, expressed in extended BNF, can be found at the end of the Lua 5.1 reference manual

(<http://www.lua.org/manual/5.1/manual.html>). Construct data type definitions for the syntactic categories defined there.

Note that the manual's authors use square brackets to enclose optional components and curly braces to enclose components that can be repeated any number of times. So the extended BNF rule that they write as

```
funcname ::= Name {'.' Name} [':' Name]
```

would, if written using the conventions adopted in this handout, appear instead as

```
<funcname> ::= <name> { . <name> } * { : <name> } ?
```

This work is licensed under the Creative Commons Attribution–ShareAlike 3.0 United States License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/3.0/us/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.