Traducción dirigida por sintaxis

Pedro O. Pérez M., PhD

Diseño de compiladores
Tecnológico de Monterrey

pperezm@tec.mx

05-2023

1. 5.1 - 5.3
   - 5.1 Definiciones dirigidas por sintaxis
   - 5.2 Orden de evaluación de SDD
   - 5.3 Aplicaciones de traducciones dirigidas por sintaxis

2. 5.4 - 5.6
   - 5.4 Esquemas de traducción dirigidos por sintaxis

Una definición dirigida por sintaxis (*syntax-directed definition*, SDD) es una gramática libre de contexto a la cuál se le agregan atributos y reglas. Los atributos están asociados a los símbolos gramaticales y las reglas están asociadas a las producción. Si X es un símbolo y a es uno de sus atributos, entonces $X.a$ denota el valor de a de un nodo del árbol de análisis en particular.

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

Existen dos tipos de atributos para los no terminales:

- Un `atributo sintetizado` para un no terminal A en un nodo N del árbol de análisis está definido por una regla semántica asociada con la producción N. Nota que la producción debe tener A como encabezado. Un `atributo sintetizado` del nodo N está definido SÓLO en términos de los valores de los atributos de los hijos de N y de él mismo.

	PRODUCTION	SEMANTIC RULES
1)	$L \rightarrow E$ **n**	$L.val = E.val$
2)	$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3)	$E \rightarrow T$	$E.val = T.val$
4)	$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5)	$T \rightarrow F$	$T.val = F.val$
6)	$F \rightarrow (E)$	$F.val = E.val$
7)	$F \rightarrow$ **digit**	$F.val =$ **digit**.lexval

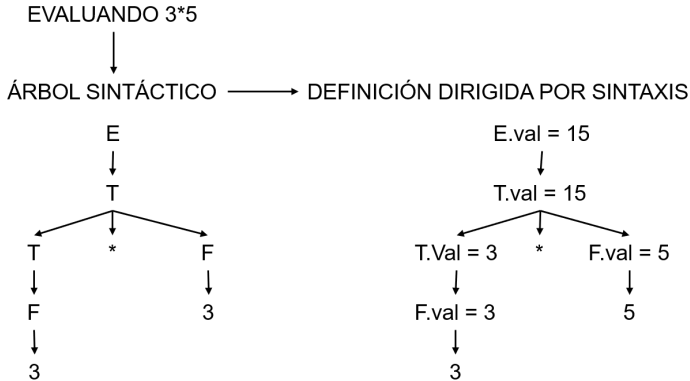
Figure 5.1: Syntax-directed definition of a simple desk calculator

- Un SSD que involucre sólo atributos sintetizados es llamado "`S-attributed`". En un `S-attributed` SSD, cada reglas calcula un atributo para el no terminal del encabezado de una producción a partir de los atributos obtenidos del cuerpo de la producción.
- Un `S-attributed SSD` puede ser implementado, de manera natural, con un analizador LR.
- La gramática anterior es considerada como una gramática libre de efectos laterales debido a su cálculo directo. Un SSD sin efectos laterales es llamada `gramática atributo`.

- Un SDD que involucra sólo atributos sintetizados es llamado *S-attributed*. Un *S-attributed* SDD puede ser implementado de forma fácil con un analizador LR.
- En la práctica es conveniente tiene reglas semánticas que tengan efectos laterales (imprimir valores). Un SDD sin efecto lateral es llamada una gramática atributo.

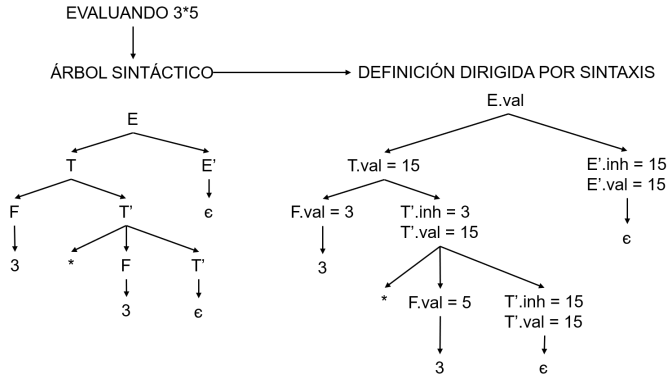
- Un `atributo heredado` para un no terminal B está definido por una regla semántica asociada con la producción del nodo padre de N. Nota que la producción debe tener a B como un símbolo del cuerpo. Un `atributo heredado` en el nodo N está definido solo en terminados de los valores de los atributos del padre y hermanos de N, así como de él mismo.

	Production	Semantic Rules
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



- Mientras que no está permitido que un atributo heredado en el nodo N se encuentre definido en términos de los valores de los atributos de los hijos de N, si está permitido que un atributo sintetizado en el nodo N se defina en términos de atributos heredados del nodo N.
- Los terminales pude tener atributos sintetizados, pero nunca heredados.

	PRODUCTION	SEMANTIC RULES
1)	$T \to F\ T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \to *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \to \epsilon$	$T'.syn = T'.inh$
4)	$F \to \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$



Evaluando un SDD con los nodos de un árbol de análisis

Empleado un SDD no es necesario construir un árbol de análisis para realizar la traducción. Sin embargo, un árbol de análisis mostrando los valores de los atributos es llamado `annotated parse tree` (árbol de análisis anotado).

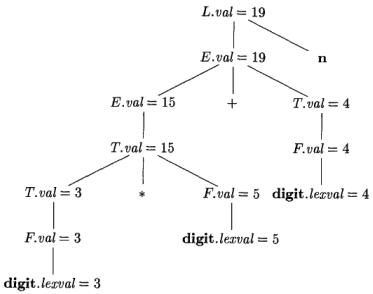


Figure 5.3: Annotated parse tree for $3 * 5 + 4$ **n**

- Un SDD con atributos sintetizados, lo podemos evaluar los atributos en cualquier recorrido “bottom-up”, por ejemplo, un recorrido post-order.
- Un SDD con atributos sintetizados y heredados, no tenemos garantizado en que orden evaluar los nodos. Por ejemplo...

PRODUCTION	SEMANTIC RULES
$A \to B$	$A.s = B.i;$
	$B.i = A.s + 1$

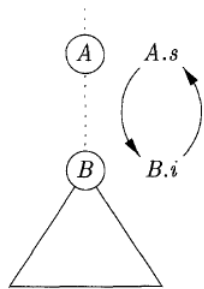


Figure 5.2: The circular dependency of $A.s$ and $B.i$ on one another

Los atributos heredados son útiles cuando la estructura del árbol de análisis no es idéntico al árbol de sintaxis abstracta del código fuente.

	PRODUCTION	SEMANTIC RULES
1)	$T \rightarrow F\, T'$	$T'.inh = F.val$ $T.val = T'.syn$
2)	$T' \rightarrow *\, F\, T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3)	$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4)	$F \rightarrow \mathbf{digit}$	$F.val = \mathbf{digit}.lexval$

Figure 5.4: An SDD based on a grammar suitable for top-down parsing

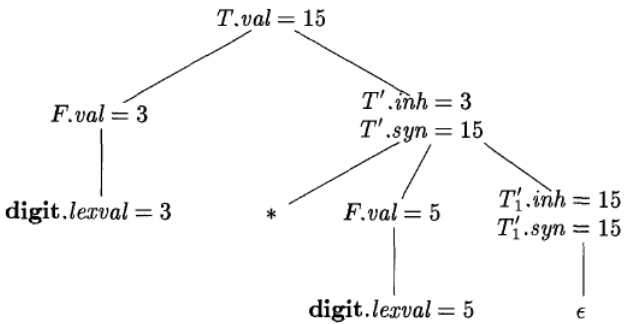


Figure 5.5: Annotated parse tree for $3 * 5$

Orden de evaluación de SDD

Los grafos de dependencia son una herramienta útil para determinar el orden de evaluación de los atributos de un árbol de análisis determinado. Mientras que árbol de análisis anotado muestra los valores de los atributos, una grafo de dependencia nos ayuda a determinar cómo los valores son calculados.

Un grafo de dependencia indica el flujo de información entre los atributos de un árbol de análisis particular. El arco de un atributo a otro indica que el valor del primero es requerido para calcular el segundo. Los arcos indican restricciones implicadas en las reglas semánticas.

- Por cada nodo, X, del árbol de análisis, el grafo de dependencia tiene un vértice por cada nodo asociado con X.

- Supón que una regla semántica asociada con una producción p define el valor del atributo sintetizado $A.b$ en términos del valor de $X.c$ (la regla puede definir $A.b$ en términos de otros atributos además de $X.c$). Entonces, el gro de dependencia tiene un arco de $X.c$ a $A.b$. Más precisamente, en cada nota N etiquedata A donde la producción P es aplicada, se crea un arco del atributo b en N, al atributo c en el hijo de N correspondiente con la instancia del símbolo X en el cuerpo de la producción.

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 \ + \ T$	$E.val = E_1.val + T.val$

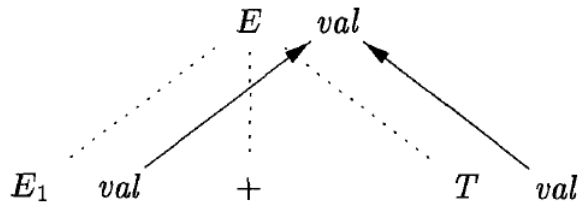


Figure 5.6: $E.val$ is synthesized from $E_1.val$ and $E_2.val$

- Supón que una regla semántica asociada con una producción p define el valor del atributo heredado $B.c$ en términos del valor de $X.a$. Entonces, el grafo de dependencia tiene una arco de $X.a$ a $B.c$. Para cada nodo N etiquetado como B que corresponda a una ocurrencia de B en el cuerpo de producción p, se crea un arco del atributo c en N al atributo a en el nodo M que corresponde a la ocurrencia de X. Observa que M puede ser el padre o el hermano de N.

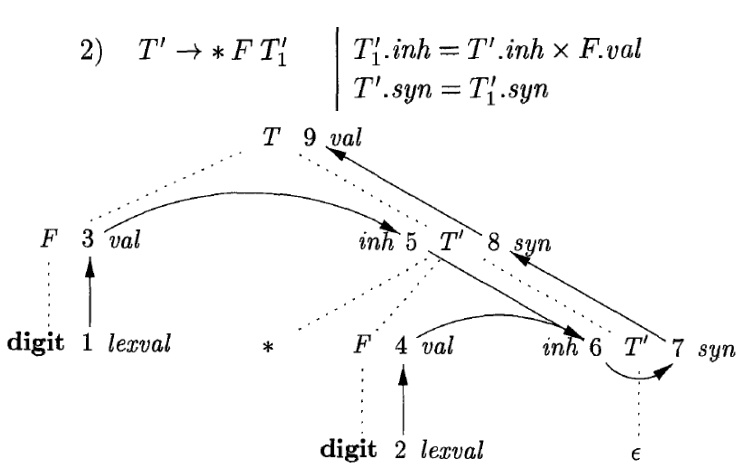


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Orden de evaluación de los atributos

El grado de dependencia caracteriza las posibles formas en las cuales podemos evaluar los atributos de un árbol de análisis. Estas formas de evaluación tienden a ser lineales (*Topological Sort*).

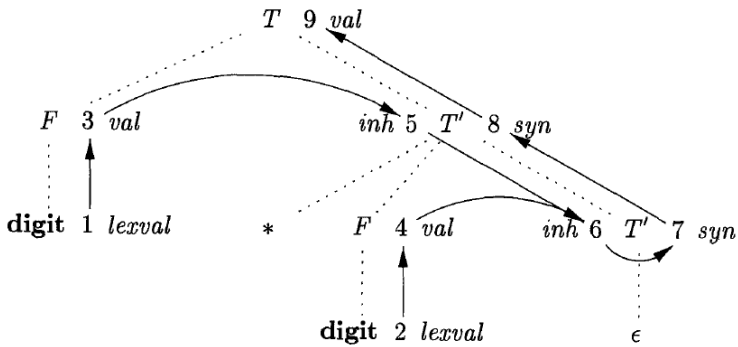


Figure 5.7: Dependency graph for the annotated parse tree of Fig. 5.5

Topological sort: 1, 3, 5, 2, 4, 6, 7, 8, 9

La traducción debe ser implementada usando alguna clase de SDD que garantice el orden de evaluación. Existen dos clases que puede ser implementadas eficientemente en conexión de un análisis sintáctico “top-down” o “bottom-up”.

Cómo se mencionó antes, un SDD donde todos los atributos son sintetizadas es llamado `S-attributed`. Cuando un SDD es `S-attributed`, podemos evaluar sus tributos en cualquier recorrido "bottom-up".

```
postorder(N) {
        for ( each child C of N, from the left ) postorder(C);
        evaluate the attributes associated with node N;
}
```

Existe una segunda clase de SDD's llamada `L-attributed`. La idea detrás de esta clase es que, entre los atributos asociados al cuerpo de una producción, los arcos de dependencia sólo pueden ir de izquierda a derecha (de ahí su nombre).

Más precisamente, cada atributo debe ser:

- Sintetizado,
- Heredado, pero empleado la siguientes reglas. Supón que hay una producción $A \rightarrow X_1 X_2 ... X_n$, y que hay un atributo heredado $X_i.a$ calculado con una regla asociada con esta producción. Entonces la regla puede ser usada solo:
  - El atributo heredado está asociado al encabezado de A.
  - Los atributos heredados o sintetizados asociados con los simbolos X_1, X_2, ... , X_{i-1} están localizados a la izquierda de X_i.
  - Los atributos sintetizados o heredados asociados con X_i, pero solo si no hay ciclos en el grafo de dependencia formando por los atributos de X_i.

PRODUCTION	SEMANTIC RULE
$T \rightarrow F\ T'$	$T'.inh = F.val$
$T' \rightarrow *\ F\ T'_1$	$T'_1.inh = T'.inh \times F.val$

PRODUCTION	SEMANTIC RULES
$A \rightarrow B\ C$	$A.s = B.b;$
	$B.i = f(C.c, A.s)$

Reglas semánticas con efectos laterales controlados

En la práctica, las traducciones suelen involucrar efectos laterales. Podemos controlar los efectos laterales en un SDD con alguna de las siguientes estrategias:

- Permitir efecto laterales que no restrinjan la evaluación de los atributos, basado en el "topological sort" correcto.
- Restringir los órdenes de evaluación posibles, de forma que la traducción sea la misma sin importar el orden de evaluación.

	PRODUCTION	SEMANTIC RULE
1)	$D \rightarrow T\ L$	$L.inh = T.type$
2)	$T \rightarrow$ **int**	$T.type =$ integer
3)	$T \rightarrow$ **float**	$T.type =$ float
4)	$L \rightarrow L_1$, **id**	$L_1.inh = L.inh$
		$addType(\mathbf{id}.entry, L.$
5)	$L \rightarrow$ **id**	$addType(\mathbf{id}.entry, L.$

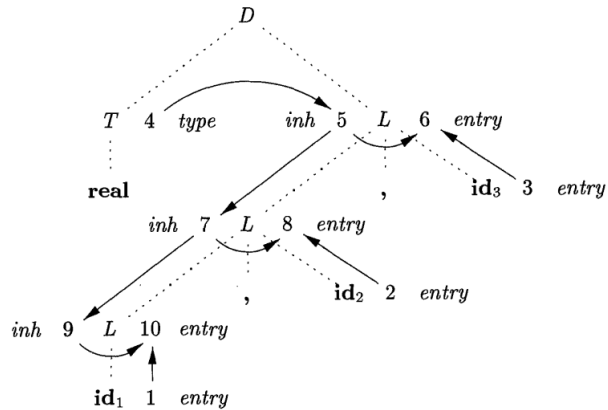


Figure 5.9: Dependency graph for a declaration **float** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Exercise 5.2.3: Suppose that we have a production $A \to BCD$. Each of the four nonterminals A, B, C, and D have two attributes: s is a synthesized attribute, and i is an inherited attribute. For each of the sets of rules below, tell whether (i) the rules are consistent with an S-attributed definition (ii) the rules are consistent with an L-attributed definition, and (iii) whether the rules are consistent with any evaluation order at all?

a) $A.s = B.i + C.s$.

b) $A.s = B.i + C.s$ and $D.i = A.i + B.s$.

c) $A.s = B.s + D.s$.

Como ya se ha mencionado antes, un árbol de sintaxis representa una producción. Un árbol de sintaxis se implementa con nodos que almacenan información relevante. Cada nodo contiene, al menos, un campo *op* que es la etiqueta del nodo y puede contener alguno de los siguientes campos:

- Si el nodo es una hoja, éste tiene un campo conteniendo el valor lexicográfico de la hoja.
- Si es un nodo interno, éste puede tener campos adicionales como pueden ser los hijos del nodo.

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node =$ **new** $Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node =$ **new** $Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow$ **id**	$T.node =$ **new** $Leaf($**id**, **id**.$entry)$
6)	$T \rightarrow$ **num**	$T.node =$ **new** $Leaf($**num**, **num**.$val)$

Figure 5.10: Constructing syntax trees for simple expressions

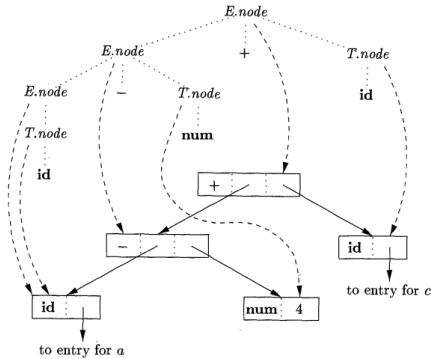


Figure 5.11: Syntax tree for $a - 4 + c$

1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a});$
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c});$
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);$

Figure 5.12: Steps in the construction of the syntax tree for $a - 4 + c$

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow T\ E'$	$E.node = E'.syn$ $E'.inh = T.node$
2)	$E' \rightarrow +\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('+', E'.inh, T.node)$ $E'.syn = E'_1.syn$
3)	$E' \rightarrow -\ T\ E'_1$	$E'_1.inh = \textbf{new}\ Node('-', E'.inh, T.node)$ $E'.syn = E'_1.syn$
4)	$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
5)	$T \rightarrow (\ E\ )$	$T.node = E.node$
6)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
7)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Figure 5.13: Constructing syntax trees during top-down parsing

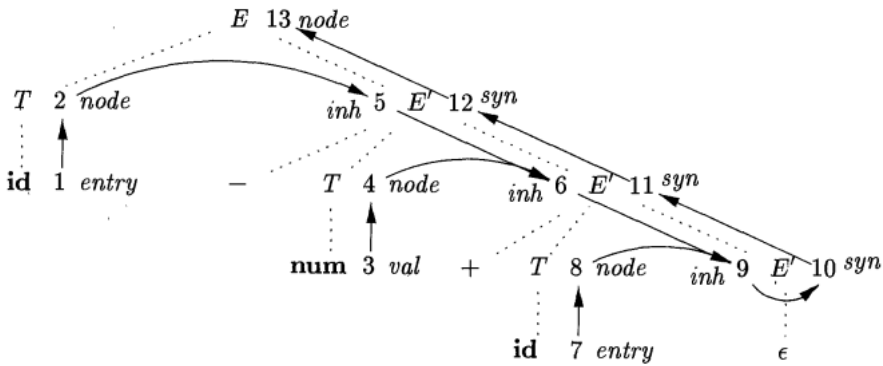


Figure 5.14: Dependency graph for $a - 4 + c$, with the SDD of Fig. 5.13

Los atributos heredados son empleados cuando la estructura de un árbol de análisis difiere de la sintaxis abstracta; los atributos se usan para llevar información de una parte del árbol a otra.

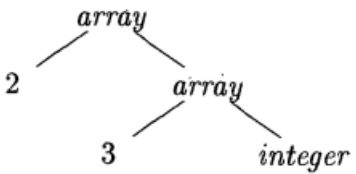


Figure 5.15: Type expression for **int**[2][3]

PRODUCTION	SEMANTIC RULES
$T \rightarrow B\ C$	$T.t = C.t$
	$C.b = B.t$
$B \rightarrow$ **int**	$B.t = integer$
$B \rightarrow$ **float**	$B.t = float$
$C \rightarrow$ [**num**] C_1	$C.t = array(\mathbf{num}.val,\ C_1.t)$
	$C_1.b = C.b$
$C \rightarrow \epsilon$	$C.t = C.b$

Figure 5.16: T generates either a basic type or an array type

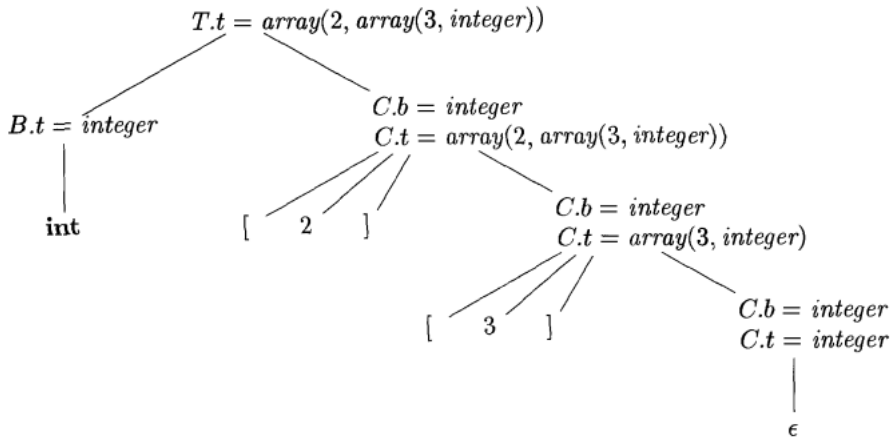


Figure 5.17: Syntax-directed translation of array types

Exercise 5.3.1 : Below is a grammar for expressions involving operator + and integer or floating-point operands. Floating-point numbers are distinguished by having a decimal point.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow \textbf{num} \ . \ \textbf{num} \mid \textbf{num}$$

a) Give an SDD to determine the type of each term T and expression E.

b) Extend your SDD of (a) to translate expressions into postfix notation. Use the unary operator **intToFloat** to turn an integer into an equivalent float.

Esquemas de traducción dirigidos por sintaxis

- Los esquemas de traducción dirigidos por sintaxis tienen una notación complementaria a las definiciones sintácticas. Todo aquellas aplicaciones que se implementen usando definiciones dirigidos por sintaxis pueden ser también implementados usando esquemas de traducción por sintaxis.
- Un esquema de traducción dirigidas por sintaxis (SDT) es un gramática libre de contexto con fragmentos de programas embebidos dentro del cuerpo de la producción. Estos fragmentos de código son llamadas acciones semánticas y pueden aparecer en cualquier posición dentro del cuerpo de la producción.

- Cualquier SDT puede ser implementado construyendo primero un árbol de análisis y luego realizado las acciones de izquierda a derecha.
- Típicamente, los SDT's son implementados durante el análisis, sin construir el árbol de análisis.
  - Gramática LR, el SDD es `S-attributed`.
  - Gramática LL, el SDD es `L-attributed`.

De lejos, la implementación SDD más simple ocurre cuando analizados una gramática bottom-up y el SDD es *S-attributed*. En ese caso, nosotros podemos construir un SDT en el cual la acción está al final de la producción y es ejecutta cuando se hace la reducción del cuerpo de la producción. Este tipo de SDT es conocido postfijo.

$$
\begin{array}{lcll}
L & \rightarrow & E\ \mathbf{n} & \{\ \mathrm{print}(E.val);\ \} \\
E & \rightarrow & E_1 + T & \{\ E.val = E_1.val + T.val;\ \} \\
E & \rightarrow & T & \{\ E.val = T.val;\ \} \\
T & \rightarrow & T_1 * F & \{\ T.val = T_1.val \times F.val;\ \} \\
T & \rightarrow & F & \{\ T.val = F.val;\ \} \\
F & \rightarrow & (\ E\ ) & \{\ F.val = E.val;\ \} \\
F & \rightarrow & \mathbf{digit} & \{\ F.val = \mathbf{digit}.lexval;\ \}
\end{array}
$$

Figure 5.18: Postfix SDT implementing the desk calculator

Implementación por pila del SDT postfijo

- Un SDT postfijo puede ser implementado en un analizador LR cuando se realiza una reducción. Los atributos de cada símbolo gramatical pueden ser colocan en la pila e un lugar donde pueden ser encontrados durante la reducción.

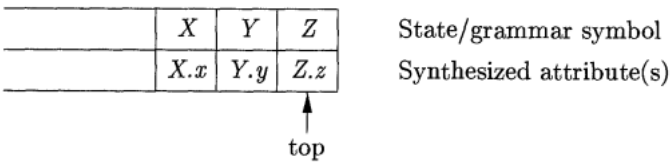


Figure 5.19: Parser stack with a field for synthesized attributes

PRODUCTION	ACTIONS
$L \to E\ \mathbf{n}$	$\{\ \text{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \to E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \to T$	
$T \to T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \to F$	
$F \to (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \to \mathbf{digit}$	

Figure 5.20: Implementing the desk calculator on a bottom-up parsing stack

SDT's con acciones dentro las producciones

Una acción puede ser colocada en cualquier posición dentro del cuerpo de una producción. La acción es realizada inmediatamente después de procesar todos los símbolos que están a su izquierda. Entonces, si tenemos una producción $B \to X \ \{a\} \ Y$, la acción a es realizada después de reconocer X (si X es un terminal) o todas las terminales derivadas de C (si X es un no terminal). Más precisamente:

- Si el analizador es bottom-up, entonces realizamos la acción a tan pronto como esta ocurrencia de X aparece en el tope de la pila de análisis.
- Si el analizado es top-down, realizamos la acción a antes de intentar expandir la ocurrencia de Y (si Y es un no terminal) o revisar por Y en la entrada (si Y es un terminal).

$$
\begin{array}{lllll}
1) & L & \rightarrow & E\ \mathbf{n} & \\
2) & E & \rightarrow & \{\ \text{print}('+');\ \}\ \ E_1 + T & \\
3) & E & \rightarrow & T & \\
4) & T & \rightarrow & \{\ \text{print}('*');\ \}\ \ T_1 * F & \\
5) & T & \rightarrow & F & \\
6) & F & \rightarrow & (\ E\ ) & \\
7) & F & \rightarrow & \mathbf{digit}\ \ \{\ \text{print}(\mathbf{digit}.\mathit{lexval});\ \} &
\end{array}
$$

Figure 5.21: Problematic SDT for infix-to-prefix translation during parsing

Es imposible implementar este SDT, ya sea "bottom-up" or "top-down", porque el analizador puede tener que realizar alguna acción, como imprimir * o +, mucho antes de conocer si estos símbolos aparecen en la entrada.

Si tenemos que construir un SDT podemos hacer lo siguiente:

- Ignora las acciones, analiza la entrada y construye un árbol de análisis.
- Examina cada nodo interior N que representan alguna producción $A \to \alpha$. Agrega hijos adicionales a N para las acciones en α, de tal forma que los hijos de N, de izquierda a derecha, tiene exactamente los símbolos y acciones de α.
- Realiza un recorrido preorder del árbol, y tan pronto se visite un nodo etiquetado con una acción, ejecutarla.

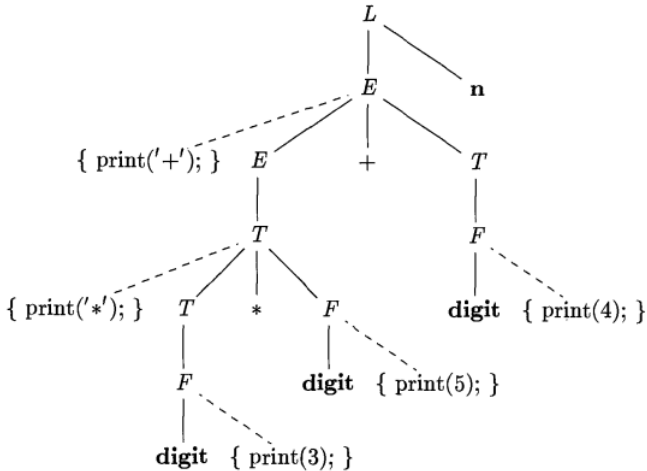


Figure 5.22: Parse tree with actions embedded

Eliminando la recursividad por la izquierda de un SDT

Cuando transformes una gramática para quitar la recursión por la izquierda, trata las acciones gramaticales como si fueran símbolos terminales.

Example 5.17: Consider the following E-productions from an SDT for translating infix expressions into postfix notation:

$$\begin{array}{lcll} E & \rightarrow & E_1 + T & \{ \text{ print}('+'); \} \\ E & \rightarrow & T & \end{array}$$

If we apply the standard transformation to E, the remainder of the left-recursive production is

$$\alpha \;=\; +\; T\; \{ \text{ print}('+'); \}$$

and β, the body of the other production is T. If we introduce R for the remainder of E, we get the set of productions:

$$\begin{array}{lcl} E & \rightarrow & T\ R \\ R & \rightarrow & +\ T\ \{ \text{ print}('+'); \}\ R \\ R & \rightarrow & \epsilon \end{array}$$

Algunas veces, remover la recursión por la izquierda puede no se tan sencillo...

$$
\begin{array}{lcl}
A & \rightarrow & A_1\ Y\ \{A.a = g(A_1.a, Y.y)\} \\
A & \rightarrow & X\ \{A.a = f(X.x)\}
\end{array}
$$

$$\Downarrow$$

$$
\begin{array}{lcl}
A & \rightarrow & X\ R \\
R & \rightarrow & Y\ R \mid \epsilon
\end{array}
$$

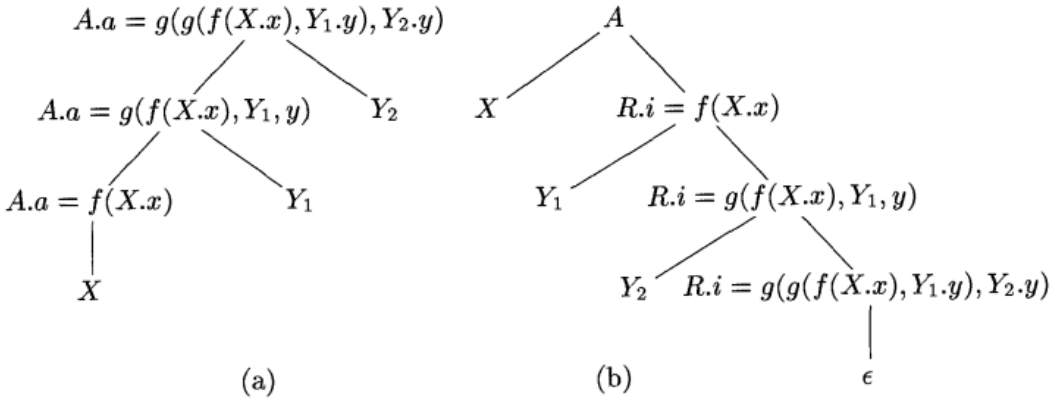


Figure 5.23: Eliminating left recursion from a postfix SDT

$$
\begin{array}{lcl}
A & \rightarrow & X \ \ \{R.i = f(X.x)\} \ \ R \ \ \{A.a = R.s\} \\
R & \rightarrow & Y \ \ \{R_1.i = g(R.i, Y.y\} \ \ R_1 \ \ \{R.s = R_1.s\} \\
R & \rightarrow & \epsilon \ \ \{R.s = R.i\}
\end{array}
$$

SDT's para definiciones L-attributed

Ya revisamos cómo convertir un S-attributed SDD en un SDT postfijo. Esta se puede hacer, en tanto se hable de una gramática LR, a través de un análisis "bottom-up".
Ahora consideremos el caso más general: SDD L-attributed. Para ello, debemos considerar que debe ser una gramática que pueda ser analizada "top-down" .

Las reglas para convertir un SDD `L-attributed` en una SDT son:

- Embebe la acción que calcula el atributo heredado de un o terminal A inmediatamente antes de la ocurrencia de A en el cuerpo de la producción. Sí varios atributos de A dependen de cualquier otro de una manera acíclica, ordena la evaluación de los atributos de la manera en que se evalúen adecuadamente.
- Coloca las acciones que calcula el atributo sintetizado del encabezado de una producción al final del cuerpo de la misma.

Veamos el siguiente ejemplo. Queremos generar el código intermedio de la siguiente producción:

$$S \rightarrow \textbf{while}\,(\,C\,)\,S_1$$

Esta producción debe generar los siguientes atributos:

- El atributo generado $S.next$ que indica el inicio del código que debe ser ejecutado después de S.
- El atributo sintetizado $S.code$ es la secuencia de código intermedio que implementa S y que termina con un salto a $S.next$.

- El atributo heredado $C.true$ que indica el inicio del código que debe ser ejecutado si C es verdadero.
- El atributo heredado $C.false$ que indica el inicio del códgio que debe ser ejecutado si C es falso.
- El atributo sintetizado $C.code$ es la secuencia de código intermedio que implementa la condición C y salta, ya sea a $C.true$ o a $C.false$.

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$	$L1 = new();$ $L2 = new();$ $S_1.next = L1;$ $C.false = S.next;$ $C.true = L2;$ $S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

Figure 5.27: SDD for while-statements

$$
\begin{array}{lll}
S \rightarrow \textbf{while} \; (& \{ \; L1 = new(); \; L2 = new(); \; C.false = S.next; \; C.true = L2; \; \} \\
\quad C \;) & \{ \; S_1.next = L1; \; \} \\
\quad S_1 & \{ \; S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code; \; \}
\end{array}
$$

Figure 5.28: SDT for while-statements

Exercise 5.4.2: Rewrite the following SDT:

$$A \to A\ \{a\}\ B \mid A\ B\ \{b\} \mid 0$$
$$B \to B\ \{c\}\ A \mid B\ A\ \{d\} \mid 1$$

so that the underlying grammar becomes non-left-recursive. Here, a, b, c, and d are actions, and 0 and 1 are terminals.

Dado que muchas traducciones pueden ser manejadas con definiciones L-attributed, revisaremos su implementación con más detalle.
Ya hemos revisados dos métodos para realizar la traducción recorriendo el árbol de análisis.

- Construye el árbol y realizar anotaciones. (Sección 5.1.2).
- Construye el árbol, agrega acciones, y ejecuta las acciones en preorder. (Sección 5.4.4).

A continuación, discutiremos los métodos para realizar la traducción durante el análisis:

- Usar un analizador de descenso recursivo. En este caso, la función relacionada con un no terminal A recibe los atributos heredados de A como argumentos y regresa los atributos sintetizados de A.
- Generar el código "on the fly", usando un analizador de descenso recursivo.
- Implementar un SDT en conjunción con un analizador LL.
- Implementar un SDT en conjunción con un analizador LR.

Traducción durante el análisis de descenso recursivo

Un analizador de descenso recursivo tiene una función A para cada no terminal A (Sección 4.4.1). Es posible convertir este analizador en un traductor...

- Los argumento de la función A son los atributos heredados del no terminal A.
- El valor que regresa la función A es la colección de atributos sintetizados del no terminal A.

En el cuerdo de la función A, necesitamos tanto analizar y manejar los atributos para...

- Decidir sobre la producción usada para expandir A.
- Verificar que cada terminar aparezca en la entrada en el orden requerido. Si es necesario hacer "backtracking", se puede hacer restaurando la entrada a la posición que tenía antes del error.
- Preservar, en variables locales, el valor de todos los atributos necesarios para calcular los atributos heredados para los no terminales en el cuerpo o atributos sintetizados en el encabezado.
- Llamar a las funciones correspondientes a los no terminales en el cuerpo de la producción seleccionada con los parámetros adecuados. Dado que estamos trabajando con un SDD con `L.attributed`, ya debimos haber calculado estos atributos y se encuentran almacenados en variables locales.

```
string S(label next) {
        string Scode, Ccode; /* local variables holding code fragments */
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                Ccode = C(next, L2);
                check ')' is next on the input, and advance;
                Scode = S(L1);
                return("label" || L1 || Ccode || "label" || L2 || Scode);
        }
        else /* other statement types */
}
```

Figure 5.29: Implementing while-statements with a recursive-descent parser

Generación de código “al vuelo”

Muchas veces, la construcción de largas cadenas de código como resultado de atributos no es deseable por varias razones: copiar o mover cadenas largas. En la mayoría de los casos, se colocando pedazos de código en un arreglo o un archivo de salida.

Los elementos que debemos considerar para que esta técnica funcione son:

- Hay, para cada no terminal, un atributo principal. Por conveniencia, este atributo principal es una cadena de caracteres.
- Los atributos principales son sintetizados.
- Las reglas para evaluar un atributo principal deben tener en cuenta:
  - El atributo principal es la concatenación de los atributos principales que aparece en el cuerpo de la producción involucrada, incluyendo, otros elementos que no son atributos principales.
  - Los atributos principales de los no terminales aparecen en la regla en el mismo orden en que aparecen en el cuerpo de la producción.

```
void S(label next) {
        label L1, L2; /* the local labels */
        if ( current input == token while ) {
                advance input;
                check '(' is next on the input, and advance;
                L1 = new();
                L2 = new();
                print("label", L1);
                C(next, L2);
                check ')' is next on the input, and advance;
                print("label", L2);
                S(L1);
        }
        else /* other statement types */
}
```

Figure 5.31: On-the-fly recursive-descent code generation for while-statements

Incidentalmente, podemos hacer los mismo cambios en el SDT: convertir la construcción de un atributo principal en acciones que generen los elementos de ese atributo.

$$
\begin{array}{lll}
S \rightarrow \textbf{while} (& \{\ L1 = new();\ L2 = new();\ C.false = S.next; & \\
 & \quad C.true = L2;\ print(\texttt{"label"}, L1);\ \} & \\
C\ ) & \{\ S_1.next = L1;\ print(\texttt{"label"}, L2);\ \} & \\
S_1 & &
\end{array}
$$

Figure 5.32: SDT for on-the-fly code generation for while statements

SDD L-attributed y un analizador LL

En la sección 5.4.5 se revisó como era posible convertir un SDD L-attributed basado en un analizador LL en un SDT con acciones embebidas en la producciones: extendiendo la pila de análisis que albergue las acciones y ciertos datos para la evaluación de atributos.

Adicionalmente, para registrar la representación de terminales y no terminales, la pila del analizador debe mantener las acciones registro que representan las acciones a ser ejecutadas y registros sintetizados que almacenan los atributos sintetizados para no terminal.

Para hacerlo, debemos de tomar en cuenta la siguiente:

- Los atributos heredados de un no terminal A se colocan en el registro de pila que representa ese no terminal. El código para evaluar estos atributos usualmente se representan por un registro acción inmediatamente por encima del registro de pila para A.
- Los atributos sintetizados para un no terminal A se colocan en un registro sintetizado separado por debajo de A en la pila.

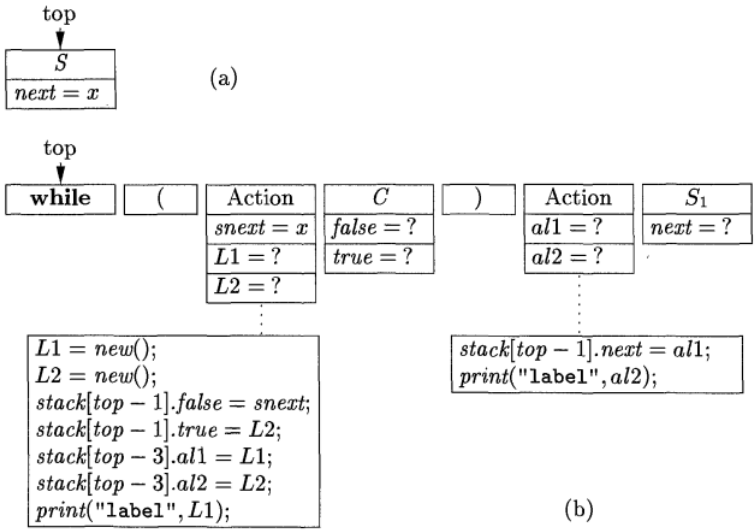


Figure 5.33: Expansion of S according to the while-statement production

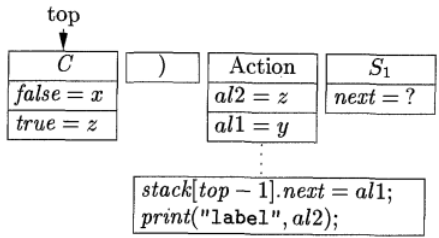


Figure 5.34: After the action above C is performed

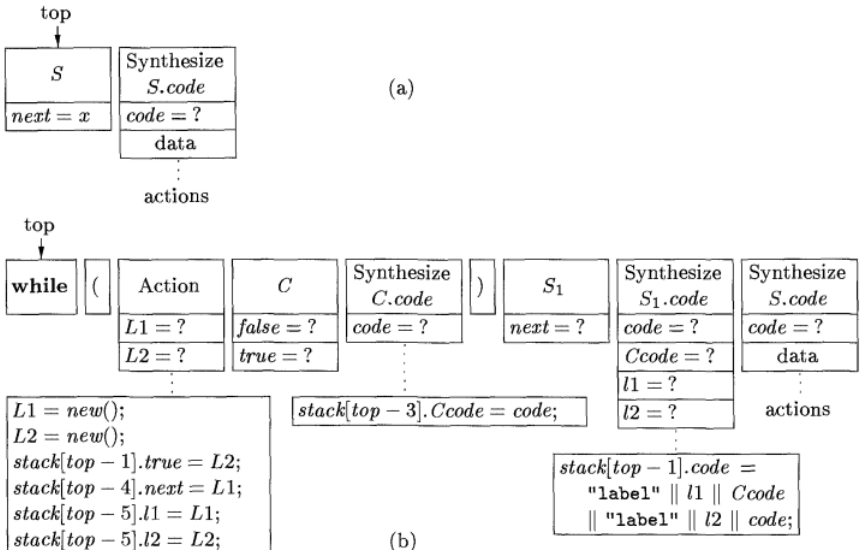


Figure 5.35: Expansion of S with synthesized attribute constructed on the stack

Analizador "bottom-up" de un SDD L-attributed

Podemos hacer un traducción "bottom-up" con un analizador LR, para ello debemos tener en cuenta tres puntos:

- Construir un SDT (Sección 5.4.5), el cual coloca acciones embebidas antes de cada no terminal para calcular sus atributos heredados y una acción al final de la producción para calcular atributos sintetizados.
- Introducir en la gramática un marcador no terminal en lugar de cada acción embebida. Cada lugar tiene un marcador distintivo, y hay una producción para cada marcador M ($M \to \epsilon$).

- Modificar la acción *a* si el marcador no terminal M lo reemplaza en alguna producción $A \rightarrow \alpha a \beta$, y asocia $M \rightarrow \epsilon$ con una acción a' que...
  - Copiar, como atributos heredados de *m*, cualquier atributo de A o símbolos de α que la acción *a* necesite.
  - Calcula los atributos de la misma manera que A ,pero hacer que estos atributos sean atributos sintetizados de M.

Example 5.25 : Suppose that there is a production $A \to B\,C$ in an LL grammar, and the inherited attribute $B.i$ is computed from inherited attribute $A.i$ by some formula $B.i = f(A.i)$. That is, the fragment of an SDT we care about is

$$A \to \{B.i = f(A.i);\}\ B\ C$$

We introduce marker M with inherited attribute $M.i$ and synthesized attribute $M.s$. The former will be a copy of $A.i$ and the latter will be $B.i$. The SDT will be written

$$\begin{aligned} A &\to M\ B\ C \\ M &\to \{M.i = A.i;\ M.s = f(M.i);\} \end{aligned}$$

Notice that the rule for M does not have $A.i$ available to it, but in fact we shall arrange that every inherited attribute for a nonterminal such as A appears on the stack immediately below where the reduction to A will later take place. Thus, when we reduce ϵ to M, we shall find $A.i$ immediately below it, from where it may be read. Also, the value of $M.s$, which is left on the stack along with M, is really $B.i$ and properly is found right below where the reduction to B will later occur. $\square$

Ejercicio 5.4

Exercise 5.5.1: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.1.

Exercise 5.5.2: Implement each of your SDD's of Exercise 5.4.4 as a recursive-descent parser in the style of Section 5.5.2.