

Generación automática de Analizadores de Léxico y Sintaxis

Lex&Yacc:

Plataforma y lenguaje base.

- ❑ Lex&Yacc son programas separados pero que se utilizan de forma conjunta para generar análisis sintáctico, que es tomar un código base y descubrir su estructura. Lex&Yacc están hechos en C.

Características básicas (formato requerido en las entradas, tipo de ejecución, etc.)

- ❑ Los archivos Lex&Yacc se dividen en tres secciones las cuales son divididas con línea independientes que cuentan solamente con dos signos “%”. En la primera sección se ponen las declaraciones, en la segunda las reglas y en el tercero se pone el código escrito en C.
- ❑ En las declaraciones se definen los macros y archivos que se importaran, en las reglas se asocian expresiones regulares a sentencias y en la última sección se ponen las funciones y código que será usado para el funcionamiento del código.

Herramientas teóricas en las que se basan: exp. regulares, métodos sintácticos empleados, etc.

- ❑ Ambas herramientas se basan en expresiones regulares, usan tokens para manejar estas mismas.

Tipo de interfaz. Facilidad para añadir código propio y cómo se haría.

- ❑ En la siguiente imagen se muestra un ejemplo de cómo se estructuran ambos programas, para añadir código propio simplemente hay que escribirlo en la parte correcta, es necesario saber en qué sección se debe poner cada cosa para evitar problemas

```
/** Sección de declaraciones **/  
  
%{  
/* Código en C que será copiado */  
#include <stdio.h>  
%}  
  
/* Esto indica a Flex que lea sólo un fichero de entrada */  
%option noyywrap  
  
%%  
/** Sección de reglas **/  
  
/* [0-9]+ identifica una cadena de uno o más dígitos */  
[0-9]+ {  
    /* yytext es una cadena que contiene el texto coincidente. */  
    printf("Encontrado un entero: %s\n", yytext);  
}  
  
. { /* Ignora todos los demás caracteres. */ }  
  
%%  
/** Sección de código en C **/  
  
int main(void)  
{  
    /* Ejecuta el 'lexer', y después termina. */  
    yylex();  
    return 0;  
}
```

Ejemplo: [https://es.wikipedia.org/wiki/lex_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/lex_(inform%C3%A1tica))

PLY:

Plataforma y lenguaje base.

- ❑ Ply está basado en las herramientas de Lex&Yacc para ser implementado en Python.

Características básicas (formato requerido en las entradas, tipo de ejecución, etc.)

- ❑ Uso de parseo-LR lo cual hace muy eficaz para procesar gramáticas largas
- ❑ Ply provee herramientas para revisar de forma extensa y meticulosa los errores en la revisión de la gramática.
- ❑ Para ser implementado hay que bajar el módulo de Ply en <https://github.com/dabeaz/ply>, este se debe agregar al proyecto en el que se trabajara y se debe importar. Al ser basado en Lex&Yacc la forma en que se estructura el programa es muy parecido, se divide en tres secciones: Declaraciones reglas y código. Las declaraciones definen los macros y archivos que se importarán, en las reglas se asocian expresiones regulares a sentencias y en la última sección se ponen las funciones y código que será usado para el funcionamiento del código.

Herramientas teóricas en las que se basan: exp. regulares, métodos sintácticos empleados, etc.

- ❑ Al ser basado en Lex&Yacc también basa su herramientas en expresiones regulares

Tipo de interfaz. Facilidad para añadir código propio y cómo se haría.

- ❑ Es muy fácil agregar código propio, abajo dejare un ejemplo sacado de la página de github del creador de Ply. El código propio debe ir en su respectivo lugar en el formato correcto.

```
tokens = (
    'NAME', 'NUMBER',
    'PLUS', 'MINUS', 'TIMES', 'DIVIDE', 'EQUALS',
    'LPAREN', 'RPAREN',
)

# Tokens

t_PLUS = r'\+'
t_MINUS = r'\-'
t_TIMES = r'\*'
t_DIVIDE = r'\/'
t_EQUALS = r'\='
```

```
t_LPAREN = r'\('
t_RPAREN = r'\)'

t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'

def t_NUMBER(t):

    r'\d+'

    t.value = int(t.value)

    return t

# Ignored characters

t_ignore = " \t"

def t_newline(t):

    r'\n+'

    t.lexer.lineno += t.value.count("\n")

def t_error(t):

    print(f"Illegal character {t.value[0]!r}")

    t.lexer.skip(1)

# Build the lexer

import ply.lex as lex

lex.lex()

# Precedence rules for the arithmetic operators

precedence = (

    ('left', 'PLUS', 'MINUS'),

    ('left', 'TIMES', 'DIVIDE'),

    ('right', 'UMINUS'),

)

# dictionary of names (for storing variables)

names = { }

def p_statement_assign(p):

    'statement : NAME EQUALS expression'

    names[p[1]] = p[3]

def p_statement_expr(p):

    'statement : expression'

    print(p[1])

def p_expression_binop(p):

    '''expression : expression PLUS expression

                  | expression MINUS expression

                  | expression TIMES expression'''
```

```
        | expression DIVIDE expression'''

    if p[2] == '+': p[0] = p[1] + p[3]

    elif p[2] == '-': p[0] = p[1] - p[3]

    elif p[2] == '*': p[0] = p[1] * p[3]

    elif p[2] == '/': p[0] = p[1] / p[3]

def p_expression_uminus(p):

    'expression : MINUS expression %prec UMINUS'

    p[0] = -p[2]

def p_expression_group(p):

    'expression : LPAREN expression RPAREN'

    p[0] = p[2]

def p_expression_number(p):

    'expression : NUMBER'

    p[0] = p[1]

def p_expression_name(p):

    'expression : NAME'

    try:

        p[0] = names[p[1]]

    except LookupError:

        print(f"Undefined name {p[1]!r}")

        p[0] = 0

def p_error(p):

    print(f"Syntax error at {p.value!r}")

import ply.yacc as yacc

yacc.yacc()

while True:

    try:

        s = input('calc > ')

    except EOFError:

        break

    yacc.parse(s)
```

Además, investigarás una segunda herramienta, comercial o no, que se “conecte” con cualquier lenguaje de programación (*preferentemente con el lenguaje que tengas planeado usar para tu desarrollo*).

ANTLR:

Plataforma y lenguaje base.

- ❑ ANTLR (Another tool for language recognition) es una herramienta para construir parsers, interpreters y compilers a partir de gramáticas. Está hecho en Java pero tiene disponibles versiones para diferentes lenguajes de programación como Python o C. ANTLR se puede usar en Linux, Windows y Mac OS.

Características básicas (formato requerido en las entradas, tipo de ejecución, etc.)

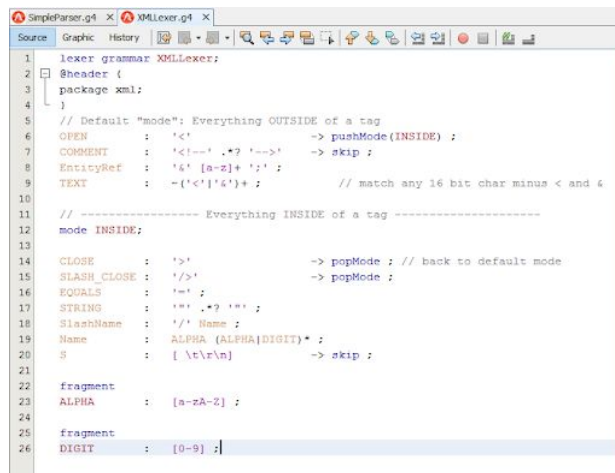
- ❑ ANTLR es un programa que escribe otros programas, recibe una descripción gramatical y se genera un programa para determinar si la sentencia o palabra es de cierto lenguaje en específico. ANTLR también destaca por dar mucha facilidad en la creación de estructuras intermedias de análisis.

Herramientas teóricas en las que se basan: exp. regulares, métodos sintácticos empleados, etc.

- ❑ Creación y manipulación de AST's, clases léxicas, manejo de errores, clases de recuperación de fallos, expresiones regulares.

Tipo de interfaz. Facilidad para añadir código propio y cómo se haría.

- ❑ Ejemplo de cómo se ve ANTLR, al igual que en las herramientas pasadas, simplemente se debe escribir las funciones que el programador quiere meter en los espacios correspondientes y de la forma correcta



```
1 lexer grammar XMLLexer;
2 $header {
3   package xml;
4 }
5 // Default "mode": Everything OUTSIDE of a tag
6 OPEN      : '<'          -> pushMode(INSIDE) ;
7 COMMENT   : '<!--' .*? '-->' -> skip ;
8 EntityRef : '&' [a-z]+ ';' ;
9 TEXT      : ~('<'|'&')+ ; // match any 16 bit char minus < and &
10
11 // ----- Everything INSIDE of a tag -----
12 mode INSIDE;
13
14 CLOSE     : '>'          -> popMode ; // back to default mode
15 SLASH_CLOSE : '/>'       -> popMode ;
16 EQUALS    : '=' ;
17 STRING    : '"' .*? '"' ;
18 SlashName : '/' Name ;
19 Name      : ALPHA (ALPHA|DIGIT)* ;
20 S         : [ \t\r\n] -> skip ;
21
22 fragment
23 ALPHA     : [a-zA-Z] ;
24
25 fragment
26 DIGIT     : [0-9] ;
```

Bibliografía:

<https://www.dabeaz.com/ply/>

<http://dinosaur.compilertools.net/>

<https://pragprog.com/titles/tpantlr2/the-definitive-antlr-4-reference/>