Trabalho prático 2 - Compiladores

Grupo:

- Lucas Caetano Lopes Rodrigues / 2016006670
- Lucas Starling De Paula Salles / 2016006697

Introdução

A tarefa proposta foi a implementação de um analisador sintático de expressoões matemáticas para um conjunto de convenções lexicas e uma gramática definidos. A gramática conta com operadores lógicos, de adição, multiplicação, parêntesis, sinais, constantes, variavéis e funções. O objetivo era gerar um codigo interativo, capaz de receber do usuario uma expressão, ou lista de expressões, para avaliação. O programa deve pedir para o usuario inserir o valor para variaveis, no momento de sua avaliação, caso existam. Por fim o avaliador deve resolver as expressões, em ordem entrada caso sua sintaxe esteja correta, e informar o resultado para o usuario. Uma expressão com sintaxe invalida deve exibir um erro e interromper a execução de programa, assim que o erro for verificado.

O primeiro passo para desenvolver o analisador foi o tratamento da gramática. Esse processo foi feito manualmente, e para tal foi nescessario entendê-la. Percebemos que a definição para os sinais da gramática existiam como uma convenção léxica além de uma regra da gramática, por isso escolhemos tratá-los como símbolos léxicos, ou seja, sinais foram usados como um símbolo terminal. Além disso, alteramos as produções `function_ref := identifier` e `function_ref := identifier ( expr_list )` para que a gramática consiga diferenciar entre identificadores (equivalentes a variáveis no programa) de identificadores de funções matemáticas (`sin`, `cos`, `log`). Tendo isso definido, seguimos para a expansão da gramática seguida pelo cálculo do fechamento.

A Gramática:

```
G' := expr_list
expr_list := expr_list , expr
expr_list := expr
expr := simple_expr
expr := simple_expr RELOP simple_expr
simple_expr := term
simple_expr := sign term
simple_expr := simple_expr ADDOP term
term := factor
term := term MULOP factor
factor := identifier
factor := constant
factor := ( expr )
factor := function_ref
factor := NOT factor
function_ref := func_identifier
function_ref := func_identifier ( expr_list )
```

Closure:

```
CLOSURE:
        I0: {
                G' → .expr_list;
                expr_list → .expr_list , expr; expr_list → .expr;
                expr → .simple_expr;
                expr → .simple_expr RELOP simple_expr;
                simple_expr → .term;
                simple_expr → .sign term;
                simple_expr → .simple_expr ADDOP term;
                term → .factor; term → .term MULOP factor;
                factor → .identifier; factor → .constant; factor → .( expr );
                factor → .function_ref;
                factor → .NOT factor;
                function_ref → .func_identifier;
                function_ref → .func_identifier ( expr_list );
        },
        I1: {
                G' → expr_list.;
                expr_list → expr_list., expr;
        },
        I2: {
                expr_list → expr.;
        },
        I3: {
                expr → simple_expr.;
                expr → simple_expr.RELOP simple_expr;
                simple_expr → simple_expr.ADDOP term;
        },
        I4: {
                simple_expr → term.;
                term → term.MULOP factor ;
        },
        I5: {
                simple_expr → sign.term;
                term → .factor;
                term → .term MULOP factor;
                factor → .identifier;
                factor → .constant;
                factor → .( expr );
                factor → .function_ref;
                factor → .NOT factor;
                function_ref → .func_identifier;
                function_ref → .func_identifier ( expr_list );
        },
```

```
I6: {
        term → factor.;
},
I7: {
        factor → identifier.;
},
I8: {
        factor → constant.;
},
I9: {
        factor → (.expr );
        expr → .simple_expr;
        expr → .simple_expr RELOP simple_expr;
        simple_expr → .term;
        simple_expr → .sign term;
        simple_expr → .simple_expr ADDOP term;
        term → .factor;
        term → .term MULOP factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
I10: {
        factor → function_ref.;
},
I11: {
        factor → NOT.factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
```

```
I12: {
        function_ref → func_identifier.;
        function_ref → func_identifier.( expr_list );
},
I13: {
        expr_list → expr_list ,.expr;
        expr → .simple_expr;
        expr → .simple_expr RELOP simple_expr;
        simple_expr → .term;
        simple_expr → .sign term;
        simple_expr → .simple_expr ADDOP term;
        term → .factor;
        term → .term MULOP factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
I14: {
       expr → simple_expr RELOP.simple_expr;
       simple_expr → .term;
       simple_expr → .sign term;
       simple_expr → .simple_expr ADDOP term;
       term → .factor;
       term → .term MULOP factor;
       factor → .identifier;
       factor → .constant;
       factor → .( expr );
       factor → .function_ref;
       factor → .NOT factor;
       function_ref → .func_identifier;
       function_ref → .func_identifier ( expr_list );
},
```

```
I15: {
        simple_expr → simple_expr ADDOP.term;
        term → .factor;
        term → .term MULOP factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
I16: {
        term → term MULOP.factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
I17: {
        simple_expr → sign term.;
        term → term.MULOP factor;
},
I18: {
        factor → ( expr.);
},
I19: {
        factor → NOT factor.;
},
```

```
I20: {
        function_ref → func_identifier (.expr_list );
        expr_list → .expr_list , expr;
        expr_list → .expr;
        expr → .simple_expr;
        expr → .simple_expr RELOP simple_expr;
        simple_expr → .term;
        simple_expr → .sign term;
        simple_expr → .simple_expr ADDOP term;
        term → .factor; term → .term MULOP factor;
        factor → .identifier;
        factor → .constant;
        factor → .( expr );
        factor → .function_ref;
        factor → .NOT factor;
        function_ref → .func_identifier;
        function_ref → .func_identifier ( expr_list );
},
I21: {
        expr_list → expr_list , expr.;
},
I22: {
        expr → simple_expr RELOP simple_expr.;
        simple_expr → simple_expr.ADDOP term;
},
I23: {
        simple_expr → simple_expr ADDOP term.;
        term → term.MULOP factor;
},
I24: {
        term → term MULOP factor.;
},
I25: {
        factor → ( expr ).;
},
I26: {
       function_ref → func_identifier ( expr_list.);
       expr_list → expr_list., expr;
},
I27: {
        function_ref → func_identifier ( expr_list ).;
}
```

Desenvolvimento

Para desenvolver o projeto com o funcionamento correto, implementamos não apenas do analisador sintático mas também de um léxico. O analisador léxico é responsável pela tokenização da entrada do usuário e o encaminhamento dos tokens gerados para a análise sintática. O projeto foi desenvolvido completamente em C++.

Analisador Léxico

O Analisador léxico é o primeiro componente do compilador. Sua função é receber as expressões inseridas pelo usuário e separá-la em *tokens* de acordo com os símbolos da gramática, para, então, enviar o conjunto de *tokens* ao analisador sintático.

Durante a leitura da expressão inserida pelo usuário (ou conjunto de expressões separadas por vírgula), foram utilizadas expressões regulares para reconhecimento de *tokens* e símbolos terminais da gramática, assim como destacar as operações (ADDOP, RELOP, MULOP) e caracteres especiais (e.g., parêntesis). As seguintes expressões regulares foram utilizadas:

```
std::regex RE_id("[a-zA-Z]([a-zA-Z]|[0-9])*");
std::regex RE_constant("(\\+|-)?([0-9][0-9]*(\\.[0-9]+)?(E(\\+|-)?[0-9][0-9]*)?|[0-9][0-9]*)");
std::regex RE_constant_exp("(\\+|-)?[0-9][0-9]*(\\.[0-9]+)?(E(\\+|-)[0-9][0-9]*)");
std::regex RE_relop("(<=|>=|<>|=|<|>)(?![^()]*\\))(?!\\([^)]*)");
std::regex RE_addop("\\b(\\+|-|(or))(?![^()]*\\))");
std::regex RE_mulop("(\\*|/|div|mod|and)(?![^()]*\\))");
std::regex RE_fun("(sin|cos|log)");
std::regex RE_parenthesis("(\\(*\\))");
std::regex RE_not("NOT");
std::regex RE_sign("(\\+|-|or)");
```

Após a leitura dos caracteres e separação dos *tokens*, o analisador léxico invoca o método `parse()` do analisador sintático, que é explicado na seção seguinte, enviando-lhe o vetor criado.

Analisador Sintático

A análise sintática de gramáticas SLR é feita usando a tecnica de `shift reduce` e pode ser entendida como uma máquina de estados. Sua função de transição representa as possíveis transformações descritas pela gramática. Essa função transforma o estado atual em um próximo estado determinado pelo símbolo mais recente vindo de uma pilha que os armazena em ordem de entrada. A função transição é derivada da gramática e definida por um conjunto de regras de transição e um estado final, representando a aceitação sintática da expressão avaliada.

O algotitmo de análise foi implementado conforme o exemplo provido:

```
var parsing : lógico;
      tipo : terminal;
      valor : value;
      ACTION : array[estado, terminal] of ação;
      GOTO : array[estado, não-terminal] of estado;
      SIN : array[0..MAX] of estado;
      topo : 0..MAX;
      k, S: estado;
      A : não-terminal;
      p : número-de-produção;
begin
      parsing = true
      S := estado inicial; topo := 0; SIN[topo] := S;
      ... preenche tabelas ACTION e GOTO ...
      Scan(tipo, valor);
      while parsing do
            case ACTION[S, tipo]
                  shift k: topo := topo + 1
                        S := k
                        SIN[topo] := k
                        Scan(tipo, valor)
                  reduce p: A := LE da produção p
                        n := tamanho LD de p
                        topo := topo - n
                        S := GOTO[SIN[topo],A]
                        topo := topo + 1
                        SIN[topo] := S
                  accept: parsing := false
                  error: parsing := erro
            end
      end
```

Assim sendo a implementação contou com: uma pilha de estado, uma pilha de símbolos e duas tabelas, `Action` e `Goto`. O *parser* recebe como entrada os *tokens* do analisador léxico. As tabelas foram modeladas como uma classe:

```
class SLR_Table {
    public:
        SLR_Table();
        void printAction();
        void printGoto();
        std::tuple<char, int> get_action(int, int);
        int get_go_to(int, int);
    private:
        std::string action[STATES][TERMINALS];
        int go_to[STATES][NON_TERMINALS];
        std::string composeAction(int, bool );
        void assembleAction();
        void assembleGoto();
}
```

Essas tabelas foram preenchidas usando o conjuto de fechamento de forma a compor de forma correta a tabela SLR(1) para a gramática. As figuras a seguir foram geradas pelo programa para verificação do formato das tabelas:

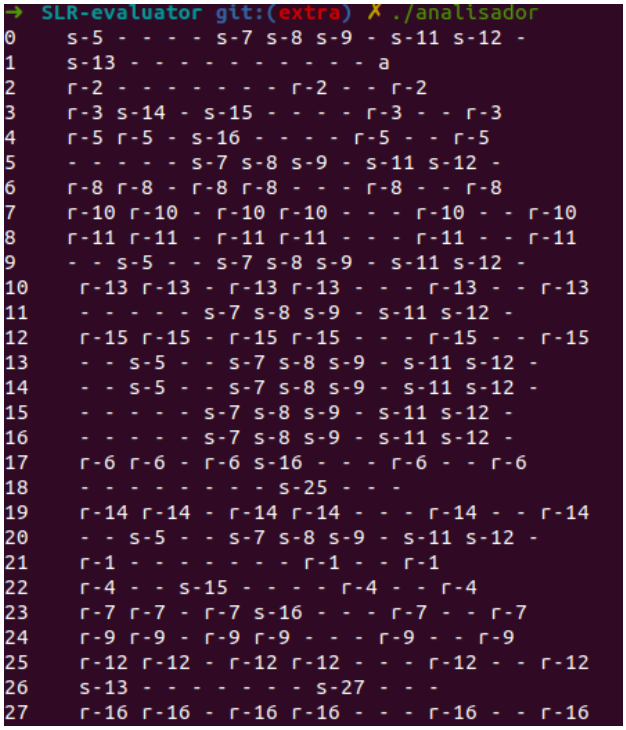
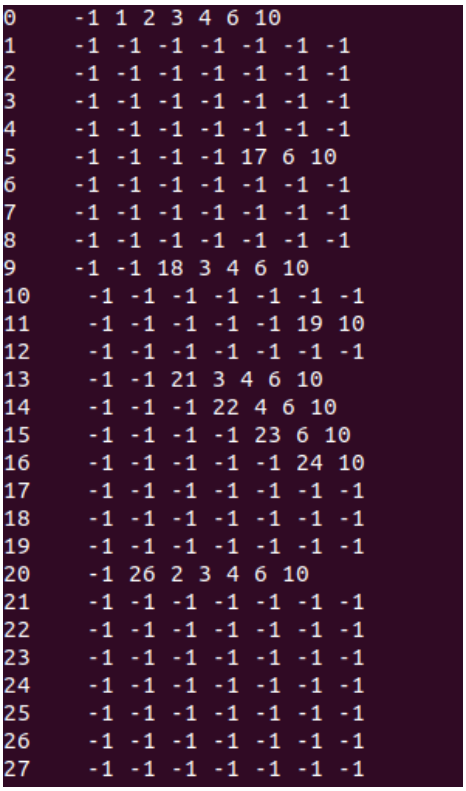
A ordem das colunas (implementada no código) é a seguinte:

- Tabela Action:

`, | RELOP | sign | ADDOP | MULOP | identifier | constant | ( | ) | NOT | func_identifier | $`

- Tabela GOTO:

`E' | expr_list | expr | simple_expr | term | factor | function_ref`

Tabela Action	Tabela GOTO
	

Esse objeto, `SLR_Table`, conta com os métodos responsaveis pela codificação e decodificação das ações esperadas para cada transição de estados válida: `composeAction()` e `get_action()`, respectivamente. Ambas recebem o estado atual e símbolo atual como entrada. A notação usada armazenou *shifts* para um estado `x` como `s-x` e *reduces* usando a regra gramátical `g` como `r-g`. Além disso, o método `get_go_to()` também recebe o estado atual e o símbolo de entrada e retorna o próximo estado da máquina. O valor `-1` foi utilizado como código de erro na Tabela GOTO e o valor `-` foi utilizado como código de erro na Tabela Action. Por fim `printAction()` e `printGoto()` imprimem as respectivas tabelas, seus resultados podem ser observados acima.

O algoritmo de análise sintática foi implementato também com o auxílio de uma classe:

```
class SLR_Parser {
    SLR_Table *table;
    std::vector<int> state_stack;
    std::vector<int> symbol_stack;
    public:
        SLR_Parser();
        bool parse(std::vector<int>);
    private:
        bool reduce(int);
        void parse_action(std::string);
        void clean_parser();
}
```

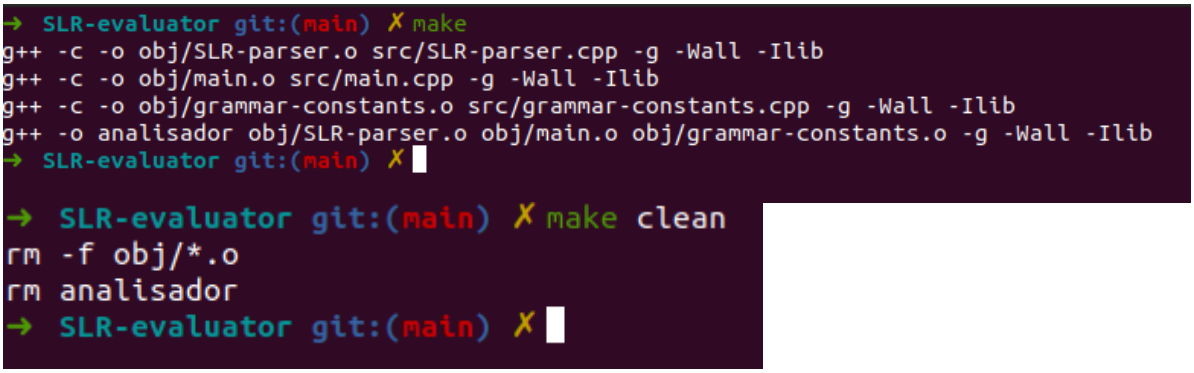
Um objeto `SLR_Parser` conta com uma tabela `SLR_Table`, uma pilha de estados `state_stack` e uma pilha de símbolos percorridos, `symbol_stack`. O método `clean_parser()` esvazia essas pilhas e `reduce()` é um auxiliar que recebe o número da regra gramatical e realiza a redução corretamente. Por fim, `parse()` implementa o algoritmo mencionado, seu retorno indica se a expressão está sintaticamente correta.

```
bool SLR_Parser::parse(std::vector<int> expression){
    char type;
    int cur_state, new_state, symbol;

    int position = 0;
    symbol = expression[position];
    expression.push_back(EOE);

    bool is_valid = true;
    bool parsing = true;

    while(parsing){
        cur_state = this->state_stack.back();
        if (DEBBUGER) {
            std::cout << cur_state << "," << symbol << ": ";
        }
        std::tuple<char, int> action =
            this->table->get_action(cur_state, symbol);

        type = std::get<0>(action); //Type of action to be executed
        new_state = std::get<1>(action); //State or rule

        if (DEBBUGER) {
            std::cout << type << "-" << new_state << std::endl;
        }

        if( type == 's'){ //Shift
            this->state_stack.push_back(new_state);
            this->symbol_stack.push_back(symbol);
            position++;
            symbol = expression[position];

        } else if( type == 'r'){ //Reduce
            this->reduce(new_state);
            new_state = this->table->get_go_to(this->state_stack.back(),
                this->symbol_stack.back());
            if(new_state == GOTO_ERROR){
```

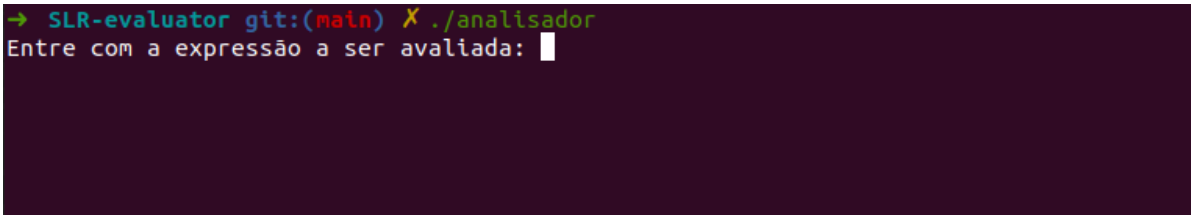
```
                is_valid = false;
                parsing = false;
            }
            this->state_stack.push_back(new_state);

        } else if(type == ACTION_ERROR){

            is_valid = false;
            parsing = false;

        } else if(type == ACTION_ACC){
            parsing = false;
        }else{
            is_valid = false;
            parsing = false;
        }
    }
    this->clean_parser();
    return is_valid;
}
```

Operação

- Compilação: Feita através de linha de comando. No diretório do projeto basta executar `make` no terminal. O executável `analisador` será gerado na raiz do projeto. O comando `make clean` também foi implementado para remover os objetos gerados durante a compilação e o executável.



- Execução: Também por linha de comando, no diretorio executar `./analisador`.



- Uso: O terminal que roda o programa espera uma expressão do usuário, se houverem variáveis seus valores serão pedidos ao usuario quando forem avaliados. Se a expressão estiver correta o resultado, ou resultados, são expostos na tela e o usuario poderá realizar o input novamente. Caso contário o programa exibirá uma mensagem de erro e terminará sua execução.

Testes

As figuras a seguir mostram os testes realizados com o *parser*.

```
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 8+3
Expressão válida.
Resultado: 11
Entre com a expressão a ser avaliada: 10-7
Expressão válida.
Resultado: 3
Entre com a expressão a ser avaliada: 5*5
Expressão válida.
Resultado: 25
Entre com a expressão a ser avaliada: 80div4
Expressão válida.
Resultado: 20
Entre com a expressão a ser avaliada: 44/11
Expressão válida.
Resultado: 4
```

```
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 25>10
Expressão válida.
Resultado: true
Entre com a expressão a ser avaliada: 32<=66
Expressão válida.
Resultado: true
```

```
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 8-2,3*4,25+3
Expressão válida.
Resultado: 6
Expressão válida.
Resultado: 12
Expressão válida.
Resultado: 28
```

```
Entre com a expressão a ser avaliada: sin(45)
Expressão válida.
Resultado: 0.850904
Entre com a expressão a ser avaliada: cos(90)
Expressão válida.
Resultado: -0.448074
Entre com a expressão a ser avaliada: log(64)
Expressão válida.
Resultado: 4.15888
```

```
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: sin(x)
Insira o valor de x: 3.14
Expressão válida.
Resultado: 0.00159255
Entre com a expressão a ser avaliada: x*9
Insira o valor de x: 20
Expressão válida.
Resultado: 180
```

```
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 8!>>
Expressão inválida.
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 5,,!
Expressão válida.

Expressão inválida.
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 5+))(
Expressão inválida.
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: 9<)(
Expressão inválida.
(base) doc@tardis:~/workdir/UFMG/comp1/SLR-evaluator$ ./analisador
Entre com a expressão a ser avaliada: ..ç
Expressão inválida.
```

Listagem dos programas

A estrutura do diretório do projeto é a seguinte:

```
.
├── analisador
├── doc
│   ├── Doc.md
│   └── Trabalho_Pratico_2_Compiladores_1.pdf
├── images
├── lib
│   ├── grammar-constants.hpp
│   └── SLR-parser.hpp
├── makefile
├── obj
│   ├── grammar-constants.o
│   ├── main.o
│   └── SLR-parser.o
├── README.md
├── src
│   ├── analisador-lexico.cpp
│   ├── grammar-constants.cpp
│   ├── main.cpp
│   ├── parser.cpp
│   └── SLR-parser.cpp
└── testes.txt
```