Introduction

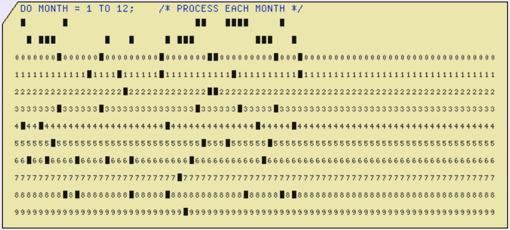
WANG Hanfei

February 13, 2022

1. 程序设计语言的历史
2. 程序设计语言的组成
   - 字符
   - 单词
   - 语句
   - 语义
3. 编译器的结构
   - 编译器的定义
   - 编译器的结构
4. XL 语言编译器
   - XL 语言的形式规则
   - 词法分析器的设计
   - 递归下降语法分析器的设计
   - 语义分析及代码生成

Prehistory(40 年代) — 机器语言

(a) Grace Hopper



(b) Punch card

Intel 机器码写的阶乘计算程序

```
10111000 00000001 00000000 00000000 00000000
10111010 00000010 00000000 00000000 00000000
00111001 11011010
01111111 00000110
00001111 10101111 11000010
01000010
11101011 11110110
11000011
```

远古 (1950) — 汇编语言及汇编器

用文本表示机器语言

1. 机器指令用助记符表示.
2. 内存地址和指令地址用标识符表示.
3. 允许有注释.
4. 汇编器完成汇编语言到机器语言的翻译.

Intel~汇编语言的阶乘计算程序

```
;; 输入参数 N 放入寄存器EBX中,计算结果放入寄存器EAX中
Factorial:
        mov eax, 1      ;; 初始化输出result = 1
        mov edx, 2      ;; 初始化循环参数index = 2
L1:     cmp edx, ebx    ;; 如果 index <= N ...
        jg L2
        imul eax, edx   ;; result乘上index
        inc edx         ;; index递增1
        jmp L1          ;; 转移到循环始点
L2:     ret             ;; 返回
```

复兴 (1957) — 算术表达式的翻译

FORTRAN — FORmula TRANslator

- 与机器无关.
- 数学表达式，不需要计算机专业知识即可阅读和书写.
- 编译器完成数学表达式到汇编语言到翻译.

二次方程的求解

```
In FORTRAN :
D = SQRT(B*B - 4*A*C)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

```
;In assembly language :
mul t1, b, b      sub x1, d, b
mul t2, a, c      div x1, x1, t3
mul t2, t2, 4     neg x2, b
sub t1, t1, t2    sub x2, x2, d
sqrt d, t1        div x2, x2, t3
mul t3, a, 2
```

曙光 (60 年代) — 递归和循环

ALGOL (ALGOrithmic Language) — 算法语言的诞生

- Backus-Naur 范式对语言形式描述.
- 递归调用，控制结构和调用方式 (传值与传名).
- 现代程序设计语言的雏形.

阶乘函数 --- 用C语言书写

Iteration (loops):

```
r = 1;
for (i = 2; i <= n; i++)
  r = r * i;
```

Recursion:

```
int  fact (int n)
{ if (n <= 1)
     return 1;
  else return fact(n - 1) * n;
}
```

现代 (80 年代) — 数据结构的自动表示

ML — 函数式程序设计语言

- 抽象数据类型.
- 数据在内存的表示由编译器控制.
- 编译器管理内存的分配.
- 说明式语言.

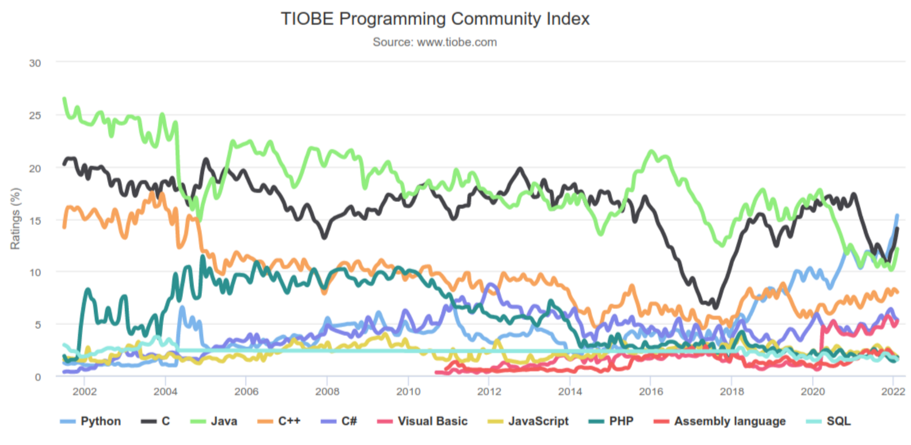
阶乘函数 --- 用ML语言书写

```
let rec fact n =
  if n <= 1
  then 1
  else n * fact (n - 1)
```

程序语言的分类

- 汇编语言
- general purpose： Fortran, Algol, Pascal, Basic, Simula, Ada, Java, Scala (Twitter: Ruby → Scala) ···
- system programming languages: C.
- 脚本语言： Shell, Awk, Perl, Tcl, Python, Javascript, PHP, ···
- 文字处理语言： TEX, LATEX, XƎLATEX(本幻灯片用此编写).
- 超文本语言：HTML, XML;
- 数据库语言： SQL, LINQ (Language Integrated Query).
- 数据表处理语言： VisiCalc, Lotus1-2-3, Excel,···
- 页面描述语言： PostScript, PDF (Portable Document Format).
- more http://en.wikipedia.org/wiki/Programming_language.

程序设计语言的使用统计



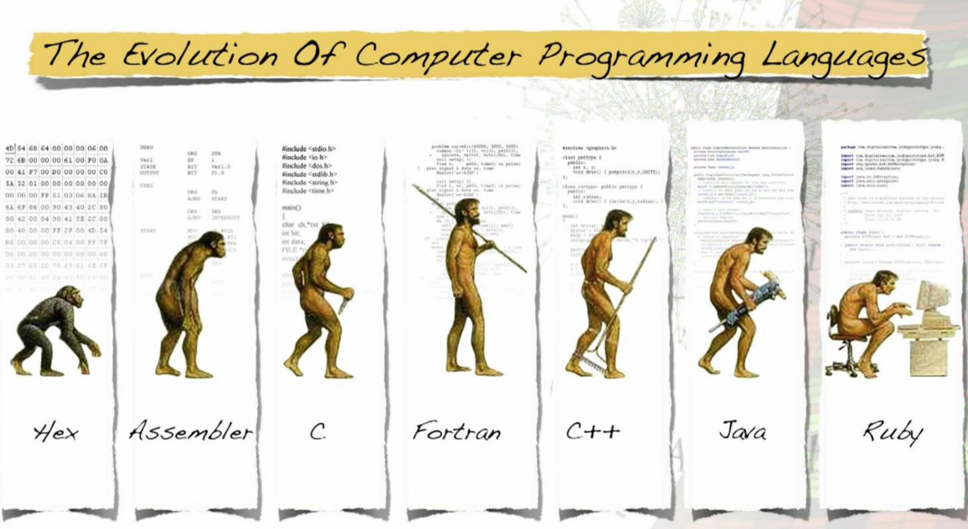
程序设计语言的范型 (programming paradigm)

Programming paradigm is a fundamental style of computer programming (http://en.wikipedia.org/wiki/Programming_paradigm).

- 过程式语言 (Procedural languages): FORTRAN, C, Pascal...
- 面向对象语言 (Object-oriented languages): Simula, Smalltalk, Eiffel, Java, Ruby (Seeing Everything as an Object).
- 说明式语言 (Declarative programming): 与上述命令式 (Imperative language) 不同，没有控制结构，甚至没有赋值，仅有问题说明，或者说纯数学定义：
  - 函数式语言：Lisp, OCaml, Haskell, Scala...
  - 逻辑程序设计语言：Prolog...

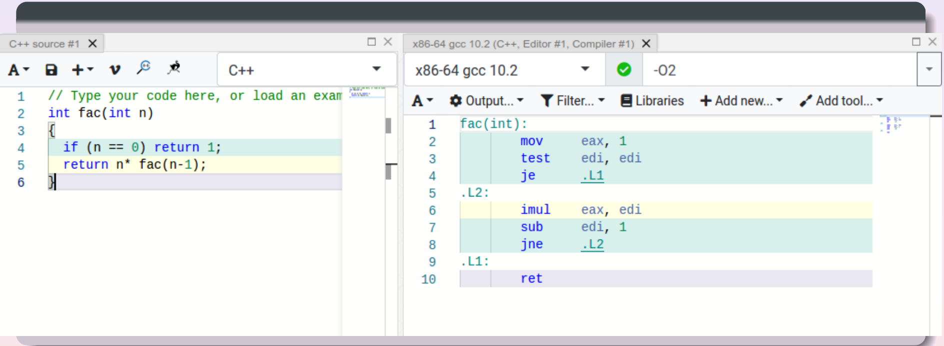
小知识

- Hello World Collection (http://helloworldcollection.de/) 收集了 428 个不同语言写的 “Hello World” 程序.
- More than 2500 PL now. every 2 weeks, a new PL borns since 1954 (see http://cdn.oreillystatic.com/news/graphics/prog_lang_poster.pdf).


The Evolution Of Computer Programming Languages
Hex
Assembler
C
Fortran
C++
Java
Ruby

More Humors

Compiler Explorer: https://godbolt.org



1. 程序设计语言的历史
2. 程序设计语言的组成
   - 字符
   - 单词
   - 语句
   - 语义
3. 编译器的结构
   - 编译器的定义
   - 编译器的结构
4. XL 语言编译器
   - XL 语言的形式规则
   - 词法分析器的设计
   - 递归下降语法分析器的设计
   - 语义分析及代码生成

字符 (characters)

字符

- 组成程序语言的最小单位.
- 字符编码.

Example

- ASCII, EBCDIC: `'Z' - 'A' + 1`.
- UNICODE(UCS), UTF-7, UTF-8, ISO-Latin, GB2312, GBK. `iconv`, `enca`, `enconv` 查看或更改文件编码.
- MIME(Multipurpose Internet Mail Extensions): `mpack`, `munpack`.
- Linux 程序运行的语言环境的设置: `locale`.
- MySQL: Charset and Collation (字符集与校对).

单词 (tokens)

单词 (tokens)

- 一组连续的字符，程序语言处理的最小单位.
- 程序语言规定了单词构成的规则和单词类别.

Example of C

- operators, separators: ! % + - ++ -- >> == ;
- identifiers, keywords.
- constants: 'A', "ABC", 0XAA, 0XAAL.

Example of C

```
int test
 (
  int * x
 )
 { return  * x +++= 1 ; }
```

语句 (sentences)

语句

- 对单词的再次重组.
- 程序设计语言规定了语句的重组规则和语句的类别.

Example of C expressions 和 statements 的递归定义

- 变量名和常量是表达式 (归纳基础).
- if `expr1` and `expr2` are expressions, then:
  `expr1 '+' expr2`, `expr1 '=' expr2`, ... are expressions. (归纳条款)
- `';'` is a statement.
- if `expr` is an expression, then `expr ';'` is a stmt.
- if `stmt` is a statement and `expr` is an expr, then:
  `'if' '(' expr ')' stmt` is a stmt.

Example of C statements

- `i = 0; while ( i < 10 ); {s = s + i ; i++;}`
- `printf ("%c\n", 3["ABCD"]);`

语义 (semantics)

语义

- the meaning of the language.

Example of C expressions

- type, value, l-value and side-effect.
- `expr1 '+' expr2` 要求两表达式的类型是可求和类型, 并且一致 (可能会有 implicite conversion 发生).
- `expr1 '=' expr2` 要求表达式 1 有左值.

Example

```
char a = 129, b = 2;
unsigned int c;
c = a + b;
```

Which result of c?

131, 3, 65411?

类型与语义

How the implicit conversion works

int → char → int → operation → unsigned int

129 → -127 → -127 → -127 + 2 → $2^{16} - 125$

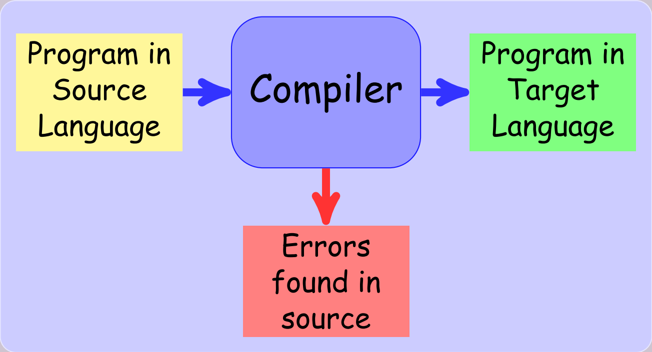
注意:

char 参于运算时将转换为 int!

How to get 131 as correct result

```
c = (unsigned char) a + (unsigned char) b;
```

编译器的定义


Program in Source Language
Compiler
Program in Target Language
Errors found in source

编译器 (compiler)

广义定义

编译器是一个将 Lang1 的任意的一条源语言语句翻译为 Lang2 对应的目标语言语句的一段程序:

$$\text{Lang1} \longrightarrow \text{Lang2}$$
$$l_1 \longmapsto l_2$$

狭义定义

Lang1 是程序设计语言, Lang2 是计算机可以 “理解” 并且可以 “执行” 的机器语言.

本课程将主要关注狭义的编译器

注释

- 如果程序看成是数学函数，则编译器就是函数的函数 — 泛函.
- 能够准确地判断给定的 l_1 是否属于Lang1，如果不属于，编译能够正确地指出错误所在.
- "翻译" 意味着 l_2 和 l_1 要保持相同的含义.
- 编译器的质量：时间、空间、返回信息、调试支持和目标代码质量.

Example in OCaml

```
# let rec len l = match l with
    [] -> 0
  | a::l1 ->  1 + (len l1);;
val len : 'a list -> int = <fun>
# len [1; 2; 3];;
- : int = 3
# let rec sum l = match l with
    [] -> 0
  | a::l1 ->  a + (sum l1);;
val sum : int list -> int = <fun>
# sum [1; 2; 3];;
- : int = 6
# let rec rev l = match l with
    [] -> []
  | a::l1 -> (rev l1) @ [a];;
val rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Evaluation Processus of len

```
len [1; 2; 3]
```

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

Evaluation Processus of len

```
  len [1; 2; 3]
= 1 + (len [2; 3])
```

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

Evaluation Processus of len

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
```

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
= f(1, len [2; 3])
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
= f(1, len [2; 3])
= f(1, f(2, len [3]))
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
= f(1, len [2; 3])
= f(1, f(2, len [3]))
= f(1, f(2, f(3, len [])))
```

Evaluation Processus of len

```
let rec len l = match l with
| [] -> 0
| a::l1 -> 1 + len l1;;
```

```
  len [1; 2; 3]
= 1 + (len [2; 3])
= 1 + (1 + (len [3]))
= 1 + (1 + (1 + (len [])))
= 1 + (1 + (1 + (  0   )))
```

Abstraction `1 + (len l)` with function `f(a, len l)`, we have

```
  len [1; 2; 3]
= f(1, len [2; 3])
= f(1, f(2, len [3]))
= f(1, f(2, f(3, len [])))
= f(1, f(2, f(3,   0   )))
```

Evaluation Processus of sum

sum [1; 2; 3]

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

Evaluation Processus of sum

```
sum [1; 2; 3]
= 1 + (sum [2; 3])
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

Evaluation Processus of sum

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
= f(1, sum [2; 3])
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
= f(1, sum [2; 3])
= f(1, f(2, sum [3]))
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
= f(1, sum [2; 3])
= f(1, f(2, sum [3]))
= f(1, f(2, f(3, sum [])))
```

Evaluation Processus of sum

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> 1 + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

Abstraction `a + (sum l)` with function `f(a, sum l)`, we have

```
  sum [1; 2; 3]
= f(1, sum [2; 3])
= f(1, f(2, sum [3]))
= f(1, f(2, f(3, sum [])))
= f(1, f(2, f(3,   0   )))
```

Evaluation Processus of rev

```
rev [1; 2; 3]
```

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

Evaluation Processus of rev

```
rev [1; 2; 3]
= (rev [2; 3]) @ [1]
```

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> l + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> l + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []  ) @ [3])@ [2]) @ [1]
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
    rev [1; 2; 3]
  = (rev [2; 3]) @ [1]
  = ((rev [3]) @ [2]) @ [1]
  = (((rev []) @ [3])@ [2]) @ [1]
  = (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
     rev [1; 2; 3]
   = (rev [2; 3]) @ [1]
   = ((rev [3]) @ [2]) @ [1]
   = (((rev []) @ [3])@ [2]) @ [1]
   = (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
rev [1; 2; 3]
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
   rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
  rev [1; 2; 3]
= f(1, rev [2; 3])
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
  rev [1; 2; 3]
= f(1, rev [2; 3])
= f(1, f(2, rev [3]))
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> 1 + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []   ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
  rev [1; 2; 3]
= f(1, rev [2; 3])
= f(1, f(2, rev [3]))
= f(1, f(2, f(3, rev [])))
```

Evaluation Processus of rev

```
let rec rev l = match l with
| [] -> []
| a::l1 -> l + rev l1;;
```

```
  rev [1; 2; 3]
= (rev [2; 3]) @ [1]
= ((rev [3]) @ [2]) @ [1]
= (((rev []) @ [3])@ [2]) @ [1]
= (((  []  ) @ [3])@ [2]) @ [1]
```

Abstraction `(rev l) @ a` with function `f(a, rev l)`, we have

```
  rev [1; 2; 3]
= f(1, rev [2; 3])
= f(1, f(2, rev [3]))
= f(1, f(2, f(3, rev [])))
= f(1, f(2, f(3,   []   )))
```

Function as data

- The above 3 functions have the same behaviors: applying consecutively the every list element from right to left to a function f:

$$f(a_1, f(a_2, f(a_3, f(\cdots f(a_n, b)\cdots)))).$$

where $(a_1, a_2, \ldots a_n)$ is the list, and $f : X \times Y \to Y$ is the abstract function which operates on a list element & the result of the application f to the rest of the list. The intial element b will correspond the result of the empty list.
- for `len`, f can be taked $(x, y) \mapsto 1 + y$, $b = 0$.
- for `sum`, f can be taked $(x, y) \mapsto x + y$, $b = 0$.
- for `rev`, f can be taked $(x, y) \mapsto y@[x]$, $b = []$.

Evaluation Processus of sum

define new function `fold_right`, take `sum` as an argument

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

Evaluation Processus of sum

define new function `fold_right`, take `sum` as an argument

```
sum [1; 2; 3]
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
fold_right f [1; 2; 3] b
```

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
```

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
= f(1, f(2, fold_right f [3] b))
```

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
```

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
= f(1, f(2, fold_right f [3] b))
= f(1, f(2, f(3, fold_right f [] b)))
```

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
= f(1, f(2, fold_right f [3] b))
= f(1, f(2, f(3, fold_right f [] b)))
= f(1, f(2, f(3,         b        )))
```

Evaluation Processus of sum

define new function fold_right, take sum as an argument

```
let rec sum l = match l with
| [] -> 0
| a::l1 -> a + sum l1;;
```

```
  sum [1; 2; 3]
= 1 + (sum [2; 3])
= 1 + (2 + (sum [3]))
= 1 + (2 + (3 + (sum [])))
= 1 + (2 + (3 + (  0   )))
```

```
let rec fold_right f l b = match l
with
| [] -> b
| a::l1 ->f a (fold_right f l1 b);;
```

```
  fold_right f [1; 2; 3] b
= f(1, fold_right f [2; 3] b)
= f(1, f(2, fold_right f [3] b))
= f(1, f(2, f(3, fold_right f [] b)))
= f(1, f(2, f(3,          b        )))
```

b is the initial element.

Example in OCaml

```
# let rec fold_right f l b = match l with
    [] -> b
  | a::l1 -> f a (fold_right f l1 b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
# let len l = fold_right (fun x y -> 1 + y)  l 0;;
val len : 'a list -> int = <fun>
# len [1; 2; 3];;
- : int = 3
# let sum l = fold_right (+) l 0;;
val sum : int list -> int = <fun>
# sum [1; 2; 3];;
- : int = 6
# let rev l = fold_right (fun a l1 -> l1 @ [a]) l [];;
val rev : 'a list -> 'a list = <fun>
# rev [1; 2; 3];;
- : int list = [3; 2; 1]
```

Change to tail recursion

- `fold_right` is not tail recursive, so the execution is not efficient.
- Because compiler can transform the tail recursion to while-loop, the more efficient way is define the function as tail recursion.
- The tips is change the recursion result to recursion argument, so called "accumulator":

```
let rec sum l = match l with
    [] -> 0
| a::l1 -> a + (sum l1);;
```

could transform to:

```
let rec sum a l = match l with
    [] -> a
| b::l1 -> sum (a + b) l1;;
```

- The same way, define `fold_left` as

$$f(f(\cdots f(f(f(a,b_1),b_2),b_3),\ldots,b_{n-1}),b_n).$$

where $(b_1,b_2,\ldots b_n)$ is the list, and $f : X \times Y \rightarrow X$ is the abstract function which operates on an intial element a and list element, produces the element of same type of the initial element.

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

```
sum  0 [1; 2; 3]
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
fold_left f a [1; 2; 3]
```

Evaluation Processus of sum

define new function fold_left, take sum as an argument f

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left f (f(a, 1)) [2; 3]
```

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
= sum (0 + 1 + 2) [3]
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left f (f(a, 1)) [2; 3]
= fold_left f (f(f(a, 1), 2)) [3]
```

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
= sum (0 + 1 + 2) [3]
= sum (0 + 1 + 2 + 3) []
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left f (f(a, 1)) [2; 3]
= fold_left f (f(f(a, 1), 2)) [3]
= fold_left f (f(f(f(a, 1), 2), 3)) []
```

Evaluation Processus of sum

define new function `fold_left`, take `sum` as an argument `f`

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
= sum (0 + 1 + 2) [3]
= sum (0 + 1 + 2 + 3) []
=      0 + 1 + 2 + 3
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left f (f(a, 1)) [2; 3]
= fold_left f (f(f(a, 1), 2)) [3]
= fold_left f (f(f(f(a, 1), 2), 3)) []
=             (f(f(f(a, 1), 2), 3))
```

Evaluation Processus of sum

define new function fold_left, take sum as an argument f

```
  sum  0 [1; 2; 3]
= sum (0 + 1) [2; 3]
= sum (0 + 1 + 2) [3]
= sum (0 + 1 + 2 + 3) []
=      0 + 1 + 2 + 3
```

```
let rec sum a l = match l with
| [] -> a
| b::l1 -> sum (a + b) l1;;
```

```
let rec fold_left f a l = match l with
| [] -> a
| b::l1 -> fold_left (f a b) l1;;
```

```
  fold_left f a [1; 2; 3]
= fold_left f (f(a, 1)) [2; 3]
= fold_left f (f(f(a, 1), 2)) [3]
= fold_left f (f(f(f(a, 1), 2), 3)) []
=             (f(f(f(a, 1), 2), 3))
```

a is the initial element.

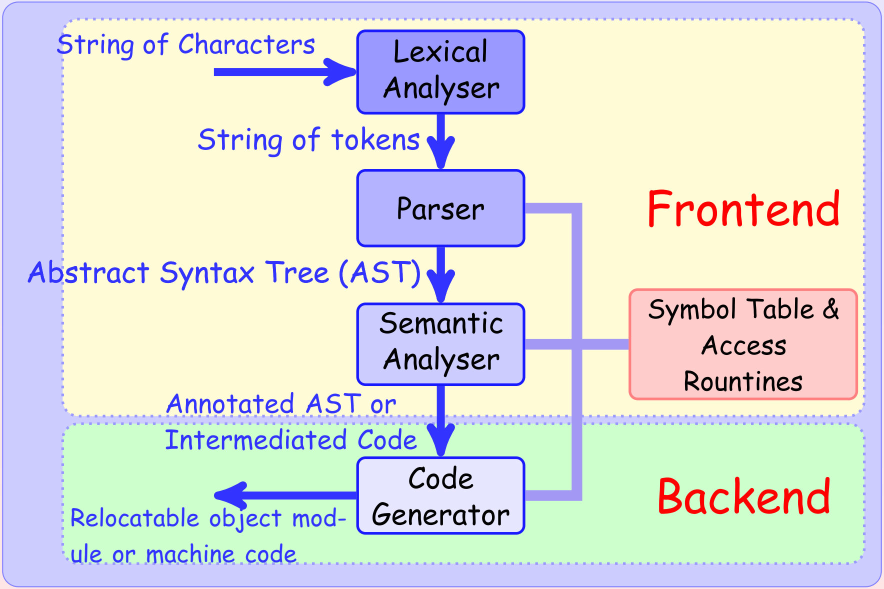
Example in OCaml

```
# let rec fold_left f a l = match l with
    [] -> a
  | b::l1 -> fold_left f (f a b) l1;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# let len l = fold_left (fun x y -> x + 1) 0 l;;
val len : 'a list -> int = <fun>
# len [1;2;3];;
- : int = 3
# let sum l = fold_left (+) 0 l;;
val sum : int list -> int = <fun>
# sum [1;2;3];;
- : int = 6
# let rev l = fold_left (fun l1 a -> a::l1) [] l;;
val rev : 'a list -> 'a list = <fun>
# reve [1;2;3];;
- : int list = [3; 2; 1]
```

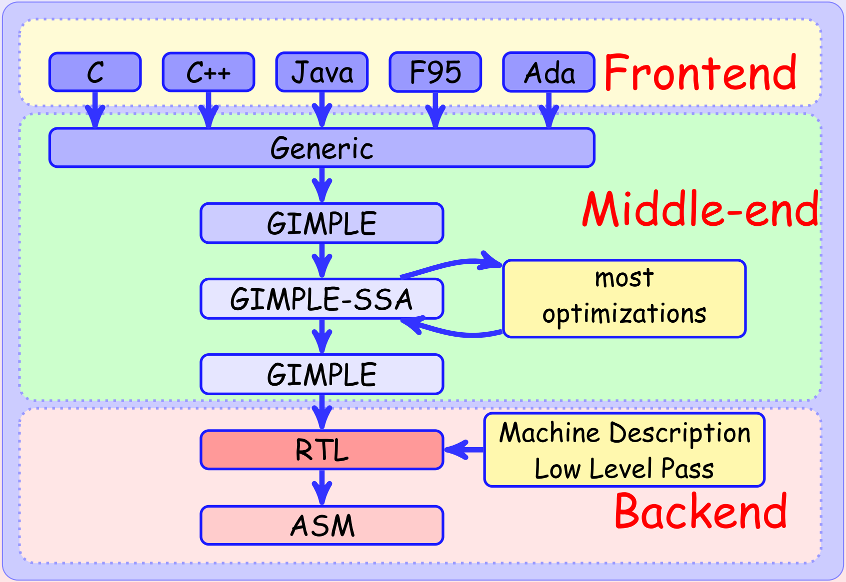
fold_left is an iterator

```
# let rec aux l a = match l with
| [] -> [a]
| b :: l1 -> if a <= b then a::l else b::(aux l1 a);;
val insert_sort : 'a list -> 'a list = <fun>
# let insert_sort l = fold_left aux [] l;;
val insert_sort : 'a list -> 'a list = <fun>
# insert_sort [3; 1; 6; 2; 4; 5];;
- : int list = [1; 2; 3; 4; 5; 6]
# insert_sort [3; 1; 6; 2; 4; 5; 1; 2]
- : int list = [1; 1; 2; 2; 3; 4; 5; 6]
```

编译器的结构


String of Characters
Lexical Analyser
String of tokens
Parser
Frontend
Abstract Syntax Tree (AST)
Semantic Analyser
Symbol Table & Access Rountines
Annotated AST or Intermediated Code
Code Generator
Backend
Relocatable object mod-ule or machine code

Example: GCC 4.xx 版代码编译流程和优化框架



形式系统 (Formal Systems) 的两次合成

- 程序语言语句的合成：一维的字符流 $\longrightarrow$ 二维的树结构.
- 程序语言语义的合成：二维的树结构 $\longrightarrow$ 一维的目标码.

形式系统 (Formal Systems) 的两次合成

- 程序语言语句的合成：一维的字符流 $\longrightarrow$ 二维的树结构.
- 程序语言语义的合成：二维的树结构 $\longrightarrow$ 一维的目标码.

1+a*3

形式系统 (Formal Systems) 的两次合成

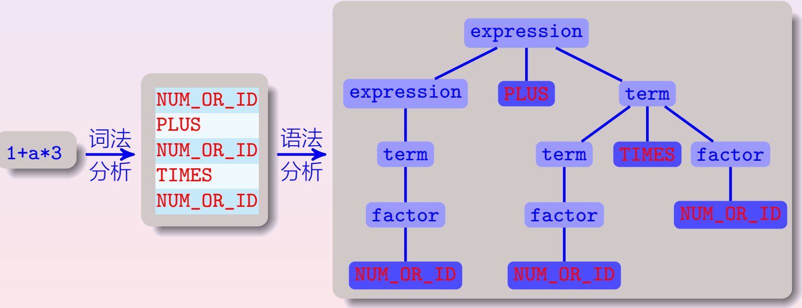
- 程序语言语句的合成：一维的字符流 $\longrightarrow$ 二维的**树结构**.
- 程序语言语义的合成：二维的树结构 $\longrightarrow$ 一维的目标码.

1+a*3 $\xrightarrow{\text{词法分析}}$

NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID

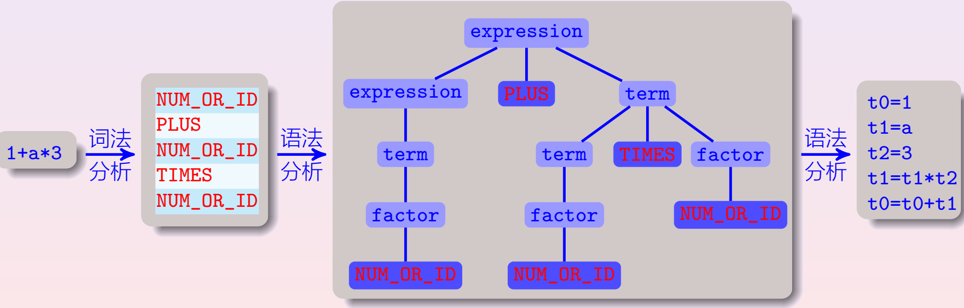
形式系统 (Formal Systems) 的两次合成

- 程序语言语句的合成：一维的字符流 $\longrightarrow$ 二维的**树结构**.
- 程序语言语义的合成：二维的树结构 $\longrightarrow$ 一维的目标码.



形式系统 (Formal Systems) 的两次合成

- 程序语言语句的合成：一维的字符流 ⟶ 二维的**树结构**.
- 程序语言语义的合成：二维的树结构 ⟶ 一维的目标码.



注释

- 预处理器 (preprocessor) 和连结程序 (linker)，如：C 语言：
  - “gcc -E” 输出经过预处理后的 C 源程序.
  - “gcc -S” 输出汇编代码.
  - “ln” 对.obj 文件进行连接生成执行文件.
- Native code and Bytecode，如：
  - “GCC” 仅能输出特定 CPU 的机器码: Sparc, MIPS, Alpha, Intel, Motorola, Arm.
  - “Java” 输出仅能在 Java virtual Machine 运行的 bytecode 机器码.
  - “OCaml” 能输出 Bytecode 和 Native code 两种形式的机器码.
  - “.NET” 输出 CLR(Common Language Runtime).
- Cross Compiler: 编译生成的目标语言不是 Native Code, 而是另一种 CPU 架构的机器代码，如：嵌入式系统的开发.
- Backend 还包含一个特别重要的内容就是代码优化，它是保证目标代码质量和执行效率的关键.

Why study compiler

- 认识形式系统
  - 程序设计语言手册完全是编译器的产品说明书 (See C: A Reference Manual (Fifth Ed.) by Harbision S.P. el al. Prentice Hall, 2002).
  - 词法、语法和语义是计算机形式系统必须具有的层次结构 (http://en.wikipedia.org/wiki/Formal_system).
- 理论和实现的最佳结合点.
- 计算机科学的小百科:
  - 算法：栈，图论，Hash 表，动态规划等.
  - 人工智能：合一算法，greedy algorithm，learning algorithm.
  - 理论: 自动机，形式文法，形式语义学.
  - 体系结构: 内存管理，指令选取与调度，Cache 优化等与 CPU 架构密切相关的优化.

编译技术面对的挑战

新的问题

- New Computer Architectures: VLIW (Very Long Instruction Word), instead of CISC (Complex Instruction Set Computer) and RISC (Reduced Instruction Set Computer), Multi-core processor (superscalar execution, pipelining, and multithreading).
- Embedded environment, SoC – Small ROM & RAM, 特殊的指令集.
- Program Security.

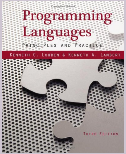
面对的挑战

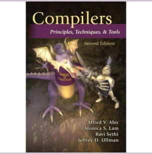
- 指令调度以支持指令的并行执行，降低 CPU 能耗（移动设备）.
- 代码的空间和内存的优化，以支持微处理器对 ROM 和 RAM 的限制.
- 程序的静态与动态分析以检测程序的安全漏洞.
- JIT (Just In Time) or AOT (Ahead Of Time) compiler for Android.
- 认证编译器 (Certified Compiler): 保证相关性质经编译后还能保持.

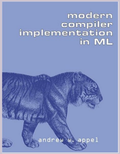
主要的参考书目

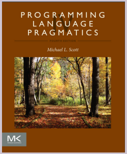
References

- **Kenneth C Louden** Programming Languages: Principles and Practices (Third Edition), Course Technology, 2011
- **Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman** Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, 2006 (Dragon book)
- **Andrew W. Appel** Modern Compiler Implementation in ML, Cambridge University Press, 2004. 2nd Ed. (Tiger Book)
- **Michael L. Scott** Programming Language Pragmatics, Fourth Edition, Morgan Kaufmann, 2015. (中译本：《程序设计语言：实践之路》裘宗燕，电子工业出版社)









重要链接

- My compiler_cd directory;
- http://compilers.iecc.com/index.phtml ： The comp.compilers newsgroup
- Dragon Book sources: http://dragonbook.stanford.edu/
- Alex Aiken Compiler Course.
- Ask questions： mailto:hanfei.wang@gmail.com
- 编程作业提交到： mailto:hanfei.wang@gmail.com.
  邮件主题： 学号 (n)
  n为编程作业的次数
  注意： 学号、括号和次数均为 ASCII 码!

1. 程序设计语言的历史

2. 程序设计语言的组成
   - 字符
   - 单词
   - 语句
   - 语义

3. 编译器的结构
   - 编译器的定义
   - 编译器的结构

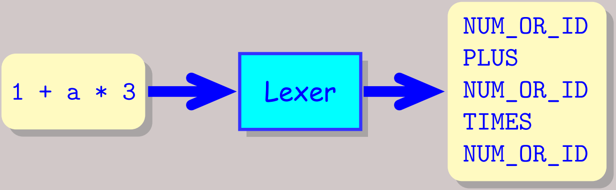
4. XL 语言编译器
   - XL 语言的形式规则
   - 词法分析器的设计
   - 递归下降语法分析器的设计
   - 语义分析及代码生成

词法

组成单词的规则

- 整数和标识符 (NUM_OR_ID): 0, ..., 9 组成的字符串，以字母开始并以字母和数字组成的字符串.
- 运算符号： + (PLUS), * (TIMES)
- 分隔符号: ; (SEMI), ((LP),) (RP)

XL 语言的词法分析



语法

Grammar

```
statements -> expression SEMI EOI
            | expression SEMI statements

expression -> expression PLUS term
            | term

term -> term TIMES factor
      | factor

factor -> NUM_OR_ID
        | LP expression RP
```

XL 语言的语法分析与语法树的建立

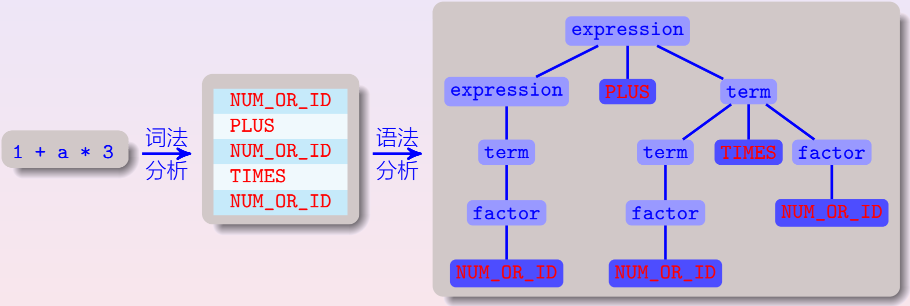
```
1 + a * 3
```

XL 语言的语法分析与语法树的建立

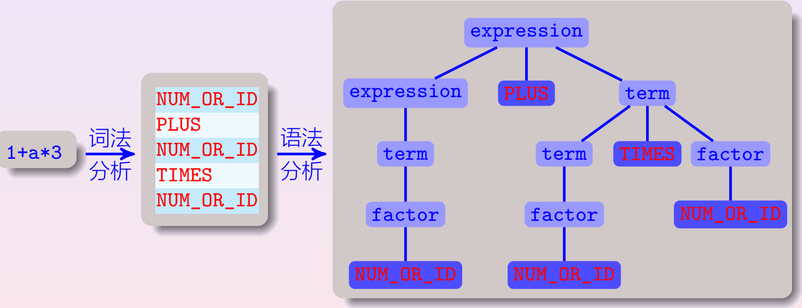
`1 + a * 3` $\xrightarrow{\text{词法分析}}$

NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID

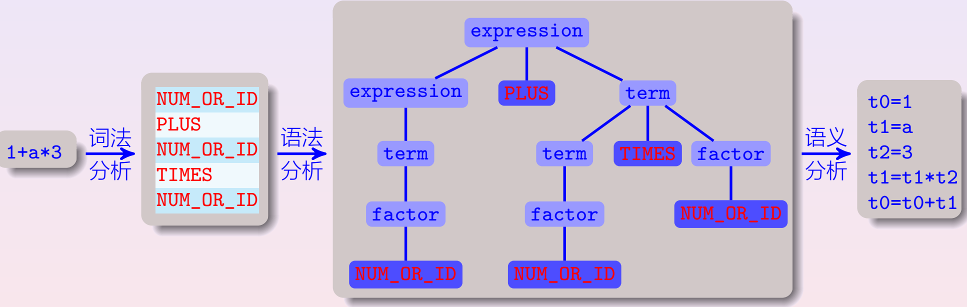
XL 语言的语法分析与语法树的建立



XL 语言的语义


1+a*3
词法分析
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
语法分析
expression
expression
PLUS
term
term
term
TIMES
factor
factor
factor
NUM_OR_ID
NUM_OR_ID
NUM_OR_ID

XL 语言的语义


1+a*3
词法分析
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
语法分析
expression
expression
PLUS
term
term
term
TIMES
factor
factor
factor
NUM_OR_ID
NUM_OR_ID
NUM_OR_ID
语义分析
t0=1
t1=a
t2=3
t1=t1*t2
t0=t0+t1

词法分析器的设计

Lexer Design

- 对输入字符串按照单词的规则进行分组，输出相应单词的编码.
- 白字符(white space, 空格，横向跳格，换行等格式控制符号) 等作为单词之间的分割符号，在语法分析时不再需要，分析器要对其进行过滤.
- 当分析器扫描到输入文件结束标记时，返回EOI编码.
- 间断式：对输入的扫描不是一次完成，分析器在获得语法分析器的请求后，返回一个单词编码后停止扫描. 等待语法分析器的下次调用. 因此，分析器必须用静态变量的方式保留分析其返回时的现场，以便再次调用时能够继续工作.

词法分析器的设计

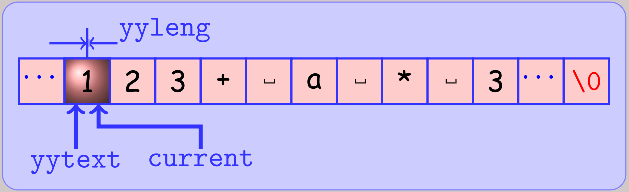
全局变量：

```
char *yytext ; /* 指向当前单词的开始字符 */
int yyleng; /* 当前扫描单词的的长度 */
static char input_buffer[128]; /* 输入缓冲区 */
```

局部变量：

```
char *current; /* 指向当前扫描的字符 */
```

缓冲区结构图



词法分析器的设计

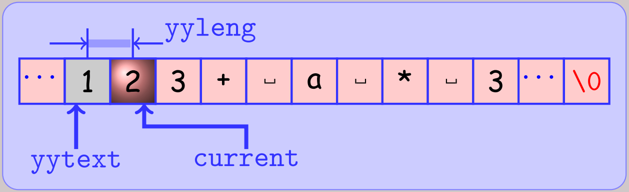
全局变量:

```
char *yytext ; /* 指向当前单词的开始字符 */
int yyleng; /* 当前扫描单词的的长度 */
static char input_buffer[128]; /* 输入缓冲区 */
```

局部变量:

```
char *current; /* 指向当前扫描的字符 */
```

缓冲区结构图



词法分析器的设计

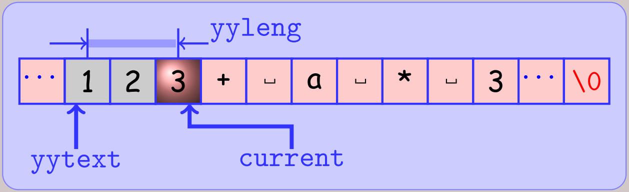
全局变量：

```
char *yytext ; /* 指向当前单词的开始字符 */
int yyleng; /* 当前扫描单词的的长度 */
static char input_buffer[128]; /* 输入缓冲区 */
```

局部变量：

```
char *current; /* 指向当前扫描的字符 */
```

缓冲区结构图



词法分析器的设计

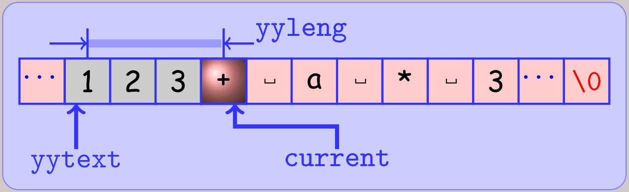
全局变量:

```
char *yytext ; /* 指向当前单词的开始字符 */
int yyleng; /* 当前扫描单词的的长度 */
static char input_buffer[128]; /* 输入缓冲区 */
```

局部变量:

```
char *current; /* 指向当前扫描的字符 */
```

缓冲区结构图



算法 lex.c

```
current = yytext + yyleng;

    /* 跳过已读过的单词 */
while ( 1 ) {  /* 读下一个单词  */
    while ( !*current ) {
      /* 当前缓冲区已读完,重新从键盘
         读入新的一行. 并且跳过空格 */
      current = input_buffer;
          /* 如果读行有误,返回 EOI */
      if ( !gets( input_buffer ) ) {
        *current = '\0' ;
        return EOI;
      }
      while ( isspace(*current) )
        ++current;
    }
    for( ; *current ; ++current ) {
          /* Get the next token */
      yytext = current;
      yyleng = 1;
          /* 返回不同的单词代码 */
```

```
      switch ( *current ) {
        case ';': return SEMI;
        case '+': return PLUS;
        case '*': return TIMES;
        case '(': return LP;
        case ')': return RP;
        case '\n': case '\t':
        case ' ' : break;
        default:
        if ( !isalnum(*current) )
          printf("Ignore illegal \
            input <%c>\n", *current);
        else {
          while ( isalnum(*current) )
            ++current;
          yyleng = current - yytext;
          return NUM_OR_ID;
        }
        break;
      }
    }
}
```

算法 lex.c

准备识别单词状态

```
current = yytext + yyleng;

    /* 跳过已读过的单词 */
while ( 1 ) {  /* 读下一个单词  */
    while ( !*current ) {
      /* 当前缓冲区已读完,重新从键盘
         读入新的一行. 并且跳过空格 */
      current = input_buffer;
          /* 如果读行有误,返回 EOI */
      if ( !gets( input_buffer ) ) {
        *current = '\0' ;
        return EOI;
      }
      while ( isspace(*current) )
        ++current;
    }
    for( ; *current ; ++current ) {
          /* Get the next token */
      yytext = current;
      yyleng = 1;
          /* 返回不同的单词代码 */
      switch ( *current ) {
        case ';': return SEMI;
        case '+': return PLUS;
        case '*': return TIMES;
        case '(': return LP;
        case ')': return RP;
        case '\n': case '\t':
        case ' ' : break;
        default:
        if ( !isalnum(*current) )
          printf("Ignore illegal \
            input <%c>\n", *current);
        else {
          while ( isalnum(*current) )
            ++current;
          yyleng = current - yytext;
          return NUM_OR_ID;
        }
        break;
      }
    }
}
```

算法 lex.c

准备识别单词状态

```
current = yytext + yyleng;

  /* 跳过已读过的单词 */
while ( 1 ) {  /* 读下一个单词  */
  while ( !*current ) {
    /* 当前缓冲区已读完,重新从键盘
       读入新的一行. 并且跳过空格 */
    current = input_buffer;
        /* 如果读行有误,返回 EOI */
    if ( !gets( input_buffer ) ) {
      *current = '\0' ;
      return EOI;
    }
    while ( isspace(*current) )
      ++current;
  }
  for( ; *current ; ++current ) {
        /* Get the next token */
    yytext = current;
    yyleng = 1;
        /* 返回不同的单词代码 */
```

缓冲区已空状态

```
      switch ( *current ) {
        case ';': return SEMI;
        case '+': return PLUS;
        case '*': return TIMES;
        case '(': return LP;
        case ')': return RP;
        case '\n': case '\t':
        case ' ' : break;
        default:
        if ( !isalnum(*current) )
          printf("Ignore illegal \
            input <%c>\n", *current);
        else {
          while ( isalnum(*current) )
            ++current;
          yyleng = current - yytext;
          return NUM_OR_ID;
        }
        break;
    }
  }
}
```

算法 lex.c

准备识别单词状态

```
current = yytext + yyleng;

   /* 跳过已读过的单词 */
while ( 1 ) {  /* 读下一个单词  */
    while ( !*current ) {
      /* 当前缓冲区已读完,重新从键盘
         读入新的一行. 并且跳过空格 */
      current = input_buffer;
          /* 如果读行有误,返回 EOI */
      if ( !gets( input_buffer ) ) {
        *current = '\0' ;
        return EOI;
      }
      while ( isspace(*current) )
        ++current;
    }
    for( ; *current ; ++current ) {
          /* Get the next token */
      yytext = current;
      yyleng = 1;
          /* 返回不同的单词代码 */
```

缓冲区已空状态

```
      switch ( *current ) {
        case ';': return SEMI;
        case '+': return PLUS;
        case '*': return TIMES;
        case '(': return LP;
        case ')': return RP;
        case '\n': case '\t':
        case ' ' : break;
        default:
        if ( !isalnum(*current) )
          printf("Ignore illegal \
            input <%c>\n", *current);
        else {
          while ( isalnum(*current) )
            ++current;
          yyleng = current - yytext;
          return NUM_OR_ID;
        }
        break;
      }
    }
}
```

最后识别的是字母数字状态

©hfwang

接口

lex.c

```
static int Lookahead = -1;
  /* 向前查看的单词,设初值为-1. 表示第一次调用
     match函数时, 必须要读取一个单词 */

int match( token ) int token;
{
  /* 判断token是否和当前向前查看的单词相同. */
  if  (Lookahead == -1)
     Lookahead = lex();
  return token == Lookahead;
}

void advance()
{
  Lookahead = lex();/* 向前都一个词汇 */
}
```

递归下降语法分析器的设计

设计思想

- 函数递归调用的结构和语法树结构一致.
- 利用函数递归调用来建立语法树.
- 每个语句结构对应一个函数, 每个单词用词法分析器的 match() 来匹配, 如果匹配则该单词已经满足语法规则, 不再需要, 调用 advance() 准备下一个单词.

Example

expression -> expression PLUS term

对应下列函数:

```
void expression (void)
{
  expression ();
  if (match(PLUS)) { advance();
    term(); }
}
```

递归下降语法分析器的设计

设计思想

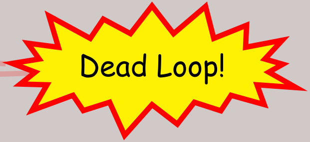
- 函数递归调用的结构和语法树结构一致.
- 利用函数递归调用来建立语法树.
- 每个语句结构对应一个函数，每个单词用词法分析器的 match() 来匹配，如果匹配则该单词已经满足语法规则，不再需要，调用 advance() 准备下一个单词.

Example

```
expression -> expression PLUS term
```

对应下列函数：

```
void expression (void)
{
  expression ();
  if (match(PLUS)) { advance();
    term(); }
}
```



文法的等价变换

原文法

```
statements -> expression SEMI EOI
            |  expression SEMI
               statements

expression -> expression PLUS term
            |  term

term -> term TIMES factor
      |  factor

factor -> NUM_OR_ID
        |  LP expression  RP
```

⇔

等价文法

```
statements -> expression SEMI EOI
            |  expression SEMI
               statements

expression -> term expression'

expression' -> PLUS term expression'
             |  epsilon   /* 空串 */

term -> factor term'

term' -> TIMES factor term'
       |  epsilon    /* 空串 */

factor -> NUM_OR_ID
        | LP expression RP
```

算法

递归函数

```
/* expression -> term expression' */

void expression(void)
{
  term();
  expr_prime();
}

/* expression' -> PLUS term expression'
 *              | epsilon
 */

void expr_prime(void)
{
  if ( match( PLUS ) ) {
    advance();
    term();
    expr_prime();
    }
}
```

⇒

简化算法

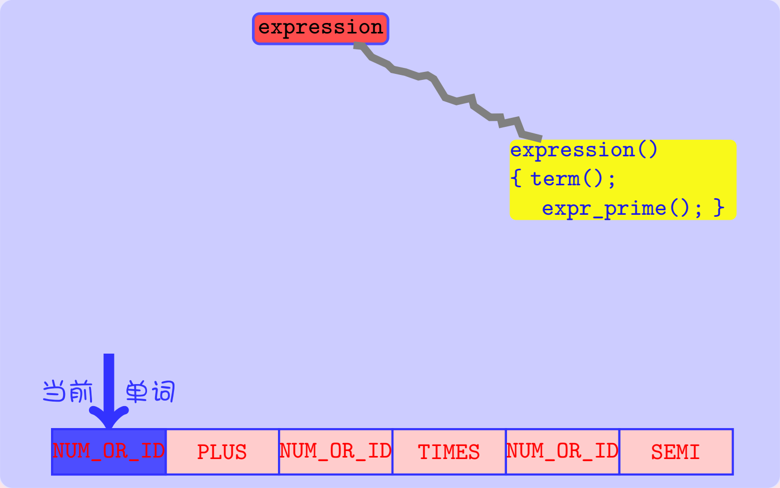
```
/*
   expression ->
     term ( PLUS term expression')*
 */

expression()
{
  term();
  while ( match( PLUS) ) {
    advance();
    term();
  }
}
```

分析过程演示

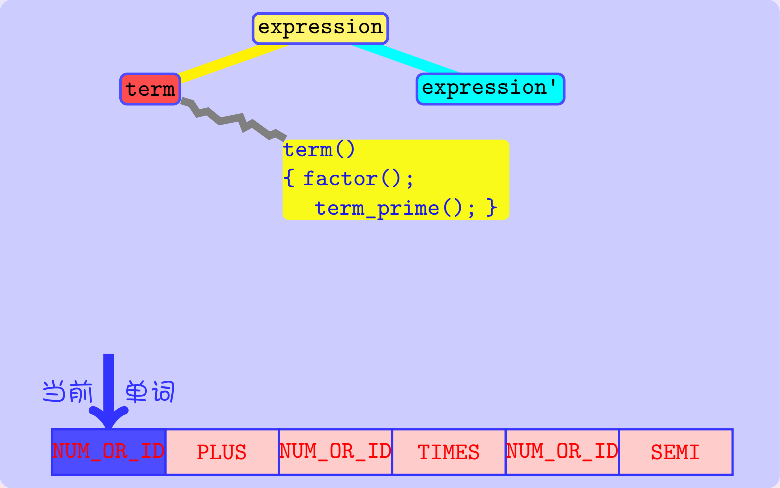
当前 单词

NUM_OR_ID	PLUS	NUM_OR_ID	TIMES	NUM_OR_ID	SEMI

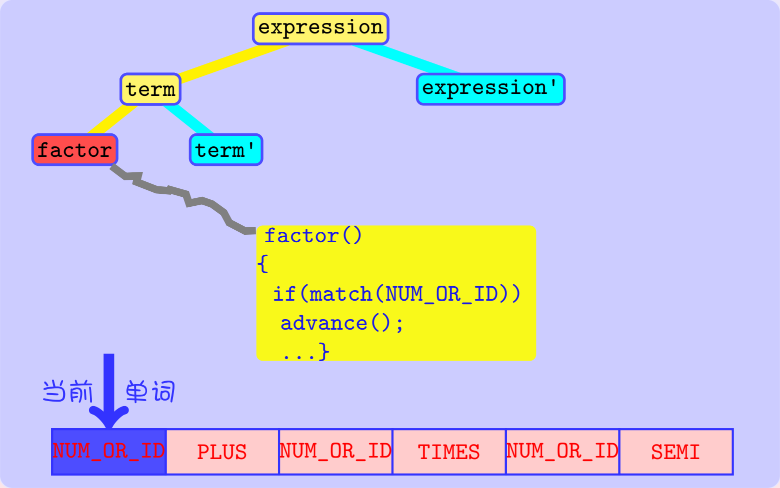
分析过程演示


expression
expression()
{ term();
expr_prime(); }
当前
单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

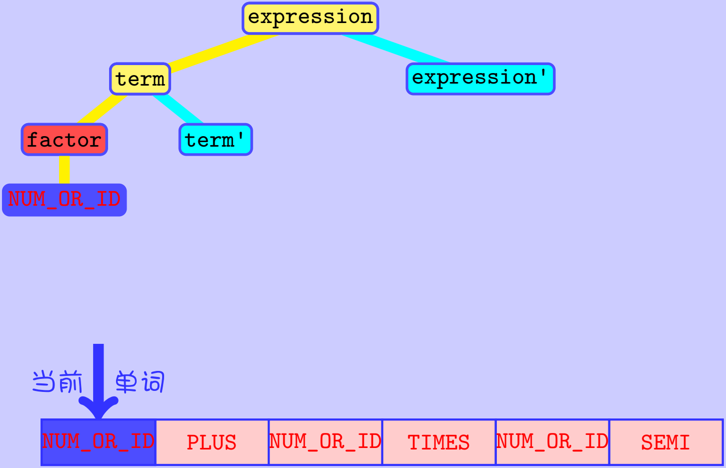
分析过程演示


expression
term
expression'
term()
{ factor();
term_prime(); }
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

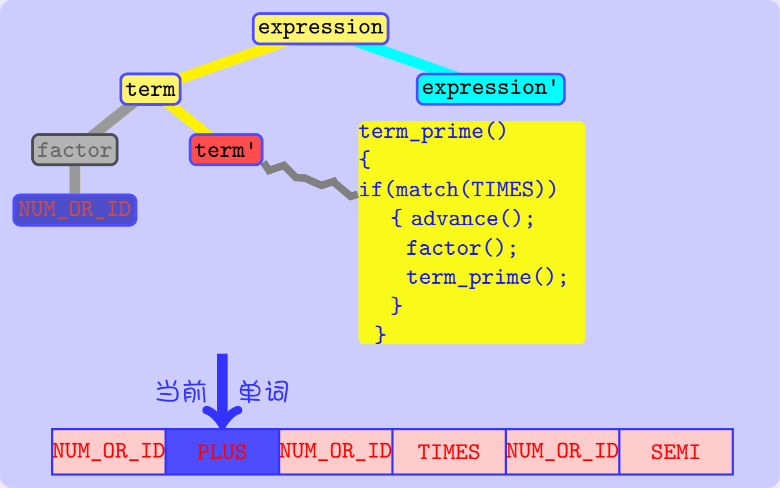
分析过程演示


expression
term
expression'
factor
term'
factor()
{
if(match(NUM_OR_ID))
advance();
...}
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

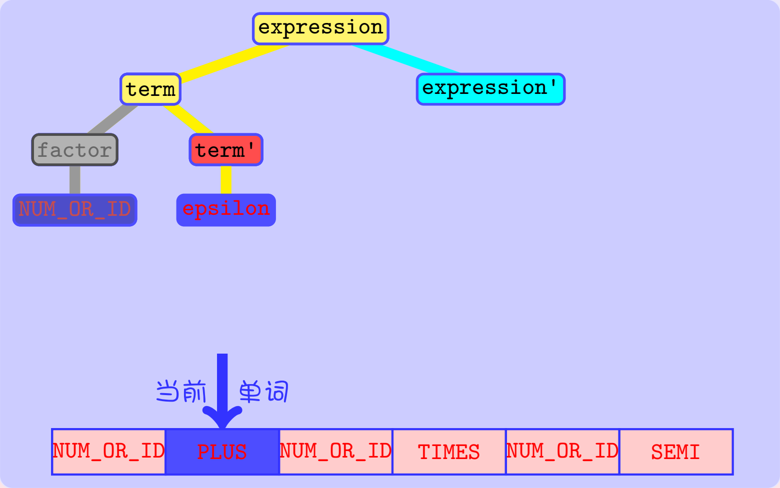
分析过程演示


expression
term
expression'
factor
term'
NUM_OR_ID
当前
单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

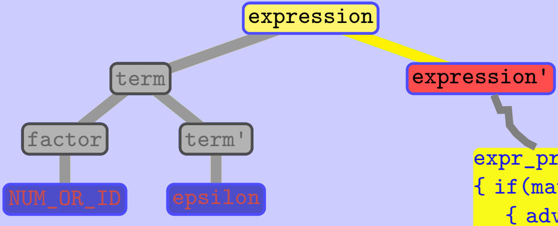
分析过程演示


expression
term
expression'
factor
term'
NUM_OR_ID
term_prime()
{
if(match(TIMES))
{ advance();
factor();
term_prime();
}
}
当前单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

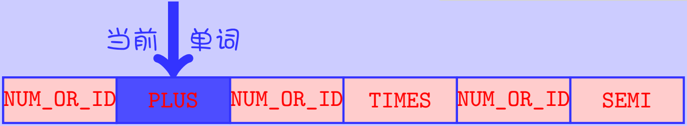
分析过程演示


expression
term
expression'
factor
term'
NUM_OR_ID
epsilon
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

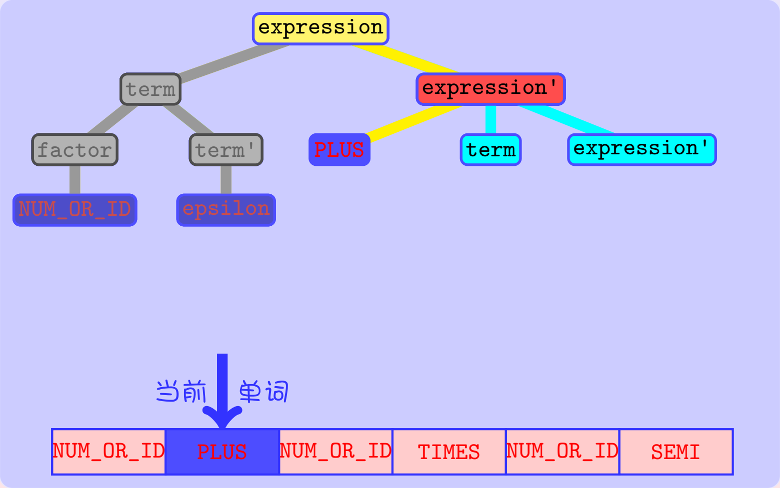
分析过程演示



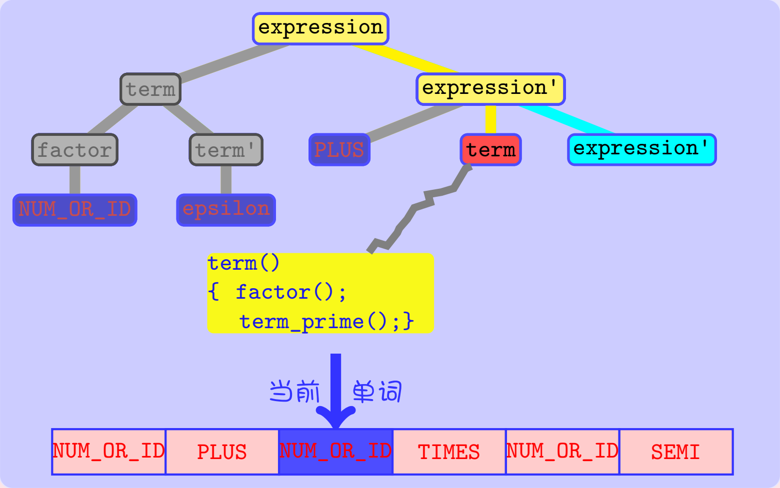

```
expr_prime()
{ if(match(PLUS))
   { advance();
     term();
     expr_prime();
   }
 }
```



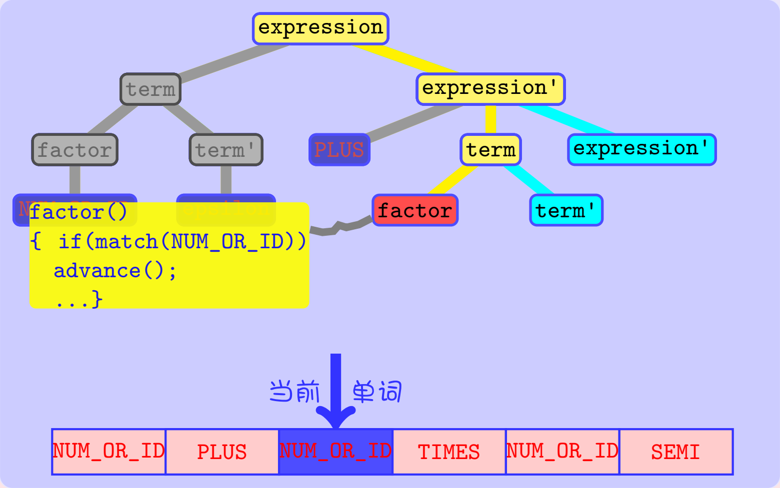
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

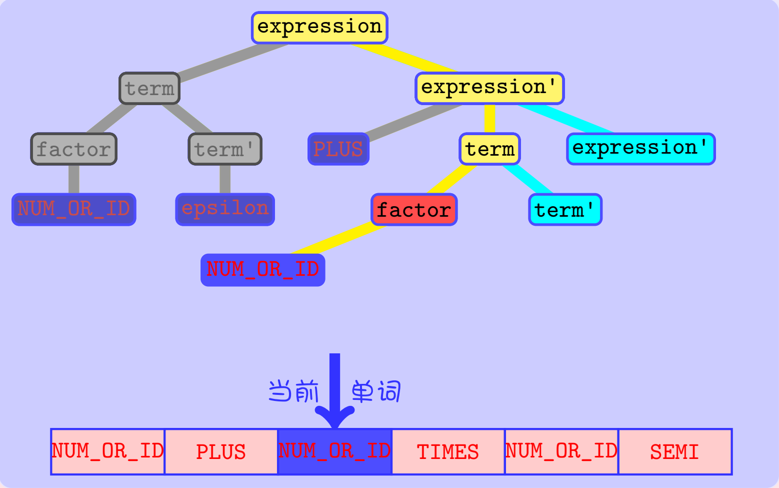
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
term()
{ factor();
term_prime();}
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

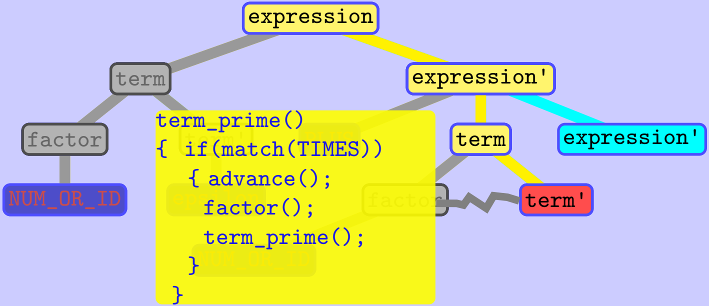
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
factor
term'
factor()
{ if(match(NUM_OR_ID))
advance();
...}
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
NUM_OR_ID
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

分析过程演示

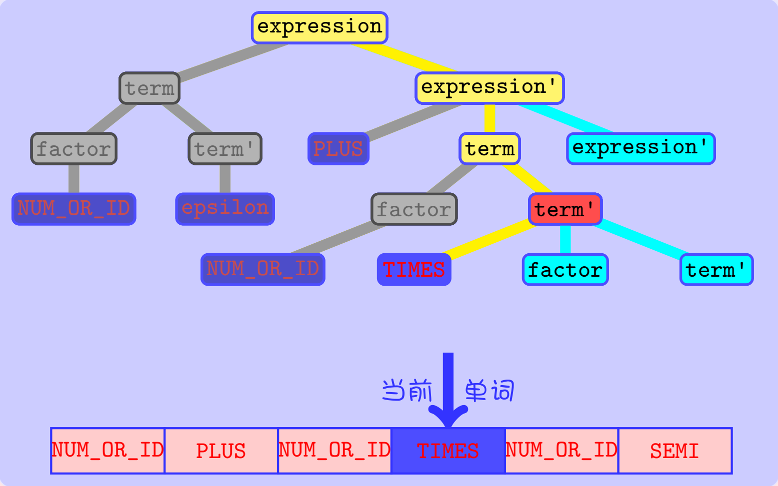



```
term_prime()
{  if(match(TIMES))
   { advance();
     factor();
     term_prime();
   }
 }
```

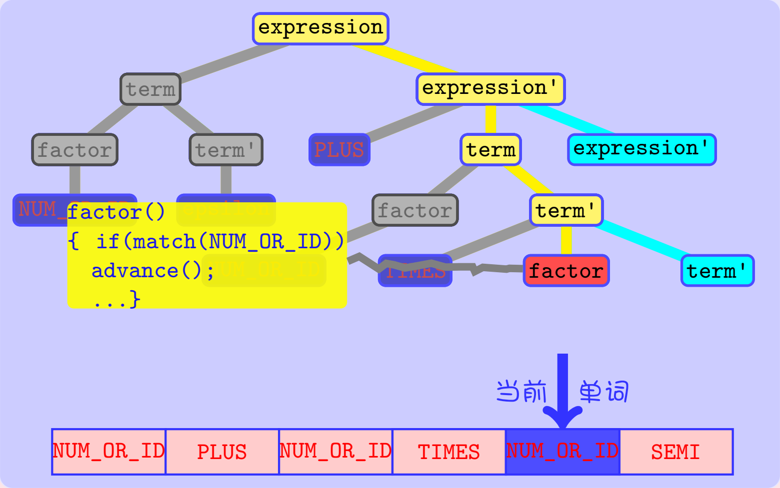
当前 单词

NUM_OR_ID	PLUS	NUM_OR_ID	TIMES	NUM_OR_ID	SEMI

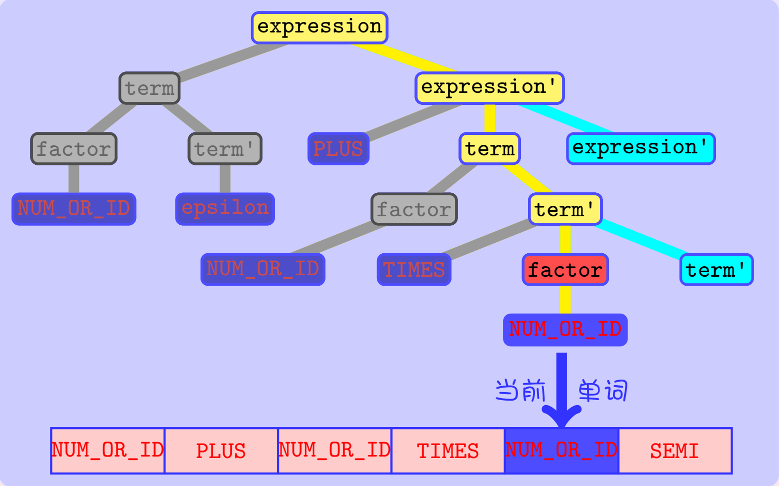
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
NUM_OR_ID
TIMES
factor
term'
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

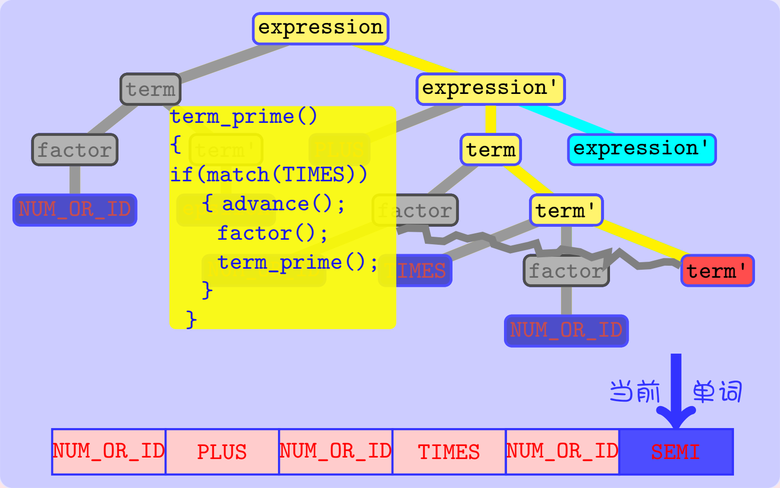
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
factor
term'
factor()
{ if(match(NUM_OR_ID))
advance();
...}
factor
term'
当前
单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

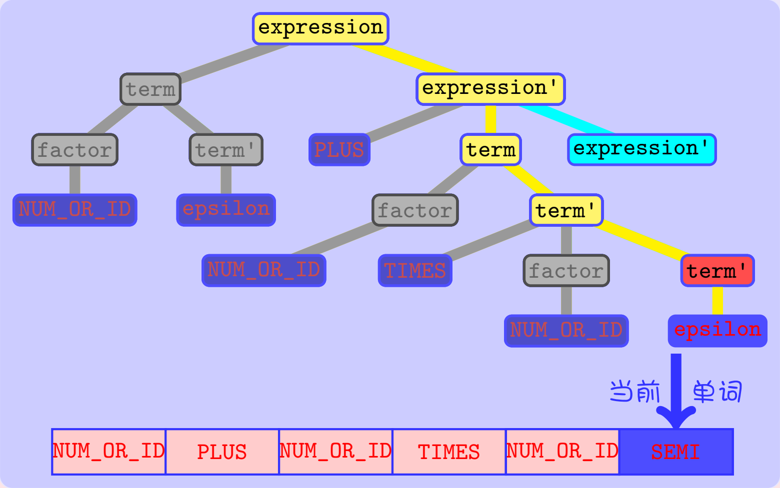
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
NUM_OR_ID
TIMES
factor
term'
NUM_OR_ID
当前
单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

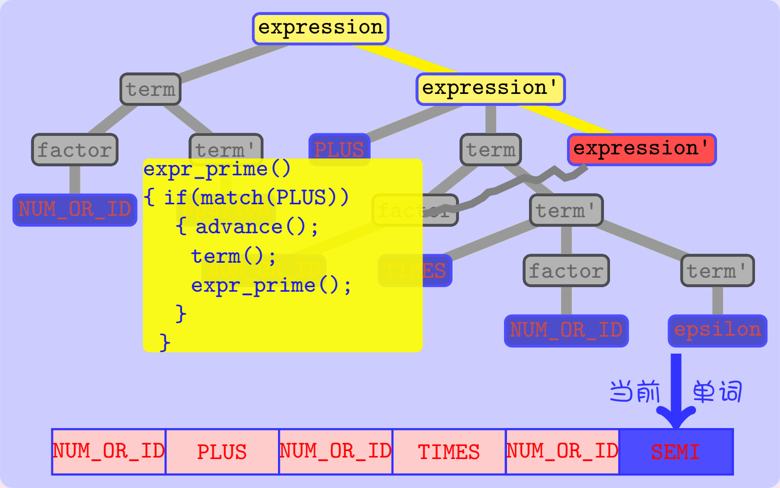
分析过程演示


expression
term
expression'
factor
term
expression'
NUM_OR_ID
term'
term_prime()
{
if(match(TIMES))
{ advance();
factor();
term_prime();
}
}
factor
IMES
factor
term'
NUM_OR_ID
当前
单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

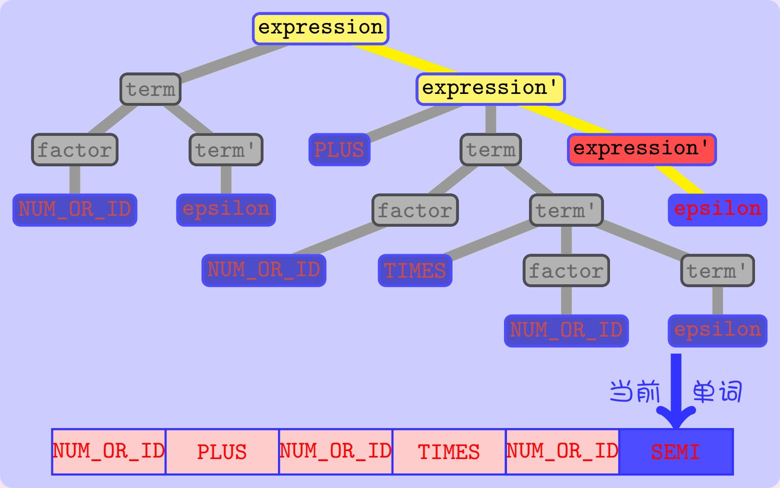
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
NUM_OR_ID
TIMES
factor
term'
NUM_OR_ID
epsilon
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

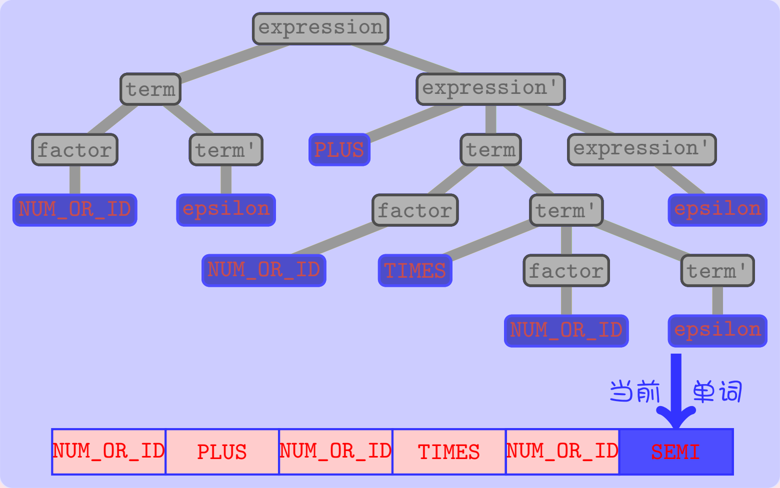
分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
term'
factor
TIMES
factor
term'
NUM_OR_ID
epsilon
expr_prime()
{ if(match(PLUS))
{ advance();
term();
expr_prime();
}
}
当前 单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
epsilon
NUM_OR_ID
TIMES
factor
term'
NUM_OR_ID
epsilon
当前单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

分析过程演示


expression
term
expression'
factor
term'
PLUS
term
expression'
NUM_OR_ID
epsilon
factor
term'
epsilon
NUM_OR_ID
TIMES
factor
term'
NUM_OR_ID
epsilon
当前单词
NUM_OR_ID
PLUS
NUM_OR_ID
TIMES
NUM_OR_ID
SEMI

几点说明

Remarks

- 语义可以按照语法树的结构进行组合.
- 语法树的每个节点和其子节点的关系只有有限种可能，即是语法中相关语句成分对应的规则.
- 形式语言的合法语句正是按固定规则组成的语法树的叶节点从左导右顺序排列的单词串.
- 也正是因为这有限的可能，才使得一个语法分析程序能够分析可能无限多的语句，从而满足编译器必须示泛函的条件.
- 语法分析通过语法树的建立的过程判断给定的语句是否合法，以及为下一步的语义分析提供必要的信息.

语义分析及代码生成

- 语义可以按照语法树的结构进行组合 – 语法制导.
- 每个语法规则规定了其对应语句成分的基本语义信息.

expression规则

```
expression -> term1 ( PLUS term2 )*

- 设term1的计算结果保存在
  临时变量t1中.
- 设表达式term2的计算结果
  保存在临时变量t2中,

则上述语法规则对应的语义:

1/ 产生代码: t1 = t1 + t2
2/ 释放临时变量t2
3/ 将t1作为expression的计算
   结果返回
```

对应算法

```
char *expression()
{
  char *tempvar, *tempvar2;

  tempvar = term();
  while ( match( PLUS ) ) {
    advance();
    tempvar2 = term();
    printf("%s += %s\n",
      tempvar, tempvar2 );
    freename( tempvar2 );
  }
  return tempvar;
}
```

另一种语义解释 — 中缀表达式翻译为前缀表达式

expression规则

```
expression -> term1 ( PLUS term2 )*
```

设term1的前缀式保存在临时变量t1中，设表达式term2的计算结果保存在临时变量t2中. 则上述语法规则对应的语义:

1/ 生成新的字符串 "+" + t1 + t2
2/ 释放临时变量t1和t2
3/ 将t1作为expression的计算结果返回

为了支持语义分析能同时计算中间代码和前缀码两个不同的语义解释，特对递归函数的返回值的类型做如下修改:

```
typedef struct {
  char * val; /* 临时变量指针 */
  char * exp; /* 前缀表达式 */
} YYLVAL;
```

对应算法

```
YYLVAL *expression()
{
  YYLVAL *tempvar, *tempvar2;
  char *affix;

  tempvar = term();
  while ( match( PLUS ) ) {
    advance();
    tempvar2 = term();
    处理中间代码;
    affix = "+ " + tempvar->exp +
            " " + tempvar2->exp;
    释放tempvar->val, tempvar->exp,
      tempvar2->val, tempvar2->exp,
      tempvar2;
    tempvar->exp = affix;
  }
  return tempvar;
}
```

Make all together

lex.h	词汇宏定义头文件
lex.c	词法分析模块
plain.c	语法分析模块(不含语义分析)
improved.c	plain.c的改进，提通过legal_lookahead函数将 expresssion 和expression'; term和term'合为一个函数处理,简化plain.c 的分析程序
name.c	临时变量分配函数
retval.c	语法分析模块(含有语义分析)
main.c	主函数
plain.prj	工程文件，以lex.h，lex.c，plain.c和main.c组成
unixmake.mak	Unix Makefile
tcmake.mak	Turbo C Makefile

编译方法

- Linux 环境：`make -f unixmake.mak`生成执行文件.
- Turbo C 2.0 + DOS 命令行：`make -ftcmake.mak`. 或
- Turbo C 2.0 IDE: “`open project`” + `F9`.

What we learned from XL 1/2

浅层

- 模块化程序设计.
- 词法、语法和语义交互进行，语法分析为核心.
- `while (1)` 循环处理词法的各种可能.
- 函数的递归调用和语法树的一致性.
- 语义对应于递归函数的返回值.

What we learned from XL 2/2

深层

- 形式语言的两级抽象结构：词法和语法.
- 每层抽象都有一组有限的规则描述：词法和语法.
- 有限的规则组成无限可能的语句.
- 编译器的首要问题是识别语言.
- 一个形式语言的语句可能是千变万化，但是语句的组成规则是固定的，编译器就是利用这有限规则对语句的合成机制编程，从而使编译器是泛函成为可能.
- 寻找形式系统的相对不变性是编译技术要解决的核心问题.

本章小节

1. 程序设计语言的历史
2. 程序设计语言的组成
   - 字符
   - 单词
   - 语句
   - 语义
3. 编译器的结构
   - 编译器的定义
   - 编译器的结构
4. XL 语言编译器
   - XL 语言的形式规则
   - 词法分析器的设计
   - 递归下降语法分析器的设计
   - 语义分析及代码生成