Manual do Usuário - Simple C

Felipe Gomes da Silva
Luis Henrique Salomão Lobato

Setembro, 2025

Sumário

1 Introdução

Este manual do usuário fornece uma visão geral do Simple C, uma linguagem de programação que visa simplicar o uso de conceitos fundamentais da linguagem C. O Simple C é projetado para ser fácil de aprender e usar, tornando-o ideal para iniciantes em programação, mas removendo algumas funcionalidades avançadas da linguagem C.

Neste momento, abordaremos apenas o analisador léxico da linguagem. Utilizamos a ferramenta Flex para construir o analisador léxico, que é responsável por identificar e classificar os tokens na entrada do código-fonte.

2 Testando o Analisador Léxico

Para testar o analisador léxico do Simple C, siga os passos abaixo:

1. Certifique-se de ter o Flex instalado em seu sistema. Você pode verificar isso executando o comando `flex -version` no terminal.

2. Clone o repositório do Simple C do GitHub:

   ```
   git clone github.com/felipe-gsilva/simple-c
   ```

3. Navegue até o diretório do projeto:

   ```
   cd simple-c/
   ```

4. Gere o analisador léxico usando o Flex e o compile com nosso script de build

   ```
   chmod +x build.sh
   ./build.sh
   ```

5. Execute o analisador léxico com um arquivo de entrada que contenha código Simple C:

   ```
   ./build/simplec < input_file.c
   ```

   Substitua `input_file.c` pelo caminho do arquivo que você deseja analisar.

6. Teste os analisadores com:

   ```
   chmod +x test.sh
   ./test.sh
   ```

Foram criados alguns arquivos de teste na pasta `tests` do repositório. Você pode usar esses arquivos para verificar o funcionamento do analisador léxico. Cada arquivo de teste contém exemplos de código Simple C que abrangem diferentes aspectos da linguagem, como declarações de variáveis, estruturas de controle, funções, entre outros.

Tenha em vista que o analisador sintático ainda não foi implementado, portanto, mudanças na gramática podem ocorrer futuramente.

3 Tokens Reconhecidos

O analisador léxico da linguagem Simple C toma a decisão de aceitar ou não determinada sentença com base nos tokens destacados na Tabela 1

Tabela 1: Tabela de Tokens do Simple C

Nome do Token	Lexema(s)	Descrição
Palavras-chave: Tipos de Dados		
KEYWORD_INT	`int`	Palavra-chave para tipo inteiro.
KEYWORD_FLOAT	`float`	Palavra-chave para tipo ponto flutuante.
KEYWORD_CHAR	`char`	Palavra-chave para tipo caractere.
KEYWORD_STRING	`string`	Palavra-chave para tipo string (cadeia de caracteres).
KEYWORD_VOID	`void`	Palavra-chave para tipo vazio/nulo.
KEYWORD_BOOL	`bool`	Palavra-chave para tipo booleano.
Palavras-chave: Controle de Fluxo e Comandos		
KEYWORD_IF	`if`	Inicia uma estrutura condicional.
KEYWORD_ELSE	`else`	Bloco alternativo de uma estrutura condicional.
KEYWORD_FOR	`for`	Inicia um laço de repetição ‘for‘.
KEYWORD_WHILE	`while`	Inicia um laço de repetição ‘while‘.
KEYWORD_DO	`do`	Inicia um laço de repetição ‘do-while‘.
KEYWORD_SWITCH	`switch`	Inicia uma estrutura de seleção múltipla.
KEYWORD_CASE	`case`	Define um rótulo dentro de um ‘switch‘.
KEYWORD_DEFAULT	`default`	Define o rótulo padrão de um ‘switch‘.
KEYWORD_BREAK	`break`	Interrompe a execução de um laço ou ‘switch‘.
KEYWORD_CONTINUE	`continue`	Pula para a próxima iteração de um laço.
KEYWORD_RETURN	`return`	Retorna um valor de uma função.
Identificadores e Literais		
IDENTIFICADOR	`var`, `_x`, ...	Nome de variável, função, etc.
INT	`123`, `42`	Valor literal inteiro.
FLOAT	`3.14`, `0.5`	Valor literal de ponto flutuante.
CHAR	`'a'`, `'\n'`	Valor literal de caractere.
STRING	`"hello"`	Valor literal de string.
BOOLEAN_LITERAL	`true`, `false`	Valor literal booleano.
Operadores		
OP_SOMA	`+`	Operador de adição.
OP_SUB	`-`	Operador de subtração.
OP_MULT	`*`	Operador de multiplicação.
OP_DIV	`/`	Operador de divisão.
OP_MOD	`%`	Operador de módulo (resto da divisão).
OP_ATRIBUICAO	`=`	Operador de atribuição simples.
OP_INC_ATRIBUICAO	`+=`	Operador de atribuição com adição.
OP_DEC_ATRIBUICAO	`-=`	Operador de atribuição com subtração.
OP_MULT_ATRIBUICAO	`*=`	Operador de atribuição com multiplicação.
OP_DIV_ATRIBUICAO	`/=`	Operador de atribuição com divisão.
OP_INC	`++`	Operador de incremento.

Tabela 1: Tabela de Tokens do Simple C (Continuação)

Nome do Token	Lexema(s)	Descrição
OP_DEC	`--`	Operador de decremento.
OP_IGUAL	`==`	Operador relacional de igualdade.
OP_DIFERENTE	`!=`	Operador relacional de desigualdade.
OP_MENOR	`<`	Operador relacional menor que.
OP_MAIOR	`>`	Operador relacional maior que.
OP_MENOR_IGUAL	`<=`	Operador relacional menor ou igual que.
OP_MAIOR_IGUAL	`>=`	Operador relacional maior ou igual que.
OP_AND	`&&`	Operador lógico E (AND).
OP_OR	`||`	Operador lógico OU (OR).
OP_NOT	`!`	Operador lógico de negação (NOT).
Pontuadores e Delimitadores		
PONTO_VIRGULA	`;`	Finalizador de instrução.
DOIS_PONTOS	`:`	Usado em casos de 'switch'.
VIRGULA	`,`	Separador de elementos (ex: em listas).
ABRE_PARENTESES	`(`	Abre lista de parâmetros ou expressão.
FECHA_PARENTESES	`)`	Fecha lista de parâmetros ou expressão.
ABRE_CHAVES	`{`	Abre um bloco de código.
FECHA_CHAVES	`}`	Fecha um bloco de código.
ABRE_COLCHETES	`[`	Abre a declaração/acesso de um array.
FECHA_COLCHETES	`]`	Fecha a declaração/acesso de um array.

4 Análise de Erros

O Simple C implementa um tratamento de erros em duas fases: léxica (Flex) e sintática (Bison).

4.1 Tratamento de Erros Léxicos (Flex)

O analisador léxico é configurado para detectar e reportar falhas específicas, além de caracteres desconhecidos, garantindo um diagnóstico preciso no código-fonte. Para isto, são utilizadas as variáveis globais `current_line` e `current_col`, que rastreiam a posição exata de cada token reconhecido.

A função de erro léxico, `print_error`, reporta a linha, coluna, o tipo de erro e o texto problemático (`yytext`). Erros capturados incluem:

- Identificadores que iniciam com dígitos.
- Literais de caracteres contendo múltiplos caracteres.
- Construtos não fechados, como comentários de bloco (`/*`) ou strings (`"..."`) no fim do arquivo ou com quebras de linha inesperadas.

4.2 Tratamento de Erros Sintáticos (Bison)

A detecção de erros sintáticos é responsabilidade do Analisador Sintático (Bison). Um erro ocorre quando a sequência de tokens fornecida pelo Flex não pode ser reduzida a nenhuma regra de produção da Gramática Livre de Contexto (G).

4.2.1 Função de Erro Sintático e Rastreamento

Quando um erro de sintaxe é encontrado, o Bison invoca a função `yyerror(...)`. Esta função aproveita as variáveis globais `current_line` e `current_col` (mantidas pelo Flex) para informar ao usuário a linha e coluna exatas onde a violação sintática foi detectada.

A função de erro, definida no arquivo Bison, é:

```
void yyerror(const char *s) {
    fprintf(stderr,
    "Erro Sintatico na Linha %d, Coluna %d: %s\n",
    current_line,
    current_col,
    s
    );
    exit(1);
}
```

Código 1: Função para tratamento de erros

4.2.2 Precedência e Associatividade

A ambiguidade sintática, especialmente em expressões complexas, é resolvida pela definição explícita de Precedência e Associatividade dos operadores no arquivo Bison (usando diretivas como ‘left‘, ‘right‘). Isto garante que a estrutura sintática preferida siga a ordem de avaliação matemática e lógica esperada (ex: multiplicação antes da soma).

5 Gerenciamento de Estados

Para melhor gerenciar o estado do analisador léxico, utilizamos a funcionalidade de estados do Flex. Em especial, foram criados 2 estados diferentes: `IN_COMMENT` e `IN_STRING`.

5.1 Comentários

Os comentários podem aparecer de 2 formas no código: (i) comentários de linha única, que começam com `//` e se estendem até o final da linha; (ii) comentários de bloco, que começam com `/*` e terminam com `*/`.

5.2 Strings

As strings são sequências de caracteres delimitadas por aspas duplas (¨). O analisador léxico reconhece strings e trata caracteres de escape, como `\n` para nova linha e `\t` para tabulação.

Para isso, foi-se utilizado a função `yymore()` do Flex, que permite concatenar a string lida em `yytext` com a próxima parte da string lida, até que o caractere de fechamento (¨) seja encontrado.

Com isso, é possível a declaração e atribuição de strings com múltiplas linhas. Segue no Código 2, exemplos diversos sobre este tipo de dado.

```
// Teste de literais de string, incluindo escapes
string s = "Uma string de teste com \"aspas\" escapadas.";
string outra_string_123 = "outra string";
string string_com_3_linhas =
"""
Linha 1\nLinha 2\nLinha 3
""";
```

Código 2: Exemplos para o tratamento de strings.

6 Exemplos de uso

Nesta seção serão descritos exemplos do código da Linguagem Simple C, visando introduzir conceitos básicos sobre a sintaxe para o uso da mesma.

6.1 Declaração de Variáveis e Tipos de Dados

Abaixo é apresentado um trecho de código de exemplo da linguagem Simple C, que será processado pelo Analisador Léxico (Seção 3). As cores indicam os tokens reconhecidos, conforme o estilo definido:

```
    int i = 12345;
    int i_2 = -12345;
    float _f = .5; // Ponto flutuante implicito
    float _f2 = 0.555555;
    float _f3 = 1.000000;
    char c = 'A';
    bool flag = true;
    char newline = '\n'; // Teste de escape em char
    int resultado_final = 0;
```

Código 3: Exemplo de Código com Tokens da Simple C

6.2 Estruturas Condicionais e de Controle

Seguem exemplos para estruturas condicionais e de controle. No Código 4, apresenta-se a utilização de um "If-Else". Já no Código 5, está definida a estrutura "Switch-Case".

```

    // Estrutura condicional 'if-else' com operadores logicos
    //e relacionais
    if (flag == true && (i < 1000 || i >= 2000)) {
        resultado_final = 1;
    } else if (!flag) {
        resultado_final = -1;
    }
```

Código 4: Exemplo de estrutura de Controle If-Else

```
    switch (c) {
        case 'A':
            i = 1;
            break;
        case 'B':
            i = 2;
            break;
        default:
            i = 0;
    }
```

Código 5: Exemplo de estrutura de Controle Switch-Case

6.3 Laços de Repetição

A linguagem Simple C suporta três estruturas de repetição fundamentais, que permitem a execução iterativa de blocos de código: o laço `for`, o laço `while` e o laço `do-while`. O

uso dos comandos `break` e `continue` (Tokens listados na Tabela 1) é permitido dentro dos corpos de todos os laços para controlar o fluxo de execução.

6.3.1 Exemplo de Laço `for`

O laço `for` no Simple C segue a estrutura padrão, onde a inicialização, a condição de parada e o passo de iteração são explicitados entre parênteses.

```
    for (int j = 0; j != 10; j++) {
        if (j <= 1) {
        continue; // Pula para a proxima iteracao
    }
        if (j > 8) {
            break; // Interrompe o laco
        }
    }
```

Código 6: “Laço de repetição For Loop”

6.3.2 Exemplo de Laços `while` e `do-while`

Os laços condicionais `while` e `do-while` são implementados com base na avaliação de uma expressão relacional. Note que o `do-while` garante pelo menos uma execução do bloco.

```
    int k = 10;
    while(k > 5) {
        k--;
    }

    do {
        k++;
    } while (k < 10);
```

Código 7: “Laços de Repetição While e Do While”