Instituto Tecnológico de Buenos Aires

Proyecto Especial

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

G - 79

Integrantes:

Nacuzzi, Gianna Lucia - 64006

Squillari, Maria Agostina - 64047

Frutos, Pilar - 64225

Docentes:

Mario Agustín Golmar

Manuel Ader

Franco Morroni

1. Introducción

Este trabajo consiste en el desarrollo de un compilador que toma como entrada un DSL propio y genera como salida una página en HTML con estilos CSS embebidos. La idea del proyecto fue recorrer todas las etapas clave de construcción de un compilador a través de las tres entregas: desde el diseño del lenguaje, pasando por el frontend (donde se hizo el análisis léxico y sintáctico), hasta llegar al backend (donde se validó el contenido y se generó el código final).

El compilador fue desarrollado en el lenguaje de programación C. Para el análisis léxico se utilizó la herramienta Flex, encargada de identificar los distintos componentes léxicos definidos en el DSL, como palabras clave, identificadores, símbolos y comentarios. Para el análisis sintáctico se empleó Bison, que permitió construir el parser a partir de una gramática libre de contexto, validando que la estructura del programa escrito en el DSL respete las reglas del lenguaje. Además, en esta etapa se generaron las estructuras internas necesarias, como el Árbol de Sintaxis Abstracta (AST), que luego es utilizado por el backend para generar el código HTML con estilos CSS embebidos.

A lo largo del informe se explican las decisiones que se fueron tomando en cada etapa del proyecto: la definición del dominio y el lenguaje, cómo está armado el compilador por dentro, las dificultades encontradas durante el desarrollo, y las posibles mejoras para implementar a futuro.

2. Modelo Computacional

2.1 Dominio

El dominio de este proyecto se centra en la generación de contenido web estático a partir de un lenguaje específico de dominio (DSL) propio. Este lenguaje está orientado a facilitar la creación de páginas HTML con estructuras comunes como títulos, párrafos, listas, botones, formularios, tarjetas y otros bloques visuales, sin necesidad de escribir directamente en HTML o CSS.

Esta elección de dominio influye completamente en cómo funciona el lenguaje diseñado. Cada cosa que se escribe en el DSL representa un bloque visual en el HTML. Por ejemplo, cuando se pone un header1, eso implica un título principal, de primer nivel. Y si se utiliza un form, se espera que aparezca un formulario.

En definitiva, el DSL actúa como un puente entre el contenido de alto nivel y su representación en HTML, brindando un entorno más ágil para la creación de páginas web estáticas.

2.2 Lenguaje

El lenguaje de entrada es de tipo específico de dominio (DSL) que describe de forma declarativa componentes de una página web usando bloques análogos a etiquetas, estilos en línea y contenido estructurado. Cada constructor se introduce con una marca @ seguida de un identificador y puede incluir parámetros entre paréntesis. Los estilos se insertan entre llaves. Véase el ejemplo a continuación:

```
@define simpleButton(text)
     @button     {     padding:10px;     color:white;
background:blue; }
    {{text}}
     @end
@enddefine
```

En este fragmento, @define simpleButton(text) declara una componente llamada simpleButton con un parámetro text. Dentro del cuerpo, @button { … } … @end genera un elemento <button> al que se aplica el estilo CSS dado. La variable {{text}} se sustituirá por el valor real cuando se invoque el widget. Para hacer uso del componente se declara:

```
@use simpleButton("Enviar formulario")
```

Por otra parte se pueden definir otros elementos más clásicos de html, como los títulos (h1, h2 y h3) y los párrafos (p):

```
# "Titulo 1"
## “Titulo 2”
### “Titulo 3”
"Este es un párrafo sencillo"
```

Para listas se soportan tanto ordenas como desordenadas, declarandose de la siguiente manera:

```
@list
1. "Primero"
2. "Segundo"
3. "Tercero"
@end

@list
- "Elemento A"
```

```
- "Elemento B"
@end
```

También, se pueden insertar otros elementos HTML como formularios, navbar, imágenes, tablas, cards y pies de página. A continuación se adjuntan a modo de ejemplo la definición de cada una:

```
@form [ action: /api/submit;  method: POST; ]
    @item('Name', 'Your name')
    @item('Email', 'your@email.com')
@end

@nav
    @item('Home', 'home.html')
    @item('Service', 'services.html')
    @item('Contact', 'contact.html')
@end

@img('https://example.com/imagen.jpg',       'Example
image')

@table
| "Name "     | "Age" | "City" |
| "Juan"        | "25"   | "Buenos Aires"|
| "Maria"       | "30"   | "Córdoba"     |
@end

@card { background: red; border: rounded; }
#"title"
"text"
@end

@footer { background: red; color: grey; }
#"My Website. All rights reserved."
@end
```

Finalmente, se admiten bloques de diseño más avanzado:

```
@row
```

```
    @column { width:50%; }
      "Columna izquierda"
    @end

    @column { width:50%; }
      "Columna derecha"
    @end
  @end
```

3. Implementación

3.1 Frontend

El desarrollo de esta capa de análisis sintáctico se centró en definir una gramática formal capaz de reflejar cada uno de los elementos de nuestro DSL. Para ello se distinguieron entre dos grande categorías de símbolos:

Los terminales, como por ejemplo literales con comillas y sin comillas, identificadores, marcadores de encabezado, números de lista, viñetas y los signos de puntuación y delimitadores.

Y los no terminales, que representan estructuras de más alto nivel. Como por ejemplo: program (símbolo inicial), statement_list y statement (agrupan desde definiciones y usos hasta formularios, listas, botones, etc).

Por otro lado, para hacer la gramática más manejable, se agruparon las acciones de Bison en estructuras reutilizables, como un ParameterList para cualquier colección de parámetros o estilos, un StatementList para concatenar declaraciones, NavItem y FormItem para gestionar ítems de navegación y formularios.

Gracias a todo esto se logró descartar en esta capa múltiples casos inválidos: falta de cierre de bloques, literales sin comillas, encabezados vacíos o anidamientos incorrectos de @end.

3.2 Backend

El backend del compilador tiene como objetivo principal realizar el análisis semántico del programa y generar como salida un documento HTML. Para comenzar con el desarrollo del backend se tuvo que tener en cuenta la definición de gramática y del lenguaje elaborado en las etapas anteriores para poder generar la tabla de símbolos adecuadamente.

La tabla de símbolos almacena la información de cada variable o función, incluyendo su nombre, valor, tipo y el bloque o función a la que pertenece. Lo que permite verificar la

existencia de una variable antes de ser utilizada, evitar redefiniciones innecesarias y validar las llamadas a funciones con respecto a la cantidad de parámetros que reciben.

En el back, también se manejan los errores semánticos, los cuales se registran en la estructura de ErrorManager. Los mismos se manejan de dicha manera para poder ser impresos al final de la ejecución. Los tipos de errores incluyen: uso de variables no definidas, redefinición de variables o funciones, cantidad incorrecta de argumentos en llamadas a funciones e ítems no válidos en listas ordenadas.

Una vez validado el programa, el backend continúa con la etapa de generación de código. A partir del árbol de sintaxis abstracta (AST), se recorre cada nodo desde la función _generateProgram que itera por una lista de statements. La misma llama a la función _generateStatement donde se genera el correspondiente bloque de HTML para cada statement específico.

Este proceso incluye la escritura de etiquetas como encabezados, párrafos, listas, tablas, imágenes, botones, formularios y otros componentes comunes de una estructura web. Además, se permite la reutilización de componentes mediante declaraciones y usos de bloques definidos por el usuario, lo cual agrega flexibilidad y modularidad al lenguaje.

Se definió el struct DefineStatementList para almacenar todas las definiciones creadas mediante sentencias @define. Cada nodo de esta lista contiene una definición con su nombre, parámetros, estilos y cuerpo asociado. Esta implementación permite construir un entorno de componentes reutilizables, similar a funciones o macros, que luego pueden ser invocados mediante sentencias @use. Esta implementación facilita la búsqueda secuencial cuando se encuentra una sentencia @use. En ese caso, el backend recorre la lista buscando el nombre de la definición a utilizar y, si lo encuentra, reemplaza los parámetros con los valores pasados y genera el contenido HTML correspondiente reutilizando el cuerpo de la definición original.

El archivo de salida se crea automáticamente en formato HTML /src/output/output.htm, se puede modificar utilizando variables de entorno. Comienza con un prólogo que incluye la declaración del documento y las etiquetas de apertura necesarias, sigue con el cuerpo del programa traducido y finaliza con el epílogo que cierra correctamente el documento.

3.3 Dificultades Encontradas

Un inconveniente fue con la gestión de memoria, más precisamente al momento de liberar el CompilerState. Algunos strings en Flex no se encontraban disponibles una vez hecho el free de CompilerState. Para solucionar este problema, se decidió no liberar esos strings directamente para asegurar que permanecieran disponibles durante todo el proceso de análisis y generación de códigos.

4. Futuras Extensiones

Aunque el compilador ya cumple con lo que se propuso en un principio, hay varias ideas que podrían sumarse a futuro para hacerlo más completo y atractivo.

- Agregar scopes: Hoy en día, todo lo que se escribe en el lenguaje se interpreta dentro de un mismo contexto. Poder definir scopes o bloques con cierto alcance permitiría organizar mejor el contenido, aplicar estilos distintos a diferentes secciones o incluso preparar el terreno para funcionalidades más avanzadas más adelante.
- Mejorar la estética de la página generada: Si bien el HTML que se genera es correcto y funcional, todavía tiene un diseño bastante básico. Sería bueno trabajar en una versión más prolija y visualmente atractiva, con mejores estilos, colores definidos y distintos tipos de tipografías para generar una presentación más moderna y atractiva al usuario.

Estas mejoras no eran parte del alcance inicial, pero sin dudas son un buen punto de partida para seguir desarrollando el proyecto en el futuro.

5. Conclusiones

El desarrollo de este proyecto cumplió con los objetivos que fueron propuestos desde un principio: diseñar un lenguaje específico de dominio (DSL) orientado a la generación declarativa de páginas web, y construir un compilador completo capaz de traducirlo a HTML con estilos CSS embebidos.

El DSL definido permite expresar de forma sencilla y estructurada contenido web estático, con una sintaxis accesible y elementos comunes como encabezados, listas, botones, formularios y tarjetas. Por otra parte, el compilador puede procesar correctamente los programas escritos en este lenguaje, generando como salida archivos HTML funcionales, bien definidos y reconocibles en un navegador.

La división del proyecto en tres etapas, una para el diseño del lenguaje, otra para el desarrollo del frontend y una última para el backend, resultó muy beneficiosa. Esta estructura por entregas permitió organizar el trabajo de manera progresiva, abordando una parte a la vez y asegurando que cada componente estuviera bien definido antes de avanzar al siguiente. Gracias a esto, se logró una mejor planificación, mayor claridad en los objetivos de cada fase, y una integración más ordenada de todas las partes del compilador.

En resumen, se obtuvo una herramienta funcional y sólida con potencial para futuras extensiones. El lenguaje y el compilador están bien formados, y podrían servir como base si en el futuro se quiere sumar más funcionalidades, hacer validaciones más complejas o incluso generar otros tipos de salida además de HTML.