CAPÍTULO 7: EXPRESSIONS AND ASSIGNMENT STATEMENTS EXPRESIONES Y SENTENCIAS DE ASIGNACIÓN

Estructura de los lenguajes

Dr. Christian von Lücken

Tópicos

Introducción
Expresiones aritméticas
Operadores sobrecargados
Conversiones de tipo
Expresiones relacionales y booleanas
Evaluación en corto-circuito
Sentencias de asignación
Asignaciones de modo mixto

Introducción

Las expresiones son los medios fundamentales para especificar cómputos en los lenguajes de programación

Para entender la evaluación de expresiones es necesario familiarizarse con el orden de los operadores y la evaluación de los operandos

Un elemento esencial de los lenguajes imperativos es el rol dominante que tienen las sentencias de asignación

Expresiones aritméticas

La evaluación aritmética fue una de las motivaciones para el desarrollo de los primeros lenguajes de programación

Las expresiones aritméticas consisten de operadores, operandos, paréntesis, y llamadas a funciones

En la mayoría de los lenguajes, los operadores binaries son infijos, excepto en Scheme y en LISP, en donde estas son prefijas; Perl también tiene algunos operadores binarios prefijos

La mayoría de los operadores unarios son prefijos, pero los operadores ++ y – en los lenguajes basados en C pueden ser tanto prefijos como posfijos

Cuestiones de diseño de las expresiones artiméticas

Cuestiones de diseño

- reglas de precedencia de operadores
- reglas de asociatividad de operadores
- orden de la evaluación de operandos
- efectos colaterales de la evaluación de operandos
- sobrecarga de operadores
- expresiones de modo mixto

Expresiones aritméticas: operadores

Aridad : número de operandos de un operador

Un operador unario tiene un operando

Un operador binario dos

Un operador ternario tres

Expresiones aritméticas: reglas de precedencia de operadores

Las reglas de precedencia de operadores para la evaluación de expresiones define el orden en el cual operadores "adyacentes" de niveles de precedencia diferentes son evaluados

Niveles típicos de precedencia

paréntesis

operadores unarios

** (si el lenguaje soporta)

*, /

+, -

Expresiones aritméticas: reglas de asociatividad de operadores

Las reglas de asociatividad de operadores para la evaluación de expresiones define el orden en el cual operadores adyacentes con el mismo nivel de precedencia son evaluados

Reglas típicas de asociatividad

- De izquierda a derecha, excepto **, que es de derecha a izquierda
- A veces los operadores unarios se asocian de derecha a izquierda (ej., en FORTRAN)

APL es diferente; todos los operadores tienen igual nivel de precedencia y todos los operadores se asocian de derecha a izquierda

Las reglas de precedencia y asociatividad pueden ser modificadas con paréntesis

Expresiones en Ruby y Scheme

Ruby

- Todos los operadores aritméticos, relacionales y de asignación, así como el indexamiento de arrays, operadores de corrimiento y bit, son implementados como métodos
- Un resultada de esto es que estos operadores pueden ser sobreescritos por los programas

Scheme (y Common Lisp)

- Todas las operaciones aritméticas y lógicas son llamadas explicitas a subprogramas
  - `a + b * c` se codifica como `(+ a (* b c))`

Expresiones aritméticas: condicionales

Expresiones condicionales

- Lenguajes basados en C (ej., C, C++)
- Ejemplo:

```
average = (count == 0)? 0 : sum / count
```

- Evalúa como si se escribiera

```
if (count == 0)
        average = 0
else
        average = sum /count
```

Expresiones aritméticas: orden de evaluación de operandos

Orden de evaluación de operandos

1. Variables: buscan el valor de la memoria
2. Constantes: a veces buscan el valor de la memoria; a veces la constante esta en instrucción de lenguaje máquina
3. Expresiones entre paréntesis: evalúa todos los operadores y operandos primero
4. El caso más interesante es cuando un operando es una llamada a función

Expresiones aritméticas: potenciales para efectos colaterales

Efectos colaterales funcionales: cuando una función cambia un parámetro de dos vías o una variable no-local

Problemas con los side-effects de las funciones:

Cuando una función referenciada en una expresión altera a otro operando de la expresión; ej. para un cambio de parámetros:

```
a = 10;
/* asuma que fun cambia el valor de a*/
b = a + fun(&a);
```

Efectos colaterales funcionales

Dos posibles soluciones

1. Escribir la definición de los lenguajes para eliminar los efectos colaterales funcionales

   No existen parámetros de dos vías

   No permitir referencias no-locales en las funciones

   Ventaja: funciona!

   Desventaja: inflexibilidad de parámetros de dos vías y referencias no-locales

2. Escribir la definición del lenguaje para demandar que el orden de evaluación de operandos sea fijo

   Desventajas: limita algunas optimizaciones del compilador

   Java require que los operandos aparezcan para ser evaluados de izquierda a derecha

Transparecia referencial

Un programa tiene la propiedad de transparencia referencial si cualquiera de dos expresiones en el programa que tienen el mismo valor pueden ser sustituida una por otra en cualquier parte del programa, sin afectar la acción del programa

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

si `fun` no tiene side effects, `result1 = result2`

De otra forma, no, y la transparencia referencial es violada

Transparencia referencial (continuación)

La ventaja de la transparencia referencial

- La semántica de un programa es mucho más fácil de entender si este tiene transparencia referencial

Debido a que estos no tienen variables, los programas en los lenguajes puramente funcionales son referencialmente transparentes

- Las funciones no pueden tener estados, el cual puede ser guardado en variables locales
- Si una función usa un valor desde afuera, este debe ser un valor constante (no existen variables). Entonces, el valor de la función depende sólo de sus parámetros

Operadores sobrecargados

El uso de un operador para más de un propósito

Algunos son comunes (ej., + para `int` y `float`)

Algunos son potencialmente problemáticos (ej., * en C y C++)

- Perdida de detección de errores del compilador (omisión de un operando debería ser un error detectable)
- Alguna pérdida de facilidad de lectura
- Puede evitarse incluyendo nuevos símbolos(ej., el **`div`** de Pascal para la división de enteros)

Operadores sobrecargados(continuación)

C++ , C#, F# y Ada permiten operadores sobrecargados definidos por el usuario

Cuando son utilizados adecuadamente, tales operadores pueden ayudar a la legibilidad (evitando llamada a funciones, expresiones que parecen naturales)

Problemas potenciales:

- Los usuarios pueden definir operaciones sin sentido
- La facilidad de lectura puede sufrir, incluso cuando los operadores tienen sentido

Conversiones de tipo

Una conversión *narrowing* es una que convierte un objeto de un tipo a otro y que no puede incluir todos los valores del tipo original ej., `float` a `int`

Una conversión *widening* es una en donde un objeto se convierte de un tipo a otro que puede incluir al menos aproximaciones a todos los valores del tipo original ej., `int` a `float`

Conversiones de tipo: Modo Mixto

Una expresión de modo mixto es una que tiene operandos de diferentes tipos

Una *coercion* es una conversión de tipo implícita

La desventaja de las *coerciones*:

- Disminuyen la habilidad de la detección de errores del compilador

En la mayoría de los lenguajes, todos los tipos numéricos son convertidos en expresiones, utilizando conversiones widening

En ML y F#, no existen coerciones en expresiones (Ada)

Conversiones de tipo explícitas

Llamado *casting* en lenguajes basados en C

Ejemplos

```
C: (int) angle
Ada: Float (sum)
F#: float(sum)
```

Note que la sintaxis de Ada (F#) es similar al de las llamadas a funciones

Conversiones de tipo: Errores en Expresiones

- Causas
  - Limitaciones inherentes de la aritmética ej., división por cero
  - Limitaciones de la aritmética del computador ej. overflow
- Usualmente ignorado por el sistema de tiempo de ejecución

Expresiones relacionales y booleanas

Expresiones relacionales

- Uso de operadores relacionales y operandos de varios tipos
- Evalua algunas representaciones booleanas
- Simbolos de operadores varian entre lenguajes (`!=`, `/=`, `.NE.`, `<>`, `#`)

JavaScript y PHP tienen dos operadores relacionales adicionales, `===` y `!==`

- Similar a sus primos, `==` y `!=`, excepto que estos no forzan la conversión de sus operandos
- Ruby utiliza `==` para la relación de igualdad que utiliza coerción y `eql?` Para las que no

Expresiones relacionales y booleanas

Expresiones booleanas

- Los operandos son booleanos y el resultado es booleano
- Operadores de ejemplo

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not
			xor

Expresiones relacionales y booleanas: no existe tipo booleano en C89

C no tiene un tipo booleano, usa el tipo `int` con 0 para falso y no-cero para verdadero

Una característica inesperada de C:

`a < b < c` es una expresión legal el resultado es:

- El operador de la izquierda se evalua, produciendo 0 o 1
- El resultado de la evaluación se compara con el tercer operando (i.e., `c`)

Expresiones relacionales y booleanas: precedencia de operadores

Precedencia de los operadores basados en C

```
prefix ++, --
unary +, -, prefix ++, --, !
*,/,%
binary +, -
<, >, <=, >=
=, !=
&&
||
```

Evaluación en corto circuito

Una expresión en donde el resultado está determinado sin evaluar todos los operandos y/o operadores

Ejemplo: `(13*a) * (b/13-1)`

Si `a` es cero, no existe necesidad de evaluar `(b/13-1)`

El problema con la evaluación sin corto circuito

```
index = 0;
while (index <= length) && (LIST[index] != value)
        index++;
```

Cuando `index=length, LIST [index]` causará un problema de índices (asumiendo que `LIST` tiene `length -1` elementos)

Evaluación en corto circuito(continuación)

C, C++, y Java: utilizan evaluación en corto circuito para los operadores booleanos usuales (**&&** and **||**), pero también provee operadores a nivel de bit que no son corto-circuitados (**&** and **|**)

Ada: el programador puede especificar ambos (short-circuit se especifica con `and then` y `or else`)

Todos los operadores lógicos en Ruby, Perl, ML, F#, y Python son evaluados en corto circuito

La evaluación en corto circuito expone el problema potencial de efectos secundarios en expresiones ej. `(a > b) || (b++ / 3)`

Sentencias de asignación

Sintaxis general

```
<target_var> <assign_operator> <expression>
```

El operador de asignación

= FORTRAN, BASIC, PL/I, C, C++, Java

:= ALGOLs, Pascal, Ada

= puede ser malo cuando es sobrecargado con el operador relacional para la igualdad (esto es el porqué los lenguajes basados en C utilizan == como el operador relacional)

Sentencias de asignación: Destinos condicionales

Destinos condicionales (C, C++, y Java)

```
(flag)? total : subtotal = 0
```

Lo que equivale a

```
if (flag)
  total = 0
else
  subtotal = 0
```

Sentencias de asignación: Destinos condicionales

Destinos condicionales (Perl)

```
($flag ? $total : $subtotal) = 0
```

Lo que equivale a

```
if ($flag){
  $total = 0
} else {
  $subtotal = 0
}
```

Sentencias de asignación: operadores compuestos

Un método abreviado para especificar una forma común de asignación

Introducido en ALGOL; adoptado por C y lenguajes basados en C

Ejemplo

```
a = a + b
```

se escribe como

```
a += b
```

Sentencias de asignación: operadores de asignación unarios

Los operadores unarios de asignación en los lenguajes basados en C combinan operaciones de incrementos y decrementos con asignaciones

Ejemplos

- `sum = ++count` (`count` incrementado, sumado a `sum`)
- `sum = count++` (`count` incrementado, sumado a `sum`)
- `count++` (`count` incrementado)
- `-count++` (`count` incrementado luego negado)

Asignación como expresión

En lenguajes basados en C, Perl y JavaScript, la sentencia de asignación produce un resultado que puede ser utilizado como un operando

```
while ((ch = getchar())!= EOF){...}
```

`ch = getchar()` se ejecuta; el resultado asignado a `ch` se utiliza como un valor condicional para la sentencia `while`

Desventaja: otro tipo de efecto colateral de expresión

Asignaciones múltiples

Perl y Ruby permiten asignaciones múltiples

```
($first, $second, $third) = (20, 30, 40);
```

También, lo siguiente es legal y realiza un intercambio:

```
($first, $second) = ($second, $first);
```

Asignación en lenguajes funcionales

Los identificadores en lenguajes funcionales son solo nombres de valores

ML

Los nombres son ligados a los valores con **`val`**

```
val fruit = apples + oranges;
```

- Si hay otro val para fruit, este será uno nuevo y un nombre diferente

F#

El **`let`** de F# es como el **`val`** de ML, excepto que **`let`** también crea un nuevo alcance

Asignación de modo mixto

Las sentencias de asignación pueden ser también de modo mixto, por ejemplo

```
int a, b;
float c;
c = a / b;
```

En Pascal, las variables enteras pueden ser asignadas a variables reales, pero las variables reales no pueden ser asignadas a enteros

En Java y C#, solo las asignaciones tipo widening son permitidas

En Ada, no existe forma de *coerción*

En Fortran, C, Perl, y C++, cualquier tipo número puede ser asignado a cualquier variable de tipo numerico

Resumen

Expresiones

Precedencia de operadores y asociatividad

Sobrecarga de operadores

Expresiones de modo mixto

Varias formas de asignación