RELATÓRIO DO TRABALHO 01
ANÁLISE LÉXICA E SINTÁTICA DESCENDENTE RECURSIVA

BCC328 - Construção de compiladores I

Autores:
20.2.4182 - Leandro Marcos Mendes Zanetti
20.1.4016 - Pedro Parentoni de Almeida
Semestre 24.2

1 Introdução

Este relatório aborda detalhadamente o projeto em Haskell que se propõe a implementar um compilador para uma linguagem chamada "lang". O projeto utiliza três componentes principais: a análise léxica, a análise sintática e a geração de uma representação estruturada de árvore sintática (AST). Este relatório descreve a estrutura do projeto, a estrutura dos arquivos e pastas, e as funcionalidades principais desenvolvidas, além de desafios e sugestões de melhorias. Além de cada módulo, e as funcionalidades desenvolvidas, bem como os desafios enfrentados e sugestões de melhorias.

2 Contexto Geral do Projeto

O projeto visa simular o funcionamento de um compilador, abordando etapas fundamentais do processo de compilação. Como por exemplo:

1. Analisar a entrada fornecida por arquivos de código-fonte.
2. Reconhecer tokens utilizando um analisador léxico (lexer).
3. Construir uma árvore sintática que represente a estrutura hierárquica do código.
4. Exibir o resultado de forma legível no terminal.

Essas funcionalidades cobrem aspectos fundamentais da compilação, desde a identificação de palavras-chave e símbolos até a construção de uma representação estruturada que pode ser utilizada em fases posteriores, como a geração de código intermediário ou final.

3 Estrutura do projeto

O projeto está organizado em três principais diretórios: app, src e test.

1. Pasta app: Contém o arquivo principal Main.hs, responsável por iniciar o programa, interagir com o usuário e integrar os módulos.
2. Pasta src: Contém os módulos principais do sistema:
   - Lexer.hs: Implementação do analisador léxico que converte o código de entrada em uma lista de tokens.
   - Parser.hs: Responsável pela análise sintática e geração da árvore sintática.
   - Simplecombinators.hs: Biblioteca de combinadores disponibilizada pelo professor.
3. Pasta test: Inclui arquivos de entrada como exemplo.lang, que servem como base para testar as funcionalidades do compilador.

4 Como Compilar e Executar

4.1 Compilar

```
stack build
```

4.2 Executar

```
stack exec lang-compiler-exe
```

4.3 Dentro do Programa

Será exibido um menu de escolhas, se uma das duas primeiras opções forem selecionadas, o arquivo `exemplo.lang` é lido automaticamente. Para alterar a entrada do programa, basta alterar o conteúdo de `exemplo.lang`.

5 Arquivo Principal

O módulo `Main.hs` é responsável por gerenciar a interação entre o usuário e as funcionalidades do compilador. Ao ser executado, o compilador exibe um menu inicial, apresentando ao usuário três opções: o modo `lexer`, o modo `recursive` e o modo `help`. Dependendo da escolha do usuário, uma das funções correspondentes será acionada. A principal função do módulo é servir de controlador entre a entrada do usuário e a execução das funcionalidades do compilador, como a análise léxica e a análise sintática.
A implementação atual é funcional e oferece uma boa base para futuras extensões do compilador. Por exemplo, seria possível adicionar mais modos de operação, como um modo de geração de código intermediário ou final. Além disso, a implementação de verificações adicionais de erros no fluxo de entrada do usuário poderia melhorar ainda mais a robustez do sistema.

5.1 Interação com o Usuário

Como dito, logo no início da execução do compilador, o programa saúda o usuário e apresenta um menu simples de opções. O usuário é então solicitado a fazer uma escolha, digitando um número correspondente a uma das opções disponíveis. As opções disponíveis são:

- **Modo Lexer**: Este modo executa a análise léxica do código-fonte, processando o texto de entrada e gerando uma lista de tokens. O lexer divide o código-fonte em unidades significativas, como palavras-chave, operadores e identificadores.
- **Modo Recursive**: Neste modo, o compilador executa a análise sintática recursiva, processando o código-fonte para gerar a árvore sintática correspondente. A árvore gerada representa a estrutura hierárquica do programa e é fundamental para as fases posteriores do compilador, como a geração de código intermediário.
- **Modo Help**: Caso o usuário precise de esclarecimentos, o modo ajuda exibe uma descrição das opções disponíveis no menu, fornecendo informações sobre o que cada modo faz.

O processo de interação é simples e intuitivo. Após o usuário selecionar uma opção, o programa executa a funcionalidade correspondente, chamando a função apropriada para processar o código-fonte ou exibir informações úteis.

5.2 Fluxo de Execução

O fluxo de execução do módulo `Main.hs` é baseado em um menu interativo. Primeiramente, o programa exibe as opções para o usuário e aguarda a escolha dele. Com base na escolha, o programa tem um comportamento distinto:

- No modo `lexer`, o programa lê o caminho do arquivo de entrada e verifica seu conteúdo, após isso, passa para a função responsável pela análise léxica, que irá gerar e exibir os tokens encontrados.
- No modo `recursive`, o programa lê o caminho do arquivo de entrada e verifica seu conteúdo, após isso, passa esse conteúdo para o analisador sintático recursivo, que processa os tokens e gera a árvore sintática correspondente.
- No modo `help`, o programa exibe informações explicativas sobre o funcionamento do compilador e as opções do menu.

Caso o usuário escolha uma opção inválida, o programa solicita que ele tente novamente, garantindo que a interação seja contínua e sem falhas.

6 Lexer

O módulo Lexer.hs implementa um analisador léxico que:

- Identifica palavras-chave como print, if, else, return.
- Reconhece símbolos de operação (+, -, *, /, etc.) e pontuação (;, , , etc.).
- Gera uma lista de tokens que servem como entrada para a próxima etapa.

O módulo `Lexer.hs` contém as definições e funções responsáveis pela análise léxica, que tem como objetivo identificar as palavras-chave, identificadores, números e símbolos da linguagem "lang". A análise léxica é a primeira etapa no processo de compilação, onde o código-fonte é transformado em uma sequência de tokens.
A função principal do módulo, lexer, percorre o código-fonte caracter por caracter, agrupando símbolos e palavras-chave em tokens adequados.

6.1 Definição do Tipo de Token

O primeiro elemento do módulo é a definição do tipo de dados `Token`, que representa as diferentes categorias de tokens reconhecidas pelo lexer. Cada token pode ser um identificador (`ID String`), um número inteiro (`INT Int`), ou uma palavra-chave, como `PRINT`, `IF`, `ELSE`, `THEN`, `RETURN`. Além disso, o lexer também reconhece operadores (`OP String`) e pontuação (`PUNC String`), como parênteses e operadores aritméticos.

```
data Token = ID String | INT Int | PRINT | IF | ELSE | THEN | RETURN
           | OP String | PUNC String deriving (Show)
```

Cada construtor de `Token` representa uma categoria específica, e a função `Show` é derivada para permitir a exibição dos tokens no terminal durante a execução do programa.

6.2 Função Principal `lexer`

A função `lexer` é o núcleo do analisador léxico. Ela recebe uma string (o código-fonte) e retorna uma lista de tokens. A função percorre o código caractere por caractere e, com base nas características de cada caractere, decide qual token gerar.

```
lexer :: String -> [Token]
lexer [] = []
lexer (c:cs)
  | isSpace c = lexer cs
  | isAlpha c = lexIdentifier (c:cs)
  | isDigit c = lexNumber (c:cs)
  | c `elem` "(){}[];,:" = PUNC [c] : lexer cs
  | c `elem` "+-*/<>=!%" = OP [c] : lexer cs
  | otherwise = error $ "caractere inesperado: " ++ [c]
```

A função começa verificando se o caractere atual é um espaço, um identificador, um número ou um símbolo de pontuação/operador. Se o caractere for um espaço, ele é ignorado e a função continua com os caracteres restantes. Se for um caractere alfabético (`isAlpha`), a função `lexIdentifier` é chamada para identificar palavras-chave ou identificadores. Se o caractere for numérico (`isDigit`), a função `lexNumber` é chamada para processar o número. Caso o caractere seja um dos símbolos de pontuação ou operadores definidos, o token correspondente é gerado. Se o caractere não se encaixar em nenhuma dessas categorias, um erro é gerado indicando um caractere inesperado.
O trecho de código a seguir define uma instância da classe de tipo `Eq` para o tipo de dado `Token`. Isso permite que valores do tipo `Token` sejam comparados utilizando o operador `==`. A implementação considera diferentes casos de correspondência para os construtores do tipo `Token`, conforme descrito abaixo:

```
instance Eq Token where
    (ID s1) == (ID s2) = s1 == s2
    (INT i1) == (INT i2) = i1 == i2
    PRINT == PRINT = True
    IF == IF = True
    ELSE == ELSE = True
    THEN == THEN = True
    RETURN == RETURN = True
    (OP op1) == (OP op2) = op1 == op2
    (PUNC p1) == (PUNC p2) = p1 == p2
    _ == _ = False
```

- `(ID s1) == (ID s2)`: Dois tokens do tipo `ID` são iguais se suas `String`s associadas (`s1` e `s2`) forem iguais.
- `(INT i1) == (INT i2)`: Dois tokens do tipo `INT` são iguais se os números inteiros associados (`i1` e `i2`) forem iguais.
- `PRINT == PRINT`, `IF == IF`, `ELSE == ELSE`, `THEN == THEN`, `RETURN == RETURN`: Esses tokens literais são iguais se forem exatamente os mesmos.
- `(OP op1) == (OP op2)`: Dois tokens do tipo `OP` são iguais se as suas `String`s (`op1` e `op2`) forem iguais.
- `(PUNC p1) == (PUNC p2)`: Dois tokens do tipo `PUNC` são iguais se as suas `String`s (`p1` e `p2`) forem iguais.
- `_ == _`: Para todos os outros casos (quando os construtores dos tokens são diferentes), o resultado da comparação é `False`.

Esse comportamento assegura que o operador `==` realiza uma comparação exata entre os valores do tipo `Token`, respeitando as características de cada construtor.

6.3 Função `lexIdentifier`

A função `lexIdentifier` é responsável por identificar identificadores e palavras-chave. Ela começa lendo o máximo possível de caracteres alfabéticos e, em seguida, verifica se a palavra formada corresponde a uma palavra-chave (como `print`, `if`, `else`, `then`, `return`). Se for uma palavra-chave, o token correspondente (como `PRINT` ou `IF`) é gerado. Caso contrário, um token `ID` é criado para representar o identificador.

```
lexIdentifier :: String -> [Token]
lexIdentifier str =
  let (name, rest) = span isAlpha str
  in case name of
       "print"  -> PRINT : lexer rest
       "if"     -> IF : lexer rest
       "else"   -> ELSE : lexer rest
       "then"   -> THEN: lexer rest
       "return" -> RETURN : lexer rest
       _        -> ID name : lexer rest
```

Essa função utiliza a função `span` para separar o nome do identificador ou palavra-chave da parte restante da string. Em seguida, ela compara o nome com as palavras-chave e retorna o token apropriado.

6.4 Função `lexNumber`

A função `lexNumber` é responsável por identificar números inteiros no código. Ela lê os caracteres numéricos consecutivos e os converte em um valor inteiro. Após a conversão, um token `INT` é criado com o valor do número.

```
lexNumber :: String -> [Token]
lexNumber str =
  let (num, rest) = span isDigit str
  in INT (read num) : lexer rest
```

A função `span` é utilizada novamente, desta vez para separar os caracteres numéricos, que são então convertidos em um valor do tipo `Int`.

6.5 Função `printTokens`

O trecho de código a seguir define uma função auxiliar chamada `printTokens`, que serve para imprimir uma lista de tokens (`[Token]`) no terminal. Essa função vai permitir a visualização dos tokens no terminal.

```
printTokens :: [Token] -> IO ()
printTokens tokens = mapM_ print tokens
```

Explicação:

- `mapM_ print tokens`:
  - `mapM_`: Uma função padrão de Haskell que aplica uma função com efeitos colaterais (`print`, neste caso) a cada elemento de uma lista (`tokens`) e descarta o resultado acumulado.
  - `print`: Função que converte um valor para uma `String` e imprime no terminal.
  - `tokens`: A lista de tokens que será processada.
- Resultado: Para cada elemento na lista `tokens`, a função `print` é chamada, imprimindo o token no terminal em linhas separadas.

7 Parser

O módulo `Parser.hs` é responsável pela análise sintática do compilador "lang". Ele recebe uma sequência de tokens, gerada pelo analisador léxico (lexer), e constrói a Árvore de Sintaxe Abstrata (AST) que representa a estrutura hierárquica, a qual facilita a compreensão e o processamento do código-fonte. É importante ressaltar que o grupo teve muita dificuldade nesta etapa de parsing, além das dificuldades encontradas na implementação, houve problemas com a identação no momento da impressão da árvore no terminal. A seguir, será apresentada uma descrição detalhada sobre as principais funções presentes neste módulo. Ele utiliza técnicas de correspondência de padrões e funções auxiliares para processar estruturas como funções, blocos e expressões. A `AST` gerada é formatada e impressa na tela.
Definição do tipo de Árvore de Sintaxe Abstrata:

```
data AST
    = Node String [AST] -- Um nó com um rótulo e subárvores
    | Leaf String       -- Uma folha com um valor
    deriving (Show)
```

7.1 Definição da Árvore de Sintaxe (AST)

A estrutura principal do módulo é a Árvore de Sintaxe (AST), que é definida como um tipo algébrico com duas variantes:

- **Node**: Representa um nó da árvore que possui um rótulo (uma string) e uma lista de subárvores (ou filhos).
- **Leaf**: Representa uma folha da árvore que contém um valor simples (também uma string).

7.2 Função Principal: `parseAndPrintAST`

```
parseAndPrintAST :: String -> IO ()
parseAndPrintAST content = do
  let tokens = lexer content
  let ast = buildAST tokens
  putStrLn "\nÁrvore de Sintaxe Gerada:"
  putStrLn (formatAST (Node "Program" ast))
```

Essa função é responsável por:

1. Tokenizar o conteúdo da entrada usando `lexer`.
2. Construir a `AST` utilizando `buildAST`.
3. Exibir a árvore formatada com `formatAST`.

7.3 Função `formatAST`

Essa função exibe a `AST` em formato estilizado como uma árvore hierárquica.

```
formatAST :: AST -> String
formatAST ast = unlines (formatHelper ast "")
```

Descrição dos Auxiliares Que Existem Dentro da Função:

- `formatHelper`: Processa recursivamente os nós da `AST`, adicionando prefixos que representam a hierarquia.
- `concatMapWithPrefix`: Ajusta os prefixos ao concatenar subárvores.
- `replaceLastBar`: Remove a última barra vertical do prefixo para o último filho, ajustando a formatação.

7.4 Funções de Parsing

As funções a seguir processam `Token`s e constroem a `AST`.

7.4.1 Função `parseBlock`

```
parseBlock :: [Token] -> ([AST], [Token])
parseBlock (PUNC "{" : rest) =
  let (blockTokens, remaining) = span (/= PUNC "}") rest
  in (buildAST blockTokens, drop 1 remaining)
parseBlock tokens = ([], tokens)
```

`parseBlock` processa blocos delimitados por chaves ({ }), retornando a `AST` do bloco de código com seus respectivos tokens, além dos `Token`s restantes.

7.4.2 Função `parseExpression`

```
parseExpression :: [Token] -> ([AST], [Token])
parseExpression (ID name : rest) = ([Leaf ("ID: " ++ name)], rest)
parseExpression (INT val : rest) = ([Leaf ("INT: " ++ show val)], rest)
parseExpression (OP op : rest) = ([Leaf ("OP: " ++ op)], rest)
parseExpression (PUNC p : rest) = ([Leaf ("PUNC: " ++ p)], rest)
parseExpression tokens = ([], tokens)
```

Essa função identifica diferentes tipos de expressões (`ID`, `INT`, `OP`, etc.) e constrói folhas da `AST` para cada uma delas.

7.4.3 Função `buildAST`

```
buildAST :: [Token] -> [AST]
buildAST [] = []
buildAST (ID name : PUNC "(" : rest) =
  let (params, remaining1) = span (/= PUNC ")") rest
      (body, remaining2) = parseBlock (drop 1 remaining1)
```

```
  in Node ("Function: " ++ name) (map (Leaf . show) params ++ body) :
  buildAST remaining2
  ...
```

Essa função é maior dentro do arquivo "Parser.hs", para esse relatório foi selecionado apenas uma parte dela para explicação, por isso o uso das reticências, essa função é responsável por construir a `AST` completa com base em uma sequência de `Token`s. Os principais casos incluem:

- `Funções`: Identificadas por um `ID` seguido de parênteses, com parâmetros e corpo processados como nós e subárvores.
- `Print`: Cria um nó "Print" com subárvores representando as expressões.
- `If`: Constrói uma árvore com nós "If", "Then" e "Else", processando os respectivos blocos.
- `Return`: Constrói um nó "Return" com a expressão associada.
- `Outros casos`: Blocos são representados por nós "Block".

8 Teste de Entrada e Saída

8.1 Entrada

O programa abaixo é um exemplo de uso do compilador, ele está alocado dentro da pasta "test" e serve para testar todo o projeto:

```
main() {
  print fat(10)[0];
}

fat(num :: Int) : Int {
  if (num < 1)
    return 1;
  else
    return num * fat(num-1)[0];
}

divmod(num :: Int, div :: Int) : Int, Int {
  q = num / div;
  r = num % div;
  return q, r;
}
```

8.2 Saída

8.2.1 Saída do Lexer

Abaixo tem-se um exemplo de saída que o compilador fornece, quando se seleciona a opção "Lexer":

```
Tokens encontrados:
ID "main"
PUNC "("
PUNC ")"
PUNC "{"
PRINT
ID "fat"
PUNC "("
INT 10
PUNC ")"
PUNC "["
INT 0
PUNC "]"
PUNC ";"
PUNC "}"
ID "fat"
PUNC "("
ID "num"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ")"
PUNC ":"
ID "Int"
PUNC "{"
IF
PUNC "("
ID "num"
OP "<"
INT 1
PUNC ")"
RETURN
INT 1
PUNC ";"
ELSE
RETURN
ID "num"
OP "*"
ID "fat"
PUNC "("
ID "num"
OP "-"
INT 1
PUNC ")"
PUNC "["
```

```
INT 0
PUNC "]"
PUNC ";"
PUNC "}"
ID "divmod"
PUNC "("
ID "num"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ","
ID "div"
PUNC ":"
PUNC ":"
ID "Int"
PUNC ")"
PUNC ":"
ID "Int"
PUNC ","
ID "Int"
PUNC "{"
ID "q"
OP "="
ID "num"
OP "/"
ID "div"
PUNC ";"
ID "r"
OP "="
ID "num"
OP "%"
ID "div"
PUNC ";"
RETURN
ID "q"
PUNC ","
ID "r"
PUNC ";"
PUNC "}"
```

8.2.2 Saída do Recursive (Parser)

Devido aos problemas e dificuldades com a implementação e depois com a identação da árvore, o compilador "entrega" como resultado uma árvore com os tokens do programa lido em sequência, buscando obedecer a hierarquia entre eles.
A árvore para o programa de entrada `exemplo.lang` foi construída e teve como saída:

```
|- Program
|  |- Function: main
|  |  |- Print
|  |  |  L- ID: fat
|  |  L- PUNC "("
|  |  L- INT 10
|  |  L- PUNC ")"
|  |  L- PUNC "["
|  |  L- INT 0
|  |  L- PUNC "]"
|  |  L- PUNC ";"
|  |- Function: fat
|  |  L- ID "num"
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  L- PUNC ":"
|  L- ID "Int"
|  |- Block
|  |- If
|  |  L- PUNC: (
|  |  |- Then
|  |  |- Else
|  L- ID "num"
|  L- OP "<"
|  L- INT 1
|  L- PUNC ")"
|  |- Return
|  |  L- INT: 1
|  L- PUNC ";"
|  L- ELSE
|  |- Return
|  |  L- ID: num
|  L- OP "*"
|  |- Function: fat
|  |  L- ID "num"
|  |  L- OP "-"
|  |  L- INT 1
|  L- PUNC "["
|  L- INT 0
|  L- PUNC "]"
|  L- PUNC ";"
|  L- PUNC "}"
|  |- Function: divmod
|  |  L- ID "num"
```

```
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  |  L- PUNC ","
|  |  L- ID "div"
|  |  L- PUNC ":"
|  |  L- PUNC ":"
|  |  L- ID "Int"
|  L- PUNC ":"
|  L- ID "Int"
|  L- PUNC ","
|  L- ID "Int"
|  |- Block
|  L- ID "q"
|  L- OP "="
|  L- ID "num"
|  L- OP "/"
|  L- ID "div"
|  L- PUNC ";"
|  L- ID "r"
|  L- OP "="
|  L- ID "num"
|  L- OP "%"
|  L- ID "div"
|  L- PUNC ";"
|  |- Return
|  |  L- ID: q
|  L- PUNC ","
|  L- ID "r"
|  L- PUNC ";"
|  L- PUNC "}"
```

9 Desafios Encontrados

1. Parsing de expressões complexas: A manipulação de expressões aninhadas exigiu tratamentos específicos para evitar erros de interpretação.

2. Geração e Visualização da árvore: Diante do arquivo `exemplo.lang` o grupo não conseguiu fazer a árvore ficar corretamente identada e bem apresentável na tela, houveram diversas tentativas sem sucesso e esse foi o principal dificultador do projeto, o que impossibilitou de entregá-lo totalmente completo.

10 Recomendações de Melhoria

1. Melhoria no Lexer: Ampliar a capacidade de reconhecimento de tipos e símbolos, permitindo maior flexibilidade na linguagem.

2. Tratamento de Erros: Adicionar mensagens de erro mais descritivas no parser para auxiliar na depuração.

3. Melhorar a Árvore de Sintaxe: Entender melhor como se dá a construção da árvore para em uma próxima etapa conseguir exib-la na tela da forma correta.

11 Conclusão

O projeto do compilador "lang" foi desenvolvido com o objetivo de implementar as fases iniciais de um compilador simples, consistindo principalmente da análise léxica e da análise sintática. O módulo de análise léxica (`Lexer.hs`), responsável por identificar e classificar os tokens de um programa, juntamente com o módulo de análise sintática (`Parser.hs`), que constrói a Árvore de Sintaxe Abstracta (AST) a partir dos tokens gerados, formam a base para a transformação do código-fonte em uma representação estruturada que pode ser utilizada em etapas posteriores, como a geração de código intermediário ou otimizações.

11.1 Análise Léxica

A função de análise léxica (`lexer`) tem um papel crucial em reconhecer a linguagem do código-fonte. Ela é responsável por processar a entrada caractere por caractere, agrupando-os em tokens, que são as unidades de significado para o compilador. O módulo trata de diferentes tipos de tokens, como identificadores, números, operadores e pontuações, assegurando que o código seja segmentado de forma adequada para a análise sintática subsequente. A principal vantagem dessa abordagem é a clareza e modularidade do código, o que facilita a manutenção e a expansão. A implementação do lexer permite uma rápida identificação de erros de sintaxe simples, como caracteres inesperados. Além disso, a inclusão de palavras-chave como `if`, `else`, `then` e `print` no lexer permite que o compilador compreenda e trate essas construções de forma apropriada.

11.2 Análise Sintática

Apesar dessa etapa não ter sido concluída da forma que o grupo gostaria, devido as dificuldades, é importante entender seu funcionamento teórico: O código desenvolvido implementa um parser simples e funcional para a construção de uma Árvore de Sintaxe Abstrata (AST), a partir de uma sequência de tokens gerada pelo analisador léxico (*lexer*). Esse parser foi projetado para identificar estruturas sintáticas específicas, como blocos de código delimitados por chaves, expressões matemáticas e estruturas condicionais, representando-as de forma hierárquica por meio de nós e folhas na AST.

O processo é dividido em três etapas principais. Primeiramente, o texto de entrada é convertido em uma lista de tokens que descreve os componentes básicos do programa. Em seguida, a função `buildAST` percorre recursivamente essa lista, construindo a AST com base em padrões sintáticos identificados. Por fim, a função `formatAST` apresenta a AST de forma visual, enquanto `parseAndPrintAST` integra todas as etapas, facilitando a compreensão do resultado.

Um analisador descendente recursivo, como o desenvolvido, oferece vantagens como a simplicidade de implementação, maior clareza no código e facilidade de depuração. Ele é especialmente útil para linguagens com gramáticas simples, e como dito anteriormente, pode servir como base para futuras extensões, como análise semântica ou geração de código.