Centro Universitario de Ciencias Exactas e Ingenierías

Compiladores

Hands-on 3: Implementación de Analizadores Sintácticos

Carrera: Ingeniería en computación.

Alumno: Fabián Joheshua Escalante Fernández

Materia: Compiladores

Calendario: 2025A

Fecha: 04/05/2024

Hands-on 3: Implementación de Analizadores Sintácticos

Índice:

Índice de figuras:

Hands-on 3: Implementación de Analizadores Sintácticos

Introducción:

Vamos a ver como se hacen expresiones sintácticas usando BNG, que contiene símbolos de tipo no terminales o abstractos, estos son los que están entre menos que y mayor que, terminales o tokens/símbolos finales, siendo TOKEN, = o />, y las reglas de producción que definen como los no terminales interactúan con los terminales.

Desarrollo:

Primero intente instalar MSYS2 pero como mi equipo ya contaba con la integración de C y python entonces fui directo al paso 2.2 que era manejar un subsistema de Windows para Linux. Se instalo correctamente habilitando las características en Windows y reiniciando la PC. Adentro de este entorno instale flex y bison, herramientas desconocidas para mi, que después vi que eran para construir analizadores.

Flex:

Fast Lexical Analyzer Generator, esta herramienta toma patrones y genera en C un scanner que lee los inputs y outputs de los símbolos terminales. Osea que las expresiones regulares de un autómata finito son minimizadas y se genera el código de C para el reconocimiento de patrones.

Bison:

Usa una gramática libre de contexto y en C produce un parser que prueba la validez de una secuencia de tokens y ejecuta las acciones asociadas, en términos semánticos.

Pues Flex tokeniza y Bison analiza. Fundamentos de los compiladores e intérpretes modernos. Regresando al tema, después se cree el archivo validar.l y validar.y aunque para validar.y me estaba dando unos fallos muy raros por no tener main y algunos de yywrap que no pude solucionar bien, entonces puse la función de main que solo imprime lo del imput y funciona mientras no de fallos yyparse, para yywrap puse un return solamente pero esto no funciono totalmente, aunque la verificación si es exitosa.

Hands-on 3 > validar.y

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
input:  expr '\n' { printf("Expresión válida\n"); }
  | error '\n' { yyerror("Expresión inválida"); yyerrok; }
  ;

expr: expr '+' expr
  | expr '-' expr
  | expr '*' expr
  | expr '/' expr
  | '-' expr %prec UMINUS
  | '(' expr ')'
  | NUMBER
  ;
%%

int yyerror(char *s) {
    fprintf(stderr, "%s\n", s);
    return 0;
}
int yywrap(void) {
    return 1;
}
int main(void) {
    printf("Añade la expresión que quieras:\n");
    while (yyparse() == 0) {}
    return 0;
}
```

Figura 1: validar.y Corregido

```
%{
#include "y.tab.h"
%}
%%
[0-9]+       { yylval = atoi(yytext); return NUMBER; }
[ \t]        ;
[\n]         { return '\n'; }
[+\-*/()]    { return yytext[0]; }
.            { return yytext[0]; }
%%
```

Figura 2: validar.l

Y la ejecución fue exitosa:

```
root@Skylon:~# bison -y -d validar.y
root@Skylon:~# ls
validar.l  validar.y  y.tab.c  y.tab.h
root@Skylon:~# flex validar.l
root@Skylon:~# gcc lex.yy.c y.tab.c -o validar -lfl -lm
```

Figura 3: Ejecución

Aplicación de ejemplos

```
root@Skylon:~# ./validar
Añade la expresión que quieras:
3 + 5 * (2 - 1)
Expresión válida
```

Figura 4: Ejemplo 1

```
root@Skylon:~# ./validar
Añade la expresión que quieras:
(8 - 4) / 2
Expresión válida
```

Figura 5: Ejemplo 2

```
Añade la expresión que quieras:
3 + * 2
syntax error
Expresión inválida
```

Figura 6: Ejemplo 3

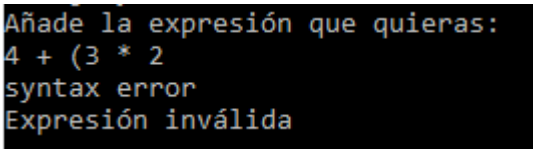

Figura 7: Ejemplo 4

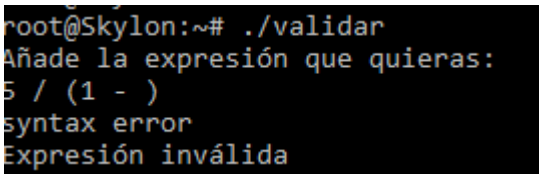

Figura 8: Ejemplo 5

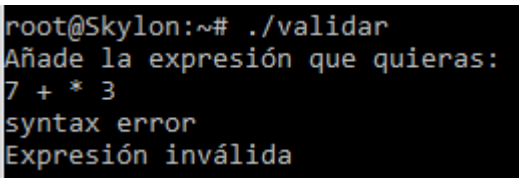

Figura 9: Ejemplo 6

Aplicación en Python y sus ejemplos

En la tarea no esta actualizada la forma en que se ejecuta en python, por lo que investigue en stack overflow y comentaban varias formas de hacerlo, entonces intente hacerlo como hice un script en otra materia de autómatas, usando PLY y reutilizando el código que usamos de Flex y Bison.

Hands-on 3 > validador.py > ...

```python
import sys
import ply.lex  as lex
import ply.yacc as yacc

tokens = ('NUMBER',)
literals = ['+', '-', '*', '/', '(', ')']

t_ignore = ' \t'

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_error(t):
    print(f"Caracter ilegal '{t.value[0]}'")
    t.lexer.skip(1)

precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
)

def p_input_valid(p):
    'input : expr'
    print("Expresión válida")

def p_input_error(p):
    'input : error'
    print("Expresión inválida")

def p_expr_binop(p):
    '''expr : expr '+' expr
            | expr '-' expr
            | expr '*' expr
            | expr '/' expr'''
    pass
```

Figura 10: Validador en python parte 1

```python
def p_expr_uminus(p):
    'expr : \'-\' expr %prec UMINUS'
    pass

def p_expr_group(p):
    'expr : \'(\' expr \')\''
    pass

def p_expr_number(p):
    'expr : NUMBER'
    pass

def p_error(p):
    raise SyntaxError

lexer = lex.lex()
parser = yacc.yacc()

def iniciar():
    print("Añade la expresión que quieras:")
    for line in sys.stdin:
        line = line.strip()
        if not line:
            continue
        try:
            parser.parse(line, lexer=lexer)
        except Exception:
            print("Expresión inválida")


iniciar()
```

Figura 11: Validador en python parte 2

```
(root) root@Skylon:~# python3 'validador - copia.py'
Añade la expresión que quieras:
3 + 5 * (2 - 1)
Expresión válida
(8 - 4) / 2
Expresión válida
3 + * 2
Expresión inválida
(3 + 5 * 2 -
Expresión inválida
5 / (3 - 1
Expresión inválida
2 * (4 + )
Expresión inválida
```

Figura 12: Ejemplos generales y ejemplos de la invitación al estudiante en el validador en python

Ya con el conocimiento de la creación de estos validadores en Python entonces decidí hacer el hands on con Python. Cuidando también que se apaguen a PEP8 y si hay algo que de verdad no entienda entonces pedirle a ChatGpt explicación y que me ayude para que funcione.

Hands-on 3.1: Validador de Expresiones Aritméticas (Nivel Básico)

Hands-on 3 > hands_on__validador_basico.py > ...

```python
import sys
import ply.lex as lex
import ply.yacc as yacc

tokens = ('NUMERO',)
literals = ['+', '-', '*', '/', '(', ')']
t_ignore = ' \t'


def t_NUMERO(t):
    r'\d+'
    t.value = int(t.value)
    return t


def t_error(t):
    t.lexer.skip(1)


precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
)


def p_expresion(p):
    '''expresion : expresion '+' termino
                 | expresion '-' termino
                 | termino'''
    pass


def p_termino(p):
    '''termino : termino '*' factor
               | termino '/' factor
               | factor'''
    pass
```

Figura 13: Validador de Expresiones Aritméticas (Nivel Básico) Parte 1

```python

def p_factor(p):
    '''factor : '-' factor %prec UMINUS
              | '(' expresion ')'
              | NUMERO'''
    pass


def p_error(p):
    print('Expresión inválida')


analizador = yacc.yacc()
lexer = lex.lex()

print('Ingrese expresiones aritméticas:')
for linea in sys.stdin:
    linea = linea.strip()
    if not linea:
        continue
    try:
        analizador.parse(linea, lexer=lexer)
        print('Expresión válida')
    except Exception:
        print('Expresión inválida')

```

Figura 14: Validador de Expresiones Aritméticas (Nivel Básico) Parte 2

Ejemplo Validador de Expresiones Aritméticas (Nivel Básico):

```
(root) root@Skylon:~# python3 hands_on__validador_basico.py
Generating LALR tables
Ingrese expresiones aritméticas:
(4 + 5) * 2
Expresión válida
3 - (2 + )
Expresión inválida
```

Figura 15: Ejemplo Validador de Expresiones Aritméticas (Nivel Básico)

Invitación al Estudiante Validador de Expresiones Aritméticas (Nivel Básico):

```
(4 + 5) * 2
Expresión válida
2 * 3 * 4
Expresión válida
23 * 3/2
Expresión válida
```

Figura 16: Invitación al Estudiante Validador de Expresiones Aritméticas (Nivel Básico):

Hands-on 3.2: Validador de Expresiones Lógicas (Nivel Intermedio):

Hands-on 3 > hands_on__validador_medio.py > ...

```python
import sys
import ply.lex as lex
import ply.yacc as yacc

tokens = ('BOOLEANO', 'AND', 'OR', 'NOT')
literals = ['(', ')']
t_AND = r'AND'
t_OR = r'OR'
t_NOT = r'NOT'
t_ignore = ' \t'


def t_BOOLEANO(t):
    r'[01]'
    t.value = int(t.value)
    return t


def t_error(t):
    t.lexer.skip(1)


precedence = (
    ('right', 'NOT'),
    ('left', 'AND', 'OR'),
)


def p_expresion(p):
    '''expresion : expresion AND termino
                 | expresion OR termino
                 | termino'''
    pass


def p_termino(p):
    '''termino : NOT factor
               | factor'''
    pass


```


Figura 17: Hands-on 3.2: Validador de Expresiones Lógicas (Nivel Intermedio) Parte 1

```python
def p_factor(p):
    '''factor : '(' expresion ')'
              | BOOLEANO'''
    pass


def p_error(p):
    print('Expresión inválida')


analizador = yacc.yacc()
lexer = lex.lex()

print('Ingrese expresiones lógicas:')
for linea in sys.stdin:
    linea = linea.strip()
    if not linea:
        continue
    try:
        analizador.parse(linea, lexer=lexer)
        print('Expresión válida')
    except Exception:
        print('Expresión inválida')

```

Figura 18: Validador de Expresiones Lógicas (Nivel Intermedio) Parte 2

Ejemplos Validador de Expresiones Lógicas (Nivel Intermedio):

```
(root) root@Skylon:~# python3 hands_on__validador_medio.py
Generating LALR tables
Ingrese expresiones lógicas:
(1 AND 0) OR (NOT 1)
Expresión válida
(1 AND (0 OR 1)
Expresión inválida
```

Figura 19: Ejemplos Validador de Expresiones Lógicas (Nivel Intermedio)

Invitación al Estudiante Validador de Expresiones Lógicas (Nivel Intermedio):

```
(root) root@Skylon:~# python3 hands_on__validador_medio.py
Ingrese expresiones lógicas:
1 AND 0
Expresión válida
NOT (0 OR 1)
Expresión válida
5 + (2 AND NOT 0)
Expresión inválida
```

Figura 20: Invitación al Estudiante Validador de Expresiones Lógicas (Nivel Intermedio)

Hands-on 3.3 Validador de Expresiones Complejas (Nivel Avanzado):

Hands-on 3 > hands_on__validador_avanzado.py > ...

```python
import sys
import ply.lex as lex
import ply.yacc as yacc

tokens = ('NUMERO', 'BOOLEANO', 'AND', 'OR', 'NOT')
literals = ['+', '-', '*', '/', '(', ')']
t_AND = r'AND'
t_OR = r'OR'
t_NOT = r'NOT'
t_ignore = ' \t'

def t_NUMERO(t):
    r'\d+'
    t.value = int(t.value)
    return t

def t_BOOLEANO(t):
    r'[01]'
    t.value = int(t.value)
    return t

def t_error(t):
    t.lexer.skip(1)

precedence = (
    ('right', 'NOT', 'UMINUS'),
    ('left', 'AND'),
    ('left', 'OR'),
    ('left', '+', '-'),
    ('left', '*', '/'),
)

def p_expr(p):
    '''expr : expr OR expr
            | expr AND expr
            | arit'''
    pass

```

Figura 21: Validador de Expresiones Complejas (Nivel Avanzado) Parte 1

```
39  def p_arit(p):
40      '''arit : arit '+' arit
41              | arit '-' arit
42              | arit '*' arit
43              | arit '/' arit
44              | '-' arit %prec UMINUS
45              | '(' expr ')'
46              | NUMERO'''
47      pass
48
49  def p_error(p):
50      raise SyntaxError
51
52  analizador = yacc.yacc()
53  lexer = lex.lex()
54
55  print('Ingrese expresiones complejas:')
56  for linea in sys.stdin:
57      linea = linea.strip()
58      if not linea:
59          continue
60      try:
61          analizador.parse(linea, lexer=lexer)
62          print('Expresión válida')
63      except SyntaxError:
64          print('Expresión inválida')
65
```

Figura 22: Validador de Expresiones Complejas (Nivel Avanzado) Parte 2

Ejemplos Validador de Expresiones Complejas (Nivel Avanzado):

```
Generating LALR tables
Ingrese expresiones complejas:
(4 + 5) * (2 AND 1)
Expresión válida
(2 AND 3) / (4 - 1
Expresión inválida
```

Figura 23: Ejemplos Validador de Expresiones Complejas (Nivel Avanzado)

Invitación al Estudiante Validador de Expresiones Complejas (Nivel Avanzado):

```
(root) root@Skylon:~# python3 hands_on__validador_avanzado.py
Ingrese expresiones complejas:
(4 + 5) * 2
Expresión válida
(4 + 5) * (2 AND 1)
Expresión válida
(3 * 4) OR 1
Expresión válida
```

Figura 24: Invitación al Estudiante Validador de Expresiones Complejas (Nivel Avanzado):

Conclusión:

Me gusto esta actividad aunque fue un poco difícil empezar a entender cómo funcionaba cada cosa y luego pasar de C a Python, por eso al final use Python pero no estoy seguro que haya sido más fácil. Esta forma de expresiones me recordó mucho a los autómatas y como podemos determinar si p es un valor correcto o no, de ahí en más fue buscar información y poner banderas.

Bibliografía:

[1] GeeksforGeeks. (2024, 17 septiembre). FLEX (Fast Lexical Analyzer Generator). Recuperado de https://www.geeksforgeeks.org/flex-fast-lexical-analyzer-generator/?utm_source=chatgpt.com

[2] Wikipedia contributors. (2025, 14 abril). Flex (lexical analyser generator). Recuperado de https://en.wikipedia.org/wiki/Flex_%28lexical_analyser_generator%29?utm_source=chatgpt.com

[3] BisOn - GNU Project - Free Software Foundation. (s. f.). Recuperado de https://www.gnu.org/software/bison/?utm_source=chatgpt.com