Adriana Abella Kuri, A01329591

Proyecto de compiladores –
Documentación

Repositorio con código completo: https://github.com/adriabellak/Compilador/tree/main

Gramática

Inicialmente, estas fueron las reglas gramaticales que definí a partir de los diagramas proporcionados para la actividad. Están definidas en pseudocódigo solo para darme mayor claridad en cuanto a la estructura que sigue el lenguaje.

```
PROGRAMA
<PROGRAMA> --> program <ID>; <VARS>? <FUNCS>* main <BODY> end

VARS
<VARS> --> var (<ID>(, <ID>)*: <TYPE> ;)+

TYPE
<TYPE> --> int|float

BODY
<BODY> --> {<STATEMENT>+}

STATEMENT
<STATEMENT> --> [<ASSIGN><CONDITION><CYCLE><F_CALL><PRINT>]

PRINT
<PRINT> --> print '(' [<EXPRESION><cte.string>] (, [<EXPRESION><cte.string>])*
')';

ASSIGN
<ASSIGN> --> <ID> = <EXPRESION> ;

CYCLE
<CYCLE> --> while <BODY> do ( <EXPRESION> ) ;

CONDITION
<CONDITION> --> if ( <EXPRESION> ) <BODY> (else <BODY>)? ;

EXPRESION
<EXPRESION> --> <EXP> ([><!=] <EXP>)?

EXP
<EXP> --> <TERMINO> ([+-] <TERMINO>)*

TERMINO
<TERMINO> --> <FACTOR> ([*/] <FACTOR>)*

FACTOR
<FACTOR> --> [ ( <EXPRESION> )  |  [+-]? [<ID><CTE> ] ]

FUNCS
```

```
<FUNCS> --> void <ID> "(" ((<ID> : <TYPE>) (, <ID> : <TYPE>)*)? ")" "[" <VARS>?
<BODY> "]" ;

F_CALL
<F_CALL> --> <ID> "(" (<EXPRESION> (, <EXPRESION>)*)? ")" ;
```

Tokens

Definí los tokens de la misma manera.

```
ID          : ([A-Za-z_]+[0-9]*)+ ;
CTE_INT     : [0-9]+ ;
CTE_FLOAT   : [0-9]+('.'[0-9]+)? ;
CTE_STRING  : '"' .*? '"';
```

Gramática definida en ANTLR:

Una vez que tuve clara la gramática, la definí de la siguiente manera utilizando la notación de ANTLR.

```
grammar baby_duck;
//-----------------gramatica
programa    : 'program' ID ';' vars? funcs* 'main' body 'end' ;
vars        : 'var' (ID (',' ID)* ':' type ';')+ ;
type        : 'int' | 'float' ;
body        : '{' statement* '}' ;
statement   : assign
            | condition
            | cycle
            | f_call
            | print
            ;
print       : 'print' '(' (expresion | CTE_STRING) (',' (expresion |
CTE_STRING))* ')' ';' ;
assign      : ID '=' expresion ';' ;
cycle       : 'while' body 'do' '(' expresion ')' ';' ;
condition   : 'if' '(' expresion ')' body ('else' body)? ';' ;
expresion   : exp ( ('<' | '>' | '!=') exp)? ;
exp         : termino (('+'|'-') termino)* ;
termino     : factor (('*'|'/') factor)* ;
factor      : '(' expresion ')'
            | ('+' | '-')? (ID | cte)
            ;
funcs       : 'void' ID '(' ( (ID ':' type) (',' ID ':' type)* )? ')' '[' vars?
body ']' ';' ;
f_call      : ID '(' (expresion (',' expresion)*)? ')' ';' ;
cte         : CTE_INT | CTE_FLOAT ;
// --------------- tokens
ID          : ([A-Za-z_]+[0-9]*)+ ;
CTE_INT     : [0-9]+ ;
```

```
CTE_FLOAT   : [0-9]+('.'[0-9]+)? ;
CTE_STRING  : '"' .*? '"';
WHITESPACE  : [ \t\n\r]+ -> skip;
```

Cabe recalcar que, aunque sigue teniendo esencialmente la misma estructura, realicé varios cambios a la gramática y los tokens para incorporar las siguientes funcionalidades de mi compilador. El código proporcionado arriba es únicamente el paso inicial. Así es como se ve la gramática en mi última entrega:

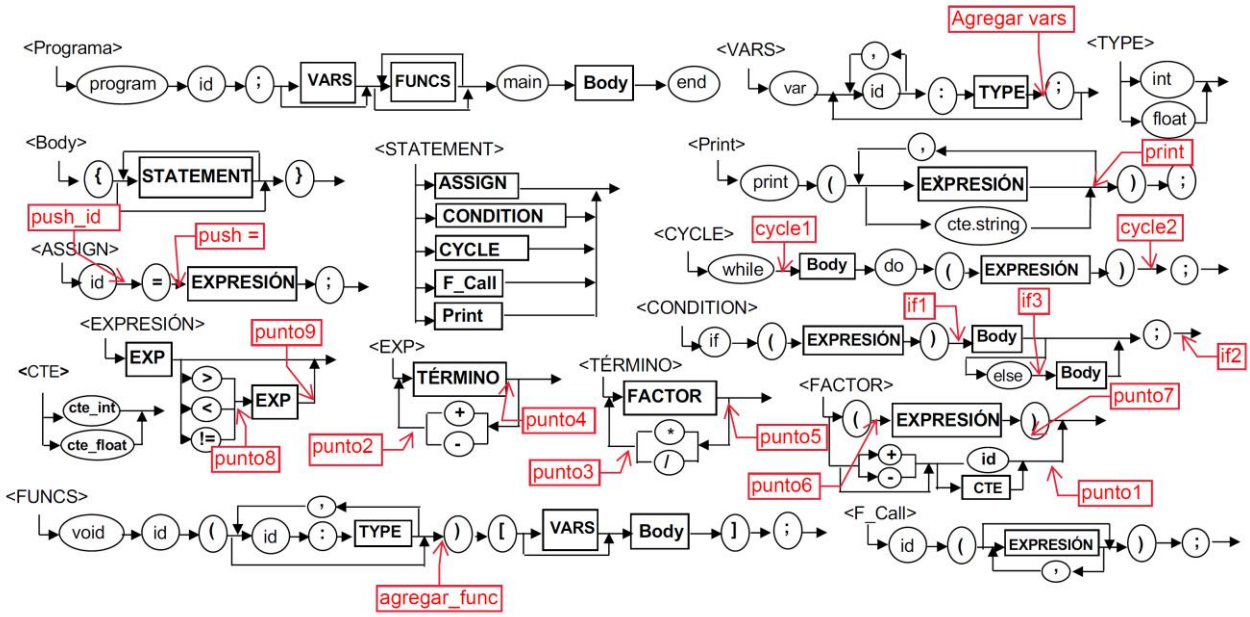
```
grammar baby_duck;

@parser::header {
from funciones import *
}
// antlr4 -Dlanguage=Python3 baby_duck.g4
programa    : 'program' ID ';' vars? funcs* 'main' body 'end' ;
vars        : 'var' (varlist ':' TYPE {agregar_vars($varlist.text, $TYPE.text)}
';')+ ;
varlist     : ID (',' ID)*;
body        : '{' statement* '}' ;
statement   : assign
            | condition
            | cycle
            | f_call
            | print
            ;
print       : 'print' '(' (expresion {print_expresion()} | CTE_STRING
{print_string($CTE_STRING.text)}) (',' (expresion {print_expresion()} | CTE_STRING
{print_string($CTE_STRING.text)}))* ')' ';' ;
assign      : ID {push_operando($ID.text, 'ID')} EQ {push_operador('=')} expresion
{pop_assign()}';' ;
cycle       : 'while' {cycle1()} body 'do' '(' expresion ')' {cycle2()} ';' ;
condition   : 'if' '(' expresion ')' {if1()} body ('else' {if3()} body)? ';'
{if2()};
expresion   : exp ( oper_rel {punto8($oper_rel.text)} exp {punto9()})? ;
oper_rel    : ('<' | '>' | '!=') ;
exp         : termino {punto4()} (OPER_EXP {punto2($OPER_EXP.text)} termino
{punto4()})* ;
termino     : factor {punto5()} (OPER_TERM {punto3($OPER_TERM.text)} factor
{punto5()})* ;
factor      : '(' {punto6()} expresion ')' {punto7()}
            | OPER_EXP? (ID {punto1($ID.text, 'ID', $OPER_EXP.text)} | cte
{punto1($cte.text, 'cte', $OPER_EXP.text)})
            ;
funcs       : 'void' ID params {nueva_func($ID.text, $params.text)} '[' vars? body
']' ';' ;
params      : '(' ( (ID ':' TYPE) (',' ID ':' TYPE)* )? ')' ;
f_call      : ID '(' (expresion (',' expresion)*)? ')' ';' ;
cte         : CTE_INT | CTE_FLOAT ;
// ---------------- tokens
EQ          : '=';
```

Adriana Abella Kuri, A01329591

```
OPER_EXP    : ('+'|'-');
OPER_TERM   : ('*'|'/');
TYPE        : ('float'|'int') ;
ID          : ([A-Za-z_]+[0-9]*)+ ;
CTE_INT     : [0-9]+ ;
CTE_FLOAT   : [0-9]+('.'[0-9]+)? ;
CTE_STRING  : '"' .*? '"';
WHITESPACE  : [ \t\n\r]+ -> skip;
```

Realicé mi proyecto en Python. Como se puede ver en el código anterior, definí diferentes funciones que utilicé como acciones para puntos neurálgicos dentro de mi gramática. Al final, estos puntos neurálgicos se vieron de la siguiente manera:



- Agregar_vars
  - Añadir nueva variable a tabla de variables
- Push_id
  - Push id a stack de operandos
- Push =
  - Push operador = a stack de operadores
- Print
  - Genera cuádruplo de print con el string o expresión anterior haciendo pop a stack de operandos
- Cycle1
  - Push salto
- Cycle 2
  - Pop tipos
  - Pop operandos
  - Pop saltos
  - Cuádruplo gotov, operando, , salto
- If1
  - pop tipos
  - pop operandos
  - push salto
  - cuádruplo gotof, operando, , ___

- If2
  - Pop saltos
  - Llenar cuádruplo pendiente
- If3
  - Cuádruplo goto, , , ___
  - Pop saltos
  - Push salto
  - Llenar cuádruplo pendiente
- Agregar_func
  - Añade función y sus parámetros a directorio de funciones
- Punto1
  - Push operando con signo
- Punto2
  - Push operador
- Punto3
  - Push operador
- Punto4
  - if top operador == ‘+’ o ‘-‘
    - pop operandos (x2)
    - pop tipos (x2)
    - pop operador
    - cuadruplo operador, op1, op2, temporal
    - push operando temporal
- punto5
  - if top operador == ‘*’ o ‘/‘
    - pop operandos (x2)
    - pop tipos (x2)
    - pop operador
    - cuadruplo operador, op1, op2, temporal
    - push operando temporal
- punto6
  - push fondo falso operadores
- punto7
  - pop fondo falso
- punto8
  - push operador
- punto9
  - if top operador == ‘*’ o ‘/‘
    - pop operandos (x2)
    - pop tipos (x2)
    - pop operador
    - cuadruplo operador, op1, op2, temporal
    - push operando temporal
- pop_assign:
  - if top operador == ‘=’
    - pop operandos (x2)
    - pop tipos (x2)
    - pop operador
    - cuadruplo operador, op1, , op2

Para el control de mis variables, definí una tabla de variables utilizando la estructura de datos de diccionario. Este es un ejemplo de como se ve dicha estructura:

```
  {'n1':
     {'tipo': 'int',
     'valor': 5,
     'scope': scope},
  'n2':
     {'tipo': 'int',
     'valor': 2,
     'scope': scope},
  'n3':
     {'tipo': 'int',
     'valor': 9,
     'scope': scope},
  'num':
     {'tipo': 'int',
     'valor': 10,
     'scope': scope},
  'cont':
     {'tipo': 'int',
     'valor': 0,
     'scope': scope}
}
```

Para llenar este diccionario utilicé puntos neurálgicos en la declaración de variables de mi gramática.

También utilicé un diccionario para mi directorio de funciones, el cual se ve de la siguiente manera:

```
  {
     'simplefunc': {
        'parametros': {
           'param1': 'float',
           'param2': 'int'}
        }
}
```

Para llenar el directorio de funciones también utilicé puntos neurálgicos con llamados de funciones directamente en la gramática.

Máquina Virtual

Una vez generados los cuádruplos, iteré con un ciclo a través de todos ellos procesando cada operación y haciendo los saltos que fueran necesarios. Utilicé mi tabla de variables (diccionario) para llevar el control de las variables y sus valores, y creé una función que ejecutaba cada operación.