实验任务4.3 语法分析器设计与实现

实验目的

程序要能够查出SysY源代码中可能包含的词法错误和语法错误：

- 词法错误(错误类型代码为A)：出现SysY词法中未定义的字符以及任何不符合SysY词法单元定义的字符。
- 语法错误(错误类型代码为B)：程序在输出错误提示信息时，需要输出具体的错误类型、出错的位置（源程序的行号）以及相关的说明文字。

实验内容

词法分析部分与实验4.2内容相同，不再赘述。下面进行语法分析器设计的详细介绍。

语法树

语法分析的最终结果为语法树，我们需要通过一个数据结构来高效地保存语法树。为此我们设计了语法树的节点类 `Node` 来封装节点的具体信息，并在其中封装了 `NodeType` 来表示节点的类型和用途。 `Node` 的具体内容如下：

- `NodeType` 节点的类型，使用 `NodeType` 枚举表示。
- `start` 和 `end` 用于标记当前节点在源代码中的起始位置和结束位置，以此可以在输出错误信息的时候定位语法元素的位置。

其中，通过调用 `Node::set_range` 方法可以设置当前节点的错误信息对应的范围

调用 `Node::show` 方法可以用于可视化节点结构，用于最终的输出。

我们的语法分析最终传回一个 `Vec<Node>` ，在输出函数当中，我们通过维护偏移量来进行类树形结构的输出行为控制。

```
pub struct Node {
    pub ntype: NodeType,
    pub btype: BasicType,
    pub start: usize,
    pub end: usize,
}
```

`Parser` 基本定义

我们的语法分析器的主要目标是通过输入的Token串，通过递归下降法生成一棵语法树，以便后续代码生成或语义分析使用。因此需要定义核心类 `Parser` 来实现语法分析的过程。简单来说， `Parser` 会通过接收标记流 `tokens` 、源文件路径 `path` 进行初始化，并使用 `comp_unit` 来进行编译单元的解析操作。

```rust
pub struct Parser {
    tokens: Vec<Token>,
    pos: usize,
    path: String,
    code: Vec<char>,
}
```

辅助函数/类

我们在 `Parser` 内定义了若干函数用于进行语法分析的辅助，下面将简单介绍它们的功能。

`expect` 用于验证当前标记是否匹配预期的类型。如果匹配，则当前指针向前移动；否则抛出错误并提示预期的标记类型。这是一个强制性的匹配检查，用于确保当前解析的语法结构正确。

`seek` 函数检查当前标记是否为目标类型。如果是，则当前指针向前移动并返回 `true`。这一函数的主要作用在于非强制性的匹配检查，常用于处理可选标记或可变数量的语法结构。

`get_current_token()` 用于获取当前指针所在的标记，借此解析器可以随时访问当前标记的详细信息，例如标记的类型和位置。

作用域类型

为了方便表示作用域，我们使用 `Scope` 枚举定义了作用域的范围，包括局部作用域 `Local`、全局作用域 `Global` 和参数作用域 `Param`。这一枚举用于区分声明节点的作用域范围。

```rust
pub enum Scope {
    Local,
    Global,
    Param,
}
```

语法分析过程

解析函数部分按照操作符优先级自顶向下实现，包括从基本表达式到复杂语句的解析逻辑。以下是对关键解析函数的详细说明。

表达式解析

我们通过若干函数实现了表达式的解析。

`primary_expr` 是处理基本表达式的函数。它能够解析常量数字、变量标识符、函数调用或数组访问。如果当前标记是一个左括号，说明表达式是一个包含子表达式的括号表达式；否则解析数字、标识符或抛出错误。

`unary_expr` 负责解析一元运算符，例如正号、负号和逻辑非。它还能够处理条件表达式中特殊的一元逻辑运算。这一函数的实现包含了对 `primary_expr` 的递归调用，从而将解析结果组合成语法树的节点。

`mul_expr` 用于进行乘、除、取模操作的处理。它通过循环实现左结合运算符的解析，确保按照从左至右的优先级顺序构建语法树。

`add_expr` 的具体处理流程类似 `mul_expr`，用于进行加减运算的处理。

`rel_expr` 和 `eq_expr` 处理比较运算符、相等运算符。它们基于更低优先级的 `add_expr` 或 `rel_expr`，并构造二元操作节点。

`l_and_expr` 和 `l_or_expr` 分别处理 `&&` 和 `||` 等逻辑操作的处理。

```
fn primary_expr(&mut self, is_cond: bool) -> Node { ... }
fn unary_expr(&mut self, is_cond: bool) -> Node { ... }
fn mul_expr(&mut self, is_cond: bool) -> Node { ... }
fn add_expr(&mut self, is_cond: bool) -> Node { ... }
fn l_and_expr(&mut self) -> Node { ... }
fn l_or_expr(&mut self) -> Node { ... }
```

语句解析

语句部分的解析通过 `stmt(&mut self) -> Node` 函数完成。该函数根据当前标记的类型处理不同的语句结构，包括赋值语句、变量声明语句、条件语句 `if` / `else`、循环语句 `while`、以及 `break`、`continue` 和 `return` 语句。对于每种类型的语句，都会生成相应的 `Node` 节点，捕捉语法结构及其组成部分。

```
fn stmt(&mut self) -> Node {
    let startpos = self.start_count();
    let t = self.get_current_token();
    self.pos += 1;
    match t.kind {
        TokenType::Ident(id) => { ... }
        TokenType::Int | TokenType::Const => { ... }
        TokenType::LeftBrace => { ... }
        TokenType::If => { ... }
        TokenType::While => { ... }
        TokenType::Break => { ... }
        TokenType::Continue => { ... }
        TokenType::Return => { ... }
        _ => {}
    }
}
```

声明解析

声明解析由 `decl_stmt(&mut self, scope: Scope) -> Node` 实现。它能够解析变量和常量声明，并支持数组维度和初始化列表。函数首先识别基本数据类型，然后递归处理声明列表，最终构建一个表示声明语句的节点。省略部分中间步骤后的代码如下。

```
fn decl_stmt(&mut self, scope: Scope) -> Node {
    let startpos = self.start_count();
    let t = self.get_current_token();
    self.pos += 1;
    let btype = match t.kind {
        TokenType::Const => { self.expect(TokenType::Int);
    Some(BasicType::Const) }
```

```
            TokenType::Int => Some(BasicType::Int),
            _ => { None }
        }
        let mut first = true;
        let mut decl_list = vec![];
        while !self.seek(TokenType::Semicolon) {
            if first { first = false; } else { self.expect(TokenType::Comma); }
            if self.seek(TokenType::Assign) { init = Some(self.init_val()); }
            else if btype == BasicType::Const {  ...error }
            else { init = None; }
            let endpos = self.stop_count();
            decl_list.push();
        }
        let endpos = self.stop_count();
        Node::new(NodeType::DeclStmt(decl_list)).set_range(startpos, endpos)
    }
```

代码块解析

代码块解析使用 `block(&mut self) -> Node` 函数实现。它将左大括号后的所有语句解析为一个语句列表，最终返回一个 `Block` 类型的节点。

```
    fn block(&mut self) -> Node {
        self.expect(TokenType::LeftBrace);
        while !self.seek(TokenType::RightBrace) { stmts.push(self.stmt()); }
        Node::new(NodeType::Block(stmts)).set_range(startpos, endpos)
    }
```

`comp_unit(&mut self) -> Node` 函数解析程序的顶层单元，包括全局变量声明和函数定义。需要注意的是，如果当前结构是一个函数定义，函数会处理参数列表并递归解析函数体；否则视为一个全局声明。这个函数会作为解析过程的主函数被调用，所有的语法分析过程都由此函数调用完成。

```
    fn comp_unit(&mut self) -> Node {
        let startpos = self.start_count();
        let pos = self.pos;
        let btype = self.basic_type();
        let name = self.ident();
        if self.seek(TokenType::LeftParen) {
            let mut args = vec![];
            if !self.seek(TokenType::RightParen) {
                args.push(self.func_f_param());
                while self.seek(TokenType::Comma) {
                    args.push(self.func_f_param());
                }
                self.expect(TokenType::RightParen);
            }
            let body = self.block();
            let endpos = self.stop_count();
            return Node::new(NodeType::Func(btype, name, args, Box::new(body)))
            .set_range(startpos, endpos);
```

```
19        }
20        self.pos = pos;
21        self.decl_stmt(Scope::Global)
22    }
```

测试

这里运行的部分样例与4.2中样例相同，具体期望输出不再赘述.

样例1

与4.2样例相同，检测到了词法错误.

```
1   int main()
2   {
3   int i = 1;
4   int j = ~1;
5   return 0;
6   }
```

```
/mnt/e/co/ref/YuXuaan-Compile/task4/task4_3 master !1 ?10 ❯ ./ss test/1.sy
lexer error: Invalid Symbol ~
  --> test/1.sy:4:9
   |
4  | int j = ~1;
   |         ^ Check the symbol is valid
   |
```

样例2

输出:该程序存在两处语法错误，一是二维数组的正确访问方式是`a[5][3]`而非`a[5,3]`，二是第7行最后少了分号。

```
1   int main()
2   {
3       int a[10][12];
4       int i;
5       a[5, 3]=5;
6       if (a[1][2]==0)
7           i=1
8       else
9           i=0;
10  }
```

可以看到，程序确实成功地输出了两个语法错误处的错误信息.

```
lexer error: expect RightBracket
  --> test/2.sy:5:1
   |
5  | , 3]=5;
   | ^ Unexpected token
   |
lexer error: expect RightBracket
  --> test/2.sy:5:1
   |
5  | , 3]=5;
   | ^ Unexpected token
   |
lexer error: expect Semicolon
  --> test/2.sy:5:1
   |
5  | 3]=5;
   | ^ Unexpected token
   |
lexer error: expect "expression"
  --> test/2.sy:5:1
   |
5  | =5;
   | ^ Unexpected token
   |
thread 'main' panicked at src/syntax.rs:345:14:
Wrong expression
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

样例3

期望输出： 成功进行语法分析

```
int inc()
{
 int i;
 i = i+1;
}
```

执行结果： 语法分析成功

```
/mnt/e/compiler/ref/YuXuaan-Compile/task4/task4_3 master !1 ?12 ❯ ./ss test/3.sy
Token{  type:Int  start:0  end:3  lineno:1  }
Token{  type:Ident("inc")  start:4  end:7  lineno:1  }
Token{  type:LeftParen  start:7  end:8  lineno:1  }
Token{  type:RightParen  start:8  end:9  lineno:1  }
Token{  type:LeftBrace  start:10  end:11  lineno:2  }
Token{  type:Int  start:13  end:16  lineno:3  }
Token{  type:Ident("i")  start:17  end:18  lineno:3  }
Token{  type:Semicolon  start:18  end:19  lineno:3  }
Token{  type:Ident("i")  start:21  end:22  lineno:4  }
Token{  type:Assign  start:23  end:24  lineno:4  }
Token{  type:Ident("i")  start:25  end:26  lineno:4  }
Token{  type:Plus  start:26  end:27  lineno:4  }
Token{  type:Number(1)  start:27  end:28  lineno:4  }
Token{  type:Semicolon  start:28  end:29  lineno:4  }
Token{  type:RightBrace  start:30  end:31  lineno:5  }

Func(inc)
  Block
    DeclStmt
      Declare(i)
    Assign(i)
```

样例4

期望输出：无语法错误.

```
1   int main()
2   {
3    int i= 0123;
4    int j= 0x3F;
5   }
```

成功进行了语法分析

```
/mnt/e/compiler/ref/YuXuaan-Compile/task4/task4_3 master !1 ?12 ❯ ./ss test/4.sy
Token{  type:Int  start:0  end:3  lineno:1  }
Token{  type:Ident("main")  start:4  end:8  lineno:1  }
Token{  type:LeftParen  start:8  end:9  lineno:1  }
Token{  type:RightParen  start:9  end:10  lineno:1  }
Token{  type:LeftBrace  start:11  end:12  lineno:2  }
Token{  type:Int  start:14  end:17  lineno:3  }
Token{  type:Ident("i")  start:18  end:19  lineno:3  }
Token{  type:Assign  start:19  end:20  lineno:3  }
Token{  type:Number(123)  start:21  end:25  lineno:3  }
Token{  type:Semicolon  start:25  end:26  lineno:3  }
Token{  type:Int  start:28  end:31  lineno:4  }
Token{  type:Ident("j")  start:32  end:33  lineno:4  }
Token{  type:Assign  start:33  end:34  lineno:4  }
Token{  type:Number(0)  start:37  end:39  lineno:4  }
Token{  type:Semicolon  start:39  end:40  lineno:4  }
Token{  type:RightBrace  start:41  end:42  lineno:5  }

Func(main)
  Block
    DeclStmt
      Declare(i)
    DeclStmt
      Declare(j)
```

假如将 `i` 的初始值改为 `09`， `j` 的初始值改为 `0x3G`，则会发生词法分析报错.

```
/mnt/e/compiler/ref/YuXuaan-Compile/task4/task4_3 master !1 ?13 ❯ ./ss test/5.sy
lexer error: Invalid Octal Number 09
  --> test/5.sy:3:11
   |
3  |  int i= 09;
   |           ^ Check if it is valid
   |
lexer error: Invalid Hexadecimal Number 3G
  --> test/5.sy:4:13
   |
4  |  int j= 0x3G;
   |             ^ Check if it is valid
   |
```