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. →

```
condition-or-
expression-tail ->
EPSILON | XX
condition-and-
expression condition-
or-expression-tail
condition-and-
expression ->
equality-expression
condition-and-
expression-tail
condition-and-
expression-tail -> &&
equality-expression
equality-expression-
tail | EPSILON
equality-expression -
> rel-expression
equality-expression-
tail
```

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
condition-expression	condition-expression
=	=
*=	*=
/=	/=
+=	+=
-=	-=
&&=	&&=

XX=	XX=
XX	XX
&&	&&
==	==
!=	!=
<	<
<=	<=
>	>
>=	>=
+	+
-	-
*	*
/	/
!	!
++	++
--	--
.	.
identifier	identifier
(	(
)	)
INT-LITERAL	INT-LITERAL
BOOL-LITERAL	BOOL-LITERAL
,	,
func	func
->	->
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, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
continue-statement	continue
return-statement	return
expression-statement	;, ε, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
expression-list	ε, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
variable-declaration-list	ε, var
assignment-operator	=, *=, /=, +=, -=, &&=, XX=
condition-or-expression-tail	ε, XX
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
post-expression-tail	., ++, --, ε
primary-expression	identifier, (, INT-LITERAL, BOOL-LITERAL
para-list	(
proper-para-list-tail	,, ε
para-declaration	identifier
arg-list	(
proper-arg-list-tail	,, ε
function-definition	func
variable-declaration	var
class-declaration	class
class-body	{
class-member	ε, const, class, var, func
constant-declaration	const
init-expression	=
type-annotation	:
type	int, bool
top-level	ε, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func
	{, while, continue, if, return, break, const, class, var, ;, ε, -, !, ++, --, identifier, (, INT-LITERAL,

statement	BOOL-LITERAL
post-expression	identifier, (, INT-LITERAL, BOOL-LITERAL
proper-para-list	identifier
declaration-statement	const, class, var
m-d-expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
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	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
arg	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
proper-arg-list	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL

Follow Set

Non-Terminal Symbol	Follow Set
statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
if-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
while-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
break-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
compound-statement	else, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, $, }
statement-list	}
continue-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
return-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
expression-statement	$, {, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
expression-list	;
variable-declaration-list	
expression	), ;, {, ,
assignment-expression	), ;, {, ,
assignment-operator	condition-expression
condition-or-expression	), ;, {, ,
condition-or-expression-tail	), ;, {, ,
condition-and-expression	XX,), ;, {, ,
condition-and-expression-tail	XX,), ;, {, ,
equality-expression	==, !=, &&, XX,), ;, {, ,
equality-expression-tail	==, !=, &&, XX,), ;, {, ,
rel-expression	==, !=, &&, XX,), ;, {, ,
rel-expression-tail	==, !=, &&, XX,), ;, {, ,
additive-expression	<, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
additive-expression-tail	<, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
m-d-expression	+, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
m-d-expression-tail	+, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
u-expression	*, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
post-expression	*, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
post-expression-tail	*, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
primary-expression	., ++, --, *, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
para-list	->
proper-para-list	)
proper-para-list-tail	)
para-declaration	,,)
arg-list	., ++, --, *, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
proper-arg-list	)
proper-arg-list-tail	)
arg	,,)
declaration-statement	const, class, var, func, }, $, {, while, continue, if, return, break, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL

function-definition	{, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL, func, }
variable-declaration	var, const, class, func, }, $, {, while, continue, if, return, break, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
class-declaration	const, class, var, func, }, $, {, while, continue, if, return, break, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
class-body	;
class-member	}
constant-declaration	const, class, var, func, }, $, {, while, continue, if, return, break, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
init-expression	;
type-annotation	;, ,,)
type	{, ;, ,,)
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
7	statement → expression-statement	;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
8	statement → declaration-statement	const, class, var
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, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
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	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
20	expression-list → ε	;
21	variable-declaration-list → variable-declaration variable-declaration-list	var
22	variable-declaration-list → ε	
23	expression → assignment-expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
24	assignment-expression → condition-or-expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
25	assignment-expression → u-expression assignment-operator condition-expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
26	assignment-operator → =	=
27	assignment-operator → *=	*=
28	assignment-operator → /=	/=
29	assignment-operator → +=	+=
30	assignment-operator → -=	-=
31	assignment-operator → &&=	&&=
32	assignment-operator → XX=	XX=
33	condition-or-expression → condition-and-expression condition-or-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
34	condition-or-expression-tail → ε	), ;, {, ,
35	condition-or-expression-tail → XX condition-and-expression condition-or-expression-tail	XX
36	condition-and-expression → equality-expression condition-and-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
37	condition-and-expression-tail → && equality-expression equality-expression-tail	&&
38	condition-and-expression-tail → ε	XX,), ;, {, ,
39	equality-expression → rel-expression equality-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
40	equality-expression-tail → ε	==, !=, &&, XX,), ;, {, ,
41	equality-expression-tail → == rel-expression equality-expression-tail	==
42	equality-expression-tail → != rel-expression equality-expression-tail	!=
43	rel-expression → additive-expression rel-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL

44	rel-expression-tail → ε	==, !=, &&, XX,), ;, {, ,
45	rel-expression-tail → < additive-expression rel-expression-tail	<
46	rel-expression-tail → <= additive-expression rel-expression-tail	<=
47	rel-expression-tail → > additive-expression rel-expression-tail	>
48	rel-expression-tail → >= additive-expression rel-expression-tail	>=
49	additive-expression → m-d-expression additive-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
50	additive-expression-tail → ε	<, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
51	additive-expression-tail → + m-d-expression additive-expression-tail	+
52	additive-expression-tail → - m-d-expression additive-expression-tail	-
53	m-d-expression → u-expression m-d-expression-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
54	m-d-expression-tail → ε	+, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
55	m-d-expression-tail → * u-expression m-d-expression-tail	*
56	m-d-expression-tail → / u-expression m-d-expression-tail	/
57	u-expression → - u-expression	-
58	u-expression → ! u-expression	!
59	u-expression → ++ u-expression	++
60	u-expression → -- u-expression	--
61	u-expression → post-expression	identifier, (, INT-LITERAL, BOOL-LITERAL
62	post-expression → primary-expression	identifier, (, INT-LITERAL, BOOL-LITERAL
63	post-expression → primary-expression post-expression-tail	identifier, (, INT-LITERAL, BOOL-LITERAL
64	post-expression-tail → . identifier post-expression-tail	.
65	post-expression-tail → ++ post-expression-tail	++
66	post-expression-tail → -- post-expression-tail	--
67	post-expression-tail → ε	*, /, =, *=, /=, +=, -=, &&=, XX=, +, -, <, <=, >, >=, ==, !=, &&, XX,), ;, {, ,
68	primary-expression → identifier	identifier
69	primary-expression → identifier arg-list	identifier
70	primary-expression → (expression)	(
71	primary-expression → INT-LITERAL	INT-LITERAL
72	primary-expression → BOOL-LITERAL	BOOL-LITERAL
73	para-list → ( )	(
74	para-list → (proper-para-list)	(
75	proper-para-list → para-declaration proper-para-list-tail	identifier
76	proper-para-list-tail → , para-declaration proper-para-list-tail	,
77	proper-para-list-tail → ε	)
78	para-declaration → identifier type-annotation	identifier
79	arg-list → ( )	(
80	arg-list → (proper-arg-list)	(
81	proper-arg-list → arg proper-arg-list-tail	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
82	proper-arg-list-tail → , arg proper-arg-list-tail	,
83	proper-arg-list-tail → ε	)
84	arg → expression	-, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
85	declaration-statement → constant-declaration	const
86	declaration-statement → variable-declaration	var
87	declaration-statement → class-declaration	class
88	function-definition → func identifier para-list → type compound-statement	func
89	variable-declaration → var identifier init-expression ;	var
90	variable-declaration → var identifier type-annotation ;	var
91	class-declaration → class identifier class-body ;	class
92	class-body → { class-member }	{
93	class-member → declaration-statement class-member	const, class, var
94	class-member → function-definition class-member	func
95	class-member → ε	}
96	constant-declaration → const identifier init-expression ;	const
97	constant-declaration → const identifier type-annotation ;	const
98	init-expression → = expression	=

99	type-annotation → : type	:
100	type → int	int
101	type → bool	bool
102	top-level → statement top-level	{, while, continue, if, return, break, const, class, var, ;, -, !, ++, --, identifier, (, INT-LITERAL, BOOL-LITERAL
103	top-level → function-definition top-level	func
104	top-level → ε	

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)

```
[[0,"if","else","while","break",";","{","}","continue","return","condition-expression","=","*=","/=","+=","-
=","&&=","XX=","XX","&&","==","!=","<","<=",">",">=","+","-","*","/","!","++","--",".","identifier","(",")","INT-LITERAL","BOOL-
LITERAL",",","func","->","var","class","const",":","int","bool","$"],
[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0],
[0,2,106,3,4,7,1,105,5,6,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,7,106,106,7,7,7,106,7,7,106,7,7,106,105,1
[0,9,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,1
[0,105,106,10,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,
[0,105,106,105,11,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,
[0,105,105,105,105,105,12,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,
[0,14,106,14,14,14,14,13,14,14,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,14,106,106,14,14,14,106,14,14,106,1
[0,105,106,105,105,105,105,105,15,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,
[0,105,106,105,105,105,105,105,105,17,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,106,
[0,105,106,105,105,18,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,18,106,106,18,18,18,106,18,1
[0,106,106,106,106,20,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,19,106,106,19,19,19,106,19,1
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,23,106,106,23,23,23,106,23,
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,25,106,106,25,25,25,106,25,
[0,106,106,106,106,106,106,106,106,106,105,26,27,28,29,30,31,32,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,33,106,106,33,33,33,106,33,
[0,106,106,106,106,34,34,106,106,106,106,106,106,106,106,106,106,106,35,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,105,106,106,106,106,106,106,106,106,36,106,106,36,36,36,106,36,
[0,106,106,106,106,38,38,106,106,106,106,106,106,106,106,106,106,106,38,37,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,105,105,105,105,106,106,106,106,106,39,106,106,39,39,39,106,39,
[0,106,106,106,106,40,40,106,106,106,106,106,106,106,106,106,106,106,40,40,41,42,106,106,106,106,106,106,106,106,106,106,106,106,106,1
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,105,105,105,105,106,106,106,106,106,43,106,106,43,43,43,106,43,
[0,106,106,106,106,44,44,106,106,106,106,106,106,106,106,106,106,106,44,44,44,44,45,46,47,48,106,106,106,106,106,106,106,106,106,106,4
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,105,105,105,105,105,105,105,105,106,49,106,106,49,49,49,106,49,
[0,106,106,106,106,50,50,106,106,106,106,106,106,106,106,106,106,106,50,50,50,50,50,50,50,50,51,52,106,106,106,106,106,106,106,106,50,
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,105,105,105,105,105,105,105,105,105,53,106,106,53,53,53,106,53,
[0,106,106,106,106,54,54,106,106,106,106,106,106,106,106,106,106,106,54,54,54,54,54,54,54,54,54,54,55,56,106,106,106,106,106,106,54,10
[0,106,106,106,106,105,105,106,106,106,106,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,57,105,105,58,59,60,106,61,
[0,106,106,106,106,105,105,106,106,106,106,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,106,106,106,10
[0,106,106,106,106,67,67,106,106,106,106,67,67,67,67,67,67,67,67,67,67,67,67,67,67,67,67,67,67,67,106,65,66,64,106,106,67,106,106,67,1
[0,106,106,106,106,105,105,106,106,106,106,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,106,105,105,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,105,106,105,105,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,81,106,106,81,81,81,106,81,
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,84,106,106,84,84,84,106,84,
[0,105,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,10
[0,105,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,10
[0,105,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,10
[0,105,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,10
[0,106,106,106,106,105,92,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,
[0,106,106,106,106,106,106,95,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,
[0,105,106,105,105,105,105,105,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,105,106,106,105,105,105,10
[0,106,106,106,106,105,106,106,106,106,106,98,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,
[0,106,106,106,106,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
[0,106,106,106,106,105,105,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,10
```

[0,102,106,102,102,102,102,106,102,102,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,106,102,106,106,102,102,102,106

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":[39],"9":[5,-2,5,12,-1],"10":[5,12,-3],"11":[-5,-4],"12":[-7,6,-6],"14":[6,1],"15":[-5,-8],"16":[-5,12,-9],"17":[-5,-9],"18":[-5,10],"19":[12],"21":[11,41],"23":[13],"24":[15],"25":[-10,14,27],"26":[-11],"27":[-12],"28":[-13],"29":[-14],"30":[-15],"31":[-16],"32":[-17],"33":[16,17],"35":[16,17,-18],"36":[18,19],"37":[20,19,-19],"39":[20,21],"41":[20,21,-20],"42":[20,21,-21],"43":[22,23],"45":[22,23,-22],"46":[22,23,-23],"47":[22,23,-24],"48":[22,23,-25],"49":[24,25],"51":[24,25,-26],"52":[24,25,-27],"53":[26,27],"55":[26,27,-28],"56":[26,27,-29],"57":[27,-27],"58":[27,-30],"59":[27,-31],"60":[27,-32],"61":[28],"62":[30],"63":[29,30],"64":[29,-34,-33],"65":[29,-31],"66":[29,-32],"68":[-34],"69":[35,-34],"70":[-36,12,-35],"71":[-37],"72":[-38],"73":[-36,-35],"74":[-36,32,-35],"75":[33,34],"76":[33,34,-39],"78":[47,-34],"79":[-36,-35],"80":[-36,36,-35],"81":[37,38],"82":[37,38,-39],"84":[12],"85":[45],"86":[41],"87":[42],"88":[31,-34,-40],"89":[-5,46,-34,-42],"90":[-5,47,-34,-42],"91":[-5,43,-34,-43],"92":[-7,44,-6],"93":[44,39],"94":[44,40],"96":[-5,46,-34,-44],"97":[-5,47,-34,-44],"98":[12,-11],"99":[48,-45],"100":[-46],"101":[-47],"102":[49,1],"103":[49,40]}

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.
- "->" designates definition, "|" designates alternation, and newlines designate termination.
- x -> y | 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 -> b | EPSILON.
- Be certain to place spaces between things you don't want read as one symbol. (A) ≠ (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)