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.

FRANCIELE PORTUGAL

KEVIN OLIVEIRA

2017.2

1. Introdução

O Projeto Compilador Cmm é um projeto com o objetivo de implementar o compilador da linguagem Cmm, utilizando os fundamentos de um compilador. Para tal, o projeto foi dividido em fases, sendo elas: Análise léxica com o AFD (Autômatos Finitos Determinísticos), Análise sintática com gerenciador de símbolos e a Análise semântica com a geração de código.

A primeira fase consistiu em construir o AFD correspondente a linguagem Cmm e gerar os tokens de um código-fonte com base no mesmo, ou seja, essa fase basicamente consistiu na construção de um analisador que reconhece os símbolos que a linguagem permite e classifica-os. Essa fase é fundamental para a realização da próxima fase, pois ela separa os tokens para que o analisador sintático opere com eles.

A segunda fase consistiu em construir um analisador sintático, ou seja, a segunda fase teve como objetivo construir um analisador que verifica a estrutura de um código-fonte da linguagem com base na gramática da mesma, validando apenas se a estrutura do conjunto de tokens puder ser gerado pela gramática. Nesta fase, também foi realizado o gerenciamento da tabela de símbolos, onde os identificadores passaram a ser armazenados na mesma com suas respectivas informações de nome, tipo, categoria, escopo, entre outras informações. Essas informações foram necessárias para controlar a existência deles no programa, além disso, a próxima fase faz uso da tabela de símbolos para verificar as regras semânticas.

A terceira fase resumiu-se na construção de um analisador para verificar as regras semânticas da linguagem, como por exemplo analisar a utilização de identificadores e expressões. Além de coletar dados, como por exemplo a base e o endereço relativo de identificadores, que são importantes para a geração de código.

Contudo, este relatório apresenta as decisões que foram tomadas ao decorrer do projeto.

2. Análise Léxica

O analisador léxico escolhido para dar continuidade ao projeto foi o analisador do componente Kevin Oliveira por apresentar funcionalidades adicionais que auxiliariam na implementação do projeto. Como por exemplo: a funcionalidade de debugar o processo de reconhecimento dos tokens, sendo possível acompanhar a mudança de estados no AFD e também, a funcionalidade de indicar não só a linha, mas também a coluna em que um erro ocorre através da função *void mensagemDeErro(FILE* fp,char c,int linha,int coluna)*.

As estruturas de dados definidas são referentes ao símbolos que a linguagem permite e ao token em si. A tabela abaixo apresenta-as de forma mais clara.

Estrutura	**Descrição**

struct token	Para armazenar as principais informações de um token, a categoria e seu respectivo código.
char ID_TABLE[1000][100] int ID_TABLE_TOPO	Para controle do armazenamento de identificadores.
enum PR char PR_TABLE[][50]	Indicam as palavras reservadas da linguagem.
enum SN char SN_TABLE[][50]	Indicam os sinais da linguagem.
int CT_I_TABLE[100] float CT_R_TABLE[100] char CT_C_TABLE[100] int CT_I_TABLE_TOPO int CT_R_TABLE_TOPO int CT_C_TABLE_TOPO	Para controle do armazenamento de constantes dos tipos: inteiro, real e caracter.
char LT[1000][100]; int LT_TOPO=0;	Para controle do armazenamento de literal.
enum categoria char tabela_categoria[][100]	Indicam as categorias de tokens da linguagem.
FILE* fp	Para controle do arquivo que possui o código-fonte a ser analisado.

Contudo, a função que realiza a análise léxica retorna o token reconhecido na estrutura do tipo "token" apenas ao chegar em um estado final no AFD da linguagem.

3. Análise Sintática

No processo de construção do analisador sintático, podemos observar que em alguns casos seria necessário não só trabalhar com o token que será tratado (*tokenAtual*), mas que seria necessário ter uma visão a frente. Por isso, criamos variáveis para indicar os próximos tokens (*tokenProx, tokenProxProx*). E, a partir disso, criamos funções para manter o controle sobre essas variáveis, que estão descritas na tabela abaixo.

Função	**Descrição**
void getToken()	Consome o token atual e ordena as variáveis (*tokenAtual, tokenProx e tokenProxProx*).
token viewToken() token viewNext() token viewNextNext()	Apenas retorna o token especificado pelo nome da função, de modo que possamos visualizá-los sem consumi-los.

void mostraTokens()	Imprime as informações das variáveis tokens em sua respectiva ordem: *tokenAtual -> tokenProx -> tokenProxProx*. E chamamos-a sempre que o token atual é consumido com o objetivo de acompanhar o processamento do analisador de forma clara.

Já as funções criadas para realizar análise sintática seguem a lógica das regras da gramática, que quando não obedecidas pelo código-fonte que está sendo analisado, é acusado o erro sintático. Para isto, foi criada a função *void erroSin (char * string)*, que recebe a mensagem que será impressa para que a mensagem de erro não fosse generalizada e acusasse literalmente o erro sintático de forma específica.

Algumas dessas funções da análise sintática tornaram-se funções de checagem, retornando um valor lógico para que o código ficasse mais limpo, consistente, de fácil entendimento e também de fácil manutenção. Sendo elas: *bool tipo(token token)*, que reconhece um tipo, e *bool op_rel(token token)*, que reconhece um operador relacional. Além disso, outras funções de checagem, descritas na tabela abaixo, também foram criadas com o mesmo objetivo e tornou-se essencial principalmente por ser utilizada muitas vezes no código.

Função	**Descrição**
bool categoria(token token, int cat)	Compara a categoria do token com a categoria esperada.
bool sinal(token token, int sinal)	Verifica se o token é sinal e caso seja, compara o sinal do token com o sinal esperado.
bool id(token token)	Verifica se o token é um identificador.
bool pr(token token, int pr)	Verifica se o token é uma palavra reservada e caso seja, compara a palavra reservada do token com a palavra reservada esperada.

Além das funções descritas acima, o analisador sintático também apresenta uma funcionalidade de debug indicada por uma variável booleana *debugSin*. Se a mesma estiver ativa, em toda função chamada para a análise da estrutura do programa a partir das regras gramaticais é impresso na tela qual função começará a ser executada, as variáveis de controle dos tokens (*tokenAtual, tokenProx e tokenProxProx*) e ao chegar no final da execução da função, alerta a saída da mesma. Com isso, é possível acompanhar o processo sintático e a situação dessas variáveis antes da função ser executada.

4. Análise Semântica

Para verificar se um identificador, seja variável, função ou parâmetro, está duplicado no mesmo escopo (global ou local), a função *void verificaIDGlobalDuplicado(int id)* e *void verificaIDLocalDuplicado(int id_inicio, int id)* é chamada toda vez que um novo identificador é armazenado. Basicamente, essa função verifica se o nome do identificador e o escopo são iguais. Sendo que se isso for verdadeiro, o analisador semântico acusa um erro de redefinição da mesma.

A função *void verificaExistenciaDaFuncao(char * nomeFunc)* verifica se uma função existe a fim de que controle a tentativa de chamada de uma função que não foi declarada ou que não possui corpo. Já a função *int verificaTipoTabelaDeSimbolos(char * nome)* retorna o tipo de um identificador, se encontrado. Caso contrário, um erro é acusado, informando que o identificador não possui referência. Essa função foi criada para auxiliar a verificação de compatibilidade de tipos dos parâmetros realizada pela função *void verificadorDeTipos(token token,int num_param)*.

A função *void verificadorDeTipos(token token,int num_param)* recebe o número do parâmetro e utiliza a posição do identificador da função, pois tendo a informação da posição sabe-se que o(s) próximo(s) símbolo da tabela, se forem da categoria parâmetros, o pertencem. Para isso, foi criada a variável *int id_funcao_atual* que armazena a posição de uma função na tabela de símbolos para que caso hajam parâmetros, os mesmos sejam localizados. E também, foi criada a variável *int numParamFuncao* que é incrementada apenas caso o próximo token seja uma vírgula, pois significa que há mais parâmetros e é igual a 1 quando após o primeiro parâmetro, o próximo token é um fecha parênteses, indicando o fim dos parâmetros da função. Já a função *void verificaQuantidadeDeParametros(int num_param)* verifica a quantidade de parâmetros numa chamada de função. No entanto, em toda chamada de função sua assinatura é verificada em relação à ordem, quantidade e tipo.

Diante da situação de identificadores opcionais na assinatura das funções do tipo protótipo, foi criado o campo *bool sem_nome* dentro da estrutura da tabela de símbolos com o objetivo de armazenar suas informações já que seria necessário realizar a verificação dos tipos dos parâmetros numa chamada de função.

A função *void trocaTipo(int novoTipo)* tem como objetivo verificar a compatibilidade de tipos. Essa função conta com o auxílio da variável *int tipoAtualUtilizado* , que é inicializada com o valor -1 para indicar que um tipo para a checagem não foi definido. Nesse caso, ela é atualizada para o novo tipo recebido por parâmetro, pois todos os outros identificadores ou constantes devem ser compatíveis com esse tipo. Dessa forma, caso não seja compatível, um erro semântico é sinalizado: "Expressão composta de tipos incompatíveis". Ao final do processo, o valor dessa variável retorna a possuir o valor -1.

A função *void verificaTipoDeRetorno()* tem como propósito verificar o valor de retorno das funções, incluindo o caso de funções do tipo *semretorno*. Essa função utiliza a função *int id_corpo_de_funcao_atual()* para verificar o tipo de retorno da função a qual ela retornou a posição. Contudo, essa função segue as regras semânticas especificadas, como por exemplo caracter é compatível com inteiro. Já a função *verificaRetornoObrigatorio()* apenas verifica se a função é do tipo *semretorno* para não obrigá-la a possuir retorno definida através da variável *possui_retorno* que funciona como uma flag.

A função *void verificaPrototipoCompativelComFuncao()* compara se uma função e a sua assinatura são iguais, considerando nome, tipos e retorno, indicando erro quando nenhuma dessas características são atendidas.

5. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

O gerenciador de tabela de símbolos consiste no controle dos identificadores. No entanto, criamos uma estrutura de dados que oferecesse esse suporte, *struct struct_tabela_de_simbolos*. Veja a tabela abaixo com detalhes da estrutura interna da mesma.

Estrutura *struct_tabela_de_simbolos*	**Descrição**
char nome[20]	Armazena nome do identificador.
int tipo	Armazena o código do tipo do identificador, indicado pelas estruturas *enum tipoTS* e *char tipoTS_nomes[][20]*.
int categoria	Armazena o código da categoria do identificador, indicado pelas estruturas *enum categoriaTS* e *char categoriaTS_nomes[][20]*.
int escopo	Armazena o código do escopo (global ou local) do identificador, indicado pela estrutura *enum escopoTS*.
int enderecoRelativo	Armazena o endereço relativo do identificador, que é dado de acordo com o seu escopo, indicado pelas estruturas *int contador_simbolos_globais* ou *int contador_simbolos_locais*.
bool zumbi	Armazena um valor lógico, que será true quando a categoria do identificador é parâmetro.

bool sem_nome	Armazena um valor lógico, que será true apenas quando a função possui tipo e não possui o nome do identificador.

O campo bool zumbi foi criado com o objetivo de identificar os parâmetros para que não fossem excluídos da tabela ao final da função da mesma, pois na análise semântica seria necessário verificar o tipo dos parâmetros de uma função ao ser chamada.

O campo *bool sem_nome* foi criado na fase da análise semântica devido ao caso de função, onde em parâmetros é permitido apenas colocar o tipo do parâmetro, ou seja, onde não tem o nome do identificador, mas tem seu tipo. Isso porque seria necessário saber o tipo de parâmetros aceitos pela função, uma vez que a função é chamada a fim de verificar a compatibilidade entre a chamada da função e a função.

No entanto, as funções criadas para realizar o controle da tabela de símbolos estão descritas abaixo:

Função	Descrição
void mostraSimbolo(int id)	Imprime o símbolo que corresponde a uma posição.
int armazenar_simbolo(int escopo,int categoria, int tipo, char nomeID[20])	Armazena o símbolo na tabela *tabela_de_simbolos*, onde o topo é indicado pela variável *int topo_tabela_de_simbolos* que sempre armazena a próxima posição.
void mostraTabela()	Imprime todos os símbolos armazenados na tabela de símbolos.
void limparSimbolosLocais(int i)	Remove os símbolos locais armazenados na tabela de símbolos, trocando o valor do topo da tabela. Uma variável foi criada para auxiliar neste processo. Essa função é chamada ao final de uma função.

A variável *int temp_id* foi criada na função *prog()* e armazena o valor topo da tabela logo no início da função após os parâmetros já terem sido armazenados. Isso porque os identificadores da categoria parâmetro não devem ser removidos. Por exemplo, uma função possui 2 parâmetros que ocupam as posições 0 e 1 na tabela de símbolos e o topo aponta para a próxima posição (2), a variável *temp_id* armazena o valor 2, pois é para esta posição que o topo deve voltar, já que o topo sempre aponta para a próxima posição, a fim de remover as variáveis locais da função ao fim da mesma.

O gerenciador de erros teve a mesma estrutura no analisador sintático e semântico. Basicamente, é uma função que recebe uma string que contém a devida mensagem de erro. Dessa forma, a mensagem de erro poderia ser mais específica para que não fosse uma mensagem generalizada a todos os erros. Contudo, essas funções são: *void erroSin(char * string)* e *void erroSem(char * string)*. Entretanto, para o analisador léxico esse gerenciador de

erros se comportou de forma diferente, indicando apenas em qual linha e coluna o erro léxico ocorreu. Sendo ela: *void mensagemDeErro(FILE* fp,char c,int linha,int coluna).*

6. Gerador de Código da MP

O gerenciador de código da Máquina de Pilha não foi construído por motivos relacionados ao tempo.

7. Conclusão

O projeto Compilador Cmm só não ficou completo por faltar a geração do código da MP. Entretanto, sua evolução foi acompanhando a compreensão dos fundamentos explicados em sala de aula na disciplina Fundamentos de Compiladores, que contribuiu para fixar esses fundamentos de forma mais prática através da aplicação dos mesmos neste projeto.

É fundamental esclarecer que sem esses fundamentos seria quase inviável realizar este projeto, o compilador em si, pois o mesmo se tornaria extremamente complexo implementando-o por tentativas e enfrentando muitos erros, já que vários casos precisam ser cobertos de acordo com as regras.

No entanto, observamos que todas as fases estão relacionadas e que uma depende da outra. A fase inicial da análise léxica exigiu bastante planejamento e compreensão dos conceitos, principalmente diante do funcionamento dos reconhecedores de cadeias de uma linguagem regular. Na segunda fase, entende-se realmente a dependência entre as fases e a necessidade delas existirem, pois utiliza os tokens separados pelo analisador léxico para tratar a estrutura de um conjunto de tokens. Além disso, é nessa fase que realmente é necessário possuir os fundamentos da gramática para entender as regras gramaticais da linguagem projetada para realizar este compilador. E na terceira fase, basicamente verificamos e tratamos tipos.

Contudo, a disciplina Fundamentos de Compiladores contribuiu para compreensão dos conceitos relacionados ao processo de compilação dos programas e também para compreensão da metodologia para execução de um projeto de compilador através da aplicação dos conceitos, passando por cada fase na implementação de um compilador.