Ámbitos y visibilidad

Albert Rubio

Procesadores de Lenguajes, FdI, UCM

Doble Grado Matématicas e Informática

Ámbitos y vinculación

Contenidos

Análisis semántico

La fase de análisis de la semántica estática se dedica a comprobar las *restricciones contextuales*.

- Reglas de ámbito.
- Reglas de visibilidad.
- Reglas de tipado.

Estas comprobaciones se hacen normalmente sobre el AST.

- Un programa es válido si cumple las restricciones contextuales. En tal caso se considera apto para ser traducido.
- En caso contrario, se muestran los errores encontrados y se rechaza.

Propiedades estáticas y dinámicas

- Propiedades estáticas: no dependen de la ejecución. Se pueden determinar en tiempo de compilación y se satisfacen en toda ejecución del programa.
  Ejemplo: el tipo de una variable en los lenguajes con tipado estricto.
- Propiedades dinámicas: dependen de la ejecución (pueden variar de una ejecución a otra). En general, no se pueden determinar en tiempo de compilación.
  Ejemplo: la dirección de una variable local de un procedimiento.

La semántica del lenguaje determina si las restricciones contextuales son propiedades estáticas o dinámicas. Por ejemplo, en C++ son estáticas pero en Python son dinámicas.

En este curso nos centraremos en las propiedades estáticas.

Vínculo

El *vínculo* ("binding") es la asociación entre un identificador y el objeto que designa.
Un identificador puede designar por ejemplo:

- una variable,
- una constante,
- un parámetro formal,
- una función, ...

El vínculo se produce en un punto concreto del texto, pero es en ejecución cuando los identificadores se vínculan a objetos concretos.

- un identificador puede designar más de un objeto en un punto del texto, por ejemplo en la ejecución de una función recursiva. Aunque se trata de distintas versiones del identificador.
- varios identificadores pueden estar vinculados al mismo objeto (alias), por ejemplo en presencia de punteros.

Tiempo de vida

El *tiempo de vida* de un objeto es el tiempo que transcurre entre que se crea y se destruye el objeto.

- objetos estáticos. Su tiempo de vida es toda la ejecución del programa. Por ejemplo los `static` de C++. El vínculo se produce estáticamente y no varía en ejecución.
- objetos asignados en la pila. Su tiempo de vida es el de la ejecución de la función al que pertenecen. Por ejemplo, variables locales.
- objetos asignados en el montón (heap). Su tiempo de vida va de su creación a su destrucción, que puede ser explícita (C++) o mediante *garbage collection* (Java).

Asignación de objetos en la pila

Al llamar a una función:

- se apila en la pila de ejecución un *marco (o bloque) de activación*.
- Tiene espacio para los parámetros (y el resultado), las variables locales y temporales de una ejecución concreta.
- Cuando el procedimiento acaba, el marco es desapilado y las variables dejan de existir.
- Se minimiza el consumo de memoria: solo es necesario espacio para los marcos de activación de la cadena de llamadas en curso.

Asignación de objetos en el montón

Se utiliza para la creación de estructuras de datos dinámicas, para las que no es posible predecir el tamaño en tiempo de compilación.

- La creación es normalmente explícita con una operación especial.
- La destrucción puede ser
  - Explícita con una operación especial:
    - En general más eficiente.
    - Requiere un sistema de gestión de memoria (evitar/reducir fragmentación).
    - Riesgo de pérdida de memoria (*memory leak*) por no liberar.
    - Riesgo de uso de punteros desvinculados. Acceso a memoria liberada.
  - Implícita mediante recolector de basura.
    - Detecta objetos no vinculados a ningún identificador. Puede requerir borrado en cascada.
    - También requiere gestión de memoria.
    - Es más seguro.
    - Al ser un proceso complejo puede ralentizar notablemente la ejecución.

Contenidos

Identificación definición/uso

Los identificadores se suelen introducir con una declaración.
Distinguiremos entre:

- *aparición de definición*, en su declaración.
- *apariciones de uso*, en el resto de las apariciones.

La fase de identificación (definición/uso) de los identificadores establece la relación entre las apariciones de uso y su aparición de definición.

Eso implica saber a que objeto esta designando cada identificador en un punto concreto del programa.

Este vínculo se basa en:

- las reglas de ámbito
- las reglas de visibilidad

del lenguaje.

Ámbito

Llamamos *ámbito* de una declaración a la porción del texto del programa en que está activa.

Ejemplos de ámbitos:

- todo el programa,
- una función,
- un bloque,
- una clase, ...

En la mayoría de lenguajes (y en este curso) los ámbitos son textuales y, por tanto, estáticos.

Notad que en algunos lenguajes los ámbitos son dinámicos y dependen de la cadena de llamadas en curso. Puede dar mucha flexibilidad pero es fácil cometer errores.

Sobrecarga

La *sobrecarga* consiste en que un mismo identificador puede utilizarse para designar objetos distintos dentro de un mismo ámbito.

Normalmente se permite en operaciones predefinidas del lenguaje ($+$, $*$, ...)
La operación concreta se determina mediante un análisis de tipos.

Los lenguajes modernos lo permiten en definiciones del usuario, como funciones o constantes. En funciones, nuevamente, el análisis de tipos determina la identificación.

En este caso, o bien

- la identificación asocia una lista de posibles definiciones a un identificador, o
- la identificación se realiza a la vez que el análisis de tipos.

Bloques

Los lenguajes modernos tienen *estructura de bloques*:

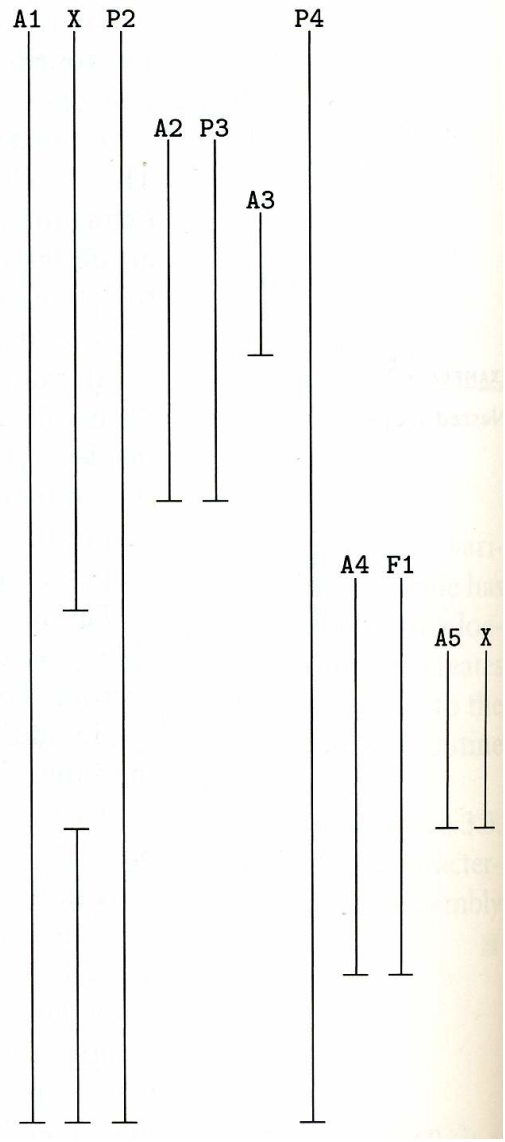
- Un vínculo está activo y es visible en el bloque en que está su declaración y en todos los bloques anidados en él.

  Puede haber apariciones de uso de las variables declaradas en cualquiera de los bloques ancestros.

- Un vínculo en un bloque más interno para un mismo identificador *oculta* el vínculo más externo y deja de ser visible hasta la finalización del bloque.

Bloques

```
procedure P1(A1 : T1);
var X : real;
    ...
    procedure P2(A2 : T2);
        ...
        procedure P3(A3 : T3);
        ...
        begin
            ...      (* body of P3 *)
        end;
        ...
    begin
        ...          (* body of P2 *)
    end;
    ...
    procedure P4(A4 : T4);
        ...
        function F1(A5 : T5) : T6;
        var X : integer;
        ...
        begin
            ...      (* body of F1 *)
        end;
        ...
    begin
        ...          (* body of P4 *)
    end;
    ...
begin
    ...              (* body of P1 *)
end
```



Bloques

En C++ tenemos algo parecido con las funciones anónimas:

```
#include <stdio.h>
int X = 0;

int F1 (int A1){
  int X=3;
  auto F2 = [X] (int A2){
    return X+A2;
  };
  return F2(A1);
}

int main () {
  printf("%d\n",F1(0));
}
```

Ámbitos anidados

- La implementación en tiempo de ejecución de los ámbitos anidados tiene cierta complejidad: lo veremos en generación de código.

  Requiere:

  - Asignar direcciones relativas al inicio.
  - Calcular el espacio máximo requerido.
  - Mantener una pila de marcos de activación con enlaces estáticos.

- Es necesario definir desde qué momento es válida la definición en un bloque.

  - Todo el bloque al que pertenece (Haskell)
  - Desde el punto de su declaración hasta el fin del bloque (C++). Esta opción simplifica la compilación.

Módulos/Clases

Existen otras construcciones que definen ámbitos de visibilidad:

- Módulos.
  - Por defecto los vínculos no son visibles fuera (importación/exportación explícita).
  - El tiempo de vida es todo el programa.
- Clases.
  - Introducen un ámbito global que circunda a todas sus funciones (métodos).
  - El tiempo de vida es la vida del objeto.

Contenidos

Reglas de visibilidad

- El vínculo más interno oculta al más externo: bloques, módulos, clases, ...
- Una cláusula de exportación puede hacer visibles fuera de un módulo identificadores declarados en ese módulo (solo para objetos estáticos).
- Una cláusula de importación puede hacer visibles en un módulo vínculos estáticos de otros módulos.
- La regla de ocultamiento no es aplicable en general a los vínculos importados (conflicto). Uso de nombres cualificados: `nombreModulo.identificador`
- Algunos lenguajes permiten crear regiones con visibilidad privilegiada. Por ejemplo el `with` de Python.
  Evitan uso excesivo de nombres cualificados.

Contenidos

Resultado de la identificación

El resultado del proceso de identificación puede anotarse de diversas formas en el AST:

1. Se anota cada aparición de uso con la información de la definición que le corresponde (no recomendable).
2. Se conecta cada aparición de uso con una referencia con el nodo definición que le corresponde.
3. Se conecta cada aparición de uso con una referencia a una tabla de símbolos donde se guarda la información de la definición que le corresponde.

La opción (2) es la más económica, aunque la (3) no es muy distinta.

En ambos casos utilizaremos tablas de símbolos, pero en (2) es solo para realizar el proceso.

Proceso de identificación

Supongamos que tomamos la opción (2).

- Par capturar las reglas de ámbito en una estructura de bloques podemos trabajar con una *pila de tablas de símbolos*. Cada tabla está asociada a un ámbito.
- En un momento del proceso, la pila muestra todos los niveles de anidamiento, siendo la cima las declaraciones del ámbito más reciente.
- Para encontrar un símbolo: recorremos la pila de tablas desde la cima.
- Recorremos el AST manteniendo la pila de tablas de símbolos con los símbolos definidos y añadiendo las referencias en los nodos del AST que contengan usos de identificadores.

Podemos expresar el proceso sobre la sintaxis abstracta (o como código Java).

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
   crea una pila de tablas vacía.

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
2. Cada vez que se entra en un nuevo ámbito
   `abreBloque()`
   que empieza un nuevo bloque apilando una nueva tabla vacía.

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
2. Cada vez que se entra en un nuevo ámbito
   `abreBloque()`
3. Cada vez que termina un ámbito
   `cierraBloque()`
   que desapila la tabla de la cima.

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
2. Cada vez que se entra en un nuevo ámbito
   `abreBloque()`
3. Cada vez que termina un ámbito
   `cierraBloque()`
4. Cada vez que encontramos una definición
   `insertaId (id,puntero)`
   que inserta `id` con su referencia al AST en la tabla de la cima.

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
2. Cada vez que se entra en un nuevo ámbito
   `abreBloque()`
3. Cada vez que termina un ámbito
   `cierraBloque()`
4. Cada vez que encontramos una definición
   `insertaId (id,puntero)`
5. Cada vez que encontramos un uso
   `puntero buscaId (id)`
   busca la primera aparición de `id` en la pila de tablas empezando por la cima y devuelve su referencia.

Proceso de identificación

Para realizar la identificación necesitamos las siguientes operaciones:

1. `inicializa()`
2. Cada vez que se entra en un nuevo ámbito
   `abreBloque()`
3. Cada vez que termina un ámbito
   `cierraBloque()`
4. Cada vez que encontramos una definición
   `insertaId (id,puntero)`
5. Cada vez que encontramos un uso
   `puntero buscaId (id)`

Se puede extender para tratar sobrecarga, importaciones, etc.