UNISAGRADO

ANA CAROLINA DE OLIVEIRA

DIOGO BISSOLI MORENO

IAN REPKER MALAVAZI

JOÃO RENATO REPKER VOROS

RELATÓRIO COMPILADORES

BAURU
2021

ANA CAROLINA DE OLIVEIRA

DIOGO BISSOLI MORENO

IAN REPKER MALAVAZI

JOÃO RENATO REPKER VOROS

RELATÓRIO COMPILADORES

Projeto de Pesquisa apresentado à Unisagrado como parte integrante da disciplina de Compiladores, sob orientação da/o Prof. Patrick Pedreira.

BAURU
2021

SUMÁRIO

1. INTRODUÇÃO

O presente documento visa documentar o projeto desenvolvido durante a disciplina de Compiladores, ministrada no segundo semestre de 2022 pelo professor Patrick Pedreira. O projeto em questão consiste no desenvolvimento de uma linguagem de programação e seu “compilador”.

O compilador, no caso, realiza apenas as duas primeiras etapas das seis presentes no processo de compilação real, sendo essas duas a análise léxica e a análise sintática.

A análise léxica foi realizada por autômatos. Foram definidas as características da linguagem, foram desenvolvidos autômatos e depois estes foram implementados formando o analisador léxico.

A segunda fase da implementação foi realizada desenvolvendo um analisador sintático recursivo para “compilar” o código da linguagem idealizada.

2. LINGUAGEM DESENVOLVIDA

A linguagem desenvolvida foi feita seguindo os padrões da maioria das linguagens de programação convencionais de alto nível modernas. Existem palavras para definir o começo e final do programa, declaração de variáveis (de forma estática), atribuição de valores com um ou dois operandos, operações matemáticas de adição e subtração e comandos para leitura e escrita.

As palavras de início e final de programa são dadas, respectivamente por: “COMECO” e “FINAL”.

Declarar uma variável segue o seguinte padrão de escrita: <tipo> <id> = <valor>. No caso, existem apenas dois tipos aceitos pela linguagem: os tipos numéricos inteiros e reais. Eles são definidos pelas palavras reservadas: “INT” e “FLOAT”. Um número real é dado com a separação de casas decimais por uma vírgula, como o padrão brasileiro (Ex: 3,14). Os identificadores de variáveis podem conter qualquer caractere de a até z (minúsculos) e qualquer número de 0 a 9, desde que não estejam no primeiro caractere do identificador.

As operações de atribuição podem ser definidas por: <id> = <operação>. Uma <operação>, por sua vez, pode ser: <operação> = <id> | <id> <sinal> <id>. E um <sinal> representa ou uma adição (<+>) ou uma subtração (<->).

Os comandos de escrita e leitura são: “ESCREVER” e “LER”. Suas estruturas são parecidas, sendo respectivamente: ESCREVER <id> e LER <id>.

Por fim, o sinal de finalização de comando foi definido como um ponto (“ . ”).

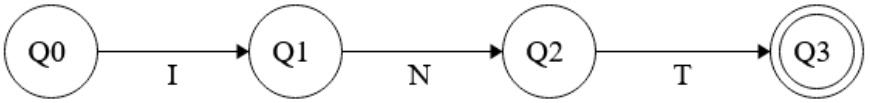
3. AUTÔMATOS

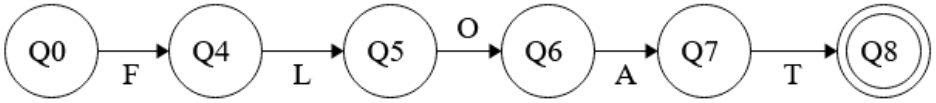
Para reconhecer os tokens da linguagem foram desenvolvidos autômatos. Os tokens da linguagem pode ser representados pela tabela abaixo:

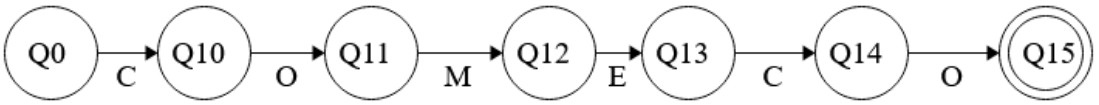
CÓDIGO	TOKEN
COMECO	ini
FINAL	fim
ESCREVER	esc
LER	ler
INT	int
FLOAT	float
=	=
+	+
-	-
[número inteiro]	num
[número real]	numr
[nome de variável]	id

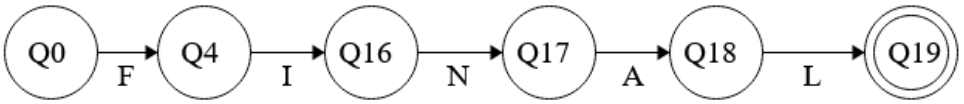
Os autómatos podem ser vistos nos desenhos que se seguem. Vale ressaltar que, embora estejam separados no presente documento, no código eles foram implementados de forma única, ou seja, existe na prática apenas um autômato que é a “soma” dos apresentados a seguir.

Reconhecimento das palavras reservadas INT, FLOAT, COMECO e FINAL

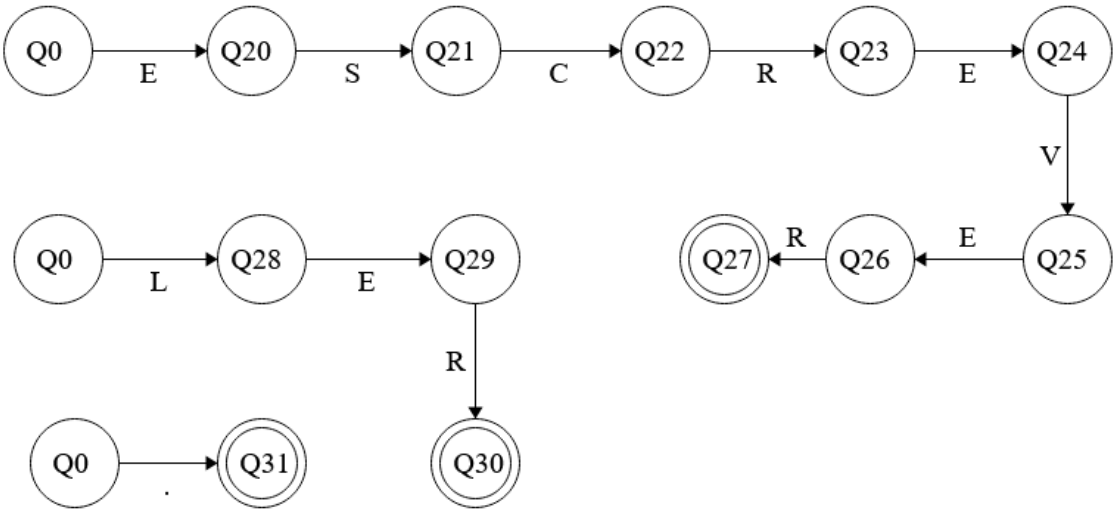




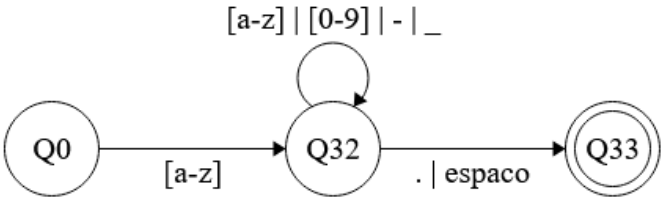


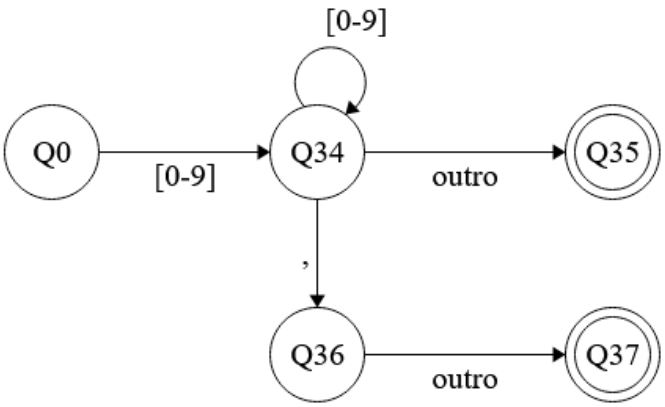


Reconhecimento dos comandos ESCREVER e LER e o símbolo de finalização de programa.



Autômatos para o reconhecimento de identificadores de variáveis, números inteiros e reais e sinais de operações matemáticas (somas, subtrações e símbolos de atribuição)





Ressaltando novamente que os autômatos foram separados nos desenhos a fim de simplificar a visualização no presente documento, porém, no código do projeto todos eles são unificados.

Quando um token é reconhecido, o estado do autômato é resetado para o estado inicial (Q0). O estado de morte foi tratado da seguinte maneira: caso não haja caminho definido para aquele caractere no estado atual, o autômato entra automaticamente em estado de morte e a execução passa para a próxima linha do código. Dessa forma é possível detectar mais erros por vez, não sendo uma regra ter que “corrigir o erro de agora”, para poder corrigir os demais.

Há também estados específicos onde ocorre uma “regressão”, ou seja, o ponteiro do autômato é retrocedido uma posição para atrás. Isso ocorre pois há casos onde somente se sabe que a classificação terminou quando outro caractere é lido. Sem esse retrocesso do ponteiro esse “caractere de gatilho” é perdido e a leitura do próximo token é prejudicada.

A seguir uma tabela com todos os estados finais do autômato e suas classes:

ESTADO	CLASSIFICAÇÃO
Q3	INT (int)
Q8	FLOAT (float)
Q15	INICIO (ini)
Q19	FINAL (fim)
Q27	ESCREVER (esc)
Q30	LER (ler)
Q31	símbolo de finalização de comando (.)
Q33	identificador de variável (id)
Q35	número inteiro (num)
Q37	número real (numr)
Q38	símbolo de atribuição (=)
Q39	símbolo de adição (+)
Q40	símbolo de subtração (-)

4. IMPLEMENTAÇÃO E SAÍDA DO ANALISADOR LÉXICO

Todos os códigos foram implementados na linguagem de programação Python. A escolha dessa linguagem se deu pelo conhecimento prévio dos membros do grupo e pelas suas facilidades de leitura de arquivos.

O analisador léxico foi feito implementando os autômatos anteriormente apresentados, usando para isso a estrutura de dados dicionário, nativa da linguagem Python. Também como já explicado, há apenas um código que contém todos os autômatos.

A saída do analisador léxico é um vetor de tuplas. Cada tupla contém a classe do token analisado e seu “valor”, ou seja, qual é o texto que representa a classe no código escrito pelo usuário.

Sendo assim, o código abaixo teria o seguinte vetor de tokens:

```
INT a = 5.

```

```
[[('int', 'INT'), ('id', 'a'), ('=', '='), ('num', '5'), ('.', '.')]]
```

O processamento é realizado por linha de arquivo, isto é, se analisa todos os tokens de uma linha e depois adiciona-se a lista de tokens classificados em uma outra lista que contém as demais linhas analisadas. Esse processo se repete até que todas as linhas estejam analisadas. Um código completo, ficaria com um formato semelhante a este:

```
INT a = 5.
INT teste = 2.
COMECO
    a = teste.
    ESCREVER a.
FINAL
```

```
[
    [('int', 'INT'), ('id', 'a'), ('=', '='), ('num', '5'), ('.', '.')],
    [('int', 'INT'), ('id', 'teste'), ('=', '='), ('num', '2'), ('.', '.')],
    [('ini', 'COMECO')],
    [('id', 'a'), ('=', '='), ('id', 'teste'), ('.', '.')],
    [('esc', 'ESCREVER'), ('id', 'a'), ('.', '.')],
    [('fim', 'FINAL')]
]
```

A partir dessa lista de tokens, o analisador sintático “compila” e executa as operações solicitadas pelo código escrito.

5. IMPLEMENTAÇÃO E SAÍDA DO ANALISADOR SINTÁTICO

O analisador sintático foi implementado de forma recursiva, onde uma “função mãe”, chama “funções filhas” e estas chamam mais funções e assim sucessivamente até que a linha seja completamente analisada. Também é nessa etapa que se tem a maior parte das mensagens de erros.

Para executar as operações exigidas pelo projeto, é simulada uma tabela de símbolos. Essa tabela é um dicionário Python onde cada item é dado por: uma chave, o identificador da variável; um tipo, se a variável em questão é INT ou

FLOAT; e o seu valor propriamente dito. A tabela é montada durante a leitura do cabeçalho de variáveis.

Quando uma operação de escrita é encontrada pelo analisador, ela dispara a função *build-in* do Python *print()* imprimindo o que for passado para ele. Ocorre um processo semelhante para a operação de leitura, porém obviamente chamando a função *input()*.

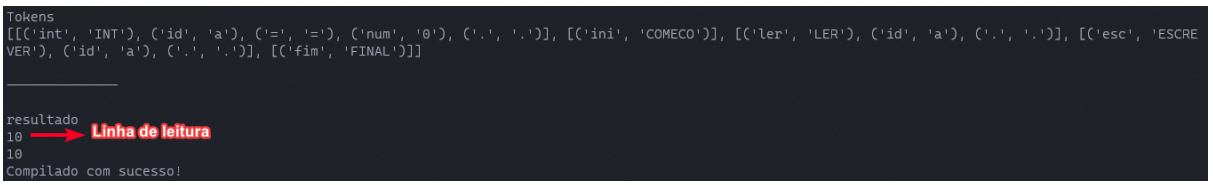
A tabela de símbolos é atualizada toda vez que ocorre uma atribuição ou comando de leitura. Caso o código tente acessar uma variável que não esteja registrada, um erro é exibido e o processo de compilação para.

Também é possível fazer atribuições com um ou dois operadores realizando operações de adição ou de subtração. A seguir alguns exemplos de códigos da linguagem, seguidos de seus respectivos resultados. Para fins de explanação, também serão mostrados os tokens gerados.

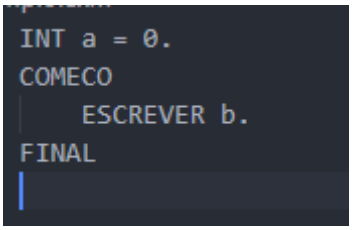
```
INT a = 5.
INT teste = 2.
COMECO
    a = teste.
    ESCREVER a.
FINAL
```

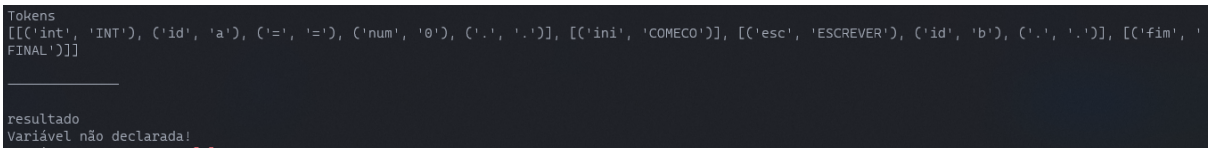
```
Tokens
[[('int', 'INT'), ('id', 'a'), ('=', '='), ('num', '5'), ('.', '.')], [('int', 'INT'), ('id', 'teste'), ('=', '='), ('num', '2'), ('.', '.')], [('ini
', 'COMECO')], [('id', 'a'), ('=', '='), ('id', 'teste'), ('.', '.')], [('esc', 'ESCREVER'), ('id', 'a'), ('.', '.')], [('fim', 'FINAL')]]

____________


resultado
2
Compilado com sucesso!
compiladores on master [!]
>>>
```

```
INT a = 0.
COMECO
    LER a.
    ESCREVER a.
FINAL
```



Tentando acessar uma variável que não foi declarada.





6. CONCLUSÃO

O projeto foi de grande proveito para os integrantes do grupo, auxiliando no entendimento dos conteúdos vistos durante a disciplina de Compiladores. O projeto agiu, mesmo que tenha sido de forma simplificada, como uma aplicação prática das teorias ministradas, simplificando o entendimento das mesmas.