
**INSTITUTO SUPERIOR POLITÉCNICO DE TECNOLOGIAS E CIÊNCIAS**

**DEPARTAMENTO DE ENGENHARIAS E TECNOLOGIAS**

**COMPILADORES**

RELATÓRIO DO TRABALHO PRÁTICO (3º fase)

EXAME

# MANUAL DO PROGRAMADOR


| **ESTUDANTE** | |
|---|---|
| Rui Yuri Joaquim Malemba - 20201580 | |
| **CURSO:** ENGENHARIA INF6<br>**TURMA:** M1 | **DOCENTE:**<br>André Filemon |
| Data de Realização do Projecto: Julho de 2023 | |

## ✓ ANÁLISADOR LÉXICO: TOKENS

Fornecemos uma tabela com os significados dos tokens que o analisador léxico reconhece. Essa tabela é muito útil para entender como o analisador léxico implementado está funcionando e como ele classifica diferentes partes do código ou do texto que está sendo analisado.

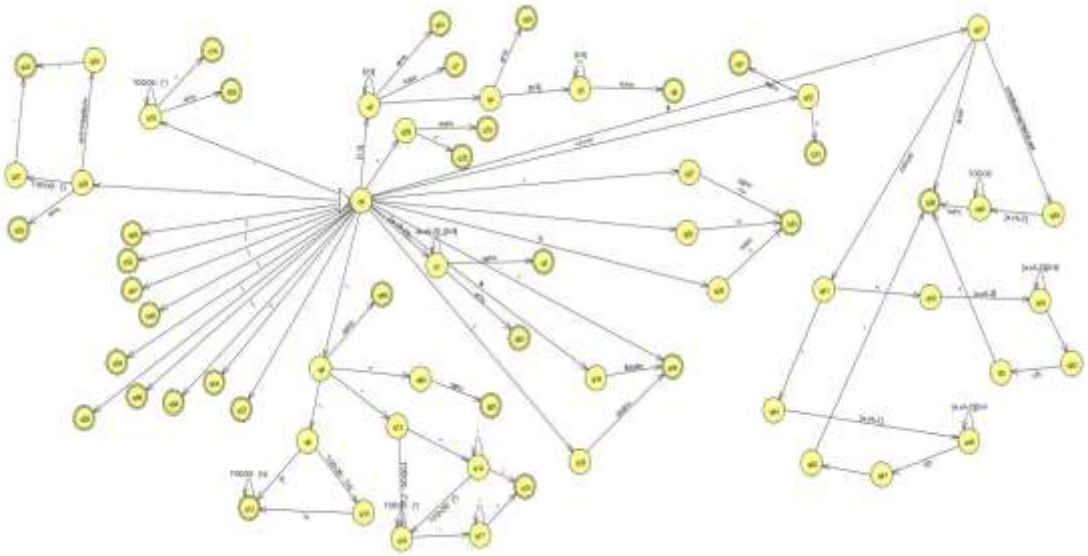
- TK_ID: Identificador ou nome de uma variável.
- TK_CM: Comentário. É uma parte do código que não é executada e serve apenas como notas explicativas para os programadores.
- TK_DIR: Diretiva. É uma instrução especial que é processada antes do código ser compilado ou executado.
- TK_AP: Abre parênteses.
- TK_FP: Fecha parênteses.
- TK_AC: Abre chaves.
- TK_FC: Fecha chaves.
- TK_APR: Abre colchetes ou parênteses retos.
- TK_FPR: Fecha colchetes ou parênteses retos.
- TK_VIRG: Vírgula. É um símbolo utilizado para separar itens em uma lista ou parâmetros de uma função.
- TK_PVIRG: Ponto e vírgula. É um símbolo utilizado para separar instruções em um programa ou código.
- TK_PINTER: Ponto de interrogação. É um símbolo utilizado para indicar uma pergunta ou dúvida em uma frase.
- TK_DPTO: Dois pontos. É um símbolo utilizado para introduzir uma lista, uma enumeração ou uma explicação.
- TK_PTO: Ponto. É um símbolo utilizado para indicar o final de uma frase ou abreviação.

- TK_C: Caractere. É um símbolo ou letra individual utilizado em um texto ou código.
- TK_S: String. É uma sequência de caracteres ou texto, geralmente utilizado para armazenar palavras ou frases em um programa.
- TK_OPL: Operador lógico. É um símbolo utilizado em expressões booleanas para indicar uma operação lógica (como "e" ou "ou").
- TK_OPR: Operador relacional. É um símbolo utilizado para comparar valores em expressões condicionais (como "maior que" ou "igual a").
- TK_OPA: Operador aritmético. É um símbolo utilizado para realizar operações matemáticas (como adição, subtração, multiplicação ou divisão).
- TK_OPATB: Operador de atribuição. É um símbolo utilizado para atribuir um valor a uma variável (como "=" ou "+=").
- TK_NUMINT: Número inteiro. É um número inteiro positivo ou negativo.
- TK_FIM: Final da execução. É um símbolo utilizado para indicar o fim de um programa ou código.

| TOKEN | Significado |
|---|---|
| TK_ID | dentificador/Variável |
| TK_CM | Comentário |
| TK_DIR | Directiva |
| TK_AP | Abre parenteses |
| TK_FP | Fecha parenteses |
| TK_AC | Abre chavetas |
| TK_FC | Abre chavetas |
| TK_APR | Abre parenteses reto |
| TK_FPR | Fecha parenteses reto |
| TK_VIRG | Vírgula |
| TK_PVIRG | Ponto e vírgula |
| TK_PINTER | Ponto de interrogação |
| TK_DPTO | Dois pontos |
| TK_PTO | Ponto |
| TK_C | Caractere |
| TK_S | String |
| TK_OPL | Operador Lógico |
| TK_OPR | Operador relacional |
| TK_OPA | Operador aritmético |
| TK_OPATB | Operador atribuição |

| | |
|---|---|
| TK_NUMINT | Número inteiro |
| TK_FIM | Final da execução |
| TK_NUMREAL | Número real |

## ✓ AUTOMATO FINITO DETERMINISTICO (AFD)



## ✓ PSEUDOCÓDIGO

```
função analex():
    enquanto houver linhas no arquivo lido:
        textoDaLinha = próxima linha do arquivo lido
        contarLinhasLidas = contarLinhasLidas + 1
        indice = 0
        enquanto indice < comprimento de textoDaLinha:
            lerChar = próximo caractere de textoDaLinha
            estado = 0
            se lerChar é um caractere em branco:
                continue para o próximo caractere
```

```
        senão, se lerChar é um dígito:
            textoLexema = lerChar
            estado = 1
            enquanto próximo caractere de textoDaLinha é um dígito:
                textoLexema = textoLexema + próximo caractere de textoDaLinha
            saída: "número, " + textoLexema
        senão, se lerChar é uma letra:
            textoLexema = lerChar
            estado = 2
            enquanto próximo caractere de textoDaLinha é uma letra ou dígito:
                textoLexema = textoLexema + próximo caractere de textoDaLinha
            se textoLexema é uma palavra-chave:
                saída: "palavra-chave, " + textoLexema
            senão:
                saída: "identificador, " + textoLexema
        senão, se lerChar é um operador:
            textoLexema = lerChar
            estado = 3
            enquanto próximo caractere de textoDaLinha é um operador:
                textoLexema = textoLexema + próximo caractere de textoDaLinha
            saída: "operador, " + textoLexema
        senão, se lerChar é um delimitador:
            saída: "delimitador, " + lerChar
        senão:
            saída: "erro léxico: caractere inválido, " + lerChar + ", linha " +
contarLinhasLidas
        indice = indice + 1
```

- ✓ Linguagem de estudo: C
- ✓ Linguagem de implementação: Java

## ✓ ANLISADOR SINTÁTICO: PARSER

Antes de explanar sobre a gramática utilizada, sua arrumação etc, e a forma de implementação do mini parser, é importante definir o que é um analisador sintático e o que é uma gramática. Basicamente, analisador sintático ou parser é uma ferramenta que realiza a análise dos símbolos validados pelo automáto(Analisador léxico), com base na gramática, ou seja com base nas sintaxes da linguagem, já estabelecida. Todas as linguagens de programação possuem uma gramática. Uma gramática é um conjunto de regras de produção também chamadas de sintaxe. Através da gramática, o programador consegue criar um compilador e ou um interpretador e ou uma linguagem de programação. Na fase 2 do projecto Mini Compilador, foi implementado um melhoramento do analisador léxico, e o anlisador sintático pela primeira vez, agora com uma interface gráfica mais amigável e sugestiva. Foi utilizada uma mini gramática da linguagem C para a construção do parser. Utilizou-se a linguagem java na implementação do parser, tal como na fase anterior foi utilizada para implementar o analex.

Funcionalidades: O meu analisador para além das funcionalidades anteriores, possui agora o reconhecimento de palavras reservadas, leitura de um código C normal, diferente do anterior que só lia o texto e validava os símbolos linha a linha, foi otmizado no que diz respeito ao analisador léxico, hoje ele apresenta os erros de maneira elegante, com a cor vermelha. As novas funcionalidade são validação da sintaxe de acordo a linguagem C, ou seja não aceita códigos que não obedecem as sintaxes do C, limitando-se a gramática que foi utilizada.

Destacamos agora algumas das novas funcionalidades: Declaração de variável validada, declaração e definição de função validada, parâmetros e argumentos validados, expressões validada, estrutura de condição validada(no caso if e if else), estrutura de repetição validada, printfs e scanfs validados, foi

introduzida a criação de escopo no código, instruções como return, break, operador ternário, impressão do número total de erros e de cada erro e sua respetiva linha a cor vermelha, mensagem de sucesso, quando assim acontece, e etc.

## Gramática do C utilizada (Mini)

- **Antes de organizar (gramática inicial)**

✓ <Program> ::= <Decl_List>

✓ <Decl_List> ::= <Decl_List> <Decl> | <Decl>

✓ <Decl> ::= <Var_Decl> | <Fun_Decl>

✓ <Var_Decl> ::= <Type_Spec> ID ";" | <Type_Spec> ID "[" "]" ";"

✓ <Type_Spec> ::= "void" | "float" | "char" | "int" | "double"

✓ <Fun_Decl> ::= <Type_Spec> ID "(" <Paramas> ")" <Com_Stmt>

✓ <Paramas> ::= <Param_List> | "void"

✓ <Param_List> ::= <Param_List> "," <Param> | <Param>

✓ <Param> ::= <Type_Spec> ID | <Type_Spec> ID "[" "]"

✓ <Stmt_List> ::= <Stmt_List> <Stmt> | vazio

- ✓ <Stmt> ::= <Exp_Stmt> | <Com_Stmt> | <If_Stmt> | <While_Stmt> | <Return_Stmt> | <Break_Stmt>
- ✓ <Exp_Stmt> ::= <Exp> “;” | “;”
- ✓ <While_Stmt> ::= “While” “(“ <Exp> “)” <Stmt>
- ✓ <Com_Stmt> ::= “{“ <Local_Decls> <Stmt_List> “}”
- ✓ <Local_Decls> ::= <Local_Decls> <Local_Decl> | vazio
- ✓ <Local_Decl> ::= <Type_Spec> ID “;” | <Type_Spec> ID “[” “]” “;”
- ✓ <If_Stmt> ::= “if” “(“ <Exp> “)” <Stmt> | “if” “(“ <Exp> “)” <Stmt> “else” <Stmt>
- ✓ <Return_Stmt> ::= “return” “;” | “return” <Exp> “;”
- ✓ <Exp> ::= ID “=” <Exp> | ID “[“ <Exp> “]” = <Exp> | <Exp> “||” <Exp> | <Exp> “=” <Exp> | <Exp> “==” <Exp> | <Exp> “<=” <Exp> | <Exp> “<” <Exp> | <Exp> “>=” <Exp> | <Exp> “&&” <Exp> | <Exp> “+” <Exp> | <Exp> “-” <Exp> | <Exp> “*” <Exp> | <Exp> “/” <Exp> | <Exp> “%” <Exp> | “!” <Exp> | “-” <Exp> | “+” <Exp> | “(“ <Exp> “)” | ID | ID “(“ <Args> “)” | ID “.” “size” | BOOL_LIT | INT_LIT | FLOAT_LIT | CHAR_LIT | “new” <Type_Spec> “[“ <Exp> “]”

- ✓ <Arg_List> ::= <Arg_List> “,” <Exp> | <Exp>

- ✓ <Arg> ::= <Arg_List> | vazio

- ➢ **Depois de organizar (gramática final)**

Como o método utilizado para o desenvolvimento do analisador sintático é top-down (recursivo descendente sem retrocesso(backtracking)), precisei aplicar algumas transformações à gramática, eliminando a ambiguidade e recursividade à esquerda nas regras da gramática acima, bem como fazer derivação de algumas regras para que me fosse fácil implementar a lógica toda no código. Asseguir é listado a gramática já organizada que foi utilizada para a criação do analisador sintático:

- ✓ <Program> ::= <Decl_List>

- ✓ <Decl_List> ::= <Decl> <Decl_List1>

- ✓ <Decl_List1> ::= <Decl> <Decl_List1> | vazio

- ✓ <Decl> ::= <Type_Spec> ID <Decl1>

- ✓ <Decl1> ::= “;” | “[”<Exp> “]” “;” | “(” <Paramas> “)” <Com_Stmt0> | “=” <Exp_Stmt>

- ✓ <Type_Spec> ::= “void” | “float” | “char” | “int” | “double” | “”

- ✓ <Paramas> ::= <Param_List> | “void”

- ✓ <Param_List> ::= <Param> <Param_List1>
- ✓ <Param_List1> ::= “,” <Param> <Param_List1> | vazio
- ✓ <Param> ::= <Type_Spec> ID <Param1>
- ✓ <Param1> ::= vazio | “[” “]”
- ✓ <Stmt_List> ::= <Stmt_List1>
- ✓ <Stmt_List1> ::= <Stmt> <Stmt_List1> | vazio
- ✓ <Stmt> ::= <Exp_Stmt>|<Com_Stmt>|<If_Stmt>|<While_Stmt> |<Do_While_Stmt> |<Return_Stmt> |<Break_Stmt> |<Printf_Stmt> |<Scanf_Stmt> |<For_Stmt>
- ✓ <Exp_Stmt> ::= <Exp> “;” | “;”
- ✓ <While_Stmt> ::= “While” “(“ <Exp> “)” <Stmt>
- ✓ <Do_While_Stmt> ::= “do” <Stmt> “while” “(“ <Exp> “)” “;”
- ✓ Com_Stmt0> ::= “;“ | <Com_Stmt>
- ✓ Com_Stmt> ::= “{“ <Local_Decls> <Stmt_List> “}”
- ✓ <For_Stmt> ::= “for” “(“ <Exp> “;” <Exp> “;” <Exp> “)” <Stmt>

- ✓ <Break_Stmt> ::= “break” “;”
- ✓ <Printf_Stmt> ::= “printf” “(“ Lit_String <IDS>
- ✓ <IDS> ::= “,” ID <IDS> |”)” “;”
- ✓ <Scanf_Stmt> ::= “Scanf” “(“ <Lit_String_Spec> <FF>
- ✓ <FF> ::= “,” “&” ID <FF1>
- ✓ <FF1> ::= “)” “;” | <FF>
- ✓ <Lit_String_Spec> ::= “%d” | “%i” | “%f” | “%lf” | “&c” | “&s”
- ✓ <Local_Decls> ::= <Local_Decls1>
- ✓ <Local_Decls1> ::= <Local_Decl> <Local_Decls1> | vazio
- ✓ <Local_Decl> ::= <Type_Spec> ID <Local_Decl1>
- ✓ <Local_Decl1> ::= “;” | “[” <Exp> “]” “;” | “=” <Exp_Stmt>
- ✓ <If_Stmt> ::= “if” “(“ <Exp> “)” <Stmt> <If_Stmt1>
- ✓ <If_Stmt1> ::= vazio | “else” <Stmt>
- ✓ <Return_Stmt> ::= “return” <Return_Stmt1>
- ✓ <Return_Stmt1> ::= “;” | <Exp> “;”

- ✓ <Arg_List> ::= <Exp> <Arg_List1>
- ✓ <Arg_List1> ::= “,” <Exp> <Arg_List1> | vazio
- ✓ <Arg> ::= <Arg_List> | vazio
- ✓ <Exp> ::= ID <Exp1><Exp3> | “!” <Exp><Exp3> | “-” <Exp><Exp3> | “+” <Exp><Exp3> | INT_LIT<Exp3> | FLOAT_LIT<Exp3> | CHAR_LIT<Exp3> | double<Exp3>
- ✓ <Exp1> :: = “[“ <Exp> “]” <Exp2> | “.” “sizeof” “(” <Exp> “)” | “(” <Args> “)” | vazio
- ✓ <Exp2> ::= “=” <Exp> | vazio
- ✓ <Exp3> ::= “||” <Exp><Exp3> | “=” <Exp><Exp3> |“==” <Exp><Exp3> |“<=” <Exp><Exp3> |“<” <Exp><Exp3> |“>” <Exp><Exp3> |“>=” <Exp><Exp3> |“&&” <Exp><Exp3> |“-” <Exp><Exp3> | “*” <Exp><Exp3> |“/” <Exp><Exp3> |“&” <Exp><Exp3> |“|” <Exp><Exp3> |“+=” <Exp><Exp3> |“-=” <Exp><Exp3> |“*=” <Exp><Exp3> |“/=” <Exp><Exp3> |“++” <Exp><Exp3> |“--”

<Exp><Exp3> | “+” <Exp><Exp3> | “%” <Exp><Exp3> | “?” <Exp><Exp3> | “:”

<Exp><Exp3> | vazio

## Implementação: Decisões técnicas

A implementação foi pensada da seguinte forma: Todos não terminais com exceção de alguns se transformaram em função, o program, se tornou uma função por exemplo, o mesmo aconteceu com todas outras menos com os não terminais como o <Type_Spec> que no fundo é um terminal, o <Lit_String_Spec> que tem a mesma analogia que o <Type_Spec> é também um terminal na verdade, porque eles só assumem um valor em cada iteração, o tipo ou int, ou float, ou é double, char ou void. Alguns não deixaram de existir no código, por decisão técnica como foi o caso do <Local_Decls>, outros não terminais deixaram de existir por eu ter feito a derivação como foi o caso do <Var_Decl> e <Fun_Decl> apartir do terminal <Decl>.

Quanto aos terminais, o tratamento foi consumir sempre que necessário para verificar se o token consumido é o esperado para validar, no caso de não ser o token esperado é registado o erro num arraylist criado na estrutura TiPoLex, uma estrutura que basicamente representa a tabela de símbolos, e no final apresentar a lista de erros e o total de erros. Nessa lista de erros, também constam os erros léxicos para além dos sintáticos.

Todas as funções são do tipo void, excepto a função program que retorna o arrayList da tabela de símbolos atualizada com os campos tipo de dados, variável de atribuição e o tipo da variável de atribuição. Essa foi uma decisão técnica.

Durante a implementou e atráves de alguns estudos e experiência com outros compiladores, notou-se que em C um array(vetor) global assim como parametros de vetores podem não ser definido um tamanho, simplesmente abrimos colchetes. Mas um array definido localmente deve obrigatoriamente ter o tamanho definido, daí a razão de se adicionar o <Exp> entre colchetes na declaração de um vetor, na gramática atualizada.

## ✓ ANALISADOR SEMÂNTICO

a) Verificar se a variável usada foi declarada

   Criei um método que utiliza uma lista através de arrayList das variáveis declaradas num determinado escopo, e depois o processo foi verificar as variáveis que têm o campo tipo vazio, então nunca foi declarado, todas variáveis declaradas e usadas, têm o campo tipo preenchido com int, char, float ou double.

b) Compatibilidade de tipos, ou seja, uma variável do tipo inteiro

   Aqui a ideia foi por exemplo impedir que um valor do tipo int recebe string ou float. Para que isso funcionasse a lógica foi criar um método que através de um loop percorre os elementos da tabela de símbolos mas apenas os de token tk_numInt, tk_numReal e tk_char, para garantir que as operações aritméticas e relacionais e de atribuição ocorressem apenas entre tipos compativeis, por exemplo os tipos compativeis em c são (int, char) e (int, float, double), essa lógicas aplicam-se tantos aos identificadores variáveis como para as funções e assim como para os literais.

c) Uma variável não deve ser declarada mais de uma vez no mesmo escopo

Aqui a lógica baseou-se na lista de variáveis declaradas sem repetição e depois fiz a comparação com a lista de tokens para verificar se ocorre apenas uma vez (certo) ou mais de uma vez (errado).

d) Verificar as expressões booleanas: (IF, WHILE, FOR, DO-WHILE)

Crie um método que verifica se dentro de parenteses de um if, while ou for contem alguma expressão, senão contém ele mostra erro, apliquei também a verificação no do while para garantir que tenha pelomenos uma expressão entre do e o while, outras verificações foi basicamente verificar se as operações aritméticas e relacionais feita dentro dos parentêses acontece de forma correcta.

e) Verificar compatibilidade de tipos nos parametros da declaração da (FUNÇÃO ou MÉTODO) com os argumentos da chamada da (FUNÇÃO ou MÉTODO);

Criei um método que itera na forma n*n a lista de tokens e verifica apenas as funções declaradas e as suas chamadas, primeiros conta os parametros de cada função e depois os argumentos passados nela na sua chamada, se forem diferentes, mostra erro, se forem iguais então parte para comparação TipoArgumento-TipoParâmetro e conta a quantidade de vezes que a comparação der verdadeira, para comparar com a quantidade de parametros, se for igual então o código é compilado, se não mostra erro.

f) Verificar os valores de saídas (entradas) do printf e scanf com os códigos (%d, %f)

Neste caso crie duas funções uma para verificar os scanfs e outra para os printfs, mais por questão de código limpo, senão a lógica não mudou

muito entre elas. A lógica foi a seguinte: Verificar se na string existe um especificador de formato, no caso %d ou %f, e também verificar se existe identificadores após a vírgula(este é mais para o printf, o scanf já é validado no parser para ele tenha sempre um id ou mais após a vírgula). Na função validaPrintf, se não existit nem especificador nem identificador algum na instrução printf, então o código é compilado, claro desde que tenha uma string para se imprimir. Se existir um e não o outro, o programa mostra erro, se existir os dois, ele faz o que acontece na lógica da compatibilidade dos parâmetros e argumentos e valida. No método validaScanf, praticamente acontece a lógica da compatibilidade dos parâmetros e argumentos.

g) Fazer também a verificação do identificador principal (main) se foi bem

declarada

Aqui criei um método tal como nos casos anteriores para verificar se o main tem o tipo de retorno e se o tipo é inteiro e se o main não recebe parâmetros. Uma vez que segundo alguns estudos, percebi que o main não deve nunca receber parâmetros, excepto vazio e void. Portanto, foi basicamente isso que implementei.

Por fim uma nota vai para as funções, variáveis e a tabela de símbolos. Validei também através de um método a questão das funções declaradas mais de uma vez no programa. Quanto as variáveis, elas podem ter o mesmo nome que as funções, é uma boa curiosidade que descobri. Sobre a tabela de símbolos, importa aqui dizer que tive que mexer um pouco na sua estrutura para acrescentar o campo função, que gurda o nome_ da _função() que cada token pertence, se for o caso.