expressões matemáticas em C++

@o-santi

1 o parser

tempos atrás fiz um post sobre a implementação de uma calculadora de expressões matemáticas e terminei dizendo que voltaria com uma implementação. de fato eu esqueci disso durante muito tempo. mas eu lembrei hoje!

e eu voltei com não apenas uma, mas DUAS implementações distintas. veremos primeiro a solução mais bruta e complexa, que foi "descoberta" (ou desenvolvida) primeiro nos primórdios da computação (por volta de 1960~1970), e depois veremos uma solução muito mais simples e elegante.

o que irei apresentar será um *Recursive Descent Parser* utilizando Flex e C++. o programa Flex não é realmente necessário, mas ele oferece uma sintaxe fácil e concisa para implementar um **lexer**, que transformará o input do programa em *tokens* e facilitará as coisas na hora de executar o parsing. utilizamos C++ pura e simplesmente para poder utilizar o tipo `string` sem tem que mexer com ponteiros. não utilizaremos nenhuma classe (yuck) nem nada do gênero.

a ideia principal desse tipo de parser é transformar cada símbolo não-terminal em uma função que chama (executa) os símbolos à direita da regra, e transformar os símbolos terminais em funções que comparam o input atual e dizem se é ou não igual ao que era esperado.

entretanto, não é suficiente simplesmente utilizar essa forma de parsing para calcular, já que iremos apenas gerar uma árvore sintática. para de fato calcular o valor da expressão, iremos também implementar um interpretador de pilha, que irá utilizar a ordem que os elementos são colocados na pilha para executar as operações.

felizmente, podemos utilizar o nosso parser para inserir os elementos na ordem correta. de fato, a função do nosso parser neste programa é de transformar os elementos na notação comum ou padrão, que nós humanos gostamos e entendemos, para a notação que o interpretador de pilha gosta e entende, chamada de notação polonesa reversa. o nome parece complicado, mas é uma maneira simples de representar sem ambiguidade a ordem das operações **sem precisar utilizar parênteses**.

2 sobre a gramática

infelizmente esse tipo de parser é um pouco fresco para com as gramáticas. primeiramente, a gramática que ele aceita não pode ser ambígua (ou seja, mais de uma possível derivação para a ex-

pressão). felizmente, vimos no post inicial que podemos transformar a gramática simples (e ambígua) de expressões matemáticas em uma gramática não ambígua adicionando novas regras.

entretanto, o parser é ainda mais frágil: a gramática também não pode ser recursiva à esquerda. isso significa que não podemos ter uma regra do seguinte tipo:

$$A \to AB$$

também temos como contornar esse problema, mas é um pouco mais chato. para cada regra da forma:

$$A \to A\alpha$$
$$A \to \beta$$

criamos as seguintes regras:

$$A \to \beta A'$$
$$A' \to \alpha A'$$
$$A' \to \varepsilon$$

onde ε representa a palavra vazia.

dessa forma, podemos transformar a seguinte gramática original não ambígua

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid num$$

em

$$E \to TE'$$
$$E' \to +TE' \mid -TE' \mid \varepsilon$$
$$T \to FT'$$
$$T' \to *FT' \mid \varepsilon$$
$$F \to (E) \mid num$$

que é um pouco mais chato de se implementar (mas ainda fácil)

3 implementação

poderíamos dividir tudo em vários arquivos e simplificar as coisas, mas como não é muito código, colocarei tudo no mesmo arquivo `.lex`.

começamos nosso `expressoes_matematicas.lex` com o boilerplate padrão de arquivos C++

```
%{
#include <iostream>
#include <string>
#include <stack>
// para implementar a pilha

int token;

extern "C" int yylex();
// para que o linker entenda que é uma função externa
// definida na biblioteca do lex

using namespace std;

string lexema;

stack<string> pilha;

// definimos as funções que utilizaremos no código c++
int Token(int);
void push(string);
void push_int(int);
string pop();
int pop_int();

void E();
void E_linha();
void T();
void T_linha();
void F();

enum {INT=256}; // enumeracao dos tokens
// como só temos um token, não faz muito sentido usar enum
// usei mais por conveniência, poderia apenas dar um #define
%}
```

depois da declaração de funções e inclusão das bibliotecas relevantes, precisamos definir os tokens que o flex irá gerar. para isso, dividimos as declarações em duas partes: primeiro declaramos "variáveis" (ou nomes) que guardarão as expressões regulares que utilizaremos:

```
WHITESPACE [ \n\T]
NUM [0-9]+
%%
```

e depois, dizemos o que o lexer deve fazer quando encontrar cada uma das possibilidades:

```
{WHITESPACE} { }
{NUM} { return Token(INT); }

. { return Token(*yytext); }

%%

int Token( int tk ) {
  lexema = yytext;
  return tk;
}
```

as linhas de código acima dizem para o lexer: se você encontrar um espaço, um tab ou uma quebra de linha, simplesmente ignore (não faça nada); se encontrar um ou mais números de 0 a 9 em sequência, leia todos e depois retorne o token `INT` (representado por um número na nossa enumeração); e por fim, se encontrar alguma sequência de caractéres que não se encaixa em nenhum desses dois padrões, retorne o valor deles em ascii (como inteiro). tudo que a função `Token` faz é guardar a string encontrada pelo lexer na variável `lexema` e depois retornar o token (inteiro).

também criaremos uma função simples que retorna o próximo token (já que ao invés de procurar todos os tokens de uma vez, estaremos pegando um por um conforme for necessário):

```
int next_token() {
  return yylex();
}
```

em seguida, criamos também uma variável que procura por um caracter específico no input (a função que representará símbolos terminais):

```
string nome (int ch) {
  if (ch >0 && ch < 256 )
    return string(1, ch); // retorna o caracter em ascii
  else
    return "INT";
  }
```

```
void match( int esperado ) {
  if( token == esperado )
    token = next_token();
  else {
    cout << endl;
    cout << "[ERRO] Esperava: " << nome(esperado)
         << " mas achou: " << nome(token) << endl;
    exit( 1 );
  }
}
```

a função compara os dois, e se não forem iguais levanta um erro e termina o programa. ela será a responsável por terminar o programa caso o input de fato não seja uma expressão válida, e portanto colocamos alguns casos para o debug.

por fim, a parte essencial para computar os valores: push e pop. usamos a função push toda vez que queremos colocar um operador na pilha, e ao colocar o operador precisamos imediatamente realizar a operação (para que a prioridade das operações sejam respeitadas). ao colocar um número na pilha, não precisamos fazer nada então podemos usar outra função (push_{int}) para tal. as funções pop e pop_{int} só retiram o valor mais no topo da pilha e retornam-o como string e inteiro, respectivamente.

```
void push(string value) {
  int op1, op2;
  if (value == "+"){
    op2 = pop_int();
    op1 = pop_int();
    push_int(op1 + op2);
  }
  else if (value == "-") {
    op2 = pop_int();
    op1 = pop_int();
    push_int(op1 - op2);
  }
  else if (value == "*"){
    op2 = pop_int();
    op1 = pop_int();
    push_int(op1 * op2);
  }
  else {
    cout << "[ERRO] operador não identificado: " << value << endl;
    exit (123);
```

```
  }
}

void push_int(int value) {
  pilha.push( to_string(value) );
}

string pop() {
  string temp = pilha.top();
  pilha.pop();
  return temp;
}

int pop_int() {
  string temp = pilha.top();
  pilha.pop();
  return stoi(temp);
}
```

podemos finalmente implementar o parser!

```
void E() {
  T();
  E_linha();
}

void T() {
  F();
  T_linha();
}

void E_linha() {
  switch (token) {
  case '+': match('+'); T(); push("+"); E_linha(); break;
  case '-': match('-'); T(); push("-"); E_linha(); break;
  }
}

void T_linha() {
  switch (token) {
  case '*': match('*'); F(); push("*"); T_linha(); break;
```

```
  }
}

void F() {
  switch (token) {
  case INT: {
    string temp = lexema;
    match(INT);
    push_int(stoi(temp));
    break;
  }
  case '(': {
    match('('); E(); match(')'); break;
  }
  default: {
    cout << "[ERRO] Operando esperado, encontrado: " << lexema <<
     ↪  endl;
    exit(3);
    }
  }
}
```

as funções acima simplesmente chamam o lado direito das respectivas regras. utilizamos um switch case toda vez que podemos ter ε como possível saida (nesse caso, a função simplesmente não fará nada).

```
int main() {
  token = next_token(); // inicializamos o valor do token
  E(); // chamamos o ponto inicial do parser
  cout << "O resultado da operação é: " << pilha.top() << endl;
  return 0;
}
```

e pronto! está tudo implementado. conforme o parser processa o input, o valor da pilha irá automaticamente ser calculado, e no final, o único elemento na pilha deverá ser o resultado da computação. para compilar o programa, executamos as seguintes linhas de código:

```
lex expressoes_matematicas.lex
g++ lex.yy.c -o output -lfl
```

e para calcular, devemos utilizar uma stream de caracteres terminada em EOF como input para o programa, como por exemplo:

```
echo "2 * 3 + 5" | ./output
```

que printará na tela "O resultado da operação é: 11". **eureka**!

podemos até testar a prioridade com parênteses

```
echo "(2 - 3) * 5" | ./output
```

que corretamente nos dá o valor -5.

como só temos 3 operações implementadas (soma, subtração e multiplicação), as contas que podem ser feitas são um pouco limitadas. porém, não é dificil de adicionar novas operações, só precisamos implementá-la na gramática e na função push. a mais intuitiva (e óbvia) seria a de divisão, mas isso requeriria trabalharmos com floats, e confesso que fiquei um pouco com preguiça de usar floats assim, já que causaria varios impecílhos.

devemos também lembrar de um fator muito importante: esse parser irá procurar pela primeira expressão válida na string. isso significa que caso tenhamos uma expressão válida seguida de uma expressão inválida, ele irá calcular o valor da expressão válida e terminar, ao invés de levantar um erro. os casos em que ele dará erro serão aqueles cuja primeira expressão é inválida (apenas). por exemplo

```
echo "1 + 1 7" | ./output
```

resulta no valor 2, mas

```
echo "1 * + 1 + 2" | ./output
```

levanta o erro "[ERRO] Operando esperado, encontrado: +".

vale notar que esse comportamento de processar apenas a primeira expressão válida é esperado, e é essencial no parser que implementaremos no próximo post.

o código fonte está disponível aqui (e no repositório do github deste site).