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.

João Víctor e Marcus Filipe
2017.2

1. Introdução

O projeto do Compilador Cmm foi desenvolvido em três fases principais, sendo elas análise léxica, sintática e semântica. Na análise sintática, assim como na semântica, tivemos fases adicionais construindo a tabela de símbolos após a verificação do código sintaticamente e gerando código ao longo do processo de compilação. Isso foi possível pois o modo de desenvolvimento de um compilador é, por natureza, modular.
No total, foram dedicadas aproximadamente 200 horas de trabalho ao desenvolvimento deste projeto, incluindo pesquisa, planejamento, codificação, testes e resolução de erros.

2. Análise Léxica

A primeira fase da análise léxica é a criação de um Autômato finito determinístico, construído em cima da linguagem alvo, e que tem como função estabelecer os passos necessários para a formação de um token da linguagem. É com base nele que a codificação será feita, excluindo caracteres inválidos e reconhecendo cadeias de caracteres válidos de acordo com a especificação da linguagem. A segunda fase é a codificação propriamente dita, nessa parte, optamos por utilizar o *enum*, para enumerar as categorias e os códigos dos tokens, a fim de facilitar a atribuição dos valores e a clareza do código. Utilizamos a leitura de um arquivo, cujo preenchimento é feito pelo usuário, e percorremos ele com o *file descriptor* montando gradativamente cada token. Caso ocorra uma parte no arquivo que não tenha significado pra linguagem, o programa emite o aviso de erro e aborta a execução do programa.

3. Análise Sintática

A segunda parte de desenvolvimento do compilador foi a Análise Sintática, aqui, a ideia é que o compilador verifique se os tokens seguem uma determinada ordem pré-definida pela linguagem, denominada de gramática. A partir desta fase o projeto deixou de ser individual e passou a ser em dupla. Nesta parte, a modularização do compilador ajudou bastante, pois não houve a necessidade de mudar nada no analisador sintático, bem como facilitou a adaptação para o analisador léxico escolhido.

A escolha do analisador léxico ocorreu de forma gradativa, mais como uma forma de padronização. No começo, começamos a codificar de forma genérica, pois como ambos estavam funcionando corretamente, adiamos a escolha. Porém, logo sentimos a necessidade de testarmos o funcionamento do que estávamos desenvolvendo, desta forma optamos por escolher um analisador léxico e então adaptamos todo o código para ele.
O *Anasint* foi construído de acordo com as definições da gramática, começamos convertendo as regras de produção em funções, e dentro delas utilizamos as estruturas condicionais da linguagem C (IF e Else) para determinar se um determinado token aparecia fora da gama de sequências possíveis, se sim, acusando um erro e abortando o programa.
Nesta fase do desenvolvimento, foi incluída a tabela de símbolos, um vetor global de mil posições, somente por questões de didática, mas que simula o funcionamento de uma estrutura de dados conhecida como "pilha". Nessa tabela são inseridos todos os identificadores encontrados, bem como seu tipo, categoria e escopo. Essas informações não foram utilizadas aqui, na análise sintática, mas lá na frente, na semântica.
Para gerenciar a tabela de símbolos foi criado também um módulo separado, que será abordado mais à frente.

4. Análise Semântica

Na fase final de análise, é verificada a "lógica" do código, de acordo com várias regras da linguagem. Como é o caso da verificação de tipos, existência de uma função principal, etc. Para isso, foram criadas algumas funções e verificações, que foram inseridas ao longo do código de análise sintática, porém de forma a manter a modularidade das mesmas.
A principal estrutura de dados da qual a análise semântica se utiliza e da tabela de símbolos. Dessa forma, uma grande parte dessas funções executam variadas operações de consulta e comparação dos dados que estão sendo inseridos com os dados já presentes na tabela de símbolos, de modo a verificar erros semânticos no código analisado. Essas checagens de erros são fundamentais para manter a integridade do programa quando este for, de fato, executado.

5. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

Como foi dito anteriormente, a tabela de símbolos é essencial para a análise semântica. Com ela, foi possível validar muitas regras semânticas presentes na especificação da linguagem. A sua construção foi feita na codificação do analisador sintático sempre quando há declarações, seja de variáveis ou funções. Como seu comportamento é da uma estrutura de dados pilha, a inserção e remoção das variáveis e funções é sempre no topo. Esta estrutura foi escolhida pois sempre que uma função chega ao fim, é necessário remover suas variáveis locais já que é possível declarar uma variável de mesmo nome em outra função. Os parâmetros da função não

são removidos porque serão necessários para verificar a quantidade deles na análise semântica, então colocamos um indicador "*zombie"* para permitir declarações de identificadores homônimos. Antes de começar a desenvolver a parte de análise semântica, quando uma função era declarada através do protótipo não era inserida na tabela de símbolos. Porém, tivemos a necessidade de inserir, para analisar ,na ocorrência daquela função já declarada no protótipo, se ela está de acordo (quantidade de parâmetros e tipo) com o protótipo. Com essa mudança, adicionamos a variável *free* à estrutura da tabela que é do tipo int e serve para simular a sua retir ada da tabela não sendo necessário fazer movimento de dados no vetor.

Para o gerenciamento de erros utilizamos mais uma vez o enum com o nome dos erros. Escolhemos o enum para eliminar a necessidade de ter que saber o código definido para cada erro. E então, através de um switch, mostrar a mensagem de erro correspondente a aquele valor ao usuário.

6. Gerador de Código da MP

Para gerar o código utilizamos o *file descriptor* e ao longo do código da análise sintática, instruções são escritas num arquivo de extensão .obj. Nos orientamos pelos documentos disponibilizados no AVA que mostra o que deve ser gerado para cada comando suportado pela linguagem e criamos os que ficaram como exercício. Nesta fase sentimos a necessidade de mais uma alteração na, já feita, tabela de símbolos adicionando uma string que guarda label. No código gerado, os labels são muito importantes porque através deles que os "saltos",como as chamadas de funções, são orientados. Por isso fizemos a alteração, para que quando uma função é criada seja guardado seu label e assim quando alguma função faça uma chamada para outra, o label da que está sendo chamada é pesquisado na tabela. Aqui percebemos como foi importante seguir a gramática na criação do analisador sintático, porque o processo de geração de código foi norteado pela sintaxe. Assim somente nos preocupamos em fazer a construção das instruções de cada parte separada. Exemplificando, se a função atrib chama expr cada um será responsável em gerar cada parte do código de forma independente.

7. Conclusão

No geral, se envolver em um projeto de desenvolvimento de um compilador muda radicalmente o panorama que se tinha sobre o assunto. Deixamos de ser usuários e passamos a destrinchar a lógica envolvida por trás de algo tão corriqueiro na vida de um estudante de Sistemas de Informação, porém tão desconhecida ao mesmo tempo. Aprender como funciona um compilador é tão básico quanto aprender a programar, entendemos como é possível que uma

máquina consiga interpretar uma linguagem e transformá-la em uma série de comandos, processá-los e executá-los. Enfim, aprendemos a conversar com um ser inanimado.

Neste processo, aprendemos que o desenvolvimento de um compilador é sim um processo que requer tempo e dedicação, porém, a metodologia de desenvolvimento é clara, e se seguida à risca, o resultado final não tem como ser diferente do esperado.

No final, acreditamos que desenvolver um compilador melhorou nossas habilidades como programadores, agora sabemos o que o compilador está realmente dizendo quando emite uma mensagem de erro, aprendemos suas vantagens e limitações.