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 **L**eft-to-right, using **L**eftmost-derivations).

Follow sets are used in top-down parsers, but also in LR parsers (bottom-up parsers, reading **L**eft-to-right, using **R**ightmost 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. →

```
Program -> main ( )
{ declarations
statement_list }

declarations ->
data_type
identifier_list ;
declarations |
EPSILON

data_type -> int |
char

identifier_list ->
id
identifier_list_fac
tors

identifier_list_fac
tors -> EPSILON | ,
```

Click for Predict, First, and Follow Sets

First Set

Non-Terminal Symbol	First Set
main	main
(	(
)	)
{	{
}	}
;	;
ε	ε
int	int
char	char
id	id
,	,
[	[
number	number
]	]
=	=
simple-expn	simple-expn
num	num
if	if
else	else
while	while
for	for
==	==
!=	!=
<=	<=
>=	>=
>	>
<	<
+	+
-	-
*	*
/	/
%	%
Program	main
declarations	ε, int, char
data_type	int, char
identifier_list	id
identifier_list_factors	ε, ,, [
statement_list	ε, id, while, for, if
assign_stat	id
eprime	ε, ==, !=, <=, >=, >, <
seprime	ε, +, -

tprime	ε, *, /, %
factor	id, num
decision_stat	if
dprime	else, ε
looping_stat	while, for
relop	==, !=, <=, >=, >, <
addop	+, -
mulop	*, /, %
statement	id, while, for, if
term	id, num
simple_expn	id, num
expn	id, num

Follow Set

Non-Terminal Symbol	Follow Set
Program	$
declarations	), id, while, for, if
data_type	id
identifier_list	;
identifier_list_factors	;
statement_list	}
statement	id, while, for, if, }
assign_stat	;,)
expn	), ;
eprime	), ;
simple_expn	==, !=, <=, >=, >, <,), ;
seprime	==, !=, <=, >=, >, <,), ;
term	+, -, ==, !=, <=, >=, >, <,), ;
tprime	+, -, ==, !=, <=, >=, >, <,), ;
factor	*, /, %, +, -, ==, !=, <=, >=, >, <,), ;
decision_stat	id, while, for, if, }
dprime	id, while, for, if, }
looping_stat	id, while, for, if, }
relop	simple-expn
addop	id, num
mulop	id, num

Predict Set

#	Expression	Predict
1	Program → main ( ) { declarations statement_list }	main
2	declarations → data_type identifier_list ; declarations	int, char
3	declarations → ε	), id, while, for, if
4	data_type → int	int

5	data_type → char	char
6	identifier_list → id identifier_list_factors	id
7	identifier_list_factors → ε	;
8	identifier_list_factors → , identifier_list	,
9	identifier_list_factors → [number] , identifier_list	
10	identifier_list_factors → [number]	
11	statement_list → statement statement_list	id, while, for, if
12	statement_list → ε	}
13	statement → assign_stat ;	id
14	statement → decision_stat	if
15	statement → looping_stat	while, for
16	assign_stat → id = expn	id
17	expn → simple_expn eprime	id, num
18	eprime → relop simple-expn	==, !=, <=, >=, >, <
19	eprime → ε	), ;
20	simple_expn → term seprime	id, num
21	seprime → addop term seprime	+, -
22	seprime → ε	==, !=, <=, >=, >, <,), ;
23	term → factor tprime	id, num
24	tprime → mulop factor tprime	*, /, %
25	tprime → ε	+, -, ==, !=, <=, >=, >, <,), ;
26	factor → id	id
27	factor → num	num
28	decision_stat → if (expn) { statement_list } dprime	if
29	dprime → else { statement_list }	else
30	dprime → ε	id, while, for, if, }
31	looping_stat → while (expn) { statement_list }	while
32	looping_stat → for (assign_stat ; expn ; assign_stat) { statement_list }	for
33	relop → ==	==
34	relop → !=	!=
35	relop → <=	<=
36	relop → >=	>=
37	relop → >	>
38	relop → <	<
39	addop → +	+
40	addop → -	-
41	mulop → *	*
42	mulop → /	/
43	mulop → %	%

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,"main","(",")","{","}",";","int","char","id",",","[","number","]","=","simple-
expn","num","if","else","while","for","==","!=","<=",">=",">","<","+","-","*","/","%","$"],
[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],
[0,1,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,44],
[0,45,45,3,45,45,45,2,2,3,45,45,45,45,45,45,45,3,45,3,3,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,45,45,4,5,44,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,45,44,45,45,6,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,45,7,45,45,45,8,10,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,12,45,45,45,11,45,45,45,45,45,45,45,11,45,11,11,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,44,45,45,45,13,45,45,45,45,45,45,45,14,45,15,15,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,44,45,45,44,45,45,16,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,44,45,45,44,45,45,17,45,45,45,45,45,45,17,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,19,45,45,19,45,45,45,45,45,45,45,45,45,45,45,45,45,45,18,18,18,18,18,18,45,45,45,45,45,45],
[0,45,45,44,45,45,44,45,45,20,45,45,45,45,45,45,20,45,45,45,45,44,44,44,44,44,44,45,45,45,45,45,45],
[0,45,45,22,45,45,22,45,45,45,45,45,45,45,45,45,45,45,45,45,45,22,22,22,22,22,22,21,21,45,45,45,45],
[0,45,45,44,45,45,44,45,45,23,45,45,45,45,45,45,23,45,45,45,45,44,44,44,44,44,44,44,44,45,45,45,45],
[0,45,45,25,45,45,25,45,45,45,45,45,45,45,45,45,45,45,45,45,45,25,25,25,25,25,25,25,25,24,24,24,45],
[0,45,45,44,45,45,44,45,45,26,45,45,45,45,45,45,27,45,45,45,45,44,44,44,44,44,44,44,44,44,44,44,45],
[0,45,45,45,45,44,45,45,45,44,45,45,45,45,45,45,45,28,45,44,44,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,30,45,45,45,30,45,45,45,45,45,45,45,30,29,30,30,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,44,45,45,45,44,45,45,45,45,45,45,45,44,45,31,32,45,45,45,45,45,45,45,45,45,45,45,45],
[0,45,45,45,45,45,45,45,45,45,45,45,45,45,45,44,45,45,45,45,45,33,34,35,36,37,38,45,45,45,45,45,45],
[0,45,45,45,45,45,45,45,45,44,45,45,45,45,45,45,44,45,45,45,45,45,45,45,45,45,45,39,40,45,45,45,45],
[0,45,45,45,45,45,45,45,45,44,45,45,45,45,45,45,44,45,45,45,45,45,45,45,45,45,45,45,45,41,42,43,45]]

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":[-5,6,2,-4,-3,-2,-1],"2":[2,-6,4,3],"4":[-7],"5":[-8],"6":[5,-9],"8":[4,-10],"9":[4,-10,-13,-12,-11],"10":
[-13,-12,-11],"11":[6,7],"13":[-6,8],"14":[16],"15":[18],"16":[9,-14,-9],"17":[10,11],"18":[-15,19],"20":
[12,13],"21":[12,13,20],"23":[14,15],"24":[14,15,21],"26":[-9],"27":[-16],"28":[17,-5,6,-4,-3,9,-2,-17],"29":
[-5,6,-4,-18],"31":[-5,6,-4,-3,9,-2,-19],"32":[-5,6,-4,-3,8,-6,9,-6,8,-2,-20],"33":[-21],"34":[-22],"35":
[-23],"36":[-24],"37":[-25],"38":[-26],"39":[-27],"40":[-28],"41":[-29],"42":[-30],"43":[-31]}

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)