Trabajo Práctico Especial

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

2021 1Q

Fecha de entrega: 27/06/2021

BERNAD, Francisco (59.538)
MANFREDI, Ignacio (58.513)
LEGAMMARE, Joaquín (59.376)
RAMPOLDI, Nicolas (59.066)

Índice

1. Idea subyacente y objetivo del lenguaje

Cuando se diseñó e ideó este lenguaje, se tuvo en mente a aquellas personas cuya habilidad se encuentra más inclinada al backend y necesitan una interfaz de manera rápida para probar las implementaciones.

Percy es un lenguaje de programación que permite crear de forma sencilla tags html, con una sintaxis similar al lenguaje javascript. Al compilar el archivo .percy, se obtiene como salida un .html que renderiza el contenido que haya sido ingresado en ese primer archivo.

El objetivo fundamental de esta implementación es encontrar el balance adecuado entre la reutilización de código y la velocidad de compilación. Al generar el contenido html y servirlo de forma estática, se logra que se presente de manera correcta y eficiente.

2. Consideraciones realizadas

La primera consideración que se tuvo en cuenta fue el hecho de que el código html creado por el usuario de Percy fue escrito correctamente. Es decir, no anidar tags que no pueden ser anidados (que no se cumpla la sintaxis html). Percy no válida esto, por lo tanto se asume que está contemplado por el usuario.

Por otro lado, se tomó la decisión de que el usuario no sea el responsable del manejo de memoria cuando utiliza Percy, para simplificar este proceso, cuando se termina el parseo y compilación, se liberan todos los recursos alocados previamente.

Además, el set de etiquetas html provisto por Percy es fijo, el usuario no puede agregar sus propias etiquetas. De esta forma se evita generar código inválido y se disminuye al máximo posible el margen de error.

Existe un manejo de errores por parte de Percy, para los cuales se hace un exit y se imprime a salida estándar el error y, en algunos casos, la línea en la cual se provocó el fallo.

3. Descripción del desarrollo del TP

En primer lugar se utilizó FLEX, el cual brinda un analizador léxico que permite reconocer los lexemas de la gramática. Luego con el uso de Yacc se definieron las reglas de la gramática del lenguaje Percy, dando origen al parser.

A continuación, una vez definida la gramática del lenguaje, se decidió implementar un árbol para parsear y analizar el archivo .percy brindado por el usuario . En este árbol se colocan los nodos que representan una construcción en el código fuente.

Luego de haber generado el árbol, se ejecuta su nodo raíz (main) y a partir de allí se empieza a recorrer el árbol procesando las funciones, expresiones, declaraciones, asignaciones y definiciones.

Dada esta forma de implementar tanto el árbol como la ejecución del mismo, se puede deducir que, como fue mencionado con anterioridad, no hay problema alguno en que el usuario forme un documento html que no es válido, ya que las sentencias se ejecutan y en ningún momento se analiza si es válido o no. Por lo tanto, es responsabilidad del usuario escribir los tags html de forma coherente.

Se utilizó además una tabla de hashing para almacenar tanto nombre de variables como de funciones. Se utilizan sus nombres como identificadores, y se los utiliza como key en la tabla de hash. Esto brinda la posibilidad de controlar que no se redefinen variables y también de asegurarse que al utilizar una variable haya sido declarada previamente.

4. Descripción de la gramática

Toda sentencia en Percy debe ser finalizada con el carácter “;”

Tipos de datos

- Cadenas de caracteres - string
- Números enteros - int
- Etiqueta-element
  - *html:* Representa la raíz del documento HTML.
  - *navbar:* Barra de navegación.
  - *footer:* Footer para el documento o sección, suele contener información relevante.
  - *container:* Una división o sección en un documento HTML.
  - *header:* Representa un tag de tipo “h1”
  - *text:* representa un tag de tipo “p”
  - *body:* representa el tag de tipo “body”

Las variables se declaran anteponiendo el tipo de dato seguido de su nombre (identificador). Adicionalmente se puede definir su valor luego del nombre con un “=” y a continuación su valor. En el caso de tratarse de una etiqueta-element, para asignarle un valor se debe colocar la palabra “new” luego del igual y seguido el valor de etiqueta. Todos los tags se encuentran con un estilo css predeterminado.

Ejemplos:

1. Sintaxis de las declaraciones:

```
int a;
string b;
element tag;
```

2. Sintaxis de las definiciones:

```
int a = 3;
string b = "Hola Mundo!";
element tag = new html();
```

3. Sintaxis de las asignaciones (tener en cuenta que deben haber sido declaradas las variables previamente).

```
a = 3;
b = "Hola mundo!";
tag = new html();
```

Operadores aritméticos

- Se permiten el uso de los operadores ‘+’, ‘-’, ‘*’ y ‘/’ y “%”.

Operadores lógicos

- Se permite el uso de los operadores AND (“&&”) y OR (“||”).

Operadores relacionales

- Se permite el uso de los operadores MAYOR (">"), MAYOR O IGUAL (">="), MENOR ("<") , MENOR O IGUAL ("<=") , IGUAL (“==") y NEGACIÓN (“!=").

Operadores de asignación

- Se permite el uso del operador ASIGNACIÓN (“=”)

Condicionales

- Se permite la sentencia condicional “if” y “else”

Bucles

- Se permite las estructuras de control “for”, “while” y “”do while” para modificar el flujo de ejecución de las instrucciones de un programa.

Ejemplos:

1. Sintaxis de un ciclo For:

```
main(){
    element html_em = new html();
    element body_em = new body();

    element nav_em = new navbar("For loop example");

    html_em.insert(body_em);
    body_em.insert(nav_em);

    int a = 0;

    element cont_em;
    element text_em;

    for(int b = a + 5; a < b ; a = a+1 ){
        cont_em = new container();
        text_em = new text(a);
        cont_em.insert(text_em);
        body_em.insert(cont_em);
    }

    element footer_em = new footer("This is a for loop example");
    body_em.insert(footer_em);

    render(html_em);
}
```

2. Sintaxis de un ciclo *While*:

```
main(){
    element html_em = new html();
    element body_em = new body();

    element nav_em = new navbar("While loop example");

    html_em.insert(body_em);
    body_em.insert(nav_em);

    int a = 0;
    int b = a + 5;

    element cont_em;
    element text_em;

    while(a < b){
        cont_em = new container();
        text_em = new text(a);
        cont_em.insert(text_em);
        body_em.insert(cont_em);
        a = a+1 ;
    }

    element footer_em = new footer("This is a while loop
example");
    body_em.insert(footer_em);

    render(html_em);
}
```

3. Sintaxis de un ciclo *Do While*:

```
main(){
    element html_em = new html();
    element body_em = new body();

    element nav_em = new navbar("Do-while loop example");

    html_em.insert(body_em);
    body_em.insert(nav_em);

    int a = 0;
    int b = a + 5;

    element cont_em;
    element text_em;

    do{
        cont_em = new container();
        text_em = new text(a);
        cont_em.insert(text_em);
        body_em.insert(cont_em);
        a = a+1 ;
    } while(a < b);

    element footer_em = new footer("This is a do-while loop
example");
    body_em.insert(footer_em);

    render(html_em);
}
```

5. Dificultades encontradas en el desarrollo del TP

Una de las primeras dificultades que se presentó fue comprender el funcionamiento del analizador léxico FLEX y de Yacc. Para poder solucionar esta problemática fue necesario investigación y de lectura de documentación sobre estas dos herramientas.

El siguiente problema encontrado fue la aparición de conflictos en la gramática. Se tuvo que replantear la gramática para solucionar los conflictos de shift reduce y reduce reduce.

Otra dificultad hallada fue realizar el correcto diagrama del AST (Abstract Syntax Tree). Se optó por utilizar distintos tipos de nodos según sea necesario (funciones, expresiones, declaraciones, asignaciones y definiciones), cada uno contiene los datos y funciones fundamentales para su funcionamiento. Entre estas funciones se encuentran las de procesamiento y las de liberación de recursos.

Por otra parte, un inconveniente que hubo que enfrentar fue liberar todos los recursos utilizados a lo largo del parseo y compilación. Para ello, una vez finalizado todo el procesamiento, se liberan los nodos correspondientes del árbol y la memoria utilizada por la tabla de hashing.

6. Futuras extensiones

Como se mencionó en el apartado de “Descripción del desarrollo del TP”, se utilizó una tabla de hashing para almacenar nombre de funciones. Si bien actualmente solo se cuenta con la función main, se pensó esta estrategia para que en una futura implementación, se puedan incorporar como funcionalidad otras funciones que puedan ser definidas por el usuario

Por otro lado, en una versión posterior se podrían implementar una mayor cantidad de tags, que le brinden al usuario mayor agilidad al momento de escribir código, para tener de esta forma un frontend rápido para poder probar funcionalidades del backend.

Además, se podría analizar el código html generado y ver que este sea válido y de esta forma evitar que el usuario anide erróneamente los tags para evitar paginas mal formadas.

Si bien se tienen en cuenta los valores enteros, no sucede lo mismo con los valores de punto flotante. En una próxima versión se podría agregar el manejo de estos y permitir operaciones aritméticas con los mismos.

El único tipo de CSS soportado por Percy es el inline css, posteriormente se podría añadir la opción de crear un archivo externo que contenga todas las reglas de estilo, de esta forma se generaría un código más compacto y modularizable. En un futuro lanzamiento se podría agregar la funcionalidad de permitirle al usuario definir los estilos y sobrescribir los predeterminados.

7. Referencias

- Tabla de hashing
- Lex and Yacc samples and tools repository for Languages and Compiler class