编译原理：实验一

姓名：王诲喆

学号：131220159

程序功能

程序使用方法 `$ ./parser [-g|greedy] source.cmm`

输出：

1. 如果source.cmm没有词法和语法错误，那么parser会打印符合实验要求的语法树；
2. 如果source.cmm有词法和/或语法错误，那么parser会在（尽可能地）输出错误提示，像实验所要求的那样，词法错误为类型A，语法错误为类型B；
3. 如果在parser和源文件之间加入 `-g` 或 `greedy` 选项，那么parser会尽可能地多报错误，一般会出现同一行重复报错的情况（具体见程序特点）。

注意：

1. 语法树和**错误提示**都打印在标准输出中；
2. source.cmm中不能使用Windows的换行 `CR LF` ，会被识别为词法错误；
3. parser注重对分号丢失时的错误恢复和行号提示进行优化，代价是**极少数情况**下会在上一行（或几行）报错（具体见程序特点）。

编译方法

使用实验默认提供的makefile即可编译，该makefile已经包含在项目目录中，没有经过修改，所以如果上层目录没有Test/test1.cmm的话， `make test` 无效。

需要注意的是yytname.h和lexical.l含有自动生成的代码。在tools目录下有减少重复代码书写的脚本工具。 gen_pattern会根据同目录下的lexical.template填写大部分终结符的匹配行为。 gen_enum会根据上层目录（项目根目录）的syntax.y中终结符和非终结符的声明顺序，生成yytname.h以备他用。脚本执行需要安装Ruby，不需要参数，直接运行。

yytname.h和lexical.l已经事先准备好，如无改动，不需要通过脚本去更新。

程序特点

充分利用Bison生成的c代码

要打印语法树，就需要存储每个结点的符号名字符串，我发现Bison在调试模式下能自己打印出符号名，就猜想它一定自行存储了所有符号的字符串。 通过搜索，我在生成的 syntax.tab.c 里发现了所

有符号的字符串表yytname。 ytname对应字符串的下标刚好可以作为语法结点的类型标志type。而且区分终结符与非终结符非常方便，只需要跟 `YYNTOKENS` 比较就可以了。

不过不像终结符，Bison并没有给非终结符准备一套 enum，而且生成的终结符的enum并不能作为下标（偏移量是一致的……），所以还需要我自己来准备一套enum。 yytname中符号字符串的出现顺序总是和 syntax.y 中终结符和非终结符作为产生式头时的出现顺序是一致的， 然后头三个是 `$end` , `$error` , `$undefined` , 终结符和非终结符之间插入 `$accept` ，这四个也算在 `YYNTOKENS` 里。所以我写了一个脚本读取syntax.y，读取终结符和非终结符，并生成yytname.h。 现在每发现一个结点，我只需要赋值对应的枚举值，然后遍历的时候，只需要通过这个枚举值就可以找到结点的名字了。

在syntax.y里书写语义行为时，由于现在只需要链接全部结点，操作相对简单重复，完全可以通过循环来处理。但是 `$$` ， `$1` ， `$2` 这样的符号，必须显式地书写， Bison也没有提供基于下标的访问方法和产生式体的符号数量。但是我观察生成的 `yyparse()` 的c代码发现， `$$` 直接替换成了 yyval， `$n` 直接替换成了 `yyvsp[0]` ， `$(n-1)` 替换成了 `yyvsp[-1]` 。语义行为的代码是直接暴露在 `yyparse()` 里的，所以我可以直接访问它的局部变量。 `yyvsp` 是parsing过程的三个栈之一，用来存储语义值(semantic value)，另外两个是 `yyssp` 和 `yylsp` ，分别存储状态和位置信息。

现在对于每个语义行为，我只需要提供产生式头的枚举值和产生式体的符号数量就可以了。具体地，我使用了一个宏（使用函数就访问不了局部变量 `yyvsp` 了）来抽象每次归约过程中局部语法树的建立，先是建立头的结点，接着将体的第一个结点链接到 `child` 域， 然后将剩下的结点挨个链接到前一个结点的 `sibling` 域上。对于能归约成ε的产生式，头被赋值成NULL，在上层链接过程中会被跳过以达到实验要求的效果。

这样的方法主要是为了追求建树过程的自动化和语义行为代码（现阶段）的整洁，不过还是有一些潜在的问题：

1. 由于没有出现 `$$` 和 `$n` ，Bison生成c代码时会认为我没有对符号做任何操作，在使用尝试使用 `%destructor` 时出现了大量的warning，最后 `%destructor` 自动free结点会导致段错误，很可能也是由于这个原因。
2. 如果Bison的某个版本把 `yyvsp` 这个栈的名字给换了，这份代码就无法通过编译了。

错误恢复

我花了大量的时间考虑丢失分号时的恢复问题，和组员张帆进交流后发现在产生式末尾加error有比较好的效果。 不过产生式末尾加error有个问题，有些可以通过寻找终结符来恢复的错误会被这一模式抢占，可能会重复触发错误，在使用 `yyerrok` 的情况下就是如此（开启 `-g` 或 `greedy` 选项）。 不过Bison默认的模式是消费掉3个token后才可以报新的错误，所以默认情况下从表面上看起来不一定会出现重复报错的情况。

行号提示

在丢失分号的情况下，parser要向前看一个token，发现无法进一步归约了，才会报错，此时往往报

错在下一行。 gcc就是在下一行报错，但是clang做到了在分号丢失的那一行报错！

我发现，使用局部语法树可以比较好地处理这种情况。具体来说，局部语法树指的是**最近一次**进行归约的产生式头为根的那个语法树。每次归约时，我都将 `$$` 对应的 结点指针指向一个全局变量，这样通过这个全局变量，我就可以访问到最近一次进行归约的语法结构。这样只需要缺少分号的那一行进行过一次归约，错误提示时使用语法树里的行号， 往往比较准确。缺少分号时常常会进行一次归约，因为分号之前的，都是非终结符，向前看到了TYPE或ID这样的符号，会先进行一次归约，比如Exp，然后再向前看，才发现不对。 这时候我就可以访问Exp代表的树，用它的行信息来提示位置。

但是这种手段在在一些个别情况下会导致提示的行号会往上一行甚至多行，原因是出错的行没有进行过归约。

现在的改进的判断方法是，如果是在首行第一个符号处产生了错误，且语法树第二层最右结点不是分号或右花括号，就使用该结点的行号，否则就使用当前行号。一般临时归约的树不会太深，而且二层最右结点往往是终结符，此时可以认为就是当前行的第一个符号有问题。

错误信息使用了 `%error-verbose` ，并且发挥一贯的作风去 syntax.tab.c 里找那些错误信息的定义，这些信息是由 `yyformat` 和 `YYCASE_(N, S)` （开了verbose后的多种输出）决定的，但是这些宏和变量没有办法从syntax.y里进行进一步处理，所以错误提示信息还是没办法很方便地自定义。

一些思考

数据结构

我们是采用子女兄弟树来模拟语法树，只使用一种结点类型。 但是我在GitHub上观察了一些语言的官方的Bison文件， 发现他们还是偏向于为不同类型的符号建立不同的结点类型。 不知道这其中的有怎样的考量？

手写parser与代码生成器

代码生成器虽然方便，但是要想加点什么功能的话还是比较麻烦，错误恢复和错误提示基本上是被牵制住了。 但是看了一些网络上的讨论，似乎这种功能上的不爽快也有一部分是算法的原因。如果是GLR的话，似乎能比LALR(1)更好地处理这些问题？