第七讲

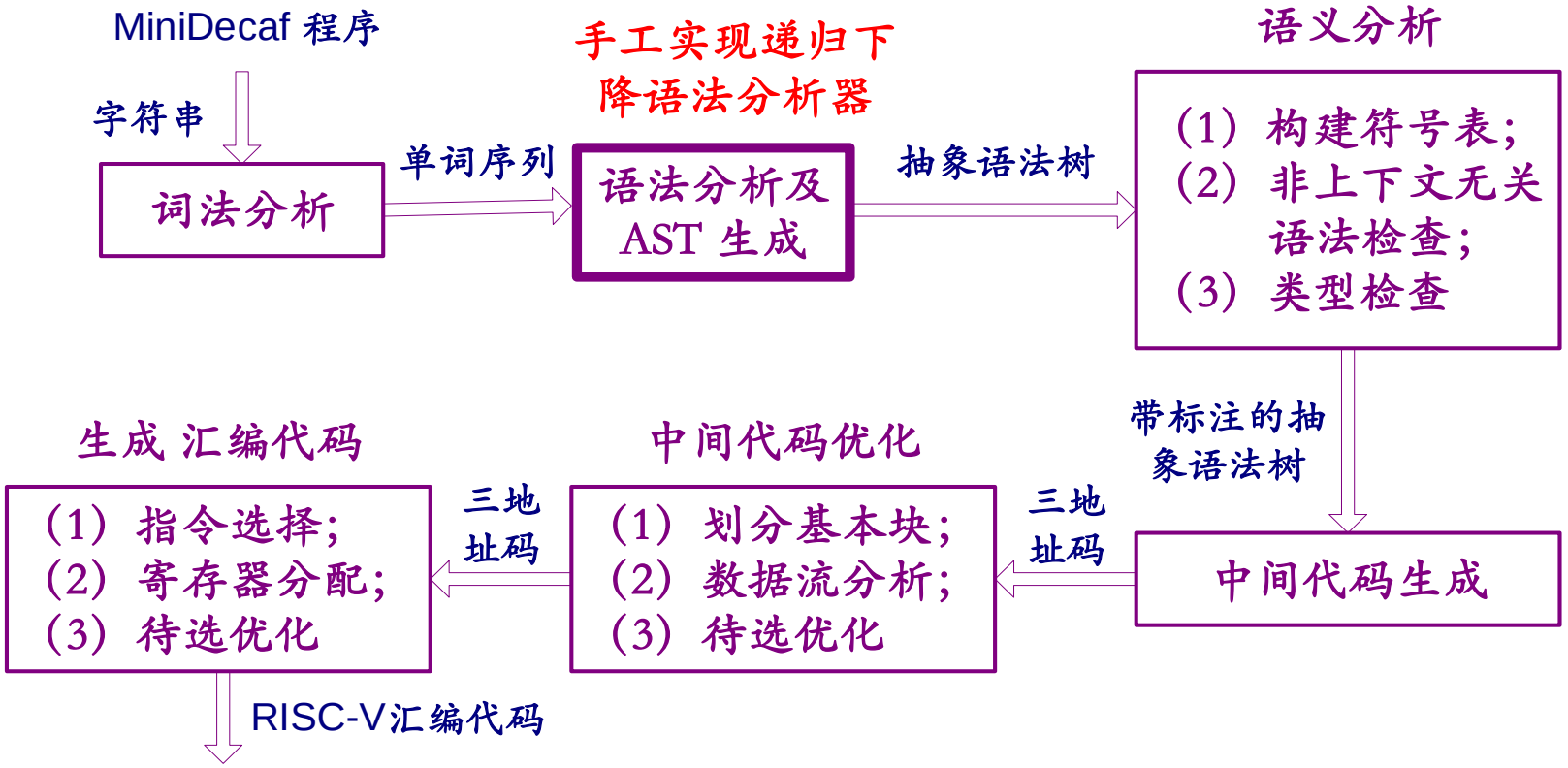
✧ 实验导引（三）

实验导引（三）

- Parser-stage 实验任务简述
- Parser-stage 框架介绍

Parser-stage 实验任务简述

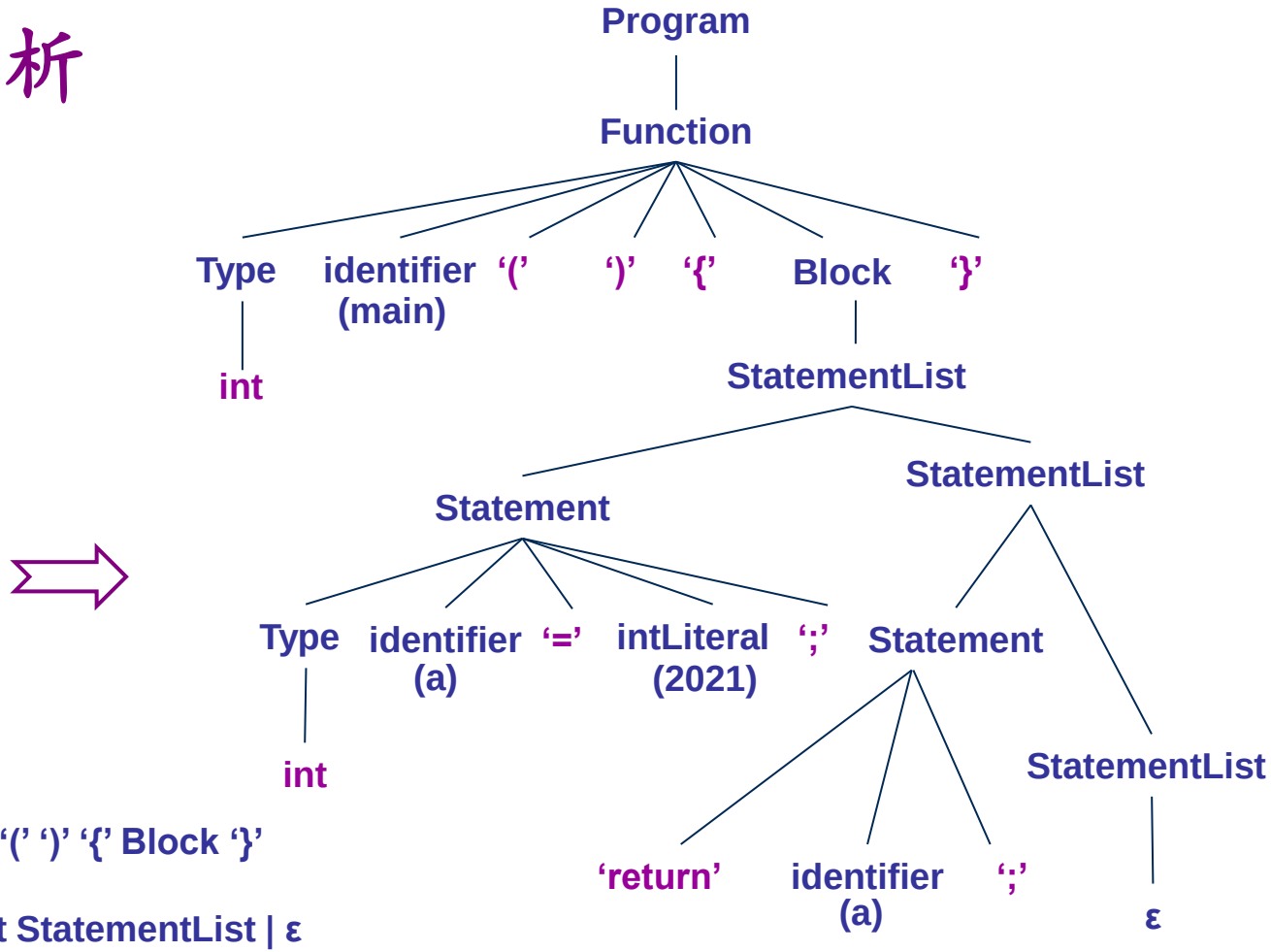
✧ 实验框架的逻辑结构



Parser-stage 实验任务简述

✧ 语法分析

```
int main() {
    int a = 2021;
    return a;
}
```



Program -> Function
Function -> Type identifier ‘(’ ‘)’ ‘{’ Block ‘}’
Block -> StatementList
StatementList -> Statement StatementList | ε
Statement -> Type identifier ‘=’ intLiteral ‘;’ | ‘return’ identifier ‘;’
……

Parser-stage 实验任务简述

✧ 语法分析

- 在 Stage1-2 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 程序并生成 AST
- 在 parser-stage 中，我们将结合课堂上学习的 LL(1)分析方法，完成一个手工实现的递归下降语法分析器
- 为了降低难度和工作量，将提供分析器的基本框架和部分实现，同学们只需要补全代码片段即可
- 所实现的手工语法分析器，只需要支持 Step1-6 的语法。

Parser-stage 实验任务简述

✧ MiniDecaf 语法（Step1-6）

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

Parser-stage 框架介绍

✧ 拓展巴克斯范式（EBNF）

- MiniDecaf 语法在描述二元运算时，存在左递归的情况
- 为了便于处理，需要将其转化为相应的拓展巴克斯范式（EBNF），并使用拓展的递归下降分析方法

✧ EBNF 的元符号

- ‘|’ 表示‘或’，即左部可由多个右部定义
- ‘{}’ 表示花括号内的语法成分可以重复；在不加上下界时可重复0到任意次数
- ‘[]’ 表示方括号内的成分为任选项
- ‘()’ 表示圆括号内的成分优先

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: primary : Integer | '(' expression ')' | Identifier

Equivalent EBNF: primary : Integer | '(' expression ')' | Identifier

非终结符	EBNF	预测集合 PS
primary	Integer	Integer
	'(' expression ')'	'('
	Identifier	Identifier

MiniDecaf语法: unary : primary | ('-'|'~'|'!') unary

Equivalent EBNF: unary : primary | ('-'|'~'|'!') unary

非终结符	EBNF	预测集合 PS
unary	primary	'(', Identifier, Integer
	'-' unary	'-'
	'~' unary	'~'
	'!' unary	'!'

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: multiplicative : unary | multiplicative ('*'|'/'|'%') unary
Equivalent EBNF: multiplicative : unary {'*' unary | '/' unary |'%' unary}

非终结符	EBNF	预测集合 PS
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: { '%' }

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: additive : multiplicative | additive ('+'|'-') multiplicative

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

非终结符	EBNF	预测集合 PS
additive	multiplicative {'+' multiplicative}	PS(multiplicative) 辅助集合: { '+' }
	multiplicative {'-' multiplicative}	PS(multiplicative) 辅助集合: { '-' }

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: relational : additive | relational ('<'|'>'|'<='|'>=') additive

Equivalent EBNF: relational : additive { '<' additive | '>' additive | '<=' additive | '>=' additive }

MiniDecaf语法: equality : relational | equality ('=='|'!=') relational

Equivalent EBNF: equality : relational {'==' relational | '!=' relational }

MiniDecaf语法: logical_and : equality | logical_and '&&' equality

Equivalent EBNF: logical_and : equality { '&&' equality }

非终结符	预测集合 PS
relational	PS(unary) 辅助集合：{ '<', '>', '<=', '>=' }
equality	PS(relational) 辅助集合：{ '==', '!=' }
logical_and	PS(equality) 辅助集合：{ '&&' }

为节省空间，进行了表格内容删减与合并

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: logical_or : logical_and | logical_or '||' logical_and
Equivalent EBNF: logical_or : logical_and { '||' logical_and }

MiniDecaf语法: conditional : logical_or | logical_or '?' expression ':' conditional
Equivalent EBNF: conditional : logical_or ['?' expression ':' conditional]

非终结符	预测集合 PS
logical_or	PS(logical_and) 辅助集合: { '||' }
conditional	PS(logical_or) 辅助集合: { '?' }

为节省空间，进行了表格内容删减与合并

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: assignment : conditional | Identifier '=' expression
Equivalent EBNF: assignment : conditional | Identifier '=' expression

MiniDecaf语法: expression : assignment
Equivalent EBNF: expression : assignment

MiniDecaf语法: declaration : type Identifier ('=' expression)? ';'
Equivalent EBNF: declaration : type Identifier ['=' expression] ';'

非终结符	EBNF	预测集合 PS
assignment	conditional	PS(conditional)
	Identifier '=' expression	Identifier
expression	assignment	PS(assignment)
declaration	type Identifier ['=' expression] ';'	PS(type) 辅助集合: { '=' }

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: statement : 'return' expression ';' | expression? ';' | 'if' '(' expression ')' statement ('else' statement)?

Equivalent EBNF: statement : 'return' expression ';' | [expression] ';' | 'if' '(' expression ')' statement ['else' statement]

MiniDecaf语法: block_item : statement | declaration

Equivalent EBNF: block_item : statement | declaration

MiniDecaf语法: type : 'int'

Equivalent EBNF: type : 'int'

非终结符	EBNF	预测集合 PS
statement	'return' expression ';'	'return'
	[expression] ';'	PS(expression) ∪ { ';' }
	'if' '(' expression ')' statement ['else' statement]	'if' 辅助集合: { 'else' }
block_item	statement	PS(statement)
	declaration	PS(declaration)
type	'int'	'int'

Parser-stage 框架介绍

✧ 计算预测集合 PS

MiniDecaf语法: function : type Identifier '(' ')' '{' block_item* '}'
Equivalent EBNF: function : type Identifier '(' ')' '{' {block_item} '}'

MiniDecaf语法: program : function
Equivalent EBNF: program : function

非终结符	EBNF	预测集合 PS
function	type Identifier '(' ')' '{' {block_item} '}'	PS(type) 辅助集合: PS(block_item)
program	function	PS(function)

Parser-stage 框架介绍

✧ 框架接口

- 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
- 提供了递归下降分析器的函数接口，同学们需要根据注释（以 TODO 标识）补全分析功能
- 请使用 startcode中提供的对应于 Step1-6 语法特性的 AST 结点构建 AST，并与中端、后端完成对接。要求所完成的语法分析器可以通过 Step1-6的原有测例。

Parser-stage 框架介绍

✧ 实现示例（Python）

```
# frontend/parser/my_parser.py

@first("Minus", "Not", "BitNot", *p_primary_expression.first)  # first 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)
```

Parser-stage 框架介绍

✧ 实现示例（C++）

```
# 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]) {
       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;
}
```

✧ EBNF文法处理示例（Python）

```
# 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 } # 大括号表示出现任意次
    """
    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
```

Parser-stage 框架介绍

✧ EBNF文法处理示例（C++）

```
# src/frontend/my_parser.cpp
static ast::Expr* p_Additive() {
  /*additive : additive '+' multiplicative | additive '-' multiplicative | multiplicative
  equivalent EBNF:
  additive : multiplicative { ‘+’ multiplicative | ‘-’ multiplicative } 大括号表示出现任意次*/
  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 节点
      break;
    case TokenType::MINUS:
      node = new ast::SubExpr(node,operand2,operation.loc); // 构造 Sub 节点
      break;
    default:
      break; } }
  return node; }
```

That's all for today.

Thank You