Compilers

Lab Session 1

Advanced calculator grammar

Calc.g4

```
prog:  stat+ EOF ;

stat:  expr NEWLINE           # print
    |  ID '=' expr NEWLINE    # assign
    |  NEWLINE                # blank
    ;

expr:  expr (MUL|DIV) expr    # prod
    |  expr (ADD|SUB) expr    # plus
    |  INT                    # int
    |  ID                     # id
    ;

MUL :  '*' ;
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID  :  [a-zA-Z]+ ;       // match identifiers
INT :  [0-9]+ ;          // match integers
NEWLINE: '\r'? '\n' ;    // return newlines to parser
WS  :  [ \t]+ -> skip ;  // toss out whitespace
```

Advanced calculator grammar

Calc.g4

```
prog:  stat+ EOF ;

stat:  expr NEWLINE          # print
    |  ID '=' expr NEWLINE   # assign
    |  NEWLINE               # blank
    ;

expr:  expr (MUL|DIV) expr   # prod
    |  expr (ADD|SUB) expr   # plus
    |  INT                   # int
    |  ID                    # id
    ;

MUL :  '*' ;
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID  :  [a-zA-Z]+ ;       // match identifiers
INT :  [0-9]+ ;          // match integers
NEWLINE: '\r'? '\n' ;    // return newlines to
parser
WS  :  [ \t]+ -> skip ;  // toss out whitespace
```

main.cpp

```
// create a lexer that consumes the character
// stream, and produces a token stream
CalcLexer lexer(&input);
antlr4::CommonTokenStream tokens(&lexer);

// create a parser that consumes the token
// stream, and parses it
CalcParser parser(&tokens);

// call the parser and get the parse tree
antlr4::tree::ParseTree *tree = parser.prog();

// print the parse tree (debugging purposes)
std::cout << tree->toStringTree(&parser)
          << std::endl;
```

Advanced calculator grammar

Calc.g4

```
prog:  stat+ EOF ;

stat:  expr NEWLINE                # print
    |  ID '=' expr NEWLINE         # assign
    |  NEWLINE                     # blank
    ;

expr:  expr (MUL|DIV) expr         # prod
    |  expr (ADD|SUB) expr         # plus
    |  INT                         # int
    |  ID                          # id
    ;

MUL :  '*' ;
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID  :  [a-zA-Z]+ ;          // match identifiers
INT :  [0-9]+ ;             // match integers
NEWLINE: '\r'? '\n' ;       // return newlines to
parser
WS  :  [ \t]+ -> skip ;     // toss out whitespace
```

main.cpp

```
// create a lexer that consumes the character
// stream, and produces a token stream
CalcLexer lexer(&input);
antlr4::CommonTokenStream tokens(&lexer);

// create a parser that consumes the token
// stream, and parses it
CalcParser parser(&tokens);

// call the parser and get the parse tree
antlr4::tree::ParseTree *tree = parser.prog();

// print the parse tree (debugging purposes)
std::cout << tree->toStringTree(&parser)
          << std::endl;
```

Advanced calculator grammar

Calc.g4

```
prog:  stat+ EOF ;

stat:  expr NEWLINE           # print
    |  ID '=' expr NEWLINE    # assign
    |  NEWLINE                # blank
    ;

expr:  expr (MUL|DIV) expr    # prod
    |  expr (ADD|SUB) expr    # plus
    |  INT                    # int
    |  ID                     # id
    ;

MUL :  '*' ;
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID  :  [a-zA-Z]+ ;        // match identifiers
INT :  [0-9]+ ;           // match integers
NEWLINE: '\r'? '\n' ;     // return newlines to
parser
WS  :  [ \t]+ -> skip ;   // toss out whitespace
```

Rule *labels*
(Not comments!)

Advanced calculator grammar

Calc.g4

```
prog:  stat+ EOF ;

stat:  expr NEWLINE           # print
    |  ID '=' expr NEWLINE    # assign
    |  NEWLINE                # blank
    ;

expr:  expr (MUL|DIV) expr    # prod
    |  expr (ADD|SUB) expr    # plus
    |  INT                    # int
    |  ID                     # id
    ;

MUL :  '*' ;
DIV :  '/' ;
ADD :  '+' ;
SUB :  '-' ;
ID  :  [a-zA-Z]+ ;            // match identifiers
INT :  [0-9]+ ;               // match integers
NEWLINE: '\r'? '\n' ;         // return newlines to
parser
WS  :  [ \t]+ -> skip ;       // toss out whitespace
```

Rule *labels*
(Not comments!)

Those are comments

Advanced calculator grammar

main.cpp

```
//////////////////////////////////////////////////////////////////
// Sample "calculator" using visitors
class Calculator : public CalcBaseVisitor {
public:

  // stat : expr NEWLINE    # print
  antlrcpp::Any visitPrint(CalcParser::PrintContext *ctx) {
    int value = std::any_cast<int>(visit(ctx->expr()));  // evaluate the 'expr' child
    std::cout << value << endl;      // print resulting value
    return 0;                        // return dummy value
  }


  // expr : INT    # int
  antlrcpp::Any visitInt(CalcParser::IntContext *ctx) {
    return std::stoi(ctx->INT()->getText());  // get integer value
  }


 // expr : expr (MUL|DIV) expr    # prod
  antlrcpp::Any visitProd(CalcParser::ProdContext *ctx) {
    int left  = std::any_cast<int>(visit(ctx->expr(0)));  // get value of left subexpr.
    int right = std::any_cast<int>(visit(ctx->expr(1)));  // get value of right subexpr.
    if (ctx->MUL()) return left*right; // compute and return result
    else return left/right;
  }
};
```

Advanced calculator grammar

main.cpp

```
//////////////////////////////////////////////////////////////////
// Sample "calculator" using visitors
class Calculator : public CalcBaseVisitor {
public:

  // stat : expr NEWLINE    # print
  antlrcpp::Any visitPrint(CalcParser::PrintContext *ctx) {
    int value = std::any_cast<int>(visit(ctx->expr()));  // evaluate the 'expr' child
    std::cout << value << endl;      // print resulting value
    return 0;                        // return dummy value
  }


  // expr : INT    # int
  antlrcpp::Any visitInt(CalcParser::IntContext *ctx) {
    return std::stoi(ctx->INT()->getText());  // get integer value
  }


 // expr : expr (MUL|DIV) expr    # prod
  antlrcpp::Any visitProd(CalcParser::ProdContext *ctx) {
    int left  = std::any_cast<int>(visit(ctx->expr(0)));  // get value of left subexpr.
    int right = std::any_cast<int>(visit(ctx->expr(1)));  // get value of right subexpr.
    if (ctx->MUL()) return left*right; // compute and return result
    else return left/right;
  }
};
```

Rule labels generate diferent *visitors* for each subrule

Advanced calculator grammar

main.cpp

```
//////////////////////////////////////////////////////////////////////
// Sample "calculator" using visitors
class Calculator : public CalcBaseVisitor {
public:
  // "memory" for the calculator; stores current value for each variable
  std::map<std::string, int> memory;

  // stat : ID '=' expr NEWLINE    # assign
  antlrcpp::Any visitAssign(CalcParser::AssignContext *ctx) {
    std::string id = ctx->ID()->getText();  // id is left-hand side of '='
    int value = std::any_cast<int>(visit(ctx->expr()));  // compute value of expr. on right
    memory[id] = value;                     // store it in the memory
    return 0;                               // return dummy value
  }

  // expr : ID    # id
  antlrcpp::Any visitId(CalcParser::IdContext *ctx) {
    std::string id = ctx->ID()->getText();
    if (memory.find(id) != memory.end())
       return memory[id];  // retrieve current variable value
    else
       return 0;           // ...or zero if it does not exist
  }

};
```

Advanced calculator grammar

main.cpp

```
//////////////////////////////////////////////////////////////////////
// Sample "calculator" using visitors
class Calculator : public CalcBaseVisitor {
public:
  // "memory" for the calculator; stores current value for each variable
  std::map<std::string, int> memory;

  // stat : ID '=' expr NEWLINE    # assign
  antlrcpp::Any visitAssign(CalcParser::AssignContext *ctx) {
    std::string id = ctx->ID()->getText();  // id is left-hand side of '='
    int value = std::any_cast<int>(visit(ctx->expr()));  // compute value of expr. on right
    memory[id] = value;                     // store it in the memory
    return 0;                               // return dummy value
  }

  // expr : ID    # id
  antlrcpp::Any visitId(CalcParser::IdContext *ctx) {
    std::string id = ctx->ID()->getText();
    if (memory.find(id) != memory.end())
       return memory[id];  // retrieve current variable value
    else
       return 0;           // ...or zero if it does not exist
  }

};
```

We need to store and retrieve values for variables

Advanced calculator grammar

Exercise

- Complete the expression grammar to handle other operators:
  - Unary minus
  - Parenthesis
  - Comparison operators (>, <, ==, !=, >=, <=)
  - Boolean operators (and, or, not)
  - Unary/binary Predefined functions (e.g. abs(a), pow(a,b))
  - N-ary predefined functions (e.g. max(a,b,c,...), min(a,b,c,...), sum(a,b,c,...))
  - Conditional expression (e.g. [a>b ? x+1 : y-2])
- Extend the Calculator **visitor** to handle the missing operators and compute the result in each case. Use **rule labels.**
- Extend your Calc language with additional statements (IF, WHILE, ...)

Summary

Key concepts learnt in this session

- How to write simple antlr4 grammars
- How to create a main program that calls a Lexer and a Parser to get a parse tree.
- How to traverse the parse tree using *Visitors* that return a result after visting each node
- How to use rule labels in antlr4 to get cleaner code for our visitors
- How to use attributes (e.g. `memory` map) to store information that persists and is accesible from any node.