

Table of Contents

AMK语言简介	0
项目时间与工作安排	1
AMK语言设计	2
命题逻辑部分语言设计	3
Lexer设计文档	4
Parser设计文档	5

AMK: Anti-Min-Ke

AMK (Anti-Min-Ke) 是为书写和检查证明的人们而设计、实现的程序设计语言。设计的目标是：

- 标准化证明书写；
- 自动检查证明的正误；
- 提供在线的编辑器提供Web服务做到上面两点；
- 做一定程度上的（半）自动证明。

AMK (Anti-Min-Ke) is a home-made language for people writing and checking mathematical proofs. It aims to

- standardize proof writings,
- automatically check the correctness of proofs, and
- (future) do semi-automatically proof.

具体实现领域

作为一学期的课程项目，我们打算做数理逻辑方面的内容，不失一般性地实现该领域下的功能，之后再考虑拓展领域。

As a course project, we decided to implement AMK in the area of Mathematical Logics (propositional logics). Later it can be extended to other areas of mathematics.

项目源码链接

从 [Github](#) 访问该项目。

Visit the project at [Github](#)

时间和工作安排

开题报告

- 开题报告: 倪盛恺

10.18~10.25: 语言设计

- 语言设计文档: 史舒扬
- 实现工具和阶段划分调查: 倪盛恺
- 可能有参考性的工具 coq 的调研: 张浩千

10.25~11.1: 词法分析

- 词法分析实现: 倪盛恺
- 语法分析调研与草稿: 张浩千
- 工具调研(flex+bison vs ANTLR): 史舒扬

11.1~11.12: 语法分析

- 实现语法分析 (构件AST) : 史舒扬
- 修改词法分析器Lexer代码, 并完成它的文档: 倪盛恺
- SDT过程设计与草稿: 张浩千

11.12~11.18: 语法制导翻译SDT

- 语法制导翻译SDT: 张浩千
- 写语法分析器Parser的文档: 史舒扬
- 继续修改Lexer的Bug: 倪盛恺

11.19: 中期报告

- 中期报告: 史舒扬

11.20~12.10: 错误报告与完善测试

- 完善Translator的SDT过程: 张浩千

- 增加更复杂的例子（添加完整的公理体系）
- Debug
- Parser完善: 史舒扬
 - 错误报告
 - 增加对括号 '()' 的支持
 - 存储AST节点的行号
- 实现外壳及模块的方式: 史舒扬
- Web 编辑器调研: 倪盛恺

12.10~12.24: 进一步的工作

- 增加打印导出功能 (PDF) : 张浩千
- Web编辑器与网站搭建: 倪盛恺
- 尝试实现跳 (一) 步功能: 史舒扬

12.31: 结题报告

- 结题报告: 张浩千
-

Schedule and Work Allocation

Current Status: Dec 10 -- Dec 24 Further Work

- Print Function: zhqc
- Web Interface and server: sanzunonyasama
- One-step inference: bsnsk

Work done

Oct 18 -- Oct 25 : language design

- Language design: bsnsk

- Tool and stage investigation: sanzunonyasama
- Coq investigation: zhqc

Oct 25 -- Nov 1 : lexical analysis

- Lexical analysis using flex: sanzunonyasama
- Syntactical analysis draft: zhqc
- Investigation (Bison vs ANTLR, feature and usage): bsnск

Nov 1 -- Nov 12 : syntactical analysis

- Fix bugs and write doc for lexical part: sanzunonyasama
- Syntactical analysis and build AST: bsnск
- Design Syntax-directed translation and write doc for that: zhqc

Nov 12 -- Nov 18: (First demo) Syntax-directed translation

- Write doc for syntactical part: bsnск
- Syntax-directed translation: zhqc
- Prepare slides for interim report: sanzunonyasama

Nov 19 : Mid-Term Report

- Mid-Term Report

Nov 20 -- Dec 10 : Error Reporting

- Syntax Refinement: bsnск
 - Error Reporting
 - Support for '(')'
 - Storage of Line Number in AST Nodes
- Shell to implement modules: bsnск
- Translator Refinement: zhqc
 - Debug
 - More Complex Examples
- Web Editors Investigation: sanzunonyasama

AMK Language Design: Overview

Drafted by *Shuyang Shi* @ Oct, 2015

下面是关于AMK语言的总述，会出现一些例子，这些例子默认来自于古典命题逻辑领域。具体各领域的內容在各自的说明文件中查阅。

模块 Module

AMK的设计初衷是为了在数学各个领域的证明中发挥相当的作用，由于这样的扩展性，决定采用模块(module)的形式，每个模块对应一个数学领域，包含该领域一些必须的内容等。不导入模块的解释器基本上可以认为只包含了简单的推理逻辑。

标准模块

预置的标准模块中包含的內容有：

- 运算符
- 变量类型（以及默认类型）
- 公理、基本定理及其证明

基本上一个模块对应一个领域，对于有嵌套的领域，用“.”来表示从属关系（见模块导入的例子，logcs模块下有classic 和 intuitionism 两个分支）。

用户定义模块

在导入模块的时候，解释器会根据模块中的证明对其中的定理证明进行验证（公理不验证，仅提示使用者），这样的一方面能每次检查模块是否被修改，保证证明基础的正确性，更重要的是，允许了用户自己撰写模块并使用（用户定义模块）。这一定程度上保证了AMK的可扩展性。

模块导入

模块一旦导入，那么模块中提供的运算符、变量类型、公理定理等都将可以使用。如果导入了不同模块导致使用有冲突，需要在使用的时候用"."标明来自于哪个模块。（当然，目前我们还是暂时考虑一次运行只导入一个模块的情况）

下面是一个在源文件中导入命题逻辑模块的例子（用**import**关键字在文件头导入模块）：

```
import logics.proposition
```

源文件语法

模块源文件 (*.mamk*, module for **amk**) 分为四部分，分别是

- 类型
- 运算符
- 公理
- 基本定理或引理。

这里不仔细展开了，有兴趣的读者可以看完后文的语法介绍以后根据下面的一个例子自行琢磨。有几点注意。

- 分为四大部分，分别是TYPE, OPERATOR, AXIOM, THEOREM
- OPERATOR 中的前缀表示结合性(left, right, nonasc, func)，其中nonasc表示没有结合性，func表示不采用中缀的方式而是类似函数调用传参数的方式使用(e.g. fun(a, b))。
- AXIOM部分的公理不需要证明，THEOREM部分的定理需要证明。

classic.mamk

```

TYPE
statement {default}
set[statement] # [statement] here means there is a list of several (finite) statements
numeric # numeric values like int, float, etc

OPERATOR # operators are arranged in decreasing order of their priorities
nonasc not (statement) --> statement

right wedge (statement, statement) --> statement

right vee (statement, statement) --> statement

right -> (statement, statement) --> statement

right <-> (statement, statement) --> statement

right |- (set[statement], statement) --> statement

right |-| (set[statement], statement) --> statement

AXIOM
axiom1:
require:
define a of statement
define b of list[statement]
conclude: a, b |- a

axiom2:
...

THEOREM
naive_theorem:
require:
define a of statement
define b of statement
conclude:
a, b |- a
proof:
a, b |- a [axiom1]

```

用户代码作为模块 (*)

有时候证明较长，用户书写的时候需要用到另一个文件里的定理等。这区别于标准模块和用户定义模块，因为这并不是领域的拓新，也不会产生新的运算符、类型、公理等，因此，这一类模块尽管也称为模块，但是解释器实现的时候仅仅是将两个文件顺序拼接当成一个新文件而已。

导入的示例（用引号以与之前的模块区分）：

```
import "logic_theorems" # import theorems in file 'logic_theorems.amk'
```

元素 Elements

类型

AMK本身并不支持类型（theorem, lemma, axiom除外），所有类型都来自于模块定义。

AMK提供元类型，即**list**, **set** 分别表示有序、无序的列表（有限数量的元素）。使用方式：

1. 在模块的源代码声明类型，如

`list[typeA] set[typeB]`

2. 运算符默认使用逗号，用

`a, b`

表示a, b两个元素组成的set，用

`[a, b]`

表示a, b两个元素组成的list。

运算符

AMK本身并没有运算符，所有的相关运算符都来自模块定义（除了numeric，它在模块中声明，一旦声明不需在模块中再定义即可使用）。

运算符的表示理论上可以使用任意符号、单词，甚至Unicode符号如 。暂时使用ASCII码的符号、字母等，后期可以加上运算符别名以适应不同输入环境下的用户需求。

公理、引理、定理

在AMK中，**theorem**, **lemma**, **axiom** 等关键字分别用作定理、引理和公理的声明。作这样的区分，主要是为了用户的书写习惯考虑，辅助用户理清楚要证明的是什么，什么是辅助。由于这些“x理”都有用户命名，因此在使用的时候可以不标明前缀是三种类型中的哪一种。

不失一般性，下面以**theorem**为例说明语法。定义方式类似于Python里的函数，首行为关键字**theorem**和定理名，后加冒号。之后两行分别用"**require:**"和"**conclude:**"标明已知条件和证明的结论。已知条件可以用

- 表达式；
- 变量的定义；
- 定理、引理等的名字或标号。

结论在表达清楚的情况下建议直接写表达式。当结论非常复杂的时候可以用缩进的**where**来声明变量代替长而重复出现的表达式。

再之后"**proof:**"后跟证明过程。具体的证明格式后面再详述。当声明了定理不跟**proof**引导的证明过程的时候，类似函数的声明，表示在后面会找到关于这个定理的证明。这一点是出于证明习惯的考虑，有时候我们会喜欢用分析法，从结论往回分阶段推倒，那么就需要留下一个假设等待后面证明。

下面是一个简单的例子（不要在意例子的具体内容）。

```
theorem A:
    require:
        P, R, Q
    conclude:
        a, b |- c
    where
        state a = R -> P Vee Q
        state b = not (R wedge Q)
        state c = R -> P
    proof:
        a, b |- c [lemma B]
```

注释

采用#作为注释的符号，注释内容从#开始一直到行末。可以看下面的例子：

```
I am not a comment # I am a comment
```

证明过程

下面是一个简单的证明示例：

```

theorem T:
  require:
    define a of statement
    define b of statement
  conclude:
    a -> b |- a -> (a wedge b)
  proof:
    a -> b, a |- a -[axiom belong] <1>
    a -> b, a |- a -> b <2> # using of axiom can be omitted
    a -> b, a |- b -[axiom ->- : <1><2>] <t3>
    a -> b, a |- a wedge b -[:<t3>] <4>
    a -> b |- a -> (a wedge b) [4]

```

首先来看写在定理 / 引理的"proof:"后面的内容，每行一句推理过程。

- 每行是一个已定义类型的表达式（或者变量），标示该表达式已经由require的内容和前面的推理步骤，在该定理内得到证明。
- （可选）尖括号<>内标示的是推理过程的标号，可以用数字、字母、下划线进行组合。为了方便书写，推理过程的标号**仅在当前函数内有效**。标号在一行的最右边（注释除外）。
- （可选）符号对-[和]内标示的是使用的定理、引理。
 - 如果是公理，可以不写清楚具体是哪条公理。但是定理、引理的使用必须明确指明其名字和应用的变量（这点和手写证明的要求是一样的）。
 - 括号内的内容可以用'::'分隔作为两部分
 - 前面是使用的定理 / 引理 / 公理的名字，前缀 axiom / lemma / theorem 可以省略。
 - 后面是对那几行推理得到的结果使用该定理 / 引理 / 公理，用标号**按定理条件的顺序**指明。
 - 如果只有前一半，可以省略'::'，但是只有后一半不能省略'::'

编写 Coding

缩进与括号

类似Python，采用缩进作为层次结构的标明，不使用大括号。定理的require, conclude, proof等都要相对theorem缩进，具体的where, 证明内容等也要相应缩进。

注意，这里缩进我们**仅使用TAB**。

大小写与命名规则

AMK大小写敏感，内置关键字等统一用小写。推荐用户使用小写字母和符号、数字等命名。变量、定理等的命名需要

高亮与编辑器支持

已有`.amk`和`.mamk`的最基本Vim语法高亮插件。安装方式：

```
$ cp vim-plugin/mamk.vim ~/.vim/syntax/  
$ cp vim-plugin/amk.vim ~/.vim/syntax/  
$ cp vim-plugin/filetype.vim ~/.vim/
```

支持各模块类型的高亮插件、自动补全插件后期可以补上。

AMK Language Design: "Mathematical Logics" Module

Drafted by *Shuyang Shi* @ Oct, 2015

前言

我们希望，能从数理逻辑(Mathematical Logics)这一领域作为例子，实现AMK的第一个模块，从而证明该语言的可行性和实用性。相比分析等领域，逻辑较为简单，但是具备了基本的证明要素，十分适宜作为试水的模块。

继承

继承AMK本身的所有语言特征。

具体的定义与说明

命题逻辑部分，参见 `modules/logics/proposition.mamk`。

Source Code Example

A nonsense proof:

```
theorem A:  
    require: P, R, Q  
    conclude: a, b |- c  
        where  
            state a = R -> P Vee Q  
            state b = not (R wedge Q)  
            state c = R -> P  
    proof:  
        conclude a, b |- c [lemma B]
```

A sensible proof:

```
theorem T:  
    require: a, b  
    conclude: a -> b |- a -> (a wedge b)  
    proof:  
        a -> b, a |- a [axiom belong] <1>  
        a -> b, a |- a -> b <2> # using of axiom can be omitted  
        a -> b, a |- b [axiom ->- 1,2] <t3>  
        a -> b, a |- a wedge b [t3] <4>  
        a -> b |- a -> (a wedge b) [4]
```

Documentation:Lex

By *Shengkai Ni* @ 2015

Keyword

```
define import
therom axiom lemma
require conclude proof
where of
not wedge vee
```

Operator

Spaces in operators are not allowed.

```
not vee wedge
|- | -> <->
```

Identifier

Identifiers are strings that

1. starts with letters;
2. may contain letters, numbers, and '_'.

Label

Labels contain letters, numbers and '_'.

Labels are wrapped with '<' and '>'.

Spaces in labels are not allowed.

Here are some examples below.

```
<1>
<belong>
<_____yoooooo_____>
<3q_>
```

Other Symbol

The symble to show which lemma or conclusion appeared before and now is used in this proof line.

```
-[]

Example:
a |- b -[theorem t: <1> <2>] <1>
```

Parrens.

```
(  
)
```

Comments

```
#  
Example:  
#This is a comment.
```

Square brackets

```
[  
]
```

Colons

```
:
```

Documentation: Syntax

By *Shuyang Shi* @ Nov 2015

Data Union

The union for data passing during **yylex()** is defined as follows.

```
/* union */
%union {
    char *str;
    struct ast_node *ptr;
};
```

AST Node

Struct Definition

struct ast_node is defined as follows.

```
/* node in AST */
struct ast_node {
    /* info of this node */
    enum node_types node_type;
    void * data;

    /* links */
    int num_links;
    struct ast_node ** links;
};
```

Explanation for Node Types

For different node types, we define the member variables as the following table indicates.

node_type	data	num_links	links
nd_program	NULL	2	import_part, proof_part
nd_import_part	NULL	n	import_expr
nd_proof_part	NULL	n	proof_block
nd_import_expr	(char *) str	0	NULL
nd_rich_expressions	NULL	n	rich_expr
nd_expressions	NULL	n	expr
nd_proof_block	(char *) name	3	proof_require, proof_conclude, proof_body / NULL
nd_rich_expr	(struct ast_node *) expr	2	theorem_ref, label
nd_ref_body	(char *) theorem_name	2	ref_pref, ref_labels
nd_ref_labels	NULL	n	identifier
nd_expr	(enum operators) op (NULL if none)	k = 1 or 2	(ast_node) expr / (char) var
nd_of_expr	var	1	type
nd_of_expressions	NULL	n	of_expr
nd_proof_req	NULL	2	of_expressions, expressions
nd_type	(char *) identifier / "set" / "list"	0 or 1	(sub-)type

Operator Precedence and Associativity

```
/* operator precedence and associativity */
%right dget
%right get
%right dcontain
%right contain
%right vee
%right wedge
%nonassoc not
```