



SISTEMAS DE INFORMAÇÃO
FUNDAMENTOS DE COMPILADORES

COMPILADOR Cmm

Trabalho apresentado como parte das avaliações parciais da disciplina Fundamentos de Compiladores do curso de Sistemas de Informação do Campus I da UNEB.

MATEUS NASCIMENTO SANTOS
WELLINGTON CORREIA DOS SANTOS

2017.2

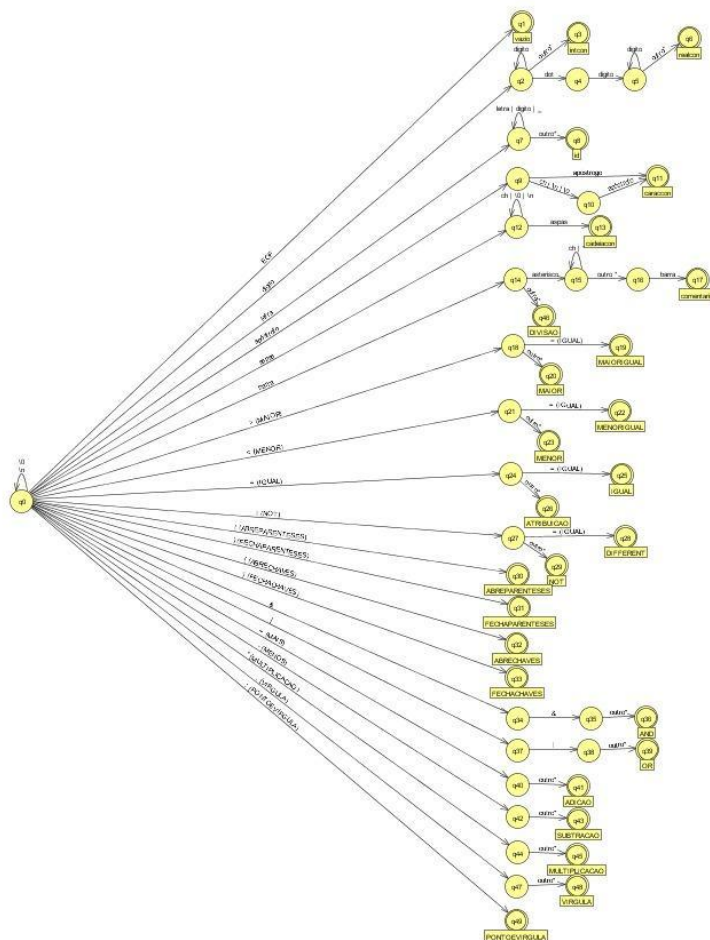
1. Introdução

Como proposto na matéria de Fundamentos de Compiladores, foi necessário criar um compilador básico que conseguisse interpretar a linguagem criada, chamada de Cmm. Com base nessa linguagem, foi criado esse compilador que será descrito mais a seguir. Nesse compilador foi desenvolvido os analisadores léxico, sintático e semântico, os gerenciadores de tabela de sinais e de erro e a máquina de pilha. O sistema teve em torno de uma semana de desenvolvimento para cada parte citada anteriormente. E com base nesse tempo, tivemos uma estimativas de 85 horas de dedicação ao projeto.

2. Análise Léxica

Foram desenvolvidos dois analisadores léxicos, cada um por um componente da equipe, onde ao final, se fez necessário a escolha de um, para o procedimento da construção do compilador. Os dois estavam completos, abrangiam todos os tokens que a documentação informava como elementos da linguagem Cmm.

Houve a troca para o analisador léxico para um que houvesse uma melhor organização em sua estrutura. O desenvolvimento desse analisador se deu em cima do seguinte AFD.

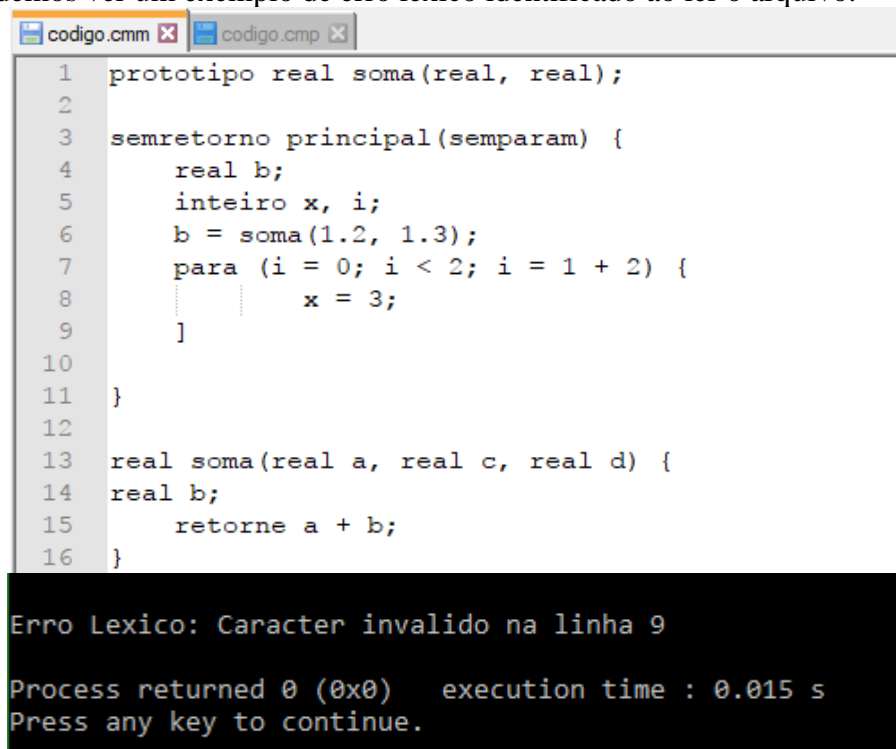


Com base neste AFD, foi desenvolvido o analisador, onde todos os caminhos que existem no AFD foram implementados, considerando as identificações de cada um dos tokens, e separando os identificadores em palavras reservadas ou ID.

As estruturas de dados utilizadas no analisador léxico foram os vetores de caracteres, onde armazenavam os dados referentes a sinais, palavras reservadas, identificadores, além de valores numéricos e caracteres. Com a utilização de uma struct, podemos nela armazenar os dados e suas identificações como categoria, linha, lexema e código. As outras estruturas utilizadas foram as funções que são referente a ler o arquivo do código Cmm e identificar que tipo de dado ele é, e armazená-lo.

Foi identificado um erro no analisador, que foi corrigido para a segunda entrega. Este erro acontecia quando era necessário armazenar uma literal com caracteres especiais. Estava sendo utilizada uma função para guardar estes dados, onde está não abrangia todos os caracteres necessários.

A seguir podemos ver um exemplo de erro léxico identificado ao ler o arquivo.



```
1 prototipo real soma(real, real);
2
3 semretorno principal(semparam) {
4     real b;
5     inteiro x, i;
6     b = soma(1.2, 1.3);
7     para (i = 0; i < 2; i = 1 + 2) {
8         x = 3;
9     }
10
11 }
12
13 real soma(real a, real c, real d) {
14     real b;
15     retorne a + b;
16 }
```

Erro Lexico: Caracter invalido na linha 9

Process returned 0 (0x0) execution time : 0.015 s
Press any key to continue.

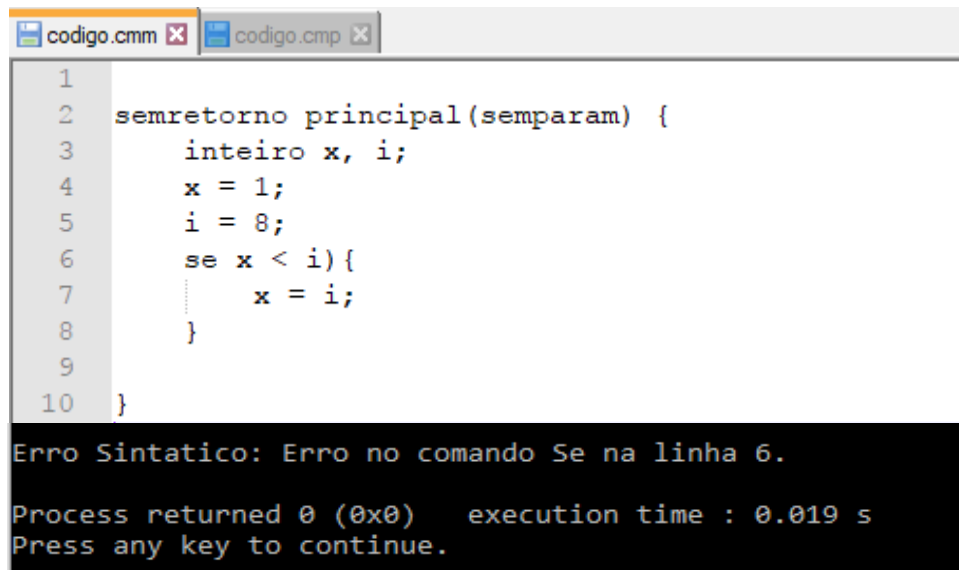
3. Análise Sintática

O analisador sintático, não tivemos muito o que discutir sobre a forma em que seria implementada uma vez em que deveríamos seguir a documentação proposta para a linguagem Cmm. O código se baseia basicamente em análise dos dados encontrados pelo analisador léxico e chamada da função que irá fazer o reconhecimento da estrutura. Na estrutura do código do analisador sintático é que se encontra a maior parte de toda a codificação do compilador, nessa parte são utilizados os analisadores léxico, sintático e semântico. Também é utilizado a parte gerenciador de erros, pois nessa parte do código é quase feita todas as verificações do código em relação a erro de análise, gerenciador da tabela de símbolos e onde é chamada todas as funções referentes a máquina de pilha. Tivemos um problema que não conseguimos resolver, quando há duas estruturas, uma dentro da outra, se houver um “SE” e dentro dele houver um “ENQUANTO” o

sistema não irá conseguir compilar, e vai dá um erro referente a falta de uma chave no final do arquivo. Caso seja colocada a chave onde o sistema irá indicar, o compilador irá conseguir ler todo o arquivo, porém sintaticamente estará errado. Esse foi um erro em que não conseguimos corrigir.

Com isso conseguimos construir quase todas as funcionalidades relacionadas com a análise sintática. Não tivemos muitas dificuldades no decorrer do processo do analisador sintático, a não ser esse último problema citado em que não conseguimos corrigir.

Podemos ver um exemplo de erro sintático na imagem a seguir:



```
1
2 semretorno principal(semparam) {
3     inteiro x, i;
4     x = 1;
5     i = 8;
6     se x < i){
7         x = i;
8     }
9
10 }

Erro Sintatico: Erro no comando Se na linha 6.

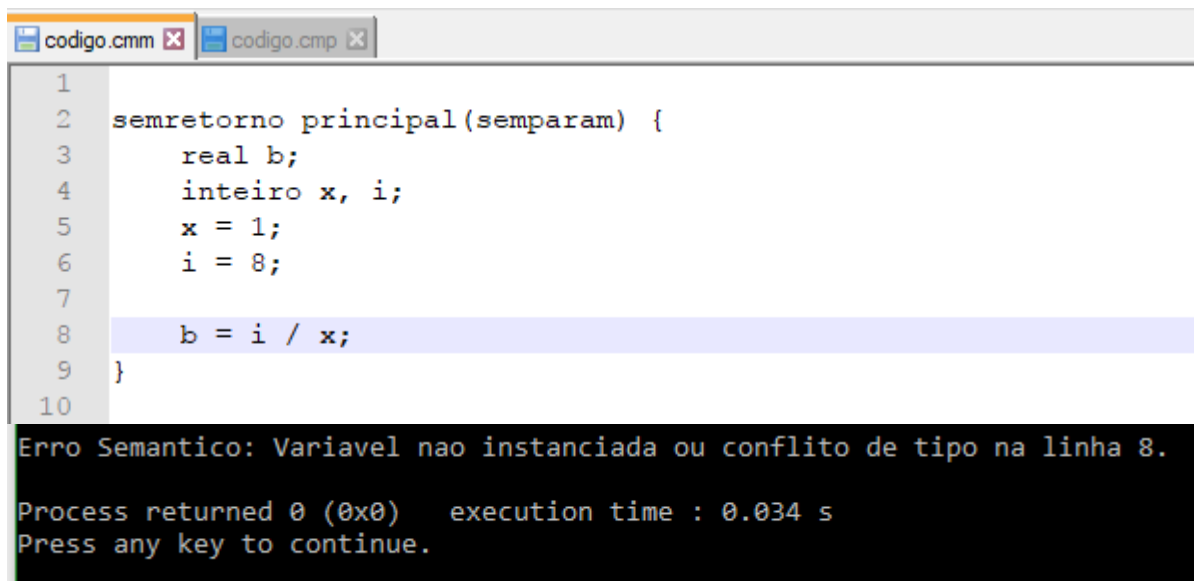
Process returned 0 (0x0)   execution time : 0.019 s
Press any key to continue.
```

4. Análise Semântica

Para o analisador semântico as decisões tomadas foram referentes entender e aplicar o nosso entendimento ao desenvolvimento, pois ficamos com dúvidas em relação a forma que iríamos por exemplo analisar o retorno de uma função e atribuir a uma variável ou a entrada de uma função que tenham o mesmo tipo do retorno. Esse também é um problema em nosso desenvolvimento, pois tivemos algumas dúvidas, mas com o decorrer dos estudos e duvidas tiradas, podemos entender como poderíamos fazer e conseguimos concluir essa parte. A partir disso podemos tomar a decisão de analisar primeiramente o tipo de retorno do método, já que ele é armazenado no nosso gerenciador de símbolos e o tipo de entrada ou variável que estamos atribuindo e verificar se os dois tipos eram iguais. Parecido a com a forma de fazer calculo entre dois números onde verificamos o tipo do primeiro que é pego pelo analisador léxico e o tipo do segundo e caso tenha um lugar onde esteja sendo atribuído, verificar o seu tipo também e com isso dizer se é permitida ou não a operação. Tudo dentro do que foi proposto pelo semântico foi implementado como atribuir retorno de função para viável do mesmo tipo, cálculo entre dois números ou mais sendo feito com o mesmo tipo e etc.

Na imagem a seguir mostramos o funcionamento da analise semântica, onde é identificado uma divisão entre duas variáveis inteiras sendo atribuída a uma variável do tipo real. Pela regra, esse tipo de operação é inválida.

Podemos ver um exemplo de erro semântico na imagem a seguir



The screenshot shows a window titled 'codigo.cmm' and 'codigo.cmp'. The code in the editor is as follows:

```
1
2 semretorno principal(semparam) {
3     real b;
4     inteiro x, i;
5     x = 1;
6     i = 8;
7
8     b = i / x;
9 }
10
```

Below the code, a black error message box displays the following text:

```
Erro Semantico: Variavel nao instanciada ou conflito de tipo na linha 8.
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

5. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

O gerenciador de tabela de símbolos decidimos por criar uma struct que seria o responsável em armazenar o lexema, o tipo do parâmetro, em qual escopo ele se encontra, se é zumbi, linha e posição. Foi utilizado um ENUM que contém a identificação para quando o escopo é global ou local e as demais funções responsáveis em inserir na tabela, alterar, excluir quando é do escopo de alguma função e há a necessidade de exclusão. O que o gerenciador faz é que quando se inicia a leitura do arquivo, sempre que ele identificar algum parâmetro que possa ser armazenado, ele irá inserir na tabela e irá identificar qual o seu escopo, se está dentro de função, em entrada de função ou se é global. Tivemos problemas com os zumbis, pois quando tinha alguma função que tivesse como entrada alguma variável, e em outra função utilizássemos essa variável mesmo sem declarar no escopo dela, ela era reconhecida como já declarada. Porém esse problema foi resolvido. E não tivemos mais nenhum outro problema com a parte de gerenciamento da tabela de sinais.

Para o gerenciador de erros decidimos por criar um arquivo separado para ele, como as demais funcionalidades e por criamos 25 (vinte e cinco) erros possíveis de ocorrer no sistemas, para isso criamos um ENUM apenas para ficar melhor para identificarmos qual tipo de erro está dando e onde está essa parte no código em possíveis manutenções, e uma função que irá apenas printar o erro na tela no momento em que ocorre um e identifica em qual linha ocorre o erro, com exceção de alguns erros que não irão identificar em que ele ocorre, exemplo seria “Erro ao abrir o arquivo”. Com o exemplo desse erro podemos citar que utilizamos uma cadeia de caracteres que irá armazenar todas as mensagens de erro.

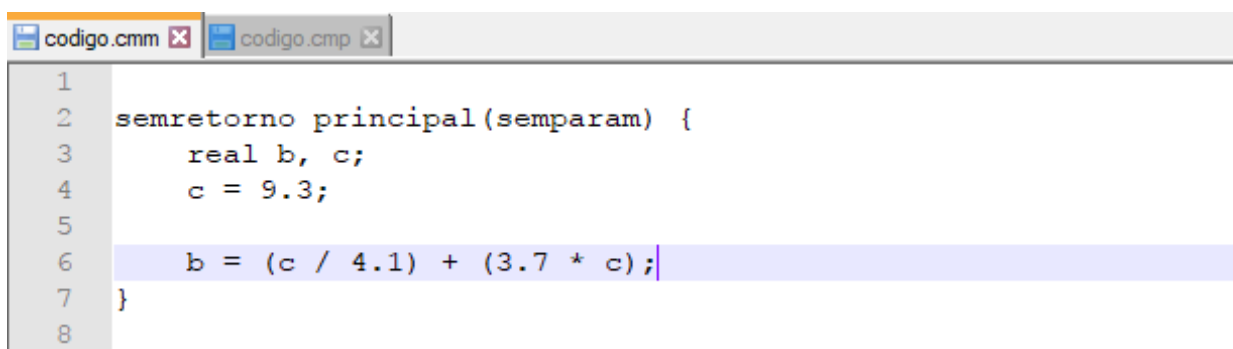
Quando ocorre algum erro no sistema, quando falta algum ponto e vírgula “;” será chamado o método de erro, nele deverá ser enviado qual tipo de erro que está acontecendo e em qual linha.

6. Gerador de Código da MP

A máquina de pilha, decidimos por criar o arquivo que iria conter o código gerado, todos os prints seriam colocados no arquivo através de funções que seriam criadas em um arquivo destinado apenas para a máquina de pilha, o “maqpilha.h” e “maqpilha.c”. O que foi feito nessa parte foi analisar em quais partes poderiam ser colocado cada método com a finalidade de que fosse

impresso de forma certa no arquivo. Tivemos uma facilidade no início para implementar as funções que iriam colocar os códigos relacionado às operações de SOMA, SUBTRAÇÃO, MULTIPLICAÇÃO E DIVISÃO, porém as operações de “maior e igual” e “menor e igual” tivemos uma certa dificuldade para implementar, pois não estávamos encontrando um local correto para colocar as impressões de cada operação que ocorreria dentro dessas das mesmas. Basicamente o que ocorre na execução dessa funcionalidade é abrir o arquivo e ir colocando cada informação dentro do mesmo de acordo com que vai ocorrendo as operações em sua execução, caso ocorra algum erro na execução, fechamos o arquivo que está sendo armazenado as informações e encerramos o programa.

Podemos ver a seguir um exemplo do funcionamento da máquina de pilha, na primeira parte vemos o código que será lido e na segunda vemos o código gerado por ela.



```
1
2 semretorno principal(semparam) {
3     real b, c;
4     c = 9.3;
5
6     b = (c / 4.1) + (3.7 * c);
7 }
8
```



```
1 INIP
2 GOTO L1
3 LABEL L2
4 INIPR 1
5 AMEM 2
6 PUSH 9.3
7 STOR 1,2
8 LOAD 1,2
9 PUSH 4.1
10 DIVF
11 PUSH 3.7
12 LOAD 1,2
13 MULF
14 ADDF
15 STOR 1,1
16 DMEM 2
17 RET 0
18 LABEL L1
19 CALL L2
20 DMEM 0
21 HALT
```

7. Conclusão

As primeiras partes do projeto foram simples, como o analisador léxico, onde há a necessidade apenas de criar um AFD e implementar estrutura de decisão em código, baseado no AFD e na estrutura do código Cmm. Porém com o decorrer do desenvolvimento, foi complicando, porém conseguimos resolver os problemas que foram ocorrendo no decorrer de sua construção.

Os impactos são referentes a complexidade de alguns sistemas e a partir disso observar os mínimos detalhes em um desenvolvimento, pois uma parte bem desenvolvida, não fará com que tenhamos problemas futuros em outras partes em decorrência de uma anterior mal feita.

Com base em todo o estudo sobre cada parte de um compilador, por mais que não tenha sido para criar um compilador profissional, mas criar um que pudesse ser feito em um tempo relativamente curto, para pessoas que apenas o utilizam e não imaginam todas as etapas necessárias para que um compilador pudesse realmente compilar um código, é de extrema necessidade a forma como a matéria aborda o tema, porém muita teoria mesmo que necessária, faz com que alguns passos de criação do compilador pareçam mais complexos do que realmente são.