Relatório: Compilador Basic

Thiago Cordeiro da Fonseca 8993080

0. Infraestrutura

Para o desenvolvimento do compilador, utilizou-se de uma abordagem orientada a eventos implementada em Python. A escolha da linguagem se deu pelo alto nível de abstração da mesma e a grande quantidade de estruturas de dados já implementadas.

Para o motor de eventos e os eventos, se criou o arquivo *base.py.* Neste arquivo encontram-se as classes *MotorDeEventos* e *Evento*, que servem como classes pai para todos os eventos e motores de eventos desse projeto.

A classe *MotorDeEventos* possui um atributo *lista*, que contém a lista de eventos a serem executados, e um outro atributo *rotinasDeTratamento*, que contém uma tabela *de-para* que relaciona um evento com a sua respectiva *rotinaDeTratamento*. Existem também o método *iterar()* que executa a rotina correspondente ao evento do topo da lista, caso o atributo *tempo* do evento corresponda ao tempo de simulação. Há também o método *logar()* que imprime o estado atual do motor de eventos (tempo de simulação, qual o motor de eventos e uma mensagem que é passada como argumento da função).

A classe *Evento* possui somente dois atributos: *tempo*, que indica o tempo em que o evento tem que ser executado; e *tarefa* que indica a qual tarefa o evento se relaciona.

Também há o arquivo *main.py* que deve ser executado para se iniciar a compilação. Nele, são declarados todos os motores de eventos a serem utilizados e o arquivo que deverá ser compilado. Então, é chamado a função principal *loopDeSimulacao()* que recebe todos os motores de eventos declarados, cria uma variável *t*, que representa o tempo de simulação e chama o método *iterar()* de todos os motores de eventos passados como argumento.

1. Analisador Léxico

O analisador léxico é composto por uma sequência de motores de eventos encadeados. Se encapsularmos o analisador Léxico como um motor de eventos só, ele recebe um arquivo e retorna eventos de *tokens* que serão utilizados pelo analisador léxico e semântico. Aqui, serão descritos como cada um desses motores foi implementado e os seus respectivos testes.

a. Sistema de Arquivos

O sistema de arquivos se encontra implementado no arquivo *SistemaDeArquivos.py*, que possui a implementação da classe *SistemaDeArquivos*, filha de *MotorDeEventos.* Também possui um ponteiro para o motor de eventos do Filtro ASCII e um atributo adicional chamado *arquivoAtual,* um ponteiro para o último arquivo aberto. Esse motor pode receber três tipos de eventos:

- Evento *Arquivo*: Chama a rotina de tratamento *abrirArquivo()* , que atualiza o atributo *arquivoAtual* com o ponteiro para o arquivo aberto a partir do *path* passado pelo próprio evento *Arquivo*. Insere um evento *ProximaLinha* no próprio Sistema de Arquivos.

- Evento *ProximaLinha*: Chama a rotina de tratamento *lerProximaLinha()*, que lê o contéudo da próxima linha do arquivo gravado em *arquivoAtual* e, se a linha não for vazia, envia um evento *Linha* para o Filtro ASCII e insere um evento *ProximaLinha* no próprio Sistema de Arquivos. Se a linha for vazia, envia um evento *FimDeArquivo* para o Filtro ASCII e insere um evento *FecharArquivo* no próprio Sistema de Arquivos.

- Evento *FecharArquivo*: Chama a rotina de tratamento *fecharArquivo(),* que simplesmente fecha o arquivo apontado por *arquivoAtual*.

Para se testar, foi usado como entrada o arquivo que se encontra em *./testes/testeArquivos.txt*.

```
0 LET X1 = 10
10 REM comentario
20 END
```

Espera-se que o sistema de arquivos consiga dividir o arquivo em suas linhas e fechá-lo com êxito. Essa foi a saída obtida:

```
Tempo: 0
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Arquivo ./testes/testeArquivos.txt aberto
```

Tempo: 1
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Linha Lida: 0 LET X1 = 10

Tempo: 2
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Linha Lida: 10 REM comentario

Tempo: 2
Maquina: <class 'FiltroAscii.FiltroAscii'>
Log: A linha recebida eh: 0 LET X1 = 10

Tempo: 3
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Linha Lida: 20 END

Tempo: 3
Maquina: <class 'FiltroAscii.FiltroAscii'>
Log: A linha recebida eh: 10 REM comentario

Tempo: 4
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Linha Lida:

Tempo: 4
Maquina: <class 'FiltroAscii.FiltroAscii'>
Log: A linha recebida eh: 20 END

Tempo: 5
Maquina: <class 'SistemaDeArquivos.SistemaDeArquivos'>
Log: Arquivo fechado

Pode-se ver que o resultado foi bem-sucedido. Também se observa a defasagem de 1 tempo entre uma linha ser processada no *SistemaDeArquivos* para depois aparecer no *FiltroASCII*.

b. Filtro ASCII

O Filtro ASCII se encontra implementado no arquivo *FiltroAscii.py*, que possui a implementação da classe *FiltroAscii*, filha de *MotorDeEventos*. Possui somente um atributo adicional que a referência para o Categorizador ASCII. Esse motor tem como função receber linhas e dividir cada uma em seus

caracteres ASCII e classificá-los. Para auxiliar, foi criado um arquivo adicional *constants.py* que contém:

- Uma lista com todos os caracteres minúsculos
- Uma lista com todos os caracteres maiúsculos
- Uma lista com todos os dígitos
- Uma lista com os caracteres especiais da linguagem
- Uma lista com palavras predefinidas e reservadas da linguagem

Esse motor pode receber dois tipos de eventos:

- Evento *Linha:* Chama a rotina *lerLinha()*. A rotina percorre caracter a caracter da linha e gera um evento *AsciiUtil*, se o caracter pertencer a uma das 4 primeiras listas, um evento *AsciiDescartavel* se o caracter for um espaço e um evento *AsciiControle* se o caracter for alguma coisa coisa (como um caracter *\n*). Esses eventos são enviados para o Categorizador ASCII.

- Evento *FimDeArquivo:* Chama a rotina *finalizarArquivo(),* que não faz nada.

Para se testar, se criou o arquivo *testes/testeFiltroAscii.txt*.

Conteudo do arquivo:
0 REM a=

A saída obtida foi:
Tempo: 2
Maquina: <class 'FiltroAscii.FiltroAscii'>
Log: A linha recebida eh: 0 REM a=

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:0 do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi: do tipo <class 'eventos.AsciiDescartavel'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:R do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:E do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>

Log: O caracter recebido foi:M do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi: do tipo <class 'eventos.AsciiDescartavel'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:a do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:= do tipo <class 'eventos.AsciiUtil'>

Tempo: 3
Maquina: <class 'CategorizadorAscii.CategorizadorAscii'>
Log: O caracter recebido foi:
 do tipo <class 'eventos.AsciiControle'>

Pode-se ver que o Categorizador ASCII está recebendo os caracteres corretamente classificados.

c. Categorizador ASCII

O Categorizador ASCII se encontra implementado no arquivo *CategorizadorAscii.py*, que possui a implementação da classe *CategorizadorAscii*, filha de *MotorDeEventos*. Possui somente um atributo adicional que é a referência para o Categorizador Léxico. Pode receber 3 tipos de eventos:

- Evento *AsciiUtil:* Chama a rotina *categorizarAsciiUtil(),* que pode inserir um evento *Digito* no Categorizador Lexico caso o caracter seja um digito; um evento *Especial* caso o caracter esteja na lista de caracteres especiais; ou um evento *Letra*, caso contrário.

- Evento *AsciiControle:* Chama a rotina *categorizarAsciiControle()*, que insere um evento *Controle* no Categorizador Léxico.

- Evento *AsciiDescartavel:* Chama a rotina *categorizarAsciiDescartavel(),* que insere um evento *Delimitador* no Categorizador Léxico.

Foi-se testado com o mesmo arquivo usado para o teste do Filtro ASCII. O resultado obtido foi:

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:0 do tipo <class 'eventos.Digito'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi: do tipo <class 'eventos.Delimitador'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:R do tipo <class 'eventos.Letra'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:E do tipo <class 'eventos.Letra'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:M do tipo <class 'eventos.Letra'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi: do tipo <class 'eventos.Delimitador'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:a do tipo <class 'eventos.Letra'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:= do tipo <class 'eventos.Especial'>

Tempo: 4
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: O caracter recebido foi:
 do tipo <class 'eventos.Controle'>

Pode-se ver que o CategorizadorLexico recebeu todos os caracteres já devidamente categorizados.

d. Categorizador Léxico

O categorizador léxico recebe caracteres ASCII já categorizados e deve agrupá-los em *Tokens*. Para isso, foi implementado um autômato finito determinístico, aliado de um acumulador que vai guardando quais os caracteres que já foram utilizados.

O Categorizador Léxico se encontra implementado no arquivo *CategorizadorLexico.py*, que possui a implementação da classe *CategorizadorLexico*, filha de *MotorDeEventos*. Ela possui um atributo *estadoAtual* que grava em qual estado do autômato o categorizador se encontra; um atributo *acumulador* que armazena os caracteres até um estado de aceitação ser encontrado; um atributo *automato* que possui o autômato em questão; e um último atributo que é um ponteiro para o Recategorizador Léxico.

Os eventos que o categorizador pode receber são:

- Evento *Delimitador*
- Evento *Digito*
- Evento *Letra*
- Evento *Especial*
- Evento *Controle*

Todos esses eventos chamam a mesma rotina *rodarAutomato*. Essa rotina verifica em qual estado o autômato se encontra e qual o próximo estado dada a entrada recebida. Os eventos de saída para o recategorizador léxico são:

- Evento *TokenId:* Regido pela expressão regular Letra{Letra}{Digito}
- Evento *TokenNumero:* Regido pela expressão regular Digito{Digito}
- Evento *TokenEspecial*: Regido pela expressão Especial{Especial}
- Evento *TokenLinha*: Regido pela expressão \n

Como pode ser visto, da maneira que está descrito, construções do tipo DigitoLetra não são permitidas. Então foram feitos dois testes. Um deles deve categorizar os tokens corretamente, sem acusar erros. Ele se encontra em *testes/testeLexico1.txt.*

Conteudo do Arquivo:

0 LET X1 = 10
20 GOTO 40
30 GO TO 40
40 END

O recategorizador léxico deve receber os seguintes tokens, na ordem: *TokenNumero, TokenId, TokenId, TokenEspecial, TokenNumero, Token Linha, TokenNumero, TokenId, TokenNumero, TokenLinha, TokenNumero, TokenId, TokenId, TokenNumero, TokenLinha, TokenNumero, TokenId, TokenLinha*. A saída obtida foi:

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 0

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh LET

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh X1

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenEspecial'> e seu conteudo eh =

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 10

Tempo: 5
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 6
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 20

Tempo: 6
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh GOTO

Tempo: 6
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 6
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 7
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 30

Tempo: 7
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh GO

Tempo: 7
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh TO

Tempo: 7
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 7
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 8
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 8
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh END

Tempo: 8
Maquina: <class 'RecategorizadorLexico.RecategorizadorLexico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Que nem o esperado. Agora, para testar que o categorizar acusando uma sequência inválida, executou-se o teste com o arquivo *teste/testeLexico2.txt.*

Conteudo:
00 LET Y = 1X

A saída recebida foi:
Tempo: 5
Maquina: <class 'CategorizadorLexico.CategorizadorLexico'>
Log: Token "1X" invalido

O resultado foi o esperado, junto com a interrupção do compilador.

e. Recategorizador Léxico

O Recategorizador Léxico se encontra implementado no arquivo Re*categorizadorLexico.py*, que possui a implementação da classe *RecategorizadorLexico*, filha de *MotorDeEventos*. Ela funciona de maneira análoga ao categorizador léxico, com exceção de que ela deve agrupar certos *TokensIds* em eventos *Tokens Reservados*. Todos os outros tokens são repassados diretamente para o *CategorizadorSintatico*. Os *TokenIDs* possuem um autômato que checa:

- Se for uma palavra reservada diferente de *DEF e GO,* transforma em *TokenReservado* e passara para o *CategorizadorSintatico*.
- Se for uma palavra reservada *DEF ou GO,* espera o próximo *token*. Se for *FN* ou *TO* respectivamente, envia um *TokenReservado* com valor *DEF FN* ou *GOTO,* respectivamente. Caso contrário envia um *TokenId* com os valores anteriores e repente o processo para o *token* que foi recebido agora.

Para se analisar o funcionamento, foi utilizado mais uma vez o arquivo *teste/testeLexico1.txt.* O resultado obtido foi:

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 0

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh LET

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>

Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh X1

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenEspecial'> e seu conteudo eh =

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 10

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 20

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh GOTO

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 30

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>

Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh GOTO

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Tempo: 9
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 40

Tempo: 9
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh END

Tempo: 9
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Pode-se ver que todos as palavras reservadas se transformaram em *tokenReservado*, inclusive ambos os GOTO, sendo que um, no código-fonte, estava com espaço e o outro sem.

2. Analisador Sintático

Para o analisador sintático, foi implementado um autômato de pilha não-determinístico. Para isso, a classe *CategorizadorSintatico()* possui um atributo adicional *categorizadores*, que é uma lista de categorizadores. Cada categorizador possui os atributos:

- *maquina:* Indica qual a máquina atual em que o categorizador se encontra.
- *estado:* Indica em qual estado da máquina atual o categorizador se encontra.

- *pilha:* Grava qual a máquina e qual estado da máquina o categorizador deve ir caso um *pop* ocorra.
- *histórico:* Lista que grava todas os estados pelo qual a analise passou.

O *categorizadorSintático* funciona da seguinte forma:

- Para todos os categorizadores disponíveis, adiciona-se uma *flag* com valor *True*, indicando que o *token* de entrada ainda precisa ser consumido por aquele categorizador.
- É chamada a função *próximos* que recebe uma lista de categorizadores com *flags*, o evento e o tempo de simulação e deve retornar uma lista de novos categorizadores a serem usados quando o próximo *token* chegar.
- Na função *proximos()*
    - Para cada categorizador*:*
        - Cria-se uma lista auxiliar
        - Caso tenha a *flag* com valor *False:*
            - Adiciona-se ele a lista auxiliar
        - Caso contrário:
            - Verifica-se o estado e máquina do categorizador e consulta-se quais as transições possíveis dada aquela entrada.
            - Para cada transição:
                - Se for uma transição que consome o *token* e o tipo e conteúdo do *token* recebido também coincidem com o tipo e conteúdo do token da transição:
                    - Cria-se um novo categorizador que possui a mesma máquina, mesma pilha, o novo estado que a transição indica e com a flag *False,* já que o token foi consumido. Incluí-se esse categorizador na lista auxiliar.
                - Se for uma transição de estado de aceitação da sub-máquina:
                    - Se a pilha estiver vazia, a análise foi bem-sucedida.
                    - Se a pilha não estiver vazia, cria-se um novo categorizador cujo estado e máquina são os do topo da pilha. Retira-se o elemento do topo da pilha. Também se mantém a flag *True* e é incluído na lista auxiliar.
                - Se for uma transição que aponta para uma nova submáquina:
                    - Cria-se um novo categorizador que possui a máquina que a transição indica, no estado 1 e empilha-se o estado e

máquina de destino quando essa sub-máquina for resolvida. Também se mantém a flag *True* e é incluído na lista auxliar.
    - Se todos os categorizadores dessa lista auxiliar possuem a flag *False*, retorna-se essa lista.
    - Caso contrário, a função *proximos()* é chamada recursivamente com essa lista.
- Verifica-se se essa lista recebida é vazia. Se for, indica-se que houve erro na análise sintática.

Para se testar seu funcionamento, implementou-se, incialmente somente as máquinas *Program, BS* e *Remark.* As máquinas se encontram no arquivo *machines.py*. Para se testar o caso em que tudo está certo, utilizou-se o arquivo *teste/testeSinstatico1.txt*. Para se montar estes, e outros autômatos, foi-se utilizado o algoritmo visto em aula para transformar gramáticas na notação de Wirth em autômatos.

Conteúdo do arquivo:

0 REM 231231
14 REM Tx31
20 END

Dado o tamanho dos logs intermediários, aqui só serão mostrados os pontos mais relevantes. O log completo pode ser visto no Apêndice A.

O analisador sintático acusou êxito. O último log antes de encerrar foi:

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh END
maquina:Program , estado: 4, pilha: [], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('Program', 3), ('Program', 4)]

O histórico indica que o analisador fez o caminho esperado, começando em *Program,* indo para *BS*, indo em *Remark*, saindo de *Remark,* saindo de *BS*, indo pra *BS*, indo pra *Remark,* saindo de *Remark,* saindo de *BS* e por fim chegando em (‘Program’,4), que é o estado final da máquina *Program.*

É também importante mostar o não determinismo que acontece nesse exemplo. Sempre ao receber os *tokens* de número após o fim de um *BS*, ao receber um novo *tokenNumero*, o analisador pode ir para dois lugares: Pode ir para o estado 3 da máquina *Program* (que depois, se receber um *tokenReservado END* vai para o estado

4 e final) e também pode ir para o estado 2 da máquina *BS* (porque todo *BS* começa com um *tokenNumero* também). Isso pode ser visto no seguinte log:

```
Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 14
maquina:BS , estado: 2, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2)]
maquina:Program , estado: 3, pilha: [], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('Program', 3)]
```

No qual pode ser observado que há 2 categorizadores concomitantes, um deles na máquina *BS* e o outro na máquina *Program.* No instante seguinte, somente a primeira máquina continua, pois ao receber um *tokenReservado REM*, a segunda máquina não tem para onde ir.

```
Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh REM
maquina:Remark , estado: 2, pilha: [('Program', 2), ('BS', 3)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2)]
```

Agora, para testar um caso em que o programa quebre. Podemos fazer isso escrevendo errado o comando *REM*. Foi isso que foi feito em *teste/testeSintatico2.txt.*

Conteúdo do arquivo:

```
0 RAM 231231
20 END
```

O resultado obtido foi:

```
Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 0
maquina:BS , estado: 2, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2)]

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh RAM
```

ERRO NA ANALISE SINTÁTICA

Como pode ser visto, o compilador acusou o erro e encerrou a execução.

3. Analisador Semântico e geração de código

O analisador semântico não foi construído em uma classe a parte. Ao invés disso, a geração de código se dá durante a análise sintática, durante as transições. Como, **em geral**, cada sub-máquina gera um bloco de código independente do resto, a lógica genérica para geração de código foi a seguinte:

- Cada *categorizador* possui um atributo adicional chamado *chainCode*, que é uma pilha no qual são gravadas todos os parâmetros necessários para geração do código daquele sub-máquina. Cada elemento no *chainCode* tem um atributo chamado *valueCode* que é responsável por armazenar o código gerado por sub-máquinas chamadas por aquela sub-máquina.
- Sempre que uma transição é feita:
    - Se for uma transição de entrada em uma sub-máquina, empilha-se no *chainCode* os parâmetros necessários para aquela sub-máquina que acabou de ser chamada. Ele será atualizado durante as atualizações do *chainCode*;
    - Se for uma transição que consome um não terminal, é chamada a função correspondente que trata aquele não terminal. Algumas transições desse tipo podem não chamar função alguma;
    - Se for uma transição que encerra aquela sub-máquina, é chamada a função de geração de código daquela sub-máquina que utilizará os parâmetros armazenados em *chainCode* para gerar um código que será anexado ao fim do *valueCode* atual, dará um *pop()* em *chainCode* e colocará o valor de *valueCode* que foi desempilhado no atributo *valueCode* do elemento que agora se encontra no topo do *chainCode*. A exceção a essa regra é no caso da saída dá sub-máquina *BStatement*, na qual o seu *valueCode* é anexado no próprio atributo *code* do categorizador.

Ao fim da análise, é gerado o código objeto final no arquivo *output.txt*. Para se montar o código-objeto, foi-se utilizado o montador desenvolvido na matéria de Sistemas de Programação: https://sites.google.com/view/2017-pcs3216-8993080/p%C3%A1gina-inicial.

Outras estruturas de dados adicionais que foram utilizadas:

- *data:* similar ao atributo *code* do *categorizador*, mas que armazena as pseudo-instruções de reserva de área de memória para as variáveis;
- *forStack:* utilizado para os grupos de comando *for/next* para empilhar qual a última variável que deve ser controlada para o *next* e o *label* de retorno

- *forFlag:* ligado logo após o comando *for* para indicar que o label de retorno do *jump* vai ser o próximo *int* que aparecer no começo da linha seguinte. Ao chegar na linha seguinte, o *int* é gravado no *forStack* e a *flag* é desligada
- *variaveisDeclaradas:* indica quais variáveis foram declaradas no programa. Sempre que algo vai ser adicionado em *data,* antes se checa se aquela variável já foi declarada ou não para não haver conflitos de label.
- *goSubFlag:* similar ao *forFlag*, mas para o par de comandos *Gosub/Return.*
- *tempCount:* contador para indicar o número de variáveis temporárias criadas.

Abaixo, serão explicados como cada comando se traduz para código e seus respectivos testes. Todos os códigos começam a excução no endereço de memória 0x100 e : terminam com um jump para a posição 0x000, na qual existe um halt-machine.

a. BStament

Ao entrar no BStatement, o *chainCode* anexado possui o parâmetro adicional *lineCode*, onde vai ser armazenado o número que inicia a linha atual, e *sub* que indica se estamos em uma subrotina.

Ao se sair de BStatement, na primeira linha se coloca como label *lineCode* e, se *sub* estiver ligado, a primeira linha é só uma reserva de espaço (para o uso da instrução SC da MVN). Caso contrário (e para as outras linhas) o código é o que está no *valueCode* do *chainCode.* Para o BStatement não há nenhum teste em específico pelo fato de ser algo que permeia todos os comandos.

b. Exp

Ao entrar em Exp, o *chainCode* anexado possui os parâmetros:

- *variableList:* lista de variáveis que estão sendo operadas por aquela expressão.
- *operationList:* lista de operações a serem realizadas com as variáveis.
- *returnVariable:* nome da label (variável) onde o valor da expressão deve ser armazenado.

Cada terminal de operação consumido, é adicionado a *operationList* e cada vez que entra-se na submáquina *Eb* uma nova variável temporária é adicionada a *variableList.*

Ao sair de Exp, anexa-se ao *valueCode* LV /0 para se carregar o valor no acumulador. Em seguida, varre-se a lista de operações atrás de multiplicações e divisões para calculá-las primeiro entre aquelas variáveis. Para cada operação e variável adiciona-se a instrução correspondente da operação. Por fim, adiciona-se a instrução para carregar esse valor no espaço de memória correspondente ao *returnVariable* (instrução MM). Então, o *valueCode* sofre

um *pop()* e esse bloco de código é adicionado ao *valueCode* do novo topo. O Exemplo de Exp será mostrado na seção referente ao Assign, já que Exp por si só não monta nenhum código final por conta própria.

c. Eb

Ao entrar em *Eb*, como *Eb* é chamado somente por *Exp,* o primeiro passo é calcular qual a variável de retorno de Eb. Sempre são variáveis temporárias, então cria-se uma variável temporária com o nome T + *tempCount.* Então, incrementa-se o *tempCount.*

Caso *Eb* seja somente uma constante, adiciona-se a instrução LV /+constante ao seu *valueCode.* Caso seja uma variável, adiciona-se a instrução LD + variável. Caso seja uma outra *Exp*, a submáquina *Exp* é chamada e uma nova variável temporária é criada e essa submáquina *Exp,* ao encerrar, escreverá seu código no *valueCode* de *Eb*.

Por fim, ao se sair de *Eb,* anexa-se a instrução MM + *returnVariable* e desempilha-se *valueCode* e o código gerado é anexado ao *valueCode* que agora que se encontra no topo. O teste se encontra, também, na parte referente ao Assign.

d. Print

O comando Print foi adaptado para simplesmente carregar no acumulador o valor que deveria ser impresso. Portanto, ao entrar em *Print,* o seu *chainCode* possui somente um parâmetro adicional que é variável que deve ser impressa.

Ao se passar por uma variável na segunda transição de *Print,* essa variável é armazenada no *chainCode*.

Ao se sair de *Print*, o código LD + *variable* é anexado ao *valueCode*, desempilha-se e esse código é anexado ao *valueCode* do *chainCode* que agora se encontra no topo. O exemplo do print também será mostrando no assign.

e. Assign

Ao se entrar em Assign, o seu *chainCode* só possui um atributo adicional que se refere a variável que está sendo atribuída.

Ao se passar pela transição que verifica a variável, ela é gravada ao *chainCode* e, caso ainda não tenha sido declarada, ela é declarada dentro do atributo *data* do categorizador, tendo como label o próprio nome da variável.

Ao se passar pela transição que espera um *Exp,* a submáquina *Exp* irá escrever no *valueCode* de *Assign* o código gerado por ela.
Ao se sair de assign, por fim, o seu *valueCode* é anexado ao *valueCode* anterior.

Para se testar o funcionamento de *Assign* e todos as outras submáquinas já mostradas anteriormente, se montou o seguinte código (*teste/testeSemantico1.txt)*:

```
00 LET X1 = 2
10 LET X2 = 1 + X1
20 LET X3 = 2 * (X2 + 2 + 1)
30 PRINT X3
40 END
```

Espera-se que X3 possua o valor 12 (ou C em hexa) ao final que o acumulador tenha o valor de X3. O código objeto compilado foi o seguinte (comentários após --):

```
NAME MAIN
 ORG /100
00 LV /02 – Eb para calcular o valor 2
 MM T0 – Variavel temporária 1
 LV /0 – Exp sendo resolvida
 + T0 – Soma-se T0
 MM X1 – Atribuição da variável, fim do primeiro Assign
10 LV /01 – Carrega o valor 1
 MM T1 -- e salva em T1 (Eb)
 LD X1 – Carrega o valor de X1
 MM T2 – e salva em T2 (Eb)
 LV /0
 + T1
 + T2
 MM X2 - - Fim da expressão.
20 LV /02 – Eb do primeiro operando da multiplicação
 MM T3 – Salva em T3 (eb)
 LD X2 – Agora entramos na expressão secundária dentro dos parênteses
 MM T5 – Salva em T5
 LV /02 – Carrega 2
 MM T6 – Salva em T6
 LV /01 --  Carrega 1
 MM T7 – Salva em T7
 LV /0
 + T5
 + T6
 + T7
 MM T4 – Fim da expressão dentro dos parênteses
```

```
 LV /0 – Carrega 0
 + T3 – Soma T3
 * T4 – Multiplica com T4
 MM X3 – Atribui em X3, fim do assign
30 LD X3 – Carrega em X3 (Print)
X1 DS /1 – Área de Dados
T0 DS /1
X2 DS /1
T1 DS /1
T2 DS /1
X3 DS /1
T3 DS /1
T4 DS /1
T5 DS /1
T6 DS /1
T7 DS /1
 END 00
```

Estado final da simulação e *snapshot* da memória:

```
O estado atual da maquina eh:
Acumulador: 000c
C.I.: 000
Instrucao: 000c
OP: 000
```

```
10  30  02  91  41  30  00  41  41  91  40  30  01  91  43  81  40
11  91  44  30  00  41  43  41  44  91  42  30  02  91  46  81  42
12  91  48  30  02  91  49  30  01  91  4a  30  00  41  48  41  49
13  41  4a  91  47  91  47  30  00  41  46  61  47  91  45  81  45
14  02  02  03  01  02  0c  02  06  03  02  01  00  00  00  00  00
```

f. Goto

Ao entrar em Goto, o seu *chainCode* só tem como atributo adicional o *label* de destino. Ao se consumir o não terminal de número após o GOTO, esse número é atribuído a variável label do *chainCode*. Por fim, ao se sair de Goto, é gerado somente um código de JP + label. O código elaborado para se testar foi (*teste/testeSemantico2.txt)*:

```
00 LET X1 = 5
05 GOTO 20
10 LET X1 = X1 + 5
20 PRINT X1
30 END
```

Com isso, espera-se que a variável X1 tenha o valor 5 ao fim do programa e não 10, caso o programa passe pela linha 10. O código-objeto obtido foi:

```
NAME MAIN
 ORG /100
00 LV /05
 MM T0
 LV /0
 + T0
 MM X1 -- Fim do primeiro Assign
05 JP 20 -- Jump
10 LD X1
 MM T1
 LV /05
 MM T2
 LV /0
 + T1
 + T2
 MM X1
20 LD X1 -- Print
X1 DS /1
T0 DS /1
X1 DS /1
T1 DS /1
T2 DS /1
 END 00
```

Estado final da simulação e *snapshot* da memória:

```
O estado atual da maquina eh:
Acumulador: 0005
C.I.: 000
Instrucao: 000c
OP: 000
```

```
10   30   05   91   1f   30   00   41   1f   91   1e   01   1c   81   1e   91   20
11   30   05   91   21   30   00   41   20   41   21   91   1e   81   1e   05   05
```

g. If

Ao se entrar no bloco *If*, o seu *chainCode* possui os seguintes atributos adicionais:

- *Variable1:* variável temporária que armazena qual o valor da expressão a esquerda do if
- *Variable2:* análoga a anterior, só que para o lado direito do if
- *Compare:* que armazena qual o tipo de comparação sendo feita
- *Destination:* label de destino do if, caso a expressão seja verdade.

Ao se sair do bloco *If,* o código gerado depende do tipo de compare, mas em geral ele carrega uma das variáveis, subtrai a outra e aí realiza o jump específico para se tratar aquele caso. No caso de *compare* sendo "=", por

exemplo, carrega-se *variable1* com LD + variable 1, subtrai-se *variable2* com - variable2 e por fim realiza-se um JZ destination. Para se testar o comando, elaborou-se o seguinte código (*teste/testeSemantico3.txt)*:

```
00 LET X1 = 3
10 LET X2 = 5
20 IF X1 < X2 THEN 40
30 LET X3 = 4
35 GOTO 50
40 LET X3 = 8
50 PRINT X3
60 END
```

Espera-se que, com esse código, X3 tenha o valor 8, como X2 é maior que X1, dessa forma evitando o código 30 e 35. O código-objeto gerado foi:

```
NAME MAIN
 ORG /100
00 LV /03
 MM T0
 LV /0
 + T0
 MM X1 – Primeiro Assing
10 LV /05
 MM T1
 LV /0
 + T1
 MM X2 – Segundo Assign
20 LD X1
 MM T4
 LV /0
 + T4
 MM T2 – Primeiro Exp do If
 LD X2
 MM T5
 LV /0
 + T5
 MM T3 – Segundo Exp do IF
 LD T2
 - T3
 JN 40 – Fim do IF
30 LV /04
 MM T6
 LV /0
 + T6
 MM X3 – Se der certo, o código não deveria entrar nessa área
35 JP 50
```

```
40 LV /08
 MM T7
 LV /0
 + T7
 MM X3 – Fim do IF
50 LD X3 -- Print
X1 DS /1
T0 DS /1
X2 DS /1
T1 DS /1
T4 DS /1
T5 DS /1
X3 DS /1
T6 DS /1
X3 DS /1
T7 DS /1
 END 00
```

Estado final da simulação e *snapshot* da memória:

```
O estado atual da maquina eh:
Acumulador: 0008
C.I.: 000
Instrucao: 000c
OP: 000
```

```
10  30  03  91  47  30  00  41  47  91  46  30  05  91  49  30  00
11  41  49  91  48  81  46  91  4c  30  00  41  4c  91  4a  81  48
12  91  4d  30  00  41  4d  91  4b  81  4a  51  4b  21  3a  30  04
13  91  4f  30  00  41  4f  91  4e  01  44  30  08  91  50  30  00
14  41  50  91  4e  81  4e  03  03  05  05  03  05  03  05  08  00
```

h. For/Next

Ao entrar na submáquina *For*, são criadas 3 variáveis temporárias, uma para o valor inicial variável de controle do for, outra para a variável que vai armazenar o valor de término do for e a última que armazena o valor de incremento a cada step. Além disso, há um flag para caso o *step* seja declarado e também um atributo que guarda o nome da variável de controle.

Ao se sair da submáquina for, é feito o assign da variável de controle com o seu valor inicial, e são gravados os valores das variáveis de valor máximo da guarda e valor do Step. Caso a flag do step esteja desligada, é carregado o valor um (LV /1 MM stepVariable) na variável de step. Além disso, no próprio *categorizador*, se armazena a variável de controle e o *forFlag* é ligado e a variável é armazenada no *forStack.*

Para o comando Next, a variável é incrementada com o valor de step e subtraída do valor máximo da guarda. É feito um JN para ver se podemos sair

do for ou não de acordo com o endereço que foi desempilhado do *forStack.* Para se testar, elaborou-se o seguinte código (*teste/testeSemantico4.txt)*:

```
00 LET X1 = 1
05 FOR I = 0 TO 2
10 LET X2 = 0
14 LET X1 = X1 * X2
15 FOR J = 0 TO 6 STEP 2
16 LET X2 = X2 + 1
17 NEXT J
20 NEXT I
25 PRINT X1
30 END
```

Espera-se que o *for* interno rode 4 vezes, fazendo com que X2 tenha o valor 4. Já espera-se que o for externo rode 3 vezes, fazendo com que a o valor final de X1 seja
(((1*4)*4)*4) ou seja, 4**3 = 0x40.

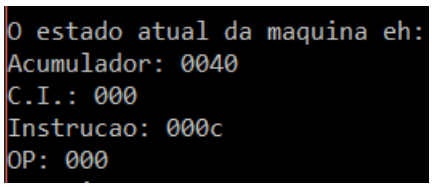
O código objeto obtido foi:

```
NAME MAIN
 ORG /100
00 LV /01
 MM T0
 LV /0
 + T0
 MM X1 – Fim do primeiro assign
05 LV /00
 MM T4
 LV /0
 + T4
 MM T1 – Primeira expressão do For
 LV /02
 MM T5
 LV /0
 + T5
 MM T2 – Guarda de controle do For
 LV /1
 MM T3 – Step 1
 LD T1
 MM I – Assign do For
10 LD X1
 MM T6
 LD X2
 MM T7
 LV /0
```

```
 + T6
 * T7
 MM X1 – Expressão do primeiro de X1 = X1*X2
14 LV /00
 MM T8
 LV /0
 + T8
 MM X2 – Assign de X2
15 LV /00
 MM T12
 LV /0
 + T12
 MM T9 – Valor inicial do segundo For
 LV /06
 MM T13
 LV /0
 + T13
 MM T10 – Valor máximo do Segundo For
 LV /02
 MM T14
 LV /0
 + T14
 MM T11 -- Step do Segundo For
 LD T9
 MM J – Assign do Segundo for
16 LD X2
 MM T15
 LV /01
 MM T16
 LV /0
 + T15
 + T16
 MM X2 – Assign de X2 dentro do for
17 LD J
 + T11
 MM J – Incremento de J
 - T10
 JN 16 – Destino do Segundo for
20 LD I
 + T3
 MM I – Incremento de I
 - T2
 JN 10 – Destino do primeiro For
25 LD X1 – Data Area
X1 DS /1
T0 DS /1
T1 DS /1
```

```
T2 DS /1
T3 DS /1
I DS /1
T4 DS /1
T5 DS /1
T3 DS /1
T6 DS /1
T7 DS /1
X2 DS /1
T8 DS /1
T9 DS /1
T10 DS /1
T11 DS /1
J DS /1
T12 DS /1
T13 DS /1
T14 DS /1
T15 DS /1
T16 DS /1
 END 00
```

Estado final da simulação e *snapshot* da memória:



i. Gosub/Return

Ao se entrar na máquina Gosub, seu *chainCode* possui como único atributo adicional o destino, que é atualizado ao se passar pelo não terminal seguinte ao comando GOSUB. Ao sair de Gosub, é feito um SC para esse destino e a gosubFlag no *categorizador* é ligada para que o número da próxima linha seja gravado para o *return*.

No *return*, é desempilhado o endereço de onde o Gosub foi chamado e é chamada a instrução RS para essa label. Para se testar esse código, foi elaborado o seguinte código (*teste/testeSemantico5.txt)*:

```
00 LET X1 = 0
10 GOSUB 30
20 GOTO 50
30 LET X1 = 5
40 RETURN
50 PRINT X1
```

60 END

O comportamento esperado é que, ao final, X1 tenha o valor 5 pois GOSUB fez ele redirecionar para a linha 30 e só depois retornar para o GOTO que encerra o programa com o print. O código gerado foi o seguinte:

```
NAME MAIN
 ORG /100
00 LV /00
 MM T0
 LV /0
 + T0
 MM X1 – Assign de X1
10 SC 30 – Subroutine Call para 30
20 JP 50 – Pula pro jump
30 DW /2 – Reserva de espaço para o endereço de retorno
 LV /05
 MM T1
 LV /0
 + T1
 MM X1 – Incremento de X1
40 RS 30 – Volta para o endereço original
50 LD X1 – Print X1
X1 DS /1
T0 DS /1
T1 DS /1
 END 00
```

Estado final da simulação e *snapshot* da memória:

```
O estado atual da maquina eh:
Acumulador: 0005
C.I.: 000
Instrucao: 000c
OP: 000
Memoria:
```

```
10   30   00   91   1f   30   00   41   1f   91   1e   a1   0e   01   1c   01   0c
11   30   05   91   20   30   00   41   20   91   1e   b1   0e   81   1e   05   00
```

4. Apêndice

a. Apêndice A

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>

Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 0
maquina:BS , estado: 2, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2)]

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh REM
maquina:Remark , estado: 2, pilha: [('Program', 2), ('BS', 3)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2)]

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 231231
maquina:Remark , estado: 2, pilha: [('Program', 2), ('BS', 3)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2)]

Tempo: 6
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

maquina:BS , estado: 4, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4)]

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 14
maquina:BS , estado: 2, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2)]
maquina:Program , estado: 3, pilha: [], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('Program', 3)]

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh REM
maquina:Remark , estado: 2, pilha: [('Program', 2), ('BS', 3)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2)]

Tempo: 7

Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenId'> e seu conteudo eh Tx31
maquina:Remark , estado: 2, pilha: [('Program', 2), ('BS', 3)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2)]

Tempo: 7
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

maquina:BS , estado: 4, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4)]

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenNumero'> e seu conteudo eh 20
maquina:BS , estado: 2, pilha: [('Program', 2)], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2)]
maquina:Program , estado: 3, pilha: [], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('Program', 3)]

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenReservado'> e seu conteudo eh END
maquina:Program , estado: 4, pilha: [], historico: [('Program', 1), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('BS', 1), ('BS', 2), ('Remark', 1), ('Remark', 2), ('Remark', 2), ('BS', 3), ('BS', 4), ('Program', 2), ('Program', 3), ('Program', 4)]

Tempo: 8
Maquina: <class 'CategorizadorSintatico.CategorizadorSintatico'>
Log: O token recebido eh do tipo <class 'eventos.TokenLinha'> e seu conteudo eh

Fim da análise sintática!