第七讲　实验导引（三）

2021-10

1. 实验任务简述

1.1　回顾：实验框架分五个阶段

我们将 MiniDecaf 项目的实验框架分成如图 1 所示的 5 个阶段：

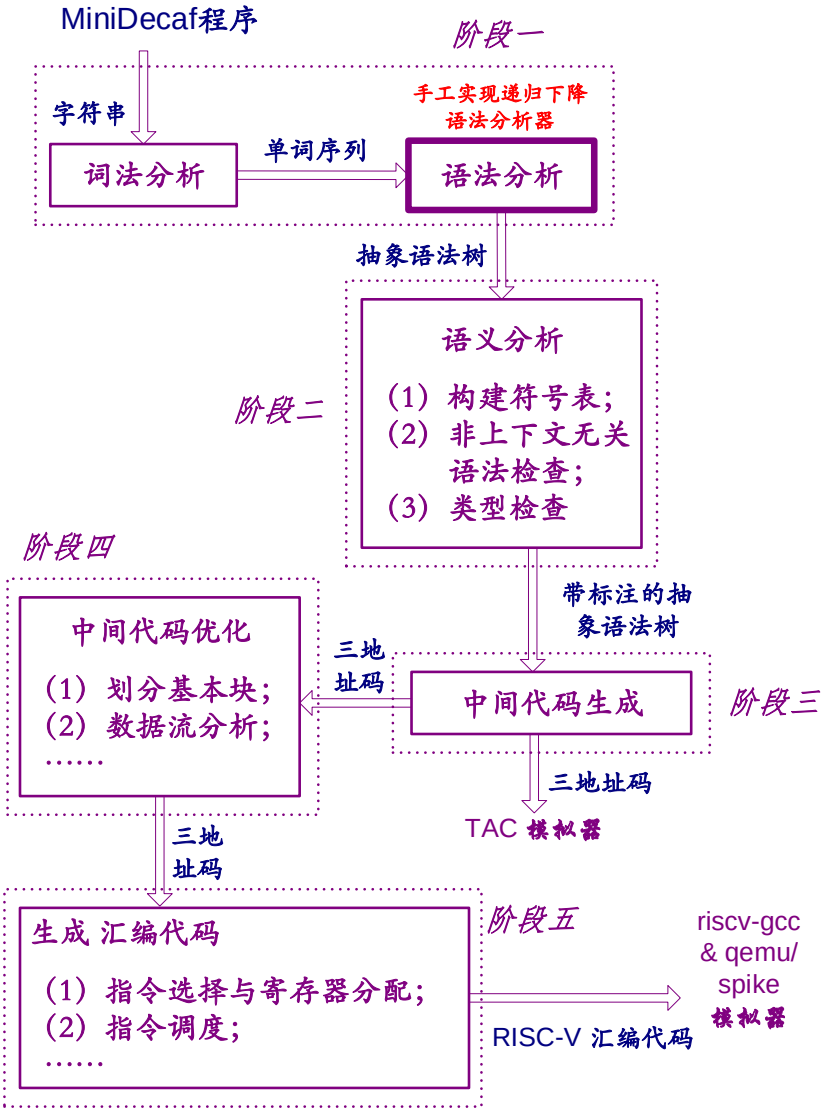

图 1　MiniDecaf 项目实验框架的五个阶段

如图 1 所示，阶段一进行词法和语法分析，并产生一种高级中间表示（实验指定的抽象语法树 AST）；阶段二为语义分析；阶段三生成三地址码（TAC）；阶段四实现一些简单的

数据流分析；阶段五生成 RISC-V 汇编代码。

1.2 手工实现递归下降语法分析器

在阶段一中，我们将 MiniDecaf 源程序转化为抽象语法树。在阶段一词法分析中，我们从左到右扫描 MiniDecaf 源程序，识别出标识符、保留字、整数常量、算符、分界符等单词符号（即终结符），并返回给语法分析器使用。在阶段一语法分析中，我们针对所输入的终结符串进行语法分析，并根据分析结果（具体语法树）建立抽象语法树，在分析过程中对不符合语法规则的 MiniDecaf 源程序进行报错处理。

Parser-stage 的任务是实现一个递归下降的语法分析器，重现阶段一的语法分析及建立抽象语法树功能。该 Stage 的任务独立于 Stage1-5，未来不需要 merge 到 Stage-3 中，但是由于需要支持 Stage-1 和 Stage-2 中的语法规则，所以依赖于同学们在 Stage-2 中已经完成的实验框架。在 Stage-1 和 Stage-2 中，实验框架使用了 bison（C++框架）或 ply（Python 框架）作为语法分析器，解析 MiniDecaf 源程序并生成 AST。Bison 和 ply 可以依据输入的语法规则，自动进行基于 LALR(1) 的自底向上语法分析。同学们在学习自底向上语法分析过程中会发现，若是手工实现一个自底向上的语法分析器工作量很大。为了不明显增加课程实验的工作量，同时使课程实验与授课内容紧密贴合，在 Parser-stage 中，我们结合课堂上学习的 LL(1)分析方法，完成一个手工实现的递归下降语法分析器（如图 1 标红部分所示）。所实现的手工语法分析器，只需要支持 Step1-6 的语法规则，同时产生类似于大家在 Stage-2 中所生成的 AST。

为了进一步降低实验难度和工作量，我们提供了递归下降分析器的基本框架和部分实现，同学们只需要根据实验文档和注释补全代码片段即可。

1.3 需要支持的语法简介

在 Parser-stage 中，我们只需要支持 Step1-6 的语法规则，如图 2 所示。需要注意的是，MiniDecaf 的语法规则中采用了正规表达式，并不是标准的 LL(1)文法。因此，在实现语法分析器时，需要对部分产生式进行转换操作，所需要进行的转换操作将在第 2 节中详细介绍。

```
program : function
function : type Identifier '(' ')' '{' block_item* '}'
type : 'int'
block_item : statement | declaration
statement : 'return' expression ';' | expression? ';' | 'if' '(' expression ')' statement ('else' statement)?
declaration : type Identifier ('=' expression)? ';'
expression : assignment
assignment : conditional | Identifier '=' expression
conditional : logical_or | logical_or '?' expression ':' conditional
logical_or : logical_and | logical_or '||' logical_and
logical_and : equality | logical_and '&&' equality
equality : relational | equality ('=='|'!=') relational
relational : additive | relational ('<'|'>'|'<='|'>=') additive
additive : multiplicative | additive ('+'|'-') multiplicative
multiplicative : unary | multiplicative ('*'|'/'|'%') unary
unary : primary | ('-'|'~'|'!') unary
primary : Integer | '(' expression ')' | Identifier
```

图 2　Parser-stage 需要支持的语法规范

2. 框架介绍

2.1　框架接口

在 Parser-stage 中，我们仍然使用 flex/ply 作为词法分析器，不需要手工实现词法分析器。词法分析器将源程序字符串转换为 token 流，作为语法分析器的输入。在框架中，我们可以使用 next_token 变量查看下一个输入的 token，对应于 LL（1）分析方法中向前查看一个单词的需求。如果检验并移除所查看的 token，可以调用 lookahead 函数消耗掉当前 token，并获取下一个 token，将其赋值给 next_token 变量用于向前查看操作。请注意，lookahead 函数有两个重载的版本。一个版本不带参数，直接读取一个新的 token；另一个版本带有一个 token 类型做参数，表示希望读取一个特定类型的 token，如果类型不符则报错。

2.2　框架结构

Parser-stage 框架的总体结构与第六讲中介绍的递归下降 LL(1)分析程序基本一致，在细节上略有区别。一个递归下降程序通常由若干个 parse 函数构成，每个函数对应于一个非终结符的解析。举例而言，设 LL(1)文法中某一非终结符 A 对应的所有产生式的集合为：

$$A \rightarrow u_1 \mid u_2 \mid ... \mid u_n$$

那么相对于非终结符 A 的分析函数 ParseA() 可以具有如下形式的一般结构：

```
void ParseA()
```

```
{
    switch (lookahead)  {
        case PS(A→u1):
                ……    /*根据 u1 设计的分析过程*/
            break;
        case PS(A→u2):
                ……    /*根据 u2 设计的分析过程*/
            break;
        ...
        case PS(A→ un):
                ……    /*根据 un 设计的分析过程*/
            break;
        default:
            printf("syntax error \n");
            exit(0);
    }
}
```

在 Parser-stage 框架中，所有名为 p_XXX 的函数，为相应非终结符的解析函数，其返回值为抽象语法树（AST）结点（对应于上述的 ParseXXX）。例如，p_unary 函数从当前的 token 流中，解析出一个 unary 表达式，并返回其 AST 结点。

下面是 C++框架里的 p_Unary 函数：

```
// src/frontend/my_parser.cpp
static ast::Expr* p_Unary(){
    /* unary : Minus unary | BitNot unary | Not unary | primary  文法供参考*/
    if (isFirst[SymbolType::Primary][next_token.type]) { //  判断产生式对应的 PS 集
        return p_Primary();     //  使用  unary -> primary  产生式
    } else {
        if (next_token.type == TokenType::MINUS) { //  使用一元运算的产生式
            Token minus = lookahead(TokenType::MINUS);
            return new ast::NegExpr(p_Unary(), minus.loc);
        } else if (next_token. type == TokenType::BNOT) {
            Token bnot = lookahead(TokenType::BNOT);
            return new ast::BitNotExpr(p_Unary(),bnot.loc);
        } else if (next_token.type == TokenType::LNOT) {
            Token lnot = lookahead(TokenType::LNOT);
            return new ast::NotExpr(p_Unary(),lnot.loc);
```

```
            }
        }
        mind::err::issue(next_token.loc, new mind::err::SyntaxError("expect unary
expression get" + TokenName[next_token.type]));
        return NULL;
    }
```

下面是 Python 框架里的 p_unary 函数：

```
    # frontend/parser/my_parser.py
    @first("Minus", "Not", "BitNot", *p_primary_expression.first) #first 集合即为 PS 集合
    def p_unary(self: Parser) -> Expression:
        # unary:Minus unary | BitNot unary | Not unary | primary #文法供参考使用
        lookahead = self.lookahead
        if self.next in p_primary_expression.first:   #使用 unary->primary 产生式
            return p_primary_expression(self)
        elif self.next in ("Minus", "Not", "BitNot"):   # 使用一元运算的产生式
            # MatchToken 并由 token 获取运算类型
            op = UnaryOp.backward_search(self.lookahead())
            #递归下降 parse unary
            oprand = p_unary(self)
            return Unary(op, oprand)
        raise DecafSyntaxError(self.next_token)
```

不难看出，这两个函数的结构和上述的 ParseA 函数是基本一致的。部分实验代码细节在实验文档和程序注释中进行了说明，同学们在完成实验的过程中可以参考。

2.3 拓展巴克斯范式

Parser-stage 的实验框架并不完全是课堂讲授的基于 LL(1)文法的递归下降分析方法，为了简便，有些地方采用了等价的拓展巴克斯范式（EBNF）文法，并通过 while 循环来解析多个连续的、左结合的表达式。我们使用 p_additive 函数介绍框架里所采用的 EBNF 文法及其解析方法。

根据语法规范，additive 对应的语法为：

```
    additive : additive '+' multiplicative
             | additive '/' multiplicative
```

| additive '%' multiplicative

| multiplicative

易于发现，这个产生式是左递归的，不适合基于 LL(1)的递归下降分析器直接处理。我们将其转换为 EBNF 的形式进行程序解析：

additive : multiplicative { '+' multiplicative | '-' multiplicative }

其中，EBNF 中的大括号表示重复零次或任意多次。

注意到产生式的开头总有一个 Multiplicative 非终结符，所以我们递归调用 p_multiplicative 函数解析对应的 Multiplicative 非终结符，如果通过 next_token 检查到后续符号不是 '+' 或 '-'，就可以结束循环并返回 Multiplicative AST 结点。否则，通过 lookahead 消耗运算符('+' 或 '-')，并按照左结合的方法，循环解析更多的 Multiplicative 非终结符。最终完成 Additive 对应 AST 结点的构建。

下面是 C++框架里的 p_Additive 函数：

```
// src/frontend/my_parser.cpp
static ast::Expr* p_Additive() {
    /*additive : additive '+' multiplicative | additive '-' multiplicative | multiplicative
    equivalent EBNF:
    additive : multiplicative { ‘+’ multiplicative | ‘-’ multiplicative } */
    ast::Expr* node = p_Multiplicative();
    while (next_token.type == TokenType::PLUS || next_token.type == TokenType::MINUS) {
        Token operation = lookahead(); //使用 while 循环处理出现任意次的产生式
        ast::Expr* operand2 = p_Multiplicative();
        switch(operation.type) {
        case TokenType::PLUS:
            node = new ast::AddExpr(node,operand2,operation.loc); //构造 Add 结点
            break;
        case TokenType::MINUS:
            node = new ast::SubExpr(node,operand2,operation.loc); //构造 Sub 结点
            break;
        default: break;
        }
    }
```

```
        return node;
    }
```

下面是 Python 框架里的 p_additive 函数：

```
    # frontend/parser/my_parser.py
    @first(*p_multiplicative.first)
    def p_additive(self: Parser) -> Expression:
        """ additive : additive '+' multiplicative | additive '-' multiplicative | multiplicative
        equivalent EBNF:
        additive : multiplicative { ‘+’ multiplicative | ‘-’ multiplicative } """
        lookahead = self.lookahead
        node = p_multiplicative(self)
        #  使用 while 循环处理大括号表示的部分
        while self.next in ("Plus", "Minus"):
            op = BinaryOp.backward_search(lookahead())
            rhs = p_multiplicative(self)
            #  迭代构造 AST 节点
            node = Binary(op, node, rhs)
        return node
```

下面，我们给出所有 Parser-stage 需要支持的语法（必要时转化为相应的 EBNF）及其预测集合，以便同学们在实现中进行参考（注：其中的辅助集合为处理 EBNF 文法时需要循环检查的符号）：

非终结符	EBNF	预测集合 PS
Primary	Integer	Integer
	'(' expression ')'	'('
	Identifier	Identifier
Unary	Primary	'(', Identifier, Integer
	'-' unary	'-'
	'~' unary	'~'

	'!' unary	'!'
multiplicative	unary {'*' unary}	PS(unary) 辅助集合：{ '*' }
	unary {'/' unary}	PS(unary) 辅助集合：{ '/' }
	unary {'%' unary}	PS(unary) 辅助集合：{ '%' }
additive	multiplicative {'+' multiplicative}	PS(multiplicative) 辅助集合：{ '+' }
	multiplicative {'-' multiplicative}	PS(multiplicative) 辅助集合：{ '-' }
relational	relational : additive { '<' additive | '>' additive | '<=' additive | '>=' additive }	PS(unary) 辅助集合：{ '<', '>', '<=', '>=' }
equality	equality : relational {'==' relational | '!=' relational }	PS(relational) 辅助集合：{ '==', '!=' }
logical_and	logical_and : equality { '&&' equality }	PS(equality) 辅助集合：{ '&&' }
logical_or	logical_or : logical_and { '||' logical_and }	PS(logical_and) 辅助集合：{ '||' }
conditional	conditional : logical_or ['?' expression ':' conditional]	PS(logical_or) 辅助集合：{ '?' }
assignment	conditional	PS(conditional)
	Identifier '=' expression	Identifier
expression	assignment	PS(assignment)
declaration	type Identifier ['=' expression] ';'	PS(type) 辅助集合：{ '=' }
statement	'return' expression ';’	'return'

	'if' '(' expression ')' statement ['else' statement]	'if' 辅助集合：{ 'else' }
	[expression] ';'	PS(expression) ∪ { ';' }
block_item	statement	PS(statement)
	declaration	PS(declaration)
type	'int'	'int'
function	type Identifier '(' ')' '{' {block_item} '}'	PS(type) 辅助集合：PS(block_item)
program	function	PS(function)