BNF and EBNF

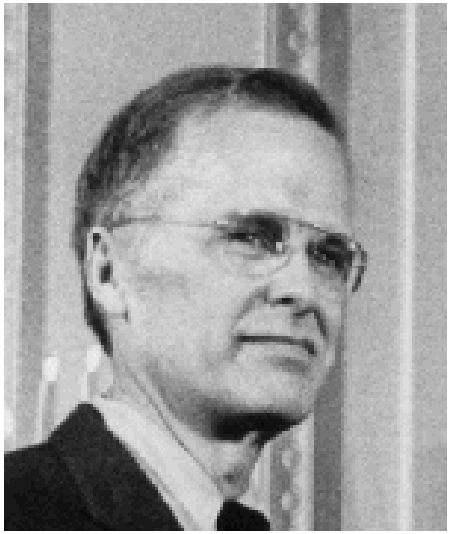
What is BNF?

- It stands for Backus-Naur Form
- It is a formal, mathematical way to specify context-free grammars
- It is precise and unambiguous
- Before BNF, people specified programming languages ambiguously, i.e., with English

How did BNF come about?

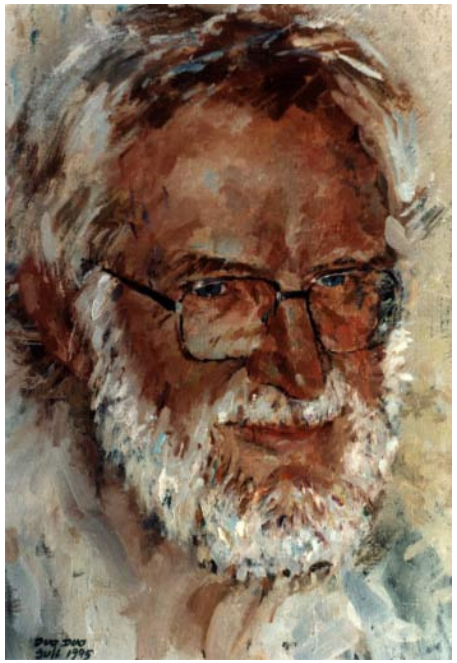
- John Backus presented a new notation containing most of the elements of BNF at a UNESCO conference
- His presentation was about Algol 58
- Peter Naur read this report and found that he and Backus interpreted Algol differently
- He wanted even more precision
- So he created what we now know as BNF for Algol 60
- Thus BNF was first published in *Algol 60 Report*

Who was John Backus?



- Backus invented FORTRAN ("FORMula TRANslator"), the first high-level language *ever*, circa 1954
- Major influence on the invention of functional programming in 1970's
- Won the 1977 Turing Award for BNF and FORTRAN

Who was Peter Naur?



- Danish astronomer turned computer scientist
- Born in 1928; picture on left is from 1968

A Bit More History…

- BNF originally stood for “Backus Normal Form”
- In 1964, Donald Knuth wrote a letter published in Communications of the ACM in which he suggests it stand for Backus-Naur form instead
- This was for two reasons:
- To recognize Naur’s contribution
- BNF is not technically a “normal form”; this would imply that there would be only one correct way of writing a grammar

What does BNF look like?

- Like this:

```
<number> ::= <digit> | <number> <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- "::=" means "is defined as" (some variants use ":=" instead)
- "|" means "or"
- Angle brackets mean a nonterminal
- Symbols without angle brackets are terminals

More BNF Examples

- <while loop> ::= while (<condition>) <statement>
- <assignment statement> ::= <variable> = <expression>
- <statement list> ::= <statement> | <statement list> <statement>
- <unsigned integer> ::= <digit> | <unsigned integer><digit>

BNF for Expressions

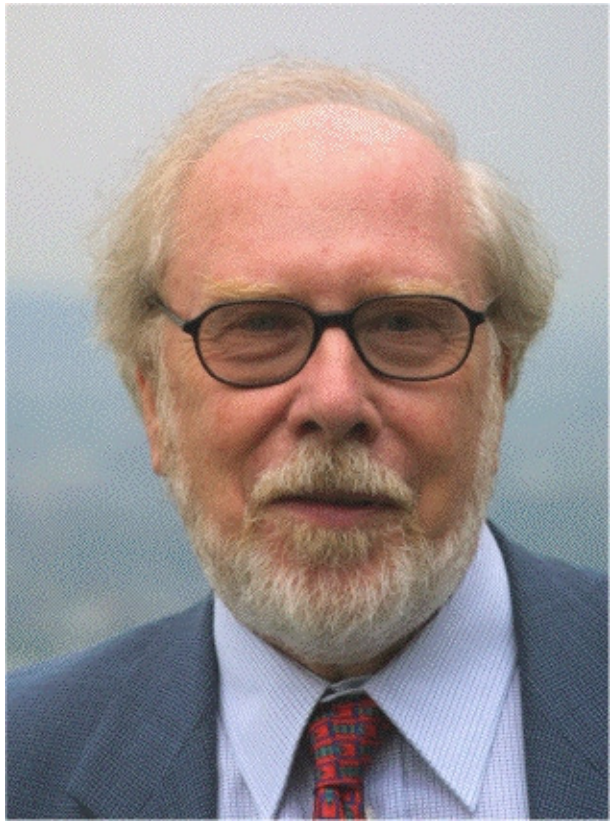
```
<expression> ::= <expression> + <term>
               | <expression> - <term>
               | <term>
<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>
<factor> ::= <primary> ^ <factor>
           | <primary>
<primary> ::= <primary>
            | <element>
<element> ::= ( <expression> )
            | <variable>
            | <number>
```

Origin of EBNF

- Stands for “Extended Backus-Naur Form”
- After BNF appeared with Algol 60, lots of people added their own extensions
- Niklaus Wirth wanted to see one form, so he published “What Can We Do About the Unnecessary Diversity of Notation for Syntactic Definitions” in Communications of the ACM in 1977
- He suggested the use of “[..]” for optional symbols (0 or 1 occurrences), “{ .. }” for 0 or more occurrences.
- Did not mention “EBNF” or Kleene cross

Who was Niklaus Wirth?

- Born in Switzerland, 1934
- Desgined Pascal (1970), Modula, Modula-2 (1980), and Oberon (1988)
- Won the Turing Award in 1984
- Won the IEEE Computer Pioneer Award in 1987



What is EBNF?

- EBNF is a few simple extensions to BNF which make expressing grammars more convenient; adds “syntactic sugar”
- Thus it is more concise than BNF
- EBNF is no more “powerful” than BNF; that is, anything that can be expressed in EBNF can also be expressed in BNF
- EBNF is widely used as the de facto standard to define programming languages

What are the Extensions?

- They vary, but often are derived from regular expression syntax
- "*****" (The Kleene Star): means 0 or more occurrences
- "**+**" (The Kleene Cross): means 1 or more occurrences
- "**?**": means 0 or 1 occurrences (sometimes "[…]" used instead)
- Use of parentheses for grouping

BNF vs EBNF

- Grammar for decimal numbers in plain BNF:

```
<expr> ::= '-' <num> | <num>
<num> ::= <digits>
        | <digits> '.' <digits>
<digits> ::= <digit>
           | <digit> <digits>
<digit> ::= '0' | '1' | '2' | '3' |
  '4' | '5' | '6' | '7' | '8' | '9'
```

BNF vs EBNF

- Same grammar in EBNF:

```
<expr> := '-'? <digit>+ ('.' <digit>+)?
<digit> := '0' | '1' | '2' | '3' | '4' |
  '5' | '6' | '7' | '8' | '9'
```

- So much more concise!
- An optional ‘-’, one or more digits, an optional decimal point followed by one or more digits

Simple Conversions

- If you have a rule such as:

```
<expr> ::= <digits>
<digits> ::= <digit>
           | <digit> <digits>
```

- You can replace it with:

```
<expr> ::= <digit>+
```

More Simple Conversions

- If you have a rule such as:

```
<expr> ::= <digits> | empty
```

- You can replace it with:

```
<expr> ::= <digit>*
```

More Simple Conversions

- If you have a rule such as:

```
<id> ::= <letter>
       | <id><letter>
       | <id><digit>
```

- You can replace it with:

```
<id> ::= <letter> (<letter> | <digit>)*
```

EBNF for Lisp

```
s_expression ::= atomic_symbol
       | "(" s_expression "." s_expression ")"
       | list
list ::= "(" s_expression* ")"
atomic_symbol ::= letter atom_part
atom_part ::= empty
            | letter atom_part
            | number atom_part
letter ::= "a" | "b" | " ..." | "z"
number ::= "1" | "2" | " ..." | "9"
```

Summary-BNF

- BNF uses following notations:
  (i) Non-terminals enclosed in < and >.
  (ii) Rules written as
  $X ::= Y$
  (1) X is LHS of rule and can only be a NT.
  (2) Y is RHS of rule: Y can be
  (a) a terminal, nonterminal, or concatenation of terminal and nonterminals, or
  (b) a set of strings separated by alternation symbol /.

Example:
$< S >::= a < S > / a$
• Notation ε: : Used to represent an empty string (a string of length 0).

Extended BNF (EBNF)

- EBNF: adding more meta-notation ⇒ shorter productions
- NTS begin with uppercase letters (discard <>)
- Repetitions (zero or more) are enclosed in {}

- Zero or one (options) are enclosed in []:

Ifstmt ::= if Cond then Stmt |
if Cond then Stmt else Stmt

⇒ Ifstmt ::= if Cond then Stmt [else Stmt]

- Use () to group items together:

Exp ::= Item {+ Item} | Item {- Item}

⇒ Exp ::= Item {(+|-) Item}

- Terminals that are grammar symbols ('[' for instance) are enclosed in quotes (‘’).

Summary

Conversion from EBNF to BNF and Vice Versa

• BNF to EBNF:

(i) Look for recursion in grammar:

A ::= a A | B

⇒ A ::= a { a } B

(ii) Look for common string that can be factored out with grouping and options.

A ::= a B | a

⇒ A := a [B]

• EBNF to BNF:

(i) Options: []

A ::= a [B] C

⇒ A' ::= a N C N ::= B | ε

(ii) Repetition: {}

A ::= a { B1 B2 ... Bn } C

⇒ A' ::= a N C N ::= B1 B2 ... Bn N | ε