EBNF: Extended BNF

EBNF notation supplements BNF notation with (...), [...], and { ... } to allow simpler specifications.

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[γ] = (γ | <empty>) means "γ is optional".
{ γ } = (<empty> | (γ) | (γ)(γ) | (γ)(γ)(γ) | ...)
means "*zero or more* γs".

Examples

Expr ::= Term (+ | –) Term
is equivalent to the following 2 BNF productions:
Expr ::= Term + Term
| Term – Term

Expr ::= [+ | –] Term (+ | –) Term
is equivalent to
Expr ::= (+ | – | <empty>) Term (+ | –) Term
which is equivalent to these 6 BNF productions:
Expr ::= + Term + Term | – Term + Term | Term + Term
| + Term – Term | – Term – Term | Term – Term

(γ_1 | ... | γ_k) means "pick any one of γ_1, ..., γ_k".
[γ] = (γ | <empty>) means "γ is optional".
{ γ } = (<empty> | (γ) | (γ)(γ) | (γ)(γ)(γ) | ...) means "*0 or more* γs".

Examples

Expr ::= Term (+ | –) Term

is equivalent to the following 2 BNF productions:

Expr ::= Term + Term
| Term – Term

Expr ::= [+ | –] Term (+ | –) Term

is equivalent to

Expr ::= (+ | – | <empty>) Term (+ | –) Term

which is equivalent to these 6 BNF productions:

Expr ::= + Term + Term | – Term + Term | Term + Term
| + Term – Term | – Term – Term | Term – Term

Expr ::= Term {(+ | –) Term}

is equivalent to an ***infinite*** collection of BNF productions, including productions such as

Expr ::= Term + Term + Term – Term + Term – Term – Term

Parse Trees Based on EBNF Specifications

As with nonterminals of a grammar, a nonterminal ***N*** of an EBNF specification denotes **the set of all sequences of terminals t_1 ... t_k for which there is a parse tree with root *N* that generates t_1 ... t_k**.

Here we define parse tree with root *N* that generates t_1 ... t_k in the same way as we defined such trees for a grammar, but:

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a BNF production* (whose right side consists only of terminals and/or nonterminals) *that can be obtained from one of the EBNF rules of the EBNF specification*.

  For example, the productions

  ```
  Expr ::= Term
  Expr ::= – Term + Term + Term – Term
  ```

  are two examples of productions that can be obtained from this EBNF rule: `Expr ::= [+ | –] Term {(+ | –) Term}`

Parse Trees Based on EBNF Specifications

We define parse tree with root *N* that generates t_1 ... t_k in the same way as we defined such trees for a grammar, but:

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a BNF production* (whose right side consists only of terminals and/or nonterminals) *that can be obtained from one of the EBNF rules of the EBNF specification*.

  For example, the productions

  Expr ::= Term

  Expr ::= – Term + Term + Term – Term

  are two examples of productions that can be obtained from this EBNF rule: Expr ::= [+ | –] Term {(+ | –) Term}

Example: Based on TinyJ's EBNF specification on page 1 of the TinyJ Assignment 1 document, a parse tree (whose root is <expr4>) that proves

UNSIGNEDINT + IDENTIFIER * UNSIGNEDINT ∈ <expr4>

is shown on page 2 of the TinyJ Assignment 1 document.

Parse Trees Based on EBNF Specifications

We define parse tree with root *N* that generates t_1 ... t_k in the same way as we defined such trees for a grammar, but:

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a BNF production* (whose right side consists only of terminals and/or nonterminals) *that can be obtained from one of the EBNF rules of the EBNF specification*.

  For example, the productions

  Expr ::= Term

  Expr ::= – Term + Term + Term – Term

  are two examples of productions that can be obtained from this EBNF rule: Expr ::= [+ | –] Term {(+ | –) Term}

As in the case of BNF, an EBNF specification has a ***starting nonterminal***. Unless otherwise indicated, ***this is the nonterminal on the left of the 1st EBNF rule***, and ***parse tree*** means ***parse tree whose root is the starting nonterminal***.

As in the case of BNF, the set (of sequences of terminals) denoted by the starting nonterminal of an EBNF specification is called the ***language generated by*** that EBNF specification.

Use ' ' to Distinguish Terminals from EBNF Metasymbols

In EBNF, when any of the characters | ( ) [ ] { } is a terminal, that terminal should be ***put in single quotes*** to make it clear that the character is ***not*** being used with its EBNF meaning!

Sethi says the following about this on p. 47 of his book (p. 48 of the course reader):

Symbols such as { and }, which have a special status in a language description, are called *metasymbols*.

EBNF has many more metasymbols than BNF. Furthermore, these same symbols can also appear in the syntax of a language—the index i in $A[i]$ is not optional—so care is needed to distinguish tokens from metasymbols. Confusion between tokens and metasymbols will be avoided by enclosing tokens within single quotes if needed, as in '('.

An EBNF version of the grammar in Fig. 2.6 is

⟨*expression*⟩ ::= ⟨*term*⟩ { (+|−) ⟨*term*⟩ }
⟨*term*⟩ ::= ⟨*factor*⟩ { (∗|/) ⟨*factor*⟩ }
⟨*factor*⟩ ::= '(' ⟨*expression*⟩ ')' | **name** | **number**

Use of BNF or EBNF to Define *Syntactically* Valid Code

If a piece of source code should be decomposed by a compiler into a sequence of token instances t_1 ... t_n in which each t_i is an instance of token T_i, then we say T_1 ... T_n is the ***sequence of tokens*** of that source code.

Java Example: **IDENTIFIER = UNSIGNED-INT-LITERAL ;**
is the sequence of tokens of **x23 = 4;**

To define syntactic validity, the designer of a language L can write a BNF or EBNF specification *G* with these properties:

1. The terminals are the tokens of L.
2. The starting nonterminal corresponds to "L source file".
3. Some other nonterminals correspond to language constructs—e.g., "L if statement".
4. Whenever *X* is "L source file" or one of the language constructs mentioned in item 3, and N_X is the nonterminal corresponding to *X*, we have that:
   - If T_1 ... T_n is the sequence of tokens of a *possibly legal X*, then T_1 ... $T_n \in N_X$.
   - If T_1 ... $T_n \in N_X$, then T_1 ... T_n is the sequence of tokens of a piece of source code that "looks like" a *possibly legal X*.

Use of BNF or EBNF to Define *Syntactically* Valid Code

To define syntactic validity, the designer of a language L can write a BNF or EBNF specification *G* with these properties:

1. The terminals are the tokens of L.
2. The starting nonterminal corresponds to "L source file".
3. Some other nonterminals correspond to language constructs—e.g., "L if statement".
4. Whenever *X* is "L source file" or one of the language constructs mentioned in item 3, and N_X is the nonterminal corresponding to *X*, we have that:
   - If $T_1 \ldots T_n$ is the sequence of tokens of a *possibly legal X*, then $T_1 \ldots T_n \in N_X$.
   - If $T_1 \ldots T_n \in N_X$, then $T_1 \ldots T_n$ is the sequence of tokens of a piece of source code that "looks like" a *possibly legal X*.

Here possibly legal *X* means text that is a legal *X* or would be a legal *X* in some context (e.g., with appropriate declarations).

"looks like" isn't precisely defined; we may be able to ensure a higher degree of resemblance by using less simple BNF / EBNF.

IMPORTANT: An example of an EBNF specification with these properties (for the TinyJ language) is given on p. 1 of the TinyJ Assignment 1 document.

Use of BNF or EBNF to Define *Syntactically* Valid Code

To define syntactic validity, the designer of a language L can write a BNF or EBNF specification *G* with these properties:

1. The terminals are the tokens of L.
2. The starting nonterminal corresponds to "L source file".
3. Some other nonterminals correspond to language constructs—e.g., "L if statement".
4. Whenever *X* is "L source file" or one of the language constructs mentioned in item 3, and N_X is the nonterminal corresponding to *X*, we have that:
   - If $T_1 \ldots T_n$ is the sequence of tokens of a *possibly legal X*, then $T_1 \ldots T_n \in N_X$.

EXAMPLE Such a *G* for Java in which there is a nonterminal <stmt> that corresponds to "Java statement" must satisfy
IDENTIFIER [IDENTIFIER] = IDENTIFIER / UNSIGNED-INT-LITERAL ; ∈ <stmt>
because this is the sequence of tokens of **a[b] = c/2;** (which is a *possibly legal X* when *X* is "Java statement" and N_X is <stmt>).

After such a *G* is written, any sequence of tokens in N_X is said to be ***syntactically valid for X***, and any source code whose sequence of tokens is in N_X is called a ***syntactically valid X***.

To define syntactic validity, the designer of a language L can write a BNF or EBNF specification *G* with these properties:

1. The terminals are the tokens of L.
2. The starting nonterminal corresponds to "L source file".
3. Some other nonterminals correspond to language constructs—e.g., "L if statement".
4. Whenever *X* is "L source file" or one of the language constructs mentioned in item 3, and N_X is the nonterminal corresponding to *X*, we have that:
   - If $T_1 \ldots T_n$ is the sequence of tokens of a *possibly legal X*, then $T_1 \ldots T_n \in N_X$.

EXAMPLE Such a *G* for Java in which there is a nonterminal <stmt> that corresponds to "Java statement" must satisfy

`IDENTIFIER [ IDENTIFIER ] = IDENTIFIER / UNSIGNED-INT-LITERAL ;` ∈ <stmt>

because this is the sequence of tokens of: **`a[b] = c/2;`**

After such a *G* is written, any sequence of tokens in N_X is said to be ***syntactically valid for X***, and any source code whose sequence of tokens is in N_X is called a ***syntactically valid X***.

When *X* = "Java statement" (so N_X = <stmt>), the sequence of tokens `IDENTIFIER [ IDENTIFIER ] = IDENTIFIER / UNSIGNED-INT-LITERAL ;` is in N_X and is therefore ***syntactically valid for X = "Java statement"***, so **`x[x] = y/0;`** is a ***syntactically valid Java statement***.

To define syntactic validity, the designer of a language L can write a BNF or EBNF specification *G* with these properties:

1. The terminals are the tokens of L.
2. The starting nonterminal corresponds to "L source file".
3. Some other nonterminals correspond to language constructs—e.g., "L if statement".
4. Whenever *X* is "L source file" or one of the language constructs mentioned in item 3, and N_X is the nonterminal corresponding to *X*, we have that:
   - If $T_1 \ldots T_n$ is the sequence of tokens of a *possibly legal X*, then $T_1 \ldots T_n \in N_X$.
   - If $T_1 \ldots T_n \in N_X$, then $T_1 \ldots T_n$ is the sequence of tokens of a piece of source code that "looks like" a *possibly legal X*.

After such a *G* is written, any sequence of tokens in N_X is said to be ***syntactically valid for X***, and any source code whose sequence of tokens is in N_X is called a ***syntactically valid X***.

In particular, when *X* = L source file (so N_X is the starting nonterminal and N_X denotes *the language generated by G*), we have that:

- Any source code whose sequence of tokens is in the language generated by *G* is called a ***syntactically valid*** L ***source file***.

Recall: Any source code whose sequence of tokens is in N_X is called a ***syntactically valid X***.

Any piece of source code that is a possibly legal *X* is certainly a syntactically valid *X* (as its sequence of tokens must be in N_X), ***but a piece of source code that is not a possibly legal X may still be a syntactically valid X***!

Indeed, any piece of source code that has the same sequence of tokens as a syntactically valid *X* is itself a syntactically valid *X*, *and so any piece of source code that has the same sequence of tokens as a possibly legal X is a syntactically valid X*. **Example:**

a[b] = c/2; is a possibly legal Java statement whose sequence of tokens is: `IDENTIFIER [ IDENTIFIER ] = IDENTIFIER / UNSIGNED-INT-LITERAL ;`
x[x] = y/0; ***isn't*** a possibly legal Java statement, but *has the same sequence of tokens* and so is a syntactically valid Java statement.

Indeed, any piece of source code that has the same sequence of tokens as a syntactically valid *X* is itself a syntactically valid *X*, *and so any piece of source code that has the same sequence of tokens as a possibly legal X is a syntactically valid X*. **Example**:

> **a[b] = c/2;** is a possibly legal Java statement whose sequence of tokens is: **IDENTIFIER [IDENTIFIER] = IDENTIFIER / UNSIGNED-INT-LITERAL ;**
> **x[x] = y/0;** ***isn't*** a possibly legal Java statement, but *has the same sequence of tokens* and so is a syntactically valid Java statement.

More generally, a syntactically valid *X* will still be a syntactically valid *X* after we make one or more changes of the following two kinds (because these changes don't affect the sequence of tokens):

- Replace an identifier with another identifier.
- Replace a literal with another literal of the same kind.