Proyecto 3: Analizador dependiente del contexto.

Compiladores 2017-1

Diana Olivia Montes Aguilar

November 2, 2016

1 Descripción

Hasta el momento hemos verificado que el código fuente sea válido de acuerdo a la gramática del lenguaje, pero tenemos que recordar que el análisis libre del contexto tiene un enfoque en categorías sintácticas más que en palabras específicas. Por ejemplo, valida que aparezca un átomo identificador en el lugar adecuado según la gramática, pero no se asegura que la instancia de ese identificador ya haya sido definida anteriormente. Si se quisiera validar la *definición antes del uso* de una variable únicamente con la sintaxis, la gramática no podría ser libre del contexto; se requeriría un nivel mayor de detalle. Pero esto implicaría mayor complejidad del reconocimiento. Por lo que se opta por técnicas *ad-hoc* para llevar a cabo esta tarea.

La etapa anterior concluyó con la elaboración de una representación intermedia, el AST. En abstracto, cada uno de los componentes del árbol es un *Nodo*, pero cada uno pertenece a una subclase con comportamientos y atributos muy particulares.

Sobre esta estructura, el AST, realizaremos, con recorridos, el análisis dependiente del contexto y la generación de código. Es claro que cada una de las instancias de las diferentes subclases debe ser analizada según su tipo. La manera ingenua es: identificar el tipo particular del nodo y operarlo en consecuencia. La implementación correspondiente sería un *switch case* con casos en igualdad de número a las subclases de nodos para cada uno de los recorridos que se realicen.

El patrón Visitor provee una manera modular y transparente de definir las acciones específicas que se requieren realizar con cada uno de los tipos específicos de nodos en cada recorrido realizado sobre el árbol.

2 Patrón Visitante

El patrón Visitante será la interfaz encargada de personalizar el análisis dependiente del contexto por cada nodo.

El patrón Visitante está conformado por las siguientes clases:

1. **Visitante Abstracto**: define un método abstracto de visita para cada uno de los elementos concretos.

```
public interface Visitante {
    public void visita(Nodo n);
    public void visita(HojaEntera h);
    ...
    public void visita(StmtNodo s);
}
```

2. **Visitante Concreto**: implementa la interfaz `Visitante` con las acciones que se desean realizar cuando se esté visitando cada elemento concreto.

```
public class VisitantePrint implements Visitante {
    public void visita(Nodo n){
        System.out.print("Nodo Genérico");
    }
    public void visita(HojaEntera h){
        System.out.print("Hoja Entera");
        System.out.print("valor: " + h.valor);
    }
    ...
    public void visita(StmtNodo s){
        System.out.print("Nodo Sentencia");
          // código para imprimir los hijos que
          // pueda tener.
    }
}
```

3. **Elemento**: Es la clase Nodo. Es decir, el objeto genérico que se visita.

```
public class Nodo{
  ...
  public void acepta(Visitante v){
```

```
        v.visita(this);
    }
}
```

4. **Elemento Concreto**: Cada uno de los tipos específicos de nodos. En cada una de las clases deberá haber un método que permita la entrada del `Visitante`.

```
public class HojoEntera{
    ...
    public void acepta(Visitante v){
        v.visita(this);
    }
}
```

3 Análisis dependiente del contexto

Lo que se verificará durante este análisis es lo siguiente:

- Que todo identificador sea definido antes de ser usado. En caso contrario, reportar error.
- Que los operadores se utilicen con tipos adecuados, en caso contrario, reportar error. Es decir que las reglas del sistema de tipos sean respetadas.

Durante la creación de algunos nodos hoja, se puede conocer el tipo del mismo. Es decir, si creo una `HojaEntera` se puede determinar que su tipo es entero. Pero hay nodos que en el momento de su creación no puede determinar el tipo al que pertenecen, ejemplo: `x+4`, ya que no se conoce si el identificador `x` ya fue definido previamente. La etapa del análisis dependiente del contexto también se encarga de dotar de tipos a los nodos que no son hojas pero que son suceptibles a tener un tipo, como el ejemplo mencionado anteriormente. Lo anterior es posible porqué se cuenta con un depósito de información de los identificadores y sus tipos, la tabla de símbolos.

La característica de cambiar el tipo de un identificador no estará permitida, ya que requiere de un nivel de análisis mas sofisticado, como un algoritmo de inferencia de tipos. El valor con el que fue creado si podrá cambiar, pero deberá conservar el tipo.

4 Tabla de símbolos

La tabla de símbolos es la estructura que utilizaremos para almacenar el contexto en el que aparece cada identificador y conocer el tipo con el que fue creado.

Las funciones que debe soportar la tabla de símbolos `h` son las siguientes:

1. `LookUp(name)`: regresa el valor asociado a `name` en la tabla de símbolos o `null` en caso de que no haya sido encontrado en la tabla.
2. `Insert(name,info)`: guarda `info` en `h(name)`.

La tabla de símbolos será implementada con tablas hash de la biblioteca `import java.util.Hashtable` de java.

5 Sistema de tipos

Un sistema de tipos consiste en la definición de los mismos y las reglas de uso. Los tipos soportados por el lenguaje son :

1. Booleanos
2. Enteros
3. Reales
4. Cadenas

Las reglas de su uso deben definirse para cada operador, ejemplo: Para el operador suma (+):

	Booleano	**Entero**	**Real**	**Cadena**
Booleano	Booleano	Entero	Real	Cadena
Entero		Entero	Real	Cadena
Real			Real	Cadena
Cadena				Cadena

Sólo se define la mitad de la matriz porqué es simétrica.

6 Ejercicios:

1. Implementar la Tabla de símbolos.
2. Implementar las reglas del sistema de tipos del lenguaje.

3. Implementar el Visitante Abstracto.

4. Implementar el Visitante Concreto que se encargue de hacer el análisis dependiente del contexto sobre el AST.

7 Administrativos(1pts):

1. El proyecto deberá se entregado antes del 21.11.16.

2. El código deberá estar comentado.

3. Estará en /Proyectos/Proyecto3