CRITÉRIOS DE CORREÇÃO

E-FÓLIO GLOBAL

21018 - Compilação

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

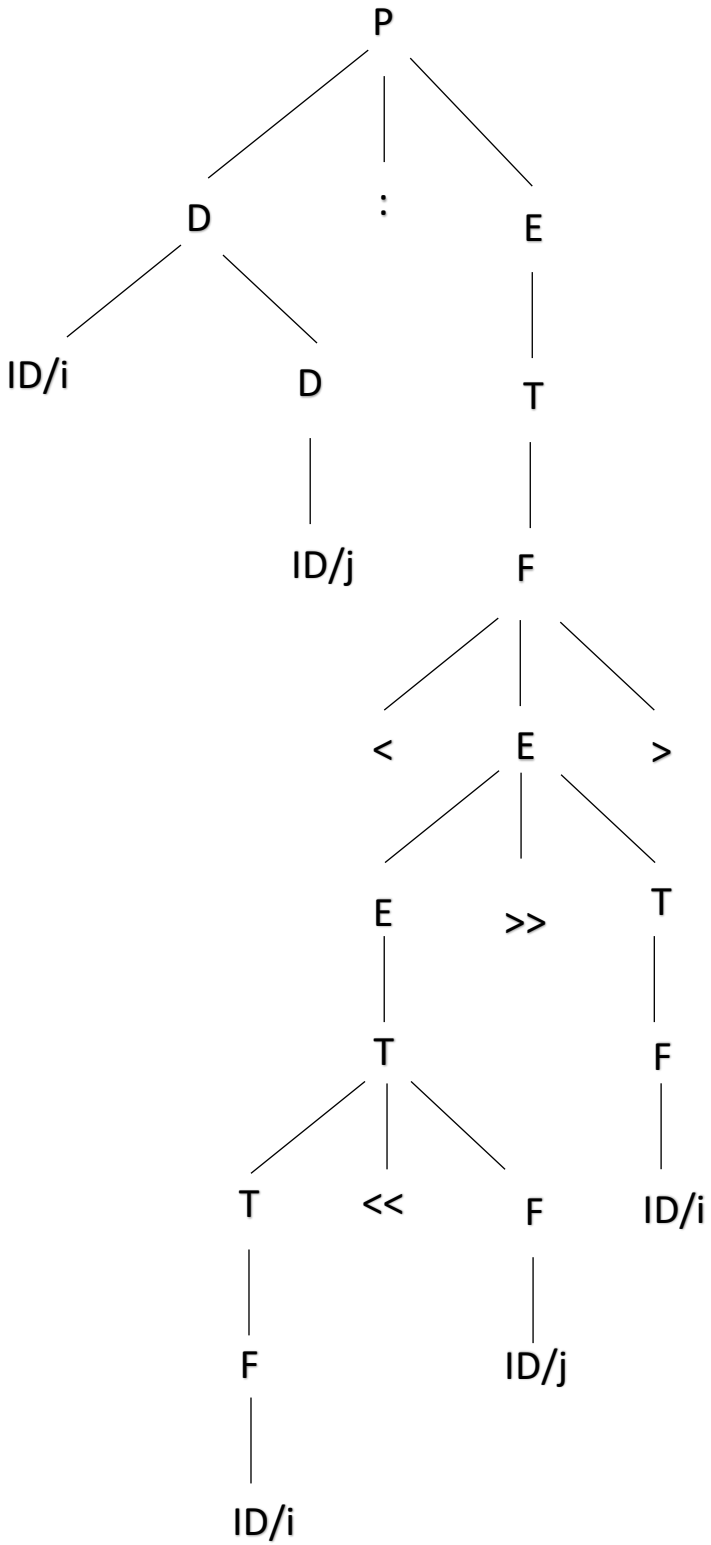
Tokens

Token ID	**Descrição**
ID	Identificador
MULT	Símbolo de multiplicação
INCR	Sequência “++”, incremento
DECR	Sequência “--", decremento
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
]	PDIR
	WS
=	ATRIB
	WS
j	ID
++	INCR
	WS
*	MULT
	WS
--	DECR
i	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 -> P $

P -> D : E

D -> ID D | ID

E -> E >> 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.

No caso, podemos partir de F e de D, que são os únicos que têm símbolos terminais no início.

FIRST(D) = { ID }

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 }

Como temos P→D : E e S→P$, e não existem símbolos iniciais em nenhuma produção de P e S, então temos que

FIRST(S) = FIRST (P) = FIRST(D) = {ID}

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

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

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

Desta produção, temos também que FOLLOW(P)={'$'}, e como P também não aparece no lado direito de nenhuma produção, fica por aqui.

No caso de D, temos ':', da produção P→D : E. Assim, FOLLOW(D) = {':'}.

No caso do E, temos 2 símbolos que se podem ver imediatamente:

>>, da produção E→E>>T;

>, da produção F→ <E>.

Além disso, como temos P→D : E, então FOLLOW(P) também fará parte de FOLLOW(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 um símbolo:

<<, 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. Usando a ferramenta disponível (ParsingEmu, disponível em http://www.supereasyfree.com/software/simulators/compilers/principles-techniques-and-tools/parsing-simulator/download.php), podem ver o resultado final, mas devem justificar os passos:
   - no diagrama, explicar como é construído cada item: 0,5; explicar como é construído cada ramo: 0,5; diagrama: 1;
   - tabela: explicar como se constrói a tabela: 0,5; tabela: 0,5.

5. O código inicial:

Ciclo while corretamente gerado: 1; tratamento do array: 1; restante código 1.

Começamos por substituir N por 50 em todas as suas ocorrências. Além disso, notar que em C, a instrução:

```
a[i] = j++ * --i;
```

é equivalente a estas três:

```
i = i - 1;
a[i] = j * i;
j = j + 1;
```

Geramos o seguinte código TAC:

```
_t1 = 50
i = _t1
_t2 = 0
j = _t2
L1:
_t3 = i
_t4 = 0
ifz _t2>_t3 goto L2
i = i - 1
_t5 = i
_t6 = _t5 * 4
_t7 = j
_t8 = i
_t9 = _t7 * _t8
a[_t6] = _t9
j = j +1
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 = 50`		X	
`i = _t1`	`i = 50`		
`_t2 = 0`		X	
`j = _t2`	`j = 0`		
`L1:`			
`_t3 = i`		X	
`_t4 = 0`		X	
`ifz _t2>_t3 goto L2`	`ifz i>0 goto L2`		
`i = i - 1`			
`_t5 = i`		X	
`_t6 = _t5 * 4`	`_t6 = i * 4`		`_t6` → `_t1`
`_t7 = j`		X	
`_t8 = i`		X	
`_t9 = _t7 * _t8`	`_t9 = j * i`	X	
`a[_t6] = _t9`	`a[_t6] = j * i`		`_t6` → `_t1`
`j = j +1`			
`goto L1`			_
`L2: //resto do código`			_

Ficamos assim com o seguinte código final:

```
i = _50

j = _0

L1:

ifz i>0 goto L2

i = i – 1

_t1 = i * 4

a[_t1] = j * i

j = j +1

goto L1

L2: //resto do código
```