实验报告

PA1

1. 源代码结构

目的是修改cool.flex，构建一个针对Cool语言的词法分析器。其中，主要需要构建flex文件的rules部分，这一部分定义了**匹配相应的正则表达式后执行的动作**。

大体上，我将rules部分从上至下分为如下几个部分：

1. 嵌套注释(Nested Comments)
2. 单行注释(Inline Comments)
3. 除了true和false的关键词(Keywords)
4. 字符串(String Constant)
5. 整数(Integer)
6. 布尔值，即true和false(Boolean)
7. 对象标识符(Object ID)和类型标识符(Type ID)
8. 换行和空白符(New Line and Blank Characters)
9. 多字符操作符(Multi-Character Operator)和单字符操作符(Single-Character Operator)
10. 其他字符，即(一类)错误(Error)

下面，在第二部分，我会先叙述flex是如何分析输入文本的，并从上至下详细叙述每条规则的细节构成，并在第三部分谈到一些细节和问题，以及相应的解决思路。

2. 核心代码工作原理

2.1 flex匹配原则

根据flex-manual 7.How the Input Is Matched，flex会从头开始**寻找与任一规则定义的正则表达式所匹配的字符串**。当发现一个匹配的字符串时，将该字符串(称作**token**)的值写到变量`yytext`中，然后flex会**执行该规则对应的C代码**。之后flex将`yytext`清空，并从剩下的输入开始继续寻找下一个匹配的token。

这里提到了两种特殊情况：

1. 如果某个字符串可以匹配多条规则，则匹配**最长的那条规则**；
2. 如果某个字符串可以匹配多条同样长的规则，则匹配**最先出现的那条规则**。

通过这种方式，flex遍历完所有的输入，并给出每个token所对应的token class，完成词法分析。

此外，flex还提供了start condition，作为条件匹配的一种有效方式。只有在该condition下，flex才能匹配该start condition下的规则。

2.2 规则细节

2.2.1 嵌套注释

包含在$(*\ *)$中的内容是嵌套注释。因此，当发现一个"$*($"时，应该进入相应的**start condition: <ncomments>**，表示接下来分析的内容都属于嵌套注释。

对于注释，我们不应该将其识别为任何token class，所以对于注释的任何内容，都与之匹配并不做任何动作(见2.1 flex匹配原则，就算**不做任何动作我们也需要将其匹配,保证匹配的正确性**)。

当我们碰到"$*)$"时，我们**不能直接退出ncomments条件**，因为**嵌套注释可以嵌套**，例如如下字符串(蓝色表示注释部分)：

$$(*aaa(*bbb*)*)*)$$

如果我们直接在碰到第一个"$*)$"就退出的话，那么上述字符串应该显示为：

$$(*aaa(*bbb*)*)*)$$

在实际的cool文件中，显示的效果为前者。因此我们必须处理嵌套注释的嵌套性。方法是在flex文件的Declarations部分添加一个全局变量`int nestDepth = 0`，记录嵌套的层数。每遇到一次"$(*$"，我们将`nestDepth`递增1；每遇到一次"$*)$"，我们将`nestDepth`递减1。当处于<ncomments>下nestDepth为0时，退出<ncomments>表示嵌套注释结束。

```
<INITIAL,ncomments>"(*"         {++nestDepth;BEGIN(ncomments);}
<ncomments>.                    {}
<ncomments>"*)"                 {
--nestDepth;
    if(nestDepth == 0)
        BEGIN(INITIAL);
}
```

此外，我们还需要保持变量`curr_lineno`的值。它是一个全局变量，使词法分析器能正确输入每个token class对应的行号。当在嵌套注释中发现一个换行符"$\backslash n$"，则我们应该将`curr_lineno`加1。

```
<ncomments>"\n"                 {++curr_lineno;}
```

需要说明的是，嵌套注释中有一些是属于错误的情况，我将其总结在了2.2.10中。后面的错误情况也是如此。

2.2.2 单行注释

以－－开头的整行内容都作为单行注释。其匹配思路与嵌套注释类似，且因为不用考虑嵌套问题更简单。我们只需要在发现－－时进入对应的start condition <icomments>，然后当发现换行符时，将`curr_lineno`加1并退出<icomments>即可。对于其他字符的处理同样是匹配但什么都不做。

```
"--"                {BEGIN(icomments);}
<icomments>"\n"     {BEGIN(INITIAL);++curr_lineno;}
<icomments>.        {}
```

2.2.3 关键词

在cool-manual 10.4中给出了Cool中所有的关键词，并指出，除了true和false的关键词都是**大小写不敏感的**。在flex-manual 6.Patterns中提到如何进行大小写不敏感的匹配：使用$(?\mathbf{i}:\mathbf{pattern})$的方式，其中$?\mathbf{i}$表示大小写不敏感。

因此，对于所有这些除了true和false的关键词我们都可以如此表示。例如：

```
(?i:class)      {return (CLASS);}
```

对应要返回的token class可以在`cool-parse.h`中查看。

2.2.4 字符串

字符串是在一对双引号("...")之内的内容。因此当我们碰到一个 " 时，进入对应的start condition <string>。当碰到第二个 " 时，就退出。

这里有四个注意点：

1. 在PA1 Handout 5. 和cool-tour 3. 中提到，对于**字符串、整数、标识符、布尔值和错误**，由于它们有具体的内容(值)，所以还需要将其记录下来。我们需要将其记录到对应的变量中，如string应该记录到`stringtable`中。
2. 不像其他类型的token可以直接将`yytext`记录下来，对于string，由于**每匹配一次`yytext`就会清空**，因此如果我们要将字符串完整的内容记录到`stringtable`中，我们**必须用一个其他变量保存字符串的内容**。事实上，`cool.flex`中已经提供了一个字符数组`string_buf`用于记录字符串。但这里我选择在Declarations中定义了一个C++中的string `buffer`，因为string类型提供的各种操作会方便一些。

3. 对于转义字符的处理：flex-manual 6.Patterns中指出，可以用”\X”的表达式来表示转义字符。因此，如果要表示如”\n”的形式，要用”\\n”来匹配。另外，我们还需要为这些转移字符做一些修正，比如”\\n”对应加入到buffer的内容应该是”\n”，而不是`yytext`的”\\n”。
4. 对于字符串内换行的情况：我们需要能正确识别

```
"Hello\
World!"
```

这种字符串，因此需要设定相应的规则与之匹配。相应的匹配规则是”\\\n”，前两个字符表示一个斜杠，后两个字符表示换行符。将换行符记到buffer中，并将`curr_lineno`加1。

```
"\""                    {BEGIN(string);buffer="";}
<string>"\""            {
    cool_yylval.symbol = stringtable.add_string((char*)buffer.c_str());
    BEGIN(INITIAL);
    return (STR_CONST);
}
<string>"\\n"         {buffer += "\n";}
<string>"\\t"         {buffer += "\t";}
<string>"\\b"         {buffer += "\b";}
<string>"\\f"         {buffer += "\f";}
<string>"\\\n"        {buffer += "\n"; ++curr_lineno;}
<string>\\.           {buffer += yytext[1];}            // 注意这个不能加双引号...识别不了通配符.的
<string>.             {buffer += yytext;}
```

2.2.5 整数

整数是**由0-9构成的非空字符串**。因此可以表示为：

```
[0-9]+          {
    cool_yylval.symbol = inttable.add_string(yytext);
    return (INT_CONST);
}
```

2.2.6 布尔值

布尔值包括true和false。它们也是关键词，但我们应该返回它们为`BOOLEAN_CONST`，即：

```
t(?i:rue)       {cool_yylval.boolean = 1; return (BOOL_CONST);}
f(?i:alse)      {cool_yylval.boolean = 0; return (BOOL_CONST);}
```

这里需要注意，在cool-manual 10.4中提到，true和false的**首字母必须是小写**，剩余的字符大小写无关。

2.2.7 对象标识符和类型标识符

cool-manual 10.1中定义标识符为**除了关键词外，由字母、数字和下划线构成的字符串**。细分可以将标识符分为两类：对象标识符，其**必须以小写字母开头**；类型标识符，其**必须以大写字母开头**。

```
[a-z][a-zA-z0-9_]*    {
    cool_yylval.symbol = idtable.add_string(yytext);
    return (OBJECTID);
}

[A-Z][a-zA-z0-9_]*    {
    cool_yylval.symbol = idtable.add_string(yytext);
    return (TYPEID);
}
```

2.2.8 换行和空白符

换行符其实包含在空白符中。cool-manual 10.5中给出的空白符有: ''、'\n'、'\f'、'\r'、'\t'、'\v'。对于换行符将`curr_lineno`加1，匹配其余字符则什么都不做。

```
"\n"        {++curr_lineno;}
[ \f\r\t\v]  {}
```

2.2.9 操作符

操作符可以分为两类：一种是多字符操作符，Cool中有3个：=>(右箭头，在case stmt中用到)、<-(赋值)、<=(小于等于)。对于多字符操作符，返回对应定义的token class(见`cool-parse.h`)。例如:

```
=>          {return (DARROW);}
```

对于单字符操作符，直接返回对应的ASCII值。我们可以利用C语言的特性，直接返回对应的字符。例如:

```
"+"       {return '+';}
```

对于所有可能的单字符，可以参考`utilities.cc`的`cool_token_to_string()`函数给出的case。这个函数将token class转换为可以输出的字符串形式，所以我们可以看对应的case知道我们在flex文件中应该返回什么形式。

2.2.10 错误

错误大体上分为以下五类：(见PA1 Handout 4.1)

1. 遇到非法字符时(无法构成任何token的字符，如[、]等)，应该将其**识别为错误，并考察下一个字符**。对于这种情况，我们只需要在最后用通配符匹配与上面所有规则都不匹配的字符即可。

```
.                   {cool_yylval.error_msg = yytext;return (ERROR);}
```

2. 如果字符串中有一个未转义的换行，例如：

```
"Hello
World"         // This is Error!
```

并继续考察下一行。我们只需要在<string>下让单独匹配一个”\n”的规则作为错误即可。

```
<string>"\n"         {
    BEGIN(INITIAL);
    cool_yylval.error_msg = "Unterminated string constant";
    ++curr_lineno;
    return (ERROR);
}
```

3. 如果字符串过长，或字符串包含空字符('\0')。此时应该在字符串结束后重新开始分析。(即第二个"之后，或是遇到一个未转义换行后的下一行)。对于这两种情况主要是要**准确描述什么时候重新开始**。显然**不应该检测到过长的字符串或包含空字符就直接报错**，因为这样的话接下来还是分析(字符串的)下一个字符，与要求不符。

事实上，我们应该用变量记录是否遇到空字符或是否超出长度限制，**并进入新的start**

condition<stringError>(专门设置新的start condition的理由见3.3 一些细节)。当在<stringError>下遇见第二个”或换行时，就结束并检查错误类型。相应地，在该start condition下也要匹配(正常)换行符和其他字符。另外如果以换行结束，还要计入行号。

```
<stringError>"\""|"\n"    {
  if(yytext[0]=='\n')
    ++errorLine;
  switch(whatError){
        case tooLong:
        cool_yylval.error_msg = "String constant too long";
        BEGIN(INITIAL);
        return (ERROR);
        case nullCharacter:
        cool_yylval.error_msg = "String contains null character.";
        BEGIN(INITIAL);
        return (ERROR);
    }
}
<stringError>"\\\n"     {++errorLine;}
<stringError>.          {}
```

4. 如果注释或字符串中遇到EOF。例如对字符串中EOF处理如下：

```
<string><<EOF>> {
    cool_yylval.error_msg = "EOF in string constant";
    BEGIN(INITIAL);
    return (ERROR);
}
```

对注释中的实现是类似的。
5. 如果在注释外碰到*)。

```
"*)"    {yylval.error_msg = "Unmatched *)";return (ERROR);}
```

3. 遇到的问题与解决思路

3.1 无法匹配的'\n'

最开始我发现始终无法匹配test.cl中第17行的"\n"，每次都提示”Unterminated string constant”。我一直无法理解为什么规则"\n"与字符串"\n"匹配了，因为按理来说规则"\n"应该匹配的是换行符。后来我发现问题其实在于**优先级问题**：

```
<string>.         {...}
<string>"\""      {...}
```

我将结束字符串的规则放在了最后，而按照flex的匹配规则，**同长度下匹配更先出现的规则**。因此实际上字符串的第二个引号**实际上被通配符匹配了，作为了字符串的一部分**。从而flex会继续考察字符串外的内容，直到换行（这也就是为什么会匹配到一个换行符）

解决方法很简单，调换一下两个规则的顺序就可以了。

3.2 关于优先级

首先是类型之间的优先级，我是参考的`cool-parse.h`中给出的优先级。我的个人理解如下：

1. 注释和字符串由于有start condition保护，因此没有明显的优先级。但对于进入对应start condition的规则需要有优先级，(* 、--、" 这三个符号至少需要高于错误的优先级（不符合标识符、整数等的规则）
2. 关键词的优先级高于标识符，否则，由于关键词也符合标识符的规则，会被错误识别。
3. 非法字符(错误)的优先级必须是最低的，否则通配符会匹配其他字符。

另外，在每一start condition内部也有相应的优先级。比如4.1中提到的问题。总结下来原则就是：如果要使用通配符，**通配符一定在该start condition的最低优先级**。这一点很重要。至于其他内部规则的优先级一般是无所谓的，因为匹配项一般很难重叠且长度相同。

3.3 一些细节

1. 在字符串一节提到了字符串过长和含有空字符的问题。那么，如果某个字符串同时出现这两个错误，应该报告哪个错误？经过与官方lexer对比和研究，结果是**只报告第一个出现的错误**。因此我们只需要用`whatError`记录第一个出现的错误类型即可。

2. 测试时发现了一种特殊情况：下划线后接空字符(”\\\0”)会报告为特殊的”String contains escaped null character.”。
3. 测试字符串过长的边界时，发现我的最大长度比官方大1，我最初以为是因为**我是先测试是否超长再添加字符到buffer的**。但测试时发现有一种特殊情况：**第1026个字符是空字符或非转义换行**。此时报告不是超长而是对应的错误。我猜测实现是这样的：**仍然是先测试是否超长再添加字符，但在字符串结束时要检查是否超长**。经测试，此时表现和官方一致。
4. 当在<stringError>下出现EOF时，也应该算作结束，并报告**之前发现的错误**(这个好像manual里也没说)。
5. 测试字符串过长错误时，发现**官方lexer报告错误出错行在刚好超长度的那一行**；空字符错误也是报告在**出现空字符的那行**。这应该属于额外的实现，因为本次PA的Handout里没有提到这个要求。

如果要实现这个特性，我们需要设置一个专门的start condition<stringError>，表示字符串已经进入错误状态，不需要计入后面的内容。(这和只报告第一个出现的错误的特性是相匹配的)。然后在**发生错误时用一个临时变量记录发送错误的行号** `errorLine` (因为报告错误的行号用的都是 `curr_lineno`)。在错误状态下我们**将换行符增加的行数都记到** `errorLine` **上**，并在报告完后再恢复 `curr_lineno` 。

问题是要如何恢复行号？因为在 `return (ERROR)` 后面的内容就不再执行，所以需要在别的地方恢复行号。我的做法比较笨，**在所有结束可能匹配的规则(即所有INITIAL条件下的规则)里检查** `errorLine` **是否大于** `curr_lineno` **，如果大于则恢复**。这样也满足了**在字符串结束后或非转义换行后开始检查的要求**。(应该有更好更优雅的方法，但我暂时没想到...)

4. 感想

本次实验基于flex实现了一个词法分析器，加深了对编译中词法分析阶段的理解。
总的来说，本次实验中遇到的问题还是比较多的：

- 优先级问题。由于我偏向于使用通配符匹配错误或普通字符，但由于放错了位置导致出现了错误的结果且很难发现这种错误。
- 字符串的转义符号处理。用户输入的换行、用户输入的"\n"、与它们分别匹配的文本、记录到buffer的文本，这些概念是比较绕的，让我十分苦恼。
- 很多细节上的问题，如3.3中所列举的。

为解决这些问题提供最大帮助的是各种手册，包括handout、cool-manual、cool-tour等。其中帮助最大的是flex-manual，从flex构建词法分析器的过程、原理，到匹配的各种pattern、start condition的使用方法，各个部分都给了我非常大的帮助。比如我最开始的时候发现flex文件一直编译失败，后来在manual 5.4中发现，**rules部分的注释不能出现在一行的开头，必须空一格，否则会被认为也是一条rule**。这让我深刻理解到手册的重要性。

另外，官方提供了一个lexer，与其分析得到的结果相对比，也很好地帮助我不断构建并完善我自己的

lexer。尤其是很多细节上的东西Handout和manual里没有提到，需要和官方lexer对比才能得出具体实现方式。我们自己也可以构造一些刁钻的测试集，比如：

- 错误出现的优先级：输入过长文本后接空字符，和空字符后过长文本对比。
- 错误行号：测试空字符和过长文本错误就能发现。
- 错误后非转义换行：输入过长文本或空字符后接一个非转移换行以测试。
- 过长错误：具体有多长？是否跟官方是一样的？
- 空字符、非转义换行符出现在恰好过长位置应该是什么表现？
- 如果在字符串已经错误的情况下，出现EOF是什么表现？