Lecture 3.1

编译器前端

徐 辉

xuh@fudan.edu.cn

大纲

- 一、目标语言
- 二、前端功能需求分析
- 三、词法设计和解析
- 四、句法设计和解析

我们想要设计一门怎样的语言？

C++ (Bjarne Stroustrup)
1985-now (C++ 20)

VM/GC/静态类型

Java (James Gosling @ Sun)
1995-now（JDK 19）

脚本语言
动态强类型

Python (Guido van Rossum)
1991-now (3.10)

脚本语言
动态弱类型

JavaScript (Brendan Eich @ Netscape)
1995-now (ECMAScript 2021)

Rust (Graydon Hoare @ Mozilla)
2010 – now（1.64）

GC/静态类型

Golang (R. Griesemer @ Google)
2009-2012（1.19）

考量因素

- 基本组成（风格）
  - 变量声明
  - 函数声明
  - 控制流语句
  - . . .
- 功能特性和范式
  - 虚拟机（后端）
  - 内存管理：垃圾回收、智能指针
  - 并行
  - 类型系统：缺省类型、类型推断、泛型
  - 函数式：lambda算子、Monad
  - 面向对象
  - …

基本组成（风格）：变量声明

```
int a = 1;
auto a = 1;
```

缺省类型，需要auto占位

```
a = 1;
```

```
let a = 1i32;
let a:i32 = 1;
let a = 1;
```

类型后缀，可缺省，易于递归下降解析

```
int a = 1;
```

```
var a = 1;
let a = 1;
a = 1;
```

```
var a int
a := 1;
```

类型后缀，不可缺省

基本组成（风格）：函数声明

```
int add (int x, int x);
```

```
def add (x, y):
    x + y;
```

✔

```
fn add_one (x: i32, y: i32) -> i32 {
    x + y
}
```

```
public class MyNumber {
    public Integer add (Integer y);
}
```

```
function add (x, y) {
    return x + y;
}
```

```
func add(x int, y int) int {
    return x + y
}
```

类似，易于递归下降解析

功能差异（值传递vs引用传递）：C++

```cpp
#include <iostream>
using namespace std;
int test(std::string& s){
    s = "changed!";
    return 0;
}

int main() {
    std::string s = "new string";
    test(s);
    cout << s << endl;
}
```

```cpp
#include <iostream>
using namespace std;
int test(std::string* p){
    std::string s1 = "changed";
    p = &s1;
    return 0;
}
int main() {
    std::string s = "new string";
    test(&s);
    cout << s << endl;
}
```

```
changed!
```

```
new string
```

功能差异（值传递vs引用传递）：Java

```
public class Main {
    public static void Test(String s){
        s = "changed!";
        System.out.println(s);
    }
    public static void main(String args[]){
        String s = "new string";
        System.out.println(s);
        Test(s);
        System.out.println(s);
   }
}
```

```
new string
changed!
```

功能差异（值传递vs引用传递）：Go

```go
package main
import ("fmt")
func test(s string) {
    s = "changed"
}

func main() {
    s := "new string"
    test(s)
    fmt.Println(s)
}
```

```go
package main
import ("fmt")
func test(p *string) {
    s := "changed"
    p = &s
}
func main() {
    s := "new string"
    test(&s)
    fmt.Println(s)
}
```

```
changed!
```

```
new string
```

功能差异（值传递vs引用传递）：Rust

```
fn test(s:&mut String){
    s.clear();
    s.insert_str(0,"changed!");
    println!("{}",s);
}
fn main() {
    let mut s = String::from("new string");
    println!("{}",s);
    test(&mut s);
    println!("{}",s);
}
```

控制流语句：for

```rust
let mut sum = 0;
for x in 1..100 {
    sum = i + 1;
}
```

```rust
let vec v = vec!["a", "b", "c"];
for s in v.iter() {
    println!("{}", s);
}
```

```go
sum := 0
for i := 1; i < 5; i++ {
    sum += i
}
```

```go
strings := []string{"a", "b"}
for i, s := range strings {
    fmt.Println(i, s)
}
```

控制流语句风格：loop

```rust
let mut sum = 1
while sum < 100 {
    sum += sum
}
```

```rust
let mut sum = 1
loop {
    sum += sum
    if sum >= 100
        break;
}
```

```go
sum := 1
for sum < 100 {
    sum += sum
}
```

```go
sum := 1
for {
    sum += sum
    if sum >= 100
        break
}
```

控制流语句风格：match-case

```rust
let number = 23;
let mut r;
match number {
    0 => { println!("error");
           r = -1; }
    _ => r = 100 % number,
}
println!("{}",r);
```

```rust
let number = 23;
let mut r = match number {
    0 => { println!("error"); -1 }
    _ => 100 % number
}
println!("{}",r);
```

```go
num := 23
var r int
switch num {
    case 0:
        fmt.Println("error")
        r = -1
    default:
        r = 100 % num
}
fmt.Println(r)
```

函数式风格

```rust
let a = 10;
let cl= |i: i32| -> i32 { i + a };
println!("{}", cl(1));
```

```cpp
int a = 10;
auto cl = [=](int i) {
    return a + i;
};
std::cout << cl(1) << std::endl;
```

捕获选项：

[ ]：不捕获

[&]：所有变量的引用

[=]：所有变量的值

[a]：仅捕获特定变量

```go
func test(i int) func() int {
   a := 10
   return func () int {
       return i + a
   }
}

func main() {
   cl  := test(1)
   fmt.Println(cl())
}
```

没有直接的实现方式，
通过返回匿名函数实现

Go的语言特色：并发

```go
func main() {
    go foo()
    foo()
}
```

Goroutines：协程（用户态线程）运行

```go
func foo(c chan int) {
    for i := 0; ; i++ {
        c <- i
    }
}
func main() {
    var c chan int = make(chan int)
    go foo(c)
    for {
        msg := <- c
        fmt.Println(msg)
    }
}
```

channel：线程间通讯

- ->表示数据流方向
- 阻塞模式

https://go.dev/talks/2012/concurrency.slide

Rust的语言特色：内存安全 + 并发安全

每一个时刻只有一个所有者（写权限）

```
fn main(){
  let mut alice = Box::new(1);
  let bob = &alice;
  let carol = alice;
}
```

bob借用Box所有权，自动归还

转移Box所有权给carol，不归还

```
unsafe impl Send for MyStruct{}
unsafe impl Sync for MyStruct{}
let mut v = MyStruct{...};
let tid = thread::spawn(move || {
    for i in 1..100001{
        v.add(i);
    }
});
```

MyStruct可以在线程间转移所有权

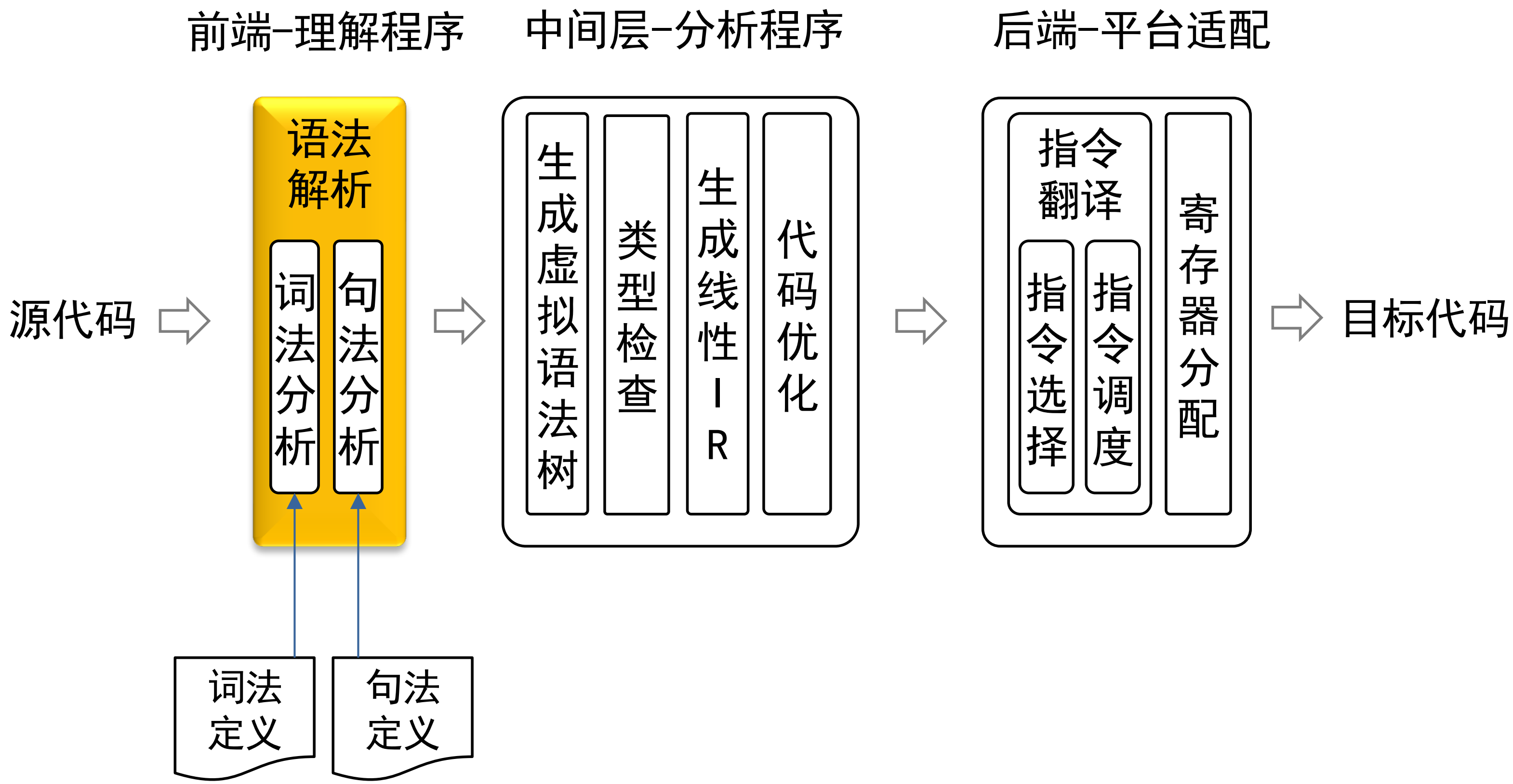
MyStruct可以在线程间借用

在子线程访问v

大纲

- 一、目标语言
- 二、前端功能需求分析
- 三、词法设计和解析
- 四、句法设计和解析

基本功能



前端-理解程序
中间层-分析程序
后端-平台适配
源代码
语法解析
词法分析
句法分析
词法定义
句法定义
生成虚拟语法树
类型检查
生成线性IR
代码优化
指令翻译
指令选择
指令调度
寄存器分配
目标代码

前端要求

- 语法定义易于解析
- 解析算法支持错误提示
  - 可精确定位错误字符位置（行号，位置）
- 解析结果便于后续操作
  - 如类型检查和生成线性IR

大纲

- 一、目标语言
- 二、前端功能需求分析
- 三、词法设计和解析
- 四、句法设计和解析

有哪些标签需要定义？

- 数据：
  - 标识符
  - 常量
- 符号：
  - 运算符
  - 分隔符
- 保留字

如何定义常量和标识符？

$$
\begin{aligned}
\mathtt{DIGIT} &:= [0\text{-}9] \\
\mathtt{DIGITS} &:= \mathtt{DIGIT}^{+} \\
\mathtt{FRACTION} &:= .\mathtt{DIGITS} \mid \epsilon \\
\mathtt{EXPONENT} &:= (e(+\mid - \mid \epsilon)\mathtt{DIGITS}) \mid \epsilon \\
\mathtt{UNUM} &:= \mathtt{DIGITS}\ \mathtt{FRACTION}\ \mathtt{EXPONENT}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{DIGIT} &:= [0\text{-}9] \\
\mathtt{LETTER} &:= [a\text{-}zA\text{-}Z] \\
\mathtt{IDENT} &:= \mathtt{LETTER}^{+}\ (\mathtt{LETTER} \mid \mathtt{DIGIT} \mid \_)^{*}
\end{aligned}
$$

运算符

二元运算符

```
<ADD>   :=  +
<SUB>   :=  -
<MUL>   :=  *
<DIV>   :=  /
<REM>   :=  %
```

比较运算符

```
<CEQ>   :=  ==
<NEQ>   :=  !=
 <GT>   :=  >
<GTE>   :=  >=
 <LT>   :=  <
<LTE>   :=  <=
```

位运算符

```
 <SHL>   :=  <<
 <SHR>   :=  >>
<BAND>   :=  &
 <BOR>   :=  |
```

赋值/一元运算符

```
 <EQ>   :=  =
<ADD>   :=  ++
<SUB>   :=  --
<MUL>   :=  +=
<DIV>   :=  -=
```

逻辑运算符

```
<AND>   :=  &&
 <OR>   :=  ||
<NOT>   :=  !
```

更多符号

域		
<LPAR>	:=	(
<RPAR>	:=	)
<LSQ>	:=	[
<RSQ>	:=	]
<LBRA>	:=	{
<RBRA>	:=	}
<SQUT>	:=	'
<DQUT>	:=	"

			用途
<COMMA>	:=	,	分隔多个元素
<SEMI>	:=	;	分隔多条语句
<RARROW>	:=	->	类型声明
<COLON>	:=	:	函数声明
<DOT>	:=	.	访问结构体内部
<DOTS>	:=	..	范围
<QUE>	:=	?	三元运算
<POUND>	:=	#	属性

注释		
<SLASHES>	:=	//
<LSTAR>	:=	/*
<RSTAR>	:=	*/

保留字

函数、变量声明

```
<FN>  := fn
<LET> := let
```

类型

```
<BOOL>   := bool
<CHAR>   := char
<INT>    := int
<LONG>   := long
<FLOAT>  := float
<DOUBLE> := double
<STR>    := str
<STRUCT> := struct
```

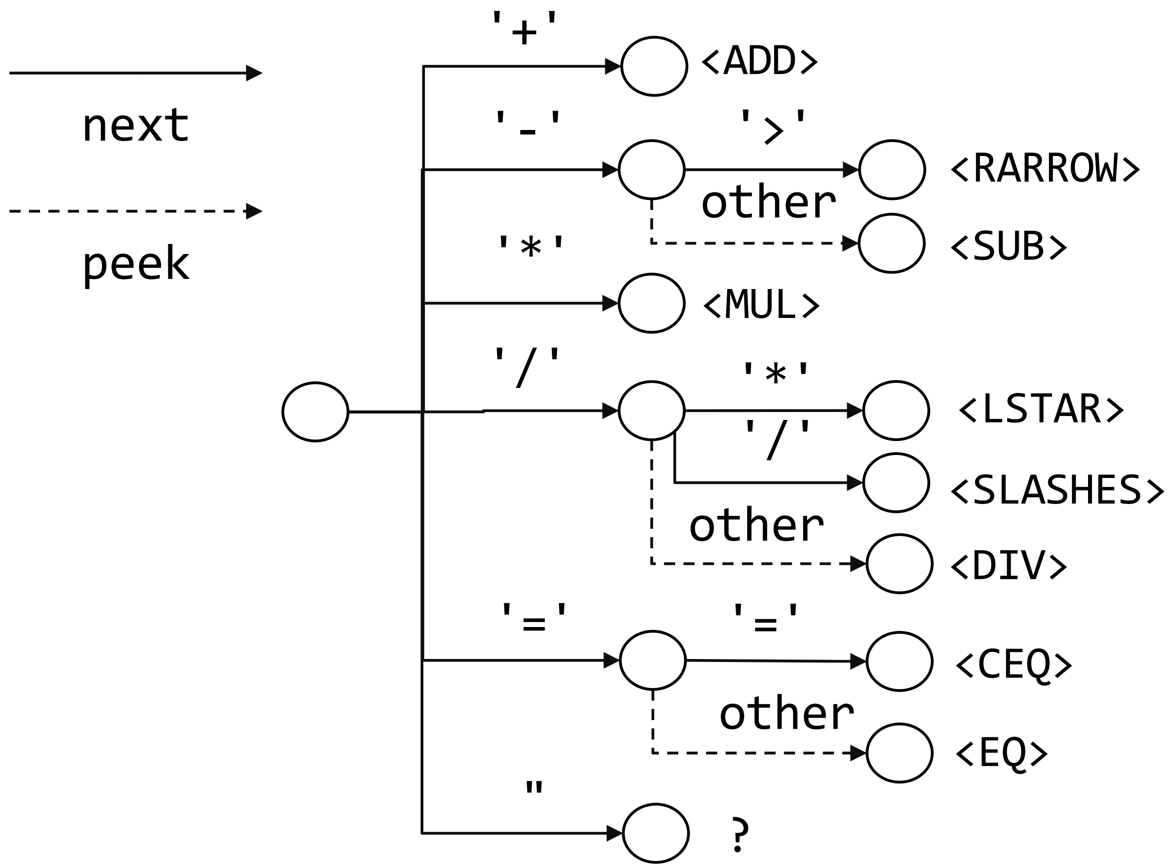
控制流

```
<IF>      := if
<ELSE>    := else
<FOR>     := for
<IN>      := in
<WHILE>   := while
<MATCH>   := match
<CASE>    := =>
<DEFAULT> := _
<BREAK>   := break
```

冲突处理

- 保留字 vs 标识符
  - 保留字优先级高于标识符，如
    - if应识别为<IF>，非<IDENT>
- 多种匹配方案时
  - 选择最长的匹配，如
    - <=不应识别为<和=
    - ifabc不应识别为if和abc

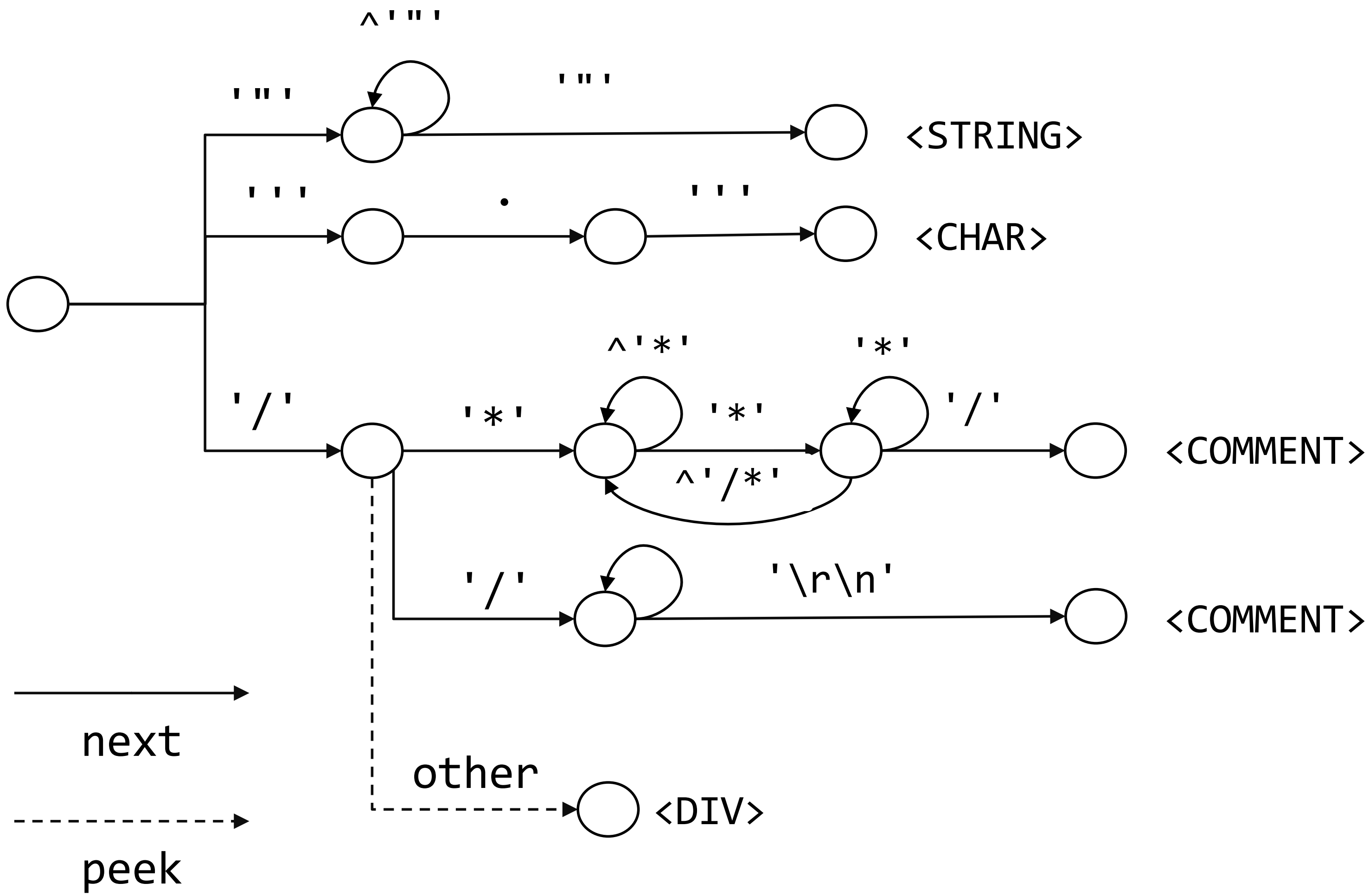
标签识别：前瞻若干个字符



问题：引号应如何处理？

注译和引号

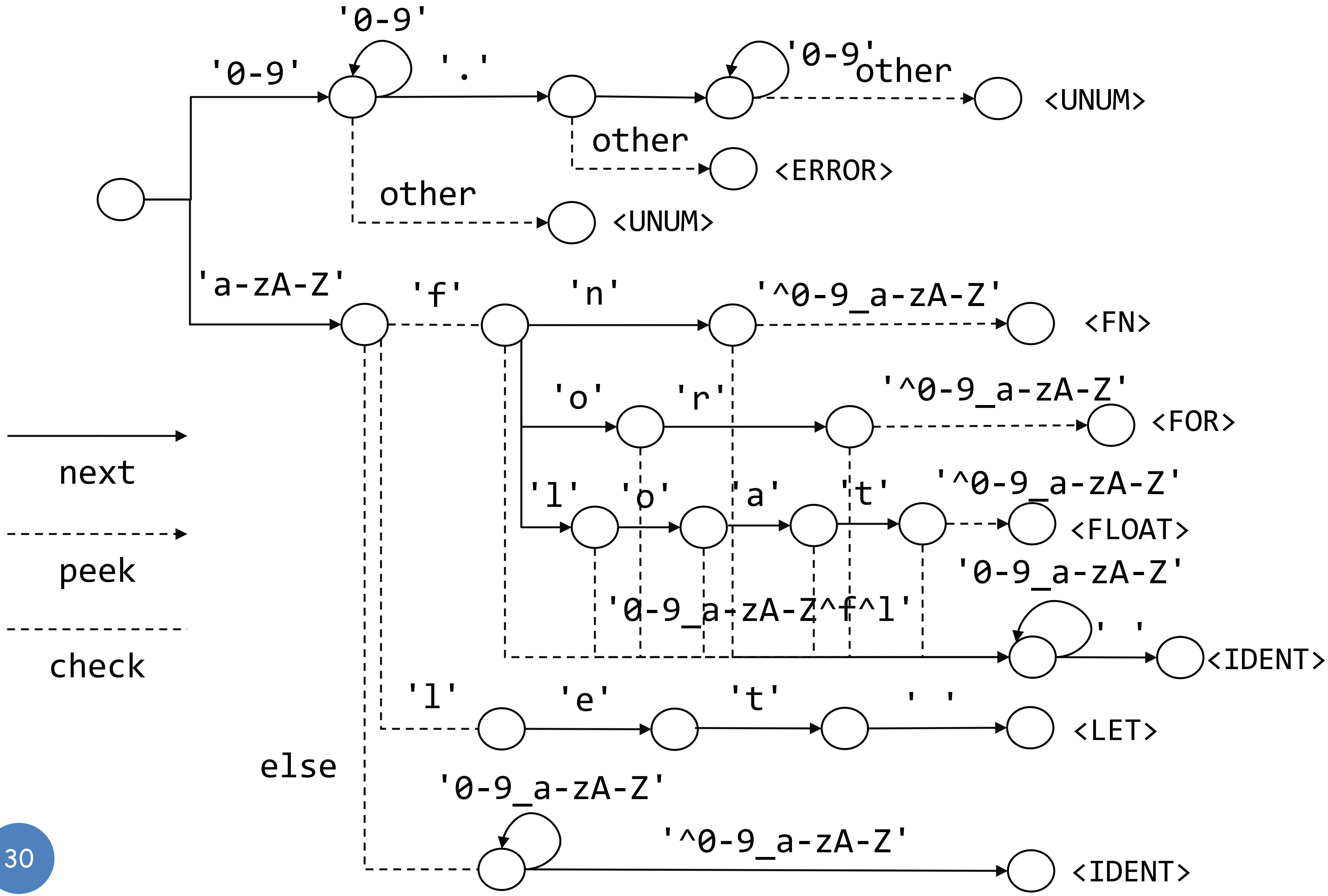
- 引号或注释内部的单词是否应识别为单独的标签？



更新标签定义

```
   <CHAR>  :=  '.'
 <STRING>  :=  "[^"]*"
<COMMENT>  :=  请自己写出正则表达式
```

识别数字和标识符



'0-9'
'0-9'
'.'
'0-9'
other
<UNUM>
other
<ERROR>
other
<UNUM>
'a-zA-Z'
'f'
'n'
'^0-9_a-zA-Z'
<FN>
'o'
'r'
'^0-9_a-zA-Z'
<FOR>
'l'
'o'
'a'
't'
'^0-9_a-zA-Z'
<FLOAT>
'0-9_a-zA-Z'
'0-9_a-zA-Z^f^l'
' '
<IDENT>
'l'
'e'
't'
' '
<LET>
else
'0-9_a-zA-Z'
'^0-9_a-zA-Z'
<IDENT>
next
peek
check

Token类应包含哪些信息

```
struct Token {
    TokType type;
    Position pos;
}
```

父类型或Union

行号、位置

```
enum TokType {
    1; // <ADD>;
    2; // <SUB>;
    ...
    101; <IDENT>
    102; <UNUM>
}
```

```
struct IdentTok : Token {
    char* ident;
}

struct UnumTok : Token {
    char* unum; // float/int?
}
```

如何编写词法分析程序？

```
cur = cstream.next();
match (cur.value) {
    '+' => tstream.add(Token::new(ADD,cur.pos));
    '-' => {
        if cstream.peek(1) == '>' {
            tstream.add(Token::new(RARROW,cur.pos));
            cur = cstream.next ();
        } else tvec.add(Token::new(SUB ,cur.pos));
    'a'-'z' || 'A'-'Z' => {
        pos = cur.pos;
        char value[256];
        if (cur.value == 'f') {
                value[0] = 'f';
                if (cstream.peek(1) == 'n') {
                    if (!isAlphanum(cstream.peek(2))) {
                        tvec.add(Token::new(FN,cur.pos));
                } else if (cstream.peek(1) == 'o') { ...
                } else {
                    int i = 1;
                    ch = cstream.next().value;
                    while (isAlphanum(ch)) {
                        value[i] = ch;
                        ch = cstream.peek(++i).value
                    }
                    tstream.add(IdentTok::new(value,pos));
                }
    ...
```

练习

- 将下列代码转化为标签序列？

```
fn main() {
    let x = factorial(10);
    println("factorial(10) = {}", x);
}
```

```
<FN><IDENT,main><LPAR><RPAR><LBRACE>
<LET><IDENT,x><EQ><IDENT><LPAR><IDENT><RPAR><SEMI>
<IDENT,println><LPAR><STRING,"factorial(10) = {}"><COMMA>
                              <IDENT,X><RPAR><SEMI>
<RBRACE>
```

如何处理字符串打印格式化符号：{}？

更多标签属性

```
println("factorial(10) = {}", x);
```

词素:	(标签, 属性, ..)
println	<IDENT>, 名称, 指向字符表中的printf函数
(	<LPAREN>
"factorial(10) = {}"	<STRING>, 字符串内容
,	<COMMA>
x	<IDENT>, 名称, 指向字符表中的score标识符
)	<RPAREN>
;	<SEMI>

ID	Scope	Type
println	global	function, (const char* int,...) -> int
x	local	int

大纲

- 一、目标语言
- 二、前端功能需求分析
- 三、词法设计和解析
- 四、句法设计和解析

程序组成

$$
\begin{aligned}
\texttt{program} &\longmapsto \texttt{program progComponent} \mid \varepsilon \\
\texttt{progComponent} &\longmapsto \texttt{varDeclStmt} \\
&\quad \mid \texttt{fnDeclStmt} \mid \texttt{fnDecl} \\
&\quad \mid \texttt{macro} \\
&\quad \mid \texttt{comment}
\end{aligned}
$$

变量声明形式

- 类型 + 变量名的形式，如 `let a:int;`
- 在变量声明时赋值，如 `let a:int = 0;`
- 同时声明多个变量，如 `let a:int,b:float,c:char`
- 支持数组类型的变量，如 `let a[n]:int;`

变量声明

```
varDeclStmt  ⟼  <LET> varList var <SEMI> | <LET> varAssign <SEMI>
    varList  ⟼  varList var <COMMA> | ε
        var  ⟼  <IDENT> | <IDENT> <COLON> type
                | <IDENT> <LSQUARE> arraySize <RSQUARE> <COLON> type
                | <IDENT> <LSQUARE> arraySize <RSQUARE>
       type  ⟼  nativeType | <IDENT> | ptrType
    ptrType  ⟼  <STAR> ptrType | <STAR> nativeType | <STAR> <IDENT>
 nativeType  ⟼  <INT> | <LONG> | <FLOAT> | <DOUBLE> | <BOOL> | <CHAR>
  arraySize  ⟼  <UNUM> | <IDENT>
  varAssign  ⟼  <IDENT> <COLON> type <EQ> expr
                | <IDENT> <EQ> expr
```

变量声明

```
  expr   ⟼  expr opL0 exprL1 | exprL1
exprL1   ⟼  exprL1 OpL1 exprL2 | exprL2
exprL2   ⟼  exprL3 OpL2 exprL2 | exprL3
exprL3   ⟼  num | <IDENT> | fnCall | deref | addr | <STRING>
             | <LPAREN> expr <RPAREN>
   num   ⟼  <UNUM> | <SUB> UNUM
 deref   ⟼  <STAR> expr
  addr   ⟼  <BITAND> <IDENT>
  opL0   ⟼  <ADD> | <SUB>
  opL1   ⟼  <STAR> | <SLASH>
  opL2   ⟼  <POW>
```

函数声明形式

- 函数声明，如
  - `fn foo(a:int, b:int)->int;`
  - `fn foo();`
- 函数声明 + 定义，如
  - `fn foo(a:int, b:int)->int {return a + b;}`

函数声明

```
fnDeclStmt  ⟼ <FN> <IDENT> <LPAREN> paramDecl <RPAREN> <SEMI>
              | <FN> <IDENT> <LPAREN> paramDecl <RPAREN> <RARROW> type <SEMI>
    fnDecl  ⟼ | <FN> <IDENT> <LPAREN> paramDecl <RPAREN>
                <LBRACE> stmts <RBRACE>
              | <FN> <IDENT> <LPAREN> paramDecl <RPAREN> <RARROW> type
                <LBRACE> stmts <RBRACE>
 paramDecl  ⟼ paramList param | ε
 paramList  ⟼ paramList param <COMMA > | ε
     param  ⟼ <IDENT> < COLON> type
```

基本代码块

$$
\begin{aligned}
\texttt{stmts} &\longmapsto \texttt{stmts stmt} \mid \varepsilon \\
\texttt{stmt} &\longmapsto \texttt{varDeclStmt} \mid \texttt{assignStmt} \mid \texttt{callStmt} \mid \texttt{ifStmt} \mid \texttt{matchStmt} \\
&\quad \mid \texttt{forStmt} \mid \texttt{whileStmt} \\
\texttt{assignStmt} &\longmapsto \texttt{leftVal <EQ> rightVal <SEMI>} \\
\texttt{leftVal} &\longmapsto \texttt{<IDENT>} \mid \texttt{<IDENT> <LSQUARE> arrayIndex <RSQUARE>} \\
&\quad \mid \texttt{fnCall} \\
\texttt{rightVal} &\longmapsto \mid \texttt{leftVal} \mid \texttt{NUM} \\
\texttt{arrayIndex} &\longmapsto \texttt{NUM} \mid \texttt{<IDENT>}
\end{aligned}
$$

函数调用

- `foo(a, b);`

```
callStmt  ⟼ fnCall <SEMI>
  fnCall  ⟼ <IDENT> <LPAREN> paramVal <RPAREN>
paramVal  ⟼ valList expr | ε
 valList  ⟼ valList expr <COLON> | ε
```

练习：用CFG定义下列控制流语法

```
if(a){
    return 0;
} else {
    return 1;
}
```

```
match(a){
    0 => { return 0; }
    1 => { return 1; }
    _ => { return -1; }
}
```

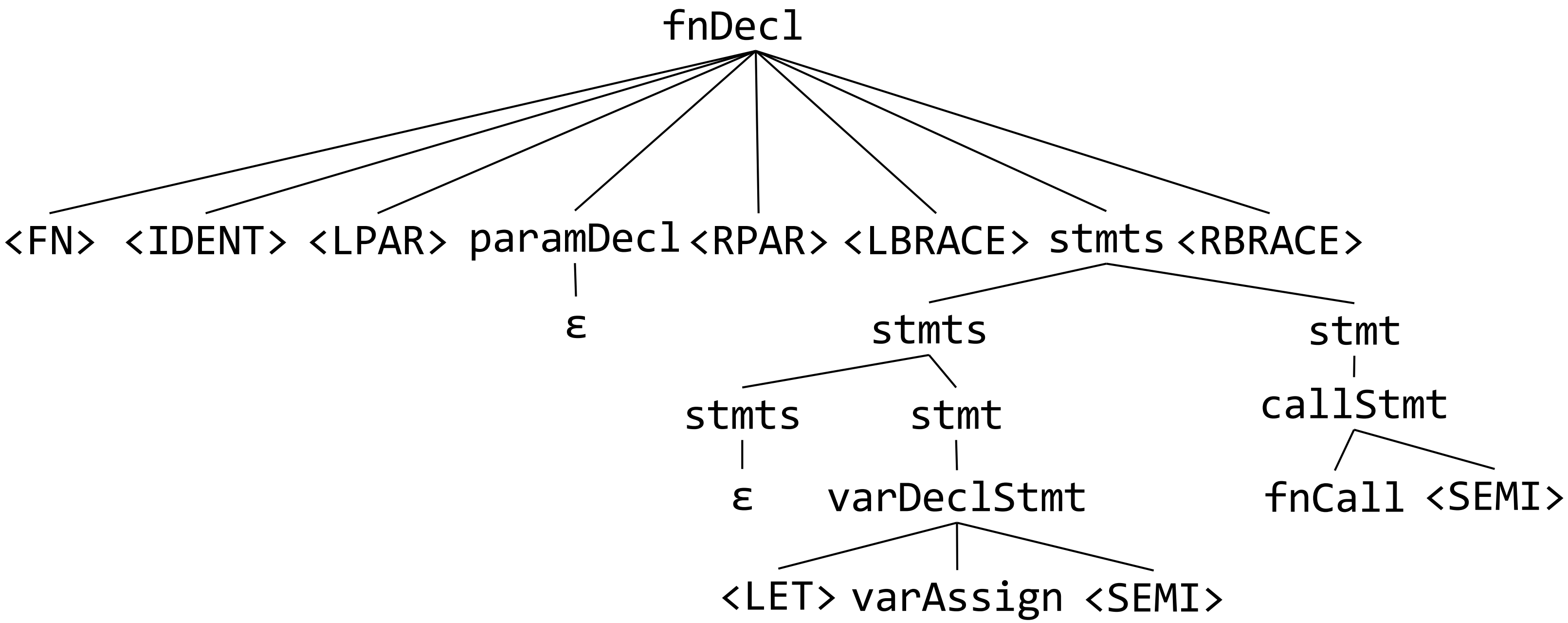
```
for(a in 0..100){
    foo();
}
```

```
while(a){
    foo();
}
```

示例

```
fn main() {
    let x = factorial(10);
    println("factorial(10) = {}", x);
}
```

<FN><IDENT,main><LPAR><RPAR><LBRACE>
<LET><IDENT,x><EQ><IDENT><LPAR><IDENT><RPAR><SEMI>
<IDENT,println><LPAR><STRING,"factorial(10) = {}"><COMMA>
<IDENT,X><RPAR><SEMI>
<RBRACE>



语法解析树节点类型

```
Struct Node {
    NodeType type;
    Vec<*Node> children;
}
```

```
enum NodeType {
    1; // varDecl
    2; // fnDecl
    3; // varList
    4; // var
    5; // type
    ...
    1001; leaf node
}
```

```
struct leafNode : Node {
    struct Token* tok;
}
```

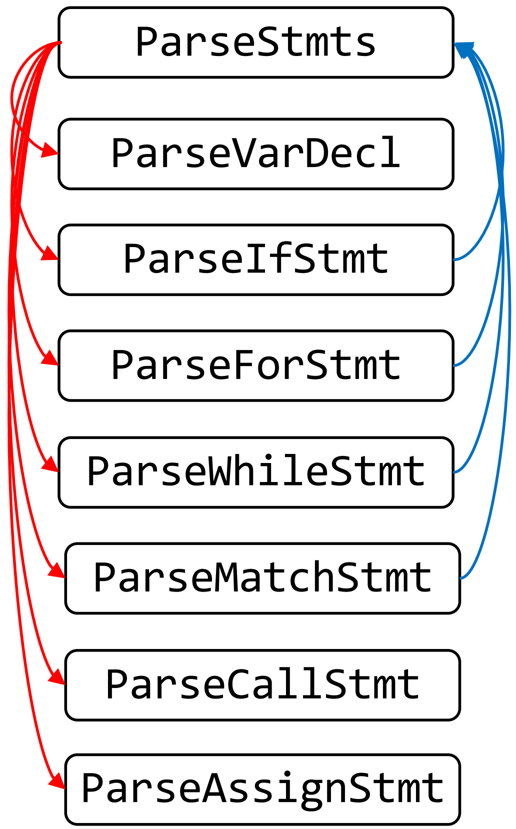
算法实现思路

- 和词法解析类似：前瞻若干个字符
- 递归下降
  - 可以先转换成LL(K)：消除左递归和回溯
  - 表达式解析可使用操作符优先级解析算法

递归下降：以基本代码块解析为例

```
stmts  ⟼ stmts stmt | ε
 stmt  ⟼ varDeclStmt | assignStmt | callStmt | ifStmt | matchStmt
          | forStmt | whileStmt
```

```
ParseStmts(tstream) -> Node {
    cur = tstream.next();
    while (cur != tok::RBRACE) {
        match (cur.type) {
            tok::LET => ParseVarDecl(tstream);
            tok::IF => ParseIfStmt(tstream);
            tok::FOR => ParseForStmt(tstream);
            tok::WHILE => ParseWhileStmt(tstream);
            tok::MATCH => ParseMatchStmt(tstream);
            tok::IDENT => {
                if(tstream.peek() == tok::LPAREN) {
                    ParseCallStmt(tstream);
                }
                else ParseAssignStmt(tstream);
            }
        }
        cur = tstream.next();
    }
}
```



递归下降：构造解析树

```
ParseStmts(tstream) -> Node {
    root = StmtsNode::new();
    root.children = Vec::new();
    cur = tstream.next();
    while (cur != tok::RBRACE) {
        stmtNode = StmtNode::new();
        match (cur.type) {
            tok::LET => {
                varDeclNode = ParseVarDecl(tstream);
                stmtNode.children = Vec::newadd(varDeclNode);
                left = root;
                root.children = Vec::newadd(left,stmtNode);
            }
            tok::IF => ...
        }
        cur = tstream.next();
    }
}
```

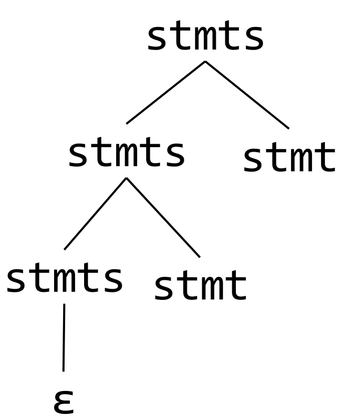
消除左递归

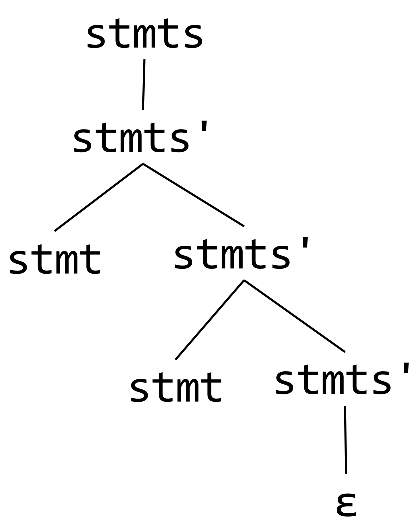
$$\boxed{\begin{aligned} E &\rightarrow E\ \alpha \\ &\mid \beta \end{aligned}} \quad \Rightarrow \quad \boxed{\begin{aligned} E &\rightarrow \beta\ E' \\ E' &\rightarrow \alpha\ E' \\ &\mid \epsilon \end{aligned}}$$

```
stmts   ⟼  stmts stmt | ε
```

$\Rightarrow$

```
 stmts   ⟼  stmts'
stmts'   ⟼  stmt stmts' | ε
```

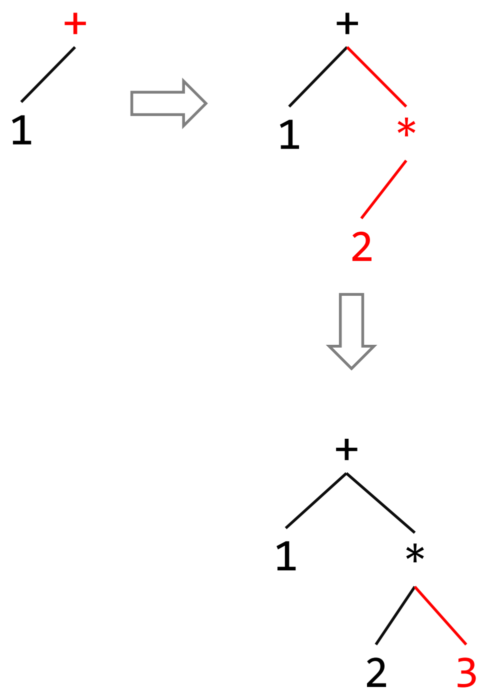




操作符优先级解析

初始化优先级　0 1 2 3 4

算式　1 + 2 * 3




```
Pred[ADD] = 1,2
Pred[SUB] = 1,2
Pred[MUL] = 3,4
Pred[DIV] = 3,4

Parse(token, precedence) {
  left = token.next();
  if left.type != tok::num
    return -1;
  while true:
    op = token.peek();
    if op.tokentype != tok::binop
      return -1;
    lp, rp = Pred[op];
    if lp < precedence
      break;
    token.next();
    right = Parse(token, rp)
    left = (op, left, right)
  return left
}
```

总结

- 前端的基本功能：
  - 代码=>语法解析树
- 前端的基本要求：
  - 语法定义易于解析
  - 解析算法支持错误提示
  - 解析结果便于后续操作
- 语法定义和解析：
  - 词法
  - 句法