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.

ANA CECÍLIA SANTOS SOUZA E STEPHANIE LIMA ROCHA

2017.2

1. Introdução

O desenvolvimento do Projeto do Compilador Cmm deu-se início no segundo semestre de 2017 e manteve-se até a metade do mês de dezembro começando a partir do estudo da teoria apresentada pelo professor da disciplina e em seguida o desenvolvimento do Analisador Léxico. Através de uma sequência lógica, as outras etapas, bem como o Analisador Sintático, Tabela de Símbolos, Analisador Semântico e a Máquina de Pilha foram desenvolvidas. Todas as etapas foram versionadas usando o Git através de uma plataforma de hospedagem de códigos-fontes conhecida como GitHub.

Para o progresso dessas etapas, sucederam-se uma dedicação total, onde as atividades relacionadas a cada etapa foram executadas de forma iterativa, ou seja, cada execução de uma determinada atividade, como criar uma função, testar uma função gerava um resultado que contribuía de forma significativa e necessária para a implementação da atividade que estaria por vir posteriormente. Em linhas gerais, o projeto originou-se com o objetivo de apresentar através da prática como de fato um compilador se comporta desde à leitura dos caracteres (*scanner*) até a geração de código.

O cronograma abaixo apresenta as atividades e uma estimativa de tempo considerando somente os meses despendido para a execução de cada uma delas.

Atividade	**Estimativa de tempo**				
	AGO	**SET**	**OUT**	**NOV**	**DEZ**
Estudo teórico dos conceitos necessários para o desenvolvimento do analisador léxico					
Criação do autômato					
Implementação do Analisador Léxico					
Testes no Analisador Léxico					
Correções no Analisador Léxico					
Estudo teórico dos conceitos necessários para o desenvolvimento do Analisador Sintático					
Implementação do Analisador Sintático					
Implementação da Tabela de Símbolos					
Testes no Analisador Sintático					
Testes na Tabela de Símbolos					
Correções no Analisador Sintático					

Correções na Tabela de Símbolos					
Estudo teórico dos conceitos necessários para o desenvolvimento do Analisador Semântico e Máquina de Pilha					
Implementação do Analisador Semântico					
Implementação da Máquina de Pilha					

2. Análise Léxica

O Analisador Léxico consiste na primeira fase do compilador. O objetivo dessa fase é analisar a entrada de caracteres que pode ser um código-fonte de um programa qualquer de computador e gerar uma sequência de *tokens* que podem ser interpretados por um *parser.* Para a implementação desta etapa, foi necessário entender como os *tokens* deveriam ser tratados. Para isso, um AFD (Autômato Finito Determinístico), apresentado na Figura 1, foi gerado considerando as regras léxicas mencionadas na especificação do projeto.

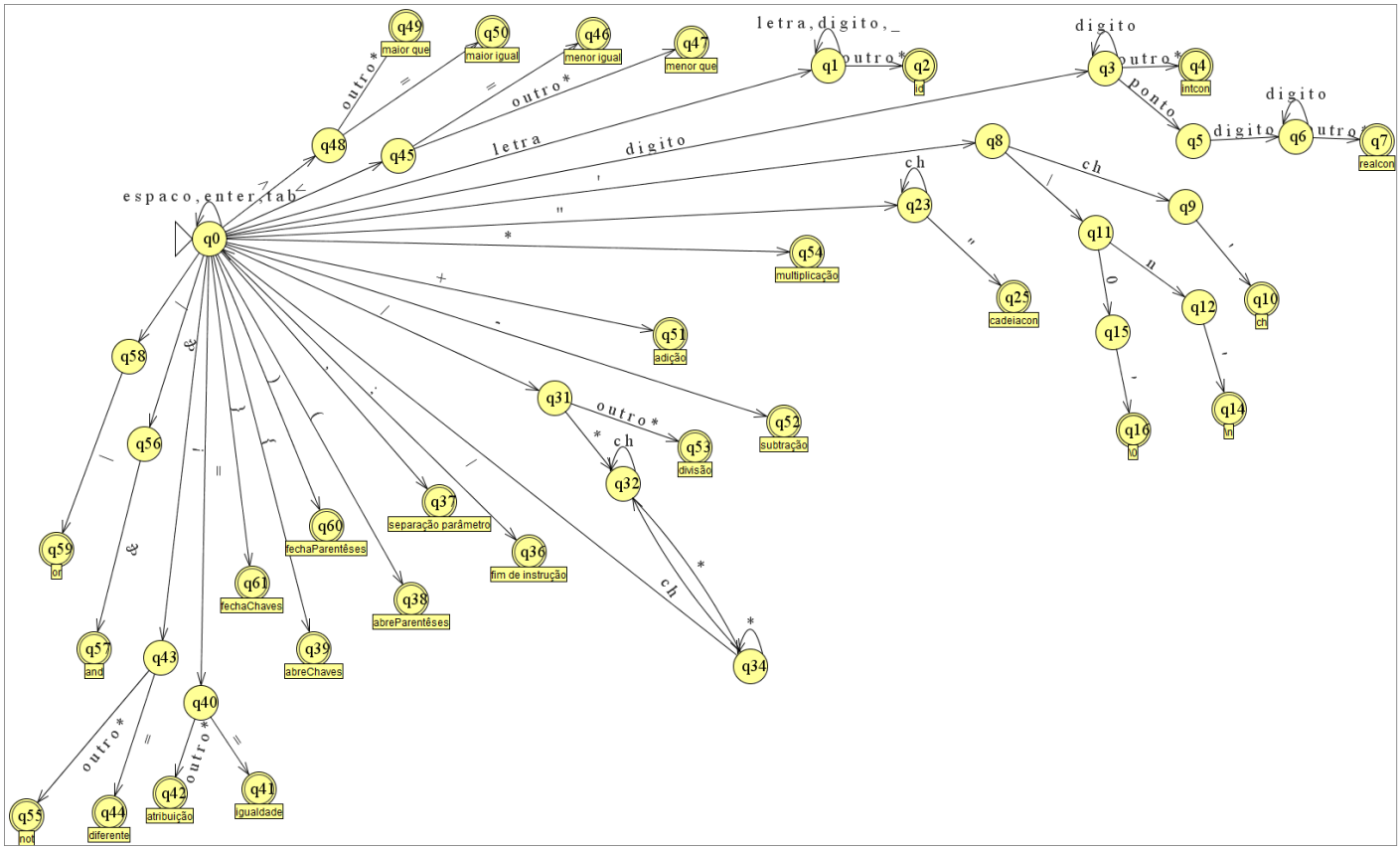

Figura 1: Autômato Finito Determinístico (AFD)

O AFD foi criado utilizando um *software* chamado JFLAP. Com o AFD definido, a implementação do código se tornou mais simples e de fácil compreensão. Um arquivo chamado lexico.c foi criado contendo todas as funções que foram utilizadas para definir as sequências de *tokens.* As assinaturas dessas funções foram previamentes definidas em um arquivo chamado lexico.h. No arquivo lexico.c possui 5 funções. A primeira função denominada abrirArquivo, recebe um arquivo e abre somente para leitura, caso o arquivo não seja encontrado a função retornará uma mensagem acusando erro ao abrir e sairá do programa.

A segunda função denominada verificarPR é chamada toda vez que um lexema é iniciado por uma letra, podendo receber outras letras, números ou *underline.* A função recebe como parâmetro uma string e compara com os nomes que estão definidos em uma

tabela chamada tabelaPR. Essa tabela armazena somente os nomes das palavras reservadas que foram descritas nas regras léxicas da especificação do projeto. Caso a string passada como parâmetro não seja igual a nenhuma dessas palavras que estão na tabela, a função retorna -1 e, portanto, sabe-se que a string é um identificador, do contrário, seria uma palavra reservada.

A terceira função, erro_Lexico, imprime na tela uma mensagem de erro, reportando a linha em que o erro aconteceu. O contLinha é uma variável global que foi inicializada com 1 e na função principal do programa é incrementada. A quarta função, inicializaEstrutura, foi criada com o objetivo de inicializar as variáveis da estrutura toda vez que um comentário for lido. Nesse caso, o estado atual sempre retornará para o estado inicial e iniciará a variável Estrutura com o próximo token.

E por fim, a última função, denominada analisadorLexico recebe o arquivo que foi aberto pela primeira função e percorre todo conteúdo do arquivo. Nesta função, o comando *switch case* foi usado com a finalidade de implementar comportamentos diferentes para cada estado que foi especificado no autômato. Cada estado é montado um lexema e retornado para a variável Estrutura.

Apesar do analisador léxico parecer muito simples e fácil de implementar, existiram algumas dificuldades no início da codificação. A primeira dificuldade foi em relação ao autômato, pois se o mesmo for elaborado errado, a codificação estará errada também, já que o mesmo reflete todo o processo de desenvolvimento do léxico. Então, existiu a necessidade de corrigi-lo mesmo depois de ter começado a implementá-lo. Como consequência, os estados do autômato ficaram desorganizados em relação a sequência. E a segunda dificuldade, foi perceber que os estados do autômato refletiam os *cases* do código. Por esse motivo, na primeira implementação, os estados no código foram criados em uma ordem sequencial em desacordo com o AFD.

3. Análise Sintática

O Analisador Sintático consiste na segunda fase do compilador. O objetivo dessa fase é determinar se uma cadeia de *tokens* pode ser gerada pela gramática do Cmm. Nessa etapa do projeto, um marco muito importante foi estabelecido pelo professor da disciplina. A possibilidade de dar continuidade ao projeto escolhendo um colega de classe. Diante desse ponto significativo, o projeto passou a ter uma contribuição muito maior. Para prosseguir com a implementação, foi necessário escolher um analisador léxico que pudesse compor o módulo do sintático. Então decidiu-se escolher o código que tivesse menos correções para serem feitas após a entrega da primeira fase.

O analisador sintático foi implementado considerando todas as regras de produção da gramática mencionadas na especificação. Assim como o léxico, um arquivo chamado sintatico.h foi definido com as assinaturas das funções e para a implementação do escopo das funções, um arquivo chamado sintatico.c foi gerado. No início da implementação, um autômato foi definindo, onde os estados caracterizavam cada regra de produção. Porém, percebeu-se que era necessário um tempo muito maior para a criação dos autômatos para cada regra de produção, e, portanto, essa estratégia não vingou.

Então, foi decidido implementar sem o uso dos autômatos. A segunda estratégia utilizada foi codificar as regras de produção todas de uma vez, e validar todas as funções através de arquivos textos que continham a estrutura de um programa. Desta forma, foi possível encontrar inúmeros *bugs* e corrigi-los no momento da validação. O sintático possui doze funções, que estão descritas abaixo, respectivamente.

1. erro: essa função consiste em imprimir na tela uma mensagem de erro sintático informando a linha onde o erro aconteceu.
2. sintatico: essa função foi implementada com o objetivo de inicializar as variáveis Estrutura e nextEstrutura com os *tokens* que foram gerados pelo analisador léxico e chamar a função principal do sintático, prog. A variável nextEstrutura foi criada nessa fase, pois em algumas funções era necessário saber qual seria o próximo *token* gerado para tomar uma determinada decisão. Por exemplo, quando um identificador é encontrado é preciso saber se o próximo *token* a ser gerado será uma atribuição ou um abre parêntese que define o início de uma função. Por esse motivo, o analisador léxico é chamado duas vezes para preencher essas variáveis.
3. prog: essa função é chamada pelo sintático e pode ser considerada como a função principal, pois é a partir dela que as outras são chamadas. Nesta função, é verificada a estrutura das declarações de variáveis, protótipos de funções e funções com e sem retorno.
4. tipos_param: essa função é chamada por prog no momento em que a estrutura das funções está sendo analisadas. Nessa função, os identificadores são imprescindíveis.
5. tipos_p_opc: essa função é muito semelhante a tipos_param, a única diferença é que ela atende à estrutura de protótipos de funções e os identificadores não são obrigados a serem especificados. A maior dificuldade para implementá-la foi no

momento de decidir como o identificador poderia ser ou não definido sem impactar a função.

6. cmd: essa função foi a mais difícil de implementar. No momento da validação, pode-se perceber que era necessário dedicar um tempo maior para corrigir cada condição. Ela é uma função que é chamada dentro de um *loop* no prog. A função só é finalizada quando um fecha chaves é encontrado. Enquanto isso, a função vai sendo executada chamando ela mesma ou outras funções. A primeira dificuldade foi em relação aos identificadores quanto ao uso do próximo *token*. O processo de entendimento foi muito lento, e por isso, as correções do sintático foram as que mais demoraram. Outra dificuldade encontrada era poder identificar os pontos em que era necessário chamar o analisador léxico para poder gerar o *token.* Na maioria das vezes, o *token* retornava errado para a função que a chamou, e por consequência, o ato de debugar tornou-se necessário em intervalos de tempo muito curtos. Durante as correções na função cmd, existiram duas situações que talvez a solução aplicada para resolver o problema não tenha sido uma das melhores. Na condição do RETORNE, o *token* não retornava das funções anteriores com o valor que deveria vir, por esse motivo o nextEstrutura foi utilizado para verificar se o próximo *token* satisfazia a situação, se sim, então o analisador léxico era chamado, caso contrário, saía por vazio. Outra situação foi descoberta enquanto a condição PARA era validada. Foi visto que o analisador léxico não estava retornando o *token* separado, ou seja, no caso de i=0, o correto seria trazer três *tokens,* mas o que tava sendo retornado foi o i separado, o sinal de igual junto com o zero estava sendo considerado como um *token* único, e por fim o zero se repetia, e portanto a análise sintática finalizava acusando erro sintático. Para solucionar este problema, foi necessário no arquivo texto colocar espaço em branco dentre os *tokens*, e desta forma o código executou corretamente.
7. atrib: essa função verifica se o identificador é uma atribuição, caso seja, a função chama expr.
8. expr: essa função é chamada por atrib ou fator. Ela pode chamar somente a função expr_simp e sair do escopo, ou ainda pode chamar op_rel e após chamar expr_simp novamente. A maior dificuldade para implementá-la foi no momento de decidir como as funções poderiam ser chamadas ou não sem impactar na função propriamente dita.
9. expr_simp: essa função é chamada por expr e analisa os *tokens* de adição, subtração e o operador lógico *or* dentro de um *loop.*
10. termo: essa função é chamada por expr_simp e analisa os *tokens* de divisão, multiplicação e o operador lógico *and* dentro de um *loop.*
11. fator: essa função é chamada por termo e analisa os *tokens* as categorias que foram definidas no analisador léxico além de verificar as expressões chamando a função expr ou ser recursivo chamando a si próprio toda vez que encontrar uma negação. Uma das dificuldades para implementar esta função foi no momento em que o *token* é gerado por um identificador. Essa dificuldade foi bem comum sempre que o token é gerado por um identificador, mas quando resolvido, a solução foi implementada nos outros lugares em que a situação se repetia.

12. op_rel: essa função é chamada por expr para especificar os operadores booleanos. Ela foi a mais fácil de ser implementada.

4. Análise Semântica

A análise semântica foi iniciada com a análise da especificação para identificar em que consistia essa etapa do projeto e que tipo de validações deveriam ser realizadas no código. Em seguida, foi identificado em quais pontos no código da análise sintática essas funções de validações deveriam ser feitas. Após isso, começou a implementação dessa funções de validação, e logo, foi percebido a necessidade de utilizar a tabela de símbolo, definida na fase do sintático.

Devido a isso, foi necessário revisar as funções de gerenciamento da tabela de símbolos. Uma das mudanças realizadas foi a criação da função de gerenciadorTabelaSimbolos que tinha como objetivo executar as outras funções de tabela, como: inserir, excluir e consultar. Essa decisão foi feita para que fosse chamada apenas uma função no analisador sintático, o que deixou o código mais organizado e mais fácil de realizar mudanças.

Após a revisão das funções de gerenciamento da tabela de símbolo, o foco se tornou a implementação das funções de validação da análise semântica. Primeiramente, foi implementado as funções de menor complexidade, por exemplo, a função de verificar lógica de declaração(verificarDeclaracao) e redeclaração(verificarDeclaracaoVariavel) de variável. A próxima função a ser implementada foi a responsável por validar a declaração de função. Nesse ponto foi identificado a primeira dependência entre as funções, pois antes de validar a função era necessário realizar as funções de validação do protótipo da função, caso existisse.

Uma grande parte das funções implementadas na análise semântica foram referentes a validação de função e algumas delas estão elencadas abaixo:

1. temPrototipo: função responsável por verificar se a função possui protótipo e validar o protótipo.
2. verificarRepetePrototipo: função responsável por verificar se um protótipo foi declarado mais de uma vez.
3. verificarRetorno: função responsável por verificar os retornos das funções
4. verificarFuncaoPrincipal: função responsável por verificar se existe uma função com o nome do identificador chamado principal no programa.

5. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

A tabela de símbolo foi implementada seguindo a orientação do professor da disciplina e em razão disso, foi utilizada a estrutura de dados de pilha, onde foi armazenada as informações dos identificadores. A tabela foi desenvolvida em paralelo ao sintático. A nível do sintático, a tabela de símbolos armazenava todos os identificadores encontrados no código. Quanto ao nível do semântico, a tabela de símbolos precisou ser modificada para se ajustar às necessidades da terceira fase.

A tabela de símbolos foi definida juntamente com algumas variáveis que compõem a sua estrutura, sendo elas: nome, codigoTipo, escopo, categoria, zumbi e qntdParam. Onde o nome diz respeito ao nome propriamente dito do identificador, codigoTipo se refere ao valor numérico equivalente ao tipo do identificador, o escopo define se o identificador é global ou local, a categoria especifica quando o identificador é uma função, protótipo, parâmetro ou variável, o zumbi age como uma *flag* que atribui valor zero para todos os parâmetros de uma função e a qntdParam age como um contador que sempre é incrementado quando existem mais de um parâmetro para uma determinada função.

Assim como nas fases anteriores do compilador, a tabela de símbolos também teve seus arquivos chamados de gerenciador.c e gerenciador.h criados. Para manipulação dos identificadores, foram implementadas as seguintes funções no gerenciador da tabela de símbolos:

1. inserirTabela: responsável por inserir os identificadores na tabela de símbolos. Na fase do sintático, ela era chamada sempre que um identificador era encontrado no código.
2. imprimirTabelaSimbolos: responsável por exibir todos os identificadores da tabela de símbolos.
3. removerItemTabSimbolos: responsável por remover os identificadores que possuem escopo local, não são parâmetros de funções e possuem o valor de zumbi igual a zero na tabela de símbolos.
4. gerenciadorTabSimbolos: responsável por controlar as operações de inserção e remoção dos identificadores da tabela. Essa função é chamada no sintático sempre que um identificador é encontrado, a fim de manipulá-lo.

6. Gerador de Código da MP

A geração de código foi baseada nos documentos disponibilizados pelo professor no avate. Em primeiro lugar, foi criada as funções de criação e fechamento de arquivo para possibilitar a validação ao passo que os comandos de “prints” fossem sendo inseridos no código de análise sintática. Em seguida, foi inserido as inserções com o menor grau de complexidade, por exemplo, *INIP* e *HALT*.

Após isso, foi inserido os “prints” de alocação e desalocação das variáveis, através dos comandos *AMEM* e *DMEM*. Posteriormente, foi inserido os “prints” relacionados a funções e procedimentos, no caso, o *INIPR* no início do procedimento. Além disso, foi implementado a função para adicionar os *LABEL* A tabela de símbolo também precisou ser utilizada na geração de código, onde foi inserido uma nova coluna na tabela chamada “Label” e também para identificar o local de armazenamento das funções, procedimentos e variáveis. Depois da criação das funções relacionadas a *LABEL*, foi identificado os pontos código onde era necessário inserir os “prints” de *GOTO* e *LABEL*. Devido ao tempo, não foi possível completar as outras inserções da máquina de pilha. O tempo utilizado para a implementação da análise semântica foi muito maior do que estimado.

7. Conclusão

Ao término deste projeto do compilador Cmm, alguns pontos merecem destaque. O primeiro ponto diz respeito ao conhecimento que nos foi agregado durante o processo de desenvolvimento. As especificidades de cada etapa foram exploradas minuciosamente com o objetivo de entender o comportamento de cada fase do compilador, embora o projeto esteja parcialmente concluído.

O projeto foi iniciado com a implementação do analisador léxico, seguindo para o analisador sintático, onde nessas primeiras etapas, o processo de desenvolvimento foi concluído. Porém, ao chegar na fase do semântico, as dificuldades para implementá-lo foram maiores. Contudo, desenvolvemos algumas funções que foram inseridas no analisador sintático, além de termos inseridos alguns comandos da máquina de pilha para geração de código.

Quanto aos impactos para a nossa formação acadêmica, o projeto do compilador permitiu uma compreensão mais ampla de como funciona as linguagens de programação, diferenciar o que são interpretadores e compiladores, bem como entender as diversas aplicações que podem ser desenvolvidas utilizando a teoria de compiladores. Após a implementação desse projeto foi possível perceber a importância de ter conhecimento teórico antes de começar a fase de implementação, pois sem o estudo da teoria previamente e esquematização do que deveria ser implementado e como a implementação seria feita, provavelmente não seria possível realizar o projeto.

Em diversos momentos durante a implementação do projeto foi necessário recorrer a literatura e documentos disponibilizados pelo professor da disciplina. Com isso algumas decisões de projeto precisaram serem alteradas. Além disso, existiu uma facilidade em identificação de eventuais erros, pois com o conhecimento aprofundado era possível deduzir que tipo de erro tinha ocorrido e em que ponto do código era preciso revisar ou reescrever.