Universidad La Salle

Compiladores

Informe del Trabajo Parcial

Algoritmo LL1

Karlo Emigdio Pacha Curimayhua

Sexto Semestre - Ingeniería de Software

2023

1 Introducción

En muchos de los lenguajes actuales el tipo de declaración del código es algo compleja y a veces innecesariamente "larga" como lo puede ser el '||' (or), en muchos lenguajes el símbolo '|' sólo se usa para la condición, sería más simple escribir sólo una vez el símbolo.

Este lenguaje busca simplificar la forma de escribir código, con el propósito de ahorrar algo de tiempo y para que el código sea más legible y ocupe menos espacio. Drachen Script se basa en lenguajes como: Python, Go, JavaScript y C#. El objetivo principal es la comodidad.

2 Especificación Léxica

2.1 Comentarios

En Drachen Script, los comentarios empiezan con el símbolo // y /**/ para comentarios multilinea. Estos deben estar dentro de una función o en el main

```
1         // esto es un comentario de una linea
2         /* comentario multilinea
3         mas lineas del comentario*/
4
```

2.2 Identificadores

En Drachen Script, los identificadores empiezan con una letra (mayúscula o minúscula), le puede seguir un número, más letras o el subguión |.

```
1         func MiFuncion (miParametro) {...}
2
3         var MiIdentificador = 10,
4         var mi_segundo_identificador = 20,
5
```

2.3 Palabras clave

En Drachen Script, las palabras clave serán para estructuras como: if, while y for. Algunas otras se usarán para llamar a funciones como main(), read() o print(). Otras palabras reservadas serán para las funciones: func, exec y return. Finalmente para la declaración de variables y los tipos de booleanos: var, null, true y false.

```
1     main() {
2         var a = true,
3         var b = exec MiFuncion(),
4         if a {
5             for i = 0, 10, 2 {
```

```
                print(i),
            }
        }
        return 0,
    }

```

2.4 Operadores

En Drachen Script, los operadores serán: + (suma), - (resta), * (multiplicación) y / (división). Los operadores lógicos serán: == (igual), > (mayor),< (menor), <= (menor o igual), >= (mayor o igual), y != (diferente).

```
    main() {
        var a = true,
        var b = 10,
        if a == false {
        }

        if b < 11 {
        }

        if a != false {
        }
    }

```

2.5 Literales

En Drachen Script, los literales comparten similitud con la mayoría de lenguajes de programación actuales.

```
        var entero = 123,
        var flotante = 1.23,
        var cadena = "abc",
        var cadena2 = 'abc',
        var booleano = true,
        var nulo = null,

```

3 Expresiones regulares

Token	Expresión Regular
Identifier	$[a-zA-Z]([a-zA-Z]\|\|[0-9]\|\|[\|])*$
IntNumber	$[0-9]+$
FloatNumber	$[0-9]*(.)[0-9]+$
String	$(\text{"}\|')([a-z]\|\|[A-Z]\|\|[0-9]\|(\backslash n)\|(\epsilon))*(\text{"}\|')$
Var	$'var'$
Plus	+
Minus	−
Times	*
Divide	/
LParen	(
RParen	)
RKey	{
LKey	}
Equal	=
Greater	>
Less	<
EqualEqual	==
GreaterEqual	>=
LessEqual	<=
Diferent	! =
While	$'while'$
For	$'for'$
If	$'if'$
Elif	$'elif'$
Else	$'else'$
Function	$'func'$
Execute	$'exec'$
Return	$'return'$
Main	$'main'$
Var	$'var'$
Print	$'print'$
Read	$'read'$
Null	$'null'$
Comment	//.*
MultiComment	\/*([^*]|[\r\n]|(*+([^*/]|[\r\n])))**+\/
Comma	,
Or	|
And	&

4 Gramática

La siguiente gramática cumple con las condiciones para ser una Gramática LL1, no es ambigua y está factorizada por la izquierda.

4.1 Gramática LL1

La siguiente gramática usa símbolos y el vacío es igual a: ''.

```
Program -> FunctionDeclaration' MainFunction

FunctionDeclaration' -> FunctionDeclaration
FunctionDeclaration' -> ''
FunctionDeclaration -> func FunctionName ( Parameters ) {
    Statements } FunctionDeclaration'

MainFunction -> main ( ) { Statements }

FunctionName -> id

Parameters -> id Parameters'
Parameters -> Parameters'
Parameters' -> , id Parameters'
Parameters' -> ''

Statements -> Statement Statements'
Statements' -> Statement Statements'
Statements' -> ''
Statement -> VariableDeclaration ,
Statement -> Assignment ,
Statement -> Conditional
Statement -> Loop
Statement -> FunctionCall ,
Statement -> ReturnStatement ,
Statement -> Comment
Statement -> Print ,

Print -> print ( SentParameters )
VariableDeclaration -> var id = Expressions
Assignment -> id = Expressions

Expressions -> Terms Expressions'
Expressions' -> ArithOperator Terms Expressions'
Expressions' -> ComparisonOperator Terms Expressions'
Expressions' -> ''

Terms -> Expression' Factor
Factor -> ArithOperator' Expression' Factor
Factor -> ''

Expression' -> ( Expressions )
Expression' -> Value
Expression' -> id
Expression' -> FunctionCall
Expression' -> read ( )

FunctionCall -> exec FunctionName ( SentParameters )

SentParameters -> Expressions SentParameters'
SentParameters ->  ''
SentParameters' -> , Expressions SentParameters'
SentParameters' -> ''

Conditional -> if Condition { Statements' } Conditional'
```

```
Conditional' -> ElseIf
Conditional' -> ''

ElseIf -> Else
ElseIf -> Elif Else

Elif -> elif Condition { Statements' } Elif'
Elif' -> Elif
Elif' -> ''
Else -> else { Statements' }

Loop -> While
Loop -> For
While -> while Condition { Statements' }

For -> for Expressions , Expressions For'
For' -> , Expressions { Statements' }
For' -> { Statements' }

Condition -> Expressions Condition'
Condition' -> LogicalOperator Condition
Condition' -> ''

ReturnStatement -> return Expressions
Comment -> comment
Comment -> multicomment

ArithOperator -> +
ArithOperator -> -
ArithOperator' -> *
ArithOperator' -> /

ComparisonOperator -> <
ComparisonOperator -> >
ComparisonOperator -> ==
ComparisonOperator -> <=
ComparisonOperator -> >=
ComparisonOperator -> !=

LogicalOperator -> &
LogicalOperator -> |

Value -> string
Value -> Number
Value -> Boolean
Value -> null

Number -> int
Number -> float
Boolean -> true
Boolean -> false
```

4.1.1 Ejemplo de Input LL1

```
func id ( id ) {
    comment
    while id + ( null ) {
```

```
        return exec id ( ( int ) ) ,
    }
}

func id ( ) {
    for id + ( int ) , id < int , true {
        var id = int ,
        print ( id , int , ( string ) ) ,
    }
}

func id ( ) {
    if int < id {
        var id = int ,
        var id = read ( ) ,
    }
}

func id ( ) {
    var id = ( exec id ( exec id ( int * id ) ) ) + id ,
}

main ( ) {
    var id = ( id != int ) + read ( ) + read ( ) ,
}

func id ( id ) { comment while id + ( null ) { return exec id ( (
    int ) ) , } } func id ( ) { for id + ( int ) , id < int , true
    { var id = int , print ( id , int , ( string ) ) , } } func id
    ( ) { if int < id { var id = int , var id = read ( ) , } } func
     id ( ) { var id = ( exec id ( exec id ( int * id ) ) ) + id ,
    } main ( ) { var id = ( id != int ) + read ( ) + read ( ) , }
```

4.2 Gramática LL1 - Parser

La siguiente gramática no usa símbolos, usa palabras, es lo más cercano a la tabla LL1 para parsear; el vacío es igual a: ''.

```
Program -> FunctionDeclaration' MainFunction

FunctionDeclaration' -> FunctionDeclaration
FunctionDeclaration' -> ''
FunctionDeclaration -> func FunctionName lparen Parameters rparen
    lkey Statements rkey FunctionDeclaration'

MainFunction -> main lparen rparen lkey Statements rkey

FunctionName -> id

Parameters -> id Parameters'
Parameters -> Parameters'
Parameters' -> comma id Parameters'
Parameters' -> ''

Statements -> Statement Statements'
Statements' -> Statement Statements'
Statements' -> ''
Statement -> VariableDeclaration comma
Statement -> Assignment comma
```

```
Statement -> Conditional
Statement -> Loop
Statement -> FunctionCall comma
Statement -> ReturnStatement comma
Statement -> Comment
Statement -> Print comma

Print -> print lparen SentParameters rparen
VariableDeclaration -> var id equal Expressions
Assignment -> id equal Expressions

Expressions -> Terms Expressions'
Expressions' -> ArithOperator Terms Expressions'
Expressions' -> ComparisonOperator Terms Expressions'
Expressions' -> ''

Terms -> Expression' Factor
Factor -> ArithOperator' Expression' Factor
Factor -> ''

Expression' -> lparen Expressions rparen
Expression' -> Value
Expression' -> id
Expression' -> FunctionCall
Expression' -> read lparen rparen

FunctionCall -> exec FunctionName lparen SentParameters rparen

SentParameters -> Expressions SentParameters'
SentParameters ->  ''
SentParameters' -> comma Expressions SentParameters'
SentParameters' -> ''

Conditional -> if Condition lkey Statements' rkey Conditional'
Conditional' -> ElseIf
Conditional' -> ''

ElseIf -> Else
ElseIf -> Elif Else

Elif -> elif Condition lkey Statements' rkey Elif'
Elif' -> Elif
Elif' -> ''
Else -> else lkey Statements' rkey

Loop -> While
Loop -> For
While -> while Condition lkey Statements' rkey

For -> for Expressions comma Expressions For'
For' -> comma Expressions lkey Statements' rkey
For' -> lkey Statements' rkey

Condition -> Expressions Condition'
Condition' -> LogicalOperator Condition
Condition' -> ''

ReturnStatement -> return Expressions
```

```
Comment -> comment
Comment -> multicomment

ArithOperator -> plus
ArithOperator -> minus
ArithOperator' -> times
ArithOperator' -> divide

ComparisonOperator -> less
ComparisonOperator -> greater
ComparisonOperator -> equalequal
ComparisonOperator -> lessequal
ComparisonOperator -> greaterequal
ComparisonOperator -> diferent

LogicalOperator -> and
LogicalOperator -> or

Value -> string
Value -> Number
Value -> Boolean
Value -> null

Number -> int
Number -> float
Boolean -> true
Boolean -> false
```

4.2.1 Ejemplo de Input LL1 - Parser

```
func id lparen id rparen lkey
    comment
    while id plus lparen null rparen lkey
        return exec id lparen lparen int rparen rparen comma
    rkey
rkey

func id lparen rparen lkey
    for id plus lparen int rparen comma id less int comma true lkey
        var id equal int comma
        print lparen id comma int comma lparen string rparen rparen
     comma
    rkey
rkey

func id lparen rparen lkey
    if int less id lkey
        var id equal int comma
        var id equal read lparen rparen comma
    rkey
rkey

func id lparen rparen lkey
    var id equal lparen exec id lparen exec id lparen int times id
    rparen rparen rparen plus id comma
rkey

```

```
main lparen rparen lkey
    var id equal lparen id diferent int rparen plus read lparen
    rparen plus read lparen rparen comma
rkey

func id lparen id rparen lkey comment while id plus lparen null
    rparen lkey return exec id lparen lparen int rparen rparen
    comma rkey rkey func id lparen rparen lkey for id plus lparen
    int rparen comma id less int comma true lkey var id equal int
    comma print lparen id comma int comma lparen string rparen
    rparen comma rkey rkey func id lparen rparen lkey if int less
    id lkey var id equal int comma var id equal read lparen rparen
    comma rkey rkey func id lparen rparen lkey var id equal lparen
    exec id lparen exec id lparen int times id rparen rparen rparen
     plus id comma rkey main lparen rparen lkey var id equal lparen
     id diferent int rparen plus read lparen rparen plus read
    lparen rparen comma rkey
```

5 Ejemplos de Código

5.1 Ejemplos Aceptados

Los siguientes ejemplos fueron aceptados por el Algoritmo LL1 implementado.

5.1.1 Example.ds

```
func Square (a) {
  return a*a,
}

main() {
    var a = 0,
    var b = 10.5,
    var c = null,
    var d = "abc",
    var e = 'abc',
    var f = false,

    if a < b & c == null {
        for 0, b, 1 {
            a = a + 1,
        }
    }
    elif a == b | f == false {
        for i , 2*b {
            a = a + 1,
        }
    }
    else {
        while a > b {
            a = a - 1,
        }
    }


    exec Square(a),
```

```
31
32 }
```

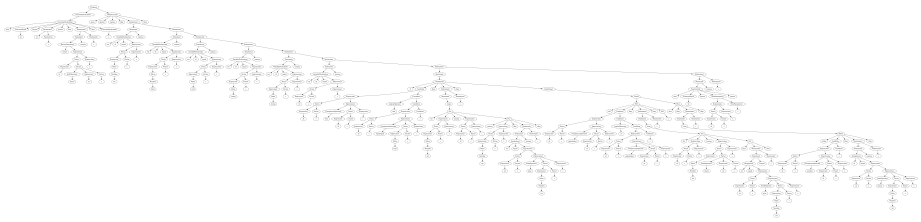

Figure 1: Árbol Sintáctico Generado

5.1.2 Example2.ds

```
main () {
  // esto es un comentario de una linea
  print(1),
  var a = read(),
  var MiIdentificador = 10,
  var mi_segundo_identificador = 20,
  var a = true,
  if a {
      for 0, 10 , 2 { }
  }

  if a == false {
  }

  if b < 11 {
      var entero = 123,
  }

  if a != false {
      var flotante = 1.23,
  }

  /* comentario multilinea
  mas lineas del comentario */

  var booleano = true,
  var nulo = null,

  return 0,
}
```

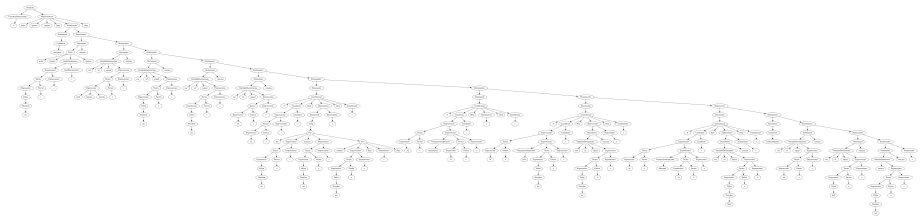

Figure 2: Árbol Sintáctico Generado

5.1.3 Example3.ds

```
func AddTwo (a) {
  return a + 2,
}

main () {
  var a = exec AddTwo(1),
  if a > 5 {
    print("True"),
  }
  else {
    print("False"),
  }
}
```

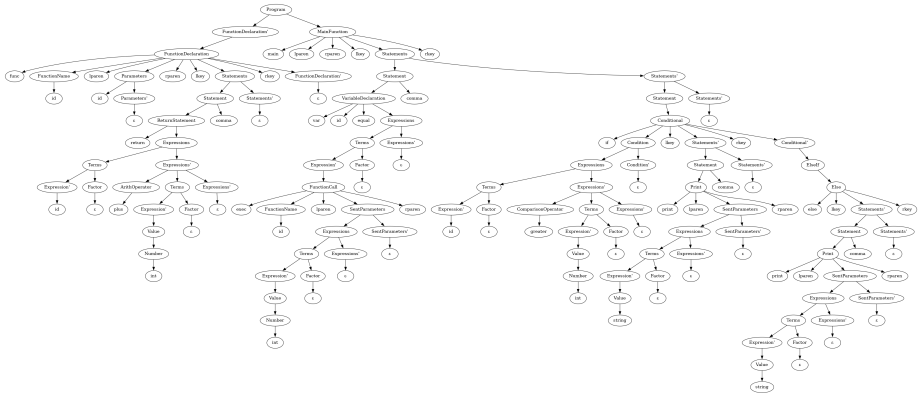

Figure 3: Árbol Sintáctico Generado

5.2 Ejemplos No Aceptados

Los siguientes ejemplos no fueron aceptados por el Algoritmo LL1 implementado.

5.2.1 BExample.ds

El primer error se encuentra en la línea 2, se espera que: "return a*a" termine con coma (,).

```
func Square (a) {
  return a*a
}

main() {
  var a = 0,
  var f = false,

  if a < b & c = null {
    for 0, b ,1 {
      a = a + 1,
    }
  }
  elif a = b | f = false {
    for i , 2*b {
      a = a + 1,
    }
  }
  else {
    while a > b {
      a = a - 1,
    }
  }

  exec Square(a),
}
```

5.2.2 BExample2.ds

El primer error se encuentra en la línea 1, no se espera que haya un comentario fuera de una función o el main.

```
// esto es un comentario de una linea

main () {
  print(1),
  var a = read(),
  var MiIdentificador = 10,
  var mi_segundo_identificador = 20,
  var a = true,
  if a {
    for 0 {
      print(i),
    }
  }

  if a == false {
  }

  if b < 11 {
    var entero = 123,
  }

  if a != false {
    var flotante = 1.23,
  }

```

```
26   /* comentario multilinea
27   mas lineas del comentario */
28
29   var booleano = true,
30   var nulo = null,
31
32   return 0
33 }
```

5.2.3 BExample3.ds

El primer error se encuentra en la línea 6, no se espera que haya un id (AddOne) sin usar la palabra reservada func antes.

```
1  func AddTwo (a) {
2    a = a + a + (a),
3    return a + 2,
4  }
5
6  AddOne (a) {
7    return a + 1,
8  }
9
10 main () {
11   var a = exec AddOne(1),
12   var b = exec AddTwo(1),
13   if a > 5 {
14     print("True"),
15   }
16   else {
17     print("False"),
18 }
```

```
Error in bexample.ds file:
 Expected: comma in [2,30]

Error in bexample2.ds file:
 Not expected: comment : "// esto es un comentario de una linea" in [1,0]

Error in bexample3.ds file:
 Not expected: id : "AddOne" in [6,56]
```

Figure 4: Errores de los 3 ejemplos no aceptados

6 Conclusiones

El objetivo de este trabajo es crear un lenguaje que permita mayor comodidad al programador, con su forma de declarar tan fácil se espera crear un código legible y que sea fácil de identificar. Para facilitar ciertas labores lo mejor es usar herramientas web como LL1 Machine, y HTML-to-CSV. Durante la implementación del Algoritmo LL1 es mejor personalizar y agregar ciertas funciones para ver el progreso, en este caso se utilizaron funciones para generar archivos para la lista de Tokens (.out) y

archivos para generear el árbol sintáctico (.dot), además de trabajar con el código dividido para tener un mejor rendimiento y entendimiento.

Repositorio GitHub