Programación Funcional

Dr. Víctor de la Cueva
vcueva@itesm.mx

Tecnológico de Monterrey

Introducción

- Se inicia a finales de la década de los 50's.
- Su paradigma se centra en el Cálculo Lambda, introducido por Alonzo Church y Stephen Kleen en los 30's.
- El CL se puede considerar como el más pequeño lenguaje de programación.
- Consiste en una regla de transformación simple (sustitución de variables) y un esquema simple para definir funciones.
- La PF es muy útil en la IA: cálculo simbólico, pruebas de teoremas, sistemas basados en reglas y procesamiento de lenguaje natural.

Programación funcional

- Su características esencial es que los cálculos se ven como una función matemática que hace corresponder entradas y salidas.
- Sus objetivos:
  - Crear un lenguaje expresivo y matemáticamente elegante que evite el concepto de estado de cómputo.
  - Acercar su notación a la notación normal de la matemática (cosa que no ocurre en los lenguajes imperativos).

Estado de cómputo

- El estado de cómputo o estado de programa, se entiende como un registro (con una o más variables) del estado en el que se encuentra el programa en un momento dado.
- En la práctica, este registro de estado de programa se implementa con variables globales, de las que depende el curso de ejecución de alguna parte del programa.

Características de los LF

- Los programas escritos en LF están constituidos únicamente por definiciones de funciones, entendiendo estas, no como subprogramas clásicos sino como funciones matemáticas puras, en las que se verifican ciertas propiedades como la transparencia referencial.
- La transparencia referencial, quiere decir que el significado de una expresión depende únicamente del significado de sus subexpresiones o parámetros, no depende de cálculos previos, ni del orden de evaluación de sus parámetros o subexpresiones y, por lo tanto, implica la carencia total de efectos colaterales.

Otras características

- Otras características propias de estos lenguajes (consecuencia directa de la ausencia de estado de cómputo y de la transparencia referencial) son:
  - La NO EXISTENCIA de asignación de variables.
  - La falta de constructores estructurados como la secuencia o la iteración (no hay instrucciones para ciclos como for o while)
- Esto obliga en la práctica a que toda las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas.
- Nota: hay algunos LF híbridos (no puros) que usan asignación, e.g. LISP.

Ejemplos de LF

- Puros: Haskell y Miranda
- Híbridos: LISP, Scheme y Ocaml. Vamos a usar Scheme como si fuera puro.
- Erlang es un LF con programación concurrente.
- R es un LF dedicado a la estadística.
- Mathematica y Máxima, son lenguajes/entornos funcionales orientados totalmente al álgebra simbólica.
- Perl es un LF si sólo se usan funciones definidas por el usuario.
- Python, es un lenguaje que incorpora el paradigma funcional.

Efecto colateral (side effect)

- Cualquier cosa que haga un procedimiento, que persista después de que éste regresa un valor, se llama efecto colateral.
- La asignación a una variable sería evaluada para que produzca el efecto colateral de asignar un valor a una variable, el cual se mantiene después de hecha la asignación.
- Un procedimiento que sólo calcula y regresa un valor es llamado una función.
- Estrictamente hablando, de acuerdo a la convención matemática, los procedimientos que tienen efectos colaterales no son funciones (como la asignación).

Recursión

- La recursión es una de las herramientas fundamentales de los LF.
- Una función f es recursiva si su cuerpo contienen al menos una aplicación de f.
- De manera más general, una función f es recursiva si f puede activarse a sí misma, posiblemente de manera indirecta a través de otras funciones.
- E.g. la secuencia de Fibonacci, factorial, etc.

$$fib(n) = \begin{cases} 1 & \text{si } n = \{0, 1\} \\ fib(n-1) + fib(n-2) & \end{cases}$$

$1, 1, 2, 3, 5, \ldots$

$fib(0) = 1$

$fib(1) = 1$

$fib(2) = fib(1) + fib(0)$

$fib(4) = fib(3) + fib(2)$

$$f(n) = \begin{cases} 1 & n = 0 \\ n * f(n-1) & n > 0 \end{cases}$$

```
fun f(n) {
    if n=0
        return 1
    else
        n*f(n-1)
}
```

Recursión lineal

- La definición de una función f es recursiva lineal, si la activación f(a) de f puede iniciar como máximo UNA nueva activación de f.
- E.g. lineal: factorial, e.g. no lineal: Fibonacci.
- La evaluación de una función recursiva lineal tiene dos fases:
  - Fase de activación: en la cual se inician las nuevas activaciones.
  - Fase de solución: en la cual el control regresa a las activaciones con una modalidad LIFO (última entrada – primera salida).
- E.g. Factorial(3)

$$
\begin{aligned}
f(3) &= 3 * f(2) \\
&= 3 * (2 * f(1)) \\
&= 3 * (2 * (1 * f(0))) \\
&= 3 * (2 * (1 * 1)) \\
&= 3 * (2 * 1) \\
&= 3 * (2) \\
&= 6
\end{aligned}
$$

activación

solución

Recursión de cola (Tail recursión)

- Una función f tiene recursión de cola si devuelve un valor sin necesidad de recursión o si devuelve simplemente el resultado de una activación recursiva (la llamada recursiva es la última operación en la función).
- Todo el trabajo de una función recursiva de cola se realiza en la fase de activación, cuando se inician activaciones nuevas.
- La fase de solución es trivial debido a que el valor calculado por la activación final se convierte en el valor de toda la evaluación.
- Las funciones recursivas pueden implementarse con eficiencia si tienen recursión de cola ya que muchos compiladores pueden transformar esto a procesos iterativos (ciclos).

fun $g(n, \underline{a}) = \ldots$

$$g(n,a) = \begin{cases} a & \text{si } n = 0 \\ g(n-1, n * a) & \end{cases}$$

ej

$$g(3,1) \downarrow g(2,3) \downarrow g(1,6) \downarrow \underbrace{g(0,6)}_{6}$$

Trasformación

- Una función que es *no-tail recursive* SIEMPRE puede transformarse en tail recursive agregando en ella una función local que use un acumulador.
- NOTA: La función puede no ser local, pero si es local es mejor.
- E.g. Factorial tail y factorial no-tail.

No-tail

```
fact(n) {
  if (n==0)
    return 1;
  else
    return n*fact(n-1);
end
```

Tail

```
fact(n) {
  return
    fact1(n,1)
}

fact1(n,a) {
  if (n==0)
    return a;
  else
    return
    fact1(n-1, n*a);
}
```

Referencias

- R. Sethi. Programming Languages: concepts and constructs. Addison-Wesley, 2nd edition (1996).
- E. Navas. Programando con Racket 5. Versión 1.0 (2010).