Universidad de San Carlos de Guatemala

Facultad de ingeniería

Ingeniería en Ciencias y Sistemas

Gramática JPR

Elian Saúl Estrada Urbina

201806838

Especificación de la Gramática

G = {NT, T, S, P}

Donde:

1. G: Representa la gramática
2. NT: Representa los símbolos no terminales
3. T: Representa los símbolos terminales
4. S: Representa el símbolo inicial de la gramática
5. P: Representa las reglas de producción.

NT:

Símbolos no terminales, son aquellos símbolos que pueden ser reemplazados, ya que estos no afectan directamente al reconocimiento de cadenas, debido a que las reglas de producción serían las mismas solo cambiando el nombre. A continuación, se listan todos los símbolos no terminales utilizados en este proyecto:

- init
- instructions
- instruction

- statement
- statementP
- assignmet
- statement_array
- type
- list_brackets
- brackets
- list_expression
- expression_bra
- values_array
- list_values_array
- values
- assignment_array
- print
- inc_dec
- conditional
- con_if
- con_switch
- list_case
- case
- default
- loops
- loop_while
- loop_for

- for_init
- for_advance
- transfer
- functions
- function_main
- list_params
- params
- call_function
- list_params_call
- params_call
- ptcommaP
- empty
- error
- expression

T:

Símbolos terminales, estas son las literales que pueden aparecer en los resultados de nuestras reglas de producción, y estos no pueden ser cambiados usando las reglas de producción. A continuación se listarán los símbolos terminales asociados con su expresión regular

```
#Token        #Expresion_Regular
res_new             = 'new'
res_if              = 'if'
res_else            = 'else'
res_switch          = 'switch'
res_case            = 'case'
res_print           = 'print'
res_break           = 'break'
res_default         = 'default'
res_while           = 'while'
res_for             = 'for'
res_continue        = 'continue'
res_return          = 'return'
res_main            = 'main'
res_true            = 'true'
res_false           = 'false'
res_var             = 'var'
res_null            = 'null'
res_int             = 'int'
res_double          = 'double'
res_char            = 'char'
res_string          = 'string'
res_boolean         = 'boolean'
res_func            = 'func'
res_read            = 'read'


tk_key_o            = '{'
tk_key_c            = '}'
tk_par_o            = '('
tk_par_c            = ')'
tk_brackets_o       = '['
tk_brackets_c       = ']'
tk_dotcomma         = ';'
tk_comma            = ','
tk_twodot           = ':'
tk_inc              = '\+\+'
tk_dec              = '--'
tk_pow              = '\*\*'
tk_add              = '\+'
tk_sub              = '-'
tk_mult             = '\*'
tk_div              = '/'
tk_module           = '\%'
tk_equals           = '=='
tk_different        = '=!'
tk_assig            = '='
```

```
tk_greater_equals   = '>='
tk_greater          = '>'
tk_less_equals      = '<='
tk_less             = '<'
tk_or               = '\|\|'
tk_and              = '&&'
tk_not              = '!'


tk_decimal          = '\d+\.\d+'
tk_int              = '\d+'
tk_id               = '[a-zA-Z][a-zA-Z_0-9]*'
tk_string           = '\"(\\\'|\\"|\\\\|\\n|\\t|[^\'\\\"])*?\"'
tk_char             = '\'(\\\'|\\"|\\t|\\n|\\\\|[^\'\\\"])?\''
```

Expresiones regulares extras

```
multipleline_comment    = '\#\*(.|\n)*?\*\#'
simple_comment          = '\#.*\n'
ignore                  = ' \t'
```

S:

Símbolo de Inicio, es el símbolo no terminal que actúa como principal, es decir, le dará comienzo a todo el seguimiento de la gramática.

```
S = {init}
```

Precedencia

En este caso debido a que estamos trabajando con una gramática ascendente (LALR) para ser precisos, la recursividad a la izquierda y la factorización no son ningún inconveniente, sin embargo, la ambigüedad si lo sigue siendo, y debido a que trabajamos con una gramática ambigua para la expression es necesario indicar precedencia:

```
precedence = (
    ('left', 'tk_or'),
    ('left', 'tk_and'),
    ('right', 'tk_unot'),
    ('left', 'tk_equals', 'tk_different', 'tk_greater_equals', 'tk_greater',
'tk_less_equals', 'tk_less_equals'),
    ('left', 'tk_add', 'tk_sub'),
    ('left', 'tk_mult', 'tk_div', 'tk_module'),
    ('left', 'tk_pow'),
    ('right', 'tk_uminus'),
    ('right', 'tk_fcast'),
    ('left', 'tk_inc', 'tk_dec'),
)
```

P:

Reglas de Producción, son las especificaciones por las que una gramática se define, esta indica que símbolos pueden reemplazar a que otros símbolos, con ellas podemos analizar cadenas y también podemos generarlas. Estas reglas tienen una "cabeza" ó lado izquierdo que es nuestro símbolo no terminal, y un "cuerpo" ó lado derecho que consiste en lo que que reemplazara a la cabeza. A continuación se listarán y explicaran las reglas gramaticales utilizadas para este proyecto: **Nota: La explicación de las clases mencionadas se explicara mas a detalle en el manual técnico.**

- **init:**

  Nuestra producción inicial, deriva en el símbolo no terminal instructions, y su reducción es lo que retorna instructions. De esta forma lo que se devolverá en init será el árbol construido.

  ```
  init : instructions
  ```

- **instructions:**

  En esta producción, podemos observar que lo que se realiza es una lista de instruction debido a la recursividad que presenta, su reducción es una lista de instruction.

  ```
  instructions : instructions instruction
               | instruction
  ```

- **instruction:**

La producción instruction tiene varias alternativas, dependiendo de los símbolos terminales que vayan apareciendo el analizador tomará la decisión de ingresar a cada una de las opciones. La reducción sera el primer no terminal a la derecha.

```
instruction : statement ptcommaP
            | assignment ptcommaP
            | statement_array ptcommaP
            | assignment_array ptcommaP
            | print ptcommaP
            | inc_dec ptcommaP
            | conditional
            | loops
            | transfer ptcommaP
            | functions
            | call_functions ptcommaP
            | error tk_dotcomma
```

- **statement:**

  Esta producción ya comienza con símbolos terminales, en este caso si el 3er símbolo a la derecha es vacío la reducción sería una instancia de la clase Declaración en el cual el valor es null, si en dado caso no lo es la instancia de la misma clase tendría el valor que se reduce en statementP.

  ```
  statement : res_var tk_id statementP
  ```

- **statementP:**

  Esta producción tiene 2 opciones, que venga un terminal '=' seguido de una expresión, o que no venga nada (producción empty explicada mas abajo), si el primer termino a la derecha es nulo se devuelve empty, de lo contrario se devuelve expression.

  ```
  statementP : tk_assig expression
             | empty
  ```

- **statement_array:**

  En cualquiera de las 3 opciones, la reducción es una instacia de la clase Array, esta producción nos indica que puede venir 3 tipos de declaración de arreglos.

```
statement_array : type list_brackets tk_id tk_assig res_new type
list_expression
                | type list_brackets tk_id tk_assig values_array
                | type list_brackets tk_id tk_assig tk_id
```

- **list_backets:**

  La reducción de la list_brackets es una lista de los terminales '[' ']'

```
list_brackets : list_brackets brackets
              | brackets
```

- **brackets:**

  Producción que devuelve '[]'

```
brackets : tk_brackets_o tk_brackets_c
```

- **list_expression:**

  producción que reduce a una lista de expressiones entre los terminales '[' ']'

```
list_expression : list_expression expression_bra
                | expression_bra
```

- **expression_bra:**

  Producción que se reduce a una expression

```
expression_bra : tk_brackets_o expression tk_brackets_c
```

- **values_array:**

  Producción que se reduce en un formato '{' list_values_array '}'

```
values_array : tk_key_o list_values_array tk_key_c
```

- **list_values_array:**

lista de values separados por el terminal ','

```
list_values_array : list_values_array tk_comma values
                  | values
```

- **values:**

  Reduce a una producción de values_array (siendo así recursiva) o a expression

```
values : values_array
       | expression
```

- **assignment:**

  Producción reducida a una instancia de la clase Assignment

```
assignment : tk_id tk_assig expression
```

- **assignment_array:**

  Produccion reducida a una instancia de la clase Access_Array

```
assignment_array : tk_id list_expression tk_assig expression
```

- **functions:**

  Producción que es reducida en una instancia de la clase Function o clase Main con una lista de instructions a ejecutar, si es de tipo Function puede ser que tenga parametros o no.

```
functions : function_main
          | res_func tk_id tk_par_o tk_par_c tk_key_o instructions tk_key_c
          | res_func tk_id tk_par_o list_params tk_par_c tk_key_o
instructions tk_key_c
```

- **function_main:**

  Producción que reduce en una instacia de clase Main, teniendo una lista de instrucciones a ejecutar.

```
function_main : res_main tk_par_o tk_par_c tk_key_o instructions tk_key_c
```

- **list_params:**

  producción reducida a una lista de params separados por el terminal ','

```
list_params : list_params tk_comma params
            | params
```

- **params:**

  en este caso la producción no devuelve una instancia de clase, si no un diccionario propio de python con las siguientes claves:

  tipo, nombre; si es un arreglo serían tipo: nombre, longitud y sub_tipo

```
params : type tk_id
       | type list_brackets tk_id
```

- **call_function:**

  Producción reducida en una instancia de la clase Call, en donde puede tener o no tener parametros.

```
call_function : tk_id tk_par_o tk_par_c
              | tk_id tk_par_o list_params_call tk_par_c
```

- **list_params_call:**

  Producción reducida a una lista de params_call separados por el terminal ','

```
list_params_call : list_params_call tk_comma params_call
                 | params_call
```

- **params_call:**

  producción reducida a una expression

```
params_call : expression
```

- **print:**

  Producción que reduce en una instancia de la clase Print

  ```
  print : res_print tk_par_o expression tk_par_c
  ```

- **inc_dec:**

  dependiendo del terminal 2do a la derecha se devolvera una instancia de la clase Int_Dec con el operador de tipo dec o inc.

  ```
  inc_dec : tk_id tk_inc
          | tk_id tk_dec
  ```

- **conditional:**

  producción que es reducida a una instancia de la clase If o Switch

  ```
  conditional : con_if
              | con_switch
  ```

- **con_if:**

  producción reducida a una instancia de la clase If, en donde se tienen 3 opciones, la primera que solo venga el if y su lista de instrucciones, la 2da que venga el if, su lista y un else con su lista de instrucciones, o la 3ra que venga el if y una lista de else-if con su lista de instrucciones etc.

  ```
  con_if : res_if tk_par_o expression tk_par_c tk_key_o instructions tk_key_c
         | res_if tk_par_o expression tk_par_c tk_key_o instructions tk_key_c
  res_else tk_key_o instructions tk_key_c
         | res_if tk_par_o expression tk_par_c tk_key_o instructions tk_key_c
  res_else con_if
  ```

- **con_switch:**

  Producción reducida a una instancia de la clase switch, se tienen 3 opciones, la 1 ra que solo venga el default y sus instrucciones, la 2da que solo venga la lista de cases y sus instrucciones y al 3ra que vengan las 2 anteriormente mencionadas.

```
con_switch : res_switch tk_par_o expression tk_par_c tk_key_o default
tk_key_c
           | res_switch tk_par_o expression tk_par_c tk_key_o list_case
tk_key_c
           | res_switch tk_par_o expression tk_par_c tk_key_o list_case
default tk_keyt_c
```

- **list_case:**

  produccion reducida a una lista de case

```
list_case : list_case case
          | case
```

- **case:**

  Producción que se reduce a una instancia de la clase Case donde tendremos una expression y una lista de instructions

```
case : res_case expression tk_twodot instructions
```

- **default:**

  producción reducida a una lista de instruction

```
default : res_default tk_twodot instructions
```

- **loops:**

  Producción reducida a una instancia de clase For o While

```
loops : loop_while
      | loop_for
```

- **loop_while:**

  Producción reducida a una instancia de clase While con una expression y una lista de instruction

```
loop_while : res_while tk_par_o expression tk_par_c tk_key_o instructions
tk_key_c
```

- **loop_for:**

  Producción reducida a una instancia de clase For, aqui esta instancia recibira una instancia de Declaración o asignación en el for_init o una asignacion o incrementeo_decremento en for_advance.

```
loop_for : res_for tk_par_o for_init tk_dotcomma expression tk_dotcomma
for_advance tk_par_c tk_key_o instructions tk_key_c
```

- **for_init:**

  Producción reducida a una instancia de Declaración o Asignación

```
for_init : statement
         | assigment
```

- **for_advance:**

  producción reducida a una instancia de la clase Asignación o Inc_dec

```
for_advance : assignment
            | inc_dec
```

- **transfer:**

  Producción que reduce en una instancia de clase Break, Continue o Return donde tanto break como continue no reciben nada, en cambio Return espera una expression

```
transfer : res_break
         | res_continue
         | res_return expression
```

- **type:**

  Producción que reduce en un typo de dato, puede ser INTEGGER, FLOAT, CHAR, STRING o BOOLEAN

```
type : res_int
     | res_char
     | res_string
     | res_double
     | res_boolean
```

- **ptcommaP:**

  producción que puede reconocer si viene un terminal ';' o vacío

```
ptcommaP : tk_dotcomma
         | empty
```

- **empty:**

  producción que reduce en nada.

```
empty :
```

- **expression:**

  Esta expression se separa en varias: Agrupada, Binaria, Unaria, Función y Primitivas. Esta es la Agrupada devolvera una agrupacion de expressiones

```
expression : tk_par_o expression tk_par_c
```

- **expression:**

  Expresión Binaria: Producción que dependiendo del terminal 2do del lado derecho puede reducir en la clase Aritmetica, Relacional o Lógica, dichos terminales son: Aritmetica (+ , -, **, *, /, %), Relacional (<, <=, >=, >, == , =!) y Lógica (||, &&)

```
expression : expression tk_add expression
           | expression tk_sub expression
           | expression tk_mult expression
           | expression tk_div expression
           | expression tk_module expression
           | expression tk_pow expression
           | expression tk_equals expression
           | expression tk_differnt expression
           | expression tk_greater expression
           | expression tk_greater_equals expression
           | expression tk_less expression
           | expression tk_less_equals expression
           | expression tk_and expression
           | expression tk_or expression
```

- **expression:**

  Expresión Unaria: Producción que puede reducir en intancias de clase como Aritmetica, Logica o Casteo, dependiendo del terminal, opciones para Aritmetica (-, ++, --), Lógica (!) y Casteo ((tipo)), aclaración, el termino %prec así como los tokens tk_uminos, tk_unot y tk_fcast son especiales para este tipo de precedencia y le dicen al interprete que maneje los tokens que si existen como un nuevo token solo para esta ocación y poder preceder de mejor manera.

  ```
  expression : tk_sub expression %prec tk_uminus
             | tk_not expression %prec tk_unot
             | tk_par_o type tk_par_c expression %prec tk_fcast
             | expression tk_inc
             | expression tk_dec
  ```

- **expression:**

  Expresión de Función: Producción que es reducida por la clase Call o la clase Read, dependiendo del primer terminal que aparezca.

  ```
  expression : call_function
             | res_read tk_par_o tk_par_c
  ```

- **expression:**

  Expresión Primitivos: Producción que puede reducir en instancias de Primitivo, Access_Array y Identificador, siendo estas de la siguiente forma, Primitivo (tk_int,

tk_decimal, tk_string, tk_char, res_true, res_false, res_null), Identificador (tk_id) y Access_Array (tk_id list_expression)

```
expression : tk_int
           | tk_decimal
           | tk_string
           | tk_char
           | res_true
           | res_false
           | res_null
           | tk_id
           | tk_id list_expression
```

Importación de PLY y puesta en marcha

Importacion de lex

```
import ply.lex as lex
lexer = lex.lex()
```

Importacoin de yacc

```
import ply.yacc as yacc
parser = yacc.yacc()
```

Puesta en marcha

Para esta ejecución se hizo un metodo que recibiría la entrada en string para posteriormente pasarlo al parser:

```
def parser(str_input):

    return parser.parse(str_input)apa
```