Anatomia de um Tokenizador Simples em C

Taina Pereira Pinto da Silva[1], Marcos Vinicius Machado dos Santos[2]

1 Universidade Unijorge, Curso de Engenharia da Computação, Salvador, BA, Brasil
tainapereira@unijorge.br
2 Universidade Unijorge, Curso de Engenharia da Computação, Salvador, BA, Brasil
marcosvinicius@unijorge.br

Resumo

Este trabalho apresenta uma implementação de um tokenizador simples escrito em C. O código tem como objetivo dividir uma string de entrada em tokens, como identificadores, números e operadores. Através deste código, é possível compreender a aplicação de conceitos fundamentais da linguagem C, como enumerações, estruturas e manipulação de strings. O tokenizador serve como uma etapa inicial importante no desenvolvimento de compiladores e interpretadores, realizando a análise léxica da entrada.

1 Informações Gerais

O código desenvolvido é um tokenizador simples que processa uma string de entrada e identifica os tokens presentes nela. Os tokens são componentes básicos que formam a estrutura de uma linguagem de programação, como identificadores (nomes de variáveis), números e operadores. Este processo de divisão da entrada em tokens é fundamental para a construção de compiladores e interpretadores.

O tokenizador é composto por uma função que percorre cada caractere da string de entrada, identificando e classificando-os de acordo com suas

características. A partir dessa análise, o código permite a separação da string em partes significativas, que podem ser usadas em estágios subsequentes de análise sintática.

2 Funcionamento do Código

A implementação do código é feita em C e utiliza enumerações e estruturas para representar os tokens encontrados na entrada. O código define um tipo `Token`, que contém o tipo e o valor do token. A enumeração `TokenType` é usada para classificar os tokens em diferentes tipos: identificador, número, operador, fim de entrada e erro.

A função principal do código é a `getNextToken`, que percorre a string de entrada, caractere por caractere, verificando se cada um deles pertence a um identificador, número, operador ou outro tipo de token. Quando encontra um caractere que corresponde a um tipo de token, ela o armazena em uma estrutura `Token` e avança para o próximo caractere da string.

2.1 Estrutura do Token

A estrutura `Token` foi definida da seguinte forma:

```
typedef struct {
    TokenType type;  // Tipo do token
    char value[MAX_TOKEN_LENGTH];  // Valor do token
} Token;
```

Aqui, `type` representa o tipo do token (como identificador, número, operador), e `value` armazena o conteúdo do token identificado.

2.2 A Função getNextToken

A função `getNextToken` lê a string de entrada e identifica cada token. A função realiza verificações para cada tipo de token, como espaços em branco, identificadores, números e operadores. Quando encontra um caractere que corresponde a um tipo de token, ele é processado e armazenado na estrutura `Token`.

Aqui está a implementação da função `getNextToken`:

```
Token getNextToken(const char **input) {
    Token token;
    token.type = TOKEN_ERROR;
    token.value[0] = '\0';

    while (**input) {
        if (isspace(**input)) {
            (*input)++;
            continue;
        }

        if (isalpha(**input)) {
            int len = 0;
            while (isalnum(**input) && len < MAX_TOKEN_LENGTH - 1) {
                token.value[len++] = *(*input)++;
            }
            token.value[len] = '\0';
            token.type = TOKEN_IDENTIFIER;
            return token;
        }

        if (isdigit(**input)) {
            int len = 0;
            while (isdigit(**input) && len < MAX_TOKEN_LENGTH - 1) {
                token.value[len++] = *(*input)++;
            }
            token.value[len] = '\0';
            token.type = TOKEN_NUMBER;
            return token;
        }

        if (strchr("+-*/", **input)) {
            token.value[0] = *(*input)++;
            token.value[1] = '\0';
            token.type = TOKEN_OPERATOR;
            return token;
        }
```

```
        if (**input == '\0') {
            token.type = TOKEN_END;
            return token;
        }

        token.value[0] = *(*input)++;
        token.value[1] = '\0';
        token.type = TOKEN_ERROR;
        return token;
    }

    token.type = TOKEN_END;
    return token;
}
```

2.3 Impressão dos Tokens

Uma vez que um token é identificado, ele pode ser impresso para o usuário. Para isso, o código inclui a função `printToken`, que exibe o tipo e o valor do token encontrado. A função utiliza um array de strings para mapear os tipos de tokens, como identificador, número, operador, etc.

Aqui está a função de impressão:

```
void printToken(Token token) {
    const char *typeNames[] = {
        "IDENTIFICADOR",
        "NÚMERO",
        "OPERADOR",
        "FIM",
        "ERRO"
    };
    printf("Token: { Tipo: %s, Valor: '%s' }\n", typeNames[token.type], token.valu
}
```

3 Exemplo de Entrada e Saída

Considerando a string de entrada:

```
const char *input = "x = 42 + y";
```

A saída do programa será a seguinte:

```
Token: { Tipo: IDENTIFICADOR, Valor: 'x' }
Token: { Tipo: OPERADOR, Valor: '=' }
Token: { Tipo: NÚMERO, Valor: '42' }
Token: { Tipo: OPERADOR, Valor: '+' }
Token: { Tipo: IDENTIFICADOR, Valor: 'y' }
```

Esses tokens representam a decomposição da string de entrada em unidades significativas que podem ser processadas em etapas subsequentes.

4 Tokens: Uma Visão Geral

Tokens são unidades fundamentais em um processo de análise léxica. Assim como cartões de crédito ou débito são utilizados em transações financeiras, os tokens são elementos que representam partes significativas de um programa de computador. No exemplo de código, os tokens identificados são:

- **Identificadores**: Como 'x' e 'y', que são nomes de variáveis.
- **Números**: Como '42', que é um valor numérico.
- **Operadores**: Como '=', '+', que são utilizados em operações.

Esses tokens são a base para uma análise sintática mais profunda.

5 Conclusão

O código apresentado é um exemplo simples, mas eficaz, de um tokenizador em C. Ele pode ser utilizado como base para entender como a análise léxica funciona, além de servir como ponto de partida para implementações mais complexas em compiladores ou interpretadores. A estrutura modular e organizada facilita a adaptação e a expansão para outras necessidades de análise de linguagem.

6 Referências

As referências são importantes para dar o devido crédito às fontes de conhecimento. Para este trabalho, uma referência comum sobre análise lexical é:

- Aho, A. V., Sethi, R., & Ullman, J. D. (2002). *Compilers: Principles, Techniques, and Tools*. Addison-Wesley.