Trabajo Práctico Especial
Autómatas, Teoría de Lenguajes y Compiladores
Sauerkraut

Grupo **CHUCRUT**
CARACCIOLO, Juan Franco
OTA, Matías
RAIES, Tomás
SAQUÉS, M. Alejo

Resumen

`Sauerkraut` es un lenguaje de programación débilmente y dinámicamente tipado, *subyacentemente orientado a objetos*, que soporta un conjunto de tipos *built-in* inspirado en las estructuras usadas con mayor habitualidad: `Integer`, `String`, `Array` y `Object` (este último sintácticamente similar a los objetos de `JavaScript`).

El compilador se ha realizado en `C++`, utilizando el entorno `llvm` para la generación de código intermedio, y las herramientas `lex` e `yacc` para el *tokenizado* y *parsing* de la gramática respectivamente. `clang`, el *frontend* de `llvm`, es utilizado para la generación de código ejecutable.

Índice

1. Idea subyacente y descripción funcional

Como se ha detallado en el resumen, el objetivo del lenguaje diseñado fue el proveer un entorno de programación con herramientas *built-in* sumamente útiles a la hora de programar. Además de los tipos de datos especificados como mínimos por la Cátedra, `Integer` y `String`, `Array` y `Object` constituyen dos herramientas útiles de la *librería estándar* del lenguaje. Nótese que el último tipo mencionado, `Object`, se comporta internamente como una *hash table*, la cuál utiliza una librería públicamente disponible implementada con un algoritmo de *linear hashing* [1].

La intención fue desarrollar un lenguaje orientado a *scripting*, por lo que las instrucciones se ejecutan en el orden de aparición en el archivo fuente. Es decir, no existe una función `main()` declarada explícitamente por el usuario que sea el punto de entrada al programa.

Por otro lado, en el resumen se ha mencionado que el lenguaje es *subyacentemente orientado a objetos*. Esto quiere decir que, si bien actualmente el lenguaje no soporta la definición de nuevas clases, el diseño interno de la librería de `Sauerkraut` está basado sobre una lógica de envío de métodos a objetos. A continuación se exhibirán algunas de las estructuras de la librería hecha en `C` sobre las cuales se construyó dicha lógica:

```
typedef struct Object {
        void * instance;
        Class * class;
} Object ;
```

A toda variable definida por el programador se le asigna un puntero a `Object` que, como podrá verse, contiene a su vez una referencia a `Class`:

```
typedef struct Class {
        const char * name;
        Object ** methods;
        int nMethods;
} Class;
```

De esta forma, al acceder a la referencia de la clase del objeto en cuestión, se accede a su vez a los métodos a los que la misma responde. Así pues, el programador podría realizar lo siguiente:

```
var obj = {"array":[5];}

print(obj.contains("array"))
```

Código cuya salida sería `1`. Internamente, la librería del lenguaje buscará linealmente en la lista de métodos para la clase `Object` (apuntada por el objeto que contiene la variable `obj`) un método cuyo nombre sea `contains`. En el caso de que el método invocado no exista para un tipo dado, se producirá una terminación anormal del programa, exhibiendo un mensaje de error adecuado al usuario, especificando qúe método de qué clase se quiso invocar pero no fue encontrado. Nótese que `struct Object` y la clase `Object` son entidades diferentes: la primera es una estructura interna de la librería del lenguaje, y la segunda es un tipo expuesto al programador.

Se puede observar que el arreglo de métodos está definido como un arreglo de `Object`: esto se relaciona con uno de los planteos originales del lenguaje, que consistía en que las funciones definidas por el programador también se comportaran como objetos, lo cual hubiera permitido su almacenamiento en variables. Dado que la *infraestructura* de la librería lo permite, sería cuestión de futuras extensiones el modificar el compilador para que las nuevas funciones definidas se comporten como tal.

2. Consideraciones adicionales

Además de la implementación de los tipos adicionales `Array` y `Object`, la implementación del compilador se destaca por el uso del entorno `llvm` (*low level virtual machine*), el cual proveyó de herramientas sumamente útiles a la hora de generar código. La consideración con respecto a este punto está vinculada a la necesidad de escribir el compilador en `C++` en vez de en `C`, tal como se ha requerido en el punto 3 del enunciado del trabajo.

A medida que se va realizando el *parsing* con `yacc`, se van agregando nodos a un `Árbol de Sintaxis Abstracta (AST)`, que luego se recorrerá para generar código con ayuda de `llvm`. Este código se genera en LLVM IR, que se vuelca como salida a un archivo `.ll` en formato texto plano. Posteriormente, aprovechando la funcionalidad del compilador `clang` para compilar tanto codgo en `C` como código `LLVM IR`, se *linkedita* el `.ll` con las librerías estáticas del lenguaje.

Cabe aclarar que, por la naturaleza dinámica de los tipos en `Sauerkraut`, hubiera sido complejo realizar operaciones directamente en código `LLVM IR`, tal es así que la totalidad del código que involucre interactuar con los objetos se resuelve mediante llamadas a la librería estándar `sklib`, no directamente en el archivo .ll.

Esta librería provee una serie de funciones que actúan de interfaz entre el código compilado en `llvm` y el *back end* de `Sauerkraut`:

```
void *newIntegerObj(int i)
void *newStringObj(char * s)
void *newArrayObj(void ** e, int n)
void *newKVObjectObj(void ** keys,void ** vals, int n)
void *funcexec(void * obj, char * name, void ** args, int nArgs)
```

Todas las funciones de la interfaz retornan un puntero `struct Object`, pero frente a `llvm` y `clang` lo único relevante es que devuelven `i8*`.

Como nota adicional, las versiones de `llvm` y `clang` utilizadas son la 4.0, pero es posible que el compilador pueda ser compilado en otras versiones.

3. Descripción del desarrollo del trabajo

En una instancia inicial, fue central el hecho de definir una gramática sobre la cuál se desarrollaría el lenguaje. Esto no solo incluyó el definir la sintaxis del mismo, sino el establecer las estructuras que el lenguaje soportaría por defecto.

Una vez definido esto, se procedió a transcribir la gramática a un formato estándar (BNF), y a diseñar la lógica interna de objetos y métodos. Esto llevó a la realización de una librería cuya compilación es independiente de la compilación del compilador en sí, siendo las funciones de la interfaz especificada más arriba el nexo entre los dos entornos. El *desligamiento* entre los dos entornos permitió una mejor división de tareas y una dinámica de trabajo por lo general ágil.

Por el lado de `llvm`, lo importante es notar que, al definir una nueva estructura, el proceso para que el compilador genere su código es relativamente simple: se debe(n) especificar el(los) nodo(s) para la traducción, añadir los atributos en el código en `llvm`, especificar la función de la interfaz utilizada para generar el puntero a la estructura en el archivo `corefn.cpp` y definir el(los) método(s) `codeGen` correspondiente(s) en `codegen.cpp`. Si todo se ha hecho correctamente, la estructura definida ya debería ser funcional.

Parte de la gracia de utilizar `llvm` radica en que no se tuvo que implementar tablas de simbolos propias, sino que se aprovecharon las funcionalidades que ofrece dicho entorno para tal fin.`Sauerkraut` cuenta con *scoping* en bloques para las variables tanto dentro de funciones como dentro de bloques `if` y `while`. Por otro lado, como limitación del lenguaje se puede mencionar que las declaración de funciones tiene alcance global.

Típicamente, al incluir una nueva estructura o funcionalidad, lo habitual fue escribir un fragmento de código en `Sauerkraut` que buscara evaluar su correcto funcionamiento. Esto podría eventualmente dar lugar a una serie de escenarios: que la aplicación no compile, que su ejecución finalice correctamente (`exit(0)`),

o bien que su terminación sea anormal (`exit(1)`), en el caso de que, por ejemplo, se busque ejecutar intencionalmente un método no existente para un tipo dado. Para corroborar rápidamente que todos los *tests* cuyo valor de retorno debe ser (`exit(0)`) ejecuten correctamente, se ha incluído un simple *script* de prueba que verifica esta condición, e informa los casos en los que, eventualmente, haya un fallo.

4. Descripción de la gramática

La gramática toma muchos elementos de la de `JavaScript`, sobre todo considerando la sintáxis de los objetos, en cuyo caso no existen diferencias con respecto al lenguaje mencionado. Sin embargo, a diferencia de `JavaScript`, no es posible definir una variable sin la palabra reservada `var` antepuesta al identificador, algo que en dicho lenguaje es posible.

De manera opcional, a modo de separación entre instrucciones, es posible colocar el tradicional `;` entre las mismas. La definición de funciones es idéntica a la de `JavaScript`, anteponiendo la palabra reservada `function` al identificador de la función, luego del mismo colocando la lista de variables que admite separados por `,`.

Tal como pudo observarse en el ejemplo expuesto más arriba, la invocación de métodos se realiza de la manera tradicional.

En el repositorio entregado se encontrará la gramática escrita en un formato BNF.

5. Dificultades encontradas

A pesar de las bondades de `llvm`, es de notar que, al menos durante el tiempo en el que se desarrolló el trabajo, muchos de los archivos de la documentación arrojaban un error `404 Not Found` al intentar acceder a los mismos, dificultando el entendimiento de su documentación. A los efectos de aprender su uso, hubo en gran medida que depender de fuentes y ejemplos en foros. Sin embargo, se pudo constatar que la versión más nueva de dicho entorno, la 5.0, posee una documentación mucho más rigurosa e incluso más didáctica.

Con respecto a la versión utilizada de `llvm`, durante gran parte del desarrollo el equipo se basó en la 3.8. Advirtiéndo que en Pampero la versión presente es la 4.0, se procedió a intentar compilar el proyecto en dicha versión, constatándose en el acto que la nueva versión no era, en efecto, retrocompatible. Exactamente dos cambios hubo que realizar: se cambió de la manera de acceder al *contexto*, que anteriormente se realizaba con el método `getGlobalContext()`, además del método que realizaba la generación de código intermedio.

6. Futuras extensiones

6.1. Garbage collection

Muchos de los lenguajes que sirvieron de inspiración para `Sauerkraut` implementan algún tipo de *garbage collection*. En el estado actual del lenguaje, no sólo las variables fuera de *scope* no se liberan, sino que es posible que haya *memory leaks* fruto de la constante instanciación de nuevos objetos en el `return` de los métodos implementados.

Una de las opciones sería el utilizar librerías conocidas de *garbage collection* en `C`, tales como la de `Boehm-Demers-Weiser`.

6.2. Funciones como objetos

Tal como se ha mencionado más arriba, una de las intenciones originales fue que este ítem fuese una característica del lenguaje. Este ítem presenta una serie de dificultades:

Por un lado, es necesario conocer a partir de qué dirección de memoria se encuentra compilada la función a los efectos de poder invocarla programáticamente. Por otro lado, se deben tener precauciones a la hora del pasaje de argumentos y de la verificación de que su orden y cantidad sea la correcta. En el código realizado, esto se ha planteado especificando una *signature* única para todas las funciones: `typedef Object * (*function)(void *,void**,int);`, siendo el primer argumento el objeto que responde a la invocación (en este caso sería el objeto `Method` en sí), y los siguientes dos los argumentos y su cantidad respectivamente.

6.3. Clases

Sería interesante el poder ofrecer la posibilidad de definir nuevas clases. Las dificultades asociadas a este ítem están vinculadas a las del ítem anterior: una clase seguramente responderá a una serie de métodos, los cuales deberán ser compilados de manera tal que puedan ser encontrados de la misma forma que lo son los métodos de los tipos *built-in*.

6.4. Otros tipos *built-in*

Dada la infraestructura provista por `llvm`, crear nuevos tipos con su sintáxis propia no sería un problema mayor. La librería estándar podría ofrecer, por ejemplo, un tipo `Deque` que haga las veces de *stack* y de *queue*.

Referencias

[1] *https* : *//github.com/mkfifo/linear_hash* : Librería de *hash tables* en `C` con un 93 % de cobertura de *tests*. Provista bajo la licencia MIT.

[2] *http* : *//gnuu.org/*2009/09/18/*writing* − *your* − *own* − *toy* − *compiler/* : Writing Your Own Toy Compiler Using Flex, Bison and LLVM.

[3] *http* : *//llvm.org/docs/tutorial/* Kaleidoscope: Implementing a Language with LLVM

[4] *https* : *//github.com/llvm*−*mirror/llvm/tree/master/examples/BrainF* : LLVM BrainF example.