编译原理

Compiler Construction Principles

朱 青

信息学院计算机系，

中国人民大学，

zqruc2012@aliyun.com

第1章：编译原理引论

- 1.0 课程简介
- 1.1 编译与解释
- 1.2 编译程序概述
- 1.3 编译程序的结构
- 1.4 编译程序的设计与实现
- 1.5 Summary (Compiler Structure)

1.0 课程简介

<u>引例：翻译（编译）程序的组成部分：</u>

如：“ I wish you success. ”

翻译成汉语。

1） 单词进行<u>词法分析</u>：

“ I ”	“ wish”	“ you”	“ success ”
（代）	（动）	（代）	（名）
我	希望	你	成功

2）语法分析：

（主语）（谓语）（间宾）（直宾）

<u>结论：</u> 是一个合乎英语语法的句子。

3）语义分析：

我希望你成功。

4）汉语句子进行修饰（优化）：

祝你成功。

为什么学习编译原理?

- ⌘ 常规用户——
  - ⌘ “编译原理”实际是翻译程序，无处不在
  - ⌘ 熟练使用与掌握编译原理， 理解计算机的编译与自动翻译程序。
- ⌘ 程序员与高级研发——
  - ⌘ 加深对使用的编译原理的理解，有利于深入编程
  - ⌘ 编程时借鉴编译原理的设计思想和算法
  - ⌘ 设计翻译程序 或者 修改现有的系统（智能翻译等）
  - ⌘ 高级研发经常与编译器打交道（自然语言处理等）
  - ⌘ 编译原理中概念和技术可推广应用领域（信息检索等）

课程简介

- 什么是编译程序？
- 计算机语言
- 计算机语言的组成结构
- 学习内容
- 考核要求
- 教学参考书

课程简介

- 什么叫编译程序
  - 将高级程序设计语言(C, C++, Java, pascal等)翻译成逻辑上等价的低级语言(汇编语言,机器语言)的翻译程序。

源程序 ⟹ **编译程序** ⟹ 目标程序

(*.c / *.cpp)　　　　(*.obj / *.exe)

高级语言程序

- Example:
  Bubblesort program that sorts array A allocated in static storage

```
for (i = n-2; i >= 0; i--) {
    for (j = 0; j <= i; j++) {
        if (A[j] > A[j+1]) {
            temp = A[j];
            A[j] = A[j+1];
            A[j+1] = temp;
        }
    }
}
```

Code Generated by the Front End

```
   i = n-2
S5:if i<0 goto s1
   j = 0
s4:if j>i goto s2
   t1 = 4*j
   t2 = &A
   t3 = t2+t1
   t4 = *t3      ;A[j]
   t5 = j+1
   t6 = 4*t5
   t7 = &A
   t8 = t7+t6
   t9 = *t8      ;A[j+1]
   if t4 <= t9 goto s3
   t10 = 4*j
   t11 = &A
   t12 = t11+t10
   temp = *t12   ;temp=A[j]
```

```
   t13 = j+1
   t14 = 4*t13
   t15 = &A
   t16 = t15+t14
   t17 = *t16       ;A[j+1]
   t18 = 4*j
   t19 = &A
   t20 = t19+t18    ;&A[j]
  *t20 = t17        ;A[j]=A[j+1]
   t21 = j+1
   t22 = 4*t21
   t23 = &A
   t24 = t23+t22
  *t24 = temp       ;A[j+1]=temp
s3:j = j+1
   goto S4
S2:i = i-1
   goto s5
s1:
```

(t4=*t3 means read memory at address in t3 and write to t4:
*t20=t17 :store value of t17 into memory at address in t20)

低级语言程序 After Optimization

Result of applying

- global common subexpression
- loop invariant code motion
- induction variable elimination
- dead-code elimination

to all the scalar and temp. variables

These traditional optimizations can make a big difference!

```
    i = n-2
    t27 = 4*i
    t28 = &A
    t29 = t27+t28
    t30 = t28+4
S5:if t29 < t28 goto s1
    t25 = t28
    t26 = t30
s4:if t25 > t29 goto s2
    t4 = *t25       ;A[j]
    t9 = *t26       ;A[j+1]
    if t4 <= t9 goto s3
    temp = *t25     ;temp=A[j]
    t17 = *t26      ;A[j+1]
    *t25 = t17      ;A[j]=A[j+1]
    *t26 = temp     ;A[j+1]=temp
s3:t25 = t25+4
    t26 = t26+4
    goto S4
S2:t29 = t29-4
    goto s5
s1:
```

代码生成 Code Generation

- Mapping machine independent assembly code to the target architecture （汇编语言到机器语言的映射）
- Virtual to physical binding （物理绑定）
  - Instruction selection – best machine op-codes to implement generic op-codes
    （指令选择）
  - Register allocation – infinite virtual registers to N physical registers
    （寄存器分配）
  - Scheduling – binding to resources (adder1)（调度）
  - Assembly emission（汇编组装， 发布）
- Machine assembly is our output, assembler, linker take over to create binary
  （汇编语言，链接 等创建二进制代码 → （机器语言））

实例：C程序的编译过程

- 源程序：

```
main( )
{
   printf("hello");
}
```

- 编译过程：

- 1. 词法分析
- 2. 语法分析
- 3. 语义分析
- 4. 中间代码与优化
- 5. 目标代码生成

1. 词法分析

- 源程序：

```
main( )
{
  printf("hello");
}
```

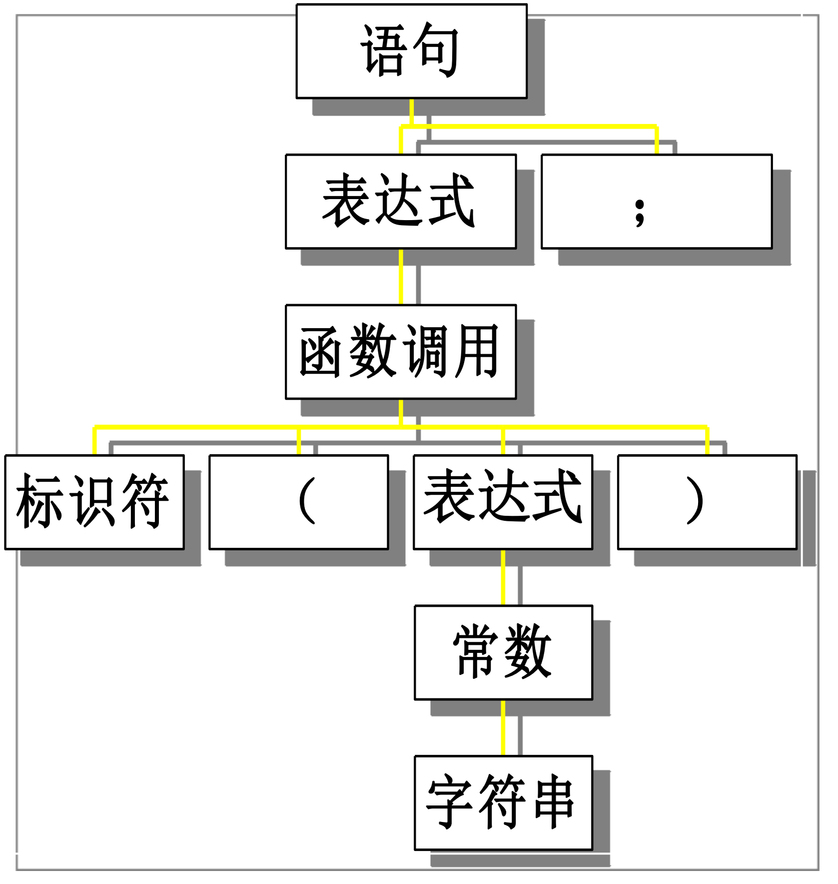
- 1. 词法分析
  - 切词
  - 属性
  - 分类
  - 填写符号表

结果

```
IDN
      main
'('
')'
'{'
IDN
      printf
'('
STR
      hello
')'
';'
'}'
```

2.语法分析

- 分析单词序列；
- 识别语法结构；



- 查语法错误；
- 构造分析树；

3.语义分析

- 确认标识符的属性
  - 类型、作用域等
- 语义检查
  - 运算的合法性、取值范围等
- 子程序的静态绑定
  - 代码存储的相对地址
- 变量的静态绑定
  - 数据存储的相对地址

4.中间代码与优化

- 中间语言（逆波兰表达式）
  - 简单规范
  - 机器无关
  - 易于优化与转换
- 按照语法分析树生成中间语言代码
  - 运算指令
  - 控制指令
- 中间代码与优化
  - 中间代码的优化处理，以求提高执行效率

例（三地址代码）

x := s （赋值）

param x （参数）

call f （函数调用）

注释

s 是 hello 的地址

f 是函数 printf 的地址

5. 目标代码生成

- 将中间代码转换成目标机上的机器指令代码或汇编代码

MOV R0, #12,	10000001 0001 1100
ADD R0, #4	10000010 0001 0100
MUL R0, R2	11000100 0001 0010
汇编指令代码	机器指令代码

计算机语言的共同点

- 语法：
  - 语句的组成规则
  - 描述方法：BNF范式、语法描述图
- 词法：
  - 单词的组成规则
  - 描述方法：BNF范式、正规式
- 单词：
  - 具有语义的最小字符串（可区分的）

语言的描述方法

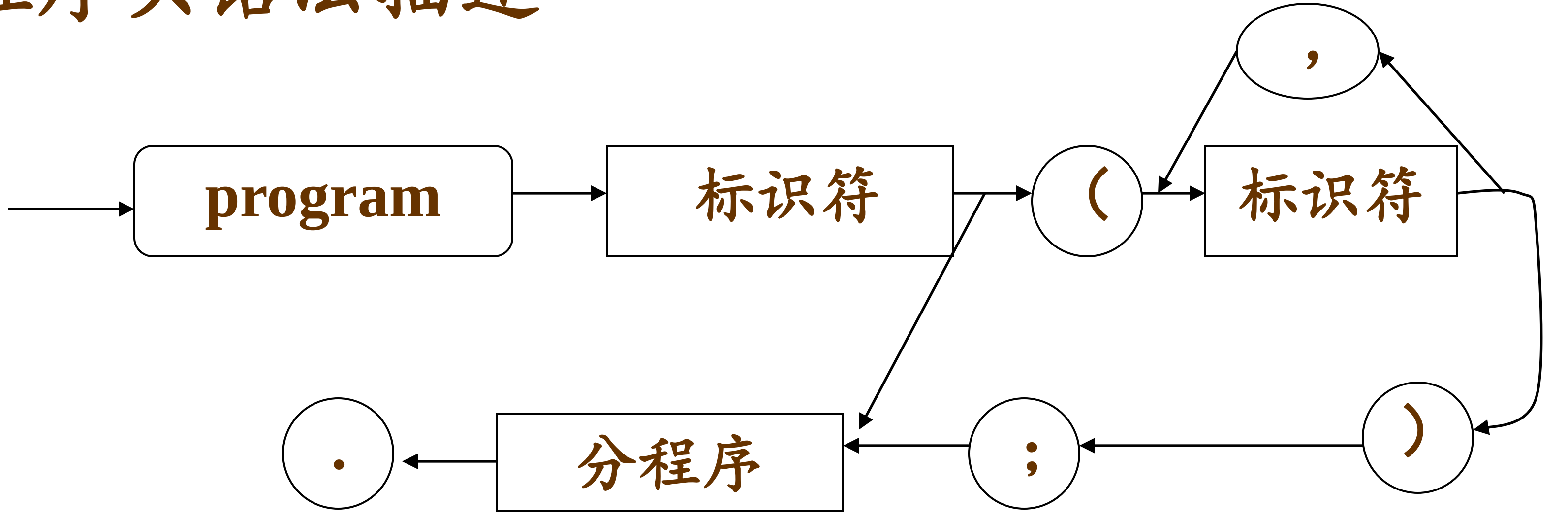
- 叙述性方法
  - 自然语言（非形式化描述）
- 记号方法
  - 数学方法（形式化描述）
  - 保证描述清晰准确
- 形式化描述的作用
  - 理论基础和抽象分析方法

程序设计语言的定义（语法）

- 语法：是指规定如何由基本符号组成一个完整的程序的规则。可以分文一般的语法规则和词法规则。
- 定义语法的方式
  - 语法图：直观，篇幅大。
  - BNF表示法：简洁，严谨，精确。
  - 自然语言：一般不使用在正式的文本中。

语法图（pascal）

程序头语法描述



BNF表示方式（范式定义）

- 〈程序〉 ::=〈程序首部〉；〈程序分程序〉
- 〈程序首部〉::= program〈标识符〉（〈程序参数〉）
- 〈程序分程序〉::=
- 〈标识符〉::=
- 〈程序参数〉::=
- ……

程序设计语言的定义（语义）

- 程序设计语言中按照语法规则所构成的各个语法成分的意义。
- 定义方式：
  - 一般使用比较严格的自然语言进行描述。
  - 形式化的方法：使用数学符号以没有歧义的方式定义。

教学内容

总学时：51 学时授课，实验：约51 学时

- 引论
- 词法分析
- 文法与形式语言
- 语法描述与分析（自顶向下，自底向上）
- 中间代码的生成
- 符号表
- 运行时存储空间组织
- 代码优化
- 目标代码生成
- 错误的检查和修复
- 编译扩展应用介绍

教学要求

1　讲授为主，课堂表现（出勤率、随堂练习等）

2　掌握每章的重点难点，按时完成作业（认真、独立）

3　理论与实践结合，重视上机实验（按时、认真完成）

4　鼓励举一反三，教材+PPT+复习+阅读参考资料+网站信息

5　鼓励开拓思路，创新思路，融会贯通

6　~~~~~~

让课堂不仅输送知识，还要引领价值；不仅传授学业，更要教会方法；提升学习力与创新力。

成绩评定

- 平时考核+期中考试（作业、实验、课堂表现等）
  - 成绩占总评成绩的50%
- 期末考试
  - 成绩占总评成绩的50%

教材与参考书

- 教材： 陈火旺，程序设计语言《编译原理》（第3版），国防工业出版社
- 教材参考书：与书配套的习题解答。
- **Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "Compilers: Principles, Techniques, and Tools"(second) Addison-Wesley** （英文版）人民邮电出版社，
  （中文版） 机械工业出版社
- **Kenneth C.Louden, "Compiler Construction Principles and Practice"**
  （中文版） 机械工业出版社，
- 陈意云，编译原理和技术，中国科学技术大学出版社
- 教材参考书：与书配套的习题解答。
- 教材PPT+参考资料+网站信息

第1章：编译原理引论

- ⌘1.0 课程简介
- ⌘1.1 编译与解释
- ⌘1.2 编译程序概述
- ⌘1.3 编译程序的结构
- ⌘1.4 编译程序的设计与实现
- ⌘1.5 Summary (Compiler Structure)

1.1 编译与解释

<u>翻译程序</u>：把一种程序语言翻译成另一种程序语言的程序，两者逻辑上等价。 如：编译，汇编，反汇编 等。

<u>编译程序</u>： 把高级程序设计语言翻译成等价的低级语言程序，最终生成可执行代码。

高级语言 ⟶ <u>编译程序</u> ⟶ 目标程序
源程序

解释程序：　它以用该语言编写的源程序作为输入，但不 产生目标程序，而是按照语言的定义，边解释边执行源程序本身。 通常把源程序翻译成某种中间程序，逐条解释并执行，从而完成翻译。

混合编码：是一种折衷形式。即对运行较慢的部分采用编译，其它部分采取解释执行。

- 许多情况，用于编译程序的构造技术同样也适用于解释程序。

examples

Compilation: source $\xrightarrow{\textit{translate}}$ real machine language $\xrightarrow{\textit{execute}}$ actions/results

Interpretation: source $\xrightarrow{\textit{translate}}$ virtual machine language $\xrightarrow{\textit{interpret}}$ actions/results

Direct Execution: source $\xrightarrow{\textit{interpret}}$ actions/results

Most C/C++ systems are examples of compilation. Lisp interpretation is (in effect) an example of direct execution (the “translation” performed by the reader is trivial).

第1章：编译原理引论

- ⌘ 1.0 课程简介
- ⌘ 1.1 编译与解释
- ⌘ 1.2 编译程序概述
- ⌘ 1.3 编译程序的结构
- ⌘ 1.4 编译程序的设计与实现
- ⌘ 1.5 Summary (Compiler Structure)

1.2 编译程序概述

五个阶段： （源程序 ⟶ 目标程序）

1。词法分析： 输入源程序，对其扫描并分析，识别各单词符号（基本字、标识符、常数、算符、界符等） 如：FORTRAN语句

DO 55 I = 1 , 100

基本字 标号 整变量 赋值号 整常量 界符 整常量

- <u>词法分析</u>的依据是<u>词法规则</u>。

2。 语法分析。 根据 语法规则 把单词串分解成各类

语法范畴（短语、子句、表达式、程序段、程序）。

- 语法分析所依循的是语言的 语法规则。

3。中间代码生成。

语义解释，语法制导翻译。

根据各语法成分的语义写出其中间代码。

它是一种记号系统，其结构简单，含义明确，

与硬件无关，但很容易变换成机器指令。

常用的中间代码有：逆波兰、三元式、四元式，间接三元式、树等。如：$M = I + M$

四元式表示为：

操作码	第一操作数	第二操作数	结果
+	I	M	M

4。 优化。

对中间代码进行加工变换， 以期在最后阶段 产生出更为高效的(省时间和空间)的目标代码。

常做的优化有：公共子表达式的外提、循环优化、算符归约等。如

```
for  k := 1  to 100  do
begin
    m := i + 10 * k ;
    n := j + 10 * k ;
end;
```

其四元式代码：

序号	操作码	第一	第二	结果	
1	: =	1		k	
2	j<	100	k	(9)	\\ for k:=1 to 100
3	*	10	k	t1	
4	+	i	t1	m	\\ m:=i+10*k
5	*	10	k	t2	
6	+	j	t2	n	\\ n:=j+10*k
7	+	1	k	k	
8	j			(2)	
9	- - - - - - -				

3 和 5： 200 次乘法。（优化）

序号	操作码	第一	第二	结果
1	:=	i		m
2	:=	j		n \\ m,n初值
3	:=	1		k
4	j<	100	k	(9) \\ for k:=1 to 100
5	+	m	10	m
6	+	n	10	n \\ m,n循环+10
7	+	1	k	k
8	j			(4)
9。。。。。。				

- 优化后的四元式

优化所依循的原则是程序的等价变换原则。

5.目标代码的生成：

在中间代码（或经过优化）的基础上，生成某具体的硬件代码，可以是绝对机器代码，或汇编代码。

联接装配程序

总之， 编译 要先分析，以确定源程序的功能；

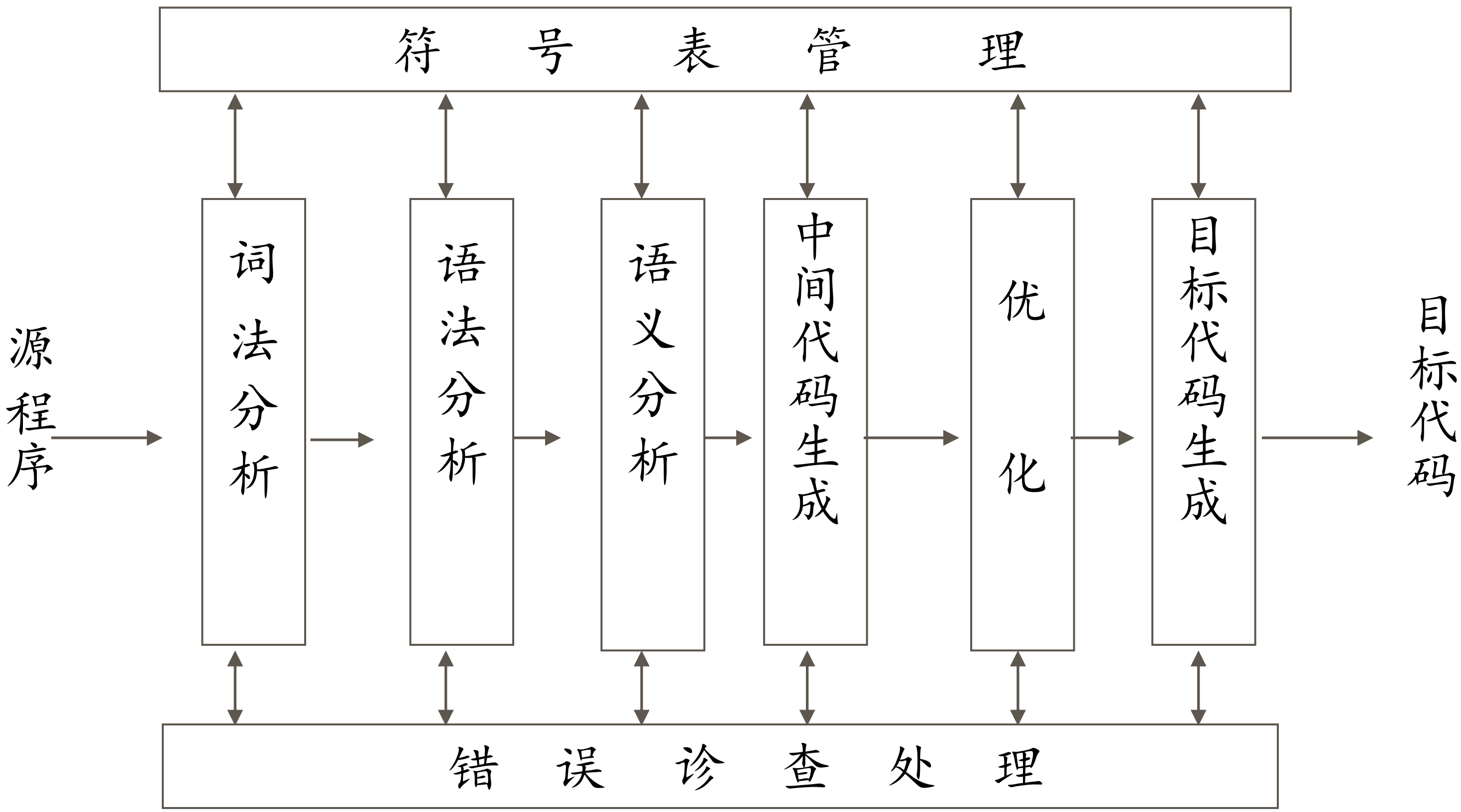
然后综合，以生成该功能的目标程序。

第1章：编译原理引论

- ⌘1.0 课程简介
- ⌘1.1 编译与解释
- ⌘1.2 编译程序概述
- ⌘1.3 编译程序的结构
- ⌘1.4 编译程序的设计与实现
- ⌘1.5 Summary (Compiler Structure)

1.3 编译程序结构

编译程序的总框图：



分析

分析阶段　　是对源程序进行 结构分析和 语义分析。

结构分析　　包括词法分析、语法分析。

语义分析　　用于决定源程序的含义。

综合

中间代码生成程序

代码生成程序

表格管理(构造、查找、更新)

符号名表　标号表

入口名表　过程引用表

表格管理的格式：

NAME	INFORMATION

信息栏：名字的属性、状态。

（书P 5）

表的种类：

循环特征表、等价名表、公共链表、中间代码表（四元式表）

错误处理（指出错误的性质、地点）

单词的拼写错位

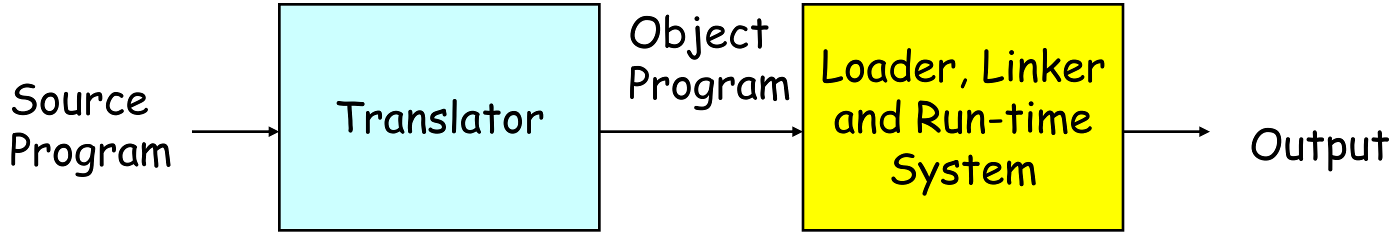
语法错位，括号不配对

语义错位，运算符不相容。

目标程序区超界

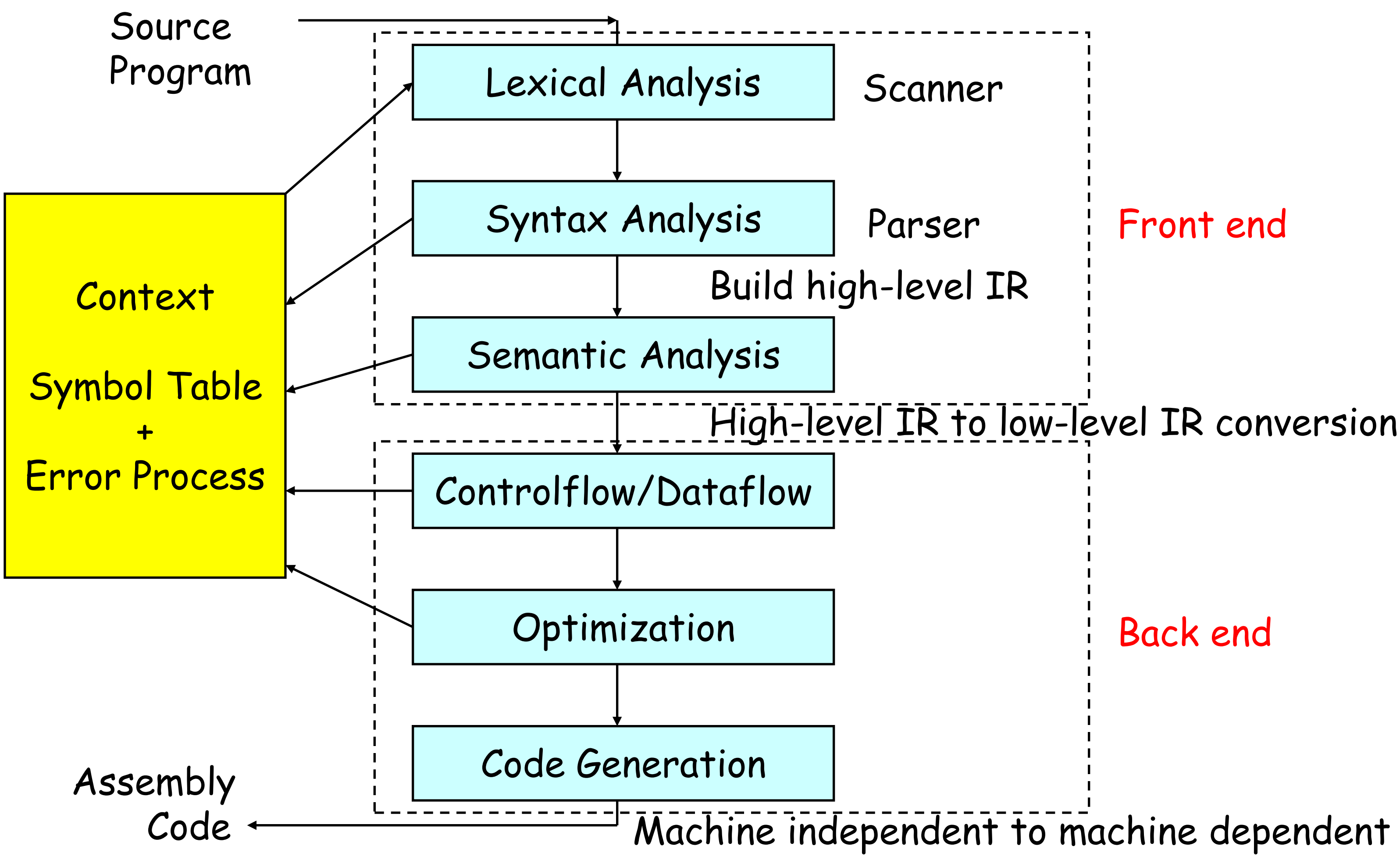
计算机容量受限

附：扩展知识 (Compiler Structure)



- **Source language**
  - **Fortran, Pascal, C, C++**
- **Target language**
  - **Machine code, assembly**
  - **High-level languages, simply actions**
- **Compile time vs run time**
  - **Compile time or statically – Positioning of variables**
  - **Run time or dynamically –heap allocation**

General Structure of a Modern Compiler



IR (intermediate representation)

Lexical Analysis (Scanner)

- Extracts and identifies lowest level lexical elements from a source stream(从源数据流中提取和识别词法元素)
  - Reserved words（关键字）: for, if, switch
  - Identifiers（标示符）: “i”, “j”, “table”
  - Constants（常数）: 3.14159, 17, “%d\n”
  - Punctuation symbols（标点符号）: “(“, “)”, “,”, “+”
- Removes non-grammatical elements from the stream – ie spaces, comments（去除非语法元素）
- Implemented with a Finite State Automata (FSA)（有限状态自动机）
  - Set of states – partial inputs
  - Transition functions to move between states

Lex/Flex（词法自动生成器）

- Automatic generation of scanners
  - Hand-coded ones are faster
  - But tedious to write, and error prone!
- Lex/Flex
  - Given a specification of regular expressions
  - Generate a table driven FSA（有限状态自动机）
  - Output is a C program that you compile to produce your scanner

Parser（分析程序）

- Check input stream for syntactic correctness
  - Framework for subsequent semantic processing
  - Implemented as a push down automaton (PDA)
- Lots of variations
  - Hand coded, recursive descent（递归下降）
  - Table driven (top-down or bottom-up)
  - For any non-trivial language, writing a correct parser is a challenge
- Yacc (yet another compiler's compiler)
  - Given a context free grammar
  - Generate a parser for that language (again a C program)

Static Semantic Analysis

- Several distinct actions to perform
  - Check definition of identifiers, ascertain that the usage is correct（检查标识符定义，确定其适用范围的正确性）
  - Disambiguate overloaded operators（消除操作歧义）
  - Translate from source to IR (intermediate representation)
- Standard formalism used to define the application of semantic rules is the Attribute Grammar (AG)
  - Graph that provides for the migration of information around the parse tree
  - Functions to apply to each node in the tree

Backend（后端）

- Frontend –
  - Statements, loops, etc
  - These broken down into multiple assembly statements
- Machine independent assembly code
  - 3-address code
  - Infinite virtual registers,
  - infinite resources
  - “Standard” op-code， load/store architecture
- Goals
  - Optimize code quality
  - Map application to real hardware

Dataflow and Control Flow Analysis

- Provide the necessary information about variable usage and execution behavior to determine when a transformation is legal/illegal
- Dataflow analysis
  - Identify when variables contain "interesting" values
  - Which instructions created values or consume values
  - DEF, USE, GEN, KILL
- Control flow analysis
  - Execution behavior caused by control statements
  - If's, for/while loops, goto's
  - Control flow graph

Optimization（优化）

- How to make the code go faster
- Classical optimizations
  - Dead code elimination（消除子代码）
    - – remove useless code
  - Common sub-expression elimination （消除公共子表达式）
    - – re-computing the same thing multiple times
- Machine independent (classical)
  - Focus of this class
  - Useful for almost all architectures
- Machine dependent
  - Depends on processor architecture
  - Memory system, branches, dependences

第1章：编译原理引论

- ⌘1.0 课程简介
- ⌘1.1 编译与解释
- ⌘1.2 编译程序概述
- ⌘1.3 编译程序的结构
- ⌘1.4 编译程序的设计与实现
- ⌘1.5 Summary (Compiler Structure)

1.4 编译程序的设计与实现

交叉编译

用“T”型图表示编译程序。

称：

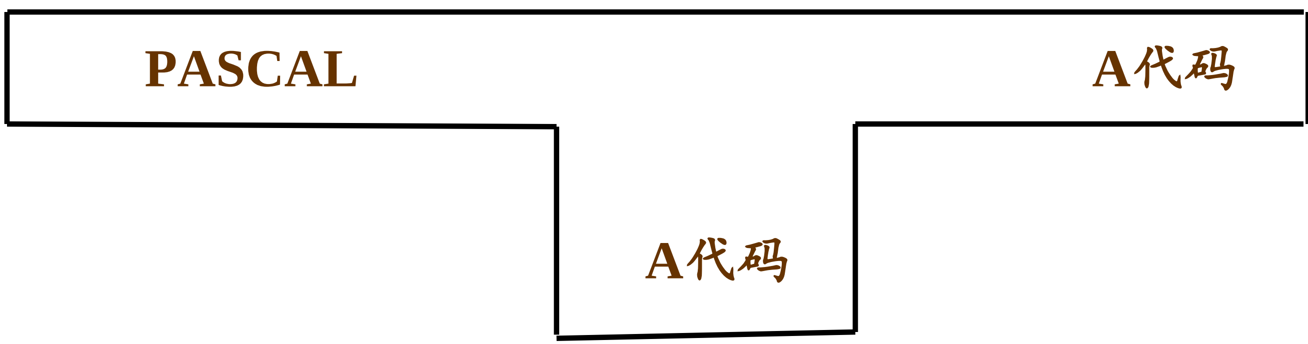
用APPLE代码编写的，生成APPLE代码的PL/1编译程序。

- 用 实现（编译）语言编写的，生成目标程序的 源语言编译程序。

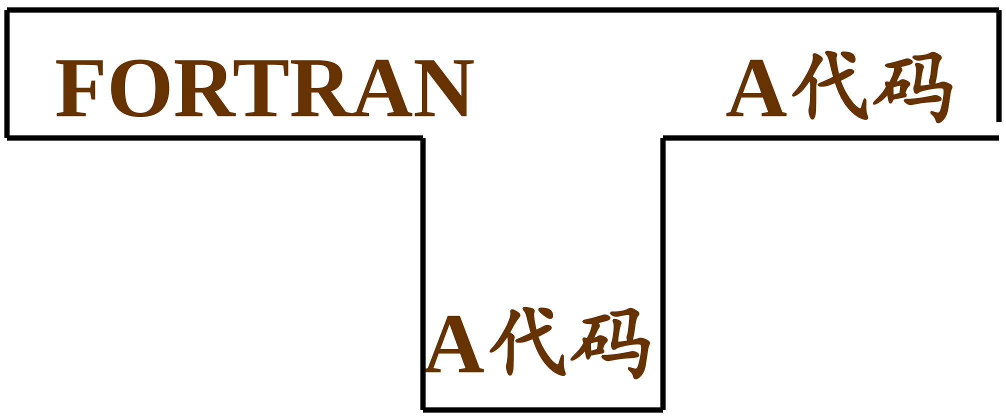
一般不采用低级语言作为实现语言。

- <u>交叉编译</u>　例如：一个PASCAL编译。

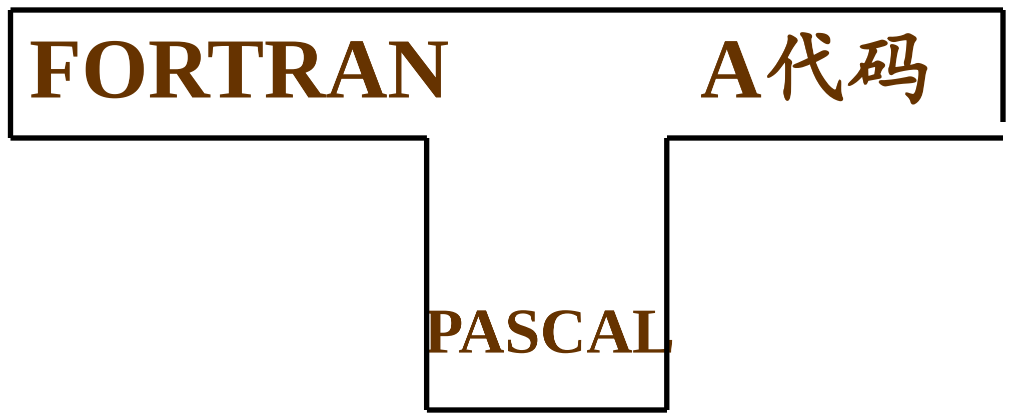
已有的



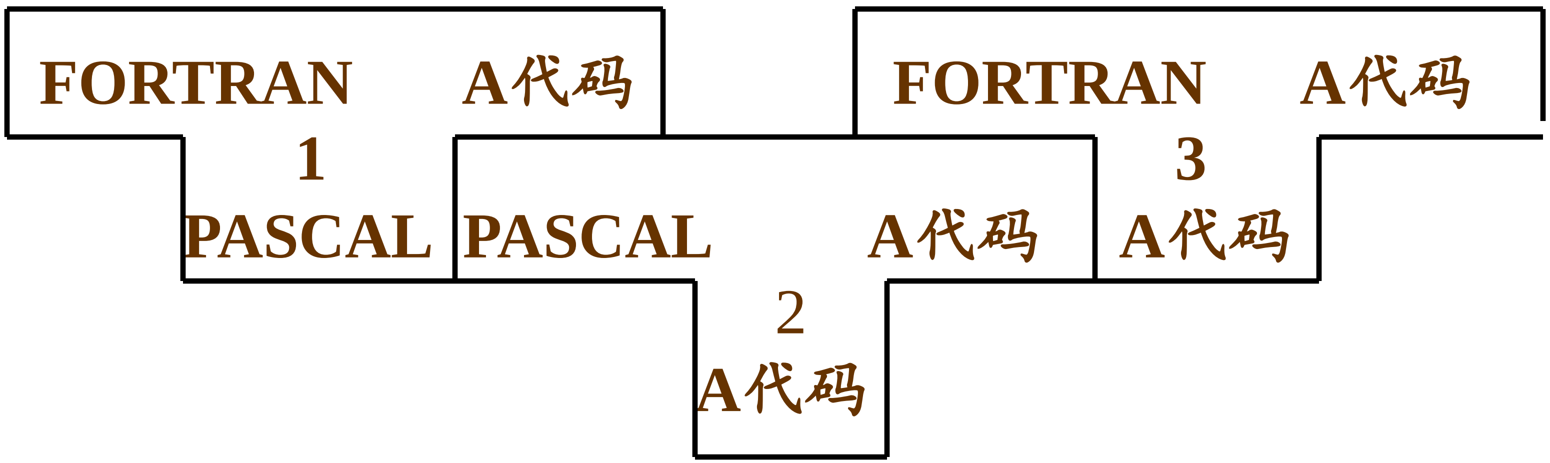
要得到的



需要编写的



用第2个编译去 编译第1个，得到第3个编译程序



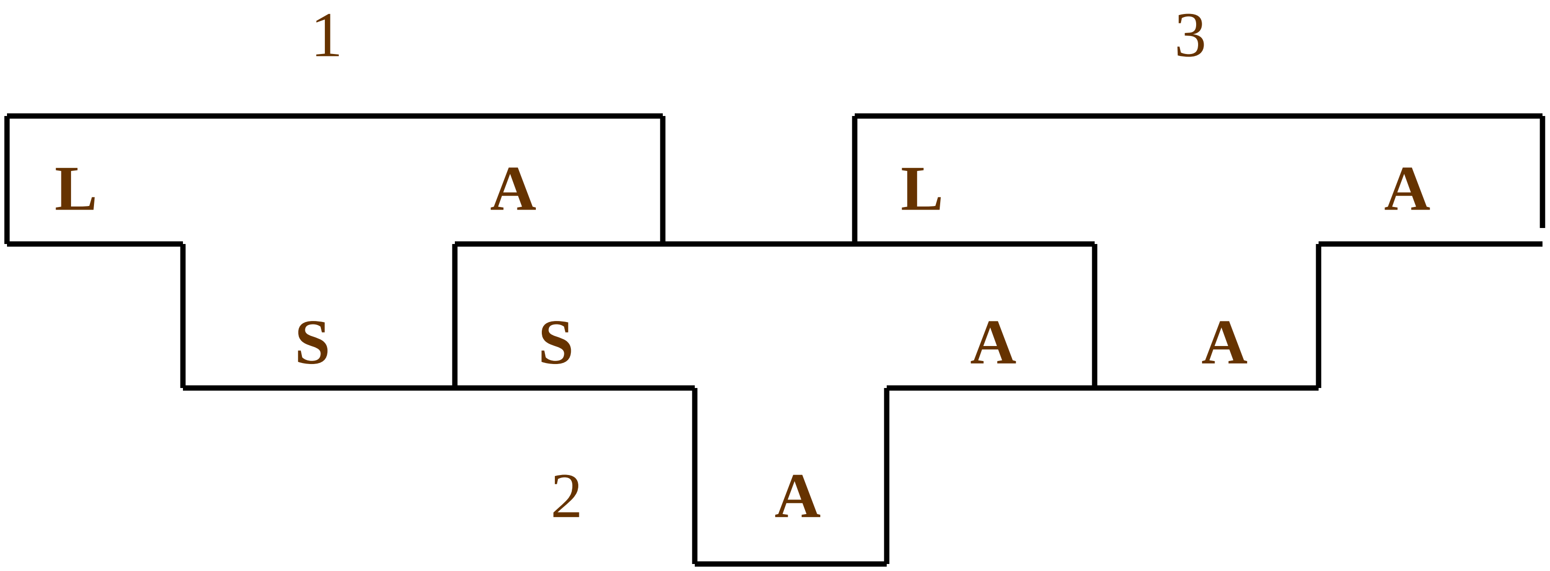
这个编译程序PASCAL程序，可得到具同样功能的用A代码表示的目标程序。

T型图的结合规则：

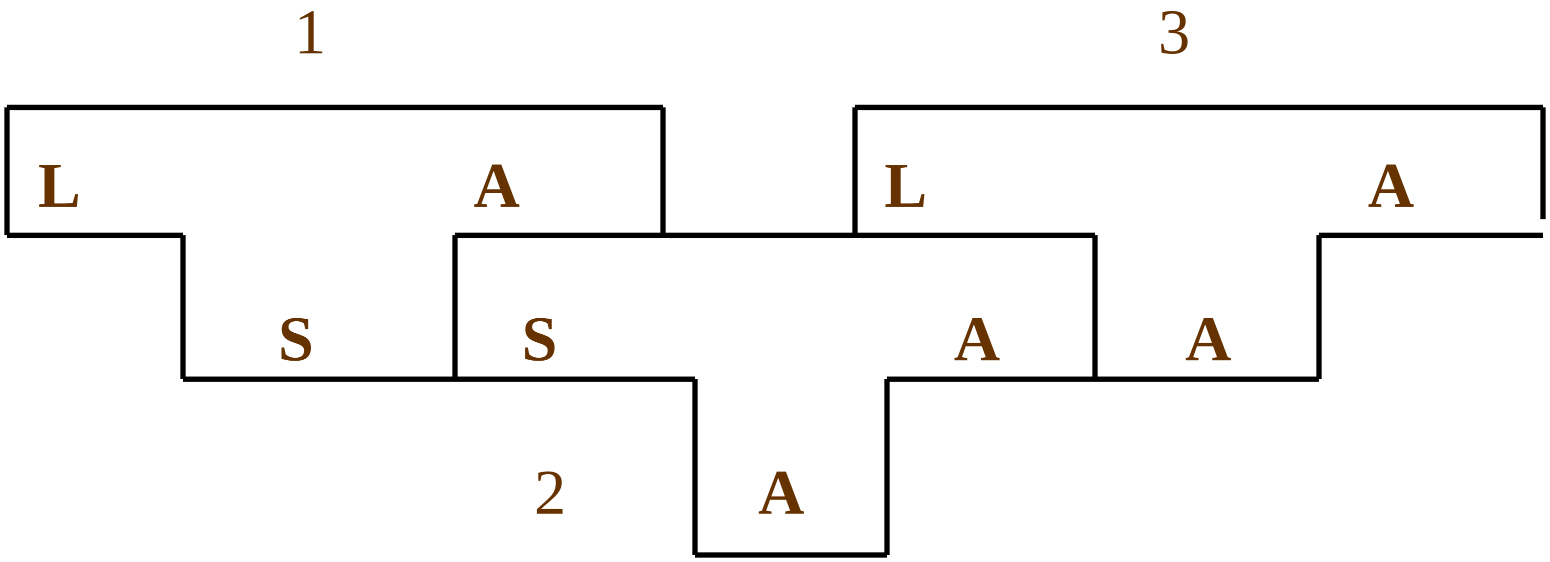
（1） 中间T图的两臂上的语言分别与左右两个T图脚上的语言相同。

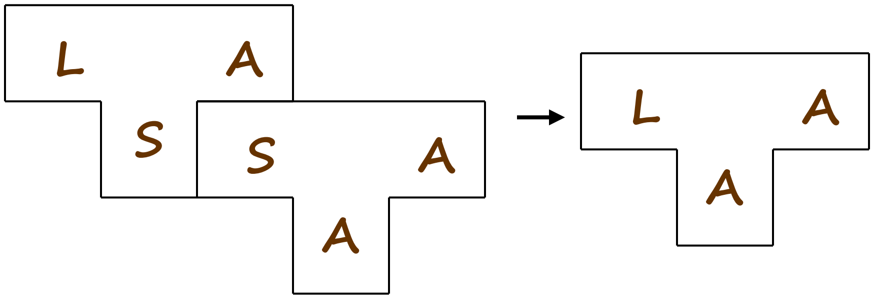
（2） 对于左右两个T图而言，其两个左端的语言必须相同，两个右端的语言亦必须相同，

用第2个编译去编译第1个，得到第3个编译程序



用第2个编译去编译第1个，得到第3个编译程序

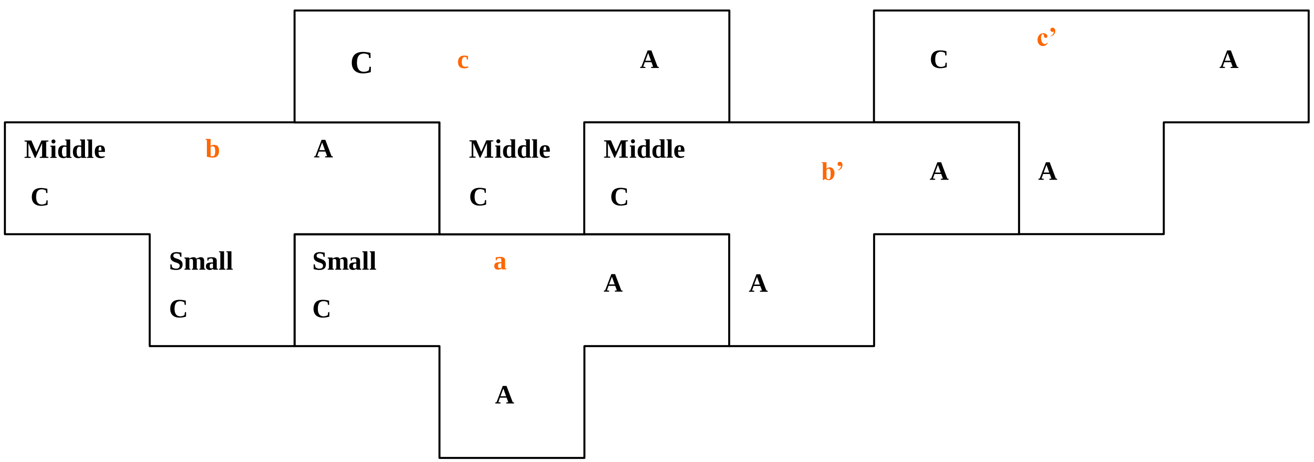




- 如何获得A机器上第一个可书写编译程序的高级语言？
- 是否一定要用低级语言来编写？
- 有两个便捷的途径
  - 自展
  - 移植

自展

- 1）在A机器上用机器语言或汇编语言编写高级语言的子集，例如，Small C的编译程序(a）
- 2）经Small C 编写的Middle C的编译程序（b）
- 3）经Small C 的编译程序编译后，得Middle C的编译程序（b’）
- 4）由Middle C 编写全C的编译程序（c）
- 5）经Middle C 的编译程序编译后，得C的编译程序（c’）

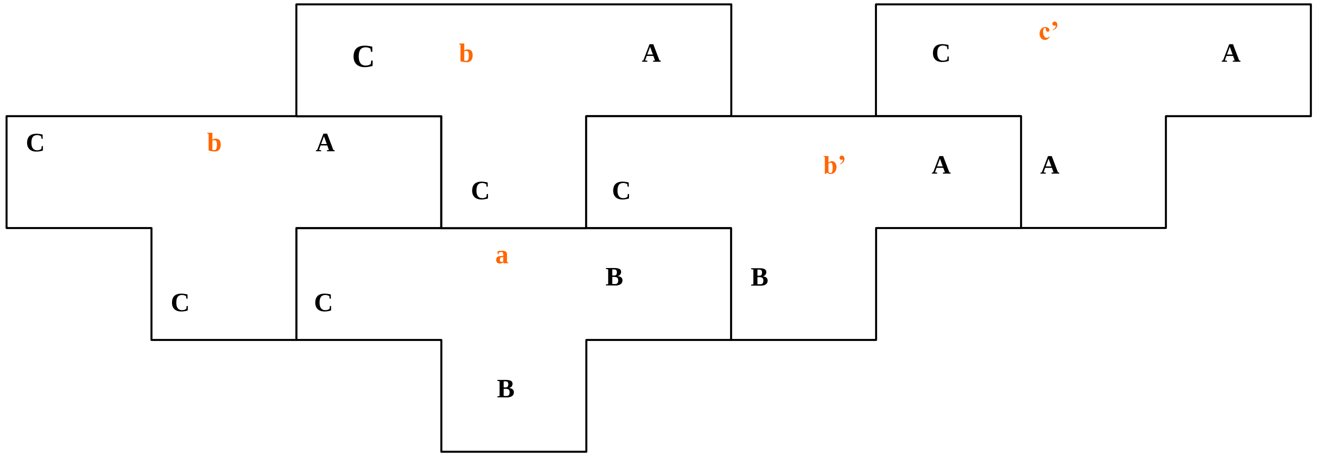


自 展

- 自展方式就像滚雪球那样扩大语言的编译程序。
- 利用自展的方式使我们用机器语言或汇编语言编写高级语言的编译程序的工作量减少到最低限度。
- 注意：实际实现要做多次的自编译才能得到可靠的编译。

移 植

- 在B机器上已有一个可运行的C编译程序（a）
- 只要我们编写一个用C语言书写的，产生A机器代码的C编译程序（b）,按如下T型图的运算去做，就得到A机器上所要的C编译程序。
- (a), (b) ⟶ (b’);　　移植；　　(b), (b’) ⟶ (c’);



移 植

- 用高级语言书写的编译程序，生产的周期短，可靠性高，易修改、扩充与维护，并且易于移植。
- 代码量长，但随着存储能力的提高、运行速度越来越快，程序正确性是主要矛盾。

其它设计方案：

UNIX操作系统的实用程序 LEX，YACC，自动地生成词法分析器和语法分析器。

可采用自编译的方法生成编译程序。

遍：编译程序对源程序或中间形式扫描一次叫一遍.从外存取上一次的中间结果工作后，将结果存于外存之中。

典型编译器可以划分的主要逻辑阶段。

《编译程序》的设计目标

（1）生成尽量小的目标程序。

（2）目标程序运行速度尽量快。

（3）编译程序尽可能小。

（4）编译所花的时间尽可能少。

（5）有较强的查错和改正错误的能力。

（6）可靠性好。

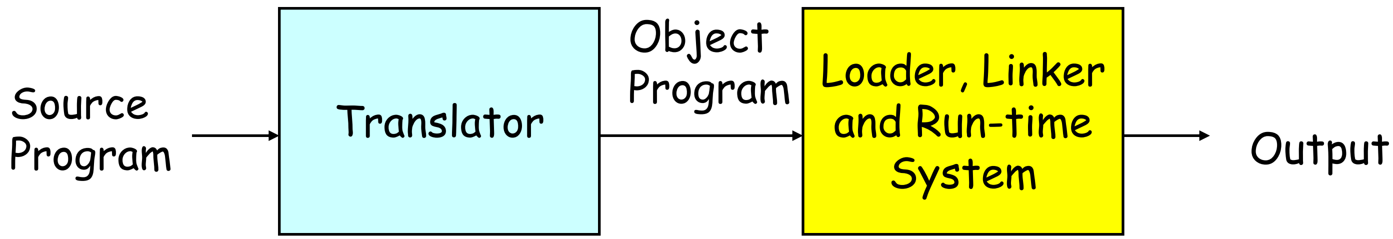
第1章：编译原理引论

- ⌘1.0 课程简介
- ⌘1.1 编译与解释
- ⌘1.2 编译程序概述
- ⌘1.3 编译程序的结构
- ⌘1.4 编译程序的设计与实现
- ⌘**1.5 Summary (Compiler Structure)**

1.5 Summary (Compiler Structure)

- ⌘ 编译、翻译与解释
- ⌘ 源语言，目标语言
- ⌘ 编译程序概念与功能
- ⌘ 编译实例
- ⌘ 编译程序的结构
- ⌘ 编译器的主要逻辑阶段，各阶段的功能
- ⌘ 编译程序的设计与实现
- ⌘ 学习编译原理的意义

附：扩展知识 (Compiler Structure)



- **Source language**
  - **Fortran, Pascal, C, C++**
- **Target language**
  - **Machine code, assembly**
  - **High-level languages, simply actions**
- **Compile time vs run time**
  - **Compile time or statically – Positioning of variables**
  - **Run time or dynamically –heap allocation**

Why Compilers?

- Compiler
  - A program that translates from 1 language to another
  - It must preserve semantics of the source
  - It should create an efficient version of the target language
- In the beginning, there was machine language
  - Ugly – writing code, debugging
  - Then came textual assembly
  - High-level languages – Fortran, Pascal, C, C++
  - Machine structures became too complex and software management too difficult to continue with low-level languages

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation

Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.

source program → COMPILER → target program

(Normally a program written in
a high-level programming language)

(Normally the equivalent program in
machine code – relocatable object file)

↓

error messages

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  – Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  – Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

- Each phase transforms the source program from one representation into another representation.
- They communicate with error handlers.
- They communicate with the symbol table.

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

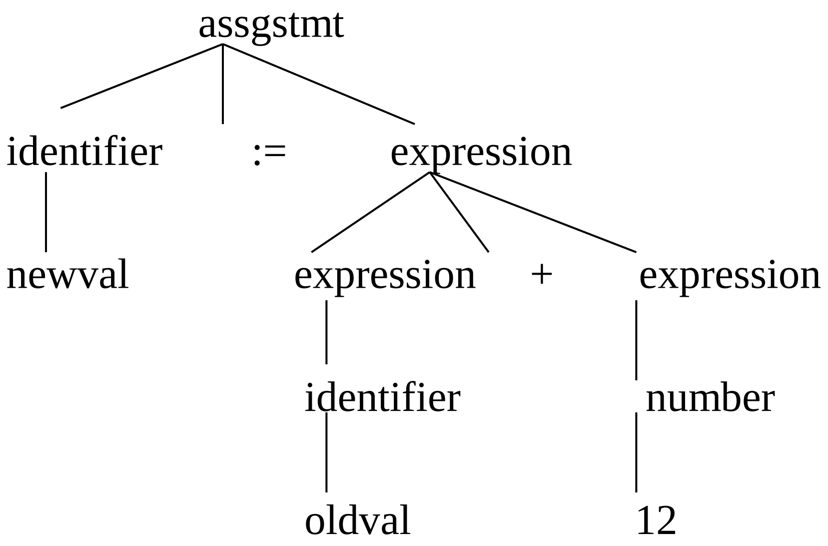
  Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

```
assgstmt    ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  +  expression
```

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest meaningful units (tokens) in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize meaningful structures in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - *Top-Down Parsing, Bottom-Up Parsing*
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

    - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- Ex:

newval := oldval * fact + 1

↓

id1 := id2 * id3 + 1

↓

MULT	id2,id3,temp1	*Intermediates Codes (Quadraples)*
ADD	temp1,#1,temp2	
MOV	temp2,,id1	

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

- Ex:

  (assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```