左递归只会对**自定向下（如 LL1）**的语法产生二义性

符合文法的 string 可能对应不同的 parsing tree（如四则运算的加、乘顺序）

一个 **unambiguous grammar** 只对应一棵 parse tree

LALR1 文法只有 reduce-reduce conflict（当对映 LR1 文法没有 reduce-shift conflict）

Finding the next handle is the man task of a LR parser（**handler 句柄**，为生成式右侧可归约的串）

Parse tree 不能反映一个 string 的产生顺序（derivation，如等高 parse tree 左右可任意选择生成先后）

Left-recursion 常常暗含了 **left associate（左结合）**

YACC 使用的文法是 LALR1

Left factoring（LL1 文法中提取左公因子的操作）

viable prefix(es)**可行前缀**，句柄中任意不超句柄长度的子串

LL1 table 有 match、generate、accept 的概念

Parser(解析器，进行**文法分析(**通常指语法分析))生成 syntax tree（parser tree 的一种，更加简洁，冗余信息更少），但 parser tree 即可反映 derivation 的顺序

An **LR(1) parser** can detect errors **earlier** than an **LR(0) parser**.

semantic analysis 语义分析，接受输入：抽象 syntax tree 作文法分析时可以先看看能不能**化简**

Follow 集合计算时特别注意**空集**，follow 列填写 parsing table 时注意是写**产生 null 的对应文法**

Tiger 语言

变量定义与 C 一致，换行符输出与 C 一致('\n')

数组记录从 0 开始，循环有 **break** 操作（没 continue）

Var a := 1 、Var a: int := 1(**nil** 只能用此方法，nil 匹配所有类型，但类型需要明确)

Function abc(a:int[**参数**]):int[**返回值**] = 函数体

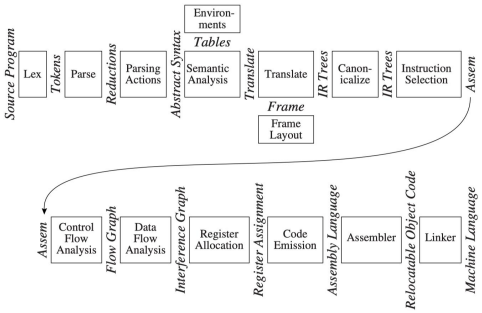
Let 用于定义内、**In** 表示实际的运算内容、**End** 与 let 匹配使用，表示 let 块结束

不等号 **<>** 相等号 **=** 并 **&** 或 **|**

没有连等操作、**While** exp1 do exp2（根据 for 推导）

```
let   type any = {any : int}
      var buffer := getchar()

 function readint(any: any) : int =
  let var i := 0
      function isdigit(s : string) : int =
          ord(buffer)>=ord("0") & ord(buffer)<=ord("9")
    in while buffer=" " | buffer="\n" do buffer := getchar()
       any.any := isdigit(buffer);
       while isdigit(buffer)
        do (i := i*10+ord(buffer)-ord("0");
            buffer := getchar());
       i
  end

 type list = {first: int, rest: list}

 function readlist() : list =
    let var any := any{any=0}
        var i := readint(any)
     in if any.any
         then list{first=i,rest=readlist()}
         else (buffer := getchar(); nil)
    end

 function merge(a: list, b: list) : list =
   if a=nil then b
   else if b=nil then a
   else if a.first < b.first
      then list{first=a.first,rest=merge(a.rest,b)}
      else list{first=b.first,rest=merge(a,b.rest)}

 function printint(i: int) =
  let function f(i:int) = if i>0
          then (f(i/10); print(chr(i-i/10*10+ord("0"))))
    in if i<0 then (print("-"); f(-i))
       else if i>0 then f(i)
       else print("0")
  end

 function printlist(l: list) =
   if l=nil then print("\n")
   else (printint(l.first); print(" "); printlist(l.rest))

  /* BODY OF MAIN PROGRAM */
 in printlist(merge(readlist(), readlist()))
end
let
  var N := 8

  type intArray = array of int

  var row := intArray [ N ] of 0
  var col := intArray [ N ] of 0
  var diag1 := intArray [N+N-1] of 0
  var diag2 := intArray [N+N-1] of 0

  function printboard() =
     (for i := 0 to N-1
       do (for j := 0 to N-1
            do print(if col[i]=j then " O" else " .");
           print("\n"));
        print("\n"))

  function try(c:int) =
   if c=N
   then printboard()
   else for r := 0 to N-1
         do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0
            then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;
                  col[c]:=r;
                  try(c+1);
                  row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

 in try(0)
end
```

标准函数：**print()**输出字符串、**getchar()**获得一个字符，文件尾返回空、**ord(string):int** 返回第一个字符 ASCII 码、**chr(int):string** 根据 ASCII 码转变为字符串（超值报错）、**size(string):int** 获得字符串个数、**substring(string,int,int):string** 获得制定字符串从（第二个参数）开始长（第三个参数）的子串、**concat(string,string):string** 获得串联字符串、**exit(int)**以指定状态码退出、**not(int)**判断是不是 0



Source language and target language

Phases：一个或多个模块、interfaces：模块间接口

Parse 阶段只是分析，parse action 阶段建立抽象 tree

Frame layout 就是栈空间的分配：Canonicalize 简化、清楚条件分支；code emission:替换寄存器为真实名称

两阶段：front end：do analysis、back end：do synthesis

Lex 分析：接受输入流、生成 token 流、解决空格注释

Non_token：注释、空格、**预处理指令**

Lex 对**大小写敏感**，_ 被当作字母

[abcd] means (a | b | c | d),

[b-g] means [bcdefg],

[b-gM-Qkr] means [bcdefgMNOPQkr],

M? means (M | ∈), and M+ means (M·M*).

.	**A period stands for any single character except newline.**
"a.+*"	**Quotation, a string in quotes stands for itself literally.**

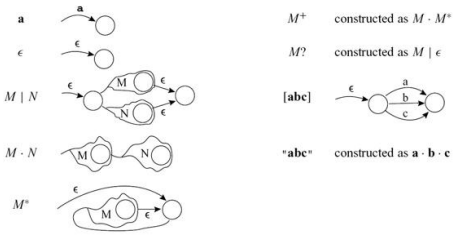
Regular 中“**空**”与**{“”}**等价

("--"[a-z]*"\n")|(" "|"\n"|"\t")+ tiger 空格注释匹配

Longest match（token 最长匹配）、**rule priority**（同长情况下根据 rule 优先级匹配）原则

Lex 进行 token 匹配时，需要 maintain **last 匹配** string **当前位置，开始匹配位置**

Thompson's Construction（regular->NFA）**基本单元：**



Distinguishes（可区分）状态，同一 string 不同结果

状态等价--两个状态发出的每一条 edge 指向相同的 next state

Last Final	Current State	Current Input	Accept Action
0	1	i f --not-a-com	
2	2	i f --not-a-com	
3	3	i f --not-a-com	
3	0	i f --not-a-com	*return* IF
0	1	i f --not-a-com	
12	12	i f --not-a-com	
12	0	i f --not-a-com	*found white space; resume*
0	1	i f --not-a-com	
9	9	i f --not-a-com	
9	10	i f --not-a-com	
9	10	i f --not-a-com	
9	10	i f --not-a-com	
9	10	i f --not-a-com	
9	0	i f --not-a-com	*error, illegal token '-'; resume*
0	1	i f --not-a-com	
9	9	i f --not-a-com	
9	0	i f --not-a-com	*error, illegal token '-'; resume*

计算 state 等价的算法：不停的划分类（只有类中 state edge 转移与其他类中 state 不同时，划分新类）

Flex 结构：

%{ definitions %} C 语言库及辅助函数定义

此处可以添加等价定义如：digit [0-9]

%% { rules } %% 匹配 token 的正则表达式

[yytext 表示返回的 token、yyleng 记录长度]

{ auxiliary routines} C 语言执行函数（可融入生成部分）

Parse 阶段接受 token 流，输出 abstract syntax

Derivation：从 start symbol 开始生成 string

Derivation 的过程可以用 parse tree 表示

Left-most derivation(每次都展开最左侧的非终结符)

A **derivation** defines a parse tree

one **parse tree** may have many derivations

A **grammar is ambiguous（二义性）** if it can derive a string with two different parse trees

四则运算的 left-association 和**优先级**(precedence)

1. E -> E + T	4. T → T * F	7. F -> id
2. E -> E – T	5. T → T / F	8. F -> num
3. E -> T	6. T → F	9. F -> (E)

递归下降（recursive descent parsing）

top-down parsing， can parse LL(1)[Left-to-right parse; Leftmost-derivation; 1 symbol look ahead]

S→(S) S | ε

M[N, T]	(	)	$
S	S -> (S)S	S -> ε	S -> ε

Steps	Parsing Stack	Input	Action
1	$S	()$	S -> (S)S
2	$S)S(	()$	match
3	$S)S	)$	S -> ε
4	$S)	)$	match
5	$S	$	S -> ε
6	$	$	accept

Parsing Stack 列需要从右向左看

FIRST(γ) is the set of terminals that can begin strings derived from γ； **OLLOW(X)** is the set of terminals that can immediately follow X.

```
for each production X → Y1Y2 ··· Yk
  for each i from 1 to k, each j from i + 1 to k,
    if all the Yi are nullable                    Iteration
      then nullable[X] ← true
    if Y1 ··· Yi−1 are all nullable
      then FIRST[X] ← FIRST[X] ∪ FIRST[Yi]
    if Yi+1 ··· Yk are all nullable
      then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FOLLOW[X]
    if Yi+1 ··· Yj−1 are all nullable
      then FOLLOW[Yi] ← FOLLOW[Yi] ∪ FIRST[Yj]
```

LL（1）Parse table 法则（注意 Z 那行，**完整性**）

Grammar:

Z → X Y Z　Y → c　X → a
Z → d　Y →　X → Y

	nullable	first	follow
Z	no	d,a,c	
Y	yes	c	a,c,d
X	yes	a,c	a,c,d

	a	c	d
Z	Z->XYZ	Z->XYZ	Z->XYZ Z->d
Y	Y->	Y->c Y->	Y->
X	X->a X->Y	X->Y	X->Y

- if T ∈ First(γ) then enter (X → γ) in row X, col T
- if γ is Nullable and T ∈ Follow(X) enter (X → γ) in row X, col T

Z->XYZ 也能产生 first(d)

certain to cause duplicate entries in the LL(1) parsing table

- E -> E+T: First(E+T) ⊆ First(E)
- E+T: First(E) ⊆ First(E+T)
- => **First(E) = First(E+T)**
- E -> T: **First(T) ⊆ First(E)**
- => if a ∈ First(T), then a ∈ First(E) and a ∈ First(E+T)
- For any such a, we can choose any of the two productions

A ->Aα
A -> β

A ->βA'
A' -> αA'
A' ->

LL(1) 需要消除 left recursion

左递归修改策略

Left factoring 提取左公因子也可以

Recover from error：inserting（可能无法终止）、deleting（skip token 直到合法，一定会停止--EOF）

LR(k) Left-to-right parse、Rightmost derivation

bottom-up parsing：reduce the string to the start symbol

有 shift、reduce、goto、accept、error 五种表现

State stack 同 symbol stack 一样根据 action 被操作

RHS：right hand side（LHS 同理），表示产生式右侧

LR(0) parsing；state：S'-> . S$（再打开 S，不停迭代）

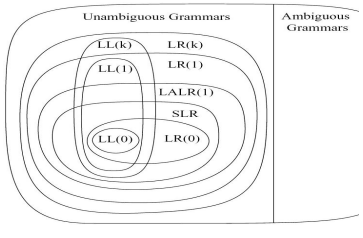
SLR parsing（simple LR parsing）：Put reduce actions into the table only indicated by the FOLLOW set（非全部符号）

LR(1) parsing: An item (A → α.β, x) indicates that the sequence α is on top of the (symbol) stack, and at the head of the input is a string derivable from βx.

LALR(1) parsing: the parsing table is made by merging any two states whose items are identical except for look ahead sets in the LR(1) parsing table.

Stack (states)	Stack (symbols)	Input	Action
1		(x) $	shift 3
1, 3	(	x) $	shift 2
1, 3, 2	(x	) $	reduce 2 S -> x
1, 3	(S	) $	goto 7
1, 3, 7	(S	) $	reduce 3 L -> S
1, 3	(L	) $	goto 5
1, 3, 5	(L	) $	shift 6
1, 3, 5, 6	(L)	$	reduce 1 S -> (L)
1	S	$	goto 4
1, 4	S	$	accept

LALR 可能存在 reduce-reduce conflict



YACC 结构

%{definitions%} 同 flex

%union { double val; char op;} token 可存储类型（或）

%token <op> 定义可能的 token（terminal，non-terminal 自动被声明为 token，<>对映于 union 中类型）

%% {rules} %% 语法操作（可用 {} 来声明 action code）

{auxiliary routines} 其余辅助函数 error、main 函数等

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex(void);
int yyerror (char * s);
%}
%token NUMBER
%%
command: exp {printf("%d\n", $1);};
exp: exp '+' term {$$ = $1 + $3;}
   | exp '-' term {$$ = $1 - $3;}
   | term {$$ = $1}
;
term: term '*' factor {$$ = $1 * $3;}
    | factor {$$ = $1;}
;
factor: NUMBER {$$ = $1;}
      | '(' exp ')' {$$ = $2;}
;
```

Example: 3 * 4

Action	Symbol Stack	Value Stack
Shift	Num	3 (from yylval)
reduce	factor	3
reduce	term	3
shift	term, '*'	3, 0 (default)
shift	term, '*', Num	3, 0, 4
reduce	term, '*', factor	3, 0, 4
reduce	term	12
reduce	exp	12

yylex 会返回 token 或 0，**yylval** 会返回 token 语义值

Yacc resolves **shift reduce conflicts** by shifting 、 **reduce reduce conflicts** by use rule appears earlier in grammar

%nonassoc EQ NEQ（没有结合性，不能连续运算）

%left PLUS MINUS（left 定义左结合）

%left TIMES DIV

%right EXP（从上至下优先级不断提升）

%prec abc（定义 label，在特定位置使用改变优先级）

同优先级左结合先 reduce 右结合先 shift

Local error recovery: 增加一个转换到 error 的语法，在发生错误时，清空 synchronizing（同步） token 之间的 stack 中所有内容，并写入 error 符号

Global error repair: 找到将源字符串转换为语法正确的字符串的最小插入和删除集，即使插入和删除不在 LL 或 LR 解析器首先报告错误的位置。

Burke-Fisher Error Repair：在解析器报告错误点前不早于 K token 的每个点尝试插入、删除或替换每个 token。一般来说，如果修复将解析器带到它最初卡住的地方之后 4 个标记，即表示修复已足够好。

要求：Maintain two parse stacks: the current stack and the old stack，Keep a queue of K tokens(两个 stack 之间)

```
int T( void) {switch (tok) {
  case ID:
  case NUM:
  case LPAREN: return Tprime(F());
  default: print("expected ID, NUM, or left-paren");
           skipto(T_follow);
           return 0;
}}

int Tprime(int a) {switch (tok) {
  case TIMES: eat(TIMES); return Tprime(a*F());
  case PLUS:
  case RPAREN:
  case EOF: return a;
  default: ...
}}
```

```
struct Ty_ty_ {
  enum {Ty_record, Ty_nil, Ty_int, Ty_string,
        Ty_array, Ty_name, Ty_void} kind;
  union {
    Ty_fieldList record;
    Ty_ty array;
    struct {S_symbol sym; Ty_ty ty;} name;
  } u;
```

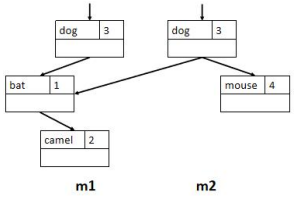
连乘的语义分析：

Concrete parse tree 记录了具体的 grammar rule reduce

Abstract syntax tree 即具体语法树的化简（yacc 生成）

Symbol table： mapping identifiers to their **types** and **locations**（**scope**）

Functional style 函数式：旧环境不发生变化（仍可被访问），常使用二叉树结构（如果我们在树深度 d 处添加新节点，只需创建 d 个新节点即可--对应 root 到新节点的父节点）



Imperative style 命令式：旧环境发生变化以转变为新环境（多为散列表结构）

Tiger 使用命令式 symbol table，维护 type table 和 value table（变量和函数（分别记录出参数与返回值类型））

destructive-update 环境， 所以使用 begin_scope 和 end_scope 作为划分符，配合额外 stack mark 变量

[] 表示方括号内包含的项可被重复 0 次或 1 次

{} 表示花括号内包含的项可被重复 0 次或多次

Tiger 中不同**类型定义**占用空间不同，即使结构一致也不能相互赋值

初始化 Ty_Nil 类型表达式必须受到 Ty_Record 类型约束

处理函数类型时，第一遍仅处理（记录）函数头，第二遍再详细处理函数体

一组相互递归的类型声明中，每个 cycle 都必须依赖一个记录或数组声明。