Primeiro Trabalho Compiladores

Luan Marques Batista
Otavio Augusto Teixeira
Instituto de Biociências,Letras e Ciências Exatas, Unesp -
Univ Estadual Paulista (São Paulo State University)
Rua Cristóvão Colombo 2265, Jd Nazareth, 15054-000, São José do Rio Preto - SP, Brazil
e-mail: luan.batista@unesp.br
e-mail: otavio.a.teixeira@unesp.br

August 30, 2024

Abstract

Neste relatório, abordaremos a construção de uma linguagem de programação que chamaremos de: KNOT. Também discutiremos o que é um Analisador Léxico, as ferramentas utilizadas e o que é uma análise léxica.

1 Introdução

Para a construção deste relatório, introduziremos alguns conceitos básicos para o entendimento do leitor.

1.1 Análise Léxica

Aos leitores que já possuem conhecimentos prévios sobre o tópico, podem pular esta seção. Aos que não conhecem, terei o prazer de explicar.

A análise léxica é a primeira fase do processo de compilação ou interpretação de um programa de computador. Ela é responsável por ler o código-fonte, que é uma sequência de caracteres, e transformá-lo em uma sequência de unidades de significado chamadas *tokens*. Cada *token* representa um elemento atômico da linguagem de programação, como palavras-chave, operadores, identificadores, literais, ou símbolos de pontuação.

O principal objetivo da análise léxica é simplificar a complexidade do código-fonte, segmentando-o em componentes menores e mais manejáveis. Ao transformar o texto em *tokens*, a análise léxica facilita as fases subsequentes de compilação, como a análise sintática e semântica, que serão abordadas futuramente em outros relatórios.

1.2 Exemplo

Um exemplo de uma análise léxica pode ser visto a seguir. Observe a seguinte expressão:

```
int x = 10 + y;
```

Como visto anteriormente, a análise léxica quebra o código em *tokens*, e os *tokens* gerados a partir desta expressão são os seguintes:

- O caracter **int** será identificada como uma palavra-chave.
- A letra **x** será identificada como um identificador
- O símbolo = será identificado como um operador de atribuição.
- O decimal 10 será identificado como um número inteiro.
- O símbolo + será identificado como um operador aritmético.
- A letra *y* será identificada como um identificador.
- O símbolo ; será identificado como um delimitador.

1.3 Flex / Lex

O Flex é uma ferramenta que possibilita a geração automática de analisadores léxicos, também conhecidos como lexers ou scanners. Esses analisadores léxicos são usados para transformar a entrada de texto, como o código-fonte de um programa, em uma sequência de *tokens*, que são unidades significativas para a análise sintática posterior. O Flex é frequentemente usado em conjunto com ferramentas de análise sintática, como o Bison, para construir compiladores, interpretadores, e outras aplicações que necessitam de processamento de linguagem.

Para a criação de regras léxicas usando o Flex, é preciso que o desenvolvedor utilize expressões regulares, como a expressão a seguir:

```
[0-9]+     { printf("Número: %s\n", yytext); }
[a-zA-Z]+  { printf("Palavra: %s\n", yytext); }
.          { printf("Caractere: %s\n", yytext); }
```

Dessa forma, o analisador será capaz de identificar um número e um identificador quando lidos.

2 KNOT

Knot é uma linguagem de programação baseada na linguagem C. Ela foi pensada e desenvolvida durante uma das aulas de Compiladores. A seguir, podemos visualizar a definição formal da linguagem.

2.1 Definição Formal

Vamos definir a linguagem usando a notação formal para as expressões regulares e descrever o comportamento do lexer.

2.1.1 Alfabeto

O alfabeto Σ da linguagem inclui:

- Letras: $a - z$, $A - Z$
- Dígitos: 0 – 9
- Símbolos: +, −, ∗, /, =, <, >, !, &, |, ”, {, }, (,), [,], ,, ;
- Espaços em branco: \t, \r, \n, ” ” (espaço)
- Caracteres de escape: \

2.1.2 Tokens

Cada *token* na linguagem é definido por uma expressão regular:

- Número Inteiro: `NUMBER` → `[0-9]+`
- Identificador: `ID` → `[A-Za-z][A-Za-z0-9]*`
- Espaço: `ESPACO` → `[\t\r" "]+`
- Operadores Relacionais: `OP_REL` → `"=="|"!="|"<"|">"|"<="|">="`
- Operadores Aritméticos: `OP_ARIT` → `+|-|*|/`
- Operadores Lógicos: `OP_LOG` → `"&&" | "||" | "!"`
- Operador de Atribuição: `ATRIBUICAO` → `"="`
- Comentário: `DELIMITADOR_COMENTARIO` → ”\\”
- Delimitadores: `DELIMITADOR` → `[(){}[],;]`
- Palavras-chave: `KEY_WORD` → `"if"|"else"|"while"|"for"|"int"|"float"|"bool"|"string"`
- String: `STRING` → `\" ([^\\"]|\\.)* \"`

2.1.3 Regras Léxicas

As regras do lexer associam essas expressões regulares a ações específicas, como a impressão do tipo de *token* ou a manipulação de erros:

- {ESPACO}* → `printf("Espaço:  %s \n", yytext);`
- \n → `linhas++;`
- {KEY_WORD} → `printf("PALAVRA RESERVADA: %s \n", yytext);`
- {DELIMITADOR} → `printf("DELIMITADOR: %s \n", yytext);`
- {COMENTARIO}[^\n]* → `printf("COMENTÁRIO \n");`

- {ID} → printf("IDENTIFICADOR: %s \n", yytext);
- {ID}{NUMBER}*{ID}* → printf("IDENTIFICADOR: %s \n", yytext);
- {NUMBER}+ → printf("INTEIRO: %s \n", yytext);
- "-"|"+"{NUMBER}+ →
  printf("INTEIRO COM SINAL: %s \n", yytext);
- {NUMBER}+"."{NUMBER}+ → printf("REAL: %s \n", yytext);
- "-"|"+"{NUMBER}+"."{NUMBER}+ →
  printf("REAL COM SINAL: %s \n", yytext);
- {STRING} → printf("STRING: %s \n", yytext);
- {OP_ARIT} → printf("OPERADOR ARITMÉTICO: %s \n", yytext);
- {OP_LOG} → printf("OPERADOR LÓGICO: %s \n", yytext);
- {OP_REL} → printf("OPERADOR RELACIONAL: %s \n", yytext);
- {ATRIBUICAO} → printf("OPERADOR DE ATRIBUIÇÃO: %s \n", yytext);
- {NUMBER}+"."{NUMBER}*{ID}+ →
  printf("Número não suportado: %s (Utilize . como separador) - ",
  yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- {NUMBER}+","{NUMBER}+ →
  printf("Número mal formatado: %s - ", yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- {NUMBER}+{ID} →
  printf("Identificador inválido: %s - ", yytext);
  erros++;
  printf("Linha: %d \n", linhas);
- . →
  printf("ERRO: %c \n", yytext[0]);
  erros++;
  printf("Linha: %d \n", linhas);

Dessa forma, o lexer conseguirá realizar a quebra do código em diversos *tokens*.

3 Modo de Usar

Para utilizar o analisador léxico, siga os passos abaixo:

1. Primeiramente, execute o comando `flex` para gerar o analisador léxico a partir do arquivo `knot.l`. O comando é:

   ```
   flex knot.l
   ```

2. Após gerar o código fonte do analisador, compile-o utilizando o `gcc` para criar o arquivo executável. Use o comando:

   ```
   gcc lex.yy.c -o knot
   ```

3. Por fim, execute o analisador com um arquivo de entrada para ver a análise no terminal. Suponha que o arquivo de entrada seja `teste.txt`. Use o comando:

   ```
   ./knot < teste.txt
   ```

4. Como resultado, você verá a análise do conteúdo de `teste.txt` exibida no terminal.

4 Conclusão

Neste relatório, exploramos a construção da linguagem KNOT, inspirada em C, e discutimos a importância da análise léxica no processo de compilação. Através do uso de ferramentas como o Flex, conseguimos automatizar a geração de analisadores léxicos, simplificando a tarefa de segmentar e classificar o código-fonte em *tokens* significativos. A formalização de KNOT nos permitiu entender como uma linguagem de programação é estruturada desde os componentes mais fundamentais. Este conhecimento é crucial para o desenvolvimento de compiladores e interpretadores eficientes e robustos.