Lexical Syntax: Tokens

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical Syntax: Tokens

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a language might be:

;   <   --   -   )   {   IDENTIFIER   UNSIGNED-INT-LITERAL   while   if

Lexical Syntax: Tokens

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

Lexical Syntax: Tokens

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:
3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**
2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

If a token has just one instance, then it can be denoted by the instance--e.g., **if** denotes the token whose only instance is **if**.

Notes:

Lexical Syntax: Tokens

An important part of the work of a compiler or interpreter is ***lexical analysis*** or ***lexical scanning***.

Lexical analysis decomposes the source program into **token instances** (i.e., **instances of *tokens***).
Ten examples of tokens of a language might be:
; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:
3 instances of **IDENTIFIER** are: **x prevVal pi_2**
2 instances of **UNSIGNED-INT-LITERAL** are: **23 0x1A1D**

If a token has just one instance, then it can be denoted by the instance--e.g., **if** denotes the token whose only instance is **if**.

Notes: In sec. 2.3 of Sethi, the tokens **IDENTIFIER** and **UNSIGNED-INT-LITERAL** are called **name** and **number**, and a token instance is called a ***spelling***.
Many authors call a token instance a ***lexeme***.

Lexical Syntax: Tokens

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*.

For Java:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java**:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java**:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

For most programming languages, ***there are 5 kinds of token***:

1.

2.

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java:**

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

For most programming languages, ***there are 5 kinds of token***:

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   -

2.

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java**:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

For most programming languages, ***there are 5 kinds of token***:

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.

2.

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java**:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

For most programming languages, ***there are 5 kinds of token***:

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String).

Lexical Syntax: The Five Kinds of Token

Ten examples of tokens of a language might be:

; < -- -) { IDENTIFIER UNSIGNED-INT-LITERAL while if

Each token *T* is a set of strings of characters; each member of that set is called an ***instance*** of *T*. **For Java**:

3 instances of **IDENTIFIER** are: **x** **prevVal** **pi_2**

2 instances of **UNSIGNED-INT-LITERAL** are: **23** **0x1A1D**

For most programming languages, ***there are 5 kinds of token***:

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:

   Instances of the *floating-pt. literal token*: **2.3**, **4.1f**, 3e-4

   Instances of the *String literal token*: **"The cat"**, **"apple"**

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:
   Instances of the *floating-pt. literal token*: **2.3**, **4.1f**, 3e-4
   Instances of the *String literal token*: **"The cat"**, **"apple"**
3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   -

Lexical Syntax: The Five Kinds of Token

1. There is a single token (which we call IDENTIFIER) whose instances are used as ***names*** of entities such as variables, functions/methods, classes, packages, and labels.
   - Each instance of this token is called an *identifier*.
2. There are tokens called ***literal tokens***. Each literal token is associated with a different type / kind of type (e.g., integer, floating-point, boolean, char, String). *Each instance of such a token represents a fixed value of the associated type / kind of type* (and is called an integer literal, String literal, etc.). Java examples:

   Instances of the *floating-pt. literal token*: **2.3**, **4.1f**, 3e-4

   Instances of the *String literal token*: **"The cat"**, **"apple"**
3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but **cannot** be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
   -

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
   - Reserved words are also called ***keywords***.

   Note:

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but **cannot** be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
   - Reserved words are also called ***keywords***.

   Note: In some languages there are "words" that have an entirely different role from that of an identifier, but which are ***not*** reserved words because it is legal to use them as identifiers in some contexts: *Such "words" are also called keywords*. In Lisp, special operator names (e.g., IF, LET, QUOTE) are keywords of this kind: They can be used as identifiers, as in

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
   - Reserved words are also called ***keywords***.

Note: In some languages there are "words" that have an entirely different role from that of an identifier, but which are ***not*** reserved words because it is legal to use them as identifiers in some contexts: *Such "words" are also called keywords*. In Lisp, special operator names (e.g., IF, LET, QUOTE) are keywords of this kind: They can be used as identifiers, as in

(defun f (if let quote) (+ if let quote)),

though it'd be a bad idea to write such code.

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
4. For each ***operator*** (e.g., **!**, *****, **++**, **+=**, **>=**, &&, **:**, **?** in Java) ***there is a token whose only instance is that operator***.

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
4. For each ***operator*** (e.g., **!**, *****, **++**, **+=**, **>=**, **&&**, **:**, **?** in Java) ***there is a token whose only instance is that operator***.
5. Languages usually have certain other characters or sequences of characters that are used as a "punctuation" symbols. Java examples: **,**, **;**, **.**, **{**, **}**, **[**, **]**, **(**, **)**, **::**

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
4. For each ***operator*** (e.g., **!**, *****, **++**, **+=**, **>=**, **&&**, **:**, **?** in Java) ***there is a token whose only instance is that operator***.
5. Languages usually have certain other characters or sequences of characters that are used as a "punctuation" symbols. Java examples: **,**, **;**, **.**, **{**, **}**, **[**, **]**, **(**, **)**, **::**

   These are called ***delimiters*** or ***separators***. For each of them ***there's a token whose only instance is that symbol***.

Lexical Syntax: The Five Kinds of Token

3. A ***reserved word*** looks like an identifier *but* ***cannot*** *be used as an identifier and instead plays an entirely different role*. Java examples are: **for**, **if**, **case**, **return**
   - For each reserved word ***there is a token whose only instance is that reserved word*** (unless reserved words are case-insensitive, in which case all ways of writing a given reserved word are instances of the same token).
4. For each ***operator*** (e.g., **!**, *****, **++**, **+=**, **>=**, **&&**, **:**, **?** in Java) ***there is a token whose only instance is that operator***.
5. Languages usually have certain other characters or sequences of characters that are used as a "punctuation" symbols. Java examples: **,**, **;**, **.**, **{**, **}**, **[**, **]**, **(**, **)**, **::**

   These are called ***delimiters*** or ***separators***. For each of them ***there's a token whose only instance is that symbol***.

A ***lexical syntax specification*** of a programming language specifies ***its tokens*** and ***the sequence of token instances into which any given piece of source code should be decomposed***.

Use of Grammars to Define *Syntactically* Valid Code

Use of Grammars 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, we say T_1 ... T_n is the ***sequence of tokens*** of that source code.

Java Example:

Use of Grammars 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, 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;**

Use of Grammars 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, 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;**

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

Use of Grammars 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, 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;**

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

-
-

Use of Grammars to Define *Syntactically* Valid Code

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

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

For many programming languages L, the language designer can construct a grammar G (whose terminals are L's tokens) such that:

$T_1 \ldots T_n$ belongs to the language generated by G if (and, roughly speaking, only if) $T_1 \ldots T_n$ is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
-

Use of Grammars 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, 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;**

For many programming languages L, the language designer can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar *G*.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar *G*.

- More generally, when a nonterminal of *G* corresponds to a language construct *X* (e.g., *statement*), we say a piece of source code is a ***syntactically valid*** *X* if its sequence of tokens belongs to the set denoted by the nonterminal.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar *G*.

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar *G* (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by *G* if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar *G*.

- Replacing one identifier with another and replacing a literal constant with another of the same type (e.g., changing 9/x to 3/y) will ***not*** affect the ***syntactic*** validity of a piece of source code, as it won't change its sequence of tokens!

Use of Grammars to Define *Syntactically* Valid Code

For many programming languages L, the language designer can construct a grammar G (whose terminals are L's tokens) such that:

T_1 ... T_n belongs to the language generated by G if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly legal L source file.

- We say a file is "possibly legal" if it's either legal or legal under certain conditions (e.g., if certain variables and functions are appropriately defined in other files).
- "roughly speaking" means some exceptions to the condition's "only if" part are allowed.

We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the grammar G.

- Replacing one identifier with another and replacing a literal constant with another of the same type (e.g., changing 9/x to 3/y) will ***not*** affect the ***syntactic*** validity of a piece of source code, as it won't change its sequence of tokens!

See https://euclid.cs.qc.cuny.edu/316/Syntactic-Validity.pdf for more on syntactic validity.

EBNF: Extended BNF

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

$(\gamma_1 \mid ... \mid \gamma_k)$ means
$[\ \gamma\ ]$ =
$\{\ \gamma\ \}$ =

EBNF: Extended BNF

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

$(\gamma_1 \mid \ldots \mid \gamma_k)$ means "pick any one of γ_1, ..., γ_k".
$[\ \gamma\ ]$ =
$\{\ \gamma\ \}$ =

EBNF: Extended BNF

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

$(\gamma_1 \mid ... \mid \gamma_k)$ means "pick any one of γ_1, ..., γ_k".
[γ] = (γ | <empty>)   means "γ is optional".
{ γ } =

EBNF: Extended BNF

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

$(\gamma_1 \mid \ldots \mid \gamma_k)$ means "pick any one of $\gamma_1, \ldots, \gamma_k$".

$[\ \gamma\ ] = (\gamma \mid$ <empty>$)$ means "γ is optional".

$\{\ \gamma\ \} = ($<empty> $\mid (\gamma) \mid (\gamma)(\gamma) \mid (\gamma)(\gamma)(\gamma) \mid \ldots)$

means "*zero or more* γs".

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:

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
```

EBNF: Extended BNF

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

$(\gamma_1 \mid \ldots \mid \gamma_k)$ means "pick any one of $\gamma_1, \ldots, \gamma_k$".
$[\ \gamma\ ] = (\gamma \mid$ <empty>$)$ means "γ is optional".
$\{\ \gamma\ \} = ($<empty> $\mid (\gamma) \mid (\gamma)(\gamma) \mid (\gamma)(\gamma)(\gamma) \mid \ldots)$
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:

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 “*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 "*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
```

(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

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

-

-

-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each (x_1 | ... | x_k) with a new nonterminal (*D*, say) that is defined by these *k* productions:

  D ::= x_1 | ... | x_k

-
-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each (x_1 | ... | x_k) with a new nonterminal (*D*, say) that is defined by these *k* productions:

  D ::= x_1 | ... | x_k

- Replace each [x_1 | ... | x_k] with a new nonterminal (*D*, say) that is defined by these *k*+1 productions:

  D ::= *<empty>* | x_1 | ... | x_k

-

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each $(x_1 \mid \ldots \mid x_k)$ with a new nonterminal (D, say) that is defined by these k productions:

  $$D ::= x_1 \mid \ldots \mid x_k$$

- Replace each $[x_1 \mid \ldots \mid x_k]$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  $$D ::= \langle empty \rangle \mid x_1 \mid \ldots \mid x_k$$

- Replace each $\{x_1 \mid \ldots \mid x_k\}$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  $$D ::= \langle empty \rangle \mid Dx_1 \mid \ldots \mid Dx_k$$

A grammar can only have ***finitely*** many productions. However, any EBNF rule can be translated into an equivalent ***finite*** set of BNF productions as follows.

Working from the inside outwards, eliminate all occurrences of (...), [...], and { ... }:

- Replace each $(x_1 \mid \ldots \mid x_k)$ with a new nonterminal (D, say) that is defined by these k productions:

  D ::= $x_1 \mid \ldots \mid x_k$

- Replace each $[x_1 \mid \ldots \mid x_k]$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $x_1 \mid \ldots \mid x_k$

- Replace each $\{x_1 \mid \ldots \mid x_k\}$ with a new nonterminal (D, say) that is defined by these k+1 productions:

  D ::= *<empty>* | $Dx_1 \mid \ldots \mid Dx_k$

Here k may be 1. Thus { *Digit* } can be replaced with a new nonterminal (*DigitSeq*, say) that is defined by:

DigitSeq ::= *<empty>* | *DigitSeq Digit*

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}          (*)
```

into a finite set of BNF productions.

Example: We now use the above method to translate

```
      Expr ::= [+ | -] Term {(+ | -) Term}          (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op defined by: Op ::= + | -

   (*) becomes:

Example: We now use the above method to translate

    Expr ::= [+ | -] Term {(+ | -) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)

Example: We now use the above method to translate

    Expr ::= [+ | -] Term {(+ | -) Term}   (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term}   (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes:

Example: We now use the above method to translate

Expr ::= [+ | -] Term {(+ | -) Term} (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op
   defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term} (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest (***)

Example: We now use the above method to translate

Expr ::= [+ | -] Term {(+ | -) Term} (*)

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op defined by: Op ::= + | -
   (*) becomes: Expr ::= [+ | -] Term {Op Term} (**)
2. Next, replace {Op Term} with a nonterminal Rest defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | -] Term Rest (***)
3. Finally, replace [+ | -] with a nonterminal OptSign defined by OptSign ::= <*empty*> | + | -
   (***) becomes:

Example: We now use the above method to translate

Expr ::= [+ | –] Term {(+ | –) Term} (*)

into a finite set of BNF productions.

1. First, replace (+ | –) with a nonterminal Op
   defined by: Op ::= + | –
   (*) becomes: Expr ::= [+ | –] Term {Op Term} (**)
2. Next, replace {Op Term} with a nonterminal Rest
   defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: Expr ::= [+ | –] Term Rest (***)
3. Finally, replace [+ | –] with a nonterminal OptSign
   defined by OptSign ::= <*empty*> | + | –
   (***) becomes: Expr ::= OptSign Term Rest

Example: We now use the above method to translate

```
    Expr ::= [+ | -] Term {(+ | -) Term}        (*)
```

into a finite set of BNF productions.

1. First, replace (+ | -) with a nonterminal Op defined by: `Op ::= + | -`
   (*) becomes: `Expr ::= [+ | -] Term {Op Term}   (**)`
2. Next, replace {Op Term} with a nonterminal Rest defined by: Rest ::= <*empty*> | Rest Op Term
   (**) becomes: `Expr ::= [+ | -] Term Rest      (***)`
3. Finally, replace [+ | -] with a nonterminal OptSign defined by OptSign ::= <*empty*> | + | -
   (***) becomes: `Expr ::= OptSign Term Rest`

The result is the following set of 8 BNF productions:

```
   Expr ::= OptSign Term Rest
OptSign ::= <empty> | + | -
   Rest ::= <empty> | Rest Op Term
     Op ::= + | -
```

While the above method always works, it will often ***not*** find a simplest finite set of grammar productions that is equivalent to the given EBNF rule!

While the above method always works, it will often ***not*** find a simplest finite set of grammar productions that is equivalent to the given EBNF rule!

For example, here is a simpler set of grammar productions that is equivalent to the EBNF rule

```
      Expr ::= [+ | -] Term {(+ | -) Term}
```

considered above:

```
  Expr ::=   Term
          |  + Term
          |  - Term
          |  Expr + Term
          |  Expr - Term
```

A Rule to Follow When Writing EBNF Specifications

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

$$
\begin{array}{rcl}
\langle expression \rangle & ::= & \langle term \rangle \ \{ \ (+ \mid -) \ \langle term \rangle \ \} \\
\langle term \rangle & ::= & \langle factor \rangle \ \{ \ (* \mid /) \ \langle factor \rangle \ \} \\
\langle factor \rangle & ::= & \text{'('} \ \langle expression \rangle \ \text{')'} \mid \textbf{name} \mid \textbf{number}
\end{array}
$$

Parse Trees Based on EBNF Specifications

Given an EBNF specification, we define its parse trees in the same way as parse trees based on a grammar; parse trees can be used to define the set of sequences of terminals that is denoted by each nonterminal, as in the case of a grammar.

-

-

Parse Trees Based on EBNF Specifications

Given an EBNF specification, we define its parse trees in the same way as parse trees based on a grammar; parse trees can be used to define the set of sequences of terminals that is denoted by each nonterminal, as in the case of a grammar.

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a 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.*
-

Parse Trees Based on EBNF Specifications

Given an EBNF specification, we define its parse trees in the same way as parse trees based on a grammar; parse trees can be used to define the set of sequences of terminals that is denoted by each nonterminal, as in the case of a grammar.

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a 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

Given an EBNF specification, we define its parse trees in the same way as parse trees based on a grammar; parse trees can be used to define the set of sequences of terminals that is denoted by each nonterminal, as in the case of a grammar.

- In rule 4 of the definition of a parse tree, we interpret "a production" to mean *a 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}`

Unless otherwise indicated, an EBNF spec's ***starting nonterminal*** is the ***nonterminal on the left of the 1st production***, 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.

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.

- EBNF can be used just like BNF to define what it means for source code to be "***<u>syntactically</u>*** valid":

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.

- EBNF can be used just like BNF to define what it means for source code to be "***syntactically*** valid":

  For many programming languages L, the language designer can construct ~~a grammar~~ an EBNF specification *G* (whose terminals are L's tokens) such that:

  > T_1 ... T_n belongs to the language generated by *G*
  > if (and, roughly speaking, only if) T_1 ... T_n is the sequence of tokens of a possibly valid L source file.

  We can then say a particular L source file is ***syntactically valid*** if its sequence of tokens belongs to the language generated by the ~~grammar~~ EBNF specification.

  - More generally, when a nonterminal of *G* corresponds to a language construct *X* (e.g., *statement*), we say a piece of source code is a ***syntactically valid*** *X* if its sequence of tokens belongs to the set denoted by the nonterminal.

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

The EBNF specification of TinyJ is an example!

This explains why “token” and “terminal” are sometimes used to mean the same thing, as mentioned in sec. 2.3 of Sethi.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*: The commonest examples of such tokens are IDENTIFIER and tokens whose instances are literal constants of some type.

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*:

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*.

Are Tokens Terminals?

Not always, though it is quite common to use BNF or EBNF specifications of programming language syntax in which the terminals are tokens of the language.

But in many other BNF and EBNF specifications of programming language syntax the terminals are *characters* and certain ***nonterminals*** are tokens that have multiple instances:

In addition to specifying syntactically valid sequences of tokens for language constructs, these BNF or EBNF specifications *also specify what sequences of characters are instances of tokens with multiple instances*.

In fact we've already seen an example of the use of BNF to specify such a token:

$$
\begin{aligned}
\langle \textit{real-number} \rangle &::= \langle \textit{integer-part} \rangle \ . \ \langle \textit{fraction} \rangle \\
\langle \textit{integer-part} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{integer-part} \rangle \langle \textit{digit} \rangle \\
\langle \textit{fraction} \rangle &::= \langle \textit{digit} \rangle \mid \langle \textit{digit} \rangle \langle \textit{fraction} \rangle \\
\langle \textit{digit} \rangle &::= 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Figure 2.3 BNF rules for real numbers.

A grammar given by by Sethi to specify unsigned floating point literals in a simple language.