Instituto Tecnológico de Buenos Aires

Facultad de Ingeniería

Texler

Diseño e implementación de un lenguaje

Autor:

Zahnd, Martín E. (60401)

mzahnd@itba.edu.ar

Co-autores:

Larroudé Álvarez, Santiago Andrés (60460)

Hinojo, Nicole (57440)

Borracci, Ángeles (56648)

Junio 2022

Índice

1. Introducción

Texler es un lenguaje diseñado para procesar texto y ser utilizado como herramienta de reporte de información. Permite convertir archivos de texto en lo pedido por el usuario. Por ejemplo, dado un CSV, el usuario puede diseñar como salida una tabla utilizando caracteres ASCII con el contenido del archivo.

2. Consideraciones adicionales

El lenguaje Texler incluye tipos de datos de números enteros, o decimales o cadenas de caracteres (*strings*), sumado a eso, se flexibiliza el esquema fuertemente tipado del lenguaje de programación C utilizando un tipado dinámico, lo cual genera un enfoque más abstracto y simple, por lo que no tendrán que preocuparse por los tipos de datos y sus variables.

Es un lenguaje diseñado con un paradigma funcional, donde los condicionales fuerzan la cláusula *else* y los archivos de texto se recorren utilizando un ciclo *for-each*. Además, la gramática está diseñada pensando en el uso extensivo de las funciones.

El lenguaje Texler genera código C como salida al compilar, que luego es compilado con GCC o Clang para obtener el archivo binario ejecutable ELF. Para eliminar la noción de la biblioteca estándar de C, nuestro lenguaje incorpora funciones built-in para manipulación de cadenas de caracteres o imprimir en salida estándar.

3. Descripción de la gramática

La gramática está diseñada con el objetivo de desarrollar un lenguaje con paradigma funcional. En consecuencia, es obligatorio declarar al menos una función en la cual se ejecutará el código Texler. Toda función puede recibir ninguno o múltiples parámetros, y devolver un identificador (declarado dentro de la misma) o nada.

Al igual que los párrafos en español, las expresiones de Texler terminan en un punto y aparte ('.' seguido de un salto de línea).

Una declaración de variable imita al lenguaje natural, debiéndose escribir como: `let <variable_name> be <initial_value>.`

Su asignación, por otra parte, es más breve para obtener código menos cargado de información para quien lo lee: `<value> -> <variable_name>`.

El siguiente caso ejemplifica lo que acabamos de mencionar inicializando la variable `abc` con "`True`z luego cambiando su valor a `False`. Notar que la función `r30` no recibe argumentos ni devuelve una variable.

```
function r30()
    let abc be True.

    False -> abc.

    return.
end
```

Código 1:

El manejo de archivos requiere de dos acciones por parte del desarrollador: declarar el archivo con el cual se desea trabajar y establecer sus separadores (si corresponde), e indicar a Texler en qué archivo desea realizar determinadas acciones.

Para declarar el archivo, debemos escribir la expresión

`File "path_to_file" as <file_variable_name>`.

En cuyo caso Texler comprende que en "path_to_file"se encuentra un archivo que deseamos llamar `<file_variable_name>` durante la ejecución de nuestro programa. En el caso de que nuestro archivo no tuviera los separadores básicos (el espacio,

las tabulaciones y los saltos de línea), es posible declarar los utilizando la palabra reservada `with` y una lista de los separadores.

Por ejemplo, un archivo que se desea separar en columnas cada vez que se encuentra un caracter ',' o ':' se declararía como:

`File "path_to_file" with [":", ","] as <file_variable_name>.`

Debemos saber que es posible establecer la salida estándar como .ªrchivo.en el cual se imprimirá: `File STDOUT as output.`

A continuación, cuando precisamos trabajar con el archivo que declaramos al comienzo de nuestra función utilizamos `with <file_variable_name>:`

Por ejemplo, el siguiente programa filtra todas las líneas que contienen error en los archivos dentro de una carpeta:

```
function r322()
    File "path" with [","] as input.
    File "" as output.

    # Líneas 1, 2 y 3 (el rango es inclusivo)
    with input: for line in lines().byIndex([1..3]) do
                    line -> output. # End assignment
                . # End loop
    . # End 'with'

    return output.
end
```

Código 2:

Nótese la gramática para los ciclos en Texler:

`for <variable> in <function> do \textit{expression}.`

Donde `<variable>` tiene alcance local, dentro del ciclo, y `lines()` es una función propia del lenguaje que devuelve todas las líneas de un archivo.

También podemos observar que es posible crear listas con rango numérico `[from..to]`.

Por otra parte, el lenguaje permite trabajar con *strings* concatenándolos:

`string1 ++ string2`, o repitiéndolos un número finito de veces: `string1 * <number>`.

Las operaciones aritméticas básicas, suma, resta, multiplicación, división y módulo, utilizando números adhiere al estilo usual en los lenguajes de programación.

Por último, los condicionales tienen la forma

`if <boolean> then <expression1> else <expression2> .`

Donde ambas `<expression>` puden ser una expresión cualquiera pero `<boolean>` debe ser una comparación o cualquier función o identificador que se pueda interpretar como un tipo de dato booleano.

4. Desarrollo del proyecto

La gramática del lenguaje fue escrita en dos oportunidades, la primera de ellas presentada a mediados de cuatrimestre, era frágil y fallaba consistentemente, además de estar incompleta.

A partir de las correcciones y recomendaciones del docente, se implementó una segunda versión de la misma que es más consistente y pasa todos los test automatizados del lenguaje.[1]

Una vez definida la gramática del lenguaje, en su segunda versión, se armó un escáner con Flex para ser utilizado como tokenizador y fue programada utilizando Bison.

A continuación se intentó implementar el árbol de sintaxis, obteniendo pocos resultados en este proceso.

[1] Véase el commit 583a6386889092afce7126909abb065ff0ad3f91 del proyecto

5. Dificultades encontradas

Habiendo avanzado en las clases teóricas de la materia, el rediseño de la gramática fue un proceso simple que se vio asistido por la implementación de un conjunto de test previo a su implementación en Bison.

Por otra parte, al momento de implementar el árbol y realizar la transpilación de código al lenguaje C se presentaron numerosas dificultades: el manejo del tiempo; dificultad para comprender la tarea a realizar; y una gran cantidad de errores acumulados en el código fuente del mismo que fue escrito, inicialmente, sin ser probado a medida que se iba desarrollando.

Estos factores, sumados a la descordinación de los tiempos del grupo para poder realizar el trabajo en conjunto, conllevaron a la imposibilidad de finalzar adecuadamente el proyecto.

6. Futuras extensiones

Mencionamos las funcionalidades las cuales nuestro compilador no aceptaba correctamente en cada testeo:

- Poder desarrollar una lectura en distintos tipos de números (enteros, decimales, puntos flotantes, etc.) y realizar operaciones con los mismos.
- El desarrollo de un programa donde se pueda realizar un ciclado con elementos obtenidos en un archivo de entrada (test r32).
- Filtrado de lineas de un archivo según algún predicado (test r33).
- Permitir el copiado de el contenido de un archivo una cantidad ingresada por el usuario (test r36).
- Permitir que un programa separe cada columna de un archivo de entrada CSV/TSV en lineas individuales (test r37).
- Poder agrupar 5 lineas de entrada en una salida separadas por comas (test r38).
- Poder aplicar una transformación diferente sobre cada columna de entrada (test r39).
- Buscar un string (ERROR) en todas los archivos de una carpeta (test r310).

Y ahora pasamos a enumerar los puntos donde nuestro compilador no debería aceptar los siguientes programas:

- Leer un archivo que no existe (test r311).
- Aplicar una transformación sobre una columna que no existe en un CSV (test r312).
- Realizar una operación entre tipos incompatibles de datos (test r314).
- Acceder a una posición de un string fuera de rango (test r315).

Continuamos con algunos puntos donde nos hubiese gustado continuar al haber cumplido con todo lo ya mencionado anteriormente:

- Permitir insertar código C inline (análogo a como C permite Assembly inline), para aquellos programadores que posean conocimiento del lenguaje C y deseen inyectar fragmentos de código C en su programa.
- Poder extender el lenguaje en tiempo de ejecución, agregando funciones personalizadas al set de funciones built-in de nuestro lenguaje o incorporar sets enteros personalizados de comportamiento como una especie de módulos de funciones personalizadas.
- Incorporar otra nueva estructura de datos tipo Tablas de Hash.
- Poder incluir código de nuestro lenguaje de otros archivos o librerías externas (como los `#include ``file''` o `#include <lib>` del lenguaje de programación C).
- Tener la capacidad de definir funciones e invocarlas desde otras al igual que el lenguaje de programación C.

Referencias

[1] Anthony A. Aaby, "Compiler Construction using Flex and Bison," https://dlsiis.fi.upm.es/traductores/Software/Flex-Bison.pdf, September 2003.

[2] Libros provistos por la cátedra.