Expresiones y Asignaciones

miércoles, 13 de octubre de 2021 12:14

- **Introducción**
  - Las expresiones son el medio fundamental para especificar en un lenguaje computacional lo que debe realizar la maquina
  - El valor de las expresiones depende del orden de evaluación de operadores y operandos
  - Ambigüedades en el orden de evaluación puede conducir a diferentes resultados
  - Lenguajes Imperativos:
    - Dominados por asignaciones a variables que modifican el estado del programa, con potenciales efectos laterales
  - Lenguajes Funcionales:
    - Las variables se usan como parámetros de funciones
    - No producen efectos laterales

- **Expresiones Aritméticas**
  - Consisten en operadores, operandos, paréntesis y llamadas a funciones
  - El orden de evaluación determina su valor
  - Aridad de los operadores:
    - Unario tiene un solo operando
    - Binario tiene dos operandos
    - Ternario tiene tres operandos → (a>b)? a:b 3 operandos
  - Evaluación de operadores:
    - Precedencia:
      - Se prioriza el orden de evaluación de mayor a menor nivel de precedencia
    - Asociatividad:
      - Define el orden de operadores adyacentes con el mismo nivel de precedencia
    - Paréntesis:
      - Altera y fuerza el orden de evaluación
    - El orden determina el valor y precisión:
      - Variables:
        - Leídas desde la memoria
      - Constantes:
        - Puede ser leída desde la memoria o desde las instrucciones de la maquina
      - Paréntesis:
        - Evalúa todos los operandos y operadores contenidos primero
        - Luego su valor puede ser usado como operando
    - Efectos Laterales:
      - Si la evaluación de un operando altera el valor de otro en una expresión
      - Si ninguno de los operandos de un operador tiene efectos laterales, el orden de evaluación de estos es irrelevante
      - Casos de efecto lateral:
        - Funciones con parámetros bidireccionales
        - Referencias a variables no locales
      - ¿Cómo evitarlos?
        - Definir un lenguaje que deshabilite efectos laterales en la evaluación de funciones, pero quita flexibilidad al lenguaje
        - Imponer un orden de evaluación a las funciones, pero evita que el compilador haga optimizaciones
    - Transparencia Referencial:
      - Propiedad de un programa, donde si cualquier par de expresiones en el programa producen el mismo valor pueden ser intercambiadas sin afectar la acción del programa
      - Se relaciona con efectos laterales de funciones
      - Tiene la ventaja de que los programas son más fáciles de entender

Reglas de Precedencia:

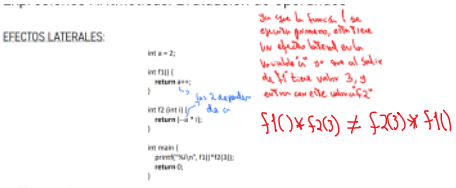
Fortran	Pascal	C
**	*, /, div, mod	Postfix ++, --
*, /	+, -	Prefix ++, --
+, -		+, - (unario)
		*, /, %
		+, - (binario)

Alta ↓ Baja

Reglas de Asociatividad:

FORTRAN	**Izq**: *, /, +, - **Der**: **
Pascal	Todos por la izquierda
C	**Izq**: ++ y -- postfijo; *, /, %, + y – binario **Der**: ++ y -- prefijo; + y – unarios
C++	**Izq**: *, /, %, + y – binario **Der**: ++, --, - y + unario

EFECTOS LATERALES:



Ejemplo:

```
res1 = (fun(a) + b) / (fun(a) – c);

temp = fun(a);
res2 = (temp + b) / (temp – c);
```

- Si fun() no tiene efectos laterales, entonces: res1 = res2
- ¡En otro caso, la transparencia referencial es violada!

```
int   a, b, c;        c = a + b;   -> suman int con int
float u, v, w;        w = u + v;   -> suman float con float
                      u = a + w;   -> suman int con float
```

en los 3 casos el operador es el +, independiente del tipo de dato

- **Sobrecarga de Operadores**
  - Un mismo operador es usado para diferentes propósitos, siendo que tienen distinto comportamiento
  - Mejor legibilidad
  - Algunos lenguajes permiten dar nuevos significados a los símbolos de operadores (C# y C++)
  - En algunos casos de sobrecarga de operadores la lógica es diametralmente diferente (& se podría entender como *AND* en un condicional, o *como dirección* de un puntero)

- **Conversión de Tipos**
  - Por expansión:
    - Es más segura
    - Convierte un objeto a un tipo que incluye todos los valores del tipo original
    - Ej: De entero a punto flotante o De subrango de enteros a entero
  - Por estrechamiento:
    - Se puede perder información
    - Convierte un objeto a un tipo que no puede incluir todos los valores del tipo original
    - Ej: De real a entero o Paso de tipo base a subrango
  - Conversión Explicita:
    - Una expresión puede tener una mezcla de operandos de diferentes tipos, por lo que el lenguaje hace una conversión implícita (coerción) para usar el operando adecuado
    - Da mayor flexibilidad al uso de operadores, pero reduce la capacidad de detectar errores
    - El programador convierte explícitamente mediante "*castings*" ((int) ángulo)
  - Errores en expresiones:
    - El lenguaje realiza verificación de tipos (estática/dinámica) para evitar errores. Pero algunos casos pueden llegar a ocurrir por las siguientes causas:
      - Coerción de operandos en las expresiones
      - Rango limitado de representación de números
      - División por cero
    - Algunos lenguajes producen **excepciones** en estos casos

- **Expresiones Relacionales y Booleanas**
  - Expresión Relacional:
    - Usa un operador relacional (binario) para comparar los valores de dos operandos
    - El valor de la expresión será booleano a menos que no exista en el lenguaje
    - ==, !=, <, <=, >, >=
  - Expresión Booleana:
    - Consiste de variables, constantes y operadores booleanos
    - Se puede combinar con expresiones relacionales
    - AND, OR , NOT, XOR
  - La precedencia de los operadores booleanos está definida de mayor a menor:
    1) NOT
    2) OR
    3) AND
  - La excepción es ADA que todos tienen igual precedencia
  - Los operadores aritméticos tienen mayor precedencia que los relaciones, y los relacionales tienen mayor que los booleanos (Excepto en Pascal)

Operadores relacionales:

Operación	Pascal	C	Ada	Fortran
Igual	=	==	=	.EQ.
No es igual	<>	!=	/=	.NE.
Mayor que	>	>	>	.GT.
Menor que	<	<	<	.LT.
Mayor o igual que	>=	>=	>=	.GE.
Menor o igual que	<=	<=	<=	.LE.

```
C:      b + 1 > b*2     /* aritméticos primero */
C:      a > 0 || a < 5  /* relacional primero */
Pascal: a > 0 OR a < 5  {es ilegal}
```

- **Cortocircuito**
  - Es el termino anticipado de evaluación
  - Por ejemplo, en el caso de un *OR*, si la expresión de la izquierda es verdadera, entonces la evaluación finalizara ahí ya que basta con que uno de los términos del *OR* sea verdadero para que toda la expresión sea verdadera
  - C, C++ y Java definen el cortocircuito para los operadores lógicos:
    - && o AND

        - || o OR
    - Perl, Python, Ruby evalúan todos sus operadores lógicos con cortocircuito
    - Pascal y Fortran no especifican a cuales les ponen cortocircuito

- **Sentencias de Asignación**
    - Mecanismo que permite cambiar dinámicamente el valor ligado a una variable
    - Fortran, Basic, PL/I, C, C++ y Java usan =
    - Algol, Pascal, ADA usan :=
    - C, C++ y Java permiten incrustar una asignación en una expresión
    - Hay ejemplos de asignaciones en el PDF y también la asignación en expresiones

Estructuras de Control y Ejecución de Sentencias

miércoles, 13 de octubre de 2021 12:14

- **Estructura del Flujo de control**
  - La ejecución es típicamente secuencial, quedando implícitamente definida por el orden de definición de sentencias
  - Se requieren dos tipos de sentencias de control para alterar esta secuencia:
    - Selección:
      - Ofrece múltiples alternativas de ejecución, controladas por una condición
    - Iteración:
      - Ejecución repetitiva de un grupo de sentencias, también controladas por una condición
  - Se requiere además una composición de sentencias (Mecanismo de agrupación de sentencias sobre las cuales se ejerce el control)

- **Sentencias Compuestas**
  - Mecanismo que permite agrupar un conjunto de sentencias como una unidad o bloque
  - Ejemplos:
    - *Begin* y *end* en derivados de Algol (Pascal)
    - Paréntesis de llave {} en derivados de C (C++ y Java)
    - Python lo realiza por indentación

- **Sentencias de Selección**
  - Selección Binaria:
    - Dos alternativas (o una o ninguna)
    - Son las condicionales de toda la vida (*if, elif, else*)
  - Selección múltiple:
    - Múltiples alternativas
    - *Case* en Pascal
    - *Switch* C, C++n y Java
    - *Elseif* en ADA
- **Sentencias de Iteración**
  - Bucles controlados por contador:
    - Se especifica el valor inicial y final de una variable que controla el número de iteraciones
    - Ej:
      - *For* en C y Pascal
  - Bucles controlados por condición:
    - Hay una condición booleana de termino
    - La condición debe incluir una variable que es modificada por el bloque
    - Ej:
      - *While, do while* en C, C++ y Java
      - *Loop-exit* en ADA
  - Bucles controlados por Estructuras de Datos:
    - *Foreach* en Perl, Python y Ruby
  - Los lenguajes de programación tienen mecanismos de escape de iteración como el *continue (*Simula el término de una iteración del *while* y vuelve al comienzo*)* o *break* en derivados de C
- **Salto Incondicional**
  - Usa rótulos o etiquetas para especificar el punto de transferencia del control cuando se ejecuta una determinada sentencia de salto
  - No son una buena forma de programar

- Ejemplo: PHP

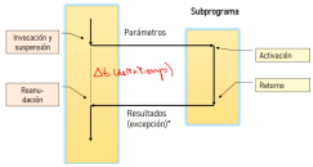
```php
<?php
        for($i=0,$j=50; $i<100; $i++) {
                while($j--) {
                if($j==17) goto end;
                }
}
echo "i = $i";
end:
echo 'j hit 17';
?>
```

Subprogramas

miércoles, 13 de octubre de 2021 12:14

- **Definición**
  - Describen una interfaz que se abstrae del proceso de computación definido por su implementación, que a través de un mecanismo de invocación permite su ejecución, con una posible transferencia de parametros y resultados
  - Encapsula código (Lo hace transparente para el invocador)
  - Permite reutilizar código
  - Requiere memoria dinámica de *stack*
- **Posibilidades**
  - Subrutinas y procedimientos (no definen valor de retorno)
  - Funciones (similar, pero con valor de retorno)
  - Métodos y constructores en lenguajes orientados a objeto

Subprogramas: Mecanismo de Invocación



- **Interfaces**
  - Se incluyen, al menos, los siguientes elementos:
    - Nombre:
      - Permite referenciar el subprograma e invocarlo
    - Parámetros:
      - Son opcionales
      - Define la comunicación de datos (nombre, orden y tipo de parámetros formales)
    - Valor de retorno:
      - Opción para funciones (tipificado)
    - Excepciones:
      - Son opcionales
      - Permite el manejo de un evento de excepción al retornar el control anormalmente
    - Firma/Prototipo:
      - Corresponde a la cabecera del subprograma
      - Define un contrato entre el invocador y el subprograma
      - Define la semántica de la interfaz para el intercambio de parámetros y resultados durante la invocación
      - Especifica sintácticamente un posible nombre, parámetros formales y retorno, con sus respectivos tipos
    - Protocolo:
      - Especifica como debe realizarse la comunicación de parámetros y resultados
      - Establece como se asocian los parámetros reales con los parámetros formales
      - Parámetros reales:
        - Son los que le pasamos al subprograma
      - Parámetros formales:
        - El subprograma asocia los parámetros reales a sus propios parámetros, los cuales son los formales

- **Parámetros**
  - Permiten comunicación explicita de datos y de otros subprogramas pasados por referencia
  - Parámetros Formales:
    - Variables mudas que se ligan a los *parámetros reales* cuando se activa el subprograma
    - El ligado se hace según la posición en la lista, definida en el protocolo
  - Comunicación Implícita/Indirecta:
    - El subprograma accede a variables no locales
    - Puede producir efectos laterales

```
procedure random(in real semilla; out real aleat);
```

```
float potencia(float base, float exp);

calculo = x * potencia(y, 2.5);
```

```
int notas[50];
...
void sort (int lista[], int largo);
...
sort(notas, 50);
```

C

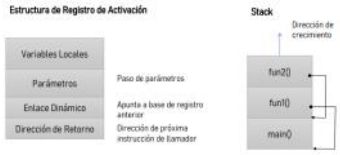
```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()
```

Python

- **Clases de Valores**
  - Según el tratamiento que permiten los lenguajes con el valor de variables que representan subprogramas (referencias o punteros), se definen las siguientes clases:
    - Primera Clase:
      - Puede ser pasado como parámetro o retornado en un subprograma
      - Puede ser asignado como variable
    - Segunda Clase:
      - Puede ser pasado como parámetro pero no retornado o asignado a una variable
    - Tercera Clase:
      - No puede ser un parámetro o ser retornado o asignado

- **Aspectos de Implementación**
  - Estructura:
    - Un subprograma consiste de dos partes:
      - Código:
        - Es el código real del subprograma
        - Es inmutable (ejecutable)
      - Registro de Activación:
        - Variables locales
        - Parámetros
        - Dirección de entorno
        - Etc.
  - Tipos de Implementación:
    - Simple:
      - No se permite anidamiento de un subprograma
      - Datos de registro de activación son estáticos
    - Stack:
      - Permite anidamiento de llamadas (O sea, puede llamar a otro subprograma dentro de un subprograma) usando variables dinámicas de *stack*
      - Soporta bien la recursión

Subprogramas: Registros de Activación



- **Paso de Parámetros: Modelos Semánticos**
  - Dirección:
    - Modo de interacción de parámetro actual a formal puede ser:
      - Entrega de valor (*IN*)
      - Recibo de valor (*OUT)*
      - Ambos (*INOUT)*
  - Implementación:
    - Implementación de transferencia de datos puede ser:
      - Copiando valores
      - Pasando referencias/punteros
  - Paso por Valor:
    - Modo *IN* es implementado normalmente con copia de valor
    - Implementación con paso de referencia requiere protección de escritura (Difícil)
    - Protege al parámetro real de posibles modificaciones (Costoso)
    - Requiere más memoria y tiempo de copiado (Mas seguro)
    - Permite usar expresiones como parámetro real
  - Paso por resultado:
    - Modo *OUT* es normalmente implementado con copia
    - Mismas complicaciones que en paso por valor
    - Parámetro formal actúa como variable local, pero al retornar copia del valor a parámetro real
    - Parámetro real debe ser una variable
    - Pueden existir colisiones en los parámetros reales (Ambigüedad)

Paso de Parámetros: Modelos Semánticos

- Paso por resultado:
  - Modo *OUT* es normalmente implementado con copia
  - Mismas complicaciones que en paso por valor
  - Parámetro formal actúa como variable local, pero al retornar copia del valor a parámetro real
  - Parámetro real debe ser una variable
  - Pueden existir colisiones en los parámetros reales (Ambigüedad)
- Paso por Valor-Resultado:
  - Modo *INOUT* con copia de parámetros en la entrega y en el retorno
  - A veces es llamada "Paso por copia"
  - Mismas dificultades que los dos anteriores
- Paso por referencia:
  - Modo *INOUT* es implementación con referencias
  - El parámetro formal y parámetro real comparten una misma variable
  - Ventajas:
    - Es más eficiente en el espacio ya que no requiere duplicar variables
    - Es más eficiente en el tiempo ya que no requiere copiar
  - Desventajas:
    - Acceso es más lento ya que usa una indirección (Es el *&* cuando se pasan las matrices en C por ejemplo)
    - Es una fuente de error ya que puede modificar el parámetro real
    - Creación de alias a través de parámetros reales
- Ejemplos:
  - C:
    - Paso por valor, y por referencia usando ***punteros*** (Parámetros deben ser des referenciados)
    - Punteros pueden ser clasificados como *const*, logra la semántica de paso por valor (sin permitir asignación)
    - Los arreglos se pasan por referencia ya que son punteros
  - C++:
    - Igual que C, más paso por referencia usando el operador *&* (Sin necesidad de des referenciar)
    - Este operador también puede ser clasificado con *const*, permitiendo la semántica de paso por valor con mayor eficiencia (ej: paso de grandes arreglos)}
  - ADA:
    - Por defecto es paso por valor, pero todos los parámetros se pueden calificar con *in, out* y *inout*
  - Pascal y Modula-2:
    - Por defecto paso por valor
    - Paso por referencia si se usa el calificativo *var*
  - Java:
    - Todos los objetos son pasados por referencia, solo los parámetros que no lo son se pasan por valor
    - No existen los punteros por lo que no se permite el paso por referencia de los tipos escalares
  - Python:
    - Se usa solo paso por referencia (paso por asignación) ya que en realidad todos los datos son objetos que tienen una referencia

Paso de Parámetros: Modelos Semánticos

Ejemplo:

```
procedure EJEMPLO1;
integer x;

procedure SUB (... integer PARAM)
begin
        x := 2;
        PARAM := PARAM +1;
end;

begin {EJEMPLO1}
        x := 1;
        SUB(X);
end
```

¿por valor?

¿por valor – resultado?

¿por referencia?

- **Paso de Parámetros: Aspectos de Implementación**
  - Comunicación de parámetro se realiza mediante el *stack*:
    - Por valor:
      - Al invocar, el valor de la variable se copia al *stack*
    - Por resultado:
      - Al retornar, el valor se copia del *stack* a la variable
    - Por valor-resultado:
      - Combinación de los dos anteriores
    - Por referencia:
      - Se escribe la dirección en el *stack* y luego se usa direccionamiento indirecto

- **Comprobación de Tipos**
  - Se hacen comprobaciones de tipos (Estática y dinámica), lo cual permite detectar errores en el mal uso de parámetros
  - Hace más confiables los programas
  - Pascal, Modula-2, Fortran 90, Java, ADA
  - ***Hay más casos de otros lenguajes en el PDF pero, aparte de para la prueba global, no creo que me sea muy útil***

- **Sobrecarga**
  - Si en el mismo ámbito, hay diferentes subprogramas con el mismo nombre
  - Cada versión debería tener una firma diferente, de manera que a partir de los parámetros reales se pueda resolver a cual versión se refiere
  - Las versiones pueden diferir en la codificación
  - Es una conveniencia notacional

Subprogramas: Sobrecarga en C++

- de Funciones:

```
double abs(double);
int abs(int);

abs(1);        // invoca int abs(int);
abs(1.0);      // invoca double abs(double);

void print(int);
void print (char*);   // se sobrecarga print
```

- de Operadores:

```
int operator* (const vector &a, const vector &b, int len) {
  int sum = 0;
  for (int i = 0; i < len; i++)
    sum += a[i] * b[i];
  return sum;
}
```

→ `vector x, y;`
`print("%i", x * y); // producto punto`

- Ejemplo de Función Genérica en C++:

```
template <class Tipo>
Tipo maximo (Tipo a, Tipo b)
{
        return a>b ? a : b;
}

int x, y, z;
char u, v, w;

z = maximo(x, y);
w = maximo(u, v);
```

- **Subprogramas Genéricos**
  - Permite crear diferentes subprogramas que implementan el mismo algoritmo, el cual actúa sobre diferentes tipos de datos
  - Mejora la reutilización y productividad en el proceso de desarrollo de software
  - Tipos de polimorfismos:
    - Polimorfismo Dinámico/Paramétrico:
      - El código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja
      - Puede ser usado con todo tipo de datos compatible
      - Ej: Funciones Genéricas
    - Polimorfismo Estático/*ad hoc*:
      - Los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizado
- **Clausura**
  - Es un subprograma y un ambiente referencial donde este fue definido (que permite invocarlo en diferentes ámbitos)

- Ejemplo: JavaScript

```
function makeAdder(x) {
    return function(y) {return x + y;}
}

var add10 = makeAdder(10);
var add5 = makeAdder(5);

document.write("add 10 to 20: " + add10(20) + "<br />");
document.write("add 5 to 20: " + add5(20) + "<br />");
```