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> -> program
id (
<identifier_list> )
; <declarations>
<subprogram_declarat
ions>
<compound_statement>
.

<identifier_list> ->
id
<resto_identifier_li
st>

<resto_identifier_li
st> -> , id
<resto_identifier_li
st> | LAMBDA

<declarations> ->
```

Click for Predict, First, and Follow Sets

First Set

Non-Terminal Symbol	First Set
program	program
id	id
(	(
)	)
;	;
.	.
,	,
λ	λ
var	var
:	:
array	array
[	[
num	num
..	..
]	]
of	of
integer	integer
real	real
<subprogram_declarion>	<subprogram_declarion>
function	function
procedure	procedure
begin	begin
end	end
assignop	assignop
while	while
do	do
if	if
then	then
else	else
relop	relop
addop	addop
mulop	mulop
not	not
<program>	program
<identifier_list>	id
<resto_identifier_list>	,, λ
<declarations>	var, λ
<type>	array, integer, real
<standard_type>	integer, real
<subprogram_declarations>	<subprogram_declarion>, λ
<subprogram_head>	function, procedure
<arguments>	(, λ
<resto_parameter_list>	;, λ
<compound_statement>	begin
<optional_statements>	λ, while, id, begin, if
<resto_statement_list>	;, λ
<statement>	while, id, begin, if
<if_statement>	if
<opc_else>	else, λ
<variable>	id

<opc_index>	[, λ
<procedure_statement>	id
<opc_parameters>	(, λ
<resto_expression_list>	,, λ
<resto_expression>	relop, λ
<resto_simple_expression>	addop, λ
<resto_term>	mulop, λ
<uno>	addop, id, num, (, not
<factor>	id, num, (, not
<resto_id>	(, λ
<subprogram_declaration>	function, procedure
<parameter_list>	id
<term>	addop, id, num, (, not
<simple_expression>	addop, id, num, (, not
<statement_list>	while, id, begin, if
<expression>	addop, id, num, (, not
<expression_list>	addop, id, num, (, not

Follow Set

Non-Terminal Symbol	Follow Set
<program>	$
<identifier_list>	:,)
<resto_identifier_list>	:,)
<declarations>	begin, (, <subprogram_declarion>
<type>	;,)
<standard_type>	;,)
<subprogram_declarations>	begin
<subprogram_declaration>	
<subprogram_head>	begin, var
<arguments>	:, ;
<parameter_list>	)
<resto_parameter_list>	)
<compound_statement>	., else, ;, end
<optional_statements>	end
<statement_list>	end
<resto_statement_list>	end
<statement>	else, ;, end
<if_statement>	else, ;, end
<opc_else>	else, ;, end
<variable>	assignop
<opc_index>	assignop
<procedure_statement>	else, ;, end
<opc_parameters>	else, ;, end
<expression_list>	)
<resto_expression_list>	)
<expression>	), ,,], then, do, else, ;, end
<resto_expression>	), ,,], then, do, else, ;, end
<simple_expression>	relop,), ,,], then, do, else, ;, end
<resto_simple_expression>	relop,), ,,], then, do, else, ;, end
<term>	addop, relop,), ,,], then, do, else, ;, end
<resto_term>	addop, relop,), ,,], then, do, else, ;, end
<uno>	mulop, addop, relop,), ,,], then, do, else, ;, end
<factor>	mulop, addop, relop,), ,,], then, do, else, ;, end
<resto_id>	mulop, addop, relop,), ,,], then, do, else, ;, end

Predict Set

#	Expression	Predict
1	<program> → program id (<identifier_list>) ; <declarations> <subprogram_declarations> <compound_statement> .	program
2	<identifier_list> → id <resto_identifier_list>	id
3	<resto_identifier_list> → , id <resto_identifier_list>	,
4	<resto_identifier_list> → λ	:,)
5	<declarations> → var <identifier_list> : <type> ; <declarations>	var
6	<declarations> → λ	begin, (, <subprogram_declarion>
7	<type> → <standard_type>	integer, real
8	<type> → array [num .. num] of <standard_type>	array
9	<standard_type> → integer	integer
10	<standard_type> → real	real
11	<subprogram_declarations> → <subprogram_declarion> ; <subprogram_declarations>	<subprogram_declarion>
12	<subprogram_declarations> → λ	begin
13	<subprogram_declaration> → <subprogram_head> <declarations> <compound_statement>	function, procedure
14	<subprogram_head> → function id <arguments> : <standard_type> ;	function
15	<subprogram_head> → procedure id <arguments> ;	procedure
16	<arguments> → (<parameter_list>)	(
17	<arguments> → λ	:, ;
18	<parameter_list> → <identifier_list> : <type> <resto_parameter_list>	id
19	<resto_parameter_list> → ; <identifier_list> : <type> <resto_parameter_list>	;
20	<resto_parameter_list> → λ	)
21	<compound_statement> → begin <optional_statements> end	begin
22	<optional_statements> → <statement_list>	while, id, begin, if
23	<optional_statements> → λ	end
24	<statement_list> → <statement> <resto_statement_list>	while, id, begin, if
25	<resto_statement_list> → ; <statement> <resto_statement_list>	;
26	<resto_statement_list> → λ	end
27	<statement> → <variable> assignop <expression>	id
28	<statement> → <procedure_statement>	id
29	<statement> → <compound_statement>	begin
30	<statement> → <if_statement>	if
31	<statement> → while <expression> do <statement>	while
32	<if_statement> → if <expression> then <statement> <opc_else>	if
33	<opc_else> → else <statement>	else
34	<opc_else> → λ	else, ;, end
35	<variable> → id <opc_index>	id
36	<opc_index> → [<expression>]	
37	<opc_index> → λ	assignop
38	<procedure_statement> → id <opc_parameters>	id
39	<opc_parameters> → (<expression_list>)	(

40 <opc_parameters> → λ	else, ;, end
41 <expression_list> → <expression> <resto_expression_list>	addop, id, num, (, not
42 <resto_expression_list> → , <expression> <resto_expression_list>	,
43 <resto_expression_list> → λ	)
44 <expression> → <simple_expression> <resto_expression>	addop, id, num, (, not
45 <resto_expression> → relop <simple_expression> <resto_expression>	relop
46 <resto_expression> → λ	), ,, , then, do, else, ;, end
47 <simple_expression> → <term> <resto_simple_expression>	addop, id, num, (, not
48 <resto_simple_expression> → addop <term> <resto_simple_expression>	addop
49 <resto_simple_expression> → λ	relop,), ,, , then, do, else, ;, end
50 <term> → <uno> <resto_term>	addop, id, num, (, not
51 <resto_term> → mulop <uno> <resto_term>	mulop
52 <resto_term> → λ	addop, relop,), ,, , then, do, else, ;, end
53 <uno> → <factor>	id, num, (, not
54 <uno> → addop <factor>	addop
55 <factor> → id <resto_id>	id
56 <factor> → num	num
57 <factor> → (<expression>)	(
58 <factor> → not <factor>	not
59 <resto_id> → (<expression_list>)	(
60 <resto_id> → λ	mulop, addop, relop,), ,, , then, do, else, ;, end

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,"program","id","(",")",";",".",",","var",":","array","[","num","..","]","of","integer","real","<subprogram_declarion>","function","procedure","begin","end","assignop","while","do","if","then","else","relop","addop","mulop","not","$"],
[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],[0,1,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,61],
[0,62,2,62,61,62,62,62,62,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,62,62,4,62,62,3,62,4,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],
[0,62,62,6,62,62,62,62,5,62,62,62,62,62,62,62,62,6,62,62,6,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,62,62,61,61,62,62,62,62,8,62,62,62,62,62,7,7,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],
[0,62,62,62,61,61,62,62,62,62,62,62,62,62,62,62,9,10,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,11,62,62,12,62,62,62,62,62,62,62,62,62,62,62],
[0,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,13,13,62,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,62,62,62,62,62,62,61,62,62,62,62,62,62,62,62,62,62,14,15,61,62,62,62,62,62,62,62,62,62,62,62,62],
[0,62,62,16,62,17,62,62,62,17,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,18,62,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],
[0,62,62,62,20,19,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],[0,62,62,62,62,61,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,21,61,62,62,62,62,62,61,62,62,62,62,62],
[0,62,22,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,22,23,62,22,62,22,62,62,62,62,62,62,62],[0,62,24,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,24,61,62,24,62,24,62,62,62,62,62,62,62],
[0,62,62,62,62,25,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,26,62,62,62,62,62,62,62,62,62,62,62],[0,62,28,62,62,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,29,61,62,31,62,30,62,61,62,62,62,62,62],
[0,62,62,62,62,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,61,62,62,62,32,62,61,62,62,62,62,62],[0,62,62,62,62,34,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,34,62,62,62,62,62,34,62,62,62,62,62],
[0,62,35,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,61,62,62,62,62,62,62,62,62,62],[0,62,62,62,62,62,62,62,62,62,62,36,62,62,62,62,62,62,62,62,62,62,62,37,62,62,62,62,62,62,62,62,62,62],
[0,62,38,62,62,61,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,61,62,62,62,62,62,61,62,62,62,62,62],[0,62,62,39,62,40,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,40,62,62,62,62,62,40,62,62,62,62,62],
[0,62,41,41,61,62,62,62,62,62,62,62,41,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,41,62,41,62],[0,62,62,62,43,62,62,42,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62,62],
[0,62,44,44,61,61,62,61,62,62,62,62,44,62,61,62,62,62,62,62,62,62,61,62,62,61,62,61,61,62,44,62,44,62],[0,62,62,62,46,46,62,46,62,62,62,62,62,62,46,62,62,62,62,62,62,62,46,62,62,46,62,46,46,45,62,62,62,62],
[0,62,47,47,61,61,62,61,62,62,62,62,47,62,61,62,62,62,62,62,62,62,61,62,62,61,62,61,61,61,47,62,47,62],[0,62,62,62,49,49,62,49,62,62,62,62,62,62,49,62,62,62,62,62,62,62,49,62,62,49,62,49,49,49,48,62,62,62],
[0,62,50,50,61,61,62,61,62,62,62,62,50,62,61,62,62,62,62,62,62,62,61,62,62,61,62,61,61,61,50,62,50,62],[0,62,62,62,52,52,62,52,62,62,62,62,62,62,52,62,62,62,62,62,62,62,52,62,62,52,62,52,52,52,52,51,62,62],
[0,62,53,53,61,61,62,61,62,62,62,62,53,62,61,62,62,62,62,62,62,62,61,62,62,61,62,61,61,61,54,61,53,62],[0,62,55,57,61,61,62,61,62,62,62,62,56,62,61,62,62,62,62,62,62,62,61,62,62,61,62,61,61,61,61,61,58,62],
[0,62,62,59,60,60,62,60,62,62,62,62,62,62,60,62,62,62,62,62,62,62,60,62,62,60,62,60,60,60,60,60,62,62]]

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":[-6,13,7,4,-5,-4,2,-3,-2,-1],"2":[3,-2],"3":[3,-2,-7],"5":[4,-5,5,-9,2,-8],"7":[6],"8":[6,-15,-14,-12,-13,-12,-11,-10],"9":[-16],"10":[-17],"11":[7,-5,-18],"13":[13,4,9],"14":[-5,6,-9,10,-2,-19],"15":[-5,10,-2,-20],"16":[-4,11,-3],"18":[12,5,-9,2],"19":[12,5,-9,2,-5],"21":[-22,14,-21],"22":[15],"24":[16,17],"25":[16,17,-5],"27":[26,-23,20],"28":[22],"29":[13],"30":[18],"31":[17,-25,26,-24],"32":[19,17,-27,26,-26],"33":[17,-28],"35":[21,-2],"36":[-14,26,-11],"38":[23,-2],"39":[-4,24,-3],"41":[25,26],"42":[25,26,-7],"44":[27,28],"45":[27,28,-29],"47":[29,30],"48":[29,30,-30],"50":[31,32],"51":[31,32,-31],"53":[33],"54":[33,-30],"55":[34,-2],"56":[-12],"57":[-4,26,-3],"58":[33,-32],"59":[-4,24,-3]}

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)