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.

JOSÉ DIÔGO DA SILVA CARNEIRO

FERNANDO AZEVEDO MAIA JUNIOR

2017.2

1. Introdução

Este documento tem o objetivo de apresentar uma visão geral do processo de construção de um compilador para a linguagem de programação Cmm. Para o desenvolvimento do projeto foi adotada a linguagem de programação C e o ambiente de desenvolvimento integrado Code Blocks.

O período de construção do projeto foi de 3 meses tendo seu início em Setembro e término em Dezembro de 2017, com uma estimativa de 85 horas de dedicação ao projeto.

A construção do compilador Cmm foi dividido em três fases de construção, análise léxica, análise sintática e análise semântica. Cada fase será destrinchada nas seções seguintes.

2. Análise Léxica

A primeira parte da construção de um compilador é chamada de análise léxica, cujo objetivo é ler o código fonte caracter a caracter em busca de identificar os elementos pertencentes a linguagem como: identificadores, palavras reservadas, operadores e eliminar espaços e comentários, facilitando assim o processamento nas etapas posteriores. Ao término da análise léxica um conjunto de tokens é passado para a próxima fase.

Para validação dos Tokens encontrados foi utilizado a estrutura de autômatos finitos, onde a cada leitura de um caracter no arquivo um estado era avançado no autômato, a **Figura 1** demonstra os estados extraídos para a linguagem CMM:

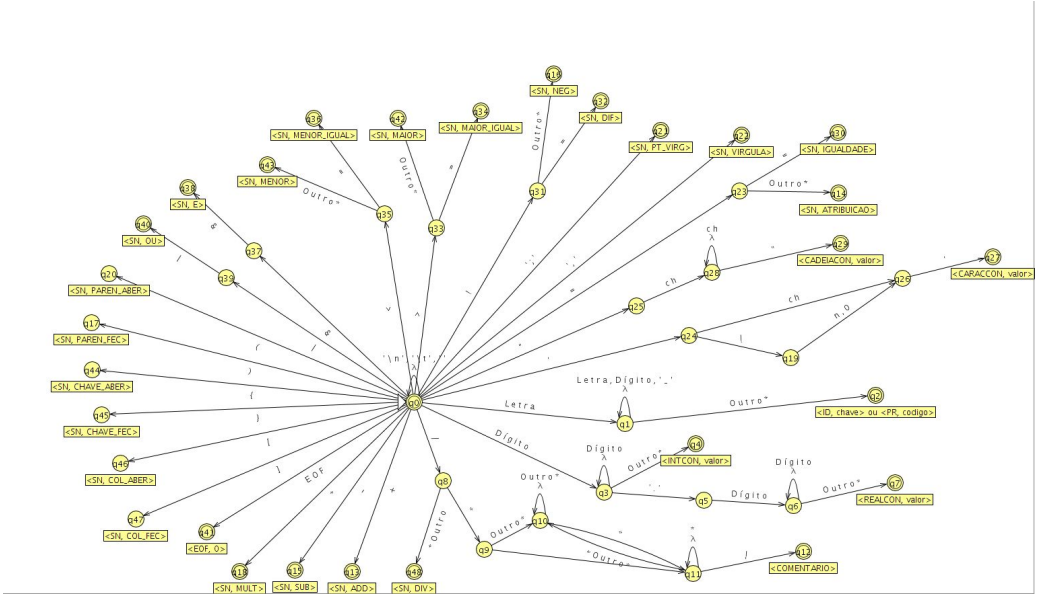

Figura 1: Autômato finito para a representação dos Tokens da linguagem Cmm

Nesta fase a implementação do analisador léxico consistiu em adaptar os estados do autômato para a programação, que foi realizada através da implementação de uma máquina de estado com a estrutura de seleção switch, que mapeava todos os estados disponíveis no autômato para cada nova entrada de caractere. Em Cmm assim como em outras linguagens, seus tokens são comumente maiores que um caracter, para solucionar este problema foi criado um vetor de caractere para servir de buffer, e guardar todos os valores acumulados diante dos estados até que chegasse a um dos estados de aceitação, onde o seu valor é zerado e todo processo recomeçava. A cada token encontrado o mesmo é retornado pela função **check_state** e passado para o analisador sintático, cujo qual vai avaliar a sintaxe, ou seja, vai validar se a sequência de tokens colhidas pelo analisador léxico fazem sentido para a gramática da linguagem. A estrutura **Token** também foi criada para representar cada Token da linguagem Cmm encontrado no arquivo fonte, e está representada na **Figura 2**.

```
// It represents each type of token possible on CMM
enum token_types {ID, PR, SN, INTCON, REALCON, CADEIACON, CARACCON,
                  eOF, COMMENT};
typedef enum token_types token_type;

/**
* This struct holds and identifier token, with it's value and
* table position
*/
typedef struct {
  int table_position;
  char value[300];
} lexem;

/*
** Structure of a Token in CMM language
*/
typedef struct {
  token_type type; //holds the type (integer, character, real)
  union {
    char cValue; // caracter values
    float dValue; // float values
    int iValue; // integer values
    char pr[15]; // keeps reserved words
    char signal[2]; // keeps signal values(+, -, *, / e etc)
    char word[1000]; // literal values
    lexem lexem; // Identifiers
  };
} Token;
```

Figura 2: Representação da estrutura Token

3. Análise Sintática

A construção do módulo de análise sintática foi baseado na análise sintática descendente recursiva seguindo as regras de reprodução da gramática de Cmm. Cada símbolo não-terminal foi representado por funções no arquivo **parser,** algumas destas funções foram implementadas como booleanas para fácil a verificação de existência de seus terminais.

Todas as funções foram implementadas respeitando a gramática de Cmm. Nesta fase também foi implementada a tabela de símbolo, que é responsável por gerenciar todos os identificadores encontrados na linguagem Cmm, variáveis locais, globais, funções e seus parâmetros, para representar esta estrutura foi criada uma **struct symbol** com a sua definição sendo representada na **Figura 3**:

```
enum token_cat{VAR,PARAN,FUNC};
typedef enum token_cat token_cat;

typedef struct{
    char name[500];
    int zumbi;
    token_cat cat; //holds category
    char type[10]; //holds type
    int scope; //keeps identifier scope
    int init; // verify if a function has been loaded
    int matchParam; //boolean verify if
    int fullfill; //boolean helper to handle if a parameter was completed
    char mem_space[50];//keeps mem space
    union {
      char cValue; //caracter values
      float dValue; //float values
      int iValue; //integer values
      int bValue; //boolean values
    };
} symbol;
```

Figura 3: Representação da estrutura símbolo

Para o gerenciamento da mesma foi criado um novo arquivo **symbol_table** que possui todas as funções de **CRUD (**criação, remoção, atualização e leitura**)** dos identificadores**.** Após a validação da estrutura gramatical do analisador sintático, foi iniciado o acoplamento da análise semântica ao mesmo que será destrinchado na próxima seção.

4. Análise Semântica

A análise semântica tem por objetivo verificar se as construções realizadas no analisador sintático estão corretas semanticamente, se os tipos das atribuições são compatíveis, assim como a resolução das expressões e validação de declarações em geral. Todas as validações semânticas descritas na especificação da linguagem Cmm foram implementadas, para o acompanhamento do valor e tipos retornados pelas chamadas das funções **expr, expr_simp, termo** e **fator** foi criado um novo tipo de estrutura no arquivo global de estruturas **structures** chamada **expression,** onde a mesma é responsável por manter o tipo dos retornos (inteiro, real, caracter, booleano), e um campo union que guarda estes valores, para cada tipo de valor tem um tipo representado no **union**, a interpretação da análise semântica foi realizada com o uso da definição dirigida pela sintaxe, onde a análise sintática e a semântica são feitas em um passo, ou seja, a validação semântica é realizada logo após a validação da estrutura sintática, isso é

possível, através da utilização da árvore sintática que é formada através das chamadas recursivas. A estrutura **expression** está definida da seguinte maneira:

```
/*
** A Pair value to store the type and value of an expression
*/
typedef struct{
  char type[10]; //holds the type (integer, character, real, boolean)
  union {
    char cValue; // caracter values
    float dValue; // float values
    int iValue; // integer values
    int bValue; // boolean values
    char word[1000]; // literal values
  };
} expression;

#endif // STRUCTURES
```

Figura 4: Estrutura **expression**

Verificações implementadas: validação de declarações, validação de parâmetros, validação de funções, validação de tipos, verificação de protótipo, geração de espaço de memória, validação de expressões e verificação de existência da função principal no programa. A maior dificuldade na implementação desta etapa foi encontrar uma estrutura geral que pudesse representar tantos os tipos quantos os valores passados em cada fase no reconhecimento de uma expressão, este problema foi resolvido com a estrutura **expression**.

5. Gerenciador de Tabela de Símbolos e Gerenciador de Erros

O gerenciamento da tabela de símbolos como mencionado na seção 3, foi realizado em um arquivo separado mantendo o conceito de **single responsability,** com todas as suas funções de criação, remoção, atualização e leitura implementadas. A tabela de símbolos foi representada por um vetor da estrutura **symbol** demonstrada na figura 3, que para o uso de avaliação da matéria foi criado estaticamente com no máximo 1000 posições.

Para o gerenciador de erro também foi criado um arquivo separado chamado **errors** que possui em seu header constantes padrões para os erros emitidos em todo o programa, a sua função principal é servir de máquina de estados de erro, onde uma constante de erro é passada e a mensagem referente a mesma é exibida no console, logo após o programa é finalizado com status -1.

Após a fase de validação do código fonte, através da análise léxica, sintática e semântica o próximo passo realizado foi a implementação da geração de código através da máquina de pilha que será abordado na próxima seção.

6. Gerador de Código da MP (Máquina de Pilha

Nesta seção iremos abordar os aspectos de maior relevância para a implementação da máquina de pilha, abrangendo as etapas que foram realizadas para a geração de código, explicando o uso das structs, funções e estrutura de dados utilizadas durante o processo, sendo que todos os comandos da geração de código são armazenados no arquivo stack_file.txt.

6.1 Condicionais

Nas condicionais utilizamos como base o documento encontrado no AVA que possui o exemplo de como estruturar a geração de código, contudo os valores LABEL Lx e LABEL Ly são adquiridos logo quando entramos na análise de condicional do analisador sintático localizado na função **cmd**. No comando **Se** utilizamos a função **get_label** para pegar o valor de LABEL Lx para utilizarmos na estrutura da condicional e atribuímos o valor a variável **labelx**. Já na condicional com da estrutura **senão** fazemos uso da variável **labely** que é o valor de LABEL Ly, usado para moldar toda nossa estrutura **senão**.

6.2 Estruturas de Repetição

Para a estrutura de repetição **enquanto** usamos também o documento para estruturar a base fazendo com que, atribuimos o valor de LABEL Lx para a variável **labelx**, o valor de LABEL Ly é representado pela variável **labely** e em ambos os casos os valores dos labels são gerados através da função **get_label**,

Já para a estrutura **para**, os valores que representam, LABEL Lw, LABEL Lx, LABEL Ly e LABEL Lz são previamente definidos em variáveis denominadas: **labelw, labelx, labely e labelz** sendo que esses valores são previamente armazenados para definir a estrutura, pois, quando começamos a analisar uma expressão ou um novo bloco de comandos (componentes definidos no próprio modelo da estrutura) serão atribuídos novos valores para os labels através da chamada de outras funções ou na análise de expressões, então a estratégia de armazenar previamente os valores referente a cada "LABEL" para utilizar durante o processo ajuda a formar o comando tanto os de repetição quantos os de condição.

6.3 Operadores Relacionais

Para a geração de código foram utilizados conceitos prévios adaptados a necessidade que encontramos em algumas situações no uso de operadores relacionais. No caso dos operadores "==","<" e ">" foi utilizado o exemplo encontrado no documento **A Máquina de Pilha (MP) - Parte 1**, disponível no Ava, todavia os operadores "!=",">=" e "<=" foram construídos com base do modelo predefinido dos exemplos que constavam no documento. Para isso foi criada uma função denominada **operator_check** que ao passar uma estrutura do tipo **Token** como

parâmetro identificamos o operador associado a ela e assim imprimimos no documento os comandos daquela operação.

6.4 Operadores Lógicos

6.4.1 Operador AND (**&&**)

Para o operador AND utilizamos em algumas estruturas a variável **aux_and** como passagem de parâmetro, ela é utilizada para caso pelo menos uma operação seja falsa será executado um GOFALSE para o label que identifica o término da execução de um comando, é passada por parâmetro pelas funções **expr, expr_simp, termo e fator** com o objetivo de que ao chegar na função **termo,** caso na expressão seja identificado esse operador lógico, efetuar a devida inserção dos parâmetros na pilha.

```
while(token.type == SN && (strcmp(token.signal,"*") == 0
      || strcmp(token.signal,"/") == 0 || strcmp(token.signal,"&&") == 0)){

    if(strcmp(token.signal,"&&") == 0){
      fprintf(stack_file,"COPY\n");
      fprintf(stack_file,"GOFALSE L%d\n",aux_and);
      fprintf(stack_file,"POP\n");
    }
```

Figura 5: *Parte da função **termo**, onde executa o bloco de comandos ao encontrar um AND*

6.4.2 Operador OR (**ll**)

O operador OR foi estruturado de acordo ao material disponibilizado no AVA, é declarado uma variável na função **cmd** denominada **aux_or** e passamos por parâmetro para as outras funções durante a análise sintática, sendo elas: **expr, expr_simp, termo e fator,** com o intuito de quando chegar na função **expr_simp** caso na operação exista um operador lógico do tipo OR, após analisar uma das expressões, sendo o resultado verdadeiro, ele executa o comando dentro do bloco. Para esse operador sempre antes de executar um bloco de comandos seja dentro de uma condicional ou em uma estrutura de repetição é adicionado um label, pois caso pelo menos uma expressão seja verdadeira daremos um GOTRUE para esse label em específico para que os comandos dentro da estrutura sejam executados.

```
if(strcmp(token.signal,"||") == 0){

  fprintf(stack_file,"COPY\n");
  fprintf(stack_file,"GOTRUE L%d\n",aux_or);
  fprintf(stack_file,"POP\n");
}
```

Figura 6: *Execução do bloco de comandos do operador OR na função **expr_simp***

6.4.3 Operador NOT (**!**)

No operador NOT utilizamos uma função denominada **operator_check_not_iqual** que através da identificação do token "**!**" é ativada a flag "**cont_not_iqual**" para que na expressão após a execução do bloco de comandos dos operadores relacionais atribuídos em uma expressão quando executamos a função **operator_check,** caso o valor daflag seja correspondente executamos imediatamente a função **operator_check_not_iqual**" para que seja executado o bloco de comando da negação que também foi especificado no AVA.

6.5 Variáveis

Para conferir quantas variáveis globais foram declaradas, fizemos uso de um contador do tipo inteiro chamado **global_var** que também é utilizado no final do programa para desalocar o espaço de memória previamente alocado. Já na declaração de variáveis globais criamos uma variável de escopo local na função **prog** que é responsável por armazenar a quantidade de variáveis declaradas de cada tipo para seja impresso no arquivo a quantidade de AMEMs equivalente a respectiva função e em conjunto temos uma variável de escopo global nomeada de **cont_local_var** que armazena a quantidade total de espaços alocados para declaração de variáveis naquele escopo local, para que possa ser desalocado no final do procedimento através de um DEMEM.

6.6 Funções e Assinaturas.

Ao encontrar uma assinatura de uma função armazenamos seu ID através da função **check_id_label_true,** na qual passamos o nome (string) do ID que queremos armazenar, caso o nome não tenha sido armazenado, é chamada internamente a função **get_store_id,** sendo que nessa mesma função esse nome é armazenado em um vetor do tipo **storeid,** que foi uma struct criada com o objetivo de armazenar o nome e label de uma função específica, para facilitar a alocação de um comando CALL quando encontramos uma chamada de uma função, para pegar o label específico de um ID de uma função usamos a função **load_label_id,** na qual passamos o ID como parâmetro e ele retornar o label associado ao início da janela de ativação daquela função em específico.

```
typedef struct storeid{
    char idname[500];
    int labelnumber;
}storeid;
```

Figura 7: *Estrutura utilizada para armazenar o ID e o valor do Label relacionado*

Caso a função não tenha assinatura faremos a chamada de **check_id_label_true** na declaração da mesma, isso não gera nenhum problema, pois mesmo chamando o método **check_id_label_true** duas vezes para efetuar o mesmo procedimento, a função permite inserir uma única vez conservando a integridade da informação. Antes de inicializar a leitura dos comandos de uma função damos um GOFALSE para não seja executado o processo logo de primeira, para isso geramos um novo label com a função **get_label** que retorna sempre o próximo valor do label e armazenamos na variável **global_jump_function,** sendo que ao final da função imprimimos um novo label com o valor armazenado nessa variável. Ao iniciar uma janela de ativação imprimimos o comando INIPR 1 que indica o início da janela de ativação e

ao término de uma função alocamos uma espaço da área negativa da pilha para armazenar o retorno da função, damos um DEMEM para destruir as variáveis de escopo global e damos um RET para o valor de retorno com relação a quantidade de parâmetros que a função possui.

7. Conclusão

A construção desse projeto de forma evolutiva e dividida em etapas, contribuiu muito para a evolução da equipe em questão de foco incremental, ao invés de tentar fazer tudo de uma vez, passamos a ter consciência da necessidade e valor de implementar cada módulo de cada vez e ter certeza que o mesmo já está validado antes de prosseguir para o próximo, mesmo que na etapa seguinte fosse necessário fazer alguns ajustes no que já tinha sido feito. Alguns fatores pessoais acabaram influenciando para que não pudéssemos dedicar mais tempo ao projeto, contudo, conseguimos implementar a metodologia de **sprints** onde separamos determinados períodos em algumas semanas para concluir x tarefas para o projeto, o que nos ajudou bastante no processo. Como sugestão para as próximas edições da disciplina acredito que seja válido a adesão deste modelo de uma prova e duas notas no trabalho,pois ao colocar o conteúdo aprendido em sala de aula em forma prática fica mais intuitivo entender o desenvolvimento do processo ao mesmo tempo que auxilia muito no processo de construção já que o foco não é desviado com um foco mais teórico nas provas e sim um complemento maior entre a teoria dada em sala de aula e a prática aplicada ao desenvolvimento do compilador. Outra sugestão é se possível a liberação dos materiais mais previamente principalmente dos slides para que a implementação venha ser realizada a medida que novos assuntos forem introduzidos.