

[HackingOff](#)

- [Home](#)
- [Blog](#)
- [Compiler Construction Toolkit](#)
 - [Overview](#)
 - [Scanner Generator](#)
 - [Regex to NFA & DFA](#)
 - [NFA to DFA](#)
 - [BNF to First, Follow, & Predict sets](#)
 - [Parser Generator Overview](#)
 - [LL\(1\) Parser Generator](#)
 - [LR\(0\) Parser Generator](#)
 - [SLR\(1\) Parser Generator](#)

Generate Predict, First, and Follow Sets from EBNF (Extended Backus Naur Form) Grammar

Provide a grammar in Extended Backus-Naur form (EBNF) to automatically calculate its first, follow, and predict sets. See the sidebar for an example.

First sets are used in LL parsers (top-down parsers reading Left-to-right, using Leftmost-derivations).

Follow sets are used in top-down parsers, but also in LR parsers (bottom-up parsers, reading Left-to-right, using Rightmost derivations). These include LR(0), SLR(1), LR(k), and LALR parsers.

Predict sets, derived from the above two, are used by [Fischer & LeBlanc](#) to construct LL(1) top-down parsers.

Input Your Grammar

For more details, and a well-formed example, check out the sidebar. →

```
statement ->
compound-statement |
if-statement | while-
statement | break-
statement | continue-
statement | return-
statement |
expression-statement
| declaration-
statement
if-statement -> if
expression compound-
statement else
compound-statement
while-statement ->
while expression
compound-statement
break-statement ->
break ;
compound-statement ->
```

[Click for Predict, First, and Follow Sets](#)

First Set

Non-Terminal	Symbol	First Set
if	if	
else	else	
while	while	
break	break	

;	;
{	{
}	}
ϵ	ϵ
continue	continue
return	return
=	=
&&	&&
==	==
!=	!=
<	<
<=	<=
>	>
>=	>=
+	+
-	-
*	*
/	/
!	!
identifier	identifier
((
))
INT-LITERAL	INT-LITERAL
BOOL-LITERAL	BOOL-LITERAL
,	,
var	var
class	class
const	const
:	:
int	int
bool	bool
if-statement	if
while-statement	while
break-statement	break
compound-statement	{
statement-list	ϵ , {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class
continue-statement	continue
return-statement	return
expression-statement	;, ϵ , -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
expression-list	ϵ , -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
class-body	{
variable-declaration-list	ϵ , var
condition-or-expression-list	ϵ , -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
condition-or-expression-tail	ϵ , -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
condition-and-expression-tail	&&, ϵ
equality-expression-tail	ϵ , ==, !=
rel-expression-tail	ϵ , <, <=, >, >=
additive-expression-tail	ϵ , +, -
m-d-expression-tail	ϵ , *, /
u-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
primary-expression	identifier, (, INT-LITERAL, BOOL-LITERAL

para-list	(
proper-para-list-tail	,, ε
arg-list	(
proper-arg-list-tail	,, ε
function-declaration	identifier
variable-declaration	var
class-declaration	class
constant-declaration	const
init-expression	=
type-annotation	:
type	int, bool
top-level	ε, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class
statement	{, while, continue, if, return, break, ;, ε, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class
para-declaration	int, bool
declaration-statement	identifier, var, const, class
m-d-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
proper-para-list	int, bool
additive-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
rel-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
equality-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
condition-and-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
condition-or-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
assignment-expression	ε, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
expression	ε, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
arg	ε, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL
proper-arg-list	ε, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL

Follow Set

Non-Terminal	Symbol	Follow Set
statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
if-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
while-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
break-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
compound-statement		else, \$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
statement-list		}
continue-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
return-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
expression-statement		\$, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class, }
expression-list		;
class-body		
variable-declaration-list		}
expression), -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;, {, ,
assignment-expression), -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;, {, ,
condition-or-expression-list		-, !, identifier, (, INT-LITERAL, BOOL-LITERAL

```

condition-or-expression    =, ), -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;, {, ,
condition-or-expression-tail    =, ), -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;, {, ,
condition-and-expression      -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
condition-and-expression-tail  -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
equality-expression          ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
equality-expression-tail      ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
rel-expression               ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
rel-expression-tail          ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {, ,
additive-expression          <, <=, >, >=, ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {,
additive-expression-tail      <, <=, >, >=, ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;, {,
m-d-expression               +, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;,
m-d-expression-tail          +, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =, ), ;,
u-expression                  *, /, +, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,
primary-expression           *, /, +, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,
para-list                     {
proper-para-list              )
proper-para-list-tail        )
para-declaration              ,, )
arg-list                      *, /, +, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,
proper-arg-list               )
proper-arg-list-tail         )
arg                           ,, )
declaration-statement         $, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-
function-declaration          $, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-
variable-declaration          var, $, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL,
class-declaration             $, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-
constant-declaration          $, {, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-
init-expression               ;
type-annotation               ;
type                          identifier, ;
top-level

```

Predict Set

#	Expression	Predict
1	statement → compound-statement	{
2	statement → if-statement	if
3	statement → while-statement	while
4	statement → break-statement	break
5	statement → continue-statement	continue
6	statement → return-statement	return
		;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL

7	statement → expression-statement	
8	statement → declaration-statement	identifier, var, const, class
9	if-statement → if expression compound-statement else compound-statement	if
10	while-statement → while expression compound-statement	while
11	break-statement → break ;	break
12	compound-statement → { statement-list }	{
13	statement-list → ε	}
14	statement-list → statement statement-list	{, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const, class
15	continue-statement → continue ;	continue
16	return-statement → return expression ;	return
17	return-statement → return ;	return
18	expression-statement → expression-list ;	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;
19	expression-list → expression expression-list	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
20	expression-list → ε	;
21	class-body → { variable-declaration-list }	{
22	variable-declaration-list → variable-declaration variable-declaration-list	var
23	variable-declaration-list → ε	}
24	expression → assignment-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
25	assignment-expression → condition-or-expression-list condition-or-expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
26	condition-or-expression-list → condition-or-expression = condition-or-expression-list	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
27	condition-or-expression-list → ε	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
28	condition-or-expression → condition-and-expression condition-or-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
29	condition-or-expression-tail → ε	=,), -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, ;, {, ,
30	condition-or-expression-tail → condition-and-expression condition-or-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
31	condition-and-expression → equality-expression condition-and-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
32	condition-and-expression-tail → && equality-expression equality-expression-tail	&&
33	condition-and-expression-tail → ε	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,), ;, {, ,
34	equality-expression → rel-expression equality-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
35	equality-expression-tail → ε	==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,), ;, {, ,
36	equality-expression-tail → == rel-expression equality-expression-tail	==
37	equality-expression-tail → != rel-expression equality-expression-tail	!=
38	rel-expression → additive-expression rel-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
39	rel-expression-tail → ε	==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,), ;, {, ,
40	rel-expression-tail → < additive-expression rel-expression-tail	<
41	rel-expression-tail → <= additive-expression rel-expression-tail	<=
42	rel-expression-tail → > additive-expression rel-expression-tail	>
43	rel-expression-tail → >= additive-expression rel-expression-tail	>=
44	additive-expression → m-d-expression additive-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL

45	additive-expression-tail $\rightarrow \epsilon$	<, <=, >, >=, ==, !=, &&, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,), ;, {, ,
46	additive-expression-tail $\rightarrow +$ m-d-expression additive-expression-tail	+
47	additive-expression-tail $\rightarrow -$ m-d-expression additive-expression-tail	-
48	m-d-expression \rightarrow u-expression m-d-expression-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
49	m-d-expression-tail $\rightarrow \epsilon$	+, -, <, <=, >, >=, ==, !=, &&, !, identifier, (, INT-LITERAL, BOOL-LITERAL, =,), ;, {, ,
50	m-d-expression-tail $\rightarrow *$ u-expression m-d-expression-tail	*
51	m-d-expression-tail $\rightarrow /$ u-expression m-d-expression-tail	/
52	u-expression $\rightarrow -$ u-expression	-
53	u-expression $\rightarrow !$ u-expression	!
54	u-expression \rightarrow primary-expression	identifier, (, INT-LITERAL, BOOL-LITERAL
55	primary-expression \rightarrow identifier	identifier
56	primary-expression \rightarrow identifier arg-list	identifier
57	primary-expression $\rightarrow ($ expression $)$	(
58	primary-expression \rightarrow INT-LITERAL	INT-LITERAL
59	primary-expression \rightarrow BOOL-LITERAL	BOOL-LITERAL
60	para-list $\rightarrow ($ $)$	(
61	para-list $\rightarrow ($ proper-para-list $)$	(
62	proper-para-list \rightarrow para-declaration proper-para-list-tail	int, bool
63	proper-para-list-tail $\rightarrow ,$ para-declaration proper-para-list-tail	,
64	proper-para-list-tail $\rightarrow \epsilon$)
65	para-declaration \rightarrow type identifier	int, bool
66	arg-list $\rightarrow ($ $)$	(
67	arg-list $\rightarrow ($ proper-arg-list $)$	(
68	proper-arg-list \rightarrow arg proper-arg-list-tail	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
69	proper-arg-list-tail $\rightarrow ,$ arg proper-arg-list-tail	,
70	proper-arg-list-tail $\rightarrow \epsilon$)
71	arg \rightarrow expression	-, !, identifier, (, INT-LITERAL, BOOL-LITERAL
72	declaration-statement \rightarrow function-declaration	identifier
73	declaration-statement \rightarrow constant-declaration	const
74	declaration-statement \rightarrow variable-declaration	var
75	declaration-statement \rightarrow class-declaration	class
76	function-declaration \rightarrow identifier para-list compound-statement	identifier
77	variable-declaration \rightarrow var identifier init-expression ;	var
78	variable-declaration \rightarrow var identifier type-annotation ;	var
79	class-declaration \rightarrow class identifier init-expression ;	class
80	class-declaration \rightarrow class identifier type-annotation ;	class
81	constant-declaration \rightarrow const identifier init-expression ;	const
82	constant-declaration \rightarrow const identifier type-annotation ;	const
83	init-expression $\rightarrow =$ expression	=
84	type-annotation $\rightarrow :$ type	:
85	type \rightarrow int	int
86	type \rightarrow bool	bool
87	top-level \rightarrow statement top-level	{, while, continue, if, return, break, ;, -, !, identifier, (, INT-LITERAL, BOOL-LITERAL, var, const,

class

$$88 \text{ top-level} \rightarrow \varepsilon$$

LL(1) Parsing Table

On the LL(1) Parsing Table's Meaning and Construction

- The top row corresponds to the columns for all the potential terminal symbols, augmented with \$ to represent the end of the parse.
- The leftmost column and second row are all zero filled, to accomodate the way Fischer and LeBlanc wrote their parser's handling of abs().
- The remaining rows correspond to production rules in the original grammar that you typed in.
- Each entry in that row maps the left-hand-side (LHS) of a production rule onto a line-number. That number is the line in which the LHS had that specific column symbol in its predict set.
- If a terminal is absent from a non-terminal's predict set, an error code is placed in the table. If that terminal is in follow(that non-terminal), the error is a POP error. Else, it's a SCAN error.

POP error code = # of predict table productions + 1

SCAN error code = # of predict table productions + 2

In practice, you'd want to tear the top, label row off of the table and stick it in a comment, so that you can make sense of your table. The remaining table can be used as is.

LL(1) Parsing Table as JSON (for Easy Import)

[illegible]

```
[0, 89, 90, 89, 89, 89, 89, 89, 89, 89, 89, 90, 90, 90, 90, 90, 90, 90, 90, 89, 90, 90, 89, 76, 89, 90, 89, 89, 90, 89, 89, 89, 90, 90, 90, 89],
[0, 89, 90, 89, 89, 89, 89, 89, 89, 89, 89, 90, 90, 90, 90, 90, 90, 90, 90, 89, 90, 90, 89, 89, 89, 90, 89, 89, 90, 78, 89, 89, 90, 90, 90, 89],
[0, 89, 90, 89, 89, 89, 89, 89, 89, 89, 89, 90, 90, 90, 90, 90, 90, 90, 90, 89, 90, 90, 89, 89, 89, 90, 89, 89, 90, 89, 80, 89, 90, 90, 90, 89],
[0, 89, 90, 89, 89, 89, 89, 89, 89, 89, 89, 90, 90, 90, 90, 90, 90, 90, 90, 89, 90, 90, 89, 89, 89, 90, 89, 89, 90, 89, 89, 82, 90, 90, 90, 89],
[0, 90, 90, 90, 90, 89, 90, 90, 90, 90, 83, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90],
[0, 90, 90, 90, 90, 89, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90],
[0, 90, 90, 90, 90, 89, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90, 90],
[0, 87, 90, 87, 87, 87, 87, 90, 87, 87, 90, 90, 90, 90, 90, 90, 90, 90, 90, 87, 90, 90, 87, 87, 87, 90, 87, 87, 87, 90, 87, 87, 87, 90, 90, 90, 90]]
```

LL(1) Parsing Push-Map (as JSON)

This structure maps each production rule in the expanded grammar (seen as the middle column in the predict table above) to a series of states that the LL parser pushes onto the stack.

```
{"1": [5], "2": [2], "3": [3], "4": [4], "5": [7], "6": [8], "7": [9], "8": [38], "9": [5, -2, 5, 13, -1], "10": [5, 13, -3], "11": [-5, -4], "12": [-7, 6, -6], "14": [6, 1], "15": [-5, -8], "16": [-5, 13, -9], "17": [-5, -9], "18": [-5, 10], "19": [10, 13], "21": [-7, 12, -6], "22": [12, 40], "24": [14], "25": [16, 15], "26": [15, -10, 16], "28": [17, 18], "30": [17, 18], "31": [19, 20], "32": [21, 20, -11], "34": [21, 22], "36": [21, 22, -12], "37": [21, 22, -13], "38": [23, 24], "40": [23, 24, -14], "41": [23, 24, -15], "42": [23, 24, -16], "43": [23, 24, -17], "44": [25, 26], "46": [25, 26, -18], "47": [25, 26, -19], "48": [27, 28], "50": [27, 28, -20], "51": [27, 28, -21], "52": [28, -19], "53": [28, -22], "54": [29], "55": [-23], "56": [34, -23], "57": [-25, 13, -24], "58": [-26], "59": [-27], "60": [-25, -24], "61": [-25, 31, -24], "62": [32, 33], "63": [32, 33, -28], "65": [-23, 45], "66": [-25, -24], "67": [-25, 35, -24], "68": [36, 37], "69": [36, 37, -28], "71": [13], "72": [39], "73": [42], "74": [40], "75": [41], "76": [5, 30, -23], "77": [-5, 43, -23, -29], "78": [-5, 44, -23, -29], "79": [-5, 43, -23, -30], "80": [-5, 44, -23, -30], "81": [-5, 43, -23, -31], "82": [-5, 44, -23, -31], "83": [13, -10], "84": [45, -32], "85": [-33], "86": [-34], "87": [46, 1]}
```

How to Calculate First, Follow, & Predict Sets

Specify your grammar in EBNF and slam the button. That's it.

EBNF Grammar Specification Requirements

Productions use the following format:

```
Goal -> A
A -> ( A ) | Two
Two -> a
Two -> b
```

- Symbols are inferred as terminal by absence from the left hand side of production rules.
- " \rightarrow " designates definition, " $|$ " designates alternation, and newlines designate termination.
- $x \rightarrow y \mid z$ is EBNF short-hand for


```

      x -> y
      x -> z
      
```
- Use "EPSILON" to represent ϵ or "LAMBDA" for λ productions. (The two function identically.) E.g., $A \rightarrow b \mid \text{EPSILON}$.
- Be certain to place spaces between things you don't want read as one symbol. $(A) \neq A$

About This Tool

Intended Audience

Computer science students & autodidacts studying compiler design or parsing.

Purpose

Automatic generation of first sets, follow sets, and predict sets speeds up the process of writing parsers. Generating these sets by hands is tedious; this tool helps ameliorate that. Goals:

- Tight feedback loops for faster learning.
- Convenient experimentation with language tweaks. (Write a generic, table/dictionary-driven parser and just plug in the JSON output to get off the ground quickly.)
- Help with tackling existing coursework or creating new course material.

Underlying Theory

I'll do a write-up on this soon. In the interim, you can read about:

- [how to determine first and follow sets \(PDF from Programming Languages course at University of Alaska Fairbanks\)](#)
- [significance of first and follow sets in top-down \(LL\(1\)\) parsing.](#)
- [follow sets' involvement in bottom-up parsing \(LALR, in this case\)](#)

© HackingOff.com 2012