CRITÉRIOS DE CORREÇÃO

E-FÓLIO GLOBAL E EXAME

21018 - Compilação

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

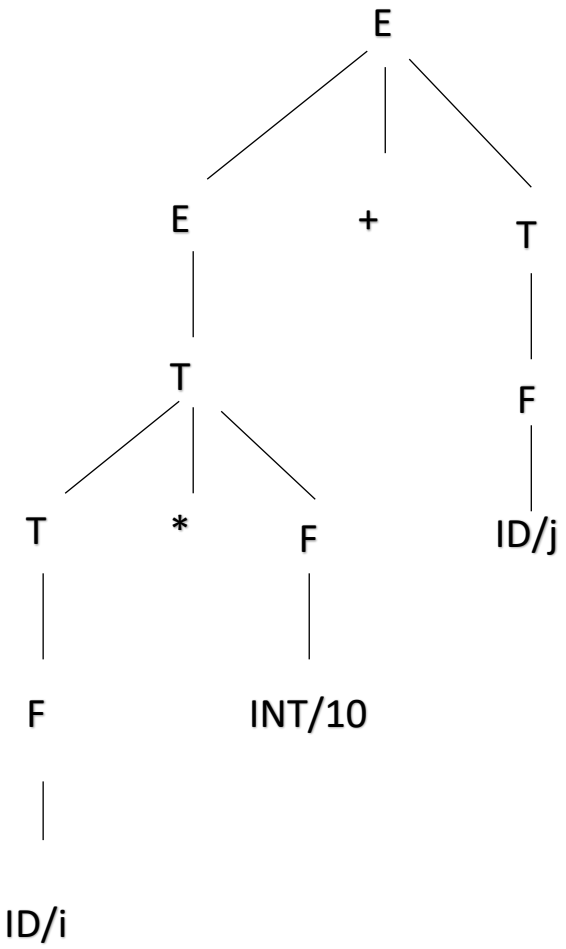
Tokens

Token ID	**Descrição**
ID	Identificador
INT	Número inteiro
MULT	Símbolo de multiplicação
PLUS	Símbolo de adição
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
i	ID
+	PLUS
j	ID
;	DLM

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



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 → E $

E → E + T | T ^ E | T

T → T * F | F

F → (E) | ID | INT

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

No caso, podemos partir de F, que é o único que tem símbolos terminais no início.

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

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

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

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

Começando pelo S, temos apenas $, a partir de S→E$.

Como S não aparece no lado direito de nenhuma produção, temos FOLLOW(S)={'$'}.

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

$, da produção S→E$;

+, da produção E→E+T;

), da produção F→(E).

Desta forma, temos FOLLOW(E) = { '$', '+', ')' }.

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 dois símbolos:

^, da produção E→T^E;

*, da produção T→T*F.

Assim, FOLLOW(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) = { ‘$’, ‘+’, ‘)’, ‘^’, ‘*’ }.

4. Antes de construir qualquer parser, convém sempre verificar se a gramática é ou não ambígua. Neste caso, verifica-se imediatamente que existe ambiguidade por causa das produções E→E+T e E →T^E.
   Por exemplo, a expressão x^y+z, poderá ter duas derivações diferentes:
   E => E + T => T ^ E + T => …
   E => T ^ E => T ^E + T => …

   Assim, caso não se resolva esta ambiguidade, teremos conflitos posteriormente na gramática. Uma solução seria fazer:
   E → E + T | T E'
   E'→ ^ E' | ε

   Tendo em conta isto, podemos resolver com a gramática original, sabendo que vai haver conflitos, ou com a modificada, sendo aceites ambas as respostas. Usando a ferramenta disponível, podem ver o resultado final, mas devem justificar os passos:
   - notar a 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 = i
_t14 = j
_t15 = _t13 + _t14
a[_t12] = _t15
_t16 = j
_t17 = 1
_t18 = _t16 + _t17
j = _t18
goto L3
```

```
L4:
_t19 = i
_t20 = 1
_t21 = _t19 + _t20
i = _t21
goto L1
L2: // resto do código
```

6. Vão ser necessários 3 tipos de otimização:
   CP (1 valor) – Copy propagation (propagação de cópia)
   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 a DCE e, no fim, a TVR.

Instrução	CP	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 = i`		X	
`_t14 = j`		X	
`_t15 = _t13 + _t14`	`_t15 = i + j`	X	
`a[_t12] = _t15`	`a[_t12] = i + j`		
`_t16 = j`		X	
`_t17 = 1`		X	
`_t18 = _t16 + _t17`	`_t18 = j + 1`	X	
`j = _t18`	`j = j + 1`		
`goto L3`			
`L4:`			
`_t19 = i`		X	
`_t20 = 1`		X	
`_t21 = _t19 + _t20`	`_t21 = i + 1`	X	
`i = _t21`	`i = i + 1`		
`goto L1`			
`L2: // resto do 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
a[_t3] = i + 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; INT – 1; OP_COND – 0,5; restantes tokens 0,5.

Análise sintática:

Gramática (0,5 por produções de cada símbolo não terminal; 0,5 conversão de números nas bases hexadecimal e octal):

Prog → Prog Inst FIM_INST | Inst FIM_INST

Inst → ID ATRIB Factor Rhs

Rhs → OP_COND Factor PI Factor Resto | ε

Resto → P2 Factor | ε

Factor → INT | ID

Geração de código:

Instrução ID = Factor ; (0,5)

Instrução ID = Factor OP_COND Factor ? Factor : Factor ; (instrução ifz 0,5; numeração correta dos labels 0,5; atribuições e goto corretamente gerados 0,5)

Ficheiro exame.l

```
%{
#include "y.tab.h"

%}


%%
[ \t\n]+                                    ;
[a-zA-Z][a-zA-Z0-9]*                        {return ID;}
[0-9]+|"\\o"[0-7]+|"\\h"[0-9A-Fa-f]+        {return INT;}
"="                                         {return ATRIB;}
"<"|">"|"<="|">="|"!="|"=="                 {return OP_COND;}
";"                                         {return FIM_INST;}
"?"                                         {return PI;}
":"                                         {return P2;}

%%
```

Ficheiro exame.y

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define YYSTYPE char *

extern char *yytext;

void conv(char *s);
int yyerror(char *s);
extern int yylex();

int label=0;
char tmpstr[30],tmpfact[30];
%}

%token ID INT ATRIB OP_COND FIM_INST PI P2


%%

prog: prog inst FIM_INST
| inst FIM_INST

inst: ID {strcpy(tmpstr,yytext);} ATRIB factor rhs
```

```
rhs: {printf("%s = %s\n",tmpstr,tmpfact);}
| OP_COND {printf("ifz %s %s ",tmpfact,yytext);} factor
{printf("%s goto L%d\n",yytext,++label);} PI factor
{printf("%s = %s\n",tmpstr,tmpfact);} resto

resto: {printf("L%d:\n",label); }
| P2 factor {printf("goto
L%d\nL%d:\n",++label,label);printf("%s =
%s\n",tmpstr,tmpfact);printf("L%d:\n",label);}

factor: INT {if(yytext[0]=='\\') conv(yytext);
strcpy(tmpfact,yytext);$$=yytext;}
| ID {strcpy(tmpfact,yytext);$$=yytext;}

%%

void main(){
yyparse();
}

void conv(char *s){
long int x;
if(s[1]=='h')
  x=strtol(s+2,NULL,16);
else if(s[1]=='o')
  x=strtol(s+2,NULL,8);
sprintf(yytext,"%ld",x);
}

int yyerror(char *s){puts(s);}
```

Ficheiro teste.txt

```
x=1;
x=a;
x=a<b?1:0;
x=a>1?1:0;
x=a>\h10?1:0;
x=\o10!=\h10?a:b;
x=1<2?a12;
```

Geração do compilador (em Linux):

```
bison -dy exame.y
flex exame.l
gcc *.c -lfl
```

Código gerado para o ficheiro teste.txt

```
x = 1
x = a
ifz a < b goto L1
x = 1
goto L2
L1:
x = 0
L2:
ifz a > 1 goto L3
x = 1
goto L4
L3:
x = 0
L4:
ifz a > 16 goto L5
x = 1
goto L6
L5:
x = 0
L6:
ifz 8 != 16 goto L7
x = a
goto L8
L7:
x = b
L8:
ifz 1 < 2 goto L9
x = a12
L9:
```