Compiladors: Examen de teoria
25 de juny de 2015, 8:00

Duració de l'examen: 2.5 hores

Publicació de les notes: 29 de juny
Revisió de notes: 30 de juny (10:00-11:00)

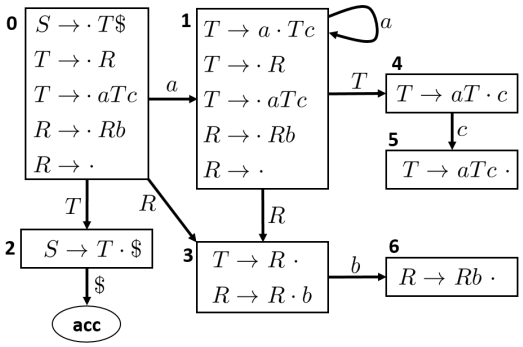
1 Analitzador sintàctic (3 punts)

Donada la següent gramàtica:

(1)	$S \to T\$$
(2)	$T \to R$
(3)	$T \to aTc$
(4)	$R \to Rb$
(5)	$R \to \varepsilon$

1. Calcular `First` i `Follow` pels símbols T i R. Afegir ε a `First` quan el símbol sigui anul.lable.
2. Dissenyar l'autòmat per construir un analitzador SLR(1).
3. Dissenyar la taula SLR(1).

Resposta:



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

	a	b	c	$	R	T
0	s1	r5	r5	r5	3	2
1	s1	r5	r5	r5	3	4
2				acc		
3		s6	r2	r2		
4			s5			
5			r3	r3		
6		r4	r4	r4		

2 Analitzador semàntic (2.5 punts)

Tenim un llenguatge senzill d'expressions on no hi ha ni instruccions condicionals ni iteratives ni crides a funcions. Els únics tipus de dades escalars són els *enters* (`int`) i els strings (`string`). També tenim variables de tipus `pila` que poden emmagatzemar enters o strings, però no elements de diferents tipus a la vegada. En el llenguatge no hi ha declaracions de variables. Les variables es creen quan no existeixen i es fa una assignació o un `push`. Les variables són destruïdes (i poden canviar de tipus) cada vegada que se'ls hi fa una nova assignació. Només es permet destruir variables de tipus pila quan aquestes estan buides. Un programa és una seqüència d'instruccions com l'exemple de l'esquerra. A la dreta es mostren exemples d'errors semàntics.

```
a = 3 + 5;
b.push(a+4);
a = b.top;
b.pop;
b = "Hello ";
c = b + "world";
a.push("Bye")
a.push(c)
```

```
a = 3 + b;                          // error: b no declarada
```

```
b.push("Hello");
b.push(5);          // error: pila amb tipus diferents
```

```
c.push(5);
c = 2 + 3;           // error: pila destruida no buida
```

```
a = "Hello" + 3;    // error: suma amb tipus diferents
```

```
x.push(5)
x.pop
a = x + 3;  // Error: x es una pila (encara que buida)
```

La gramàtica del llenguatge és la següent, on els símbols en majúscules representen tokens emesos per l'analitzador lèxic:

$progr \rightarrow (instr\ \texttt{;})^+$
$instr \rightarrow \texttt{ID = } expr$
$instr \rightarrow \texttt{ID.push(}expr\texttt{)}$
$instr \rightarrow \texttt{ID.pop}$
$expr \rightarrow expr\ \texttt{+}\ expr$
$expr \rightarrow \texttt{ID.top}$
$expr \rightarrow \texttt{ID}$
$expr \rightarrow \texttt{NUMBER}$
$expr \rightarrow \texttt{STRING}$

Suposem que cada regla de la gramàtica va associada a un node de l'AST. Es demana dissenyar un analitzador semàntic que faci les següents comprovacions:

- Les expressions (*expr*) només poden ser de tipus enter o string. L'operador *suma* (+) només pot ser utilitzat amb dades escalars del mateix tipus.
- No es pot llegir una variable que no ha estat prèviament creada.
- No es pot apilar un element d'un tipus en una pila que conté elements d'un altre tipus.
- No es pot llegir (top) o eliminar (pop) un element d'una pila buida.
- No es pot destruir una pila no buida.

Per a fer les comprovacions, suposarem que tenim una taula de símbols global amb dues funcions: `exists(ID)` i `findOrAdd(ID)`. Ambdues reben un identificador com a paràmetre. La funció `exists` retorna un booleà que ens diu si l'identificador existeix a la taula de símbols. La funció `findOrAdd` ens retorna una referència `r` a l'estructura que emmagatzema els atributs de l'identificador a la taula de símbols. Aquests atributs són:

- `r.stack`: booleà que indica si és una pila.
- `r.size`: mida (nombre d'elements) de la pila.
- `r.type`: tipus del símbol (`int` o `string`). En cas de ser una pila, representa el tipus dels seus elements.

En cas que un identificador no existeixi a la taula, la funció `findOrAdd(ID)` crea una entrada `r` amb `r.stack=false`, `r.size=0` i `r.type=int`. En els atributs de la taula de símbols cal mantenir el següent invariant:

$$\neg \texttt{r.stack} \Rightarrow \texttt{r.size} = 0.$$

Podeu suposar que els nodes de l'AST poden tenir atributs per ajudar en la comprovació d'errors. Per a un node n, s'accedeix a l'atribut `attr` mitjançant n`.attr`. Definir els atributs que siguin estrictament necessaris per a la detecció d'errors. La detecció cal fer-la utilitzant assercions. Com a exemple es mostren les accions

semàntiques associades a una de les regles de la gramàtica:

$instr \rightarrow$ `ID.pop`
```
{assert(exists(ID));
 r = findOrAdd(ID);
 assert(r.size > 0);
 r.size--;}
```

Resposta: Suposarem que cada node `n` de l'AST té un atribut anomenat `n.type` que representa el tipus del node.

$instr \rightarrow$ `ID =` $expr$
```
{r = findOrAdd(ID);
 assert (r.size == 0);
 r.stack = false;
 r.type = expr.type;}
```

$instr \rightarrow$ `ID.push(`$expr$`)`
```
{r = findOrAdd(ID);
 assert(r.size == 0 or r.type == expr.type);
 r.stack = true;
 r.type = expr.type;
 r.size++;}
```

$instr \rightarrow$ `ID.pop`
```
{r = findOrAdd(ID);
 assert(r.size > 0);
 r.size--;}
```

$expr \rightarrow expr_1$ `+` $expr_2$
```
{assert(expr_1.type == expr_2.type);
 expr.type = expr_1.type;}
```

$expr \rightarrow$ `ID.top`
```
{r = findOrAdd(ID);
 assert(r.size > 0);
 expr.type = r.type; }
```

$expr \rightarrow$ `ID`
```
{assert(exists(ID));
 r = findOrAdd(ID);
 assert(not r.stack);
 expr.type = r.type; }
```

$expr \rightarrow$ `NUMBER`
```
{ expr.type = int; }
```

$expr \rightarrow$ `STRING`
```
{ expr.type = string; }
```

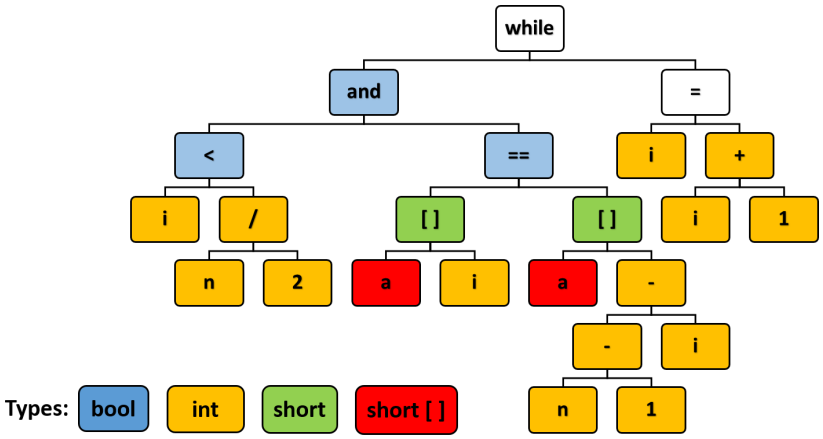
3 Generació i optimització de codi (3 punts)

Donat el següent codi:

```
bool palindrom(short a[], int n) {
  int i = 0;
  while (i < n/2 and a[i] == a[n-1-i]) i = i+1;
  return i == n/2;
}
```

1. Dibuixar l'AST de la instrucció while, indicant el tipus de cada node de l'AST.
2. Escriure el codi de 3 adreces no optimitzat de tota la funció fent servir *backpatching* per les condicions booleanes. Suposeu que un `short` ocupa 2 bytes.
3. Escriure el codi després d'optimitzar.

Resposta:



Codi no optimitzat

```
          t1 = 0
          i = t1
while:    t2 = n/2
          ifFalse i < t2 goto endwhile
          t3 = i*2
          t4 = a[t3]
          t5 = n-1
          t6 = t5-i
          t7 = t6*2
          t8 = a[t7]
          ifFalse t4 == t8 goto endwhile
          t9 = i+1
          i = t9
          goto while
endwhile: t10 = n/2
          t11 = i == t10
          return t11
```

Codi optimitzat

```
          i = 0
          t2 = n/2
          t5 = n-1
while:    ifFalse i < t2 goto endwhile
          t3 = i*2
          t4 = a[t3]
          t6 = t5-i
          t7 = t6*2
          t8 = a[t7]
          ifFalse t4 == t8 goto endwhile
          i = i+1
          goto while
endwhile: t11 = i == t2
          return t11
```

4 Assignació de registres (1.5 punts)

```
1: a = y + c
2: d = a + x
3: b = x + y
4: a = b * d
5: e = x - a
6: y = b + x
7: a = e * b
```

Suposem que el conjunt de variables vives al final del codi de l'esquerra és $\{a, y\}$.

- Indicar quines variables estan vives a cada punt del codi.
- Reescriure el codi utilitzant el mínim nombre de registres possible. Anomeneu `R1`, `R2`, `R3`, `R4`, etc, als registres.

Resposta:

```
   {c, x, y}
1: a = y + c
   {a, x, y}
2: d = a + x
   {d, x, y}
3: b = x + y
   {b, d, x}
4: a = b * d
   {a, b, x}
5: e = x - a
   {b, e, x}
6: y = b + x
   {b, e, y}
7: a = e * b
  {a, y}
```

$\Rightarrow$

El codi es pot reescriure amb només tres registres:

```
   {R1:c, R2:x, R3:y}
1: R1 = R3 + R1
   {R1:a, R2:x, R3:y}
2: R1 = R1 + R2
   {R1:d, R2:x, R3:y}
3: R3 = R2 + R3
   {R1:d, R2:x, R3:b}
4: R1 = R3 * R1
   {R1:a, R2:x, R3:b}
5: R1 = R2 - R1
   {R1:e, R2:x, R3:b}
6: R2 = R3 + R2
   {R1:e, R2:y R3:b}
7: R1 = R1 * R3
  {R1:a, R2:y}
```