Universidad Veracruzana

Inteligencia Artificial

Autómatas y Lenguajes Formales

Proyecto: LR(1) Parser

Leonardo Flores Torres

5 de enero de 2023

La gramática libre de contexto a utilizar para este proyecto es

$$\begin{aligned} \mathrm{S} &\to \mathrm{xSz} \\ \mathrm{S} &\to \mathrm{xyTyz} \\ \mathrm{T} &\to \lambda \end{aligned} \tag{1}$$

pero también se quiso tomar a la gramática libre de contexto vista en clase

$$\begin{aligned} \mathrm{S} &\to \mathrm{zMNz} \\ \mathrm{M} &\to \mathrm{aMa} \\ \mathrm{M} &\to \mathrm{z} \\ \mathrm{N} &\to \mathrm{bNb} \\ \mathrm{N} &\to \mathrm{z} \end{aligned} \tag{2}$$

para corroborar los resultados del autómata obtenido, la tabla del parser, y para probar la generalidad de la implementación del algoritmo escrito en `julia` [1]. La implementación se incluye en el Apéndice C de este trabajo.

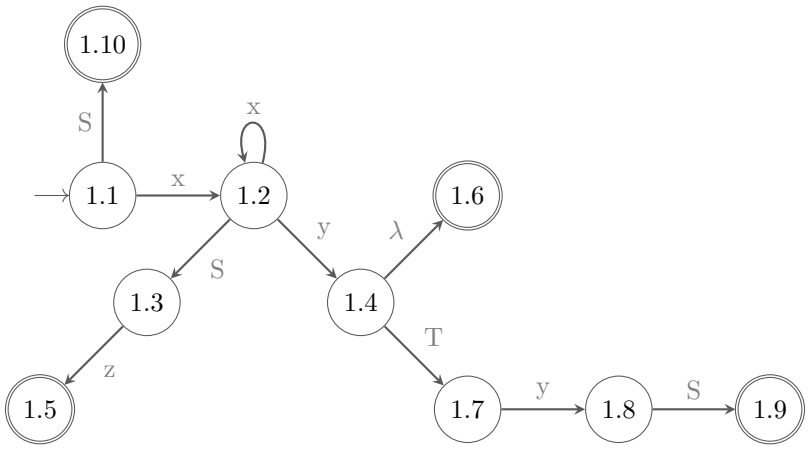

Figura 1: Autómata finito para la gramática 1.

Los estados del autómata de la gramática (1) se incluyen en el Apéndice A, y los del autómata correspondiente a la gramática (2) en el Apéndice B, en ambos casos los estados aparecen enumerados para coincidir con los diagramas de sus respectivos autómatas finitos. De igual manera, los autómatas finitos de las gramáticas (1) y (2) se muestran en las figuras 1 y 2, y sus respectivas tablas de parsers en las tablas 1 y 2 al final del documento.

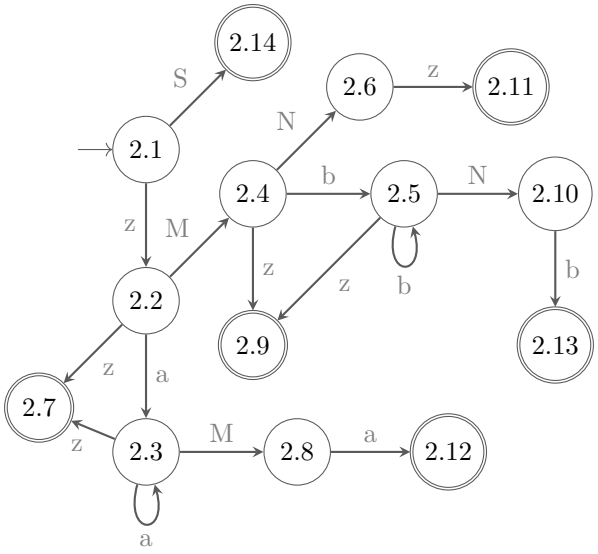

Figura 2: Autómata finito para la gramática 2.

Se puede utilizar a la gramática (2) para encontrar algunas cadenas, por ejemplo:

S	S → zMNz
zMNz	N → bNb
zMbNbz	N → z
zMbzbz	M → aMa
zaMabzbz	M → z
zazabzbz	

La siguiente cadena también es válida para esta gramática:

S	S → zMNz
zMNz	N → bNb
zMbNbz	N → bNb
zMbbNbbz	N → z
zMbbzbbz	M → z
zzbbzbbz	

Lo interesante ahora sería probar estas cadenas con la implementación del algoritmo. Primero se carga el módulo hecho en `julia`, se define la cadena de entrada en la variable `input_string` y se llama a la rutina del algoritmo llamando a la función `grammar_two` donde se encuentra definida a la gramática (2) y al final de esta función se llama internamente a la función `parse_string`. Lo anteriormente mencionado se muestra a continuación:

```
julia> using Revise; using LR1Parser; lrp = LR1Parser;

julia> input_string = "zazabzbz"; lrp.grammar_two(input_string)    # Primera cadena de prueba
Iteration 1: Any[1]

Current symbol: z
```

```
Iteration 2: Any[1, "z", 2]
Current symbol: a
Iteration 3: Any[1, "z", 2, "a", 3]
Current symbol: z
Iteration 4: Any[1, "z", 2, "a", 3, "z", 7]
Current symbol: a
Iteration 5: Any[1, "z", 2, "a", 3, "M", 8]
Iteration 6: Any[1, "z", 2, "a", 3, "M", 8, "a", 12]
Current symbol: b
Iteration 7: Any[1, "z", 2, "M", 4]
Iteration 8: Any[1, "z", 2, "M", 4, "b", 5]
Current symbol: z
Iteration 9: Any[1, "z", 2, "M", 4, "b", 5, "z", 9]
Current symbol: b
Iteration 10: Any[1, "z", 2, "M", 4, "b", 5, "N", 10]
Iteration 11: Any[1, "z", 2, "M", 4, "b", 5, "N", 10, "b", 13]
Current symbol: z
Iteration 12: Any[1, "z", 2, "M", 4, "N", 6]
Iteration 13: Any[1, "z", 2, "M", 4, "N", 6, "z", 11]
Current symbol: eos
Iteration 14: Any[1, "S", 14]
The string "zazabzbz" was succesfully parsed.
julia> input_string = "zzbbzbbz"; lrp.grammar_two(input_string)    # Segunda cadena de prueba
Iteration 1: Any[1]
Current symbol: z
Iteration 2: Any[1, "z", 2]
Current symbol: z
```

```
Iteration 3: Any[1, "z", 2, "z", 7]
Current symbol: b
Iteration 4: Any[1, "z", 2, "M", 4]
Iteration 5: Any[1, "z", 2, "M", 4, "b", 5]
Current symbol: b
Iteration 6: Any[1, "z", 2, "M", 4, "b", 5, "b", 5]
Current symbol: z
Iteration 7: Any[1, "z", 2, "M", 4, "b", 5, "b", 5, "z", 9]
Current symbol: b
Iteration 8: Any[1, "z", 2, "M", 4, "b", 5, "b", 5, "N", 10]
Iteration 9: Any[1, "z", 2, "M", 4, "b", 5, "b", 5, "N", 10, "b", 13]
Current symbol: b
Iteration 10: Any[1, "z", 2, "M", 4, "b", 5, "N", 10]
Iteration 11: Any[1, "z", 2, "M", 4, "b", 5, "N", 10, "b", 13]
Current symbol: z
Iteration 12: Any[1, "z", 2, "M", 4, "N", 6]
Iteration 13: Any[1, "z", 2, "M", 4, "N", 6, "z", 11]
Current symbol: eos
Iteration 14: Any[1, "S", 14]
The string "zzbbzbbz" was succesfully parsed.
```

Se pensó que sería útil mostrar los pasos de cada iteración del `while-loop` dentro de la implementación del algoritmo, de esta manera se podría corroborar el estado del `stack` conforme se agregan símbolos y los números correspondientes a los estados en el autómata finito definidos como `token`s. Además, para cadenas cortas, esto serviría para seguir los pasos a mano y realizar una comparación mas detalladamente de los contenidos del stack cada que se realiza un `pop` o un `push`.

Lo mismo se puede hacer para la gramática (1). Una cadena que se puede deducir a partir de esta gramática es la

siguiente:

S	$S \rightarrow xSz$
xSz	$S \rightarrow xyTyz$
xxyTyzz	$T \rightarrow \lambda$
xxyyzz	

Una segunda cadena perteneciente a esta gramática es:

S	$S \rightarrow xSz$
xSz	$S \rightarrow xSz$
xxSzz	$S \rightarrow xyTyz$
xxxyTyzzz	$T \rightarrow \lambda$
xxxyyzzz	

Con estas dos cadenas se probará el algoritmo. De manera similar al caso anterior, se definió una función `grammar_one` en donde se define a la gramática (1), e internamente se llama a `parse_string`:

```
julia> input_string = "xxyyzz"; lrp.grammar_one(input_string)    # Primera cadena de prueba
Iteration 1: Any[1]

Current symbol: x

Iteration 2: Any[1, "x", 2]

Current symbol: x

Iteration 3: Any[1, "x", 2, "x", 2]

Current symbol: y

Iteration 4: Any[1, "x", 2, "x", 2, "y", 4]

Current symbol: y

Iteration 5: Any[1, "x", 2, "x", 2, "y", 4, "T", 7]

Iteration 6: Any[1, "x", 2, "x", 2, "y", 4, "T", 7, "y", 8]

Current symbol: z

Iteration 7: Any[1, "x", 2, "x", 2, "y", 4, "T", 7, "y", 8, "z", 9]

Current symbol: z

Iteration 8: Any[1, "x", 2, "S", 3]

Iteration 9: Any[1, "x", 2, "S", 3, "z", 5]

Current symbol: eos

Iteration 10: Any[1, "S", 10]
```

```
The string "xxyyzz" was succesfully parsed.

julia> input_string = "xxxyyzzz"; lrp.grammar_one(input_string)    # Segunda cadena de prueba
Iteration 1: Any[1]

Current symbol: x

Iteration 2: Any[1, "x", 2]

Current symbol: x

Iteration 3: Any[1, "x", 2, "x", 2]

Current symbol: x

Iteration 4: Any[1, "x", 2, "x", 2, "x", 2]

Current symbol: y

Iteration 5: Any[1, "x", 2, "x", 2, "x", 2, "y", 4]

Current symbol: y

Iteration 6: Any[1, "x", 2, "x", 2, "x", 2, "y", 4, "T", 7]

Iteration 7: Any[1, "x", 2, "x", 2, "x", 2, "y", 4, "T", 7, "y", 8]

Current symbol: z

Iteration 8: Any[1, "x", 2, "x", 2, "x", 2, "y", 4, "T", 7, "y", 8, "z", 9]

Current symbol: z

Iteration 9: Any[1, "x", 2, "x", 2, "S", 3]

Iteration 10: Any[1, "x", 2, "x", 2, "S", 3, "z", 5]

Current symbol: z

Iteration 11: Any[1, "x", 2, "S", 3]

Iteration 12: Any[1, "x", 2, "S", 3, "z", 5]

Current symbol: eos

Iteration 13: Any[1, "S", 10]

The string "xxxyyzzz" was succesfully parsed.
```

Un par de elementos que se añadió al algoritmo en `parse_string` es entrar tempranamente a la rutina de error si detecta que un caracter en la cadena de entrada no pertenece al alfabeto de la gramática. Esto se implementó como se muestra a continuación (se puede observar más detalladamente este extracto de código en el Apéndice C):

```
    string_ = split(input_string, "")

    # Early termination if a character in the input string is not part of the
    # alphabet of the grammar
    if !all(x -> x in keys(symbols), string_)
        error("One or more characters in the input string \"$(input_string)\" do not belong to the symbols
        ↪  $(keys(symbols)) of the grammar.")
    end
```

Por ejemplo, si se usara la siguiente cadena de entrada `"xxybyzz"` con la gramática (1), el caracter `"b"` no es parte del alfabeto de esa gramática, y el algoritmo debería entrar a la rutina de error:

```
julia> input_string = "xxybyzz"; lrp.grammar_one(input_string)
ERROR: One or more characters in the input string "xxybyzz" do not belong to the symbols ["eos", "S", "T",
↪  "x", "z", "y"] of the grammar.
```

Recuérdese que a pesar de que se llame a la función `grammar_one`, esta a su vez llama internamente a `parse_string` entonces siempre se está llamando al algoritmo del parser. Otra rutina de error que se añadió fue si se llegara a detectar el caso en que el símbolo al tope del `stack`, por algún motivo, no coincide con el símbolo en curso del lado derecho de la regla de reescritura:

```
            for rs_symbol in reverse(right_side)
                # This condition assumes that the right side of the rewrite rule is of
                # the form A -> λ
                if !isequal(rs_symbol, "λ")
                    symbol_to_pop = stack[begin+1]    # First comes the node's number, and second is the
                    ↪  symbol to pop

                    if !isequal(rs_symbol, symbol_to_pop)
                        error("A symbol from the current rewrite rule $(right_side) does not match the symbol
                        ↪  \"$(symbol_to_pop)\" at the top of the stack.")
                    end

                    popfirst!(stack)    # To pop the table entry
                    popfirst!(stack)    # To pop the symbol
                end
            end
```

Esta última rutina de error se añadió al momento de ir probando el buen funcionamiento del algoritmo y se decidió mantenerla. Se mantuvo la última operación de vaciar el `stack` al final del algoritmo, aunque en `julia` no es necesario ya que al momento de salir de la función `parse_string` toda variable local (variable definida al interior de una función) que no sea regresada por la función misma se limpia.

La implementación solamente se hizo para un parser del tipo LR($k = 1$), esto quiere decir que el parser solamente puede leer un símbolo a la vez. Como trabajo posterior sería interesante pensar en cómo realizar la implementación para $k \neq 1$, con esto me refiero a poder elegir un valor arbitrario de k sin que la implementación sea dependiente del valor asignado. El parser hecho en este trabajo no puede manejar otro valor de k que no sea 1. Lo difícil en esto no es el algoritmo en `parse_string` per se sino en idear un algoritmo que genere automáticamente la tabla de transiciones a partir de k.

Finalmente, se quisiera incluir un caso extra que resulta en error. El algoritmo puede manejar 3 condiciones dentro del `while-loop`. La primera es cuando la entrada en la tabla de su respectiva gramática es un `"shift"`, la segunda condición es cuando la entrada es una regla de reescritura (también llamada reducción) la cual en este trabajo se identificó como `"redux"`, y la tercera condición es cuando la entrada de la tabla está vacía. Este último caso, cuando la entrada está vacía, es el que detona una rutina de error y se dá cuando todos los símbolos de la cadena de entrada sí pertenecen al alfabeto de la gramática pero la cadena no puede ser generada a partir de la misma. Esto se ejemplificará con las cadenas `"xxxyyyzzz"` y `"zzzbbzbbz"` de las gramáticas (1) y (2), respectivamente. A continuación se muestra lo descrito anteriormente:

```
julia> input_string = "xxxyyyzzz"; lrp.grammar_one(input_string)    # Primera prueba de error
Iteration 1: Any[1]

Current symbol: x

Iteration 2: Any[1, "x", 2]

Current symbol: x

Iteration 3: Any[1, "x", 2, "x", 2]

Current symbol: x

Iteration 4: Any[1, "x", 2, "x", 2, "x", 2]

Current symbol: y

Iteration 5: Any[1, "x", 2, "x", 2, "x", 2, "y", 4]

Current symbol: y

Iteration 6: Any[1, "x", 2, "x", 2, "x", 2, "y", 4, "T", 7]

Iteration 7: Any[1, "x", 2, "x", 2, "x", 2, "y", 4, "T", 7, "y", 8]

Current symbol: y

ERROR: Value of current table entry [8, "y"] is missing/blank.

julia> input_string = "zzzbbzbbz"; lrp.grammar_two(input_string)    # Segunda prueba de error
Iteration 1: Any[1]

Current symbol: z

Iteration 2: Any[1, "z", 2]

Current symbol: z

Iteration 3: Any[1, "z", 2, "z", 7]

Current symbol: z

Iteration 4: Any[1, "z", 2, "M", 4]
```

```
Iteration 5: Any[1, "z", 2, "M", 4, "z", 9]

Current symbol: b

Iteration 6: Any[1, "z", 2, "M", 4, "N", 6]

ERROR: Value of current table entry [6, "b"] is missing/blank.
```

	x	y	z	EOS	S	T
1	shift 2				10	
2	shift 2	shift 4			3	
3			shift 5			
4		T $\rightarrow\lambda$				7
5			S $\rightarrow$xSz	S $\rightarrow$xSz		
6		T $\rightarrow\lambda$				
7		shift 8				
8			shift 9			
9			S $\rightarrow$xyTyz	S $\rightarrow$xyTyz		
10				accept		

Cuadro 1: Tabla del parser LR(1) basado en la gramática (1).

	a	b	z	EOS	S	N	N
1			shift 2		14		
2	shift 3		shift 7			4	
3	shift 3		shift 7			8	
4		shift 5	shift 9				6
5		shift 5	shift 9				10
6			shift 11				
7	M $\rightarrow$z	M $\rightarrow$z	M $\rightarrow$z				
8	shift 12						
9		N $\rightarrow$z	N $\rightarrow$z				
10		shift 13					
11				S $\rightarrow$zMNz			
12	M $\rightarrow$aMa	M $\rightarrow$aMa	M $\rightarrow$aMa				
13		N $\rightarrow$bNb	N $\rightarrow$bNb				
14				accept			

Cuadro 2: Tabla del parser LR(1) basado en la gramática (2).

Aquí termina el trabajo de la implementación del parser LR(1), el leer y entender el trabajo hecho por Brookshear y Sipser [2, 3] lo suficiente como para implementar un parser de este tipo me hace preguntarme que cosas tan interesantes estan por venir y que más hay por aprender acerca de la teoría de la computación ¿Qué más hay por aprender sobre autómatas y lenguajes formales? Habrá que averiguarlo.

Apéndice A Estados de la primera gramática

Estados del autómata finito mostrado en la figura 1 generado a partir de la gramática (1). El símbolo ▽ funge como el marcador de lectura.

$$\begin{aligned} S' &\to \nabla S \\ S &\to \nabla xSz \\ S &\to \nabla xyTyz \end{aligned} \tag{1.1}$$

$$\begin{aligned} S &\to x\nabla Sz \\ S &\to x\nabla yTyz \\ S &\to \nabla xSz \\ S &\to \nabla xyTyz \end{aligned} \tag{1.2}$$

$$S \to xS\nabla z \tag{1.3}$$

$$\begin{aligned} S &\to xy\nabla Tyz \\ T &\to \nabla\lambda \end{aligned} \tag{1.4}$$

$$S \to xSz\nabla \tag{1.5}$$

$$T \to \lambda\nabla \tag{1.6}$$

$$S \to xyT\nabla yz \tag{1.7}$$

$$S \to xyTy\nabla z \tag{1.8}$$

$$S \to xyTyz\nabla \tag{1.9}$$

$$S' \to S\nabla \tag{1.10}$$

Apéndice B Estados de la segunda gramática

Estados del autómata finito mostrado en la figura 2 generado a partir de la gramática (2). El símbolo ▽ funge como el marcador de lectura.

$$S' \rightarrow \nabla S \qquad (2.1)$$
$$S \rightarrow \nabla zMNz$$

$$S \rightarrow z\nabla MNz \qquad (2.2)$$
$$M \rightarrow \nabla aMa$$
$$M \rightarrow \nabla z$$

$$M \rightarrow a\nabla Ma \qquad (2.3)$$
$$M \rightarrow \nabla aMa$$
$$M \rightarrow \nabla z$$

$$S \rightarrow zM\nabla Nz \qquad (2.4)$$
$$N \rightarrow \nabla bNb$$
$$N \rightarrow \nabla z$$

$$N \rightarrow b\nabla Nb \qquad (2.5)$$
$$N \rightarrow \nabla bNb$$
$$N \rightarrow \nabla z$$

$$S \rightarrow zMN\nabla z \qquad (2.6)$$

$$M \rightarrow z\nabla \qquad (2.7)$$

$$M \rightarrow aM\nabla a \qquad (2.8)$$

$$N \rightarrow z\nabla \qquad (2.9)$$

$$N \rightarrow bN\nabla b \qquad (2.10)$$

$$S \rightarrow zMNz\nabla \qquad (2.11)$$

$$M \rightarrow aMa\nabla \qquad (2.12)$$

$$N \rightarrow bNb\nabla \qquad (2.13)$$

$$S' \rightarrow S\nabla \qquad (2.14)$$

Apéndice C Implementación del algoritmo

```
module LR1Parser

"""
    TableEntry

Stores each entry of the parse table. Each entry corresponds to a usr defined value and its
respective type which can be "shift", "redux" or "notype".

The first type corresponds to shift operations, the second type to reductions by
rewrite rules, and the third type is for entries that do not correspond to the first
two aforementioned types.
"""
struct TableEntry
    evalue    # Entry value
    etype    # Entry type
end

"""
    symbols(list_of_symbols)

From an ordered list of symbols create its respective dictionary to store their respective column indexes.
"""
symbols(list_of_symbols) = map(x -> [x[2], x[1]], enumerate(list_of_symbols)) ▷ Dict

"""
    table(number_of_nodes, number_of_symbols)

Creates an empty parse table to be filled by the usr.
"""
table(number_of_nodes, number_of_symbols) = Array{Any}(missing, number_of_nodes, number_of_symbols)

function parse_string(input_string, symbols, table)
    table = replace(x -> ismissing(x) ? TableEntry(missing, "notype") : x, table)

    string_ = split(input_string, "")

    # Early termination if a character in the input string is not part of the
    # alphabet of the grammar
    if !all(x -> x in keys(symbols), string_)
        error("One or more characters in the input string \"$(input_string)\" do not belong to the symbols
        ↪  $(keys(symbols)) of the grammar.")
    end

    append!(string_, ["eos"])
    stack = []

    token = 1
    pushfirst!(stack, token)
    symbol = popfirst!(string_)    # read first symbol and pop it out from string
```

```
    table_entry = table[token, symbols[symbol]]

    i = 2
    println("Iteration $(i-1): $(reverse(stack)) \n")
    println("Current symbol: $(symbol) \n")

    while !isequal(table_entry.evalue, "accept")
        if isequal(table_entry.evalue, missing)
            error("Value of current table entry [$(stack[begin]), \"$(symbol)\"] is missing/blank.")
        end
        if isequal(table_entry.etype, "shift")
            pushfirst!(stack, symbol)
            token = table_entry.evalue
            pushfirst!(stack, token)
            symbol = popfirst!(string_)

            println("Iteration $(i): $(reverse(stack)) \n")
            println("Current symbol: $(symbol) \n")
        end
        if isequal(table_entry.etype, "redux")
            rewrite_rule = table_entry.evalue
            left_side = rewrite_rule[1]    # The symbol to be substituted back into the stack
            right_side = rewrite_rule[2] |> x -> split(x, "")    # The symbols to be backtracked

            for rs_symbol in reverse(right_side)
                # This condition assumes that the right side of the rewrite rule is of
                # the form A -> λ
                if !isequal(rs_symbol, "λ")
                    symbol_to_pop = stack[begin+1]    # First comes the node's number, and second is the
                    ↪  symbol to pop

                    if !isequal(rs_symbol, symbol_to_pop)
                        error("A symbol from the current rewrite rule $(right_side) does not match the symbol
                        ↪  \"$(symbol_to_pop)\" at the top of the stack.")
                    end

                    popfirst!(stack)    # To pop the table entry
                    popfirst!(stack)    # To pop the symbol
                end
            end

            token = stack[begin]    # Update token to the symbol at the top of the stack
            pushfirst!(stack, left_side)
            token = table[token, symbols[left_side]].evalue
            pushfirst!(stack, token)

            println("Iteration $(i): $(reverse(stack)) \n")
        end

        i += 1
        table_entry = table[token, symbols[symbol]]
    end

    if !isequal(symbol, "eos")
```

```
        return "Error, last symbol is \"$(symbol)\" but should be \"eos\"."
    end

    empty!(stack)
    println("The string \"$(input_string)\" was succesfully parsed.")
end

function grammar_two(input_string)
    # Test the grammar of the example seen in class:
    # S -> zMNz
    # M -> aMa
    # M -> z
    # N -> bNb
    # N -> z

    number_of_nodes = 14
    list_of_symbols = ["a", "b", "z", "eos", "S", "M", "N"]
    symbols_ = symbols(list_of_symbols)
    table_ = table(number_of_nodes, length(list_of_symbols))

    map_transitions!(symbol, symbol_transitions) = map(x -> table_[x[1], symbols_[symbol]] = TableEntry(x[2],
    ↪  x[3]) , symbol_transitions)

    a_transitions = [[2, 3, "shift"],
                     [3, 3, "shift"],
                     [7, ["M", "z"], "redux"],
                     [8, 12, "shift"],
                     [12, ["M", "aMa"], "redux"]]

    b_transitions = [[4, 5, "shift"],
                     [5, 5, "shift"],
                     [7, ["M", "z"], "redux"],
                     [9, ["N", "z"], "redux"],
                     [10, 13, "shift"],
                     [12, ["M", "aMa"], "redux"],
                     [13, ["N", "bNb"], "redux"]]

    z_transitions = [[1, 2, "shift"],
                     [2, 7, "shift"],
                     [3, 7, "shift"],
                     [4, 9, "shift"],
                     [5, 9, "shift"],
                     [6, 11, "shift"],
                     [7, ["M", "z"], "redux"],
                     [9, ["N", "z"], "redux"],
                     [12, ["M", "aMa"], "redux"],
                     [13, ["N", "bNb"], "redux"]]

    eos_transitions = [[11, ["S", "zMNz"], "redux"],
                       [14, "accept", "notype"]]

    S_transitions = [[1, 14, "notype"]]

    M_transitions = [[2, 4, "notype"], [3, 8, "notype"]]
```

```
154
155     N_transitions = [[4, 6, "notype"], [5, 10, "notype"]]
156
157     map_transitions!("a", a_transitions)
158     map_transitions!("b", b_transitions)
159     map_transitions!("z", z_transitions)
160     map_transitions!("eos", eos_transitions)
161     map_transitions!("S", S_transitions)
162     map_transitions!("M", M_transitions)
163     map_transitions!("N", N_transitions)
164
165     parse_string(input_string, symbols_, table_)
166 end
167
168 function grammar_one(input_string)
169     # Test the grammar corresponding to the project:
170     # S -> xSz
171     # S -> xyTyz
172     # T -> λ
173
174     number_of_nodes = 10
175     list_of_symbols = ["x", "y", "z", "eos", "S", "T"]
176     symbols_ = symbols(list_of_symbols)
177     table_ = table(number_of_nodes, length(list_of_symbols))
178
179     map_transitions!(symbol, symbol_transitions) = map(x -> table_[x[1], symbols_[symbol]] = TableEntry(x[2],
    ↪  x[3]) , symbol_transitions)
180
181     x_transitions = [[1, 2, "shift"],
182                      [2, 2, "shift"]]
183
184     y_transitions = [[2, 4, "shift"],
185                      [4, ["T", "λ"], "redux"],
186                      [6, ["T", "λ"], "redux"],
187                      [7, 8, "shift"]]
188
189     z_transitions = [[3, 5, "shift"],
190                      [5, ["S", "xSz"], "redux"],
191                      [8, 9, "shift"],
192                      [9, ["S", "xyTyz"], "redux"]]
193
194     eos_transitions = [[5, ["S", "xSz"], "redux"],
195                        [9, ["S", "xyTyz"], "redux"],
196                        [10, "accept", "notype"]]
197
198     S_transitions = [[1, 10, "notype"],
199                      [2, 3, "notype"]]
200
201     T_transitions = [[4, 7, "notype"]]
202
203     map_transitions!("x", x_transitions)
204     map_transitions!("y", y_transitions)
205     map_transitions!("z", z_transitions)
206     map_transitions!("eos", eos_transitions)
```

```
    map_transitions!("S", S_transitions)
    map_transitions!("T", T_transitions)

    parse_string(input_string, symbols_, table_)
end

end # module LR1Parser
```

Referencias

[1] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.

[2] J Glenn Brookshear. *Theory of computation: formal languages, automata, and complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[3] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2021.