Compiler Project Report
Arian Jalalian - 99243026
Mahdieh Moghisheh 99243068

In this project we aim to design a compiler frontend for a simple language with only two kinds of statements:

Type int a, b, c;
a = (b * c) + a / b

Which are defining a variable of type int and assigning values and expressions to it.
We first define the grammar describing this language :

Goal -> (Statement;)*
Statement -> Define | Assign
Define -> type int Id (, Id)*
Assign -> Id = Expr
Expr -> Term ((+|-) Term)*
Term -> Factor ((*|/) Factor)*
Factor -> Id | Number | (Expr)
Id -> (^[0-9][a-zA-Z0-9])+
Number -> [0-9]+

We seek **Goal** in this grammar, which is a sequence of **Statements**, separated by semicolons. Each statement is either a definition of a variable or an assignment.
In an assignment an id (which basically is a variable name) is assigned and **Expr.**
An Expr is terms that are mathematically operated together. For satisfying the correct presence, first observing non-terminals (or shallow nodes in AST tree) contains addition and subtraction and in more deep heights we see multiplication and division.
Ids (or variable names) contain alphabetic letters and digits but it should be at least of length 1 and should not start with a digit.

At the first layer of frontend, in Lexer, we process streams of characters and make the proper token based on them. In other words, lexical analysis happens.

We first ignore any kind of whitespace character such as ‘ ‘, ‘\n’, ‘\t’ ...

```
while (*BufferPtr && charinfo::isWhitespace(*BufferPtr))
 {
   ++BufferPtr;
 }
```

If the condition does not met, then either it is a null character or a non whitespace

```
  if (!*BufferPtr)
  {
    token.Kind = Token::eoi;
    return;
  }
```

If it is null, so form an end of input (eoi) token.

Then we check if it is a letter or not, in this case we find the whole word to check for keyword or id detection

```
  if (charinfo::isLetter(*BufferPtr))
  {
    const char *end = BufferPtr + 1;

    while (charinfo::isLetter(*end))
      ++end;


    llvm::StringRef Name(BufferPtr, end - BufferPtr);
    Token::TokenKind kind =
        Name == "type" ? Token::KW_type : (Name == "int" ? Token::KW_int :
Token::ident);
    formToken(token, end, kind);
    return;
  }
```

The Name function gets the string of the specified sequence of chars in the buffer. Then based on the text, we assign the proper token kind .

If it is a number we do the same as above and form the token

```
  else if (charinfo::isDigit(*BufferPtr))
  {
    const char *end = BufferPtr + 1;
    while (charinfo::isDigit(*end))
      end = end + 1;
    formToken(token, end, Token::number);
    return;
  }
```

If it is not none of above, then it must be some special characters like + or ;

```
  else
  {
    switch (*BufferPtr)
    {
#define CASE(ch, tok)                       \
  case ch:                                  \
    formToken(token, BufferPtr + 1, tok); \
    break
      CASE('+', Token::plus);
      CASE('-', Token::minus);
      CASE('*', Token::star);
      CASE('/', Token::slash);
      CASE('=', Token::equal);
      CASE('(', Token::Token::l_paren);
      CASE(')', Token::Token::r_paren);
      CASE(';', Token::Token::semi_colon);
      CASE(',', Token::Token::comma);
#undef CASE
    default:
      formToken(token, BufferPtr + 1, Token::unknown);
    }
    return;
  }
```

If it doesn't fit into any category, then it's not a valid character and it is of type unknown token which should be dealt with later on.

So now we get to the parser which processes the generated tokens and make the AST tree, and check if the given code follows the language rules or not .

```
AST *Parser::parseGoal()
{
  llvm::SmallVector<Statement *, 8> Statements;
  Statement *statement;
  while (!Tok.is(Token::eoi) && (statement = parseStatement()))
  {
    Statements.push_back(statement);
  }

  return new Goal(Statements);
}
```

The parseGoal() function iteratively parses statements until it encounters the end of input token (eoi). It collects the parsed statements into a vector and returns an AST object representing the root of the parsed code's AST. If parseStatement() returns a non-null value, meaning a valid statement was parsed, the loop continues.
Inside the while loop, Statements.push_back(statement); adds the parsed statement to the Statements vector. After the loop, return new Goal(Statements); creates a new Goal object, passing the Statements vector as an argument. The Goal object represents the root of the Abstract Syntax Tree (AST) for the parsed code.

In parseStatement :

```
if (Tok.is(Token::KW_type))
  {
    llvm::SmallVector<llvm::StringRef, 8> Vars;
    advance();
    if (expect(Token::KW_int))
    {
      goto _error;
    }

    advance();
    if (expect(Token::ident))
      goto _error;
    Vars.push_back(Tok.getText());

    advance();
    while (Tok.is(Token::comma))
    {
```

```
      advance();
      if (expect(Token::ident))
        goto _error;
      Vars.push_back(Tok.getText());
      advance();
    }

    if (consume(Token::semi_colon))
      goto _error;

    return new typeDecl(Vars);
  }
```

if (Tok.is(Token::KW_type)): This condition checks if the current token (Tok) is a **type** keyword token indicating a type declaration. If it is, the code inside this condition is executed. if (expect(Token::KW_int)) { goto _error; } checks if the next token is the keyword "int". If it is not, it jumps to the _error label. advance(); moves the lexer to the next token. And so on

```
if (Tok.is(Token::ident))
 {

    Expr *Left = new Factor(Factor::Ident, Tok.getText());

    advance();
    if (expect(Token::equal))
      goto _error;

    BinaryOp::Operator Op = BinaryOp::Equal;
    Expr *E;

    advance();
    E = parseExpr();

    if (consume(Token::semi_colon))
      goto _error;

    Left = new BinaryOp(Op, Left, E);
    return Left;
  }
```

If the first condition is not satisfied (Tok.is(Token::KW_type)), the code checks the next condition if (Tok.is(Token::ident)).
This condition handles statements with an identifier as the left side of an assignment.Expr *Left = new Factor(Factor::Ident, Tok.getText()); creates a new expression node Left representing an identifier. advance(); moves the lexer to the next token. if (expect(Token::equal)) { goto _error; } checks if the next token is an equal sign. If it is not, it jumps to the _error label. BinaryOp::Operator Op = BinaryOp::Equal; assigns the operator equal to Op. Expr *E; declares a pointer E to an expression node. advance(); moves the lexer to the next token. E = parseExpr(); calls a function parseExpr() to parse the expression on the right side of the assignment and assigns the result to E.
"if (consume(Token::semi_colon)) { goto _error; }" checks if the next token is a semicolon. If it is not, it jumps to the _error label.
"Left = new BinaryOp(Op, Left, E);" creates a new binary operation node

At the end with the help of AST the IR of the code is generated.