Um compilador simples de uma passagem

Juntando as técnicas

Prof. Edson Alves

Faculdade UnB Gama

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa
- Ele será implementado a partir da tradução dirigida a sintaxe
- As expressões conterão números, identificadores e os operadores `+`, `-`, `*`, `/`, `div` e `mod`
- Os tokens são: **id**, para identificadores, **num**, para constantes inteiras e **eof** para fim de arquivo
- Cada módulo será implementado em um par de arquivos `.cpp` e `.h`, exceto pelo módulo principal (`main.cpp`)

Especificação para um tradutor infixa-posfixa

$$
\begin{array}{lcll}
\mathit{inicio} & \rightarrow & \mathit{lista}\ \textbf{eof} & \\
\mathit{lista} & \rightarrow & \mathit{expr}\ \texttt{;}\ \{\ \mathit{imprimir}(\texttt{'\textbackslash n'})\ \}\ \mathit{lista} & \\
 & | & \epsilon & \\
\mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{termo} & \{\ \mathit{imprimir}(\texttt{'+'})\ \} \\
 & | & \mathit{expr}\ \texttt{-}\ \mathit{termo} & \{\ \mathit{imprimir}(\texttt{'-'})\ \} \\
 & | & \mathit{termo} & \\
\mathit{termo} & \rightarrow & \mathit{termo}\ \texttt{*}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{'*'})\ \} \\
 & | & \mathit{termo}\ \texttt{/}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{'/'})\ \} \\
 & | & \mathit{termo}\ \textbf{div}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{"DIV"})\ \} \\
 & | & \mathit{termo}\ \textbf{mod}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{"MOD"})\ \} \\
 & | & \mathit{fator} & \\
\mathit{fator} & \rightarrow & (\mathit{expr}) & \\
 & | & \textbf{id} & \{\ \mathit{imprimir}(\textbf{id}.\mathit{lexema})\ \} \\
 & | & \textbf{num} & \{\ \mathit{imprimir}(\textbf{id}.\mathit{valor})\ \}
\end{array}
$$

Descrição dos tokens

Lexema	Token	Atributo
espaço em branco		
sequência de dígitos	`NUM`	valor númerico da sequência
div	`DIV`	
mod	`MOD`	
sequências iniciada em letra e seguida de letras e dígitos	`ID`	lexema
caractere de fim de arquivo	`DONE`	
qualquer outro caractere	o próprio caractere	

Módulo `main.cpp`

- Este módulo é responsável pela início da execução do programa
- Ele simplesmente invoca o tradutor

```
1 #include "parser.h"
2
3 int main()
4 {
5     parser::parse();
6
7     return 0;
8 }
```

Módulo `token`

- Este módulo define uma estrutura para a representação de um token
- Cada token tem um tipo e um atributo associado
- Como o atributo de `NUM` é inteiro e de `ID` é string, foi utilizado o tipo `variant<int, string>` de C++17
- Os tipos dos tokens foram codificados como inteiros, com valores foram da faixa ASCII, para evitar conflitos com os tokens compostos por um único caractere
- Também foi definida uma função para a impressão de um token, que trata internamente as diferenças entre os tipos

Arquivo token.h

```
#ifndef TOKEN_H
#define TOKEN_H

#include <string>
#include <variant>

enum TokenType { NUM = 256, DIV, MOD, ID, DONE };

struct Token {
    int type;
    std::variant<int, std::string> value;

    Token(int t, int v) : type(t), value(v) { }
    Token(int t = DONE, std::string s = "") : type(t), value(s) { }
};

std::ostream& operator<<(std::ostream& os, const Token& token);

#endif
```

Arquivo token.cpp

```
1 #include <ostream>
2 #include "token.h"
3
4 std::ostream&
5 operator<<(std::ostream& os, const Token& token)
6 {
7     switch (token.type) {
8     case NUM:
9         os << "NUM (" << std::get<int>(token.value) << ")";
10        break;
11
12    case ID:
13        os << "ID (" << std::get<std::string>(token.value) << ")";
14        break;
15
16    case DIV:
17        os << "DIV";
18        break;
```

Arquivo token.cpp

```
20      case MOD:
21          os << "MOD";
22          break;
23
24      case DONE:
25          os << "DONE";
26          break;
27
28      default:
29          os << (char) token.type;
30      }
31
32      return os;
33  }
```

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador
- A estrutura que armazena os símbolos é um dicionário (classe `map` de C++), onde as chaves são os lexemas e os valores são os tokens
- Na inicialização da tabela são adicionadas, ao dicionário, todas as palavras reservadas
- O método `insert()` insere um novo símbolo no dicionário
- O método `find()` localizar um símbolo já inserido, ou retorna vazio, caso o dicionário não tenha nenhum token associado ao lexema passado como parâmetro

Arquivo table.h

```
#ifndef SYMBOL_TABLE_H
#define SYMBOL_TABLE_H

#include <bits/stdc++.h>
#include "token.h"

class SymbolTable {
public:
    static SymbolTable& get_instance();

    void insert(const std::string& lexema, const Token& token);
    std::optional<Token> find(const std::string& lexema) const;

private:
    SymbolTable();
    std::map<std::string, Token> table;
};

#endif
```

Arquivo table.cpp

```cpp
#include "table.h"

SymbolTable& SymbolTable::get_instance()
{
    static SymbolTable instance;
    return instance;
}

SymbolTable::SymbolTable()
{
    insert("div", Token(DIV));
    insert("mod", Token(MOD));
}

void
SymbolTable::insert(const std::string& lexema, const Token& token)
{
    table[lexema] = token;
}
```

Arquivo table.cpp

```
21 std::optional<Token>
22 SymbolTable::find(const std::string& lexema) const
23 {
24     if (table.count(lexema))
25         return table.at(lexema);
26
27     return { };
28 }
```

Módulo `scanner`

- Este módulo implementa o analisador léxico do tradutor
- Como este analisador não tem estado, ele foi implementado por meio de um `namespace`, o que permite usar a mesma notação de método estático de um classe, embora a implementação seja a de uma função regular de C++
- A função `next_token()` extrai o próximo token da entrada padrão
- O *scanner* ignora todos os espaços em branco
- Os demais tokens são extraídos conforme a especificação

Arquivo scanner.h

```
1  #ifndef SCANNER_H
2  #define SCANNER_H
3
4  #include "token.h"
5
6  namespace scanner {
7
8      Token next_token();
9
10 };
11
12 #endif
```

Arquivo scanner.cpp

```cpp
#include <iostream>
#include "scanner.h"
#include "table.h"

namespace scanner {

    Token next_token()
    {
        auto c = std::cin.get();

        while (isspace(c))
            c = std::cin.get();

        if (c == EOF)
            return { DONE, "" };

        if (isdigit(c))
        {
            std::cin.unget();
```

Arquivo scanner.cpp

```
21              int value;
22              std::cin >> value;
23
24              return { NUM, value };
25          }
26
27          if (isalpha(c))
28          {
29              std::string lexema;
30
31              while (isalpha(c))
32              {
33                  lexema.push_back(c);
34                  c = std::cin.get();
35              }
36
37              std::cin.unget();
```

Arquivo `scanner.cpp`

```
39              auto table = SymbolTable::get_instance();
40              auto token = table.find(lexema);
41
42              if (not token)
43              {
44                  table.insert(lexema, Token(ID, lexema));
45                  return { ID, lexema };
46              } else
47                  return token.value();
48          }
49
50          if (isgraph(c))
51              return Token(c);
52
53          return Token(DONE);
54      }
55  }
```

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa
- Ele invoca o *scanner* para obter os tokens da entrada, um por vez
- Cada não-terminal da gramática é implementado por meio de um procedimento
- Cada expressão da entrada será traduzida para uma linha da saída, em notação posfixa
- As expressões da entrada devem ser terminadas por **;**

Arquivo parser.h

```
1  #ifndef PARSER_H
2  #define PARSER_H
3
4  namespace parser {
5
6      void parse();
7
8  };
9
10 #endif
```

Arquivo parser.cpp

```cpp
#include "scanner.h"
#include "parser.h"
#include "table.h"
#include "error.h"

Token lookahead;

namespace parser {

    void expr();
    void termo();
    void fator();

    void reconhecer(const Token& token);
    void print(const Token& token);
```

Arquivo parser.cpp

```
17      void parse()
18      {
19          lookahead = scanner::next_token();
20
21          while (lookahead.type != DONE)
22          {
23              expr();
24              reconhecer(Token(';'));
25              std::cout << '\n';
26          }
27      }
28
29      void expr()
30      {
31          termo();
```

Arquivo parser.cpp

```
33          while (true)
34          {
35              if (lookahead.type == '+' or lookahead.type == '-')
36              {
37                  auto t = lookahead;
38
39                  reconhecer(lookahead);
40                  termo();
41                  print(t);
42              }
43              else
44                  break;
45          }
46      }
47
48      void termo()
49      {
50          fator();
```

Arquivo parser.cpp

```
52          while (true) {
53              switch (lookahead.type) {
54              case '*':
55              case '/':
56              case DIV:
57              case MOD:
58              {
59                  auto t = lookahead;
60                  reconhecer(lookahead);
61                  fator();
62                  print(t);
63                  break;
64              }
65
66              default:
67                  return;
68              }
69          }
70      }
```

Arquivo parser.cpp

```
72    void fator()
73    {
74        switch (lookahead.type) {
75        case '(':
76            reconhecer(Token('('));
77            expr();
78            reconhecer(Token(')'));
79            break;
80
81        case NUM:
82        case ID:
83            print(lookahead);
84            reconhecer(lookahead);
85            break;
86
87        default:
88            erro("Erro de sintaxe em fator");
89        }
90    }
```

Arquivo parser.cpp

```
92      void reconhecer(const Token& token)
93      {
94          if (token.type == lookahead.type)
95              lookahead = scanner::next_token();
96          else
97              erro("Erro de sintaxe em reconhecer");
98      }
99
100     void print(const Token& token)
101     {
102         switch (token.type) {
103         case '+':
104         case '-':
105         case '/':
106         case '*':
107             std::cout << (char) token.type;
108             break;
```

Arquivo parser.cpp

```
110         case DIV:
111         case MOD:
112             std::cout << token;
113             break;
114
115         case ID:
116             std::cout << std::get<std::string>(token.value);
117             break;
118
119         case NUM:
120             std::cout << std::get<int>(token.value);
121             break;
122
123         default:
124             std::cout << "token desconhecido = " << token << '\n';
125         }
126     }
127 }
```

Módulo `error`

- Este módulo é responsável pelo tratamento de erros
- A abordagem utilizada é simplificada: é impressa a mensagem indicada e o programa é encerrado por meio da função `exit()`

```
1 #include <iostream>
2 #include "error.h"
3
4 void erro(const std::string& message)
5 {
6     std::cerr << message << '\n';
7     exit(-1);
8 }
```

Referências

1. **AHO**, Alfred V, **SETHI**, Ravi, **ULLMAN**, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*, LTC Editora, 1995.
2. GNU.org. GNU Bison, acesso em 23/05/2022.
3. Wikipédia. Flex (lexical analyser generator), acesso em 23/05/2022.