72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Trabajo Práctico Especial
Grupo Golf

Integrantes:
Digon, Lucia (59030)
Lopez Guzman, Pedro Jeremías (60711)
Sartorio, Alan Gabriel (61379)

Profesores:
Santos, Juan Miguel
Arias Roig, Ana María
Ramele, Rodrigo Ezequiel

Fecha de Entrega: 28 de Noviembre 2021

Introducción

Los compiladores son elementos clave de la informática y los que permiten convertir prosa digital en acción. El objetivo principal de este trabajo práctico especial es el desarrollo completo de un lenguaje y su compilador, construyendo los dos componentes principales de un compilador: el analizador léxico y el analizador sintáctico.

Para lograrlo, se pensó una idea para el lenguaje y se creó la gramática y el compilador a partir de la misma. El compilador genera un programa ejecutable. Acorde a lo solicitado en la consigna, el programa fue desarrollado en C, utilizando Lex como Scanner (analizador léxico) y Yacc como Parser (analizador sintáctico).

Idea y objetivo del lenguaje

La idea para el lenguaje surgió a partir de Code Golfing, una disciplina recreativa entre programadores en las que se realizan competencias con el objetivo de escribir programas que resuelvan determinados problemas en la menor cantidad de caracteres posibles. Si bien comenzó como una idea simple para jugar, generó una comunidad dedicada que llegó a crear lenguajes especializados para esto.

El objetivo es crear un lenguaje que sea útil para Code Golfing mientras que siga siendo más o menos legible. Los tokens tratarían de tener la menor longitud posible para poder escribir el programa de la manera más corta posible.

Consideraciones realizadas

Algunas consideraciones a tener en cuenta:

- No se cuenta con una función “main”, sino que empieza a ejecutarse desde la primera línea.
- Los delimitadores son el salto de línea.

Desarrollo del TP

Al momento de buscar una idea para el trabajo práctico, se priorizó la creación de un lenguaje que fuese útil, y que pueda ser utilizado con algún propósito. Por este motivo, conociendo las competencias de Code Golfing, y que ya existen lenguajes creados para dichas competencias, se optó por crear un lenguaje con este fin.

Se decidió crear un parser que traduzca el lenguaje a C y que luego concatene una compilación de dicha traducción utilizando gcc. Algo que se tuvo muy en cuenta fue no permitir, dentro de lo posible, que sea gcc quien detecte errores en el código.

Para lograr dicho objetivo se decidió implementar una estructura de Abstract Syntax Tree con la cual, a medida que se leen los tokens, se van generando los nodos correspondientes a las instrucciones. Por ejemplo: nodos de inicialización, nodos de asignación, nodos de operaciones o de impresión. Una vez leído el documento y con el árbol construido, se pasó a realizar la traducción, chequeando y validando los nodos creados. Por ejemplo, en un nodo de operación aritmética se detecta un error si se observa que uno de los operandos es un string, deteniendo el proceso e indicando al usuario el error.

Cabe aclarar que se tomó la decisión de implementar dos tipos de datos: enteros y strings. Se decidió que, al igual que en otros lenguajes, los strings serían inmutables.

Gramática

Acorde a los requerimientos del trabajo práctico, la gramática que produce el lenguaje cuenta con tipos de datos numéricos y cadenas, constantes, delimitadores, operadores aritméticos, relacionales, lógicos, de asignación. Además, cuenta con un bloque condicional (if/else) y un bloque while (repetición hasta condición). Las instrucciones de uso del lenguaje se encuentran en el archivo “README.md” en el repositorio. A continuación, se detallarán los elementos de la gramática y luego sus producciones.

Elementos de la gramática:

En la representación se decidió que los símbolos terminales se escribirían con mayúsculas, mientras que los no terminales con minúscula. Además se tienen los símbolos terminales { } ( ), los cuales al no ser operadores, no fue necesario definir su asociatividad.

Para empezar con los símbolos terminales tenemos:

- NEW_LINE: salto de línea
- PRINT: símbolo “_”
- INPUT: símbolo “~”
- INIT_INT: símbolo “#”
- INIT_STR: símbolo “^”
- VARIABLE: cadena de caracteres que comienzan con una letra y pueden tener números

- STRING: cadena de caracteres encerrados por “ ”
- NUMBER: cadena de caracteres numéricos que no pueden comenzar por 0
- WHILE: símbolo “@”
- IF: símbolo “?”
- ELSE: símbolo “:”
- El resto de terminales corresponden a símbolos de operaciones, los cuales coinciden con la gramática de C a excepción de AND y OR que son “&” y “|” respectivamente. Esta decisión se tomó para hacer más pequeños los tokens.

Símbolos no terminales:

- line_list: Una lista de bloques separados por newlines.
- block: Una instrucción o múltiples instrucciones agrupadas, se utiliza para los cuerpos de los if, else y while.
- instruction: Un wrapper de las distintas instrucciones que se puede tener.
- print_statement: una instrucción que acepta una expresión a imprimir.
- input_statement: una instrucción que acepta el nombre de una variable entera y permite ingresar un número por entrada estándar en la misma.
- expr: Wrapper para las distintas expresiones que tenemos.
- initialization: Declara e inicializa una variable, o solo la inicializa.
- declaration: Declara una variable con un tipo (int o string).
- assignation: Asigna un valor a una variable previamente declarada.
- single_var: Expresión que representa el nombre de una variable.
- var_exp: Expresión que representa operaciones aritméticas entre variables o constantes.
- str_exp: Representa un string, por ejemplo: “prueba”.
- num_exp: Constantes numéricas positivas y negativas.
- while: Loop while, con condición y cuerpo.
- if: Estructura de control “if”, admite condición, cuerpo y, opcionalmente, cuerpo de “else”.

Producciones de la gramática:

- line_list:
  block | line_list **NEW_LINE** block | line_list **NEW_LINE**
- block:
  instruction | while | if | **‘{‘ NEW_LINE** line_list **NEW_LINE ‘}’**
- instruction:
  initialization | declaration | print_statement | assignation | input_statement
- print_statement:
  PRINT expr
- input_statement:
  INPUT single_var
- expr:
  var_exp | str_exp
- initialization:
  INIT_INT single_var ‘=’ var_exp | **INIT_STR** single_var ‘=’ str_exp
- declaration:

INIT_INT single_var

- assignation:
  single_var ‘=’ var_exp | single_var ‘=’ str_exp
- single_var:
  VARIABLE
- var_exp:
  single_var | num_exp |
  var_exp **PLUS** var_exp | var_exp **MINUS** var_exp |
  var_exp **TIMES** var_exp | var_exp **DIVIDED_BY** var_exp |
  var_exp **REMAINDER** var_exp | var_exp **LESS_THAN** var_exp |
  var_exp **LESS_EQUAL_THAN** var_exp |
  var_exp **GREATER_THAN** var_exp |
  var_exp **GREATER_EQUAL_THAN** var_exp |
  var_exp **AND** var_exp | var_exp **OR** var_exp | var_exp **IS_EQUAL** var_exp |
  NOT var_exp | **‘(‘** var_exp **‘)’**
- str_exp:
  STRING
- num_exp:
  NUMBER | **MINUS NUMBER**
- while:
  WHILE expr **WHILE** block
- if:
  expr **IF** block | expr **IF** block **ELSE** block

Dificultades encontradas

A continuación, se comentarán todos los problemas o desafíos que se presentaron a lo largo del desarrollo del proyecto y qué decisiones se tomaron para resolverlos.

La primera dificultad que se presentó fue cómo definir la gramática para que sea escalable. Es decir, que al momento de agregar cosas nuevas a la misma, no hubiese que reestructurar toda la gramática. Para encarar este problema, se utilizó como ayuda el pdf de Tom Niemann citado en las referencias. La mejor solución fue hacer una lista de bloques, donde cada bloque puede tener adentro una instrucción (inicialización, declaración, asignación), un while, un if, etc, y se pueden ir agregando más cosas y más estructuras si se requiere.

Otro problema que surgió fue encontrar la manera de detectar errores desde el árbol de sintaxis, chequear que no se estuviese operando con una variable no declarada o con variables de distinto tipo. Para ello, se creó una tabla de símbolo de 200 variables, y una estructura con el nombre de la variable y su tipo. De esta forma, cada vez que se crea una variable, se chequea si ya existe y cuando se realiza una operación, se hace un chequeo de que las variables existan y que los tipos sean iguales. Por ejemplo, al realizar una suma, se debe chequear que ambos sumandos sean del mismo tipo y que no se esté intentando sumar enteros con strings.

Luego, se presentó la dificultad de ver en qué línea imprimir los errores. Esto es debido a que el reductor del árbol no tiene acceso al yylineno porque cuando se reduce el árbol, el yylineno se encuentra en la última línea. La solución consistió en hacer que cada nodo del árbol almacene el número de línea en que se leyó. De esta manera, al encontrar

un error, se obtiene del nodo el número de línea para imprimirlo junto con la descripción del error.

Otro problema a resolver fue cómo diferenciar los números negativos del menos de la resta. Para ello se investigaron posibles soluciones al problema, y lo que finalmente se decidió fue no utilizar expresiones regulares que contemplen números negativos, y que al encontrar un menos sea el parser quien se encargue de detectar qué tipo de operación es, si es una resta o una constante negativa.

También a la hora de implementar la instrucción if/else se pudo ver en Yacc un conflicto de shift/reduce. Investigando un poco se encontró que este problema es bastante común, incluso apodado “Dangling else”, el cual surge por la asociación entre if’s y sus respectivos else’s. Para solucionar este problema se acudió a una solución presentada en stackOverflow, referenciada en la última sección de este informe.

Finalmente se buscó cómo resolver el tema de la precedencia de operaciones, y se decidió seguir las mismas precedencias que C. Se investigó que Yacc setea las precedencias en base a cómo son declaradas en el parser. Por este motivo se declararon en orden, para que automáticamente se seteen cuáles son las precedencias. Esto conlleva la decisión de crear un token por operación. Además Yacc permite definir la asociatividad de tokens para evitar ambigüedades, así se definió utilizando *%left* o *%right* todos los operadores.

Futuras extensiones

En el futuro, el lenguaje podría ser expandido de varias maneras. Debido al objetivo del lenguaje, siempre se puede expandir creando funciones útiles de manera compacta para poder resolver los problemas que se plantean en las competencias de Code Golfing en menos caracteres. Algunos ejemplos

- Loops en un rango que manejen la variable por su cuenta como un ciclo “for”.
- Soporte para más tipos de datos, primeramente números de punto flotante.
- Soporte para arrays con operaciones para los mismos.
- Una librería estándar que defina operadores más específicos para golf coding. (Por ej: una operación para saber si un número es primo)

Referencias

A continuación, se encuentra una lista del material consultado durante el desarrollo del tp, además del material subido por la cátedra al campus.

- Part 01: Tutorial on lex/yacc: https://www.youtube.com/watch?v=54bo1qaHAfk&t=4s
- Part 02: Tutorial on lex/yacc: https://www.youtube.com/watch?v=__-wUHG2rfM&t=983s
- LEX & YACC TUTORIAL by Tom Niemann: https://lafibre.info/images/doc/201705_lex_yacc_tutorial.pdf
- https://stackoverflow.com/questions/12731922/reforming-the-grammar-to-remove-shift-reduce-conflict-in-if-then-else