

preparation

- 上传之后图片无法正常显示，想看图片可以看同一级目录下的文件：record.pdf

评分

1. 下载评分脚本：

```
1 CODE
2 wget
https://courses.edx.org/assets/courseware/v1/2aa4dec0c84ec3a8d
91e0c1d8814452b/asset-
v1:StanfordOnline+SOE.YCSCS1+1T2020+type@asset+block/pa1-
grading.pl
```

编译：

```
1 CODE
2 make lexer
```

测试：

```
1 CODE
2 perl pa1-grading.pl
```

报错处理

如果

```
1 SHELL
2 make lexer
```

产生如下报错：

```
1 CODE
2 /usr/bin/ld: /usr/lib/gcc/x86_64-linux-gnu/9/../../../../x86_64-
linux-gnu/libfl.so: undefined reference to `yylex' collect2:
error: ld returned 1 exit status
```

那么在definitions和rules之间增加如下内容即可：

```
1 CODE
2 %option noyywrap
```

git push前:

```
1 git config --global http.proxy
2 git config --global https.proxy
3 git config --global --unset http.proxy
4 git config --global --unset https.proxy
5
6 (windows需要管理员权限)
7
8 如果上面的方法不行, 可以试下面的方法 (同样需要管理员权限)
9 git config --global http.sslVerify false
10 git config --global https.sslVerify false
```

```
1 git push github_remote thread:thread --force
```

token:

```
1 ghp_McG8oqfkZok41nJUEMaqHkFgRCMObp0y67xr
```

push问题也可以用ssh方式解决:

你现在遇到的问题是SSH公钥认证失败, 表示GitHub没有识别你的公钥, 所以拒绝了你的连接请求。为了解决这个问题, 你需要确保你的本地SSH公钥已经添加到GitHub账号的SSH Key部分。

按照以下步骤操作:

1. 检查你的SSH密钥是否已存在:

在你的终端中输入:

```
1 bashCopy code
2 cat ~/.ssh/id_rsa.pub
```

如果你看到了一长串的密钥, 那么你的SSH公钥已经存在。

如果返回"没有此文件或目录"或类似信息, 你需要生成一个新的SSH密钥:

```
1 bashCopy code
2 ssh-keygen -t rsa -b 4096 -c "your_email@example.com"
```

替换"your_email@example.com"为你在GitHub上注册的邮箱地址。

2. 将SSH公钥添加到GitHub:

如果你在第一步中找到了你的公钥或生成了一个新的, 使用以下命令复制你的公钥:

```
1 bashCopy code  
2 cat ~/.ssh/id_rsa.pub | xclip
```

(注：如果 `xclip` 没有安装，你可以手动打开 `id_rsa.pub` 文件并复制内容。)

- 登录到你的GitHub账户。
- 点击右上角的用户头像，然后选择"Settings"（设置）。
- 在左侧菜单中，选择"SSH and GPG keys"（SSH 和 GPG 密钥）。
- 点击"New SSH key"（新建SSH密钥）按钮。
- 在"Title"字段中为你的密钥起一个名字，例如"Personal Laptop"。
- 在"Key"字段中粘贴你的公钥。
- 最后，点击"Add SSH key"（添加SSH密钥）按钮。

3. 再次尝试推送：

```
1 bashCopy code  
2 git push cs143 master:main
```

如果按照以上步骤操作后仍然遇到问题，请提供更多的背景信息，我会尽量为你提供进一步的帮助。

language specification

使用正则语言和正则表达式来进行language specification

Notation

- Languages are sets of strings.
- Need some notation for specifying which sets we want
- The standard notation for regular languages is regular expressions.



你如何理解下面这段话：The standard notation for regular languages is regular expressions.。请结合实例详细解释

这句话说的是，对于正则语言（Regular Languages），通常使用正则表达式（Regular Expressions）作为标准表示方法。

首先，我们来简单了解一下“正则语言”和“正则表达式”。

1. **正则语言 (Regular Languages)**: 在计算机科学和形式语言理论中，正则语言是一种特定类型的语言，可以由正则表达式描述或由有限自动机接受。正则语言的主要特点是它们的结构相对简单，并且可以通过计算机上的算法进行高效处理。
2. **正则表达式 (Regular Expressions)**: 正则表达式是一种字符串，用来描述一类特定模式的字符串集合。它们通常用于字符串搜索和替换操作，以及数据验证等场景。正则表达式本身有一套特定的语法，可以用来描述复杂的字符串模式。

现在，举一个实例来进一步解释：

假设我们想描述所有由三个字母组成的字符串模式。对应的正则表达式可以是：

```
1 cssCopy code
2 ^[a-zA-Z]{3}$
```

这个正则表达式可以分为以下部分进行解释：

- `^`: 表示字符串的开始。
- `[a-zA-Z]`: 表示一个字符，可以是任意的小写或大写字母。
- `{3}`: 表示前面的模式（即一个字母）重复三次。
- `$`: 表示字符串的结束。

所以，整个正则表达式描述的是：一个恰好由三个字母（不区分大小写）组成的字符串。

这种表达方式使得我们可以非常精确地描述和匹配特定的字符串模式。所以正则表达式成为了描述正则语言的标准表示方法。

对于下面五个规则的介绍，仅仅看课件不够，参阅视频后可以掌握[链接](#)

Regular Expressions

- Def. The regular expressions over Σ are the smallest set of expressions including

ϵ

' c ' where $c \in \Sigma$

$A + B$ where A, B are rexp over Σ

AB " " "

A^* where A is a rexp over Σ

- Concatenation

$$\underline{AB} = \{ \underline{ab} \mid \underline{a} \in A \wedge \underline{b} \in B \}$$

- 为什么integer = digit digit*, 而integer = digit * 是不够的?

因为digit*必然包括digit(0)={""}

language implementation

啊? ? NFA->DFA也行?

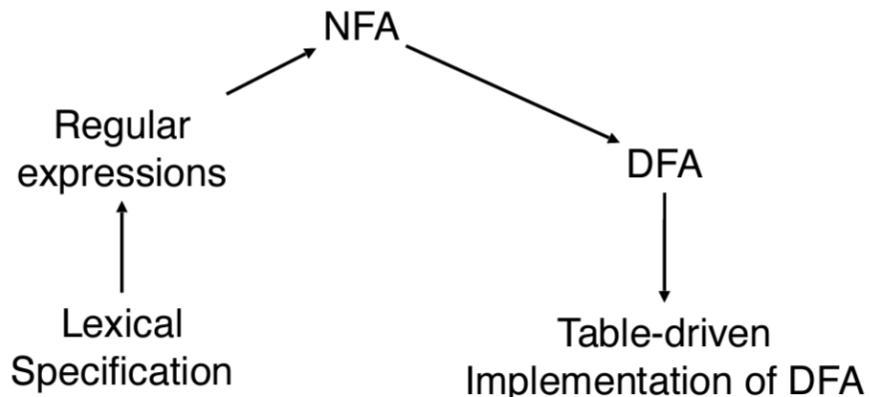
确实是可行的：把NFA的一个epsilon闭包整体视为一个状态，且在此基础上发生的transition都是非epsilon的。而且start state和final state也很好定义。

能看懂下面这张图即可：

NFA		NFA to DFA	
states	S	states	subsets of S except the empty set
start	$s \in S$	start	$\epsilon\text{-clos}(s)$
final	$F \subseteq S$	final	$\{ X \mid X \cap F \neq \emptyset \}$
$a(X) = \{ y \mid x \in X \wedge x \xrightarrow{a} y \}$		$X \xrightarrow{a} Y$ if $Y = \underline{\epsilon\text{-clos}(a(X))}$	
$\epsilon\text{-clos}$			

Convert Regular Expressions to Finite Automata

- High-level sketch



正则表达式的notation略作改变：

Notation

- There is variation in regular expression notation
- Union: $A + B$ $\equiv A \mid B$
- Option: $A + \epsilon$ $\equiv A?$
- Range: $'a' + 'b' + \dots + 'z'$ $\equiv [a-z]$
- Excluded range:
complement of $[a-z] \equiv [\wedge a-z]$

NFA->DFA转换规则

主要是add transition的部分不太好理解，结合具体例子理解。

值得注意得的是，这里的iff的意思是充分必要条件的意思，也就是说，仅仅在这种case下才可以为新建的DFA新增一个transition

NFA to DFA: The Trick

- Simulate the NFA
- Each state of DFA
= a non-empty subset of states of the NFA
- Start state
= the set of NFA states reachable through ϵ -moves from NFA start state
- Add a transition $S \xrightarrow{a} S'$ to DFA iff
 - S' is the set of NFA states reachable from any state in S after seeing the input a , considering ϵ -moves as well

对于这段话中的any我不是很明确，它是否意味着，对于 S' 中的任何一个状态，都可以由 S 中的任何一个状态在input为a的情况下得到？

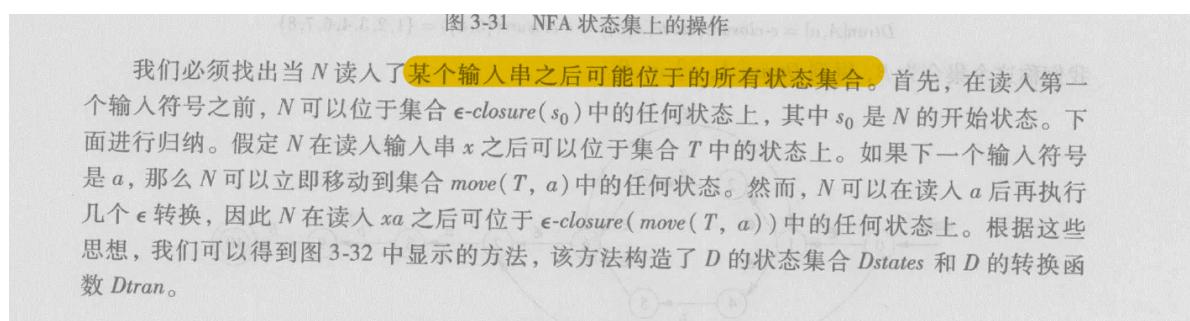
对于你提到的那段话，“any”不意味着“对于 s' 中的任何一个状态，都可以由 s 中的任何一个状态在input为a的情况下得到”。

实际上，它的意思是：“存在一个状态在 s 中，当看到输入 a 时，可以到达 s' 中的某些状态”。

换句话说，只要 s 中的某个状态在输入 a 时可以到达 s' 中的任何状态（考虑到 ϵ -moves），我们就会在DFA中为 s 和 s' 之间添加一个转换。

emmmmm,感觉gpt的解释仍然不是很明确：

还是参考龙书的表述比较准确： S' 是 S 在读入了某个输入串之后可能谓语的所有状态的集合



PA2

首先，需要学习flex的相关知识，以下面的示例程序为例，说明一些注意点：

```

1  /* scanner for a toy Pascal-like language */
2
3  %}
4  /* need this for the call to atof() below */
5  #include <math.h>
6  %}
7
8  DIGIT    [0-9]
9  ID       [a-z][a-zA-Z]*
10
11 %%
12
13 {DIGIT}+   {
14     printf( "An integer: %s (%d)\n", yytext,
15             atoi( yytext ) );
16 }
17
18 {DIGIT}+."{DIGIT}*          {
19     printf( "A float: %s (%g)\n", yytext,
20             atof( yytext ) );
21 }
22
23 if|then|begin|end|procedure|function      {
24     printf( "A keyword: %s\n", yytext );
25 }
26
27 {ID}        printf( "An identifier: %s\n", yytext );
28
29 "+" | "-" | "*" | "/"   printf( "An operator: %s\n", yytext );
30
31 "{"[^{}]\n}*"}"      /* eat up one-line comments */
32
33 [ \t\n]+           /* eat up whitespace */
34
35 .                printf( "Unrecognized character: %s\n", yytext );
36
37 %%
38
39 int main( int argc, char **argv )
40 {
41     ++argv, --argc; /* skip over program name */
42     if ( argc > 0 )
43         yyin = fopen( argv[0], "r" );
44     else
45         yyin = stdin;
46
47     yylex();
48 }
```

- 对于正则表达式的匹配，存在如下规则：

```
"x" : an "x", even if x is an operator.
```

这里其实容易引发歧义，主要是对于operator的理解，这里的operator是指正则表达式中的operator，而不是Expression中的operator，加双引号的意义就在于此。比如+号在正则表达式和算术运算中有着不同的意义

在flex中，当正则表达式用双引号包围时，它通常表示字符串中的字面值。这意味着它会尝试匹配输入中的确切字符序列，而不是正则表达式中可能表示的特殊字符。

因此，当你在flex规则中写：

```
1 flexCopy code
2 "+" { /* action for the + character */ }
```

这意味着它将匹配输入中的实际+字符，而不是正则表达式中的其他特殊意义（例如，+在正则表达式中通常表示“一个或多个前面的元素”）。

因此，对于你的问题："x" an "x", even if x is an operator。这句话的意思是，即使x是一个正则表达式的操作符，当它被双引号包围时，它将被识别为字面字符x而不是操作符。

- 上面的规则定义中，最后有一条：

```
1 .
   printf( "Unrecognized character: %s\n", yytext
 );
```

这里利用到了规则定义中的一个特性，或者说是yylex函数执行时的一个特性：按照从前到后的顺序进行匹配，一旦匹配到符合条件的就停止匹配。也就是说，这些规则的先后顺序本质上就是他们的优先级顺序。

这里我的理解其实不是很准确，因为我只考虑到了顺序原则：

在flex定义的规则中，当输入的文本可以被多个模式匹配时，flex遵循以下的规则：

1. **最长匹配原则**：flex会选择与输入文本匹配最长的那个模式。例如，如果输入是"ab"，而规则中有模式"a"和"ab"，则会选择"ab"这个模式。
2. **顺序原则**：如果有多个模式与输入文本匹配的长度相同，flex会选择在文件中出现最早的那个模式。例如，如果规则文件中先定义了模式"a"，然后是模式"ab"，当输入为"ab"时，尽管两个模式都可以匹配，但flex会选择第一个模式"a"。

所以，在你提到的那条规则中：

```
1 perlCopy code
2 .
   printf( "Unrecognized character: %s\n", yytext );
```

- 这个模式会匹配任何单个字符。因为它是规则文件中的最后一条，所以只有当输入的文本没有被前面的任何模式匹配时，这条规则才会生效。这确保了任何未被其他规则识别的字符都会被这条规则捕获并输出错误消息。这是一个常见的模式，用于报告扫描器中的词法错误。

Definitions Section

The definition can subsequently be referred to using '{name}', which will expand to '(definition)'. For example:

定义：

```
1 DIGIT [0-9]
2 ID      [a-z] [a-zA-Z0-9]*
```

那么在规则部分，

```
1 {DIGIT}+."{DIGIT}*
```

就等价于：

```
1 ([0-9])+."([0-9])*
```

User Code Section

The user code section is simply copied to lex.yy.c verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second '%%' in the input file may be skipped, too.

所以说，对于我自己定义的rule.flex文件，我必须要定义一个main函数，并且在其中调用yylex()，对吗？

ChatGPT

不完全是。你是否需要在 .flex 文件中定义一个 main 函数取决于你的用例和扫描器如何与其他代码协同工作。

- 如果 .flex 文件是独立的：**如果你想要生成一个独立的扫描器程序，那么你确实需要在 .flex 文件中定义一个 main 函数并在其中调用 yylex()。这样，当你生成、编译并运行扫描器时，它会开始读取输入，并按照定义的规则进行匹配。
- 与其他代码集成：**如果你的扫描器是与其他代码（例如 Bison 生成的解析器）一起工作的，那么你可能不需要在 .flex 文件中定义 main 函数。在这种情况下，其他代码可能会负责调用 yylex()。

3. 默认的 `main` 函数: 如果你没有为 `.flex` 文件提供 `main` 函数, `flex` 会为生成的 C 代码提供一个默认的 `main` 函数, 这个函数简单地调用 `yylex()` 并返回。这可以方便地用于测试扫描器。

总之, 你是否需要在 `.flex` 文件中定义一个 `main` 函数取决于你的需求。如果你只是想测试扫描器, 使用默认的 `main` 函数或自己定义一个就足够了。如果你的扫描器是与其他系统集成的一部分, 你可能需要确保适当的代码调用了 `yylex()`。

Patterns

\X'

if X is 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of '\x'. Otherwise, a literal 'X' (used to escape operators such as '*')

实验细节

- 其实它已经帮你做了很多了——先后顺序已经排好了, 这意味着优先级顺序不需要列入我们的考虑范围之内, 我们只需要专注于正则表达式的匹配即可
- 在cool-parse.h中, 搞清楚美剧类型的yytokentype和YYSTYPE之间的关系:
首先给出两者的定义:

```
1 #if ! defined YYSTYPE && ! defined YYSTYPE_IS_DEFINED
2 typedef union YYSTYPE
3 #line 46 "cool.y"
4 {
5     Boolean boolean;
6     Symbol symbol;
7     Program program;
8     Class_ class_;
9     Classes classes;
10    Feature feature;
11    Features features;
12    Formal formal;
13    Formals formals;
14    Case case_;
15    Cases cases;
16    Expression expression;
17    Expressions expressions;
18    char *error_msg;
19 }
20 /* Line 1489 of yacc.c. */
21 #line 124 "cool.tab.h"
22     YYSTYPE;
23 # define yytype YYSTYPE /* obsolescent; will be withdrawn */
24 # define YYSTYPE_IS_DEFINED 1
25 # define YYSTYPE_IS_TRIVIAL 1
```

```
26 #endif  
27
```

- 关于字符串+换行:其匹配规则为:

```
1 /* 换行 */  
2 <STRING>\\n  
3 {  
4     curr_lineno++;  
5     yymore();  
6 }
```

而不是你想当然认为的\n

事实上，匹配到\n是会报错的！

```
1 <STRING>\\n  
2 {  
3     cool_yylval.error_msg = "Unterminated string constant";  
4     curr_lineno++;  
5     BEGIN 0;  
6     return ERROR;  
7 }
```

以c语言举例:

```
3 using namespace std;  
4 int main() {  
5     //keyWordsCount();  
6     //EmailAddressTest();  
7     //IsPalindromesTextTest();  
8     //SmartPointerTest();  
9     MyFunction(0);  
10    char test[100] = "123444\";  
11        只有这样才能继续！！；  
12    }  
13    return 0;  
14 }  
15 //SmartPointerTest();  
16 MyFunction(0);  
17 char test[100] = "123444"  
18        这样就会报错！！；  
19 }
```

- 使用 `BEGIN(STRING);` 切换到 `STRING` 状态，这意味着从此时开始，只有标记为 `<STRING>` 的模式会被考虑匹配。
- 使用 `yymore();`，这个函数是Flex提供的，它的意思是将下一个识别到的文本追加到 `yytext` 中，而不是替换它。
- 整数/标识符
- 这里就涉及你对于string table结构体的master了

对于下面的规则。把标识符添加到stringtable中我可以理解，但是为什么对于简单的证书也要将它添加到相应的inttable中？`: {DIGIT}+`

```

1  {DIGIT}+
2  {
3
4      cool_yylval.symbol =
5      inttable.add_string(yytext);
6
7  [A-Z] [0-9a-zA-Z_]*
8  {
9      cool_yylval.symbol =
10     idtable.add_string(yytext);
11
12     return TYPEID;
13
14 }
15
16 }
```

ChatGPT

在编译器设计中，把数字常量、标识符、类型标识符等加入到各自的表格中的做法是常见的。理由如下：

- 唯一性和重用:** 将词素（如数字、标识符）添加到表中确保了只为源代码中的每个不同的词素保存一个实例。例如，假设源代码中的数字 123 出现了多次，那么 `inttable` 中只存储一次 123。这可以节省内存，并使得之后对这个数字的所有引用都指向同一个地方。
- 高效的相等性检查:** 当从表中检索或比较值时，不需要对整个字符串进行字符逐字符的比较。由于每个词素在表中都是唯一的，因此可以简单地比较其指针或索引。
- 额外的信息存储:** 通常，当向这样的表中添加一个项时，除了字符串本身，还可以关联其他信息。例如，可以为每个项分配一个唯一的索引号，或者关联有关该项的附加数据。
- 代码生成:** 在代码生成阶段，特别是在目标代码或汇编代码的生成中，这些表格为常量提供了方便的管理方式。例如，为每个 `inttable` 中的数字生成全局数据段中的一个项。
- 错误检测和调试:** 当编译器发现一个错误或当你在调试模式下运行编译器时，这些表提供了一种简便的方法来引用、打印或处理词素。

考虑这样一个情境：假设你在编译的源代码中有一个大型数组，其所有元素都被初始化为相同的整数值 12345。如果不使用 `inttable`，则每次出现这个数字时，编译器都必须为其分配新的内存，并重复存储同样的字符序列。但是，使用 `inttable`，这个数字只存储一次，并为所有其他出现重复使用同一个表项。这不仅节省了内存，而且使得在语义分析和代码生成阶段处理这个数字更为高效。

在前面的代码中，解释 `code_string_table` 的作用？

ChatGPT

在给出的代码中，`code_string_table` 是 `StrTable` 和 `IntTable` 类的成员函数，但没有给出其具体实现，因此我们只能基于函数名和一般编译器设计的经验来推测其作用。

在编译器的上下文中，`code_string_table` 可能用于 **生成目标代码或中间表示**，以定义或引用字符串表或整数表中的项。由于字符串和整数常量在运行时需要特定的内存布局或位置，因此这样的函数用于生成这些布局的代码。

这是一个简化的例子来解释其可能的作用：

假设我们有以下的源代码：

```
1 coolCopy codeclass Main inherits IO {  
2     main() : IO {  
3         out_string("Hello, World!")  
4     };  
5 }
```

在这里，字符串"Hello, World!"需要在运行时被存储在某个位置。编译器首先会将其添加到 `stringtable` 中。接着，在代码生成阶段，它需要为这个字符串生成数据定义。

一个简化的汇编代码片段可能如下：

```
1 assemblyCopy code.data  
2 string_1: .asciiz "Hello, World!"
```

`code_string_table` 函数可能会负责生成上述汇编代码片段，将所有 `stringtable` 中的字符串都转换为这种格式的数据定义。

同样地，对于整数表 `inttable`，函数可能会生成整数常量的数据定义。

总之，`code_string_table` 函数可能是用于生成存储在表中的常量的目标代码或中间表示的。但没有具体的函数实现或文档，我们只能推测其精确的作用。

我注意到，对于 `IdTable`，并没有这个方法

ChatGPT

是的，从你提供的代码来看，确实只有 `strTable` 和 `IntTable` 有 `code_string_table` 方法，而 `IdTable` 没有。

这个设计选择有一定的道理：

- 字符串和整数常量需要数据存储：**字符串（如"Hello, World!"）和整数常量（如 123）在运行时都需要一个特定的内存位置。这是因为它们是程序的实际数据，所以需要为它们分配存储空间，并在生成的代码中引用这些位置。因此，对于这两种类型的表，我们需要一个方法来生成这些数据的定义。
- 标识符不需要数据存储：**相反，标识符（如变量名、类名或函数名）通常不存储为数据。而是在编译时被转换为某种引用（如内存地址、寄存器或其他编译器生成的标识符）。因此，标识符本身不需要在运行时的数据段中有一个位置。这解释了为什么 `IdTable` 不需要一个 `code_string_table` 方法：我们不需要为标识符生成数据定义。

这种区别是基于编译器如何处理不同种类的符号。字符串和整数常量被视为数据，需要在程序的数据段中分配存储空间；而标识符在编译过程中有不同的处理方式，通常它们的名字在运行时不直接出现，因此不需要生成对应的数据定义。

- operator

- multiple-character operator和single-character operator的本质区别：

/这里有一个巧妙的设计，对于ASCII码中已经定义的字符集，直接返回，就是它的“值”，而

对于multiple-character operators，就必须要返回自己预先定义的一些值，比如256, 257。

这样就实现了=>和=在semantics层面的一致性/ 同样的，关键字的返回值也是基于此，实现了semantically-equal

- 由于这些匹配规则的先后顺序也很重要，所以说，出问题了也要想想是不是顺序没有排对。

- 盘点一些逆天bug：

```
/*
 * Nested comments
 */
"(*"      {BEGIN(COMMENT);}
<COMMENT>"*)" { BEGIN(INITIAL); }
<COMMENT>.* []
<COMMENT>\n    {curr_lineno++}; /*易错点，忽略对\n的特殊处理----它会影响一些全局变量*/
<COMMENT><<EOF>>     { BEGIN(INITIAL); cool_yylval.error_msg = "EOF in comment"; return ERROR; }

/*上面这条不看答案，很难想到*/
"--".*      {}
```

- lexer的工作原理：

所以说，我的lexer是针对我写的每一条rule生成一个DFA，还是综合所有的rule生成一个总的DFA？

ChatGPT

综合所有的规则生成一个总的DFA。

具体地说，词法分析器生成工具（例如Flex）通常按照以下步骤工作：

1. **生成NFA**：针对每一个提供的规则（正则表达式），工具会首先为每一个单独的规则生成一个NFA。
2. **合并NFA**：所有这些单独的NFA被组合成一个大的NFA。这可以通过创建一个新的初始状态并从这个状态向每一个规则对应的NFA的初始状态添加 ϵ （空）转移来完成。
3. **从NFA到DFA**：接着，这个大的NFA被转换为DFA。这是通过所谓的“powerset构造法”或“子集构造法”完成的。这个过程可能会生成大量的状态，但每个状态都明确地表示一个特定的字符输入应该如何转移。

4. **状态最小化**: 为了使DFA更高效，它会被最小化，以确保它没有冗余的状态。

在执行阶段，词法分析器使用这个总的DFA来匹配输入的字符序列。当它进入一个接受状态（即一个状态代表一个有效的记号结束）时，它会查看是哪条规则首先匹配并执行该规则的关联动作。

由于是总的DFA，它可以在单次扫描中处理所有的规则。这使得DFA驱动的词法分析器非常高效，因为它只需为每个输入字符执行一次状态转移。

你的意思是，词法分析器只要遇到accepted state，就会停止对DFA的遍历，把当前的结果从input'中提取出来？但是这会不会导致不满足最长匹配原则

ChatGPT

你指出了一个重要的问题，即“最长匹配原则”。我的上一次解释可能有些简化，确实没有涉及这一点。让我们进一步探讨这个问题。

你是对的，DFA在遇到accepted state时不会立即停止。实际上，词法分析器通常使用贪婪匹配策略，尝试找到最长的匹配。

为了说明这一点，我们再考虑一个简单的例子：

规则：

1. `if`
2. `[a-z]+` (匹配任意小写字母串)

现在，如果输入是"ifelse"：

- 一个简单的DFA可能在读取"if"时即刻认为已找到一个匹配（即关键字"if"）。
- 但根据最长匹配原则，它应该继续读取，因为"ifelse"也符合第二个规则，即一个由小写字母组成的字符串。

因此，词法分析器的工作方式如下：

1. 它会持续读取，直到没有更多的转移可以接受更多的输入为止。
2. 然后它回溯到上一个accepted state，并返回该状态对应的token。

在上面的例子中，DFA会在读完"ifelse"后停止，但它会回溯到"if"并返回关键字`if`的token，因为规则1在规则列表中出现得比规则2早。

这就是词法分析器如何确保遵循最长匹配原则的。同时，当有多个规则都可以匹配同一个字符串时，它会选择规则列表中最早的那个规则。

- 关于返回的类型和值：

下面是我截取的部分规则：我发现，对于每一个keyword或者是operator，我是仅仅返回其type，而且每一个不同的operator的返回type都不一样，但是对于字符串，我们在返回时区分它们的值，返回相同的type，为什么会有这种差异？

ChatGPT

你提到了词法分析器处理关键字、运算符和字面量（如字符串、数字）时的两种不同策略。这两种策略之间的差异源于这些token的语义和用途。让我们一一分析：

1. 关键字和运算符：

- 关键字（如 `if`、`while`）和运算符（如 `+`、`-`）在大多数编程语言中具有固定的含义。对于编译器来说，识别这些token的确切类型就足够了，因为它们的语义是明确的。
- 举个例子，关键字 `if` 在源代码中的出现表示一个条件语句的开始。编译器只需要知道遇到了一个 `if` 关键字，而不需要关心它的“值”。
- 同理，对于运算符，识别出是哪个运算符就足够了（例如 `+`、`-`、`*` 等）。

2. 字面量：

- 字面量，如字符串、整数和浮点数，有一个与之关联的值。这个值对于后续的编译步骤（例如语义分析和代码生成）是非常重要的。
- 例如，字符串 `"hello"` 和 `"world"` 都是字符串，但它们的值是不同的。因此，虽然它们都是 `STR_CONST` 类型，但它们的具体值是必要的。
- 同样，数字 `42` 和 `100` 都是整数，但它们的值也是不同的。

为了解决这个问题，大多数编译器的词法分析器将为每个token返回两部分信息：

1. **Token类型**：这告诉编译器这是什么样的token（例如，关键字、运算符、字符串、整数等）。
2. **Token的属性值**：这对于字面量是必需的，表示该token的实际值（例如，字符串的内容或整数的数值）。

在你提供的例子中，关键字和运算符只返回其类型，因为它们的值对编译器的后续步骤没有意义。但对于像字符串这样的字面量，返回其类型和值都是必要的。

- 这个PA的扣分全在COMMET的部分，尤其要注意的是对于“全局状态”的把握，一般来说，写出下面的规则是很危险的；它表示在任何状态下你遇到(*)字符都会进入COMMENT状态，那么，如果是在STRING状态呢hhh

```
1 | "(*"           {cmt_depth++; BEGIN(COMMENT);}
```

- 正确的写法见源代码

- 总之做PA2的一个很大收获在于体会到了使用%**x** 定义的这些状态的妙用，以及对源码有了一个初步的认识

```
orange0o0@LAPTOP-8TA12ITF:~/cs143/assignments/PA2$ perl pal-grading.pl
Grading ....
make: Entering directory '/home/orange0o0/cs143/assignments/PA2'
make: 'lexer' is up to date.
make: Leaving directory '/home/orange0o0/cs143/assignments/PA2'
=====
submission: ..
=====
You got a score of 63 out of 63.

Submit code:
63:03afdbd66e7929b125f8597834fa83a4
```

parser

- 不是所有语言都是正则的

我不太能理解下面这段话：Strings of balanced parentheses are not regular

ChatGPT

这句话关于“平衡的括号的字符串不是正则的”需要一些形式语言理论的背景来理解。

首先，当我们谈论一个语言是“正则的”，我们指的是该语言可以由一个正则表达式描述，或者说可以由一个确定性有限自动机（DFA）或非确定性有限自动机（NFA）识别。

考虑一个语言 L ，其中包含所有的平衡的括号字符串。一个字符串被认为是平衡的，当且仅当它包含相同数量的打开括号 $($ 和关闭括号 $)$ ，并且在任何字符串的前缀中，打开括号的数量都不少于关闭括号的数量。

例如：

- $()$ 是平衡的。
- $()()$ 是平衡的。
- $(())$ 是平衡的。
- $)()$ 是不平衡的。
- $(())$ 是不平衡的。

DFA的不足：对于下面的DFA，它只能判断01字符串中1的数量是不是偶数个，但是不能判断1到达accepted state的次数。而对于诸如左右括号、if-else这种嵌套结构，其accepted的状态的判断是以来于上面提到的“次数”的，这就引出了parse



parser vs lexer:

<i>Phase</i>	<i>Input</i>	<i>Output</i>
Lexer	<u>String of characters</u>	<u>String of tokens</u>
Parser	<u>String of tokens</u>	<u>Parse tree</u> <i>(may be implicit)</i>

上下文无关法

首先结合具体事例体会它的威力：

现在，来看一个具体的例子。

假设我们有以下的CFG，它描述了简单的平衡括号的字符串：

- | | |
|---|--------------------------|
| 1 | $S \rightarrow (S)$ |
| 2 | $S \rightarrow SS$ |
| 3 | $S \rightarrow \epsilon$ |

其中， S 是开始符号， ϵ 表示空字符串，“(”和“)”是终结符。

使用这个CFG，我们可以生成如下的字符串：

1. $S \rightarrow \epsilon$: 这意味着空字符串是这个语言的一部分。
2. $S \rightarrow (S) \rightarrow ()$: 这表示字符串“()”是这个语言的一部分。
3. $S \rightarrow SS \rightarrow (S)S \rightarrow (S)(S) \rightarrow ()(S) \rightarrow ()()$: 这表示字符串“()()”也是这个语言的一部分。

以上的例子都符合定义，即：从开始符号 S 出发，我们可以经过一系列的产生式得到一个仅由终结符组成的字符串。

因此，根据给定的定义，该CFG描述的语言是所有由这个CFG生成且仅由终结符组成的字符串的集合。

结合这个例子可以理解这句话：

Let G be a context-free grammar with start symbol S . Then the language of G is:
 $\{a_1\dots a_n \mid S \rightarrow^* a_1\dots a_n \text{ and every } a_i \text{ is a terminal}\}$

同时你也可以理解start symbol存在的意义

我们常常写出下面的context-free gramer: Expr->id, 这里Expr是表达式的意思, 我想知道id的详细含义

ChatGPT

在许多编译器和形式语言的文献中, `id` 通常是一个代表标识符 (identifier) 的缩写。标识符在编程语言中是一个非常常见的概念, 它用于命名变量、函数、类型等。

在形式化语言和上下文无关文法 (Context-Free Grammar, CFG) 的描述中, `id` 是一个终结符, 它代表源代码中的一个标识符。

例如, 考虑以下的简单语法:

```
1 bashCopy codeExpr -> id
2 | Expr + Expr
3 | ( Expr )
```

这个语法描述了一个简单的算术表达式, 其中表达式可以是一个标识符, 两个表达式通过加号连接, 或者一个在括号中的表达式。

在这种情况下, `id` 可以代表任何有效的标识符, 例如: `x`, `y`, `totalAmount`, `computeValue` 等。

在实际的编译器或解释器的实现中, 通常会有更为复杂的规则来定义什么样的字符串序列构成一个有效的 `id`。例如, 在许多编程语言中, 标识符必须以字母或下划线开头, 后跟字母、数字或下划线。这些具体的词法规则通常是通过词法分析器 (lexer) 来实施的, 而不是在CFG中明确指定的。

按照你的说法, 纯数字, 比如37并不能算作一个标识符, 这就意味着37+37并不符合上述要求?

ChatGPT

你是对的, 基于我先前给出的简化语法, 表达式 `37 + 37` 确实不符合那个语法, 因为 `37` 不是一个标识符 (`id`)。

在许多实际的上下文无关文法中, 你会看到对数字 (如整数、浮点数等) 有明确的定义。例如, 我们可以扩展上述语法, 使其更加完整:

```

1 scssCopy codeExpr    -> id
2 | num
3 | Expr + Expr
4 | ( Expr )
5
6 id      -> [a-zA-Z_][a-zA-Z0-9_]* 
7 num     -> [0-9]+

```

在这个扩展的语法中，`id` 仍然代表标识符，而新添加的 `num` 代表数字。`[a-zA-Z_]` 和 `[a-zA-Z0-9_]*` 这样的正则表达式定义了标识符的模式。`[0-9]+` 定义了至少一个数字组成的数字模式。

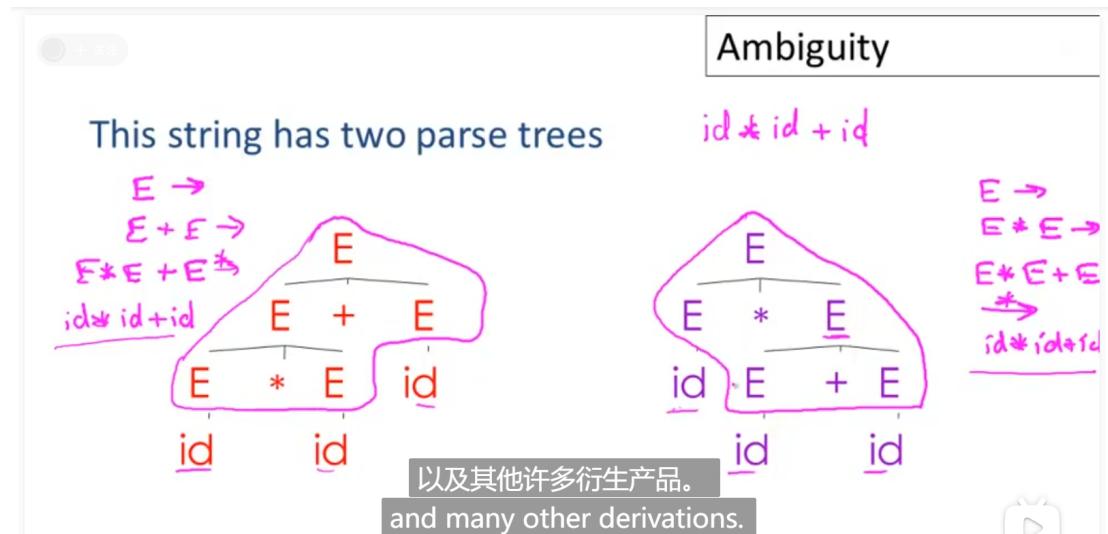
这样，表达式 `37 + 37` 就满足这个扩展后的语法了，因为 `37` 被视为一个数字 (`num`)，而不是标识符。

截至目前，我们能做的仅仅是判断 yes or no：目前只能判断 a string of tokens 是否属于某一种由 CFG 定义的语言，但是这距离我们得出语法树还是有相当的距离的。

因此，我们下面要做的工作是：对于某一个 EXPR，在判断为 yes 的基础上，对它进行 derivation，得到一个 parse tree。

这里又会引出一个问题，对于一个判定结果为 yes 的 EXPR，我们可以由多种方式（一般使用 left-most/right-most 进行解析）进行 derivation 得到它，那么，不同的 derivation 过程就对应着不同的 parse tree 吗？还是说对于一个判定结果为 yes 的 EXPR，其 parse tree 一定是固定的？

答案是否定的，考虑下面的例子：



对于同一个 expr，我们按照下面展示的 CFG，将会解析出两种不同的 parse tree，这不是编译器希望看到的

- Grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

这就引出了 **ambiguous** grammar的概念了

解决方法：

- 写出更优雅的grammar: (most used)

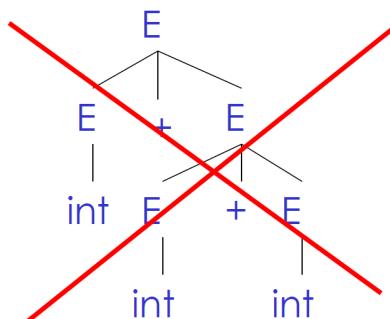
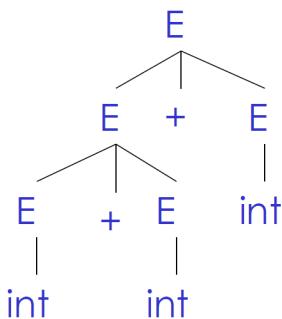
- Most direct method is to rewrite grammar unambiguously

$$\begin{cases} E \rightarrow E' \pm E \mid E' \\ E' \rightarrow id * E' \mid id \mid (E) * E' \mid (E) \end{cases}$$

- 仔细体会一下这个优化后的grammar，它实现了一个效果：所有的加法都必须在乘法之前被先进行分解，换言之，感觉它从某种程度上实现了乘法和加法的运算优先级的问题
- 另一种方法：(由于使用ambiguous的模式更贴近自然) 现在许多编译器都使用这种更自然的方法，在出问题时（出现**二义性**时）再进行特殊处理 (with disambiguation declarations)。

eg:下面的原来的grammar具有二义性，但是新增了 **Left associativity declaration**之后就明确了：

- Consider the grammar $E \rightarrow E + E \mid int$
- Ambiguous: two parse trees of $int + int + int$



- Left associativity declaration: `%left +`

error handling

以bison采取的方法为例：理解下面这张图：

- Bison: use the special terminal error to describe how much input to skip



AST

自顶向下法--recursive descent

直观画图理解--这个过程很好模拟

<https://www.bilibili.com/video/BV17K4y147Bz?t=303.7&p=23>

算法表述

首先要理解下面3个函数：

- Define boolean functions that check for a match of:
 - A given token terminal $\text{bool term}(\text{TOKEN tok}) \{ \text{return } *_{\text{next}} \text{++} == \text{tok}; \}$ true/false
 - The nth production of S:
 $\rightarrow \text{bool } S_n() \{ \dots \}$
 - Try all productions of S:
 $\text{bool } S() \{ \dots \}$ 他们检查了一下,

S返回为真的条件是any of Si返回为真，懂？（存在，而非任意）

其实还是有很多细节的：比如下面的例子，就巧妙地利用了&&和||的短路原理实现了不同production优先级排序

- For production $E \rightarrow T$
bool E₁() { return T(); }
- For production $E \rightarrow T + E$
bool E₂() { return T() && term(PLUS) && E(); }
- For all productions of E (with backtracking)

```
bool E() {
    TOKEN *save = next;
    return (next = save, E1())
        || (next = save, E2()); }
```



存在问题

考虑下面的函数，如果输入int*int，会被rejected：因为E会返回true，但是指针没有遍历整个input。从程序的角度，这正是先前的短路运算导致的：T1的成功导致E的判定成功，这导致其他的production彻底失去了被尝试的可能

Implementation

$E \rightarrow T \mid T + E$
 $T \rightarrow \text{int} \mid \text{int} * T \mid (E)$

```

bool term(TOKEN tok) { return *next++ == tok; }

bool E1() { return T(); }
bool E2() { return T() && term(PLUS) && E(); }

bool E() { TOKEN *save = next; return (next = save, E1())  

           || (next = save, E2()); }

bool T1() { return term(INT); }
bool T2() { return term(INT) && term(TIMES) && T(); }
bool T3() { return term(OPEN) && E() && term(CLOSE); }

bool T() { TOKEN *save = next; return (next = save, T1())  

           || (next = save, T2())  

           || (next = save, T3()); }

```

↑ int * int rejected

What happened?!

当我们试图找到一个适合特定非终端的产品时，



事实上，这个分解算法是由缺陷的，但是它胜在简洁。特别的，如果一种grammar，对于任何非终端节点，最多只有一个production succeed，那么这个算法就刚刚好

左递归问题

- 遇到直接或者间接地存在左递归地grammar是。这种自顶向下地算法也会失效
- 这个一般靠人们通过重写语法的方式来避免

链接

自顶向下法--predictive parsing

- Like recursive-descent but parser can “predict” which production to use
 - By looking at the next few tokens
 - No backtracking
- Predictive parsers accept LL(k) grammars

*lookahead
restricted grammars*

$\xrightarrow{\text{left-to-right}}$ $\xleftarrow{\text{k tokens}}$ $\xrightarrow{\text{lookahead}}$
 \downarrow $\text{left-most derivation}$ $(k=1)$

left factoring

- 是什么? 为什么? 怎么做?

一言以蔽之, 改写语法, 使得语法的每一步都可以在LL(1)的情况下进行predict

- Recall the grammar

$$\begin{aligned} \rightarrow E &\rightarrow T + E \mid T \\ \rightarrow T &\rightarrow \underline{\text{int}} \mid \underline{\text{int}}^* T \mid (E) \end{aligned}$$

$$\begin{aligned} E &\rightarrow T X \\ X &\rightarrow + E \mid \epsilon \\ T &\rightarrow \underline{\text{int}} + Y \mid (E) \\ Y &\rightarrow * T \mid \epsilon \end{aligned}$$



parsing table

- 怎么用? 结合例子记忆算法

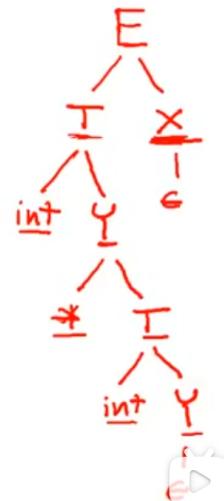
- Method similar to recursive descent, except
 - For the leftmost non-terminal S
 - We look at the next input token a
 - And choose the production shown at [S,a]
- A stack records frontier of parse tree
 - Non-terminals that have yet to be expanded
 - Terminals that have yet to be matched against the input
 - Top of stack = leftmost pending terminal or non-terminal
- Reject on reaching error state
- Accept on end of input & empty stack



	int	*	+	()	\$
E	TX			TX		
X			+ E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ



Stack	Input	Action
<u>E\$</u>	<u>int * int \$</u>	<u>TX</u>
<u>int X\$</u>	<u>int * int \$</u>	<u>int Y</u>
<u>int Y X\$</u>	<u>int * int \$</u>	terminal
<u>* T X\$</u>	<u>* int \$</u>	<u>* T</u>
<u>T X\$</u>	<u>int \$</u>	<u>int Y</u>
<u>int Y X\$</u>	<u>int \$</u>	terminal
<u>Y X\$</u>	<u>\$</u>	<u>ε</u>
<u>X\$</u>	<u>\$</u>	<u>ε</u>
<u>\$</u>	<u>\$</u>	<u>ACCEPT</u>



Alex Aiken

- 如何生成parsing table? :

按照下图所示规则:

- 第一点很好理解: 它其实很好的利用了大背景: 我们已经对原来的grammar进行了left factoring, 因此, 只要A经过了*步的推导 (这里的*表示 ≥ 0), 发现input中的t在首位, 则与之相应的a必然就是唯一应当采取的derivation, 这是由left factored的grammar保证的。
- 第二点其实是对第一种case的一种补充--它考虑到A经过了*步的推导 (这里的*表示 ≥ 0), 得出epsilon的case

- Consider non-terminal A, production $A \rightarrow \alpha$, and token t

- Add $T[A, t] = \alpha$
if $A \rightarrow \alpha \rightarrow^* t \beta$

- α can derive a t in the first position
- We say that $t \in \text{First}(\alpha)$

Greek letters denote strings of non-terminals and terminals

- Add $T[A, t] = \epsilon$

- if $A \rightarrow \alpha \rightarrow^* \epsilon$ and $S \rightarrow^* \gamma A \delta$

- Useful if stack has A, input is t, and A cannot derive t
- In this case only option is to get rid of A (by deriving ϵ)
 - Can work only if t can follow A in at least one derivation
 - We say $t \in \text{Follow}(A)$

- first set的计算:

- 规则: (这里尤其要注意第三点, 注意是for $1 < i < n$, 这意味着对于A_{1-n}都必须满足epsilon属于First(X))

Definition

$$\text{First}(X) = \{ t \mid X \xrightarrow{*} t\alpha \} \cup \{ \epsilon \mid X \xrightarrow{*} \epsilon \}$$

Algorithm sketch:

- 1. $\text{First}(t) = \{ t \}$ t is a terminal
- 2. $\epsilon \in \text{First}(X)$
 - if $X \xrightarrow{*} \epsilon$
 - if $X \xrightarrow{*} A_1 \dots A_n$ and $\epsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$
- 3. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \xrightarrow{*} A_1 \dots A_n \alpha$
 - and $\epsilon \in \text{First}(A_i)$ for $1 \leq i \leq n$



- 示例：这里，由于First (T) 中不包含epsilon，所以FirstE直接等于FirstT，如果包含epsilon的话，根据规则3，只能是子集关系。

- Recall the grammar

$\begin{array}{l} E \rightarrow T X \\ T \rightarrow (E) \mid \text{int } Y \end{array}$	$\begin{array}{l} X \rightarrow \pm E \mid \epsilon \\ Y \rightarrow * T \mid \epsilon \end{array}$
$\left\{ \begin{array}{l} \text{First}(+) = \{ + \} \\ \text{First}(*) = \{ * \} \\ \vdots \\ \{ > \} \\ \{ \text{int} \} \end{array} \right.$	$\left\{ \begin{array}{l} \text{First}(E) = \text{First}(T) \\ \text{First}(T) = \{ (, \text{int} \} \\ \text{First}(X) = \{ +, \epsilon \} \\ \text{First}(Y) = \{ *, \epsilon \} \end{array} \right.$



- follow set的计算：

- 规则：

- epsilon只在First set的考虑范围之内，而在Follow set的考虑范围之内
- 尤其是要注意对第三条的理解应用

- Definition:

$$\text{Follow}(X) = \{ t \mid S \xrightarrow{*} \beta X \underline{t} \delta \}$$

$$\begin{array}{c} S \xrightarrow{*} X t \xrightarrow{*} A B t \xrightarrow{*} \\ \underline{A} \in \end{array}$$

$$X \xrightarrow{*} A B \xrightarrow{*} \underline{A} \in B$$

- Intuition

– If $X \xrightarrow{*} A B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and

$$\text{Follow}(X) \subseteq \text{Follow}(B)$$

• if $B \xrightarrow{*} \epsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$

– If S is the start symbol then $\$ \in \text{Follow}(S)$



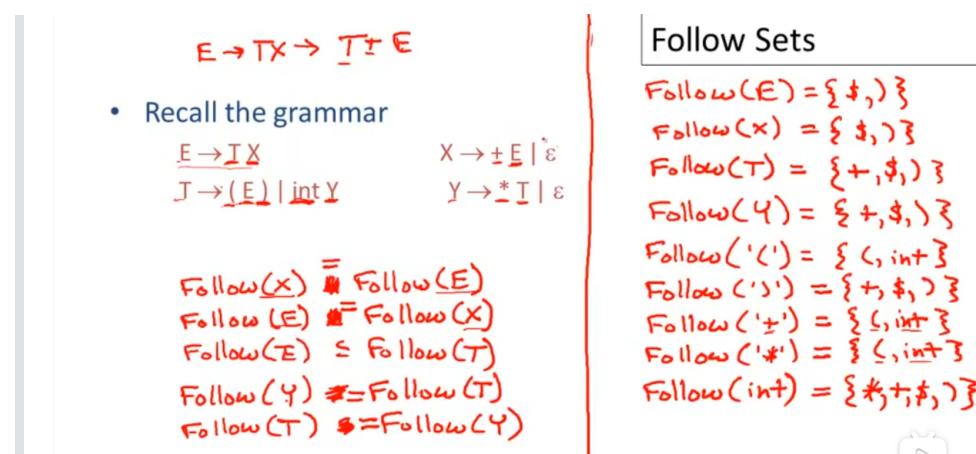
Algorithm sketch:

1. $\$ \in \text{Follow}(S)$
2. $\text{First}(\beta) - \{\epsilon\} \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$
3. $\text{Follow}(A) \subseteq \text{Follow}(X)$
 - For each production $A \rightarrow \alpha X \beta$ where $\epsilon \in \text{First}(\beta)$

这个后缀的生产可以完全消失。
this suffix of the production can completely disappear.



- 示例：尤其是要注意上面规则中的第三条。以下图为例：尤其要注意最终可以得出epsilon的rightmost non-terminal结点。比如这里，X可以得出epsilon，这导致Follow (X) 还将包括Follow(E)
- follow set的计算，主要是找目标出现在哪些位置，把这些位置遍历一遍即可



- 基于上面两个方法建立parsing table：理解了First set和Follow set的含义之后，建表规则就很好理解了：

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
 - For each terminal $t \in \text{First}(\alpha)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$, for each $t \in \text{Follow}(A)$ do
 - $T[A, t] = \alpha$
 - If $\epsilon \in \text{First}(\alpha)$ and $\$ \in \text{Follow}(A)$ do
 - $T[A, \$] = \alpha$



- parsing table可以用于反向验证先前的grammar是否符合LL (1) 的要求，唯一的判断依据是建立parsing table。看看吗=每个entry是不是唯一的
- 事实证明，大部分主流编程语言对应的语法都不是LL (1) 的，但是他的思想得到了传承doge

Bottom up法

两个重要概念：viable prefix和handle：

请解释viable prefix的概念及其和handle的关系？

ChatGPT

在自底向上 (bottom-up) 语法分析中，两个关键的概念是 "handle" 和 "viable prefix"。

Viable Prefix (可行前缀)

定义：Viable prefix 是那些可以出现在解析堆栈的顶部的输入字符串的前缀。这些前缀可能是一个句柄，或可能在后续的输入后成为一个句柄。简而言之，这是一个可以在堆栈中进一步解析的前缀，而不会导致错误。

示例：考虑我们之前提到的文法：

```
1 rCopy codeE → E + T | T  
2 T → T * F | F  
3 F → ( E ) | id
```

对于输入 `id + id * id`，以下是一些可行的前缀：

- `id`
- `E +`
- `E + id`
- `E + T *` ...等等。

这些都是可以出现在堆栈顶部的前缀，因为它们都可以在后续的输入后被进一步解析。

基于其定义，可以有如下推论：

Handle

请详细向我解释bottom up语法分析中 handle的概念

ChatGPT

自底向上语法分析，特别是在 shift-reduce 解析中，“handle”是一个核心概念。理解“handle”对于理解整个解析过程是很重要的。

定义：在自底向上解析中，“handle”是一个与文法的某个产生式直接匹配的句柄，可以立即使用这个产生式进行归约。

以下是与 handle 相关的重要概念：

1. **位置：**handle 通常位于解析堆栈的顶部。这意味着它是当前已解析符号的一个子串，并且可以立即归约。
2. **归约：**当找到一个 handle 时，解析器会将其替换为该产生式的左侧的非终结符。这就是所谓的“reduce”操作。
3. **目的：**handle 的概念是为了保证我们在正确的时机进行归约，确保产生的语法树是输入句子的正确解释。

以一个简单的文法为例：

```
1 rCopy codeE → E + T | T  
2 T → T * F | F  
3 F → ( E ) | id
```

考虑一个输入句子 `id + id * id`。

当我们解析到 `id + id *`，堆栈内容可能是 `E + T *`。在这个状态下，`T *` 是下一个产生式 `T → T * F` 的一个前缀，但它还不是个 handle，因为我们不能只使用 `T → T *` 进行归约。相反，我们需要继续移入堆栈直到找到一个完整的 handle。

当我们进一步解析并移入下一个 `id`，堆栈变为 `E + T * F`。现在，`T * F` 是一个 handle，因为它完全匹配产生式 `T → T * F`，因此可以归约为 `T`。

这个过程继续进行，直到整个输入句子被解析。

总的来说，handle 是自底向上解析策略的核心，它决定了何时以及如何归约输入句子的一部分。找到正确的 handle 是确保得到正确语法树的关键。

Viable Prefix 与 Handle 的关系

1. **所有的 handle 都是 viable prefix：**当一个字符串前缀是一个 handle 时，它可以被立即归约。但是，并不是所有的 viable prefix 都是 handle。
2. **堆栈上的内容始终是一个 viable prefix：**解析器的工作是确保堆栈上总是一个可行的前缀，这意味着它可以在后续的输入后继续解析。
3. **归约时机：**当堆栈的顶部是一个 handle 时，解析器会进行归约操作。但如果堆栈顶部只是一个可行的前缀（而不是句柄），那么解析器可能选择移入操作。

总结一下，viable prefix 是描述可以在堆栈上进一步解析的前缀，而 handle 是指示归约操作何时应该发生的特定类型的可行前缀。理解这两个概念以及它们之间的关系对于理解自底向上解析过程非常关键。

NFA构建算法：

- 1.Add a new start production $S' \rightarrow S$ to G (grammar)
- 2.The NFA states are the items of G -(Including the new start production)
- 3.For item $E \rightarrow \alpha.X\beta$ add transition $E \rightarrow \alpha.X\beta \rightarrow X \quad E \rightarrow \alpha X.\beta$
 - 这里的X既可以是terminal，又可以是non-terminal
- 4.For item $E \rightarrow \alpha.X\beta$ and production $X \rightarrow \gamma$ add $E \rightarrow \alpha.X\beta \rightarrow \epsilon \quad X \rightarrow .\gamma$
 - 这里的X必然是non-terminal
 - 要理解第四点，首先要足够理解NFA：它的状态转移应当理解成一种并行的状态的探索

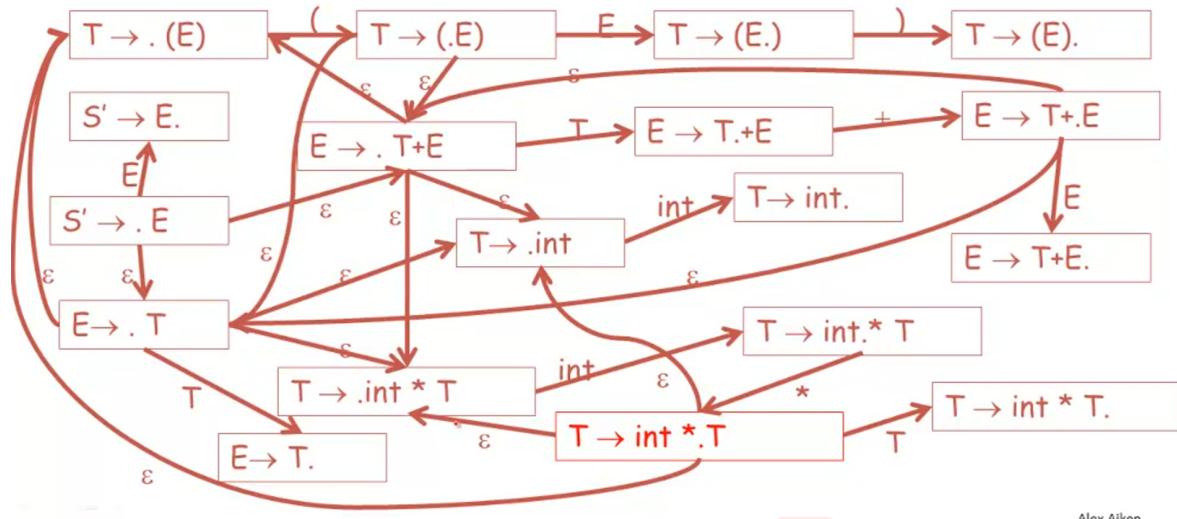
当我们看到形如 $E \rightarrow \alpha.X\beta$ 的项时，它表示当前的解析位置在X之前。此时，解析器试图决定下一步是移入 X 还是执行某种归约。为了做出决策，解析器需要考虑可能的产生式，即 X 可能如何展开。

考虑项 $E \rightarrow \alpha.X\beta$ 和产生式 $X \rightarrow \gamma$ 。虽然在当前输入位置上还没有明确的 γ ，但解析器需要知道这是一个可能的选择。因此，解析器考虑 ϵ -转移来表示这一可能性，即在没有消耗任何输入的情况下开始解析 X 。

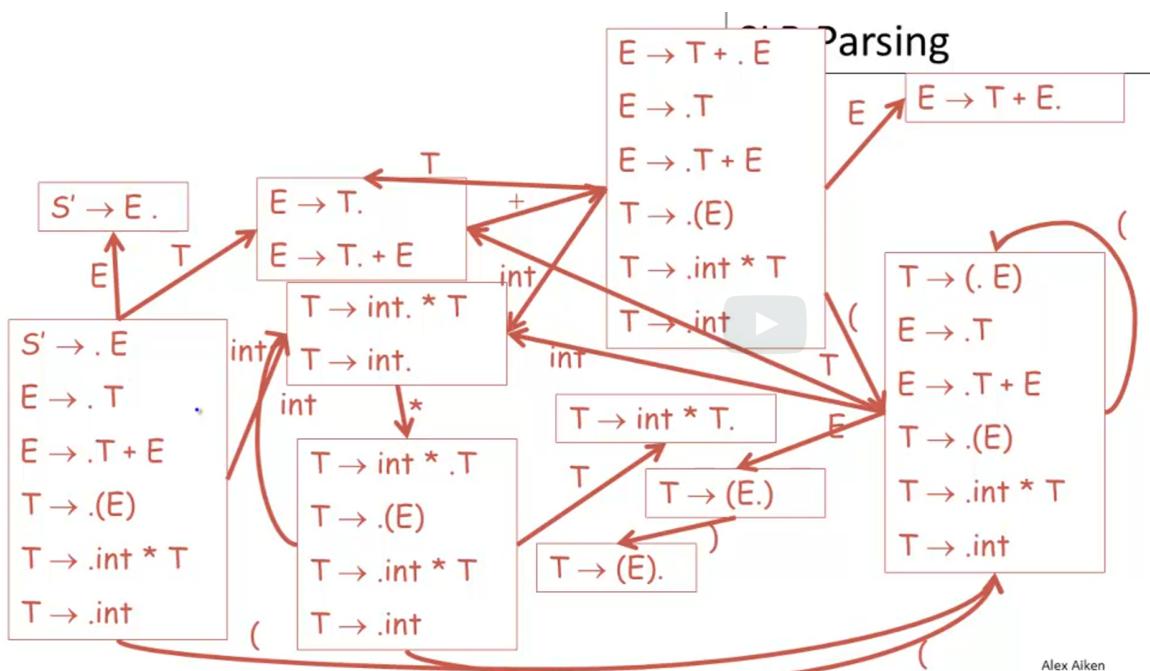
此 ϵ -转移并不意味着我们已经在输入中看到了 γ ，而是表示我们正在为解析 X 做准备，并考虑它可以如何展开。这就是为什么虽然 $E \rightarrow \alpha.X\beta$ 和 $X \rightarrow .\gamma$ 的成立条件不同，但在这种上下文中使用 ϵ -转移是有意义的。

$$\left[\begin{array}{l} S' \rightarrow E \\ \left[\begin{array}{l} E \rightarrow T + E \mid T \\ T \rightarrow \text{int} * T \mid \text{int} \mid (E) \end{array} \right] \end{array} \right]$$

目前对于下面这张图中的某些transition不是很理解（或者说对NFA还是不够理解？



转换为DFA:



SLR

- 主要为了减少conflicts
- 新增执行reduce的条件:

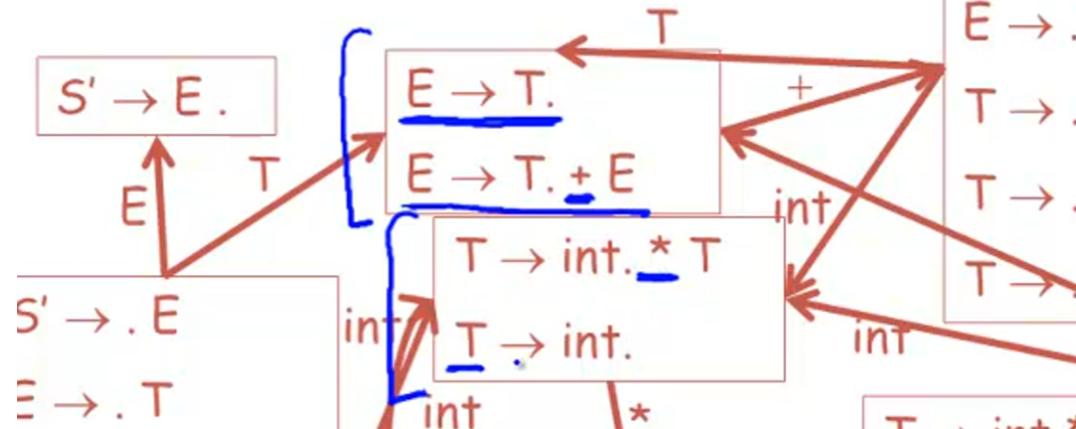
- Reduce by $X \rightarrow \beta$ if
 - s contains item $X \rightarrow \beta.$
 - $t \in \text{Follow}(X)$

$\rightarrow \dots \beta | \tau$
 $\rightarrow \dots \underline{\beta} | \tau$

举个例子: 对于下面的FDFA, 当读取到int的时候, 由于*在Follow (T) 中, 所以进行reduce而不是shift

$$\text{Follow}(E) = \{\$,)\}$$

$$\text{Follow}(T) = \{+,), +\}$$



- 算法：尤其是注意篮筐的两条

1. Let M be DFA for viable prefixes of G
2. Let $|x_1 \dots x_n \$|$ be initial configuration
3. Repeat until configuration is $|\$| \$$
 - Let $\alpha | \omega$ be current configuration
 - Run M on current stack α
 - If M rejects α , report parsing error
 - Stack α is not a viable prefix
 - If M accepts α with items I , let a be next input
 - Shift if $X \rightarrow \beta / \gamma \in I$
 - Reduce if $X \rightarrow \beta, \gamma \in I$ and $a \in \text{Follow}(X)$
 - Report parsing error if neither applies

如果在采用了避免措施的情况下，仍然在最后一步发生conflict，那么就是grammar的问题：当前grammar不是SLR (k) 的grammar

[这里有一个典型例子](#)需要注意的是，每次发生reduce之后都需要从头开始

SLR improvement

算法：求求你给我个例子吧qwq

```
Let I = w$ be initial input  
Let j = 0  
Let DFA state 1 have item  $S' \rightarrow .S$   
Let stack =  $\langle \text{dummy}, 1 \rangle$   
repeat  
    case action[top_state(stack), I[j]] of  
        shift k: push  $\langle I[j+1], k \rangle$   
        reduce X  $\rightarrow A$ :  
            pop |A| pairs,  
            push  $\langle X, \text{goto}[\text{top\_state}(stack), X] \rangle$   
        accept: halt normally  
        error: halt and report error
```

- Important Fact #1: A bottom-up parser traces a rightmost derivation in reverse

结合下面这张图片理解上面的话

$$\begin{aligned} E &\rightarrow T + E \mid T \\ T &\rightarrow \text{int} * T \mid \text{int} \mid (E) \end{aligned}$$

Observation

- Read the productions in reverse (from bottom to top)
- This is a reverse rightmost derivation!

int * int + int	$T \rightarrow \text{int}$
int * T + int	$T \rightarrow \text{int} * T$
T + int	$T \rightarrow \text{int}$
T + T	$E \rightarrow T$
T + E	$E \rightarrow T + E$
E	

- reduction: 和derivation相反的操作
- shift move& reduce move
 - 使用stack很合适，因为我们永远只可能对vertical bar左边字符的末端进行reduce
 - shift-reduce conflict: 相对来说容易解决
 - reduce-reduce conflict: 难以解决，一般是严重的语法问题
- 所以这里的核自然就在于：确定vertical bar在移动过程中，next step是shift还是reduction?
- 这样就自然引出了handle:
 - intuition:

- Intuition: Want to reduce only if the result can still be reduced to the start symbol

- Assume a rightmost derivation

$$S \xrightarrow{*} \alpha X \omega \rightarrow \alpha \beta \omega$$

- Then $\alpha\beta$ is a *handle* of $\alpha\beta\omega$



- Important Fact #2: In shift-reduce parsing, handles appear **only at the top of the stack**, never inside(可以说是根据其intuition的一个推论)
- 关于viable prefix:

请结合具体示例详细解释？

ChatGPT

当然可以。让我们使用一个简单的例子来解释。

考虑一个非常简单的语法，它只接受形如 a^*b 的字符串。这意味着我们期待一个或多个 'a' 字符，后面紧跟一个或多个 'b' 字符。

现在，我们有一个基于堆栈的解析器，它按照这个语法来解析字符串。

示例 1： aaab

1. 解析器开始读取输入的第一个字符，它是 'a'。
2. 'a' 是一个可行的前缀，因为按照我们的语法，'a' 可以出现在最前面。因此，解析器将它推入堆栈。
3. 解析器继续读取，遇到第二个 'a'，这仍然是一个可行的前缀，所以再次推入堆栈。
4. 当解析器读取到 'b' 时，根据我们的语法，'a' 后面可以跟 'b'，所以解析器开始弹出堆栈上的 'a' 并接受 'b'。
5. 解析器成功地解析了整个字符串，并且没有发现错误。

示例 2： aaba

1. 解析器开始读取第一个字符 'a' 并推入堆栈。
2. 继续读取下一个字符 'a' 并再次推入堆栈。
3. 解析器读取到 'b'，开始弹出堆栈上的 'a' 并接受 'b'。
4. 解析器读取到最后一个字符 'a'。此时，根据我们的语法，'b' 后面不应该是 'a'。此时，尽管堆栈上之前的内容（例如 'a'）都是可行的前缀，但现在解析器遇到了一个错误，因为 'b' 后面不能跟 'a'。

在第二个示例中，尽管堆栈上有可行的前缀 'a'，但由于后续输入的字符不符合期望的模式，因此检测到了一个解析错误。

这个例子的核心思想是，只要堆栈上的内容可以与之后的输入合并成有效的语法结构，解析器就认为一切正常。只有当堆栈上的内容无法与之后的输入合并成有效的语法结构时，解析器才会检测到错误。

- Important Fact #3 about bottom-up parsing: For any SLR grammar, **the set of viable prefixes is a regular language**

请结合合适的例子解释下面这段话：Important Fact #3 about bottom-up parsing: For any SLR grammar, the set of viable prefixes is a regular language

ChatGPT

好的，让我们首先分解这段话的关键概念，然后使用示例进行解释。

1. **Bottom-up parsing**: 这是一个解析方法，从输入的叶子（单个的词法单元或字符）开始，逐步构建语法结构，直到达到起始符号。
2. **SLR grammar**: SLR代表Simple LR. SLR文法是LR(0)文法的一种简化版本，它使用了更简单的方法来检查文法的冲突。
3. **Viable prefixes**: 如前所述，这些是可以出现在堆栈顶部的输入字符串的前缀，并且可能指向有效的解析。
4. **Regular language**: 正则语言是可以由正则表达式或确定性有限自动机 (DFA) 表示的语言。它们是形式语言的一个子集，具有可计算性质。

现在，重要事实#3的核心内容是：对于任何SLR文法，其可行前缀的集合都是一个正则语言。这意味着我们可以使用确定性有限自动机 (DFA) 来识别和处理这些可行前缀。

示例：

考虑一个简单的文法，它描述了加法：

```
1 mathematicaCopy codeE -> E + E  
2 E -> n
```

其中E代表表达式，n代表数字。

现在，假设我们正在使用基于堆栈的解析器进行自底向上的解析。当我们遇到输入 `n + n` 时，我们会经历以下过程：

1. 将 `n` 压入堆栈。
2. 当我们遇到 `+` 时，我们可以将前面的 `n` 识别为一个表达式E。
3. 继续推进，我们再次将 `n` 识别为另一个表达式E。

这里的关键点是，**堆栈上的内容（例如 `E +`）构成了可行前缀的集合**。这个可行前缀的集合可以通过一个确定性有限自动机 (DFA) 进行识别，因为它是一个正则语言。

这个事实是自底向上解析的一个关键组成部分，因为它意味着我们可以使用DFA来有效地识别并处理堆栈上的内容，以确定如何继续解析或何时应用文法规则。

为了真正构建这样的DFA并处理更复杂的文法，通常需要使用解析器生成工具，如Yacc或Bison，这些工具可以为给定的文法自动生成相应的DFA。

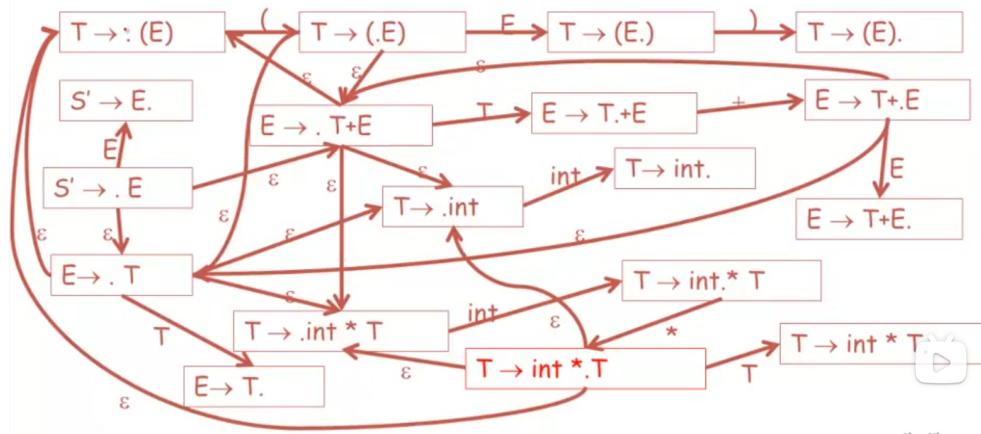
- notation: Item $T \rightarrow (E.)$ says that so far we have seen (E of this production and hope to see)
 - 额额，其实就是要标记当前vertical bar所在的位置
- 结合下面的例子理解，如何结合前面的概念，生成与之对应的NFA：

grammar:

$$\left\{ \begin{array}{l} S' \rightarrow E \\ E \rightarrow T + E \mid T \\ T \rightarrow \text{int} * T \mid \text{int} \mid (E) \end{array} \right.$$

NFA:

Recognizing VP



Alex Aiken

PA3

- 注意理解词法分析和语法分析是如何综合起来的

好好好

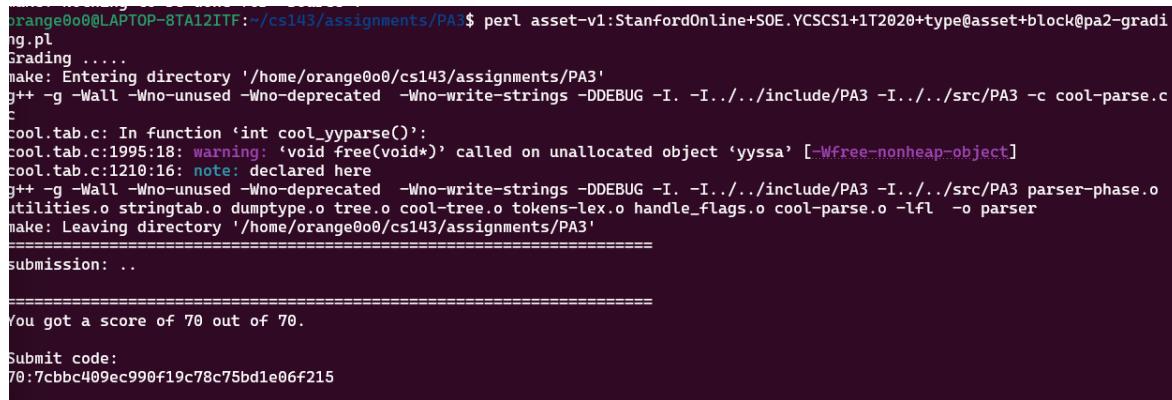
```
orange0o0@LAPTOP-8TA12ITF:~/cs143/assignments/PA3$ perl asset-v1:StanfordOnline+SOE.YCSCS1+1T2020+type@asset+block@pa2-grading.pl
Grading .....
make: Entering directory '/home/orange0o0/cs143/assignments/PA3'
g++ -g -Wall -Wno-unused -Wno-deprecated -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 -c cool-parse.c
cool.tab.c: In function 'int cool_yyparse()':
cool.tab.c:1995:18: warning: 'void free(void*)' called on unallocated object 'yyssa' [-Wfree-nonheap-object]
cool.tab.c:1210:16: note: declared here
g++ -g -Wall -Wno-unused -Wno-deprecated -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 parser-phase.o
utilities.o stringtab.o dumptype.o tree.o cool-tree.o tokens-lex.o handle_flags.o cool-parse.o -lfl -o parser
make: Leaving directory '/home/orange0o0/cs143/assignments/PA3'
=====
submission: ..

=====
You got a score of 70 out of 70.

Submit code:
70:7cbbc409ec990f19c78c75bd1e06f215
```

关于cool-tree.aps与tree.h、cool-tree.h之间的关系的理解

目前不是很理解这个.aps文件是如何在编译过程中发挥作用的。当时，对于.aps文件的理解花费了很长时间。目前来看，最好的情况是，这个所谓的aps文件+bison只是为程序员提供了一层建立AST树的抽象。



```
orange0o0@LAPTOP-8TA12ITF:~/cs143/assignments/PA3$ perl asset-v1:StanfordOnline+SOE.YCSCS1+1T2020+type@asset+block@pa2-grading.pl
Grading .....
make: Entering directory '/home/orange0o0/cs143/assignments/PA3'
g++ -g -Wall -Wno-unused -Wno-deprecated -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 -c cool-parse.c
cool.tab.c: In function 'int cool_yyparse()':
cool.tab.c:1995:18: warning: 'void free(void*)' called on unallocated object 'yyssa' [-Wfree-nonheap-object]
cool.tab.c:1210:16: note: declared here
g++ -g -Wall -Wno-unused -Wno-deprecated -Wno-write-strings -DDEBUG -I. -I../../include/PA3 -I../../src/PA3 parser-phase.o
utilities.o stringtab.o dumptype.o tree.o cool-tree.o tokens-lex.o handle_flags.o cool-parse.o -lfl -o parser
make: Leaving directory '/home/orange0o0/cs143/assignments/PA3'
=====
submission: ..

=====
You got a score of 70 out of 70.

Submit code:
70:7cbbc409ec990f19c78c75bd1e06f215
```

cool-tree.aps与tree.h、cool-tree.h其实没有必要完全读完，要建立Abstraction的概念，知道这些工具怎么用就好了。反而是仔细读读bison的文档，比如第二章的example部分对于加深对整个语法分析、词法分析过程的理解很有帮助。

%type

- 关于cool.y的一些源码分析（注意是语法不熟悉hhh）

在Bison中（这是你在示例中使用的工具，它是用于创建parsers的GNU工具），%type用于指定非终端符号的类型。

具体来说，Bison中的每一个非终端都有一个与之相关联的“语义值”。这些值是在解析过程中构建和传递的。你可以将它们视为在语法分析器规则中由\$\$和\$1、\$2等符号引用的值。

例如，在示例中：

```
1 %type <program> program
```

这一行声明了一个非终端 `program`，其语义值的类型为 `program`（这个类型在之前的 `%union` 中已经定义）。

同样：

```
1 %type <classes> class_list  
2 %type <class> class
```

上面两行分别为 `class_list` 和 `class` 这两个非终端指定了语义值的类型。

简而言之，`%type` 指令允许你为每个非终端指定一个类型，这样你在处理这些非终端的值时就知道它们的具体类型了。

TYPEID和OBJECTID

1. TYPEID 278

- `TYPEID` 很代表一个类型标识符，例如在许多面向对象的编程语言中，类的名称就是一个类型标识符。所以在COOL (Classroom Object-Oriented Language) 中，当你定义一个新的类时，例如 `class Hello {}`，这里的 `Hello` 就是一个 `TYPEID`。

2. OBJECTID 279

它可能表示对象的标识符，即变量名或方法名。在面向对象的语言中，对象通常是一个类的实例。例如，考虑以下的COOL代码：

```
1 let greeting: String <- "Hello world!";
```

在这里，`greeting` 是一个 `OBJECTID`，它是 `String` 类型的一个实例。

现在，为什么要为这些标识符提供类型，例如 `<symbol>`？这是因为，在词法分析期间，当词法分析器识别到一个字符串常量、类型标识符或对象标识符时，它不仅返回 `token` 类型（例如 `STR_CONST`、`TYPEID` 或 `OBJECTID`），还返回与该 `token` 关联的值。这些值通常存储在 `symbol table` 中，并且在语法分析或后续的编译阶段可以进一步访问。

举例来说，对于以下的代码：

```
1 let greeting: String <- "Hello world!";
```

- `greeting` 被词法分析器识别为 `OBJECTID`，它的值可能会被存储在 `symbol table` 中以供后续使用。
- `String` 被识别为 `TYPEID`，它的值也存储在 `symbol table` 中。
- `"Hello world!"` 被识别为 `STR_CONST`，与此 `token` 关联的值是实际的字符串内容 "Hello World!"。

这种机制允许 `Bison` 在解析过程中访问和操纵这些值，从而为下一编译阶段提供丰富的上下文信息。

错误处理

这部分是这次作业的难点，主要是根据PA2的错误处理部分：

- 如果类定义中有错误，但该类正确终止，并且下一个类的语法正确，则解析器应该能够在下一个类定义时重新启动。
- 类似地，解析器应该从feature（继续到下一个feature）、let绑定（继续到下一个变量）和{...}块中的表达式中的错误中恢复。

这两部分对应如下三种含义：

- 如果处理class时报错，应该忽略该错误，然后处理后续的class。
- 如果处理feature时报错，则忽略该错误，然后处理后续的feature。
- 如果处理let时报错，则忽略该错误，然后处理后续的let。

对应的代码：

```
1 CPPclass
2   : CLASS TYPEID '{' feature_list '}' ';'
3   { $$ = class_($2,idtable.add_string("Object"),$4,
4     stringtable.add_string(curr_filename)); }
5   | CLASS TYPEID INHERITS TYPEID '{' feature_list '}' ';'
6   { $$ =
7     class_($2,$4,$6,stringtable.add_string(curr_filename)); }
8   | CLASS error ';' class
9   { $$ = $4; }

10 feature_list
11   : /* empty */
12   { $$ = nil_Features(); }
13   | feature_list feature
14   { $$ = append_Features($1, single_Features($2)); };
15   | feature_list error ;
16   { $$ = $1; }

17 let
18   : OBJECTID ':' TYPEID IN expr
19   { $$ = let($1, $3, no_expr(), $5); }
20   | OBJECTID ':' TYPEID ASSIGN expr IN expr
21   { $$ = let($1, $3, $5, $7); }
22   | OBJECTID ':' TYPEID ',' let
23   { $$ = let($1, $3, no_expr(), $5); }
24   | OBJECTID ':' TYPEID ASSIGN expr ',' let
25   { $$ = let($1, $3, $5, $7); };
26   | error ',' let
27   { $$ = $3; }
```

PA2 vs PA3

两者是如何联系起来的？

1. 接口:

- 当 `flex` 识别出一个记号，它会调用 `yylex()` 函数返回这个记号的类型。
- `bison` 生成的解析器会周期性地调用 `yylex()` 函数来获取下一个记号，并根据给定的文法规则进行解析。

2. 数据交换:

- 当 `flex` 识别出一个记号，它不仅返回记号的类型，还可能设置一个全局变量（如 `yyval`）来传递与该记号相关的附加数据（例如，如果记号是一个整数常量，`yyval` 可能包含该整数的值）。
- 当 `bison` 的解析器调用 `yylex()` 并接收到一个记号时，它可以访问 `yyval` 来获取与该记号相关的数据。

3. 协作:

- `bison` 通常会提供一个函数 `yyerror()` 来报告解析错误。这个函数也可以被 `flex` 生成的词法分析器使用，以报告例如非法字符之类的词法错误。
- 通过合适的错误处理和同步策略，`flex` 和 `bison` 生成的代码可以协同工作，即使在面对错误输入时也能继续进行。

为了将 `flex` 和 `bison` 生成的代码整合到一起，通常的做法是：

1. 使用 `flex` 编写一个 `.l` 文件来定义词法规则。
2. 使用 `bison` 编写一个 `.y` 文件来定义语法规则。
3. 分别使用 `flex` 和 `bison` 工具编译 `.l` 和 `.y` 文件，生成源代码。
4. 将生成的源代码与其他代码一起编译，生成最终的编译器或解释器。

这样，`flex` 和 `bison` 就协同工作，将源代码从文本形式转化为一种更适于进一步处理（如优化和代码生成）的形式。

作用过程对比

`flex` 的作用是一个一个 `char` 的读取，按照正则表达式的规则识别 `token`，而 `bison` 的规则是一个一个 `token` 读取，按照语法规则（这里的语法规则是否也是满足正则表达式？？）识别出一个 `terminal symbol` 或者 `non terminal symbol`。从这个角度来说，它们都只是借助了 NFA/DFA 的强大功能，实现了基于特定规则的匹配。

不同之处在于，`bison` 在每次识别成功之后，都会对 AST 进一步构建。`tree.h` 和 `cool-tree.h` 的头文件就是为我们提供与之相关的所有数据类型以及接口函数（主要展示一些构造函数）对于下面的例子，就调用了 `nil_Features()` 和 `append_Features` 接口函数。

```

1  /* Feature list may be empty, but no empty features in list. */
2  feature_list
3      :      /* empty */
4      { $$ = nil_Features(); }
5      | feature_list feature
6      { $$ = append_Features($1, single_Features($2)); };
7      | feature_list error ';' //看, 错误处理!
8      { $$ = $1; }

```

所以说，tree.h和cool-tree.h的编写对于语法分析来说是很关键的一个部分。

似乎这两个关于AST的生成其实不是靠人写的，而是借助那个cool-tree.aps自动化地生成了所有与之相关的cpp代码？

您的总结大部分是正确的，但有几点需要更详细地说明或微调。

1. flex 和 bison 的角色:

- 正如您所说，flex 的作用确实是按字符读取输入并根据正则表达式的规则识别记号（token）。这个过程称为词法分析。
- bison 则按照给定的上下文无关文法的规则，一个接一个地读取记号并识别更高层次的结构，如表达式、语句等。这个过程称为语法分析。

2. 文法的类型:

- bison 使用的是上下文无关文法，而不是正则文法。这意味着它可以描述比正则表达式更复杂的语言。而正则文法（由正则表达式描述）只是上下文无关文法的子集。

3. NFA/DFA:

- 您是对的，flex 的内部机制确实利用了 NFA/DFA 来实现正则表达式的匹配。但 bison 则使用了不同的算法（如 LR(1)、LALR(1) 等）来实现上下文无关文法的解析。

4. 构建抽象语法树:

- 您正确地指出，bison 的规则中的动作通常用于构建抽象语法树（AST）。tree.h 和 cool-tree.h 确实提供了相关的数据类型和接口函数。

5. 自动化生成:

- 您也是对的，通常 tree.h 和 cool-tree.h 中的代码会由一些工具自动生成，如您提到的 cool-tree.aps。这种自动化生成可以确保与文法规则的一致性，并简化了添加新规则或修改现有规则时的工作。

综上所述，您的总结很准确，只是需要注意 bison 使用的是上下文无关文法，而不是正则文法。而且，bison 的解析算法与 flex 使用的 NFA/DFA 有所不同。

各种文法的对比

- 好吧，课本还是有东西的。确实听听课还是有价值的，高珍老师的ppt要比gpt讲的清晰、严谨很多

- ### Chomsky 文法体系分类

- 文法 $G = (V_N, V_T, S, \delta)$, $\delta : P \rightarrow \alpha$, 其中
 $P \in (V_N \cup V_T)^*$ $V_N (V_N \cup V_T)^*$, $\alpha \in (V_N \cup V_T)^*$ 属于 Chomsky 文法体系
- 该体系对产生式的形式做了一些规定，分为四类，即0型、1型、2型、3型文法

- ### 0型文法

- 0型文法：无限制文法，短语文法
 - 对应的语言：递归可枚举语言
 - 与图灵机等价。

- 例. 下列文法是0型文法

$S \rightarrow aBC|aSBC$

$CB \rightarrow BC$

$aB \rightarrow ab$

$bB \rightarrow bb$

$bB \rightarrow b$

$bC \rightarrow bc$

$cC \rightarrow cc$

$cC \rightarrow c$

1型文法

- 也称上下文有关文法 (CSG: Context-sensitive Grammar)
- 产生式的形式为 $P \rightarrow \alpha$, 其中
 - $|P| \leq |\alpha|$, $\alpha \in (V_N \cup V_T)^*$, $P \in (V_N \cup V_T)^* V_N (V_N \cup V_T)^*$
 - 对应的语言: 上下文有关语言 (CSL: Context-sensitive Language)
 - 若不考虑 ϵ , 与线性有界自动机 (LBA, Linear Bounded Automaton) 等价。
 - 举例: $\alpha A \beta \rightarrow \alpha \gamma \beta$

• 2型文法

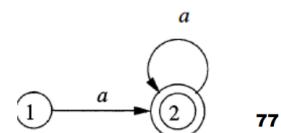
- 也称上下文无关文法 (CFG: Context-free Grammar)
- 产生式的形式为 $P \rightarrow \alpha$, 其中
 - $P \in V_N$, 且 $\alpha \in (V_N \cup V_T)^*$ 核心要求在于production的左边有且仅有
一个非终结符
 - 对应的语言: 上下文无关语言 (CFL: Context-free Language)
 - 对应的自动机: 下推自动机 (PDA: Pushdown Automaton)。

这里的“下推自动机”: 可以简单的理解为: 一个简单的有限自动机加上一个栈结构即可

• 3型文法

- 也称正规文法
 - 右线性文法 (Right-linear Grammar) : 任何产生式为 $A \rightarrow \omega B$ 或 $A \rightarrow \omega$, 其中 $A, B \in V_N$, $\omega \in V_T^*$ 。
 - 左线性文法 (Left-linear Grammar) : 任何产生式为 $A \rightarrow B\omega$ 或 $A \rightarrow \omega$, 其中 $A, B \in V_N$, $\omega \in V_T^*$ 。
 - 等价于正规式
 - 对应的语言: 正规语言
 - 对应的自动机: 有限自动机 (Finite Automaton)。
- 例. 文法 $S \rightarrow aS$, $S \rightarrow a$

对应正规式: a^+ , 或者 a^*a



77

请系统化地解释正则文法和上下文无关文法之间的关系, 对所有文法的分类和包含关系进行分析

文法的分类通常是基于其产生式 (production rules) 的复杂性。下面是常见的文法分类及其关系：

1. 正则文法 (Regular Grammars) :

- **描述能力:** 可以描述所有正则语言。正则语言是最简单的形式语言类别。
- **应用场景:** 正则文法主要用于词法分析和简单的文本模式匹配。
- **产生式示例:** $A \rightarrow aB, A \rightarrow a$
- **实现机制:** 可以由确定性有限自动机 (DFA) 或非确定性有限自动机 (NFA) 来识别。
- **常用工具:** 正则表达式。

2. 上下文无关文法 (Context-Free Grammars, CFG) :

- **描述能力:** 比正则文法的描述能力强，但不能描述所有的形式语言。例如，它不能描述所有需要平衡括号的语言。
- **应用场景:** 上下文无关文法常用于编程语言的语法分析阶段。
- **产生式示例:** $A \rightarrow aBb, B \rightarrow b$
- **实现机制:** 可以由推导自动机或LR解析器来识别。

3. 上下文相关文法 (Context-Sensitive Grammars) :

- **描述能力:** 比上下文无关文法的描述能力强。
- **应用场景:** 上下文相关文法在实际编译器设计中使用较少，但它们可以描述某些在CFG中不可能描述的语言。
- **产生式示例:** $aAb \rightarrow aBb$ (只有在A前有a时，A才可以转换为B)
- **实现机制:** 由线性有界非确定性图灵机来识别。

4. 递归可枚举文法 (Recursively Enumerable Grammars) :

- **描述能力:** 这是描述能力最强的文法类型，可以描述任何由图灵机可识别的语言。
- **应用场景:** 由于其复杂性，在实际编译器设计中很少使用。
- **实现机制:** 由图灵机来识别。

关系与包含性:

文法的描述能力有一个清晰的包含关系：正则文法是上下文无关文法的子集，上下文无关文法是上下文相关文法的子集，而上下文相关文法是递归可枚举文法的子集。或更形象地说：

1| 正则文法 \subseteq 上下文无关文法 \subseteq 上下文相关文法 \subseteq 递归可枚举文法

简而言之，当我们从正则文法向递归可枚举文法移动时，文法的描述能力增加，但文法的处理和解析变得更加复杂。

正则文法vsCFG

结合具体实例说明上下文无关文法和正则文法的区别？在我看来，它们形式上是一样的，只是具体的文法内容不同？

ChatGPT

上下文无关文法（CFG）和正则文法（RG）的确都是用于描述形式语言的，但它们的能力和适用场合是有区别的。首先，让我们通过具体的实例来理解这两者的不同。

正则文法（Regular Grammar）：

- 只能使用右线性（右递归）或左线性（左递归）的产生式。
- 通常用于描述可以通过有限状态机（FSM）进行识别的语言。

示例：

描述所有由零或多个 `a` 组成的字符串的语法：

```
1 rustCopy code
2 S -> aS | ε
```

上下文无关文法（Context-Free Grammar）：

- 可以描述更复杂的结构，如嵌套结构。
- 通常用于描述可以通过推导自动机（如LL或LR解析器）识别的语言。

示例：

描述所有正确配对的括号字符串的语法：

```
1 rustCopy code
2 S -> (S) | SS | ε
```

对比和说明：

1. 描述能力：

- RG 可以描述所有正则语言，而 CFG 可以描述更多，包括一些非正则语言。
- 例如，上述的配对括号语言是非正则的，因此不能用正则文法（或正则表达式）描述，但可以用上下文无关文法描述。

2. 形式：

- 尽管它们的形式看起来相似，但 CFG 允许更复杂的产生式。
- 在 RG 中，每个产生式的右侧要么是非终端符号后跟终端符号，要么是终端符号后跟非终端符号，或只有终端符号。而 CFG 则没有这种限制。

3. 解析方法:

- RG 可以使用有限状态机 (FSM) 来解析，而 CFG 则需要更复杂的解析技术，如递归下降解析或LR解析。

4. 应用场景:

- RG 通常用于词法分析阶段，例如在编译器前端将源代码分解为标记。
- CFG 用于语法分析阶段，例如在编译器前端建立抽象语法树。

简而言之，上下文无关文法和正则文法在形式上可能看起来类似，但它们的描述能力和应用场景有很大的不同。CFG 可以描述 RG 无法捕获的某些复杂结构。

所以说，它们形式上是一样的，只是具体的文法内容不同。我想知道具体的不同之处在哪，它们分别对于文法的书写有何区别

ChatGPT

对，您的理解是对的：正则文法 (RG) 和上下文无关文法 (CFG) 确实在形式上都使用产生式，但是它们的产生式有一些固有的约束和特点，这些约束和特点决定了它们描述语言的能力。

正则文法 (RG) :

- 右线性（或右递归）

：每个产生式的右侧由一个终端符号和一个非终端符号（或无）组成。

```
1 cssCopy code
2 A -> aB | a
```

- 左线性（或左递归）

：每个产生式的右侧由一个非终端符号和一个终端符号（或无）组成。

```
1 cssCopy code
2 A -> Ba | a
```

- 正则文法不允许描述中间结构，例如嵌套结构。

上下文无关文法 (CFG) :

- 没有上述正则文法中的那些限制。每个产生式的右侧可以是任意组合的终端符号和非终端符号。
- 可以描述复杂的中间结构，例如嵌套结构。

```
1| rustCopy code  
2| S -> asb | ε
```

这描述了形如

```
1| aabb
```

,

```
1| aaabbb
```

等具有平衡的

```
1| a
```

和

```
1| b
```

的语言。

不同之处:

1. 产生式的形式:

- 正则文法的产生式具有特定的形式：要么右递归，要么左递归。CFG 没有这种形式限制。

2. 描述能力:

- CFG 能够描述一些 RG 无法描述的语言，例如需要平衡括号或其他嵌套结构的语言。

3. 嵌套结构:

- RG 不能描述任何形式的嵌套或递归结构，而 CFG 可以。

结论:

尽管 RG 和 CFG 在形式上都使用产生式，但它们在描述能力上有很大的不同，这主要是由于它们对产生式形式的不同约束所决定的。CFG 由于其更大的灵活性，通常用于描述编程语言的语法，而 RG 由于其简单性，通常用于描述更简单的模式，如正则表达式中使用的模式。

CFG 和 LL (1) LR LALR之间的关系

LL(1)文法，LR文法，都是上下文无关文法的子集，都是CFG的在不同约定下的，便于解析的写法，对吗？

是的，您的理解是正确的。LL(1)文法和LR文法都是上下文无关文法（CFG）的子集，它们的设计是为了使解析更为简单和直接。让我们更具体地解析一下：

1. LL(1) 文法：

- LL 代表从左到右扫描输入（第一个L）并从左到右产生派生（第二个L）。
- (1) 代表使用一个符号的前瞻来决定下一步的解析动作。
- LL(1)文法特别适合**递归下降解析**，这是一种简单的、可以手写的、顶部向下的解析方法。
- LL(1)文法要求对于给定的非终端和输入符号，必须有一个明确的产生式来应用，也就是说，它不允许有二义性。

2. LR 文法：

- LR 代表从左到右扫描输入 (L) 并从右到左产生派生 (R)。
- LR**解析器**是一种底部向上的解析器，它构建一个解析树，从叶子开始并向上到根。
- LR文法比LL(1)文法能描述的语言范围更广泛。
- 常见的LR解析器有：SLR、LALR、Canonical LR等，其中**LALR** (Look-Ahead LR) 是很多实际工具（如 Yacc 和 Bison）中最常用的。

3. 关于子集关系：

- 并不是所有的上下文无关文法都是LL(1)或LR(1)。这两者都是CFG的子集，并且它们都有自己的限制。
- 通常，LR文法的范围更广，因此LR解析器可以处理更多的上下文无关文法。

4. 目的和应用：

- 这两种文法的制定是为了简化解析。特定的解析技术可能只适用于某种特定类型的文法。
- 为了使一个给定的CFG适合特定的解析技术（如LL或LR），通常需要重写或修改原始的CFG。

综上所述，您是对的：LL(1)和LR文法都是上下文无关文法的子集，它们都是为了便于特定的解析技术而定义的特定形式的CFG。