

Specification of Programming Language Syntax

by

Martin Richards

`mr10@cl.cam.ac.uk`

`http://www.cl.cam.ac.uk/users/mr10/`

Computer Laboratory

University of Cambridge

Revision date: Sun Dec 29 05:41:56 AM GMT 2024

Abstract

Normally Backus Naur Form (BNF) or extended BNF (EBNF) is used to specify the syntax of programming languages. Users typically find such specifications hard to interpret and it is often not easy to check whether a parser conforms precisely with the BNF specification. There is also the problem that the BNF may be ambiguous.

This document suggests way to specify the syntax of programming languages that is concise and easily understood by users. It essentially specifies a recursive descent parsing algorithm making it easy to implement a parser that conforms precisely to the specification. Syntax specified in this way is guaranteed to be unambiguous.

Keywords: BNF, EBNF, transition diagrams, ambiguity.

Chapter 1

Introduction

Backus Naur Form (BNF) was first used in the Algol 60 Report to give a precise description of its syntax.

John Backus and Peter Naur were members of the committee that developed Algol 60.

Since then almost all programming languages have had their syntax specified using BNF or Extended BNF (EBNF). EBNF is a slightly more compact notation to specify a BNF grammar.

Algol 60 was a tour de force since at that time the most popular languages were COBOL, FORTRAN 66, LISP and various Autocodes.

1.1 Example BNF grammar for expressions

```

Bexp ::= name
       number
       ( E )
       + E
       - E

```

```

E  ::= Bexp
      E ^ E
      E * E
      E / E
      E + E
      E - E

```

An example mathematical expression

$$x * 2^{-n^2} - 16 / -2 * 3$$

can be written satisfying the above grammar as follows:

$$x * 2 ^{-n^2} - 16 / -2 * 3$$

1.2 Syntax is difficult

At about the age of 7 we are taught that multiplication is more binding than addition. So $1 + 2 \times 3$ evaluates to 7 not 9.

I was told to remember the word BODMAS which stands for

Brackets Of Division Multiplication Addition and Subtraction

indicating that, for instance, multiplication is more binding than addition.

Unfortunately BODMAS is just wrong and misleading. For instance, what does it say about:

$$24 \times 6 \div 2 + 1 - 5 + 6$$

Or even:

$$x + y + z$$

Consider the following two assignments:

```
s = x + z;  
t = x + y + z;
```

```
Bexp := name  
      number  
      ( E0 )  
      + E1  
      - E1
```

```
E2 ::= Bexp  
      Bexp ^ E2
```

```
E1 ::= E2  
      E1 * E2  
      E1 / E2
```

```
E0 ::= E1  
      E0 + E1  
      E0 - E1
```

```
Prog ::= E0 <eof>
```

This grammar could be written using category names similar to those in the C++ grammar given in the ISO/IEC 14882:1998(E) specification

```
<basic expression> ::=
    <name>
    <number>
    ( <additive expression> )
    + <multiplicative expression>
    - <multiplicative expression>

<power expression> ::=
    <basic expression>
    <basic expression> ^ <power expression>

<multiplicative expression> ::=
    <power expressio>
    <multiplicative expression> * <power expression>
    <multiplicative expression> / <power expression>

<additive expression> ::=
    <multiplicative expression>
    <additive expression> + <multiplicative expression>
    <additive expression> - <multiplicative expression>

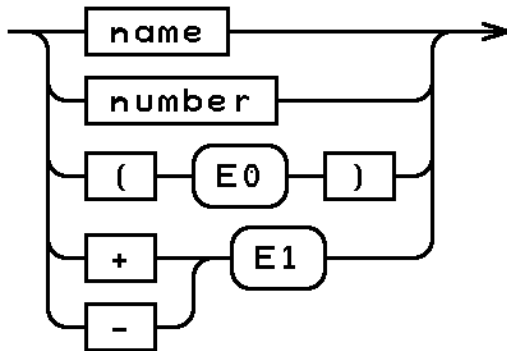
<program> ::=
    <additive expression> <eof>
```

This grammar has
6 syntactic categories, 14 productions and 10 terminal symbols.

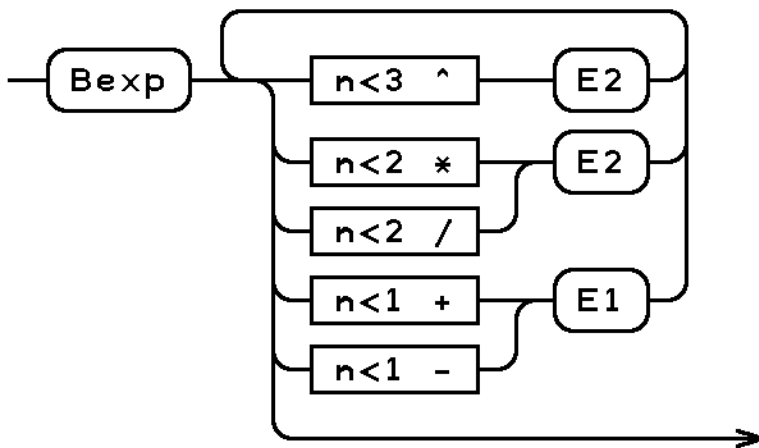
The ISO/IEC 14882:1998(E) grammar for C++ has
196 categories, 640 productions and 192 terminal symbols.

1.3 The transition diagrams

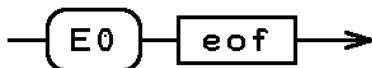
The definition of Bexp



A more compact definition of En

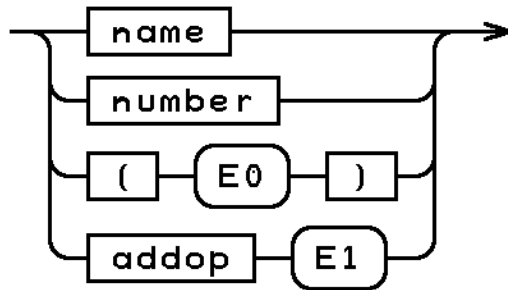


Definition of Prog

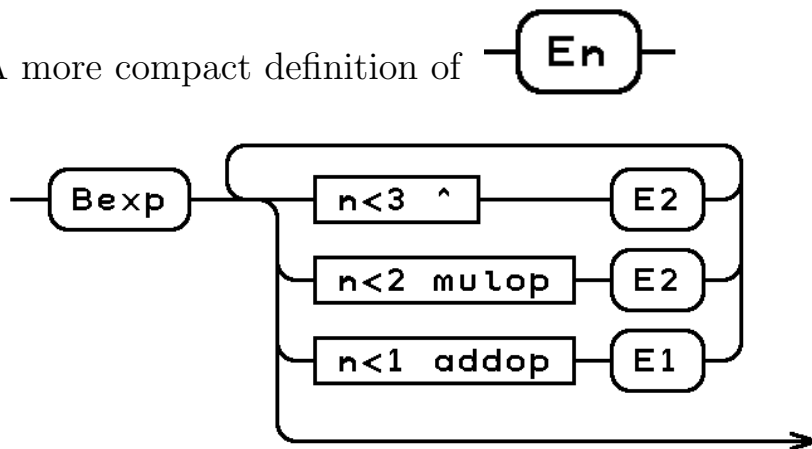


1.4 The compacted transition diagrams

A more compact definition of Bexp



A more compact definition of En



Definition of Prog



1.5 Implementing the Parser

```

LET rbexp() = VALOF
{ // Read a basic expression returning the corresponding parse tree
  LET op = token
  LET res = 0
  SWITCHON op INTO
  { DEFAULT:
    synerr("Bad token at the start of an expression")
    lex()
    RESULTIS 0

    CASE s_name:
    CASE s_number:
      res := mk2(op, lexval)
      lex()
      RESULTIS res

    CASE s_lparen:
      res := rnexp(0)
      checkfor(s_rparen, "'')' expected")
      RESULTIS res

    CASE s_add:
    CASE s_sub:
      res := rnexp(1)
      IF op=s_sub RESULTIS mk2(s_neg, res)
      RESULTIS res
  }
}

AND rnexp(n) = VALOF
{ lex()
  RESULTIS rexp(n)
}

```

```
AND rexp(n) = VALOF
{ LET res = rbexp()

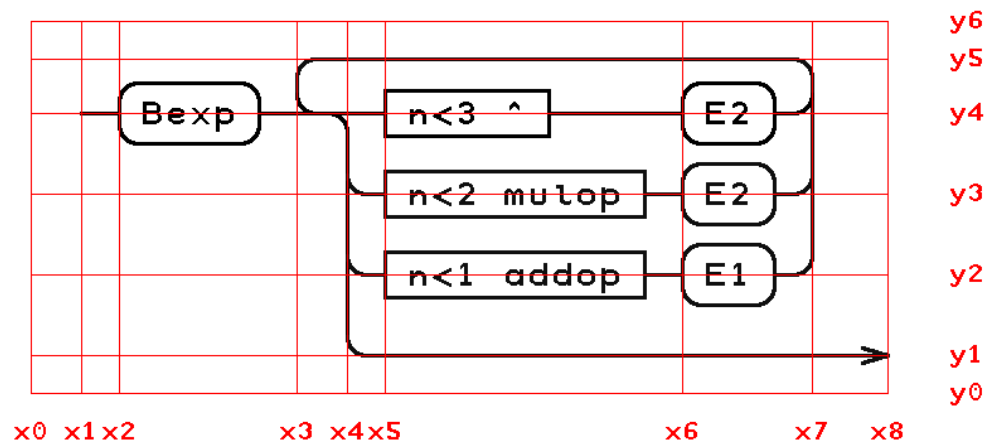
  { LET op = token
    SWITCHON op INTO
    { DEFAULT:
      RESULTIS res

      CASE s_power:
        UNLESS n<3 RESULTIS res
        res := mk3(op, res, rnexp(2))
        LOOP

      CASE s_mul:
      CASE s_div:
        UNLESS n<2 RESULTIS res
        res := mk3(op, res, rnexp(2))
        LOOP

      CASE s_add:
      CASE s_sub:
        UNLESS n<1 RESULTIS res
        res := mk3(op, res, rnexp(1))
        LOOP
    }
  } REPEAT
  RESULTIS res
}
```

1.6 How to draw the flow graphs



Code to draw the items at level y_4

```
drawcatboxL (y4, x1, x2, x5, "Bexp")
drawtestboxL(y4, x5, x5, x6, "n<3 ^")
drawcatboxL (y4, x6, x6, x7-r, "E2")
rndcorner(2, x3, y4, r)
rndcorner(0, x4, y4, r)
rndcorner(3, x7, y4, r)
moveto(x4, y4-r)
drawto(x4, y1+r)
```

1.7 Syntactic ambiguity

The rules associated with the flow graphs are:

- 1) Backtracking through a test box that matches a lexical token is not permitted.
- 2) At a path division point the left hand branch is always tried first.

The rules ensure the grammar specified by the flow graphs contains no ambiguities.

It is easy to create flow graphs in which every loop will match at least one lexical, and it is easy to check that this property holds.

Note that BNF grammars can contain ambiguities.

1.8 BNF Ambiguities

From the Web BNF I have obtained BNF specifications for the languages C, C++ and Java. All three essential contain the following productions.

```
<selection-statement> ::= if ( <expression> ) <statement>
```

```
<selection-statement> ::= if ( <expression> ) <statement> else <statement>
```

```
<statement> ::= <selection-statement>
```

This contains an ambiguity since there are two way to parse the following:

```
res = 0;  
if (x>0) if (y>0) res=2; else res = 1;
```

If x is zero this code will set `res` to either zero or one depending on how the author of the compiler interpreted the grammar.

This is exactly the same ambiguity that was present in the Algol 60 Report 64 years ago.

My conclusion is that language designers and/or people who write programming language manuals choose to specify the grammar using BNF but don't notice or care if the BNF specification is ambiguous.