第一讲

✧ 课程概述

编译原理

Principles and Practice of Compiler Construction

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - 目标

      通过渐进式开发来逐步完成一个完整编译器

      掌握实现一个编译器的完整开发过程
    - 过程

      6个阶段（12个步骤）
    - 编程语言+工具

      缺省：Python + ANTLR4

      其它：向助教提出申请，助教认可后完成实验

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA1： 一个完整编译器

      step1 仅一个 return 的 main 函数

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA2： 常量表达式

      step2 一元计算操作

      step3 二元计算操作

      step4 比较和逻辑表达式

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA3： 变量和语句

      step5 局部变量和赋值

      step6 if 语句和条件表达式

int a, int b.

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA4： 作用域与控制语句

      step7 作用域和块语句

      step8 循环语句

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA4： 作用域与控制语句

      step7 作用域和块语句

      step8 循环语句

实 验 计 划

- 基础实验项目
  - 实现一个小型语言minidecaf（C的小子集）
    - PA5： 函数和全局变量

      step9 函数

      step10 全局变量

实 验 计 划

- ✧ 基础实验项目
  - – 实现一个小型语言minidecaf（C的小子集）
    - PA6： 指针和数组

      step11 指针

      step12 数组

生成 RISC-V 目标代码
→ 用模拟器执行（不用在 RISC-V 芯片上执行）
（qemu）

实 验 计 划

✧ 扩展实验项目

– 可以参加的前提条件

课程前4周已完成基础实验，并通过老师+助教的评审

– 内容

参考教师/助教推荐选题或自荐，至少包含词法、语法和静态语义分析及生成可（模拟）执行代码等环节

– 过程

第1~4周：准备计划；第5周：评估答辩；
之后每周完成简明周进展报告，进行一次集中交流
考试周开始前一周提交总结报告和项目代码
考试周前后组织答辩（或延后）

考 核 计 划

- 成绩分布 （100 ）
  - 原理部分书面作业 +出勤（10 %）
  - 基本实验成绩（20%）
  - 期末考试
    - 如没有参加扩展实验 （70%）
    - 如参加了扩展实验，选择如下两者的最大值：

      扩展实验（50%）+期末考试（20%）

      期末考试（70%）

答疑与交流

- 通过网络
  - 清华网络学堂（课程讨论区）

    问题探讨
  - 电子邮件
- 面对面 （老师答疑可预约）
  - 助教固定答疑时间（节假日除外）

    待定
  - 地点

    东主楼 10 区 209 室

答疑与交流

✧ 课程微信群

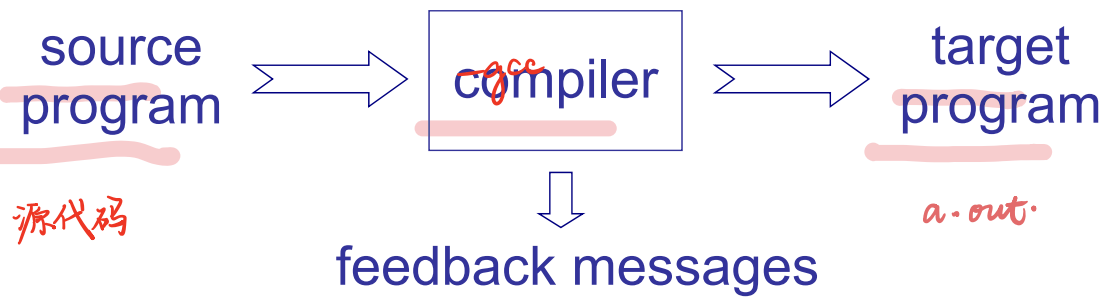
编译程序（系统）概述

- ✧ 什么是编译程序
- ✧ 编译程序的逻辑结构
- ✧ 编译程序的组织
- ✧ 编译程序的伙伴程序
- ✧ 编译程序与 T 型图

什么是编译程序

- 从基本功能来看，编译程序（*Compiler*）是一种翻译程序（*Translator*）
  - 将语言*A*的程序翻译为语言*B*的程序
  - 称语言*A*为源语言 （*Source Language*）
  - 称语言*B*为目标语言 （*Target Language*）

C语言→RISC-V.



什么是编译程序

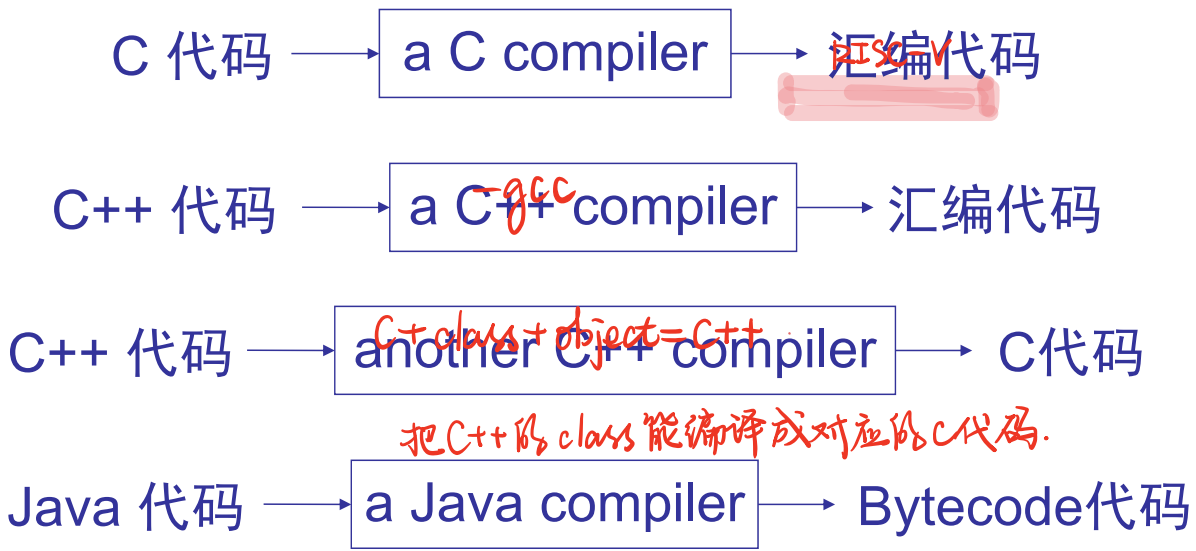
- 编译程序是较为复杂的翻译程序
  - 需要对源程序进行分析（*Analysis*）

    识别源程序的语法结构信息，理解源程序的语义信息，反馈相应的出错信息
  - 根据分析结果及目标信息进行综合（*Synthesis*）

    生成语义上等价于源程序的目标程序
- 较为简单的翻译程序如：
  - 预处理程序（*Preprocessor*）
  - 汇编程序（*Assembler*）

什么是编译程序

- 编译程序通常是从较高级语言的程序翻译至较低级语言的程序，如



什么是编译程序

- 传统的编译程序
  - 源语言通常为高级语言（*High-Level Programming Languages*）

    Fortran, Algol, C, Pascal, Ada, C++, Java, Lisp, Prolog, Python…
  - 目标语言通常为机器级语言（*Machine-Level Languages*）或较低级的虚拟机语言

    汇编语言（*Assembly Languages*）

    机器语言（*Machine Languages*）

    Bytecode（*Java* 虚拟机语言）

什么是编译程序

- 编程语言的主要范型（*Paradigms*）
  - 命令式语言（*Imperative Languages*）
    - 描述问题如何实现（*how it to be done*）
    - 程序具有状态,通过语句改变程序状态
    - *Fortran, Algol, C, C++, Pascal, Basic, Java, C#, …*
  - 陈述式语言（*Declarative Languages*）
    - 描述问题做什么（*what it to be done*）
    - 程序无状态（对纯的陈述式语言而言）
    - 函数式（*Functional*）:*Lisp, Scheme, Haskell, ML, Caml, …*
    - 逻辑型（*Logic*）: *Prolog, …*

什么是编译程序

- 编程语言的主要范型（*Paradigms*）
  - 面向对象语言（*Object-Oriented Languages*）

    基于对象（*object-based*，类,对象及对象间交互）

    面向对象（*object-oriented*，类,对象,对象间交互,继承及多态）

    如：*Smalltalk, Simula67, Java, C++, C#, …*
  - 并发/并行/分布式语言

    （*Concurrent / Parallel / Distributed Languages*）

    Ada, Java, Modula-3, Linda, HPF, OpenMP, MPI, CUDA, …

    进程/线程/任务间通信：基于共享内存（*memory/variable-sharing,* 如*OpenMP , Java* ）,基于消息传递（*message passing,* 如*MPI*），基于远方过程调用（*remote procedure/method call,* 如*Ada, Java*），基于数据并行（*data parallel,*如*HPF*）

什么是编译程序

- 编程语言的主要范型（*Paradigms*）
  - 其他

    同步语言(*Synchronous Languages*）：面向实时控制，时钟周期同步，含时钟（*clock*）和时态(*temporal*)算子，如*Signal, Lustre…*

    数据库语言(*database language*): *SQL,…*

    脚本语言(*Scripting Languages*）：解释型语言,显式的 *glue together* 算子,如 *Perl, PHP, Python, Javascript…*

  - 趋势：多范型融合

    Java （低版本：并发，命令式面向对象；高版本：新增函数式）

    Rust （混合范型：并发，面向对象，命令式，函数式）

什么是编译程序

- 编译架构（*Compiler Infrastructure*）
  - 共享的编译程序研究/开发平台
    - SUIF (Stanford)
    - Zephyr (Virginia and Princeton)
    - IMPACT, LLVM (UIUC)
    - GCC （GNU Compiler Collection ）
    - Open64（SGI, 中科院计算所, Intel, HP, Delaware, 清华, …）
    - 方舟编译器（华为，https://www.openarkcompiler.cn ）
  - 多源语言多目标机体系结构
    - 如 GCC有C, C++, Objective C, Fortran, Ada, and Java ， …
    - 等诸多前端，以及支持30多类体系结构、上百种平台的后端
  - 多级中间表示
    - 如 Open64 的中间表示语言 WHIRL分5个级别

编译程序的逻辑结构

- 编译程序逻辑结构上至少包含两大阶段
  - 分析（*Analysis*）阶段

    理解源程序，挖掘源程序的语义
  - 综合（*Synthesis*）阶段

    生成与源程序语义上等价的目标程序

编译程序的逻辑结构

✧ 编译程序的前端、中端和后端

- 前端（*Front End*）

  实现主要的分析任务

  通常以第一次生成中间代码为标志

- 后端（*Back End*）

  实现主要的综合任务（目标代码生成和优化）

  通常以从最后一级中间代码生成目标代码为标志

- 中端（*Middle End*）

  实现各级中间代码上的操作（中间代码生成与优化）

编译程序的逻辑结构

✧ 典型编译程序的逻辑过程

编译程序的逻辑结构

✧ 词法分析

单词流→字符流

– 扫描源程序字符流，识别出有词法意义的单词，返回单词的类别和单词的值，或词法错误信息

```
class Main {
   static void main() {
      Print("hello world");
   }
}
```

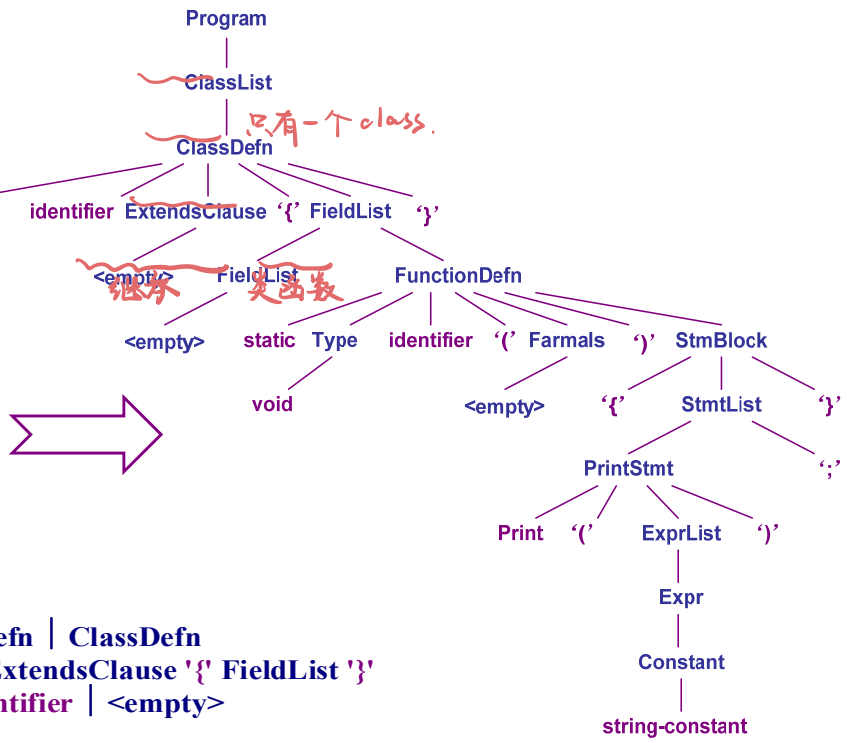
单词类别	单词值
保留字 class	
标识符	Main
分隔符 {	
保留字 static	
保留字 void	
标识符	main
分隔符 (	
分隔符)	
分隔符 {	
保留字 Print	
分隔符 (	
字符串常量	“hello world”
分隔符)	
分隔符 ;	
分隔符 }	
分隔符 }	

编译程序的逻辑结构

✧ 语法分析

生成语法分析树

```
class Main {
   static void main() {
      Print("hello world");
   }
}
```



Program → ClassList
ClassList → ClassList ClassDefn | ClassDefn
ClassDefn → class identifier ExtendsClause '{' FieldList '}'
ExtendsClause → extends identifier | <empty>
……

编译程序的逻辑结构

✧ 语义分析

– 对语法分析后的程序进行语义分析,不符合语义规则时给出语义错误信息

类型检查.

```
class Main {
   static void main() {
      int a;
      a="hello world";
      Print(a);
   }
}
```

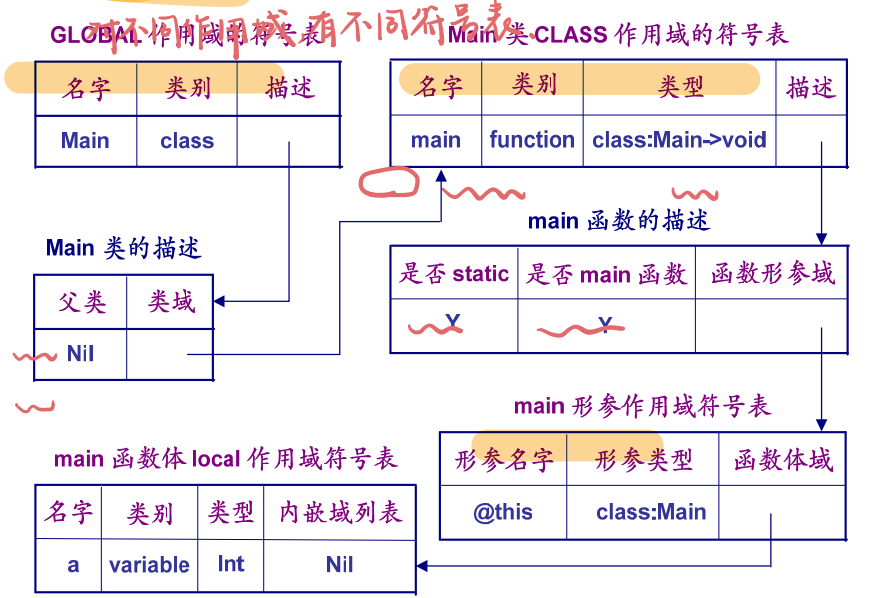
incompatible operands: int = string.

编译程序的逻辑结构

✧ 符号表

– 收集每个名字的各种属性用于语义分析及后续各阶段

```
class Main {
   static void main() {
      int a;
      a="hello world";
      Print(a);
   }
}
```



编译程序的逻辑结构

- 出错处理
  - 检查错误
    - 报告出错信息（*error reporting*）
  - 排错
    - 恢复编译工作（*error recovery*）

修复错误

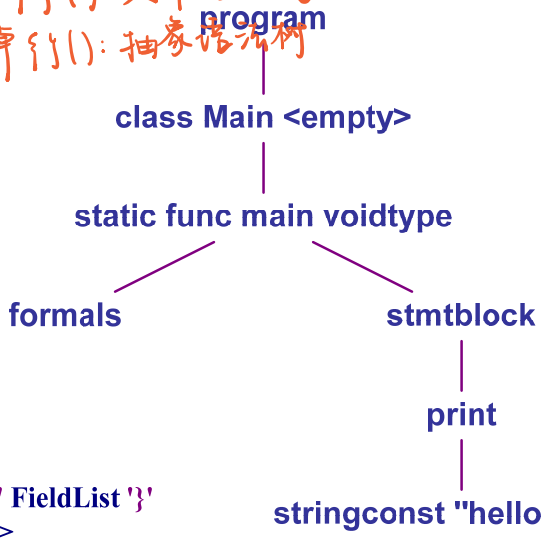
编译程序的逻辑结构

✧ 中间代码生成

– 抽象语法树 *AST*

带有{}():具体语法树
拿掉{}():抽象语法树

```
class Main {
   static void main() {
      Print("hello world");
   }
}
```



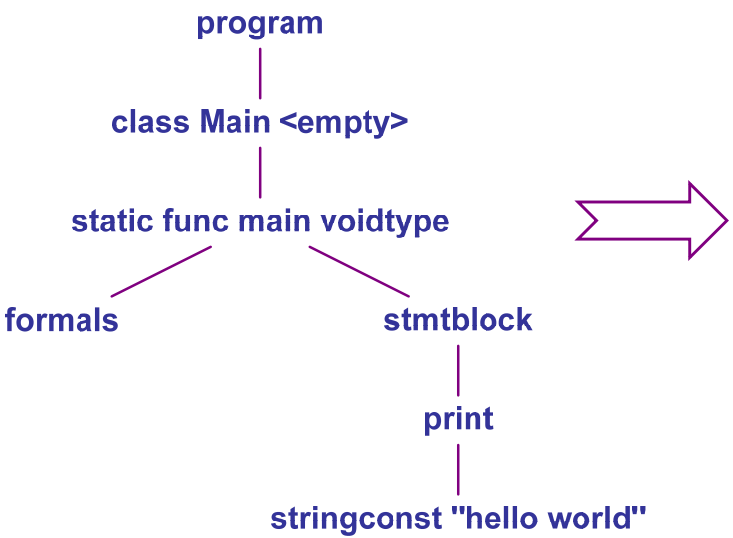
Program → ClassList
ClassList → ClassList ClassDefn | ClassDefn
ClassDefn → class identifier ExtendsClause '{' FieldList '}'
ExtendsClause → extends identifier | <empty>
……

PrintStmt → print '(' ExprList ')'
……

编译程序的逻辑结构

✧ 中间代码生成 IR

– 三地址码 *TAC*




```
VTABLE(_Main) {
    <empty>
    Main
}

FUNCTION(_Main_New) {
memo ''
_Main_New:
    _T0 = 4
    parm _T0
    _T1 =  call _Alloc
    _T2 = VTBL <_Main>
    *(_T1 + 0) = _T2
    return _T1
}

FUNCTION(main) {
memo ''
main:
    _T3 = "hello world"
    parm _T3
    call _PrintString
}
```

编译程序的逻辑结构

✧ 目标代码生成

– 生成目标机代码

```
VTABLE(_Main) {
   <empty>
   Main
}
FUNCTION(_Main_New) {
memo ''
_Main_New:
   _T0 = 4
   parm _T0
   _T1 =  call _Alloc
   _T2 = VTBL <_Main>
   *(_T1 + 0) = _T2
   return _T1
}
FUNCTION(main) {
memo ''
main:
   _T3 = "hello world"
   parm _T3
   call _PrintString
}
```

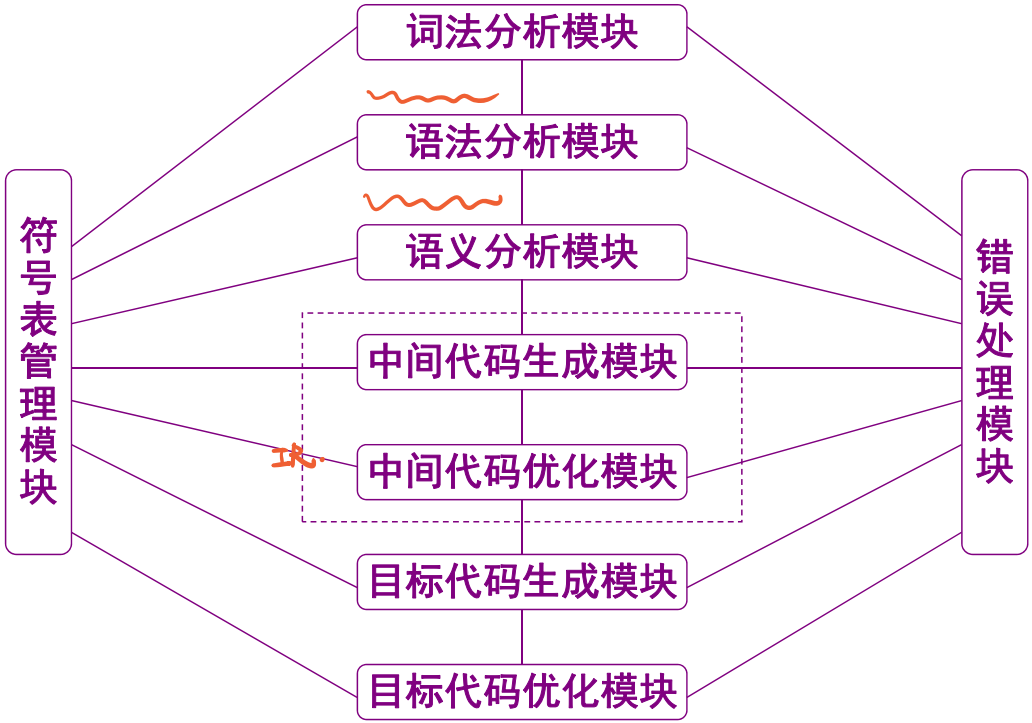
MIPS 汇编码

```
        .data
        .globl main

        .data
        .align 2
_Main:                      # virtual table
        .word 0             # parent
        .word _STRING0      # class name


        .text
_Main_New:      # function entry
        sw $fp, 0($sp)
        sw $ra, -4($sp)
        move $fp, $sp
        addiu $sp, $sp, -16
_L14:
        li   $t0, 4
        sw   $t0, 4($sp)
        jal  _Alloc
        move $t0, $v0
        la   $t1, _Main
        sw   $t1, 0($t0)
        move $v0, $t0
        move $sp, $fp
        lw   $ra, -4($fp)
        lw   $fp, 0($fp)
        jr   $ra
```

```
main:    # function entry
        sw $fp, 0($sp)
        sw $ra, -4($sp)
        move $fp, $sp
        addiu $sp, $sp, -16
_L15:
        la   $t0, _STRING1
        sw   $t0, 4($sp)
        jal  _PrintString
        move $sp, $fp
        lw   $ra, -4($fp)
        lw   $fp, 0($fp)
        jr   $ra


        .data
_STRING0:
        .asciiz "Main"
_STRING1:
        .asciiz "hello world"
```

编译程序的逻辑结构

✧ 小结：典型编译程序的主要逻辑模块

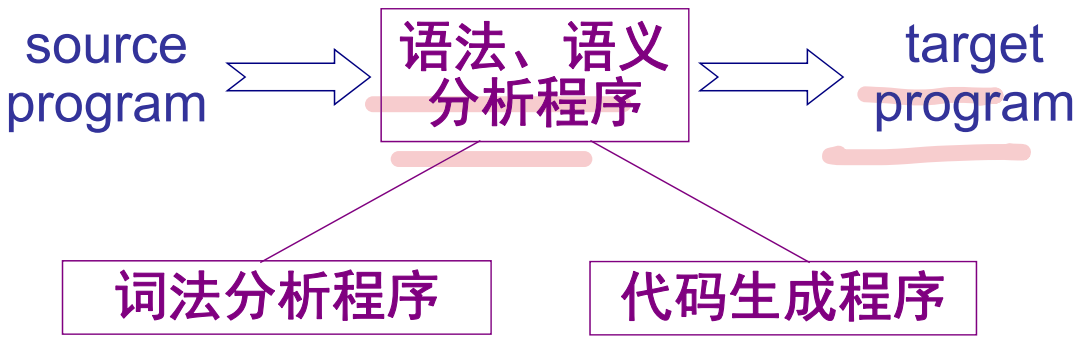


编译程序的组织

- 编译程序的遍（*Passes / Phases*）
  - 对一种代码形式从头到尾扫描一遍
  - 将一个代码空间变换到另一个代码空间
  - 代码空间 = 代码 + 符号表 + 其他有用信息
- 编译程序的组织取决于各遍的组织
  - 单遍编译程序，多遍编译程序
  - 多个遍之间有逻辑上的先后关系
  - 多个遍的实现可采用顺序结构或并发结构（后者不常用）

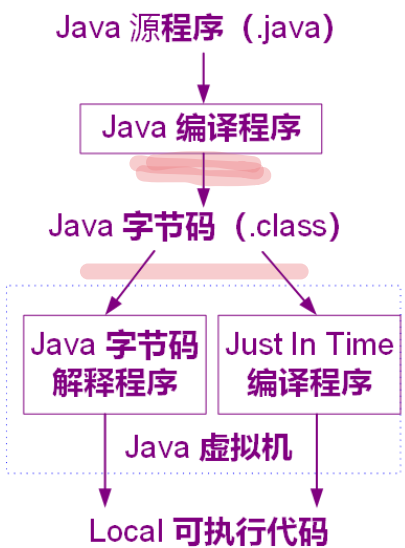
编译程序的组织

- 例：一个以语法、语义分析程序为中心的单遍编译程序组织

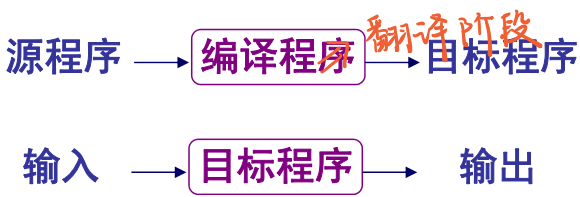


编译程序的伙伴程序

- 解释程序（Interpreter）
  - 不产生目标程序文件
  - 不区别翻译阶段和执行阶段
  - 翻译源程序的每条语句后直接执行
  - 程序执行期间一直有解释程序守候
  - 常用于实现虚拟机



- 比较编译程序和解释程序



编译程序的伙伴程序

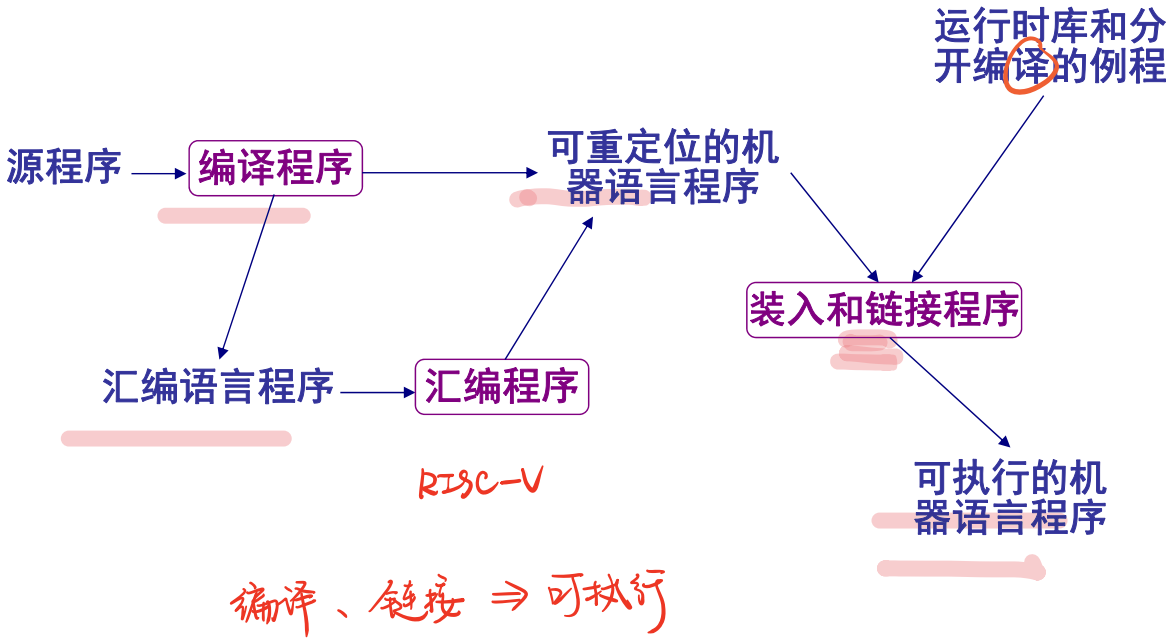
- 预处理程序（*Preprocessor*）
  - 支持宏定义（*Macro definition*）
    如C源程序中 #define 行的处理
  - 支持文件包含（*File inclusion*）
    如C源程序中 #include 行的处理
  - 支持其他更复杂的源程序扩展信息
- 预处理程序和编译程序的关系

编译程序的伙伴程序

- 汇编程序（*Assembler*）
  - 翻译汇编语言程序至可重定位的（*Relocatable*）机器语言程序
- 装入和链接程序（*Loader and Link-editor*）
  - 装入程序对可重定位机器语言程序进行修改
    将相对地址变换为机器绝对地址
  - 链接程序合并多个可重定位机器语言程序文件到同一个程序
  - 装入和链接程序产生最终可执行的机器语言程序

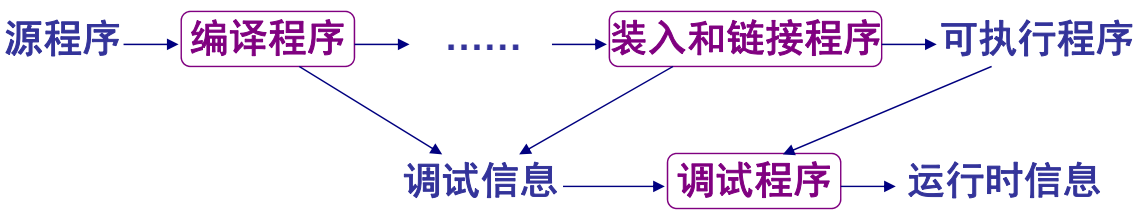
编译程序的伙伴程序

- 编译程序、汇编程序及装入和链接程序之间的典型关系



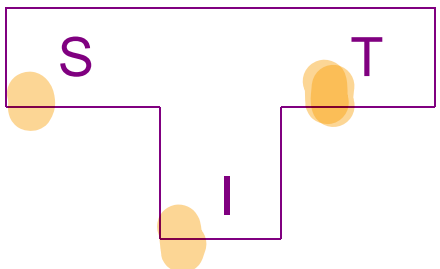
编译程序的伙伴程序

- 调试程序（*Debugger*）
  - 反馈目标程序运行时信息
  - 将目标程序运行时信息与源程序关联
  - 断点管理、单步跟踪、读/写目标机状态等功能
- 调试程序和编译程序的关系



编译程序与 T 型图

✧ T-型图 (表示一个编译程序)



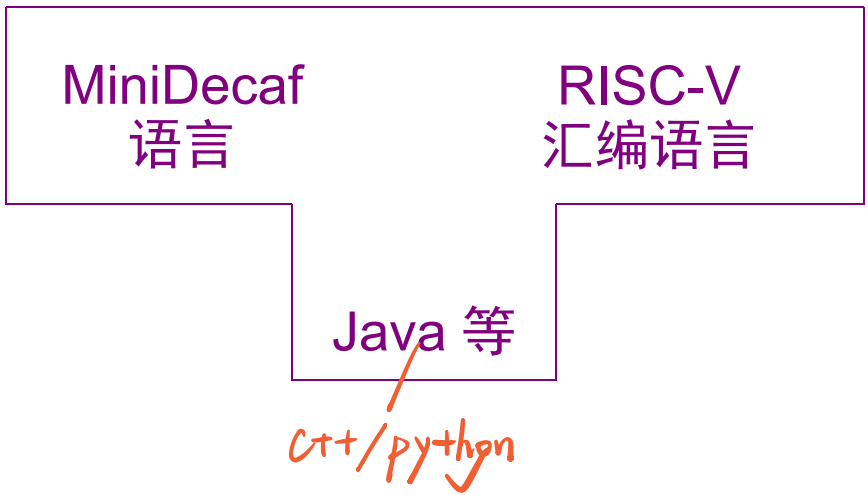
S：编译程序所实现的源语言

T：目标语言

I： 编译程序的实现语言

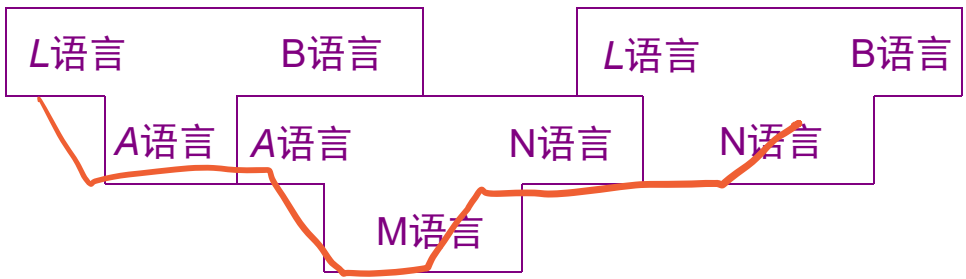
编译程序与 T 型图

✧ 例: minidecaf 项目中编译程序 T-型图



编译程序与 T 型图

✧ T-型图的叠加



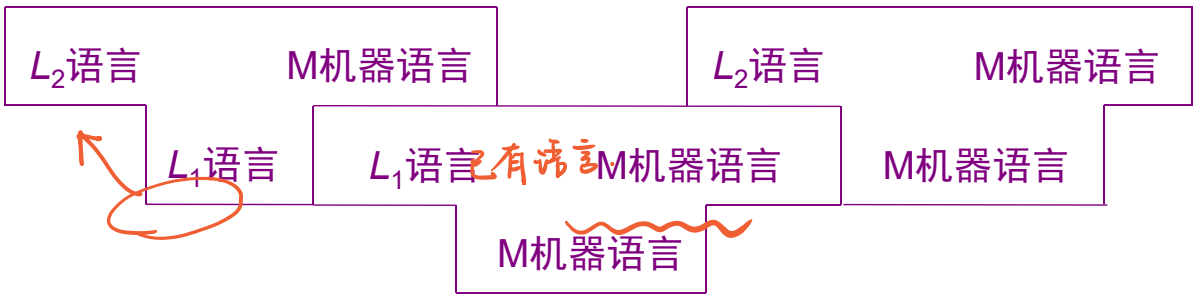
编译程序与 T 型图

- (M 机器上运行的)本地编译器

- (M 机器上运行的)交叉编译器(实验)

编译程序与 T 型图

✧ 用已有的语言 L_1 实现新的语言 L_2



步骤：

在M机器上编译，在M机器上执行.

(1）用L_1 语言编写L_2 语言到M 机器语言的编译程序

(2）将该L_2 语言编译程序用L_1 语言编译程序进行编译

编译程序与 T 型图

✧ 编译程序的移植

将机器 A 上的语言 *L* 移植到机器 B，步骤：(1）用*L* 语言编写 *L* 语言到B机器语言的编译程序X; (2）用*L*编译程序对X进行编译，产生一个能在机器 A 上运行的产生 B 机器代码的编译程序Y（**交叉编译程序**）；(3）再用Y 对X进行编译，得到可以在机器 B 上运行的*L* 语言编译程序

教学内容预览

✧ 教学形式

– 课内学习和课外学习内容互补

原理 + 技术 + 工具

课内　　课外

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

基本概念
逻辑结构
组织方式
伙伴程序
生成环境
2学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

词法分析基础
1 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

自顶向下语法分析
3 学时

自底向上语法分析
5 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

基于属性文法和翻译模式进行语义计算的基本原理及实现技术

3.5 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

强调作用域及其组织方式

1 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

以类型检查程序设计为重点

1.5学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

以常用语言机制的实现技术为主线

3.5 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

存储布局,
存储分配策略,
活动记录,
过程实现,
面向对象程序/
函数式程序存
储组织, ……
3 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

以简单但完整的指令选择、寄存器分配过程为主线

3 学时

教学内容预览

✧ 教学内容

– 课堂教学内容及课时计划

- 编译程序/系统概述
- 词法分析
- 语法分析
- 语法制导的语义计算基础
- 符号表组织
- 语义分析
- 中间代码生成
- 运行时存储组织
- 目标代码生成
- 代码优化

以基本块内的简单优化方法、控制流数据流分析基础等代码生成和优化相关的基本知识为主线，辅以优化技术的综述

2.5 学时

教学内容预览

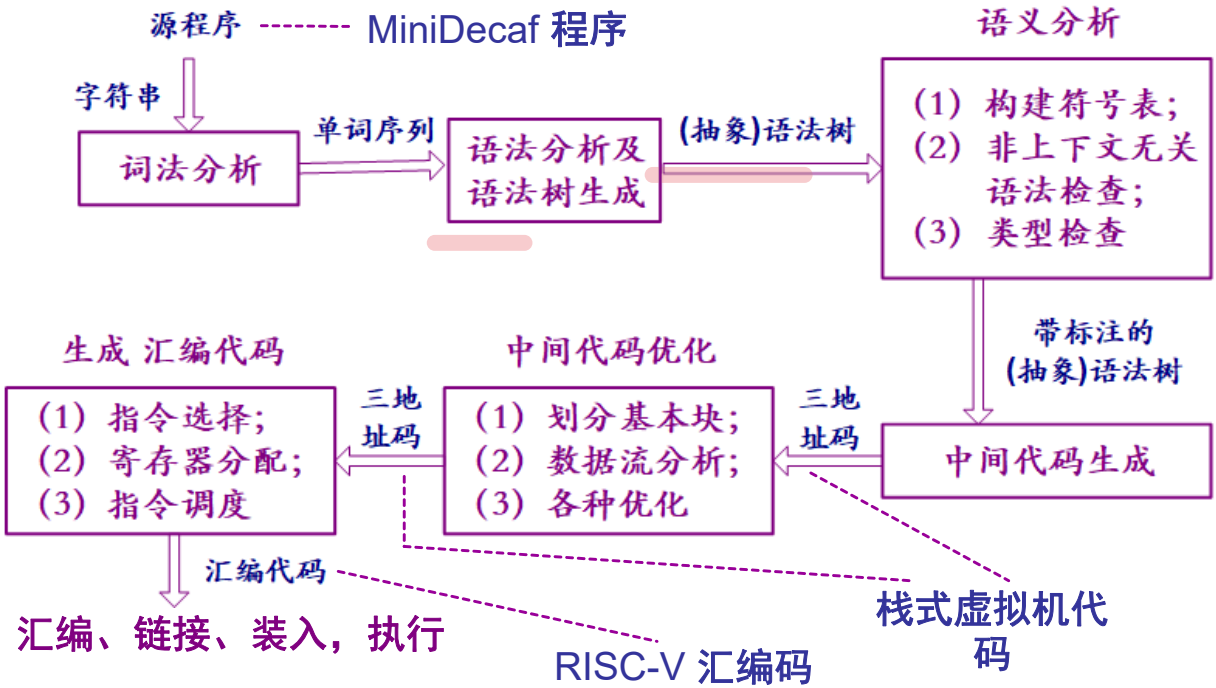
✧ 教学内容（实践部分）

– minidecaf 项目（PA1-PA6）

- 词法和语法分析
  产生（抽象）语法树（参考助教实现或自定义）
- 静态语义分析
  遍历（抽象）语法树构造符号表（可选）、实现静态语义分析、产生带标注的抽象语法树（可选）
- 中间代码生成
  生成中间表示（可选，通常是需要的）
- 数据流分析及优化（选做）
- 目标代码生成（必做）及优化（选做）

教学内容预览

✧ 实验项目minidecaf与课堂教学的关联



课后作业

1. 学习或复习《形式语言与自动机》
2. 准备实验相关工具与开发环境
3. 查阅实验指导书以及相关工具（如 ANTLR4）的技术文档

That's all for today.

Thank You