Laboratorio

Procesadores del Lenguaje

2022-2023

Presentación

- Profesores Laboratorio.

- Jesús García Herrero
- (jgherrer@inf.uc3m.es)

Objetivos

- Plantear una gramática para un lenguaje
  - Verificar si la gramática cumple propiedades para su traducción eficiente
- Generar un analizador para un lenguaje dado
- Construir los controles semánticos necesarios para verificar y traducir un lenguaje
- Conocer los principios de generación de código

Desarrollo y Herramientas

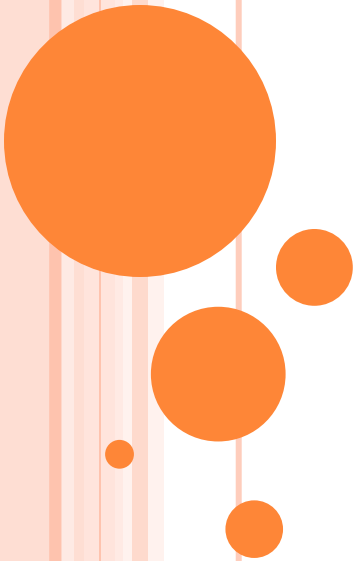
Las Prácticas de Laboratorio:

- **Se realizarán en grupos de dos**
- **Se programará en**
  - **lenguaje Python**
  - **módulo PLY**
  - **entorno libre**
- **Herramientas**
  - **PLY – lex Scanner**
  - **PLY- yacc Parsers**
    - **(http://www.dabeaz.com/ply/)**

Evaluación

Las Prácticas de Laboratorio:

- **Supondrán un 40% de la nota de la asignatura:**
  - **1ª Práctica – Introducción (0.75)**
  - **2ª Práctica – Análisis léxico (0.75)**
  - **3ª Práctica – Análisis sintáctico y semántico (2.5)**



Introducción a las herramientas de Compiladores

Procesadores del lenguaje

(lex y yacc con PLY)

Lex/Flex

- Analizador léxico (también conocido **Scanner**), escrito originalmente en C, reimplementado en Python en PLY-lex

- Diseñado para trabajar junto a Yacc, analizador sintáctico para gramáticas LALR.

https://www.cs.princeton.edu/~appel/modern/c/software/flex/flex.html

YACC/BISON

- Sirve para generar analizadores sintácticos (o ***parsers***) a partir de la especificación de una gramática independiente del contexto.

- YACC/ Bison originalmente desarrollados en lenguaje C, reimplementado en Python en PLY-yacc

https://www.gnu.org/software/bison/manual/bison.html

Instalación de PLY

- Descargar e instalar PLY:
  - Localizar la última versión: PLY-3.11
    http://www.dabeaz.com/ply/
  - Descomprimir la versión a instalar, ej.: ply-3.11.tar.gz
  - Añadir la raíz “python\ply-master” en el path del proyecto
  - Ya pueden importarse los módulos ply.lex, ply.yacc

Creación del proyecto

- Una vez instalado, podemos crear un nuevo proyecto Python de la forma usual.
  - Por facilidad, en la carpeta “python\example” tenemos algunos ejemplos ilustrativos, en un solo archivo .ply
- En este punto ya podemos empezar a rellenar los archivos de acuerdo a lo especificado en los manuales de lex y yacc

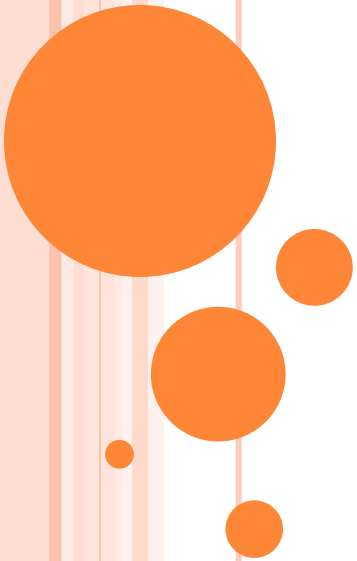
Ejemplo proyecto (PLY)

```
import sys
sys.path.insert(0, "../..")
# ...
# Especificacion de tokens y ERs
# ...
# Build the lexer
import ply.lex as lex
lex.lex()

# Parsing rules
# ...
# especificacion de gramática y acciones
# ...

import ply.yacc as yacc
yacc.yacc()
while 1:
    try:
        s = raw_input('calc > ')
    except EOFError:
        break
    if not s:
        continue
    yacc.parse(s)
```

Definir lenguaje léxico: Plantilla Flex

- Realizar las definiciones que sean necesarias para especificar los tokens y expreseiones regulares con la sintaxis de lex

- Reconocer los patrones indicados:
  - Palabras reservadas: **lang**, **es**, **eng**, ...
  - Números
  - Identificadores
  - Comentarios…



LEX

Procesadores del lenguaje (PLY-lex)

Formato de LeX

- Funcionamiento general: el analizador creado busca en la entrada ocurrencia de los patrones
  - Cuando encuentra un patrón, devuelve el token correspondiente
    - Si tiene función definida, ejecuta sus acciones, y debe devolver el token. Si no, continua la búsqueda
  - Si varios patrones encajan, selecciona el primero declarado
- Por defecto, si el texto no encaja con ningún patrón se envía a al función de error

Representación de patrones con Lex

Literal	"x"	La cadena x
Literal	*	El carácter literal: '*'
Selección	a|ab	Selección de una alternativa: {a, ab}
Rango	[ad-gB-EG2-4]	{a,d,e,f,g,B,C,D,E,G,2,3,4}
Negación del Rango	[^a-z]	Cualquiera excepto minúsculas
Agrupar	(a-z)|(0-9)	Para agrupar patrones [a-z0-9]
Numeración	r*	Ocurrencia de r >=0
	r+	Ocurrencia de r >0
	r?	Cero o una ocurrencia de r
	r{2,4}	De 2 a 4 ocurrencias de r
Cualquier carácter (no \n)	.	(.|\n)* representa cualquier fichero
Localización	^r	r al principio de la línea
	r$	r al final de la línea

Aspectos adicionales PLY-Lex

- Especificacion de tokens con ERs
  - Dar nombre a patrones complejos facilitando la legibilidad
  - Se pueden anidar, otras definiciones se pueden reutilizar

```
tokens = (
    'letter',
    'digit',
    'id',
    'nl',
)

# tokens DEFINITION

t_letter = r'[A-Za-z]'
t_digit  = r'[0-9]'
t_id     = r'(' + t_letter +
           r'(' + digit + r'|'+ t_letter + r')*)'
t_nl = r'\n|\r|\r\n'
```

Aspectos adicionales PLY-Lex

- Los tokens pueden especificarse directamente con reglas o con funciones

```
tokens = (
    'NAME', 'NUMBER',
)
# Tokens
t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

t: clase LexToken
value
type
lineno
lexpos…

- Funciones especiales para **ignorar, error**, espacio en blanco …

```
t_ignore = " \t"
def t_newline(t):
    r'\n+'
    t.lexer.lineno += t.value.count("\n")
def t_error(t):
    print("Illegal character '%s'" % t.value[0])
    t.lexer.skip(1)
```

- Definición de las **condiciones de arranque** (incl/excl.)

Aspectos adicionales PLY-Lex

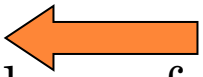
- Condiciones de arranque
  - Para modificar el flujo de análisis: reconocedores más potentes
  - Hay dos tipos de condiciones de arranque
    - Inclusivas
      - Evaluados los patrones con la condición de arranque y los que no utilizan ninguna
    - Exclusivas
      - Sólo se evalúan los de la condición de arranque
  - Cambiar de estado: función de lex **begin** ()

```
states = (
   ('foo','exclusive'),
   ('bar','inclusive'),
)
t_foo_NUMBER = r'\d+'      # Token 'NUMBER' in state 'foo'
t_bar_ID     = r'[a-zA-Z_][a-zA-Z0-9_]*' # Token 'ID' in state 'bar'

def t_foo_newline(t):
    r'\n'
    t.lexer.lineno += 1
    t.lexer.begin('foo')
def t_bar_newline(t):
    t.lexer.begin('INITIAL')
```

Orden de identificación de reglas

- Orden para la identificación de patrones
  - Si para la entrada puedan aplicarse varias reglas:
  1. Las reglas literales se ordenan por longitud decreciente
     - Con la entrada **abc**:

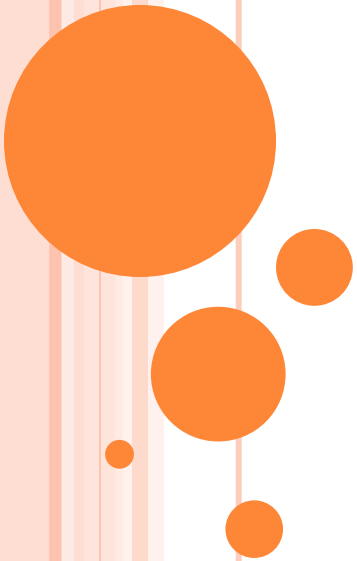
```
t_p1= r'a'
t_p2= r'ab'
t_p3= r'c'
t_p4= r'abc'
```



  2. Los patrones definidos por funciones se comprueban en orden de definición

```
def t_REAL(t):
    r'\d+\.\d+'
    t.value = float(t.value)
    return t
def t_ENT(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

(si cambiamos el orden no funciona)



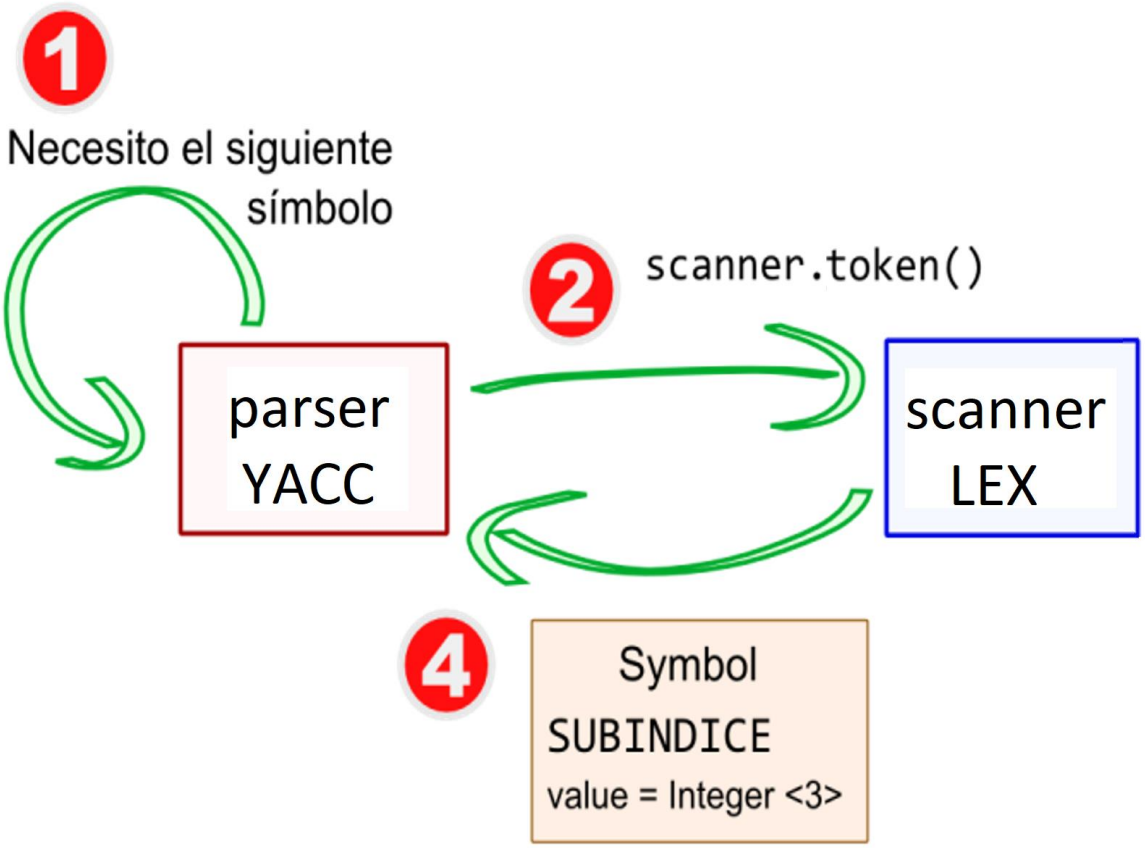
YACC

Procesadores del lenguaje (PLY-yacc)

Principio de Yacc

- Análisis previo de la gramática para construir el analizador LALR: notifica incidencias
- Funcionamiento general del analizador: pide tokens definidos al analizador léxico
  - Como es LALR(1), pide un token de pre-análisis
  - Cuando realiza desplazamiento pide el siguiente token
  - Cuando realiza reducciones de producciones, ejecuta el código en sus acciones

Comunicación Lex-Yacc



Creación del proyecto (yacc)

Terminales
No Terminales

Terminales: {ES, ENG, LANG, SALTO, PALABRA}

no terminales: {S, idioma, texto}

Gramática - acciones

```
S::=LANG idioma texto SALTO
     |LANG idioma texto SALTO S

texto::= PALABRA
          |  texto PALABRA
print("El texto introducido es " + TEXTO.text)

idioma::=ES
            |ENG
if (ES):
   print ("El Idioma elegido es Cast");
else:
   print ("El Idioma elegido es Ingles");
```

Declaraciones de terminales

- Ejemplo definición terminales
  - Lenguaje de expresiones aritméticas con enteros

```
tokens = (
    'NAME', 'NUMBER',
)

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

# Tokens
t_NAME = r'[a-zA-Z_][a-zA-Z0-9_]*'
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t…
```

Declaraciones de terminales

- Alternativa definición sin literales

```
tokens = (
    'NAME', 'NUMBER', 'EQUALS', 'PLUS', 'MINUS',
    'BY', 'DIV', 'LBRACE', 'RBRACE',
)
# Tokens
t_NAME   = r'[a-zA-Z_][a-zA-Z0-9_]*'
t_EQUALS = r'='
t_PLUS   = r'\+'
t_MINUS  = r'-'
t_BY     = r'\*'
t_DIV    = r'\/'
t_LBRACE = r'\('
t_RBRACE = r'\)'
...
```

Reglas en YACC

- Formato BNF simplificado

  LI: LD *acción*

  - **LI**: es un símbolo no-terminal del lenguaje
  - **LD**: secuencia de símbolos no-terminales y terminales

- Agrupar varias reglas del mismo no terminal :

```
expression : expression '+' expression
           | expression '-' expression
           | expression '*' expression
           | expression '/' expression
```

- Si se deja vacía es el regla de la palabra vacía

```
sentencias : sentencias ';' sentencia
           |
```

- ***acción***: sentencias en código (puede ser vacío)

Codificación de gramática en PLY

```
# dictionary of names
names = {}

def p_statement_assign(p):
    'statement : NAME "=" expression'
    names[p[1]] = p[3]

def p_statement_expr(p):
    'statement : expression'
    print(p[1])

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' 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]
...
```

p[i]: pila semántica. array de objetos del tipo definido para el atributo de cada símbolo de la gramática

Ejemplo de análisis

- Gramática

  expr→expr + term

  expr→term

  term→term * factor

  term→factor

  factor→NUMERO

  factor→(expr)

- Ejemplo de ejecución
  - 5*(6+1)+3

factor--> NUMERO (5)
term --> factor
factor--> NUMERO (6)
term --> factor
expr --> term
factor--> NUMERO (1)
term --> factor
expr --> expr MAS term
factor--> (expr)
term --> term POR factor
expr --> term
factor--> NUMERO (3)
term --> factor
expr --> expr MAS term

Resolución de ambigüedad

- Por defecto, después de avisar, yacc resuelve
  - D/R: Desplazar prioridad sobre reducir
  - R/R: Reducir por la producción primera
- Preferible resolver los conflictos explícitamente
  - Criterios de prioridad

Precedencia

- Especificación de precedencia
  - Asociatividad izquierda (mismo operador)
    - **('left', 'op'): x op y op z → (x op y) op z**
  - Asociatividad derecha (mismo operador)
    - **('right' , 'op'): x op y op z → x op (y op z)**
  - No asociatividad (mismo operador)
    - **('nonassoc' , 'op'): x op y op z INCORRECTO**
  - Con otros operadores: los declarados en líneas posteriores más precedencia
- Ejemplo de asociación:

  ('left', '+', '-')

  ('left', '*', '/')

  - El último declarado es el que tiene más precedencia

Gramática ambigüa: expresiones con prioridad

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
)
...
def p_statement_assign(p):
    'statement : NAME "=" expression'
    names[p[1]] = p[3]
def p_statement_expr(p):
    'statement : expression'
    print(p[1])

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
```

Gramática ambigüa: expresiones con prioridad

```
precedence = (
    #('left', '+', '-'),
    #('left', '*', '/'),
    ('right', 'UMINUS'),
)
...
def p_statement_assign(p):
    'statement : NAME "=" expression'
    names[p[1]] = p[3]
def p_statement_expr(p):
    'statement : expression'
    print(p[1])

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
```

```
calc > 2*3+1+2+8
28
calc >
```

Gramática ambigüa: expresiones con prioridad

```
precedence = (
    ('left', '+', '-'),
    ('left', '*', '/'),
    ('right', 'UMINUS'),
)
...
def p_statement_assign(p):
    'statement : NAME "=" expression'
    names[p[1]] = p[3]
def p_statement_expr(p):
    'statement : expression'
    print(p[1])

def p_expression_binop(p):
    '''expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression'''
```

```
calc > 2*3+1+2+8
17
calc >
```