Compiladores
Proyecto 2: *Análisis Semántico*

Carlos Acosta Karla Esquivel
Yuan Yuan Luis Mayo
Andrea González

Facultad de Ciencias UNAM
2017-2

Índice

1. Sistema de tipos

Desde que definimos la gramática para nuestro lenguaje en el *proyecto 1* se habían contemplado los tres tipos que se mencionan en la especificación de este proyecto: tenemos números enteros de tipo `int`, números con punto flotante de tipo `float` y valores booleanos `bool`. La sintáxis del lenguaje también permitía que se declararan variables con tipo estático.
Gracias a que nuestro lenguaje ya cumplía con estos requerimentos, no tuvimos que modificar la sintáxis del lenguaje en estos campos.

2. Tabla de simbolos

La *tabla de simbolos* es usada para manejar los tipos de las variables y de los valores que regresen las funciones. Dicha tabla se crea durante el análisis sintáctico usando el patrón *visitor* con la clase **VisitorCreaTabla** para el descubrimiento de variables, las cuales deben ser inicializadas cuando son declaradas, pero cumplen con la regla de "declaración antes de uso".
Las funciones pueden regresar[1] algún valor de tipo `int`, `float` o `bool` o no regresar ninguno (usando `void`).

2.1. Implementación

La tabla de simbolos fue implementada como una estructura de datos en forma de árbol donde la raíz es el alcance principal del programa y cada nuevo alcance que se abra será un hijo para el alcance en el que se abrió.

2.1.1. métodos principales

- `look_up(name)`: regresa el nodo que contiene la declaración válida de *name* en el alcance actual. Si no hay declaración, regresa un apuntador nulo.
- `insert(name, simbol)`: ingresa *name* en la tabla de simbolos bajo el alcance actual, *simbol* contiene los atributos obtenidos en la declaración de *name*.
- `open_scope()`: abre un nuevo alcance para la tabla de símbolos, es decir, crea un nodo hijo donde se ingresarán nuevos simbolos.
- `close_scope()`: cierra el alcance más reciente, revirtiendo las referencias a simbolos hacia el alcance externo, es decir, a su nodo padre.
- `declared_locally(name)`: verifica que name esté declarado en el alcance actual y regresa `true` si lo está, en caso contrario regresa `false`.

[1] La sentencia `return` solo puede ser usada dentro de una función, en cualquier otro caso habrá un error de sintáxis

3. Verificación de tipos

Usando el patrón *visitor* con la clase **VisitorVerificaTipos** y la *tabla de simbolos* creada anteriormente.

4. Análisis dependiente del contexto

Además de la verificación de tipos, el análisis semántico de este compilador está implementado de modo que para que un programa sea válido para nuestro lenguaje debe cumplir los siguientes requisitos.

- Solo se puede llamar a funciones que sean definidas previamente.
- El número de parámetros que recibe la llamada de una función debe ser el mismo que el número de parámetros que se definieron en la firma de dicha función.
- Los tipos de los parámetros que recibe la llamada de una función deben ser los mismos que los definidos en la firma de la función.
- En el caso de una asignación, el valor de retorno de la función llamada debe ser del mismo tipo de la variable a la que se está asignando.

5. Instrucciones para ejecutar y compilar el proyecto

El código fuente del proyecto, se encuentra dentro del directorio `src/`. Debe asegurarse que tenga esta forma mínima para su correcto funcionamiento:

```
src/
|-- lexer.lex
|-- Makefile
|-- nodo.cpp
|-- nodo.h
|-- parser.y
|-- tabla.cpp
|-- tabla.h
|-- visitor.cpp
|__ visitor.h
```

5.1. Compilación del proyecto

En cualquier sistema operativo basado en Unix, con `g++`, `flex` y `bison` preinstalados, basta con ejecutar **make** en el directorio `src/` desde línea de comandos.

```
[user@host src]$ make
```

Dicha orden producirá un binario ejecutable de nombre ***kyc-ip*** listo para recibir código fuente de nuestro lenguaje.
En caso de existir un problema para generar dicho ejecutable, recomendamos volver a generar el código de los analizadores sintácticos y volver a intentar construir el ejecutable con las siguientes instrucciones:

```
[user@host src]$ make flex
[user@host src]$ make bison
[user@host src]$ make
```

5.2. Ejecución del analizador

Una vez obtenido el binario, basta:

```
[user@host src]$ ./kyc-ip <archivo>
```

Donde `<archivo>` es el archivo de texto claro con el código fuente correspondiente a nuestro lenguaje de programación.

5.3. Ejemplo de un programa no válido

```
fun cmp (int a, int b) int:
    int val = 0;
    cond (a > b):
        val = 1 and true;
    | (a < b):
        val = -1 and false;
    | default:
        val = 0 * 1.23;
    ~cond
    return val;
~fun

fun gungi () void:
    int a = 100;
    int b = a/2;
    int c = 1;
    bool d = false;
    while (c > 0):
        a = (a*b)/0.2;
        b = b*b or d;
        c = cmp(a,b);
    ~while
~fun
```

5.4. Ejemplo de un programa válido

```
fun cmp (int a, int b) int:
    int val = 0;
    cond (a > b):
        val = 1;
    | (a < b):
        val = -1;
    | default:
        val = 0;
    ~cond
    return val;
~fun

fun gungi () void:
    int a = 100;
    int b = a/2;
    int c = 1;
    while (c > 0):
        a = a*b;
        b = b*b;
        c = cmp(a,b);
    ~while
~fun
```