



SISTEMAS DE INFORMAÇÃO
FUNDAMENTOS DE COMPILADORES

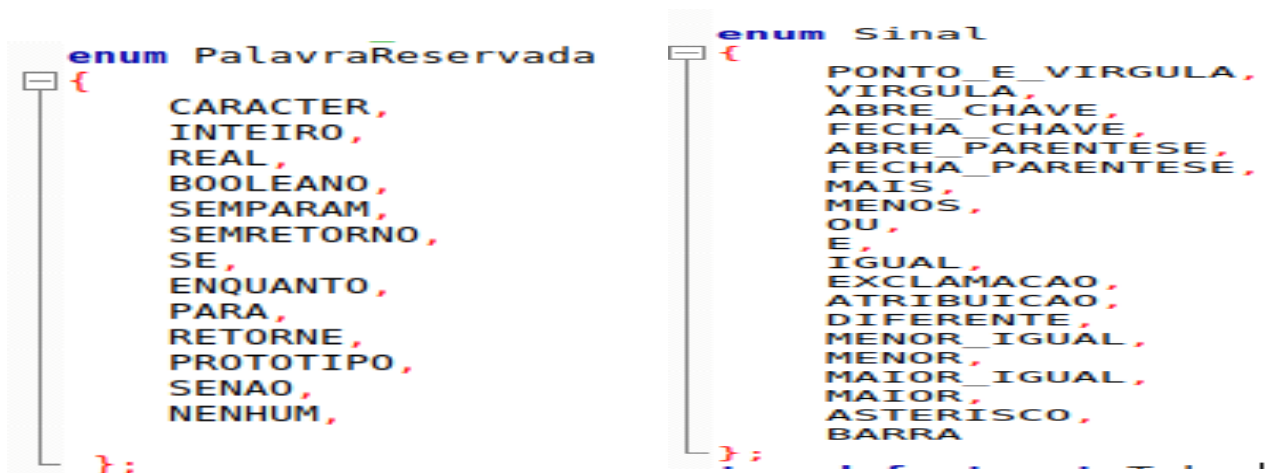
COMPILADOR Cmm

Trabalho apresentado como parte das avaliações parciais da disciplina Fundamentos de Compiladores do curso de Sistemas de Informação do Campus I da UNEB.

CAROLINE DE SOUZA DOS SANTOS

2017.2

Contém quatro estruturas usadas pelo analisador léxico, três estruturas menores que são os `enum` e a estrutura maior que abriga as outras menores contruindo o `Token`. As estruturas pode ser vistas a seguir.

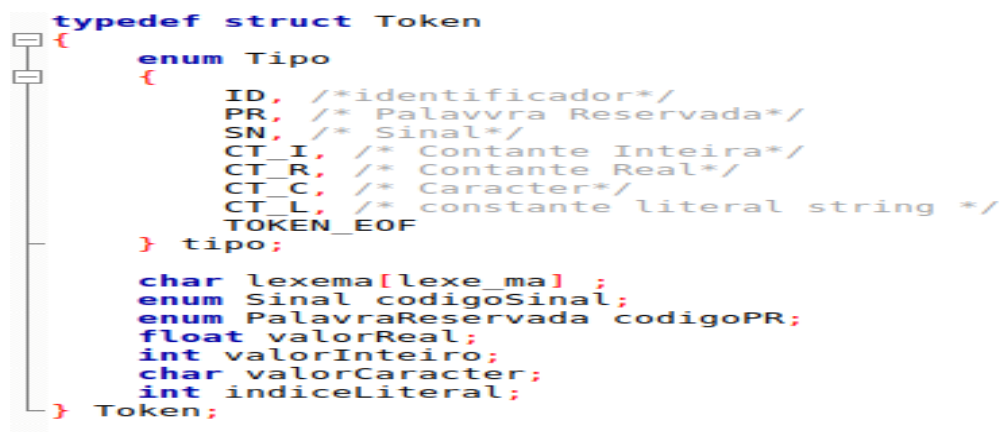


```
enum PalavraReservada
{
    CARACTER,
    INTEIRO,
    REAL,
    BOOLEANO,
    SEMPARAM,
    SEMRETORNO,
    SE,
    ENQUANTO,
    PARA,
    RETORNE,
    PROTOTIPO,
    SENAO,
    NENHUM,
};

enum Sinal
{
    PONTO_E_VIRGULA,
    VIRGULA,
    ABRE_CHAVE,
    FECHA_CHAVE,
    ABRE_PARENTESE,
    FECHA_PARENTESE,
    MAIS,
    MENOS,
    OU,
    E,
    IGUAL,
    EXCLAMACAO,
    ATRIBUICAO,
    DIFERENTE,
    MENOR_IGUAL,
    MAIOR_IGUAL,
    MAIOR,
    ASTERISCO,
    BARRA,
};
```

Figura 2: Enum das palavras reservadas

Figura 3: Enum dos sinais



```
typedef struct Token
{
    enum Tipo
    {
        ID, /*identificador*/
        PR, /* Palavra Reservada*/
        SN, /* Sinal*/
        CT_I, /* Contante Inteira*/
        CT_R, /* Contante Real*/
        CT_C, /* Caracter*/
        CT_L, /* constante literal string */
        TOKEN_EOF
    } tipo;

    char lexema[lexema];
    enum Sinal codigoSinal;
    enum PalavraReservada codigoPR;
    float valorReal;
    int valorInteiro;
    char valorCaracter;
    int indiceLiteral;
} Token;
```

Figura 4: Estrutura do Token criado

➤ **analex.c**

O arquivo inicia preenchendo a tabela de palavras reservadas com base no enum de palavras reservadas já mencionado. Nesse arquivo é feita a abertura do arquivo de entrada (teste.txt) possibilitando a leitura dos caracteres. Algumas ações foram modularizadas em funções afim de tornar a leitura do código mais legível, dentre essas funções estão a: `letraDigitoUnderline` que retorna verdadeiro (1) se for lido um numero inteiro positivo entre 0-9 ou um caracter entre a-z ou A-Z ou ainda underline (_).

A seguir no analisador léxico foram implementadas as funções referentes a cada estado principal presentes no AFD, houve uma preocupação em seguir estritamente o que havia sido construído no mesmo. As numerações das funções coincidem com as do AFD, partindo do estado inicial (estado 0). Sempre é lido um token extra das funções referentes aos estados, a sequência de caracter válidas e portanto o token achado e sempre devolvido para o estado0() que por sua vez é devolvido para o lexico(). Vale ressaltar que se um caracter lido não corresponde a sequência

esperada para ser identificado com um token válido da linguagem, o programa é abortado e é dado erro léxico. O principal problema encontrado aqui foi validar o uso do contador de linha, que na primeira versão não funcionava direito, após alguns testes houve a correção desse problema.

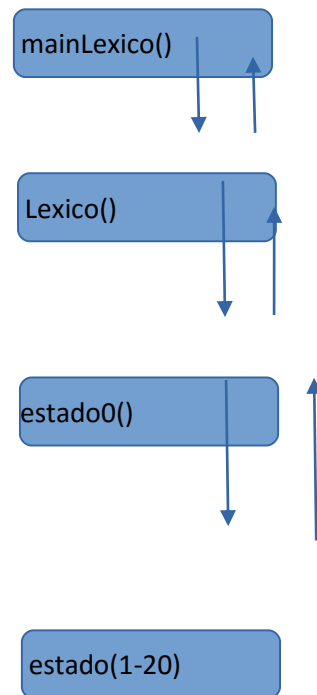


Figura 5: Fluxo de execução

3) Análise Sintática

O sintático é responsável por verificar a coerência sintática do arquivo fonte, a sintaxe das construções utilizadas foram determinadas mediante a gramática da linguagem cmm fornecidas na especificação da mesma. As funções criadas no anasin seguem as regras de produção também disponibilizadas nas especificações. Vale ressaltar que o sintático e o léxico trabalham em conjunto.



anasin.h

Neste arquivo foram implementadas as regras de produção da gramática. Para cada regra de produção foi criada uma função. Essa etapa custou um bom tempo de trabalho, a versão testada e apresentada para fins avaliativos continha alguns erros de construção, o principal problema que tive nessa etapa foi a leitura de tokens extras, isso acabava dando erros sintáticos aonde não deveria aparecer. A complexidade de algumas regras de produção também contribuíram para atrasar a finalização dessa etapa.

```
int prog();
int decl();
int decl_var();
int tipo();
int tipos_param();
int cmd();
int atrib();
int expr();
int expr_simp();
int termo();
int fator();
int op_rel();
void prog_sem_retorno();
void prog_prototipo();
void prog_tipo (TipoSimbolo tipoFuncao);

int tiposParam();
int cabecalho();

int VerificarSinal(enum Sinal);
int VerificarPR(enum PalavraReservada);

Token token;
FILE *arquivo;
```

Figura 6: Todas as regras de produção presentes no sintático



anasin.c

O arquivo começa chamado o léxico para leitura do primeiro token dentro de prog (função principal) e a depender do token lido é chamada uma função que atenda cada regra de produção apresentada. Caso o próximo token lido não atenda as regras de produção um erro semântico é indicado e o programa é abortado.

4) Análise Semântica

A construção dessa etapa não ficou tão clara pra mim, perder muito tempo consertando o sintático, e não conseguir implementa-la.

5) Gerenciador de Tabela de Símbolos e Gerenciador de Erros

A tabela construída tem a seguinte estrutura:

```
typedef enum TipoSimbolo
{
    INT,
    FLOAT,
    CHAR,
    BOOL,
    VOID
} TipoSimbolo;

typedef struct linha_tabela_simbolos
{
    char nome[TAM];
    int escopo;
    TipoSimbolo tipo;
    int ehFuncao;
    int ehParametro;
    int ehPrototipo;
} linha_tabela_simbolos;
```

Foram implementadas as

seguintes funções principais:

- void adiciona_tabela(TipoSimbolo tipo, char lexema[TAM]);
- int verifica_tipo(char lexema[TAM]);
- int busca(char lexema[TAM]);
- void remove_locais();
- void imprime_tabela();

Sendo que nem todas foram usadas porque não conseguir terminar a tempo o analisador semântico. Quando uma função é encontrada no analisador sintático a mesma é adicionada na tabela com os seguintes parâmetros (escopo global, nome, indicação de função), antes no entanto é verificado se já existe uma função com mesmo lexema, caso exista é emitido uma mensagem de erro. O mesmo vale para o protótipo de função com diferença na sinalização do parâmetro protótipo da minha tabela. A adição dos parâmetros das funções seguem o mesmo esquema com a mudança do escopo global para local.

6) Gerador de Código da MP

Não deu tempo de implementar o máquina de pilha por motivos de tempo.

7) Conclusão

A construção do compilador cmm facilitou no entendimento dos conceitos por trás de todo processo de compilação. Embora não tenha conseguido terminar a tempo todo o compilador o que foi feito até então me ajudou a ver de forma prática o que vimos em sala de aula, tornando o ensino mais eficaz.