Projeto e Implementação de uma Ferramenta de Compilação para a Linguagem TPP

Eduardo Kurek[1]
[1]Departamento Acadêmico de Computação (DACOM)
Universidade Tecnológica Federal do Paraná (UTFPR)

Abstract

Compilers are computer programs that translate from a language to another. Developing a compiler is a very complex task, most computer scientists and professionals will never write a full compiler. This work aims to construct a compiler for the **TPP** language. The development of a compiler includes 4 stages, which are: lexical analysis, syntactic analysis, semantic analysis and code generation. In lexical analysis, we must determine the marks that define the language, breaking the source code into identifiable and meaningless pieces for now. When the lexical analyzer finishes, we can receive a language source code and extract their tokens.

Resumo

Compiladores são programas de computador que traduzem de uma linguagem para outra. Desenvolver um compilador é uma tarefa muito complexa, a maioria dos cientistas e profissionais da computação jamais escreverão um compilador por completo. Esse trabalho tem por finalidade a construção de um compilador para a linguagem **TPP**. O processo de desenvolvimento de um compilador envolve 4 etapas, sendo elas: análise léxica, análise sintática, análise semântica e geração de código. Na análise léxica, devemos determinar as marcas que formam a linguagem, quebrando o código fonte em pedaços identificáveis e sem significado por enquanto. Ao finalizar o analisador léxico, podemos receber um arquivo da linguagem fonte e extrair as marcações daquele código.

1 Introdução

Compiladores são programas de computadores capazes de traduzir programas em uma linguagem para outra. Eles recebem um programa escrito na linguagem fonte e produz um equivalente na linguagem alvo (LOUDEN 2004).

Tais programas são muito importantes, pois permitem a escrita de programas em uma linguagem de alto nível, desse modo, não precisamos ter ciência completa do *hardware* em que estamos desenvolvendo. O compilador abstrai as informações e nuâncias de cada máquina, facilitando a portabilidade e simplificando muito o desenvolvimento de novos programas. Também são fundamentais pois são responsáveis por otimizar o código fonte, através de técnicas, é possível tornar o código resultante mais eficiente que o original.

As linguagens de alto nível tentam aproximar a forma de escrever programas com a forma que nós, humanos, nos comunicamos. Escondendo detalhes de baixo nível como pertinência dos registradores entre procedimentos, cálculo de endereços para desvios, gerenciamento de memória, etc.

Corresponding author(s): Eduardo Kurek, Universidade Tecnológica Federal do Paraná (UTFPR, eduardokurek@alunos.utfpr.edu.br

2 Análise Léxica

A análise léxica ou varredura é a fase em que o compilador lê o arquivo de entrada e o separa em *tokens*, que são marcações que representam as palavras daquela linguagem. Exemplos de tokens são as palavras reservadas, como **if** e **else**, identificadores, que são rótulos atribuidos pelo usuário, símbolos algébricos, entre outros.

Para realizar a varredura, precisamos de um método eficiente para identificar os padrões dessa linguagem, que são: **expressões regulares** ou **autômatos finitos** (Menezes 2011). O método utilizado para esse desenvolvimento foi a utilização de expressões regulares (JARGAS 2012), que é uma maneira formal de descrever um padrão de caracteres.

2.1 Especificações da linguagem TPP

Tpp é uma linguagem simples, sem suporte a orientação a objetos, *strings* e ponteiros. Seus tipos primitivos são **inteiro** e **flutuante**, e também não é possível definir estruturas. A linguagem suporta a criação de funções com múltiplos parâmetros, e vetores de no máximo duas dimensões. Como a linguagem não suporta ponteiros, é necessário trabalhar com vetores globais, em caso de utilização em várias funções. Os procedimentos **principal**, **escreva** e **leia** são reservados da linguagem. Comentários em tpp ficam dentro de chaves ({ Comentário }). O Códido 1 é um exemplo escrito na linguagem tpp, já a tabela 1 mostra todos os padrões de expressões regulares utilizados e seus respectivos tokens.

Código 1: Exemplo de código em tpp

```
1  inteiro: n
2
3  inteiro fatorial(inteiro: n)
4      inteiro: fat
5      se n > 0 então {não calcula se n > 0}
6          fat := 1
7          repita
8              fat := fat * n
9              n := n - 1
10         até n = 0
11         retorna(fat) {retorna o valor do fatorial de n}
12     senão
13         retorna(0)
14     fim
15 fim
16
17 inteiro principal()
18     leia(n)
19     escreva(fatorial(n))
20     retorna(0)
21 fim
```

Expressão regular	Token
\+	MAIS
-	MENOS
*	VEZES
\/	DIVIDE
\:	DOIS_PONTOS
,	VIRGULA
\<	MENOR

>	MAIOR
=	IGUAL
\<>	DIFERENTE
\<=	MENOR_IGUAL
>=	MAIOR_IGUAL
&&	E
||	OU
!	NAO
\(	ABRE_PARENTESE
\)	FECHA_PARENTESE
\[	ABRE_COLCHETE
\]	FECHA_COLCHETE
se	SE
então	ENTAO
senão	SENAO
fim	FIM
repita	REPITA
até	ATE
:=	ATRIBUICAO
leia	LEIA
escreva	ESCREVA
retorna	RETORNA
inteiro	INTEIRO
flutuante	FLUTUANTE
[0-9]+	NUM_INTEIRO
[-]?[0-9]+(\.([0-9]+)?)	NUM_PONTO_FLUTUANTE
(([-\+]?)([1-9])\.([0-9])+[eE]([-\+]?)([0-9]+))	NUM_NOTACAO_CIENTIFICA
[a-zA-ZáÁãÃàÀéÉíÍóÓõÕ][a-zA-ZáÁãÃàÀéÉíÍóÓõÕ0-9_]*	ID

Tabela 1: Expressões regulares e tokens da linguagem tpp

2.2 Implementação

Como o processo de análise léxica é muito semelhante entre as várias linguagens de programação, ferramentas de desenvolvimento para essa etapa foram criadas, que é o caso do **Flex**, feito para a linguagem C e posteriormente para C++ (Paxson 1990). Essa ferramenta permite o rápido desenvolvimento de um analisador léxico através da definição de regras. O Código 2 mostra como podemos definir essas regras.

Código 2: Código das regras do analisador léxico

```
1  %%
2
3  \{([^{}]|\n)*\} { /* IGNORA COMENTÁRIOS */ }
4  \+ yyprint("MAIS");
5  - yyprint("MENOS");
6  \* yyprint("VEZES");
7  \/ yyprint("DIVIDE");
8
9  regras omitidas...
10
```

```
11 [ \t] /* IGNORA ESPAÇOS EM BRANCO */ column++;
12 \r\n|\n  /* IGNORA QUEBRAS DE LINHA */ column = 1;
13 .|ç yyerror(Error::ERR_LEX_INV_CHAR); column++;
14
15 %%
```

No código 2, as funções `yyprint(const char*)` e `yyerror(const Error*)` imprimem a mensagem correspondente na saída padrão. A segunda função utiliza dos métodos da classe `Error` descritos no código 3 para tal ação.

As linhas 3, 11 e 12 são necessárias para o Flex ignorar comentários e caracteres de quebra de linha. Na linha 12, foi necessário incluir a expressão \r\n pois em arquivos no formato CRLF, a quebra de linha é precedida pelo caracter \r. Na linha 13 pode-se notar a presença do **ç**, isso se faz necessário pois o Flex lê caracteres byte a byte (em codificação UTF-8), e como o caracter ç é uma junção de dois bytes (C3 A7), precisa ser identificado como uma expressão à parte. Caso contrário, C3 seria identificado como o caracter Ã, e a saída não seria como o esperado.

Código 3: Error.h

```
1  #include <string>
2
3  namespace Error{
4
5      class Error{
6
7          std::string cod;
8          std::string message;
9
10
11     public:
12         Error(std::string cod, std::string message) :
13             cod(cod), message(message) {}
14
15         virtual void printCode() const;
16         virtual void printMessage() const;
17
18     };
19
20     extern const Error* ERR_LEX_USE;
21     extern const Error* ERR_LEX_NOT_TPP;
22     extern const Error* ERR_LEX_FILE_NOT_EXISTS;
23     extern const Error* ERR_LEX_INV_CHAR;
24 }
```

Pode-se utilizar da *flag* -k para que a saída do analisador léxico exiba apenas os tokens, sem mensagens adicionais. Tratamentos de erros também foram implementados. Sendo eles:

- **ERR-LEX-USE**: Erro fatal, a execução do varredor deve ser feita com a seguinte assinatura: `./lex code.tpp [-k];`
- **ERR-LEX-NOT-TPP**: Erro fatal, disparado quando o programa enviado por parâmetro não possui a extensão .tpp;
- **ERR-LEX-NOT-EXISTS**: Erro fatal, disparado quando o programa enviado por parâmetro não existe;
- **ERR-LEX-INV-CHAR**: Disparado quando o analisador encontra algum caracter inválido;

Ao compilar o varredor, podemos executar e ver a saída que ele gerou. As Figuras 1 e 2 exibem o arquivo gerado pelo lexer ao processar o codigo 1, com e sem a flag -k, respectivamente. Na Figura 2, o formato `LexToken` exibe respectivamente: O token gerado pela regra, qual palavra gerou aquele token, qual linha a palavra foi encontrada e o comprimento dessa palavra.

```
INTEIRO             FECHA_PARENTESE   ID                FECHA_PARENTESE   FECHA_PARENTESE
DOIS_PONTOS         INTEIRO           VEZES             SENAO             ESCREVA
ID                  DOIS_PONTOS       ID                RETORNA           ABRE_PARENTESE
FLUTUANTE           ID                ID                ABRE_PARENTESE    ID
DOIS_PONTOS         SE                ATRIBUICAO        NUM_INTEIRO       ABRE_PARENTESE
ID                  ID                ID                FECHA_PARENTESE   ID
ABRE_COLCHETE       MAIOR             MENOS             FIM               FECHA_PARENTESE
NUM_INTEIRO         NUM_INTEIRO       NUM_INTEIRO       FIM               FECHA_PARENTESE
FECHA_COLCHETE      ENTAO             ATE               INTEIRO           RETORNA
INTEIRO             ID                ID                ID                ABRE_PARENTESE
ID                  ATRIBUICAO        IGUAL             ABRE_PARENTESE    NUM_INTEIRO
ABRE_PARENTESE      NUM_INTEIRO       NUM_INTEIRO       FECHA_PARENTESE   FECHA_PARENTESE
INTEIRO             REPITA            RETORNA           LEIA              FIM
DOIS_PONTOS         ID                ABRE_PARENTESE    ABRE_PARENTESE
ID                  ATRIBUICAO        ID                ID
```

Figura 1: Saída do léxico ao processar o código 1 com a flag `-k`

```
LexToken(INTEIRO,'inteiro',1,7)
LexToken(DOIS_PONTOS,':',1,1)
LexToken(ID,'n',1,1)
LexToken(FLUTUANTE,'flutuante',2,9)
LexToken(DOIS_PONTOS,':',2,1)
LexToken(ID,'a',2,1)
LexToken(ABRE_COLCHETE,'[',2,1)
LexToken(NUM_INTEIRO,'10',2,2)
LexToken(FECHA_COLCHETE,']',2,1)
LexToken(INTEIRO,'inteiro',4,7)
LexToken(ID,'fatorial',4,8)
LexToken(ABRE_PARENTESE,'(',4,1)
LexToken(INTEIRO,'inteiro',4,7)
LexToken(DOIS_PONTOS,':',4,1)
LexToken(ID,'n',4,1)
LexToken(FECHA_PARENTESE,')',4,1)
LexToken(INTEIRO,'inteiro',5,7)
LexToken(DOIS_PONTOS,':',5,1)
LexToken(ID,'fat',5,3)
LexToken(SE,'se',6,2)
LexToken(ID,'n',6,1)
LexToken(MAIOR,'>',6,1)
LexToken(NUM_INTEIRO,'0',6,1)
LexToken(ENTAO,'então',6,6)
LexToken(ID,'fat',7,3)
LexToken(ATRIBUICAO,':=',7,2)
LexToken(NUM_INTEIRO,'1',7,1)
LexToken(REPITA,'repita',8,6)
LexToken(ID,'fat',9,3)
LexToken(ATRIBUICAO,':=',9,2)
LexToken(ID,'fat',9,3)
LexToken(VEZES,'*',9,1)
LexToken(ID,'n',9,1)
LexToken(ID,'n',10,1)
LexToken(ATRIBUICAO,':=',10,2)
LexToken(ID,'n',10,1)
LexToken(MENOS,'-',10,1)
LexToken(NUM_INTEIRO,'1',10,1)
LexToken(ATE,'até',11,4)
LexToken(ID,'n',11,1)
LexToken(IGUAL,'=',11,1)
LexToken(NUM_INTEIRO,'0',11,1)
LexToken(RETORNA,'retorna',12,7)
LexToken(ABRE_PARENTESE,'(',12,1)
LexToken(ID,'fat',12,3)
LexToken(FECHA_PARENTESE,')',12,1)
LexToken(SENAO,'senão',13,6)
LexToken(RETORNA,'retorna',14,7)
LexToken(ABRE_PARENTESE,'(',14,1)
LexToken(NUM_INTEIRO,'0',14,1)
LexToken(FECHA_PARENTESE,')',14,1)
LexToken(FIM,'fim',15,3)
LexToken(FIM,'fim',16,3)
LexToken(INTEIRO,'inteiro',18,7)
LexToken(ID,'principal',18,9)
LexToken(ABRE_PARENTESE,'(',18,1)
LexToken(FECHA_PARENTESE,')',18,1)
LexToken(LEIA,'leia',19,4)
LexToken(ABRE_PARENTESE,'(',19,1)
LexToken(ID,'n',19,1)
LexToken(FECHA_PARENTESE,')',19,1)
LexToken(ESCREVA,'escreva',20,7)
LexToken(ABRE_PARENTESE,'(',20,1)
LexToken(ID,'fatorial',20,8)
LexToken(ABRE_PARENTESE,'(',20,1)
LexToken(ID,'n',20,1)
LexToken(FECHA_PARENTESE,')',20,1)
LexToken(FECHA_PARENTESE,')',20,1)
LexToken(RETORNA,'retorna',21,7)
LexToken(ABRE_PARENTESE,'(',21,1)
LexToken(NUM_INTEIRO,'0',21,1)
LexToken(FECHA_PARENTESE,')',21,1)
LexToken(FIM,'fim',22,3)
```

Figura 2: Saída do léxico ao processar o código 1 sem a flag `-k`

Referências

JARGAS, Aurélio Marinho. 2012. *Expressões regulares: uma abordagem divertida*. São Paulo, SP: Novatec 4th edn.

LOUDEN, Kenneth C. 2004. *Compiladores: Princípios e práticas*. São Paulo, SP: Thomson 1st edn.

Menezes, Paulo B. 2011. *Linguagens formais e autômatos - ufrgs*, vol. 3. Porto Alegre: Bookman 6th edn. https://integrada.minhabiblioteca.com.br/reader/books/9788577807994/. E-book, p.80.

Paxson, Vern. 1990. Lexical analysis with flex. Acessado em: 20 de Outubro de 2024. https://westes.github.io/flex/manual/.