ITBA - "72.39 - Autómatas, Teoría de Lenguajes y Compiladores"

Trabajo Práctico Especial - 'Diseño e implementación de un lenguaje'

Alumnos:

De Luca, Juan Manuel	60103
Konfederak, Sol	60255
Dizenhaus, Manuel	61101
Cornídez, Milagros	61432

1 Tabla de contenidos

Contents

2 Introducción

Para la materia *"Autómatas, Teoría de Lenguajes y Compiladores"* se requirió diseñar, analizar, e implementar un lenguaje a elección nuestra. El lenguaje debía encontrarse completamente definido y, a su vez, aportar valor en cuanto a su funcionalidad. Las tecnologías utilizadas fueron `flex` y `bison`.
El lenguaje escogido busca solventar la barrera de acceso que propone la compleja sintaxis de programación HTML a nuevos programadores. El uso de tags, atributos, referencias, como también de elementos como los `div` (incluyendo lo que concierne su posicionamiento, apariencia, etc.) puede resultar dificultoso para un usuario nuevo.
Este lenguaje provee una capa de abstracción sobre esto, dejando al usuario con una sintaxis mucho mas clara para trabajar, tarea que resultaría profundamente mas ardua e dificultosa si se tuviera que aprender de 0 a generar el esqueleto de un sitio web.

3 Gramática del lenguaje

La gramática del lenguaje esta definida de la siguiente manera:

$$G =< V_{NT}, V_T, P, program >$$

donde los elementos de V_{NT} se definen de la siguiente manera:

$$V_{NT} = \{program, web\_expressions, web\_expression, title\_expression,$$
$$text\_expression, img\_expression, link\_expression, table\_expression,$$
$$div\_expression, link\_attrs, link\_attr, div\_attrs,$$
$$title\_attrs, table\_resolve, row\_resolve, row\_cell\_union,$$
$$row\_data, data\_attrs, data\_attr\}$$

Los elementos de V_T son los siguientes:

$$V_T = \{BOX, ENDBOX, START, END, TITLE, TITLE\_SIZE,$$
$$COLOR, POSITION, DEF\_DELIMITER, COMMA,$$
$$TABLE, ENDTABLE, LINK, RANDOM, IMAGE$$
$$HYPERLINK, ID, IDREF, SOURCE, BOLD, ITALIC,$$
$$UNDERLINED, NUMBER, ROW, ENDROW, DATA$$
$$TIMES, TEXT, CONTENT\}$$

Y el conjunto de producciones esta conformado por:

$P = \{program \rightarrow START\ web\_expressions\ END,$
$web\_expressions \rightarrow web\_expression \mid web\_expressions\ web\_expression,$
$web\_expression \rightarrow title\_expression \mid text\_expression \mid img\_expression \mid link\_expression$
$\mid table\_expression \mid div\_expression,$
$img\_expression \rightarrow IMAGE\ SOURCE\ DEF\_DELIMITER\ CONTENT \mid$
$IMAGE\ SOURCE\ IDREF\ DEF\_DELIMITER CONTENT \mid$
$IMAGE\ IDREF\ SOURCE\ DEF\_DELIMITER CONTENT$
$link\_expression \rightarrow HYPERLINK\ SOURCE\ link\_attrs\ DEF\_DELIMITER\ CONTENT \mid$
$HYPERLINK\ SOURCE\ DEF\_DELIMITER\ CONTENT \mid$
$HYPERLINK\ IDREF\ link\_attrs\ DEF\_DELIMITER\ CONTENT \mid$
$HYPERLINK\ IDREF\ DEF\_DELIMITER\ CONTENT,$
$link\_attrs \rightarrow link\_attr \mid link\_attrs\ COMMA\ link\_attr,$
$link\_attr \rightarrow BOLD \mid ITALIC \mid UNDERLINED \mid COLOR,$
$div\_expression \rightarrow BOX\ div\_attrs\ DEF\_DELIMITER\ web\_expression\ ENDBOX \mid$
$BOX\ DEF\_DELIMITER\ web\_expression\ ENDBOX \mid$
$BOX\ div\_attrs\ DEF\_DELIMITER\ web\_expressions\ ENDBOX \mid$
$BOX\ DEF\_DELIMITER\ web\_expression\ ENDBOX\ ,$
$div\_attrs \rightarrow POSITION \mid ID\ POSITION,$
$text\_expression \rightarrow TEXT\ data\_attrs\ DEF\_DELIMITER\ CONTENT \mid$
$TEXT\ ID\ data\_attrs\ DEF\_DELIMITER\ CONTENT \mid$
$TEXT\ DEF\_DELIMITER\ CONTENT,$
$title\_expression \rightarrow TITLE\ DEF\_DELIMITER\ CONTENT \mid$
$TITLE\ title\_attrs\ DEF\_DELIMITER\ CONTENT \mid$
$TITLE\ DEF\_DELIMITER\ LINK \mid$
$TITLE\ title\_attrs\ DEF\_DELIMITER\ LINK,$
$title\_attrs \rightarrow data\_attrs \mid TITLE\_SIZE\ data\_attrs \mid ID\ data\_attrs \mid TITLE\_SIZE \mid ID,$
$table\_expression \rightarrow TABLE\ DEF\_DELIMITER\ NUMBER\ TIMES\ NUMBER\ table\_resolve\ ENDTABLE,$
$table\_resolve \rightarrow row\_resolve \mid table\_resolve\ row\_resolve,$
$row\_resolve \rightarrow ROW\ DEF\_DELIMITER\ row\_cell\_union\ ENDROW,$
$row\_cell\_union \rightarrow row\_data \mid row\_data\ row\_data \mid row\_cell\_union\ row\_data,$
$row\_data \rightarrow DATA\ DEF\_DELIMITER\ CONTENT \mid DATA\ data\_attrs\ DEF\_DELIMITER\ CONTENT,$
$data\_attrs \rightarrow data\_attr \mid data\_attr\ COMMA\ data\_attrs,$
$data\_attr \rightarrow COLOR \mid POSITION \mid BOLD \mid ITALIC \mid UNDERLINED$
$\}$

4 Desarrollo del proyecto y de fases del compilador

4.1 Frontend

Durante el transcurso de la primer entrega, el trabajo progresó de manera mas lenta a pesar de ser la porción del trabajo mas "liviana". Esto fue producto del desconocimiento de todos sobre lenguajes, y lo que parecia como una tarea Herculeana de "crear nuestro propio lenguaje". Luego de analizar diferentes posibilidades de lenguajes, yendo de ideas mas triviales hacia mas elaboradas, finalmente nos decidimos por realizar esta simplificación de HTML para ayudar a quienes decidían incursionar en el mundo del desarrollo web. Creímos que esto le daba valor a nuestro proyecto, mas allá de ser un trabajo práctico.
Comenzando a desarrollar el trabajo, lo primero que hicimos fue un estudio del compilador inicial que se nos otorgaba. Este ejemplo resultó útil para ver el flujo de ejecución de un programa esperado (aún sin backend). Una vez que logramos comprender la dinámica de lo requerido, progresamos a la construcción de los tokens y la gramática.
Nuestra primer gramática conllevo diversas falencias, como por ejemplo el *hardcodeo* del orden de los atributos, entre otras cosas. La cantidad de producciones era abismal, y virtualmente imposible para trabajar, además del hecho de que no aportaba ningún valor tener una producción definida para, por ejemplo:

```
...
title bold, small: ...
...
```

como también para:

```
...
title small, bold: ...
...
```

4.2 Backend

Continuando con la segunda entrega del trabajo práctico, se debió implementar todo lo que conforma el backend del mismo. Esto llevó un trabajo mas demandante, considerando que ahora se debia generar el *Abstract Syntax Tree*, con todos los tipos de nodo necesarios para satisfacer a todas las producciones, y así poder reconstruir el mismo desde el generador de código.
En primer lugar, siguiendo la crítica realizada para la primer entrega, se intentó compactar la gramática en menos reglas de producción[1]dado que creíamos que varias eran redundantes y no aportaban nada mas que una sobrecarga de análisis.
Al intentar construir el arbol, hubo un proceso de prueba y error reiterativo dado que el uso de un tipo de nodo conllevaba conexiones a siguientes. De esta manera, fue necesario ir considerando "a futuro" la conveniencia de ciertas estructuras (esto fue gran parte de las limitaciones del trabajo).
Posterior a esto, hubo que realizar ciertas modificaciones a la primer entrega. Por un lado, la definición de los tokens en un `ENUM` del lado de flex ya no era útil, dado que el TokenID dejó de tener validez una vez que definimos los tipos de dato bajo la directiva `\%union`. Entonces se debió remover esa porción de código, y reemplazar los tipos `TokenID` que se devolvían en el archivo `flex-patterns.l` por `unsigned int`.
Una vez que realizamos todo esto, procedimos al trabajo de analizar el arbol desde su raíz. Recordando que `yylval` cuenta con un campo *result*, allí es donde almacenamos la raíz y comenzamos el desglose del árbol. Todo el análisis del arbol se encuentra en el archivo `generator.c`.

[1]Creemos que esta fue una fuente de problemas, mas allá que no logramos identificar exactamente donde nacía el problema, las modificaciones realizadas llevaron a distintos problemas.

5 Dificultades a la hora de desarrollar el proyecto

El trabajo propuesto llevó variadas complicaciones durante su desarrollo. Desde la partida, sabíamos que tratabamos con dos tecnologías que nos eran totalmente novedosas como son `flex` y `bison`. Estas herramientas de GNU tienen una sintaxis particular, y de por sí eso conllevo una barrera a nivel comprensión y posterior elaboración de código.
A su vez, al utilizar el *template* provisto por la catedra del lenguaje básico diseñado para la calculadora, cuando intentamos retratar ciertos fragmentos que se encontraban allí implementados, nos daban errores. En gran parte, esto se debía, por ejemplo, a la falta de comprensión de `flex`. Nosotros observamos el ejemplo donde, en el archivo `"flex-actions.c"`, capturaba el valor del entero que necesitaba de la siguiente manera:

```
TokenID IntegerPatternAction(const char * lexeme) {
    LogDebug("IntegerPatternAction: '%s'.", lexeme);
    yylval = atoi(lexeme);
    return INTEGER;
}
```

Utilizando la variable `"yylval"`, le permitía almacenar el valor del entero a capturar de manera sencilla. Al intentar emularlo pero con strings, el programa logicamente fallaba. Investigando un poco el código y consultando con algunos de nuestros compañeros, logramos notar que en el archivo `"shared.h"` se identifica a la variable de tipo int, y no se trataba de una variable genérica que permitía almacenar el lexema que recibiamos. Esto derivó en dos cuestiones centrales: Primero, ¿como capturabamos strings, si yylval era de tipo entero? Y segundo, como consecuencia de lo primero, ¿como podía un lenguaje capturar distintos tipos de dato? Nuestro lenguaje requería capturar en los tokens tipos `string` como tipos `int`.
Luego de consultar con la catedra, descubrimos que la declaración de yylval quedaría obsoleta. Esto se debía a que la implementación base unicamente necesitaba enteros como variables, y es por eso que ni siquiera utilizaba una instrucción de tipo `int`. La solución fue la utilización de la directiva `%union`, cuyo funcionamiento permitía la declaración de diferentes tipos de dato (la directiva se encuentra en el archivo `bison-grammar.y`, en la carpeta *src/frontend/syntactic-analysis*). De esta manera, pudimos capturar tanto variables de tipo `char *` como de tipo `int`.
Otra dificultad que tuvimos a la hora de programar este trabajo fue considerar en paralelo la creación de los nodos para el *AST* como para el posterior análisis de los mismos. En reiterados casos, nos encontramos con obstaculos a la hora de "desarmar" los nodos construidos para el AST y procesarlos. El principal foco de conflictos se pudo notar con los atributos, dado que en muchos casos resultaba en un para nada deseable *segmentation fault*.

6 Futuras extensiones y/o modificaciones

Creemos que este lenguaje tiene muchísimo potencial. HTML es un lenguaje que tiene miles de componentes diferentes, como también distintas formas de utilizarse. En este trabajo, se abarcaron las estructuras principales de HTML como *div, header, p, link, img* pero queda mas que en evidencia que el potencial de este lenguaje aún queda por ser explotado. Pensando en algunos posibles tags[2], sería interesante la implementación de algunos como *input, video, object, li* entre muchos otros.
Otro aspecto que resulta intrigante para analizar es la implementación de `CSS` a este lenguaje. Permitir la definición de clases en CSS, y poder definir atributos desde las mismas, le daría otra capa de profundidad con muchísimas mas posibilidades a nivel estético.
Inclusive, si el proyecto avanza lo suficiente, se podría proponer el uso de *Javascript* junto a estas dos tecnologías. De cualquier manera, creemos que la sintaxis necesaria para implementar este tercer pilar requeriría practicamente un lenguaje nuevo, dado que contiene elementos de sintaxis totalmente diferentes al implementado.

[2]Todos los tags pueden ser analizados en este enlace

7 Referencias y bibliografía

[1] Levine, J. R. *Flex & Bison*, (1.a ed.). O'Reilly Media, Inc

[2] Golmar, M. Agustín, *Flex-Bison Compiler*, 2022. Link (Sujeto a que se mantenga en caracter público)