TEMA 5. TRADUCCION DIRIGIDA POR LA SINTAXIS

TEMA 6. TABLAS DE SÍMBOLOS

Francisco José Ribadas Pena

PROCESADORES DE LENGUAJES
4º Informática
ribadas@uvigo.es

13 de marzo de 2007

5.1 Sintaxis Concreta y Sintaxis Abstracta

- Uso de árboles sintácticos como interfaz entre A. Sintáctico y A. Semántico
- Para la semántica de un lenguaje hay detalles de la gramática no relevantes
  - Elementos para mejorar legibilidad
    - separadores, palabras clave, ...
  - Símbolos auxiliares para facilitar análisis
    - elimin. recursividad, factorización, precedencia, ...

Árboles de Sintaxis Concreta

- Representación directa de la secuencia de reglas gramaticales utilizadas durante el análsis
- Incluye todos los terminales y no terminales implicados
- No "cómoda" para utilizar directamente en A. Semántico

Árboles de Sintaxis Abstractas

- Representación condensada de la estructura del programa donde se eliminan detalles supérfluos
- Facilita el procesamiento semánticos

5.2 Gramáticas de Atributos

- CFGs no son suficientes para representar ciertas características de los lenguajes ni para realizar A. Semántico
- Necesidad de un formalismo más rico para el A. Semántico y la Gen. de Código
- **Gramáticas de Atributos**
  - *Las gram. de atributos son gramáticas de contexto libre a cuyos símbolos (terminales y no terminales) se les asocia un conjunto de atributos semánticos*
  - *Además, se definen reglas semánticas asociadas a las reglas gramaticales que determinan/calculan los valores de los atributos*
  - Aplicables sobre sintaxis abstracta y concreta
  - NOTACIÓN: $\begin{cases} \text{Atributos símbolo } \mathbf{X}: & \mathbf{X}.a_1, \mathbf{X}.a_2, \ldots \\ \text{Reglas semánticas } A \rightarrow \alpha: & \{ \texttt{ acciones } \} \end{cases}$
- Atributos Semánticos: Representan cualquier tipo de información asociada a los símbolos
  - *Ej.:* tipo, valor, secuencia de código generado, punteros a entradas TDS, ...
- Reglas Semánticas:
  - Cada regla gramatical tiene asociado un conjunto de reglas semánticas
  - Calculan el valor de los atributos en base a los valores de los demás atributos y al contexto (TDS)
  - Pueden incluir efectos laterales: emisión mensajes error, modificar TDS, generar código, ...

5.2.1 Tipos de Atributos

- Dada la regla "$\mathbf{A \rightarrow A_1\ A_2\ \ldots\ A_k}$", con $\mathbf{a, a_1, a_2, \ldots, a_n}$ atributos de los símbolos de la regla
- **Atributos Sintetizados:**
  - Su valor se calcula únicamente a partir de los valores de los atributos (sintetizados/heredados) pertenecientes a sus hijos en el árbol de análisis
  - $\mathbf{A.a = f(A_1.a_1, A_2.a_2, \ldots, A_k.a_k)}$
- **Atributos Heredados:**
  - Su valor se calcula a partir de los valores de los atributos (sintetizados/heredados) pertenecientes al padre o a los hermanos de ese nodo en el árbol de análisis
  - $\mathbf{A_i.a_i = f(A_1.a_1, A_2.a_2, \ldots, A_{i-1}.a_{i-1}, A_{i-1}.a_{i-1}, \ldots, A_k.a_k)}$

- *Ejemplos*

1. Atributos <u>sintetizados</u> y reglas para evaluación de expresiones aritméticas

Atributos: $\begin{cases} \mathbf{X}.valor \rightarrow \text{valor numérico de la subexpresión} \\ \mathbf{X}.texto \rightarrow \text{string asociado a un identificador} \end{cases}$

Sintaxis	**Semántica**
$E \rightarrow E + T$	$\{\ E_0.valor := E_1.valor + T.valor\ \}$
$E \rightarrow T$	$\{\ E.valor := T.valor\ \}$
$T \rightarrow T * F$	$\{\ T_0.valor := T_1.valor * F.valor\ \}$
$T \rightarrow F$	$\{\ T.valor := F.valor\ \}$
$F \rightarrow (E)$	$\{\ F.valor := E.valor\ \}$
$F \rightarrow entero$	$\{\ F.valor := entero.valor\ \}$
$F \rightarrow ident$	$\{\ F.valor := consultarTDS(ident.texto)\ \}$

2. Atributos <u>heredados</u> y reglas para propagación de tipos

Atributos: $\begin{cases} \mathbf{X}.tipo \rightarrow \text{expresión de tipo de dato} \\ \mathbf{X}.texto \rightarrow \text{string asociado a un identificador} \end{cases}$

Sintaxis	**Semántica**
$Declarac \rightarrow NombTipo : ListaIDs;$	$\{\ ListaIDS.tipo := NomTipo.tipo\ \}$
$ListaIDs \rightarrow ident$	$\{\ insertarTDS(ident.texto, ListaIDs.tipo)\ \}$
$ListaIDs \rightarrow ListaIDs, ident$	$\{\ ListaIDs_1.tipo := ListaIDs_0.tipo$ $insertarTDS(ident.texto, ListaIDs_0.tipo)\ \}$
$NomTipo \rightarrow float$	$\{\ NomTipo.tipo := "real"\ \}$
$NomTipo \rightarrow int$	$\{\ NomTipo.tipo := "entero"\ \}$

5.2.2 Evaluación de Atributos

- *"Árbol decorado":* árbol sintáctico (concreto/abstracto) con todos sus atributos evaluados.
- Dos posibilidades en evaluación de atributos
  1. A la vez que se realiza al A. Sintáctico
     - A medida que se va analizando el fichero de entrada se aplican las reglas semánticas
     - Muy dependiente de la estrategia de análisis (ascendente/descendente)
     - Puede utilizarse para generar árboles abstractos que serán procesados posteriormente
       - Almacenar nodos en los atributos de los símbolos
       - Al completar el reconocimiento de las reglas $\longrightarrow$ crear nodo para lado IZQ y enlazar con sus sucesores en el lado DER.
  2. Después del A. Sintáctico
     - Una vez construido el árbol de análisis (concreto/abstrato) se recorre –en el orden *"adecuado"*– para evaluar las reglas semánticas en sus nodos
     - Aproximación más general
- En ambos casos obtendremos un árbol sintáctico "decorado" (implícito o explícito).
  - Donde se habrán calculado todos los atributos asociados a los nodos
  - Servirá de base a las siguientes fases de A. Semántico/Gen. Código

5.3 Decoración de Árboles Sintácticos independiente del Análisis

- Las reglas semánticas de las gramáticas de atributos :
  - Establecen una relación funcional entre atributos
  - No determinan explicitamente el orden de evaluación de las reglas
- Para extraer el orden de evaluación se construye un **grafo de dependencias** para el programa analizado a partir de la gramática
- Construcción del grafo de dependencias:
  - La regla que calcula el atributo $\mathbf{A.a}$ se representa $\mathbf{A.a = f(a_1, a_2, ..., a_k)}$, siendo $\mathbf{a_i}$ los atributos de otros símbolos.
  - Al grafo de dependencias se le añadirán arcos de la forma $(\mathbf{a_i}, \mathbf{A.a})$ $\forall \mathbf{i}$, indicando que el atributo $\mathbf{A.a}$ no puede ser calculado hasta que se calculen todos los atributos $\mathbf{a_i}$.
- A partir del grafo se determina un orden de evaluación de las reglas semánticas
- Se aplican las reglas en ese orden para calular los valores de los atributos
- Esquema general:

- *Ejemplo:*

5.4 Decoración de Árboles Sintácticos durante el Análisis

- **Idea:** Aprovechar el proceso de reconocimiento sintáctico para evaluar, a la vez, los atributos.
- No se necesita disponer explicitamente del árbol de análisis
- Uso de pilas semánticas
  - No sólo pila de símbolos gramaticales
  - Se incluyen atributos semánticos asociados a los elementos de la pila
- Se utilizan los atributos de los símbolos presentes en la pila del analizador para calcular los nuevos atributos
  - Visión parcial y local del árbol sintáctico global
- El método de A. Sintáctico impone limitaciones a la hora de calcular atributos
  - Determina un recorrido del árbol
  - No se dispone de todos los atributos del árbol

(a) TIPOS DE GRAMÁTICAS DE ATRIBUTOS

1. Gramáticas *S-atribuidas*:
   - Sólo contienen atributos sintetizados.
2. Gramáticas *L-atribuidas*:
   - Todos los atributos heredados asociados a un símbolo X_i del lado derecho de $A \rightarrow X_1 X_2 \ldots X_i \ldots X_k$ dependen sólo de:
     - Los atributos heredados del antecedente A
     - Los atributos heredados y sintetizados de los símbolos a su IZQ, $X_1, X_2, \ldots X_{i-1}$:
3. Gramáticas *LC-atribuidas*: (LC: *"left corner"*)
   - Todos los atributos heredados de los símbolos del lado DER dependen **sólo** de los atributos (heredados o sintetizados) de sus hermanos IZQ.

5.4.1 Evaluación en algoritmos descendentes (LL(1))

- LL(1) realiza análisis descendente de IZQ a DER.
- Permite el análisis y evaluación simultánea de atributos para gramáticas L-atributidas (y LC-atribuídas).
- Dada la forma en que se ejecuta al algoritmo LL(1) y la definición de gram. L-atributida, se puede demostrar que en todo momento tendremos disponible, en posiciones conocidas de la pila, todos los valores de atributos necesarios para calcular los nuevos atributos
- Ejemplo de funcionamiento:
  - Uso de pila semántica para mantener los atributos asociados a los símbolos gramaticales
  - Manejo de la pila: *al predecir la regla* $X \longrightarrow Y\,Z$

    (1) PUSH*(atrib. heredados de* X*)*
    (2) PUSH*(atrib. heredados de* Y*)*
    (3) Después de reconocer Y*:* PUSH*(atrib. sintetizados de* Y*)*
    (4) PUSH*(atrib. heredados de* Z*)*
    (5) Después de reconocer Z*:* PUSH*(atrib. sintetizados de* Z*)*
    (6) Reconocido X*:* POP*(atrib. símbolos lado DER:* Y *y* Z*)*
    PUSH*(atrib. sintetizados de* X*)*

5.4.2 Evaluación en algoritmos ascendentes (LR(1))

- LR(1) realiza análisis ascendente salto-reducc. de IZQ a DER.
- También utiliza "pila semántica"
- En LR el símbolo de la IZQ de una regla no se mete en la pila hasta que se han reconocido todos sus hijos
  $\Rightarrow$ no es posible que un símbolo herede atributos de su padre
- Tipo de gramáticas evaluables:
  1. Si se soportan atributos sintetizados
     - Al realizar la reducción de una regla están disponibles en la pila los atributos de símbolos de la parte DER
     - Se podrá calcular el valor de los atributos sintetizados del símbolo del lado IZQ
  2. Es posible soporte limitado de atributos heredados
     - Sólo se podrá heredar de símbolos de la parte DER de la regla que estén actualmente en la pila
     - Es decir, se puede heredar de los hermanos IZQ del símbolo actual

Resumen:
- Por (1): LR soporta gram. S-atribuidas
- Por (1) y (2) LR soporta gram. LC-atribuidas

TEMA 6. Tablas de Simbolos

6.1 Introducción

Tabla de Símbolos (TDS): Estructura de datos usada por el compilador para asociar a cada símbolo(nombre) del programa una representación de su contenido semántico(atributos).

- Función análoga a un *diccionario*
- Mecanismo para representar el contexto en un programa
- Mantiene la info. que se ha recogido acerca de un símbolo desde el momento de su declaración
- Mecanismo utilizado en A. Semántico para implementar las restricciones típicas de los lenguajes de programación
  - Control de unicidad de identificadores
  - Asegurar que toda variable fue declarada antes de ser usada
  - Verificación de tipos
  - Implementación de reglas de ámbito (en leng. con estructura de bloques)

TDSs sólo existen durante compilación, no suelen incorporarse al fich. objeto.

- Se puede incorporar para facilitar depuración (acceso a variables por nombre, etc.)
- También, se incorpora en los intérpretes (alternan compilación y ejecución)

La info. guardada (campos) depende del elemento almacenado, del lenguaje y de la estructura del compilador:

- **Atributos en TDS**
  - Nombre del símbolo: string
  - Dirección en memoria: dir. donde se guardarán los valores de las variables durante la ejecución.
    - No dir. absolutas, sino relativas a una zona de memoria durante la ejecución
    - No contiene valores, sólo info. de almacenamiento
  - Tipo: codificación correspondiente al tipo de dato asociado al símbolo
  - Localización (nº línea, nombre fichero): Info. auxiliar para depuración.

  TDS puede estar inicializada con info. sobre símbolos especiales del lenguaje (palabras reservadas, funciones de librería, constantes predefinidas, etc...)
- **Operaciones en TDS (interfaz)**
  - Insertar un símbolo en la tabla (comprobando repeticiones)
  - Buscar y recuperar info. de un símbolo
  - Modificar info. de un símbolo
  - Borrar
  - En lenguajes con estructura de bloques:
    - NuevoBloque: inicio de un bloque
    - FinBloque: fin del ámbito de un bloque

6.2 Implementación de TDS

Determinar cómo implementar funciones insertar y buscar.

Opciones:

1. Tablas no ordenadas: arrays, listas,...
   - Fácil implementación
   - Poco eficientes
2. Tablas ordenadas: arrays y listas ordenadas, árboles binarios, tablas hash,...
   - Más eficientes y complejos.

6.2.1 Árboles Binarios de Búsqueda

Búsqueda e inserción rápida, si se mantienen balanceados

- Búsqueda $\mathcal{O}(log_2(n))$ en el mejor caso

Eficientes en términos de espacio

- Espacio (nº nodos) necesario proporcional al nº de símbolos a almacenar
- Tablas hash tienen tamaño fijo, independiente del nº de símbolos almacenado
- Ventajoso en sistemas con múltiples TDSs

6.2.2 Tablas HASH

Muy usados por su buena eficiencia (acceso directo en el mejor caso)

Idea: Mapear un espacio grande (nombres de símbolo) a un espacio de menor tamaño (TDS), que se corresponde con una tabla hash con un nº fijo de posiciones.

Funciones de *hashing*: Responsables del mapeo.

- Propiedades deseables:
  - $hash(n)$ depende sólo de n (nombre).
  - $hash(n)$ debe poder ser calculada rápidamente
  - Debe de distribuir los nombres de forma aleatoria y uniforme sobre las posiciones de la TDS
- **Colisiones:** Función hash asigna a 2 símbolos la misma posición $\Rightarrow$ Resolución de colisiones
  - Resolución lineal: Si $hash(n)$ está ocupado, probar $(hash(n) + 1) mod\ m$, $(hash(n) + 2) mod\ m$,..., con m=tamaño tabla
  - Resolución cuadrática: Probar con: $(hash(n) + 1^2) mod\ m$, $(hash(n) + 2^2) mod\ m$, $(hash(n) + 3^2) mod\ m$,...
  - Resolución con encadenamiento
    - Los nombres no se incluyen directamente en las entradas de la tabla hash
    - Las entradas de la tabla guardan un puntero a una lista de entradas
    - Nombres con idéntica posición en hash están en una lista enlazada
    - Evita degeneración a búsqueda lineal y minimiza espacio extra necesario
    - Evita un error fatal en caso de llenar la tabla completamente

6.3 TDS estructuradas en bloques

Lenguajes de prog. suelen estructurarse en bloques de código (subprogramas, paquetes, módulos, bloques,...)

- Es posible anidar bloques y acceder a símbolos de otros bloques (ámbitos de visibilidad)
- Esta estructuración condiciona la implementación de las TDS (y viceversa).

Toda línea de código pertenece a uno o más bloques:

- Bloque "más interno" $\longrightarrow$ ámbito actual
- Ámbito actual + bloques que lo incluyesen $\longrightarrow$ ámbitos abiertos
- Bloques que no engloban a la línea actual $\longrightarrow$ ámbitos cerrados

Reglas de Visibilidad:

- En un punto dado del programa, sólo los nombres declarados en el ámbito actual y en los demás ámbitos abiertos son accesibles.
- Si un nombre está declarado en más de un ámbito abierto, se usará la declaración "más interna"(la más cercana).
- Las nuevas declaraciones sólo tienen efecto en el ámbito actual (y en los incluidos en él)
- Todas las declaraciones de un ámbito cerrado son inaccesibles.
- NOTA:
  - Los nombres de los parámetros de un subprograma son locales al cuerpo de ese subprograma
  - El nombre del subprograma está incluido en el ámbito en el que se declara el subprograma.
    - Los subprogramas recursivos también deben incluirse en el ámbito de su propio cuerpo.

6.3.1 Aproximaciones para implementar TDS en leng. con estructura de bloques

1. **Una tabla individual para cada ámbito/bloque**
   - Necesidad de un mecanismo que asegure que las búsquedas de nombres siguen las reglas de visibilidad para bloques anidados.
   - Estructuración de las distintas TDS en forma de pila
     - El ámbito más interno estará en la cima
     - Al abrir un nuevo ámbito/bloque se añade su TDS a la pila
     - Cuando se sale de un ámbito/bloque se saca de la pila su TDS
       - En compiladores de 1 pasada $\longrightarrow$ se puede eliminar esa TDS
       - En compiladores de múltiples pasadas:
         - ◇ se guarda la TDS, asociándola al bloque
         - ◇ para cada TDS habrá un puntero apuntado a la TDS del bloque padre
         - ◇ da lugar a una estructura en forma de árbol
   - Para buscar un nombre se consulta la TDS de la cima y se va bajando hasta que:
     - se encuentra el ámbito cuya TDS contiene ese nombre
     - se agota la pila.
   - Ejemplo:
   - INCONVENIENTES:
     - Necesidad de buscar a través de múltiples TDSs
       $\longrightarrow$ vars. globales $\Rightarrow$ buscar en todas las TDS
     - Problemas con tablas hash (TDS de tamaño fijo)
       - Si tabla grande $\longrightarrow$ desaprovecha espacio si hay pocos símbolos
       - Si tabla pequeña $\longrightarrow$ muchas colisiones, se enlentece búsqueda (listas largas)
     - No problema con árboles de búsqueda

2. **Una única tabla global**
   - Todos los nombres, para todos los ámbitos, en un sóla tabla
   - Cada ámbito tiene asociado un número de ámbito
   - Un nombre puede aparecer varias veces, cada una con un nº de ámbito diferente
   - Búsqueda debe considerar el "número de ámbito actual"
   - Se apoya en una pila de números de ámbito
   - VENTAJAS E INCONVENIENTES
     - Implementación eficiente con tablas hash
       - Rápido y relativamente eficiente en espacio
       - Fácil añadir símbolos e identificar los que ya no son visibles
     - Manejo complejo con árboles de búsqueda
       - Eliminar símbolos al salir de un bloque $\Rightarrow$ recorrer todo el árbol
   - Ejemplo:

6.4 Extensiones de la Estructuración en Bloques

Alteraciones de reglas de visibilidad usuales que afectan al diseño de TDS

6.4.1 Campos y Registros

Nombres de campo sólo son visibles si están calificados con el nombre del registro (Pascal: `record`. C: `struct`)

Nombres de campo sólo deben ser únicos dentro del registro, pueden aparecer de nuevo en el ámbito sin causar conflicto.

Dos posibilidades de implementación:

1. **Asociar una TDS a cada tipo registro**
   - Cada tipo registro tiene su propia TDS para almacenar info. sobre sus campos
   - Será un atributo de la definición del tipo registro
   - No se añade a la pila de ámbitos
   - Cada nueva TDS de campos no amplía el conj. de símbolos que pueden aparecer, sino que lo sustituye.
   - Búsqueda en dos fases (`nomVar.nomCampo`):
     - Nombre de variable de tipo registro en TDS del ámbito actual
     - Nombre del campo en TDS del tipo registro.
   - Implementación simple.
   - Costosa en espacio (sobretodo son tablas hash)
2. **Uso de "número de registro"**
   - Asociar a cada tipo registro un número
   - Incluir nombres de campo en TDS del ámbito incluyendo el nº de registro al que pertenecen
   - Los nombres que no se refieran a campos tienen un nº registro 0.
   - Búsqueda de una referencia a un campo (`nomVar.nomCampo`)
     - Empieza buscado el nombre de la var. de tipo registro para recuperar el nº registro.
     - Se busca entre todos los símbolos con nombre "`nomCampo`",uno con ese nº.

6.4.2 Reglas de Exportación e Importación

En ciertos lenguajes es posible que determinados símbolos de un bloques/ámbito sean visibles desde otros módulos separados, más allá de las reglas de visibilidad clásicas.

- Ej.: unidades (Pascal), packages (ADA), módulos (Modula-2), clases (C++),...
- Utilidad: { Soportar modularidad / Permitir compilación separada

Reglas de importación/exportación permiten determinar selectivamente que símbolos han de ser conocidos entre diversos módulos.

- Pueden incluirse restricciones sobre esos símbolos importados/exportados. (Ej.: sólo lectura, lectura-escritura,...)

Se amplia interfaz de la TDS con 2 funciones:

- Exportación de símbolos:
  - Toma los símbolos de un ámbito (TDS) que, según las reglas de exportación, son exportables y los incorpora a otro ámbito distinto.
    $\longrightarrow$ a ámbito importador de nivel superior o fuera de ámbitos abiertos actuales
- Importación de símbolos:
  - Toma un nombre símbolo y lo busca en la TDS del ámbito que lo exporta.
  - Copia esa entrada en la TDS del ámbito que la importa.

Cuando los módulos importador/exportador están en distintas unidades de compilación ($\approx$ ficheros), se habla de compilación separada.

- Es necesario guardar parte de las TDS en los ficheros objeto generados, para poder recuperar esa info. al compilar el módulo importador.
- NOTA: En C, es el programador quien maneja de forma explícita la importación/exportación de símbolos entre distintos fichero fuente.
  $\longrightarrow$ uso fichero cabecera `.h` y declaraciones `extern`