BUAA-2024-Compiler

刘思涵 22371386

前言

文法解读

词法分析

这是编译器项目中的第一个要完成的部分，也是建立整个编译器架构的时机
在此，我按照课程分类，结合实际编译器的设计，也将代码分为词法分析，语法分析，中间优化，目标代码生成四个部分
除此之外，还有公用的错误处理，符号表管理，输入输出和工具类几个部分
我这样设计的目的是为了能够尽可能解耦各个部分，使得每一次作业都能够只进行增量开发，但不要修改原有代码
如果被迫需要修改原有代码，也尽可能不要影响程序其他部分功能

词法组成

根据作业页面定义，本次作业中的词法一共包含以下部分：

单词名称	类别码	单词名称	类别码	单词名称	类别码	单词名称	类别码
Ident	IDENFR	else	ELSETK	void	VOIDTK	;	SEMICN
IntConst	INTCON	!	NOT	*	MULT	,	COMMA
StringConst	STRCON	&&	AND	/	DIV	(	LPARENT
CharConst	CHRCON	||	OR	%	MOD	)	RPARENT
main	MAINTK	for	FORTK	<	LSS	[	LBRACK
const	CONSTTK	getint	GETINTTK	<=	LEQ	]	RBRACK
int	INTTK	getchar	GETCHARTK	>	GRE	{	LBRACE
char	CHARTK	printf	PRINTFTK	>=	GEQ	}	RBRACE
break	BREAKTK	return	RETURNTK	==	EQL		
continue	CONTINUETK	+	PLUS	!=	NEQ		
if	IFTK	-	MINU	=	ASSIGN		

其中可以分为以下几个类别：

1. 数字开头：IntConst
2. 双引号开头：StringConst

3. 单引号开头：CharConst
4. 字母，下划线开头：分为关键字（main，const，...）或者标识符（Ident）
5. 单符号：例如（+，-，*，/）
6. 单或多符号：例如（&&，||，<，<=）

然而字符串是由双引号包裹，其中几乎包含任何字符，只需要注意双引号和反斜杠
字符由单引号包裹，其中包含一个或者两个字符
数字只由数字组成，标识符只由字母，下划线和数字组成，关键字只由字母组成
因此只需要将定义中的分割符作为界限，其余全部划分为单词，再由词法分析程序处理即可

词法分析程序设计

在词法分析程序部分，我主要分为两部分，其一是字符处理部分（IO），其二是词法分析部分（Lexer）

IO

这部分用于从制定的位置读取字符，并且将连续的字符组合成为一系列单词
根据上面词法组成的部分可知，一个单词主要由分割符决定

1. 在分割符和空白符之间的连续字母，数字，下划线可以视为一个单词
2. 分割符自身可以被视为一个单词
3. 双引号之间，单引号之间被视为一个单词
4. 注释符号之间被视为一个单词（但忽略）

为了实现以上功能，并且尽可能将分别单词的任务交给词法分析程序，我在IO部分设置了可以自由规定可忽略分割符和不可忽略分割符的读取函数

接口

建立在解耦各个模块的目的上，我为每一个模块设置了各自的接口，对于模块外部所说，应当只有接口和构造函数可见

ErrorLog

错误处理部分功能的接口，因为还没有做到错误处理部分，目前的任务只需要记录特定报错的信息和位置，所以只设置了两个函数

1. 记录错误信息

```
public void log(CompilerError error);
```

用于向日志提交一个错误，错误内容按照CompilerError模板填写 CompilerError:

```
public class CompilerError {

    protected final int line;

    protected final String message;
```

```
    public CompilerError(int line, String message) {
        this.line = line;
        this.message = message;
    }

    public int getLine() {
        return line;
    }

    public String getMessage() {
        return message;
    }
}
```

目前只需要记录错误信息和位置
2. 打印错误报告

用于在程序结束后将错误内容输出到error.txt

```
public void print(OutputSystem outputSystem);
```

需要指定输出系统，决定输出到哪个位置

IO

1. 输入一行

从当前位置，直到行末，不包括换行符

```
    /**
     * 得到一行，基于评测机要求，以'\n'为截至，返回值不包含'\n'
     * @return 当前行剩余内容，不包括换行符
     */
    public String getLine();
```

2. 输入一个单词

可以自定义分割符，也可以使用默认分割符

```
    /**
     * 得到一个单词，默认以空白符为分割，例如'\n' '\t' '\r' '\v' '\f' ' '
     * @return 以分割符隔开的单词，自动换行，不包括空白符
     */
    public String getWord();

    /**
```

```
     * 得到一个单词，但是以输入的字符为分割，默认不包括空白符
     * 在结束时得到null
     * @param noKeepSeparators 作为分割的符号集合，不作为单词输出
     * @param keepSeparators 作为分割的符号集合，作为单词输出
     * @return 以分割符隔开的单词，自动换行
     */
    public String getWord(char[] noKeepSeparators, char[] keepSeparators);
```

3. 更换默认分割符

初始默认为只有空白符，因为程序中，主要由文法中的符号进行分割，所以可以修改

```
    /**
     * 重新声明默认分割符
     * @param noKeep 跳过的分割符（空白符）
     * @param keep 保留的分割符
     */
    public void setDefault(char[] noKeep, char[] keep);
```

4. 回退一个单词

用于在试探性读取下一个单词时，注意只能使用一次，并且使用之后需要再取下一个词

```
    /**
     * 返回上一个单词，返回null表示前面没有单词，开头使用会导致getWord错误
     * 在下一次getWord之前，不要使用其他方法
     * 下一次getWord会返回当前单词
     * @return 上一个单词
     */
    public String backward();
```

5. 读取一个字符

用于细微调整，例如读取反斜杠之后的转义字符

```
    /**
     * 返回一个字符，包括空白符
     * @return 得到的字符
     */
    public String getChar();
```

6. 得到当前单词行数

用于报错时定位，注意，得到的是当前指向单词的位置（下一个单词的位置）

```
    /**
     * 得到当前行
     * @return 当前读取的行（尚未读取的部分所在行）
     */
    public int getRow();
```

7. 输出系列函数

用于在不同输出方向的公用函数

```
    /**
     * 基本打印函数
     * @param s 打印字符串
     */
    public void print(String s);

    /**
     * 基本换行打印函数
     * @param s 打印字符串
     */
    public void println(String s);

    /**
     * 换行
     */
    public void println();
```

8. Token及Error输出

特别为本次作业需要的两种输出提供功能

```
    /**
     * 错误打印函数
     * @param error 编译错误
     */
    public void error(CompilerError error);

    /**
     * Token打印函数
     * @param token Token
     */
    public void Token(Token token);
```

Lexer

词法解析程序，包括Token模板，Token种类枚举，Lexer方法几部分

1. Token模板

```
public class Token {

    // 类别码
    private final TokenType type;

    // 单词的字符，字符串，数字
    private final String value;

    /**
     * 声明一个标准的Token
     * @param type Token类别码
     * @param value Token的具体内容
     */
    public Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }

    public TokenType getType() {
        return type;
    }

    public String getValue() {
        return value;
    }
}
```

2. TokenType枚举

```
public enum TokenType {
    IDENFR, INTCON, STRCON, CHRCON,
    MAINTK, CONSTTK, INTTK, CHARTK, BREAKTK, CONTINUETK, IFTK, ELSETK,
    NOT, AND, OR,
    FORTK, GETINTTK, GETCHARTK, PRINTFTK, RETURNTK,
    PLUS, MINU,
    VOIDTK,
    MULT, DIV, MOD, LSS, LEQ, GRE, GEQ, EQL, NEQ, ASSIGN, SEMICN, COMMA,
    LPARENT, RPARENT, LBRACK, RBRACK, LBRACE, RBRACE
}
```

3. 获得下一个Token

本次词法分析使用语法分析相互交互的模型
指挥词法分析程序，从文件中读取单词，并解析成为下一个Token（如果之前回退过，会回到当前位置）

```
    /**
     * 取出下一个Token
     * @return 下一个Token，null表示结束
     */
    public Token nextToken();
```

4. 获取上一个Token

上上次输出的Token（上一次输出为当前Token），不会改变指针位置

```
    /**
     * 取出上一个Token
     * @return 上一个Token，null表示不存在
     */
    public Token previousToken();
```

5. 预览下一个Token

不会改变当前的指针位置，得到下一个Token的内容

```
    /**
     * 预览下一个Token，不改变当前指针位置
     * @return 下一个Token，null表示结束
     */
    public Token previewToken();
```

6. 重新取出当前位置Token内容

```
    /**
     * 取出当前Token，不改变当前指针位置
     * @return 当前Token，null表示尚未开始，或者结束
     */
    public Token getToken();
```

其余函数功能

语法分析

语法树

语法分析部分的主要目的在于，将判断词法分析得到的结果是否符合语法的定义顺序
在忽略部分缺少符号的错误之后，将剩下的内容，按照语法整合成语法树
为了更方便的整理语法树，我做了一些修改：

1. 所有的Token和非终结符都属于元素，因此可以将元素作为单位组织树

2. 额外添加了基本的Int，Char，String常量的非终结符，在之后的使用中，读取非终结符即可获得常量
3. 在根据语法，进行下一步解析时，直接调用对应语法成分的of方法，会自动调用词法，应对不限长度的语法成分

接口

本次在之后会被调用的内容比较单一，基本存在于生成的语法树中，因此语法树包含部分接口方法，语法解释器也包含部分方法

Parser

1. 开始解析

```
public AbstractSyntaxTree parse() throws Exception;
```

在声明一个语法解析器后，直接调用parse方法开始解释，会返回一个完整的抽象语法树

ast

1. 开始解析

```
/**
 * 开始解析
 */
public void parse();
```

2. 按树状结构打印语法树

```
/**
 * 按树状结构打印内容
 */
public void print();
```

更直观的展示语法树的结构，但与课程要求不同
3. 按平面打印语法树

```
/**
 * 按平面格式打印内容（忽略课程要求内容）
 */
public void printAns();
```

和课程要求相同
4. 返回根节点

```
    /**
     * 返回抽象语法树的根节点（CompUnit）
     * @return 根节点
     */
    public Symbol getSymbol();
```

得到CompUnit，同时也可以作为元素进行遍历