Reporte 2. Analizador semántico para el Lenguaje C+-

Matías Greco, Javier Reyes

9 de Noviembre, 2018

Introducción

El presente reporte explica el trabajo realizado para el desarrollo de un anailzador semántico para el lenguaje de programación `C+-`.

El lenguaje `C+-` corresponde a un subconjunto del lenguaje `C`, con la adición de algunas características de `C++`, como la posibilidad de definir una función con paso por valor o paso por referencia.

El análisis ' léxico y sintáctico de este lenguaje de programacin fue desarrollado previamente, obtenido un arbol sintctico abstracto (AST) el cual se recorre nodo a nodo. Dicho analizador fue desarrollado en la herramienta ANTLR4 (ANother Tool for Language Recognition).

El analizador semántico fue desarrollado en el lenguaje de programacin Python y su salida es una tabla de símbolos en la cual se encuentran todos los símbolos necesarios para desarrollar el análisis semántico correspondiente, y donde cada nodo o sub-arbol representa un ámbito dentro del programa.

El repositorio del proyecto est disponible en github[1].

Características del lenguaje

El lenguaje `C+-` incluye las siguientes características, propias del standard `C89`

- Funciones.
- Declaraciones.
- Asignaciones.
- Expresiones lógicas y de operaciones.
- If.
- Switch (case y default).
- While.

[1] https://github.com/matgreco/Cmm-Compiler

- For.
- Do While.
- Struct.

Algunas características interesantes incluidas:

- Comma expression

Caracteríticas no incluidas

- Macros:

  No se incluyó debido a que complica todo el proceso. Requeriría una precompilación y la capacidad de incluir otros archivos mediante enlazado.

- Punteros:

  No se incluyó debido a que complica la gramática, por la aparición de una infinidad de tipos distintos (del estilo `int***`). También requiere una administración de memoria que escapa de nuestro objetivo principal.

- Typedef:

  No se incluyó debido a que complica la definición de la gramática. Eso permitiría utilizar una expresión de tipo `VAR` como una definición de tipo. Al no incluirlo, el lenguaje no pierde capacidades, ya que una estructura personalizada `S` tiene tipo `struct S`.

Tabla de símbolos

En el proceso de compilación, luego de la generación del AST, es necesario conocer una serie de nombres y cierta información de para realizar el proceso de análisis semántico. Para esto, los compiladores prefieren almacenar y luego recuperar la información obtenida en las lineas anteriores del programa en vez de buscarlas o recomputarlas nuevamente. Para esto existe la tabla de simbolos, que en general corresponde a una representación de un arbol cuyos son ámbitos que almacenan tablas de hash.

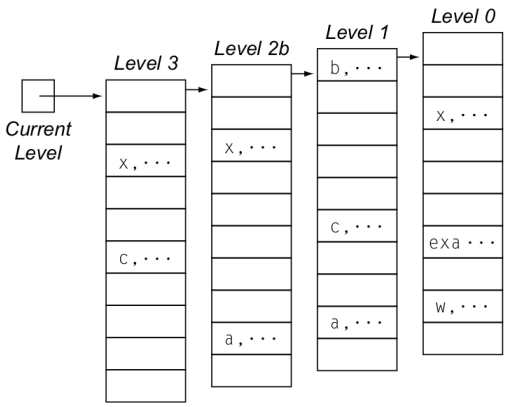

Figura 1: Representación de una tabla de simbolos. Fuente: (Cooper, 2012)

Cada tabla de hash representa la información contenida dentro de un ámbito, y dicha información solo será accesible desde los nodos sucesores. La raíz del arbol representa el ámbito global.

Para esta implementación se utilizó la biblioteca de Python AnyTree y la estructura de datos diccionario.

Representacón de la Tabla de Simbolos y Reglas semánticas

El principal comportamiento del analizador semántico es que cada paréntesis encargado de iniciar un ámbito, generará un nuevo nodo dentro del árbol de la tabla de símbolos y al momento de cerrar el ámbito volverá a su nodo anterior.

Cada nodo corresponde a un diccionario que en su índice contiene el nombre del símbolo y almacena algunos datos propios de la definición encontrada en el formáto de una estructura llamadá `simbol`.

La estructura `simbol` contiene los siguientes atributos:

- **SType:** Define el tipo de símbolo encontrado. Ejemplo: 'función', 'struct', 'variable', 'valor'.
- **Name:** Define el nombre del simbolo encontrado. Este es coincidente con el índice con el cual se agrega al diccionario.
- **VType:** Define el tipo de dato correspondiente. En el caso de una función contendrá el tipo que retorna, en el caso de una definición el tipo del cual es propia la variable.
- **Info:** Contiene antecedentes adicionales del símbolo. Por ejemplo, el número de argumentos de una función o el numero de datos miembro una estructura.
- **Line:** Corresponde a la línea en la cual fue encontrado el símbolo. Será útil para retornar mensajes de error como: R̈edeclaración de la variable X, ya fue definida en la línea Y.̈

Una particularidad de nuestro analizador semántico es que a la tabla de símbolos tambien agrega los ciclos y las funciones como un nodo del árbol. Esto permite detectar el tipo de dato a retornar por la función o si el ámbito corresponde a un ciclo, con la intención de hacer verificaciones como la instrucción continue o break estén dentro de un ciclo, o si una función de tipo void tiene un valor en su instrucción return.

Ejemplo de una tabla de símbolos con ciclos.

```
int main(){
    while(1){
        {
            break;
            while(0){
                continue;
            }
        }
    }
    return 0;
}
```

Este programa construye la siguiente tabla de símbolos:

```
{'main': (int)}
  {'%return': 'int'}
      {'%loop': 'while'}
                  {'%loop': 'while'}
```

Algúnas de los errores semánticos que es capaz de detectar este analizador son los siguientes:

- Variables no declaradas o no accesibles desde el ámbito correspondiente.
- Redeclaración de variables dentro del mismo ámbito.
- Estructuras no definidas y operaciones con estructuras.
- Retornar valor en función void y asignación desde función void.
- Instrucciones de control de flujo continue o break fuera de ciclos.

Entre otros.

Restricciónes adicionales de la gramtica

Existen dos tipos de instrucciones que fueron restringidos de la gramtica admitida por el lenguaje.

- **Switch - case:** Debido a que se complicaba el funcionamiento del analizador semntico.
- **Operadores ++ y −** Dado que al ir acompaado de una variable, era difícil su identificacin.

Ambas restricciones no le quitan capacidades al lenguaje y simplificaron la construccin del analizador semntico.

Pruebas

En el repositorio del proyecto se encuentras nueve códigos de prueba en lenguaje C+-, tres de los cuales construyen la tabla de símbolos sin detectar errores en el cdigo ingresado.

Entre las pruebas con error, se encuentran los siguientes:

- **Error semntico funcin:** El cdigo contiene una funcin de tipo VOID que retorna un valor. Adems se le asigna su resultado (de tipo VOID) a una variable, se llama a la funcin que requiere argumentos y no se le entregan argumentos, y se realizan operacines entre dos llamadas a la funcin.
- **Error semntico loop:** Contiene la expresin return sin un valor, dentro de la funcin main de tipo INT; tambien contiene las instrucciones continue y break fuera de un loop.
- **Error semntico struct:** Genera una instancia de una estructura que no existe; como tambien se realiza una operacin con una instancia de una estructura.

Entre las pruebas sin error, se encuenta:

- **Test:** Es un programa que contiene muchas de las funcionalidades del lenguajes, entre las cuales se encuentra la definicion de variables globales, definicion de funciones que retornar un valor y tambien de tipo void, declaracin de variables locales dentro de distintos ambitos con sus correspondientes operaciones; tambien contiene ciclos while y for correctamente definidos.
- **Test Scopes:** Contiene una prueba de definicin de distintas variables dentro de varios scopes. Tambien se aprecia que permite redefinir una variable pero dentro de otro mbito que la contenga.

Un ejemplo de un extracto de código existente entre las pruebas es el siguiente que detalla la construcción de la tabla de símbolos y sus correspondientes ámbitos es el siguiente:

```
void main(){
    int a;      // ok
    int b = 10; // ok
    a = 10;
    int x;
    x = 20;
    {
        int x;
        x = 30;
        {
            int a;
            a = 40;
        }
    }
}
```

El código presentado, al pasar por el analizador semántico, construye la siguiente tabla de símbolos.

```
 {'main': (void)}
  {'%return': 'void', 'a': (int), 'b': (int), 'x': (int)}
      {'x': (int)}
          {'a': (int)}
```

En ella se aprecia la definición de la variable a y x en los distintos mbitos.

Conclusiones

En este reporte se presentó el analizador semántico construido para el lenguaje llamado C+-. Dicho trabajo se realizó construyendo un árbol donde cada nodo representa un ámbito diferente del programa y permite verificar la visibilidad de las variables a través de los ámbitos antecesores como también para evitar la redeclaración de variables. En términos generales, en analizador semántico es robusto y logra detectar una gran cantidad de errores de distinto tipo sin obtener falsos positivos.

APENDICE

Definiciones léxicas

La definición de los componentes léxicos del lenguaje `C+-` es similar al lenguaje `C`, y se define de la siguiente forma:

- **Keywords:** `int`, `char`, `double`, `float`, `long`, `short`, `unsigned`, `sizeof`, `if`, `else`, `while`, `for`, `break`, `continue`, `true`, `false`, `struct`, `void`, `return`, `switch`, `case`, `default`, `do`. Tienen el mismo uso que en `C`.
- **Identificadores:** Puede componerse de letras, números y guiones bajos, pero no pueden empezar con un número.
- **Valores constantes:** Pueden ser números enteros con o sin signo (expresables en base 8, 10 y 16), números de punto flotante, caracteres y strings.
- **Operadores aritméticos:** `+` para suma, `-` para resta, `*` para multiplicación `/` para división y `%` para el resto de la división.
- **Operadores de comparación:** `==`, `!=`, `<=`, `>=`, `<`, `>`.
- **Operadores unarios:** `++`, `--`, `+`, `-`, `!`, $\sim$.
- **Operadores de shift:** `<<`, `>>`.
- **Operadores bitwise:** `&`, `^`, `|`.
- **Operadores lógicos:** `&&`, `||`.
- **Operador ternario:** `? :`
- **Operador coma:** `exp1,exp2` ejecuta `exp1`, luego `exp2` y retorna `exp2`.
- **Operadores varios:** `sizeof` retorna el tamaño en bytes de una expresión o tipo; llamadas a métodos (`f(exp1,exp2)`); acceso a miembros (`estructura.miembro`); y acceso a elementos de un array (`arr[i]`).