

完成时间: 9 月 21 日, 星期三

此项目包含两部分: 词法分析和语法分析。

词法分析

学生的词法分析器必须要能够识别 Decaf 语言中的字符数组, Decaf 语言是一种在课程 6.035 中需要编译的简单强制性语言,。在分发资料 5 中对该语言有所描述。词法分析器需要能够识别非法字符、缺失引号以及其他的词法错误, 并且要有合理且准确的错误信息。词法分析器需要能够发现尽量多的词法错误, 并且在错误被找出以后还能继续查找。词法分析器还需要能够过滤掉注释和空格。

学生无需从最开始编写词法分析器, 而是使用 JLex 来生成词法分析器, JLex 是普林斯顿大学的一个词法分析生成器。该程序可以阅读常规表达的输入规格, 然后生成一个 Java 程序来对指定语言进行词法分析。更多 JLex 相关信息(包括使用手册和示例)请访问下面网站:

<http://www.cs.princeton.edu/~appel/modern/java/JLex/>

为了让学生能尽快上手, 我们在课程服务器上提供了模板。

将这个文件拷贝至学生工作目录下, 然后用适当的常规表达和行为进行替换。这个模板里包含了一些示例命令和宏, 但是学生可能需要阅读 JLex 手册获得更多的信息才能创建完整的规范。为了能够在此文件上运行 JLex, 需要将 CLASSPATH 进行如分发资料 4 中所说明的进行设置, 并且执行下面的命令:

```
java JLex.Main scanner.lex
```

这样会产生一个名为 scanner.lex.java 的文件。请注意, JLex 仅仅是生成一个词法分析器的 Java 源程序; 它并不进行编译, 更不会检查输出的句法。因而, 任何 lex 文件中的排印错误和句法错误都会复制到输出中。

在项目的语法分析部分, 我们会使用 CUP 语法分析生成器(稍后讲述)。语法分析器预计的返回字符数组类型为 java_cup 运行时间标志。标志类提供了字符数组的基本抽象表达。它包含了有关字符数组的四个基本区域: 这些值以这个顺序传递到标志的构造器。如果学生有兴趣了解标志的类定义的话, 课程服务器上有 CUP 源文件的拷贝。

sym	符号标志符 (int 类型)
left	输入原始文件左部
right	输入原始文件右部
value	词法值 (Object 类型)

每个 Decaf 语法可识别的终端都必须有与之联系的唯一的整数, 这样语法分析器就可以识别他们了。这些值存储在标志中的 sym 区域内。当产生新的符号时, 可以使用 decafCUP 规范

(稍后详述) 自动生成的符号值。

一些字符数组也有与他们想联系的值。例子包括有整数和字符串。这些都存储在标志中的值区域。注意这个值的类型是 `java.lang.Object`，所以可以在任何类型的对象中存储。在项目的这个部分，我们只会简单的使用它来存储一些字符数组的字符串属性，在以后会用它来存储更加复杂的信息。

注意，标志类并不包含有关字符数组行数的任何信息，而是存储字符数组在输入文件的绝对位置值。有两个方法可以计算出包含行数的错误信息。这些方法在 Appel 的教材中有描述，主要是定义一个全局错误信息目标文件，这样就可以存储输入文件有错的行数。这也可以用于计算给定字符数组的行列定位位置。学生可能会觉得这种方法不是很好，另外的一种方法标志类的子集来存储有关这个字符数组更多的信息（如行数）。这两种方法在评分上都是一样的。

语法分析器

学生的词法分析器要能够遵从 Decaf 语言的语法来判断程序的词法错误。任何与语言语法不符合的语法都需要至少标注一条错误信息。

如同之前提到的一样，我们会使用 CUP LALR 语法生成器。CUP 的文档在下面的网址可以获得：

<http://www.cs.princeton.edu/~appel/modern/java/CUP/>

CUP 的调用方法如下：

```
java java_cup.Main < parser.cup
```

从 `parser.cup` 规范看来，这样会生成两个 Java 源文件。语法分析器代码包含在 `parser.java` 中，而符号常数代码包含在 `sym.java` 中。而 `-parser` 和 `-symbols` 标志则可以不用考虑这些。在 `sym.java` 中包含有语法分析器定义的每个终端和非终端整数。等到词法分析器返回字符数组后，这些数字就可以使用了。同时使用 JLex 和 CUP 的例子提供在课程服务器上。

学生需要将分发资料 5 中的相关语法（和有限规则）翻译成 CUP 的 BNF 类似输入语法表达。LALR (1) 语法是很合适的，这样 CUP 执行起来就不会有冲突。

学生的语法分析器需要能够使用 CUP 中的错误符号增加附加的纠错功能，以完成基本句法错误恢复功能。不要尝试任何和上下文有关的错误恢复。做一些简单的工作就好了。我们只希望能够完成如恢复缺失的右括弧或缺失的分号这样的错误，不期望更高的纠错功能。学生的编译器不必，或者说不应该，识别上下文相关错误，如在应该使用数组标志符的地方使用了整数标志符。这样的错误应该由静态予以检测器进行检查。不要在语法分析器中做检测上下文相关错误。

学生可能需要查阅“虎书”的第三节，或“龙书”的 4.3 节和 4.8 节，获取一些提示来解决语法的转换/降低和降低/降低冲突。另外，我们已经修改了 CUP，可以提供一个字符串，`PRODSTRING` 代表当前降低。学生会在调试过程中发现这是很有用的。

提交

在详细描写项目的硬拷贝文件时，请遵守分发资料 3 中的说明。要包含语法分析器架构的详尽描述。

在提交的电子部分，需要在小组文件放置位置里提供名为 `leNN-parser.tar.gz` 的压缩文件，NN 为小组编号。这个文件需要包含所有相关的源代码和 `makefile`。同时还需要在相同路径下提供使用 `java` 工具生成的，名为 `leNN-parser.jar` 的 Java 文档。解压 `tar` 文件运行 `make` 应该要能够产生同样的 Java 文档。（所有以后提交的都大体相似，只是文件名会有所不同。）

课程服务器上提供了一个 `makefile` 例子。

学生要把这个文件拷贝到工作目录下，这个文件会有助于生成正确的 `tar` 和 `jar` 文件。另外，学生还要生成一个驱动文件来运行词法分析器和语法分析器。驱动名为 `Compiler.class`，位置在课程的根目录下，所以可以用下面的命令运行：

```
java Compiler <输入文件>
```

课程服务器上提供由示例驱动。

驱动必须要能够对命令行进行词法分析，并且根据需求来运行语法分析器或隔离词法分析器。我们希望学生使用我们提供的 CLI 工具，而不是自己编写命令行语法分析器。CLI 的文档在课程 6.035 主页上有链接。

词法分析器格式：当 `-target` 词法分析指定以后，编译器的输出文件应该是一个表格，表格中对于一个输入文件的字符数组对应一行。每一个包含三栏：字符数组出现的行号（以 1 开始计数），字符数组类型和字符数组的值。

简单字符数组的种类（如关键字或操作数）就是该字符数组本身，如：`return` 或 `>=`。有值的字符数组为后面所列中的一个：字符、整数、布尔、字符串、标志符。

对于整数而言，值是在源程序中的一系列数字（见附录解释）。对于布尔类型，值为真或假。对于字符串和字符类型，值为字符序列转化为内码后的值。对于简单的字符数组，值的一栏为左边空白。注意字符和字符串可能包含新的一行中的字符。如果它们也如同描述的打印出来，可能会打乱表的格式——这是可以的。

如果有的话，每条错误消息需要打印在错误所在行，且要在字符数组之前。这样的信息格式如：`<文件名>: <行号>: <信息>`

下面是打印("hello,world!");的示例表格：

行号	类型	值
2	标志符	Print
2	(
2	字符串	Hello,World!
2)	
2	;	

语法分析格式：当 `-target` 词法分析指定以后，任何句法错误程序都应该至少有一条错误消息标注，这时程序应该以一个非零值退出（见 `System.exit()`）。对于可以进行错误恢复的多个句法错误要求有多条错误信息（如，缺失右括弧或者一个缺失的分号）。对于一个没有句法错误的程序，编译器应该没有输出，并且以零值退出（成功）。

测试

服务器上提供有测试词法分析和语法分析的测试案例。

我们会在这些案例上测试学生的词法分析器/语法分析器，并且还要进行一些类隐藏测试。学生的分数由成功通过多少这些测试决定。不过要想获得满分，我们希望学生也能完成分发资料 3 中所述的项目书写部分。

A 为什么我们把整数边界检测推迟到下一个项目中执行。

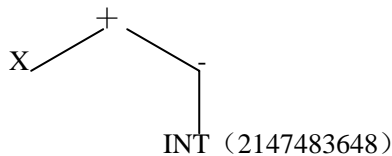
在考虑检测输入程序的合法性问题时，词法分析器、语法分析器和语义检测器之间是没有根本责任界限的。通常情况下，编译器设计者可以选择在哪一个特定阶段检测某个限制，甚至把这个检测放分割在不同阶段进行。但是，为了在整体上更好的符合课程时间安排，我们不得不把完整的词法分析/语法分析/检测分成简单的两部分。

这样做的结果，就是我们必须强制要求一些与编译器实现正确性相关不大的细节。举例来说，学生不能在构建语法树的前提下对整数是否越界进行全面检查。

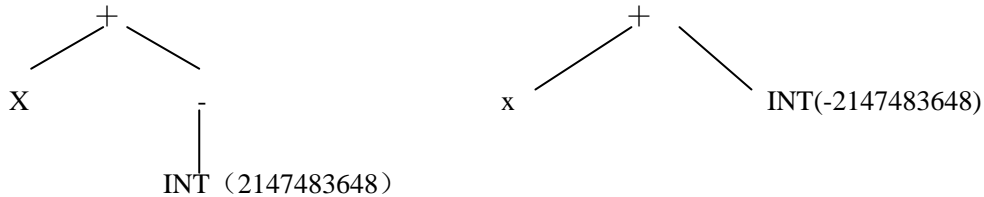
考虑输入为：

x+-2147483648

对应的语法树为：



我们不能在词法分析器中确认整数-214783648 是在范围内的，因为它不是单个字符数组。我们也不能在语法分析阶段确认，因为在这个阶段我们还没有构建器抽象语法树。只有在语义检测阶段，我们已经有了抽象语法树，我们就可以执行检测了，因为这个阶段需要一元负操作数修改它的实参，如果它的实参是如下所示整数的话：



当然，如果整数已经很明显越界了（如 9999999999），那么词法分析器应该要能够检测到，但是这样的检测在这部分是不需要的，因为无论如何，后面语义阶段都会执行这样的检测的。因此，我们决定不把检测一部分放在前面一部分放在后面，而决定将所有的整数边界检测推迟到语义阶段。所以，学生的词法分析/语法分析器务必不要判断在整数字符数组中的十进制或十六进制字符串；字符数组必须要原样保留到语义阶段。

在从词法分析器打印字符数组表时，不要将整数字符数组以十进制形式打印出来。应该将其在源程序中出现过的无论十进制还是十六进制的都原样打印出来。