Instituto Tecnológico de Buenos Aires

TRABAJO PRÁCTICO ESPECIAL

Diseño e Implementación de un Lenguaje:

Weird Flex but Ok

Materia:

(20242Q) 72.39 Autómatas, Teoría de Lenguajes y Compiladores

Profesores:

Ana María Arias Roig

Mario Agustín Golmar

Índice

1. Equipo:

Lonegro Gurfinkel, Lucas 63.738 - llonegrogurfinkel@itba.edu.ar
Noceda, Tobías Martín 63.422 - tnoceda@itba.edu.ar
Smirnoff, Sergio - 62.256 - ssmirnoff@itba.edu.ar
Torlaschi, Patrick Luca 62.273 - ptorlaschi@itba.edu.ar

2. Repositorio

La solución y su documentación serán versionadas en: TP-TLA.

3. Dominio

Desarrollar un lenguaje que permita generar analizadores léxicos en el lenguaje Java. El lenguaje debe permitir generar archivos ".l" del lenguaje Flex de una manera más amigable, incluyendo tener solo 1 espacio de declaraciones donde están todas las reglas variables y funciones, aceptando los tipos de caracteres que se requieran, como alfanuméricos, variables, entre otros.

El lenguaje deberá permitir ser granular para tener un estado por token y hacerlo inmodificable desde el exterior profundizando su contexto.

El lenguaje se compone de reglas, variables, y funciones las cuales definirán qué token se le otorga a cada carácter en cada caso.

Luego de su implementación, permitiría al usuario generar un analizador léxico de una manera fácil y sencilla para poder tokenizar los distintos programas que se les pase al analizador en, un lenguaje orientado a objetos, JAVA.

4. Construcciones

El lenguaje desarrollado debería ofrecer las siguientes construcciones, prestaciones y funcionalidades:

(I). Recibir por entrada una expresión regular.
(II). Permitir definir variables, funciones y reglas en el mismo sector.
(III). Permitir definir reglas a partir de otras reglas (macros) previamente definidas.
(IV). Permitir pasar variables previamente definidas a funciones asociadas con reglas.
(V). Flexibilizar la aceptación de escritura del .l para no tener que preocuparse por espacios o ”\n”.
(VI). Elaborar los tokens antes de entrar al bison.
(VII). Utilizar variables locales de contexto para cada token haciendo que no se puedan modificar estas variables desde otros entornos.
(VIII). Utilizar “gramática” y estilo similar a java para facilitar su transición.
(IX). Retornar un fichero con extensión .java.
(X). Permitir comentarios de una línea con // de forma tal que se ignore todo hasta el \n
(XI). Permitir comentarios de líneas múltiples con /* */ de forma que se ignore todo lo que hay entre las /
(XII). Aceptar más caracteres Unicode. (i18n)
(XIII). Crear aliases/ “Macros” predefinidos para expresiones regulares comúnmente utilizadas ej [:digit:], [:lower:], [:upper:], [:alpha:], [:alphan:]
(XIV). Permitir escapar caracteres reservados en expresiones regulares.
(XV). Aceptar múltiples archivos de entrada.

5. Casos de Prueba

Se proponen los siguientes casos iniciales de prueba de aceptación:

(I). Una Regex con una única clausura.

(II). Una Regex con una $clausura^{+}$.

(III). Una Regex únicamente con letras.

(IV). Una Regex únicamente con números.

(V). Una Regex conteniendo todo tipo de carácter.

(VI). Una Regex conteniendo todo tipo de caracteres y caracteres especiales.

(VII). Una Regex con contenido alfanumérico.

(VIII). Un programa con acción por defecto.

(IX). Un programa con creación de atributo y acción.

(X). Un programa que genere un analizador léxico para PL/0.

Además, los siguientes casos de prueba de rechazo:

(I). Un programa malformado.

(II). Un programa que tenga una Regex mal formada.

(III). Un programa que intente utilizar una Regex que no fue definida previamente.

(IV). Un programa que utilice un carácter no aceptado.

(V). Un programa que utilice una acción no existente.

(VI). Un programa con una expresión que no sea una función asociada a una regla

6. Ejemplos

Un programa que genera un analizador léxico para el lenguaje PL/0:

```
"+" -> PLUS
"-" -> MINUS
"*" -> TIMES
"/" -> SLASH
"(" -> LPAREN
")" -> RPAREN
";" -> SEMICOLON
"," -> COMMA
"." -> PERIOD
":=" -> BECOMES
"=" -> EQL
"<>" -> NEQ
"<" -> LSS
">" -> GTR
"<=" ->LEQ
">=" -> GEQ
"begin"-> BEGINSYM
"call" -> CALLSYM
"const" -> CONSTSYM
"do" -> DOSYM
"end" -> ENDSYM
"if" -> IFSYM
"odd" -> ODDSYM
"procedure" -> PROCSYM
"then" -> THENSYM
"var" -> VARSYM
"while" -> WHILESYM

Regex digit        [0-9];
Regex letter       [a-zA-Z];

{letter}({letter}|{digit})* -> (State var){
                    var.attribute.id = var.text.toString(); /* var.text is an instance of StringBuilder */
                    return IDENT;
 }

{digit}+ -> (State var){
                    var.attribute.num = Integer.parseInt(var.text.toString());
                    return NUMBER;
 }

[ \t\n\r]          /* skip whitespace */

default -> REJECT
```

En este caso se pasa una Clase State la cual contiene los datos necesarios para armar los contextos en cada caso. La clase tiene un state.instance, la cual pasa una copia suya a cada función para que no pueda ser modificada externamente.

Un programa que solo analiza las Regex con números y algunos símbolos matemáticos:

```
"+" -> PLUS
"-" -> MINUS
"*" -> TIMES
"/" -> SLASH
"("  -> LPAREN
")" -> RPAREN
"="  -> EQL
"<>" -> NEQ
"<"  -> LSS
">"  -> GTR
"<=" ->LEQ
">=" -> GEQ

Regex digit        [0-9];

{digit}+ -> (State var){
                    var.attribute.num = Integer.parseInt(var.text.toString());
                    return NUMBER;
 }

[ \t\n\r]          /* skip whitespace */

default -> { return REJECT }
```

Class State

- .instance
- .attribute
  - .id
  - .num
- .lineNumber
- .lexeme
  - .length
  - .row
  - .column
- .token