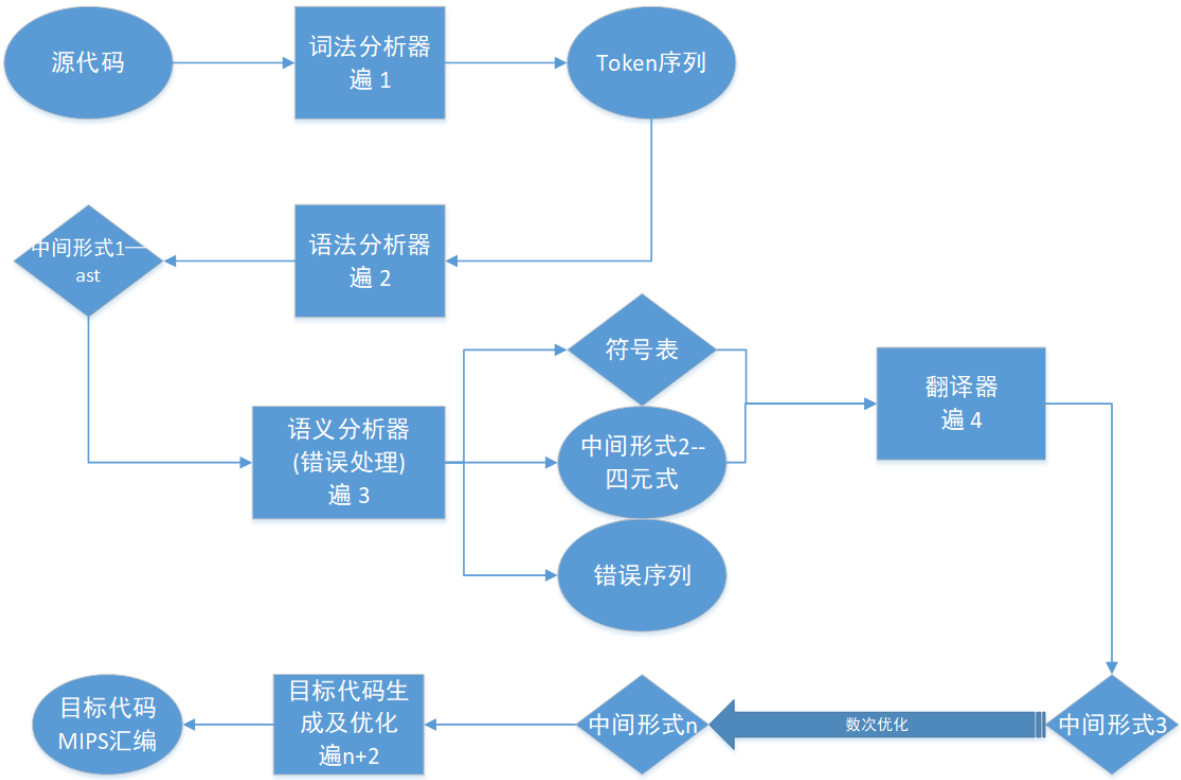
编译技术实验课程申优文档 19373126 张昊

一、总体架构

本课设采用Java设计

为了符合课程设计的评测体系，本课程设计采用了多遍的编译器架构设计。

整体架构设计如下图



图中椭圆形代表编译器的某翻译子程序的源代码/目标代码等**输入输出数据**；矩形为编译器的组成部分，为各种**翻译子程序**；菱形代表编译器运行时保存的**数据结构**；

具体设计见相应的翻译子程序的说明。

二、词法分析

1. 设计概述

根据课上内容可知，词法分析器本质上就是一个**输入字符，输出Token的DFA**，因此仅需定义所有**确定状态**，**状态转换**，**输出行为**，即可开始实现

(1). 状态定义

状态含义

本DFA的"状态"含义为**输入字符串从当前字符开始（包括当前字符）向前所能匹配的最长关键字（或其部分）**

> 如输入序列为"constab+"
>
> - 当前字符为n，则从当前字符向前，"con"匹配"**con**st"的前半部分"con"或是"**con**tinue"的前半部分，或是标识符"con"，即为当前状态`CONST_N_CONTINUE_N1`的意义

> - 当前字符为s，则从当前字符向前，"cons"匹配"**cons**t"的前半部分"cons"，或是标识符"cons"，即为当前状态`CONST_S`的意义
> - 当前字符为a，则从当前字符向前，"consta"仅能匹配标识符"consta"，则当前状态为`IDENT`

状态的确定与实现

工程化的状态定义需要将文法转化为正则文法，再通过一般化方法找到NFA定义，将NFA转化为DFA，再极简化DFA，才能获得需要的最简化词法分析DFA

但由于这样做工程量太大，而题目文法较为简单，所以省略这些步骤，而代之以两步——

1.将所有关键字的所有字符作为一个状态(标识符、字符串、字符常量这些无限长的Token可以转化为有限个状态表示)

2.合并相同出发、转换条件的状态

如开始时，若输入一个"c"，则可能为**const**关键字或**continue**关键字的"c"，原本定义有`CONST_C`与`CONTINUE_C`，而由此将两者合并为一个状态`CONST_CONTINUE_C`

通过分析所有关键字的字符，即可获得状态定义，利用Java的enum类实现如下

```
enum State {
    START,
    IDENT, INTCONST,
    FORMAT_STR_LEFT, FORMAT_STR_NORMAL, FORMAT_STR_PERSENT, FORMAT_STR_D,
FORMAT_STR_RIGHT,
    MAIN_M, MAIN_A, MAIN_I, MAIN_N,
    CONST_CONTINUE_C, CONST_CONTINUE_O, CONST_N_CONTINUE_N1, CONST_S,
CONST_T,
    INT_IF_I, INT_N, INT_T,
    BREAK_B, BREAK_R, BREAK_E, BREAK_A, BREAK_K,
    CONTINUE_T, CONTINUE_I, CONTINUE_N2, CONTINUE_U, CONTINUE_E,
    IF_F,
    ELSE_E1, ELSE_L, ELSE_S, ELSE_E2,
    NOT, AND_1, AND_2, OR_1, OR_2,
    WHILE_W, WHILE_H, WHILE_I, WHILE_L, WHILE_E,
    GETINT_G, GETINT_E, GETINT_T1, GETINT_I, GETINT_N, GETINT_T2,
    PRINTF_P, PRINTF_R, PRINTF_I, PRINTF_N, PRINTF_T, PRINTF_F,
    RETURN_R1, RETURN_E, RETURN_T, RETURN_U, RETURN_R2, RETURN_N,
    VOID_V, VOID_O, VOID_I, VOID_D,
    PLUS, MINU, MULT, DIV_CMT, SGL_LINE_CMT, MLT_LINE_CMT, MOD,
    LSS, LEQ_EQ, GTS,  GEQ_EQ, EQL_ASN, EQL_2, NEQ_EQ,
    SEMICN, COMMA,
    LPAR, RPAR, LBRACK, RBRACK, LBRACE, RBRACE,
    END,
    ERROR
}
```

其中，START状态为开始状态，也是输入为空串时的状态；END为匹配完毕一个Token后的（暂时）结束状态，ERROR则为错误状态

(2). 状态转换与输出行为

定义

本设计中将其实现为一个Mealy型状态机，因此状态转换与输出是同步的

将状态分类：

- 状态共有以下几种
  - 开始、结束、错误状态
  - 关键字状态（如`GETINT_G`、`LSS`等期待关键字或算符的字符的状态，特点为**有限长**），也分两种
    - 完成状态（如`GETINT_T2`、`VOID_D`等）
    - 未完成状态
  - 无限Token状态（如`INTCONST`等期待非关键字的状态，特点为**无限长**）

状态转换与输出行为讨论如下：

- 对于开始状态`START`，读取第一个字符，根据第一个字符进入对应关键字或非关键字状态
- 对于每一个未完成关键字状态，有以下三种情况
  - 当前字符为期待的关键字字符
    - 则下一状态为新的期待的关键字状态
    - 如当前状态为`GETINT_G`，当前字符为'e'，则下一状态为`GETINT_E`
  - 当前字符为不期待的关键字字符，或是其他构成标识符的字符
    - 则下一状态为标识符状态`IDENT`
    - 如当前状态`GETINT_G`，当前字符为't'，则下一状态为`IDENT`，即意为标识符"gt"
  - 当前字符不能构成标识符，但是是合法输入字符，或到达文件末尾
    - 则下一状态为标识符状态`IDENT`
    - 如当前状态`GETINT_G`，当前字符为'2'，则下一状态为`IDENT`，即意为标识符"g2"
    - 或是如当前状态`GETINT_G`，但当前字符为空白字符（包括换行符），或是到达文件末尾，则下一状态为`IDENT`，即意为标识符"g"
- 对于每一个完成关键字状态，有以下两种情况
  - 当前字符为其他构成标识符的字符
    - 则下一状态为标识符状态`IDENT`
    - 如当前状态`GETINT_T2`，当前字符为'a'，则下一状态为`IDENT`，即为标识符"getinta"
  - 当前字符不能构成标识符，但是是合法输入字符，或到达文件末尾
    - 则输出相应信息，下一状态转为`END`
    - 如当前状态`GETINT_T2`，当前字符为空白字符，或是'='，或是到达文件末尾，则输出`GETINTTK`，即意为匹配到了关键字"getint"
- 对于每一个无限Token状态，有以下两种情况
  - 当前字符为期待字符
    - 下一状态为原状态
    - 如当前状态为”INTCONST"，而当前输入字符为数字字符"1"，则继续保持当前状态
  - 当前字符为非期待字符，或是终止字符/空白字符
    - 输出当前字符，下一状态为`END`
    - 如当前状态为"INTCONST"，当前输入字符为字母字符'a'或是空白字符，则转为`END`状态，并输出信息`IntConst`，表示匹配到了数字常量
    - 特殊的是，对于字符串常量`FormatString`，它有一个专门的终止字符双引号' " '，只有遇到这一字符才执行上述操作，否则继续匹配
- 对于终止状态`END`

    - 下一状态为`Start`状态
- 对于任何一个状态，若输入为不合法字符
    - 状态转为错误状态`ERROR`，并输出错误信息

实现

通过switch-case语句以及if语句实现即可，体力活~

2. Debug以及细节处理

输入字符流

源程序中的代码将以行读取，方便定位行号

每次匹配成功后，会将匹配成功的字符串从输入字符流中剔除，产生的新字符串再作为后续处理的输入字符串，这样以流式处理方便定位当前匹配的输入字符串的子串（即保证每个关键字都从0开始匹配）

同时，当前处理的字符串为空时，即表示处理到了换行符，由上述状态转换可知，状态将转换为`END`状态，之后为`START`状态，在该状态中读取新行（若为文件末尾则处理完毕）

字符串常量中的特殊字符

字符串常量中的占位符"%d"与换行符"\n"是两个极为特殊的字符组合，因为百分号"%"与反斜杠"\"是两个非法字符

因此需要额外定义两个特殊状态`FORMAT_STR_PERSENT`与`FORMAT_STR_BACKSLASH`，当状态为这些状态时，若下一个字符不是'd'或'n',则状态机转为错误状态

空白符处理

通常情况下，状态机需要一个单独状态匹配空白符，本课设中将`START`与`END`状态作同样用途，即在开始匹配关键字前/匹配一个关键字结束后删除无用的空白符

不过为了提高效率，通过应用正则表达式方法`replaceAll`删除字符串开头的多个空白符

优先级

匹配关键字的优先级总是大于非关键字（如标识符`Ident`），这点由case语句中if的顺序保证（某log既视感）

其他

由于代码量突破了1k行，因此可能发生手滑打错以及copy出错等导致的bug，需要多多注意

三、语法分析

按照要求，本设计中语法分析器采用递归子程序法对文法中定义的语法进行分析

1. 设计概述

若要采用递归下降法分析语法，文法需要满足——无左递归，无回溯，而我们的SysY语言的文法中，这两个现象都存在，须予以消除

处理后，即可采用非常直观的递归下降法处理

(1). 文法处理

左递归

目标语言SysY中的左递归规则十分常见，处理方法也比较常规

如对于规则 `加减表达式 AddExp -> MulExp | AddExp ('+' | '-') MulExp`，明显在规则右侧的第二分支处存在左递归，处理成以下两条规则即可

```
AddExp -> MulExp AddExpSuffix
AddExpSuffix -> ('+'|'-') MulExp AddExpSuffix | ϵ
```

回溯

回溯主要靠**提取左公因子**和**超前扫描**处理，对于难以处理的，采用**回溯**解决

- 提取左公因子
  - 需要对非终结符所能产生的所有支路进行公因子提取
  - 但是由于文法需要输出非终结符的相关信息，在提取公因式时要尽量避免公式右侧非终结符的代换与展开，而尽量采用超前扫描的方法解决回溯，否则程序复杂度将会比较大
- 超前扫描
  - 下文详述

(2) 递归下降法的程序设计

核心设计思想

递归下降法的程序实现非常直观——输入为Token标志序列，程序中为每一个非终结符实现一个`parse*()`方法，方法中处理该非终结符的匹配

对于该终结符所在规则右侧的每个支路，从左往右扫描

- 若为终结符，则查看token标志序列中的当前标志是否为该终结符对应的Token标志，是，则继续，否则发生失配
- 若为非终结符，则调用对应的`parse*()`方法以匹配这一非终结符，再继续之后的匹配
- 若在当前支路发生失配，则切换到下一支路，从再重复上述操作，直至完全匹配或遍历所有支路为止（是错误情况，将报错）

支路切换

由于已经进行了**提取左公因式**，因此对于任一非终结符所能导出的所有支路，都可以保证其First集相交为空

这意味着对于某提取公因式后的规则

`A->ab(c|d)`

在`parseA()`方法中，先匹配终结符'a''b'，之后进入分支结构，若'c'匹配成功，则走左侧支路，否则走右侧支路，即开始匹配'd'

超前扫描

由于前述原因，提取左公因子的方法能力有限

如对于规则

```
A -> Cc|Ee
C -> acd
E -> afg
```

显然，$First(Cc) \cap First(Ee) = \{a\}$，此时若采用提取公因式的方法解决$A$中的回溯问题，需要将$C$、$D$两个非终结符代入$A$的规则中，变为

```
A->a(cd|fg)
```

这意味着`parseA()`要这样实现

```
boolean parseA() {
    match('a');
    if (match('c')) {//match 返回false时，当前非终结符不变，返回true时，当前非终结符为下一个
        match('d');
        output("CTK");
    } else {
        match('f');
        match('g');
        output("ETK");
    }
    output("ATK");
}
```

可以发现，`parseA`方法其实内联展开了`parseC`与`parseE`方法，程序的耦合度较高，而且当支路增多时程序将变得比较繁杂

因此对于这种情况多采用**超前扫描**解决，即如下方法

```
boolean parseA() {
    match('a');
    String sign = peek();//peek当前非终结符不变
    if (sign.equals("c")) {
        parseB();
    } else {
        parseC();
    }
    output("ATK");
}
boolean parseC() {
    match('a');
    match('c');
    match('d');
    output("CTK");
}
boolean parseE() {
    match('a');
    match('e');
    match('g');
    output("CTK");
}
```

降低了程序的耦合度

2. 实现与难点处理

- 流式输入Token序列
  - 通过一个Token数组与now指针指向当前Token，以流式匹配终结符——匹配成功则指针前进，否则不动，保证遍历所有Token
- 错误处理代替分支
  - 失配本质上是一种Exception，因此实现中自定义了Exception的子类`MissMatchException`，当失配时抛出，若为分支切换，则在try-catch语句中继续下一分支的处理，若为不期待的失配，则直接抛出以停止程序(或错误处理)
- 必要的回溯
  - 文法中`<stmt>`非终结符能导出的句型很多，其中隐含着一个非常tricky的回溯问题，导致了课设好几次过不去评测
  - 问题大概如下
    - 由文法有

```
Stmt -> [Exp]';' =*> LVal ...
Stmt -> LVal '=' Exp ';'
Stmt -> LVal '=' 'getint''('')'';'
LVal -> Ident {'[' Exp ']'}
```

    - 推导1、2、3的右侧三个句型均以非终结符LVal开头，而它们都是Stmt的分支，这使得需要超前扫描到LVal右侧的符号才能判断进入哪个分支
    - 然而问题在于，由规则4，LVal可能是无限长的（指无法预知其长度），因此不能通过超前扫描**常值**个终结符判断进入哪个分支，因此只能通过回溯解决问题
    - 具体说来，需要在`parseStmt`中先屏蔽输出，保存当前指针now，并调用`parseLVal`方法先将指针移动到LVal之后的非终结符，通过这个非终结符即可判断进入哪个支路，再将指针回到保存的位置，继续进行后续的分析，代码示例如下

```
void parseStmt() throws MissMatchException{
    //...
    setPointerCheckPoint();//指针保存
    setOutputEnabled(false);//屏蔽输出
    parseLVal();//提前parse
    setOutputEnabled(true);//启动输出
    String sign = peekSign();
    if (sign.equals("=")) {
        //则当前支路为 LVal '=' 'getint''('')'';' 或LVal '=' Exp ';'
        pointerFlashBack();//指针回溯
        parseLVal();//LVal的信息之前没有输出，需要回溯重新输出一次
        //继续匹配...
    } else {
        //则当前支路为 Exp';'
        pointerFlashBack();
        parseExp();
        //继续匹配...
    }
}
```

- 左递归处理与输出
  - 按照我们的左递归处理方法处理文法后，要输出对应的非终结符信息，需要略加调整如下

```java
public void parseAddExp() throws MissMatchException {
    parseMulExp();
    out("<AddExp>");
    parseInfinityAddExp();
}

public void parseInfinityAddExp() throws MissMatchException {
    String sign = peekSign();
    switch (sign) {
        case "+":
            expect('+');
            break;
        case "-":
            expect('-');
            break;
        default:
            return;//ϵ
    }
    parseMulExp();
    out("<AddExp>");
    parseInfinityAddExp();
}
```

而不是在`parse*()`方法的末尾才输出信息

四、中间代码、语义分析与错误处理

1. 设计概述

（1）抽象语法树与语义分析

语法分析结束后，要进行语义分析，有两种思路：

- 一种即在语法分析的过程中(`parse`函数中)分析语义并处理语义错误
  - 但这会使得编译程序耦合度过高，程序各部分分工不明确，使Debug等更加困难
- 另一种则是通过语法分析生成中间代码，再通过分析中间代码来分析语义
  - 但是若在较为底层的中间代码如四元式序列等进行语义分析也是比较困难的，因为生成这些中间代码的过程中会丢失很多信息（如语法分析得到的语法树）

因此课设中在语法分析过程中会生成抽象语法树(AST)，之后在抽象语法树上进行语义分析以及语义有关的错误处理

- 一方面，解耦语法分析与语义分析，程序结构明了
- 另一方面，抽象语法树很大程度上保留了语法分析的信息，便于语义分析

（2）抽象语法树的设计与构建

AST设计

抽象语法树的设计思想比较直观，基本上就是对文法中所有非终结符建立一个同名类，成为树上的节点。而每个非终结符所能导出的其他非终结符对应的节点则是它的子节点。

根节点`CompileUnit`有全局变量声明`Decl`、函数定义`FuncDef`和主函数定义`MainFuncDef`作为子节点

```java
public class CompileUnit {
    private ArrayList<Decl> globalDecls;
    private ArrayList<FuncDef> funcDefs;
    private MainFuncDef mainFuncDef;

    public CompileUnit(ArrayList<Decl> globalDecls, ArrayList<FuncDef>
funcDefs, MainFuncDef mainFuncDef) {
        this.globalDecls = globalDecls;
        this.funcDefs = funcDefs;
        this.mainFuncDef = mainFuncDef;
    }
    //...其他函数
}
```

除了对应树结构的非终结符信息外，节点内部还要保留必要的终结符信息，如`LVal`中的`Ident`，`Number`字符串等。

AST的构造

语法分析阶段就可以构造AST：

- 将原本的语法分析方法改为返回对应的类即可
    - 将原本的输出语句改为对应类的构造方法的调用语句
    - 同时将其子节点（能推出的非终结符以及需要的终结符信息）作为参数输入

如`parseCompileUnit()`方法改为

```java
public CompileUnit parseCompUnit() {
    globalDecls = parseMultiOrZeroDecl();
    funcDefs = parseMultiOrZeroFuncDef();
    mainFuncDef = parseMainFuncDef();
    //out("<CompUnit>");
    return new CompileUnit(globalDecls, funcDefs, mainFuncDef);
}
```

（3）语义分析实现

分析框架

语义分析本质上需要进行语法树的**遍历**

因此对每个语法树节点设计一个语义分析的方法`scanForErrors()`，然后递归地调用即可（深度优先）遍历完整个语法树

数据结构——符号表设计

语义分析需要分析变量的定义与引用，因此必然要用到符号表

符号表本身由一个单独的类`SignTable`实现，内部有多个`SignTableItem`对象，以保存各个符号表项

在最初版本的符号表设计中，符号表结构是按照课上内容进行设计

- 一个`ArrayList<SignTableItem> tableBody`属性保存符号表项
- 一个`ArrayList<Integer> indexTable`属性保存作用域的索引信息

符号表项的内容如下

- 符号类型

    - 变量/函数
    - 变量/常量/参数
    - 有/无返回值的函数
- 符号名
    - 标识符`ident`
- 符号对应的树节点
    - 如变量，则保存其对应的`VarDef`等

语义分析过程与符号表行为定义

语义分析的入口为`CompileUnit`的`scanForError`方法，语义分析之后调用该方法

方法内部会new一个`SignTable`对象，之后在其他所有节点的遍历过程中将该表作为参数传递

其中：

- 每当树的遍历过程中碰到新的Block时，调用`SignTable`的`intoLayer`方法
    - `SignTable`会在索引表`indexTable`栈顶添加一个指针，指向符号表`tableBody`栈顶+1，标识新作用域的符号表项将要被添加的位置
- 每当树遍历过程中碰到新的变量/函数/参数声明时，调用`SignTable`的`addItem`方法
    - `SignTable`会先检查在该作用域下是否重定义（从索引栈顶的指针到符号表栈顶）类型相同且标识符同名则重定义，返回错误，否则继续
    - `SignTable`会在符号表`tableBody`栈顶添加新`SignTableItem`，内部保存标识符名、符号类型等内容
- 每当树的遍历过程中碰到新的变量/函数/参数调用时，调用`checkDefinitionOfItem`方法
    - `SignTable`会检查该调用是否定义，类型是否对应(注意变量可以和函数同名，两者的命名空间要区分)
    - 需要从栈顶的索引指针开始，检查到栈顶，找不到则从栈顶的索引指针开始反向向栈底检查
    - 未定义则返回错误
- 每当树的遍历过程中退出Block时，调用`SignTable`的`outOfLayer`方法
    - `SignTable`会弹出当前作用域内的所有表项
    - 之后索引表`indexTable`栈顶也会弹出一个指针，表示当前作用域分析完毕

（4）错误处理

错误类型	错误类别码	解释	对应文法及出错符号(...省略该条规则后续部分)
非法符号	a	格式字符串中出现非法字符 报错行号为<FormatString>所在行数。	<FormatString> → '"'{<Char>}'"'
名字重定义	b	函数名或者变量名在当前作用域下重复定义。 注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。 报错行号为<Ident>所在行数。	<ConstDef>→<Ident> ... <VarDef>→<Ident> ... |<Ident> ... <FuncDef>→<FuncType><Ident> ...
未定义的名字	c	使用了未定义的标识符 报错行号为<Ident>所在行数。 此处包括函数未定义（在函数调用前没有函数声明）和变量/常量未定义。	<LVal>→<Ident> ... <UnaryExp>→<Ident> ...
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。 报错行号为函数名调用语句的函数名所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。 报错行号为函数名调用语句的函数名所在行数。	<UnaryExp>→<Ident>'('[FuncRParams]')'
无返回值的函数存在不匹配的return语句	f	报错行号为'return'所在行号。	<Stmt>→'return' {['Exp']}';'
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句即可，无需考虑数据流。 报错行号为函数结尾的'}'所在行号。	FuncDef → FuncType Ident '(' [FuncFParams] ')' Block
不能改变常量的值	h	<LVal>为常量时，不能对其修改。 报错行号为<LVal>所在行号。	<Stmt>→<LVal>'=' <Exp>';' |<LVal>'=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号前一个非终结符所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号前一个非终结符所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号前一个非终结符所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为'printf'所在行号。	Stmt →'printf'('FormatString{,Exp}')';'
在非循环块中使用break和continue语句	m	报错行号为'break'与'continue'所在行号。	<Stmt>→'break';' |'continue';'

确定错误处理的阶段

首先分析各种错误发生的编译阶段

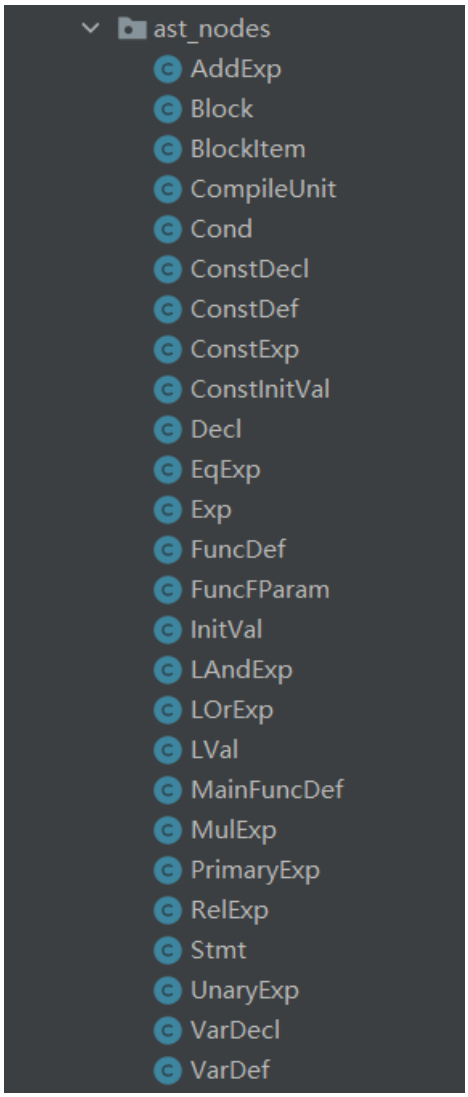
- 词法分析
  - a，词法分析状态机即可识别
- 语法分析
  - i、j、k是典型的语法错误，语法分析在匹配终结符时即可识别
- 语义分析
  - b、c、d、e、f、g、h、l、m、需要联系上下文来进行识别与处理，因此只能放在语义分析来做（实际上也有语法错误可以处理的错误，为了最大程度解耦，就都放在了语义分析部分）

错误处理方法

- 工具类
  - 自定义一个`DemandError`类保存出错信息（错误类型、行号等）
  - 包装一个`DemandError`的数组，形成错误信息管理器`ErrorManager`
- 错误处理
  - a类错误
    - 词法分析状态机中添加格式字符串错误处理功能
      - 在`FORMAT_STR_PERSENT`的状态下，识别输入字符是否valid，是则添加一个a类`DemandError`（并添加相应行号信息）到`ErrorManager`
  - b、c类错误
    - 如前文**语义分析实现**所述，在分析到变量声明/变量调用时，在符号表中分析有无重定义以及未定义错误
    - 有则添加相关信息的`DemandError`到管理器中
  - d、e类错误
    - 同样由查找符号表完成处理
      - 分析到函数调用时，查找符号表，先看有无定义，无则产生错误信息
      - 有，则从对应的符号表项中取出相应的语法树节点`FuncDef`，然后读取其中的形参信息，比较形参表和实参表的长度，判断并产生可能的错误信息
      - 将形参与实参逐一比较，查看其类型是否匹配，判断并产生可能的错误信息
  - f、m类错误
    - 在遍历语法树的过程中处理
      - 分析错误，可以发现
        - f类错误要求`return`语句所在的语法树节点不能成为`void`函数的孙子节点
        - m类错误则要求循环控制语句所在的语法树节点必须为循环语句节点的孙子结点
      - 故在遍历传参时，添加两个`boolean`参数`unexpectLoopControl`和`requiresReturn`
        - 语义分析入口的这两个参数分别为`true`和`false`
        - 每当碰到循环语句时，其子节点`stmt`的`scanForError`方法的`unexpectLoopControl`都会被置为`false`
        - 每当碰到`int`函数声明时，其子节点`block`的`scanForError`方法的`requiresReturn`都会被置为`true`
        - 其余节点均传播其父节点的参数，除非其本身也为这两种语句
      - 碰到`return`和循环控制语句时，检查当前参数是否允许它们存在，是则继续，否则生成错误信息
  - g类错误
    - 由于降低了要求，仅需在函数`int`函数定义`FuncDef`中检查其`Block`子节点中的`BlockItem`有无`return`语句，无则产生信息（不用递归检查孙子节点了）
  - h类错误
    - 本质上和b、c、d、e类错误一样，借助符号表完成
      - 碰到赋值语句时，获取左值（标识符以及数组名）对应的符号表项，查看其是否为常量，是则产生错误
  - i、j、k类错误
    - 语法分析、递归下降阶段检查括号是否匹配（终结符是否满足文法），不满足则产生信息

    - l类错误
        - 检查`printf`语句中的格式字符串和`exp`个数是否对应即可
    - 注意
        - 根据课设要求，一个输入源程序可能不止一处错误，所以不能碰到错误就停止（实际上也应该是这样，这是错误处理而非错误抛出）尤其是像括号不匹配等这些错误，遇到后保存错误信息，之后要继续分析，”假装“从来没发生过一样，除非遇到致命错误再停止

2. 实现与难点处理

- 语法树
    - 

    - 所有的语法树节点类如上图
    - 由于几乎所有类都有一个`scanForError`方法，其实令其实现一个接口可以方便管理，但是由于这里多态用处不大，所以没有实现
- 符号表重构
    - 最初版本的符号表考虑欠妥，照课本设计其实有些麻烦，而且易出bug，后续迭代中用`HashMap`重构了符号表
        - 考虑到符号表要不停的进行输入标识符，查找表项的操作，本质上是需要一个`String->SignTableItem`的映射，所以考虑采用`HashMap`
        - 整体结构为一个`HashMap`的`ArrayList`，每个`HashMap`对应一个作用域的所有`Ident->SignTableItem`映射
        - 进出`Block`时仅需弹出和压入`HashMap`即可创建和删除新作用域
        - 查表时从栈顶到栈底逐个取出`HashMap`，通过标识符查找表项
- 错误信息-行号保存

    - 错误信息需要输出错误行号
        - 对于词法、语法分析仅需保存好Token流中每个Token的行号即可保证遇到错误时能够输出
        - 语义分析这里需要在各个语法树节点上新增**标识符行号**这一属性，便于输出
- 后续在这一部分其实还有重构，见下文

五、四元式与目标代码生成

语义分析处理完毕之后将把抽象语法树转换为要求的四元式，目标代码生成则由四元式产生

1. 设计概述

（1）四元式设计

共设计四类四元式

- 赋值型
    - 包括赋值以及计算
- 声明型
    - 包括函数声明、变量声明、便于地址分配
- 动作型
    - 包括跳转语句、返回语句、输入输出语句等
- 标签
    - 跳转语句的目标位置

（2）抽象语法树到四元式

框架

仍然需要遍历抽象语法树，所以这里又对抽象语法树的每个节点类中新增方法——`generateQuaternion()`

入口为`CompileUnit`类中的同名方法，并在需要时返回存储结果的临时变量名

工具类

通过一个充满静态方法的类`QuaternionIRGenerator`集中管理生成四元式的接口以及四元式序列，同时也可以生成临时变量以及标签名

生成过程

递归地调用子节点的`generateQuaternion()`方法，在具有所有所需信息的节点处：

把所有需要的信息传入`QuaternionIrGenerator`类中的相应静态接口中，生成对应的四元式，并保存在它的四元式序列静态属性中

新符号表

由于语义分析和四元式生成两个过程之间的解耦，加之语义分析中的符号表随用随删，难以找到一个好方法保存符号表信息

但是生成四元式又需要**数组模板**和**变量作用域**等符号表信息（而块标记也没有保留到四元式序列中）

所以这里设计了两个新的专用符号表`ArrayTempletTable`和`IdentWithMarkTable`，在随着`generateQuaternion()`方法遍历抽象语法树的同时通过参数形式，参与到四元式生成的过程中

（3）四元式转化为目标代码

框架与工具类

在无优化的前提下，直接将上一步类`QuaternionIRGenerator`中的四元式序列取出，由工具类`MipsGenerator`读取四元式，生成目标代码

新符号表

我们需要变量的**地址信息**才能真正进行生成目标代码，这一信息本来应该由符号表保存，但仍然由于解耦，难以将语义分析阶段的信息保存到生成目标代码阶段

故这里又设计了新的专用符号表`IdentWithAddressTable`

先扫描一遍四元式序列，生成并记录变量在栈或是全局变量空间的地址信息，之后在真正生成MIPS代码的过程中用以给变量寻址

转换过程

先用`IdentWithAddressTable`扫描一遍四元式，在不同函数体以及全局声明区中分别记录变量的声明信息，同时记录变量的类型，占用空间等，压入栈中

同时程序中用到的临时变量也需要开辟空间

扫描完毕后逐个弹栈，分配相对地址空间，记录到HashMap中，同时记录函数的栈帧大小，以便于函数调用时栈帧指针的移动

之后扫描四元式，将不同类型的四元式翻译成对应的目标代码，输出到指定文件中即可

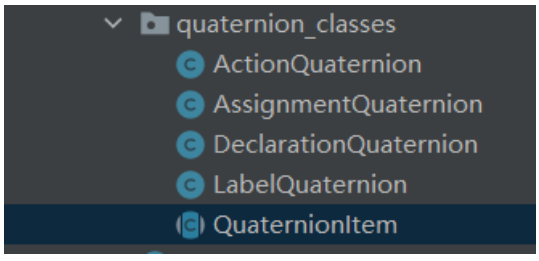
2. 实现与难点处理

AST-四元式

四元式实现类

四元式实现为一个抽象类`QuaternionItem`,具有一个`toString`方法便于输出，用多态便于管理四元式序列

对每种不同的四元式分别建立类，继承`QuaternionItem`方法，共四种，如下图



四元式补充定义

- 变\常量的初始化一律处理为普通赋值
- 数组下标a[i]、部分数组传参等处理为运算符号
- 所有数组的引用都必须按照a[b]的形式，包括多维数组。（但声明要说明维数以及每维长度，未来生成目标代码需要数组模板）
    - a为数组名，其地址中存有数组首地址，b为偏移量(/字)
- 特别地，getint处理为动作型、printf处理为printStr 和printInt两种动作
- OuterReturn在处理FuncCall后自动添加，即使是void函数也有一个返回值临时变量，但是不用

四元式变量更名

由于四元式中并未保留区块信息，难以在四元式中分析清楚引用变量所在的作用域

故在AST生成四元式时，通过`IdentWithMarkTable`表分析变量所引用的定义所在的作用域，之后将变量引用处的变量标识符更名为`identx,y`再生成四元式。ident即原变量名，`x,y`分别为变量所引用的作用域的嵌套深度以及同嵌套深度下的块标号（为了统一，定义处四元式也更名）

这样，四元式可以通过变量名区分出变量引用的定义所在的作用域了

数组模板表

数组模板生成四元式时，读取到数组声明时生成，内容为数组模板各个维对应的数组常量$P[i]$，同时也要记录数组声明的作用域并进行更名

数组引用时，既要更名引用处的数组便于查找作用域，又要通过数组模板生成计算数组偏移量的四元式代码，最终生成3/4型-赋值型四元式

表达式转化

由于AST的设计其实与文法结构类似，表达式是右结合的，不能直接把AST当作表达式树，通过后序遍历生成四元式

而应当对每个树的节点，如AddExp,要从左往右，从上到下生成四元式，计算顺序才正确

短路求值的实现

短路求值算是比较难实现的，主要是要弄清楚编译器本身的行为和程序运行的行为，想清楚后者，再来指导编译器的设计。

如`if(cond)...`的实现，由于文法简化，LOrExp和LAndExp不会被括号括起来，所以它们必须是最外层的表达式

所以要明确程序行为：

- LAndExp中每个RelExp之后，若为真，则继续执行下一RelExp，若为假，则跳转到当前LAndExp的尾部；同理
- LOrExp中每个LAndExp判断完毕之后，若为真，则直接跳转到if内部，若为假，则跳转到下一个LOrExp中

这样就可以不实现"&&"和"||"操作符也能完成if判断

之后定义编译器：

- 先生成if内部和if之后的两个Label:label1, label2
- 再在进入cond节点后，通过传参将两个Label传入
- 之后再通过传参，确定当cond为true，则跳转到label1, 若为false,则跳转到label2
- 之后再通过传参，确定当一个LOrExp为真时，跳转到cond为真的label，若LOrExp为假时，跳转到下一个LOrExp对应的标签处，这个标签需要生成并传参给到下一个LOrExp的遍历中
- ...（LAndExp同理）

如此善用标签生成与传参，即可实现好短路求值

对语义分析的修改

- ConstExp中出现const变量
    - 开始未考虑该情况，后经课程组提醒才实现
    - 通过在语义分析的符号表管理过程中添加功能

    - --在ConstExp的遍历过程中，在符号表中查找其中的所有标识符（已经保证为const变量），找到对应的声明语句节点，计算出const的值，添加到ConstExp的子节点中，进而解决问题

四元式-MIPS

变量地址表

由于通过变量名即可区分出各个变量的定义作用域，所以通过一个`HashMap`即可保存所有变量的`Ident-Address`信息

此外，本课设采用了纯压栈的方式进行传参，故在生成地址信息时，将形参的声明也要纳入考虑，将其映射到栈帧的指定位置（这里特别需要考虑清楚压栈和弹栈顺序是相反的，地址容易搞混）

栈帧结构

对于每次函数调用，其栈帧从底到顶依次为

- 所有参数
    - 保持压栈顺序
- 局部变量（包括临时变量与局部数组）
    - 保持声明顺序
    - 地址$sp + 4 + (n - i) * 4
- 返回地址
    - 地址$sp + 4

$sp指向栈顶元素的下一个元素，即栈帧的低地址（顶）-4

“优雅地”终止main函数

生成main函数时，先执行`la $ra, label`

label为一个终止`syscall`的指令，即可实现main函数return后程序停止的功能

六、代码优化

中间代码优化

基本块划分与流图生成

这是优化的基础

- 入口语句
    - 函数的第一条语句
    - Return|无条件跳转|条件跳转语句的下一条语句
    - 无条件跳转语句、条件跳转语句的目标语句
    - 末尾语句之后（exit块)
    - 注：函数调用不算，但是并不一般，需要保存上下文
- 流图组织
    - 无条件跳转->目标块
    - 条件跳转->目标块
    - 条件跳转->顺序下一块
    - Return块->exit块
    - 顺序前一块->后一块
- 实现：

- 第〇遍：
  - 按照FuncDef划分为过程块
  - 全局变量声明区也作为一个“过程块”，但不处理
- 第一遍
  - 记录可能被跳转的Label
- 第二遍
  - 扫描Label，标记Label所在语句
  - 扫描跳转语句，记录下一语句
- 处理
  - 按照前两遍的标记划分基本块
  - 若基本块以Label开头，则记录Label->基本块号的映射
- 第三遍
  - 扫描基本块的最后一句
    - 跳转语句->Label块
    - Return语句->exit块
  - 所有非无条件跳转语句以及return语句的块->顺序下一块
  - 反向设置pre链表

窥孔优化（中端）

中端指中间代码优化阶段，本阶段中可以进行一些窥孔优化以实现四元式方面的优化

合并四元式

四元式的生成过程中会产生很多只赋值使用一次的中间变量，其中很多可以合并如下：

```
%t  = a + b
d = %t
//合并为
d = a+b
```

（%t开头的是临时变量）

优化方法：

仅需顺序扫描四元式，检查相邻的2型赋值四元式和4型赋值四元式有无这种关系，有则修改前一句的被赋值变量，删除后一句四元式即可

合并跳转语句

生成四元式的过程中，在处理短路求值时会产生很多无用的跳转语句（如向下一句跳转的跳转语句）

遍历并删除那些无用的语句，既减少了周期，又扩大了基本块，优化效果更好

如生成if或while语句时，常常会有如下情况：

```
%t2 = a > b
beqz %t2 label1
bnez %t2 label2
label1:
//...
```

提取其规律可以发现——第2行的beqz和第3行的bnez条件互补，而且均需要对同一个变量%t2做判断，故可以推断若第2行的语句会做跳转，则第3行不会跳转，程序也将进入第4行

因此，第2行的跳转为无用跳转

优化方法为：

即可扫描四元式，检查相邻的条件互补的跳转语句，同时检查前一条跳转语句是否会跳转到后一条跳转语句之后，找到则删除前一句跳转即可

数据流分析——常量传播与复写传播

这两种优化均为顺序扫描数据流的优化，可以一并做

针对:常量传播：

```
a = 2
c = a * 4
//可优化为
a = 2
c = 2 * 4
```

以及复写传播：

```
a = b
c = a * d
//可优化为
a = b;
c = b * d
```

要分析这样的情景，需要记录变量-常量 与 变量-变量两种映射

通过数据流分析方法，使用Map数据结构实现：

- 对每个过程块（不跨过程块分析）
  - 从第一个基本块开始，其in集初始化为空
  - 对当前基本块，映射Map初始化为当前块的in集合
    - 在基本块内部从上到下顺序扫描四元式
    - 若为常量赋值`a=2`或复写赋值`a=b`，则删除`a`原有的映射，并在Map中添加新的变量-常量 以及变量-变量映射
    - 若为其他赋值型或是被getint赋值、被函数返回值赋值，则删除被赋值变量的所有映射
    - 扫描到基本块底，剩余的映射即为本基本块的out集合
  - 之后沿流图传播，每个基本块的out为所有前驱in集合的交集，再分析后继基本块，遍历流图
  - 重复上述操作（重新从第一个基本块开始遍历，但是不要初始化in集合），直至集合不减小（通过HashMap的equals方法即可判断in集合有无改变）

即可实现

此外，传播过程中可以顺便把所有常量运算一并在编译阶段做完，将3型赋值四元式转换为2型，可减小运算量

如

```
a = 2
c = a * 4
//可优化为
a = 2
c = 2 * 4
//进一步优化为
a = 2
c = 8
```

虽然这些优化并没有实际上消除四元式，但是实际上优化之后的代码会剩余很多死代码，通过下一步死代码删除即可删除很多冗余

数据流分析——活跃变量分析与死代码删除

逆向分析数据流，获取活跃变量相关信息

同时删除不活跃的变量赋值语句

如对于

```
a = b
c = b * d
//先分析活跃变量
//若之后a不再活跃
//优化为
c = b * d
```

活跃变量集通过Set结构即可保存

先做全局活跃变量分析：

- 先获取全局变量集，要保守处理之（认为其永远活跃）
- 对每个过程块
  - 开始时：从exit基本块开始，其out集初始化为全局变量集
  - 当前活跃变量集初始化为当前块的out集合
    - 在基本块内部反向扫描
      - 若某变量被赋值，且其不为全局变量，则将其从活跃变量集中删去
      - 若某变量的值被用到（作为赋值变量或是用于计算等），如`a = b`中的`b`，则将`b`加入活跃变量集；
    - 扫描到基本块顶，剩余的集合即为out集合
  - 之后沿流图传播，当前基本块in集合为其所有后继的out集合的并集，遍历流图
  - 重复上述操作（重复从exit块开始遍历流图，但不再初始化），直至集合不变化（通过HashSet的equals方法即可判断有无改变）

最终得到了每个基本块前(in集)后(out集)的活跃变量集

再做局部死代码删除

死代码删除：

- （无需遍历流图）对每个基本块

    - 获取其out集合，作为初始的活跃变量集
    - 在基本块内部反向扫描
        - 若某变量被赋值，且不为全局变量，
            - 若其在活跃变量集中，则从活跃变量集中删去之
            - 若不在，则删去该条四元式，并不再将用到的变量加入活跃变量集
        - 若某变量的值被用到，则将其加入活跃变量集；
    - 直至遍历完基本块到顶部即完成死代码删除

结合了常量传播和复写传播优化，死代码删除可以有效地删除很多无用四元式

体系结构有关的优化

临时寄存器分配

采用寄存器池的方法进行临时寄存器的分配，若池满则采用OPT算法选取下一次使用最远的变量，溢出到栈中，腾出寄存器以供使用

具体为：

- 在目标代码生成阶段
    - 对基本块内的所有四元式语句（除Label型外）
        - 先找到当前四元式语句用到的变量，申请临时寄存器并分配，并将值从内存中写入寄存器
        - 再找到当前四元式要赋值的变量，申请临时寄存器并分配，将值写入寄存器，并标记该变量为“脏”
        - 若寄存器不足以分配，则向后扫描四元式，选取最远被使用的变量（或该块中语句之后不再活跃的变量）溢出（若脏则写一次内存，不脏就不必写了）（即OPT算法）
        - 每当遇到函数调用，则将寄存器池的“脏”变量全部溢出，函数返回后再全部写回（包括不“脏”的变量）
        - 若基本块结束，则将“脏”并且后续活跃的变量（即处于out集合中的变量）全部写入内存（全局变量仍然保守处理）

这里没有区分全局变量（跨基本块）和局部变量，主要是时间关系不足以实现全局寄存器分配，故以此策略作弥补，虽然效果不是很理想，但是相比于优化前各种lw sw还是优化了不少

常量乘除优化

主要是变量乘以常量和变量除以常量的优化：

- 乘法(3cycles)
    - 若一个乘数为常量，讨论其性质
        - 若其为2的整数次幂((a & (a-1)) == 0)
            - 转化为位移运算（1cycle)
        - 若其+1或-1为2的整数次幂
            - 转化为位移运算和一次加减法运算(2cycles)
    - 其他则可能产生负优化(≥3cycles)
        - 如将a*10 优化为 (a<<3) + (a<<1)，本身也需要3个周期，零优化，不需要做
- 除法(100cycles)
    - 除数为常量
    - 参考论文，运算可以削弱为几次位移运算和一次乘法(~10cycles)，周期数减少一个数量级
- 取模(100cycles)

- 除数为常量
- a%b=a-a/b*b，周期数也将减少一个数量级（使用优化除法）

窥孔优化（后端）

连续的访存指令

如

```
lw $t0, 8($sp)
sw $t0, 8($sp)
```

等连续的lw sw指令，若其寄存器与地址相同，则删除后一句即可

顺序遍历目标代码，扫描其是否存在这种优化情景，有则删除后一句即可

连续的move指令

如

```
move $t0, $t1
move $t1, $t0
```

等做了临时寄存器分配之后就可能出现的优化情景

检查连续的move指令其源寄存器和目标寄存器是否相反，是则删除后一句move即可

结语

总之，虽然优化竞速还有所遗憾，整体上来说，编译课设还是让我收获颇丰。主要有一点，也算是老生常谈，那就是——架构设计很重要！以后越大的项目越要重视架构设计，有了架构设计才能更快更高效地完成任务，而且后续的重构也会更不必要，需要做也可以做得更顺畅。若上手就开写，不做任何设计，轻则效率低下，重则必须反复重构才能实现功能。

另外，实际上编译课设给我们自由设计的空间是非常大的，没有规定统一的中间代码范式、甚至没有规定地址空间。所以我们在实现自己的编译器时也可以自由一些，不必要拘泥于课本或是其他工程级编译器的设计，在自己能设计的基础上做到最优化就是最好的了。