LL(1) Parser Visualization

Write your own context-free grammar and see an LL(1) parser in action!

Written by Zak Kincaid and Shaowei Zhu based on
http://jmachines.sourceforge.net/machines/ll1.html

1. Write your LL(1) grammar (empty string '' represents ε):

```
ARG_D -> TYPE
ARG_DN
ARG_DN -> colon
ARG_D
ARG_DN -> ''
ASSIGEN_MAY -> ''
ASSIGEN_MAY ->
assign expresion
TYPE -> integer
TYPE -> nil
TYPE -> number
TYPE -> string
RANGE -> local
RANGE -> global
```

Valid LL(1) Grammars

For any production S -> A | B, it must be the case that:

- For no terminal t could A and B derive strings beginning with t
- At most one of A and B can derive the empty string
- If B can derive the empty string, then A does not derive any string beginning with a terminal in Follow(A)

Formatting Instructions

- The non-terminal on the left-hand-side of the first rule is the start non-terminal
- Write each production rule in a separate line (see example to the left)
- Separate each token using whitespace
- $ is reserved as the end-of-input symbol, and S is reserved as an artificial start symbol. The grammar is automatically augmented with the rule S -> start $

Debugging

- More information about the parser construction is printed on the console
- The source code follows the pseudocode in lecture. In particular, see `computeNullable`, `computeFirst`, `computeFollow`, and `computeAllTables`

Generate tables

2. Nullable/First/Follow Table and Transition Table

Nonterminal	Nullable?	First	Follow
S	✕	require	
START	✕	require	$
PROLOG	✕	require	id, function, while, if, local, global, $
PROG	✓	id, function, while, if, local, global	end, else, return, $
ID_NEXT	✓	comma	assigen
N_EXPRESIONS	✓	comma	id, function, while, if, local, global, end, else, return, $
WHILE	✕	while	id, function, while, if, local, global, end, else, return, $
IF	✕	if	id, function, while, if, local, global, end, else, return, $
ELSE_M	✓	else	end
FUNCTION	✕	function	id, function, while, if, local, global, end, else, return, $
ARG	✓	id	rbr, comma
ARGNEXT	✓	comma	rbr
RETURN_D	✓	integer, nil, number, string	end, return, id, function, while, if, local, global
RETURN_DN	✓	comma	end, return, id, function, while, if, local, global
RETURN	✓	return	end
RETURN_ARG	✓	expresion	end
RETURN_ARG_N	✓	comma	end
DECLARATION	✕	local, global	id, function, while, if, local, global, end, else, return, $
DECLARATION_T	✕	function, integer, nil, number, string	id, function, while, if, local, global, end, else, return, $
ARG_D	✓	integer, nil, number, string	id, function, while, if, local, global, rbr, end, else, return, $
ARG_DN	✓	colon	id, function, while, if, local, global, rbr, end, else, return, $
ASSIGEN_MAY	✓	assigen	id, function, while, if, local, global, end, else, return, $
TYPE	✕	integer, nil, number, string	rbr, comma, end, return, id, function, while, if, local, global, assigen, colon, else, $
RANGE	✕	local, global	id

	$	require	string	id	assigen	expresion	comma	while	do	end	if	then	else	function	ibr	rbr	colon	return	integer	nil	number	local	global
S		S -> START $																					
START		START -> PROLOG PROG																					
PROLOG		PROLOG -> require string																					
PROG	PROG -> ε			PROG -> id ID_NEXT assigen expresion N_EXPRESIONS PROG				PROG -> WHILE PROG		PROG -> ε	PROG -> IF PROG		PROG -> ε	PROG -> FUNCTION PROG				PROG -> ε				PROG -> DECLARATION PROG	PROG -> DECLARATION PROG
ID_NEXT					ID_NEXT -> ε		ID_NEXT -> comma id ID_NEXT																
N_EXPRESIONS	N_EXPRESIONS -> ε			N_EXPRESIONS -> ε			N_EXPRESIONS -> comma expresion N_EXPRESIONS	N_EXPRESIONS -> ε		N_EXPRESIONS -> ε	N_EXPRESIONS -> ε		N_EXPRESIONS -> ε	N_EXPRESIONS -> ε				N_EXPRESIONS -> ε			N_EXPRESIONS -> ε	N_EXPRESIONS -> ε
WHILE								WHILE -> while expresion do PROG end															
IF											IF -> if expresion then PROG ELSE_M end												
ELSE_M										ELSE_M -> ε			ELSE_M -> else PROG										
FUNCTION														FUNCTION -> function id lbr ARG ARGNEXT rbr RETURN_D PROG RETURN end									
ARG				ARG -> id colon TYPE			ARG -> ε									ARG -> ε							
ARGNEXT							ARGNEXT -> comma ARG ARGNEXT									ARGNEXT -> ε							
RETURN_D			RETURN_D -> TYPE RETURN_DN	RETURN_D -> ε				RETURN_D -> ε		RETURN_D -> ε	RETURN_D -> ε			RETURN_D -> ε				RETURN_D -> ε	RETURN_D -> TYPE RETURN_DN	RETURN_D -> TYPE RETURN_DN	RETURN_D -> TYPE RETURN_DN	RETURN_D -> ε	RETURN_D -> ε
RETURN_DN				RETURN_DN -> ε			RETURN_DN -> comma RETURN_D	RETURN_DN -> ε		RETURN_DN -> ε	RETURN_DN -> ε			RETURN_DN -> ε				RETURN_DN -> ε				RETURN_DN -> ε	RETURN_DN -> ε
RETURN										RETURN -> ε								RETURN -> return RETURN_ARG					
RETURN_ARG						RETURN_ARG -> expresion RETURN_ARG_N				RETURN_ARG -> ε													
RETURN_ARG_N							RETURN_ARG_N -> comma RETURN_ARG			RETURN_ARG_N -> ε													
DECLARATION																						DECLARATION -> RANGE id colon DECLARATION_T	DECLARATION -> RANGE id colon DECLARATION_T
DECLARATION_T			DECLARATION_T -> TYPE ASSIGEN_MAY											DECLARATION_T -> function lbr ARG_D rbr colon ARG_D					DECLARATION_T -> TYPE ASSIGEN_MAY	DECLARATION_T -> TYPE ASSIGEN_MAY	DECLARATION_T -> TYPE ASSIGEN_MAY		
ARG_D	ARG_D -> ε		ARG_D -> TYPE ARG_DN		ARG_D -> ε			ARG_D -> ε		ARG_D -> ε	ARG_D -> ε		ARG_D -> ε	ARG_D -> TYPE ARG_DN		ARG_D -> ε		ARG_D -> TYPE ARG_DN	ARG_D -> TYPE ARG_DN	ARG_D -> TYPE ARG_DN	ARG_D -> TYPE ARG_DN	ARG_D -> ε	ARG_D -> ε
ARG_DN	ARG_DN -> ε			ARG_DN -> ε				ARG_DN -> ε		ARG_DN -> ε	ARG_DN -> ε		ARG_DN -> ε	ARG_DN -> ε		ARG_DN -> ε	ARG_DN -> colon ARG_D	ARG_DN -> ε				ARG_DN -> ε	ARG_DN -> ε
ASSIGEN_MAY	ASSIGEN_MAY -> ε			ASSIGEN_MAY -> ε	ASSIGEN_MAY -> assigen expresion			ASSIGEN_MAY -> ε		ASSIGEN_MAY -> ε	ASSIGEN_MAY -> ε		ASSIGEN_MAY -> ε	ASSIGEN_MAY -> ε				ASSIGEN_MAY -> ε				ASSIGEN_MAY -> ε	ASSIGEN_MAY -> ε
TYPE			TYPE -> string																TYPE -> integer	TYPE -> nil	TYPE -> number		
RANGE																						RANGE -> local	RANGE -> global

3. Parsing

Token stream separated by spaces: id + id

Start/Reset Step Forward

Stack

Remaining Input

Rule

Partial Parse Tree