گزارش پروژه کامپایلر - فاز اول

پارسا جهانتاب- مائده دهقان - امیرحسین منصوری

در این پروژه از کدهای کتاب مرجع چهارم درس الهام گرفته شده است.

برای پروژه گرامر زیر در نظر گرفته شده است و کد ها با استفاده از گرامر زیر بازنویسی شده اند.

GOAL -> Statement;

Statement → VarDecl | Assignment

Assignment → ‘type’ ‘int’ Ident

Expr :Term (("+" | "-") Term)* ;

Term : Factor (("*" | "/") Factor)* ;

Factor : Ident | Number | "(" Expr ")" ;

Ident : ([a-zAZ])+ ;

Number : ([0-9])+ ;

Lexer:

محتوای فایل lexer.h

یک ورودی رشته می‌گیرد و جریانی از توکن‌ها در خروجی به ما می‌دهد. توکن‌های خروجی نوع های نتفاوتی دارند که با یک enum در فایل lexer.h انواع آنها را مشخص کردیم:

```
5     enum TokenKind {
6         ID,
7         NUM,
8         LEFT_PAR,
9         RIGHT_PAR,
10        COMMA,
11        KW_TYPE,
12        KW_INT,
13        PLUS,
14        MINUS,
15        STAR,
16        SLASH,
17        EQUAL,
18        SEMICOLON,
19        EOI,
20        UNKNOWN,
21    };
```

هر توکنی یک kind و یک text دارد. که kind از enum گفته شده از بالا استفاده می‌کنند. Text هر توکن، محتوای متنی توکن را نگه می‌دارد. مثلا '9' یک توکن با kind از نوع NUM و با Text یامحتوای '9' می‌باشد.

```
TokenKind getKind() {
    return kind;
}

llvm::StringRef getText() {
    return text;
}
```

با استفاده از bufferStart و bufferPtr متن ورودی داده شده را پردازش کرده و توکنایز(toknise) می‌کنیم.

```
class Lexer {
    private:
        const char* bufferStart;
        const char* bufferPtr;

    public:
        Lexer(llvm::StringRef &buffer) {
            bufferStart = buffer.begin();
            bufferPtr = bufferStart;
        }

        Token next();
};
```

محتوای فایل lexer.cpp کارکتر به کارکتر متن ورودی را می‌خواند و توکن مناسب را با استفاده از lexer.h تولید می‌کنیم.

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

تشخیص فاصله و فضای خالی

```
bool isNumeric(char c) {
    return c >= '0' && c <= '9';
}

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

تشخیص حرف یا عدد بودن کارکتر

```
Token Lexer::next() {
    while(*bufferPtr && isWhitespace(*bufferPtr)) bufferPtr++;

    if (!*bufferPtr) return Token(EOI, "");

    if (isLetter(*bufferPtr)) {
        const char* end = bufferPtr + 1;
        while(isLetter(*end)) end++;

        llvm::StringRef curTokenText(bufferPtr, end - bufferPtr);

        bufferPtr = end;

        if (curTokenText == "type") return Token(KW_TYPE, "");
        if (curTokenText == "int") return Token(KW_INT, "");
        else return Token(ID, curTokenText);
    } else if (isNumeric(*bufferPtr)) {
        const char* end = bufferPtr + 1;
        while(isNumeric(*end)) end++;

        llvm::StringRef numberLiteral(bufferPtr, end - bufferPtr);

        bufferPtr = end;

        return Token(NUM, numberLiteral);
    }
```

در اینجا محتوای bufferPtr* را بررسی می‌کنیم و با توجه به آن اگر فاصله ای ببینیم آن را اسکیپ و در غیر آن صورت توکن مناسب را به آن انتساب می‌کنیم.

:PARSER

:Parser.h

```
 8     class Parser {
 9       Lexer &Lex;
10       Token Tok;
11       bool HasError;
```

Lex و Tok نمونه هایی از کلاس های بخش قبل هستند. Tokنشانه بعدی (نگاه به جلو) را ذخیره می‌کند، در حالی که Lex برای بازیابی نشانه بعدی از آن استفاده می شود

HasError نشان می دهد که آیا خطایی شناسایی شده است یا خیر.

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

    void advance() { Lex.next(Tok); }
```

تابع error ،ارور را چاپ می‌کند و تابع advance، توکن بعدی را شناسایی می‌کند.

()consume توکن بعدی را بازیابی می کند اگر lookahead از نوع مورد انتظار باشد. اگر یک خطا منتشر شود، پرچم HasError روی true تنظیم می‌شود.

برای هر Non-terminal روشی برای pars بیان شده است:

```
35       AST *parseGoal();
36       Expr *parseStatment();
37       Expr *parseVarDeclr();
38       Expr *parseAssignment();
39       Expr *parseExpr();
40       Expr *parseTerm();
41       Expr *parseFactor();
```

:Parser.cpp

نکته اصلی متد parse() این است که کل ورودی مصرف شده است.

اولین تصمیمی که باید گرفته شود این است که آیا هر گروه اختیاری (non-Terminal)باید تجزیه شود یا خیر.

برای هر non-terminal خروجی‌های expected را بررسی می‌کنیم و متناسب با خروجی در جای مناسب error تولید می‌کنیم.

:AST

نتیجه فرآیند تجزیه یک AST است. AST یک نمایش فشرده دیگر است

از برنامه ورودی این اطلاعات ضروری را جمع آوری می کند. بسیاری از زبان های برنامه نویسی نمادهایی دارند که به عنوان جداکننده مورد نیاز هستند و معنای بیشتری ندارند.به عنوان مثال، در C++، نقطه ویرگول، ;، پایان یک عبارت واحد را نشان می دهد.

در پیاده سازی این بخش از visitor design pattern استفاده کردیم. هرکدام از visit ها& یک تعریف دیفالت دارند که کاری انجام نمی‌دهند اما د رادامه در فایل sema.cpp آن ها را override خواهیم کرد.

AST نود ریشه در درخت ماست. به همین ترتیب هر فرزند از پدر خودش ارث بری می کند.

:Semantic Analysis

تحلیل معنایی مرحله سوم طراحی کامپایلر است که بررسی می کند که آیا اعلان ها و عبارات یک برنامه از نظر معنایی درست هستند یا خیر. از درخت نحو (AST)و جدول نماد برای جمع‌آوری اطلاعات نوع و انجام checking2 استفاده می‌کند. همچنین سایر خطاهای معنایی مانند متغیرهای اعلام نشده، سوء استفاده از شناسه رزرو شده، بررسی برچسب و بررسی کنترل جریان را بررسی می کند. تجزیه و تحلیل معنایی را می توان به معنای ایستا و معنایی پویا تقسیم کرد، بسته به اینکه آنها در زمان کامپایل بررسی شوند یا زمان اجرا. تجزیه و تحلیل معنایی برای اطمینان از سازگاری و معنای کد منبع قبل از تولید کد میانی مهم است.

در این قسمت نیز از Visitor design pattern استفاده کردیم. ایده اصلی این است که نام هر یک از declerd variable ها در یک مجموعه ذخیره می شود. در حالی که مجموعه ای که ایجاد می شود، می توانیم بررسی کنیم که هر نام منحصر به فرد باشد و در مجموعه وجود داشته باشد.

همانند بخش های پیش د راین بخش نیز برای مشخص کردن error از یک flag استفاده می‌کنیم.

در آخر یک تابع Semantic تعریف می کنیم که روی درخت پیمایش انجام می‌دهد و فلگ error را برای تشخیص برمی‌گرداند.