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 con contenido alfanumérico el usuario puede obtener un archivo de texto con cada columna del archivo original en una línea nueva y con todos los campos numéricos con su valor duplicado.

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), lo cual evita la escritura de elipsis pues esta tiene otro significado. [1]

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

Su asignación se realiza de manera breve y directa: `<value> -> <variable_name>`.

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

```
1  function r30()
2      let abc be True.
3
4      False -> abc.
5
6      return.
7  end
```

Código 1:

El manejo de archivos requiere de dos acciones por parte del desarrollador: declarar el archivo con el cual se desea trabajar estableciendo sus separadores, e indicar a Texler en qué archivo desean realizar las transformaciones.

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.

[1] Utilizar un punto final en las expresiones es una idea basada en la gramática del lenguaje funcional Erlang.

Si quisiera analizarse una carpeta con archivos de texto plano, "path_to_file" debe terminar con una barra:

`File "path_to_file/" as <file_variable_name>.`

Por otra parte, en el caso de que nuestro archivo no tuviera los separadores por defecto (el espacio y la coma), es posible declararlos 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 carácter ',' o ':' se declararía como:

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

Notemos que los separadores deben ser una lista de al menos un elemento, y cada uno de ellos con único caracter: `":"` es válido, mientras que `".."` no lo es.

Para diferenciar un archivo que debe ser leído (de entrada) del archivo que se desea escribir (el de salida) los mismos deben nombrarse con el prefijo `input` y `output`, respectivamente.

A su vez, el nombre del archivo de salida puede ser:

- `File "" as output.`: Un string vació, en cuyo caso se debe devolver la variable que lo contiene al finalizar la función.
- `File STDOUT as output.`: La salida estándar.
- `File "archivo_transformado.txt" as output.`: El nombre y extensión del archivo que se desea obtener.

El siguiente programa filtra todas las líneas que contienen error en los archivos dentro de una carpeta y las escribe en el archivo "output_r310.txt":

```
function r310()
    File "r310_folder/" with [","] as input.
    File "output_r310.txt" as output.

    with input: for line in lines() do
                    line.filter("ERROR") -> output. # Fin expresión
                . # Fin ciclo
    . # Fin 'with'

    return. # No devuelve nada
end
```

Código 2:

Nótese la gramática para los ciclos en Texler:

`for <variable> in <function> do <expression>.`

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

Las funciones estándar implementadas que se pueden utilizar en un ciclo son:

- `lines()`: Obtiene cada línea del archivo. No recibe parámetros.
- `columns()`: Obtiene cada columna en una línea del archivo, por lo que debe ser utilizada junto a `lines()` concatenadas del siguiente modo: `columns().lines()`.

  De esta manera se entiende que se está ciclando 'en cada columna de cada línea'. Tampoco recibe parámetros.
- `byIndex()`: Se concatena al final de alguna de las funciones previamente mencionadas y puede recibir dos tipos de parámetros: una lista con un rango, explicada más adelante, o una constante.

También contamos con las siguientes funciones para aplicar transformaciones o obtener información de las variables:

- `toString(variable)`: Devuelve en un string (char *) el contenido de la variable, ya sea una constante numérica o no.

- `at(variable, posición)`: Devuelve el carácter que se encuentra en la posición dentro del string de la variable (variable tiene que ser un string).

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.

En cuanto a las listas, las hay de dos clases:

Listas con rango numérico `[from..to]`, donde ambos valores son inclusivos y de elementos o variables `["string", variable]`, que son idénticas a una lista en cualquier lenguaje de tipado dinámico.

Por último, los condicionales tienen la forma

`if <boolean> then <expression> else <expression> .`

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.

Resta explicar un detalle de la gramática: ¿cómo sabe Texler sobre qué archivo debe realizar una acción?

Simple: con la palabra clave `with`. El ejemplo 2 es una clara demostración de su uso: estamos indicando al compilador que deseamos leer las líneas de los archivos que se encuentran dentro de la carpeta `r310_folder/`.

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 la primera versión de los test automatizados del lenguaje. [2]

En esta segunda oportunidad, la gramática se obtuvo escribiendo primero los posibles casos de uso e implementando los mismos como tests para poder corroborar constantemente que la misma se esté escribiendo de manera correcta. Luego se desarrolló un escáner con Flex para ser utilizado como tokenizador y fue programada utilizando Bison.

Luego de esto, se implemento el árbol de sintaxis, el cual dificultó el proceso del trabajo ya que los "nodos de expresión" (`node_expression`) contienen una gran cantidad de información y pueden ser utilizados de varias maneras. Al comienzo creímos que podía ser de gran ayuda realizarlo de esta forma porque no necesitaríamos de una gran cantidad de estructuras a la hora de realizar el árbol, pero al adentrarnos en el trabajo notamos que la otra estrategia hubiera resultado en código más simple y fácil de mantener.

Al tener desarrollada la gramática y haber obtenido el árbol de sintaxis, realizamos el proceso inverso con el mismo: desarmarlo para traducir nuestro lenguaje a lenguaje C. Para lograrlo, primero desarrollamos los mismos ejemplos y casos de uso que se aprovecharon en la implementación de la gramática, completamente en el lenguaje C, de manera que pudiéramos probar y comprender el funcionamiento y sintaxis del generador de código intermedio, y luego implementamos este último, nuevamente aprovechando los test automatizados y comparando con nuestra versión propia escrita en C.

Es importante destacar que el código intermedio generado por Texler es difícil de leer pues carece de saltos de línea y comentarios. Para facilitar esta tarea, se puede utilizar la herramienta de formato de Clang: ClangFormat con el archivo de formato que se encuentra en la raíz del proyecto. Los detalles se encuentran en el README

[2] Véase el commit 583a6386889092afce7126909abb065ff0ad3f91 del proyecto.

del proyecto, junto a las instrucciones de compilación.

5. Dificultades encontradas

Si bien al comienzo tuvimos dificultades para diseñar la gramática, habiendo avanzado en las clases teóricas de la materia, el rediseño de la misma fue un proceso simple que se vio asistido por la implementación de un conjunto de test previo a su implementación en Bison, como se mencionó en una sección anterior.

La elección de un lenguaje con tipado dinámico también resultó ser errónea, pues la cantidad de consideraciones que se deben tener en cuenta al momento de transpilar el código es grande y difícil de implementar en un único cuatrimestre. Fue una apuesta interesante, de la cual ciertamente aprendimos, mas el tiempo y conocimientos necesarios para su correcta y completa implementación son mayores a los que se pueden obtener en un primer curso de compiladores.

Otra de las dificultades fue la implementación elegida a la hora de guardar la información y estructura de nuestro lenguaje (la estructura de los nodos del árbol) ya que el nodo mas importante donde guardamos la información (`node_expression`) es el que mas aparece en todas las estructuras de la gramática (por como fue pensada inicialmente) y en algunos casos no era claro cómo o de qué manera guardar la información. Si tuviéramos que rediseñar la gramática u otro lenguaje, este sería el primer aspecto que consideraríamos cambiar.

Una vez finalizada esta primera etapa del trabajo, otra de las dificultades fue al momento de hacer el proceso inverso al explicado, la transpilación de la información dentro de los nodos del árbol. Esto se debió principalmente a la elección de un tipo de nodo que contuviera al resto: cuando entrabamos en un `node_expression` teníamos muchos caminos posibles, lo cual es sumamente susceptible a errores. Cada vez que estos ocurrían, debíamos debuggear el código ya que en la mayoría de los casos no se encontraba dónde estaba el error sin ayuda de GDB, lo cual nos quitaba tiempo de avances en el resto del proyecto.

Además de esto, inicialmente no habíamos generado algún archivo con la estructura y la forma de realizar los test propuestos por la cátedra: una especie de borrador en lenguaje C, el cual podíamos realizar pruebas y testear los resultados de manera ágil. Afortunadamente notamos este error al poco tiempo y lo revertimos. Con esto ya realizado, la tarea mencionada fue realizada con una mayor rapidez.

Por otra parte, tuvimos bastantes dificultades a lo largo de la realización del

trabajo, una de ellas fue la organización y distribución de las tareas dentro del grupo; luego tuvimos dificultades a la hora de comprender la consigna en algunos aspectos o de poder implementarla, principalmente porque no teníamos claros algunos aspectos teóricos o la explicación de los mismos se realizaba más adelante en el cuatrimestre.

Por todo lo mencionado anteriormente, la finalización del trabajo no fue posible en el tiempo pautado por la cátedra.

6. Futuras extensiones

Las futuras extensiones que encontramos para Texler se basan en las limitaciones de esta primera implementación:

- 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.
- Actualmente Texler acepta una única función por archivo, esta sería una extensión importante del lenguaje. Si bien buena parte del código para implementarla está ya escrito, no se realizaron pruebas ni mucho menos consideraron los casos borde de realizar esto.
- 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.
- Extender y mejorar el manejo de errores dentro de la implementación en el código generado.
- Incorporar otra nueva estructura de datos tipo Tablas de Hash.
- Permitir que los ciclos reciban más de una expresión luego de `do`, que en esta versión se encuentra limitada a una única.
- Considerar reemplazar el punto y aparte por otro símbolo o ningún caracter visible (similar a Python).
- 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.