Profesores: José Miguel Cañete Valdeón y Francisco José Galán Morillo

PROCESADORES DE LENGUAJES Proyecto

Lenguaje de Programación P (Breve especificación y evaluación del proyecto)

P es un lenguaje de programación secuencial con asertos. La idea es usar asertos para verificar la corrección del programa. Si al ejecutar un programa, sus asertos son ciertos entonces el programa se considera correcto. El programa sería incorrecto en otro caso. El programa P se estructura en 3 secciones: una dedicada a la declaración de variables (sección `VARIABLES`), otra dedicada a la declaración de subprogramas (sección `SUBPROGRAMAS`) y otra dedicada al uso de variables y subprogramas (sección `INSTRUCCIONES`).

```
PROGRAMA

VARIABLES
  ...

SUBPROGRAMAS
   ...

INSTRUCCIONES
   ...
```

Los *tipos de datos* considerados en P son dos clases (1) tipos elementales: tipo entero (`NUM`) y tipo lógico o booleano (`LOG`) y (2) tipos no elementales: tipo secuencia entera (`SEQ(NUM)`) y tipo secuencia lógica (`SEQ(LOG)`). Las constantes enteras son las convencionales (ej. `-1,0,1`)., las lógicas son `T` y `F`. y las constantes secuencia usan corchetes para encerrar sus elementos (ej. `[3,4,2], [T], []`).

Cada **variable** declarada en un programa tiene un tipo invariable en dicho programa.

Ejemplo de declaración de variables:

```
VARIABLES
   i,max,min:NUM; //i,max y min son variables numéricas y no pueden cambiar su tipo a lo largo
                  //del programa
   s:SEQ(NUM);    //similarmente, s es una secuencia de números
```

La declaración de una variable no asigna valor a ésta.
Las variables `NUM` almacenan valores enteros, las variables `LOG` valores lógicos y las variables `SEQ(NUM)/SEQ(LOG)` almacenan secuencias de enteros/secuencias de valores lógicos.

.
P predefine las siguientes operaciones para el tipo `NUM`: suma (+), resta (-) y multiplicación (*). Se considera por defecto que `*` es más prioritario que + y − y estos dos últimos operadores tienen la misma prioridad entre sí.

P no predefine ninguna operación para el tipo `LOG`.

P predefine dos funciones para las secuencias (a) un predicado `vacia` para testar si una secuencia es vacía y (b) una función `ultima_posicion` para devolver la última posición de una secuencia no vacía. Cualquier expresión numérica puede servir para codificar una posición en una secuencia. Por ejemplo, `s[i+1]` denota la selección del elemento `i+1` de la secuencia `s`. siendo `i` una variable tipo `NUM`. La selección de un elemento desde posición inexistente es un error. El primer elemento de una secuencia está en la posición `0`.

P es un lenguaje fuertemente tipado: las expresiones con distintos tipos no se pueden combinar. Ejemplo de expresiones sin tipo: `1+T, [] y F, []+[1,2]`

Los **subprogramas** en P pueden ser funciones o procedimientos.

La *función* tiene un conjunto (posiblemente vacío) de parámetros de entrada y un conjunto (no vacío) de parámetros de salida. Los parámetros de entrada son parámetros de sólo lectura y los parámetros de salida de lectura/escritura.

La función debe incluir en algún punto de sus instrucciones un retorno explícito de los parámetros de salida (instrucción `dev`).

Por ejemplo,

```
//pre: cierto
//post: max es el elemento máximo de s e i es la posición en la que
//      se encuentra.
 FUNCION mayor(SEQ(NUM) s) dev (NUM i, NUM max)
 VARIABLES
    j: NUM;
 INSTRUCCIONES
    j=0;
    max=s[j];
    mientras(j<=ultima_posicion(s)) hacer
       si (s[j]>max) entonces
         max=s[j];
         i=j;
       fsi
       j=j+1;
    fmientras
    dev max,i;  //retorno de parámetros de salida.
 FFUNCION
```

Llamaremos *predicado* a la función que devuelve un valor lógico. Un ejemplo de predicado se muestra en la siguiente función:

```
//pre: cierto
//post: r devuelve cierto si x es mayor que 5 y falso en caso contrario.
 FUNCION mayor_que_5(NUM x) dev (LOG r)
 VARIABLES
 INSTRUCCIONES
    si (x>5) entonces
       dev T;
    sino
       dev F;
    fsi
 FFUNCION
```

El *procedimiento* tiene un conjunto (posiblemente vacío) de parámetros de entrada y salida. No hay devolución explícita de resultados (no debe usarse la instrucción `dev`).

Por ejemplo,

```
//pre: cierto
//post: max es el elemento maximo de s e i es la posición en la que
//      se encuentra.
PROCEDIMIENTO mayor(SEQ(NUM) s, NUM i, NUM max)
VARIABLES
   j: NUM;
INSTRUCCIONES
   j=0;
   max=s[j];
   mientras(j<=ultima_posicion(s)) hacer
      si (s[j]>max) entonces
        max=s[j];
        i=j;
      fsi
      j=j+1;
   fmientras
FPROCEDIMIENTO
```

Las **instrucciones** de P son: asignación, condicional, iteración, ruptura de control, llamada a procedimiento o función, devolución de resultados de una función, mostrar por consola el valor de variables y aserto. El lenguaje P también dispone de comentarios de línea (`//`) y de bloque (`/* */`)

La *asignación* en P puede ser simple o múltiple. Por ejemplo,

```
s = [3,4,2,7,9,0,11]; //asignación simple
x,y = 1,x+2;          //asignación multiple
```

La asignación múltiple se interpreta de forma paralela, es decir, todas las asignaciones se hacen al mismo tiempo. Por ejemplo,

```
x = 1;          //asignación simple: x toma valor 1
y = 2;          //asignación simple: y toma valor 2
x,y = y,x+2;    //asignación multiple: x toma valor 2 e y toma valor 3 (interpretación paralela)
                //Por interpretación secuencial se entendería, x toma valor de y (sería 2)
                //y luego y toma valor x+2 (sería 4)
```

P no admite la asignación de expresiones sin valor a una variable. En el siguiente ejemplo, se muestra una asignación errónea.

```
VARIABLES
  x:NUM;

INSTRUCCIONES
   x=x+1 //error x+1 no tiene valor porque x no tiene valor
```

La instrucción *condicional* en P define un bloque de instrucciones cuya ejecución depende del valor de verdad de una condición.

Por ejemplo,

```
si (s[i]>max) entonces  //condicion
   max=s[i];            //bloque
fsi
```

La instrucción condicional puede incluir una alternativa `sino`. Por ejemplo,

```
si (s[i]>max) entonces  //condicion
   max=s[i];            //bloque si se cumple la condicion
sino
   max=0;               //bloque si no se cumple la condicion
fsi
```

Las *condiciones* en P se construyen en base a igualdades (==) y desigualdades (`!=`, `<`, `>`, `<=`, `>=`) de expresiones del mismo tipo. Las condiciones se pueden concatenar con operadores conjunción (`&&`) y disyunción (`||`). También se puede usar la negación (`!`).
Estos operadores presentan una prioridad por defecto: (a) `!` es más prioritario que `&&` y `||` (b) `&&` y `||` tiene la misma prioridad. El uso de paréntesis permitirá romper la prioridad por defecto de estos operadores.
P incluye dos condiciones especiales: `cierto` que siempre evalúa a verdadero y `falso` que siempre evalúa a falso.

La evaluación de una condición puede tomar tres posibles valores: *verdadero, falso e indefinido*. En dicha evaluación, se propaga la indefinición, es decir, toda igualdad y desigualdad en una condición debe tener un valor de verdad definido (verdadero o falso) sino la condición evaluará a indefinido. Un ejemplo típico de condición con evaluación indefinida es la que contiene alguna variable sin valor.

La *iteración* en P define un bloque de instrucciones con una condición al principio. La ejecución del bloque se repite hasta que la condición sea falsa.
Por ejemplo,

```
i=0;
max=s[i];
mientras (i<=ultima_posicion(s)) hacer
   si (s[i]>max) entonces
      max=s[i];
   fsi
   i=i+1;
fmientras
```

A la iteración se le puede asociar una *función de avance*. La función de avance tiene por objeto testar que la iteración avanza hacia su terminación. La función de avance tiene como parámetros de entrada las variables de la condición y devuelve un número mayor o igual que cero. Por ejemplo, la función de avance para la iteración anterior:

```
FUNCION bucle_1_avance(SEQ(NUM) seq, NUM pos) dev (NUM r)
```

```
      VARIABLES
      INSTRUCCIONES
         r=ultima_posicion(s)-i;
         dev r;
    FFUNCION
```

La vinculación de una función de avance a una iteración se hace de la siguiente forma en P:

```
   i=0;
   max=s[i];
   mientras (i<=ultima_posicion(s)) hacer {avance: bucle_1_avance(s, i)}
      si (s[i]>max) entonces
         max=s[i];
      fsi
      i=i+1;
   fmientras
```

Una iteración es errónea si los valores devueltos por su función de avance no decrecen en cada iteración sucesiva. Por ejemplo, la función `bucle_1_avance` devuelve la siguiente secuencia de valores para la iteración asociada: 6,5,4,3,2,1,0 siendo `s = [3,4,2,7,9,0,11].`

P dispone de una instrucción de *ruptura de control* llamada `ruptura`. La ejecución de `ruptura` lleva al programa fuera del bloque en el que está localizada la ruptura. En el siguiente código de ejemplo, la ejecución de la ruptura lleva el control del programa fuera del bloque de instrucciones asociado a la iteración:

```
i=0;
mientras (i<=9) hacer
   x=3;
   ruptura;
   i=i+1;
fmientras
```

La declaración de subprogramas en P permite hacer *llamadas* a los mismos en la zona de instrucciones. En el siguiente ejemplo se muestra una llamada a función:

```
x,y = mayor(s); //llamada a funcion
```

La llamada a una función puede ser parte de una expresión si la declaración de la función devuelve un único parámetro. La llamada a un procedimiento nunca puede formar parte de una expresión.

La llamada a un subprograma es correcta si la secuencia de expresiones en los parámetros de la llamada coincide en número y tipo con la correspondiente secuencia de parámetros en la declaración. Por ejemplo, la siguiente llamada sería incorrecta:

```
x,y = mayor(1); //llamada a función con un parámetro tipo NUM. Se requiere un parámetro tipo SEQ
                //en su declaracióm
```

P dispone de un procedimiento predefinido llamado `mostrar` para mostrar por consola los valores de una lista de variables. Por ejemplo, `mostrar(s)` mostraría por pantalla `s = [3,4,2,7,9,0,11]`

Un *aserto* es una expresión lógica entre llaves. El aserto se diseña para que su evaluación sea verdadera o falsa. Un aserto se dice que está mal diseñado si su evaluación es indefinida. El aserto se localiza en algún punto de las instrucciones del programa o subprograma. *Un programa es incorrecto si alguno de sus asertos es falso en alguna de sus ejecuciones*. A continuación, se muestra un programa de ejemplo con asertos.

```
PROGRAMA

VARIABLES
   i,max,min:NUM;
   s:SEQ(NUM);

SUBPROGRAMAS


INSTRUCCIONES
{  cierto   }
   s=[3,4,2,7,9,0,11];
```

```
   i=0;
   mostrar(i);
   min=s[i];
   max=s[i];
   mientras (i<=ultima_posicion(s)) hacer
      si (s[i]>max) entonces
         max=s[i];
      fsi
      si (s[i]<min) entonces
         min=s[i];
      fsi
      mostrar(min,max);
      { PARATODO(p:[0,i],s[p]<=max && s[p]>=min) }
      i=i+1;
   fmientras
{ PARATODO(p:[0,ultima_posicion(s)],s[p]<=max && s[p]>=min) }
```

Desde un punto de vista sintáctico, los asertos son condiciones extendidas con dos tipos de posibles cuantificaciones:
(a) cuantificación universal, ej. `PARATODO(p:[0,ultima_posicion(s)],s[i]<=max)` y
(b) cuantificación existencial ej. `EXISTE(x:[0,ultima_posicion(s)],s[x]>10)`.
Toda cuantificación está compuesta de una variable cuantificada en un rango finito de valores enteros ej, `x:[0,ultima_posicion(s)]` y una condición, ej. `s[i]<=max && s[i]>=min`.
Las cuantificaciones no se anidan.

SE PIDE:

Especificación e Implementación de un procesador para P compuesto de 4 partes:

1.Analizador Léxico/Sintáctico

2.Analizador Semántico

3.Intérprete

4.Compilador (o Traductor) a lenguaje Java.

Llamaremos Front-end del procesador a las dos primeras partes (Analizador Léxico/Sintáctico, Analizador Semántico) y Back-end a las dos últimas (Intérprete, Compilador).

Analizador Léxico/Sintáctico

[ESPECIFICACIÓN 1] Gramática independiente de tecnología que describa la sintaxis de P.

[IMPLEMENTACIÓN 1] Lexer y Parser Antlr4 para P basado en ESPECIFICACIÓN 1.

Analizador Semántico

[ESPECIFICACIÓN 2] Especificación del analizador semántico que detecte un conjunto de errores semánticos propuestos por el alumno. La relevancia de los errores propuestos será un aspecto clave. El formato de la especificación debe ser:

```
//OBJETIVO: especificación abstracta y declarativa de los errores semánticos
// propuestos:
//          error 1,
//          …,
//          error k
//DECISIONES DE DISEÑO error 1:
// (DECISIÓN 1.1) …
// (DECISIÓN 1.2) …
//  …
```

```
//DECISIONES DE DISEÑO error k:
// (DECISIÓN k.1) …
// (DECISIÓN k.2) …

//GRAMÁTICA ATRIBUIDA: …
```

[IMPLEMENTACIÓN 2] Implementación Antlr4 de la gramática atribuida propuesta en ESPECIFICACIÓN 2.

Intérprete

[ESPECIFICACIÓN 3] Especificación de un intérprete para P.

El formato de la especificación debe ser:

```
//OBJETIVO: especificación abstracta y declarativa de la semántica de P en base a
//          la especificación de cada sección de P (enunciado propuesto para P
//          puede servir de referencia).
//DECISIONES DE DISEÑO: explicar cómo se realizarán las interpretaciones de las
//                      diferentes elementos de un programa.
// (DECISIÓN 1) …
// (DECISIÓN 2) …
//  …
//GRAMÁTICA ATRIBUIDA: …
```

[IMPLEMENTACIÓN 3] Implementación Antlr4 de la gramática atribuida propuesta en ESPECIFICACIÓN 3.

Compilador

[ESPECIFICACIÓN 4] Especificación de un compilador que traduzca programa P a Java.

El formato de la especificación debe ser:

```
//OBJETIVO: especificación abstracta y declarativa de la semántica de P en base a
//          la especificación de cada sección de P (enunciado propuesto para P
//          puede servir de referencia).
//EJEMPLOS DE COMPILACIONES: proponer un conjunto de ejemplos de traducción. Es
//          que dicho conjunto sea válido y razonablemente completo.
//      ejemplo 1
//      …
//      ejemplo n
//DECISIONES DE DISEÑO: explicar cómo se realizarán las traducciones de los diferentes
//                      elementos del programa.
// (DECISIÓN 1) …
// (DECISIÓN 2) …
//  …
//GRAMÁTICA ATRIBUIDA: …
```

[IMPLEMENTACIÓN 4] Implementación Antlr4 de la gramática atribuida propuesta en ESPECIFICACIÓN 4.

EVALUACIÓN:

Se consideran 3 niveles de expresividad en P:

Nivel1 (expresividad mínima): P sin tipo `LOG`, sin subprogramas, sin asertos y sin función de avance.

Nivel 2 (expresividad media): P sin asertos y sin función de avance.

Nivel 3 (expresividad máxima): P completo.

A cada nivel de expresividad se le asocia una valoración máxima:

Nivel 1 (expresividad mínima): Hasta un 6.

Nivel 2 (expresividad media): Hasta un 8.

Nivel 3 (expresividad máxima): Hasta un 10.

La evaluación será individual (con preguntas en un examen para verificar autoría).

El procesador consta de 2 partes principales:

Front-end (Analizador Léxico/Sintáctico y Analizador Semántico). Debe incluir al menos un Analizador Léxico/ Sintáctico.

Back-end (Intérprete y Compilador): Debe incluir al menos un Intérprete o un Compilador.

Ejemplos de procesadores mínimos viables serían:

- Analizador Léxico/Sintáctico + Intérprete
- Analizador Léxico/Sintáctico + Compilador