Procesamiento Dirigido por la Sintaxis

Albert Rubio

Procesadores de Lenguajes, FdI, UCM

Doble Grado Matématicas e Informática

Procesamiento Dirigido por la Sintaxis

Contenidos

Análisis semántico

- El análisis léxico y sintáctico no permite comprobar:
  - Identificadores usados han sido declarados.
  - A que declaración están asociados (y por tanto que tipo tienen).
  - Las operaciones reciben parámetros del tipo adecuado.
  - Los accesos a un tipo registro (struct) son correctos.
  - Los accesos a un tipo array con varias dimensiones son correctos.
  - ...
- Todas ellas son comprobaciones estáticas (textuales).
- No permiten interpretar (ejecutar) o traducir el programa aceptado.

Para poder realizar todos estos tratamientos es necesario:

- Extender el formalismo de las gramáticas incontextuales con capacidades de descripción de procesamiento.
- Definir nuevos análisis para realizar los procesamientos necesarios.

Contenidos

Sintaxis Abstracta vs sintaxis concreta

Realizar el procesamiento sobre la sintaxis concreta implica que

- Cualquier cambio en la sintaxis obliga a revisar los procesos

Por esta razón se sigue un método basado en dos fases:

1. Primera fase: transformar las frases del lenguaje en árboles de sintaxis abstracta (AST) que refleja la estructura sintáctica sin tener en cuenta aspectos sintácticos concretos.
   - Por ejemplo, eliminar delimitadores (separadores, paréntesis, llaves en bloques).
   - Requiere:
     - Diseño de la sintaxis abstracta
     - Especificación de un constructor del AST mediante una *gramática de atributos*.
2. Segunda fase: implementa los distintos procesamientos sobre el AST.

Diseño de la sintaxis abstracta

Definición de los tipos de nodos que nos podemos encontrar en el árbol sintáctico (abstracto).

- Representar declaraciones
- Representar tipos
- Representar el programa y funciones
- Representación de instrucciones
- Representación de designadores (direcciones)
- Representación de expresiones
- ...

En la práctica se trata de definir el AST como estructura de datos

Ejemplo sintaxis abstracta

Suponed que tenemos un lenguaje sencillo donde:

- Tenemos un programa principal.
- Todas las declaraciones se hacen antes de las instrucciones.
- Se pueden declarar tipos, variables y procedimientos.
- Tenemos enteros, booleanos, arrays, registros y punteros.
- Podemos definir bloques de instrucciones con nuevas declaraciones al inicio.

Representación de programas:

Constructora	Descripción	Rep. sintáctica
$prog : List(Dec) \times List(Inst) \longrightarrow Prog$	Construye un programa a partir de una lista de declaraciones y una instrucción	*Ds && Is*

Ejemplo sintaxis abstracta

Representación de declaraciones

Constructora	Descripción	Rep. sintáctica
$decVar : Tipo \times String \longrightarrow Dec$	Construye una declaración de variable	$var\ id : T$
$decTipo : Tipo \times String \longrightarrow Dec$	Construye una declaración de tipo	$type\ id : T$
$decProc : String \times Proc \longrightarrow Dec$	Construye la declaración de un procedimiento a partir de: su nombre y su definición	$proc\ id : Proc$

Representación de procedimientos

Constructora	Descripción	Rep. simbólica
$defProc : List(Param) \times List(Dec) \times Inst \longrightarrow Proc$	Construye un procedimiento: lista de parámetros, declaraciones locales e instrucciones	$proc\ Ps\ Ds\ Is$
$param : (ref \mid val) \times String \times Tipo \longrightarrow Param$	Construye un parámetro: referencia o valor, nombre y tipo	$(ref \mid val)\ id : T$

Representación de expresiones de tipo

Constructora	Descripción	Rep. simbólica
$bool : Tipo$	Construye un tipo booleano	*bool*
$int : Tipo$	Construye un tipo entero	*int*
$tid : String \longrightarrow Tipo$	Contruye un tipo con el identificador dado	*tid id*
$array : Integer \times Tipo \longrightarrow Tipo$	Construye un tipo array	$T[num]$
$struct : List(String \times Tipo) \longrightarrow Tipo$	Construye un tipo registro	*struct Cs*
$pointer : Tipo \longrightarrow Tipo$	Construye un tipo puntero	*pointer T*

Representación de expresiones de designadores

Constructora	Descripción	Rep. simbólica
$var : String \longrightarrow Desig$	Designador a identificador	id
$indxElem : Desig \times Exp \longrightarrow Desig$	Designador a posición de array	$D[E]$
$selCampo : Desig \times String \longrightarrow Desig$	Designador a campo de registro	$D.id$
$deref : Desig \longrightarrow Desig$	Designador a la dirección de un designador dado	$D->$

Representación de expresiones de instrucciones

Constructora	Descripción	Rep. simbólica
$asig : Desig \times Exp \longrightarrow Inst$	Asignación	$D = E$
$print : Exp \longrightarrow Inst$	Escritura	$print\ E$
$read : Desig \longrightarrow Inst$	Lectura	$read\ E$
$new : Desig \longrightarrow Inst$	Reserva de memoria	$new\ D$
$del : Desig \longrightarrow Inst$	Borrado de memoria	$del\ D$
$if : Exp \times Inst \longrightarrow Inst$	Instrucción if	$if\ E\ I$
$ifElse : Exp \times Inst \times Inst \longrightarrow Inst$	Instrucción if-Else	$if\ E\ I_0\ I_1$
$while : Exp \times Inst \longrightarrow Inst$	Instrucción while	$while\ E\ I$
$bloque : List(Dec) \times List(Inst) \longrightarrow Inst$	Instrucción bloque	$\{Ds\ \&\&\ Is\}$
$call : Desig \times List(Exp) \longrightarrow Inst$	Llamada a procedimiento	$D(Ps)$

Representación de expresiones de cálculo de valores

Constructora	Descripción	Rep. simbólica
$true :\longrightarrow Exp$	Expresión true	*true*
$false :\longrightarrow Exp$	Expresión false	*false*
$null :\longrightarrow Exp$	Expresión null	*null*
$num : Integer \longrightarrow Exp$	Expresión número	*num*
$mem : Desig \longrightarrow Exp$	Expresión designador	*mem D*
$igual : Exp \times Exp \longrightarrow Exp$	Expresión igual	$E_0 = E_1$
...	...	...
$suma : Exp \times Exp \longrightarrow Exp$	Expresión suma	$E_0 + E_1$
...	...	...
$and : Exp \times Exp \longrightarrow Exp$	Expresión conjunción	E_0 *and* E_1
...	...	...

Contenidos

Formalismo de las gramáticas de atributos

Formalismo propuesto por Donald E. Knuth.

Permite añadir una semántica de traducción a las frases de nuestro lenguaje.

- El significado de un no terminal A que tiene asociada una regla $A \longrightarrow \alpha$ se obtiene a partir de la semántica de α
- Se asocian atributos a los no terminales.
- Se describe recursivamente el valor de los nuevos atributos a partir de los conocidos

Atributos sintetizados

Los *atributos sintetizados* de un no terminal se obtienen exclusivamente a partir de los atributos de los terminales y los atributos sintetizados de los no terminales de sus partes derechas.

- Asumimos que los terminales tienen asociados *atributos léxicos* obtenidos en el análisis léxico.
- Se asocia un conjunto de atributos sintetizados a cada no terminal.
- Se asocia a cada regla $A \longrightarrow \alpha$ un conjunto de ecuaciones semánticas que dicen cómo obtener los atributos de A a partir de los de α.
- Las ecuaciones son de la forma $a = f(a_1, \ldots, a_n)$ donde a es un atributo de A, los a_i son atributos de los no terminales de α y f es la función de cómputo del atributo.
  Nota: Si un no terminal aparece varias veces en $A \longrightarrow \alpha$ se usan subíndices (empezando en 0 e incrementando de izquierda a derecha) para distinguir las distintas apariciones.

Ejemplo de semántica de atributos

Gramática con atributos sintetizados que computan el valor decimal de una frase que representa un número en binario.

$$Bits \longrightarrow Bits\ Bit$$
$$\quad Bits_0.valor = Bits_1.valor * 2 + Bit.valor$$
$$Bits \longrightarrow Bit$$
$$\quad Bits.valor = Bit.valor$$
$$Bit \longrightarrow 0$$
$$\quad Bit.valor = 0$$
$$Bit \longrightarrow 1$$
$$\quad Bit.valor = 1$$

Se puede entender como que el árbol sintáctico queda decorado con los atributos del no terminal de la izquierda en cada nodo.

Una gramática con solo atributos sintetizados se llama *s*-atribuida.

Ejemplo simple de construcción del AST

Supongamos que para el lenguaje de números binarios creamos la sintaxis abstracta con los constructores:

$$addBit : BinNum \times BitNum \longrightarrow BinNum$$
$$fromBit : BitNum \longrightarrow BinNum$$
$$bit : \{0 \mid 1\} \longrightarrow BitNum$$

Entonces podemos obtener un AST:

$$Bits \longrightarrow Bits\ Bit$$
$$\quad Bits_0.AST = addBit(Bits_1.AST, Bit.AST)$$
$$Bits \longrightarrow Bit$$
$$\quad Bits.AST = fromBit(Bit.AST)$$
$$Bit \longrightarrow 0$$
$$\quad Bit.AST = bit(0)$$
$$Bit \longrightarrow 1$$
$$\quad Bit.AST = bit(1)$$

Atributos heredados

Cuando el cálculo a realizar depende del contexto los atributos sintetizados pueden no ser suficientes o darnos una forma ineficiente de calcular los atributos.

El tratamiento del contexto se expresará usando *atributos heredados*.

Considerad la siguiente gramática para definir números con dígitos y calcular su valor en base 10.

$Numero \longrightarrow digito\ Numero$

$Numero_0.valor = pot10(digito.valor, Numero_1.digs) + Numero_1.valor$

$Numero_0.digs = Numero_1.digs + 1$

$Numero \longrightarrow digito$

$Numero.valor = digito.valor$

$Numero.digs = 1$

El cálculo requiere muchas multiplicaciones por 10.

Atributos heredados

Resultaría más eficiente si entendemos que el *valor* depende del contexto:

Añadimos un atributo heredado *hvalor* que nos dice el *valor* hasta el momento.

$$Numero \longrightarrow digito\ Numero$$
$$Numero_1.hvalor = Numero_0.hvalor * 10 + digito.valor$$
$$Numero_0.valor = Numero_1.valor$$
$$Numero \longrightarrow digito$$
$$Numero.valor = Numero.hvalor * 10 + digito.valor$$

Para completar la gramática es necesario añadir una producción inicial.

Atributos heredados

Resultaría más eficiente si entendemos que el *valor* depende del contexto:

Añadimos un atributo heredado *hvalor* que nos dice el *valor* hasta el momento.

$$Numero' \longrightarrow Numero$$
$$Numero.hvalor = 0$$
$$Numero'.valor = Numero.valor$$
$$Numero \longrightarrow digito\ Numero$$
$$Numero_1.hvalor = Numero_0.hvalor * 10 + valor(digito.lex)$$
$$Numero_0.valor = Numero_1.valor$$
$$Numero \longrightarrow digito$$
$$Numero.valor = Numero.hvalor * 10 + valor(digito.lex)$$

Gramáticas *l*-atribuidas

- Hay que garantizar que se pueden computar los valores de los atributos.
- En general, se pueden computar si no hay circularidad en las definiciones.
- Una condición suficiente es que la gramática sea *l*-atribuida: para cada producción $X \longrightarrow u_0 X_1 u_1 \ldots u_{n-1} X_n u_n$ Los atributos heredados de cada no terminal X_i dependen solo de:
  - Los atributos de $u_1 \ldots u_{i-1}$.
  - Los atributos sintetizados de $X_1 \ldots X_{i-1}$.
  - Los atributos heredados de X.
- Además en una gramática *l*-atribuida los atributos heredados se pueden calcular en el recorrido de izquierda a derecha y en profundidad.

Gramáticas de atributos LR(k)/LL(k)

Es fácil

- Hacer un analizador ascendente y calcular los atributos sintetizados para una gramática LR(k) *s*-atribuida.
- Hacer un analizador descendente y calcular los atributos heredados y sintetizados para una gramática LL(k) *l*-atribuida.

Los generadores automáticos de analizadores ascendentes y descendentes generan código que calcula los atributos.

De *s*-atribuidas a *l*-atribuidas

Para obtener una gramática $LL(1)$ es necesario

- factorizar
- eliminar la recursión por la izquierda

Si la gramática es *s*-atribuida podemos obtener una gramática *l*-atribuida equivalente sintáctica y semánticamente.

Factorización

Reemplazamos

$$A \longrightarrow \alpha\beta_1$$
$$A.a = f_1(\alpha.a, \beta_1.a)$$
$$\ldots$$
$$A \longrightarrow \alpha\beta_n$$
$$A.a = f_n(\alpha.a, \beta_n.a)$$

por

$$A \longrightarrow \alpha A'$$
$$A'.ha = \alpha.a$$
$$A.a = A'.a$$
$$A' \longrightarrow \beta_1$$
$$A'.a = f_1(A'.ha, \beta_1.a)$$
$$\ldots$$
$$A' \longrightarrow \beta_n$$
$$A'.a = f_n(A'.ha, \beta_n.a)$$

Eliminación de recursividad por la izquierda

Reemplazamos

$$A \longrightarrow A\alpha_1 \quad A_0.a = f_1(A_1.a, \alpha_1.a)$$
$$\dots$$
$$A \longrightarrow A\alpha_n \quad A_0.a = f_n(A_1.a, \alpha_n.a)$$
$$A \longrightarrow \beta_1 \quad A.a = g_1(\beta_1.a)$$
$$\dots$$
$$A \longrightarrow \beta_m \quad A.a = g_m(\beta_m.a)$$

por

$$A \longrightarrow \beta_1 A' \quad A'.ha = g_1(\beta_1.a) \quad A.a = A'.a$$
$$\dots$$
$$A \longrightarrow \beta_m A' \quad A'.ha = g_m(\beta_m.a) \quad A.a = A'.a$$
$$A' \longrightarrow \alpha_1 A' \quad A'_1.ha = f_1(A'_0.ha, \alpha_1.a) \quad A'_0.a = A'_1.a$$
$$\dots$$
$$A' \longrightarrow \alpha_n A' \quad A'_1.ha = f_n(A'_0.ha, \alpha_n.a) \quad A'_0.a = A'_1.a$$
$$A' \longrightarrow \epsilon \quad A'.a = A'.ha$$

Ejemplo

Considerad la siguiente gramática s-atribuida:

$$
\begin{array}{lcl}
A & \longrightarrow & A\ ,\ B \\
 & & A_0.a\ =\ lista(A_1.a, B.a) \\
A & \longrightarrow & @\ B\ A \\
 & & A_0.a\ =\ grupo(B.a, A_1.a) \\
A & \longrightarrow & @\ B\ B \\
 & & A.a\ =\ grupo(B_0.a, B_1.a) \\
B & \longrightarrow & str\ B \\
 & & B_0.a\ =\ linea(string(str.lex), B_1.a) \\
B & \longrightarrow & str\ \# \\
 & & B.a\ =\ string(str.lex)
\end{array}
$$

Transforma esta especificación para posibilitar su implementación mediante un analizador descendente predictivo.

Ejemplo

Factorizamos

$$
\begin{array}{lcl}
B & \longrightarrow & str\ B \\
 & & B_0.a = linea(string(str.lex), B_1.a) \\
B & \longrightarrow & str\ \# \\
 & & B.a = string(str.lex)
\end{array}
$$

y obtenemos

Ejemplo

Factorizamos

$$
\begin{array}{lcl}
B & \longrightarrow & str\ B \\
 & & B_0.a = linea(string(str.lex), B_1.a) \\
B & \longrightarrow & str\ \# \\
 & & B.a = string(str.lex)
\end{array}
$$

y obtenemos

$$
\begin{array}{lcl}
B & \longrightarrow & str\ B' \\
 & & B'.ha = string(str.lex) \\
 & & B.a = B'.a \\
B' & \longrightarrow & B \\
 & & B'.a = linea(B'ha, B.a) \\
B' & \longrightarrow & \# \\
 & & B'.a = B'.ha
\end{array}
$$

Ejemplo

Factorizamos

$$
\begin{array}{lcl}
A & \longrightarrow & @\ B\ A \\
 & & A_0.a = grupo(B.a, A_1.a) \\
A & \longrightarrow & @\ B\ B \\
 & & A.a = grupo(B_0.a, B_1.a)
\end{array}
$$

y obtenemos

Ejemplo

Factorizamos

$$
\begin{array}{lcl}
A & \longrightarrow & @\ B\ A \\
 & & A_0.a = grupo(B.a, A_1.a) \\
A & \longrightarrow & @\ B\ B \\
 & & A.a = grupo(B_0.a, B_1.a)
\end{array}
$$

y obtenemos

$$
\begin{array}{lcl}
A & \longrightarrow & @\ B\ A' \\
 & & A'.ha = B.a \\
 & & A.a = A'.a \\
A' & \longrightarrow & A \\
 & & A'.a = grupo(A'.ha, A.a) \\
A' & \longrightarrow & B \\
 & & A'.a = grupo(A'.ha, B.a)
\end{array}
$$

Ejemplo

Eliminamos recursividad por la izquierda

$$
\begin{array}{lcl}
A & \longrightarrow & A\ ,\ B \\
 & & A_0.a\ =\ \mathit{lista}(A_1.a, B.a) \\
A & \longrightarrow & @\ B\ A' \\
 & & A'.ha\ =\ B.a \\
 & & A.a\ =\ A'.a
\end{array}
$$

y obtenemos

Ejemplo

Eliminamos recursividad por la izquierda

$$
\begin{array}{lcl}
A & \longrightarrow & A\ ,\ B \\
 & & A_0.a\ =\ lista(A_1.a, B.a) \\
A & \longrightarrow & @\ B\ A' \\
 & & A'.ha\ =\ B.a \\
 & & A.a\ =\ A'.a
\end{array}
$$

y obtenemos

$$
\begin{array}{lcl}
A & \longrightarrow & @\ B\ A'\ A'' \\
 & & A'.ha\ =\ B.a \\
 & & A''.ha\ =\ A'.a \\
 & & A.a\ =\ A''.a
\end{array}
$$

$$
\begin{array}{lcl}
A'' & \longrightarrow & ,\ B\ A'' \\
 & & A''_1.ha\ =\ lista(A''_0.ha, B.a) \\
 & & A''_0.a\ =\ A''_1.a \\
A'' & \longrightarrow & \epsilon \\
 & & A''.a\ =\ A''.ha
\end{array}
$$

Ejercicio 1

Transformar la siguiente gramática de atributos para posibilitar su implementación durante el análisis descendente.

$$
\begin{array}{lll}
Path & \longrightarrow & step\ Path \\
 & & Path_0.form = join(step.val, Path_1.form) \\
Path & \longrightarrow & step \\
 & & Path.form = newPath(step.val) \\
Path & \longrightarrow & step\ Curve \\
 & & Path.form = blend(step.val, Curve.form) \\
Curve & \longrightarrow & Curve\ segment \\
 & & Curve_0.form = attach(Curve_1.form, segment.val) \\
Curve & \longrightarrow & segment \\
 & & Curve.form = newCurve(segment.val)
\end{array}
$$

Ejercicios 2

Transformar la siguiente gramática de atributos para posibilitar su implementación durante el análisis descendente.

$$
\begin{array}{lcl}
E & \longrightarrow & E \ @ \ T \\
 & & E_0.s = arroba(E_1.s, T.s) \\
E & \longrightarrow & T * T \\
 & & E.s = por(T_0.s, T_1.s) \\
E & \longrightarrow & T \\
 & & E.s = T.s \\
T & \longrightarrow & id \\
 & & T.s = iden(id.lex) \\
T & \longrightarrow & (\ E \ ) \\
 & & T.s = E.s
\end{array}
$$

Ejercicio 3

Transformar la siguiente gramática de atributos para posibilitar su implementación durante el análisis descendente.

$$
\begin{array}{lcl}
A & \longrightarrow & A\ a \\
 & & A_0.n = combina(A_1.n, a.lex) \\
A & \longrightarrow & A\ b \\
 & & A_0.n = junta(A_1.n, b.lex) \\
A & \longrightarrow & B\ c \\
 & & A.n = mezcla(B.n, c.lex) \\
A & \longrightarrow & B\ d \\
 & & A.n = teje(B.n, d.lex) \\
B & \longrightarrow & e \\
 & & B.n = inicia(e.lex)
\end{array}
$$