E79 Procesadores de lenguaje

Examen de teoría (10 de septiembre de 2002)

PREGUNTA 1 (6 PUNTOS)

A continuación, se presentan tres posibles extensiones del lenguaje 2KS. Elige dos de ellas y explica claramente qué modificaciones se tendrían que hacer en un compilador de 2KS a Stan para que las aceptara. Las modificaciones son independientes entre sí; no hace falta que consideres sus posibles interacciones.

En tu descripción de las modificaciones, procura ser claro, escueto y preciso. En particular, no es necesario que describas partes del compilador que no estén afectadas por las modificaciones. Puedes optar por descripciones algorítmicas o en lenguaje natural para lograr una mayor sencillez en la explicación. También puede facilitarte la exposición una estructura que siga las distintas etapas del compilador.

Explicita cualquier asunción que hagas acerca del compilador o del enunciado propuesto.

Operador condicional

Esta extensión define el nuevo operador "`? :`". Una expresión con el operador condicional tiene la estructura

exp1 `?` *exp2* `:` *exp3*

La evaluación de la expresión se realiza de la siguiente manera: en primer lugar se evalua *exp1* (que debe ser de tipo entero). Si resultara ser cierta (según las reglas de 2KS) se devuelve el resultado de evaluar *exp2*, en caso contrario, se devuelve el resultado de evaluar *exp3*. Se garantiza que sólo se evalúa *exp2* o *exp3*, pero no ambos. Por lo tanto, el siguiente código nunca provoca un error de ejecución:

```
eixida <- r=0 ? 0 : 1/r;
```

El nuevo operador define un nuevo nivel de prioridad por debajo del de la suma, resta y disyunción. El tipo del operador es el más general de los tipos de *exp2* y *exp3*.

Notas: el operador se puede anidar y es asociativo por la izquierda. Así `1?2?3:4:5` es equivalente a `1?(2?3:4):5` y la expresión `1?2:3?4:5` lo es a `(1?2:3)?4:5`.

Funciones anónimas

Esta extensión permite escribir funciones que se utilizan inmediatamente sin necesidad de declararlas. Estas funciones consisten únicamente en una expresión que se evalúa sobre los parámetros que se le pasan y devuelve el correspondiente resultado. La sintaxis es similar a la de una llamada a función convencional con la excepción de que en lugar del nombre de la función aparece la secuencia `[[` *lista parámetros* `:` *expresión* `]]`. La lista de parámetros tiene la misma sintaxis que las listas de parámetros de 2KS. La expresión puede ser cualquier expresión válida en 2KS con la salvedad de que no pueden utilizarse variables en ella, únicamente son válidos los parámetros declarados al escribir la función.

Por ejemplo, la sentencia

```
eixida <- [[ real r1, real r2: r1*r1+r2*r2]] (a+b, 2*c);
```

escribiría el valor de $(a+b)^2+(2c)^2$, pero evaluando $a+b$ y $2c$ *una sola vez*.

El tipo del resultado es el tipo de la expresión.

Inicialización de vectores mediante funciones

Esta extensión permite inicializar cómodamente los vectores en el momento de su declaración. Para ello, en la declaración del vector se puede seguir el identificador con la secuencia `<- [`*id*`]`, donde *id* es el identificador de una función que recibe un parámetro de tipo entero y devuelve un valor de tipo compatible con el tipo base del vector. En el momento de inicializar el vector se llama a esa función con los valores desde el límite

inferior al superior, inclusive y se almacena el resultado de la llamada en la posición correspondiente del vector. Por ejemplo, la declaración

```
vector enter [1..5] cuadrados <- [cuadrado];
```

almacenaría en el vector `cuadrados` los cuadrados de los cinco primeros números naturales (asumiendo una definición adecuada de la función `cuadrado`).

Nota: no se asume nada acerca del orden de las inicializaciones.

Pregunta 2 (2 puntos)

Supongamos que en un compilador de 2KΣ decides modificar el módulo que controla el flujo de entrada como se explica a continuación. Para aumentar la eficiencia global eliminando los comentarios, ese módulo, al recibir una petición para leer un carácter, comprueba si éste es una barra (/). Si lo es, mira también el siguiente carácter y, en caso de que sea una nueva barra, lee caracteres hasta encontrar un fin de línea. En ese momento lee un carácter más, que es el que devuelve. Es decir, ante la entrada

```
i<- 3; // Inicialización de i
a<- 4; // Inicialización de a
eixida <- i+a;
```

los caracteres que vería el analizador léxico serían:

¿Cuál de las siguientes posibilidades es cierta?:

(a) El compilador resultante es incorrecto; acepta programas no válidos o rechaza programas válidos.

(b) El compilador es correcto pero el rendimiento no se ve afectado.

(c) El compilador es correcto pero el rendimiento baja.

(d) El compilador es correcto y el rendimiento es superior.

Justifica tu respuesta y **explicita cualquier asunción acerca del compilador (o de metacomp) que hagas**.

Pregunta 3 (2 puntos)

Sea la gramática G_n con las siguientes reglas:

$$\begin{aligned}
\langle \mathsf{S} \rangle &\rightarrow \langle \mathsf{A}_1 \rangle \\
\langle \mathsf{A}_1 \rangle &\rightarrow \langle \mathsf{A}_2 \rangle \langle \mathsf{A}_2 \rangle \\
\langle \mathsf{A}_2 \rangle &\rightarrow \langle \mathsf{A}_3 \rangle \langle \mathsf{A}_3 \rangle \langle \mathsf{A}_3 \rangle \\
&\cdots \\
\langle \mathsf{A}_{n-1} \rangle &\rightarrow \overbrace{\langle \mathsf{A}_n \rangle \cdots \langle \mathsf{A}_n \rangle}^{n} \\
\langle \mathsf{A}_n \rangle &\rightarrow \mathsf{a}
\end{aligned}$$

Se pide:

(a) Número de estados del automata de prefijos viables de la gramática G_n.

(b) Decidir si la gramática G_n es o no SLR.

(c) Lenguaje generado por G_n.

Justifica las respuestas.