Course Design of Compiling.

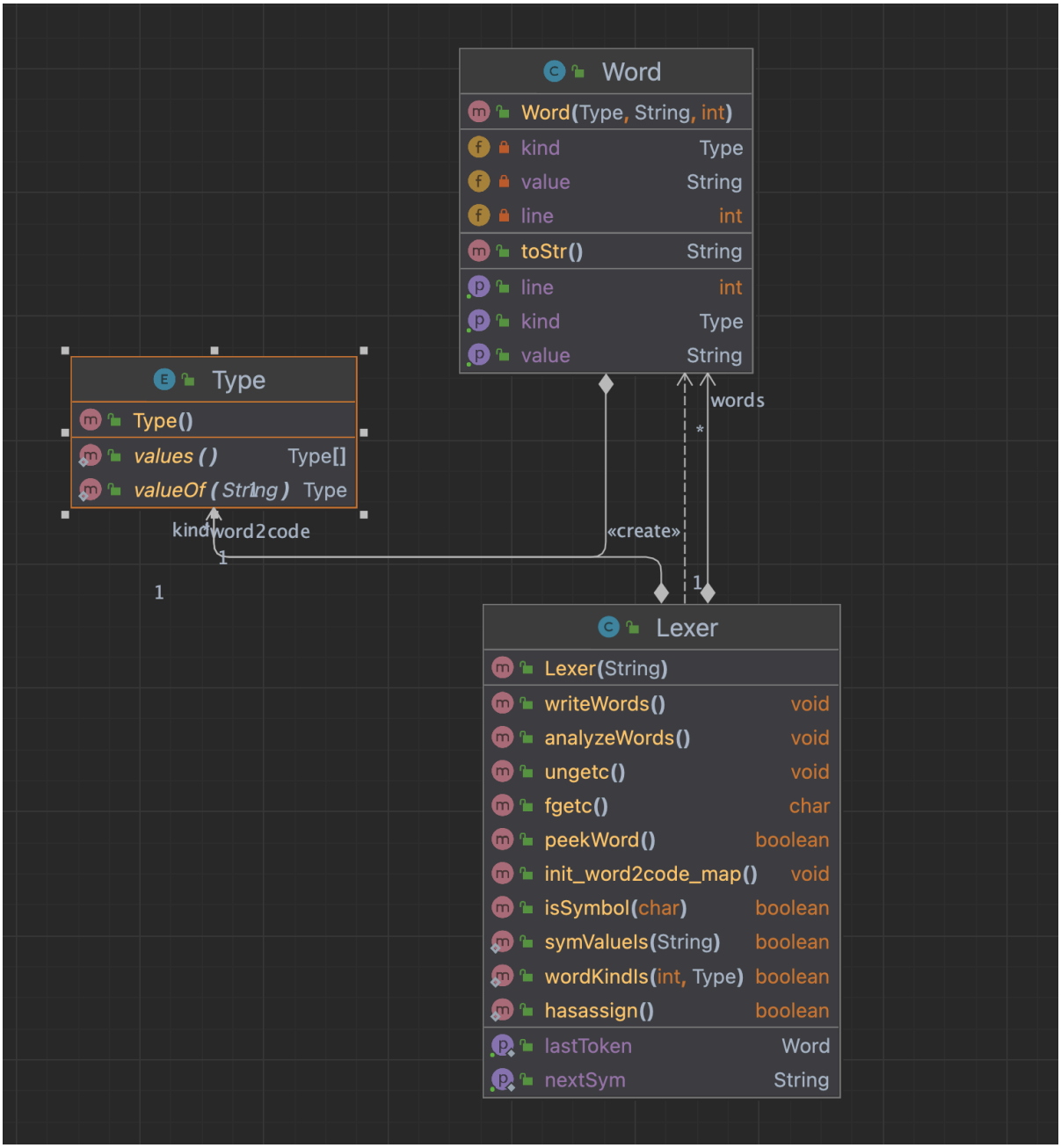
Autumn 2022. BUAA.

author:szy

编译器设计文档

1.词法分析

1.1 架构设计



1.1.1 Type Class

该类为枚举类,枚举类主要枚举了所有要识别的单词名称对应的类别码（包括下表中的所有类别码）。之所以使用枚举类，一方面可以统一的记录所有的类别码类型，我们在Word类中记录单词对应类别码时，可以直接调用已经固定好的Type内的所有名称类别，避免我们自己写类别码String时打错，造成低级Bug。另一方面，使用枚举类更可以大大增强代码的可读性。

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	!	NOT	*	MULT	=	ASSIGN
IntConst	INTCON	&&	AND	/	DIV	;	SEMICN
FormatString	STRCON	||	OR	%	MOD	,	COMMA
main	MAINTK	while	WHILETK	<	LSS	(	LPARENT
const	CONSTTK	getint	GETINTTK	<=	LEQ	)	RPARENT
int	INTTK	printf	PRINTFTK	>	GRE	[	LBRACK
break	BREAKTK	return	RETURNTK	>=	GEQ	]	RBRACK
continue	CONTINUETK	+	PLUS	==	EQL	{	LBRACE
if	IFTK	-	MINU	!=	NEQ	}	RBRACE
else	ELSETK	void	VOIDTK				

- Word类

该类记录单词的String值，单词类别码（Type类），line行数（行数是在错误处理的时候加的，因为错误处理时需要输出错误所在的行数）。

- Lexer类

词法解析器类，该类实现词法解析的主要功能，主要采用流式解析的方式，分为数字，字母，操作符（特殊字符）三类进行解析。主要分为以下三步

1. 读取文件中的所有字符，组成一个字符ArrayList，在ArrayList最后引入特殊符号～代表文件结束（EOF）。
2. 检测到符合词法的单词后，为每个单词生成Word类并存入wordList中；之后继续读取、处理后续字符。检测合法单词的顺序如下
    - 跳过空白符号，记得遇到\n换行
    - 如果遇到～符号，即终止符号，停止循环
    - 检测到字母或”_“开头,说明是Ident或者字符串类型的保留字(通过word2code是否含有该key来区分)
    - 检测到引号“，处理Formatstring
    - 检测到数字打头，处理数字
    - 检测到字符`!&|+-*/%<>=;,()[]{}`任一
        - 是`/`,是两种注释或者是除号，处理注释的时候记得换行时，保持line更新
        - 是`&`,检测到`&&`
        - 是`|`，检测到`||`
        - 是`<>=!`，看看后面还有没有`=`，分两种情况处理

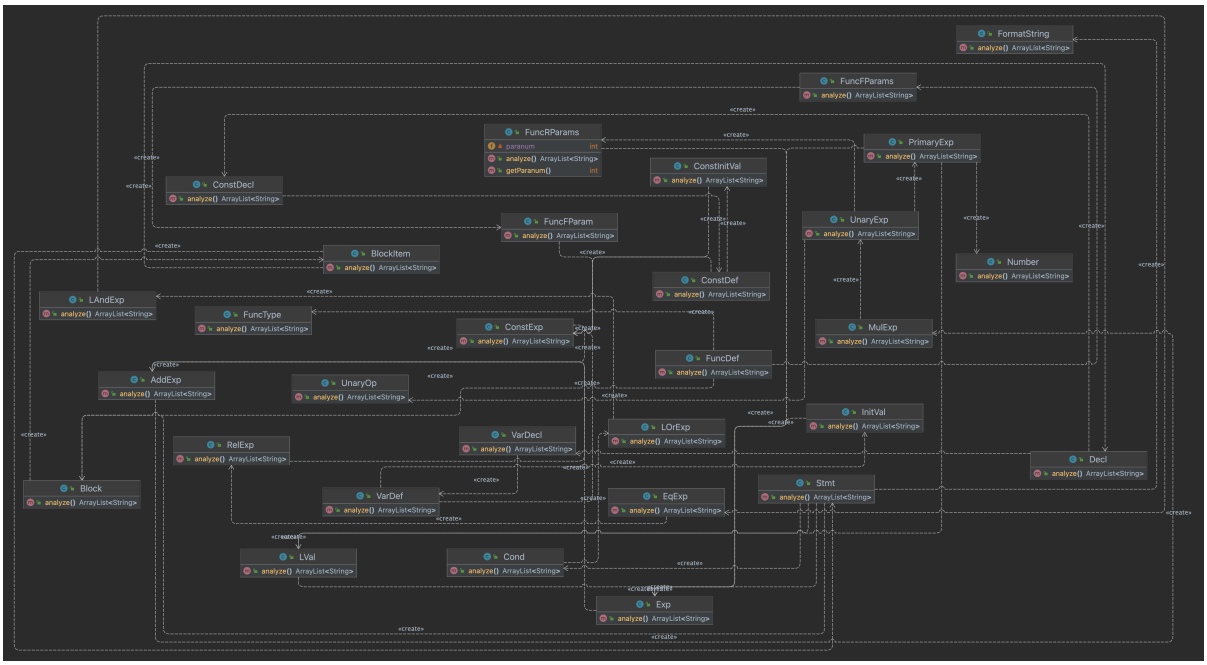
        - 其他的`+ - * % ; , ( ) [ ] { }`没有特殊情况，就一个单字符，一起处理掉
    - 循环重复上述过程

3. 额外的，在处理过程中如果遇到"\n"换行符，那么在Lexer中记录行号line，在创建word的时候可以该单词所处的line也记录到word类中。

1.2 设计修改

在进行语法分析设计时，我们把当前单词的提取，回退，类别码判断，单词值判断等功能方法全都交给了词法分析器Lexer执行，这些功能与语法分析器相分离，更好的降低了耦合度。在进行错误处理时，由于我们需要输出行数，因此在Word类中新增了line属性，用于记录每个单词所在行数。

2. 语法分析



2.1架构设计

总体下降采用递归下降的方法，从UML类图可以看到，分别给每一个语法成分编写了其对应的analyze解析函数，只需要根据文法进行对应的语法成分解析函数调用即可，思路非常简单。但问题在于，使用递归下降的方法解析意味着文法中不能出现左递归的情况存在，因此我们对文法进行了一定的改写，消除了左递归，经过观察我们发现左递归全部出现在表达式的部分文法中，只需要对这部分文法进行修改即可。此外在进行架构设计的时候，还出现了其他几个问题，统一总结如下。

2.1.1 文法左递归问题

我们利用理论课的两条规则消除左递归。

> 规则一（提因子）:U::=xy|xw|...|xz ⇒U::=x (y|w|...|z)
>
> 规则二：U::=x|y|...|z|Uv ⇒U::=(x|y|...|z) {v}

修改前

```
<MulExp>        := <UnaryExp> | <MulExp> ( '*' | '/' | '%' ) <UnaryExp>
<AddExp>        := <MulExp> | <AddExp> ( '+' | '-' ) <MulExp>
<RelExp>        := <AddExp> | <RelExp> ( '<' | '>' | '<=' | '>=' ) <AddExp>
<EqExp>         := <RelExp> | <EqExp> ( '==' | '!=' ) <RelExp>
<LAndExp>       := <EqExp> | <LAndExp> '&&' <EqExp>
<LOrExp>        := <LAndExp> | <LOrExp> '||' <LAndExp>
```

修改后

```
<MulExp>        := <UnaryExp> { ('*' | '/' | '%') <UnaryExp> }
<AddExp>        := <MulExp> { ('+' | '-') <MulExp> }
<RelExp>        := <AddExp> { ('<' | '>' | '<=' | '>=') <AddExp> }
<EqExp>         := <RelExp> { ('==' | '!=') <RelExp> }
<LAndExp>       := <EqExp> { '&&' <EqExp> }
<LOrExp>        := <LAndExp> { '||' <LAndExp> }
```

其余文法与原来保持不变。还有一个注意的的要点在于，我们解析的时候虽然改写了文法，但是在输出的时候还需要跟原来的文法保持一致。比如我们改写了MulExp，正常来说我们只需要在最后加`add("<MulExp>");`,把输出加进去，但是这并不是原文法的输出，我们需要在（1）出也加上，才是原文法的输出。

```
public class MulExp extends GrammarElement {
    @Override
    public ArrayList<String> analyze(){
        addAll(new UnaryExp().analyze());
        while (Lexer.symValueIs("*") || Lexer.symValueIs("%")
        || Lexer.symValueIs("/")){
            add("<MulExp>"); // (1) .是上一个的语法父节点，这里也要加
            add(Lexer.getNextSym()); //* / %
            addAll(new UnaryExp().analyze());
        }
        add("<MulExp>");
        return sublist;
    }
}
```

2.2.2 向前看问题

在文法中遇到"|","{}","[]"，就会存在向前看问题，因为这里存在不确定性，我们需要向前看来判断这里到底是来确定到底走哪个分支，或者这里到底是否存在这个非终结符。

2.2.3 Stmt解析中Lval开头还是Exp开头

```
Stmt →
| 'if' '( Cond ')' Stmt [ 'else' Stmt ]
| 'while' '(' Cond ')' Stmt
| 'break' ';' | 'continue' ';'
| 'return' [Exp] ';'
| 'printf''('FormatString{,Exp}')'';'
| Block
| LVal = 'getint''('')'';'
| LVal '=' Exp ';'
| [Exp] ';'
```

这里我们通过查看分号前有没有等号出现，如果Lval开头则必然会有一个等号，如果不是Lval开头，以Exp开头，Exp内部不存在等号。

```
public static boolean hasassign() {    //查找分号前有无等号
        int offset = 1;
        while (index + offset < words.size()) {
            Word newsym = words.get(index + offset);
            if (newsym.getValue().equals(";")){
                break;
            } else if (newsym.getValue().equals("=")) {
                return true;
            }
            offset += 1;
        }
        return false;
    }
```

2.2.4 递归下降架构的设计

设计递归下降时，为了程序的架构更加明晰，我将每个左部的非终结符都建立了类，这些类都继承一个父类GrammarElement。该父类包含sublist数组属性，这个数组包含该非终结符根据文法推出的所有终结符以及非终结符字符串，用于记录和输出。我们使用add和addAll方法向这个数组里添加元素，在递归调用的过程中，我们不断的add，addAll，便可以将sublist补充完毕。每一个非终结符都有sublist，这也是为什么我要创建这个父类的原因，减少代码冗余，最后CompUnit里面的字符串元素就是我们要输出的结果。同时该父类有一个analyze方法，没个非终结符子类需要重写该方法，以实现每个非终结符的解析。

```
public class GrammarElement {
    protected ArrayList<String> sublist;

    public GrammarElement(){
        sublist = new ArrayList<>();
    }
    public ArrayList<String> analyze(){
```

```
            return null;
        }
        public void add(String s){
            sublist.add(s);
        }
        public void addAll(ArrayList<String> ss){
            sublist.addAll(ss);
        }

        public void error() {
            System.err.println("Error!");
        }
    }
```

2.2 设计修改

与设计时最初的架构大概无变化。

错误处理

分别处理a-m问题
a:
问题:格式字符串中出现非法字符报错行号为所在行数
解决：formatstring类中