

Compiladores - Analisador Léxico em C++

Suelen Vicentini Fraga

Introdução

O objetivo deste trabalho é projetar e implementar um analisador léxico em C++, capaz de reconhecer os principais tokens de uma mini linguagem de programação. Esses tokens representam os elementos básicos da linguagem, como identificadores, números, operadores, palavras-chave e delimitadores, que posteriormente serão utilizados pelas fases de análise sintática e semântica. Além da implementação prática, o trabalho também envolve a definição das expressões regulares que descrevem os padrões válidos para cada tipo de token, bem como a construção de uma tabela exemplificando cadeias aceitas e rejeitadas.

Durante o desenvolvimento, foram utilizadas estruturas como a classe `Token` para armazenar informações detalhadas (tipo, valor, linha e coluna) e a classe `Lexer`, responsável por percorrer a entrada e aplicar as regras de reconhecimento. O analisador foi projetado para lidar com situações comuns em linguagens de programação, como comentários, operadores aritméticos e relacionais, além de strings literais. Também foram tratados casos de erro, como strings não finalizadas e caracteres inválidos, retornando tokens especiais do tipo `UNKNOWN`.

Etapa 1 - Expressões regulares e Tabela de Tokens

Antes da implementação, foram definidas as expressões regulares para cada tipo de Token.

Token	Expressões regulares (Regex)	Exemplos válidos	Exemplos inválidos
identificador	[A-Z, a-z][A-Z, a-z, 0-9]*	x, var1, cont	1abc, @nome, #teste
Palavra-chave	(mesma regra de identificador, mas comparado a tabela keywords)	if, else, while, int, float, string, return	iff, inta, whiles
Número(int)	[0-9]	0, 42, 123	12a, .5, 10.
Número(float)	[0-9].[0-9](apenas se houver dígito após o ponto)	3.14, 0.5, 10.0	3., .5, 10.
String	".*" ou "[^"]*"	"Olá", "123", "texto simples"	"não fecha, "abc\"

Comentário	//[^\\n]*	//comentario, //123abc	(não existe inválido, só termina na linha)
Operadores duplos	== != <= >= &&	!=, >=, ==, &&,	=<, >>
Operadores simples	[+\\-*/=\\>%]	+, -, /, %, <, >	?,>>
Pontuação	[0;{}\\[\\]]	;,(),{,}	;,. (a não ser parte de número)
Desconhecido	Qualquer caractere fora das regras acima	@, ?, ^	(qualquer caractere não tratado)

Etapa 2 - Scanner inicial

Após a definição das expressões regulares para os tokens, a segunda etapa do trabalho consistiu na implementação de um scanner inicial em C++. Esse scanner tem como função percorrer o código-fonte de entrada e retornar uma sequência de tokens, cada um representando um símbolo válido da linguagem.

A lógica utilizada segue um fluxo simples:

- O programa lê o arquivo de entrada e converte seu conteúdo em uma única string, representando todo o código-fonte.
- A classe **Lexer** é responsável por percorrer essa string caractere por caractere, utilizando funções auxiliares (**peek** e **get**) para olhar e consumir os caracteres.
- Dependendo do caractere encontrado, o analisador aplica regras de reconhecimento:
 - Se for uma letra, inicia-se o reconhecimento de um palavra-chave ou identificador.
 - Se for um dígito, inicia-se a leitura de um número inteiro ou decimal.
 - Se for " (aspas duplas), o scanner entra em um estado especial para capturar strings literais.
 - Se for / seguido de /, o texto restante da linha é classificado como comentário.
 - Se for um operador ou pontuação, o símbolo é reconhecido imediatamente, com suporte tanto para operadores simples quanto compostos (ex.: ==, !=, <=, >=)
- Cada token reconhecido retorna em um objeto da classe **Token**, contendo seu tipo, valor e também a posição no código (linha e coluna).

Com esse scanner inicial já foi possível realizar testes práticos, como processar pequenos trechos de código-fonte e verificar se a saída correspondia à sequência de tokens esperada. Isso serviu como base para evoluções nas etapas seguintes, como a criação da classe **Token** mais detalhada, modularização do código e o tratamento mais robusto de strings e erros léxicos.

Etapa 3 - Classe Token

Com o scanner funcionando, a próxima etapa foi estruturar melhor as informações geradas durante a análise. Foi então criada a classe **Token**, que tem como finalidade estruturar os dados de cada símbolo reconhecido no código-fonte.

Um token possui informações adicionais que irão auxiliar nas fases seguintes da compilação. Por isso a classe foi projetada para armazenar as seguintes variáveis:

- **Tipo:** que classifica o token de acordo com a sua natureza (**IDENTIFIER**, **NUMBER**, **STRING**, **KEYWORD**, **OPERATOR**, **PUNCTUATION**, **COMMENT**, **UNKNOWN** ou **EOF**).
- **Lexema:** Conteúdo das strings que formam o Token ("if", "x", "42", "+").
- **Linha:** A linha do código-fonte onde o token inicia.
- **Coluna:** A coluna do código-fonte onde o token inicia.

A implementação inclui também o método **toString()**, responsável por converter o token em texto legível no seguinte formato:

```
TIPO -> "Lexema" [linha, coluna]
```

Isso facilita na visualização do usuário, o código-finte também gera um arquivo **.txt** com a saída gerada.

Exemplo prático

Utilizando o pseudocódigo a seguir

```
int x = 42;
```

Produz a seguinte sequência de código

```
KEYWORD -> "int" [1,1]
IDENTIFICADOR -> "x" [1,5]
OPERADOR -> "=" [1,7]
NUMERO -> "42" [1,9]
PONTUACAO -> ";" [1,11]
```

Neste exemplo, o token **int** é classificado como palavra-chave **KEYWORD**, enquanto o **x** é um identificador **IDENTIFIER**. O operador de atribuição **=** é identificado como **OPERATOR**, o número **42** como **NUMBER**, e o **;** como **PUNCTUATION**. Além disso, cada token traz a posição exata no código, o que auxilia a reportar erros de compilação.

Importância desta etapa

Sem a estrutura da classe **token** o scanner retornaria apenas sequências de caracteres, o que dificultaria a análise sintática e a geração de mensagens de erro. Cada token reconhecido possui um formato padronizado, facilitando tanto a depuração quanto a evolução para as etapas seguintes do compilador.

Etapa 4 - Modularização

Embora a implementação inicial do analisador léxico tenha sido feita em um único arquivo, é importante pensar em modularização para tornar o código mais organizado, legível e de fácil manutenção. A modularização consiste em dividir o programa em partes menores e independentes, cada uma com uma responsabilidade bem definida.

No caso deste projeto, a modularização poderia ser aplicada da seguinte forma:

Separação em arquivos

- Criar um arquivo `token.h` contendo a definição da classe `Token` e do enum `TokenType`.
- Criar arquivos `lexer.h` e `lexer.cpp`, responsáveis pela declaração e implementação da classe `Lexer`, respectivamente.
- Manter o `main.cpp` apenas como ponto de entrada do programa, com a responsabilidade de abrir o arquivo-fonte, instanciar o analisador e exibir os tokens.

Dessa forma, cada componente do projeto fica isolado, facilitando futuras alterações sem comprometer o restante do código.

Uso de funções auxiliares

Em vez de concentrar toda a lógica dentro da função `nextToken()`, o reconhecimento de cada tipo de token pode ser delegado a funções específicas, como:

- `scanIdentifierOrKeyword()`
- `scanNumber()`
- `scanString()`
- `scanComment()`
- `scanOperator()`
- `scanPunctuation()`

Isso reduz a complexidade da função principal e torna o código mais próximo de uma máquina de estados explícita, além de facilitar a depuração e os testes.

Uso de macros ou constantes para regex simplificadas

Expressões regulares ou padrões recorrentes podem ser definidos como constantes globais ou macros. Por exemplo:

```
const string IDENTIFIER_REGEX = "[A-Za-z_][A-Za-z0-9_]*";  
const string NUMBER_REGEX = "\d+(\.\d+)?";
```

Dessa forma, se o padrão precisar ser alterado no futuro, basta ajustar a constante em um único local.

Importância da modularização

A modularização é fundamental para projetos maiores pois:

- Melhora a legibilidade
- Facilita a manutenção
- Favorece a reutilização
- Prepara para etapas futuras de compilação

Mesmo que nesta versão do trabalho a modularização ainda não tenha sido implementada, sua importância é clara e a sua adoção em etapas futuras tornará o compilador mais robusto e escalável.

Etapa 5 - Reconhecimento de strings literais

Uma das partes mais complexas da análise léxica é o reconhecimento de **strings literais**, pois elas podem conter espaços, caracteres especiais e até mesmo sequências de escape. Diferente de identificadores ou números, que seguem padrões mais simples, as strings precisam de regras específicas para serem tratadas.

Na implementação do analisador léxico, foi incluído um estado que entra em ação sempre que o scanner encontra **aspas dupla** (""). Ele então percorre os caracteres até encontrar outra aspas dupla que indique o fim da string. Durante esse processo, são levadas em considerações as seguintes situações:

- Strings comuns (Exemplo: "Olá, mundo").
- Strings com sequência de escape (Exemplo: "Linha1\nLinha2" ou "Ele disse: \"oi\"")
 - O analisador consome a barra invertida (\) e também o caractere seguinte, para então garantir que a sequencia faça parte da string.
- Strings não finalizadas (Exemplo: "Texto sem aspas finais")
 - Nesse caso, o analisador percorre até o fim do arquivo sem encontrar a aspa de fechamento. Para tratar esse erro, foi criado um token especial do tipo **UNKNOWN** informando então a mensagem "(String nunca foi fechada)".
 - Esse recurso é importante porque impede que o analisador entre o loop infinito e permite identificar facilmente o problema no código-fonte.

Exemplo prático

Entrada:

```
string s1 = "Olá, mundo!";
string s2 = "Exemplo com \"aspas\" dentro";
string s3 = "String com erro
```

Saida:

```
KEYWORD -> "string" [1,1]
IDENTIFICADOR -> "s1" [1,8]
OPERADOR -> "=" [1,11]
STRING -> "\"Olá, mundo!\"" [1,13]
PONTUACAO -> ";" [1,26]
KEYWORD -> "string" [2,1]
```

IDENTIFICADOR -> "s2" [2,8]
OPERADOR -> "=" [2,11]
STRING -> "\"Exemplo com \\"aspas\\\" dentro\\"" [2,13]
PONTUACAO -> ";" [2,41]
KEYWORD -> "string" [3,1]
IDENTIFIER -> "s3" [3,8]
OPERATOR -> "=" [3,11]
UNKNOWN -> "\"String com erro (String nunca foi fechada)" [3,13]

Importância desta etapa

O tratamento de strings é fundamental para qualquer linguagem de programação, sem ele o programador não poderia manipular textos, o que limitaria as possibilidades da linguagem. Além disso, o tratamento de erros como as strings não finalizadas, reforça o analisador léxico, permitindo que ele não apenas identifique tokens validos mas também reporte problemas.

Essa etapa torna o compilador mais próximo de um uso real, já que na prática é comum que programas contenham strings com espaços, caracteres especiais e até erros de digitação que precisam ser tratados com mensagens adequadas.

Conclusão

O desenvolvimento deste analisador léxico em C++ permitiu compreender, de forma prática, a importância da análise de tokens no processo de compilação. A partir da implementação, foi possível observar como um código-fonte é decomposto em unidades menores e significativas, que servirão de base para as próximas fases: análise sintática e análise semântica.

O programa reconhece os principais elementos de uma linguagem de programação simplificada, incluindo **identificadores, palavras-chave, números inteiros e decimais, operadores aritméticos e relacionais, strings, comentários e símbolos de pontuação**. Além disso, foram tratados casos de erro, como strings não finalizadas, retornando tokens especiais do tipo **UNKNOWN** que facilitam a identificação de problemas no código-fonte.

Embora a modularização ainda não tenha sido aplicada, a implementação atual já apresenta uma arquitetura flexível, que pode ser facilmente reorganizada em múltiplos arquivos e funções auxiliares. Esse passo será essencial para a evolução do compilador, principalmente quando for necessário integrá-lo às etapas de parsing e análise semântica

Dificuldades encontradas

Durante o desenvolvimento, algumas dificuldades práticas e conceituais foram enfrentadas:

- **Reconhecimento de operadores compostos:** inicialmente, apenas operadores simples (+, -, *, etc.) eram identificados. Foi necessário expandir a lógica para operadores compostos como ==, !=, <=, >, além dos operadores lógicos && e ||.
- **Entendimento das expressões regulares:** a definição das regex para cada token demandou atenção, especialmente no caso das strings, que envolvem escapes (\", \\n) e exigem cuidados para não aceitar sequências inválidas. A simplificação do

regex para documentação ajudou, mas compreender como implementar isso em código foi um desafio inicial.

- **Tratamento de erros em strings:** implementar a detecção de strings não finalizadas foi outro ponto que exigiu reflexão, já que sem esse cuidado o analisador poderia entrar em loop infinito ou aceitar tokens inválidos.
- **Diferenciar Identificadores de keywords:** Entender a diferença prática entre identificar um identificador e uma palavra-chave (mesmo ambos seguindo a mesma regex).