Universidade Estadual Paulista
Rio Claro

Relatório de Implementação de um Compilador Pascal Simplificado

Autores
Fábio Contatto 1219596
Ivo Dias Gregorio 1219634

Professor
Eraldo Pereira Marinho

Sumário

1. Introdução

Este trabalho tinha por objetivo construir um compilador para linguagem pascal simplificado descrita no livro Programação Sistemática em Pascal. O compilador deve ser capaz de fazer a análise léxica do código, análise sintática e realizar ações semânticas sobre esse código para geração de um código assembler, de acordo com o padrão AT&T e informar ao programador erros que ele cometeu no código.

Do ponto de vista acadêmico torna-se relevante uma vez que possibilita aos alunos a fixação de conceitos, metodologias e a experiência no desenvolvimento de um.

2. Metodologia

Utilizamos a metodologia top-down, tradução dirigida por sintaxe.

3. Implementação

Neste capítulo descrevemos a gramática utilizada, como o código foi produzido e as limitações para este compilador.

3.1. Gramática da Linguagem

A linguagem está descrita no livro através de cartas sintáticas que aqui serão descritas na forma gramatical. Para uma melhor visualização todos os símbolos terminais estão em negrito, cor vermelho.

G = {NT, T, P, S}, em que NT é o conjunto dos símbolos não-terminais, T é o conjunto dos símbolos terminais, P é o conjunto das produções e S é o símbolo não-terminal de início da gramática.

```
NT = {PROGRAMA, BLOCO, EXPRESSÃO, EXPRESSÃO_SIMPLES, TERMO, FATOR, VARIÁVEL,
TIPO, TIPO_SIMPLES, CONSTANTE, CONST_SEM_SINAL, TEXTO, NUM_SEM_SINAL,
INT_SEM_SINAL, IDENTIFICADOR, DIGITO, LETRA, CARACTER}
```

```
T = {A..Z, a..z, 0..9,
+, -, *, /, div, mod, &, or, not,
=, #, <, <=, >=, >, (, ), [, ], {, },
:=, ., .., ,, :, ;, ^, `,
program, label, const, var, begin, end,
if, then, else, case, of, while, do,
repeat, until, for, to, file, goto}
```

P:

```
PROGRAMA::= program BLOCO.

BLOCO::=
      [label IDENTIFICADOR [{, IDENTIFICADOR}];]
      [const IDENTIFICADOR = CONSTANTE;[{IDENTIFICADOR = CONSTANTE;}]]
      [type IDENTIFICADOR = TIPO;[{IDENTIFICADOR = TIPO;}]]
      [var IDENTIFICADOR[{, IDENTIFICADOR}]:TIPO;
            [{IDENTIFICADOR[{, IDENTIFICADOR}]:TIPO;}]]
     begin COMANDO [{; COMANDO}] end
```

```
COMANDO::=
      [IDENTIFICADOR_LABEL:](
      (VARIÁVEL | IDENTIFICADOR_FUNÇÃO) := EXPRESSÃO |
      IDENTIFICADOR_ROTINA[(EXPRESSÃO[{, EXPRESSÃO}] )] |
      begin COMANDO[{; COMANDO}] end |
      if EXPRESSÃO then COMANDO [else COMANDO] |
      case EXPRESSÃO of CONSTANTE [{, CONSTANTE}]: COMANDO
            [{; CONSTANTE{, CONSTANTE}: COMANDO}] end |
      while EXPRESSÃO do COMANDO |
      repeat COMANDO [{; COMANDO}] until EXPRESSÃO |
      for IDENTIFICADOR_VARIÁVEL := EXPRESSÃO to EXPRESSÃO do COMANDO |
      goto IDENTIFICADOR_LABEL |
      vazio
      )

EXPRESSÃO::=
      EXPRESSÃO_SIMPLES [(= | # | < | <= | >= | >)EXPRESSÃO_SIMPLES]

EXPRESSÃO_SIMPLES::=
      [(+|-)]TERMO [{(+|-|or)TERMO}]

TERMO::=
      FATOR [{(*|/|div|mod|&)FATOR}]

FATOR::=
      CONST_SEM_SINAL |VARIÁVEL |
      (EXPRESSÃO) |not FATOR

TIPO::=
      TIPO_SIMPLES | file of TIPO

TIPO_SIMPLES::=
      IDENTIFICADOR_TIPO |
      (IDENTIFICADOR{, IDENTIFICADOR}) |
      CONSTANTE..CONSTANTE

CONSTANTE::=
      ([(+|-)](IDENTIFICADOR_CONST | NUMBER) | TEXTO)

CONST_SEM_SINAL::=
      IDENTIFICADOR_CONST | NUMBER | TEXTO

NUMBER::= NUM_SEM_SINAL | INT_SEM_SINAL
NUM_SEM_SINAL::= INT_SEM_SINAL[.INT_SEM_SINAL][E[(+|-)]INT_SEM_SINAL]
INT_SEM_SINAL::= DIGITO[{DIGITO}]
TEXTO::= 'CARACTER [{CARACTER}]'
IDENTIFICADOR::= LETRA[{(LETRA|DIGITO)}]
DIGITO  ::= 0|1|2|3|4|5|6|7|8|9
LETRA   ::= A..Z | a..z
CARACTER::= <qualquer caracter >
```

3.2. Geração de Código

Utilizamos ambiente Linux para o desenvolvimento de nosso software. Como ferramenta de edição utilizamos o vi e para compilar nosso código (que foi todo escrito em linguagem C) utilizamos o compilador gcc da gnu.

Os símbolos não-terminais IDENIFICADOR, NUMBER e todos os símbolos terminais tornaram-se *tokens* que o analisador léxico está gerando.

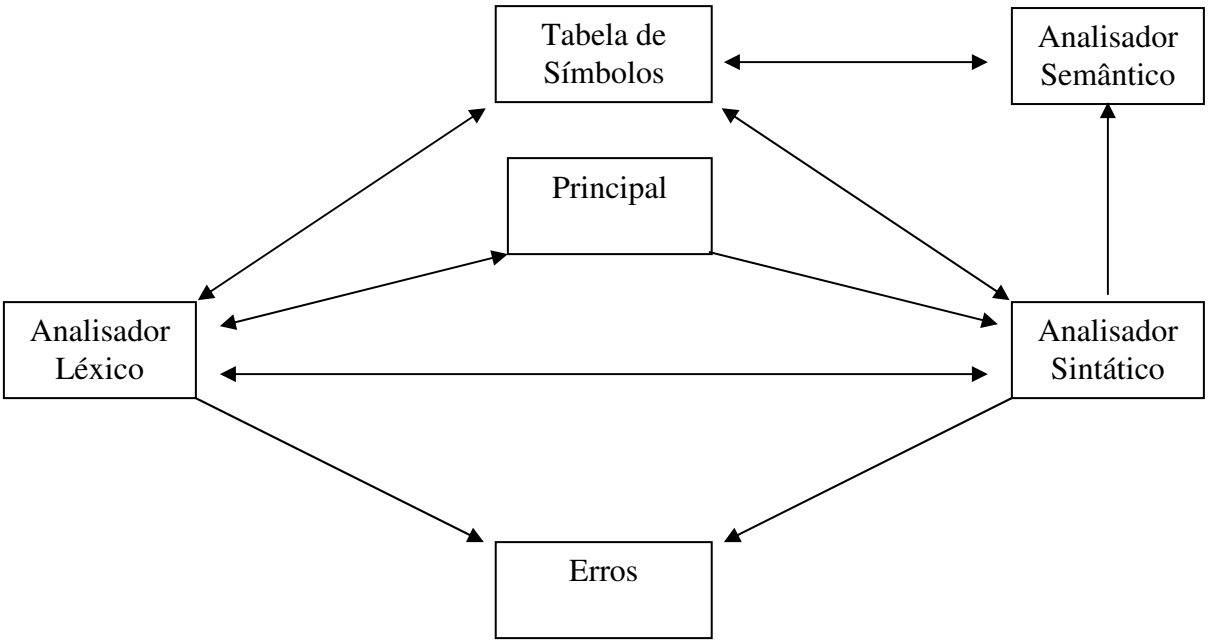
Os símbolos não terminais transformaram-se em rotinas de forma a acompanhar esta gramática, em que um símbolo não-terminal chama o outro e assim por diante até atingir símbolos terminais (Metodologia de tradução dirigida por sintáxe).

3.3. Limitações

- Este compilador não trata vetores, arquivos, ponteiros, procedimentos, funções (e por conseqüência passagem de parâmetros), vetores e a construção de tipos de dados.
- Este compilador trata diversos tipos de laços à saber: *for, while e repeat*;
- Também tratamos os cálculos matemáticos básicos: soma, subtração, divisão e multiplicação; bem como a atribuição de valores a variáveis.
- Operações comparativas, tais como <, <=, >=, >, =, # também estão implementadas.
- A operação IN **não** está definida para este compilador.
- Operações lógicas AND, OR, NOT.
- Implementamos estruturas condicionais: *if-then-else*.
- O desvio incondicional foi atendido com a instrução *goto*.
- Os rótulos para a função *goto* foram alterados de numéricos para identificadores (de rótulos).
- Este compilador não é sensível a caixa alta ou caixa baixa, ou seja, maiúsculo ou minúsculo. O lexema begin é igual ao lexema BEGIN que é igual ao lexema BegIn, por exemplo.
- As operações só estão sendo realizadas para valores inteiros.

4. Descrição dos Módulos

O compilador pascal (cp), é constituído dos módulos a saber: principal, analisador léxico, analisador sintático, analisador semântico, tabela de símbolos, erros.



4.1. Módulo Principal

O módulo principal é responsável por carregar o arquivo fonte e iniciar a variável global *lookahead* invocando a função *gettoken* do analisador léxico. Feito isso o módulo principal inicia a análise préditiva do código fonte invocando a função *programa* do analisador sintático de acordo com a gramática no capítulo 2.

4.2. Módulo do Analisador Léxico

O analisador léxico é o módulo responsável por identificar lexemas válidos pela linguagem pascal simplificado, salvá-los na tabela de símbolos e retornar um *token* que identifique o lexema lido ao analisador sintático.

Este módulo possui apenas três funções:

- `void toUpperCase(char *s);`
  - Esta função apenas converte cada caractere da *string* apontada por s de minúsculo para maiúsculo;
  - Esta sendo utilizada para eliminar a sensibilidade de caixa, pois em pascal não há distinção entre letras minúsculos e maiúsculos, e dessa forma todo o código fonte é tratado apenas como maiúsculo.

- `int gettoken(void);`
  - Além de eliminar caracteres de controle esta função é a responsável por identificar lexemas válidos da linguagem pascal simplificado, retornando um *token* para cada tipo de lexema;
  - Utiliza a função "`void error(erro_t err);`" do módulo de erros para emitir alertas sobre possíveis más formações de lexemas ou o não reconhecimento de um lexema específico pela linguagem pascal simplificado.
  - Também cabe a esta função ir atualizando o contador de linha *line* e o contador da coluna *col*. Estas variáveis globais do módulo de erros indicam a posição corrente de leitura do código fonte.
- `void match(int token, erro_t err);`
  - Esta função é responsável por validar se o *token* corrente na variável global *lookahead* é de fato válido na sintaxe do programa.
  - Caso seja válido, esta função irá se encarregar de apanhar um próximo *token* invocando a função "`int gettoken(void);`";
  - Caso não seja válido a função "`void error(erro_t err);`" será invocada para exibição da mensagem de erro.

O módulo é constituído pelos arquivos "lexer.c" e "lexer.h". Para maiores detalhes consulte a listagem dos códigos clicando nos nomes dos arquivos.

4.3. Módulo do Analisador Sintático

Este módulo é responsável pela análise sintática preditiva do código fonte e por acionar as ações semânticas a cada tipo de produção.

Este módulo possui essas funções que são equivalentes aos símbolos não-terminais descritos na gramática do capítulo 2, exceto pelas duas últimas funções que possuem tarefas especiais.

- `void programa(void);`
- `void bloco(void);`
- `void comando(void);`
- `void lista_parametros(void);`
- `void expressao(void);`
- `void expressao_simples(void);`
- `void termo(void);`
- `void fator(void);`
- `int variavel(void);`
- `void tipo(void);`
- `void tipo_simples(void);`
- `void constante(void);`
- `void qualificador(int tok);`
  - Esta função simplesmente verifica se o lexema corrente possui o token de idenfificador e o qualifica em:
    - `UNDEFINED`
      (identificador ainda não definido).
    - `IDENTIFICADOR_CON`

(um identificador de constante).
        - IDENTIFICADOR_TIP
          (um identificador de tipo definido - usuário ou sistema).
        - IDENTIFICADOR_VAR
          (um identificador de variável).
        - IDENTIFICADOR_PRO
          (um identificador de procedimento).
        - IDENTIFICADOR_FUN
          (um identificador de função).
- `boolean_t match_qualifier(int qualifier, int showerro);`
    - Esta função verifica se o lexema encontrado já está na tabela de símbolos, se é um identificador e qual a qualificação do identificador (UNDEFINED, ID_CON, ID_TIP, ID_VAR, ID_PRO, ID_FUN).
    - Caso esteja tudo ok, returna TRUE, caso contrário um erro é emitido e retornar FALSE.

Este módulo é constituído pelos arquivos "parser.c" e "parser.h".

4.4. Módulo do Analisador Semântico

O Analisador Semântico dita as ações que trechos específicos do código pascal têm, ou seja, seus significados, ações.

Este módulo possui as seguintes funções:

- `void semantic(action_t action);`
    - Esta função é um direcionador de ações. O analisador sintático pode ativar diversas funções presentes neste módulo pela chamada a esta função com o respectivo código `action` da ação.
- `void encerra(int exit_code);`
    - Esta função emite para o arquivo resultado o trecho de código em assembler para finalizar o software, através da interrupção do sistema.
- `void leitura(int descriptor, char *address, int tamanho);`
    - Esta função chama a interrupção para fazer a leitura dos caracteres do descritor.
- `void escrita(int descriptor, char *address, int tamanho);`
    - Esta função chama a interrupção para fazer a impressão dos caracteres contidos no na posição de memória apontada por `address`.
- `void abrir_arq(void);`
    - Esta função abre um arquivo.
- `void fechar_arq(void);`
    - Esta função fecha um arquivo.
- `void section_text(void);`
    - Esta função emite para o arquivo resultado o trecho de código em assembler para abrir a seção de texto.
- `void section_rodata(void);`
    - Esta função define uma seção de dados inicializados; os dados são de somente leitura.
- `void print_const_lex(void);`

    - Esta função imprime o lexema da seção *rodata*.
- `void print_const(void);`
    - Esta função imprime uma constante, seja booleana, inteira, ponto-flutuante, caractere ou texto.
- `void section_bss(void);`
    - Esta função define uma seção de dados não-inicializados.
- `void print_var(void);`
    - Esta função imprime uma variável, seja ela booleana, inteira, ponto-flutuante, caractere ou texto.
- `void jmp_start(void);`
    - Esta função imprime o rótulo de início do programa.
- `void act_then(int counter_label_if);`
    - Esta função emite o trecho de código que faz a verificação para ver se o valor retornado pela expressão é TRUE e o trecho de código que emite um *jump* para pular o trecho de código caso a expressão seja verdadeira.
- `void act_else(int counter_label_if, int END_L);`
    - Esta função emite o trecho de código que contém um *jump* de seu respectivo *then* para o fim do bloco *if-then-else* e emite o rótulo da instrução jump condicional a expressão *then* do *if-then-else* bloco em questão.
- `void put_label_if(int counter_label_if);`
    - Esta função emite o rótulo que pula o bloco *if-then-else*.
- `void push(const char *address);`
    - Esta função determina o empilhamento enviando o trecho de código ao arquivo resultado.
- `void pop(const char *address);`
    - Esta função determina o desempilhamento enviando o trecho de código ao arquivo resultado.
- `void multiplica(void);`
    - Esta função realiza a multiplicação de dois números com sinal.
- `void divide_float(void);`
    - Esta função realiza a divisão de dois números com sinal.
- `void divide_int(void);`
    - Esta função realiza a divisão de dois números inteiros com sinal.
- `void divide_resto(void);`
    - Esta função realiza a divisão de dois números e retorna o resto.
- `void logic_and(void);`
    - Esta função realiza a operação lógica AND.
- `void plus(void);`
    - Esta função realiza a soma de dois valores.
- `void minus(void);`
    - Esta função realiza a subtração de dois valores.
- `void logic_or(void);`
    - Esta função realiza a operação lógica OR de dois valores.
- `void igual(void);`
    - Esta função esta função verifica a igualdade de dois valores.
- `void diferente(void);`

    - Esta função esta função verifica se dois valores são diferentes.
- `void menor_que(void);`
    - Esta função verifica se um valor a é menor que um valor b. Retorna verdade caso verdadeiro.
- `void menor_igual_que(void);`
    - Esta função verifica se um valor a é menor ou igual que um valor b. Retorna verdade caso verdadeiro.
- `void maior_igual_que(void);`
    - Esta função verifica se um valor a é maior ou igual que um valor b. Retorna verdade caso verdadeiro.
- `void maior_que(void);`
    - Esta função verifica se um valor a é maior que um valor b. Retorna verdade caso verdadeiro.
- `void salve_pro(void);`
    - Esta função salva o nome do procedimento.
- `void call_pro(void);`
    - Esta função emite a instrução para invocar o procedimento que foi salvo em `void salve_pro(void);`
- `void put_buffer_out(void);`
    - Esta função emite o trecho de código para converter qualquer tipo de dado em string e armazenar esta string no buffer de saída *buffer_out*.
- `void put_buffer_in(void);`
    - Esta função emite o trecho de código para converter a string lida por um descritor e armazenado em *buffer_in* em qualquer tipo de dado.
- `void negar(void);`
    - Esta função realiza a operação lógica NOT.
- `void oposto(void);`
    - Esta função realiza a multiplica por –1 um valor.
- `void expression_case(void);`
    - Esta função resolve uma expressão.
- `void print_conditional_jmp_label_case(int l);`
    - Esta função emite o trecho de código para pular para a próxima possibilidade do comando *case* caso a atual não seja válida.
- `void jmp_end_label_case(int l);`
    - Esta função emite o trecho de código para um pulo forçado para o final do *case* quando alguma das possibilidades foi atendida.
- `void print_next_label_case(int l);`
    - Esta função emite o rótulo para que aconteça o *jump* do item anterior em caso de falha.
- `void put_end_label_case(int l);`
    - Esta função emite o rótulo de fim do bloco *case* corrente.
- `void put_label_while(int l);`
    - Esta função emite o rótulo de início do bloco *while*.
- `void jmp_when_false(int l);`
    - Esta função emite a instrução de desvio-incondicional para sair do bloco *while* corrente, quando a condição for falsa.

- `void jmp_to_begin(int l);`
  - Esta função emite o trecho de código para o desvio-incondicional para o início do bloco *while* após a execução de todos os seus comandos.
- `void put_end_label_while(int l);`
  - Esta função emite o rótuilo para o desvio condicional do bloco *while* corrente para o fim caso a condição não seja satisfeita, ou seja, retorne falso.
- `void print_label_repeat(int l);`
  - Esta função emite o rótulo para onde deve ocorrer o desvio condicional do bloco *repeat* caso a condição de parada **não** seja satisfeita.
- `void jmp_if_false(int l);`
  - Esta função emite a instrução de desvio condicional. O desvio ocorrerá caso a condição **não** seja atendida.
- `void salve_variable(void);`
  - Esta função salva o nome da variável;
- `void atribuir(void);`
  - Esta função emite o trecho de código em que ocorre a atribuição de algo para alguém.
- `void put_label_for(int l);`
  - Esta função emite o rótulo de retorno de um bloco *for*.
- `void verifica_contador(const char *s, int l);`
  - Esta função verifica se a variável do contador já superou ou não o valor a ser atingido e emite também a instrução para um desvio incondicional de saída do bloco *for* corrente em caso do contador ter ultrapassado o valor a ser atingido.
- `void jmp_label_for(const char *s, int l);`
  - Esta função incrementa o contador de um bloco *for,* emite a instrução de um desvio incondicional para o começo do laço e o rótulo para a saída do bloco *for*.
- `void inconditional_jmp(void);`
  - Esta função emite a instrução de um desvio incondicional . Representa o *goto* do pascal.
- `void print_label(void);`
  - Esta função emite o rótulo do desvio incondicional emitido pela função `void inconditional_jmp(void);`.
- `void print_literal_statements(void);`
  - Esta função, após ter lido todo o código fonte e reconhecido todas as constantes literais dop tipo string, cria um arquivo final e reabre o arquivo intermediário de saída sendo trabalhado até agora, para inserir no bloco *rodata* endereços fictícios (criados pelo compilador) e os valores (as próprias strings) para serem referenciadas.
- `void mark(void)`
  - Esta função marca o índice de início de uma declaração de variáveis, que vão sendo inseridas na tabela de símbolos sem um tipo, para serem “tipadas” quando um tipo for reconhecido.

Este módulo é constituído dos arquivos “semantic.c” e “semantic.h”.

4.5. Módulo da Tabela de Símbolos

A tabela de símbolos é o módulo responsável por salvar os lexemas já identificados pelo módulo do analisador léxico (função `int gettoken(void);`), evitando assim duplicidade de informações, seus com os tokens, tipos dos identificadores, qualificando-os, registrando o endereço relativo da memória e o nível.

Este módulo possui as seguintes funções:

- `int insert(const char *s, int token, const char *offset, qualifier_t qual, type_t type);`
  - Esta função é responsável por inserir um novo registro na tabela de símbolos. Caso não haja mais espaço para armazenamento de dados esta função aloca mais espaço e então salva os dados.
  - Esta função retorna o *token* do lexema inserido na tabela.
  - O argumento `const char *s` recebe o lexema reconhecido;
  - O argumento `int token` salva o token deste lexema.
  - O argumento `const char *offset` salva a forma como lexema deve ser referenciado no código assembler.
    - "strings" são referenciadas por variáveis criadas pelo compilador, *.TXTn* (n >=0), e inseridas na seção *rodata*.
    - Constantes numéricas recebem o sinal *$* antes do lexema.
    - Identificadores são referenciados com o próprio lexema.
- `int retrieve(const char *s);`
  - Esta função vai encontrar o lexema na tabela de símbolos e retorna o token do lexema.
  - Caso não encontre retorna 0 (zero).
- `void trimtab(void);`
  - Esta função não é essencial, mas evita desperdício de espaço na memória, pois ela ajusta a capacidade de armazenamento da tabela à quantidade de registros da tabela, liberando o espaço extra.
- `int findindex(const char *s);`
  - Esta função simplesmente retorna o índice da tabela de símbolos em que um determinado lexema está salvo.
  - Caso não aja nenhum lexema, será retornado o valor –1.
- `tupla_t *gettupla(int idx);`
  - Esta função retorna um ponteiro para tupla endereçada pelo índice idx.
  - Caso seja um endereço inválido será retornado o valor NULL.
- `boolean_t settupla(int idx, tupla_t tp);`
  - Esta função armazena uma tupla tp num lugar específico da tabela de símbolos, idx.
  - Retorna TRUE em caso de sucesso ou FALSE, caso contrário.
- `void printtupla(int idx);`
  - Esta função imprime o conteúdo da tupla endereçada por idx.
  - Caso idx tenha um valor inválido nada será impresso.
- `void printtab(void);`
  - Imprime na saída padrão todo o conteúdo da tabela de símbolos.
  - Caso não tenha conteúdo nada será impresso.
- `void inittab(void);`

- Esta função é responsável por iniciar o conteúdo da tabela de símbolos com palavras-chaves, tipos, variáveis, procedimentos e funções pré-definidos pela linguagem pascal simplificado.

Este módulo é constituído pelos arquivos “tabsym.c” e “tabsym.h”.

4.6. Módulo de Erros

Este módulo é responsável apenas por imprimir os diversos tipos de erros que podem ocorrer na análise léxica, sintática ou semântica.

Este módulo possui apenas uma função à saber: `void error(erro_t err);`. Esta função é responsável por pela emissão de mensagens de erros encontrados pelos analisadores ao usuário.

Abaixo temos a lista que associa cada erro a sua respectiva mensagem:

```
enum erro{
  NONE,
  UNKNOWN,
  EXP_APOST_SIMPLES,
  EXP_APOST_DUPLO,
  EXP_INT,
  EXP_DOU,
  EXP_OPERATOR,
  EXP_ASSIGN,
  EXP_ASG_OPER,
  EXP_TERM,
  EXP_PTO,
  EXP_PTOPTO,
  EXP_VIRG,
  EXP_PTOVIRG,
  EXP_ABRPAR,
  EXP_FECPAR,
  EXP_ABRCOL,
  EXP_FECCOL,
  EXP_IGUAL,
  EXP_ID,
  EXP_PROGRAM,
  EXP_BEGIN,
  EXP_END,
  EXP_IF,
  EXP_THEN,
  EXP_ELSE,
  EXP_CASE,
  EXP_OF,
  EXP_CONST,
  EXP_TYPE,
  EXP_VAR,
  EXP_WHILE,
  EXP_DO,
  EXP_REPEAT,
  EXP_UNTIL,
  EXP_FOR,
  EXP_TO,
```

```
char *error_msg[COUNT]={
  "No error",
  "Unknown symbol",
  "Expected a '''",
  "Expected a '\"'",
  "Expected an integer value",
  "Expected an double value",
  "Expected an operator",
  "Expected a ':='",
  "Expected ':=' or an operator",
  "Expected a term",
  "Expected a '.'",
  "Expected a ':'"
  "Expected a ','",
  "Expected a ';'",
  "Expected a '('",
  "Expected a ')'",
  "Expected a '['",
  "Expected a ']'",
  "Expected a '='",
  "Expected an identifier",
  "Expected a 'PROGRAM'",
  "Expected a 'BEGIN'",
  "Expected a 'END'",
  "Expected a 'IF'",
  "Expected a 'THEN'",
  "Expected a 'ELSE'",
  "Expected a 'CASE'",
  "Expected a 'OF'",
  "Expected a 'CONST'",
  "Expected a 'TYPE'",
  "Expected a 'VAR'",
  "Expected a 'WHILE'",
  "Expected a 'DO'",
  "Expected a 'REPEAT'",
  "Expected a 'UNTIL'",
  "Expected a 'FOR'",
  "Expected a 'TO'",
```

```
  EXP_GOTO,                                  "Expected a 'GOTO'",
  DIV_BY_ZERO,                               "Divizion by zero",
  TYPE_INCOMPATIBLE,                         "Tipo do identificador imcompatível",
  SINTAX_ERR,                                "Erro de sintáxe"
  COUNT      /* quantidade de erros */   };
};
typedef enum erro erro_t;
```

Este módulo é constituído pelos arquivos “error.c” e “error.h”.

5. Descrição Geral do Funcionamento

O usuário deve escrever seu código pascal em arquivo tipo texto puro, obedecendo a sintaxe do pascal simplificado, salvar este arquivo e por fim invocar o compilador pascal cp, no console com o comando *./cp <nome_do_arquivo_fonte>.pas* (em Linux).

O módulo principal carregará o arquivo fonte, e invocará a função *gettoken* do módulo analisador léxico para iniciar a variável global *lookahead* e logo em seguida invocará a função *programa* do analisador sintático que inicia uma análise sintática preditiva do código fonte de acordo com a gramática descrita acima (Capítulo 3, seção 3.1). O analisador léxico também é responsável por ir adicionando os símbolos lidos na tabela de símbolos caso ainda não estejam presentes. O analisador sintático validará a gramática ao invocar a função *match* do analisador léxico que irá verificar se o *token* corrente é o esperado pela gramática. Também é o analisador sintático que ativará as funções do módulo do analisador semântico para o trato de cada estrutura de informação lida. O analisador semântico por sua vez possui a função de gerar o arquivo com o código em linguagem de montagem para que o montador possa fazer o resto do trabalho, portanto utilizamos uma linguagem intermediária.

Por fim, cabe ao montador ‘**as’,** do próprio ambiente Linux, converter este arquivo em objeto e fazer a linkagem do mesmo.

6. Problemas

Neste capítulo comentaremos sobre os problemas na construção deste compilador e as restrições que impomos a este compilador.

- No analisador sintático tivemos problemas em algumas complexidades da gramática ou por não entendermos corretamente para que serviam alguns lexemas na linguagem pascal, ou na verificação de tipos, etc. Definir os tokens, símbolos não-terminais baseados apenas na carta sintática não foi trivial em alguns pontos por não sabermos do que se tratavam alguns blocos na linguagem pascal.
- Ainda no analisador sintático tivemos dificuldades escolher a posição e a ação semântica correta.
- Na seção de erros, também não soubemos como tratar todos os possíveis erros e nem sabemos se conseguimos de fato identificar todos os possíveis erros.

- No analisador semântico o excesso de funções nos confundiu diversas vezes em quais já haviam sido escritos ou não, pois antes de construir este módulo, varremos o analisador sintático e colocamos lá todas as ações que achamos cabíveis.
- Na definição da tabelas de símbolos. A cada etapa pronta do compilador, era necessário remodelar a tabela para que uma próxima etapa também pudesse ser atendida.
- Tivemos problemas na depuração por desconhecimento de ferramentas.
- A falta de material para consulta sobre a construção de procedimentos, funções, vetores, ponteiros e tipos de dados, sendo assim, estes itens foram removidos deste compilador.
- Sem sombra de dúvidas o maior de todos os problemas foi o pouco tempo para o desenvolvimento deste compilador.
- A maior parte dos problemas foram resolvidos empiricamente.

Referências

- Wirth, N. - Programação Sistemática em Pascal, 6ª Edição (Editora Campus) – Data 1985;
- Aho, A.V. – Compiladores – Princípios, Técnicas e Ferramentas (Editora LTC)
- Marinho, E.P. - **Elementos de GNU Assembler -** Curso Básico de GNU Assembler para Intel 386, http://sagitario.rc.unesp.br/as, Data: 25/01/2005