Teoría de Autómatas Lenguajes y Compiladores

TPE:

Yet Another Compiler Compiler

Instituto Tecnológico de Buenos Aires

Buenos Aires, Argentina

2016

Grupo Z80
Integrantes:

● Agop Hurmuz	ahurmuz@itba.edu.ar	53248
● Kevin Hirschowitz Kraus	khirscho@itba.edu.ar	52539
● Ariel Debrouvier	ndebrouvier@itba.edu.ar	55382

Profesores:

- Rodrigo Ramele
- Ana Roig
- Juan Miguel Santos

Indice

Idea subyacente y objetivo del lenguaje

El lenguaje que desarrollamos busca un primer acercamiento a aquellos que se están iniciando en la programación y no están tan familiarizados con el lenguaje inglés. Por esto mismo, el lenguaje que realizamos tiene una sintaxis más intuitiva para estos individuos y en español.

Consideraciones realizadas

Si bien nuestro analizador sintáctico está programado en C y genera lenguaje output también escrito en C, esto último se podría variar. Es decir, podríamos hacer que transforme a cualquier lenguaje, Java, Assembler, etc.

Descripción del desarrollo del TP

Se utilizó el analizador léxico Lex para reconocer los tokens o lexemas de nuestro lenguaje, definidos por la gramática. Posteriormente, a partir de la herramiento de parseo Yacc, se establecieron las reglas o producciones de la gramática, para así definir qué programas forman parte de nuestro lenguaje.

Se incluyeron los siguientes programas de ejemplo para tener una idea de como es la sintaxis del lenguaje:

- *Factorial:* Calcula el factorial de un número natural.
- *Fibonacci*: Calcula un término de la sucesión de fibonacci.
- *Potencia:* Calcula la potencia de una base con un exponente.
- *Test de primo*: Verifica si un número es primo.
- *Máximo común divisor:* Calcula el máximo común divisor entre dos enteros.

Descripción de la gramática

La gramática desarrollada en el siguiente trabajo práctico admite una estructura lógica de código similar a la del lenguaje de programación C.

Presenta tipo de datos entero y cadena de texto y las instrucciones se finalizan con un punto. Los delimitadores del comienzo y final de un programa son las palabras reservadas EMPEZAR y TERMINAR. Soporta las operaciones aritméticas de suma, resta, división,

multiplicación y módulo, así como expresiones booleanas con los conectivos and, or y not. Se presenta bloques condicionales if, con y sin else, y bloques do-while. Admite un mecanismo de salida de datos similar a printf, y un mecanismo de entrada de datos como scanf.

```
G = < NT, T, S, P >
T = { START, END, END_INSTR, VAR_NAME, OP_PLUS_ONE, OP_SUB_ONE, INT_VAR, STRING_VAR,
OP_ASSIGN, OP_PRINT, OPEN_PARENTHESIS, STRING, COMA, CLOSE_PARENTHESIS, INTEGER, OP_IN,
AMPERSAND, IF, ELSE, DO, WHILE, OPEN_BLOCK, CLOSE_BLOCK, OP_OR, OP_AND, OP_NEG, TRUE,
FALSE, OP_SUM, OP_SUB, OP_MODULO, OP_MUL, OP_DIV, OP_LT, OP_GT, OP_EQ, OP_DIST, OP_LE,
OP_GE }

NT = { S, code,instruction, declaration, type, asign, print, in, control_sequence, if,
loop, open_block, close_block, boolean_expression, boolean_term, boolean_factor, boolean,
comparation, expression, term, factor, compare_operator }
P = {S ->  START code END

code ->  | instruction code | control_sequence code

instruction -> declaration asign END_INSTR
| declaration asign_string END_INSTR
| declaration END_INSTR
| print END_INSTR
| VAR_NAME assign END_INSTR
| VAR_NAME assign_string END_INSTR
| VAR_NAME OP_PLUS_ONE
| VAR_NAME OP_SUB_ONE
| in END_INSTR

declaration -> type VAR_NAME

type -> INT_VAR | STRING_VAR

assign -> OP_ASSIGN expression

print -> OP_PRINT OPEN_PARENTHESIS STRING COMA STRING CLOSE_PARENTHESIS
|OP_PRINT OPEN_PARENTHESIS STRING COMA VAR_NAME CLOSE_PARENTHESIS
|OP_PRINT OPEN_PARENTHESIS STRING COMA INTEGER CLOSE_PARENTHESIS

in -> OP_IN OPEN_PARENTHESIS STRING COMA AMPERSAND VAR_NAME CLOSE_PARENTHESIS

control_sequence -> if | loop

if -> IF OPEN_PARENTHESIS boolean_expression CLOSE_PARENTHESIS open_block code close_block
|
 IF OPEN_PARENTHESIS boolean_expression CLOSE_PARENTHESIS open_block code close_block ELSE
open_block code close_block
```

```
loop -> DO open_block code close_block WHILE open_parenthesis boolean_expresion
close_parenthesis | WHILE open_parenthesis boolean_expresion close_parenthesis open_block
code close_block

open_block -> OPEN_BLOCK

close_block -> CLOSE_BLOCK

boolean_expression -> boolean_expression OP_OR boolean_term | boolean_term

boolean_term -> boolean_term OP_AND boolean_factor | boolean_factor

boolean_factor -> OPEN_PARENTHESIS boolean_expression CLOSE_PARENTHESIS | OP_NEG
boolean_factor | boolean

boolean -> TRUE | FALSE | comparation

comparation ->  expression compare_operator expression

expression -> OPEN_PARENTHESIS expression OP_SUM term CLOSE_PARENTHESIS | OPEN_PARENTHESIS
expression OP_SUB term CLOSE_PARENTHESIS | term
| expression OP_SUM term | expression OP_SUB term | expression OP_MODULO term

term -> OPEN_PARENTHESIS term OP_MUL factor CLOSE_PARENTHESIS | OPEN_PARENTHESIS term
OP_DIV factor CLOSE_PARENTHESIS | factor
| term OP_MUL factor | term OP_DIV factor

factor -> VAR_NAME | INTEGER

compare_operator -> OP_LT | OP_GT | OP_EQ |OP_DIST | OP_LE | OP_GE
}
```

Dificultades encontradas

No pudimos compilar el proyecto en una Macbook, sin embargo, bajo el mismo makefile y archivos compilaba sin problemas en distribuciones de Linux en pampero y Ubuntu. No pudimos detectar donde yacía el problema, probablemente se debe a una compatibilidad de librerías de compilación. Intentamos variando yacc por bison, una versión similar pero más actualizada y tampoco pudimos obtener resultados favorables.

Futuras extensiones

Se podría agregar a futuro ciclos for sin demasiada dificultad, ya que el lenguaje lo soportaría. Bastaría con definir la estructura con una posible sintaxis “repetir[inicio, hasta, entrevuelta]” y definir todo el tipo de sentencias que admitiríamos en cada una de las expresiones inicio, hasta y entrevuelta.

Se podrían agregar más alternativas para acortar código, *syntactic sugar*, esto volvería más atractivo para el uso al lenguaje o desestructurarlo un poco más, ya que aún posee los mismos principios que C en cuanto a la formación de estructuras básicas. Si bien esto último no es ni correcto ni incorrecto se podría jugar un poco más con las distintas estructuras para poder encadenarlas de distinta forma.

Otra posible extensión sería que haya múltiples tokens que mapeen a un mismo símbolo terminal de C, la gran ventaja que nos prestaría esto sería que podemos definir nuestro lenguaje en español/francés y con cualquier alternativa se traduciría a su equivalente en C-inglés.

Esto sin embargo, traería muy poca coherencia en el código, debería permitirse que una parte este escrita en C-español y la otra en C-francés y que siga compilando bien son algunas de las preguntas que nos surgen ante estas alternativas.

Referencias

- http://dinosaur.compilertools.net/yacc/
- http://www.medina-web.com/programas/documents/tutoriales/lex_yacc/core/yacc.html
- http://www.tldp.org/HOWTO/Lex-YACC-HOWTO-6.html