4º Ingeniería Informática
II26 Procesadores de lenguaje

Examen (12 de septiembre de 2008)

Pregunta 1 (5 puntos)

Explica cómo debería modificarse la versión 7 de `minicomp` (la que incluye las modificaciones hasta "promociones implícitas de entero a real", inclusive) de modo que aceptara las extensiones que se presentan a continuación. Las extensiones son independientes entre sí; no hace falta que consideres sus posibles interacciones.

Procura que tu descripción sea clara, escueta y precisa. Para ello, puede facilitarte la exposición una estructura que siga las distintas etapas del compilador.

Puedes optar por descripciones algorítmicas o en lenguaje natural para lograr una mayor sencillez en la explicación, pero dejando claro qué se modificaría y cómo. Como mínimo: las descripciones de modificaciones del nivel léxico deben explicitar, si las hay, las nuevas expresiones regulares y los atributos asociados a posibles nuevas categorías; las de modificaciones del nivel sintáctico o semántico deben aclarar las reglas, acciones semánticas y/o métodos de comprobación afectados; las de modificaciones de la generación de código deben dejar claro qué instrucciones de máquina virtual se emitirían con la modificación. Por ejemplo: sería aceptable una frase como "se comprobaría que el hijo izquierdo es de tipo entero" (es trivial transformar la comprobación en un test), pero no una como "se generaría código para sumar los dos números" (no sabemos si se generaría una instrucción o varias, cuál o cuáles serían, dónde están los números —¿en registros, en memoria, en la pila?—, etc.).

Explicita cualquier asunción que hagas acerca del enunciado propuesto.

E1. Compilación condicional (2 puntos)

Mediante esta extensión, se pueden compilar condicionalmente distintas partes del código. Para ello se introducen las directivas de compilación `#nivel`, `#si` y `#fin_si`. La primera marca un *nivel* de compilación, un entero cuya interpretación se deja al programador (por ejemplo, nivel de depuración, de optimización, de verbosidad) y sigue la sintaxis `#nivel` ***num***, donde ***num*** es un literal entero. Las directivas `#si` y `#fin_si` rodean un grupo de sentencias y permiten controlar si su código aparecerá en el ejecutable final. Por ejemplo, podemos controlar la verbosidad de un mensaje de depuración de la siguiente forma:

```
#nivel 1
...
#si #nivel > 0
  escribe "Estoy en f"; nl;
  #si #nivel >= 2
    escribe "Parámetros:"; nl:
    escribe "x:"; escribe x; nl;
    escribe "y:"; escribe y; nl;
  #fin_si
#fin_si
```

Aunque las directivas no son sentencias, sólo pueden aparecer donde sea correcto encontrar una sentencia. Como ves en el ejemplo, las estructuras `#si`–`#fin_si` pueden anidarse. Por simplicidad, los cambios de `#nivel` tienen efecto desde su aparición hasta la siguiente directiva de nivel, sin tener en cuenta los ámbitos de las funciones o si están dentro de una estructura `#si`–`#fin_si`. Antes de la primera directiva `#nivel`, su valor es cero.

La sintaxis de `#si`–`#fin_si` es la siguiente:

```
#si #nivel comp num
  Lista de sentencias
#fin_si
```

donde *comp* es un operador de comparación y *num* un literal entero.

Puedes elegir si se efectuarán o no las comprobaciones semánticas de aquellas sentencias contenidas en un bloque `#si`–`#fin_si` cuya condición sea falsa.

E2. Llamadas a función con vectores (3 puntos)

Esta extensión permite llamar a una función pasándole los parámetros mediante un vector. Para ello, se emplea la notación `llama` *f* `[`*v*`]` donde: *f* es una función cuyos parámetros tienen todos el mismo tipo elemental; *v* es un vector con tipo base igual al de los parámetros de *f* (no se aplica ninguna promoción implícita) y con talla igual al número de parámetros de *f*.

Por ejemplo, si tenemos la función `suma(x,y,z: real): real` y los vectores `lista: vector[3] de real` y `matriz: vector[4] de vector[3] de real`, el resultado de

```
i= llama suma(lista[0], lista[1], lista[2]);
j= llama suma(matriz[1][0], matriz[1][1], matriz[1][2]);
```

es el mismo que el de

```
i= llama suma[lista];
j= llama suma[matriz[1]];
```

Sólo cambia el número de instrucciones generadas; en una llamada con vectores es independiente del tamaño del vector.

Por comodidad, puedes simplificar tu respuesta si, en la parte de generación de código, no incluyes aquello que no cambia respecto a las llamadas normales.

Pregunta 2 (2 puntos)

Escribe tanto una expresión regular como un AFD para los comentarios de un lenguaje de programación tales que: comienzan por una cadena del lenguaje definido por la expresión regular `<`–$^{+}$`\|`; terminan con la primera aparición de una cadena del lenguaje definido por la expresión regular `\|`–$^{+}$`>`; y no es necesario que coincidan las longitudes de la cadenas de inicio y fin del comentario.

Esto son ejemplos de comentarios válidos:

```
<--||-->
<---| calculamos a-b |-->
<----| ejecutar si |x-y|>6 |->
<--||-|->
```

Y estos no son válidos:

```
<---|-->
<---| a|----->b |->
```

Pregunta 3 (1,5 puntos)

Obtén una gramática incontextual, sin partes derechas regulares, que sea LL(1) y que genere el mismo lenguaje que la siguiente:

$$\langle \mathsf{S} \rangle \quad \rightarrow \quad (\mathbf{a}|\mathbf{c}|\mathbf{ba}|\mathbf{bc})^{*}\mathbf{b}?$$

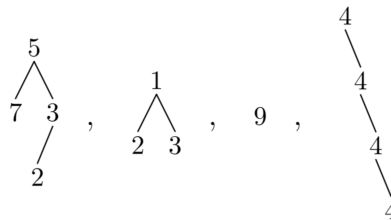
siendo $\{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ el conjunto de terminales. Demuestra que tu gramática es LL(1).

Pregunta 4 (1,5 puntos)

Considera la siguiente gramática, que modela un bosque formado por una secuencia de árboles binarios separados por comas:

$$\begin{aligned}
\langle \mathsf{Bosque} \rangle &\rightarrow \langle \mathsf{Arbin} \rangle \langle \mathsf{MásArbin} \rangle \\
\langle \mathsf{MásArbin} \rangle &\rightarrow \textbf{coma } \langle \mathsf{Arbin} \rangle \langle \mathsf{MásArbin} \rangle | \lambda \\
\langle \mathsf{Arbin} \rangle &\rightarrow \textbf{abreParéntesis } (\textbf{entero } (\langle \mathsf{Arbin} \rangle \langle \mathsf{Arbin} \rangle)?)? \textbf{ cierraParéntesis}
\end{aligned}$$

Por ejemplo, el bosque



se puede representar mediante la cadena

```
(5(7()())(3(2)())), (1(2)(3)), (9), (4()(4()(4()(4))))
```

© Universitat Jaume I 2007-2008

Añade acciones semánticas a la gramática anterior, sin modificarla, para que el esquema de traducción resultante escriba, mientras se lleva a cabo un análisis RLL(1) de la entrada, la profundidad de aquellos árboles del bosque que superen la profundidad de todos los anteriores.

Sólo puedes utilizar atributos de tipo lógico, entero o cadena (y no, por ejemplo, de tipo lista). Además, no puedes utilizar ningún objeto global ni tampoco puedes utilizar como heredados atributos sintetizados y viceversa.

Indica para cada atributo si es heredado o sintetizado y qué representa.

Nota: la profundidad de un árbol binario es la cantidad de nodos en el camino más largo desde la raíz hasta una hoja. En el ejemplo anterior, las profundidades son 3, 2, 1 y 4, respectivamente. Por lo tanto, la salida es `3 4`, correspondiente a las profundidades de los árboles primero y último.