
ITBA
INSTITUTO TECNOLOGICO DE BUENOS AIRES
•UNIVERSIDAD PRIVADA•

Informe TP Especial

Teoría de lenguajes, automatas y compiladores

Integrantes

Cristóbal Rojas - 58564

Florencia Petrikovich - 58637

Gaston Lifschitz - 58225

Manuel Luque Meijide - 57386

Fecha de entrega: 1/12/2019

Índice

Idea subyacente y objetivo del lenguaje

Nuestro objetivo fue generar un compilador que acepte un lenguaje enfocado en operaciones matemáticas, aplicando conceptos vistos en la materia. Se crearon tipos nuevos de datos y built ins para realizar esto.

En el presente informe, se detalla la descripción y las funcionalidades que provee nuestro compilador, así como también las dificultades encontradas durante el desarrollo de la misma.

Consideraciones realizadas

No se realizaron consideraciones fuera del enunciado.

Descripciones del desarrollo del TP

Se utilizó LEX como analizador léxico para la gramática. En el mismo, definimos las palabras clave que acepta la misma. Estas no se difieren mucho de las de C dado que el enfoque era matemático y no sobre el lenguaje en sí. Se utilizó la herramienta de parseo YACC como el analizador sintáctico, el cual crea un árbol desde abajo con las producciones especificadas. En las producciones se establecieron reglas las cuales creaban estructuras llamadas Tokens, donde las mismas difieren en sus campos dependiendo de que token se analizaba. Estas estructuras fueron utilizadas para traducir el lenguaje a C una vez terminado el parseo. Este código C generado se utiliza como entrada para generar .c y poder luego crear un ejecutable mediante el uso de gcc.

Descripción de la gramática

Descripción general

- El programa debe comenzar con start y terminar con end.
  Ej:
```
start
   ---------
   ---------
 end
```

- Al igual que C, los comandos finalizan con ; (punto y coma).
- Permite las operaciones aritméticas: Suma, resta, multiplicación, división y módulo.
- Permite las operaciones ++ y -- sobre los números (X++, Y--)
- Permite la definición de funciones que reciben un solo parámetro numérico y devuelven un valor numérico.

Asignaciones

- Las asignaciones son manejadas de la siguiente manera: Inicialmente, se tiene un arreglo con las variables declaradas. Estas variables son guardadas con 3 atributos: un boolean para saber si fue declarada, el nombre de la variable y una sub-estructura que contiene información relevante. Esto permite tener un mayor registro de las variables y liberarlas al final del código.
- No se permite declarar nuevamente una variable al igual que C.
- Por default, la máxima cantidad de variables que se pueden generar es 100. Sin embargo, esto se puede modificar fácilmente cambiando el valor de un define. Si no se deseara un valor fijo, se podría crear un arreglo dinámico que se va expandiendo con el aumento en la demanda.
- Una variable se puede definir luego de haberla declarado al igual que en la misma declaración:
  Ej:

```
number a=15;
number b;
b=15;
```

- Se permiten las siguientes asignaciones: = , += , -= , /= , *=

Tipos de datos

La gramática implementada, soporta los siguientes tipos de datos:

- Number: tipo de dato que encapsula todos los tipos de datos numéricos (usados, por supuesto, en las operaciones matemáticas). Por lo cual, no es necesario especificar el tipo de variable numérica específicamente.
  - En esta versión, solo se aceptan números enteros. Se desea en una próxima versión expandirlo a double y float.
  - Solo se podrán realizar operaciones con number;
- String: Debe estar entre comillas dobles. Puede tener cualquier longitud. No se permiten operaciones sobre los strings. Su propósito es servir como métodos para imprimir mensajes al usuario.

Condicionales

- El condicional "IF" va seguida de una expresión (que podría ser un conjunto de operaciones relacionales u operaciones lógicas, por ende, solamente aquellas que pueden ser verdaderas o falsas). Una operación relacional sería por ejemplo a>b o b!=a. Una operación lógica serían extensiones de las leyes morgan.

- Para ejecutar código en un ciclo se usa la estructura calculate-while. El código dentro de las llaves se ejecutará hasta que la condición luego del while resulte falsa. Ej:

```
calculate {
        --------
        --------
        --------
} while(var >= 19999);
```

- Permite las relaciones únicamente “booleanas” al igual que el IF. Estas son las que contienen como operador >, <, >=, <=, !=, ==, &&, ||. Se recomienda el uso explícito de paréntesis a la hora de encadenar operadores lógicos.

Sumatoria y productoria

- Existen dos bloques para simular la sumatoria y productoria. Se especifica la variable en la cual se guardará el resultado (donde la misma se puede declarar en el mismo bloque), el rango que deberá ciclar una variable, y la expresión que se desea sumar o multiplicar en cada iteración.
- Todas las variables involucradas deben ser números, si no lo son, no compilara el programa.
- Prototipo:

```
summation(variable que acumula el resultado; asignación inicial; expresión; asignación final);
```

- Ejemplos:

COMPILA:

```
number var;
number n;
product (number num1; var = 0; var * var + 2; var = 4);
summation(var; n = 122; var + n; n = 144);
```

NO COMPILA:

```
number var;
string n;
summation(var; n = 122; var + n; n = 144);

number var;
product(string y; var = 122; var; var = 144);
```

- Considerar que es posible especificar un número mayor en la asignación inicial que en la final. Esto causaría un loop infinito, es responsabilidad del programador evitar esto.

Funciones

- El objetivo de las funciones es el de modularizar dentro del mismo programa, creando una secuencia de instrucciones que fácilmente se podrá reutilizar las veces necesarias de manera rápida y eficiente. Se pensó como un tipo de dato nuevo que varía dependiendo del resultado de las instrucciones que contiene.
- Permite crear declarar funciones como las conocidas en la matemática.
    - Ejemplo: `f(x) = x * x`
- Las funciones se definen con la palabra clave function de la siguiente manera:

```
function function_name(number param_name) {
     //Valid code for our language
};
```

- Estás luego se llaman de la forma `function_name(expression)` donde expression debe resolver a un valor numérico. Se pueden encadenar llamados a funciones.
- Solo está permitido el uso de un argumento y este debe ser numérico.

Programas de prueba

En el GIT del proyecto, dentro del directorio /test, se pueden encontrar programas de prueba. En ellos se pueden probar las distintas funcionalidades ofrecidas por el compilador. Para ejecutarlos, se puede utilizar el bash script RunScript.sh. Los archivos generados de código C se guardarán en la carpeta *generatedCode* y los archivos ejecutables en la carpeta *executableTests.*

Dificultades encontradas en el desarrollo del TP

Una de las dificultades halladas en el desarrollo fue cómo almacenar las variables y mantener un registro de cuales fueron declaradas, al igual de como chequear el tipo de dato de una variable para evitar la definición errónea de la misma. Para resolver esto, se agrego un atributo más a las estructura de los tokens llamado DataType. El mismo se asignaba al declarar una variable. Al realizar operaciones y asignaciones, se chequeaba que este atributo coincidiera y si no, se seteaba este atributo en DATA_NULL. Luego de la creación de cada Token, se realiza un chequeo que verifique este atributo sea distinto que DATA_NULL, y si lo era, se mandaba el error adecuado con yyerror. Para el almacenamiento de las variables, se creo un array que almacena punteros a los VariableToken y de tal manera, poder acceder a ellas con su nombre.

Otro desafío hallado fue al comienzo del desarrollo y se debió a una falta de organización en el flujo de trabajo. Se codeo la mayoría sin testeo en el medio y cuando llegó la hora de empezar a probar, surgieron varios errores que resultó en cambios drásticos en el código.

Se deseaba implementar varias nuevas características, sin embargo no fueron posibles por falta de tiempo. Algunas de estas son arreglos, operaciones simples para

realizar la suma de arreglos o el producto entre ellos, coordenadas y operaciones para hallar la pendiente dado dos coordenadas.

Futuras extensiones (breve descripción de la complejidad de cada una)

Para una siguiente versión, el lenguaje podría incorporar las siguientes funcionalidades:

- Arreglos: Colección de datos de un mismo tipo, especificamente tipos numéricos para realizar operaciones matemáticas con ellos (suma, resta, producto, etc).
  - Posibles usos:
    - `sum of [1, 2, 4]` → resultado sería 7
    - `[1, 3, 6] + [3, 5, 2]` → resultado sería [4, 8, 8]
- Coordinates: Un tipo de dato que contenga dos valores. Un valor que representa la 'x' y otro que representa la 'y' en un eje cartesiano. Para su implementación habría que hacerse cambios en la gramática y que internamente almacena dos variables de tipo number.
- Slope: Una función para calcular la diferencia entre dos coordenadas (coordinates). Esta función recibirá 2 coordinates y calcular la diferencia entre puntos cartesianos.

  ```
  Number X = slope(coord1,coord2);
  ```
- For: Implementar el bloque for como alternativa al calculate-while.
- Switch: Implementar el bloque for como alternativa al if para evitar concatenar tantos elif (Este sería uno de los cambios más difíciles, ya que la gramática del switch tiene más alternativas que el if).
- Vectores: Ideal para la suma, resta, entre otras cosas.
- Comentarios: Incorporar la posibilidad de agregar comentarios, este cambio implicaría un cambio en toda la gramática ya que habría que agregarle los comentarios a todas las líneas (incluyendo expresiones y variables).
- Funciones: Se definió la gramática con un solo argumento numérico para la definición de una función. Se podría haber armado para que soporte mayor cantidad de argumentos pero se dejó en uno por simplicidad de código.

Referencias

- [Tutorial on lex and yacc - Video] https://www.youtube.com/watch?v=54bo1qaHAfk .