E-fólio B | Folha de resolução para E-fólio

UNIDADE CURRICULAR: LINGUAGENS E COMPUTAÇÃO

CÓDIGO: 21078

DOCENTE: Constantino Martins

A preencher pelo estudante

NOME:

N.º DE ESTUDANTE:

CURSO: Licenciatura em Engenharia Informática

DATA DE ENTREGA: 14/01/2026

TRABALHO / RESOLUÇÃO:

1.a – A definição formal de uma gramática assume a forma $G = (V, T, P, S)$, em que V representa o conjunto das variáveis ou símbolos não terminais da gramática, T o conjunto de símbolos terminais, P o conjunto de produções e S o símbolo/variável de início. Para definir a gramática da linguagem BLINK (G_{BLINK}), poderemos subdividir as suas características para facilitar a sua definição.

As duas primeiras características da linguagem que são definidas é que "um programa em BLINK é uma sequência de instruções" e que "cada instrução termina com ponto (.)". Assim, podemos definir a variável "S", que marca o início da linguagem a ser definida, e podemos definir a variável "I", para representar todas as instruções possíveis. Da mesma forma, podemos definir um símbolo terminal o ponto ".". Relativamente às construções, a cabeça é a variável de início da linguagem "S", e precisamos que S defina zero ou mais instruções, para tal definiram-se as seguintes produções:

- S → ε (para uma linguagem vazia);
- S → I . S (para um conjunto recursivo de instruções).

Ou seja, para estes dois atributos da linguagem BLINK, $G1_{BLINK} = (\{S, I\}, \{.\}, P, S)$, em que P representa as produções S → ε | I . S.

Existem 3 tipos de instruções, atribuição de valor a variável (A), condicional (B) e ciclo (C). As quais definiremos separadamente:

- Iniciando pela instrução de atribuição de valor a variável, podemos definir a variável A, que marca o início da definição desta condição, e ainda a variável "E" para definir uma expressão. Relativamente aos símbolos terminais, poderíamos definir igual "=" e dois pontos ":" de forma separada, contudo, para evitar interpretações erradas por parte do analisador léxico, como ": =", assim define-se como terminal o símbolo ":=", ainda como terminal podemos definir "Var" como o token da linguagem BLINK que define o

nome de uma variável. No que diz respeito às construções, a cabeça é a variável A, e sabemos que a única produção possível é A → Var := E. Ou seja para esta condição, a gramática $G2A_{BLINK} = (\{A, E\}, \{:=, Var\}, P, A)$, em que P representa a produção A → Var := E.

- Continuando para a instrução condicional, podemos definir "B", como a variável que inicia a definição desta condição, podemos utilizar "D" como a variável para uma condição e utilizámos a variável "S" para definir zero ou mais instruções, definimos como símbolos terminais os parênteses "(" e ")", as chavetas "{" e "}" e ainda o til "~". No que diz respeito às construções, a cabeça é a variável B, e sabemos que a única produção possível é B → (D) ? { S } ~ { S }. Assim, podemos definir para esta condição, a gramática $G2B_{BLINK} = (\{B, D, S\}, \{(,), \{, \}, \sim, ?\}, P, B)$, em que P representa a produção B → (D) ? { S } ~ { S }.
- Para finalizar, a instrução ciclo, podemos definir C, como a variável iniciadora desta condição, podemos utilizar "D" como a variável para uma condição, à semelhança da instrução anterior, e a variável "S", para definir zero ou mais instruções, definimos como símbolos terminais os parênteses "(" e ")", as chavetas "{" e "}" e ainda a arroba "@". A única produção possível é, C → (D) @ { S }. Daqui, surge a definição para esta condição da gramática, $G2C_{BLINK} = (\{C, D, S\}, \{(,), \{, \}, @\}, P, C)$ em que P representa a produção C → (D) @ { S }.
- Desta análise, sabemos então que para as instruções, tomamos como símbolo inicial a variável "I", e fazemos a união do conjunto das variáveis e dos símbolos terminais. Relativamente às produções, como temos apenas uma produção por instrução, não precisamos das variáveis "A", "B" e "C", sendo as produções iniciadas em I. Assim, para a

definição da parte da gramática destinada às instruções, temos que: $G2_{BLINK} = (\{I, D, E, S\}, \{(,), \{, \}, @, \sim, ?, Var, :=\}, P, I)$, sendo que P representa as produções I → Var := E | (D) ? { S } ~ { S } | (D) @ { S }.

Seguidamente, falta-nos definir a gramática para as expressões, mantemos como variável que inicia esta parte da gramática "E". Como caracteres terminais, definimos "Var" e "Const" como tokens da linguagem BLINK que definem o nome de uma variável e uma constante, respetivamente, não necessitando de ser definidos, temos ainda os símbolos das quatro operações binárias multiplicação "*", divisão "/", adição "+" e subtração "-", bem como os parênteses "(" e ")". Relativamente às produções, temos as seguintes:

- E → Var (quando a expressão é o nome de uma variável);
- E → Const (quando a expressão é uma constante);
- E → E + E (operação binária de adição);
- E → E - E (operação binária de subtração);
- E → E * E (operação binária de multiplicação);
- E → E / E (operação binária de divisão);
- E → (E) (para forçar precedência entre expressões).

Contudo, analisando estas produções, podemos verificar que a gramática é ambígua, pois por exemplo para $w = Var + Var / Var$, poderemos ter derivações diferentes, por exemplo:

- $E \Longrightarrow E + E \Longrightarrow E + E/E \Longrightarrow Var + E/E \Longrightarrow Var + Var/E \Longrightarrow Var + Var/Var$
- $E \Longrightarrow E/E \Longrightarrow E + E/E \Longrightarrow Var + E/E \Longrightarrow Var + Var/E \Longrightarrow Var + Var/Var$

Para solucionar esta ambiguidade teremos de forçar a precedência com a adição de variáveis. Para tal, vamos introduzir a variável "X" para os identificadores, ou seja, para definir as variáveis e constantes, pois não é possível a sua divisão por qualquer operador. Vamos introduzir também a variável "F", para definirmos uma expressão que não pode ser separada pelos operadores das operações binárias, ou

seja para introduzir as expressões entre parênteses e garantir que o que se encontra dentro dos parênteses não se torne operando de um operador fora de parênteses. Por último, vamos introduzir a variável "T", para definir os termos, ou seja, expressões que não podem ser partidas pelos operadores de menor precedência "+" e "-", nomeadamente "*" e "/". Vamos manter a variável E, que permite qualquer operação incluído aquelas que podem ser quebradas por operadores adjacentes. Ou seja, as produções anteriormente referidas transformam-se em:

- X → Var (para identificar o nome de uma variável);
- X → Const (para identificar uma constante);
- F → X (quando o fator é um identificador);
- F → (E) (quando o fator é uma expressão entre parênteses);
- T → F (quando o termo é um factor);
- T → T*F (quando o termo é uma operação de multiplicação);
- T → T/F (quando o termo é uma operação de divisão);
- E → T (quando a expressão é um termo);
- E → E+T (quando a expressão é uma operação de adição);
- E → E-T (quando a expressão é uma operação de subtração).

Assim, podemos definir a parte da gramática da linguagem BLINK para as expressões, como: $G3_{BLINK} = (\{E, T, F, X\}, \{(,), +, -, *, /, Var, Const\}, P, E)$, onde P são as construções: X → Var | Const, F → X | (E), T → F | T*F | T/F e E→ T | E+T | E-T.

Por último, falta definir as condições, e vamos utilizar como variável que inicia esta parte da gramática "D", e a variável que define as expressões "E", variáveis já introduzidas anteriormente. Definimos ainda como variável "Op", para melhorar a legibilidade e manutenção da gramática, para se no futuro quisermos adicionar novos símbolos terminais como >= ou <=, em que apenas necessitamos de alterar "Op". Como símbolos terminais, teremos os caracteres "<", ">", "=", e ainda "<>" para evitar interpretações erradas do analisador léxico.

Relativamente às produções, poderemos ter cada uma das operações de comparação, ou seja:

- D → E Op E
- Op → <
- Op → >
- Op → <>
- Op → =

Pelo que, podemos definir a parte da gramática da linguagem BLINK para as condições, como: $G4_{BLINK} = (\{E, D, Op\}, \{<, >, =, <>\}, P, D)$, onde P são as construções: D → E Op E e Op → < | > | <> | =.

Conclui-se, então, que a gramática final da linguagem BLINK, G_{BLINK}, resulta da junção dos componentes anteriores $G1$, $G2$, $G3$ e $G4$, nomeadamente através da união dos seus conjuntos. O conjunto de variáveis final resulta da união das variáveis introduzidas em cada etapa (S,I,E,T,F,X,D,Op), o conjunto de terminais finais é a união de todos os símbolos terminais necessários e o conjunto de produções P é a união de todas as produções. Assim, a definição formal da gramática para a linguagem de programação BLINK é:

$G_{BLINK} = (\{S, I, E, T, F, X, D, Op\}, \{., :=, (,), ?, \{, \}, \sim, @, +, -, *, /, <, >, =, <>$ $, Var, Const\}, P, S)$, em que P consiste nas seguintes produções:

- S → ε | I . S
- I → Var := E | (D) ? { S } ~ { S } | (D) @ { S }
- X → Var | Const
- F → X | (E)
- T → F | T ∗ F | T / F
- E → T | E + T | E − T
- D → E Op E
- Op → < | > | = | <>

Esta definição formal da linguagem BLINK de acordo com a descrição informal do enunciado, garante simultaneamente a estrutura de programas como sequência de instruções terminadas por ponto, a existência dos três tipos de instruções, a construção de expressões

com precedência correta e a definição de condições por comparação entre expressões.

Para verificarmos e testarmos a gramática, seguem-se três exemplos da utilização da mesma:

- Programa simples (uma instrução de atribuição):
  - w1 = Var := Const .
  - Esta palavra resulta da seguinte derivação:

Derivação	**Produção utilizada**
S	—
I . S	S → I . S
Var := E . S	I → Var := E
Var := T . S	E → T
Var := F . S	T → F
Var := X . S	F → X
Var := Const . S	X → Const
Var := Const . ε	S → ε
Var := Const .	(resultado final)

- Duas instruções e precedência nas expressões:
  - w2: Var := Var + Var * Const . Var := (Var + Var) / Const .
  - Esta palavra resulta da seguinte derivação:

Derivação	**Produção utilizada**
S	—
I . S	S → I . S
Var := E . S	I → Var := E
Var := E + T . S	E → E + T
Var := T + T . S	E → T
Var := F + T . S	T → F
Var := X + T . S	F → X
Var := Var + T . S	X → Var
Var := Var + T * F . S	T → T * F
Var := Var + F * F . S	T → F
Var := Var + X * F . S	F → X
Var := Var + Var * F . S	X → Var
Var := Var + Var * X . S	F → X
Var := Var + Var * Const . S	X → Const
Var := Var + Var * Const . I . S	S → I . S
Var := Var + Var * Const . Var := E . S	I → Var := E
Var := Var + Var * Const . Var := T . S	E → T
Var := Var + Var * Const . Var := T / F . S	T → T / F
Var := Var + Var * Const . Var := F / F . S	T → F
Var := Var + Var * Const . Var := (E) / F . S	F → (E)
Var := Var + Var * Const . Var := (E + T) / F . S	E → E + T
Var := Var + Var * Const . Var := (T + T) / F . S	E → T
Var := Var + Var * Const . Var := (F + T) / F . S	T → F

Var := Var + Var * Const . Var := (X + T) / F . S	F → X
Var := Var + Var * Const . Var := (Var + T) / F . S	X → Var
Var := Var + Var * Const . Var := (Var + F) / F . S	T → F
Var := Var + Var * Const . Var := (Var + X) / F . S	F → X
Var := Var + Var * Const . Var := (Var + Var) / F . S	X → Var
Var := Var + Var * Const . Var := (Var + Var) / X . S	F → X
Var := Var + Var * Const . Var := (Var + Var) / Const . S	X → Const
Var := Var + Var * Const . Var := (Var + Var) / Const . ε	S → ε
Var := Var + Var * Const . Var := (Var + Var) / Const .	(resultado final)

- Programa com condicional + ciclo e blocos com zero ou mais instruções
  - w3: (Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . } . } .
  - Esta palavra resulta da seguinte derivação à esquerda:

Derivação	Produção utilizada
S	—
I . S	S → I . S
(D) ? { S } ~ { S } . S	I → (D) ? { S } ~ { S }
(E Op E) ? { S } ~ { S } . S	D → E\ Op\ E
(T Op E) ? { S } ~ { S } . S	E → T
(F Op E) ? { S } ~ { S } . S	T → F
(X Op E) ? { S } ~ { S } . S	F → X
(Var Op E) ? { S } ~ { S } . S	X → Var
(Var < E) ? { S } ~ { S } . S	Op → <
(Var < T) ? { S } ~ { S } . S	E → T
(Var < F) ? { S } ~ { S } . S	T → F
(Var < X) ? { S } ~ { S } . S	F → X
(Var < Const) ? { S } ~ { S } . S	X → Const
(Var < Const) ? { I . S } ~ { S } . S	S → I . S *(no bloco "then")*
(Var < Const) ? { Var := E . S } ~ { S } . S	I → Var := E
(Var < Const) ? { Var := E - T . S } ~ { S } . S	E → E - T
(Var < Const) ? { Var := T - T . S } ~ { S } . S	E → T
(Var < Const) ? { Var := F - T . S } ~ { S } . S	T → F
(Var < Const) ? { Var := X - T . S } ~ { S } . S	F → X
(Var < Const) ? { Var := Var - T . S } ~ { S } . S	X → Var
(Var < Const) ? { Var := Var - F . S } ~ { S } . S	T → F
(Var < Const) ? { Var := Var - X . S } ~ { S } . S	F → X
(Var < Const) ? { Var := Var - Const . S } ~ { S } . S	X → Const
(Var < Const) ? { Var := Var - Const . ε } ~ { S } . S	S → ε *(fim do bloco "then")*
(Var < Const) ? { Var := Var - Const . } ~ { I . S } . S	S → I . S *(no bloco "else")*
(Var < Const) ? { Var := Var - Const . } ~ { (D) @ { S } . S } . S	I → (D) @ { S }
(Var < Const) ? { Var := Var - Const . } ~ { (E Op E) @ { S } . S } . S	D → E\ Op\ E

(Var < Const) ? { Var := Var - Const . } ~ { (T Op E) @ { S } . S } . S	E → T
(Var < Const) ? { Var := Var - Const . } ~ { (F Op E) @ { S } . S } . S	T → F
(Var < Const) ? { Var := Var - Const . } ~ { (X Op E) @ { S } . S } . S	F → X
(Var < Const) ? { Var := Var - Const . } ~ { (Var Op E) @ { S } . S } . S	X → Var
(Var < Const) ? { Var := Var - Const . } ~ { (Var > E) @ { S } . S } . S	Op → >
(Var < Const) ? { Var := Var - Const . } ~ { (Var > T) @ { S } . S } . S	E → T
(Var < Const) ? { Var := Var - Const . } ~ { (Var > F) @ { S } . S } . S	T → F
(Var < Const) ? { Var := Var - Const . } ~ { (Var > X) @ { S } . S } . S	F → X
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { S } . S } . S	X → Const
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { I . S } . S } . S	S → I . S *(no corpo do ciclo)*
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := E . S } . S } . S	I → Var := E
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := T . S } . S } . S	E → T
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := T / F . S } . S } . S	T → T / F
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := F / F . S } . S } . S	T → F
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := X / F . S } . S } . S	F → X
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / F . S } . S } . S	X → Var
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / X . S } . S } . S	F → X
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . S } . S } . S	X → Const
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . ε } . S } . S	S → ε *(fim do corpo do ciclo)*
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . } . ε } . S	S → ε *(fim do bloco "else" após o ponto do ciclo)*
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . } . } . ε	S → ε *(fim do programa)*
(Var < Const) ? { Var := Var - Const . } ~ { (Var > Const) @ { Var := Var / Const . } . } .	(resultado final)

Para finalizar, interessa referir que se optou por permitir que a gramática aceite uma palavra vazia, através da produção S → ε. Esta opção deriva de uma sequência, em linguagem formal, poder ser

entendida como zero ou mais ocorrências, esta escolha mantém a gramática uniforme e reutilizável, pois o mesmo símbolo "S" é usado para representar o "conjunto de zero ou mais instruções" dentro dos blocos { S }. Caso se optasse por não permitir $w = \varepsilon$, ou seja se se pretendesse impor que um programa deve conter pelo menos uma instrução, a solução seria separar explicitamente "programa" de "lista de instruções", por exemplo com um símbolo "P" para o programa: $P \rightarrow I.S$ e $S \rightarrow \varepsilon \mid I.S$

1.b – Na formulação inicial da parte das expressões, a gramática considerada para "E" admitia produções do tipo $E \rightarrow E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid Var \mid Const$. Contudo, esta forma de definir expressões é ambígua, pois não impõe qualquer regra de precedência ou de associação entre operadores. Assim, para uma mesma palavra é possível obter diferentes derivações (e, consequentemente, diferentes árvores de derivação). Por exemplo, para $w = Var + Var/Var$, existem derivações que interpretam primeiro a adição e outras que interpretam primeiro a divisão, o que corresponde a leituras distintas da expressão.

Para contornar os problemas inerentes a esta ambiguidade (isto é, para garantir uma interpretação única das expressões), a solução adotada consistiu em forçar explicitamente a precedência e a estrutura sintática através da introdução de variáveis adicionais e da separação das expressões em níveis. Em concreto, foram introduzidas as variáveis "X", "F" e "T" mantendo-se "E" como variável para a expressão completa:

- "X" permite identificar os operandos básicos, limitando-os a tokens "Var" e "Const";
- "F" define um fator, isto é, um identificador ou uma expressão entre parênteses, garantindo que o conteúdo de (E) não é "partido" por operadores externos;

- "T" define um termo, permitindo apenas multiplicação e divisão ($*$ e $/$), assegurando que estes operadores têm precedência sobre adição e subtração;
- "E" define a expressão completa, permitindo apenas adição e subtração ($+$ e $-$) ao nível superior, sempre em função de termos.

Desta forma, as produções foram reestruturadas para:

$$X \rightarrow Var \mid Const$$
$$F \rightarrow X \mid (E)$$
$$T \rightarrow F \mid T * F \mid T/F$$
$$E \rightarrow T \mid E + T \mid E - T$$

Esta organização elimina a ambiguidade porque impede que um operador de menor precedência (como $+$) "intercepte" uma estrutura de maior precedência (como um termo com $*$ ou $/$). Consequentemente, uma palavra como $Var + Var/Var$ passa a ter apenas uma derivação válida, correspondendo à interpretação esperada pela linguagem BLINK: primeiro a divisão (nível T) e só depois a adição (nível E). Assim, conclui-se que a ambiguidade foi contornada ao impor, na própria gramática, a precedência dos operadores e o agrupamento explícito por parênteses, evitando derivações alternativas para a mesma expressão.