编译竞赛培训（一）

计算机研究所编译系统室

全国大学生计算机系统能力大赛

COMP
CCRG Open MP

2023年全国大学生计算机系统能力大赛
编译系统设计赛技术方案

详情访问大赛技术平台：
https://compiler.educg.net

一、评价方式的基本说明

第1条 大赛要求各参赛队综合运用各种知识（包括但不局限于编译技术、操作系统、计算机体系结构等），构思并实现一个综合性的编译系统，以展示面向特定目标平台的编译器构造与编译优化的能力。

第2条 大赛鼓励各参赛队充分了解目标语言及目标硬件平台（CPU指令集、 Cache、各类并行加速能力等）特点，使编译出的目标码能够尽可能利用目标硬件平台能力以提高目标码的运行效率。

第3条 为展示参赛队的设计水平，增加竞赛的对抗性，进入决赛的参赛队还需针对目标语言或目标平台的变化，现场调整编译系统。

第4条 除本技术方案特别要求、规定和禁止事项外，各参赛队可自行决定编译器体系结构、前端与后端设计、代码优化等细节。

内容

1、编译概念

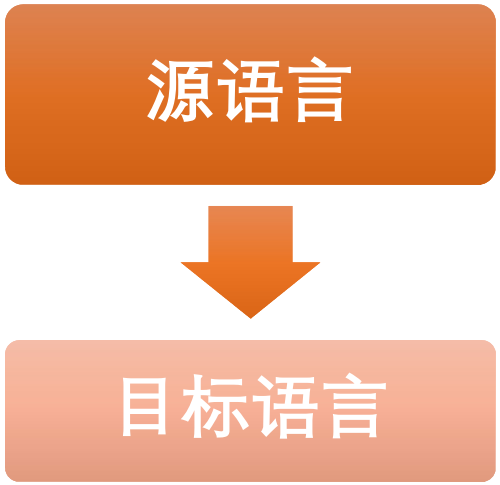
2、编译过程

3、编译器结构

4、编译器前端

1.1 什么是编译

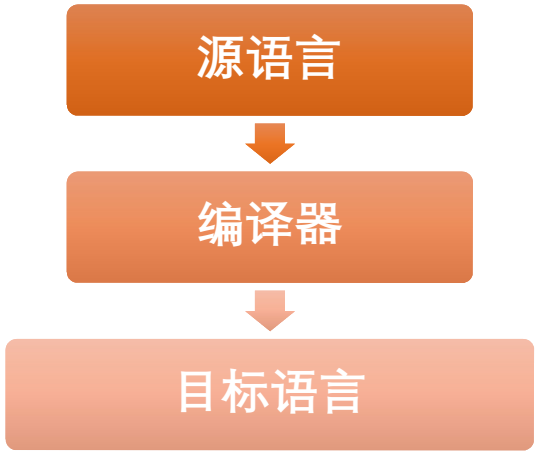
- 将一种语言（源语言）编写的程序，翻译成等价的、用另一种语言（目标语言）编写的程序

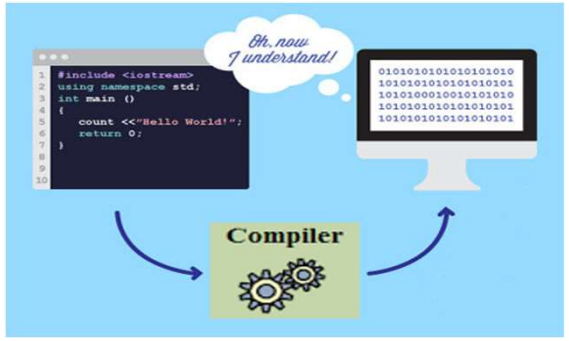


- 编译
  - 翻译成另一种语言
  - 源语言与目标语言等价

1.2 什么是编译器

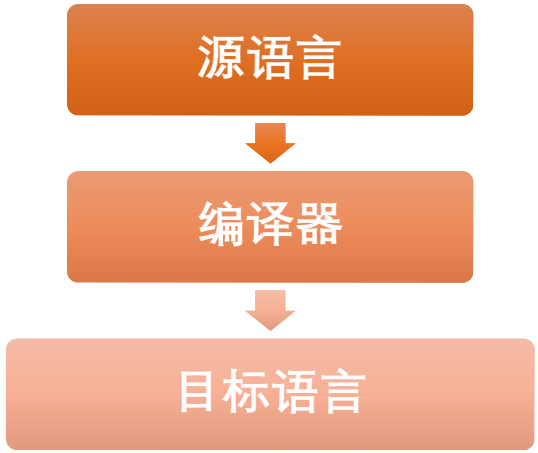
- 编译程序，将一种语言（源语言）编写的程序，翻译成等价的、用另一种语言（目标语言）编写的程序的计算机软件





1.2 什么是编译器

- 编译程序，将一种语言（源语言）编写的程序，翻译成等价的、用另一种语言（目标语言）编写的程序的计算机软件



IBM 704， 1954.05-1960.04

编译器最开始直接将高级语言编写的源程序翻译为可执行的目标代码。

1.2 什么是编译器

- 目标语言通常是目标机汇编语言

sin.c

```
int myfun(doule a)
{
  double result = sin(a);
  printf("result = %12.f\n", result);
   return 0;
}
```

1.2 什么是编译器

- 目标语言通常是目标机汇编语言

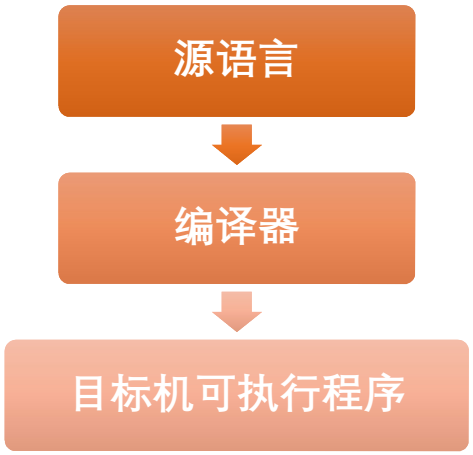
sin.c

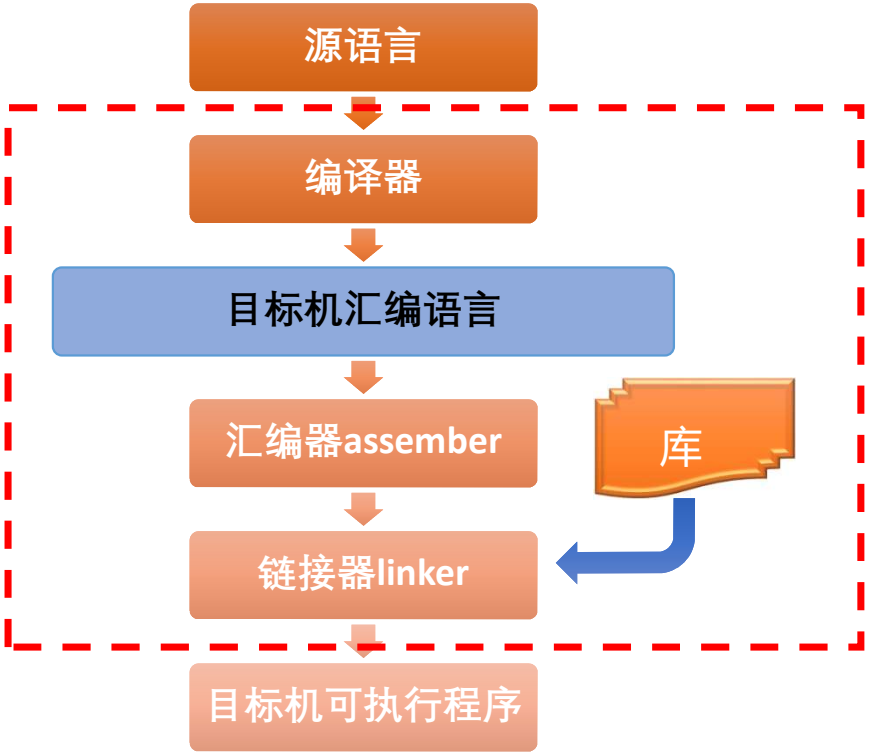
```
#include <stdio.h>
#include <math.h>
int myfun(doule a)
{
  double result = sin(a);
  printf("result = %12.f\n", result);
   return 0;
}
```

#clang sin.c –lm

1.2 什么是编译器

- 目标语言通常是目标机汇编语言

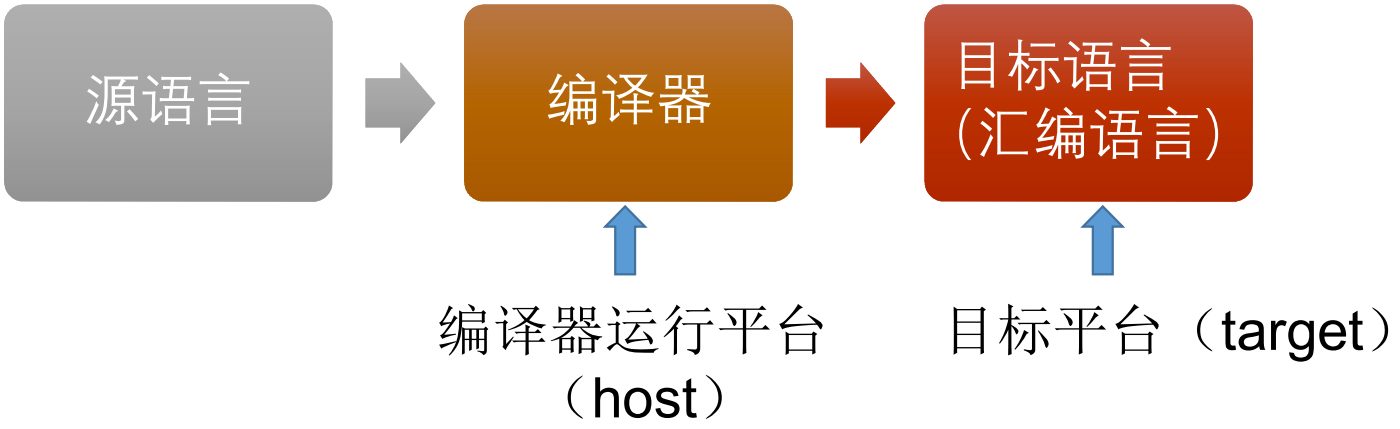




1.3 编译器分类

- 根据目标语言的不同

	目标语言
本地编译器（Native Compiler）	编译器运行平台的汇编语言
交叉编译器（Cross-Compiler）	另一种平台的汇编语言
源到源编译器（Source-to-Source Compiler）	另一种高级语言



1.3 编译器分类

- 根据源语言的不同

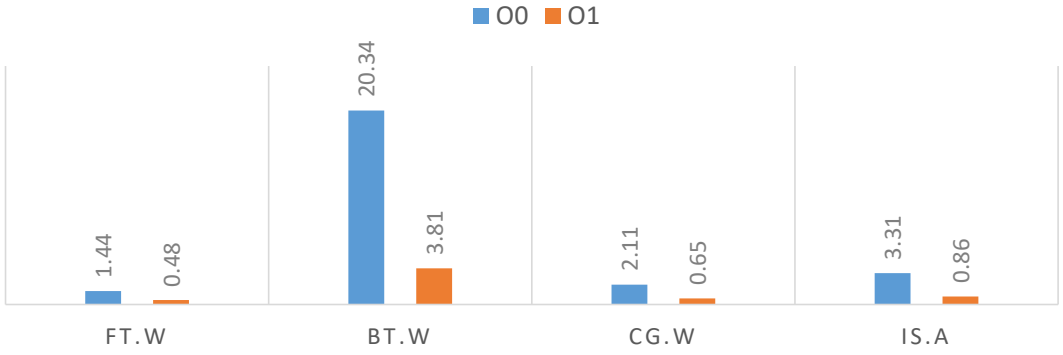
C语言编译器	icc，gcc，clang
C++语言编译器	icpc，g++，clang++
Fortran语言编译器	ifort，gfotran，flang

1.4 编译器功能

- 将源语言程序翻译为机器的汇编语言或者其他语言，使用户可以通过该源语言编写程序
  - 高级语言编写程序
- 在编译过程中，帮助用户发现、定位程序中的错误，提高程序的产出率和可靠性

1.4 编译器功能

- 将源语言程序翻译为机器的汇编语言或者其他语言，使用户可以通过该源语言编写程序
- 在编译过程中，帮助用户发现、定位程序中的错误，提高程序的产出率和可靠性
- 通过代码转换，优化产生的目标语言程序质量



GCC（10.2）FT2000+处理器
NPB3.3.1

1.5 优化编译器

- 优化编译器=翻译+优化
- 优化（Optimization）
  - 通过分析编译过程中的中间表示，对其进行转换，以便生成更好的目标代码
- 优化目标— “更好的”目标代码
  - 执行时间更短
  - 目标代码更小
  - 执行能耗更低
  - 浮点精度更高
  - ……

1.5 优化编译器

- 优化编译器=翻译+优化
- 优化必须
  - 保持程序语义（安全的）
  - 有价值的
    - 普遍适用的
    - 执行代价可接受的
- 优化 = 分析 + 转换

COMP
CCRG Open MP

1.5 优化编译器

- 关键的分析
  - 控制流分析（Control-flow analysis）
  - 数据流分析（Data-flow analysis）
  - 别名分析（Alias analysis）
  - 依赖关系分析（Dependence analysis）
  - ……
- 优化 = 分析 + 转换

1.5 优化编译器

- 优化：基于分析的结果进行代码转换

```
for(i = 0; i< a.length-foo; i++)
    sum += a[i];
```

到达-定值分析 ➔ 循环不变量外提优化

```
t = a.length-foo;
for(i = 0; i< t; i++)
    sum += a[i];
```

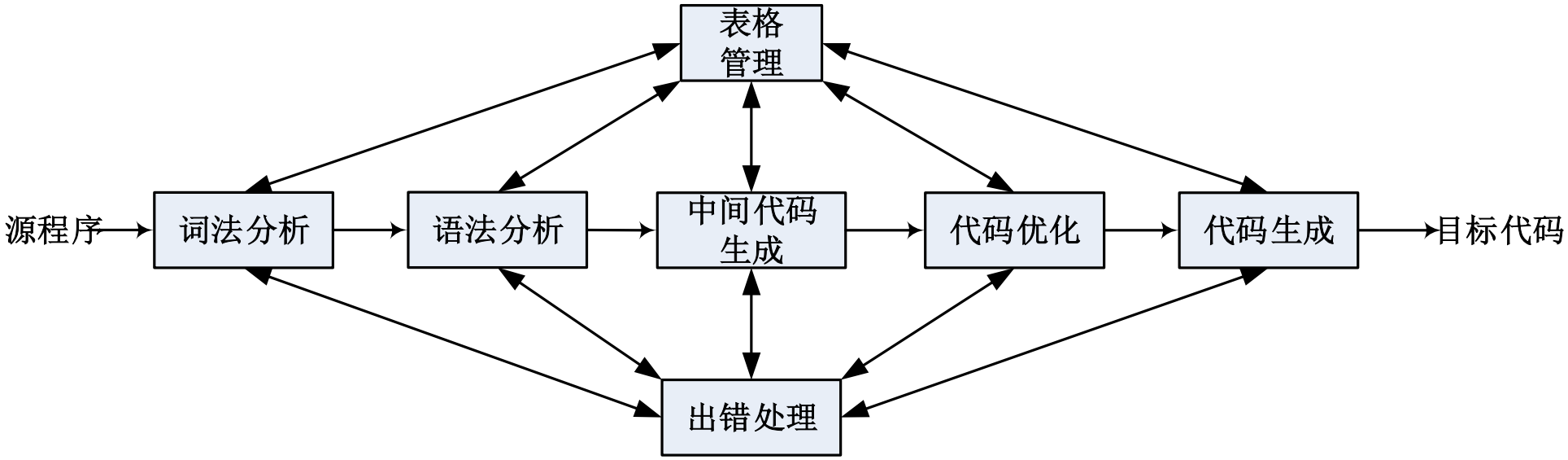
1.6 编译器的作用

- 应用、语言和体系结构之间的桥梁
- 帮助程序员发现和改正错误，提高程序可靠性和生产率
- 编译优化能够极大地提高程序性能

- 评估计算机体系结构的重要工具
- 支撑其他程序开发和调试工具
  - 性能分析工具、调试器、错误检测工具……

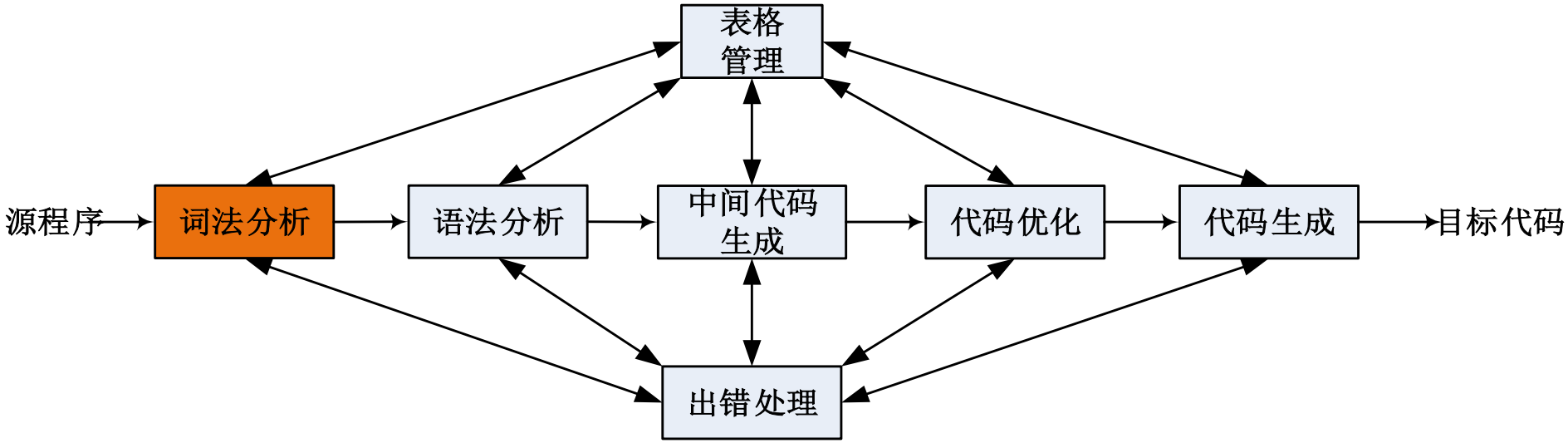
2、编译过程

编译过程


表格
管理
源程序
词法分析
语法分析
中间代码
生成
代码优化
代码生成
目标代码
出错处理

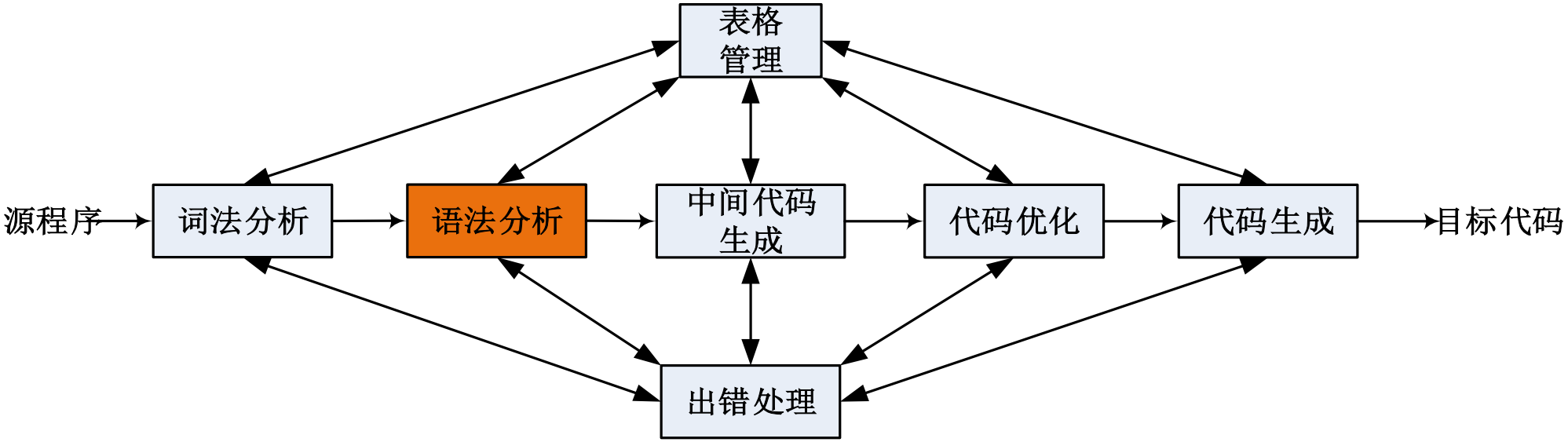

词法分析
语法分析
中间代码
生成
代码优化
代码生成

2.1 编译过程



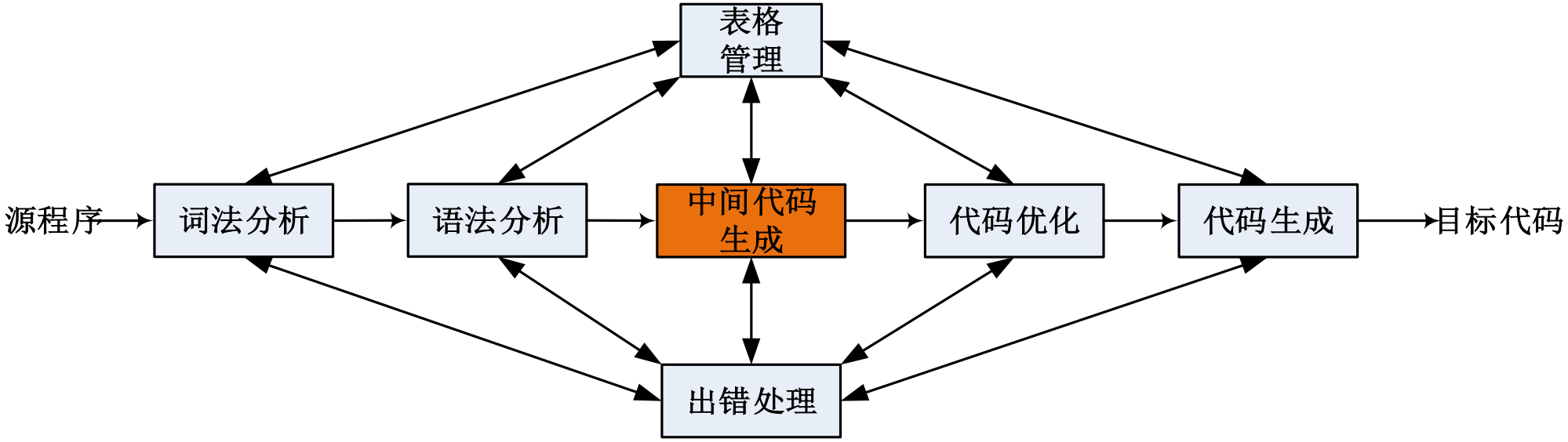
- 词法分析
  - 对构成源程序的字符流进行扫描，将字符组成有意义的单词符号序列

2.1 编译过程



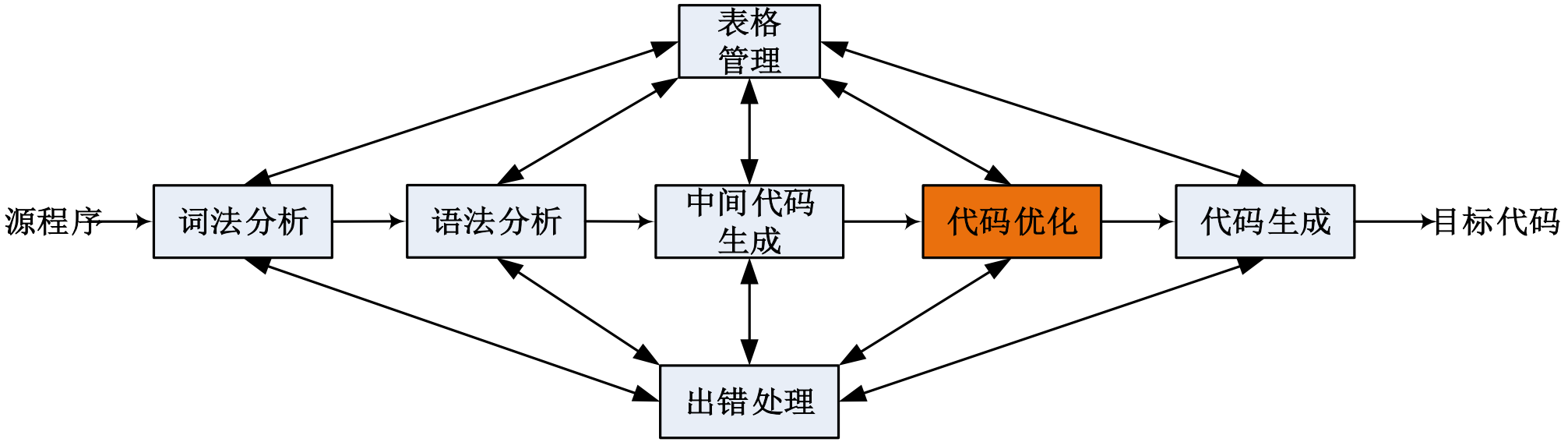
- 语法分析
  - 根据程序设计语言的语法规则，把单词符号串分解成各类语法单位，确定整个输入符号串是否符合语言的语法规范

2.1 编译过程



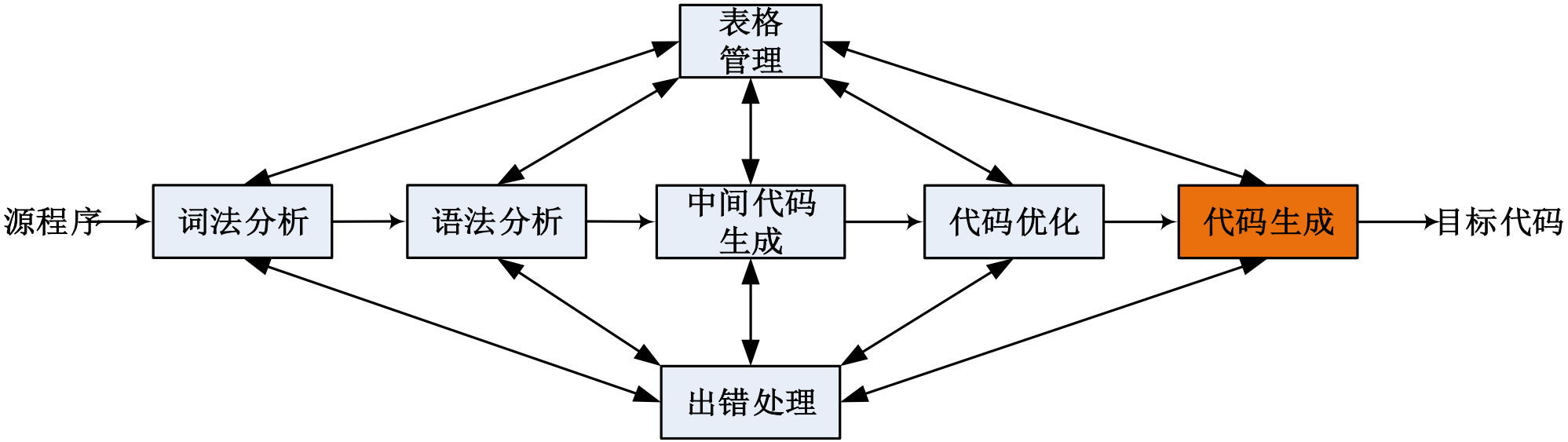
- 中间代码生成
  - 对语法分析所识别出的各类语法范畴，分析其含义，并产生中间代码

2.1 编译过程



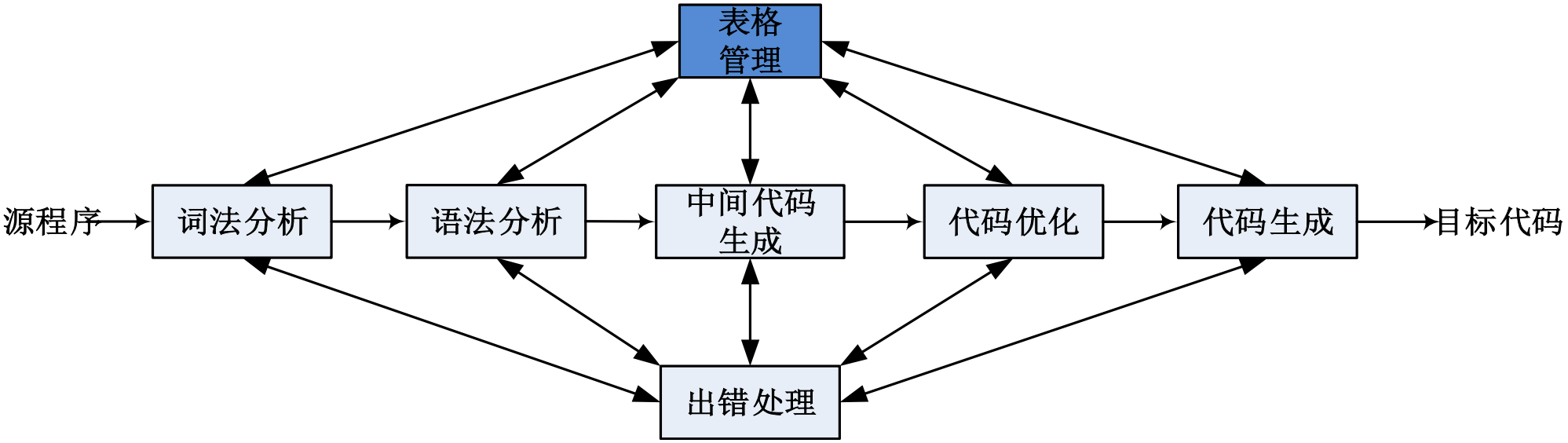
- 代码优化
  - 分析中间代码，对其进行转换，以便生成更好的目标代码

2.1 编译过程



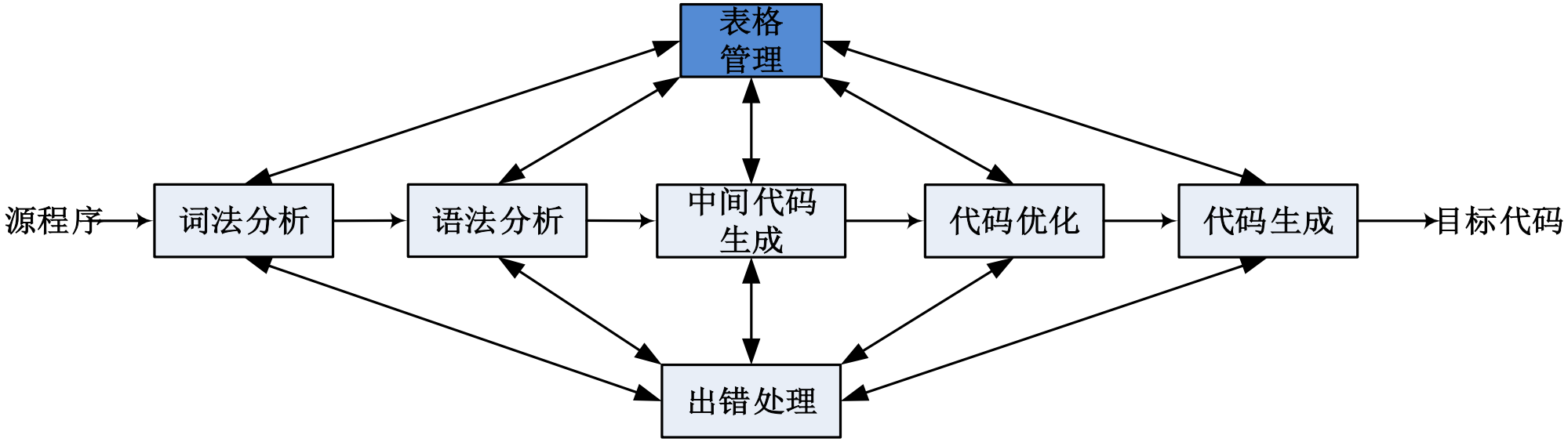
- 代码生成
  - 将中间代码翻译成目标程序

2.1 编译过程



- 表格管理
  - 维持一系列表格，记录源程序的各类信息，供编译的各个阶段使用
  - 符号表

2.1 编译遍



- 5个阶段仅仅是逻辑功能上的划分，在实现时，往往将编译器组织成若干**遍（Pass）**

2.2 编译遍

- 编译遍
  - 从头到尾扫描一遍源程序或中间代码程序并完成所规定的任务，生成新的中间代码或者目标程序
- 编译遍之间顺序执行，前一个遍的输出作为后一个遍的输入
- 为什么分遍?
  - 各遍功能独立单一，逻辑结构清晰
  - 不同的编译遍可以相对独立的开发
  - 通过遍管理器打开或关闭特定遍
  - 提高代码的可重用性，降低编译器开发的代价

2.2 编译遍

- 编译遍

  例 LLVM定义的遍

```
MODULE_ANALYSIS("callgraph", CallGraphAnalysis())
MODULE_ANALYSIS("lcg", LazyCallGraphAnalysis())
MODULE_ANALYSIS("module-summary", ModuleSummaryIndexAnalysis())
MODULE_ANALYSIS("no-op-module", NoOpModuleAnalysis())
MODULE_ANALYSIS("profile-summary", ProfileSummaryAnalysis())
MODULE_ANALYSIS("stack-safety", StackSafetyGlobalAnalysis())
MODULE_ANALYSIS("targetlibinfo", TargetLibraryAnalysis())
```

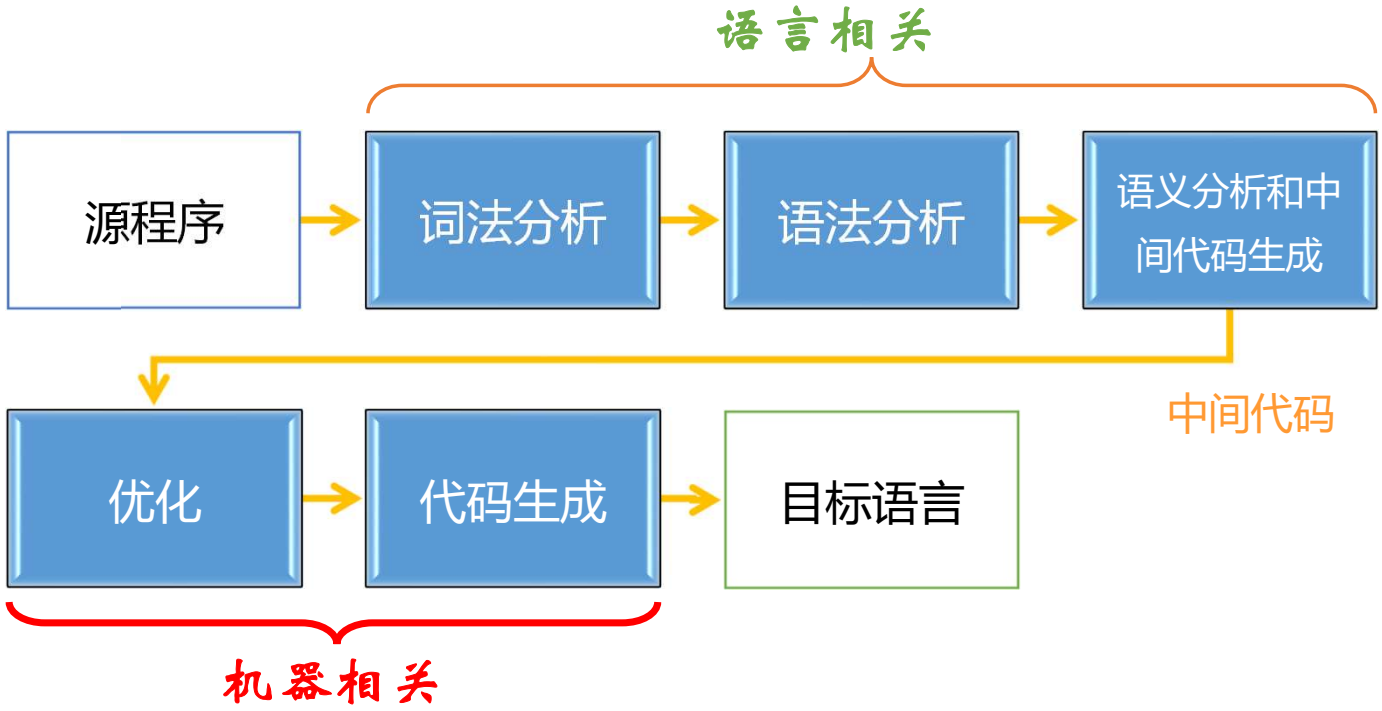
- 根据编译遍所完成的任务，可以将编译遍分成四类：分析遍、代码转换遍、优化遍及辅助遍

2.2 编译遍

- 分析遍
  - 收集信息，提供给其他遍使用，或用来调试、程序可视化等
  - 例如：控制流分析、数据流分析、构造必经节点树……
- 代码转换遍
  - 将代码从一种表示方式转换成另一种表示方式
  - 例如：源语言到编译器中间语言代码的转换……
- 优化遍
  - 对代码进行优化转换
  - 例如：常数传播、标量替换、循环展开……
- 辅助遍
  - 提供辅助功能，例如代码验证遍用于验证变换后的代码是否正确

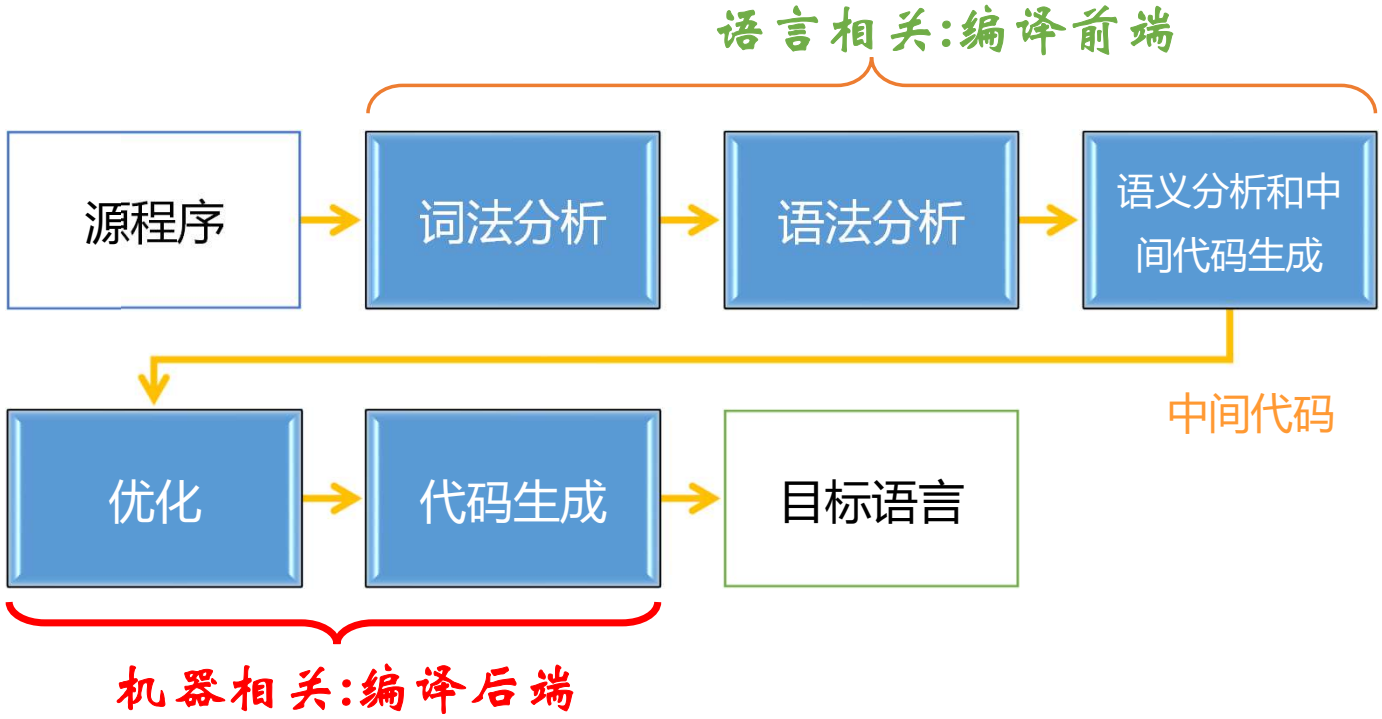
3 编译器结构

总体结构



3 编译器结构

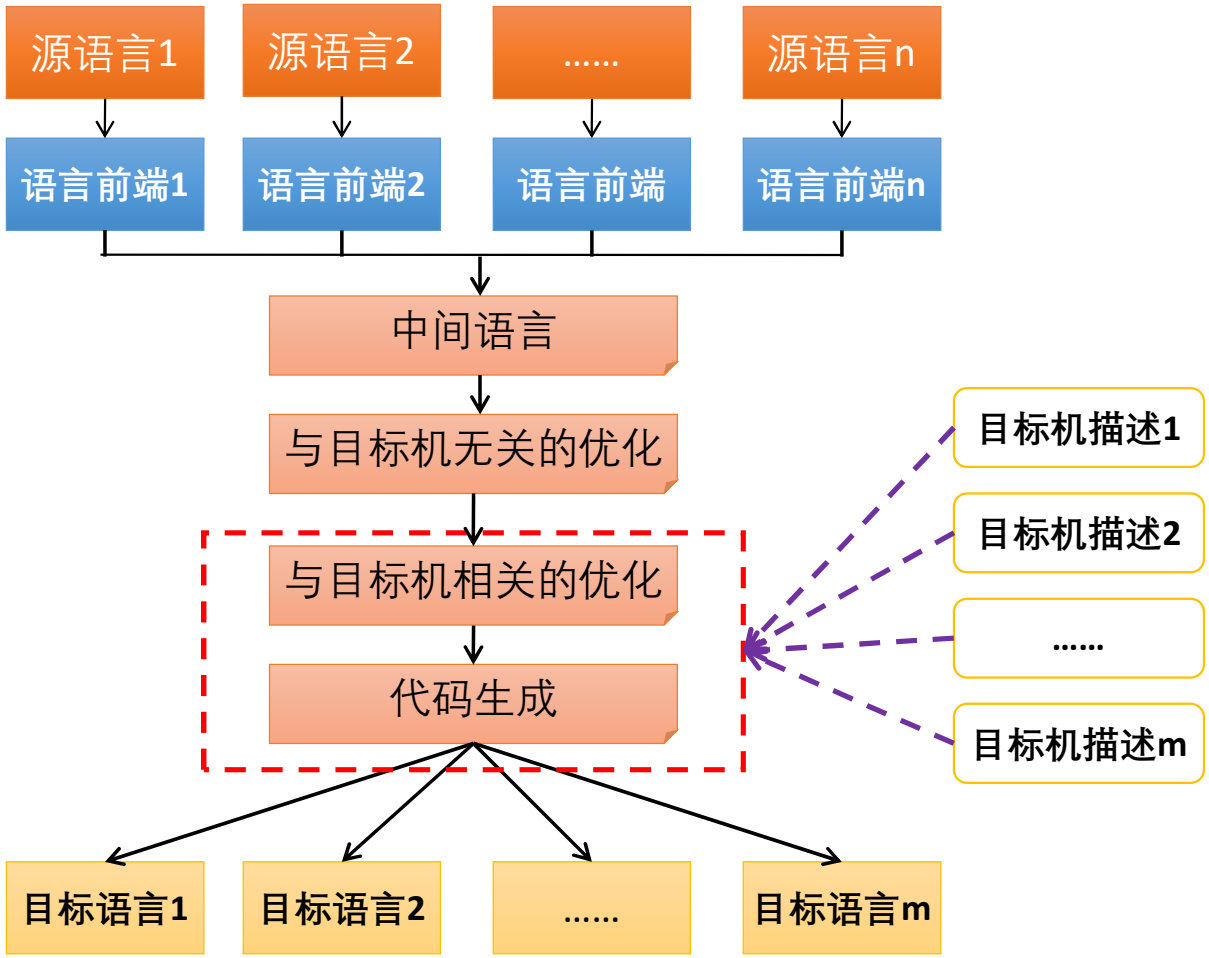
总体结构


语言相关:编译前端
源程序
词法分析
语法分析
语义分析和中间代码生成
中间代码
优化
代码生成
目标语言
机器相关:编译后端

3.1 多语言、多目标编译器

- 50%以上的代码在优化、代码生成阶段
- 并且，这部分代码大多数针对中间表示进行，与目标机无关
- 如何用n + m个模块生成m×n个编译器?
  - n种语言
  - m个目标机

3.1 多语言、多目标编译器



3.2 程序表示

- 程序表示
  - 编译过程中，编译器内部使用的、能够表示源代码和源代码相关信息的数据结构或代码。
- 关键的程序表示
  - 中间代码
- 中间代码
  - 又称中间表示或中间语言
  - 中间表示是一种含义明确、便于处理的记号系统，通常独立于硬件

3.2 程序表示

- 常用的中间表示
  - 后缀式
    - 逆波兰表示法
    - 表达式的操作数写在前面、操作符写在后面（后缀），例如a+b写成ab+
  - 三地址代码
    - x:=y op z
    - 三地址代码的具体表示方法包括：三元式、间接三元式、四元式等
  - 图表示
    - 抽象语法树
    - 很多内部数据结构采用了图的形式

3.2 程序表示

- 为什么使用中间代码
  - 有利于编译器重定向（语言和新的平台），支持多种编程语言和多种后端平台的编译器可以通过使用一种中间表示来实现
  - 便于进行与平台无关的优化
  - 编译程序本身结构更清晰、更模块化

3.3 编译遍顺序

- 编译器通过遍管理器管理编译遍
- 编译器在初始化过程将各遍按一定顺序注册到遍管理器中，遍管理器根据遍提供的信息及注册顺序来进行调度执行
- 编译遍需要向遍管理器提供的信息包括什么时候执行、如何执行、执行需要的条件等。

```
OptimizePM.addPass(LoopDistributePass());
OptimizePM.addPass(LoopVectorizePass());
OptimizePM.addPass(LoopLoadEliminationPass());
```

示例：LLVM 初始化遍顺序

3.3编译遍顺序

- 如何组织编译遍顺序?
  - 遍之间具有简单的依赖关系
    - 一个优化为另一个优化创造机会
    - 例如：Copy传播和死代码删除
  - 遍之间具有循环依赖关系
    - 常数折叠和常数传播
  - 编译之间互相反作用
    - 公用子表达式删除和寄存器分配
    - 寄存器分配和指令调度

4.1 什么是词法分析

- 词法分析
  - 对源程序进行扫描，将输入的字符流分割为最小的、有意义的单元 —— 词法单元（**tokens)**

```
C代码:    x3 = y + 3;
```

```
Fortran代码:     programtest
                    print *,'hi'
                 end
```

4.1 什么是词法分析

- 词法单元（Token）
  - <单元名, 属性值>
  - 单元名（Token name）
    - 标记词法单元种类的抽象符号
    - 例如：标识符，数字，标号，关键字等等
  - 属性值
    - 指向该token在符号表中的位置，或者该token的值

```
programtest
   print *,'hi'
end
```

```
<program>
<identify,test>
<identify,print>
<*>
<const,'hi'>
<end>
```

4.1 什么是词法分析

- 词法单元（Token）
  - <单元名, 属性值>
  - 单元名（Token name）
    - 标记词法单元种类的抽象符号
    - 例如：标识符，数字，标号，关键字等等
  - 属性值
    - 指向该token在符号表中的位置，或者该token的值

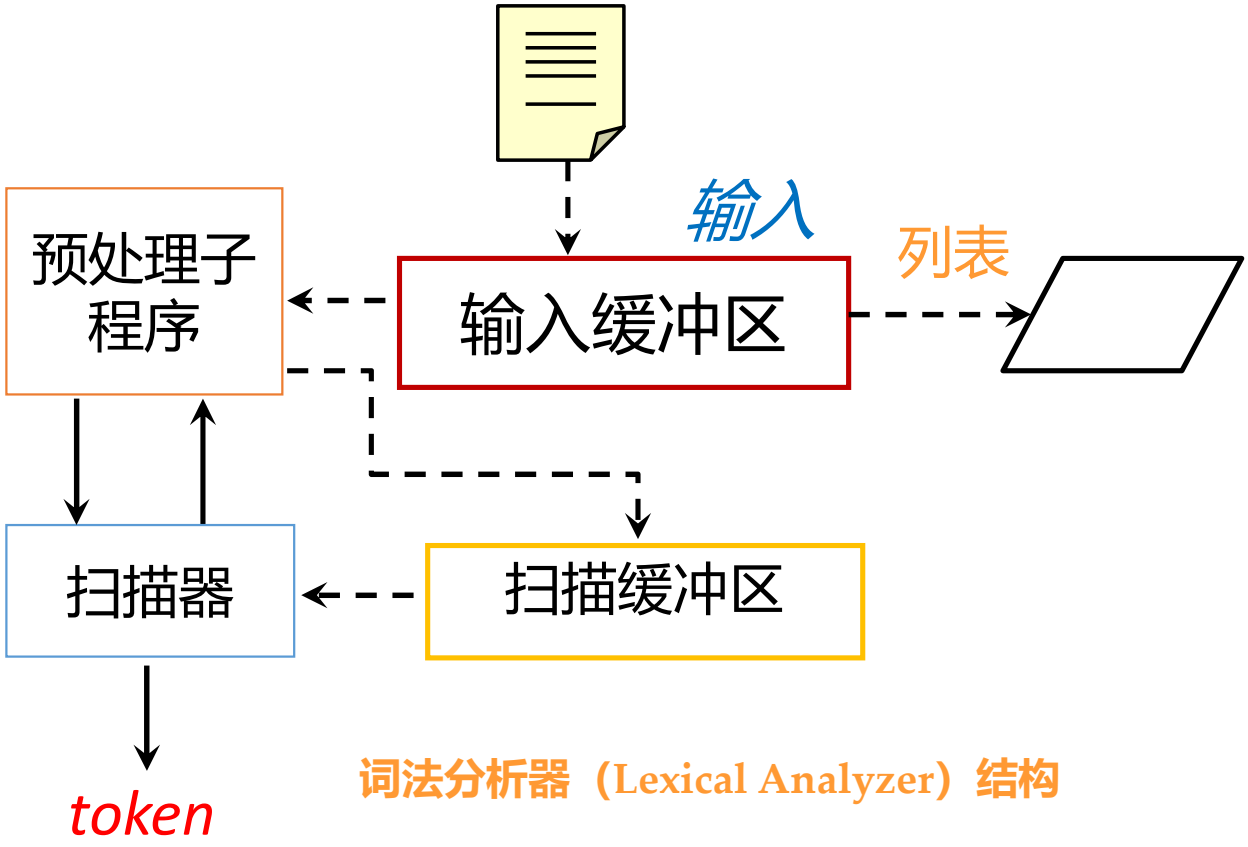
```
x3 = y + 3;
```

```
<identify, x3>
<=>
<identify, y>
<+>
<number,3>
<;>
```

4.2 词法分析器

- 词法分析器(Lexical Analyzer)
  - 又称扫描器(Scanner)：执行词法分析的程序
- 在词法分析阶段，也将错误信息和源程序的位置对应起来。

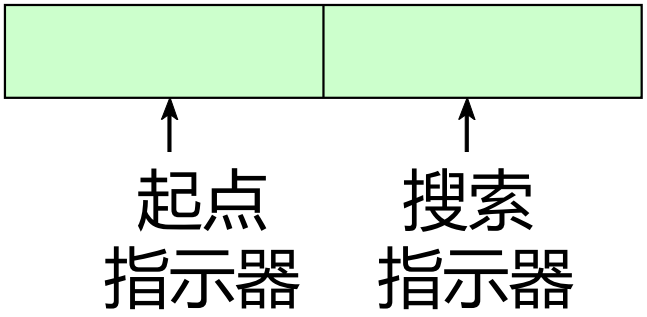
4.2 词法分析器



词法分析器（Lexical Analyzer）结构

4.2 词法分析器

- 输入串放在输入缓冲区中
- 预处理子程序：剔除无用的空白、跳格、回车和换行等编辑性字符;区分标号区、捻接续行和给出句末符等
- 扫描缓冲区



4.3 词法分析器的构建

- 如何为一个语言构建词法分析器?
  1、使用正规表达式***E***（*regular expressions*)描述语言的词法文法
  2、根据E构建一个确定有穷自动机（DFA）
  3、执行这个DFA判断输入字符串是否属于E描述的语言***L(E)***
- 可以使用***lex/flex/Antlr***等工具来构建DFA，也可以手工进行

4.3.1 正规表达式

- 大部分程序语言的词法文法（词法结构）都可以用正规表达式描述
- 每个正规表达式代表一个字符串集合

Symbol: *a*	符号*a*，正则表达式*a*表示仅包含字符串*a*.
Alternation: M|N	选择 M|N，字符串M或者N形成的正规表达式
Concatenation: M·N	联结：M·N, N跟在M后形成的正规表达式
Epsilon: *ε*	空串
Repetition: M*	重复，M重复0次或者多次形成的正规表达式

4.3.1 正规表达式

- 数字常数的正规表达式

```
digit → [0-9]
digits → digit digit*
optionalFraction →.digits|ε
optionalExponent →([eE](+|-|ε) digits )|ε
number → digits optionalFraction optionalExponent
```

```
digit → [0-9]
digits → digit+
number → digits(.digits )?( [eE] (+|-)?digits )?
```

4.3.2 确定有限状态自动机

定义：非确定有限状态自动机 (NFA) A = {S, Σ, s_0, F, f}：

1.有限状态集合S

2.输入符号集合Σ

3.初始状态$S_0 \in S$

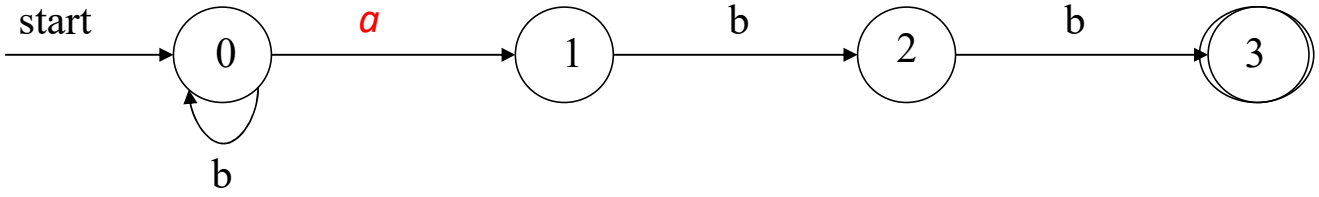
4.终态集合$F \subseteq S$

5.状态转换函数f，表示当前状态S_i，当输入符号a时，转换成下一状态S_j

如果S_j唯一，我们称该有限自动机为确定有限自动机.

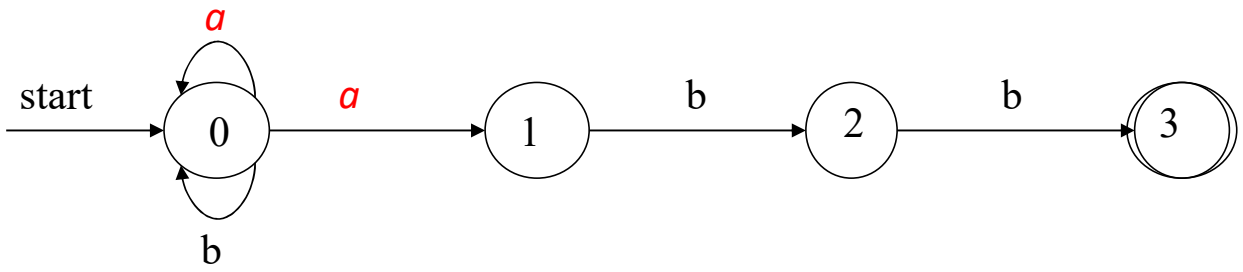
4.3.2 确定有限状态自动机

确定有限状态自动机(DFA):



b*abb

非确定有限状态自动机(DFA):



(a|b)*abb

4.3.2 确定有限状态自动机

- 确定有限状态自动机(DFA)
  - 在当前状态+输入下，转换后状态唯一
- 非确定有限状态自动机(NFA)
  - 转换状态不唯一
  - 对于某些输入，到达最终状态
  - 选择错误时，可能导致正确的词法单元不能接受，需要回退
- DFA没有回溯，速度更快

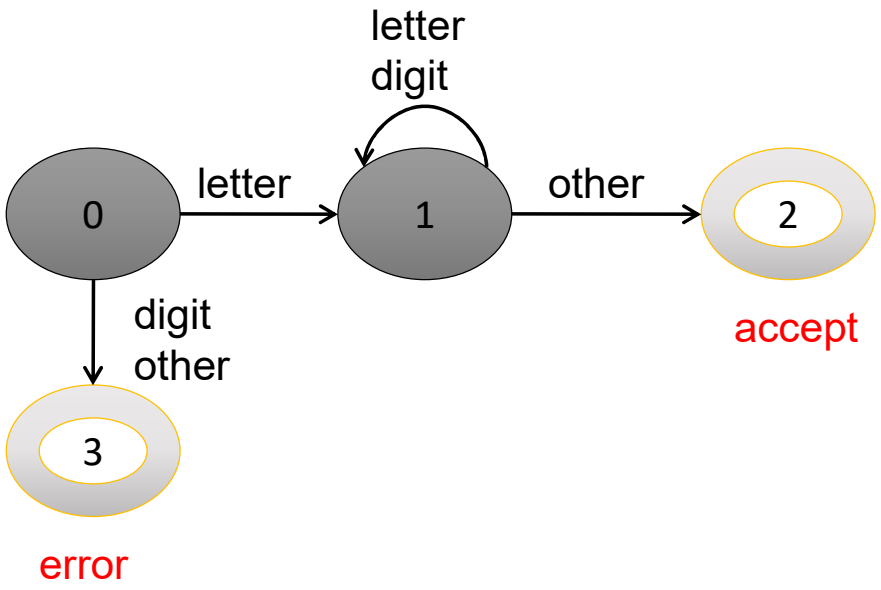
4.3.2 确定有限状态自动机

- 正规表达式只包含几种操作，如选择、联结等，很容易用NFA表示
- 从NFA转换为等价的DFA也有预定的过程

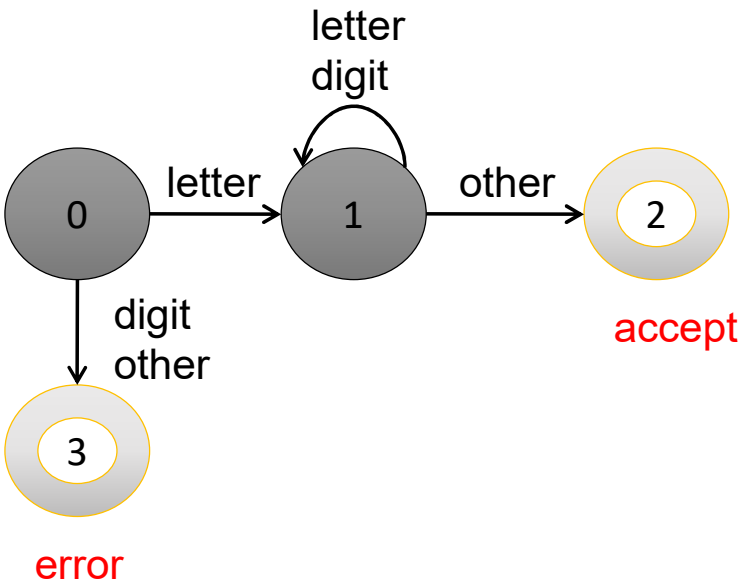
1.4 词法分析器示例

- 标识符是以字母打头，后面跟0个或若干个字母或者数字

```
letter →(a|b|c|…|z|A|B|C|…|Z)
digit →(0|1|2|3|4|5|6|7|8|9)
id →letter(letter|digit)*
```



4.4 词法分析器示例



letter →(a|b|c|…|z|A|B|C|…|Z)
digit →(0|1|2|3|4|5|6|7|8|9)
id →letter(letter|digit)*

char_class:

	A-Z	a-z	0-9	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

4.4 词法分析器示例

```
char  ← next_char();
state ← 0;
done ← false;
token_value ←"";
while (!done){
  class ←char_class[char];
  state ←next_state[class,state];
  switch(state) {
     case 1:
        char ←next_char();
        token_value←token_value|char;
        break;
     case 2:
        token=id;
        done = true;
        break;
     case 3
        done = true;
        token= error;
        break;
    }
}
return token;
```

letter →(a|b|c|…|z|A|B|C|…|Z)
digit →(0|1|2|3|4|5|6|7|8|9)
id →letter(letter|digit)*

char_class:

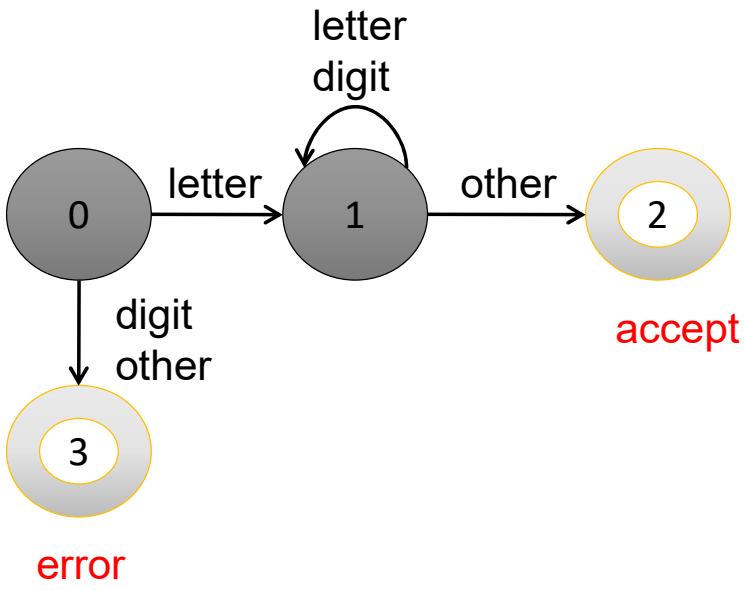
	A-Z	a-z	0-9	other
value	letter	letter	digit	other

next_state:

class	0	1	2	3
letter	1	1	—	—
digit	3	1	—	—
other	3	2	—	—

4.4 词法分析器示例

letter →(a|b|c|…|z|A|B|C|…|Z)
digit →(0|1|2|3|4|5|6|7|8|9)
id →letter(letter|digit)*



lex.l

```
%option warn noyywrap

letter  [A-Za-z]
digit   [0-9]
id      {letter}({letter}|{digit})*
%%

{id}    {printf("id = %s\n", yytext);}
{digit} {printf("num = %s\n", yytext);}

%%
```

5.1 语法分析

- 又称为解析，验证词法分析得到的词法单元流能够由该语言的文法生成

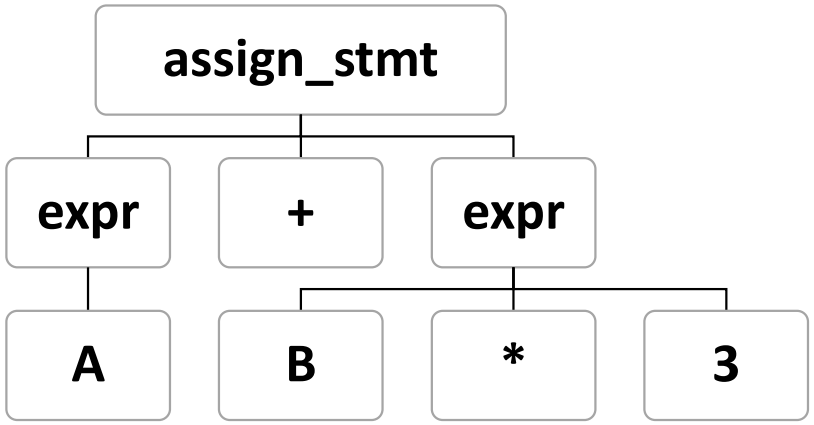
```
A = A + B * 3;
Token:
    <id, A>
    <=>
    <+>
    <id,B>
    <*>
    <num, 3>
```

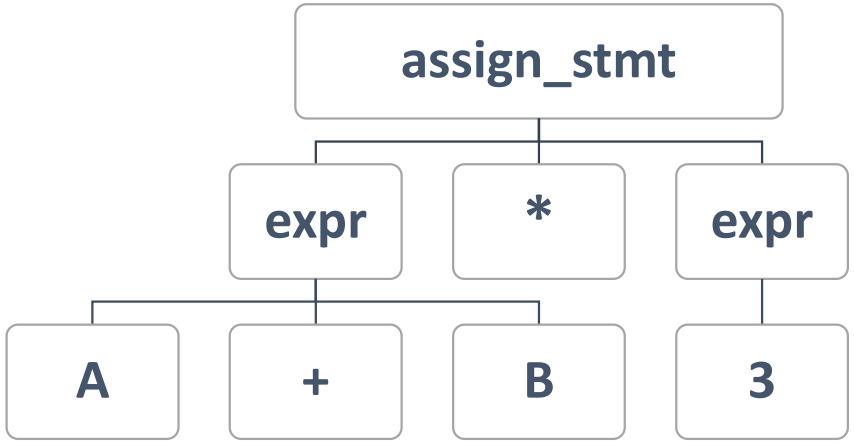
赋值语句的文法：

```
assign_stmt → lvalue '=' rvalue
lvalue → id
rvalue → expr
expr → expr '+' expr
     | expr'*' expt
     | id
     | num
```

5.1 语法分析

A = A + B * 3;





具有二义性的文法

- 结合属性
- 优先级

5.2 语法分析器

- 解析器，分析这些词法单元，看他们是否能够组织成为符合语言规范的语句
- 并且生成语法分析树给后续阶段使用（可选的）
- 要进行语法分析，必须对语言的语法结构进行描述。
  - 采用正规式和有限自动机可以描述和识别语言的单词符号
  - 用上下文无关文法来描述语法规则

5.3 上下文无关文法

- 上下文无关文法 G=(V_t,V_n,S,P)，其中
  - V_t 终结符(*tokens*)集合(非空)
  - V_n, 非终结符集合(非空)，且$V_T \cap V_N = \varnothing$
  - S 文法的开始符号，$S \in V_N$
  - P 产生式集合(有限)，每个产生式形式为

    $P \rightarrow \alpha$， $P \in V_N$， $\alpha \in (V_T \cup V_N)^*$
- 开始符S至少必须在某个产生式的左部出现一次

表达式的文法：

```
expr → expr '+' expr
     | expr'*' expt
     | id
     | num
```

5.3 上下文无关文法

- 定义：称$\alpha A\beta$直接推出$\alpha\gamma\beta$，即 $\alpha A\beta \Rightarrow \alpha\gamma\beta$

  仅当$A \to \gamma$是一个产生式， 且α， $\beta \in (V_T \cup V_N)^*$ 。

- 如果$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$，则我们称这个序列是从$\alpha_1$到$\alpha_n$的一个推导。若存在一个从$\alpha_1$到$\alpha_n$的推导，则称$\alpha_1$可以推导出$\alpha_n$ 。

例如：对表达式文法

expr ⇒ (expr+expr)⇒ (id+expr)⇒ (id+id) ⇒A+B

5.3 上下文无关文法

- 定义

用 $\alpha_1 \overset{+}{\Rightarrow} \alpha_n$ 表示：从α_1出发，经过一步或若干步，可以推出α_n。

用 $\alpha_1 \overset{*}{\Rightarrow} \alpha_n$ 表示：从α_1出发，经过0步或若干步，可以推出α_n。

- 定义：假定G是一个文法，S 是它的开始符号。如果 $S \overset{*}{\Rightarrow} \alpha$，则$\alpha$称是一个句型。仅含终结符号的句型是一个句子。文法G所产生的句子的全体是一个语言，将它记为 L(G)。

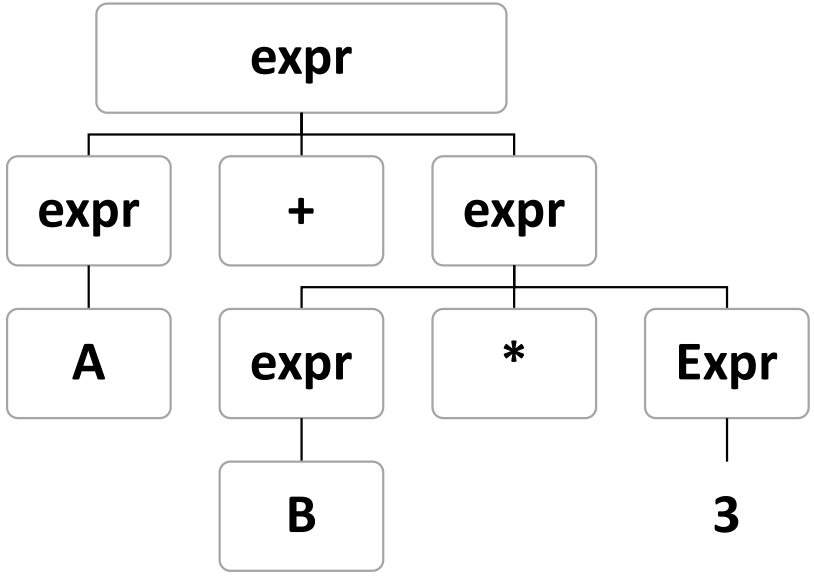
5.3 上下文无关文法

语法分析问题：为输入的token流，寻找产生式组成的导出序列。

A + B * 3;

表达式的文法：

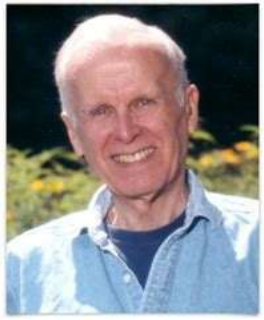
expr → expr '+' expr
| expr'*' expt
| id
| num



5.4 EBNF

语法分析第一步：使用上下文无关文法定义语言的语法规则

- BNF（Backus–Naur Form）



Dr. John Backus

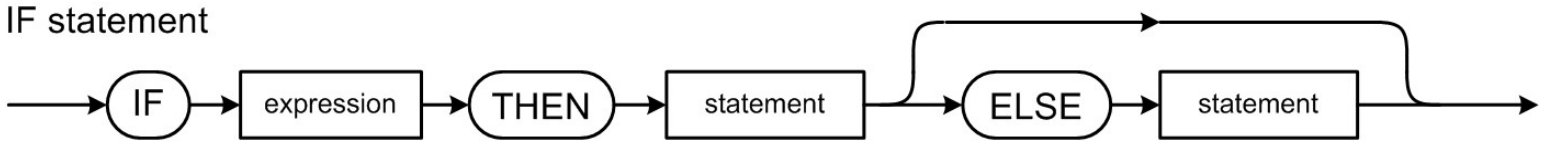
描述语言的元语言

Dr. Peter Naur

5.4 EBNF

- BNF（Backus–Naur Form）
  - 元符号（Meta-Symbol）

::=	定义为
|	或
< >	非终结符




```
<if statement> ::= IF <expression> THEN <statement>
                 | IF <expression> THEN <statement> ELSE <statement>
```

5.4 EBNF

- BNF（Backus–Naur Form）
  - 支持递归

```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
```

右递归

```
<digit sequence> ::= <digit>
                   | <digit sequence> <digit>
```

左递归

5.4 EBNF

- EBNF（Extended Backus–Naur Form）
  - 新的元符号

{ }	括号内项重复0零或者多次
[ ]	可选项

Dr. Niklaus Emil

标准 ISO-14977定义了最常用的EBNF变体

5.4 EBNF

- EBNF（Extended Backus–Naur Form）
  - BNF

```
<digit sequence> ::= <digit>
                   | <digit> <digit sequence>
```

  - EBNF

```
<digit sequence> ::= <digit> { <digit> }
```

5.4 EBNF

- EBNF（Extended Backus–Naur Form）
  - BNF

```
<if statement> ::= IF <expression> THEN <statement>
                 | IF <expression> THEN <statement>
                                  ELSE <statement>
```

  - EBNF

```
<if statement> ::= IF <expression> THEN <statement>
                                  [ ELSE <statement> ]
```

5.4 EBNF

- SysY 语言
  - C 语言的一个子集
  - 练习：

SysY 语言中数值常量可以是整型数 IntConst，用BNF或者EBNF定义

(注意：整型数可以是八进制、十进制、十六进制)

5.4 构建语法分析器

- 使用上下文无关文法定义语言的语法规则
- 选择分析器类型
  - 通用的方法，可以对任意文法进行分析
  - 自顶向下
  - 自底向上
- 自顶向下
  - 从语法分析树的根开始，按照深度优先遍历分析树，也可以看作是寻找输入串的最左推导，每次选择最左的非终结符进行推导
- 自底向上
  - 从叶子开始，可以看作是一个最右推导的逆过程

5.4.1 自顶向下的分析

- 从分析树的根开始（标记为开始符号S），重复下面步骤，直到分析成功或错误：
  1. 对一个非终结符A，选择一个产生式A→α，构建α的每个符号的子结点
  2. 当一个终结符加入到分析树，不能匹配输入串，回溯
  3. 寻找下一个非终结符结点，进行推导

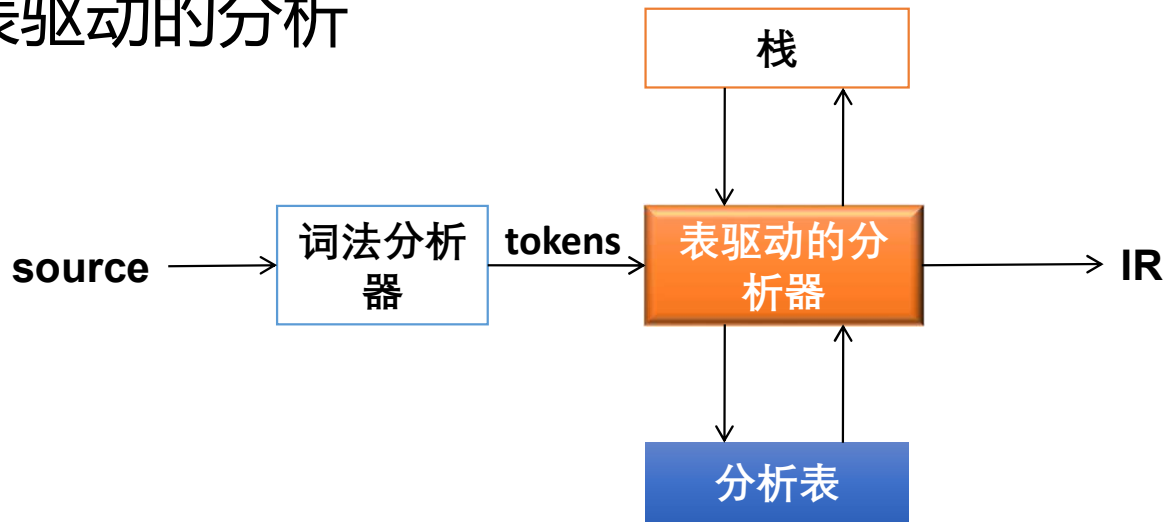
关键在于步骤1，如何选择产生式。产生式的选择通常采用“**超前看**”（Look-ahead）的方法。

5.4.1 自顶向下的分析

- 自顶向下分析按照从左至右顺序，每次选择最左的非终结符进行推导，称为LL分析（Left to right， Leftmost-derivation）
- LL(K)分析，为了选择产生式，分析器会超前看k个输入。
- LL(1)分析
  - 超前看一个符号
  - 不能处理具有左递归和具有二义性的文法外

5.4.1 自顶向下的分析

- 实现：表驱动的分析




```
E  → TE′
E′ → +TE′ | ε
T  → FT′
T′ → *FT′ | ε
F  → ( E ) | id
```

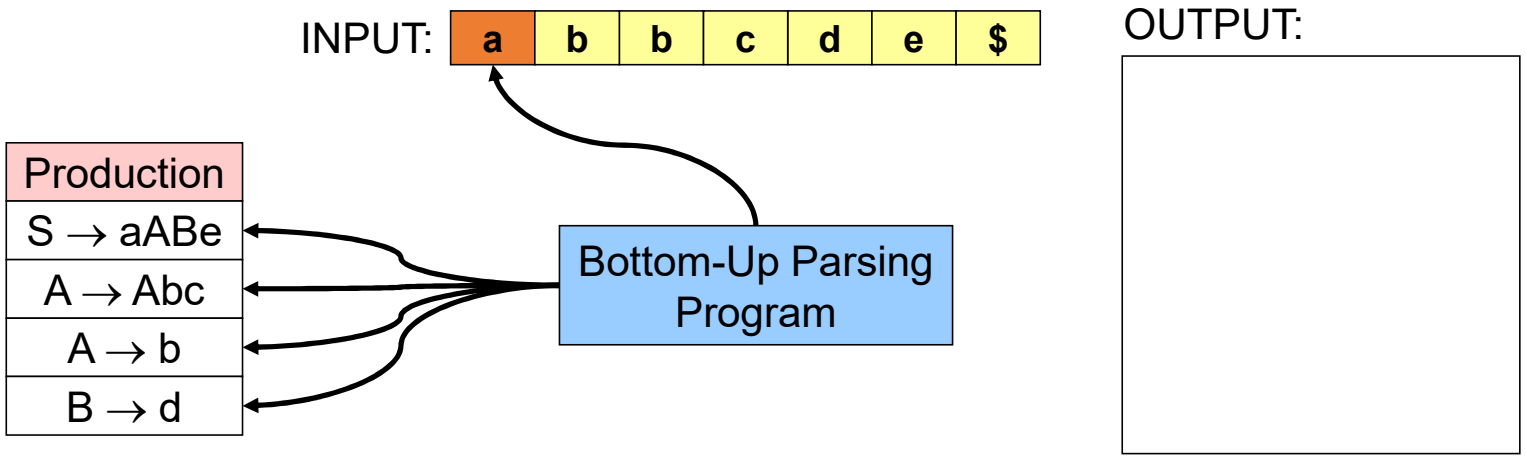
V_n	INPUT SYMBOL					
	id	+	*	(	)	$
E	E → TE′			E → TE′		
E′		E′ → +TE′			E′ → ε	E′ → ε
T	T → FT′			T → FT′		
T′		T′→ ε	T′ → *FT′		T′ → ε	T′ → ε
F	F → id			F → (E)		

5.4.2 自底向上分析

- 也称为移位-归约分析
  - 移位: 将下一个符号移入栈顶
  - 归约: 如果栈顶连续字符串和某个产生式匹配，则使用产生式左部的非终结符替换字符串
- 关键：何时进行归约，以及使用什么产生式归约

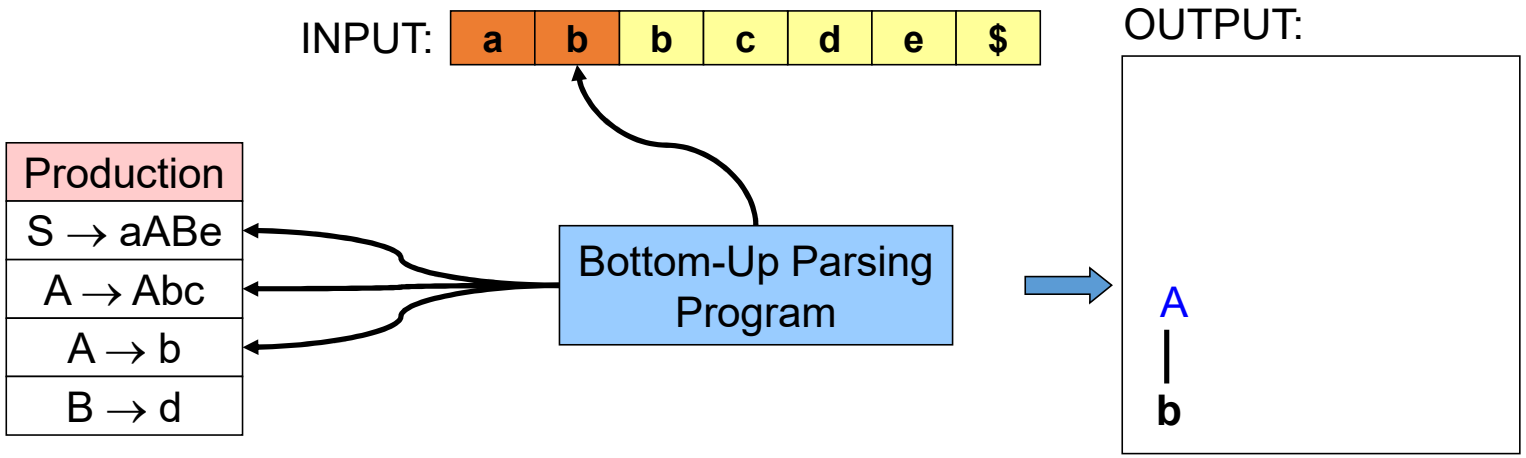
5.4.2自底向上分析

- shift-reduce parser



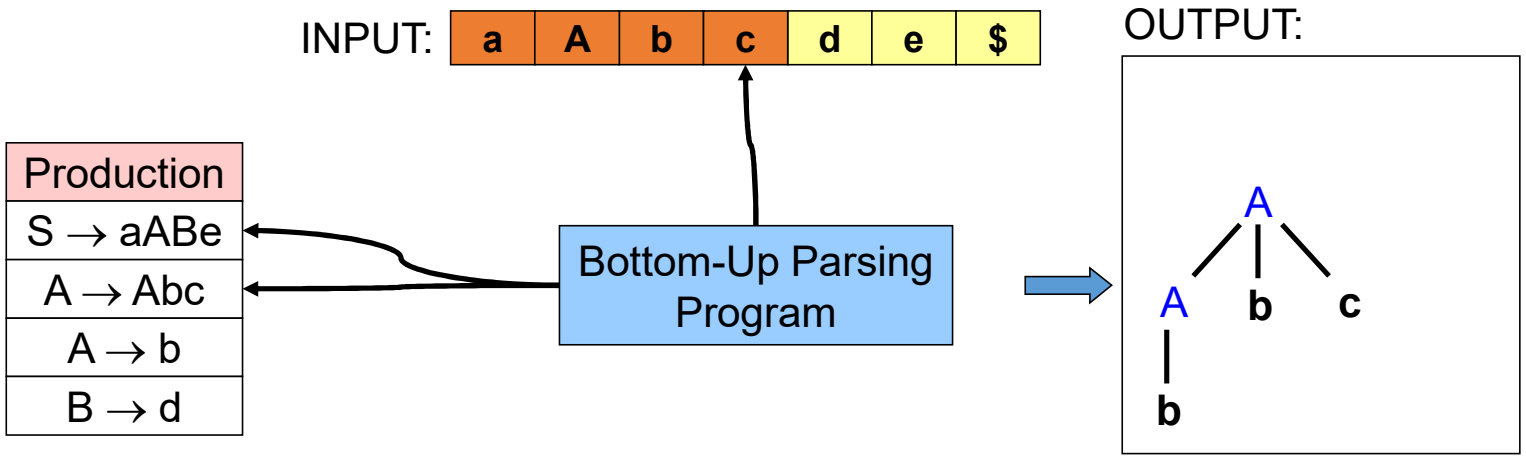
5.4.2自底向上分析

- shift-reduce parser

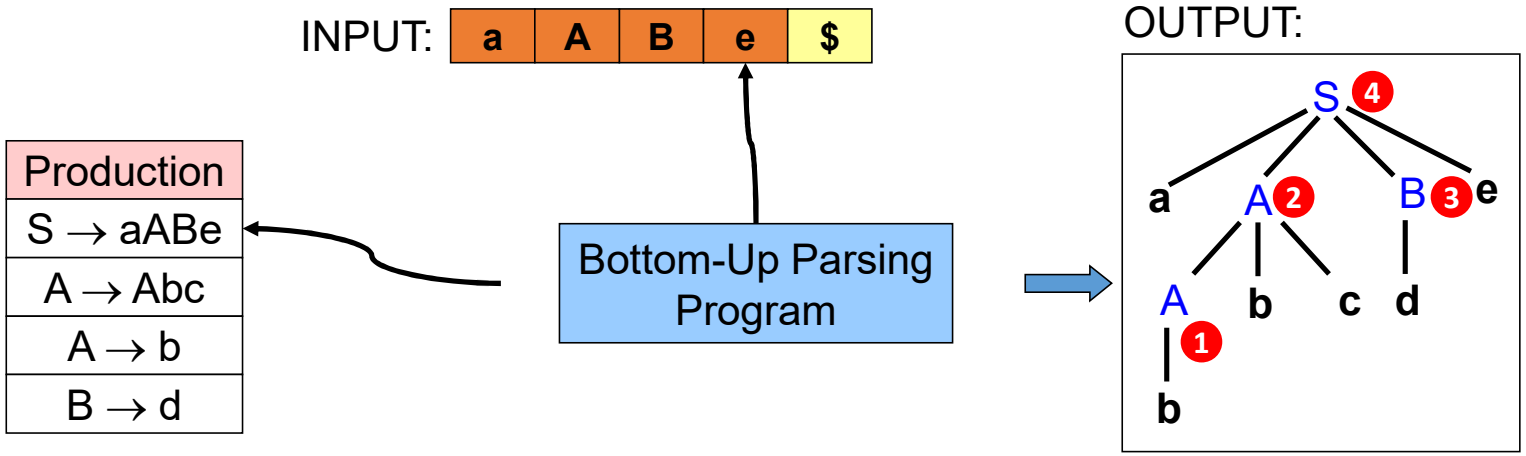


5.4.2 自底向上分析

- shift-reduce parser



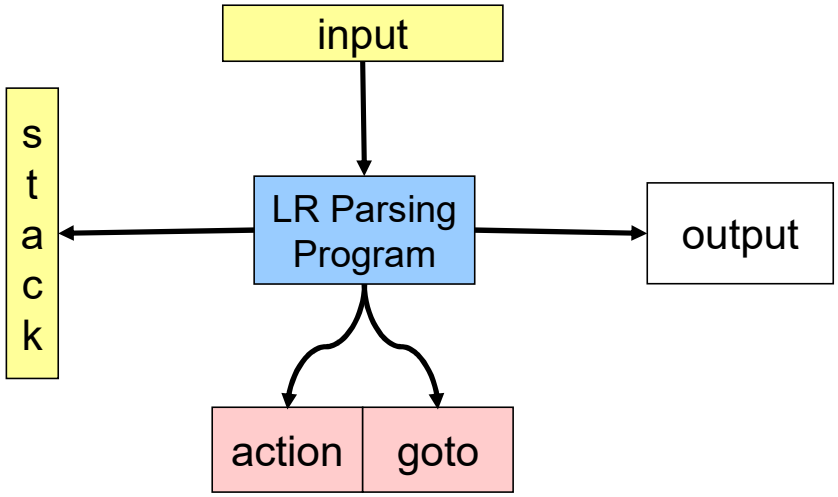
5.4.2 自底向上分析



- 最右推导的逆过程
  - LR

5.4.2自底向上分析

- LR(*k*)实现
  - Stack
  - Input
- 分析表
  - action[s_i, *token*]
    - 移位
    - 归约 A→β
    - 接受
    - 错误
  - goto[s_i,A] = s_j
    - 从状态i和非终结符A转到状态j



LR(1)

INPUT:

id	*	id	+	id	$

(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

总结

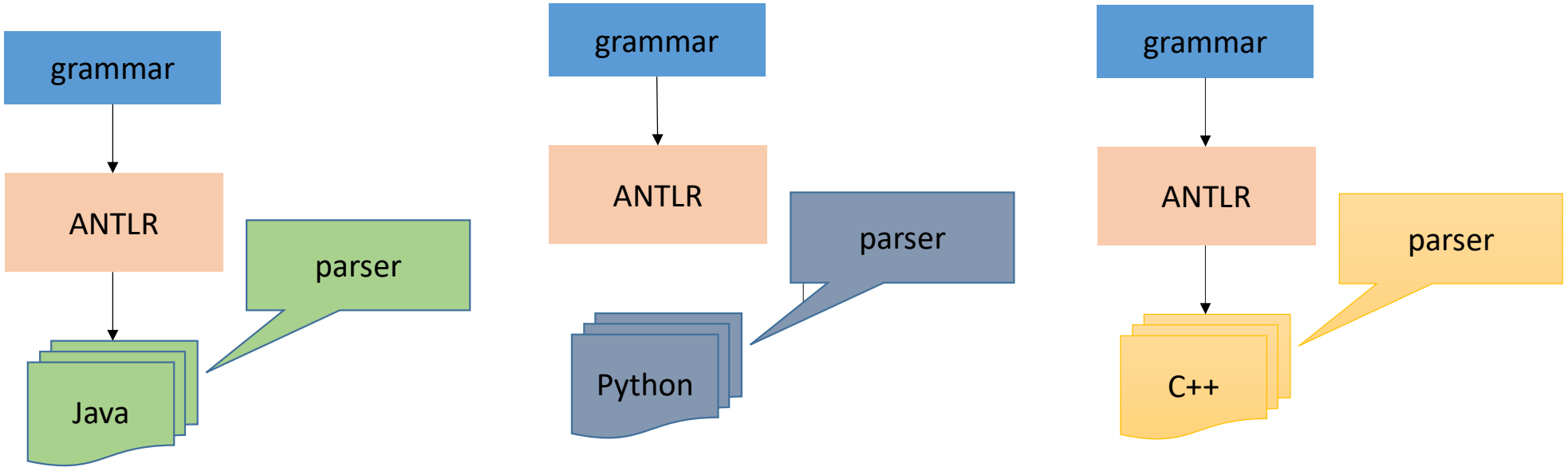
- 使用上下文无关文法定义语言的语法规则
- 选择解析器类型

Top-down	Bottom-up
手工编写	手工编写 解析器产生器
能分析的文法受到限制	处理的文法范围广
LL(*k*)	LR(*k*)

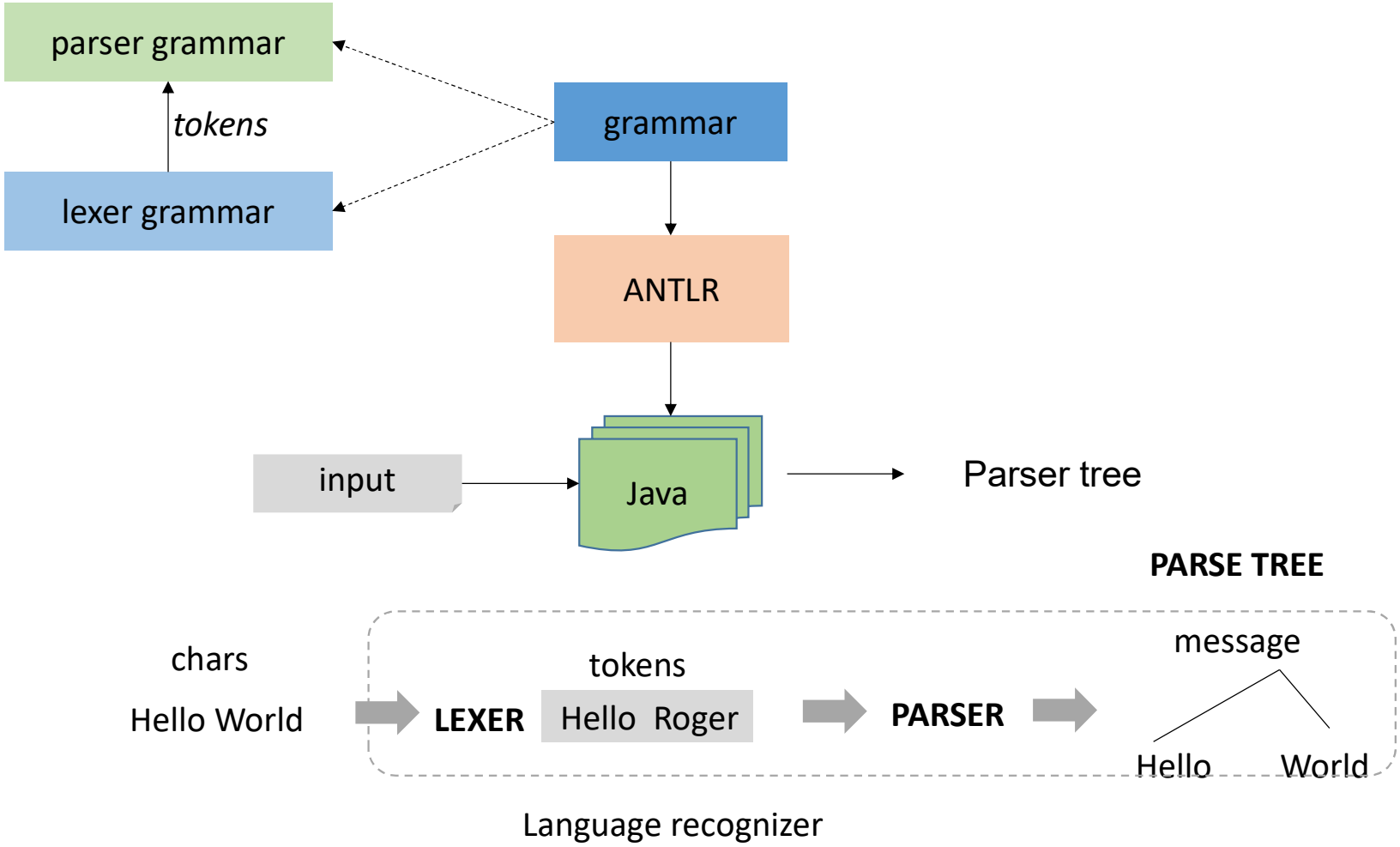
- 构建解析器
  - 手工编写代码：表驱动的语法分析
  - 使用解析器产生器

5.5 ANTLR

- ANTLR（ANother Tool for Language Recognition）
  - Parser Generator：ANTLR is a tool (program) that converts a grammar into a parser



5.5 ANTLR


parser grammar
tokens
lexer grammar
grammar
ANTLR
input
Java
Parser tree
PARSE TREE
chars
Hello World
tokens
LEXER
Hello Roger
PARSER
message
Hello
World
Language recognizer

5.5 ANTLR

- ANTLR可以一站式的解决词法与语法解析器的生成
  - Flex/yacc
- ANTLR通过在文法文件中的设置，可以生成多个语言代码

```
options {language=Cpp;}
options {language=CSharp;}
options {language=Java;}
options {language=Python3;}
```

- ANTLR可以生成语法解析树的图形化表示，方便开发与测试

计划

- 寒假
  - 熟悉Sys2022语言规范
  - 利用ANTLR实现语言的语法前端（词法分析、语法分析），完成C语言程序→AST
- 开学
  - 中间表示
  - 静态单一赋值SSA
  - 从AST→IR