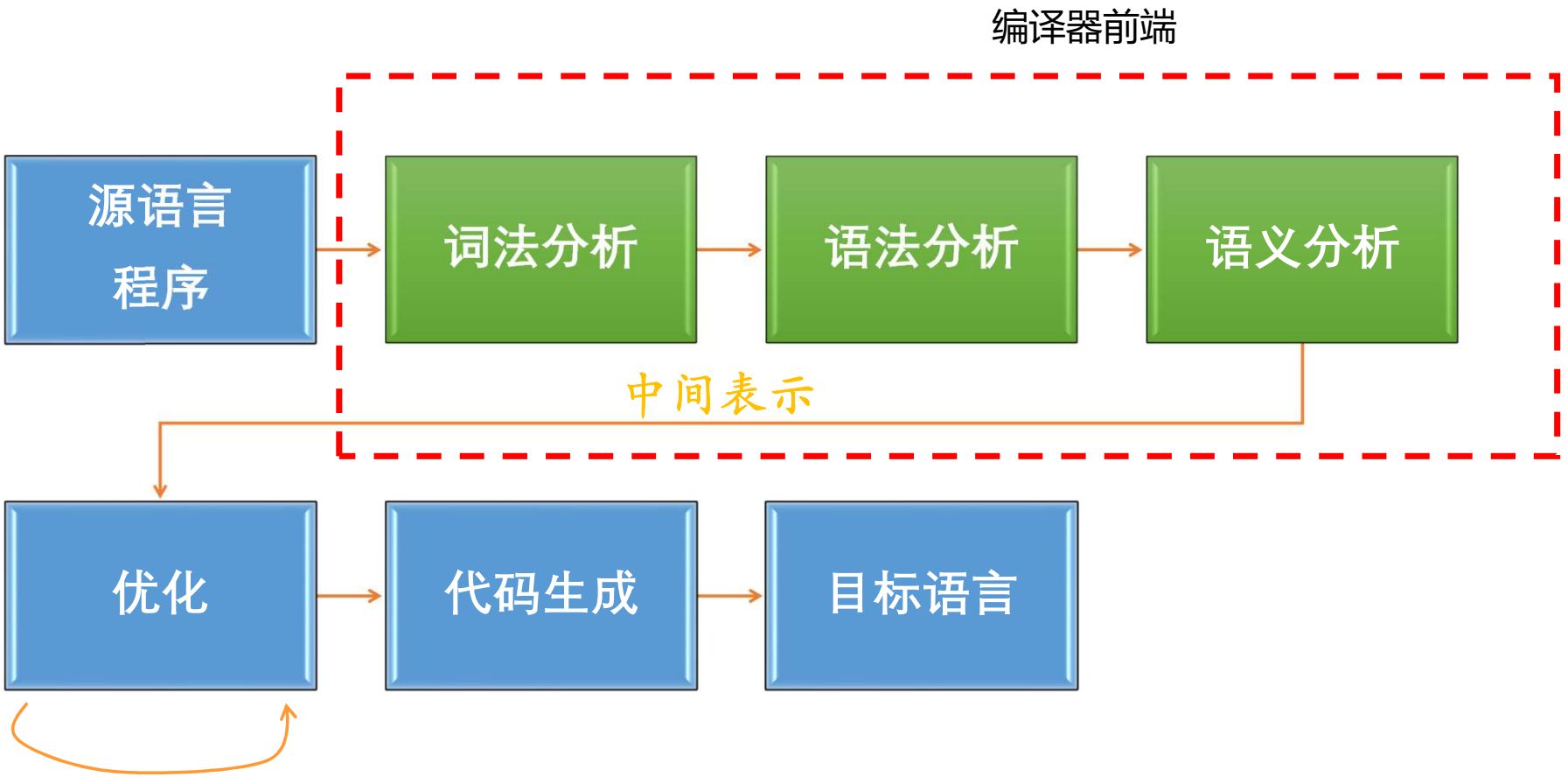
并行编译与优化
Parallel Compiler & Optimization

计算机研究所编译系统室

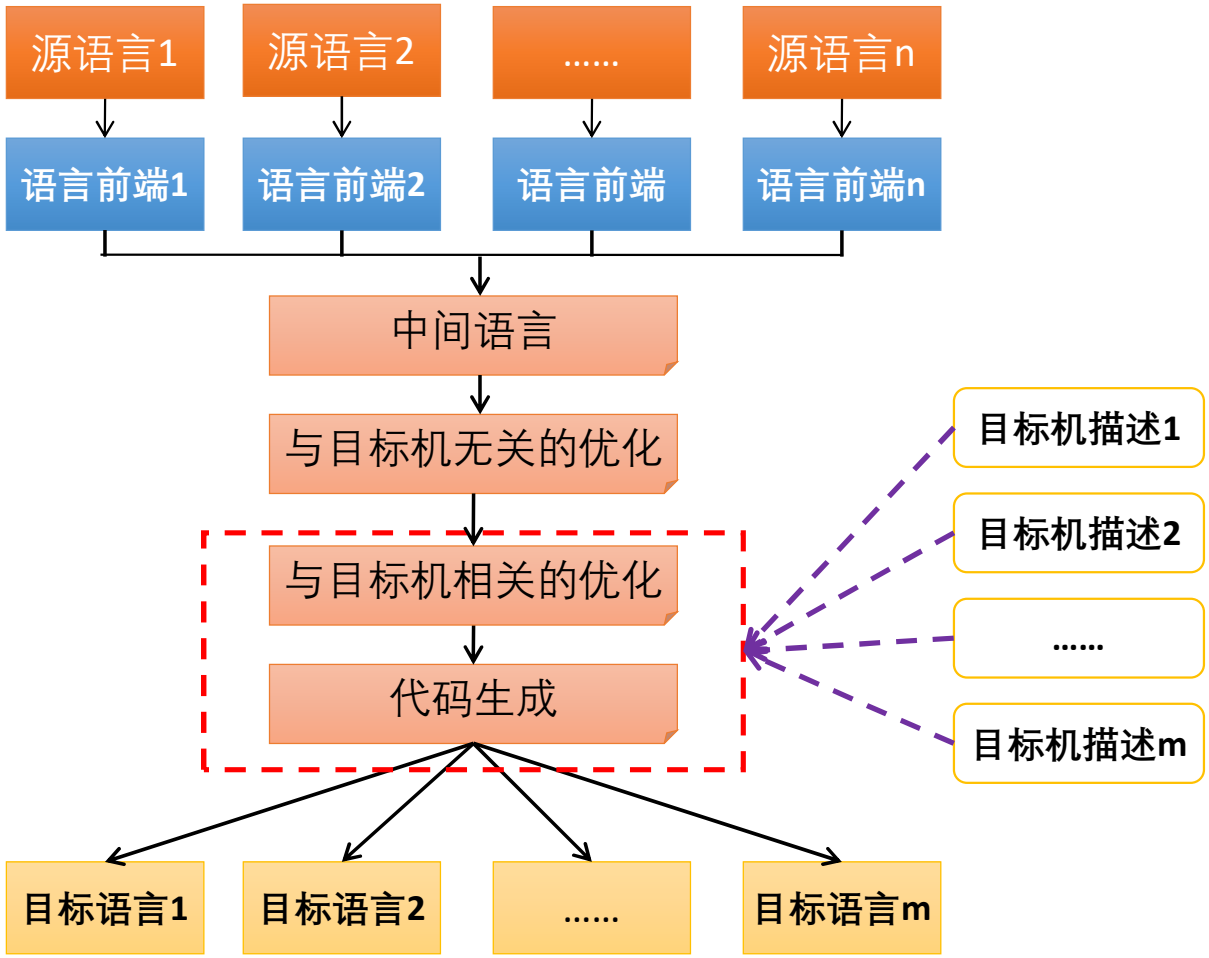
Lecture Two Compiler Front Lexical Analysis and Syntax Analysis

第二课 编译器前端：词法分析和语法分析

复习：编译器结构

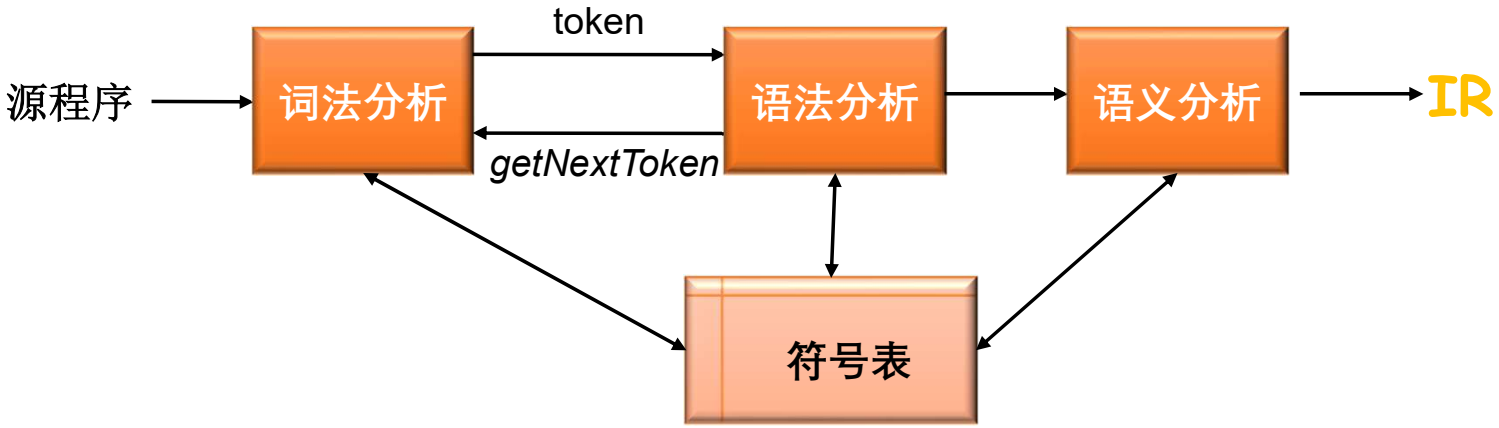


复习:多语言、多目标编译器



复习：编译器前端

- 前端
  - 扫描程序，识别合法程序
  - 给出恰当的警告/错误信息
  - 生成中间表示代码（IR）



为什么分离词法分析和语法分析

COMP
CCRG Open MP

- 功能单一，聚焦重点
  - 词法分析过滤一些无关的细节（注释、空格、包含的文件等）
  - 语法分析以词法分析的输出token为输入，更容易设计
- 高效
  - 词法分析简单、快速
- 可移植

内容

1. 词法分析
   - 正则表达式
   - 确定有限状态机
   - 构建词法分析器

2. 语法分析
   - 上下文无关文法
   - 自顶向下分析
   - 自底向上分析

1.1 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
```

1.1 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>
```

1.1 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

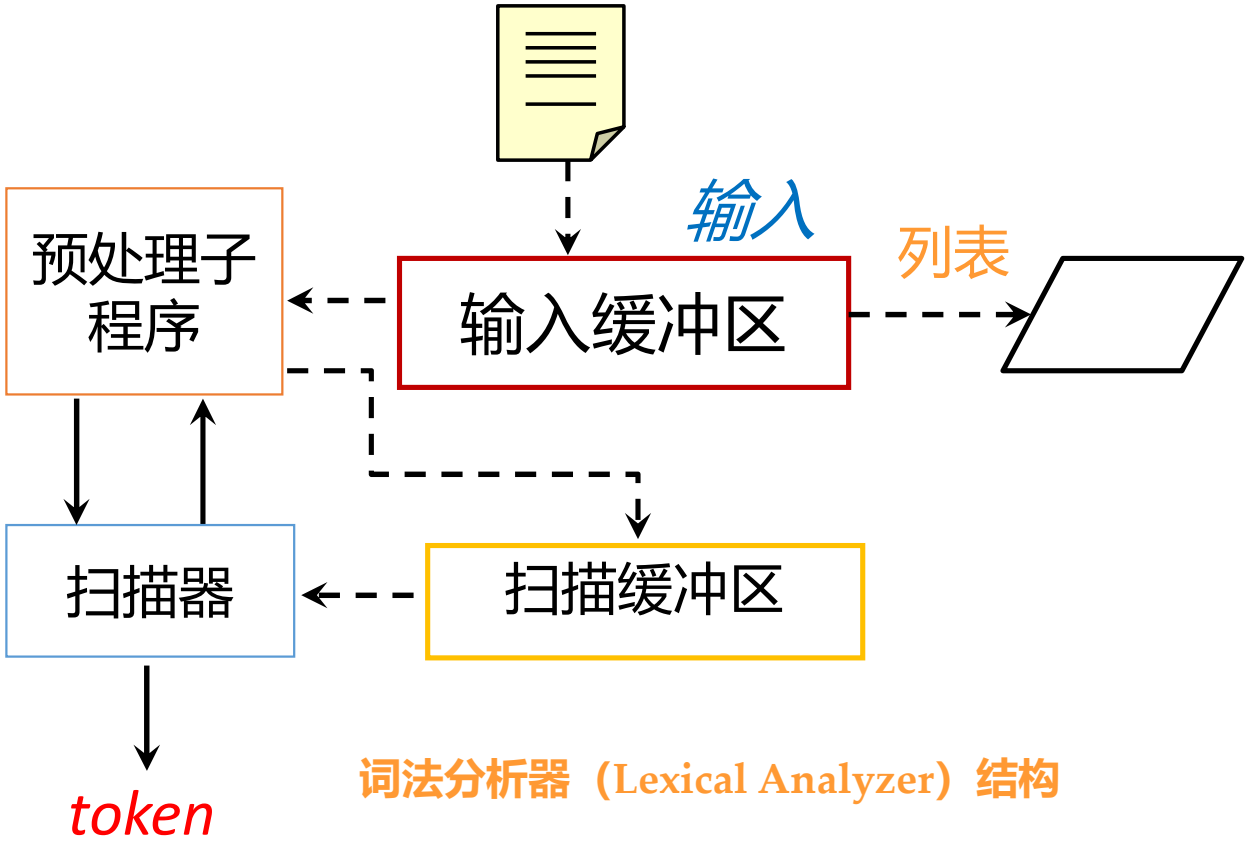
```
x3 = y + 3;
```

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

1.2 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。

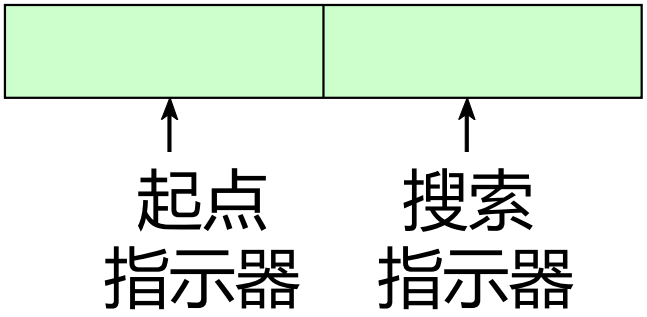
1.2 词法分析器



词法分析器（Lexical Analyzer）结构

1.2 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



1.3 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

1.3.1 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

1.3.1 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 )?
```

1.3.2 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$S_0 \in S$

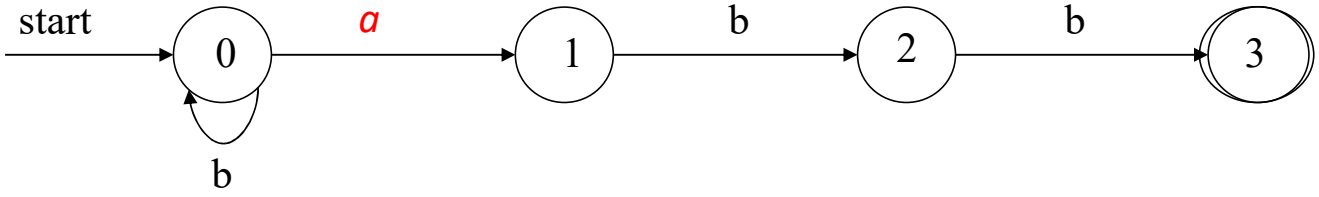
4.终态集合$F \subseteq S$

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.

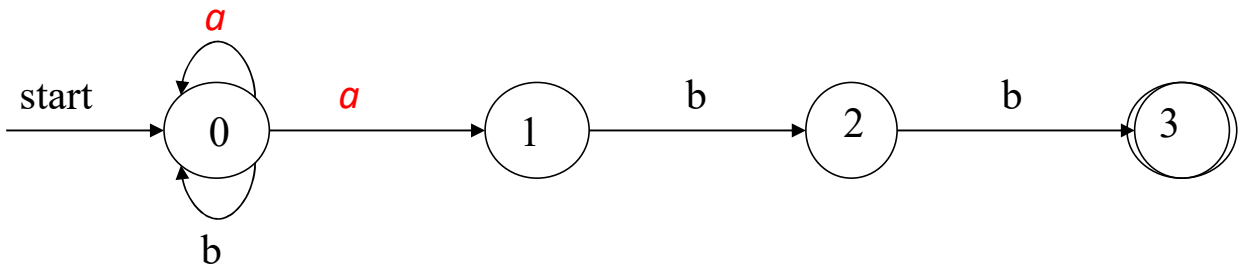
1.3.2 确定有限状态自动机

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

1.3.2 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

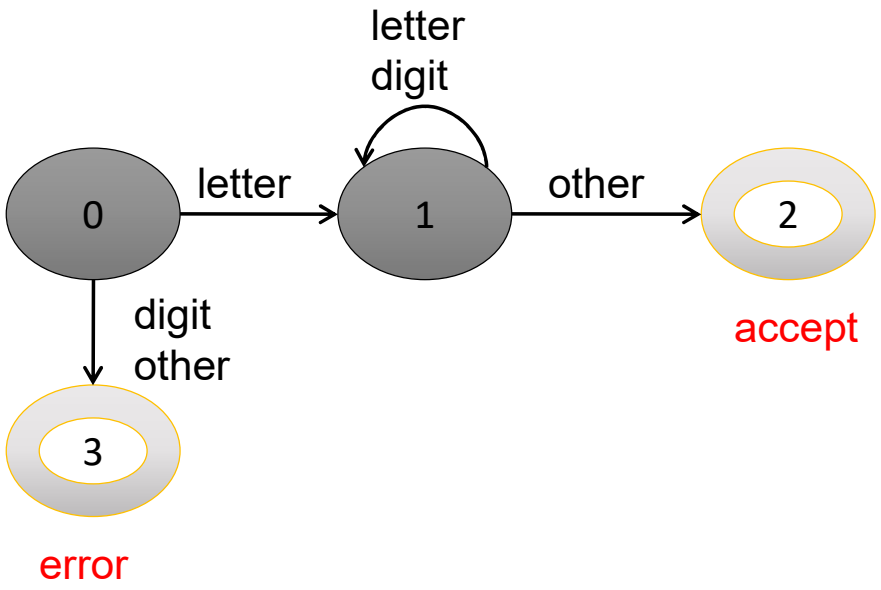
1.3.2 确定有限状态自动机

- 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

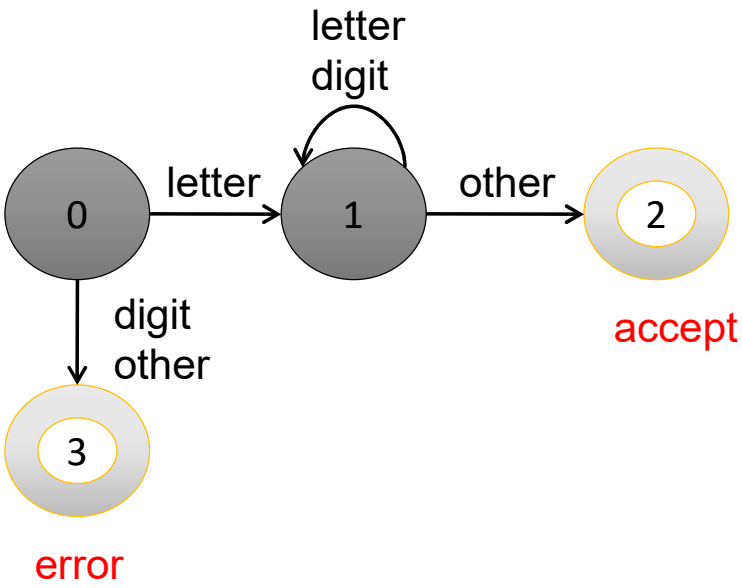
1.4 词法分析器示例

- 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)*
```



1.4 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	—	—

1.4 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:

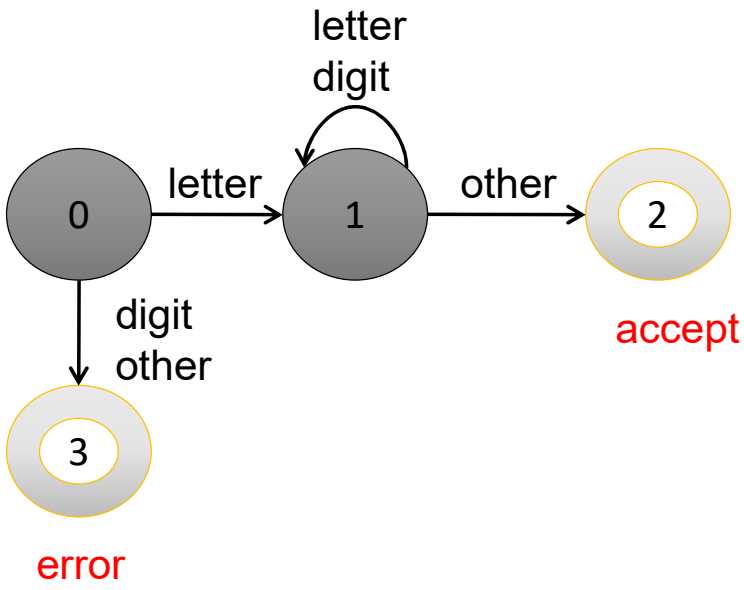
	A-Z	a-z	0-9	other
value	letter	letter	digit	other

next_state:

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

1.4 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)*

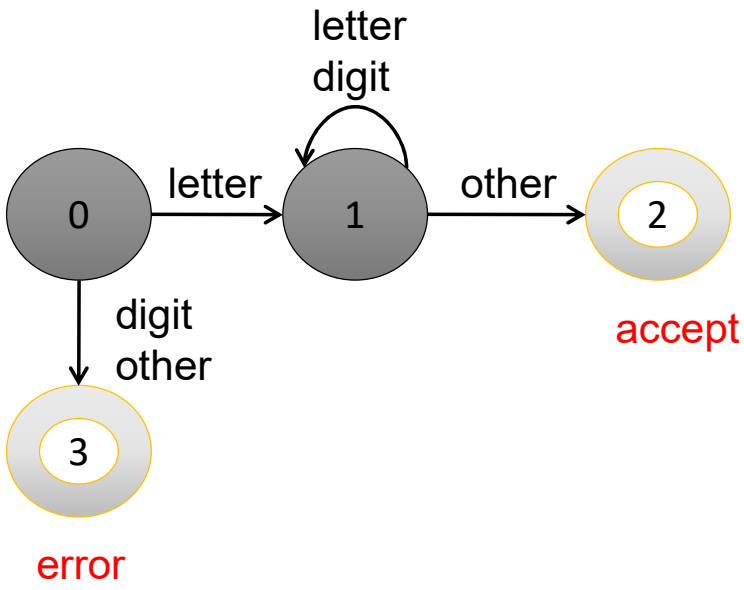


lex.l

```
%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);}

%%
```

1.5词法分析器的自动产生--*flex*

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);}

%%
```

1.5词法分析器的自动产生--*flex*

- flex
  - 快速词法分析器产生器（fast lexical analyzer generator）
- flex的输入源程序(.l文件)由正规表达式表和对应的程序片段组成
- flex生成***lex.yy.c***文件
  - 其中定义函数 *yylex()* 开始词法分析，返回**token**

1.5词法分析器的自动产生--*flex*

- flex输入程序由三个段组成，每个段之前用%%间隔

```
definitions
%%
rules
%%
user code
```

1.5词法分析器的自动产生--flex

- 定义段
  - 形如:

name definition

```
DIGIT     [0-9]
ID        [a-z][a-z0-9]*
```

1.5词法分析器的自动产生--*flex*

- 规则段
  - 形如:

pattern action

```
{ID} printf( "An identifier: %s\n", yytext );
```

- 两个特殊变量:
  - *yytext* 当前匹配的文本字符串
  - *yylength* 当前匹配的文本字符串文本长度.
- 如果**action为空，当前匹配的token被抛弃**

1.5词法分析器的自动产生--*flex*

- 规则段
  - 形如: **pattern action**
  - 预定义Pattern原语

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[]	character class

1.5词法分析器的自动产生--*flex*

- 动作**action**
  - 像C语言一样，使用 ‘{’ 和‘}‘括起来.
  - 如果动作仅包括一个‘|’，表示和下一条规则一样的动作
  - 可以包含*return* 语句
- 优先级问题
  - 例如： ‘<’ 可以匹配‘<’ ，也可以匹配 ‘<=’
  - 文本长的优先
  - 如果长度一致，源程序中规则在前的优先

1.5词法分析器的自动产生--*flex*

- 用户代码段
  - 用户代码段原封不动地复制到 *lex.yy.c*
  - 可选段
- 在定义段和规则段，使用%{和%}包围的文本原封不动地复制（删除%{ %}）

1.5词法分析器的自动产生--*flex*

示例:

```
%{
  int num_lines = 0, num_chars = 0;
%}
%%
\n      ++num_lines; ++num_chars;
.       ++num_chars;
%%
main()  {
  yylex();
  printf( "# of lines = %d, # of chars = %d\n", num_lines, num_chars );
}
```

1.5词法分析器的自动产生--*flex*

示例*ex.l*：

```
%{
   int num_lines = 0, num_chars = 0;
%}
%%
\n ++num_lines; ++num_chars;
.          ++num_chars;
%%
main()  {
   yylex();
   printf( "# of lines = %d, # of chars = %d\n",
                num_lines, num_chars );
}
```

```
[user@s138%s138 ~]$ more input
dd
dd
ss
sa
```

```
$flex ex.l
$gcc lex.yy.c -o test -lfl
$./test< input
     # of lines = 4, # of chars = 12
```

2.1 语法分析

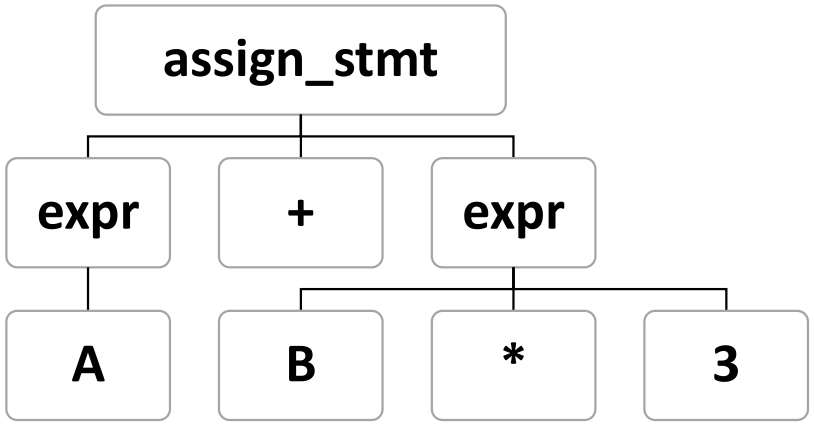
- 又称为解析，验证词法分析得到的词法单元流能够由该语言的文法生成

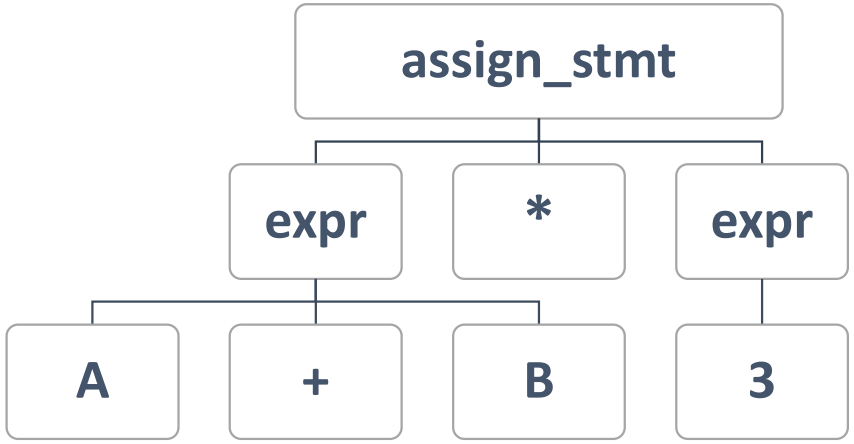
A = A + B * 3;
Token:
<id, A>
<=>
<+>
<id,B>
<*>
<num, 3>

赋值语句的文法：
assign_stmt → lvalue '=' rvalue
lvalue → id
rvalue → expr
expr → expr '+' expr
| expr'*' expt
| id
| num

2.1 语法分析

A = A + B * 3;





具有二义性的文法

- 结合属性
- 优先级

2.2 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

2.3 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
```

2.3 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

2.3 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)。

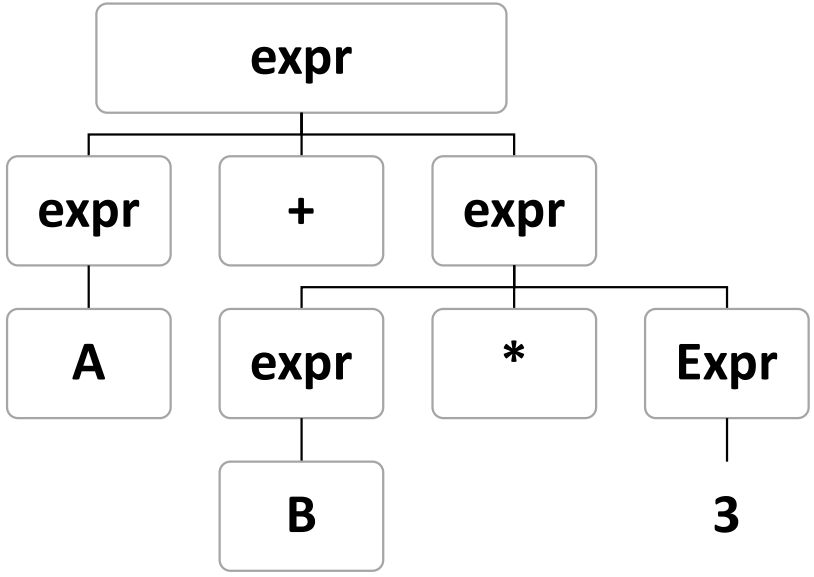
2.3 上下文无关文法

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：

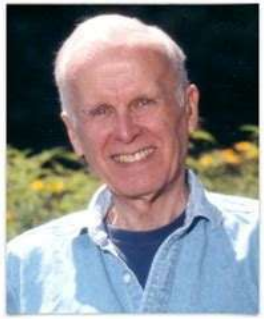
expr → expr '+' expr
| expr'*' expt
| id
| num



2.4 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

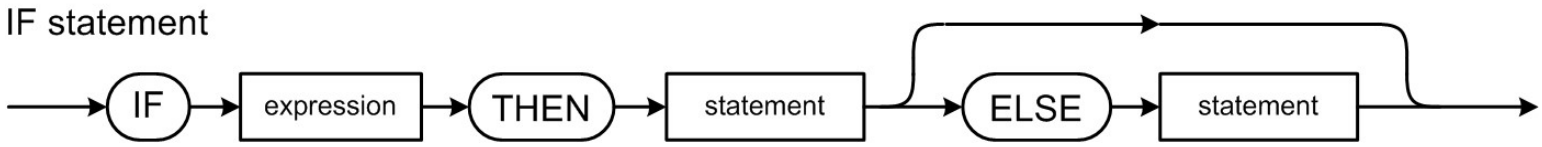
描述语言的元语言

Dr. Peter Naur

2.4 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>
```

2.4 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

2.4 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

2.4 EBNF

- EBNF（Extended Backus–Naur Form）
  - BNF

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

  - EBNF

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

2.4 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> ]
```

2.4 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)

2.4 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

2.4.1 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

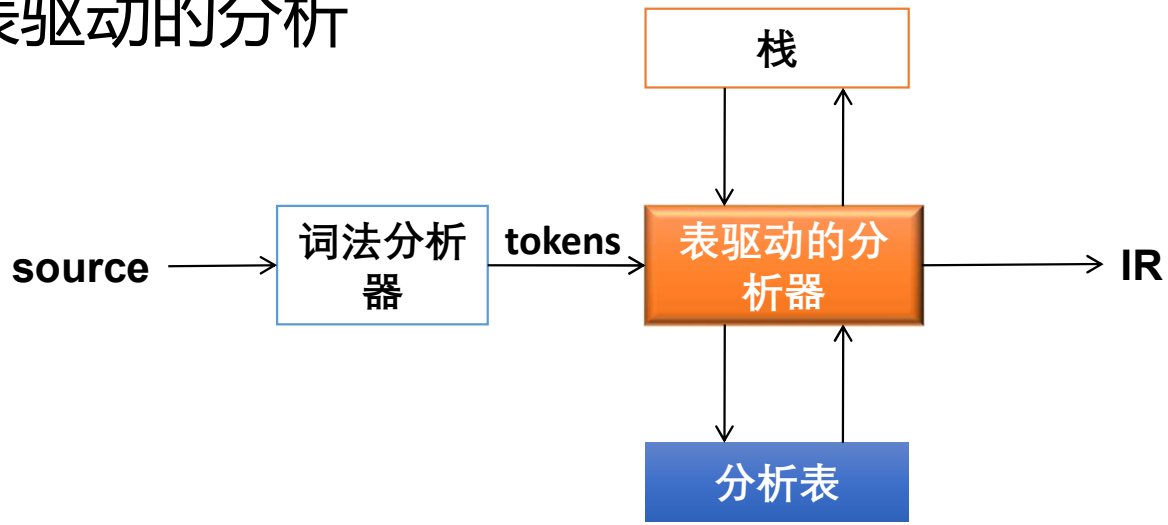
关键在于步骤1，如何选择产生式。产生式的选择通常采用“**超前看**”（Look-ahead）的方法。

2.4.1 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

2.4.1 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)		

2.4.1 自顶向下的分析

- 构建LL(1)的分析表
  - 候选首符集**FIRST(α)**：由α可以推出的任意字符串的开头终结符组成的集合

$$FIRST(\alpha) = \{\alpha | \alpha \overset{*}{\Rightarrow} a\ldots, a \in V_T\}$$

  - **FOLLOW(α)**直接跟在α后的终结符集合

$$FOLLOW(\alpha) = \{a | S \overset{*}{\Rightarrow} \ldots \alpha a\ldots, a \in V_T\}$$

2.4.1 自顶向下的分析

- 如果一个文法G满足以下条件，则称该文法G为LL(1)文法。
  - 文法不含左递归
  - 对于文法中每一个非终结符A的各个产生式的候选首符集两两不相交。即，若

$$A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$$

则 $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \varnothing \quad (i \neq j)$

  - 对文法中的每个非终结符A，若它存在某个候选首符集包含ε，则

$$FIRST(\alpha_i) \cap FOLLOW(A) = \varnothing \qquad i=1,2,...,n$$

2.4.1 自顶向下的分析

- LL(1) 分析表构建算法

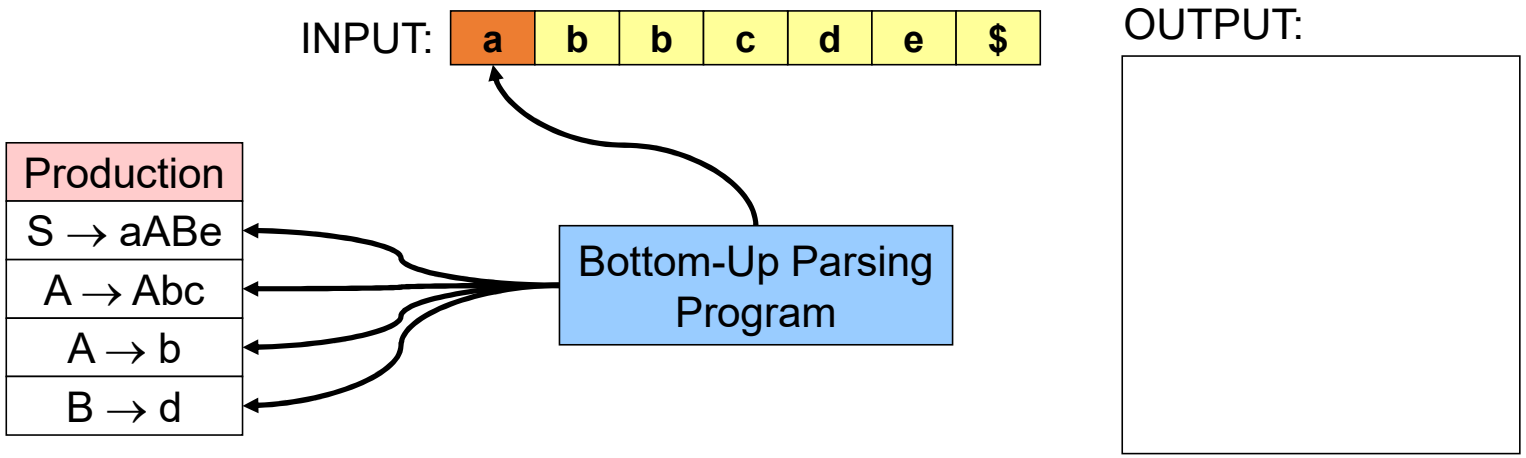
1. ∀P: $A \to \alpha$:
   1.1 ∀$a \in$ ***FIRST(α)***, 将 $A \to \alpha$ 加入到$M[A,a]$
   1.2 如果 $\varepsilon \in FIRST(\alpha)$:
      i. ∀$b \in FOLLOW(A)$, 将 $A \to \alpha$ 加入到 $M[A,b]$
      ii. 如果 \$∈ FOLLOW(A)，将$A \to \alpha$加入到$M[A,\$]$
2. 表M中其他未定义的表项设置为*error*

2.4.2 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

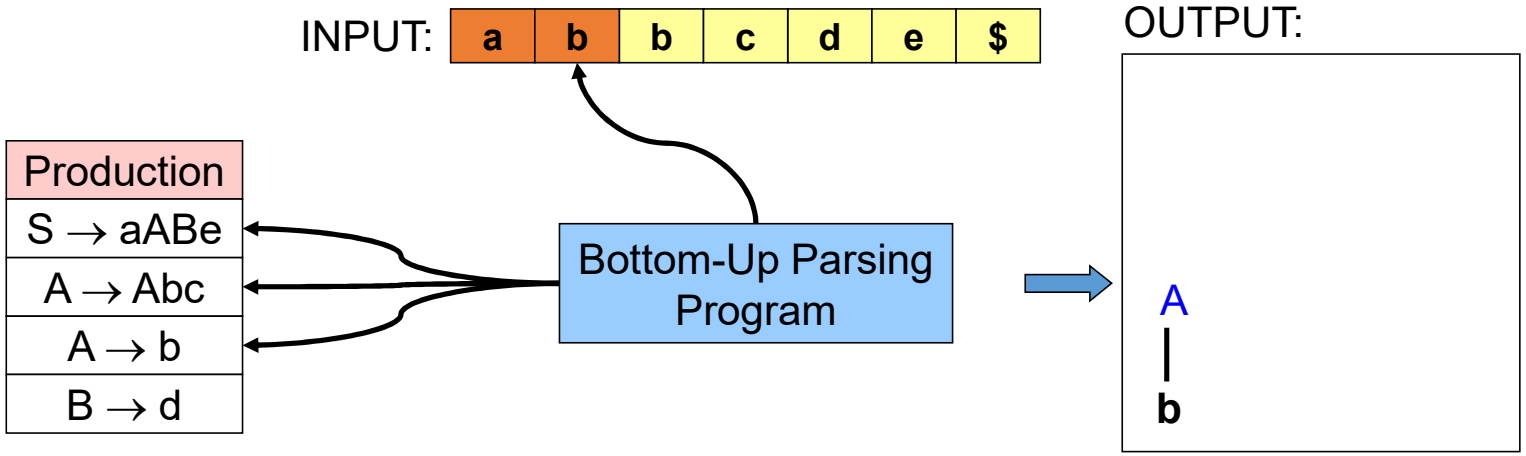
2.4.2自底向上分析

- shift-reduce parser



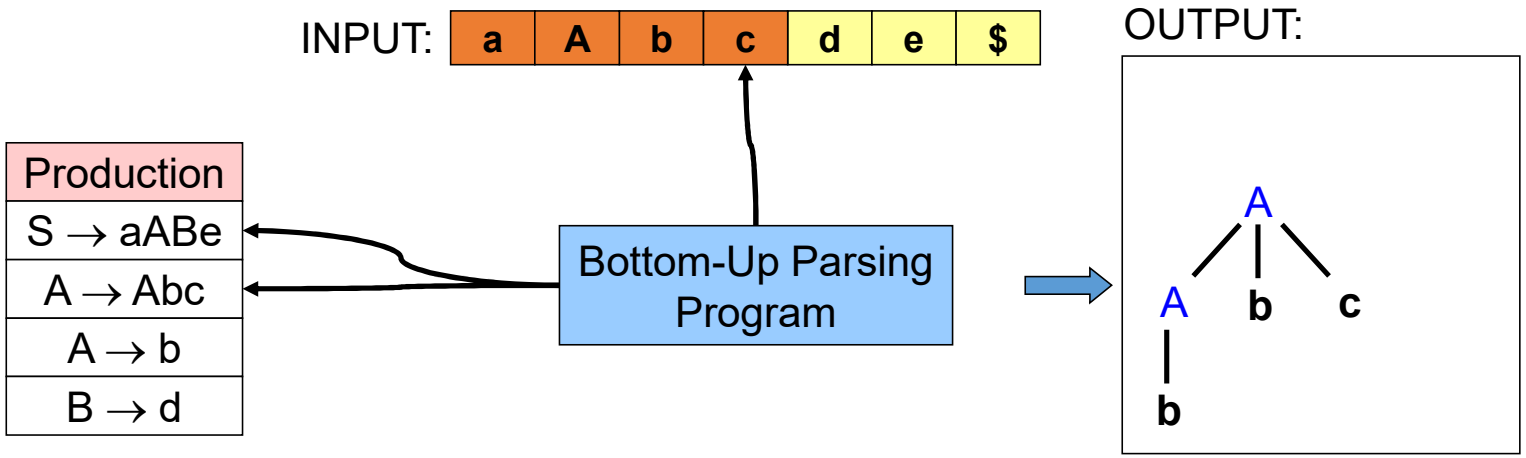
2.4.2自底向上分析

- shift-reduce parser

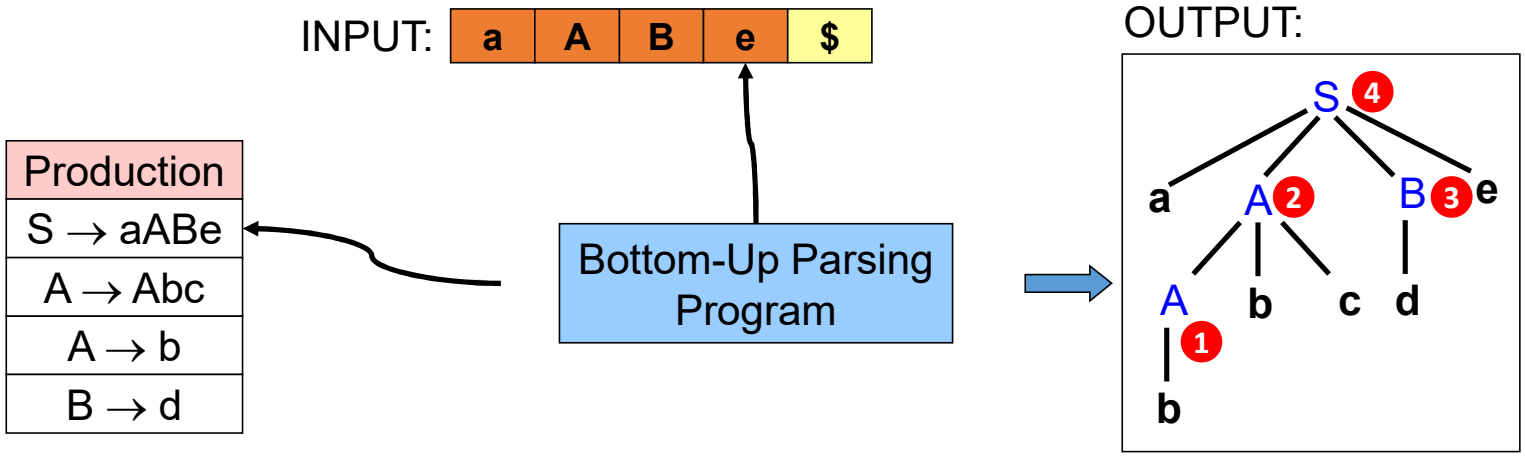


2.4.2 自底向上分析

- shift-reduce parser



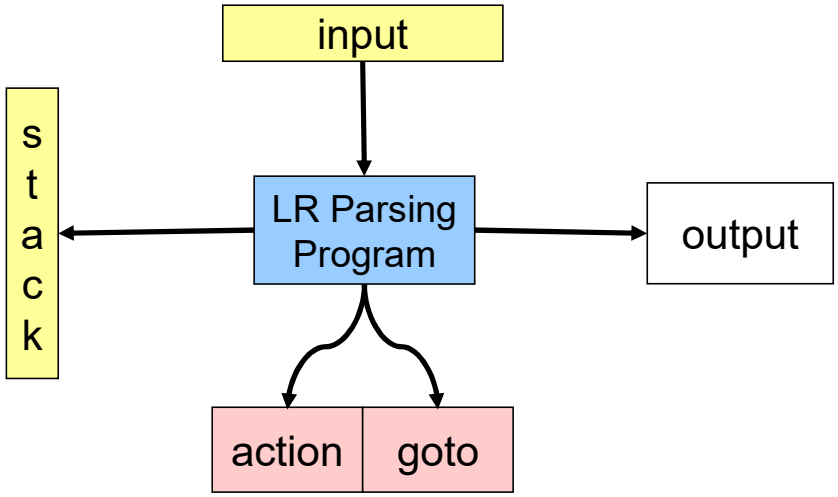
2.4.2 自底向上分析



- 最右推导的逆过程
  - LR

2.4.2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:

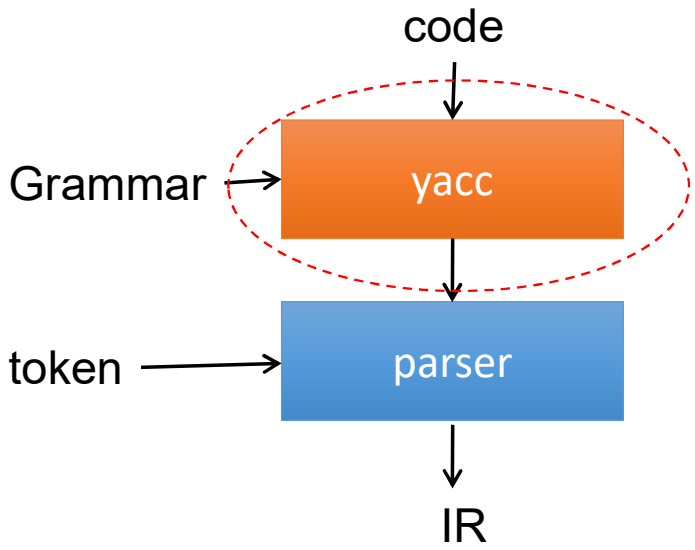
id	*	id	+	id	$

(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			

2.5 解析器产生器——yacc

- yacc/bison
  - LALR(1)(Look-ahead LR)



  - 比LL(1)处理的文法更广

2.5 解析器产生器——yacc

- yacc（Yet Another Compiler Compiler）
  - 文件格式

```
/*declare*/
    %{
          /*Ordinary C declarations*/
    }%
   /*Token Declarations*/
   %%
   The translation Rules
   %%
 Auxiliary C functions
```

  - yacc yaccfile.y 生成y.tab.c文件，包含yyparse()
  - yacc和flex配合使用，yyparse调用yylex获取token

2.5 解析器产生器——yacc

- 声明段/declaration/
  - 声明token

```
%token id INTEGER
```

  - 其他非终结符
    - 通常非终结符不需要额外声明，除非你需要将它和一个类型关联用以存储必要的属性
  - C/C++声明，使用“%{“和“%}”括起来

2.5 解析器产生器——yacc

- 转换规则段/ Translations rules /
  - 描述文法的产生式
  - 使用BNF表示上下文无关文法

```
exp : exp PLUSnumber exp
        | exp MINUSnumber exp
        | exp TIMESnumber exp
        | exp DIVIDEnumber exp
        | LPARENnumber exp RPARENnumber
        | ICONSTnumber
        ;

exp : exp PLUSnumber exp
        ;
exp : exp MINUSnumber exp
        ;
```

2.5 解析器产生器——yacc

- 示例
  - **定义语言的文法**

```
program -> program expr | ε
expr -> expr + expr | expr - expr | id
```

2.5 解析器产生器——yacc

- 示例
  - Lex程序

```
%{
#include <stdlib.h>
void yyerror(char *);
#include "y.tab.h"
%}
%%
[0-9]+        {  yylval = atoi(yytex);
                 return INTEGER;
              }
[-+\n]        return *yytext;
[\t]          /* skip*/
.             yyerror("invalid char!");
%%
int  yywarp() {
    return 1;
}
```

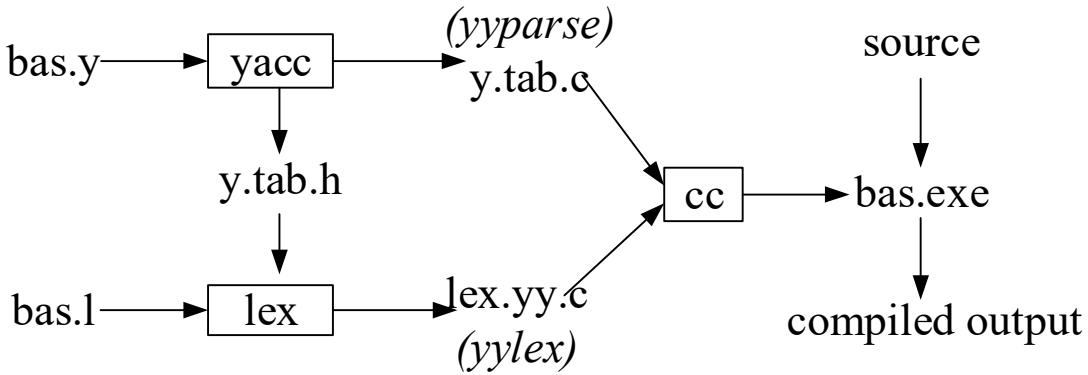
2.5 解析器产生器——yacc

- 示例
  - yacc程序

```
%{
#include <stdio.h>
void yyerror(char *);
int yylex();
%}
%token INTEGER
%%
program:
        program expr '\n'   {printf("%d\n",$2);}
        |
        ;
expr:
        INTEGER  {$$ = $1;}
        |expr'+'expr      {$$ = $1 + $3;}
        |expr'-'expr   {$$ = $1 - $3;}
        ;
%%
int  yyerror(char *s) {
    fprintf(stderr,"%s \n",s);
}
int main(){
    yyparse();
    return 0;
}
```

2.5 解析器产生器——yacc

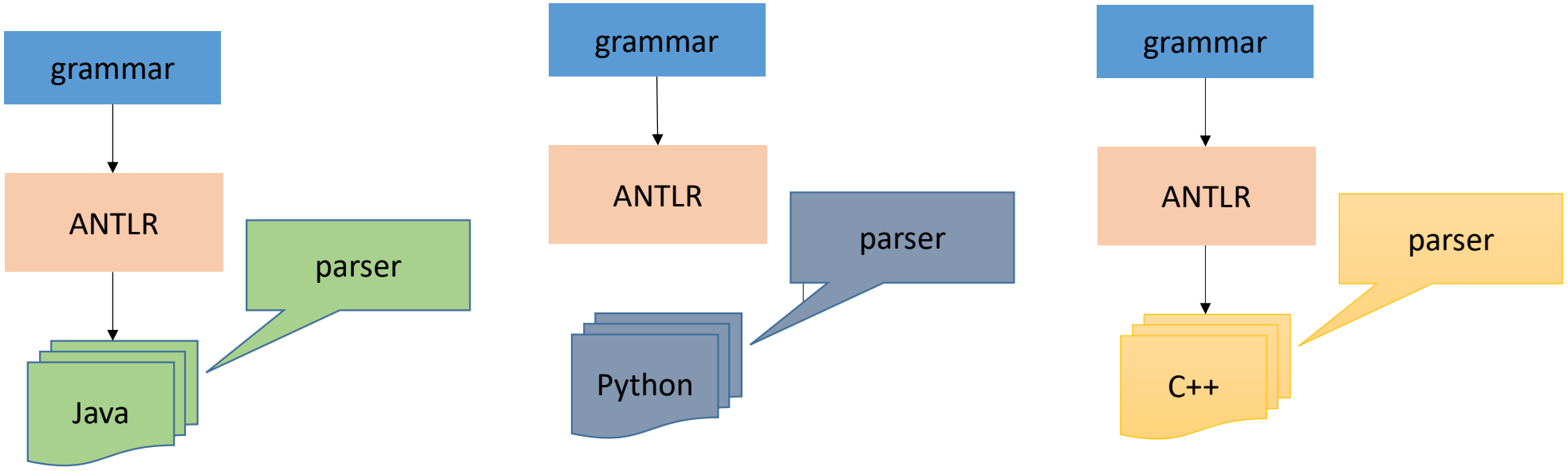
- 示例
  - 生成词法和语法分析器



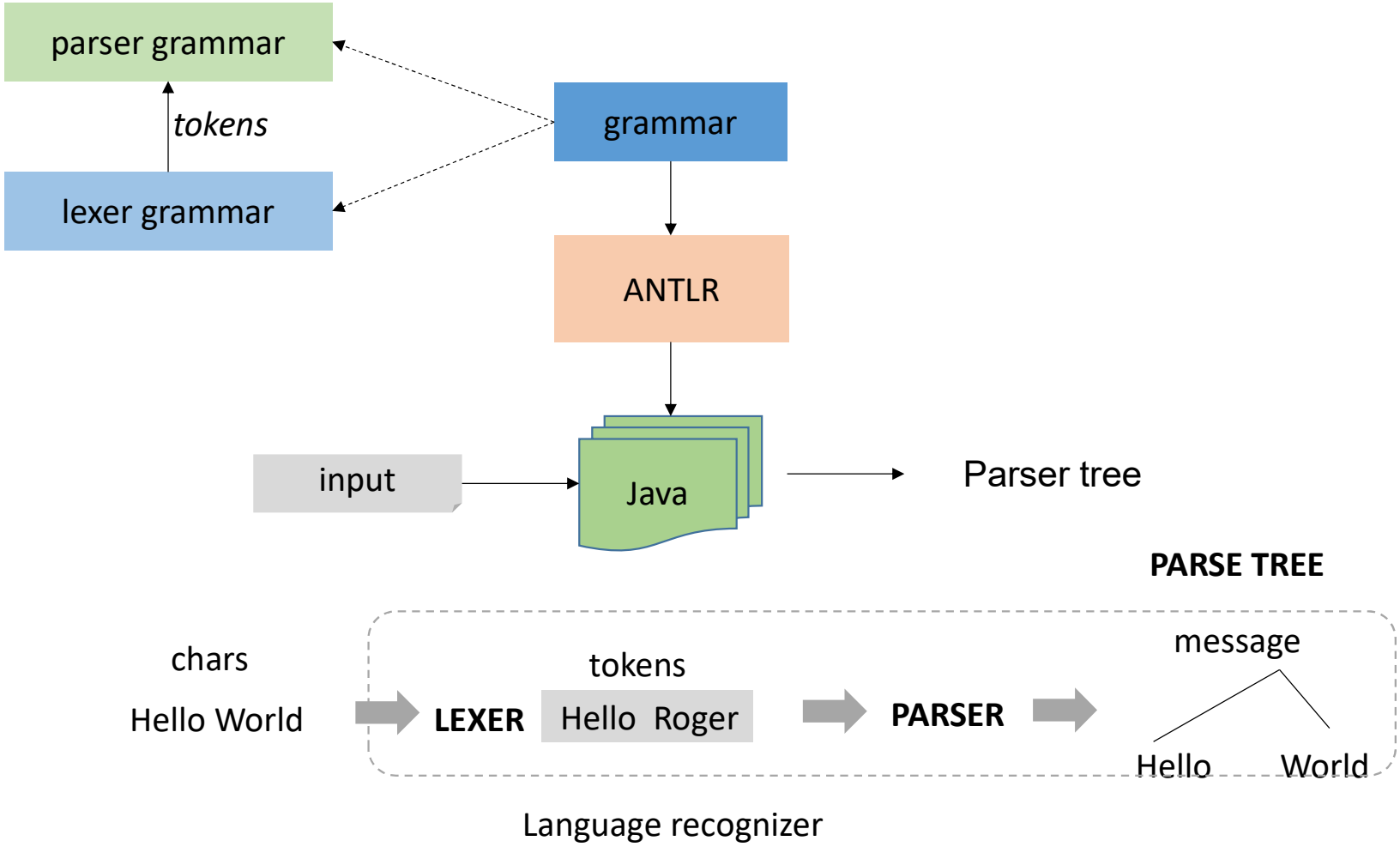

```
user@s138%s138 hc]$ gcc y.tab.c  lex.yy.c   -lfl
[user@s138%s138 hc]$ ./a.out
43+6
49
5/76
invalid char!

syntax error
```

2.6 ANTLR

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



2.6 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

2.6 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

2.7 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

参考

- 《编译原理》
- Tom Niemann. “A Compact Guide to Lex & Yacc ”. Portland, Oregon. 18 April 2010 <http://epaperpress.com>

作业

- 作业：
  - 预习SysY语言定义文档
  - 根据SysY语言定义，请用正规表达式表示整数常数（网站提交）
  - 准备ANTLR实验环境