Insight into DBMS:

Design and Implementation

Chapter 4 A: SQL to File Operations

Parse SQL

孔令波
mlinking@126.com
+86 15010255486

Outline of the chapters covered

- **Introduction**
- **Overview of the projects**
- **Demonstration of Development environment**
  - Watch and practice by yourself
- **My understanding about (R)DBMS**
  - History and D&I
- **SQL translation with 2 conversions**
  - SQL → RA (Relational Algebra)
  - RA → Sequence of File operations
- **Transaction control**
- **Deeper**
  - File, (R)DBMS, ERP, DW, Big Data (No SQL, SQL again)
  - SQL on MPP and Hadoop (Greenplum, **HAWQ**)

- **Sit back, but not relax!**
  - We'll start the most exciting yet challenging part – SQL translator!

Chapter 3: Parse SQL

- **SQL as a language – 4 GL (Generation Language) – follows some rule**
  - **Syntax** should be followed – How is syntax defined?
  - **Automata** are used to **model syntax**, and could **verify** if a sentence *s.t* a given language?
  - Precursors proposed a **framework** to derive operations from the legal sentence
- **3rd training project – a Naïve SQL parser**

  Select B,D

  From R,S

  Where R.A = “c” $\wedge$ R.C=S.C

Friendly Interface – SQL

But HOW?!

CREATE OnlineRetail (int InvNo, … , float pr
UPDATE OnlineRetail **SET** price =25.5 **WHERE** C… and
InvNo=536365 and StockCode='85123A';
SELECT AVG(quantity*price) **FROM** OnlineRetail WHERE InvNo=536365
…

To get result determined by friendly SQL of course is carried out by file operations!

Translation here is based on compiler!

FILE System supported by OS

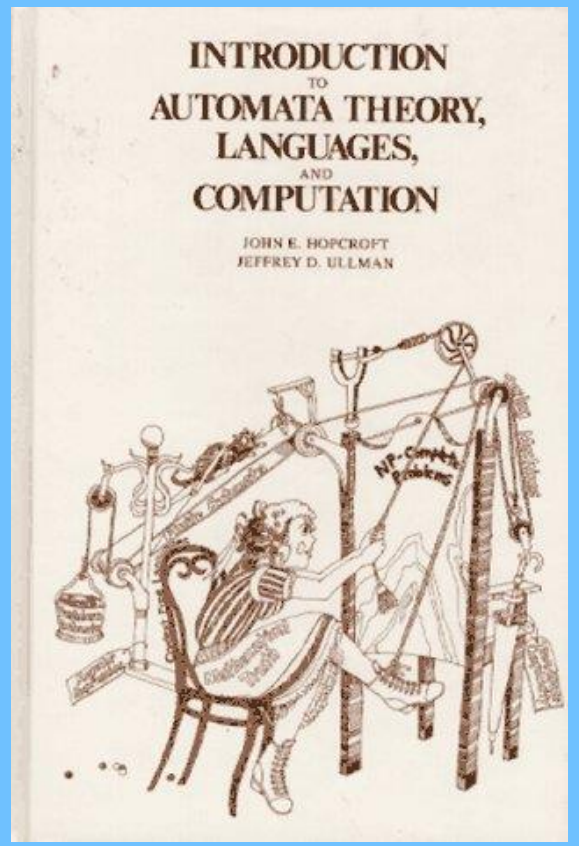
```
struct{
  time;
  money;}
```

File

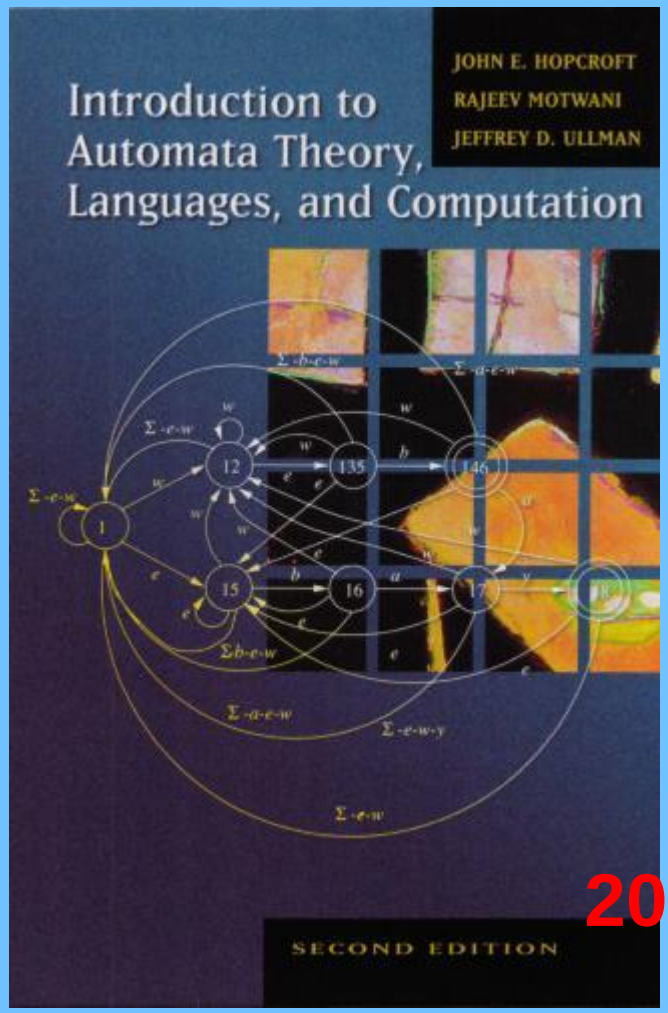
File operations (learn in OS):

- Open
- Seek
- Read/Write (fscanf(),sprintf(), …)
- EOF/Exist
- Close

Compiler? – Understanding Languages



1979

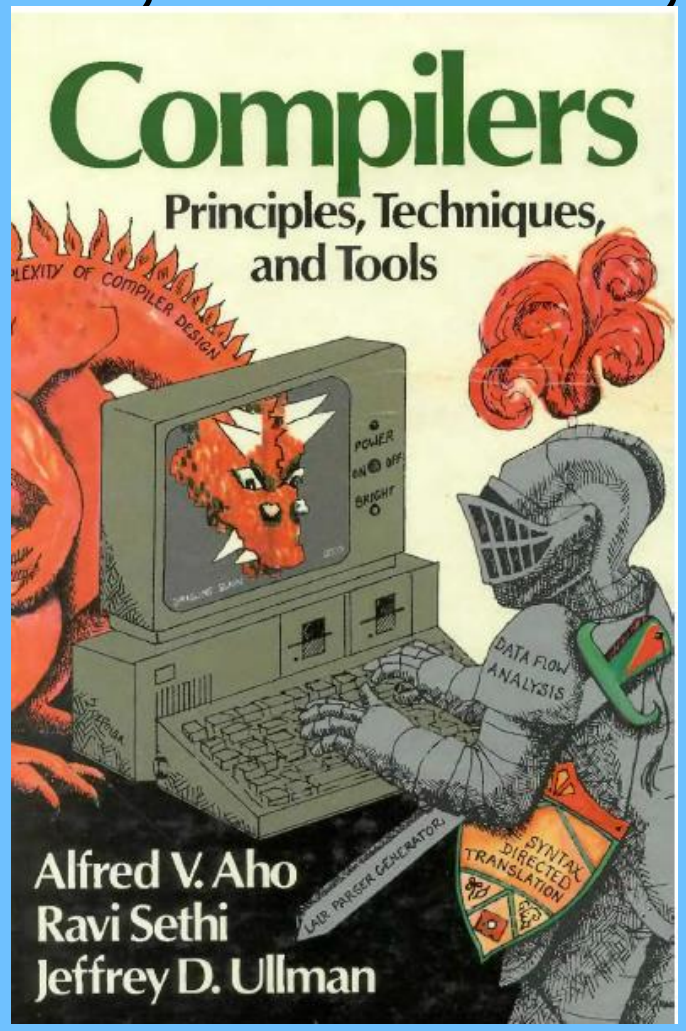


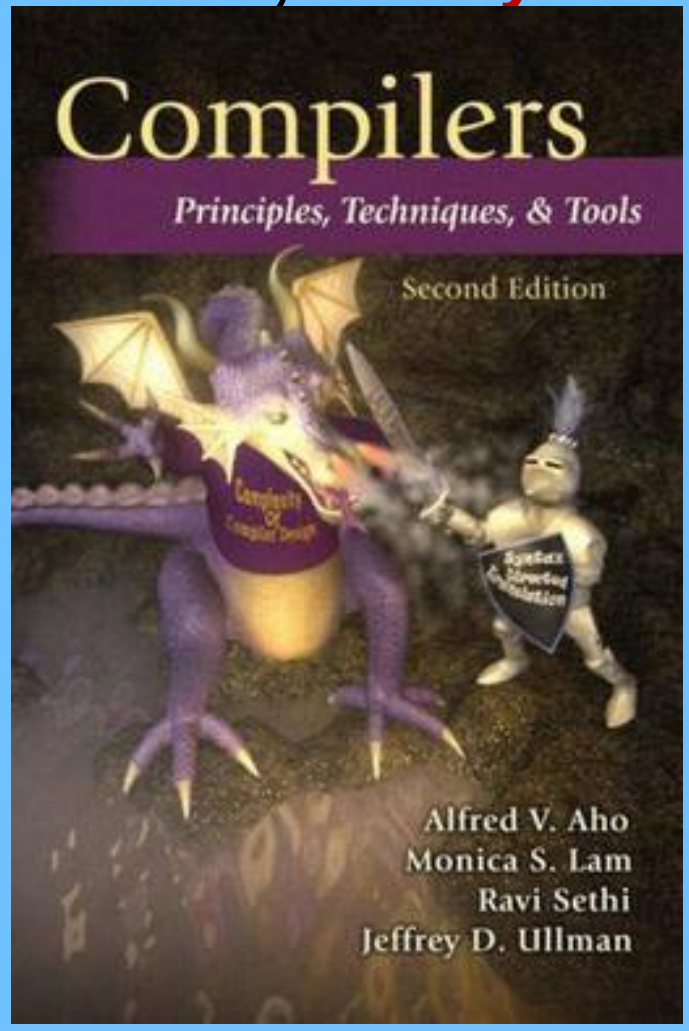
2001

Dragon book [所谓的“龙书”]

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

1986

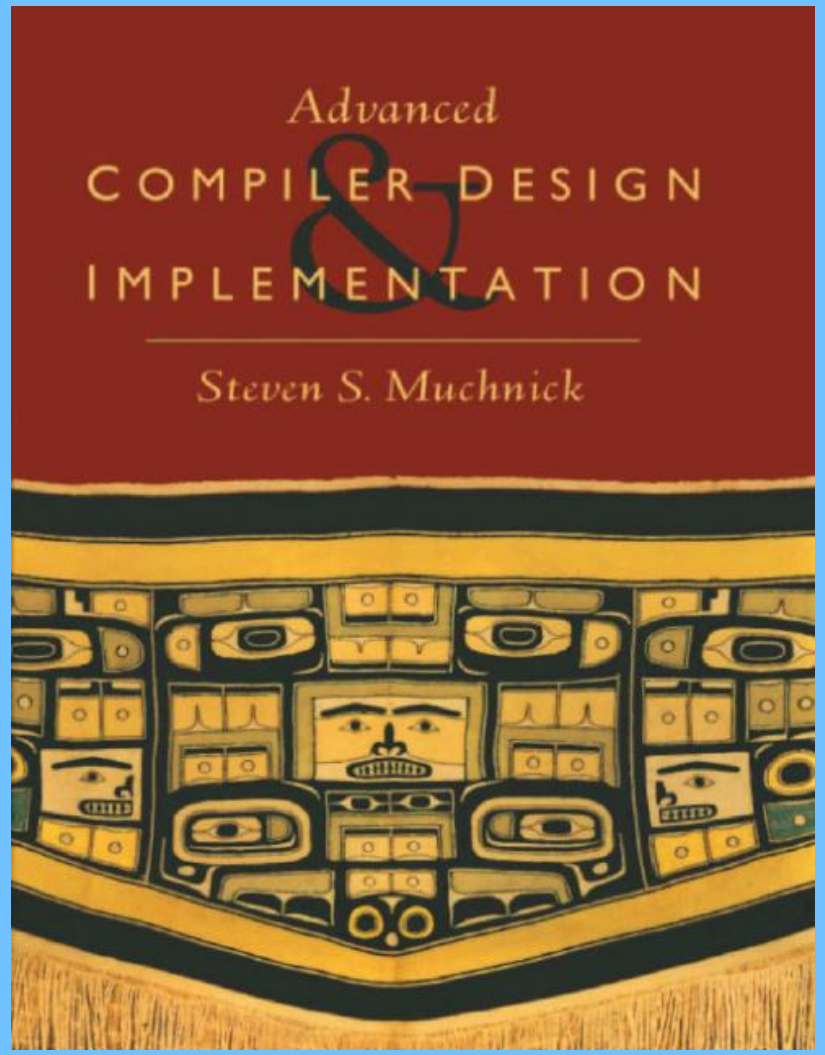




2007

- **Engineering a Compiler (2nd)**

Whale Book [所谓的“鲸书”]

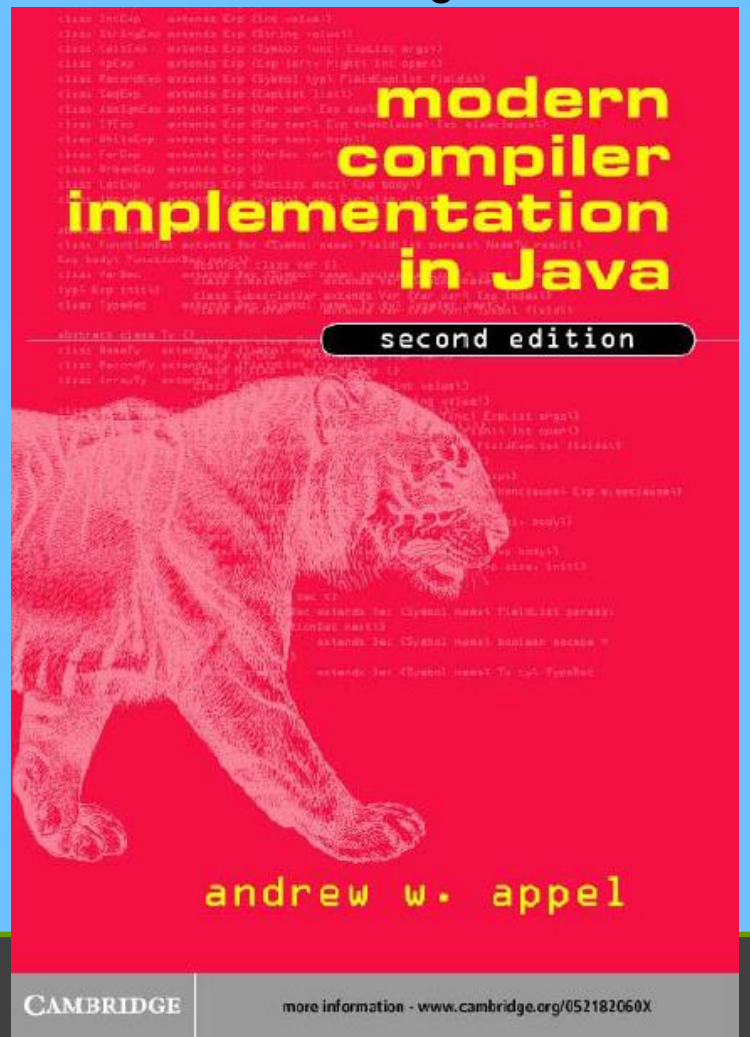


- **Advanced Compiler Design and Implementation**

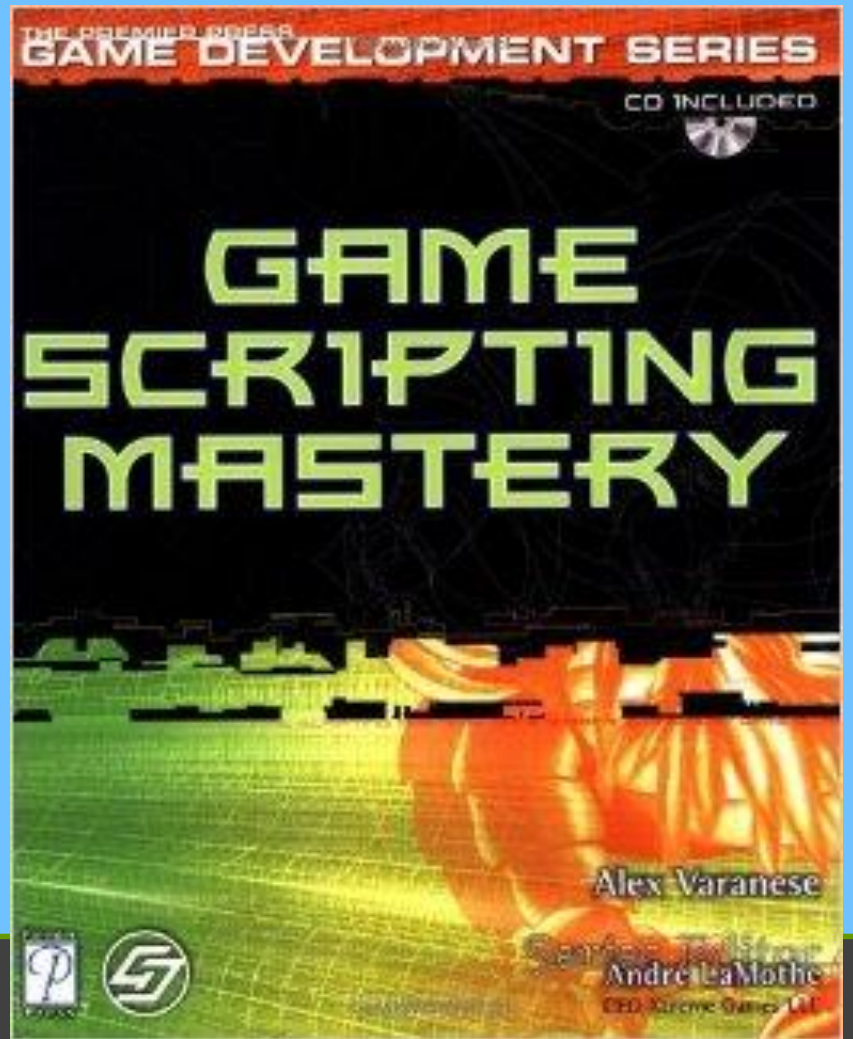
understanding Language is IMPORTANT!

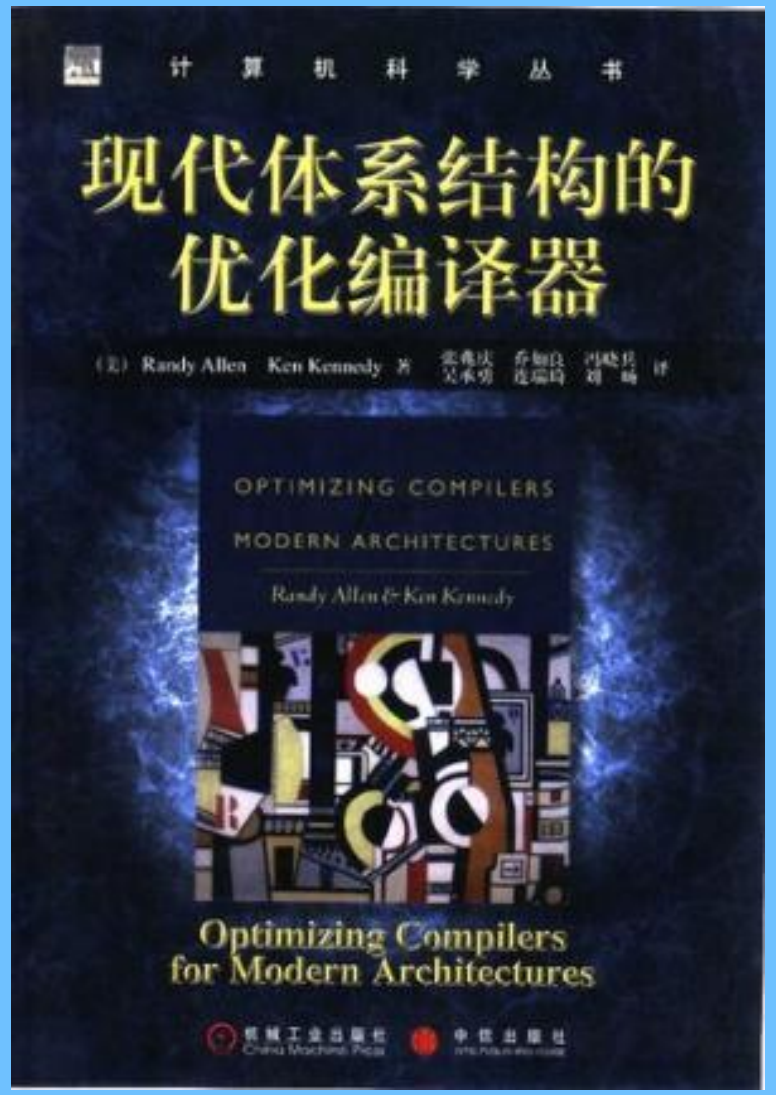
- **Compiler! Even Game Intelligence!**

So-called Tiger book “虎书”



Game Scripting Mastery

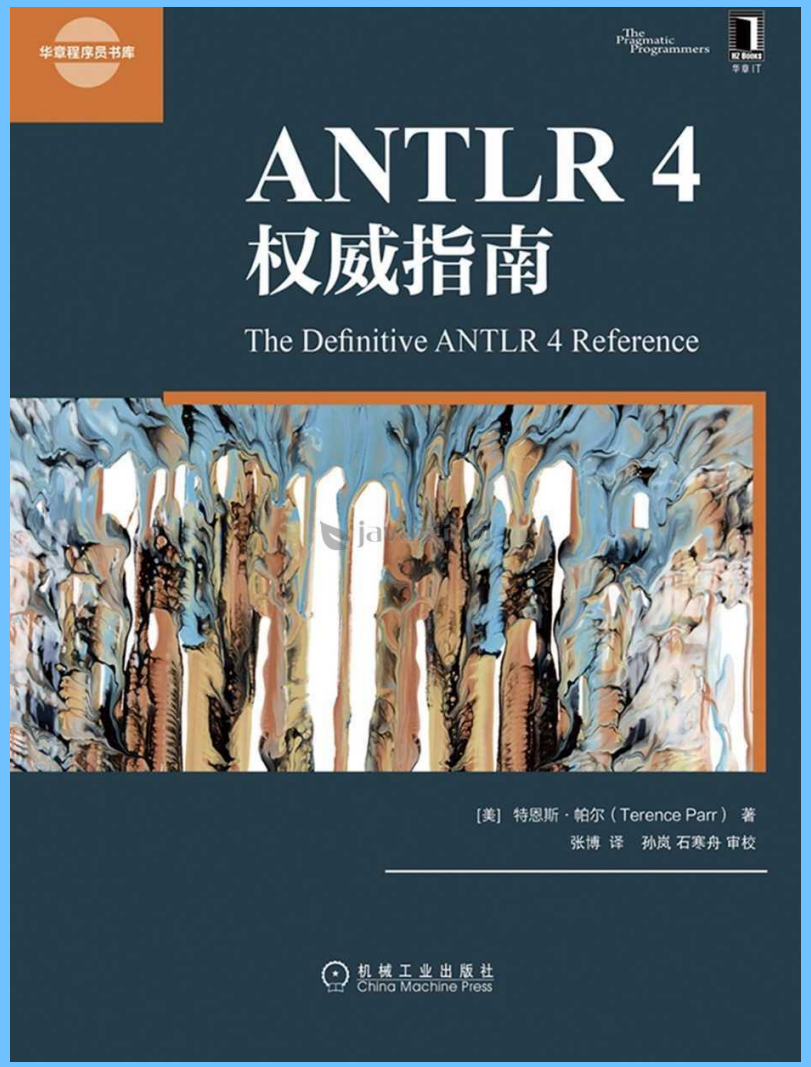




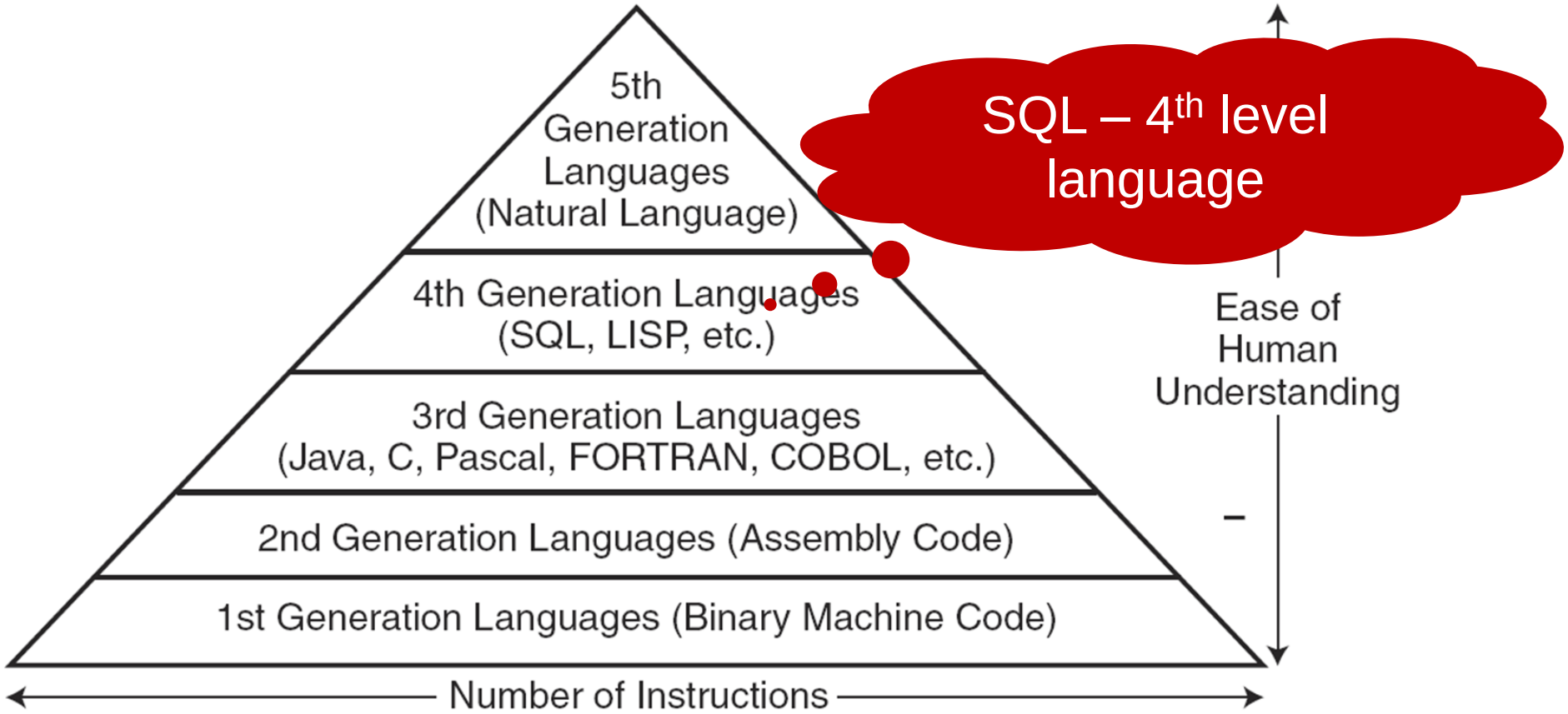
- 现代体系结构的优化编译器
- *[美] Randy Allen*

- 2004
- 机械工业出版社

- flex & bison - Unix Text Processing Tools
- *John Levine*
- O'Reilly Media
- 2009



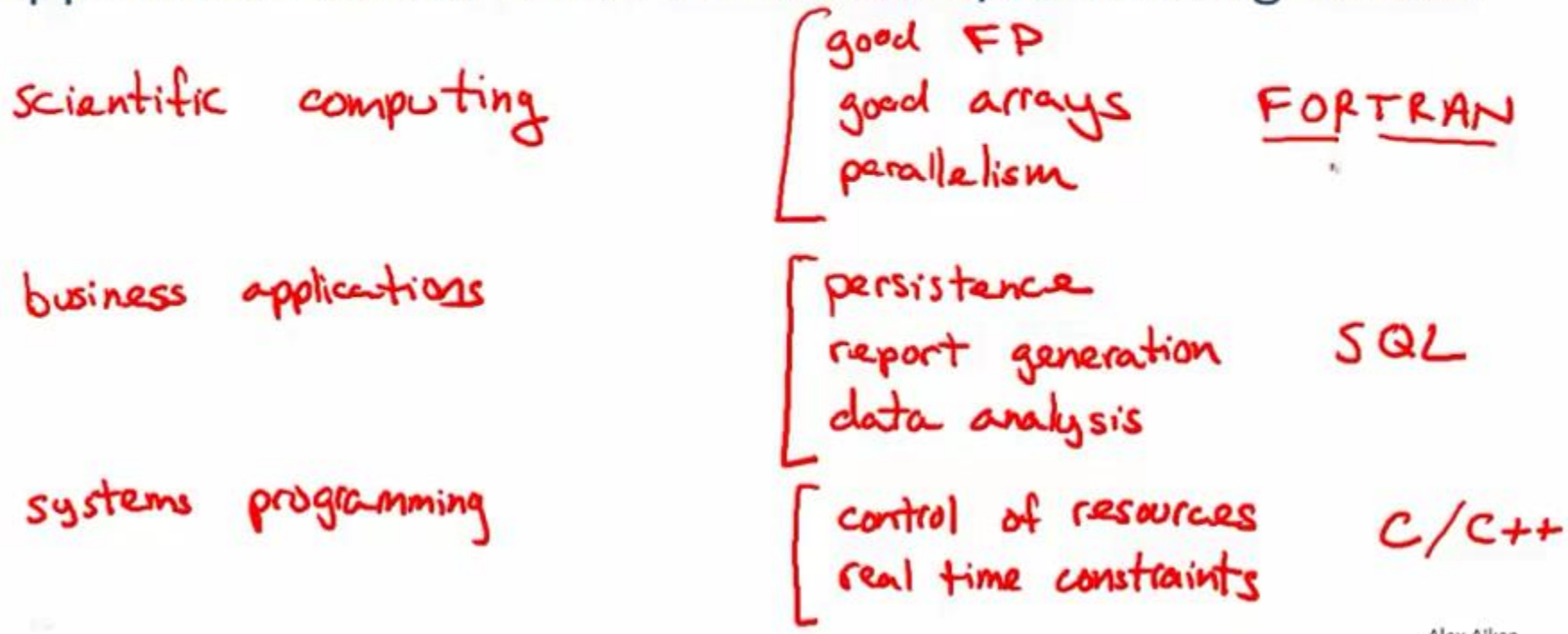
- ANTLR 4权威指南
- 作者: Terence Parr
- 出版社: 机械工业出版社
- 译者: 张博
- 出版年: 2017-5-1
- 页数: 262
- 定价: 69元
- 丛书: 华章程序员书库
- ISBN: 9787111566489


5th Generation Languages (Natural Language)
4th Generation Languages (SQL, LISP, etc.)
3rd Generation Languages (Java, C, Pascal, FORTRAN, COBOL, etc.)
2nd Generation Languages (Assembly Code)
1st Generation Languages (Binary Machine Code)
Number of Instructions
SQL – 4th level language
Ease of Human Understanding

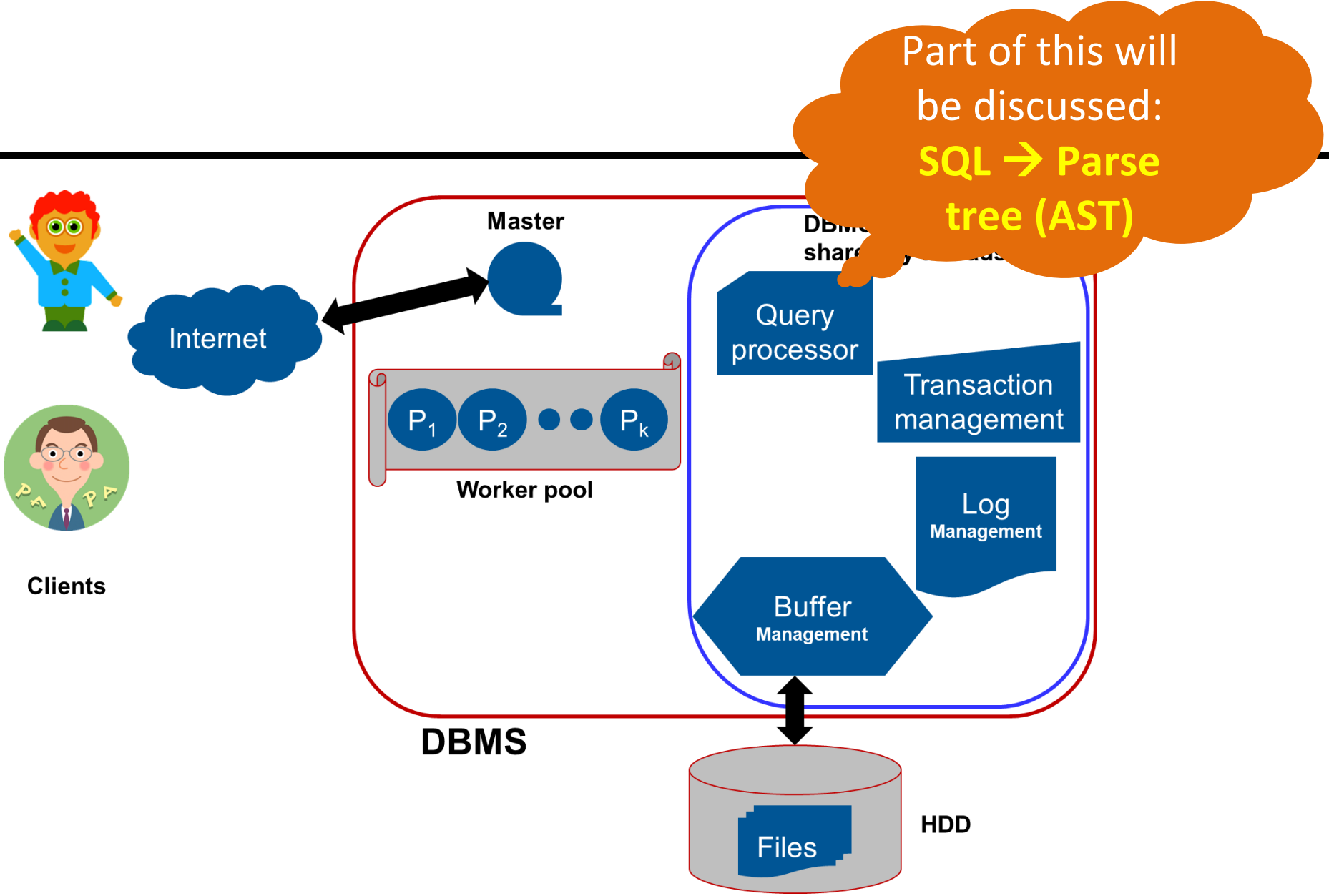
Economy

Why are there so many programming languages?

Application domains have distinctive/conflicting needs.

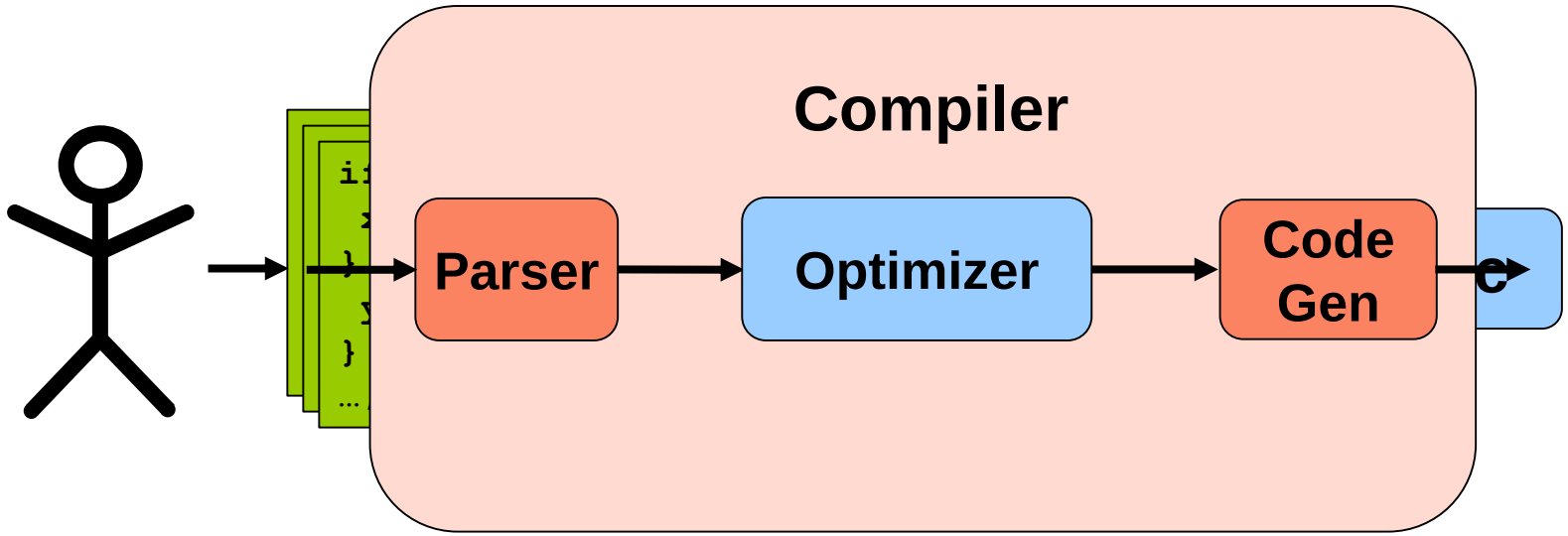


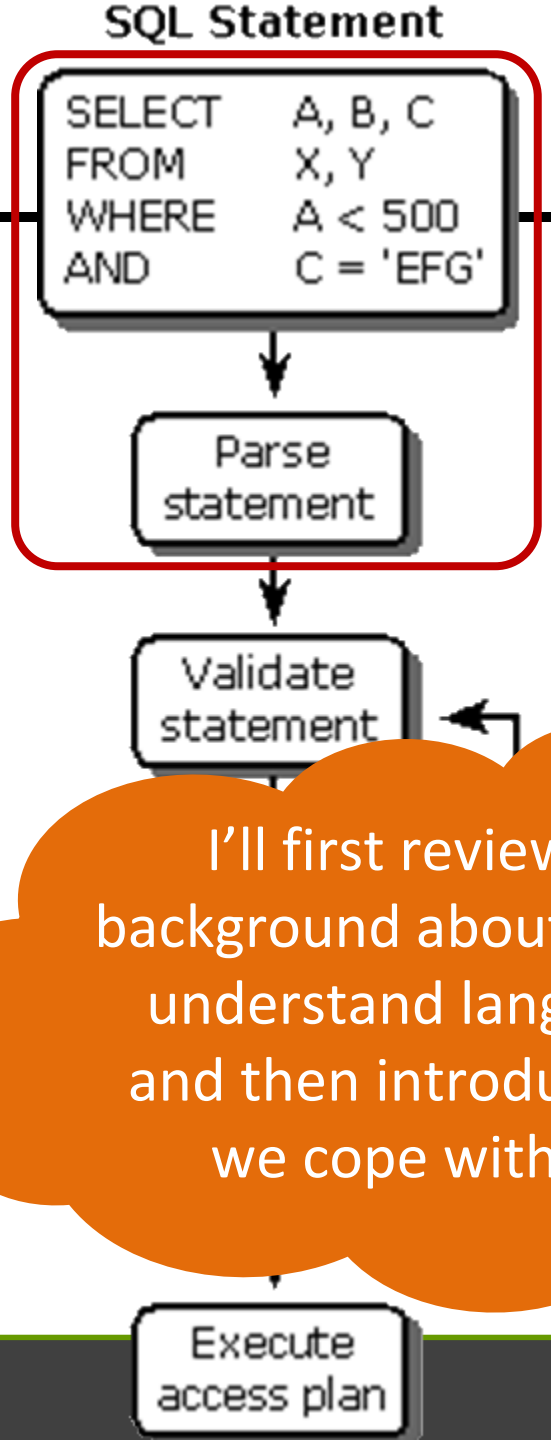
Alex Aiken


Part of this will be discussed:
SQL → Parse tree (AST)
Master
Internet
Clients
P_1 P_2 P_k
Worker pool
Query processor
Transaction management
Log Management
Buffer Management
DBMS
Files
HDD

Understanding SQL follows Compiler idea

- Let's look at a compiler





How does D

How to define a language?
– You can try …

- SQ
- By LANGUAGE –
  - **Syntax** ! – how to define the syntax of a language
    - If a sentence belongs to a language, it must satisfy the syntax of that language!
      - ✓ C, Java, SQL, …
    - utomata are used to **model syntax**, and could **rify** if a sentence s.t a given language?
    - ecursors proposed a **framework** to **derive perations** from a sentence?

I'll first review the background about how we understand languages, and then introduce how we cope with SQL

There is even a theory to define languages!

- **Do you really know "LANGUAGE"?** Or "**words**"

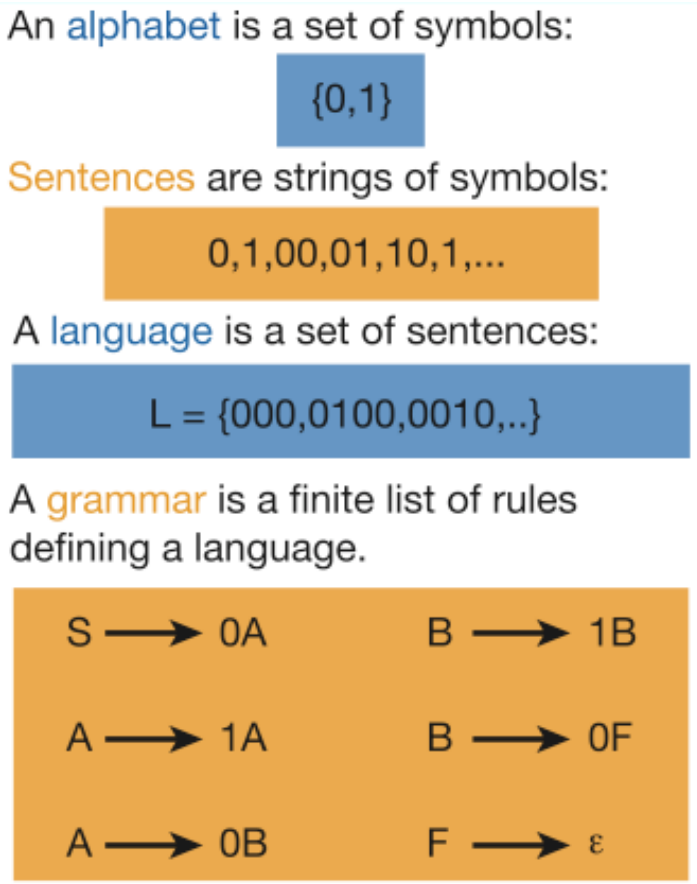


Image source: Nowak et al. Nature, vol 417, 2002

- **Languages**: "***A language is a collection of sentences of finite length*** *all constructed from a finite alphabet of symbols*"
- **Grammars**: "***A grammar can be regarded as a device that enumerates the sentences of a language***" - **nothing more, nothing less**
- ***N. Chomsky, Information and Control, Vol 2, 1959***

Formal grammar

Any idea??
– Your turn to browse the Internet if you want to know more – quite challenging!

- A formal grammar is a quad-tuple $G = (N, \Sigma, P, S)$ where
  - N is a finite set of non-terminals
  - Σ is a finite set of terminals and is disjoint from N
  - P is a finite set of production rules of the form
    $$w \in (N \cup \Sigma)^* \rightarrow w \in (N \cup \Sigma)^*$$
  - $S \in N$ is the start symbol

disjoint[disˈdʒɔint]
vi. 脱臼; 分离, 脱开
adj. 【数】不相交的

https://zh.wikipedia.org/zh-cn/%E5%B7%B4%E7%A7%91%E6%96%AF%E8%8C%83%E5%BC%8F
https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

BNF (Backus-Naur Form) [巴科斯范式]


John Backus Peter Naur

Popular way to describe GRAMMAR

- A BNF specification is a set of **derivation** rules, written as

  derivation K.K.[ˌdɛrəˈveʃən] n. 发展; 起源; 派生

  - < symbol > ::= __expression__
- Where <symbol>[5] is a *nonterminal*, and the __expression__ consists of one or more sequences of symbols;
  - more sequences are separated by the vertical bar "|", indicating a choice, the whole being a possible substitution for the symbol on the left.
  - Symbols that never appear on a left side are *terminals*.
  - On the other hand, symbols that appear on a left side are *non-terminals* and are always enclosed between the pair <>.[5]
  - The "::=" means that the symbol on the left must be replaced with the expression on the right.

https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

- As an example, consider this possible BNF for a U.S. postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>

     <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
                   | <personal-part> <name-part>

 <personal-part> ::= <initial> "." | <first-name>

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

      <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
   <opt-apt-num> ::= <apt-num> | ""
```

<name-part>	LingBo Kong <EOL – End of Line>
<street-address>	811……. <EOL>
<zip-part>	BeiJing, [no state code] 100044 <EOL>

History of Backus-Naur Form (BNF)

- In 1959 John Backus of IBM devised **Backus Normal Form** as a concise notation for describing components of a programming language (Algol 58, the first high-level programming language)
- In 1960 Peter Naur, a Danish programmer, refined Backus's notation
- Result is **Backus-Naur Form** (the preferred name for the notation in current use, though this is still usually called *Backus Normal Form*)

Use of BNF

- **Backus-Naur Form makes it possible to define SQL (and other programming languages) concisely**

Many languages, and therefore many grammars
Regular Expression [正则表达式]

- **Example of a Regular grammar**

$$
\begin{aligned}
S &\rightarrow aS \\
S &\rightarrow bA \\
A &\rightarrow \varepsilon \\
A &\rightarrow cA
\end{aligned}
$$

  - **All non-terminals locate at one side of terminals**

  - Is “aaab” one sentence defined by the above Regular grammar?
  - “aabbccc”?
  - How about “aaaaabccccccccccccccccccc”?

https://www.os3.nl/_media/2012-2013/courses/es/re.pdf

Many languages, and therefore many grammars
CFG – Context Free grammar [上下文无关语法]

- **Example of a Context-Free grammar**

$$S \rightarrow aSb$$
$$S \rightarrow ab$$

- **Non-terminals could occur between terminals**

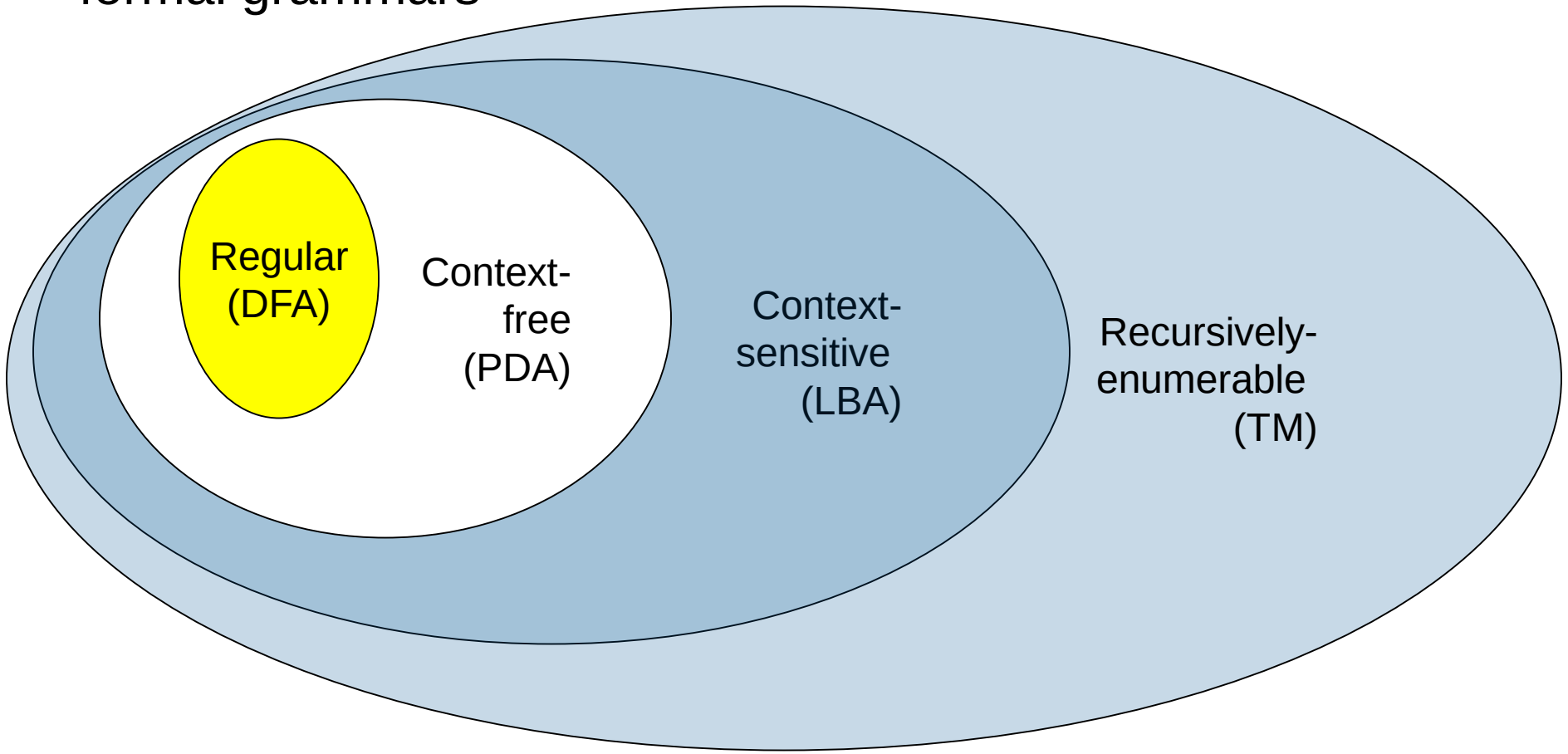
- The above grammar defines a language of $\mathbf{a^k b^k}$, k≥1
  - Regular expression grammar could not express that language

- **Regular Expressions are less expressive than context-free grammars (CFGs)**

https://www.os3.nl/_media/2012-2013/courses/es/re.pdf

We have Chomsky Hierarchy

- A containment hierarchy (strictly nested sets) of classes of formal grammars



11.2.3 乔姆斯基层次

表11-1总结了乔姆斯基层次和它的特性。

表11-1 文法、产生式规则及其等价关系一览表

	正则文法	上下文无关文法	上下文相关文法	递归可枚举文法
产生式规则	$u \rightarrow Xv$ $u \rightarrow X$	$u \rightarrow vw$ $u \rightarrow X$	$\alpha X \gamma \rightarrow \alpha \beta \gamma$	所有
闭包性质	$\cup, ., ^*$ $\cap, \bar{}$	$\cup, ., ^*$ 无$\cap$，无$\bar{}$	$\cup, ., ^*$	所有
等价的自动机	有限状态自动机	下推自动机	线性有界的图灵机	图灵机
特征语言		回文	复制语言	所有
相关性特征	无长程相关性	嵌套	交叉	所有

DFA **PDA** **LBA** **TM** Turing Machine

Deterministic Finite Automaton

确定有限状态自动机或确定有限自动机

Linear bounded automaton：线性有界自动机

Pushdown automaton：下推自动机

Programming languages

CFG is quite powerful!

- The syntax of most programming languages is context-free (or very close to it)
  - EBNF / ALGOL 60
- Due to memory constraints, long-range relations are limited
- Common strategy: a relaxed CF parser that accepts a superset of the language, invalid constructs are filtered
- Alternate grammars proposed: indexed, recording, affix, attribute, van Wijngaarden (VW)

Chapter 3: Parse SQL

- **SQL as a language – 4 GL (Generation Language) – follows some rule**
  - **Syntax** should be followed – How is syntax defined?
  - **Automata** are used to **model syntax**, and could **verify** if a sentence s.t a given language?
  - Precursors proposed a **framework** to derive operations from the legal sentence
- **3rd training project – a Naïve SQL parser**

  Select B,D

  From R,S

  Where R.A = “c” ∧ R.C=S.C

Automata?! -The “machine” to understand the language you have defined

- The hierarchy and The Automata [自动机]

Class	Grammars	Lan	
Type-0	Unrestricted	Re (T	
Type-1	Context-sensitive	Context-sensitive	L ar-bounded
Type-2	Context-free	Context-free	Pushdown
Type-3	Regular	Regular	Finite

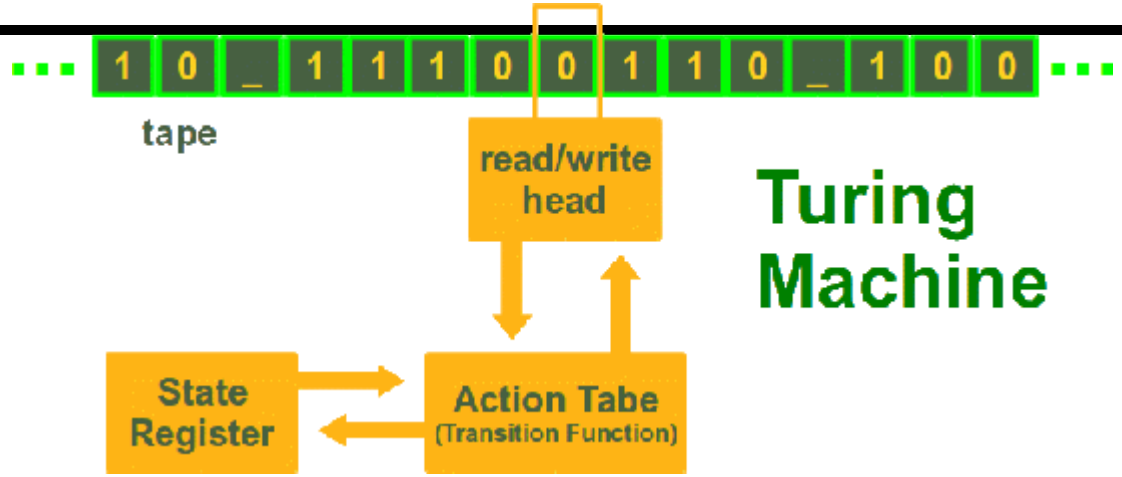
“Pushdown” here means you have to use “stack data structure” to support retrospect [回溯]

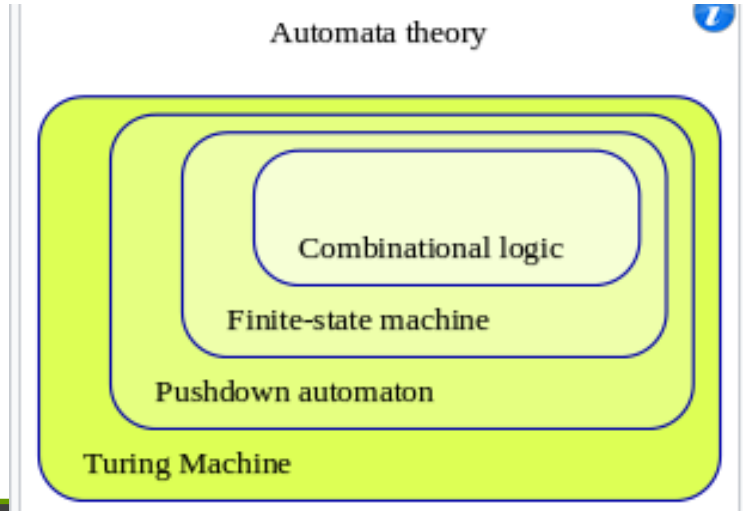
Automata?
– Turing machine is the most powerful one

https://en.wikipedia.org/wiki/Turing_machine

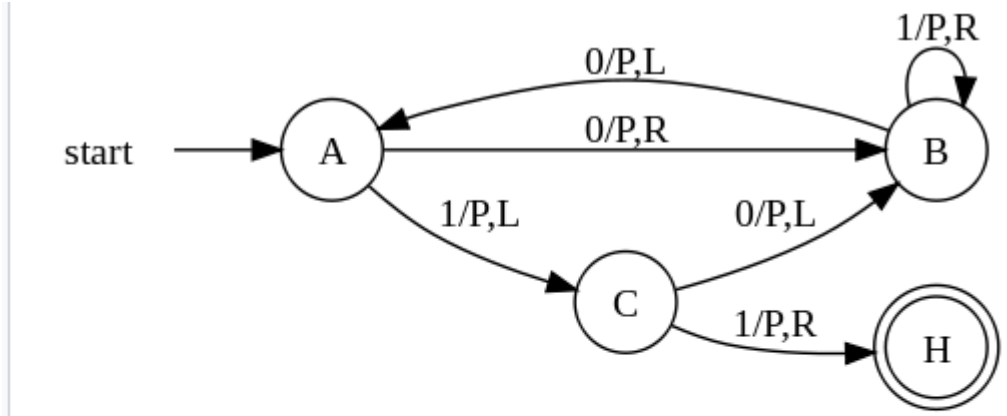
http://www.python-course.eu/turing_machine.php

- A **Turing machine** is an abstract machine[1] that manipulates symbols on a strip of tape according to a table of rules; to be more exact, it is a mathematical model of computation that defines such a device



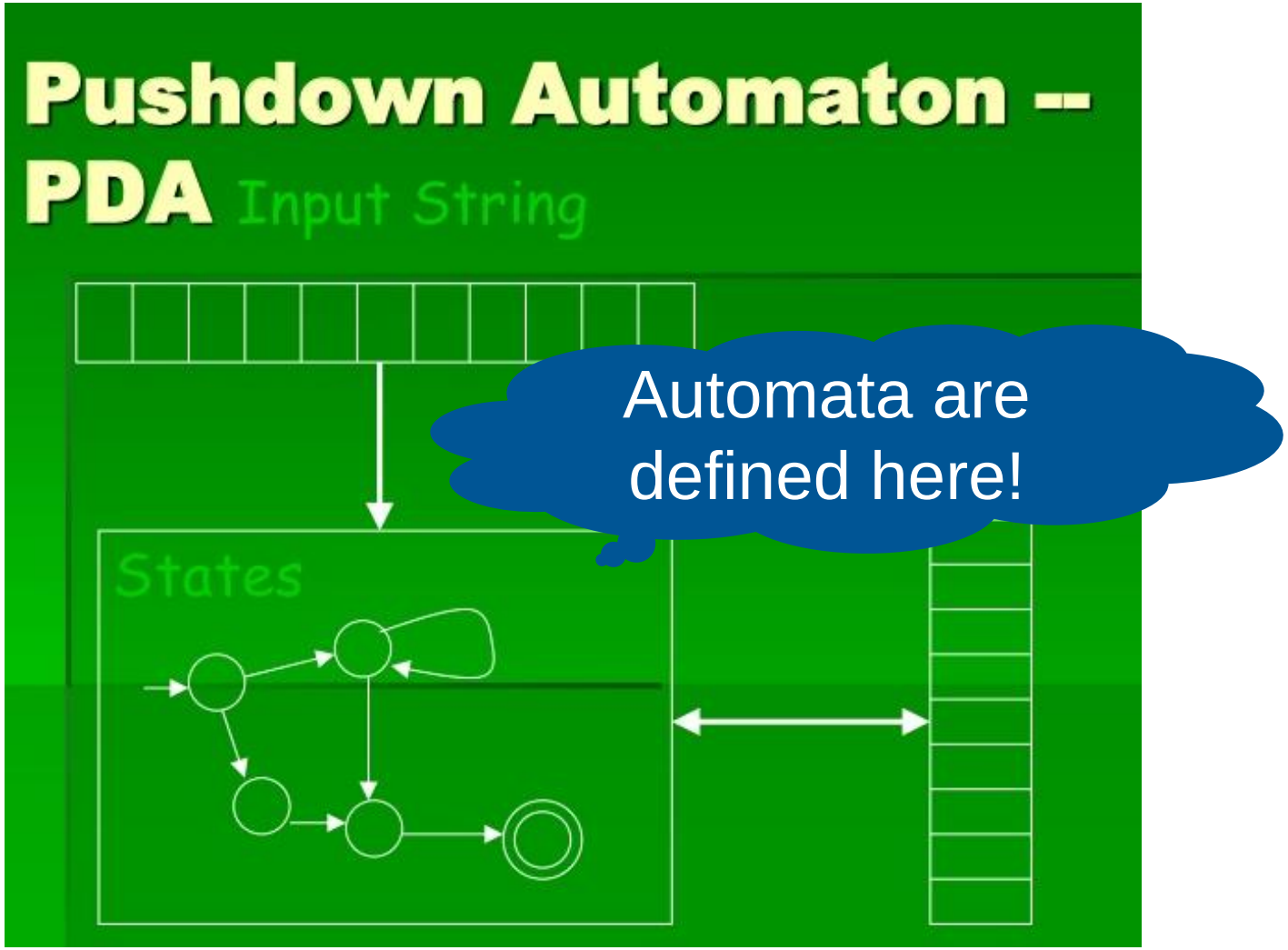


Classes of automata (Clicking on the text will take you to an article on that subject)



The "3-state busy beaver" Turing machine in a finite state representation. Each circle represents a "state" of the TABLE—an "m-configuration" or "instruction". "Direction" of a state *transition* is shown by an arrow. The label (e.g. **0/P,R**) near the outgoing state (at the "tail" of the arrow) specifies the scanned symbol that causes a particular transition (e.g. **0**) followed by a slash **/**, followed by the subsequent "behaviors" of the machine, e.g. "**P** Print" then move tape "**R** Right". No general accepted format exists. The convention shown is after McClusky (1965), Booth (1967), Hill, and Peterson (1974).

State diagram



A finite automaton is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$

Q is the set of states

Σ is the alphabet

$\delta : Q \times \Sigma \rightarrow Q$ is the transition function

$q_0 \in Q$ is the start state

$F \subseteq Q$ is the set of accept states

L(M) = the language of machine M
= set of all strings machine M accepts

$M = (Q, \Sigma, \delta, q_0, F)$ where

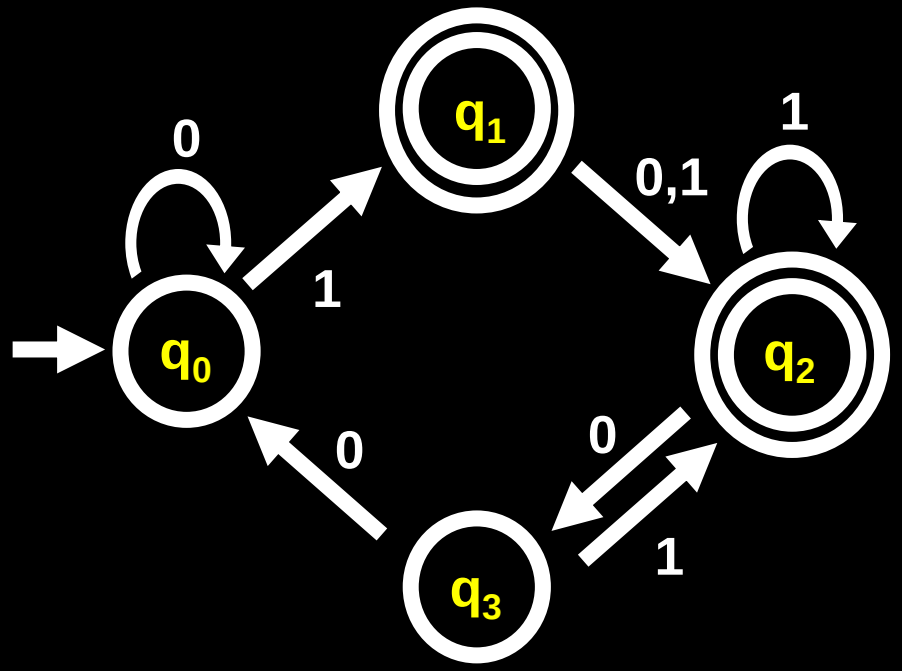
$Q = \{q_0, q_1, q_2, q_3\}$

$\Sigma = \{0,1\}$

$\delta : Q \times \Sigma \rightarrow Q$ transition function*

$q_0 \in Q$ is start state

$F = \{q_1, q_2\} \subseteq Q$ accept states



*

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_3	q_2
q_3	q_0	q_2

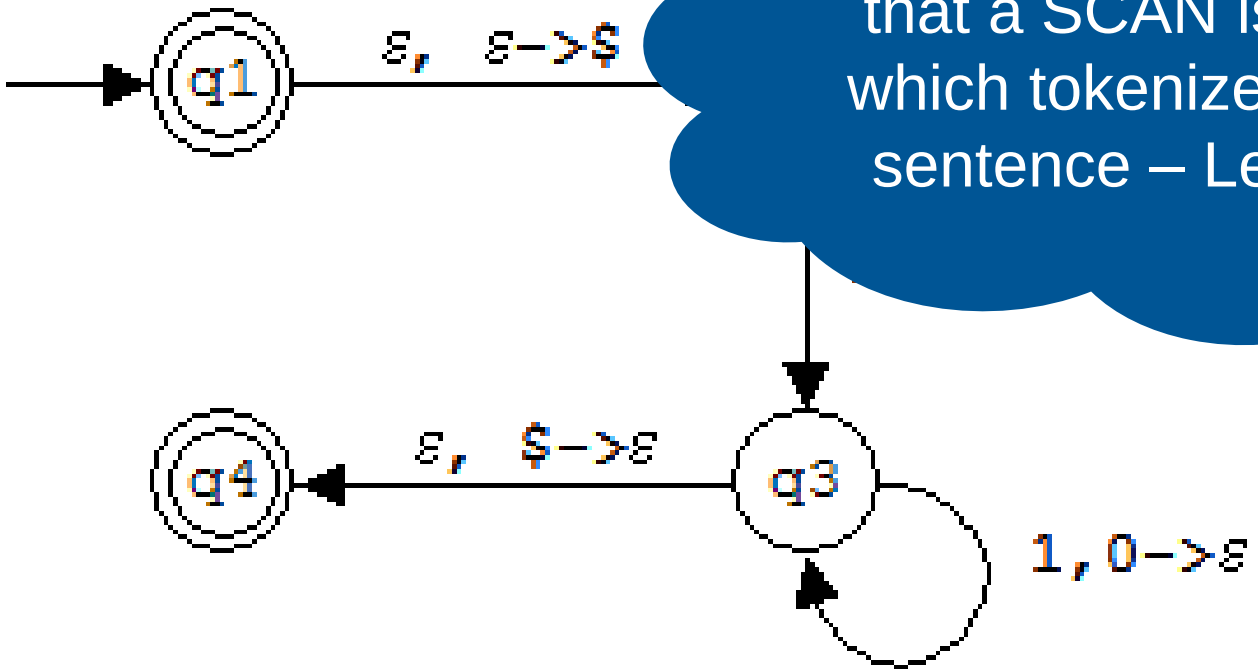
An example

http://csclab.murraystate.e

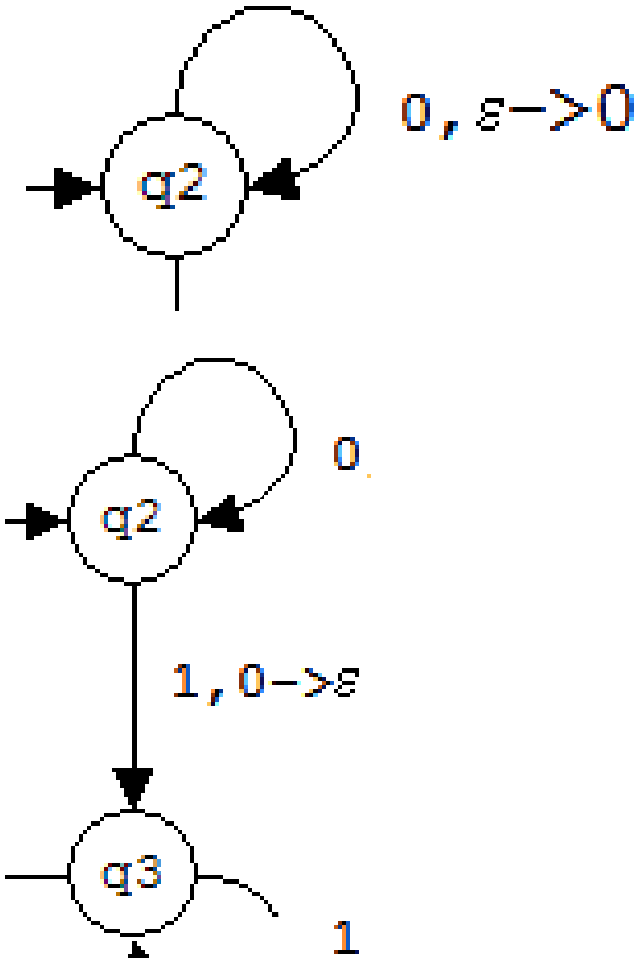
So you can finish a program to verify if “0001111” is legal or not?

- **How to understand the language {0**
  - 000111?
    - Read the input from **L**EFT to **R**IGHT with help of

At least you could derive that a SCAN is needed which tokenize the input sentence – Lexer later



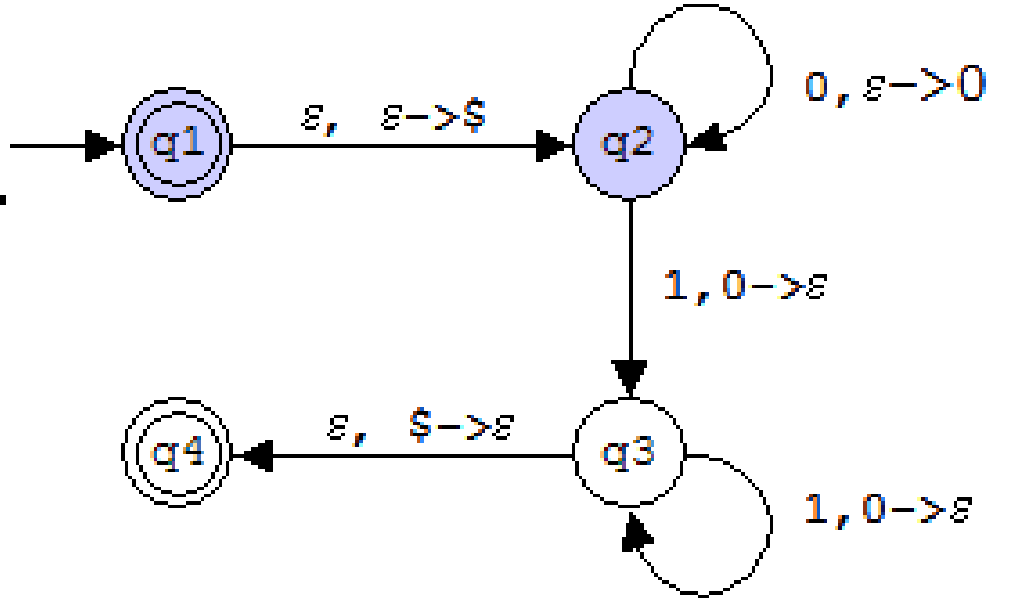
state, input/read and action (→)

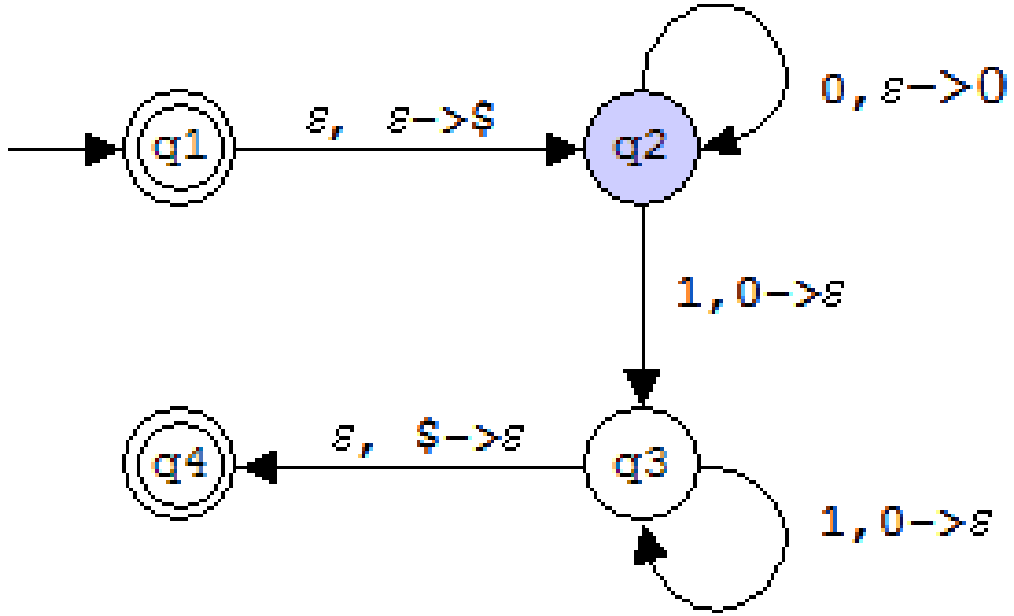


- At state q_2, if read 0, "ε→0" means you p[illegible] stac[illegible]

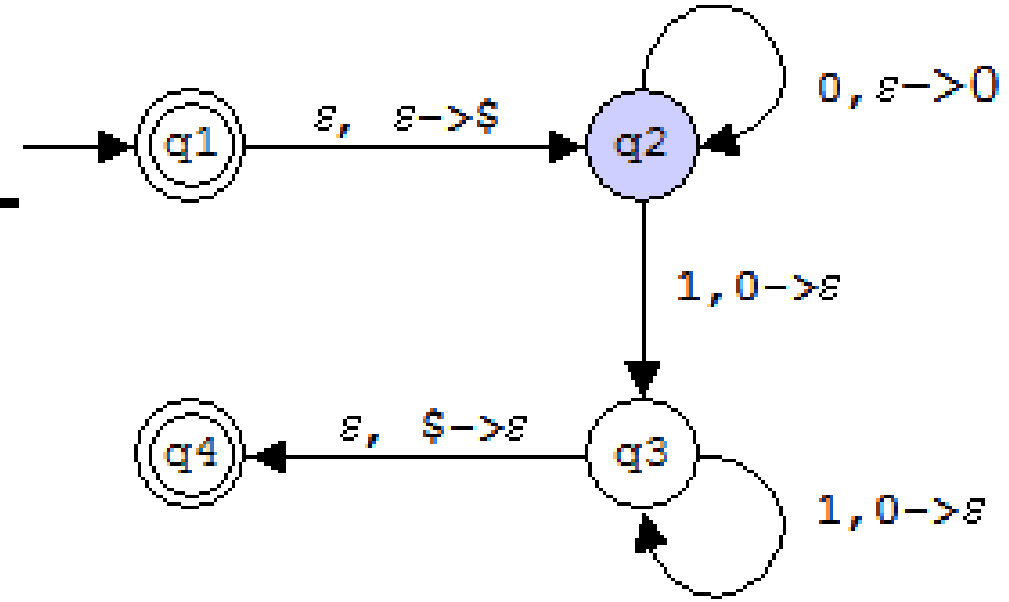
Verification of a given sentence is carried out based on STACK

- At state q_2, if read 1, "0→ ε" means to pop the top 0 from stack; and the state is moved to q_3

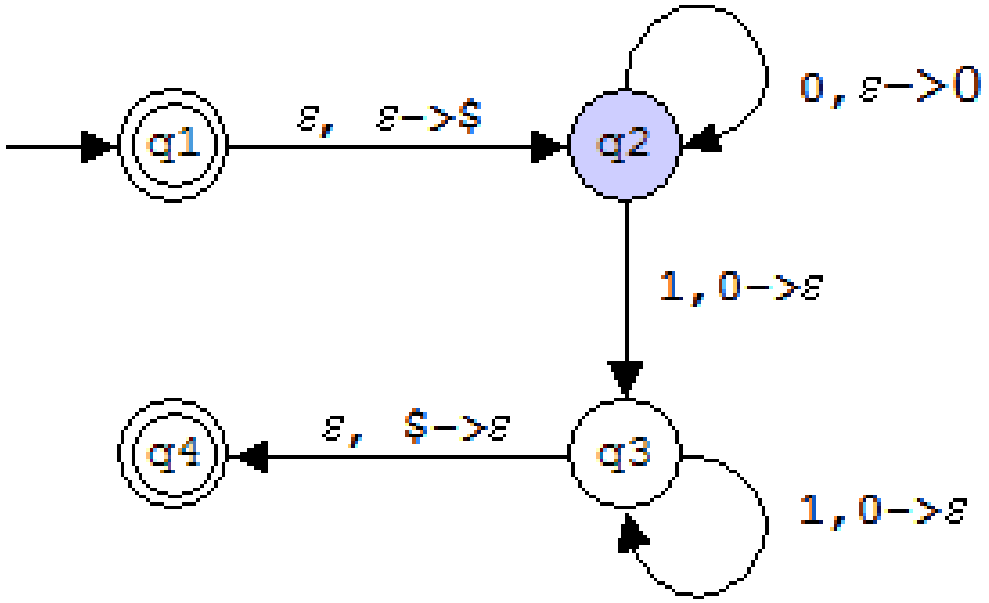




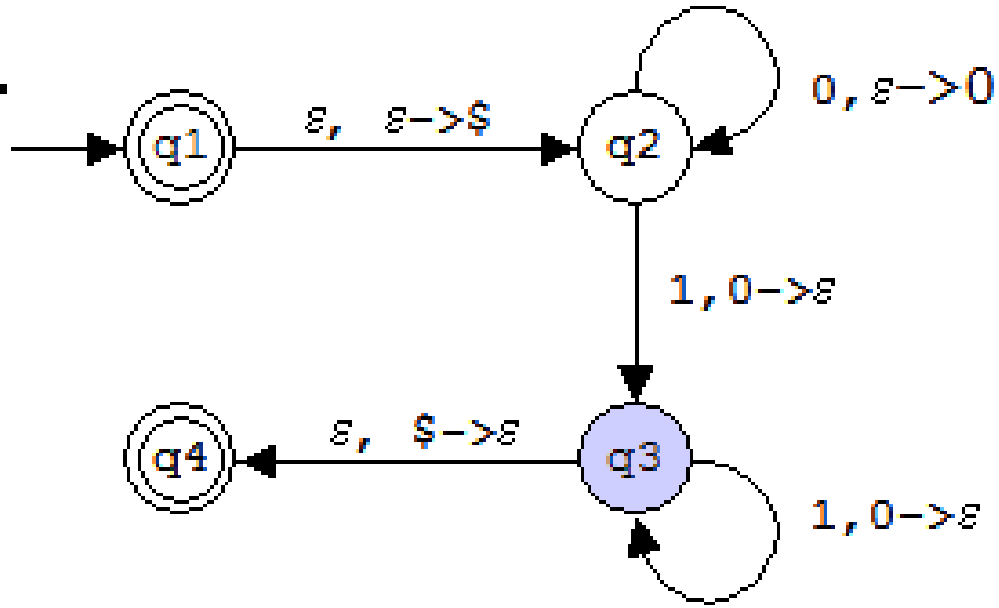
Input	Stack
0001111	$
^	
	0
0001111	$
^	



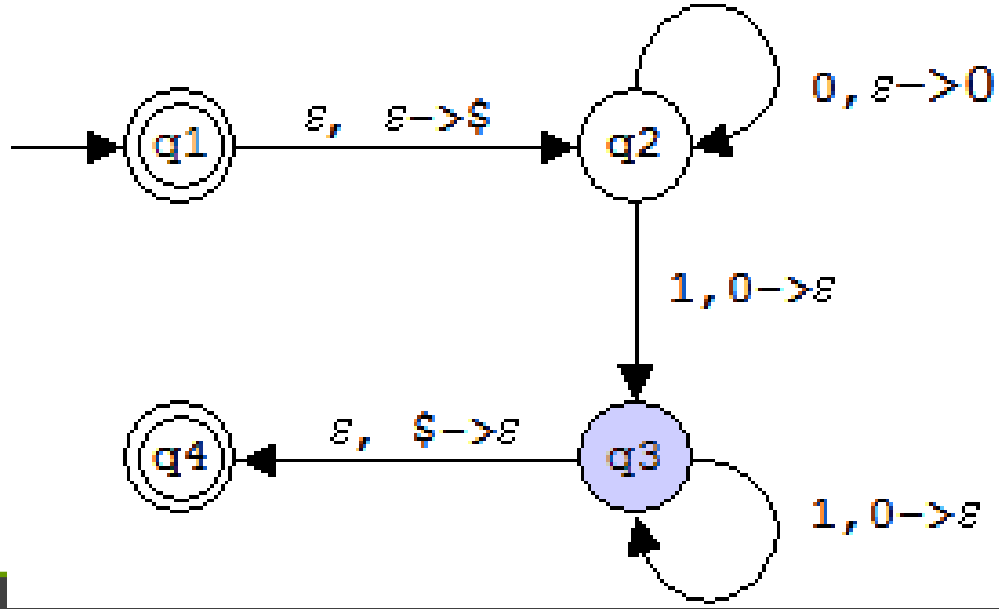
0
0
0001111 $
^



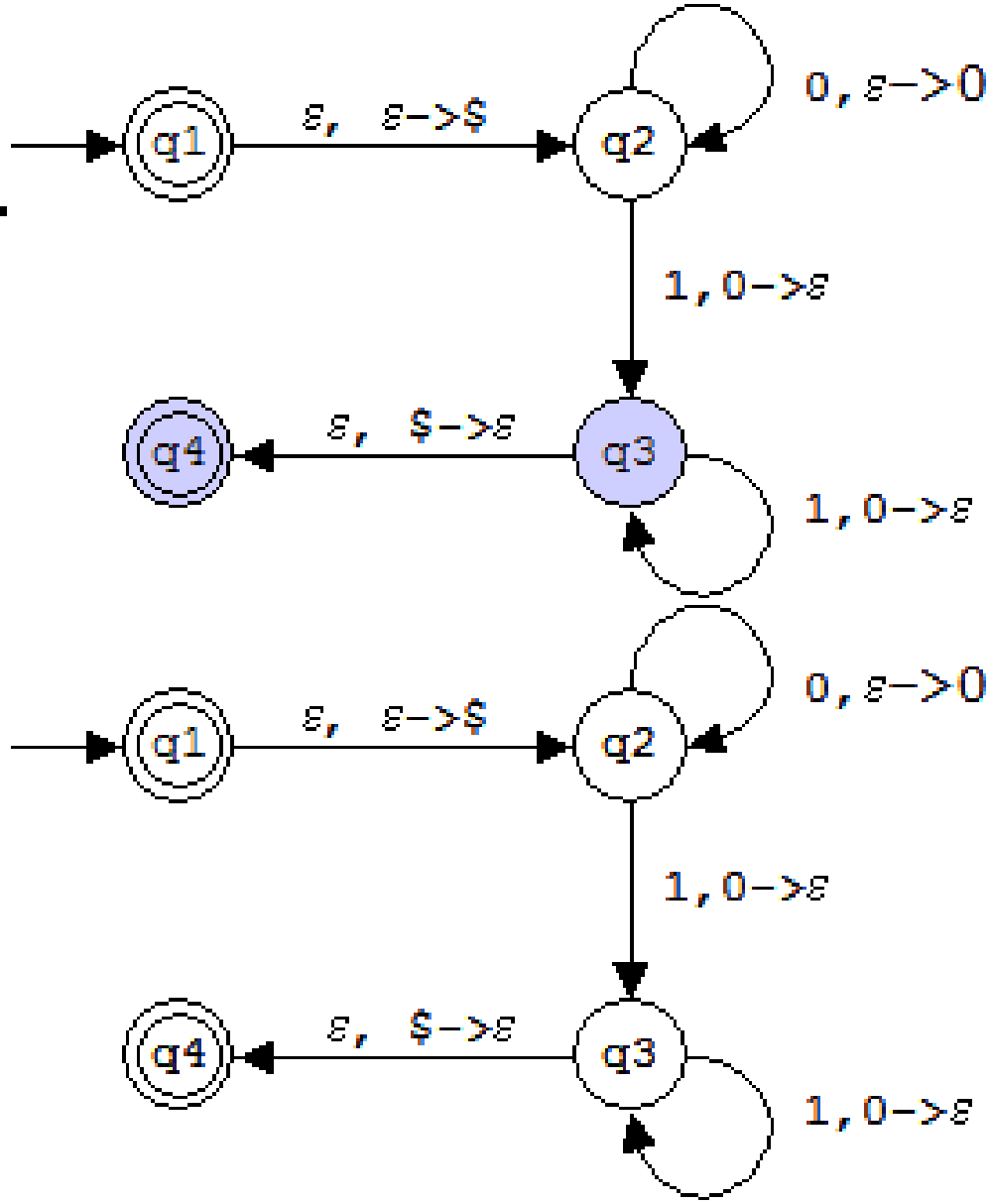
0
0
0
0001111 $
^



0
0
0001111 $
^



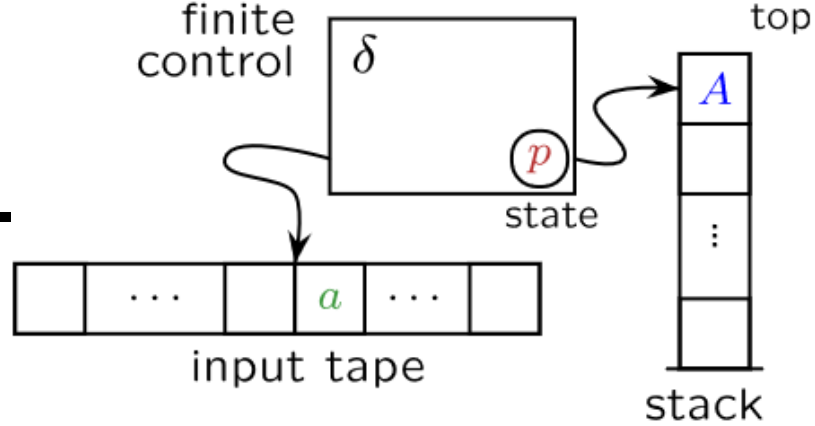
0
0001111 $
^



0001111 ___
^

0001111 ___
^

Reading the final '1' symbol in the input string we find that there is no valid transition from either q3 or q4 on a '1' input symbol AND an empty stack. In this case we are left in no state, so we do not accept.



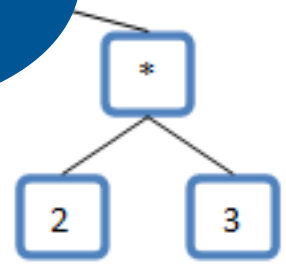
To parse 0001111?

- By “parse” here, 2 phases
  - **Lexer** [词法分析器] (**Tokenizer**) – check if the “character” from the tape s.t the conditions defined in the grammar
    - ✓ $\{0^n1^n \mid n >= 0\}$ can only accept “1” and “0”
    - ✓ For later more complex languages/grammars, the check may be based on “word” – like in English, word is the basis of a sentence
  - **Parser** [语法分析器] – check if the sequence s.t semantic grammar
    - ✓ Grammar could be understood as a collection of rules
      - » $S \rightarrow \epsilon \mid 0S1$
    - ✓ The PDA is built based on this, and could be used to verify a given sequence

- **NB**: only simple mechanism o
- More complex compiler need o
  verification/evaluation
  - **AST** (Abstract Syntax Tree) is popular in Com
  - **Parse Tree** is the corresponding
- **For example**
  - A math expression: 1+2*3
  - Its AST is sketched as
    - How to construct AST/(Pars
      to the grammar is the core of Com
  - Based on this AST, we could carry out the needed computation/execution

Of course you could use IF-ELSE to verify the input sentence legal or not.

But IF-ELSE style is not OK for framework of automatic compiler construction general for different grammars – discussed more later



```
EXP     -> TERM EXP1
EXP1    -> + TERM EXP1 |
           - TERM EXP1 |
           epsilon
TERM    -> FACTOR TERM1
TERM1   -> * FACTOR TERM1 |
           / FACTOR TERM1 |
           epsilon
FACTOR -> ( EXP ) | - EXP | number
```

We could follow the grammar here to evaluate if 1+2*3 is legal or not

Exp→ Term Exp1
→ Factor (1) Term1 (eps) Exp1
→ 1 + Term Exp1
→ 1 + Factor Term1 Exp1
→ ….

We'll build this tree by inserting semantic actions and adding nodes according to the following rules:

Production	Semantic rule
EXP -> TERM EXP1	EXP.node = mknode(Plus, TERM.node, EXP1.node)
EXP1 -> + TERM EXP1	EXP1.node = mknode(Plus, EXP1.node, TERM.node)
EXP1 -> - TERM EXP1	EXP1.node = mknode(Minus, EXP1.node, TERM.node)
EXP1 -> epsilon	EXP1.node = mknode(Number, 0)
TERM -> FACTOR TERM1	TERM.node = mknode(Mul, FACTOR.node, TERM1.node)
TERM1 -> * FACTOR TERM1	TERM1.node = mknode(Mul, TERM1.node, FACTOR.node)
TERM1 -> / FACTOR TERM1	TERM1.node = mknode(Div, TERM1.node, FACTOR.node)
TERM1 -> epsilon	TERM1.node = mknode(Number, 1)
FACTOR -> (EXP)	FACTOR.node = EXP.node
FACTOR -> - EXP	FACTOR.node = mknode(UnaryMinus, EXP.node)
FACTOR -> number	FACTOR.node = mknode(Number, number)

Data structures you've learned are used here

```
enum ASTNodeType
{
    Undefined,
    OperatorPlus,
    OperatorMinus,
    OperatorMul,
    OperatorDiv,
    UnaryMinus,
    NumberValue
};
```

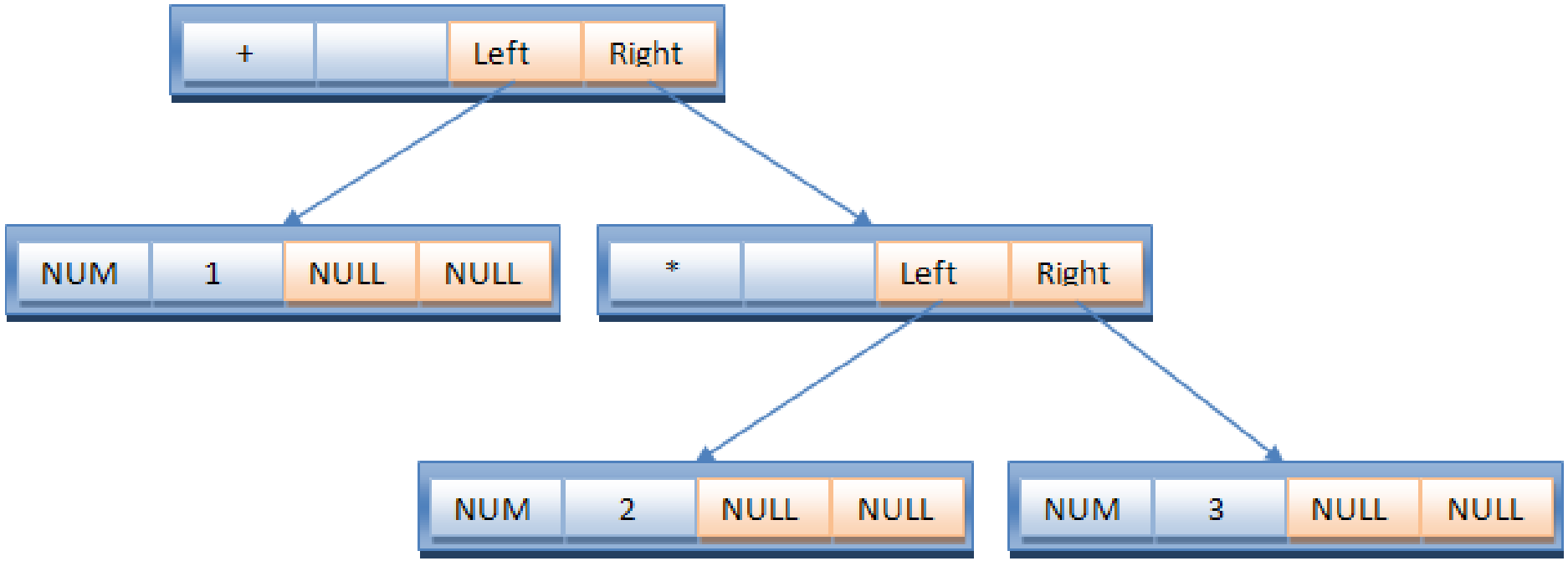
```
class ASTNode
{
public:
    ASTNodeType Type;
    double      Value;
    ASTNode*    Left;
    ASTNode*    Right;

    ASTNode()
    {
        Type = Undefined;
        Value = 0;
        Left = NULL;
        Right = NULL;
    }

    ~ASTNode()
    {
        delete Left;
        delete Right;
    }
};
```

```
class Parser
{
   Token m_crtToken;
   const char* m_Text;
   size_t m_Index;

private:

   ASTNode* Expression()
   {
      ASTNode* tnode = Term();
      ASTNode* e1node = Expression1();

      return CreateNode(OperatorPlus, tnode, e1node);
   }

   ASTNode* Expression1()
   {
      ASTNode* tnode;
      ASTNode* e1node;

      switch(m_crtToken.Type)
      {
      case Plus:
         GetNextToken();
         tnode = Term();
         e1node = Expression1();

         return CreateNode(OperatorPlus, e1node, tnode);

      case Minus:
         GetNextToken();
         tnode = Term();
         e1node = Expression1();
```

```
double GetNumber()
{
    SkipWhitespaces();

    int index = m_Index;
    while(isdigit(m_Text[m_Index])) m_Index++;
    if(m_Text[m_Index] == '.') m_Index++;
    while(isdigit(m_Text[m_Index])) m_Index++;

    if(m_Index - index == 0)
        throw ParserException("Number expected but not found!", m_Index);

    char buffer[32] = {0};
    memcpy(buffer, &m_Text[index], m_Index - index);

    return atof(buffer);
}
```

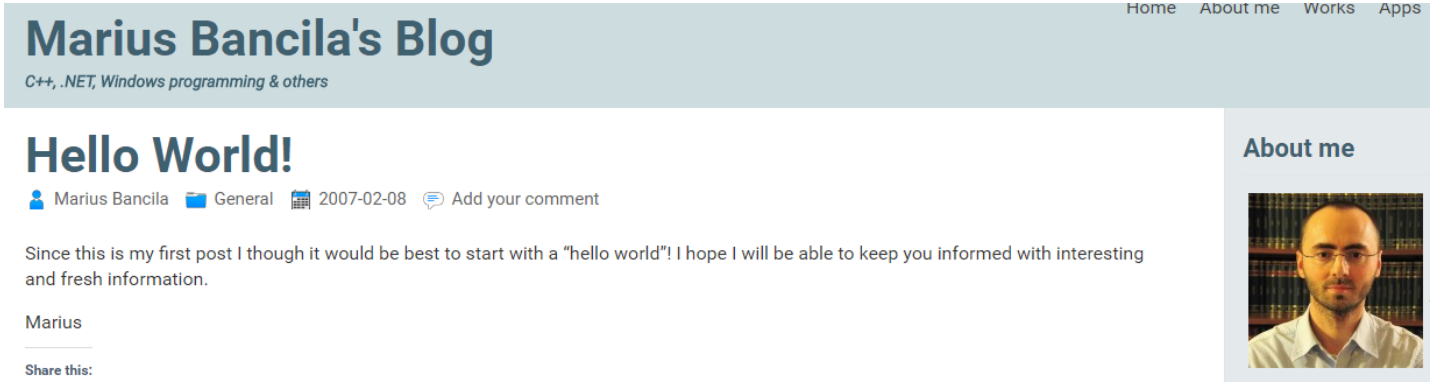
```
public:
    ASTNode* Parse(const char* text)
    {
        m_Text = text;
        m_Index = 0;
        GetNextToken();

        return Expression();
    }
};
```

For the expression 1+2*3, the AST will be:



- **Redundant, but OK for understanding**
  - How to convert language sentence into tree data structure, which is helpful to understand SQL parsing/executing
- **Post-order Traversal (LRD) to compute the result**

- **Thanks to the volunteers – a GREAT example for us to understand Compiler Theory**



- **Evaluating Expressions**
  1. The Approaches
  2. Parse the expression
  3. Build the abstract syntax tree
  4. Evaluate the abstract syntax t

Translation – from sentence (s.t some Grammar) to AST (Data structure)

- **cos, sin etc.?**
  - cos(num)
  - sin(num)
  - The num is used as radian [弧度]
- **How to implement Lexer now?**
  - Only after the Lexer can we parse the math expression
- **How to define Syntax and use later tools to finish the above Math functions?**

Chapter 3: Parse SQL

- **SQL as a language – 4 GL (Generation Language) – follows some rule**
  - **Syntax** should be followed – How is syntax defined?
  - **Automata** are used to **model syntax**, and could **verify** if a sentence *s.t* a given language?
  - Precursors proposed a **framework** to derive operations from the legal sentence
- **3rd training project – a Naïve SQL parser**

  Select B,D

  From R,S

  Where R.A = “c” ∧ R.C=S.C

- **The Languages used in CS has intimate relationship with Computation Theory – Automata Theory [自动机理论].**
  - How to design the program to understand the languages! – Compiler Theory

- **Language implementation systems must analyze source code, *regardless of the specific implementation approach***
  - Nearly all **syntax analysis** is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a **lexical analyzer** [词法分析器] (mathematically, a finite automaton based on a regular grammar)

- A high-level part called a **syntax analyzer** [语法分析器], or **parser** (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Fortunately, there are many ready-to-use Lexical/Syntax analyzers !

LR: Left to Right

precedence [priˈsi:dəns]
n.领先于…的权利; 优先权

Knuth, D. E. (July 1965). "On the translation of languages from left to right" (PDF). *Information and Control*. **8** (6): 607–639.

- In **1965** Donald Knuth invented the LR(k) parser (Left to right, Rightmost derivation parser) a type of shift-reduce parser, as a generalization of existing precedence parsers.
- This parser has the potential of recognizing all deterministic context-free languages (PDA) and can produce both left and right derivations of statements encountered in the input file.

Professor Knuth accomplished many amazing works

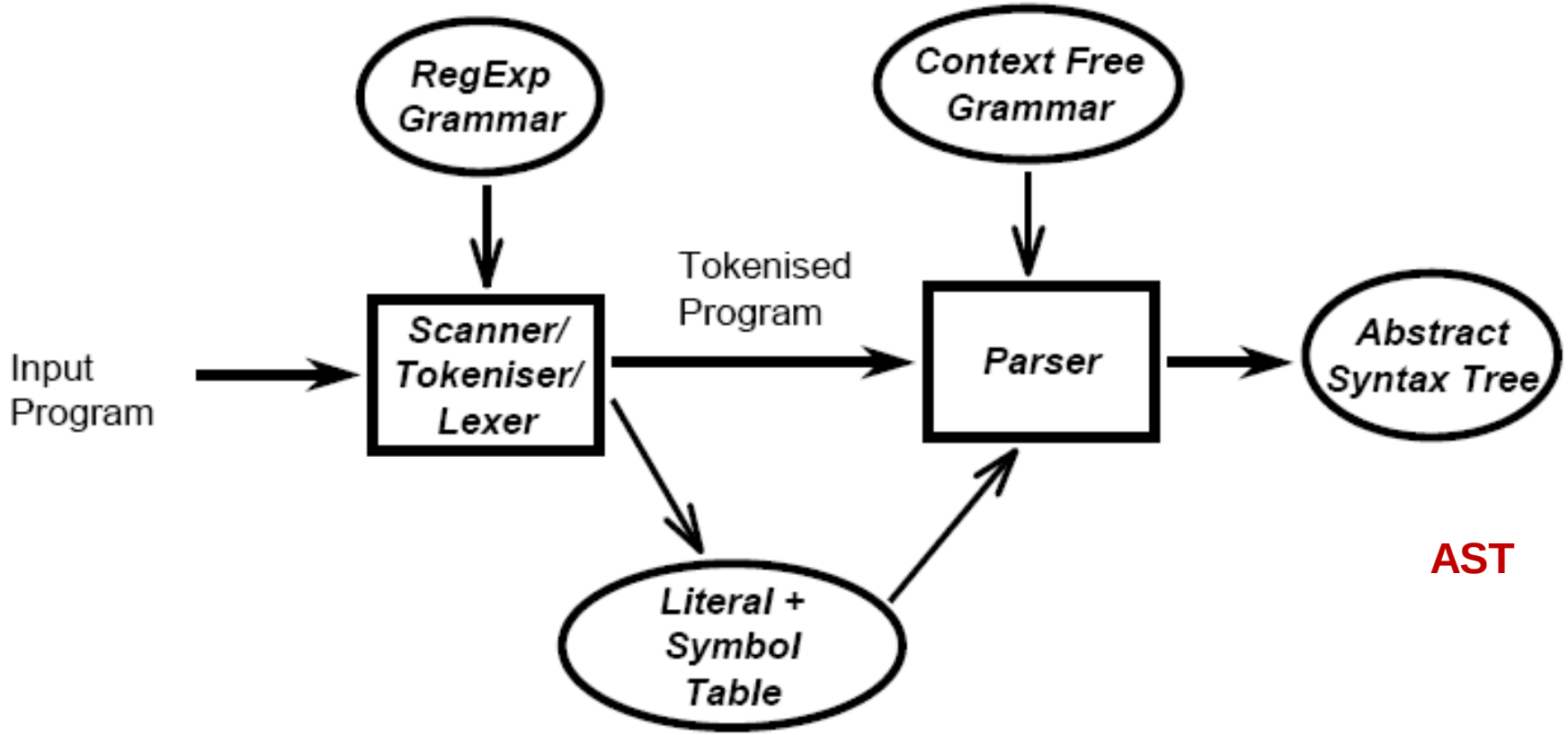
- TAOCP

- TeX
  - Scientific Typesetting

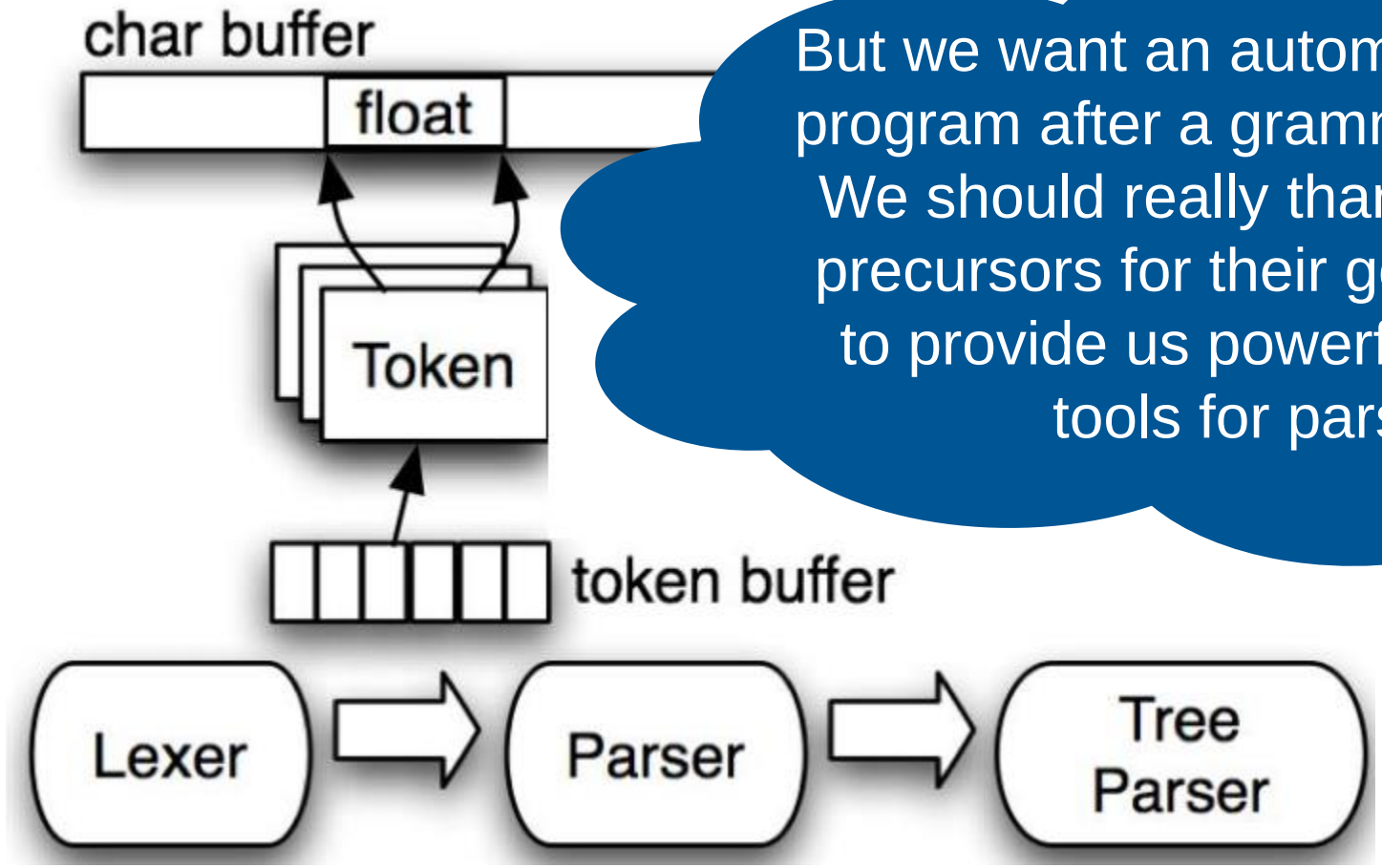
- Even Knuth's Buddy System – old method to manage Main Memory for OS

Parsers usually follow similar FRAMEWORK

- As the procedure of translating a language – Compiler



http://www.pling.org.uk/cs/lsa.html



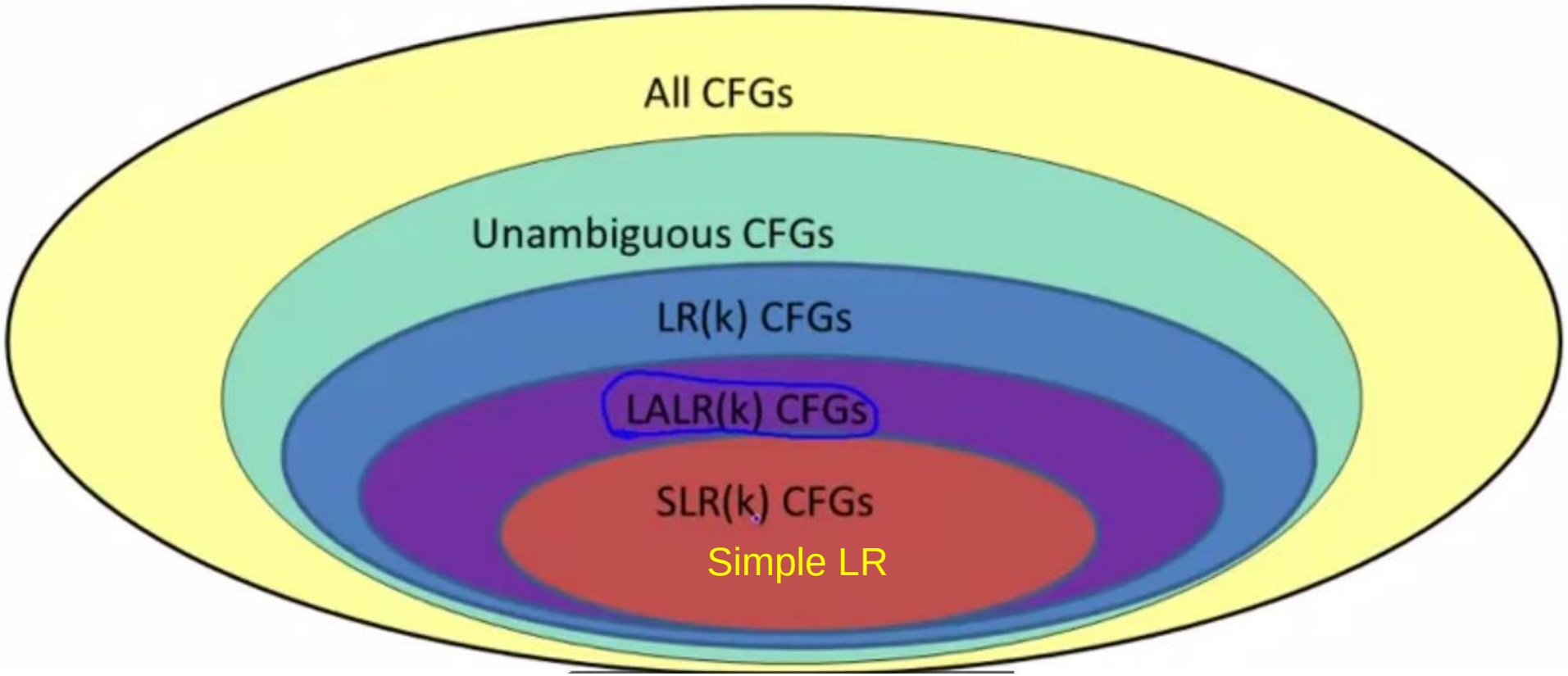
But we want an automatism to get a program after a grammar is given! – We should really thank a lot to our precursors for their gorgeous work to provide us powerful automatic tools for parsers!

http://creation.com/frank-deremer

LALR: Look-Ahead LR parser

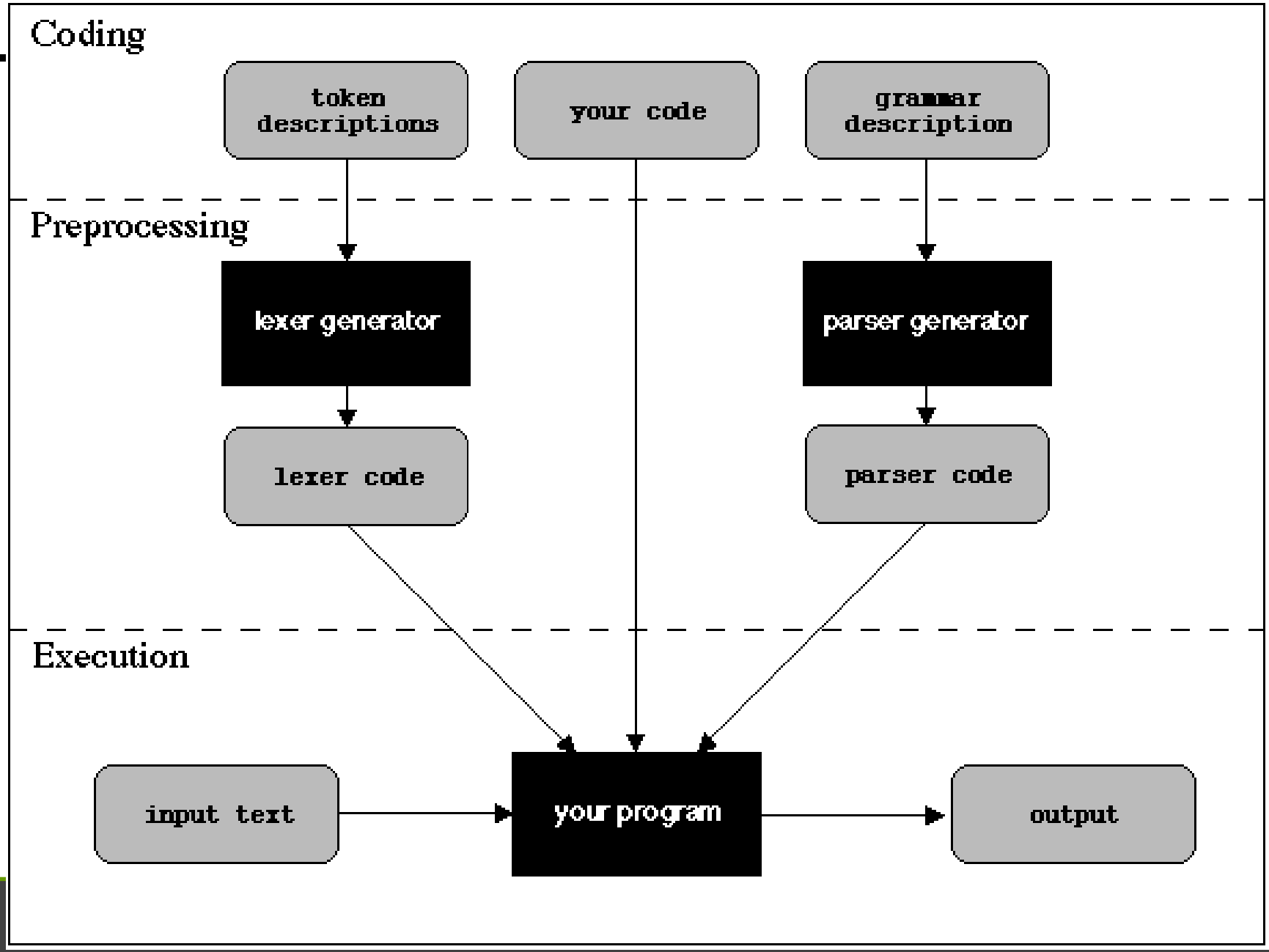
- In computer science, an LALR parser[a] or Look-Ahead LR parser is a simplified version of a canonical LR parser [规范LR], to parse (separate and analyze) a text according to a set of production rules specified by a formal grammar for a computer language.

- The LALR parser was invented by Frank DeRemer in his 1969 PhD dissertation, *Practical Translators for LR(k) languages*,[1] in his treatment of the practical difficulties at that time of implementing LR(1) parsers.



https://www.zhihu.com/question/29636774

With those tools, the compiler …



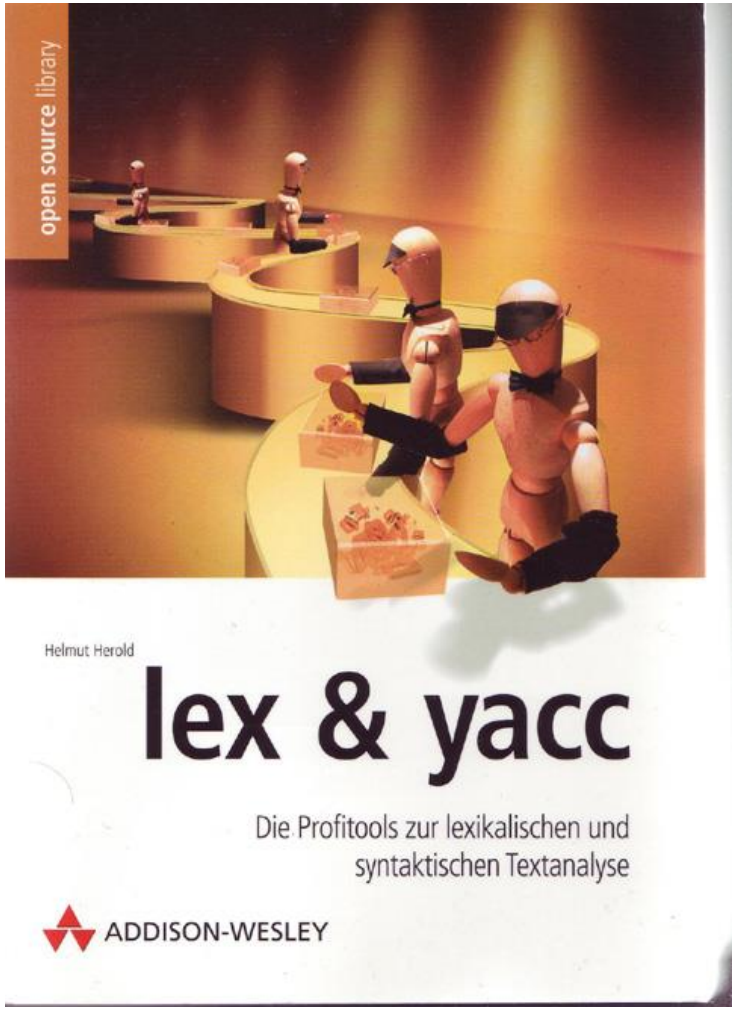
Now there are many ready-to-use tools to construct your own parsers 1975

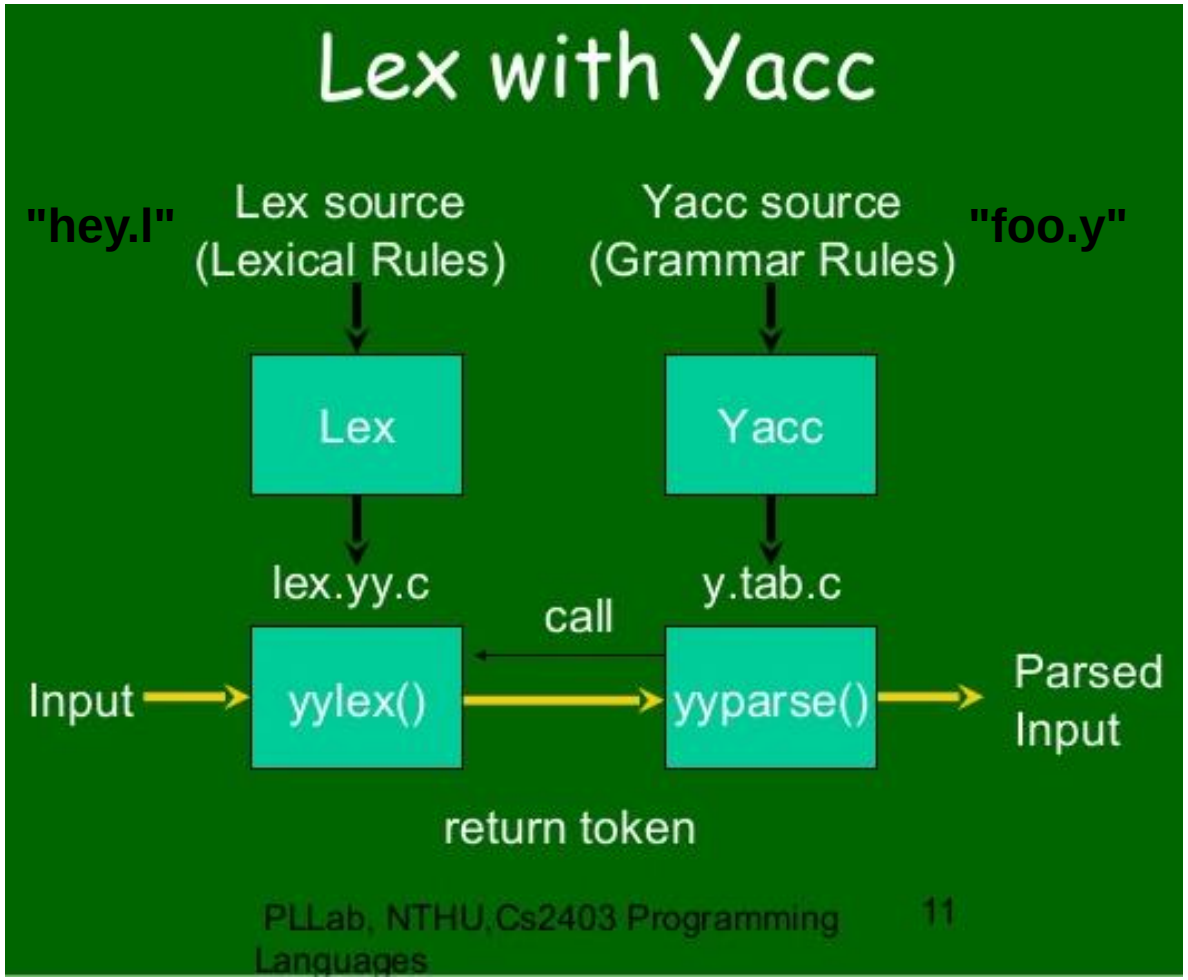
- **LEX and YACC (JYACC – Java version)**
  - A compiler or interpreter for a programming language is often decomposed into two parts:
    1. Read the source program and discover its structure.
    2. Process this structure, e.g. to generate the target program.
  - Lex and Yacc can generate program fragments that solve the first task.
  - The task of discovering the source structure again is decomposed into subtasks:
    1. Split the source file into tokens (**Lex**).
    2. Find the hierarchical structure of the program (**Yacc**).

Yacc:
Same pronunciation with Yak=牦牛

http://dinosaur.compilertools.net/

all your files with extensions (".l") and (".y")





Videos for Lex + Yacc

- Part 01- Tutorial on lex yacc
- Part 02- Tutorial on lex yacc.

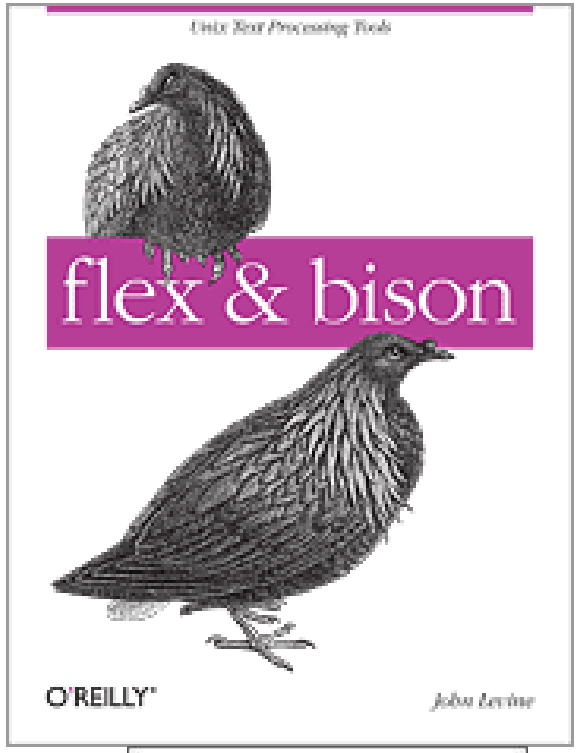
Flex, Bison

Bison [ˈbaisn] :美洲野牛

1988

lex vs. flex, yacc vs. bison

- In addition to hearing about "flex and bison", you will also hear about "lex and yacc".
  - "lex and yacc" are the original tools; **"flex and bison" are their almost completely compatible newer versions**.
  - All four of the above are **C-based tools**; they're written in C, but more important their output is C code.
- Only very old code uses lex and yacc; most of the world has moved on to Flex and Bison.
- http://aquamentus.com/flex_bison.html



Flex和Bison的起源

bison来源于yacc，一个由Stephen C. Johnson于1975年到1978年期间在贝尔实验室完成的语法分析器生成程序。正如它的名字（yacc是“yet another compiler compiler”的缩写）所暗示的那样，那时很多人都在编写语法分析器生成程序。Johnson的工具基于D.E.Knuth所研究的语法分析理论（因此yacc十分可靠）和方便的输入语法。这使得yacc在Unix用户中非常流行，尽管当时Unix所遵循的受限版权使它只能够被使用在学术界和贝尔系统里。大约在1985年，Bob Corbett，一个加州伯克利大学的研究生，使用改进的内部算法再次实现了yacc并演变成为伯克利yacc。由于这个版本比贝尔实验室的yacc更快并且使用了灵活的伯克利许可证，它很快成为最流行的yacc。来自自由软件基金会（Free Software Foundation）的Richard Stallman改写了Corbett的版本并把它用于GNU项目中，在那里，它被添加了大量的新特性并演化成为当前的bison。bison现在作为FSF的一个项目而被维护，且它基于GNU公共许可证进行发布。

在1975年，Mike Lesk和暑期实习生Eric Schmidt编写了lex，一个词法分析器生成程序，大部分编程工作由Schmidt完成。他们发现lex既可以作为一个独立的工具，也可以作为Johnson的yacc的协同程序。lex因此变得十分流行，尽管它运行起来有一点慢并且有很多错误。（不过Schmidt后来在计算机行业里拥有一份非常成功的事业，他现在是Google的CEO。）

大概在1987年，Lawrence Berkeley实验室的Vern Paxson把一种用ratfor（当时流行的一种扩展的Fortran语言）写成的lex版本改写为C语言的，被称为flex，意思是“快速词法分析器生成程序”（*Fast Lexical* Analyzer Generator）。由于它比AT&T的lex更快速和可靠，并且就像伯克利的yacc那样基于伯克利许可证，它最终也超越了原来的lex。flex现在是SourceForge的一个项目，依然基于伯克利许可证。

BNF文法

为了编写一个语法分析器，我们需要一定的方法来描述语法分析
号转化为语法分析树的规则。在计算机分析程序里最常用的语言
（*Context-Free Grammar*，CFG）[注3]。书写上下文无关文法的
*Naur*范式（BackusNaur Form，BNF），大致创立于1960年，用来
两名Algol 60委员会的成员命名。

幸运的是，BNF相当简单。这里描述简单算术表达式的BNF足以处理

注3：上下文无关文法（CFG）也被称为短语结构文法（*Phrase-Structu*
言（*Type-3 language*）。计算机理论学家和自然语言学家在20世
了它们。如果你是一名计算机科学家，你通常称呼它们为上下文
你是语言学家，你会称呼它们为短语结构文法或者3型语法，但是

这个语法分析器将基于流行的MySQL开源数据库所使用的SQL版
使用bison语法分析器来分析它的SQL输入，不过出于各种原因，
有建立在MySQL的语法分析器之上，而是基于手册中SQL语言的

实际的MySQL的语法分析器更长更复杂，我们这儿出于讲解目
多较少用到的部分。MySQL的语法分析器采用了特别的方式来
生成C语言的语法分析器，但是用C++编译器来编译，而词法分
码。此外还有一些细节，不过MySQL的许可证不允许它们被摘录
你感兴趣的话，你可以查阅文件sql/sql_yacc.yy，它是源代码
downloads/mysql/5.1.html）的一部分。

JFlex, CUP

1999

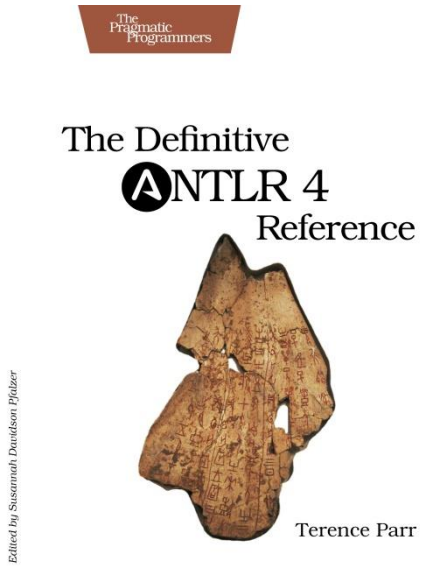
- **JFlex**
  - http://jflex.de/manual.html
    - JFlex is a lexical analyzer generator for **Java**[1] written in Java. A lexical analyser generates a program (a lexer) .
      - ✓ **Lexers usually are the first front-end step in compilers, matching keywords, comments, operators, etc., and generating an input token stream for parsers**.
- **CUP**
  - http://www2.cs.tum.edu/projects/cup/
    - CUP stands for **Construction of Useful Parsers** and is an **LALR parser generator** for **Java**. It implements standard LALR(1) parser generation.
    - JFlex is suited particularly well for collaboration with CUP

ANTLR

Terence Parr is the maniac behind ANTLR and has been working on language tools since 1989. He is a professor of computer science at the University of San Francisco.

1992

- **http://www.antlr.org/**
  - ANTLR (**AN**other **T**ool for **L**anguage **R**ecognition) is a powerful parser generator for reading, processing, executing, or translating structured text or binary files.
    - From a grammar, ANTLR generates a parser that can build and walk parse trees.



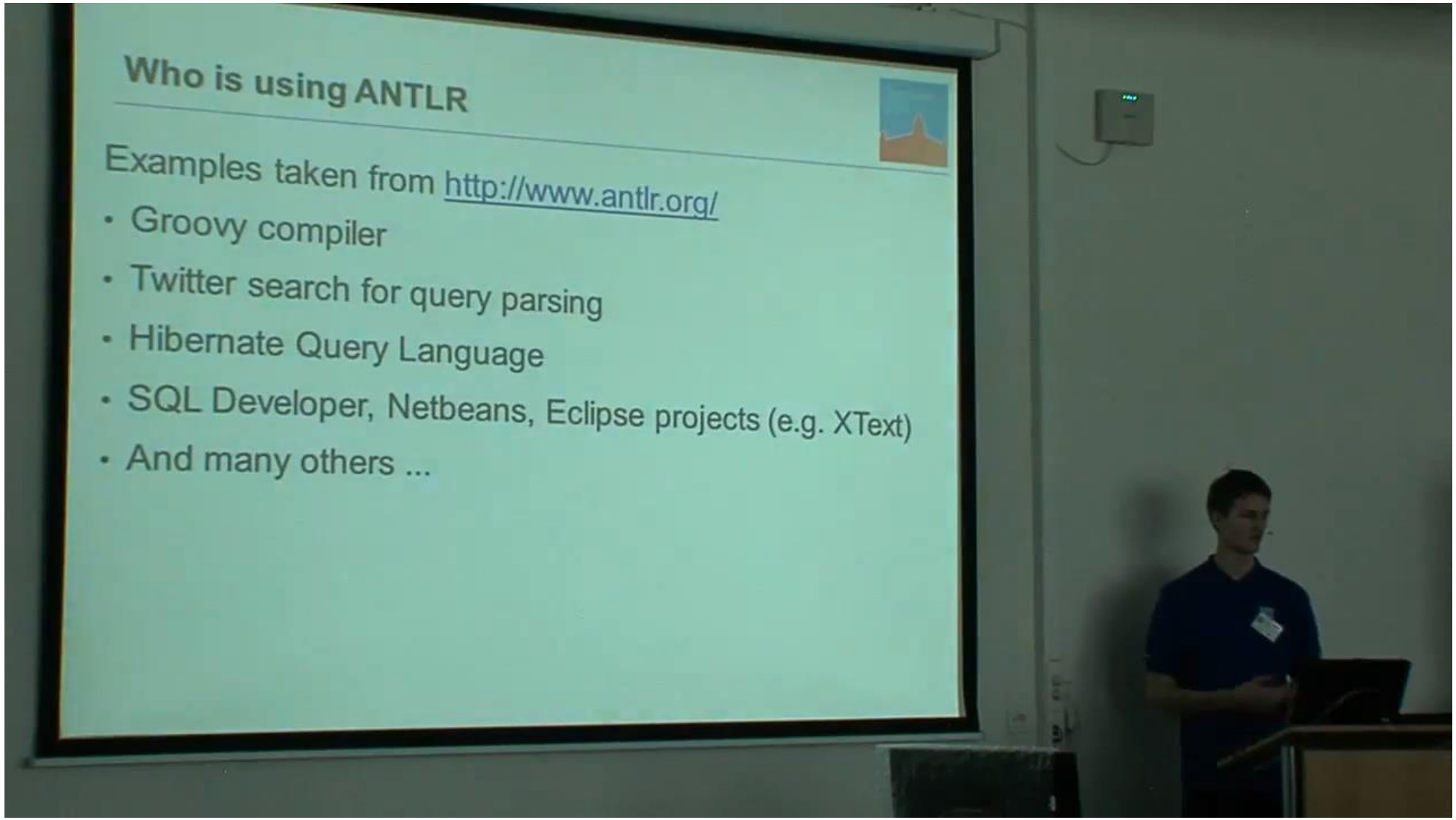
Programmers run into parsing problems all the time. Whether it's a data format like JSON, a network protocol like SMTP, a server configuration file for Apache, a PostScript/PDF file, or a simple spreadsheet macro language—ANTLR v4 and this book will demystify the process.

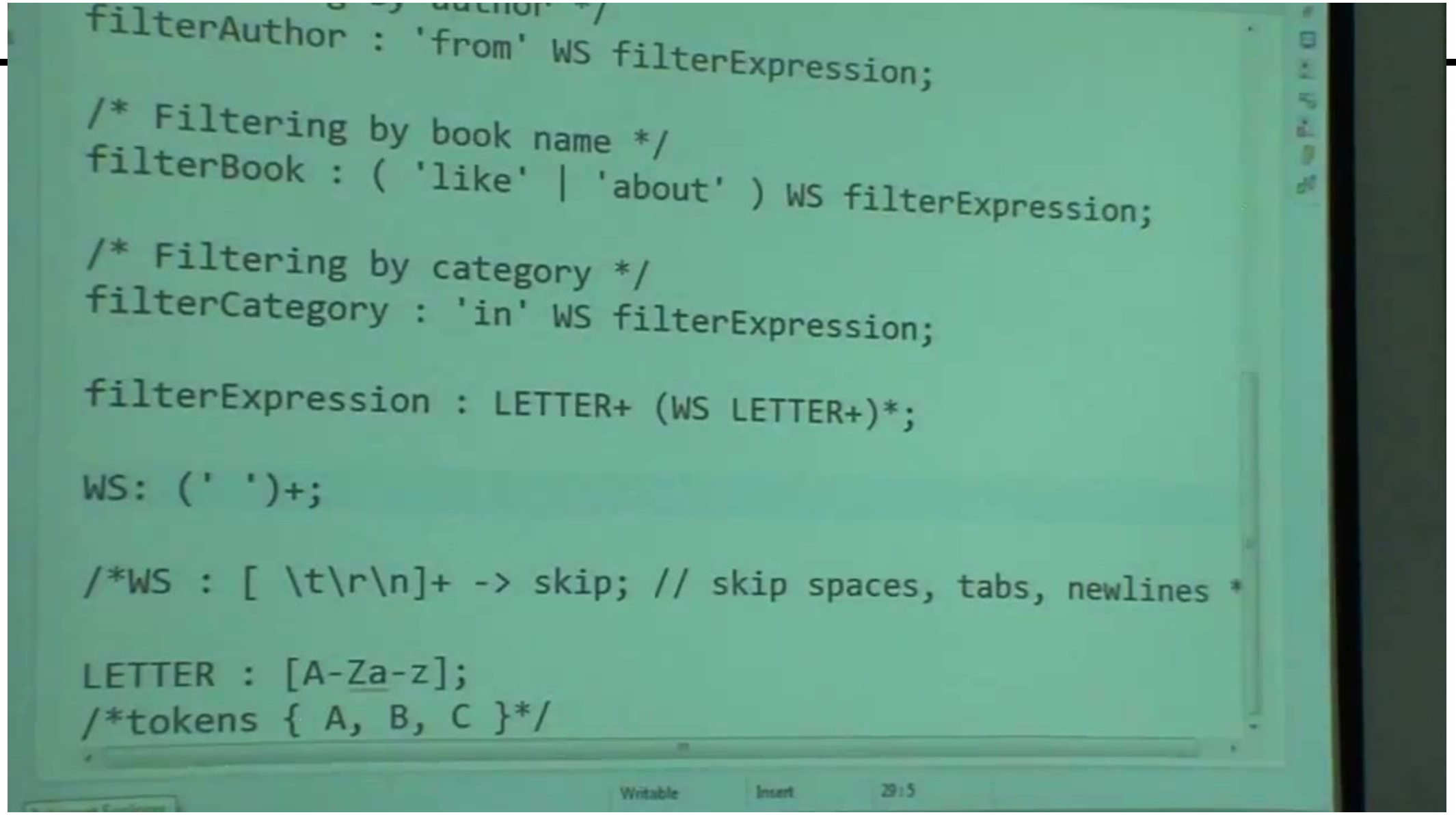
http://www.antlr.org/download.html

- The latest version of ANTLR is 4.6, released December 15, 2016. As of 4.6, we have these code generation targets:
  - Java
  - C# (and an alternate C# target)
  - Python (2 and 3)
  - JavaScript
  - Go
  - C++
  - Swift
- All users should download the ANTLR tool itself and then choose a runtime target below, unless you are using Java which is built into the tool jar.

Videos

- [Terence_Parr]_The_Definitive_ANTLR_4_Reference,_2ed.pdf
- The Definitive ANTLR 4 Reference.mp4
- ANTLR v4 with Terence Parr.mp4
- Language recognition using Antlr (Vladimir Hlavacek, Lukas Jusko).mp4
  - An example to use ANTLR


Who is using ANTLR
Examples taken from http://www.antlr.org/
• Groovy compiler
• Twitter search for query parsing
• Hibernate Query Language
• SQL Developer, Netbeans, Eclipse projects (e.g. XText)
• And many others ...



```
filterAuthor : 'from' WS filterExpression;

/* Filtering by book name */
filterBook : ( 'like' | 'about' ) WS filterExpression;

/* Filtering by category */
filterCategory : 'in' WS filterExpression;

filterExpression : LETTER+ (WS LETTER+)*;

WS: (' ')+;

/*WS : [ \t\r\n]+ -> skip; // skip spaces, tabs, newlines *

LETTER : [A-Za-z];
/*tokens { A, B, C }*/
```

Many other "HIGH-END" applications

- **There are even companies for producing parsers**
  - Due to the complexity of the SQL grammar, many people have attempted but failed to generate a successful parser. Here, at Gudu Software, we have developed a parser that can successfully reduce the difficulties associated with decoding SQL grammar.

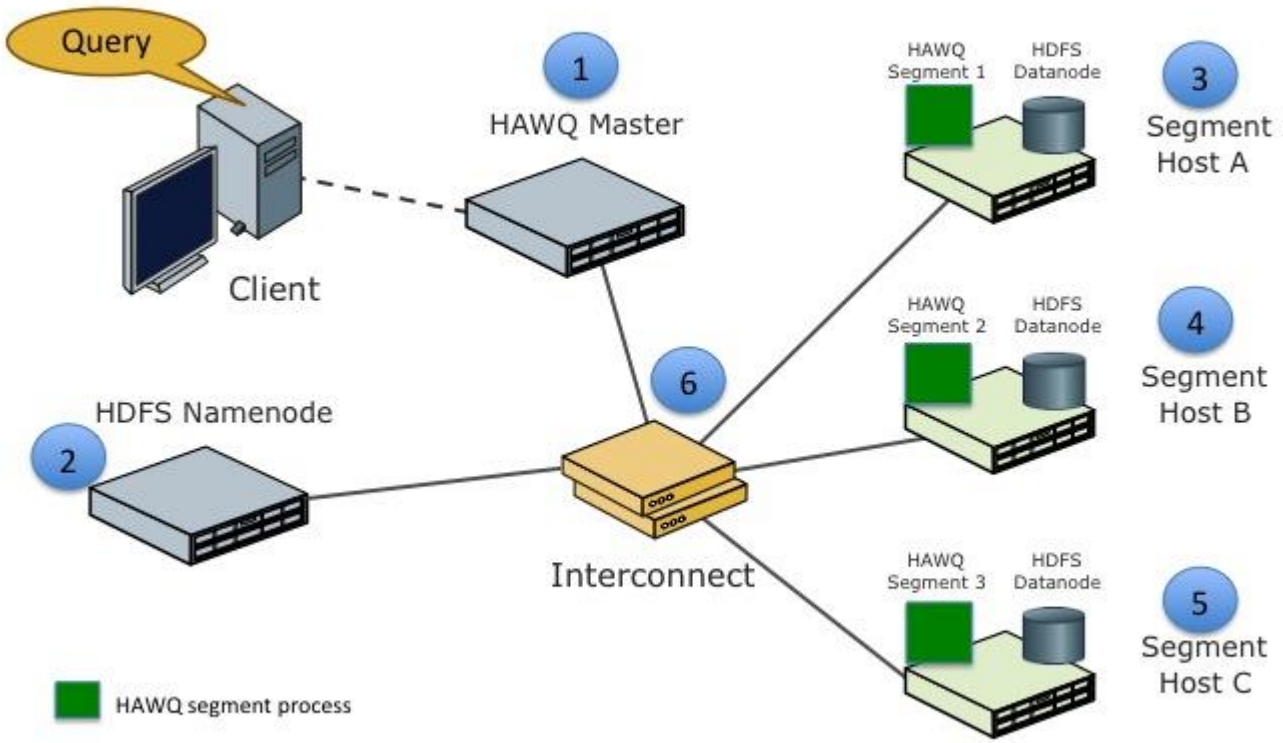
- **Data Integration**
  - Data are scattered in remote computers, how to support the execution of a SQL?
    - http://research.cs.wisc.edu/dibook/
- **Spark SQL for Big Data**
  - Deep Dive into Spark SQL's Catalyst Optimizer
  - https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

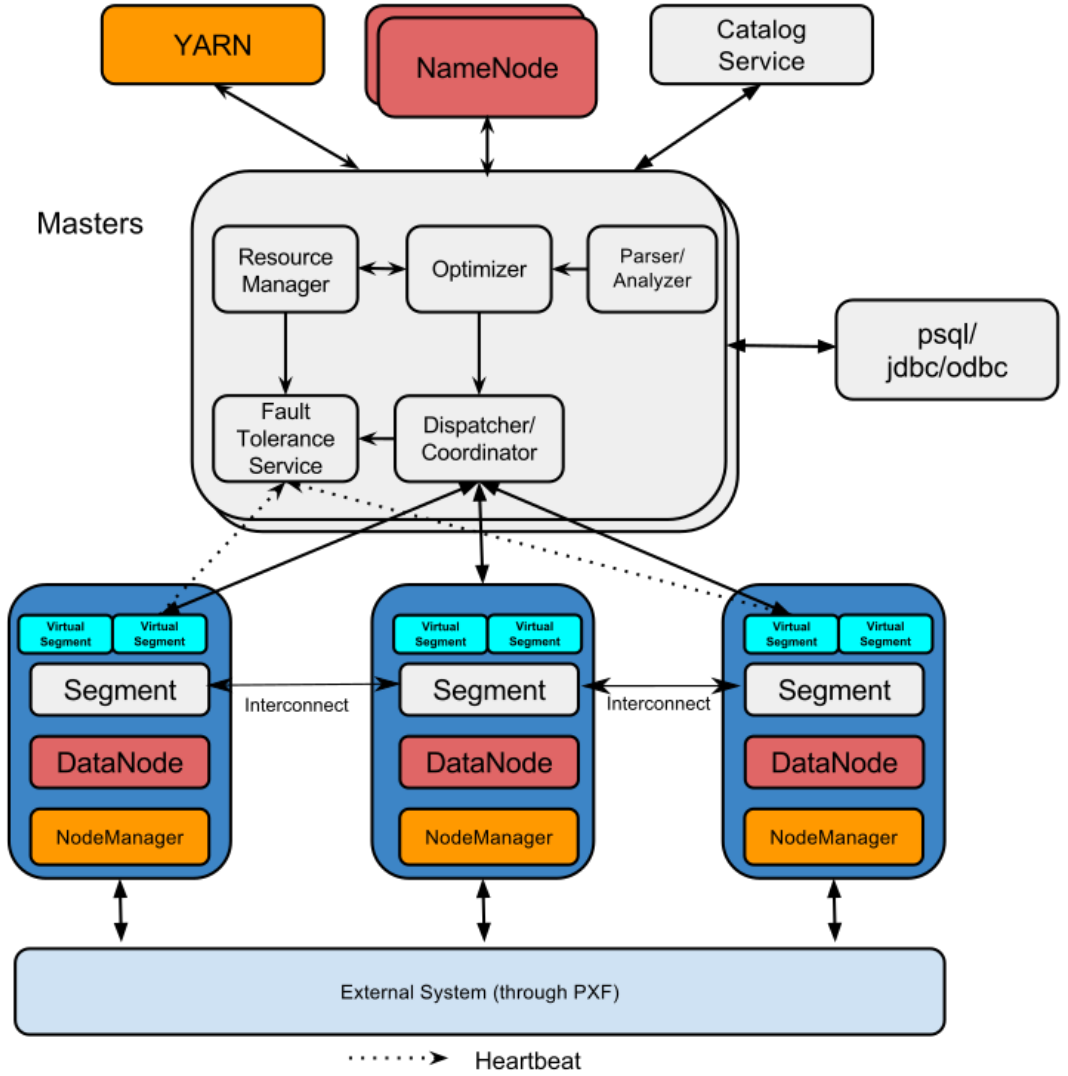
HAWQ


专访偶数科技创始人兼CEO常
雷：一个创业者的野蛮生长
2017-12-29


偶数科技获红杉中国、红点中国
投资
2017-10-31

HAWQ Physical Architecture




YARN
NameNode
Catalog Service
Masters
Resource Manager
Optimizer
Parser/ Analyzer
Fault Tolerance Service
Dispatcher/ Coordinator
psql/ jdbc/odbc
Virtual Segment
Virtual Segment
Segment
DataNode
NodeManager
Interconnect
Interconnect
External System (through PXF)
Heartbeat

Chapter 3: Parse SQL

- **SQL as a language – 4 GL (Generation Language) – follows some rule**
  - **Syntax** should be followed – How is syntax defined?
  - **Automata** are used to **model syntax**, and could **verify** if a sentence *s.t* a given language?
  - Precursors proposed a **framework** to derive operations from the legal sentence
- **3rd training project – a Naïve SQL parser**

Select B,D

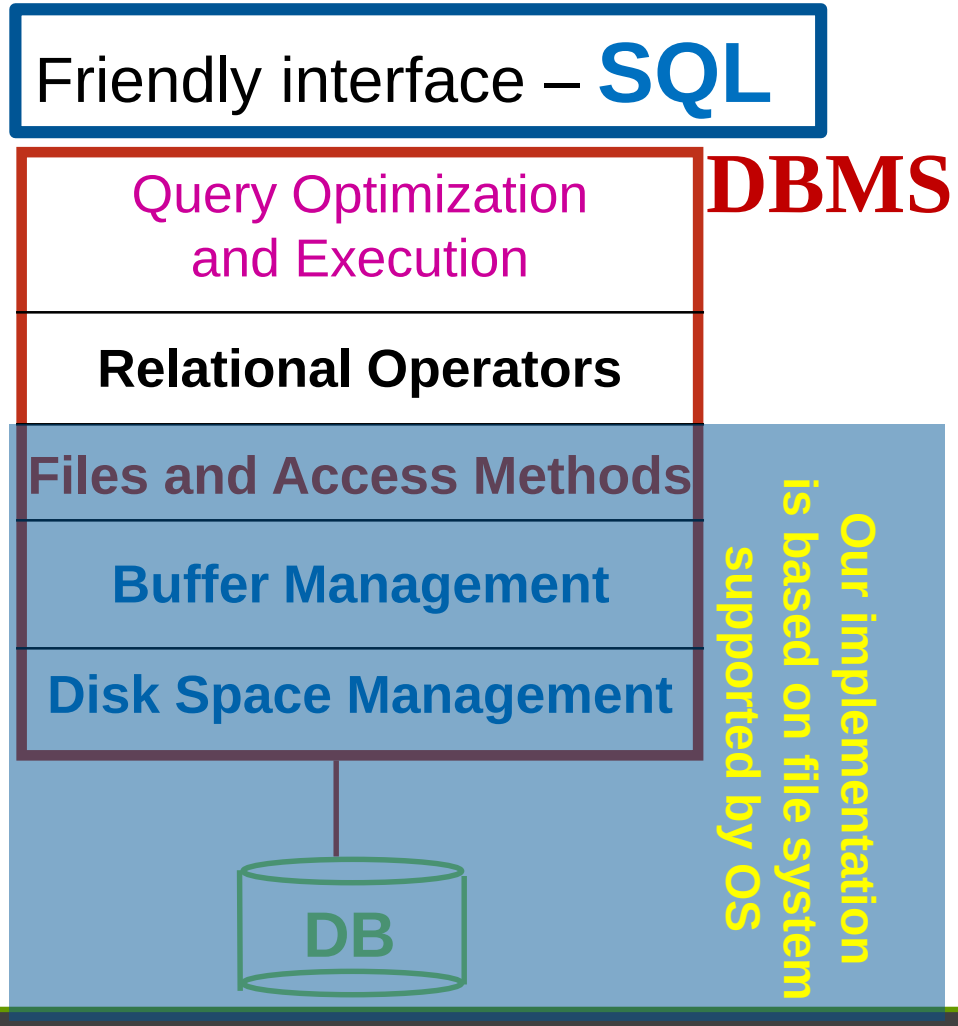
From R,S

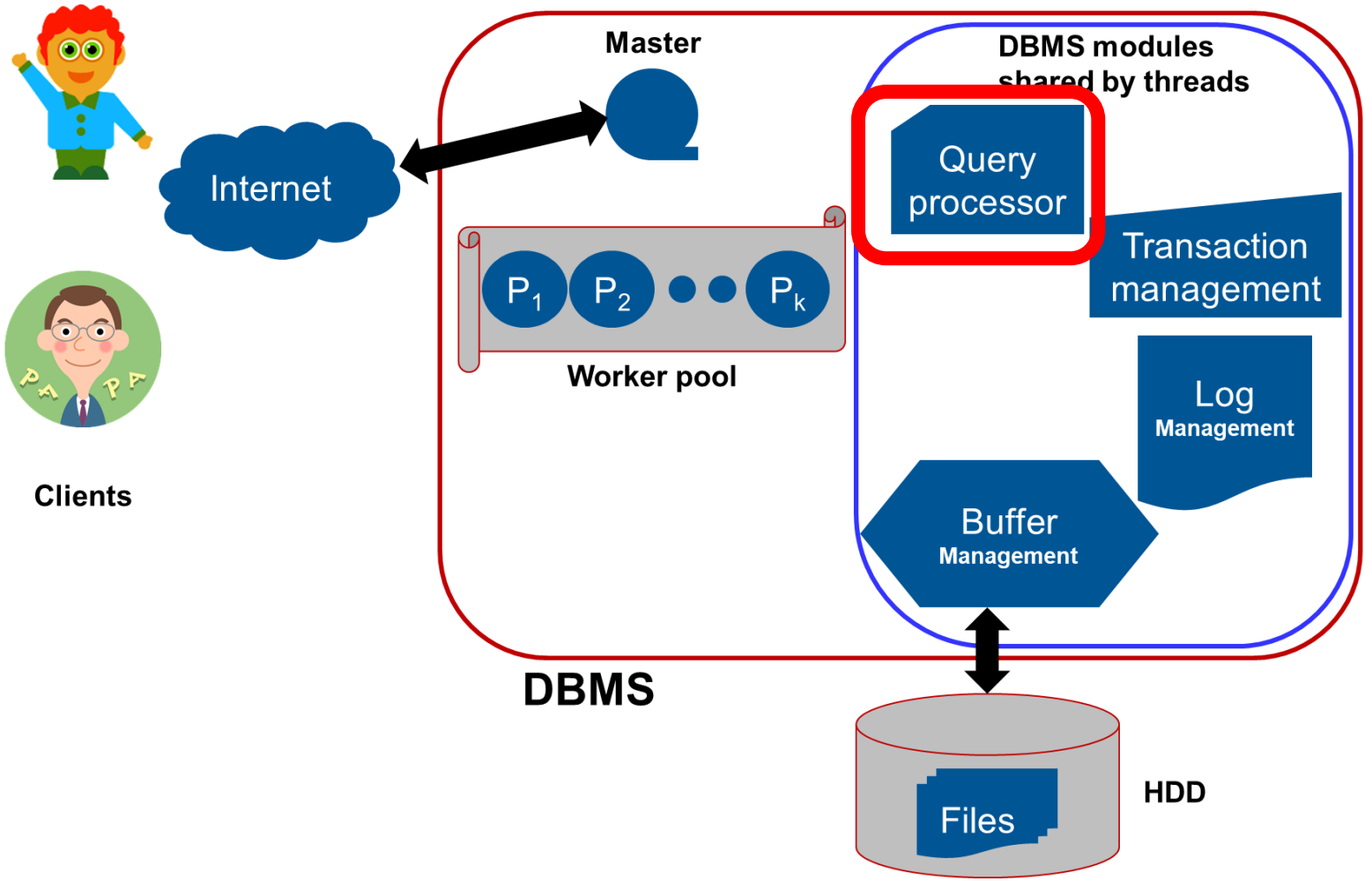
Where R.A = “c” ∧ R.C=S.C

Your project should at least understand SQL like this ☺

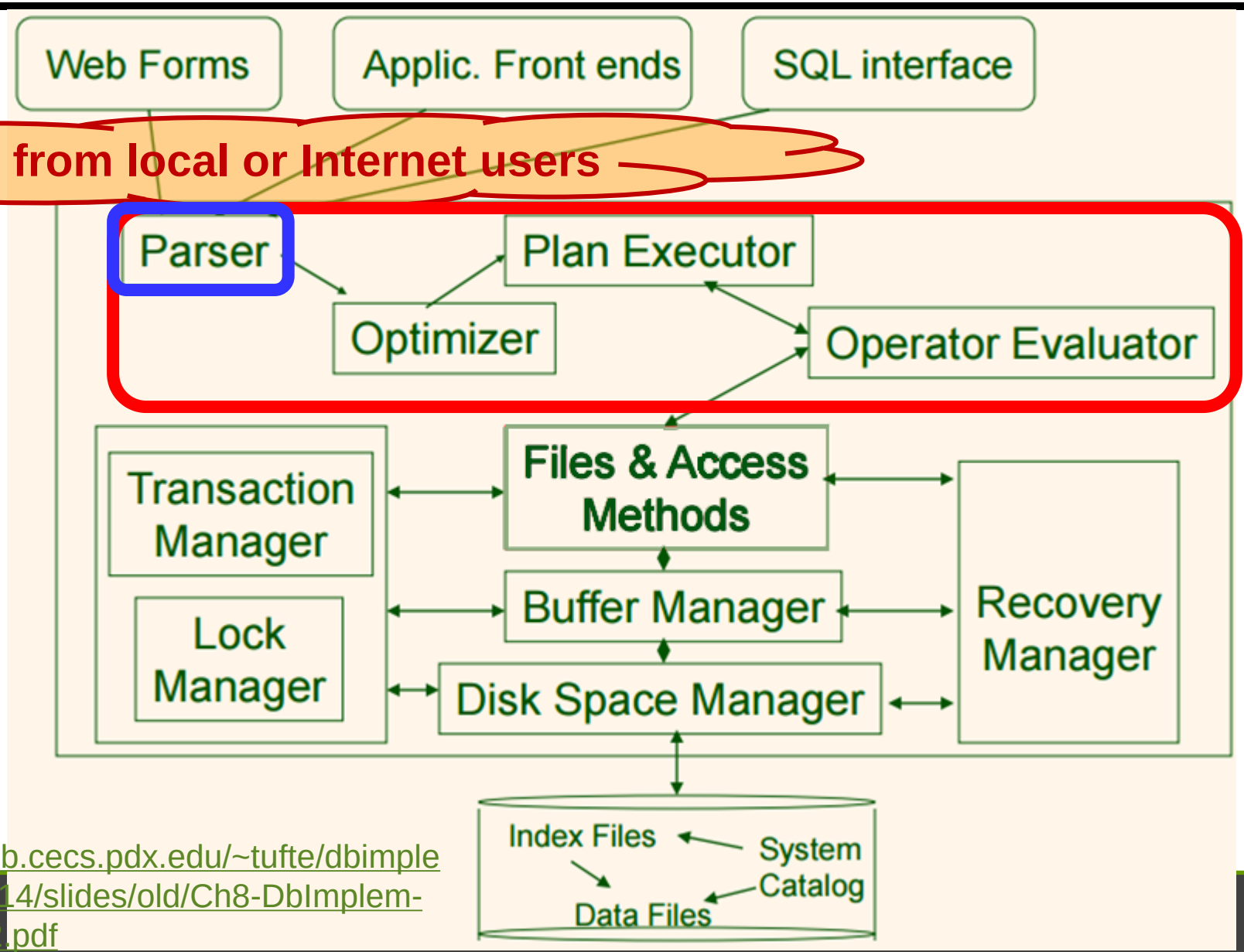
DBMS

- **Top goal of DBMS**
  - Support the **concurrent** **data access** for **many clients/users**
- **2 roles of DBMS**
  1. **Concurrent Data Management** by using files (which are supported by OS)
  2. **Provide friendly/flexible interaction for users** to use




Master
DBMS modules
shared by threads
Internet
Query
processor
Transaction
management
P_1 P_2 P_k
Worker pool
Log
Management
Buffer
Management
Clients
DBMS
Files
HDD

Database Architecture



http://web.cecs.pdx.edu/~tufte/dbimplem-spr2014/slides/old/Ch8-DbImplem-Fall2012.pdf

- **For the 2nd role, SQL (Structured Query Language) is traditionally the standard**
  - **DDL**: Data Definition Language
    - define the database structure or schema.
  - **DML**: Data Management Language
    - manage data within schema objects
  - **DCL**: Data Control Language
    - Gives/ withdraw user's access privileges to database
  - **TCL**: Transaction Control Language
    - manage the changes made by DML statements.
    - It allows statements to be grouped together into logical transactions

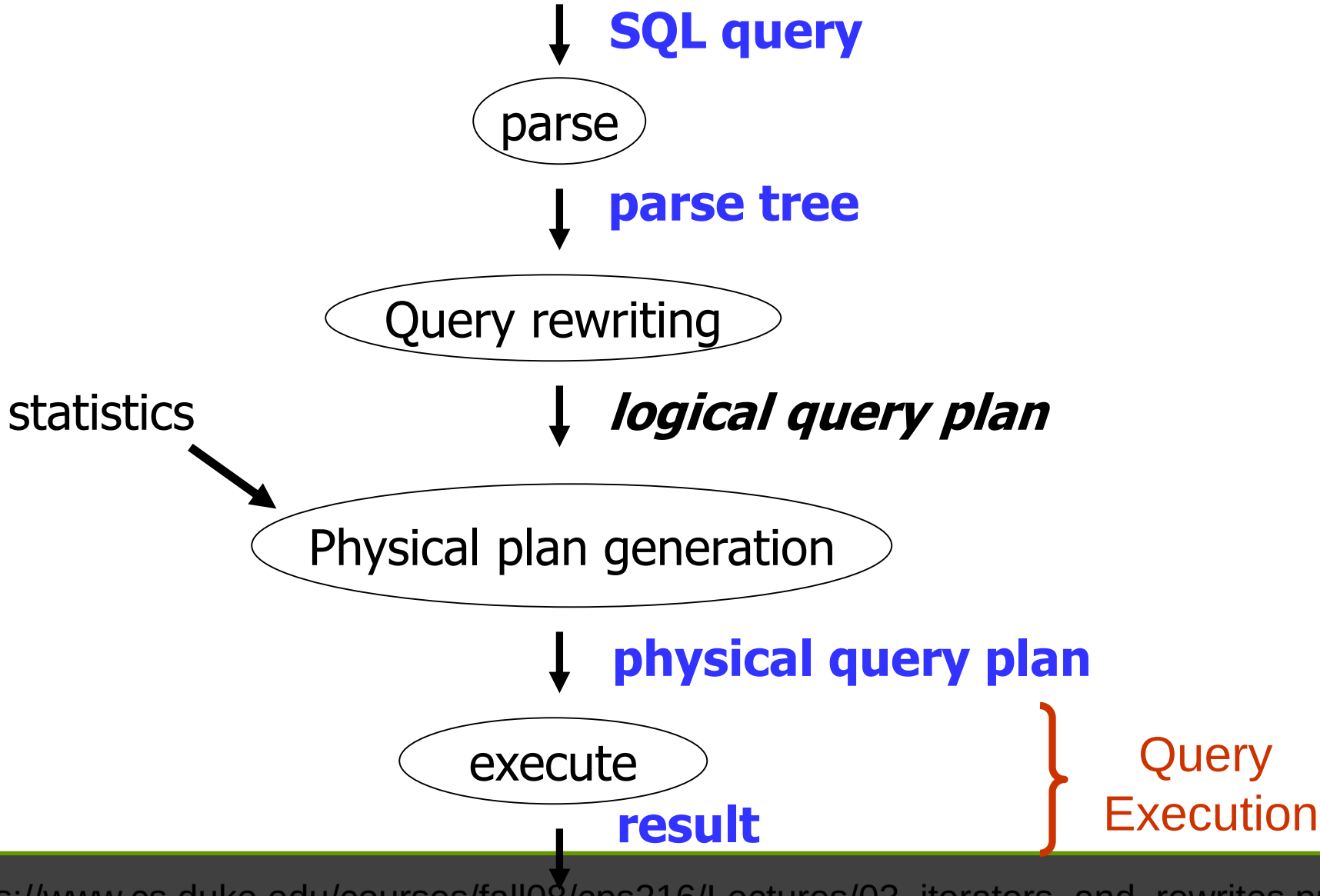
SQL examples

- CREATE moneyNote (time date, money float);
- UPDATE moneyNote SET money=200.0 WHERE time=201602221432
- SELECT AVG(money) FROM moneyNote
- SELECT title

```
FROM StarsIn
WHERE starName IN (
    SELECT name
    FROM MovieStar
    WHERE birthdate LIKE '%1960'
);
```

- …

Overview of Query Processing



History of SQL

- **In 1974, D. Chamberlin (IBM San Jose Laboratory) defined language called 'Structured English Query Language' (SEQUEL).**
  - A revised version, SEQUEL/2, was defined in 1976 but name was subsequently changed to SQL for legal reasons.
  - Still pronounced 'see-quel', though official pronunciation is 'S-Q-L'.

- **IBM subsequently produced a prototype DBMS called System R, based on SEQUEL/2.**

- **Roots of SQL, however, are in SQUARE (Specifying Queries as Relational Expressions), which predates System R project.**

- IBM San Jose Research Centre
- Members of System R Team
  - System R (1975) first prototype RDBMS
  - Pioneered architecture of modern RDBMSs
- Chamberlin: Design of SQL (SEQUEL)
- Lorie: Buffer manager, optimizer, SQL
- Mehl: First SQL parser
- to see a modern SQL parser:
  - postgresql-11.3/src/backend/parser

After the proposal of SQL, (R)DBMS becomes popular!

- In late 1970s, ORACLE appeared and [illegible] RDBMS based on SQL.
- In 1987, ANSI and ISO published an initial [illegible] QL.
- In 1989, ISO published an addendum that defined an 'Integrity Enhancement Feature'.
- In 1992, first major revision to ISO standard occurred, referred to as SQL2 or SQL/92.

- In 1999, SQL3 was released with support for object-oriented data management.

- **Use extended form of BNF notation [EBNF]:**
  - **Upper-case letters** represent reserved words.
  - **Lower-case letters** represent user-defined words.
  - **|** indicates a choice among alternatives.
  - **Curly braces** indicate a required element.
  - **Square brackets** indicate an optional element.
  - **…** indicates optional repetition (0 or more).

http://savage.net.au/SQL/sql-92.bnf.html#query%20specification

- **SQL (like other computer languages) is formally defined in a notation called "Backus-Naur Form"**
- **Subset of SQL may be**

<query specification> ::=
SELECT [<set quantifier>] <select list> <table expression>

<select list> ::=
<asterisk>
| <select sublist> [{ <comma> <select sublist> }...]

<select sublist> ::= <derived column> | <qualifier> <period> <asterisk>

<derived column> ::= <value expression> [<as clause>]

<as clause> ::= [AS] <column name>

<table expression> ::=
<from clause>
[<where clause>]
[<group by clause>]
[<having clause>]

<from clause> ::= FROM <table reference> [{ <comma> <table reference> }...]

A Simple Grammar – Queries

- **The syntactic category `<Query>` represents SQL queries**
- **For this simple grammar there is just one rule for queries**

  - The symbol `::=` means “can be expressed as”
  - The query rule omits GROUP BY, HAVING and (many) other optional clauses

```
<Query> ::= SELECT <SelList> FROM <FromList> WHERE <Condition>
```

Select Lists

- **A *select list* is a comma separated list of attributes**
  - A single attribute, or
  - An attribute, a comma and a list of attributes
- **These rules do not provide for expressions, aggregations and aliases**

```
<SelList> ::= <Attribute>, <SelList>
<SelList> ::= <Attribute>
```

From Lists

- A *from list* is a comma separated list of relations
- These rules do not provide for joins, sub-queries and tuple variables

```
<FromList> ::= <Relation>, <FromList>
<FromList> ::= < Relation>
```

Conditions

- **This abbreviated set of rules does not include**
  - OR, NOT and EXISTS
  - Comparisons not on equality or LIKE
  - ...

```
<Condition> ::= <Condition> AND <Condition>
<Condition> ::= <Attribute> IN <Query>
<Condition> ::= <Attribute> = <Attribute>
<Condition> ::= <Attribute> LIKE <Pattern>
```

Base Syntactic Categories

- **There are three base syntactic categories**
  - <Attribute>, <Relation>, and <Pattern>.
  - These categories are not defined by rules but by which atoms they can contain

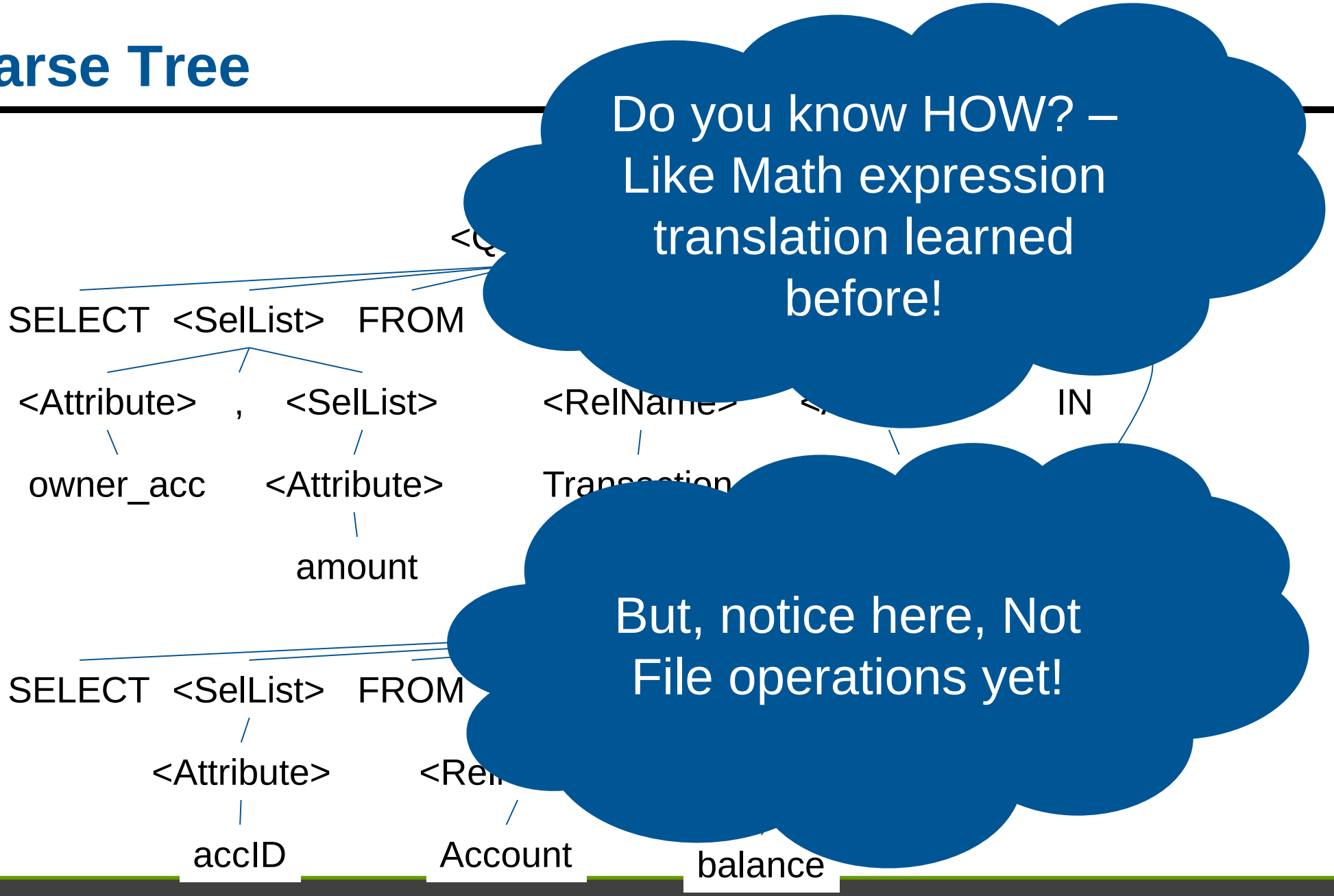
- **An <Attribute> can be any string of characters that identifies a legal attribute**
- **A <Relation> can be any string of characters that identifies a legal relation**

Example

- **Consider two relations**
  - *Account* = {*accID*, *balance*, *ownerSIN*}
  - *Transaction* = {*transID*, *amount*, *date*, *owner_acc*}
- **And a query ...**

```
SELECT owner_acc, amount
FROM Transaction
WHERE owner_acc IN(
        SELECT accID
        FROM Account
        WHERE balance = 1000000)
```

Example Parse Tree



Could you?

Select B,D
From R,S
Where R.A = “c” ∧ R.C=S.C

Chapter 3: Parse SQL

- **SQL as a language – 4 GL (Generation Language) – follows some rule**
  - **Syntax** should be followed – How is syntax defined?
  - **Automata** are used to **model syntax**, and could **verify** if a sentence *s.t* a given language?
  - Precursors proposed a **framework** to derive operations from the legal sentence
- **3rd training project – a Naïve SQL parser**

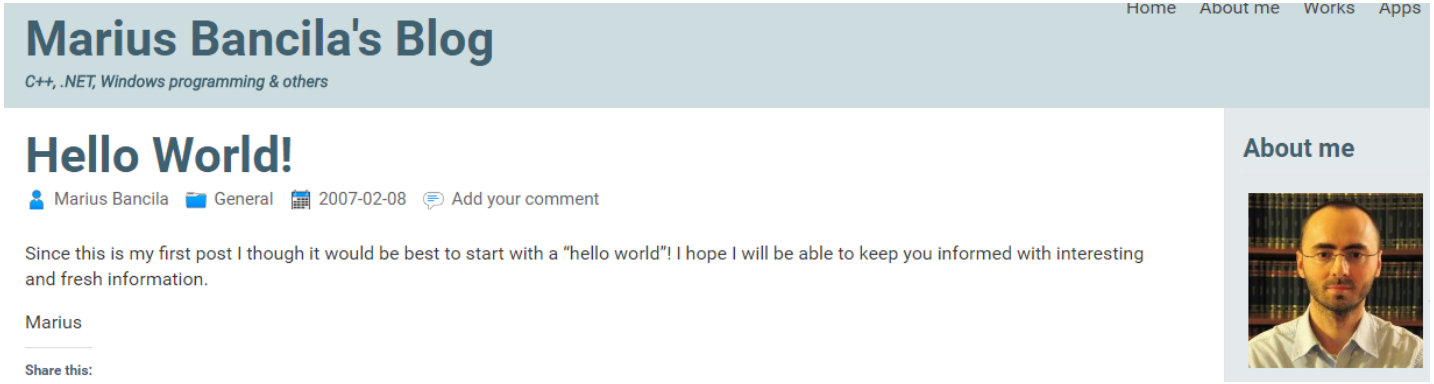
Select B,D

From R,S

Where R.A = “c” ∧ R.C=S.C

Your project should at least understand SQL like this ☺

1. Try to add cos() and sin()



- **cos, sin etc.?**
  - cos(num)
  - sin(num)
  - The num is used as radian [弧度]
- **How to implement Lexer now?**
  - Only after the Lexer can we parse the math expression
- **How to define Syntax and use later tools to finish the above Math functions?**

Select B,D

From R,S

Where R.A = “c” $\wedge$ R.C=S.C

2. Show us how to use **FLEX** and **Bison** to parse SQL (a subset will be enough)

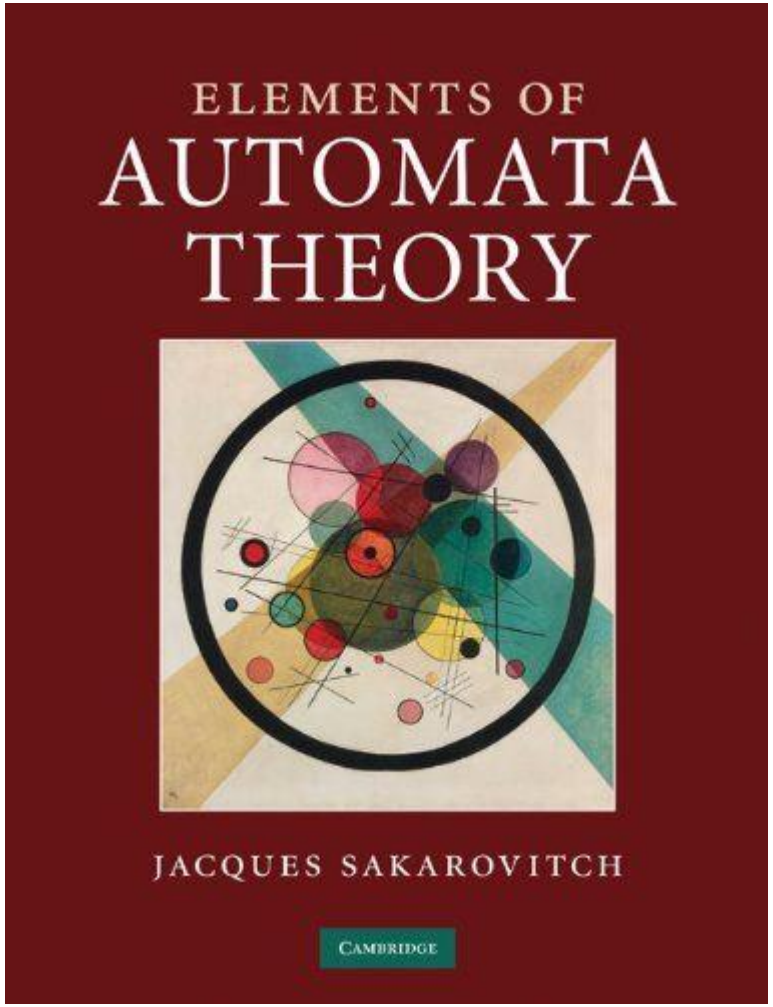
3. **Or use a popular ready-to-use compiler generator (ANTLR?) to show the parse tree of following SQL**

Select B,D

From R,S

Where R.A = “c” ∧ R.C=S.C

- You should explain well how this SQL is translated by using that tool
  - History of the background
  - The story about that tool
  - The steps and some details to use it
  - Key data structures
  - The meaning of the key functions
  - …



- Elements of Automata Theory
- Authors Jacques Sakarovitch
- Year 2009
- Pages 782
- Publisher Cambridge University Press
- Language English
- ISBN 9780521844253