Trabajo Práctico Especial

Teoría del Lenguaje, Autómatas y Compiladores

Segundo Cuatrimestre 2018

Integrantes:

- Tomás Ferrer - 57207
- Marcos Lund - 57159
- Guido Princ - 57334
- Juan Pablo Lo Coco - 57313

Titular:

- Juan Miguel Santos

Adjuntos:

- Ana María Arias Roig
- Rodrigo Ezequiel Ramele
- Cristian Adrián Ontivero

Índice

Introducción

El presente informe expone y profundiza sobre la construcción de un compilador para el lenguaje Frozone, ambos productos de este trabajo práctico. Asimismo se explicarán las decisiones tomadas y dificultades encontradas a lo largo de la implementación.

Para el desarrollo del trabajo se utilizaron las herramientas Lex, Yacc y GCC como lo sugiere el enunciado.

Idea subyacente y Objetivo del Lenguaje

Originalmente la idea era hacer un lenguaje tipado con sintaxis similar a la del lenguaje C, con un particular énfasis en la instrucción `goto` y las etiquetas relacionadas con esta. Sin embargo, se optó finalmente por hacer un lenguaje no tipado ya que esta idea resultó interesante y desafiante al utilizar como lenguaje de salida un lenguaje tipado. Haber traducido a Ruby, por ejemplo, hubiera resultado más sencillo.

Para que la idea pudiera ser desarrollada fue necesario inicialmente definir cómo estarían estructuradas las variables en el lenguaje de salida, y se decidió que las variables no consistan únicamente de un valor, sino que éstas tomarán un valor para cada tipo existente en el lenguaje diseñado. Entonces, en C, la estructura que compone a cada variable declarada en el lenguaje nuevo tendría un campo `int`, un campo `double`, un campo `char *` y un campo `bool` (donde `bool` es un `enum` definido como `FALSE = 0` y `TRUE = 1`), y todos estos campos serán modificados cada vez que se asigne un valor a la variable. Asimismo, las variables tienen también un campo que determina cuál de todos los tipos es el tipo base (esto es, el tipo preferido a usar cuando se utilice la variable).

Una novedad del lenguaje es que una instrucción termina cuando al llegar al “\n” por ende, no hace falta finalizar los comandos con un “ ; ”.

Los siguientes ejemplos resultan descriptivos y facilitan la comprensión de la idea:

Considérese la siguiente línea escrita en el lenguaje Frozone:

```
x = 5
```

En C, el código equivalente (sin tomar en cuenta el manejo de memoria) es:

```
Var x; // var es un puntero a una estructura VarCDT
x->i = 5;
x->d = 5.0;
sprintf(x->str, “%d”, 5); // se reserva espacio para esta
                          // operación
```

```
x->b = TRUE;
x->baseType = INT;
```

Ahora, si la línea fuera `x = 0`, la lógica sería la misma, con la excepción de que el valor booleano es FALSE.

Considérese la asignación de una cadena de caracteres:

```
x = "str"
```

Esto resultaría en algo similar a:

```
x->i = atoi("str");
x->d = atof("str");
strcpy(x->str, "str");
x->b = TRUE;
x->baseType = STR;
```

En este caso, el valor booleano es verdadero debido a que la cadena de caracteres no es vacía.

Consideraciones Realizadas

Todo programa arranca a ejecutarse con la función *frozone*. Esta función no tiene valor de retorno y será de la forma:

```
frozone ( ) {
        ….
}
```

Más allá de que el enunciado no lo especificara, se optó por hacer un lenguaje que soportara la creación y ejecución de funciones. Éstas necesariamente deben incluir prototipos. De lo contrario, sería obligatorio definir las funciones antes de que éstas puedan ser llamadas.

Los prototipos son declarados de la siguiente forma:

```
foo(3)
```

Donde `func` es el nombre de una función que recibe tres argumentos.

Los prototipos de funciones deben declararse obligatoriamente, con excepción a `frozone`, y al comienzo del archivo. Una vez que se comienzan a definir funciones, ya no puede declararse ningún prototipo adicional. Las funciones pueden estar definidas en cualquier orden, pero toda función que haya sido declarada mediante un prototipo debe también ser definida obligatoriamente para validar que no haya llamadas a funciones no definidas.

Las funciones son declaradas de la siguiente manera:

```
function foo(a,b,c)
```

Donde el identificador de la función es `foo`, y ésta recibe tres argumentos. Las líneas en las que haya llamados a función sólo pueden ser asignaciones a variables, y no puede aplicarse ninguna operación adicional en esa línea de código.

Por otro lado, el lenguaje Frozone cuenta con ciertos operadores lógicos y relacionales. A continuación se listan los mismos y sus respectivas equivalencias en C:

- `eq` → `==`
- `lt` → `<`
- `gt` → `>`
- `ne` → `!=`
- `and` → `&&`
- `or` → `||`
- `not` → `!`

Para utilizar los operadores lógicos no binarios (and, or y not) es necesario el uso de paréntesis entre los miembros.

Para operaciones aritméticas se cuenta con los operadores de suma (`+`), resta (`-`), multiplicación (`*`) y división (`/`).

Nota: para la resta es necesario dejar un espacio entre el caracter '-' y el literal de la derecha (si se restara un literal). De lo contrario, será tomado como el signo del literal y el programa no compilará por no encontrar un operador.

En caso de realizar una suma entre dos variables que sean de tipo cadena de caracteres se realizará la concatenación de los mismos.

Es importante entender que las operaciones aritméticas son binarias y que siempre se toma el tipo de la variable a la izquierda de la operación como base. Es decir, si por ejemplo se sumara una cadena con un entero (en ese orden), la operación a ejecutar será una concatenación, pero si se sumara un entero con una cadena, la operación no tendrá efecto. Se pensó en sumarle al entero el valor numérico equivalente de la cadena pero finalmente se decidió que una operación de ese estilo no tendría sentido alguno.

También se ignora la operación en casos de operaciones aritméticas con booleanos, operaciones con cadenas a excepción de la suma, entre otros.

Al asignar a una variable una operación aritmética, el tipo base de ésta será el tipo del operador izquierdo; si éste último fuera una variable, se tomará su tipo base.

```
x = 5
```

```
y = x + 2.5
print(y) % "Se imprime 7 al redondear el operador derecho al
tipo"
         % "base del izquierdo"
```

COMENTARIOS

Los comentarios son de la forma % Literal, esto es, cualquier constante, incluso cadenas de caracteres constantes.

```
% "Los siguientes son todos comentarios, y este también"
% 2
% 2.5
% true
```

IF

La estructura condicional del lenguaje comienza con la palabra reservada `on` seguida por la condición entre paréntesis, finalizando con la palabra reservada `do` y el código a ejecutar entre llaves en caso de que la condición sea verdadera. También se puede optar por incluir una cláusula `else` a continuación.

```
on (condition) do {
   ...
}
```

CYCLE

La sintaxis es muy similar a la de C.

```
cycle {
   ...
} on (condition)
```

PRINT

La sintaxis es del estilo Print (Identifier) o Print (Literal)

```
x = 5
print(x)
print(6)
```

SCAN

La sintaxis es del estilo Scan (Identifier, Literal). Esto es así para saber qué tipo de dato se le debe asignar a la variable ingresada. Dicha variable ya debe estar declarada. El literal puede ser un string, un integer, un double o un boolean.

```
a = 2
scan(a, 1)
```

Descripción del Desarrollo del Trabajo Práctico

Para el desarrollo del analizador lexicográfico se utilizó LEX (mediante la instalación de `flex 2.6.0`). En cuanto al analizador sintáctico, la herramienta utilizada fue YACC (`Bison 3.0.4`).

En primer lugar se definió la gramática a utilizar (luego, la misma se vería modificada, pero la base de la gramática original quedó intacta). Una vez definida la gramática fueron escritas las expresiones regulares que utilizaría el analizador lexicográfico.

Una vez terminadas estas tareas comenzó la etapa de desarrollo. Se comenzó por el analizador lexicográfico y luego se prosiguió con el analizador sintáctico.

Prácticamente desde el comienzo de desarrollo se decidió también implementar una tabla de símbolos para almacenar las declaraciones tanto de funciones como de variables, y para controlar el alcance de las mismas. La tabla de símbolos se organizó de la siguiente manera:

El scope global `gscope` posee una lista de funciones, que se va llenando a medida que se van encontrando prototipos de funciones. En la lista se guarda tanto el identificador de la función (este identificador no puede repetirse, obviamente) así como la cantidad de parámetros y las variables que se van definiendo en la definición de la misma.

La función principal (el punto de entrada) también tiene su entrada en la tabla de funciones (ocupa la primera posición) y es agregado desde LEX una vez encontrado. Nótese que esta función no tiene prototipo.

Debido al hecho de que la gramática definida tiene muchas producciones recursivas, la forma más conveniente encontrada para que el analizador sintáctico esté al tanto de cuál es la función que se está parseando, LEX modifica una variable almacenada, asignándole el índice que la función encontrada tiene en la tabla de funciones. Si no se encuentra en la tabla de funciones es porque no existía un prototipo para esta función, y la compilación falla. Entonces, al encontrar la definición de una función, se actualiza la entrada en la tabla de funciones, para advertir que esta función ha sido definida.

Cada vez que se encuentra una nueva variable, se agrega en la lista de variables de la función donde la misma es declarada.

Si del lado derecho de alguna expresión se encuentra alguna variable que aún no había sido definida, entonces la compilación falla.

Los valores de los símbolos no terminales de la gramática forman nodos con distintas propiedades. Los nodos básicos sólo cuentan con un string con el código traducido a C que se imprimirá por salida estándar (podría haberse obviado la creación de estos nodos y haber asignado el valor directamente a una cadena de caracteres). Las operaciones, entre otros no terminales, cuentan con un tipo diferenciado de nodos que permite además informar a sus padres el tipo que devuelve la operación en caso de ser una operación aritmética. Un tercer tipo de nodos cuenta con , además de la cadena a imprimir, un entero que sirve como contador, por ejemplo, del número de parámetros de una función.

Descripción de la gramática

A continuación se listan los símbolos no terminales con los que cuenta nuestra gramática, y se da una breve descripción de cada uno:

- **Program:** es el símbolo inicial de la gramática. Consta de prototipos de funciones por un lado y las definiciones de funciones por el otro.
- **FunctionPrototypes:** consta de una lista de prototipos de funciones. Puede ser vacía.
- **FunctionPrototype:** deriva en el nombre de la función y el número de parámetros que acepta entre paréntesis.
- **GlobalFunctionList:** es la lista de funciones definidas, explicitando que debe existir una función main. Hace posible sólo tener el main como función definida.
- **MainFunction:** definición de una función cuyo nombre debe coincidir con “frozone”.
- **FunctionList:** una lista de funciones, sin contar el main.
- **Function:** consta de un nombre dado, parámetros entre paréntesis y un cuerpo de función.
- **FunctionArguments:** permite tener o no parámetros en una función.
- **NonEmptyFunctionArguments:** es una lista de parámetros de la función en caso de contar con uno o más.
- **FunctionBody:** una lista de Statements que permite cuerpos de función vacíos.
- **Statement:** deriva en uno de los siete tipos de Statements definidos a continuación.
    - **VarDeclaration:** permite asignar literales, llamados a función y operaciones lógicas, relacionales y aritméticas a variables.
    - **OnStatement:** es el comando “if” de nuestro lenguaje. Acepta una condición a evaluar, seguida de un cuerpo de función y permite un “else” opcional a continuación.
    - **CycleStatement:** es el ciclo “do … while” de nuestro lenguaje. Acepta un cuerpo de función seguido de una condición a evaluar para seguir iterando o no.
    - **ReturnStatement:** permite retornar una variable o un literal de una función. Asegura que el main no pueda tener ningún “return”.

    - **PrintStatement:** permite imprimir por salida literales o variables, en cuyo caso imprimiría el tipo base de las mismas.
    - **ScanStatement:** permite leer input por entrada estándar.
    - **CommentStatement:** comienza con '%' seguido de un valor numérico, booleano, o string encerrado entre comillas dobles. Los comentarios son ignorados al generar el archivo C de salida.
- **Operation:** representa las operaciones aritméticas del lenguaje que pueden darse entre cualquier combinación de variables y literales, siempre y cuando sean binarias.
- **Literal:** permite definir valores numéricos de tipo entero y punto flotante así como también booleanos, y cadenas de caracteres encerrados entre comillas dobles. Nota: los literales negativos (tanto enteros como double) deben declararse con el signo '-' justo antes del primer dígito.
- **Condition:** permite el uso de operadores relacionales entre variables o entre variables y literales. También acepta operadores lógicos binarios entre condiciones, entre condiciones y variables o entre variables, y operadores lógicos unarios con condiciones o variables. Las condiciones van siempre encerradas por paréntesis. Son binarias si se usan operadores relacionales, a menos que consten de una sola variable o literal.
- **FunctionCall:** consta del nombre de función a llamar y sus argumentos. Se verifica que dicha función haya sido declarada en la sección de prototipos y que el número de argumentos dado coincida.
- **FunctionCallArgs:** una lista de argumentos a pasar a la función, que puede estar vacía.
- **NonEmptyFunctionCallArgs:** una lista no vacía de argumentos a pasar a una función cuando es llamada.

Dificultades Encontradas en el Desarrollo del Trabajo Práctico

La primera dificultad surgida tuvo lugar al leer la consigna, ya que, a pesar de que ésta es lo suficientemente clara, fue difícil de comprender. Inicialmente se consideró erróneamente que el programa a desarrollar sería similar al ejemplo de la calculadora dado en clase. Una vez descartada esta idea, la dificultad fue entender si el compilador a desarrollar tendría que generar directamente bytecode, o si tendría que generar código intermedio que luego fuera compilado por un compilador, que sería variable en función del lenguaje intermedio escogido.

Además, una vez comprendida la idea, resultó evidente la necesidad de implementar una robusta tabla de símbolos, que tuviera en cuenta las funciones y sus correspondientes variables. No obstante, dado que la gramática es recursiva, fue difícil pensar cómo podría saberse el alcance de la línea de código que es parseada en cierto instante. Una solución considerada (aunque no implementada) podría haber sido guardar las variables en un alcance auxiliar a medida que se van leyendo las líneas de código, y, una vez terminada la función, asignar el scope auxiliar al de la función recién parseada. No obstante, esta solución presenta un inconveniente: si el nombre de la función fuera inválido (por haber sido

utilizado anteriormente, por ejemplo), y, a la vez, existieran errores en el código, se mostrarían primero los errores en el código y, una vez corregidos estos, aparecería el error en el nombre de la función.

Entonces, se decidió que LEX modifique una variable cada vez que entra a una nueva función, asignando en esta la nueva función encontrada. Asimismo, LEX reportaría de inmediato los nombres de función inválidos.

Otro problema surgido fue cómo lograr que una función pudiera incluir llamados a otra función declarada luego en el código, ya que resultaría imposible saber si se trata de un llamado válido o no. La mejor solución a esto, así como a llamados con cantidades de argumentos inválidas, fue la implementación de prototipos.

Futuras Extensiones

Una extensión extremadamente útil y que hubiera sido deseable incluir en el trabajo práctico es la posibilidad de tener arreglos, claramente no tipados. Un lenguaje de programación que carece de esto pierde mucha potencia. Esto no necesariamente es tan complejo; es posible que con cambiar el tipo char * a void * o con agregar un campo del tipo Var * en la estructura de datos que compone cada variable sea suficiente.

También hubiera sido deseable tener un mejor manejo de memoria, ya que tanto el compilador como el programa compilado tienen memory leaks. Esta tarea probablemente tampoco sea demasiado compleja; es cuestión de revisar el código con mucha atención y mucho tiempo de dedicación.

Se podría haber agregado una lista de palabras reservadas en C para prevenir que los nombres de las variables pueden tomar dichos valores y evitar errores de compilación del archivo C de salida.

Referencias

1. https://github.com/faturita/YetAnotherCompilerClass
2. https://github.com/faturita/LlvmBasicCompiler
3. https://www.gnu.org/software/make/manual/make.html
4. http://www.goldparser.org/doc/grammars/conflict-shift-reduce.htm
5. https://stackoverflow.com/questions/37186142/goto-label-in-the-same-loop-in-bison
6. https://stackoverflow.com/questions/15665828/how-to-make-semantic-check-using-flex-bison