编译器项目 API 规范

1. 文件格式规范

1.1 词法规则文件 (`.lex`)

这个文件定义了词法分析器（Lexer）能识别的所有 Token。

1.1.1 规则格式

每一行定义一个 Token，格式如下：

```
<TOKEN名称>: <正则表达式>
```

- `<TOKEN名称>`: Token 的唯一标识符，使用**大写字母**（例如：`ID`, `INT`, `PLUS`）。
- `<正则表达式>`: 用于匹配该 Token 的 Python 正则表达式。
- **空格**: 冒号 `:` 周围的空格是可选的，建议加上以提高可读性。

1.1.2 注释

以 `#` 开头的行是注释，会被词法分析器忽略。

```
# 这是一条注释
ID: [a-zA-Z_][a-zA-Z0-9_]*  # 这是一条行内注释
```

1.1.3 示例 `lex_rules.lex`

```
# --- 关键字 ---
IF: 'if'
ELSE: 'else'
WHILE: 'while'

# --- 标识符和字面量 ---
ID: [a-zA-Z_][a-zA-Z0-9_]*    # 变量名、函数名等
INT: [0-9]+                   # 整数
STRING: '"'[^"]*'"'           # 字符串 (暂时不支持转义字符)

# --- 运算符 ---
PLUS: '+'
MINUS: '-'
MULTIPLY: '*'
DIVIDE: '/'
ASSIGN: '='

# --- 标点符号 ---
LPAREN: '('
RPAREN: ')'
LBRACE: '{'
RBRACE: '}'
SEMICOLON: ';'
```

1.2 语法规则文件 (`.bnf`)

这个文件定义了编程语言的语法结构（上下文无关文法）。

1.2.1 产生式规则格式

每一行定义一个产生式，格式如下：

```
<非终结符> -> <产生式体>
```

- `<非终结符>`: 语法成分的名称，使用**小写字母**（例如：`expr`, `stmt`, `program`）。
- `->`: 产生式运算符，读作 “推导为”。
- `<产生式体>`
  : 由终结符和 / 或非终结符组成的序列。
  - **终结符**: 要么是 `.lex` 文件中定义的 Token 名称（大写），要么是用单引号 `'` 包裹的字面量字符（如 `'+'`, `';'`）。
  - **非终结符**: 小写字母。
  - **选择项**: 同一个非终结符的多个产生式可以分开写，或者用 `|` 连接。

1.2.2 注释

以 `#` 开头的行是注释，会被语法分析器生成器忽略。

1.2.3 空产生式

空字符串（ε）用 `epsilon` 表示。

```
stmt -> epsilon  # 表示一个可选的语句
```

1.2.4 示例 `yacc_rules.bnf`

```
# --- 程序结构 ---
program -> stmt_list

stmt_list -> stmt stmt_list
stmt_list -> epsilon

# --- 语句 ---
stmt -> expr SEMICOLON
stmt -> IF LPAREN expr RPAREN LBRACE stmt_list RBRACE
stmt -> IF LPAREN expr RPAREN LBRACE stmt_list RBRACE ELSE LBRACE stmt_list
RBRACE
stmt -> WHILE LPAREN expr RPAREN LBRACE stmt_list RBRACE

# --- 表达式（支持优先级）---
expr -> additive_expr

additive_expr -> multiplicative_expr
additive_expr -> additive_expr PLUS multiplicative_expr
additive_expr -> additive_expr MINUS multiplicative_expr
```

```
multiplicative_expr -> primary_expr
multiplicative_expr -> multiplicative_expr MULTIPLY primary_expr
multiplicative_expr -> multiplicative_expr DIVIDE primary_expr

primary_expr -> ID
primary_expr -> INT
primary_expr -> LPAREN expr RPAREN
```

1.3 中间代码文件 (`.ir`)

三地址码指令格式

中间代码采用三地址码（Three-Address Code, TAC）形式，每行一条指令。

指令的通用格式为：`result = op arg1, arg2`

- `result`: 结果变量（通常是临时变量 `t1, t2, ...` 或目标变量）。
- `op`: 操作符。
- `arg1`, `arg2`: 操作数（可以是变量、常量或临时变量）。

支持的指令类型

1. **算术运算**: `t1 = a + b`, `t2 = t1 * 5`
2. **赋值运算**: `x = t2`, `y = 10`
3. **比较运算**: `t3 = a < b` (结果为布尔值，可用 1 表示真，0 表示假)
4. **控制流**:
   - `label L1`: 定义一个标签 `L1`。
   - `goto L1`: 无条件跳转到标签 `L1`。
   - `if t3 goto L1`: 如果 `t3` 为真（非零），则跳转到 `L1`。
   - `if_false t3 goto L1`: 如果 `t3` 为假（零），则跳转到 `L1`。(可选，也可用 `if t3 goto L2; goto L1` 实现)
5. **函数调用 / 返回**

   (如果需要):
   - `param a`: 传递参数 `a`。
   - `call func, n`: 调用函数 `func`，传递 `n` 个参数。
   - `return t1`: 返回值 `t1`。
6. **打印输出**

   (示例): `print a`: 打印变量 `a` 的值。

临时变量命名规则

- 临时变量统一使用 `t` 加数字的形式命名，如 `t1`, `t2`, `t3`, ...。
- 数字从 `1` 开始递增。

示例 `.ir` 文件

对于代码 `x = 5 + 3; if (x > 5) { x = x * 2; }`，其 `.ir` 文件内容可能如下：

```
t1 = 5 + 3
x = t1
t2 = x > 5
if_false t2 goto L1
t3 = x * 2
x = t3
L1:
```

2. 数据结构接口

这些是模块间传递数据时使用的 Python 类。

2.1 Token 结构

表示由词法分析器生成的一个单词（记号）。

2.1.1 Python 类定义

```python
# src/utils/token.py
class Token:
    def __init__(self, type_, value, line, column):
        self.type = type_   # Token的类型 (例如: 'ID', 'PLUS')
        self.value = value  # Token的具体值 (例如: 'x', '+')
        self.line = line    # 在源代码中的行号 (从1开始)
        self.column = column# 在源代码中的列号 (从1开始)
```

- **注意**: `type` 是 Python 的关键字，所以这里用 `type_` 作为参数名，然后赋值给 `self.type`。

2.1.2 示例

```
Token('ID', 'my_variable', 5, 3)
Token('PLUS', '+', 5, 14)
Token('INT', '100', 5, 16)
```

2.2 抽象语法树 (AST) 节点结构

表示语法分析器生成的抽象语法树中的一个节点。

2.2.1 Python 类定义

```
# src/utils/ast_node.py
from typing import List, Optional

class ASTNode:
    def __init__(self, node_type, children=None, value=None, line=None):
        self.node_type = node_type  # 节点类型 (例如: 'BinaryExpression',
'Identifier')
        self.children = children or [] # 子节点列表
        self.value = value          # 节点的值 (可选, 例如标识符的名字)
        self.line = line            # 在源代码中的行号 (可选)

        # 为语法制导翻译预留的字段
        self.ir_code = []           # 存储该节点生成的中间代码
```

- `node_type`: 建议使用 `PascalCase` (如 `BinaryExpression`) 或 `snake_case` (如 `binary_expression`)。
- `children`: 一个包含其他 `ASTNode` 对象的列表。
- `value`: 通常用于叶子节点，如 `Identifier` 节点的变量名或 `IntegerLiteral` 节点的数值字符串。
- `ir_code`: 一个字符串列表，用于在构建 AST 的同时存储生成的中间代码。

2.2.2 示例

对于代码 `x = 5 + 3;`，其 AST 结构可能如下:

```
ASTNode(node_type='Assignment', children=[
    ASTNode(node_type='Identifier', value='x', line=1),
    ASTNode(node_type='BinaryExpression', value='+', children=[
        ASTNode(node_type='IntegerLiteral', value='5', line=1),
        ASTNode(node_type='IntegerLiteral', value='3', line=1)
    ], line=1)
], line=1)
```

2.3 三地址码指令结构 (可选)

不过好像可以直接用List[str]，没必要新定义一个类

为了在内存中更结构化地表示三地址码，可以定义一个辅助类。这对于后续的代码优化或目标代码生成阶段非常有用。

```
# src/utils/ir_instruction.py
from dataclasses import dataclass
from typing import Optional

@dataclass
class IRInstruction:
    op: str
    arg1: Optional[str] = None
    arg2: Optional[str] = None
```

```
    result: Optional[str] = None
    label: Optional[str] = None # 用于 label 和 goto 指令

    def __str__(self):
        """将指令对象转换为字符串形式的三地址码。"""
        if self.label:
            return f"{self.label}:"
        elif self.op == 'goto' or self.op == 'if' or self.op == 'if_false':
            return f"{self.op} {self.arg1} {self.result}" # result 字段用于存储目标
标签
        elif self.op == 'print':
            return f"{self.op} {self.arg1}"
        else: # 通用格式
            return f"{self.result} = {self.op} {self.arg1}, {self.arg2}" if
self.arg2 else f"{self.result} = {self.op} {self.arg1}"

# 示例
# instr1 = IRInstruction(op='+', arg1='5', arg2='3', result='t1')
# print(instr1)  # 输出: t1 = + 5, 3
# instr2 = IRInstruction(label='L1')
# print(instr2)  # 输出: L1:
# instr3 = IRInstruction(op='goto', result='L1')
# print(instr3)  # 输出: goto L1
```

注意：如果使用此类，`ASTNode` 中的 `ir_code` 字段类型应改为 `List[IRInstruction]`。

3. 模块接口

3.1 词法分析器接口

```
# src/lexer_generator/__init__.py
def tokenize(source_code: str, lex_rules_path: str) -> list[Token]:
    """
    将源代码字符串转换为 Token 列表。
    :param source_code: 输入的源代码。
    :param lex_rules_path: .lex 规则文件的路径。
    :return: Token 对象列表。
    """
    pass
```

3.2 语法分析器接口

```
# src/parser_generator/__init__.py
def parse(tokens: list[Token], bnf_rules_path: str) -> ASTNode:
    """
    将 Token 列表解析为 AST。
    :param tokens: 来自词法分析器的 Token 列表。
    :param bnf_rules_path: .bnf 语法文件的路径。
    :return: AST 的根节点。
    """
    pass
```

3.3 中间代码生成器接口

```python
# src/code_generator/__init__.py
from typing import List

def generate_ir(ast_root: ASTNode) -> List[str]:
    """
    从 AST 生成三地址码。
    :param ast_root: AST 的根节点。
    :return: 三地址码指令字符串列表。
    """
    pass

def write_ir_to_file(ir_code: List[str], output_path: str):
    """
    将生成的三地址码写入到 .ir 文件。
    :param ir_code: 三地址码指令字符串列表。
    :param output_path: 输出文件路径。
    """
    pass
```

注意：如果使用了 `IRInstruction` 类，`generate_ir` 的返回类型应为 `List[IRInstruction]`，并且 `write_ir_to_file` 需要相应地处理这个列表（例如，在写入前调用每个指令的 `__str__` 方法）。上面的示例直接使用的字符串列表。