Compiladors: Examen de teoria

19 de juny de 2012

Duració de l'examen: 2.5 hores

1 Identificadors amb subíndexos (2.5 punts)

Nota: Aquest exercici està inspirat en un de semblant del llibre de Aho, Lam, Sethi i Ullman.

Volem fer un analitzador d'identificadors amb subíndexos per a un llenguatge que s'utilitza per a la descripció d'equacions. Aquí hi ha alguns exemples del llenguatge d'identificadors amb subíndexos.

Entrada	Visualització
`lambda`	*lambda*
`a[i]`	a_i
`Matriu[arg[i]]`	$Matriu_{arg_i}$
`a[i][j[min]]`	$a_{i,j_{min}}$
`Vector[i[min]]`	$Vector_{i_{min}}$

Entrada	Visualització
`Vector[i[13][min]]`	$Vector_{i_{13,min}}$
`C[1][2][arg[3]]`	$C_{1,2,arg_3}$
`M[fab[3]][2]`	$M_{fab_3,2}$
`Vector[1[i]]`	`String il·legal`

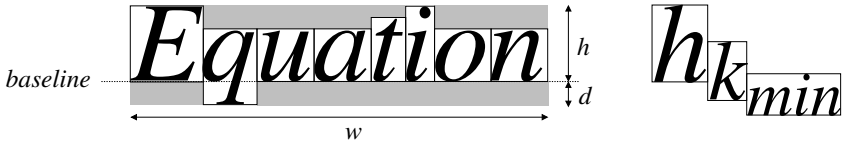
En aquest llenguatge, els identificadors només poden contenir lletres. Els subíndexos poden ser identificadors o números enters sense signe. Els subíndexos que siguin identificadors poden tenir més subíndexos. En canvi, els números no poden tenir més subíndexos (veure el darrer example).

- Dissenyar una gramàtica EBNF que accepti un identificador amb subíndexos. L'alfabet d'entrada és $\Sigma = \{\texttt{a}\ldots\texttt{z}, \texttt{A}\ldots\texttt{Z}, \texttt{0}\ldots\texttt{9}, \texttt{[}, \texttt{]}\}$.

Resposta:

$$
\begin{array}{rcl}
\texttt{S} & \rightarrow & \texttt{Id Subindex}^* \\
\texttt{Subindex} & \rightarrow & \texttt{'[' S} \mid \texttt{Num ']'} \\
\texttt{Id} & \rightarrow & (\texttt{'a'..'z'} \mid \texttt{'A'..'Z'})^+ \\
\texttt{Num} & \rightarrow & (\texttt{'0'..'9'})^+
\end{array}
$$

Volem calcular els atributs de la *bounding box* d'un string del llenguatge. Donat un tamany de font (point size (p): 8, 10, 12, ...), el tamany de cada string es caracteritza amb tres atributs: *height* (h), *depth* (d) i *width* (w). Els atributs h i d representen l'alçada i la profunditat del string respecte a una línia base on descansa el text (veure la figura). Els valors w, h i d sempre són positius.



Cada vegada que especifiquem un subíndex, el tamany de font (p) és reduït un 30% i la línia base del subíndex es desplaça $0.25p$ cap avall, on p és el tamany de font del pare. El tamany de font pot tenir qualsevol valor real (per exemple, 11.83).

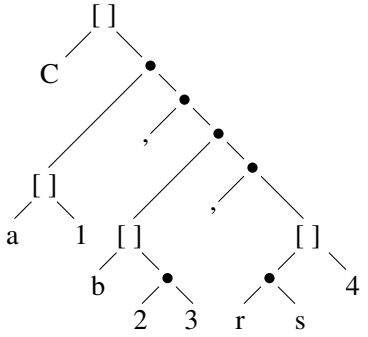
Suposem que la gramàtica genera un AST amb tres tipus de nodes: s (símbol), $\bullet$ (concatenació) i `[]` (subíndex), on s són els nodes que hi ha a les fulles. Cada node N de l'AST conté tres atributs, $N.h$, $N.d$ i $N.w$, que representen les dimensions del string associat al node. Cada node N de tipus s conté un atribut $N.val$ que enmagatzema el símbol que representa (per exemple, `'a'`, `'3'`, `','`, ...). Donat un tamany de font p i un símbol s , podem obtenir el valor dels seus atributs amb les funcions *Getw(p,s)*, *Geth(p,s)* i *Getd(p,s)*.

A continuació es mostra un exemple de l'AST que s'obtindria a partir d'una entrada. Cal observar que l'estructura de l'AST no correspon a l'estructura sintàctica de l'entrada.

Entrada: `C[a[1]][b[23]][rs[4]]`

Visualització: C_{a_1,b_{23},rs_4}

AST $\Longrightarrow$



- Dissenyar la funció `BBox`(p, N) que visita l'AST i defineix els atributs $N.h$, $N.d$ i $N.w$ de cada node N per a un tamany de font p donat. Considerar les tres possibles versions de la funció:

`BBox`$(p, N \to s)$	$\{s$ és un símbol$\}$
`BBox`$(p, N \to B_1 \bullet B_2)$	{concatenació de strings}
`BBox`$(p, N \to B_1[B_2])$	{subíndex}

Resposta:

```
BBox(p, N->s) {
  N.w = Getw(p, N.val);
  N.h = Geth(p, N.val);
  N.d = Getd(p, N.val);
}
```

```
BBox(p, N->B1*B2) {
  BBox(p, B1); BBox(p, B2);
  N.w = B1.w + B2.w;
  N.h = max(B1.h, B2.h);
  N.d = max(B1.d, B2.d);
}
```

```
BBox(p, N->B1[B2]) {
  BBox(p, B1); BBox(0.7*p, B2);
  N.w = B1.w + B2.w;
  N.h = max(B1.h, B2.h - 0.25p);
  N.d = max(B1.d, B2.d + 0.25p);
}
```

2 Anàlisi sintàctica (2.5 punts)

Donada la següent gramàtica:

$$
\begin{array}{lllll}
(1) & S & \to & T\$ \\
(2) & T & \to & R \\
(3) & T & \to & aTc \\
(4) & R & \to & \\
(5) & R & \to & bR
\end{array}
$$

- Descriure el llenguatge que genera.

  Resposta: El llenguatge generat és $a^n b^* c^n$, per a qualsevol $n \geq 0$.

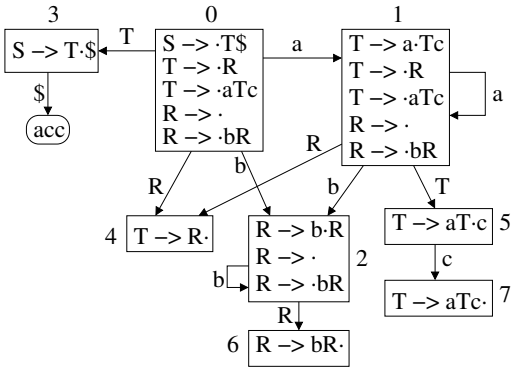
- Calcular *First* i *Follow* per a tots els símbols no terminals.

  Resposta:

	First	*Follow*
S	$\{a, b, \$\}$	$\{\}$
T	$\{a, b\}$	$\{c, \$\}$
R	$\{b\}$	$\{c, \$\}$

- Generar l'autòmat i la taula LR(1).

  Resposta:



	a	b	c	$	T	R
0	s1	s2	r4	r4	3	4
1	s1	s2	r4	r4	5	4
2		s2	r4	r4		6
3				acc		
4			r2	r2		
5			s7			
6			r5	r5		
7			r3	r3		

3 Generació de codi (2.5 punts)

Considerem un llenguatge de programació amb les dues instruccions següents:

```
continue
break n when B
```

La semàntica de `continue` és idèntica a la de la mateixa instrucció en C/C++. La instrucció `break` surt de l'execució dels n bucles més imbricats dins dels que és troba la instrucció quan es compleix l'expressió Booleana B. La constant n és coneguda en temps de compilació.

Aquest és un exemple de programa que fa servir la instrucció `break` per cercar un element en una matriu:

```
i = 0;
while i < n do
  j = 0;
  while j < m do
    break 2 when A[i][j] = x;
    j = j + 1
  endwhile;
  i = i + 1
endwhile;
if i < n then write "Element found" endif
```

Es demana:

- Explicar quines estructures de dades caldria afegir al generador de codi per a poder tenir les instruccions `continue` i `break`.

  Resposta:

  Caldria que el generador de codi tingués una pila d'etiquetes de bucles. En particular, cada posició de la pila contindria dues etiquetes: la de principi i la de final de bucle. El cim de la pila contindria les etiquetes del bucle més imbricat. Podem anomenar aquesta pila `LoopStack` i podem suposar que tenim les següents operacions: `Push(L_ini,L_end)`, `Pop()` i `Top(`i`)`. La funció `Top(`i`)` rep un valor $i \geq 1$ que indica la profunditat de la pila a la que volem accedir ($i = 1$ pel cim de la pila). Podem obtenir les etiquetes d'inici i final de bucle amb `Top(`i`).ini` i `Top(`i`).end`.

- Redefinir la següent funció de generació de codi per tal de donar suport a les instruccions de `continue` i `break`:

  `Execute(I →` `while` B `do` S`)`

- Definir les funcions de generació de codi per a `continue` i `break`:

  `Execute(I` $\rightarrow$ `continue)`
  `Execute(I` $\rightarrow$ `break` n `when` B`)`

  En tots els casos, cal fer servir l'avaluació d'expressions Booleanes amb *backpatching*.

  Resposta:

```
Execute(I -> while B do S)}
    L1 = new Label();
    L2 = new Label();
    EvalBoolExpr(B, 0, L2);
    LoopStack.Push (L1, L2);
    Execute(S);
    LoopStack.Pop();
    I.code = "L1:" ||
                B.code ||
                S.code ||
                "goto L1" ||
             "L2:";
```

```
Execute(I -> continue)
    L = LoopStack.Top(1).ini;
    I.code = "goto L"
```

```
Execute(I -> break n when B)
    L = LoopStack.Top(n).end;
    EvalBoolExpr(B, L, 0);
    I.code = B.code;
```

4 Optimització de codi (2.5 punts)

Suposem que el següent codi correspon al cos d'una funció declarada com a

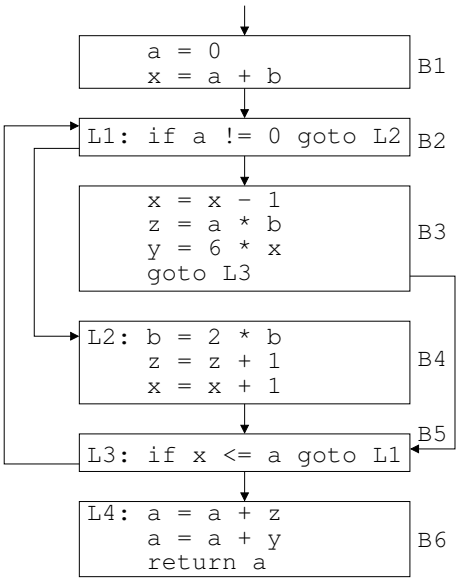
int f (int b);

on b és un paràmetre i la resta de variables són locals i enteres.

```
    a = 0
    x = a + b
L1: if a != 0 goto L2
    x = x - 1
    z = a * b
    y = 6 * x
    goto L3
L2: b = 2 * b
    z = z + 1
    x = x + 1
L3: if x > a goto L1
    a = a + z
    a = a + y
    return a
```

- Dibuixar el graf de blocs bàsics.

  Resposta:



- Quins blocs bàsics *dominen* el bloc bàsic que conté `L3`?

  Resposta: B1 i B2. Opcionalment podem dir que B5 també el domina si considerem que la relació és reflexiva.

- Indicar quines variables estan vives després d'executar les instruccions `x=x-1`, `x=x+1` i `a=a+z`.

  Resposta:

Després de	Variables vives
`x = x - 1`	`a, b, x`
`x = x + 1`	`a, b, x, y, z`
`a = a + z`	`a, y`

- Aplicar iterativament totes les optimitzacions fins que el codi no es pugui reduïr més. Explicar el raonament de cadascuna de les optimizacions i escriure el codi final.

Resposta:

Les possibles optimitzacions són mostrades en els següents fragments de codi:

```
    a = 0
    x = 0 + b
L1: if 0 != 0 goto L2
    x = x - 1
    z = 0 * b
    y = 6 * x
    goto L3
L2: b = 2 * b
    z = z + 1
    x = x + 1
L3: if x > 0 goto L1
    a = 0 + z
    a = a + y
    return a
```

```
    a = 0
    x = b
L1: x = x - 1
    z = 0
    y = 6 * x
    goto L3
L3: if x > 0 goto L1
    a = 0
    a = 0 + y
    return a
```

```
    x = b
L1: x = x - 1
    y = 6 * x
    if x > 0 goto L1
    return y
```

```
    x = b
    y = 6 * x
L1: x = x - 1
    y = y - 6
    if x > 0 goto L1
    return y
```

Inicialment és pot aplicar una propagació de còpies i constants. D'aquesta manera, l'assignació `a=0` és pot propagar a tots aquells usos en els que aquesta sigui l'única definició que hi arriba.

Posteriorment es pot observar que la condició de la instrucció a `L1` és sempre falsa. Això vol dir que el codi que va entre `L2` i `L3` mai serà executat (codi mort) i és pot eliminar.

La propagació de constants i de còpies fa que les assignacions `z=0` i `a=0` siguin a variables mortes i, per tant, es puguin eliminar.

La instrucció `goto L3` també es pot eliminar, donat que `L3` és la següent instrucció.

Finalment ens podem adonar que la variable `y` és d'inducció i la multiplicació es pot treure fora del bucle. Es podria també aplicar una propagació de còpia i transformar l'assignació `y=6*x` en `y=6*b`, tot i que això no milloraria la qualitat del codi.