Conceptos Avanzados en Lenguajes de Programación

Semanticas

<2016-08-22 lun>

Contents

Introducción

Introducción

- **Sintaxis**: La forma y estructura de las expresiones, sentencias y unidades del programa.
- **Semántica**: El significado de las expresiones, sentencias, y unidades del programa.
- Sintaxis y Semántica proveen una Definición del Lenguaje
  - Usuarios de una Definición del Lenguaje
    * Otros diseñadores del Lenguaje
    * Implementadores
    * Programadores

Definición Formal de Lenguajes

- **Reconocedores**
  - Un dispositivo de reconocimiento que lee cadenas del lenguaje y decide si las cadenas de entrada pertenecen al Lenguaje.
  - Ejemplo, el analizador sintáctico de un compilador.
- **Generadores**

  - Un dispositivo que genera sentencias de un lenguaje
  - Se puede determinar si la sintaxis de una sentencia particular es correcta comparándola con la estructura del generador.

Métodos Formales de Describir la Sintaxis

- Forma Backus-Naur y gramáticas libres de contexto
  - El método mas conocido para describir la sintaxis de un Lenguaje de Programación.
- BNF Extendida
  - Mejora la legibilidad de BNF
- Gramáticas y Reconocedores

BNF y Gramáticas Libres de Contexto

- Gramáticas libres de Contexto
  - Desarrollado por Noam Chomsky a mediados de 1950s
  - Generadores de Lenguajes, medio de describir la la sintaxis de lenguajes naturales
  - Define clases de lenguajes
- Forma Backus-Naur (1959)
  - Inventado por John Backus para describir Algol 58
  - Árboles sintacticos - ambiguedad del lenguaje

Semántica Estática

Gramáticas con atributos

- Las Gramáticas Libres de Contexto (GLC) no pueden describir toda la sintaxis de los lenguajes de programación.
- Agregados a GLC para introducir información semántica en los árboles sintácticos
- Principal aporte de las Gramáticas con atributos
  - Especificación de la semántica estática
  - Diseño de Compiladores (chequeo de semántica estática)

Gramáticas con atributos: Definición

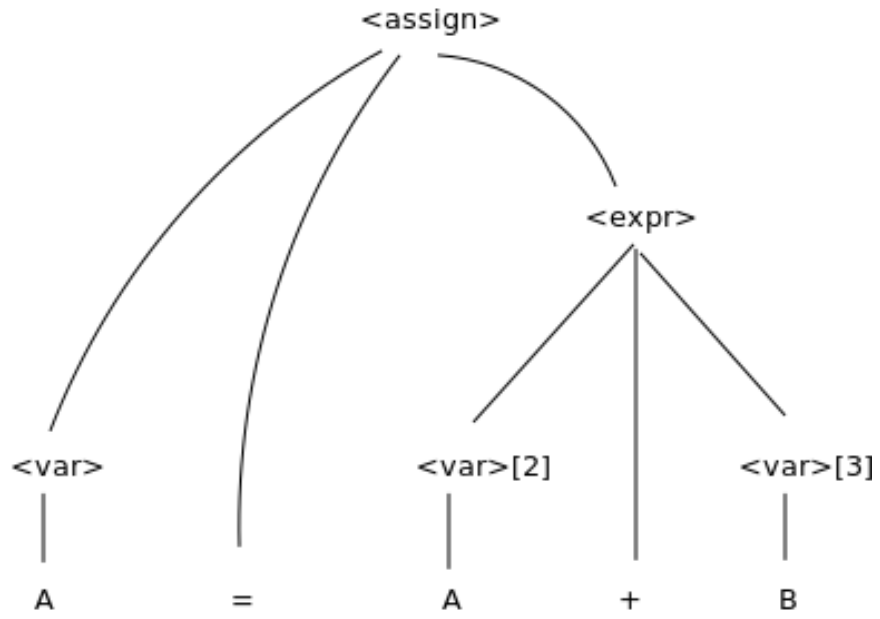
- Una Gramática con atributos es una gramática libre de contexto $G = (S, N, T, P)$ con los siguientes agregados:
  - Por cada símbolo de gramática x hay un conjunto $A(x)$ de valores de atributos
  - Cada regla tiene un conjunto de funciones que definen ciertos atributos de los no terminales en la regla
  - Cada regla tiene un conjunto posiblemente vacío de predicados para chequear la consistencia de los atributos

Gramáticas con atributos: Definición

- Sea $X_0 \rightarrow X_1...X_n$ una regla de la gramática libre de contexto
- Funciones de la forma $S(X_0) = f(A(X_1), ..., A(X_n))$ definen *atributos sintetizados*
- Funciones de la forma $I(X_j) = f(A(X_0)), ..., f(A(X_{j-1}))$ para $i <= j <= n$, definen *atributos heredados*
- Inicialmente hay *atributos intrínsecos* en las hojas de los árboles sintácticos

Gramáticas con atributos: Un Ejemplo

- Sintaxis
  - <assign> → <var> = <expr>
  - <expr> → <var> + <var> | <var>
  - <var> → A | B | C
- tipo-real: sintetizado por <var> y <expr>
- tipo-esperado: heredado por <expr>

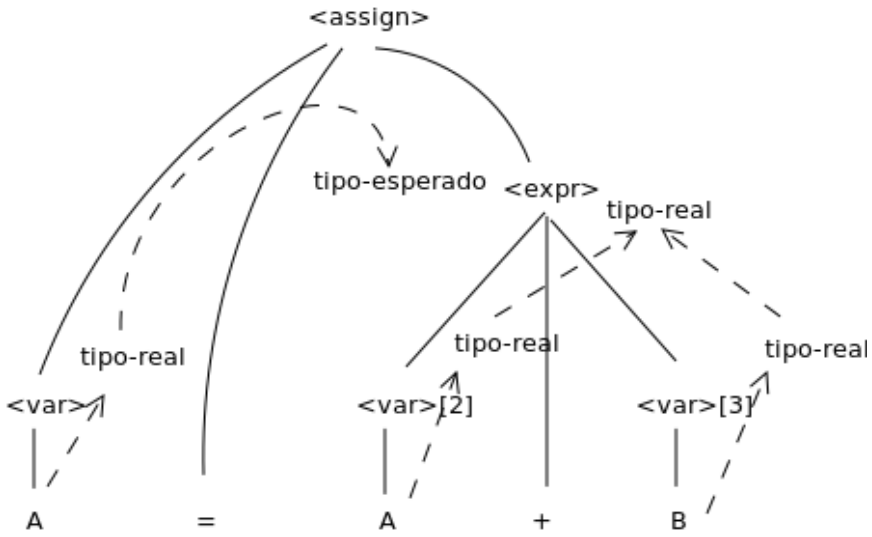


Gramáticas con atributos: Un Ejemplo

1. Regla sintáctica: <assign> → <var> = <expr>
   - Regla semántica: <expr>.tipo-esperado ← <var>.tipo-real
2. Regla sintáctica: <expr> → <var>[2] + <var>[3]
   - Regla semántica: <expr>.tipo-real ←
     if (<var>[2].tipo-real = int) and (<var>[3].tipo-real = int) then int else real end if
   - Predicado: <expr>.tipo-real = <expr>.tipo-esperado
3. Regla sintáctica: <expr> → <var>
   - Regla semántica: <expr>.tipo-real ← <var>.tipo-real
   - Predicado: <expr>.tipo-real = <expr>.tipo-esperado
4. Regla sintáctica: <var> → A | B | C
   - Regla semántica: <var>.tipo-real ← lookup (<var>.string)

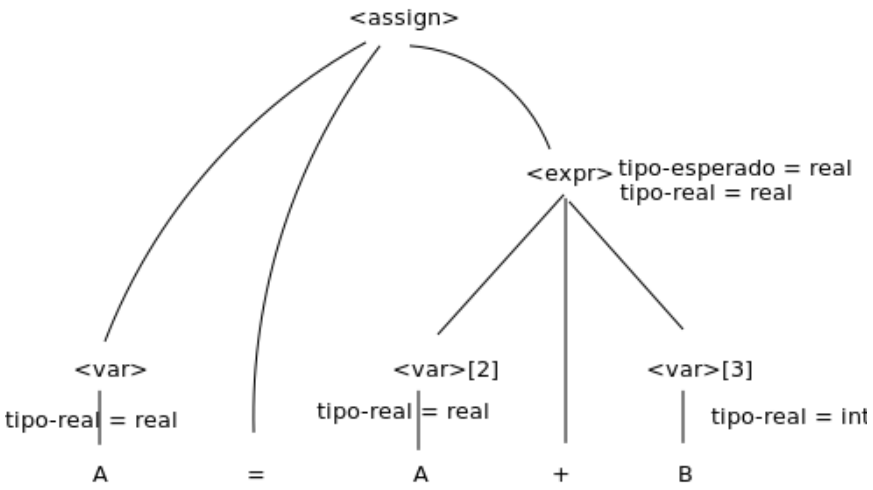
Gramáticas con atributos

- ¿Cómo se computan los valores de atributos?
  - Si todos los atributos fueran heredados, el árbol podría ser completado en un orden *top-down*.
  - Si todos los atributos fueran sintetizados, el árbol podría ser completado en un orden *bottom-up*
  - En muchos casos, ambos casos de atributos son utilizados y se necesita una combinación de ambos órdenes.



Gramáticas con atributos

1. <var>.tipo-real ← look-up(A) (Regla 4)
2. <expr>.tipo-esperado ← <var>.tipo-real (Regla 1)
3.
   - <var>[2].tipo-real ← look-up(A) (Regla 4)
   - <var>[3].tipo-real ← look-up(B) (Regla 4)
4. <expr>.tipo-real ← int o real (Regla 2)
5. <expr>.tipo-esperado = <expr>.tipo-real es VERDADERO o FALSO (Regla 2)



Semántica Dinámica

Métodos Desarrollados

- Semántica Operacional
  - Operaciones en una máquina abstracta
- Semántica Denotacional
  - Usa funciones para especificar la semántica, los programas se convierten en funciones para poder aplicar la teoría de funciones recursivas
- Semántica Axiomática
  - Aplica la lógica formal: afirmaciones (aserciones) para describir suposiciones y resultados deseados
  - Los axiomas o reglas de inferencia son usados en cada tipo de sentencias.
  - Marcelo

Semántica Operacional

Semántica Operacional

- Describe el significado de un programa ejecutando sus sentencias sobre una máquina, simulada o real. Los cambios en el estado de la máquina

(registros, memoria, etc) define el significado de la sentencia.

- Para el uso de una semántica operacional en un lenguaje de alto nivel se necesita una máquina virtual
  - Un intérprete de hardware puro podría ser muy costoso.
  - Un intérprete de software puro también tiene problemas (dependiente de la máquina)
- Una mejor alternativa: Una simulación completa de la computadora
  - Construir un traductor del codigo fuente a un codigo maquina de una computadora idealizada
  - Construir un simulador para la computadora idealizada

Semántica Operacional

- Simulador de Prolog en Prolog

```
mi(true).
mi((A,B)) :-
        mi(A),
        mi(B).
mi1(Goal) :-
        Goal \= true,
        Goal \= (_,_),
        clause(Goal, Body),
        mi(Body).
```

- Evaluación:
  - Bueno usado informalmente.
  - Extremadamente complejo usado formalmente.

Semántica Denotacional

Semántica Denotacional

- Basado en la teoría de funciones recursivas
- El método de descripción semántica mas abstracto
- Originalmente desarrollado por Scott y Strachey (1970)

- El proceso de construir una especificación denotacional para un lenguaje es definir un objeto matemático por cada entidad del Lenguaje
  - Define una función que relaciona instancias de las entidades del lenguaje con instancias de los objetos matemáticos correspondientes
- El significado de las construcciones del lenguaje son definidos solo por los valores de las variables del programa

Semántica Denotacional vs Semántica Operacional

- En la semántica operacional los cambios de estado son definidos por algoritmos codificados
- En la semántica denotacional los cambios de estado son definidos por funciones matemáticas rigurosas.
- El estado de un programa son los valores de todas las variables actuales $s = < i_1, v_1 >, < i_2, v_2 >, ..., < i_n, v_n >$
- Sea $VARMAP$ una función que, cuando recibe un nombre de variable y un estado retorna el valor actual de esa variable $VARMAP(i_j, s) = v_j$

Números Decimales

- <dec-num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
- M_{dec} ('0') = 0, M_{dec} ('1') = 1, . . . , M_{dec} ('9') = 9
- M_{dec} (<dec-num> '0') = 10 * M_{dec} (<dec-num>)
- M_{dec} (<dec-num> '1') = 10 * M_{dec} (<dec-num>) + 1
- . . .
- M_{dec} (<dec-num> '9') = 10 * M_{dec} (<dec-num>) + 9

Expresiones

- relaciona expresiones a Z ∪ { error }
- suponiendo que las expresiones son números decimales, variables, o expresiones binarias teniendo un operador aritmético y dos operandos, cada uno de los cuales puede ser una expresión.

- M_e (< expr >, s) = case < expr > of
  - < dec-num > → M_{dec} (< dec-num > , s)
  - < var > → if VARMAP(< var >, s)
  - < binary-expr > →
    - if (M_e(< binary-expr > . <left-expr > , s) = undef
      - OR M_e(< binary-expr > . < right-expr > , s) = undef)
    - then error
    - else
      - if (< binary-expr >.< operator > = '+' then
      - M_e(< binary-expr >.< left-expr >, s) + M_e(< binary-expr >.< right-expr >, s)
      - else M_e(< binary-expr >.< left-expr >, s) * M_e(< binary-expr >.< right-expr >, s)
- ...

asignación

- M_a (X := E, s) =
  - if M_e(E, s) = error
    - then error
    - else s' = { < i_1',v_1' >, < i_2',v_2' >, ... , < i_n',v_n' >},
      - where for j = 1, 2, ... n,
      - v_j' = varmap(i_j, s) if i_j <> x
      - = M_e(E, s) if i_j = x

Ciclo 'while'

- M_l(while B do L, s) =
  - if M_b(B, s) = undef
    - then error
    - else if M_b(B, s) = false
      - then s
      - else if M_{sl}(L, s) = error
      - then error
      - else M_l(while B do L, M_{sl}(L, s))

Ciclo

- El significado del ciclo es el valor de las variables del programa después de que las sentencias del ciclo han sido ejecutadas el número prescrito de veces, asumiendo que no ha habido errores
- En esencia el ciclo ha sido convertido de iterativo a recursivo, donde el control recursivo es definido por otra función recursiva de estados
- La recursión comparada con la iteración es mas facil de describir con rigor matemático

Evaluación

- Puede ser usado para probar la corrección de programas
- Provee un modo riguroso de pensar los programas
- Puede ser una ayuda al diseño de lenguajes
- Ha sido usado en sistemas de generación de compiladores
- A causa de su complejidad es de poco uso para los usuarios del lenguaje

Semántica Axiomática

Semántica Axiomática

- Basado en Lógica Formal (cálculo de predicados)
- Propósito original: Verificación formal de programas
- Axiomas o reglas de inferencia son definidas para cada tipo de sentencia del lenguaje (para permitir transformaciones de expresiones a otras expresiones)
- Las expresiones son llamadas *aserciones* (afirmaciones)

Semántica Axiomática

- Una aserción antes de una sentencia (una *precondición* establece las relaciones y restricciones entre variables que son verdaderas en ese punto de la ejecución
- Una aserción que sigue a una sentencia es una *postcondición*

- Una *precondición mas débil* es la menos restrictiva precondición que garantiza la postcondición

Semántica Axiomática

- La Forma es {P} sentencia {Q}
- Un ejemplo
  - a = b + 1 {a > 1}
  - una posible precondición: {b > 10)
  - *precondición mas débil*: {b > 0}

Proceso de prueba de programa

- La postcondición para el programa entero es el resultado deseado
  - Se trabaja hacia atrás a través del programa hasta la primer sentencia. Si la precondición sobre la primer sentencias está inferida por la especificación de entrada del programa, entonces el programa es correcto.

Axiomas

- Un axioma para la asignación
  - (x = E): $\{Q_{x \to E}\}\ x = E\ \{Q\}$
- La regla de la Consecuencia

$$\frac{\{P\}\ S\ \{Q\}, P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

Axiomas

- $x = 2 * y - 3\{x > 25\}$
- $2 * y - 3 > 25$
- $y > 14$
- $x = x + y - 3\{x > 10\}$

- $x + y - 3 > 10$
- $y > 13 - x$

Axiomas

$$\frac{\{x > 3\}\ x = x - 3\ \{x > 0\}, (x > 5) \Rightarrow (x > 3), (x > 0) \Rightarrow (X > 0)}{\{x > 5\}\ x = x - 3\ \{x > 0\}}$$

Axiomas

- Una regla de inferencia para secuencias
  - $\{P1\}S1\{P2\}$
  - $\{P2\}S2\{P3\}$

$$\frac{\{P1\}\ S1\ \{P2\}, \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1; S2\ \{P3\}}$$

Axiomas

- $y = 3 * x + 1;$
- $x = y + 3;$
- $\{x < 10\}$

La precondición para la segunda asignación es $y < 7$ la cual es usada como postcondición para la primer sentencia. La precondición para la primera asignación puede ser computada

- $3 * x + 1 < 7$
- $x < 2$

Axiomas

- regla de inferencia para sentencias de selección *if*

  {P} **if** B **then** S1 **else** S2 {Q}

$$\frac{\{B\ and\ P\}\ S1\ \{Q\}, \{(notB)\ and\ P\}\ S2\ \{Q\}}{\{P\}\ if\ B\ then\ S1\ else\ S2\ \{Q\}}$$

Ejemplo

- **if** $x > 0$ **then** $y = y - 1$ **else** $y = y + 1$
- con la postcondición $\{y > 0\}$
- el axioma de asignación para la clausula **then**: $y = y - 1\{y > 0\}$ produce $\{y - 1 > 0\}$ o $\{y > 1\}$
- el axioma de asignación para la clausula **else**: $y = y+1\{y > 0\}$ produce $\{y + 1 > 0\}$ o $\{y > -1\}$
- Como $\{y > 1\} \Rightarrow \{y > -1\}$ la regla de consecuencia nos permite usar $\{y > 1\}$ como precondición del total de la sentencia

Axiomas

- Una regla de inferencia para un ciclo *while*

  {P} **while** B **do** S **end** {Q}

$$\frac{(I \; and \; B)S\{I\}}{\{I\} \; while \; B \; do \; S\{I \; and (not B)\}}$$

  donde I es el *invariante* (la hipótesis inductiva)

Axiomas

- Características del *invariante*: I debe satisfacer las siguientes condiciones:
  - $P \Rightarrow I$ el invariante debe ser inicialmente verdadero
  - $\{I\}$ B $\{I\}$ la evaluación de la parte booleana no debe cambiar la validez de I
  - $\{I \; and \; B\}$ S $\{I\}$ I no cambia por la ejecución del cuerpo del ciclo iterativo
  - $(I \; and \; (not \; B)) \Rightarrow Q$ si I es verdadero y B es falso es implicado Q
  - El ciclo termina

Ejemplo

- **while** $y <> x$ **do** $y = y + 1$ **end** $\{y = x\}$
- Para cero iteraciones la precondición mas débil es $\{y = x\}$
- Para una iteración es:

$$wp(y = y + 1, \{y = x\}) = \{y + 1 = x\} = \{y = x - 1\}$$

- Para dos iteraciones es:

$$wp(y = y + 1, \{y = x - 1\}) = \{y + 1 = x - 1\} = \{y = x - 2\}$$

- Para tres iteraciones es:

$$wp(y = y + 1, \{y = x - 2\}) = \{y + 1 = x - 2\} = \{y = x - 3\}$$

- Es obvio que $\{y < x\}$es suficiente para los casos de uno o mas iteraciones. Combinado con $\{y = x\}$para el caso base obtenemos $\{y <= x\}$, que puede ser el invariante del ciclo.

Ejemplo

- $P \Rightarrow I$ $\{y <= x\} \Rightarrow \{y <= x\}$
- $\{I\}$ B $\{I\}$ $\{y <= x\}$ $\{y <> x\}$ $\{y <= x\}$
- $\{I \; and \; B\}$ S $\{I\}$ $\{y <= x \; and \; y <> x\}$ $y = y+1$ $\{y <= x\}$ aplicando el axioma de asignación a $y = y+1\{y <= x\}$ tenemos $\{y+1 <= x\}$ que es equivalente a $\{y < x\}$ el cual es implicado por $\{y < x \; and \; y <> x\}$.
- $(I \; and \; (not \; B)) \Rightarrow Q$ $\{(y <= x) \; and \; (not \; y <> x)\} \Rightarrow \{y = x\}$ sigue $\{(y <= x) \; and \; (y = x)\} \Rightarrow \{y = x\}$ sigue $\{y = x\} \Rightarrow \{y = x\}$
- El ciclo termina

Invariante

- El invariante es la versión mas debil de la postcondición del ciclo, y es también una precondición.
- Debe ser lo suficientemente debil para satsifacer a priori el comienzo del ciclo, pero cuando se combina con la condición de salida debe ser los suficientemente fuerte para forzar la verdad de la postcondición

Evaluación

- Desarrollar axiomas y reglas de inferencia para todas las sentencias en un lenguaje es dificultoso
- Es una buena herramienta para la verificación de programas y un excelente marco para razonar los programas, pero no es útil para los usuarios del lenguaje y desarrolladores de compiladores