گزارش تشریحی پروژه "اصول طراحی کامپایلر" – فاز اول

استاد درس : جناب آقای علیدوست نیا

اعضای گروه : نرگس دهقان بنادکی – زهرا مترصد – ریحانه ناصری مقدم

***لازم به ذکر است که فصل اول کتاب به صورت کامل اجرا و فصل دوم تا مرحله ۸ انجام شده است اما با توجه به خطاهای حاصل شده و عدم امکان رفع آنها (حتی با کمک گرفتن از دستیاران آموزشی عزیز و استاد گرامی)، امکان خروجی گرفتن از کدها میسر نبود. امید آن داریم که در فاز ۲ خروجی کدها به پیوست ارسال گردد.**

در این فاز از پروژه ، کد های مربوط به Frontend و Backend یک کامپایلر ، صرف نظر از بخش بهینه سازی، پیاده سازی شده اند.

در این پروژه بخش ها و فایل های مختلفی پیاده سازی شده اند که به ترتیب در جدول زیر آمده اند.

Lexer	Parser	Sema	CodeGen	AST.h
Lexer.h	Parser.h	Sema.h	CodeGen.h	**Calc**.cpp
Lexer.cpp	Parser.cpp	Sema.cpp	CodeGen.cpp	

پیاده سازی Lexer

بخش Lexer شامل دو فایل h. و cpp. میباشد.

فایل Lexer.h

در این فایل به بررسی مباحث ابتدایی پیاده سازی Frontend در کامپایلر میپردازیم.

برای پیاده سازی این بخش از چندین کتابخانه استفاده شده است تا با استفاده از آنها به پیاده سازی های اصلی تر پرداخته و اط جزئیات صرف نظر کنیم.

با توجه به توضیحات داده شده و مثالی که در صورت پروژه برای زبان خواسته شده آمده است و نیز مجموعه قواعدی که ساختار زبان مورد نظر را تشکیل میدهند، به شرح زیر خواهد بود:

توضیحات صورت پروژه : "زبان برنامه نویسی فرضی ما دارای دو نوع دستور قابل کامپایل است. یکی تعریف متغیرها و دیگری عملیات ریاضی چهارگانه (جمع، تفریق، ضرب و تقسیم) که نمونه کد آنها در ادامه می آید":

```
type int a, b, c;
c = a + (c- (b*c));
```

نمونه کد موجود در صورت پروژه

```
calc : ("type int" ident ("," ident)*)? expr  ;
expr : ident "=" term (( "+" | "-" ) term)*  ;
term : factor (( "*" | "/" ) term)* ;
factor : ident | number | "("expr")" ;
ident : ([a-zA-Z])+ ;
number : ([0-9]) ;
```

مجموعه قواعد نوشته شده برای زبان خواسته شده

***کلاس Token :**

در این قسمت از کد، میتوان توکن هایی که برای پیاده سازی زبان مورد نیاز هستند را از مجموعه قواعد تعریف شده استخراج کرد. در بخش enum این کلاس ، تمام توکن های استفاده شده در این گرامر جمع آوری شده اند.

در بخش تعریف متغیر های این کلاس ، پوینتری به ابتدای رشته خوانده شده تعریف میکنیم (با کمک کتابخانه StringRef)

```
public:
  enum TokenKind : unsigned short {
  // the definition of the enumeration for the unique token numbers

    eoi,
    // eoi -> stands for end of input
    // (is returned when all the characters of the input are processed)
    unknown,
    // unknown -> is used in case of an error at the lexical level
    ident,
    number,
    comma,
    equation,
    plus,
    minus,
    star,
    slash,
    semicolon,
    l_paren,
    r_paren,
    KW_TypeInt
  };
// two parts below are useful for semantic processing
// ** start of mentioned part
private:
  TokenKind Kind;
  // 1.points to the start of the text of the token
  // 2.StringRef is used
  llvm::StringRef Text;
```

در ادامه این کلاس، ۴ تابع داریم :

۱. **'getText ()': پوینتری که به رشته اشاره میکند را برمیگرداند.**
۲. **'getKind ()': رده ای از توکن را برمیگرداند.**
۳. **'is ()': بررسی میکند که آیا یک توکنی که به عنوان پارامتر به تابع داده شده است، با رده ای از توکن که در نظر گرفته ایم، تطابق دارد یا خیر.**
۴. **'isOneOf ()': بررسی میکند که آیا توکن داده شده به عنوان پارامتر، یکی از چندین دسته توکنی که در نظر گرفته ایم هست یا خیر.**

```
public:
  TokenKind getKind() const { return Kind; }
  llvm::StringRef getText() const {
    return Text;
  }
// ** end of mentioned part

// "is" , "isOneOf" -> useful to test if the token is of a certain kind
  bool is(TokenKind K) const { return Kind == K; }

 // "isOneOf" -> can be used for variable number of arguments
  bool isOneOf(TokenKind K1, TokenKind K2) const {
    return is(K1) || is(K2);
  }
  template <typename... Ts>
  bool isOneOf(TokenKind K1, TokenKind K2, Ts... Ks) const {
    return is(K1) || isOneOf(K2, Ks...);
  }
};
```

***کلاس Lexer :**

در این کلاس دو مولفه BufferStart و BufferPtr داریم که در بخش constructor به گونه ای مقداردهی میشوند که به ترتیب به ابتدای رشته و کاراکتر بعدی که هنوز پردازش نشده است، اشاره دارند.

***تابع next :**

این تابع، توکن بعدی موجود در رشته را برمیگرداند. (در فایل cpp. با جزئیات نوشته خواهد شد.)

***تابع fromToken :**

(در فایل cpp. با جزئیات نوشته خواهد شد.)

```
class Lexer {
  const char *BufferStart;
  const char *BufferPtr;

public:
  Lexer(const llvm::StringRef &Buffer) {
    // "BufferStart" , "BufferPtr" -> pointers to the beginning of the input + next unprocessed character
    // *hint : it is assumed that the buffer ends with a terminating 0 (like a C string)
    BufferStart = Buffer.begin();
    BufferPtr = BufferStart;
  }
  // "next" method -> returns the next token (acts like an iterator , advancing to the next available token)
  void next(Token &token);

private:
  void formToken(Token &Result, const char *TokEnd,
                 Token::TokenKind Kind);
};
#endif
```

فایل Lexer.cpp

در این فایل، پردازش های لازم برای آماده سازی یک رشته خوانده شده از زبان برای استفاده در مراحل بعدی کامپایلر انجام میشوند.
در ابتدا باید توابعی را پیاده سازی کنیم که بتوانند برای خواندن قدم به قدم یک رشته و آنالیز کاراکتر ها، توکن های تعریف شده در زبان و فاصله بین کاراکترها را بررسی کرده و تشخیص دهد.

```
LLVM_READNONE inline bool isWhitespace(char c) {
  return c == ' ' || c == '\t' || c == '\f' || c == '\v' ||
         c == '\r' || c == '\n';
}

LLVM_READNONE inline bool isDigit(char c) {
  return c >= '0' && c <= '9';
}

LLVM_READNONE inline bool isLetter(char c) {
  return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z');
}
```

۱. 'isWhiteSpace': **حالت های مختلفی که میتواند فضای خالی ایجاد کند را بررسی کرده و در صورتی که هر یک از آنها صادق بود، مقدار بازگشتی تابع برابر با true خواهد بود.**
۲. 'isLetter': **یک پوینتر را به عنوان ورودی گرفته و بررسی میکند که آیا مقداری که به آن اشاره شده در دسته حروف قرار دارد یا خیر. (ident در تعریف گرامر)**
۳. 'isDigit': **یک پوینتر را به عنوان ورودی گرفته و بررسی میکند که آیا مقدار اشاره شده یک رقم هست یا خیر. .(number در تعریف گرامر)**

۴. 'next': **کلیت این تابع آن است که BufferPtr را بررسی میکند.**

در واقع چک میکند که توکن در حال بررسی در چه رده ای قرار دارد و متناسب با آن، عملکرد خاص خود را خواهد داشت:

- **isLetter, isDigit <- رویکرد خاصی وجود نخواهد داشت و صرفا اشاره پوینتر به یک خانه جلوتر تغییر پیدا خواهد کرد.**
- **IsWhiteSpace <- در صورت برخورد با همچین توکنی، یک حلقه اجرا خواهد شد و تا زمانی که توکن ما برابر با فصله (از هر نوع) باشد، یک واحد به اشاره گر اضافه خواهد شد.**
- **در صورتی که BufferPtr به هیچ چیز اشاره نداشته باشد، بدین معناست که به انتهای رشته رسیده ایم و در این صورت پیمایش رشته خاتمه پیدا خواهد کرد.**
- **در حالتی که هیچ یک از حالت های بالا رخ ندهد (یعنی پوینتر به چیزی اشاره داشته باشد که در زبان تعریف نشده است) مقدار بازگشتی unknown خواهد بود.**

```
    switch (*BufferPtr) {
    // reason of usage (reduce typing): all tokens have only one character
#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::Token::l_paren);
CASE(')', Token::Token::r_paren);
CASE('=', Token::Token::equation);
CASE(',', Token::Token::comma);
CASE(';', Token::Token::semicolon);
#undef CASE
    // check for unexpected characters
    default:
      formToken(token, BufferPtr + 1, Token::unknown);
    }
    return;
  }
```

بخشی از کد که توکن را بررسی میکند
(تصویر بخش های دیگر تابع به دلیل طولانی بودن طول کد، در اینجا آورده نشده اند)

پیاده سازی Parser

بخش Lexer شامل دو فایل h. و cpp. میباشد.

در رابطه با بخش پارسر، همانطور که در درس داشتیم میتوان بیان کرد که پس از پردازش رشته توسط lexer و شناسایی توکن ها، خروجی به پارسر داده میشود تا بررسی کنیم که آیا رشته داده شده با مجموعه قواعد زبان مورد نظر تطابق دارد یا خیر. در هرجایی از این پردازش که متوجه عدم تطابق شویم، خطا برگردانده خواهد شد.

فایل Parser.h

کلاس Parser:

سه مولفه برای این کلاس در نظر گرفته شده است :

- **Lexer -> خروجی مرحله قبل را به ما میدهد**
- **AST -> همان درختی است که پردازش پارسر روی آن اتفاق می افتد. (Abstract Syntax Tree)**
- **HasError -> به عنوان یک پرچم در نظر گرفته میشود که در صورت تشخیص خطا برابر با ۱ قرار داده میشود تا پس از پایان یک دور پیمایش حلقه، در صورت نیاز فرایند پیمایش متوقف و فرایند خطا (تابع error) اجرا شود.**

```
class Parser {
  // declares some private members
  Lexer &Lex;   // instance of "Lexer" class ("Lex" -> retrieve the next token from the input)
  Token Tok;    // instance of "Token" class ("Tok" -> stores the next token (the look-ahead))
  bool HasError; // "HasError" -> a flag that indicates if an error was detected

  void error() {
    llvm::errs() << "Unexpected: " << Tok.getText() << "\n";
    HasError = true;
  }
```

توابع پیاده سازی شده :

۱. **'advance': این تابع با گرفتن توکن بعدی موجود در رشته ، پردازش را یک پله جلو میبرد. (در واقع انگار که پردازش توکن n به پایان رسیده و حال باید به پردازش N+1 بپردازیم).**

۲. **'expect': این تابع بررسی میکند که آیا توکن در حال بررسی از رده خاصی از توکن که به عنوان پارامتر به تابع داده شده ، هست یا خیر. در صورتی که از جنس توکن خواسته شده نبود، یعنی خطا رخ داده است (کاربرد این تابع را در فایل های بعدی خواهیم دید).**

۳. **'consume': در صورتیکه توکن مورد بررسی از جنس توکن پارامتر باشد، این توکن بازیابی خواهد شد. (انگار پردازش توکن فعلی با موفقیت سپری شده است)**

```
void error() {
  llvm::errs() << "Unexpected: " << Tok.getText() << "\n";
  HasError = true;
}
// "advance()" -> retrieves the next token from lexer (use the "next" method of Lexer class)
void advance() { Lex.next(Tok); }

// "expect()" -> tests if the look-ahead is of the expected kind (if not -> error message)
bool expect(Token::TokenKind Kind) {
  if (!Tok.is(Kind)) {    // the look-ahead is *not* of the expected kind
    error();              // error message
    return true;          // true -> have error in token
  }
  return false;   // the look-ahead is of the expected kind
}

// "consume()" -> retrieves the next token if the look-ahead is of the expected kind (if not -> error message)
bool consume(Token::TokenKind Kind) {
  if (expect(Kind))    // an error message is founded (expect returns 'true' for this condition)
    return true;       // the 'HasError' = true
  advance();
  return false;
}
```

در ادامه مولفه های مربوط به پیمایش تعریف شده اند که به همان ترتیب نگاشت مجموعه قواعد زبان آمده اند.

***تابع Parser:**

تعریف کلی : پیمایش روی درخت را انجام میدهد و در صورت مشاهده HasError = 1 ، خطا برگردانده خواهد شد. (پیاده سازی جزئیات این تابع را در فایل cpp. خواهیم داشت)

```
  AST  *parseCalc();
  Expr *parseExpr();
  Expr *parseTerm();
  Expr *parseFactor();

// code below : initializes all the members and retrieves the first token from the lexer
public:
  Parser(Lexer &Lex) : Lex(Lex), HasError(false) {
    advance();
  }
  AST *parse(); // the main entry point into parsing
  // "hasError" -> to get value of the error flag
  bool hasError() { return HasError; }
};
```

فایل Parse.cpp

پیاده سازی های مربوط به پارسر در این بخش انجام شده اند.

۱. **در پیمایش AST به عنوان ریشه هر رشته ای که ما به عنوان ورودی از کاربر دریافت میکنیم، ابتدا بررسی میکنیم که اصلا رشته ای برای پیمایش وجود دارد یا خیر (با بررسی توکن eoi از طریق تابع expect). در صورتیکه رشته ای وجود نداشت، طبق تعریف موجود در expect خطا برگردانده میشود و در غیر این صورت به مرحله بعد برای پیمایش Calc میرویم.**

```
// ** RECURSIVE DESCENT PARSER ** //
AST *Parser::parse() {
  AST *Res = parseCalc();
  // "parseCalc()" -> implements the corresponding rule
  expect(Token::eoi);
  return Res;
}
```

۲. **با توجه به تعریفی که برای قاعده Calc در نظر گرفته شده است، ابتدا وجود "Type int"و"ident" را در ابتدای رشته بررسی میکنیم.چرا که لازمه اینکه یک عبارت با قاعده Calc تعریف شده باشد آن است که در ابتدای آن رشته ، توکن های ذکر شده حضور داشته باشند.**

در ادامه و با توجه به اینکه میتوان متغیر های دیگری هم در همین عبارت تعریف کرد، سایر متغیر ها را در صورت وجود با استفاده از حلقه بررسی میکنیم.

پس از آنکه هیچ متغیر دیگری رویت نشد، وجود یک Expr را بررسی میکنیم. چرا که طبق دستور زبانی که داریم، پس از تعریف هر متغیر میتوان یک عبارت جهت استفاده از آن را هم در نظر داشت. بنابراین وجود یا عدم وجود عبارت را هم بررسی کرده و در صورت وجود، ادامه فرایند Parse از طریق پیمایش Expr انجام خواهد شد.

توجه داریم که در انتها عبارت حتما چک کنیم که مورد یا توکن دیگری وجود نداشته باشد (if (Vars.empty())) و تنها توکن eoi (که برای مشخص نمودن انتهای عبارت ورودی به کار میرود) تشخیص داده شود. در غیر اینصورت باید خطا برگردانده شود.

(تصویر این بخش از کد به دلیل طولانی بودن طول آن، در اینجا آورده نشده است ☺)

۳. پیمایش قاعده های Expr و Term شبیه به هم هستند؛ چرا که تنها تفاوت آنها در عملگر هایی هست که بررسی میکنند، وگرنه به لحاظ منطقی یا سبک پیاده سازی با یکدیگر تفاوت زیادی ندارند. همچنین به دلیل بررسی عملگر ها با Calc که صرفا تعریف شدن متغیر ها را تشخیص میداد، متفاوت میباشند.

برای این بخش، نحوه بررسی قاعده Expr توضیح داده شده و به دلیل تشابه با Term، از نوشتن جزئیات پیاده سازی Term صرف نظر میشود.

در پیمایش Expr در نظر داریم که با توجه به اینکه تعریف یک عبارت ریاضی برای یک متغیر انجام میشود، پس حتما در ابتدای این قاعده باید وجود ident و "=" تایید شود و در غیر این صورت به خطا میخوریم.

در ادامه و در سمت راست این تساوی (که البته بهتر است آن را انتساب بنامیم) باید عبارت معادل آورده شود که برای این امر ما میتوانیم حالت های مختلف داشته باشیم :

۱- یک متغیر دیگر (مثال : a = c)

۲- یک مقدار عددی (مثال : a = 2)

۳- یک عبارت ریاضی

برای دو حالت اول از term استفاده میکنیم که در صورت ادامه روند پیمایش و بررسی factor ، این دو حالت محقق خواهند شد.

اما برای حالت سوم، باید بحث اولویت عملگر ها را در نظر بگیریم که با توجه به پیاده سازی انجام شده، شاهد رعایت اولویت عملگر ها برای طراحی این گرامر هستیم (لازم به ذکر است که بخش مربوط به اولویت عملگر ها از جزوه درس نظریه زبان ها و ماشین ها و نیز جزوه درس کامپایلر الهام گرفته شده است. لذا با توجه به واضح بودن این بخش، از توضیح چگونگی رعایت قواعد صرف نظر میکنیم).

در اینجا توجه داریم که بررسی صحیح بودن عبارت ریاضی تعریف شده، با کمک توابع تعریف شده در کلاس BinaryOp انجام میشود.

```
// expr : ident "=" term (( "+" | "-" ) term)*  ;
Expr *Parser::parseExpr() {
  // ** THIS PART IS ADDED BY MYSELF (the first part of the expr (ident "="))
  if (Tok.is(Token::ident)) {
    advance();
    if (expect(Token::equation))
      goto _error;
    Vars.push_back(Tok.getText());
    advance();
  }
  // ** END OF ADDITION
  Expr *Left = parseTerm();
  while (Tok.isOneOf(Token::plus, Token::minus)) {
    BinaryOp::Operator Op = Tok.is(Token::plus)
                                ? BinaryOp::Plus
                                : BinaryOp::Minus;
    advance();
    Expr *Right = parseTerm();
    Left = new BinaryOp(Op, Left, Right);
  }
  return Left;
}
```

پیاده سازی بخش تجزیه قاعده Expr

۴. با توجه به اینکه در قاعده Factor ما وجود پایانه های ident و number و نیز پرانتز را مورد بررسی قرار میدهیم، بنابراین پیاده سازی این قاعده با دو قاعده قبلی تفاوت دارد. در این قاعده بررسی عملگر ها به اتمام رسیده و به توکن هایی رسیدیم که باید تشخیص دهیم که از کدام رده از توکن هایی هستند که برای این قاعده میتوان در نظر گرفت (ident, number, l_paren, r_paren)

```
// factor : ident | number | "("expr")" ;
Expr *Parser::parseFactor() {
  Expr *Res = nullptr;
  // switch-case is more suitable than if-else statement here
  switch (Tok.getKind()) {
  case Token::number:
    Res = new Factor(Factor::Number, Tok.getText());
    advance(); break;
  case Token::ident:
    Res = new Factor(Factor::Ident, Tok.getText());
    advance(); break;
  case Token::l_paren:
    advance();
    Res = parseExpr();
    if (!consume(Token::r_paren)) break;
  default:
    if (!Res)
      error();
    while (!Tok.isOneOf(Token::r_paren, Token::star,
                        Token::plus, Token::minus,
                        Token::slash, Token::eoi))
      advance();
  }
  return Res;
}
```

+ نکته ای که باید برای تمام این مراحل مد نظر داشت، آن است که در صورتی که هر یک از شروط قابل قبول واقع شود (به زبان ساده تر پیمایش توکن مورد بررسی انجام شود و خطایی رخ ندهد)، از تابع advance() برای جلوتر بردن پوینتر اشاره گر به توکن مورد بررسی و ادامه روند پیمایش استفاده میکنیم.

پیاده سازی AST

پیاده سازی درخت تجزیه در این بخش اتفاق افتاده است.

فایل AST.h

با توجه به آنکه درخت تجزیه ما طی چند مرحله پیمایش میشود و در هر مرحله ما نیاز داریم تشخیص دهیم که زیر درخت در حال پیمایش را با استفاده از کدام قاعده زبان میتوان تجزیه کرد، بنابراین از یک prototype از قاعده های موجود در زبان میسازیم.

با توجه به اینکه قاعده Calc ریشه پیمایش هر رشته ای به حساب می آید ، به وسیله AST قابل پیمایش است. اما چون برای Expr و Factor چنین قاعده ای صدق نمیکند، بنابراین کلاسی جداگانه برای آنها تعریف میکنیم .همچنین دو کلاس دیگر به نام های BinaryOp و WithDecl نیز تعریف شده است که توضیحات آنها را در ادامه خواهیم داشت.

(AST,Expr,Factor,BianryOp,WithDecl<= مجموعه کلاس های تعریف شده)

```
// ** start of declaration
class AST;
class Expr;
class Factor;
class BinaryOp;
class WithDecl;
// ** end of declaration

// "visit()" method -> same for AST and Expr + does nothing :)
class ASTVisitor {
public:
  virtual void visit(AST &){};
  virtual void visit(Expr &){};
  virtual void visit(Factor &) = 0;
  virtual void visit(BinaryOp &) = 0;
  virtual void visit(WithDecl &) = 0;
};
```

کلاس ASTVisitor:

در این کلاس پیاده سازی پیمایش درخت را خواهیم داشت. برای هر یک از نمونه کلاس هایی که در بالا تعریف شد ، تابع visit را در نظر میگیریم تا امکان پیمایش فراهم شود.

کلاس AST و Expr: این دو کلاس خیلی ساده پیاده سازی شده اند و محتوای خاصی برای توضیح ندارند.

```
// AST -> the root of hierarchy
class AST {
public:
  virtual ~AST() {}
  virtual void accept(ASTVisitor &V) = 0;
};
// Expr -> the root of the AST classes related to expressions
class Expr : public AST {
public:
  Expr() {}
};
```

کلاس Factor:

پیاده سازی این کلاس به نسبت دو کلاس قبلی دارای جزئیات بیشتری هست. چرا که با توجه به تعریفی که برای Factor داشتیم، این قاعده ident و number را در رشته زبان مورد بررسی قرار میدهد.بنابراین در پیاده سازی آن باید توجه داشته باشیم که تنها تشخیص حروف و اعداد امکان پذیر هست و در صورت تشخیص ، در صورت نیاز نوع توکن و یا مقدار آن بازگردانده شود (توابع getKind و getVal)

```
// Factor -> stores the number or the name of a variable
class Factor : public Expr {
public:
  enum ValueKind { Ident, Number };

private:
  ValueKind Kind;
  // tells us which of both cases the instances represent (ident OR number)
  llvm::StringRef Val;

public:
  Factor(ValueKind Kind, llvm::StringRef Val)
      : Kind(Kind), Val(Val) {}
  ValueKind getKind() { return Kind; }
  llvm::StringRef getVal() { return Val; }
  virtual void accept(ASTVisitor &V) override {
    V.visit(*this);
  }
};
```

کلاس BinaryOp:

این کلاس مولفه های Left,Right,Op را تعریف میکند و از آنها برای نگهداری عملوند راست، چپ و نیز عملگر عبارت استفاده میکند. تشخیص وجود و نگهداری عملوند چپ (= اولین عملوند) با استفاده از تابع getLeft ، عملوند راست (=دومین عملوند) getRight و عملگر عبارت با getOperator انجام میشود.

```
/**
"BinaryOp" ->
    1. holds the data that's needed to evaluate an expression
    2. makes no distinction between multiplicative (Mul, Div) and additive(Plus, Minus) operators
**/
class BinaryOp : public Expr {
public:
  enum Operator { Plus, Minus, Mul, Div };

private:
  Expr *Left;
  Expr *Right;
  Operator Op;

public:
  BinaryOp(Operator Op, Expr *L, Expr *R)
      : Op(Op), Left(L), Right(R) {}
  Expr *getLeft() { return Left; }      // return the first operand     (left side)
  Expr *getRight() { return Right; }    // return the second operand    (right side)
  Operator getOperator() { return Op; } // return the operator
  virtual void accept(ASTVisitor &V) override {
    V.visit(*this);
  }
};
```

*کلاس WithDecl:

این کلاس از VarVector برای نگهداری متغیرها (variables) و عبارت هایی (expressions)که توسط کاربر تعریف شده اند ، استفاده میشود.
این کار با استفاده از دو تابع begin و end (برای پیمایش این بردار (VarVector)) و تابع getExpr (برای نگهداری عبارات) انجام میشود.
با توجه به اینکه در هر یک از سه کلاس Factor, BinaryOp و WithDecl پایانه هایی برای نگهداری داریم (حروف، اعداد، عملگرها) نیاز به تایید هست که آیا مولفه های در نظر گرفته شده به درستی تشخیص داده شده اند یا نه (مثلا برای BinaryOp آیا هر سه مولفه لازم برای تعریف یک عبارت ریاضی وجود دارند یا نه)، پس تابعی تحت عنوان 'accept' را برای این کلاس در نظر میگیریم.

```
// "WithDecl" -> stores the declared variables and the expression
class WithDecl : public AST {
  using VarVector = llvm::SmallVector<llvm::StringRef, 8>;
  VarVector Vars;
  Expr *E;

public:
  WithDecl(llvm::SmallVector<llvm::StringRef, 8> Vars,
           Expr *E)
      : Vars(Vars), E(E) {}
  VarVector::const_iterator begin() { return Vars.begin(); }
  VarVector::const_iterator end() { return Vars.end(); }
  Expr *getExpr() { return E; }
  virtual void accept(ASTVisitor &V) override {
    V.visit(*this);
  }
};
```

پیاده سازی Sema

فایل Sema.h

در توضیح کوتاه بخش Sema تنها میتوان گفت که در این بخش درخت تجزیه را پله به پله جلو میرویم و قوانینی که برای تشخیص معنادار بودن عبارات باید چک شوند را بررسی میکنیم.
مثال : قبل از آنکه بخواهیم از یک متغیر استفاده کنیم، باید بررسی کنیم که آیا آن متغیر در لیست متغیر های تعریف شده (defined variables) قرار دارد یا خیر. در صورتی که وجود نداشت، باید خطا برگردانده شود.

فایل Sema.cpp

در این بخش از کد، قوانینی که برای درست بودن معنای عبارت باید لحاظ شوند، روی رشته ورودی چک میشوند.
مولفه های مورد نیاز :

1. **Scope: یک مجموعه از جنس رشته (StringSet) که برای نگهداری نام متغیر های تعریف شده مورد استفاده قرار میگیرد.**
2. **HasError: یک مقدار بول که به عنوان flag و برای تعیین رخ دادن خطا کاربرد دارد.**

+ لازم به ذکر است که در این بخش ، ۲ نوع خطا قابلیت بررسی دارند :

۱- Not: برای زمانی که یک متغیر به صورت explicit تعریف نشده اما در تعریف یک عبارت یا یک انتساب مورد استفاده قرار گرفته است.
۲- Twice: برای زمانی که یک متغیر واحد دو (یا چند) بار تعریف میشود (نه مقداردهی).

```
// implemented using a visitor
/*base idea : 'LIVEOUT' and 'UEVAR'
  1. store the name of each declared variable in a set
  2. check each name is unique
  3. check the name is in the set later
*/
llvm::StringSet<> Scope;
// -> a set to store the names
bool HasError;
// -> used to indicate that an error occurred

enum ErrorType { Twice, Not };

void error(ErrorType ET, llvm::StringRef V) {
  llvm::errs() << "Variable " << V << " "
               << (ET == Twice ? "already" : "not")
               << " declared\n";
  HasError = true;
}
```

قوانین :

۱. **اگر** ident **در یک قاعده با کمک** Factor **استفاده شده است، باید در لیست متغیرهای تعریف شده حضور داشته باشد.** (visit(BinaryOp &Node))

۲. **در تعریف یک عبارت ریاضی، حتما ۲ عملوند مورد نیاز وجود داشته و در صورتی که از جنس** ident **هستند، قانون ۱ برای آنها صادق باشد.** (visit(WithDecl &Node))

```
// RULE-2 : check that both sides of the operation exist and have been visited
// "AST.h" -> 'BinaryOp' class-> 'accept' + 'getLeft' + 'getRight' method
virtual void visit(BinaryOp &Node) override {
  if (Node.getLeft())              // check the existence of second operand
    Node.getLeft()->accept(*this);
  else
    HasError = true;
  if (Node.getRight())             // check the existence of first operand
    Node.getRight()->accept(*this);
  else
    HasError = true;
};

// "AST.h" -> 'WithDecl' class-> 'accept' + 'begin' + 'end' method
virtual void visit(WithDecl &Node) override {
  for (auto I = Node.begin(), E = Node.end(); I != E;
  /*will be continued until we reach the end of the expression*/
       ++I) {
    if (!Scope.insert(*I).second) // check if the declared variables have been stored
      error(Twice, *I);
  }
  if (Node.getExpr())              // check if the declared expression has been stored
    Node.getExpr()->accept(*this);
  else
    HasError = true;
};
};
```

❋تابع semantic **:**

این تابع در ابتدا بررسی میکند که اصلا درختی برای چک کردن قوانین وجود دارد یا خیر. در صورتی که وجود داشت، قوانین برای آن بررسی میشود(استفاده از توابع پیاده سازی شده برای قوانین و کمک گرفتن از تابع accept **که برای خود درخت نوشته شده است).**

```
/**
'Sema.h' -> 'semantic' method
    logic : only starts the tree walk and returns an error flag
**/
bool Sema::semantic(AST *Tree) {
  if (!Tree) // check if any tree exists (if not -> error)
    return false;
  DeclCheck Check;
  Tree->accept(Check);
  return Check.hasError();
}
```

پیاده سازی CodeGen

فایل CodeGen.h

تعریف خیلی ساده و خلاصه برای این بخش :

درخت تجزیه (AST) باید به IR متناسب با مدل تبدیل شده و در نهایت به یک کد ماشین بهینه شده بدل گردد.

فایل CodeGe.cpp

در اینجا تبدیلات لازم برای کد ماشین را انجام میدهیم.

برای این کار نوع داده های مورد نیاز (data type) را تعریف میکنیم تا در صورت نیاز بتوان از آنها استفاده کرد.

```
/*
each compilation unit is represented in LLVM by the 'Module' class
=> a pointer to the module call in visitor
**/
Module *M;
IRBuilder<> Builder; // -> to represent types in IR
// 'Type's below are used as caches to avoid repeated lookup
// ** start of initialization of caches
Type *VoidTy;
Type *Int32Ty;
Type *Int8PtrTy;
Type *Int8PtrPtrTy;
Constant *Int32Zero;
// ** end of initialization of caches

Value *V; // = the current calculated value (updated through tree traversal)

// "nameMap" -> maps a variable name to the value that's returned by the 'calc_read()' function
StringMap<Value *> nameMap;
```

به طور مثال در صورتی که یک گره مربوط به Factor داشته باشیم، مقادیر عددی مربوطه (در عملوند ها) با نوع داده متناظر خود sync میشوند و سپس با توجه به نوع عملگری که برای آن عبارت ریاضی مشخص کرده ایم، عملیات را انجام میدهیم (این بخش در یک switch-case پیاده سازی شده است).

```
virtual void visit(Factor &Node) override {
  if (Node.getKind() == Factor::Ident) {  // variable name
    V = nameMap[Node.getVal()];
  } else {                                // number
    int intval;
    Node.getVal().getAsInteger(10, intval);
    V = ConstantInt::get(Int32Ty, intval, true);
  }
```

بحث مربوط به متغیر ها و اشاره گر ها نیز به همین شکل در تابع visit(WithDecl &Node) پیاده سازی شده است.

در ادامه تابع run را داریم که یک مبدل به کد ماشین را ایجاد کرده، موارد تولید شده در مراحل قبلی را میگیرد و با استفاده از توابع مرتبط برای تبدیل هر یک، کد ماشین را ایجاد (generate) میکند.

```
for (auto I = Node.begin(), E = Node.end(); I != E;
     ++I) {
  StringRef Var = *I; // create a string with a variable name for each variable

  // Create IR code to call 'calc_read()' function
  Constant *StrText = ConstantDataArray::getString(
      M->getContext(), Var);
  GlobalVariable *Str = new GlobalVariable(
      *M, StrText->getType(),
      /*isConstant=*/true, GlobalValue::PrivateLinkage,
      StrText, Twine(Var).concat(".str"));
  Value *Ptr = Builder.CreateInBoundsGEP(
      Str, {Int32Zero, Int32Zero}, "ptr");
  CallInst *Call = // call 'calc_read()' (ReadFn)
      Builder.CreateCall(ReadFty, ReadFn, {Ptr});
  // the returned value is stored in the 'mapNames' map for later use
  nameMap[Var] = Call;
}
// continue tree traverse with the expression
Node.getExpr()->accept(*this);
```

بخشی از تابع run که تبدیل به کد ماشین در آن اتفاق می افتد.

*تابع compile:

به زبان خیلی ساده، کامپایل در این تابع انجام میشود (طبق همان مراحل توضیحی در فایل CodeGen.h) :)

```
/**
compile() method :
    1. creates the global context and the module
    2. runs the tree traversal
    3. dumps the general IR to the console
**/
void CodeGen::compile(AST *Tree) {
  LLVMContext Ctx;
  Module *M = new Module("calc.expr", Ctx);
  ToIRVisitor ToIR(M);
  ToIR.run(Tree);
  M->print(outs(), nullptr);
}
```

فایل Calc.cpp

این بخش از کد ، تابع main برای اجرای روند کامپایل را دارد که در آن به ترتیب توابع hasError, Pars, semantic, compile استفاده شده اند.

```
Lexer Lex(Input);
Parser Parser(Lex);
AST *Tree = Parser.parse();
if (!Tree || Parser.hasError()) { // check if errors occured
  llvm::errs() << "Syntax errors occured\n";
  return 1; // exit the compiler with a return code
}
Sema Semantic;
if (Semantic.semantic(Tree)) {    // check for semantic error
  llvm::errs() << "Semantic errors occured\n";
  return 1;
}
CodeGen CodeGenerator;
CodeGenerator.compile(Tree);
```

بخش مربوط به استفاده از مواردی که قبل تر پیاده سازی شدند