INSTITUTO TECNOLÓGICO DE BUENOS AIRES

ITBA

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

PipoScript

Profesores

Santos, Juan Miguel

Arias Roig, Ana María

Ramele, Rodrigo Ezequiel

Alumnos

Brandy, Tobias – 59527

Pannunzio, Faustino – 59223

Sagüés, Ignacio – 59244

Vazquez, Ignacio – 59309

Índice

Idea subyacente y objetivo del lenguaje

¿Por qué es necesario este lenguaje? Situándonos en el cuatrimestre pasado, durante la cursada de la materia de HCI, se evidencio una diferencia clara entre HTML y los lenguajes de programación que habíamos utilizado anteriormente. Al día de hoy, el lenguaje no provee ninguna herramienta nativa para reutilizar componentes y modularizar el diseño.

Teniendo en cuenta esta situación, consideramos que una herramienta que permita generar archivos HTML con el poder de un lenguaje de programación en tiempo de compilación sería de gran utilidad.

En base a la problemática ya detallada, ideamos e implementamos PipoScript. Este es un lenguaje tipado con sintaxis basada en C que tiene como objetivo principal permitir la modularización y parametrización del HTML en distintos componentes y funcionalidades. A partir de esto, creemos que la reutilización y la mantenibilidad del código en la programación de páginas web aumentaría.

Respecto a que sea en tiempo de compilación, consideramos que generar el contenido a ser mostrado una única vez y servirlo siempre de forma estática es muy beneficioso. Esto aplica tanto a cuestiones de performance como la garantía de que el contenido será servido de la forma correcta (sin errores de generación). Cabe destacar que las páginas estáticas también son favorables desde el punto de vista de SEO (Search Engine Optimization).

Breve descripción del lenguaje

PipoScript cuenta con 3 tipos de datos principales, números enteros, cadenas de caracteres y etiquetas. Siendo esta última central para el objetivo del lenguaje. Además, cuenta con gran parte de las operaciones aritméticas, booleanas y de control de flujo ofrecidas por C, funciones nativas para el manejo de strings, capacidad para generar y separar comportamientos en distintas funciones y a lo largo de múltiples archivos, y por supuesto operaciones para la creación y modificación de etiquetas con las que a partir de ellas se generará el HTML.

Consideraciones realizadas

Generación y renderizado de etiquetas

El producto final de la compilación es un archivo HTML. Para lograr esto se plantearon dos etapas. La primera y donde el usuario tiene el mayor control es la creación y configuración (mediante las herramientas del lenguaje) de un objeto de tipo tag que representa la etiqueta raíz del HTML que queremos generar. Vamos a poder asignarle un nombre, cuerpo, atributos (y sus valores) y sus etiquetas hijas. Esta etiqueta raíz debe ser retornada por la función main para pasar a la siguiente etapa, la renderización.

Para esto se utiliza una rutina recursiva aprovechando la forma de árbol que tiene un archivo HTML bien formado. Este proceso consiste en generar la etiqueta con su nombre, atributos y contenido seguido de la generación de todas sus etiquetas hijas. Este proceso da como resultado un archivo HTML indentado de manera correcta y valido (si es que el

usuario respetó las convenciones del estándar HTML) para ser interpretado por cualquier navegador.

Manejo de funciones

Para poder lograr el mayor grado de modularización y reutilización de componentes, el lenguaje cuenta con la capacidad de definir funciones. Estas permiten sintetizar rutinas de código de la longitud deseada en pocas palabras. Permite esconder todo el boilerplate HTML asociado a la creación de un componente, parametrizarlo mediante los argumentos de la función, y solo declarar aquellos valores que generan valor. Este es un concepto clave en el lenguaje, pues facilita la reutilización y mantenibilidad del código HTML, los dos grandes objetivos que nos planteamos con PipoScript.

Compilación de múltiples archivos

Otro factor importante es la claridad del código. Obligar al código a estar todo junto en un mismo archivo reduce el control del programador y puede llevar muy fácilmente a tener un archivo de gran tamaño casi indescifrable. Es por esto que se permite compilar una cantidad arbitraria de archivos con una cantidad arbitraria de funciones bajo la condición que exista un único main entre todos.

Además, creemos que esta funcionalidad facilita el desarrollo y utilización de librerías de terceros, siendo muy fácil incluirlas en nuestro proceso de compilación sin necesidad de modificar los archivos. Lo único importante a tener en cuenta en librerías públicas será utilizar un namespace en los nombres de las funciones, para reducir las probabilidades de colisiones de nombres.

Manejo de memoria

En PipoScript consideramos que no debe ser tarea del usuario el manejo de la memoria utilizada por el compilador durante la interpretación del código para generar el HTML. Teniendo en cuenta que el proceso de compilación se realizará una sola vez, y luego el archivo generado podrá ser interpretado por los navegadores con la misma eficiencia y efectividad que cualquier otro, decidimos que es más importante priorizar la facilidad de aprendizaje y uso del lenguaje, que su performance.

Si bien no queríamos que el manejo de memoria recaiga sobre el usuario, tampoco nos pareció que esta funcionalidad sea la más interesante del lenguaje. Es por esto que, teniendo en cuenta el corto tiempo de desarrollo del compilador, optamos por una solución simple pero funcional. Toda la memoria alocada por el usuario durante el proceso de interpretación no será liberada hasta que finalice la generación de HTML.

Manejo de strings

Dado que PipoScript es un lenguaje cuyo principal objetivo es el de manipular etiquetas y sus propiedades, y que los valores de estos son strings, se consideró crucial brindar funcionalidad al usuario tanto para facilitar el trabajo con los mismos como para aumentar el poder expresivo del lenguaje. De esta manera, se decidió implementar funciones tales como pedir la longitud de una cadena y compararla con otra, para de este

modo poder combinar estos valores con los bloques de control. Con esta posibilidad, se permite al usuario tomar decisiones en base a, por ejemplo, el valor de una propiedad de la etiqueta. Además, se posee la capacidad de concatenar cadenas. Si bien esta funcionalidad se consideró importante desde el primer momento, resultó siendo crucial para el correcto funcionamiento del lenguaje. Concretamente, en el caso de los atributos de una etiqueta, dado que estos pueden ser multivaluados, podría ocurrir que se quiera agregar un nuevo valor al atributo además de los ya existentes. Para hacer esto, se puede pisar el valor previo con la concatenación del valor actual y el atributo que se desea agregar. De este modo, una funcionalidad que sin dudas debía ser provista se encuentra presente gracias a las funciones de manejo de strings.

Mensajes descriptivos

Durante el proceso de compilación se generan tres tipos de mensajes:

El primero y menos relevante son los mensajes de estado del compilador que se generan en la salida estándar. Estos permiten al usuario saber en qué etapa se encuentra el compilador, ya sea parseando, interpretando o renderizando. Estos mensajes pueden ser muy útiles para dar feedback en procesos de compilación muy largos.

El segundo y más importante son los mensajes de error. Ante la detección de cualquier tipo de error sintáctico o semántico, el compilador corta la ejecución e imprime un mensaje de error a salida de error estándar. Consideramos una prioridad que estos mensajes de error fuesen lo más descriptivos posibles, dándole al usuario la mayor cantidad de herramientas posibles para poder solucionar el error. Como parte de esto, en todos los mensajes donde es relevante, se incluye el nombre del archivo y línea del mismo donde se detectó el error. También se informa de los posibles errores internos que puede experimentar el compilador.

Por último, incluimos de manera opcional (mediante el parámetro -D) logueos que buscan representar el proceso de interpretación del código que realiza el compilador. Consideramos que esta funcionalidad resulta muy útil tanto para los desarrolladores para poder debuggear el compilador, como para los usuarios que buscan entender con mayor detalle cómo se está ejecutando su código. Estos logs también son enviados a salida de error estándar.

Valores de retorno semánticos

Teniendo en cuenta que el proceso realizado por el compilador se separa en varias etapas las cuales realizan tareas distintas. A la hora de analizar la causa de un error de compilación, es de gran utilidad saber el estado en el que ocurrió el error. Es por esto que el compilador utiliza distintos valores de retorno para indicar con mayor precisión en qué etapa de compilación ocurrió el error. Los mismos se detallan más abajo.

Descripción del desarrollo del TP

Proceso de compilación

Este proceso puede separarse en 5 etapas.

Inicialización

La primera etapa consiste en parsear los argumentos de compilación recibidos. Estos contienen los archivos que se desean compilar y, en caso de haber sido provisto, el nombre del archivo de salida. Otros parámetros opcionales son activar los logs de debug(-D) y imprimir un mensaje de ayuda (-h). Luego, se inicializan todas las estructuras y servicios utilizados para el proceso de compilación (function y symbol tamble, servicios de tags y strings).

En caso de que haya algún problema durante el parseado de argumentos se aborta con un código de error 1 (por ejemplo, no se especifican archivos a compilar). En caso de que haya problemas en la inicialización, se aborta con un código de error 2 (por ejemplo, no se puede abrir el archivo de salida especificado, o el default).

Analisis Sintactico

Esta es la primera de las 3 etapas centrales. Consiste en realizar el analisis lexico y sintactico a todos los archivos obtenidos en el paso anterior. Este proceso consiste en abrir el archivo en cuestión y utilizar la interfaz provista por yacc y lex para realizar el análisis del código en base a la gramática definida. En caso de que el análisis haya sido exitoso, se reinician las variables necesarias, se cierra el archivo actual y se procede al siguiente en la lista.

El principal efecto de este proceso es la generación de un AST (Abstract Syntax Tree). Esta estructura junto a la tabla de símbolos y la tabla de funciones son los recursos necesarios para la etapa de interpretación.

Interpretación

Para este punto, ya se leyeron todos los archivos fuentes y se generó una estructura de árbol que permite hacer la ejecución de todo código necesario para la interpretación del programa. En este caso, teniendo en cuenta que el tipo de lenguaje que estamos generando no es procedural, es necesario ejecutar todas las sentencias que hayan sido declaradas.

Este proceso se ve favorecido por la estructura del árbol. Antes de ejecutar una función, podemos ejecutar todo el subárbol que lo siga para obtener los valores necesarios.

Se empieza ejecutando la función de entrada main que se debe encontrar en la tabla de símbolos de función, y a partir de allí se irá recorriendo el árbol según las sentencias de control y llamados a funciones indiquen. Es importante notar que solo se interpretarán las sentencias (nodos del árbol) que realmente se ejecuten. Esto tiene la consecuencia de que si existe un error semántico en una parte del código que no es ejecutada, este error no se detectará, ya que la ejecución del programa puede realizarse sin ningún problema.

El principal efecto de este proceso es la generación de la estructura de datos que representa a la etiqueta HTML raíz, la cual ahora pasará a ser renderizada.

Renderizado

Como ya se mencionó previamente, la función main devuelve una etiqueta raíz que ahora debe ser renderizada. Para esto se utiliza el archivo index.html como default o aquel archivo que haya sido indicado vía parámetros del compilador. El mecanismo para lograr el renderizado ya fue explicado en una sección previa.

Con el HTML renderizado en el archivo de salida, los tres procesos principales de la compilación ya terminaron.

Finalización

Por último, una vez se haya terminado todo el proceso de compilación, es necesario liberar todos los recursos que fueron utilizados. Específicamente se liberan la tabla de símbolos y de función (donde se libera de manera recursiva todo el árbol AST), los strings estáticos, los recursos utilizados por lex, todos los tags y strings alocados por el usuario durante la interpretación (se ejecuta el garbage collector) y se cierra el archivo de salida.

Abstract Syntax Tree

Para la implementación del AST optamos por tener un tipo de nodo genérico AstNode que representa cualquier nodo dentro del árbol. Para diferenciar que tipo de nodo es (de asignación, de llamado de función, etc), el AstNode posee una propiedad nodeType, el cual indica el tipo de nodo al que debe ser casteado, que funcion se debe usar para ejecutar dicho nodo, y su función de liberación. También se incluyen el número de línea y el nombre de archivo de donde se extrajo dicho nodo para poder realizar los mensajes de error descriptivos.

Por ejemplo, la operación de suma tiene el tipo de nodo ‘+’. Su nodo posee dos puntero a un AstNode que representan los dos valores a sumar. Estos valores son árboles y no directamente enteros, pues podrían hacer referencia a múltiples operaciones matemáticas, a una función, o una variable. Para obtener el valor entero que representa cada AstNode debemos ejecutar la función de procesamiento asociada a cada nodo.

Los nodos raíces de nuestro AST son los de declaración de función, pues todo código PipoScript se encuentra dentro de una declaración de función. Estos nodos, a su vez, se encuentran en la tabla de símbolos de función. Es así como liberar la tabla de funciones, significa liberar todo nuestro árbol.

Cabe destacar que las funciones de procesamiento reciben la tabla de símbolos a utilizar, es decir las variables con las que pueden a trabajar. De esta manera se implementa el concepto de que las variables tengan scope de función. Una función no puede acceder a variables declaradas en otra.

Symbol table y function table

Tanto para la tabla de símbolos como para la tabla de funciones, se decidió implementar un mapa. Se mapean claves strings con punteros a nodos símbolos en el primer caso, y a nodos de declaración de función en el segundo. Sin bien estas implementaciones parecen similares funcionan un tanto distintas.

En el caso de la tabla de símbolos, cada nodo de llamado de función dentro del árbol crea la tabla de símbolos que utilizará para ejecutarse. El problema es que se debe conservar una referencia global a todas las tablas pues, como se está recorriendo un árbol, en cualquier nodo de este se puede encontrar un error y abortar. En este caso habrá que acceder a la referencia global y liberar todas las tablas de símbolos inicializadas actualmente. Para esto se usó una lista doblemente encadenada. Conforme se van agregando tablas de símbolos se actualiza la lista, manteniendo una referencia al último eslabón.

Para la tabla de funciones, en cambio, el mapa se presenta de manera estática dado que esta es única para todo el programa. Esto trae la consecuencia de que no pueden haber dos funciones con el mismo nombre.

Strings Estáticos

A medida que se parsea el archivo, uno de los no terminales que pueden ser encontrados son los strings estáticos, es decir, identificadores de variables, de función y strings literales. Si bien estos son representados por `yytext` este valor es cambiante y no es posible guardar una referencia al mismo. Es por esto que la única alternativa es copiar el valor al que apuntan. Para esto se utilizó una tabla de hash, con el objetivo de almacenar una única vez cada string. Aprovechando que estos strings son estáticos, cada vez que se necesite un string estático con un valor que ya tenemos reservado en la tabla, podemos simplemente utilizar un puntero al valor ya reservado.

Teniendo en cuenta que un identificador es declarado con el fin de usarlo posteriormente, el string asociado al nombre del mismo aparecerá muchas veces. Si se guardase cada aparición del mismo sin reutilizar asignaciones previas, se generaría un uso innecesario de memoria.

Manejo de memoria del usuario

Tags

Se implementó un memory manager para poder liberar correctamente todos los tags generados por el usuario. Todo tag nuevo creado se guarda en una lista, y a la hora de la liberación de recursos hay que recorrerla y encargarse de liberar cada nodo individualmente.

Strings

Teniendo en cuenta que el lenguaje ofrece strings de longitud variable y asume la responsabilidad de manejar el almacenado y control de los mismo, se debió implementar un memory manager también para este tipo de dato. Dado que se pueden concatenar strings y convertir números a strings, ya no alcanza con el manejo de strings estáticos ya mencionado. Por esto se agregó un segundo sistema de manejo de memoria que reserva la cantidad solicitada y almacena todas las referencias para ser liberadas una vez se termine la ejecución. Su comportamiento es muy similar al realizado para los tags.

Descripción de la gramática

Tipos de dato

Se ofrecen 3 tipos de datos que poseen el mismo comportamiento base. Por un lado, existe una fase de declaración donde se registra el símbolo (se genera el valor izquierdo) y se le asocia un tipo de dato. Por el otro, está la fase de inicialización o asignación en donde se asigna un valor a ese símbolo (generación del valor derecho). Este

comportamiento es muy útil para que una misma variable posea distintos valores derechos a través de su ciclo de vida. Es importante tener en cuenta que no se permite utilizar una variable no inicializada para asignarle su valor a otra, o como valor de retorno de una función.

Estos tipos de datos se pueden separar en dos categorías: los mutables y los inmutables. Las referencias (variables) a los tipos de datos inmutables siempre son a valores distintos, es decir que nunca una operación sobre una variable con tipo de dato inmutable puede afectar a otra. Por el contrario, distintas variables con tipo de dato mutable pueden referenciar al mismo valor, es decir que una operación sobre una variable puede cambiar el contenido de otra.

- **Números enteros** (int)

  Este es el único tipo de dato para el manejo de números. Puede ser utilizado en operaciones aritméticas, lógicas y de casteo. También es utilizado para representar los valores booleanos, donde el valor 0 representa false, y un valor distinto de 0 representa true. Este tipo de dato es inmutable. Dependiendo de la arquitectura ocupa 2 o 4 bytes. Son declarados usando la palabra reservada `int`

- **Cadenas de caracteres** (string)

  Representación de las cadenas de caracteres. Pueden ser declaradas explícitamente, obtenidas casteando un número a string, o como producto de concatenar otras cadenas. Este tipo de dato es inmutable. Son declarados usando la palabra reservada `string`.

- **Etiqueta** (tag)

  Representación de una etiqueta XML/HTML. Es un tipo de dato mutable. Para declarar este tipo de variable se utiliza la palabra reservada `tag`. Cuenta con 4 propiedades que la definen:

  - **Nombre** (name <string>)

    Nombre de la etiqueta. Si se trabaja en un ambiente de HTML puro estos deberían ser nombres de etiquetas válidas dentro del lenguaje. Preferentemente aquellas con nombre semántico para favorecer la accesibilidad.

  - **Contenido** (body <string>)

    Contenido textual dentro de una etiqueta. Es la información contenida dentro de las etiquetas. Esta siempre es renderizada antes de cualquier sub etiqueta contenida.

  - **Sub etiquetas** (children <tag>)

    Todas las etiquetas contenidas por la etiqueta principal en orden de inserción. Permiten generar la representación en forma de árbol que caracteriza a XML y HTML.

  - **Atributos** (attributes <string, string>)

    Conjunto de atributos incluidos en la etiqueta de apertura. Son representados en forma clave valor. Ambas de tipo string. Permite representar tanto atributos con valor o sin. Siendo estos últimos el

equivalente a asignar como valor el mismo string que la clave (siguiendo el estándar HTML).

- **Tipo vacío** (void)

  Representación de una función sin valor de retorno. Para declarar una función de esta manera, aclarando que no devolverá ningún valor, se usa la palabra reservada `void`.

Inicialización

Las variables dentro de nuestro lenguaje, hasta que se les asigna un valor derecho, son consideradas no inicializadas. Para realizar esta inicialización se utiliza el símbolo `=` seguido del valor derecho a ser asignado. En el caso particular de los tags, la asignación se realiza utilizando la palabra reservada `new` seguida de la palabra `tag`. Luego de realizar esto, el tag posee un `name` y un `body` equivalente a un string vacío. A su vez, tampoco poseerá ningún `attribute`. Para darles un valor distinto del string vacío se usa `set`.

Cualquier operación que requiera de un valor derecho, fallará si se utiliza una variable no inicializada.

Sentencias de control

Son una de las funcionalidades más importantes para aumentar el poder expresivo de nuestro lenguaje. Permiten generar distintos flujos de ejecución. Dentro de las sentencias ofrecidas se encuentran las palabras reservadas para los controladores de ciclos como `do, while, for` correspondientes al no terminal `iteration_statement` de la gramática. Como mecanismos para generar distintos flujos de ejecución en base a condiciones se cuenta con las palabras reservadas `if, else` correspondientes al no terminal `if_statement` de la gramática.

Operaciones

Aritméticas

Tipo de operaciones entre números. Dentro de las mismas se encuentra la adición (`+`), sustracción (`-`), división (`/`), multiplicación (`*`), módulo (`%`), y opuesto (`-` unario). También se incluyen los operadores de incremento (`++`) y decremento (`--`). Por último, para afectar la precedencia, se incluyen los paréntesis (`(`, `)`).

Lógicas

Para el uso de sentencias de control es necesario utilizar condiciones que permitan determinar el comportamiento. Con este fin, el lenguaje cuenta con los operadores lógicos más tradicionales. Estos son igualdad (`==`), desigualdad (`!=`), mayor (`>`), mayor igual (`>=`), menor (`<`), menor igual (`<=`), AND (`&&`), OR (`||`) y la negación (`!`), permitiendo combinar operaciones lógicas en condiciones más complejas y específicas. También aprovechan el uso de paréntesis.

Cadenas de caracteres

Si bien los string permiten representar cadenas de caracteres de tamaño variable, no siempre alcanzan con ese valor. Es por esto que la posibilidad de operar con strings es de gran utilidad. En este caso, la principal operación para aumentar el poder expresivo es la posibilidad de concatenar strings. Para realizar dicha operación se cuenta con la palabra reservada `concat` la cual ejecutada como función, recibe dos strings y devuelve su concatenación. Esto, sumado a las sentencias de control y los strings como tipo de dato permite generar cadenas en gran cantidad y con gran precisión.

Sumado a la concatenación, el lenguaje ofrece conversión nativa tanto de string a int, como de int a string. Para la primera se utiliza la palabra reservada `str` como función, recibiendo un int como argumento. Para la segunda, se realiza algo similar, utilizando `int` como función y recibiendo un string como argumento. Además, se ofrece la posibilidad de obtener la longitud de una cadena usando la palabra reservada `len` y la opción de compararla con otra. Para esto último se utiliza `cmp` la cual recibirá dos cadenas, devolviendo 1, 0 o -1 indicando el resultado de la comparación.

Manipulación de etiquetas

Teniendo en cuenta que uno de los tipos de datos nativos son las etiquetas y que su información está dividida en 4 propiedades, es necesario contar con las herramientas para poder manipularlas y utilizarlas de manera correcta.

Property names

Para poder pedir los distintos valores de las propiedades de la tag se usan `name, body, attribute`. Estos valores serán referenciados como `<property_name>` y podrán ser obtenidos como se indicará a continuación.

Asignación

Para poder asignarle valores a las propiedades de una etiqueta existe una combinación de palabras reservadas. Cuales son y cómo se combinan dependen del tipo de dato. Todas estas operaciones son consideradas un assignment statement.

- **Nombre y Contenido**

  Ambas propiedades son de tipo string. En estos caso se utiliza `set <property_name> from <tag_variable> = <string_value>;`. Esto permite indicar cual de las propiedades de tipo string se desea modificar, cual es la variable (tag) en cuestión y por ultimo, cual es el nuevo valor que va a ser asignado. Cabe destacar que la asignación pisa los valores previos, en caso de querer anteponer o concatenar utilizando el valor original, se deberá usar la asignación en conjunto con la desreferenciación de propiedades (getters).

- **Atributos**

  Si bien esta propiedad tiene valores de tipo string, al igual que las mencionadas previamente. Los atributos se caracterizan por ser una propiedad de tipo clave-valor. Para esto, es necesario poder especificar no solo el valor sino el nombre de la clave. Para llevar a cabo esta acción se utiliza `set <property_name>`

`<attribute_key> from <tag_variable> = <attribute_value>;`. En caso de querer generar un atributo vacío es decir, una clave pero sin un valor asociado, se debe utilizar `set <property_name> <attribute_key> from <tag_variable>;`.

- **Sub etiquetas**

  Por último, para agregar nuevas etiquetas dentro de una etiqueta en cuestión, se cuenta con `append child from <tag_variable> = <tag_value>;`. Esta combinación de palabras reservada agrega la etiqueta en cuestión al final de la lista de sub etiquetas.

Obtención de los valores asignados

Si bien el principal interés respecto a una etiqueta es cargarle los valores necesarios, también es muy útil poder obtener los valores ya asignados para procesarlos y utilizarlos junto a las demás funciones. En este caso, el único tipo de dato no accesible en la implementación actual son las sub etiquetas.

- **Nombre y Contenido**

  Por defecto tanto name cómo body, al ser propiedades de tipo string, son inicializadas con un string vacío. Esto permite que, una vez esté inicializado el tag, ambas propiedades pueden ser accedidas. Para esto se utiliza una sintaxis muy similar a la del set `get <property_name> from <tag_variable>;`, obteniendo un string como valor de retorno.

- **Atributos**

  A diferencia del nombre y contenido, los atributos son un tipo de dato más complejo. El principal obstáculo es la representación y manejo de atributos que todavía no fueron declarados, es decir, que esa clave todavía no está asociada a la etiqueta. Es por esto que, antes de acceder a cualquier atributo se debe validar que la clave exista. Para esto se utiliza `has attribute <property_name> from <tag_variable>;` y se obtiene como resultado un número con valor lógico. Una vez que se sabe que la clave del atributo está presente es seguro acceder a la misma utilizando `get attribute <property_name> from <tag_variable>;` y obteniendo el valor del atributo.

Funciones

Uno de los principales objetivos y razones para definir este nuevo lenguaje es la capacidad de modularizar la generación de etiquetas. Para esto, una de las herramientas claves es la existencia de funciones o que permitan condensar en unas pocas palabras una lógica mucho mayor.

- La sintaxis de las mismas es muy similar a la de C. Se cuenta con una primera línea en la cual se indica el valor de retorno de la función (`void`, `int`, `string` o `tag`). Luego, se indica el nombre identificador de la función. Seguido, se puede proveer una lista de argumentos que la función espera recibir al ser llamada. Esta lista es una sucesión formada por tipo de dato e identificador de la variable separado de los demás argumentos por una coma (salvando el último). Por último, se cuenta con un bloque de código a ser ejecutado contenido en llaves. Si el tipo del valor

devuelto por la función no coincide con el declarado por la función, se obtendrá un error.

```
tag getTitle(string message, int size){
      tag title;
      ...
      return title;
}

tag = getTitle(“pipo is awesome”, 1);
```

En caso de utilizar una función, no es necesario que exista una declaración más arriba o en un archivo include, basta con estar declarada en alguno de los archivos recibidos por el compilador. La llamada en sí, al igual que la declaración, tiene una sintaxis muy familiar. Consiste en combinar el nombre que describe a la función con la lista de valores o variables que se desean pasar como argumentos, sin necesidad de indicar el tipo. Dependiendo del valor de retorno de la misma, puede ser llamada como una sentencia individual o como si fuese un valor (asignación, pasaje a otra función, etc).

El valor de retorno de la función será indicado antecedido por la palabra reservada `return`. Este deberá ser del mismo tipo que el indicado al declarar la función. De omitirse esta palabra se retorna la función sin valor alguno, pero esto solo será válido en el caso de las funciones de tipo `void`.

Main

Cabe destacar que, al igual que C, es necesario que exista una función main y que la misma retorne un valor de tipo tag y no espere ningún argumento. Esto se debe a que, el resultado del proceso de compilación será el efecto de renderizar el tag devuelto.

Comentarios

Dado que se está construyendo un lenguaje de programación que pretende facilitar la modularización y reutilización de código consideramos la funcionalidad de comentarios es necesaria para el proceso de documentación de las mismas. No solo esto sino que muchas veces es necesario ocultar una porción de código, otro de los casos en que la posibilidad de comentar código resulta relevante. Piposcript soporta comentarios multilínea utilizando `/*` para abrir el comentario y `*/` para concluirlo.

Dificultades encontradas en el desarrollo del TP

Palabras reservadas inconvenientes

Dado que las palabras name, attribute y body son reservadas para las operaciones pertinentes a esas propiedades en las etiquetas, actualmente no pueden ser usadas como identificadores. El caso más problemático es que imposibilita a que una variable tag sea llamada body, lo cual sería bastante común. Esto no impide renderizar una etiqueta con nombre body, ya que eso sería un string literal y no colisionará, pero es una práctica común

en los lenguajes de programación utilizar como identificador de la etiqueta su mismo nombre.

Uso intenso del heap

Hoy en día nuestra aplicación tiene un uso muy intenso del heap. Para poder interpretar y generar correctamente el programa de salida, debe tener en memoria todos los strings constantes, todos los símbolos de función y de variables, y todo el árbol AST simultáneamente en memoria. Además, por la implementación simple de manejo de memoria del usuario que realizamos, se agregan todas las tags y strings declaradas por el usuario.

Actualmente esto no está generando ningún inconveniente, pero si se pretende utilizar PipoScript en un ambiente productivo en el futuro, este problema deberá ser solventado de alguna manera.

Valores nulos

Por decisión de diseño, nuestro lenguaje no soporta el valor nulo, esto trajo consecuencias inesperadas.

En nuestra implementación, las variables declaradas, pero no inicializadas poseen valor nulo, es por esto que es un error tratar de acceder a dicho valor. Si el programador no se encarga de inicializar sus variables, entonces consideramos correcto que le salten este tipo de errores. Sin embargo, en el caso de los atributos de una tag, no es tan sencillo saber si cierto atributo está inicializado o no (si la tag posee o no el atributo).

Para solventar este problema, incorporamos la operación `has attribute` que permite consultarlo.

De todas maneras, entendemos que es una posibilidad que dicha decisión de diseño siga trayendo problemas a futuro con situaciones similares.

Futuras extensiones

Mejor Manejo de Memoria

Como previamente mencionamos, el manejo de memoria implementado es muy simple, toda la memoria reservada por el usuario se va acumulando hasta el final de la ejecución. Mejorar este sistema sería uno de los siguientes pasos en el desarrollo del compilador de PipoScript.

Colecciones

La principal mejora que podría implementarse en términos de funcionalidad sería la de soportar colecciones. Esto permitiría hacer a nuestro lenguaje mucho más dinámico. En casos como funciones que construyen listas o crean múltiples componentes, actualmente no puede usarse una colección e iterar en ella usando sus distintos valores para la construcción de los elementos previamente nombrados.

Específicamente, la siguiente funcionalidad que creemos que agregaría mucho valor al lenguaje es la incorporación de arrays.

Se podría, por ejemplo, crear una función que liste, ayudándose de un ciclo while o for, en las distintas filas de una tabla los múltiples valores de un arreglo de strings o que construya tarjetas de manera dinámica usando los datos que estén en la colección.

Acceso a sub etiquetas

Una funcionalidad interesante relacionada a las etiquetas que no se llegó a implementar es la manipulación de las etiquetas hijas de una tag. Las funcionalidades que sería ideal ir incorporando en futuras iteraciones serían las de acceder, borrar y modificar a las etiquetas hijas operando en forma de lista (con el índice). Con estas tres operaciones, se incorporaría la posibilidad de reestructurar el árbol de una etiqueta ya creada, algo que agregaría mucho valor al lenguaje.

Otra funcionalidad interesante sería, además de poder acceder con el índice, obtener el subconjunto de etiquetas hijas que posean un nombre específico.

Soporte nativo de CSS

Hoy en día, PipoScript permite modificar la capa de estilo de las páginas web creadas con el mismo utilizando CSS embebido. Es decir, utilizando los atributos style de las etiquetas HTML (inline CSS), o mediante la etiqueta style. Las únicas herramientas que se proveen para esto son las de manipulación de strings.

En un futuro, se podría dar soporte para la inclusión de CSS externo, generando un segundo archivo CSS, y brindar funcionalidad específica para el lenguaje, más allá de la manipulación de strings, similar a lo que se hace con el tipo de dato tag.

Con esta incorporación, la creación de páginas web productivas utilizando únicamente PipoScript podría ser una realidad.

Valores de punto flotante

Otra extensión posible para el lenguaje sería la de incluir valores numéricos de punto flotante, que actualmente no son soportados. Si bien no es una incorporación clave, existen propiedades de CSS que utilizan este tipo de datos, por lo que podría ser de utilidad.

Creación de una librería estándar

Pueden implementarse funciones que se ocupen de agregar atributos típicos como margen y relleno de manera sencilla (de hecho en uno de los ejemplos se realiza esto). Si bien esto no es difícil de hacer como funciones externas, para los atributos más comunes podrían hacerse funciones propias del lenguaje que se ocupen de agregarle estos atributos a las etiquetas, y así enriquecer el lenguaje.

Manejo de atributos multivaluados

Como fue explicado anteriormente, si se desea agregar un valor a un atributo multivaluado, solo puede realizarse concatenando el valor actual del atributo con el que se

desea agregar, y asignar esa cadena como nuevo valor. Si bien esto es realizable, es una consideración que debe realizarse constantemente a la hora de manipular dicha propiedad, y que genera muchos problemas a la hora de modificar un valor ya asignado sin modificar al resto. Para facilitar este manejo se podría dar la opción de manipular (agregar, borrar y modificar) los múltiples valores de un atributo como una lista, pudiendo manipular el que se necesite sin modificar al resto.

Un ejemplo claro donde esta funcionalidad ayudaría es en el atributo class.

Validación de schema

Si bien el lenguaje diseñado permite la generación de archivos de tipo XML que no respeten ningún tipo de regla ni esquema (a excepción de una única etiqueta raíz), está pensado para ser utilizado para generar HTML. Teniendo en cuenta esto, una funcionalidad que podría ser de gran utilidad sería validación del código generado respecto a la estructura de un archivo HTML bien definido. Este sistema podría ser agregado en forma de warnings opcionales (como -Wall) con el fin de generar un mejor código. Cabe destacar que, para implementar este sistema habría que primero generar una definición local de todas las reglas que componen a un archivo bien formado y luego, contrastar el código que está siendo renderizado con las mismas.

Referencias

- Referencia para generar el BNF de la gramática: parsing - extract BNF grammar rules from yacc file - Stack Overflow
- Librería utilizada para la implementación de todas las clases que requirieron una tabla de hashing.
- Flex & Bison - John R. Levine - O'REILLY. Recurso principal referenciado para el diseño de los archivos pipoScript.y, pipoScript.l, y el árbol AST.