UNIVERSITAT JAUME·I

Ingeniería Informática

Procesadores de lenguaje

Examen de teoría (12 de diciembre de 2006)

PREGUNTA 1 (5 PUNTOS)

Explica cómo debería modificarse `minicomp` (en su versión para Stan o para Rossi, pero sin ninguna de las extensiones planteadas en la página de la asignatura) 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.

En tu descripción, procura ser claro, escueto y preciso. 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 enunciado propuesto.

Comparación de cadenas (3 puntos)

El objetivo de esta modificación es permitir la utilización de los operadores de comparación (`<`, `>`, `<=`, `>=`, `=`, `!=`) con cadenas. De esta manera podemos, por ejemplo, escribir:

```
if c[1] < "hola" entonces
  escribe("menor");
si_no
  escribe("mayor o igual");
fin
```

El orden en las cadenas es el orden alfabético. Sean u y v dos cadenas. Hay tres posibilidades:

- Son iguales.
- Una es prefijo de la otra, por lo que es la menor.
- Si la primera posición en la que difieren u y v es i, será $u < v$ si $u_i < v_i$ y $u > v$ si $u_i > v_i$.

Nota: no se asume que cadenas iguales tengan la misma dirección.

Además de la descripción de la modificación, escribe el código que se generaría para el ejemplo. Asigna a `c` y a las cadenas las direcciones que creas oportunas.

Comparaciones encadenadas (2 puntos)

Esta modificación permite encadenar comparaciones de manera similar a Python. En concreto, permite expresiones de la forma e_1 **opcomp**$_1$ $e_2 \ldots e_n$**opcomp**$_n$ e_{n+1}, donde e_1, e_2,... e_{n+1} son expresiones de tipo entero y los operadores de comparación no tienen por qué ser iguales entre sí. La expresión sería equivalente a e_1 **opcomp**$_1$ e_2 **y** e_2 **opcomp**$_2$ e_3 **y** ... **y** e_n**opcomp**$_n$ e_{n+1} si existiera en `minicomp` el operador y-lógico, se evaluara por circuito corto y *no se evaluara ninguna expresión más de una vez.*

Por ejemplo, el fragmento

```
si f(1) < f(2) < f(3) entonces
  escribe("creciente");
fin
```

escribirá "`creciente`" por pantalla si $f(1) < f(2)$ y $f(2) < f(3)$. Además, llamará a $f(1)$ y $f(2)$ exactamente una vez. Sólo si $f(1) < f(2)$, llamará a $f(3)$.

Además de las modificaciones, escribe el código que generarías para el ejemplo. Dale a `c` y `f` la dirección y etiqueta que consideres oportunas.

PREGUNTA 2 (1,5 PUNTOS)

Sea r_n la expresión regular $(\mathtt{a}|\mathtt{ba}|\mathtt{baa}|\ldots|\overbrace{\mathtt{ba}\ldots\mathtt{a}}^{n})^*$ Escribe el autómata finito determinista obtenido para r_3 mediante el algoritmo de construcción usando ítems. Para un n cualquiera mayor que cero, ¿cuántos estados tiene el autómata finito determinista obtenido para r_n mediante el algoritmo de construcción usando ítems?

Pregunta 3 (1,5 puntos)

Queremos reconocer cadenas de paréntesis y corchetes de modo que las subcadenas obtenidas considerando sólo los paréntesis o sólo los corchetes estén bien parentizadas, sin importar las relaciones entre paréntesis y corchetes. Así, la cadena **(([)]()[)]** es correcta porque tanto **(()())** como **[][]** están bien parentizadas. Sin embargo, no lo es la cadena **(()])** porque **]** no está bien parentizada.

Modela estas cadenas de una de las dos formas siguientes:

- Mediante una GPDR.
- Mediante un esquema de traducción de modo que el atributo sintetizado *bien* del símbolo inicial sea cierto si la cadena generada está bien parentizada y falso en caso contrario. Si optas por esta vía, sólo puedes emplear atributos sintetizados de tipo lógico, carácter o entero y las acciones deben estar al final de las reglas.

En ambos casos, no debes preocuparte de si la gramática resultante es o no de las familias LL, RLL o LR.

Pregunta 4 (2 puntos)

Sean $G_1 = (N_1, \Sigma, P_1, \langle \mathsf{S}_1 \rangle)$ y $G_2 = (N_2, \Sigma, P_2, \langle \mathsf{S}_2 \rangle)$ dos gramáticas con $N_1 \cap N_2 = \emptyset$. Definimos las gramáticas unión, G_U, y concatenación, G_C, de la siguiente manera:

- $G_U = \big(N_1 \cup N_2 \cup \{\langle \mathsf{S}_U \rangle\}, \Sigma, P_1 \cup P_2 \cup \{\langle \mathsf{S}_U \rangle \rightarrow \langle \mathsf{S}_1 \rangle | \langle \mathsf{S}_2 \rangle\}, \langle \mathsf{S}_U \rangle\big)$.
- $G_C = \big(N_1 \cup N_2 \cup \{\langle \mathsf{S}_C \rangle\}, \Sigma, P_1 \cup P_2 \cup \{\langle \mathsf{S}_C \rangle \rightarrow \langle \mathsf{S}_1 \rangle \langle \mathsf{S}_2 \rangle\}, \langle \mathsf{S}_C \rangle\big)$.

Donde $\langle \mathsf{S}_C \rangle$ y $\langle \mathsf{S}_U \rangle$ son dos nuevos terminales no incluidos en N_1 ni N_2.

Supón que $L(G_1) \cap L(G_2) = \emptyset$. Para cada una de las afirmaciones siguientes, da un contraejemplo si es falsa o justifica su veracidad.

- Si G_1 y G_2 son LL(1), también lo es G_U.
- Si G_1 y G_2 son LL(1), también lo es G_C.
- Si G_1 y G_2 son SLR, también lo es G_U.
- Si G_1 y G_2 son SLR, también lo es G_C.

Duración del examen: 4 horas

¡Buena suerte!