UNEB
UNIVERSIDADE DO ESTADO DA BAHIA

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.

FELIPE ANDRADE SILVA E LUCAS FREITAS SILVA

2017.2

1. Introdução

O presente relatório apresenta um estudo e detalhamento em relação a construção de um projeto compilador com regras léxicas, sintáticas e semânticas em relação a linguagem especificada Cmm. No qual buscamos relacionar os conceitos e arcabouço teórico aprendido em sala de aula ao longo da disciplina e transformamos o que aprendemos em um compilador que lê, reconhece e gera código de máquina de pilha para um programa fonte descrito em Cmm.

Na contemporaneidade, para a construção de softwares são necessárias diversas ferramentas, entretanto, uma delas é de suma importância, que é o processo de compilação desse software, no qual o desenvolvedor escreve um código em linguagem de alto nível e o mesmo é traduzido pelos compiladores e transformados em códigos de máquinas e isso vem da literatura que define um compilador como programas de computador, que tem como objetivo transformar uma linguagem fonte em linguagem alvo, a grosso modo isso é a forma que o computador entenda o que o desenvolvedor quer que ele faça. A literatura define. Neste sentindo compreendemos a importância de um desenvolvedor que realmente saiba o que está fazendo busque estudar todo esse processo, pois uma pessoa desenvolver software de qualquer forma, pode ocasionar em softwares de má qualidade, não atendendo os requisitos necessários.

Em relação ao nosso projeto em específico, primeiramente foi feito um estudo em relação a teoria da disciplina envolvendo o funcionamento conceitual de um compilador, tais como as partes da compilação (Análise e Síntese) e suas fases (Codificação, Análise léxica, Análise Sintática, Análise semântica, árvore anotada, geração de código intermediário, otimização e geração de código objeto) e as principais definições para especificação e tratamento de tokens. Em seguidas fomos para a parte prática para desenvolver o Analisador Léxico e posteriormente de acordo com que aprendemos em sala de aula fomos desenvolvendo o Analisador Sintático, a tabela de símbolos, analisador semântico e a Máquina de pilha.

As etapas de desenvolvimento foram realizadas de forma interativa e incremental, de acordo com as datas estipuladas pelo professor, entretanto só iremos apresentar aqui os prazos a partir da análise sintática, que foi quando começamos a trabalhar juntos. Começamos a trabalhar juntos a partir do dia 04/11/2017 foi quando começamos a montar uma estrutura lógica para o funcionamento do sintático e finalizamos a análise sintática no dia 16/11/2017. Em relação a uma estimativa de horas, nós não conseguimos contabilizar, mas ao seguir os nossos commits no github, podemos afirmar que nos dedicamos ao projeto nas seguintes datas 11/11/2017, 12/11/2017, 13/11/2017, 14/11/2017, 15/11/2017,. 16/11/2017. O tempo médio que gastamos em cada data foram de 2 à 8hrs dia, mas não conseguimos afirmar quais com precisão. Por fim esse usamos esses tempos para desenvolver toda a análise sintática e tabela de símbolos, incluindo o estudo dos conceitos, implementação, testes e correções.

No que diz respeito ao desenvolvimento do restante do projeto, iniciamos a análise semântica

30/12/2017 e trabalhamos efetivamente nos dias 04/12/2017, 05/12/2017, 07/12/2017, 08/12/2017, 10/12/2017, 11/12/2017, 12/12/2017, 13/12/2017, 14/12/2017 e 15/12/2017. O tempo médio aqui foi maior de 3 à 12hrs por dia., no qual desenvolvemos a análise semântica a geração de código, implementação, testes e correções. O período que ficamos "ocioso" de uma etapa para outra, era o tempo que estávamos seguindo em sala de aula os novos conceitos que iriam surgindo, além de revisarmos em casa.

2. Análise Léxica

A análise léxica é a primeira fase do compilador, no qual buscou-se fazer uma varredura dos tokens que podem ser interpretados por um parser. Havendo essa fase sido realizada individualmente, não houve uma análise rigorosa para a escolha de qual projeto utilizar, optou-se pela utilização do projeto inicialmente desenvolvido pelo membro Felipe.

Antes de iniciar a implementação do analisador léxico, foi necessário a criação de um AFD (Autômato Finito Determinístico), apresentado na Figura 1. Este foi gerado considerando as regras léxicas mencionadas na especificação do projeto. O AFD pode ser comparado a um "mapa" para o projetista, no qual os símbolos reconhecidos pela linguagem e os estados são representados. Assim como no mapa o AFD parte de um ponto inicial, que no caso é o estado 0 no centro da figura 1. A partir deste estado é possível avançar para os demais estados. No entanto, enquanto em um mapa o percurso realizado se dá de acordo com o objetivo do explorador, no AFD o percurso se dá de acordo com os tokens recebidos durante o processo de compilação. Por tanto, partindo do estado 0, com exceção do Enter, Espaço e Tab, cada token encaminha para um estado diferente. Há, por tanto, estados terminais e não terminais, sendo os terminais aqueles que identificam uma cadeia, enquanto que os não terminais são os que não identificam cadeias, mas revelam sua importância ao viabilizar a correta identificação das cadeias. Mantendo a analogia com o mapa, os estados terminais seriam o local que se pretende chegar e os não terminais encruzilhadas nas quais se deve decidir qual caminho tomar.

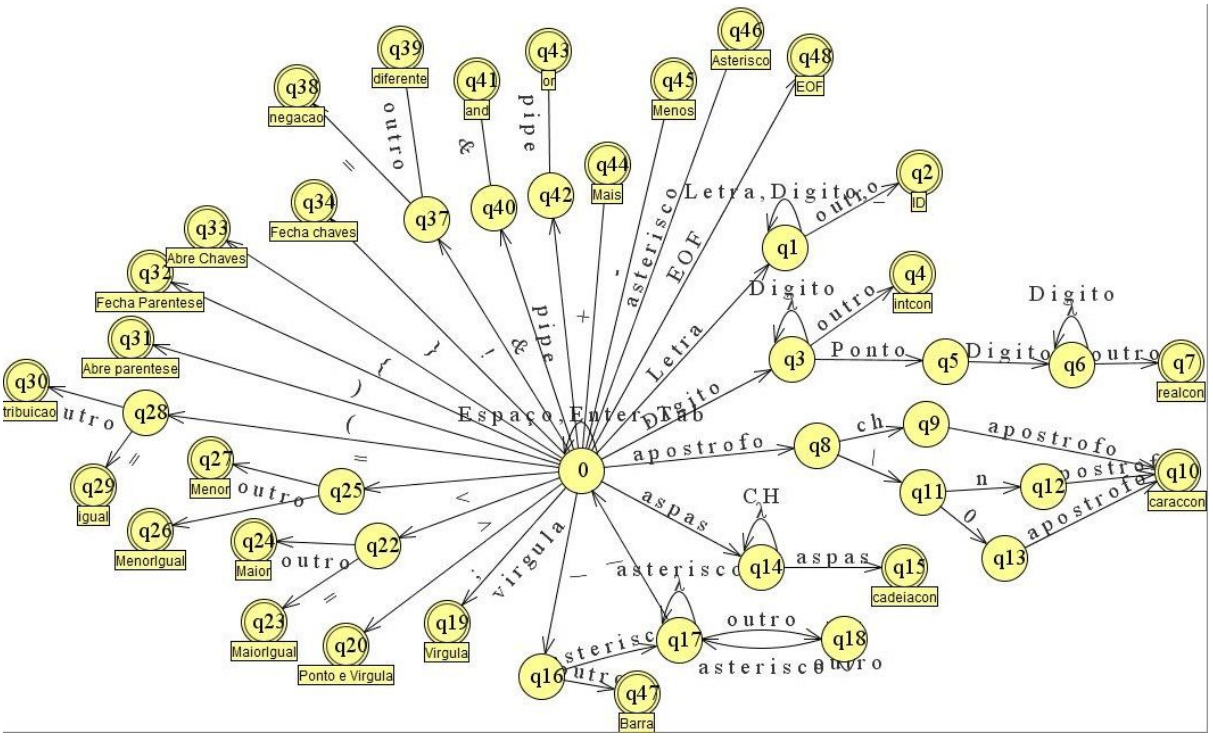

Figura 1: Autômato Finito Determinístico (AFD)

O desenho do AFD foi criado utilizando o *software* chamado JFLAP. As estruturas referente ao analisador léxico estão no Analex.c e Analex.h.

O header do analisador léxico ficou responsável por armazenar as categorias (indicando se é indentificador, palavra reservada, inteiro, etc.), estrutura de sinais (and, or, soma, multiplicação, divisão, etc), estrutura de palavra reservadas. E por fim a estrutura Token, que armazena a categoria do token, seu lexema, seu código e seus valores inteiro ou real caso fosse um número, que foi útil para as análises posteriores.

O padrão adotado para a implementação em código do equivalente ao AFD foi a utilização de switch case (na linguagem C) para representar os dois tipos de estados. Foi utilizado uma estrutura de laço do tipo *while* para fazer a leitura dos tokens e a depender do estado armazenado em uma variável de controle (variável que indicava o estado que deve ser executado no switch case) o token era identificado e a variável de controle tinha seu valor alterado para o próximo case. Este próximo case estava de acordo com os estados do AFD apresentado na figura 1 e de acordo com o token que o direciona, também como apresentado na figura 1. Por fim, se um token não estivesse de acordo com o estado indicado no momento era identificado um erro léxico que seguia para ser tratado. Por outro lado, se o token estivesse de acordo e levasse a um estado final seria feita uma identificação de toda a cadeia formada.

3. Análise Sintática

A análise sintática é a fase no qual determina se a cadeias de tokens gerada pelo analisador léxico pode gerar uma gramática. No qual a implementação da análise sintática se deu através da gramática disponibilizada pelo professor em sua versão 2.2 do Cmm e foi implementada de acordo com suas regras de produção, seguindo à forma de implementação descendente recursiva. Foi quando começamos a trabalhar juntos de fato.

Definimos o módulo sintático criando dois arquivos ao projeto Sintático.c para implementação das estruturas de dados e Sintático.h servindo como um intermediário de comunicação entre os arquivos (posteriormente mudamos esses nomes para AnaSin.c e AnaSin.h, para padronização). A implementação seguiu de forma que: Os procedimentos foram associados a cada símbolo não-terminal da gramática e foram executados de acordo com às ordens de ocorrência dos mesmos, respeitando suas diversas regras de produção. Logo, cada função implementada no analisador sintático, diz respeito à sua regra de produção, exemplo a função Boolean tipo(), diz respeito a regra de produção da gramática tipo, e void atrib () diz respeito à regra atrib.

Primeiramente definimos uma estrutura de Boolean em C, pois o mesmo não oferece suporte a esta estrutura, foi uma escolha para facilitar nossa implementação, já que escolhemos fazer nossa ASDR de forma que, à medida que o compilador fosse criando as árvores sintáticas, iria armazenando o valor de verdadeiro ou falso, pois assim quando o retorno da função fosse falso, indicaria que aquela estrutura não atendeu a regra de produção proposta, apontando o devido erro para a estrutura em questão, e quando o retorno fosse verdadeiro, o código escrito pelo programador iria seguir seu curso normal, pois ele atendeu a regra de produção associada aquela estrutura, exemplo: caso ele colocasse a expressão 1+1, logo na função Boolean expr(); iria retornar verdade, indicando que a expressão está correta.

Uma outra questão de escolha de projeto adotada, foi a de realizar as comparações para a análise sintática utilizando a própria características dos Tokens, pois assim, além de deixar o código organizado, ficava ao mesmo tempo documentavel e legível. Um exemplo disto é (Token.cat == PR && Token.tipo.codigo == CARACTER), isso indica que: Uma pergunta, questionando se o Token atual é uma palavra reservada e o seu código é um CARACTER.

4. Análise Semântica

A análise semântica foi implementada no mesmo arquivo da análise sintática, porém, respeitando a sua modularização. Tomou-se o cuidado de manter a separação entre o que seria análise sintática e análise semântica em termos de codificação. Na análise semântica, utilizamos massivamente a tabela de símbolos, que fora desenvolvida na análise sintática, porém apenas o necessário para o desenvolvimento do projeto que são as operações básicas (Inclusão, listagem, seleção, exclusão).

Nesta parte começamos a trabalhar também com uma quantidade maior de variáveis globais, principalmente para nos indicar algumas situações que iriam ser utilizadas para comparações semânticas. Temos como exemplos: tipo_proc: indica se é um procedimento ou uma função, assim podemos indicar erros no retorne com mais facilidade; posicao_parametros: indica a posição correta nos quais os parâmetros foram alocados, posteriormente foi utilizado para saber se a posição do parâmetro está na posição correta.

De acordo com o que interpretamos das regras semânticas, muitas realizaram comparações, principalmente no que diz respeito às funções, então decidimos implementar uma estrutura que apelidamos de fp, que ficou encarregada por armazenar os dados referente as assinaturas das funções/procedimentos, tais como seu tipo, nome, número de parâmetro e suas devidas posições.

Em relação às funções da análise semântica, destaca-se a verificar_consistencia_tipos, que reflete nas devidas comparações entre os tipos exigidas nas regras semânticas, além de indicar qual é o tipo de retorno da função. Este tipo de retorno é utilizado em outra validação semântica (se o retorno da função casa com o tipo da declaração). Como dito anteriormente, a tabela de símbolos foi essencial para a implementação das regras semânticas, sendo assim implementamos as seguintes funções: declarado_na_tabela_simbolos() indica se a variavel foi declarada no programa.

Gerenciador de Tabela de Símbolos e Gerenciador de Erros

O gerenciamento da tabela de símbolos foi desenvolvido de acordo com a orientação do professor. Nós a implementamos em forma de vetor, mas seu funcionamento se deu como a estrutura de dados pilha, no qual foi armazenado as informações dos identificadores. Onde criamos uma estrutura tabela, que ficou encarregada de armazenar os nomes, tipo, escopo, tipo do símbolo (indica se é variável função, função protótipo), zumbi, a quantidade de parâmetros caso fosse uma função/procedimento, label e endereço relativo.

Suas principais funções foram a de adiciona_tabela_simbolos que ficou responsável por adicionar o identificador na tabela, pesquisar_tabela_simbolos responsável por realizar a consulta na tabela de símbolos e excluir_tabela_simbolos, que diz respeito a retirar os elementos que foram adicionados na tabela anteriormente. As demais funções que foram desenvolvidas neste módulo diz respeito a estruturas para auxiliar e até mesmo realizar análise semântica como dito anteriormente.

Para o gerenciador de erros criamos uma estrutura para armazenar todos os erros, tanto os sintáticos como os semânticos em forma de um eneum, pois assim nós não precisamos nos preocupar com os valores atribuídos aos erros, em razão disso quando for identificado o erro de acordo com as regras sintáticas ou semânticas do Cmm, recorremos à este módulo e

invocamos seu devido eneum através de um case para apontar a mensagem de erro correta na tela para o programador.

5. Gerador de Código da MP

A implementação da geração de código da MP, se deu através de impressão de printf's em um arquivo nos seus devidos locais, onde primeiramente estudamos os textos em relação a geração de código disponibilizado pelo professor. Nesta etapa também utilizamos o gerenciador de tabela de símbolos, em especial para armazenar os labels das funções (função: adicionar_label_Tabela()).

No desenvolvimento do gerador de código para máquina de pilha, utilizamos duas funções para fazer o tratamento em relação ao endereço relativo e escopo da variável (retorna_endereco_relativo e retorna_escopos, ambas encontra-se no GerenciadorTS.c), para quando fosse indiciado o comando LOAD, retornasse o endereço correto.

Aqui se destaca o uso de variáveis globais para fazer o controle em relação aos números dos labels, pois para cada GOTO, GOTRUE e GOFALSE, precisamos saber para qual label essas instruções iriam "saltar", então tivemos uma variável num_label, que era incrementada toda vez que que é gerado um novo label. E depois pegamos o valor atual dessa variável para armazenar na instrução correspondente. Assim realmente o código da máquina de pilha gerava o GOTO lx , LABEL lx corretamente. Também implementamos uma função gera_label(), responsável só por gerar novos labels em cada local que ela era chamada e escrever no arquivo gerador (onde seria escrito os códigos da máquina de pilha) e nessa função que incrementamos a variável num_label. Notamos que algumas gerações de código de expressão ficariam muito extensa, e poderia poluir muito nosso código atrapalhando nossa "visão", então transformamos elas em funções no qual passamos a condição correta e a função gerador_Codigo_Expr() fica responsável por chamar o gera código correto e escrever no arquivo. O uso de variáveis globais similarmente foi utilizado para contar quantas variáveis foram armazenadas em escopo local e global, para quando fosse gerado o código de alocação e desalocação, mostra-se a quantidade corretamente.

Foram realizados testes no final desta etapa, onde foram produzidos código_objeto. Este código objeto estará no github caso o professor queira verificar. O código objeto corresponde ao código em cmm:

inteiro func(inteiro a, inteiro b) { retorne a+b; }

semretorno principal(semparam) { inteiro x; x = func(1,2); }

6. Conclusão

O trabalho nos mostrou que um projeto como este exige do desenvolvedor um profundo estudo nas questões conceituais, antes de partir para prática. A prática, por sua vez, não é uma tarefa fácil, pois exige muito do conhecimento do desenvolvedor em relação a linguagem que ele utilizará para construir o seu compilador. Apesar de toda tarefa árdua, no entanto, acrescenta muito conhecimento, pois o aluno explora mais a sua habilidade de programação, além de perceber que é possível, se ele utilizar as estruturas e codificar corretamente de acordo com a literatura, construir um compilador que atenda à suas próprias expectativas. Como um projeto de software, uma das questões mais difíceis enfrentadas pela dupla foram as massas de testes, pois achamos que por mais que tenhamos codificado seguindo a especificação acabou não funcionando 100% corretamente na correção da análise sintática, deixando o questionamento: será que as massas de testes utilizadas pela dupla foram suficientes? neste caso não. Sendo assim, uma sugestão que deixamos é a criação de uma plataforma que tenha todos os casos de teste, onde o aluno faça o upload do seu projeto, a plataforma iria realizar todos os testes e indicar a porcentagem concluída do seu trabalho e se é possível indicar quais são os erros que o compilador desenvolvido está dando de acordo com a linguagem especificada. Ex: faço o upload do projeto, a plataforma retorna: Seu compilador está 85% correto, e não passou nos seguintes casos de teste: colocar os casos de teste aqui. A participação de um projeto como este é de suma importância para um aluno de computação, pois assim ele sai do amadorismo e começa e exercitar um pouco insights profissionais, pois agora saberá como funciona o processo de compilação e buscará programar de forma otimizada.