rtex(Runnable TEX)

- 解释型强类型脚本语言
- 基于flex(https://github.com/westes/flex/)和bison(https://www.gnu.org/software/bison/)实现
- repo: https://github.com/supplient/MyBisonCompiler
- Author: 赵智源 ZY2006166

一、什么情况下会想要使用rtex?

让我们从一个情景出发。小赵亲爱的数学老师布置了一个作业：已知有10行10列学生，若每个学生的行编号是i，列编号是j，编号从0开始，则其分数为$10i + 6cosj$，现求所有学生分数的总和。

小赵知道老师布置这个作业并不是想让他求100次，而是希望他用一些高超的数学技巧来简单得到结果，但很可惜，小赵是计算机学院的学生，这学期他数学课几乎一分钟都没听，一直在做计组。所以小赵还是编写了程序来求解这个问题。他遵循了这样的步骤：

1. 在word内写下数学公式$s = \sum_{i=0}^{9} \sum_{j=0}^{9} (10i + 6cosj)$
2. 编写代码，用一个二重循环计算结果
3. 执行代码，输出计算结果
4. 把输出的结果复制黏贴到小赵的word文件里面

小赵就这样做完了今天的作业，老师给了他这次作业满分。

第二天，小赵亲爱的数学老师又布置了新的作业，令他惊喜的是这次作业几乎和昨天的作业一模一样，仅仅只是分数的计算公式改为$10j + 2sini$。他用了20秒修改了程序，并将结果覆盖到昨天的word文件中，就这么提交上去了。

但是这次老师给了小赵0分，因为他忘了修改word中数学公式的部分，老师很疑惑他是怎么根据上面的式子得到下面的结果的。

第三天，又是几乎一样的作业布置了下来。这次小赵小心谨慎地先修改了word中数学公式的部分，然后再修改了代码部分。

但是这次依然是0分，因为他忘记重新复制黏贴输出结果了。

第四天，又是一模一样的作业布置了下来。小赵意识到自己的记忆和金鱼差不多，按照之前的步骤那样繁琐地在代码和word中反复切换，对他而言太过困难了。他迫切地想要一种方式能够将代码和文档合二为一。

小赵想起他的室友Mr.Zhao之前写了个markdown的拓展，叫rtex，它似乎就能在文档中编程。

rtex非常容易上手，小赵之前学过latex，也经常使用markdown，对于他而言，这门语言几乎不存在任何的学习成本。他很快就学会了它，并用它写了作业：

1. 小赵的作业源码

已知

```
@@@
let Matrix M_{10, 10};
```

```
M_{i,j} = 3*i+cos(j) #where i=0,1...9 #where j=0,1...9;
@@@

求

@@@
let Real s;
s = \sum_{i=0}^{9} \sum_{j=0}^{9} M_{i,j};
@@@

则$s=$@@s;@@。
```

2. 小赵提交的作业

已知

$$Matrix \quad M_{10\times10}$$
$$M_{i,j} = 3 * i + cos(j) \quad i = 0,1...9 \quad j = 0,1...9$$

求

$$Real \quad s$$
$$s = \sum_{i=0}^{9}\sum_{j=0}^{9} M_{i,j}$$

则s =1362.65。

3. 小赵的结局

rtex让小赵非常满意，以后他只需要修改一次就可以完成作业，而不需要既修改数学公式又修改代码和复制黏贴输出结果了。

但很可惜这次作业中老师是按步给分的，直接输出答案的小赵依然是0分。

二、rtex是什么?

rtex是对mdmath(https://marketplace.visualstudio.com/items?itemName=goessner.mdmath)的一个拓展，而mdmath是对markdown的一个拓展。

mdmath使得可以直接在markdown中插入latex段落，它会在生成html文件时自动处理这些latex段落，例如$M_{2,3}$，经过mdmath会被转换为$M_{2,3}$。

而rtex则是可以在markdown中插入rtex段落，这些段落会被rtex解释器转换为latex段落。也就是如下的流程：

rtex段落中包含的就是rtex代码。

rtex并不是排版语言，虽然所有的rtex代码都有对应的良好排版的输出方式，但是rtex并不是用来控制排版的语言。

rtex代码是可执行代码。rtex解释器会维护符号表、变量堆，所有rtex代码描述的运算都会在解释过程中被实际地执行。

1. 所以，为什么需要rtex?

第一章中的小赵因为他的金鱼脑袋而想要找到一种文档与代码相结合的方式来省时省力。事实上，就算没有那么金鱼脑袋，我们也经常希望能够让“描述执行逻辑的文字”与“实际执行逻辑的代码”能够合二为一。

例如jupyter notebook就是在试着达成这个目的。在jupyter notebook中，文档被分割成一个个块，每个块都可以是代码或者文本，也就是在整份文档中，代码和文本可以交错摆放——并且代码块还是可执行的。

rtex就是受jupyter的启发而诞生的，但rtex更进一步。注意到jupyter中代码依然是代码，在最终生成的文档中，文本与代码依然泾渭分明。rtex想要做的就是抹去这条线——rtex想要在最后的文档中让代码和文本看起来一模一样。

例如，小赵就不可能使用jupyter来实现他的目的，因为他依然需要在文本块中输一遍数学公式，再在代码块中编写代码，而且还会将自己是编程完成作业这一事实暴露给老师。

另一方面，用jupyter编写的文档并不适合给非程序员看，因为它依然要求读者能够读懂代码。而rtex则通过对代码本身进行良好排版，使得非程序员也能阅读。例如，哪怕数学老师从来没学过编程，他也肯定能读懂第一章中最后小赵提交的作业。

总结来说，rtex迎合这样的需求：

- 我需要跑一段代码，然后把代码的执行逻辑和输出结果编撰成一份能直接给非程序员看的文档，可我不想又写文档又写代码。

三、rtex的特点是?

所以，rtex是怎样满足上一章中提到的需求的呢？它以这几点来满足：

- 所有rtex语句都可以被转换成latex，从而被良好排版输出。
- 所有rtex语句都可执行，并且rtex解释器会维护rtex上下文（即符号表、变量堆）。
- rtex代码可以以段落的形式被嵌入到任何markdown文档的任何位置，而不影响文档其他部分。

用第一章中小赵的作业为例，

- 可以被良好排版的rtex代码使得他只需要编写一次rtex代码，最后的排版结果就足以让他的数学老师看懂了，而不需要另外用latex写数学公式。
- 可执行的rtex代码让他不用另外写代码去算结果。
- 可随意插入的rtex代码使得他不需要复制黏贴转换结果、输出结果，而可以直接在文档中插入。

最终，小赵只写了一份文档，就得到了计算结果，同时还得到了一份老师也能看懂的文档。

四、rtex的基本语法

完整的ebnf记法的语法描述请参见“附录一、语法”，本章仅介绍各类语法，而不作形式化描述。

1. rtex程序的构成

你可以在任何markdown文件的任何位置插入rtex段落。插入方法分为块段落(Block Paragraph)和行段落(Inline Paragraph)。块段落以@@@为标记符，行段落以@@为标记符。

例如：

```
@@@
let Real k;
k = 3;
@@@
```

将会插入一个rtex块段落。上述段落最终的转换效果如下：

$$Real \quad k$$
$$k = 3$$

行段落也是同理。$k =$ `@@ k; @@` 会被转换成$k = 3$。

每个rtex段落都是语句(statement)的序列，每个语句都以分号结尾。每个语句又是语段(phase)的序列，语段之间用#拼接，例如：

```
@@@
let Matrix M_{2,2};
M_{i,j} = i+j #where i=0,1...1 #where j=0,1...1;
@@@
```

其中第二行就是由三个语段构成的语句。转换结果是：

$$Matrix \quad M_{2\times 2}$$
$$M_{i,j} = i + j \quad i = 0, 1...1 \quad j = 0, 1...1$$

最终会得到$M = \begin{bmatrix} 0 & 1 \\ 1 & 2 \end{bmatrix}$。

而语段中可以包含若干表达式(expression)，其数量和位置根据语段类型的不同而不同。上文中 `k=3` 的左端的 `k` 和右端的 `3` 分别为左值表达式和右值表达式。

打印语句(Print Statement)

除了上述的程序结构外，出于方便的考虑，还有一个特例，那就是打印语句。语句除了由语段构成，还可以由一个右值表达式直接表示。其效果就是打印出该右值表达式的值。例如 `@@ k; @@` 会将变量k的值打印出来：3。

2. 语段(Phase)

目前rtex有三类语段：类型语段(Type Phase)，赋值语段(Assign Phase)，where语段(Where Phase)。

- 类型语段会在rtex上下文的符号表中插入标识符，并将其标记为指定的类型。例如 `@@ let Matrix K_{3, 3}; @@` ，转化为$Matrix \quad K_{3\times 3}$会声明一个3行3列的矩阵变量。

- 赋值语段顾名思义，会将等号右边的表达式的值赋给等号左边的变量。例如 `@@ K_{0,1}=2; @@` ，转化为$K_{0,1}=2$会将2赋给K的第0行第1列。打印检查一下：$\begin{bmatrix}0&2&0\\0&0&0\\0&0&0\end{bmatrix}$。
- where语段实质上就是循环控制，它会迭代执行它左侧的所有语段，循环变量使用省略号表达式来指定。例如 `@@ K_{i,0}=1 #where i=0,1...2; @@` ，转化为$K_{i,0}=1 \quad i=0,1...2$会将矩阵$K$的第0列全部赋为1。打印检查一下：$\begin{bmatrix}1&2&0\\1&0&0\\1&0&0\end{bmatrix}$。

3. 表达式(Expression)

表达式分为两类：左值表达式和右值表达式。

任何用类型语段声明过的变量名都能形成一个左值表达式，除此之外对矩阵变量的元素的访问也构成一个左值表达式。例如之前声明的实数变量 `k` 就是一个左值表达式，而因为之前声明了一个矩阵变量K，所以 `K_{0,2}` 也是一个左值表达式。

左值表达式可以被赋值修改。与之相对的，右值表达式不可以被赋值修改。

右值表达式就是表示一堆运算的结果，不做赘述，只介绍两类rtex特有的右值表达式。

if表达式

rtex目前并不允许语段级别的条件控制，主要是因为我没想到好的对应的排版方法。目前的条件控制是在表达式级别的。例如：

```
@@
k = (
    2, if K_{0,1}>2 #
    1, else
);
@@
```

会被转换为：

$$k=\begin{cases}2,if K_{0,1}>2\\1,else\end{cases}$$

其结果因为$K_{0,1}$之前被我们赋为了2，所以$k=1$。

sum表达式

这个在小赵同学的作业中也被使用了，它能够实现表达式级别的循环控制。它的存在主要是因为排版很好看，所以我加上去了。例如：

```
@@
k = \sum_{i=0}^{2}\sum_{j=0}^{2} K_{i,j};
@@
```

会被转换为：

$$k = \sum_{i=0}^{2}\sum_{j=0}^{2} K_{i,j}$$

其结果为$k = 5$。

4. 矩阵相关

可以直接以字面值的形式给矩阵赋值：

```
@@@
let Matrix B_{2,2};
B = [(2,2)
    0, 1,
    2, 3
];
@@@
```

转换为：

$$Matrix \quad B_{2\times2}$$
$$B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

结果为$B = \begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$

矩阵乘法目前打印出来不太好看，但也实现了：

```
@@@
let Matrix C_{2,2};
C = [(2,2)
    3, 2,
    1, 0
];
C = C * B;
@@@
```

转换为：

$$Matrix \quad C_{2\times2}$$
$$C = \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}$$
$$C = C * B$$

其结果$C = \begin{bmatrix} 4 & 9 \\ 0 & 1 \end{bmatrix}$。

还有矩阵转置：

```
@@@
C=C^T;
@@@
```

转换为：

$$C = C^T$$

得到$C = \begin{bmatrix} 4 & 0 \\ 9 & 1 \end{bmatrix}$。

5. 特殊记号

分隔符转义

因为有的时候我们会想要在文档里直接写出rtex代码的源码（例如这份文档），所以会想要打把@@@直接打出来而不被rtex当作段落分隔符，此时可以打\@@@进行转义。

清空上下文

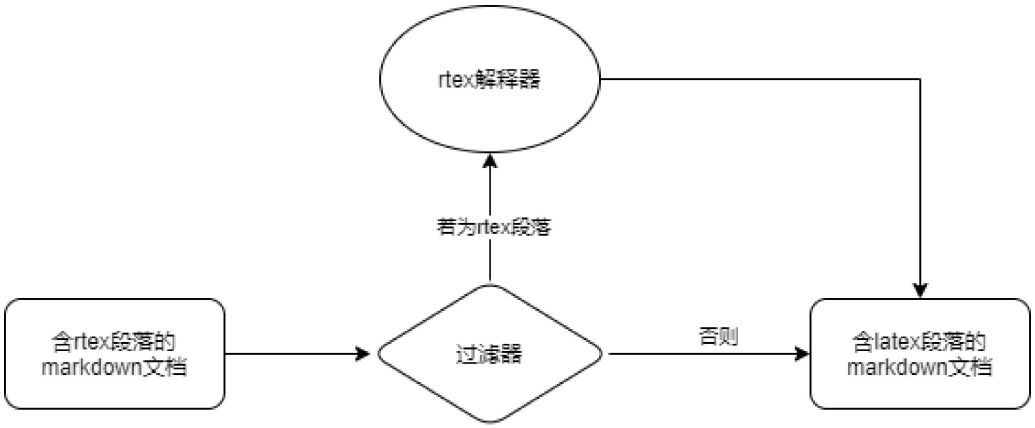
因为一份文档可能很长（例如这份文档），而我们又希望在这份文档的不同区域使用相同的变量名，但指的却并不是同一个变量。也就是我们想要多个上下文。此时可以在文档中用@@##@@来清空rtex上下文。附带一提转义也与@@@类似，是\@@##@@。

五、rtex的解释器实现

本章不会介绍实现细节（毕竟作业是设计语言，而不是写编译器），只会介绍几个与语言相关的点。

1. 文本过滤

通常，文档是直接输给词法分析器的，但是因为rtex是内嵌在markdown中的编程语言，而我不希望还要把代码复制黏贴到一个专门的文件中再执行。所以，我在词法分析器前又用flex做了一个过滤器，只把rtex段落传输给rtex解释器，而其他文本都直接输出。也就是如下的流程：



2. 以语句(statement)为执行批次

因为我不想要引入中间代码，而rtex中的where语段又是后置的循环控制，导致如果我直接在语法解析的过程中执行的话，会无法实现where语段。所以我在语法解析过程中不直接执行，而是将执行过程封装成函数（其实就类似于中间代码），把这一函数传递给上层语法元素，最后在语句这一层次再进行执行。从而实现了这一效果：虽然where语段是后置的语法元素而被较晚地解析，但却能控制前面已被解析的语法元素的执行。

3. 排版与执行共同传递

在rtex解释过程中，解释器不仅需要处理“执行”，还需要处理“排版”，也就是所有语法元素都有两层语义，一层是“它执行后的效果”，一层是“它排版后的结果”，称前者为执行语义，后者为排版语义。

排版语义会受到执行语义的影响。例如上文中提到的“打印语句”，它会将变量的值输出，本质就是之前的rtex的执行语义改变了变量值，从而影响这条语句的排版语义。不过反过来，执行语义并不受排版语义影响。

另一方面，上一条中提到，执行语义需要逐层向上传递函数，最后以语句为批次进行执行。类似的，排版语义是以段落为批次进行执行的。所以在传递过程中，排版语义与执行语义需要共同传递。

六、类型

rtex的类型设计也是从“可被良好排版”这点上出发进行的。因为后端使用的排版语言是latex，它适合用于排版数学元素，所以rtex的类型也是与数学元素一一对应。目前支持的类型为实数(Real)，整数(Integer)，矩阵(Matrix)。

1. 类型转换规则

- 实数与整数不可转换成矩阵，反之也不可。
- 整数可以转换成实数，反之不可。

2. 类型转换方法

rtex的类型转换都为隐式转换，无法显式进行。在计算右值表达式或执行赋值语段时，若发现操作数中有实数类型的，就会尝试将与之运算的操作数转换成实数类型。

七、变量

rtex的变量都是静态束定的，即rtex变量一经声明，其类型就确定，在整个程序生命周期中都不可更改。这是从文档的可读性出发而设计的。

因为最后rtex是要生成一份供人阅读的文档，而人读文档并不一定是线性阅读的，例如助教或老师读我这份文档的时候，很可能直接跳到样例部分阅读。所以如果同一个符号在同一个文章段落中前后却是两种不同的类型，容易让读者产生混乱。

特例有二：

1. where语段、sum表达式中声明的局部变量会覆盖掉之前类型语段声明的全局变量，并且这些局部变量的生命周期、作用域只有这一条语句。
    - 这一特例的设计是因为数学里本就有这样在下标中重复使用同一记号的习惯，所以依然是human readable的。
2. @@##@@记号会清空rtex上下文（即符号表、变量堆）。
    - 这一特例的设计是因为当文档很长时，前后哪怕使用同一个记号，其实也是在指不同的东西的情况是很常见的。

八、样例

最直接的样例就是这份文档，没错，这份文档就是“含rtex段落的markdown”。它的源码在(https://github.com/supplient/MyBisonCompiler/blob/master/test/testfile.md)，经过rtex解释器后生成的“含latex段落的markdown”在(https://github.com/supplient/MyBisonCompiler/blob/master/output/output.md)，再经过mdmath（不是我写的）后生成的html文件（打印后的pdf文件）就是你所见的这份文档。

这里我们做个简单的数据处理作为演示。

1. 源码

```
@@##@@

首先我们有@@ let Matrix M_{10, 10}; @@，它满足@@ M_{i, j} = cos(j+i*10) #where i=0,1...9 #where j=0,1.

然后我们想对矩阵$M$做一个过滤，让所有负项都归零，所有正项都倍增，也就是

@@@
    M_{i,j} = (
        0, if M_{i,j}<0 #
        2 * M_{i,j}, else
    )
    #where i=0,1...9
    #where j=0,1...9
    ;
@@@

经过过滤后，

$M=$@@M;@@

最后我们想得到M的第0行和第3列的内积，也就是

@@@
    let Real s;
    s = \sum_{i=0}^{9} M_{0,i}*M_{i,3};
@@@

得到结果$s=$@@s;@@。
```

2. 结果

首先我们有$Matrix \quad M_{10\times 10}$，它满足$M_{i,j} = cos(j + i * 10) \quad i = 0, 1...9 \quad j = 0, 1...9$。

然后我们想对矩阵M做一个过滤，让所有负项都归零，所有正项都倍增，也就是

$$M_{i,j} = \begin{cases} 0, if M_{i,j} < 0 \\ 2 * M_{i,j}, else \end{cases} \quad i = 0, 1...9 \quad j = 0, 1...9$$

经过过滤后，

$$M = \begin{bmatrix} 2 & 1.0806 & 0 & 0 & 0 & 0.567324 & 1.92034 & 1.5078 & 0 & 0 \\ 0 & 0.0088514 & 1.68771 & 1.81489 & 0.273474 & 0 & 0 & 0 & 1.32063 & 1.97741 \\ 0.816164 & 0 & 0 & 0 & 0.848358 & 1.98241 & 1.29384 & 0 & 0 & 0 \\ 0.308503 & 1.82948 & 1.66845 & 0 & 0 & 0 & 0 & 1.53083 & 1.91015 & 0.533286 \\ 0 & 0 & 0 & 1.11023 & 1.99969 & 1.05064 & 0 & 0 & 0 & 0.601185 \\ 1.92993 & 1.48431 & 0 & 0 & 0 & 0.0442535 & 1.70644 & 1.79973 & 0.23836 & 0 \\ 0 & 0 & 1.34701 & 1.97179 & 0.783714 & 0 & 0 & 0 & 0.880286 & 1.98678 \\ 1.26664 & 0 & 0 & 0 & 0.343435 & 1.8435 & 1.64866 & 0 & 0 & 0 \\ 0 & 1.55337 & 1.89936 & 0.49908 & 0 & 0 & 0 & 1.1395 & 1.99875 & 1.02035 \\ 0 & 0 & 0 & 0.634857 & 1.93892 & 1.46035 & 0 & 0 & 0 & 0.0796418 \end{bmatrix}$$

最后我们想得到M的第0行和第3列的内积，也就是

$$s = \sum_{i=0}^{9} M_{0,i} * M_{i,3}$$

$$Real \quad s$$

得到结果s =5.7477。

附录一、语法

注意这里没有列出正文中提到的特殊记号"@@##@@"，因为本章中只涉及rtex解释器，而不涉及在这之前的过滤器。@@##@@的处理是在过滤器中进行的。

1. Token定义

此处忽略掉下文语法描述中直接用双引号标出来的token，而只列出下文中使用全大写记号的token。

文本	token名
let	LET
\sum	SUM
...	ELLIPSIS
if	IF
else	ELSE
where	WHERE

2. 语法描述

```
program: paragraphs
;

paragraphs: paragraph paragraphs
| paragraph
;

paragraph:  block_paragraph
| inline_paragraph
```

```
;

block_paragraph: "@@@" statements "@@@"
;

inline_paragraph: "@@" statements "@@"
;

statements: statement ";" statements
| statement ";"
;

statement: phases
| right_exp
;

phases: phase "#" phases
| phase
;

phase: type_phase
| assign_phase
| where_phase
;

type_phase: LET TYPE IDENTIFIER
| LET TYPE IDENTIFIER subscript_dim
;

where_phase: WHERE IDENTIFIER "=" ellipsis_exp
;

ellipsis_exp: right_exp "," right_exp ELLIPSIS right_exp
;

assign_phase: left_exp "=" right_exp
;

left_exp: IDENTIFIER
| IDENTIFIER subscript_dim
;

right_exp: right_exp OPERATOR right_exp
| BUILT_IN_FUNCTION "(" right_exp ")"
| INTEGER
| REAL
| list_exp
| sum_exp
| "(" if_exp ")"
| "(" right_exp ")"
| left_exp
| right_exp "^T"
;

list_exp: "[" right_exp_list "]"
| "[" "(" INTEGER "," INTEGER ")" right_exp_list "]"
;

right_exp_list: right_exp "," right_exp_list
| right_exp
;

sum_exp: SUM subscript_cond superscript_cond right_exp
;

subscript_cond: "_" "{" IDENTIFIER "=" right_exp "}"
;
```

```
superscript_cond: "^" "{" right_exp "}"
;

if_exp: if_exp_phases
;

if_exp_phases: if_exp_phase "#" if_exp_phases
| if_exp_phase
;

if_exp_phase: right_exp "," IF bool_exp
| right_exp "," ELSE
;

bool_exp: right_exp OPERATOR right_exp
;

subscript_dim: "_" "{" right_exp_list "}"
;
```