CRITÉRIOS DE CORREÇÃO

E-FÓLIO GLOBAL E EXAME

21018 – Compilação

Época normal 2021/22

1. Critério de correção: tokens 1 valor; separação 1 valor (desconto por percentagem de erro).

Tokens

Token ID	**Descrição**
ID	Identificador
INT	Número inteiro
MULT	Símbolo de multiplicação
PLUS	Símbolo de adição
MINUS	Símbolo de subtração
PESQ	Parênteses retos esquerdos
PDIR	Parênteses retos direitos
ATRIB	Símbolo de atribuição
DLM	Símbolo delimitador de instruções
WS	Espaço em branco a ignorar

Análise léxica

Lexema	**Token**
a	ID
[	PESQ
i	ID
*	MULT
10	INT
+	PLUS
j	ID
]	PDIR
	WS
=	ATRIB
	WS
10	INT
*	MULT
10	INT
-	MINUS
i	ID
-	MINUS
j	ID
;	DLM

2. Critérios de correção: 0 errado, 1 certo.

Houve um erro na gramática, que ficou mais simples do que devia para reconhecer esta sequência. Tal como está, reconhece apenas uma derivação de D, pelo que apenas poderia ter i Pad ou d Link, nunca ambos. Assim, a pergunta é anulada, sendo atribuído a todos os estudantes a classificação máxima.

Considerando apenas d Link ; \ a p ar / t amen t o . teríamos:

P => D ; E =>

d Link(ID) ; E =>

d Link ; T =>

d Link ; T t F =>

d Link ; T t F t F =>

d Link ; F t F t F =>

d Link ; \ E / t F t F =>

d Link ; \ E p T / t F t F =>

d Link ; \ T p T / t F t F =>

d Link ; \ F p T / t F t F =>

d Link ; \ a(ID) p T / t F t F =>

d Link ; \ a p F / t F t F =>

d Link ; \ a p ar(ID) / t F t F =>

d Link ; \ a p ar / t amen(ID) t F =>

d Link ; \ a p ar / t amen t o(ID)

3. Critério de correção: FIRST 0,5; FOLLOW 0,5; se não houver explicação correta vale 0.

No caso do FIRST, pretendemos os símbolos terminais que estão no início de uma derivação a partir do estado não terminal dado.

Assim, vamos começar por estender a gramática com a produção inicial S com o símbolo de fim de string $:

S → P $

P → D ; E .

D → i ID | d ID

E → E p T | T

T → T t F | F

F → \ E / | ID

Sempre que na produção tenhamos um símbolo não terminal, temos de adicionar os símbolos iniciais desse símbolo não terminal ao atual.

Assim, temos:

FIRST(D) = { 'i', 'd'}

FIRST (F) = { '\', ID }

Como temos T→F e E→T, e não existem símbolos iniciais em nenhuma produção de T e E, então temos que

FIRST (E) = FIRST(T) =FIRST (F) = { '\', ID }

De forma semelhante, temos P → D ; E . e S → P $, não existindo símbolos iniciais em nenhuma produção de P e S, logo temos:

FIRST(S) = FIRST(P) = FIRST(D) = { 'i', 'd'}

No FOLLOW, vamos querer saber que símbolos poderão aparecer após uma produção do símbolo não terminal dado.

Começando pelo S, temos apenas $, a partir de S→ P $, logo temos FOLLOW(P) = FOLLOW(S) = {'$'}.

Da produção P →D ; E, temos também FOLLOW(D) = {';'}

No caso do E, temos 3 símbolos que se podem ver imediatamente:

., da produção P → D ; E .

p, da produção E → E p T

/, da produção F → \ E /

Desta forma, temos FOLLOW(E) = { '.', 'p', '/' }.

Como temos a produção E→T, então os elementos de FOLLOW(E) também farão parte de FOLLOW(T).

No caso de FOLLOW(T), vamos adicionar mais um símbolo:

t, da produção T→T t F.

Assim, FOLLOW(T) = { ‘.’, ‘p’, ‘/’, ‘t’ }.

Como temos a produção T→F, então os elementos de FOLLOW(T) também farão parte de FOLLOW(F). Como não existe nenhuma produção em que no lado direito tenhamos Fx, onde x seja um símbolo terminal, não temos nenhum novo elemento a adicionar. Assim,

FOLLOW(F) = FOLLOW(T) = { ‘.’, ‘p’, ‘/’, ‘t’ }.

4. Usar ferramenta:
   http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/parsing-simulator.php

   (NOTA: substituir, por exemplo, ID por x, caso contrário não vai funcionar corretamente)
   Usando a ferramenta disponível, podem ver o resultado final, mas devem justificar os passos:
   - notar que não há ambiguidade: 0,2
   - no diagrama, explicar como é construído cada item: 0,4; explicar como é construído cada ramo: 0,4; diagrama: 1;
   - tabela: explicar como se constrói a tabela: 0,5; tabela: 0,5.

5. O código inicial:

Ciclos for corretamente gerado: 1,5; tratamento do array: 1; restante código 0,5.

Começamos por substituir N por 10 em todas as suas ocorrências. Geramos o seguinte código TAC:

```
_t1 = 0
i = _t1
L1:
_t2 = i
_t3 = 10
ifz _t2<_t3 goto L2
_t4 = 0
j = _t4
L3:
_t5 = j
_t6 = 10
```

```
ifz _t5<_t6 goto L4
_t7 = i
_t8 = 10
_t9 = _t7 * _t8
_t10 = j
_t11 = _t9 + _t10
_t12 = _t11 * 4
_t13 = 10
_t14 = 10
_t15 = _t13 * _t14
_t16 = i
_t17 = _t15 - _t16
_t18 = j
_t19 = _t17 – t_18
a[_t12] = _t19
_t20 = j
_t21 = 1
_t22 = _t20 + _t21
j = _t22
goto L3
L4:
_t23 = i
_t24 = 1
_t25 = _t23 + _t24
i = _t25
goto L1
L2: // resto do código
```

6. Vão ser necessários 3 tipos de otimização:
   CP (0,8) – Copy propagation (propagação de cópia)
   CF (0,2) – Constant folding (cálculo de constante)
   DCE (0,5) – Dead Code Elimination (eliminação de código morto)
   TVR (0,5) – Temporary Variable Renaming (renomeação de variável temporária)
   Vamos construir uma tabela, com uma coluna para as instruções e as outras 3 para cada tipo de otimização. É importante notar que, neste caso, a ordem de otimização é de cima para baixo e da esquerda para a direita, aplicando primeiro a CP, depois CF (que pode resultar de novo em CP)depois a DCE e, no fim, a TVR.

Instrução	CP	CF	DCE	TVR
`_t1 = 0`			X	
`i = _t1`	`i = 0`			
`L1:`				
`_t2 = i`			X	
`_t3 = 10`			X	
`ifz _t2< _t3 goto L2`	`ifz i<10 goto L2`			
`_t4 = 0`			X	
`j = _t4`	`j = 0`			
`L3:`				
`_t5 = j`			X	
`_t6 = 10`			X	
`ifz _t5<_t6 goto L4`	`ifz j<10 goto L4`			
`_t7 = i`			X	
`_t8 = 10`			X	
`_t9 = _t7 * _t8`	`_t9 = i * 10`			`_t9` → `_t1`
`_t10 = j`			X	
`_t11 = _t9 + _t10`	`_t11 = _t9 + j`			`_t11`→ `_t2`
`_t12 = _t11 * 4`				`_t12`→ `_t3`
`_t13 = 10`			X	
`_t14 = 10`			X	
`_t15 = _t13 * _t14`	`_t15 = 10 * 10`	`_t15=100`	X	
`_t16 = i`			X	
`_t17 = _t15 – t16`	`_t17 = 100 - i`			`_t17`→ `_t4`
`_t18 = j`			X	
`_t19 = _t17 – t18`	`_t19 = _t17 - j`		X	
`a[_t12] = _t19`	`a[_t12] = _t17 - j`			
`_t20 = j`			X	
`_t21 = 1`			X	
`_t22 = _t16 + _t17`	`_t22 = j + 1`		X	
`j = _t22`	`j = j + 1`			
`goto L3`				
`L4:`				
`_t23 = i`			X	
`_t24 = 1`			X	
`_t25 = _t23 + _t24`	`_t25 = i + 1`		X	
`i = _t25`	`i = i + 1`			
`goto L1`				
`L2: //resto código`				

Ficamos assim com o seguinte código final:

```
i = 0
L1:
ifz i<10 goto L2
j = 0
L3:
ifz j<10 goto L4
_t1 = i*10
_t2 = _t1 + j
_t3 = _t2 * 4
_t4 = 100 – i
a[_t3] = _t4 - j
j = j + 1
goto L3
L4:
i = i + 1
goto L1
L2: // resto do código
```

EXAME – Grupo II

Critérios de correção
Análise léxica:

ID – 1; NUM – 1; OP_COND – 0,5; restantes tokens 0,5.

Análise sintática:
Ficheiro em bison baseado nesta gramática:
Prog → Prog Inst FIM_INST | Inst FIM_INST
Inst → Atrib | WhileCicle | IfThenElse
Atrib → ID ATRIB Rhs | UP ID | DOWN ID
Rhs → BIN | OCT | DEC | HEX | ID
WhileCicle → WHILE Cond Atrib
IfThenElse → if Cond Then Atrib Resto
Resto → Else Atrib | ε
Cond → Rhs OP_COND Rhs

0,5 para Prog + Inst;
0,5 para cada um dos restantes símbolos não terminais (6 x 0,5);
0,5 conversão de números das respetivas bases para a decimal):

Geração de código:
Atribuições (0,5)
While (0,5)
If-Then-Else (0,5)
Labels e gotos corretos (0,5)

Ficheiros em anexo:

exame.l
exame.y
teste.txt

Geração do compilador (em Linux):

```
bison -dy exame.y
flex exame.l
gcc *.c -lfl
./a.out < teste.txt
```

Código gerado para o ficheiro teste.txt

```
Xpto$ = 5
Yeti$ = 31
Zundapp$ = 10
ifz Xpto$ == Yeti$ goto L0
Zundapp$ = Zundapp$ + 1
goto L1
L0:
Zundapp$ = Zundapp$ - 1
L1:
ifz Xpto$ == Yeti$ goto L2
Zundapp$ = 15
L2:
L3:
ifz Xpto$ < Yeti$ goto L4
Yeti$ = Yeti$ - 1
goto L3
L4:
```