

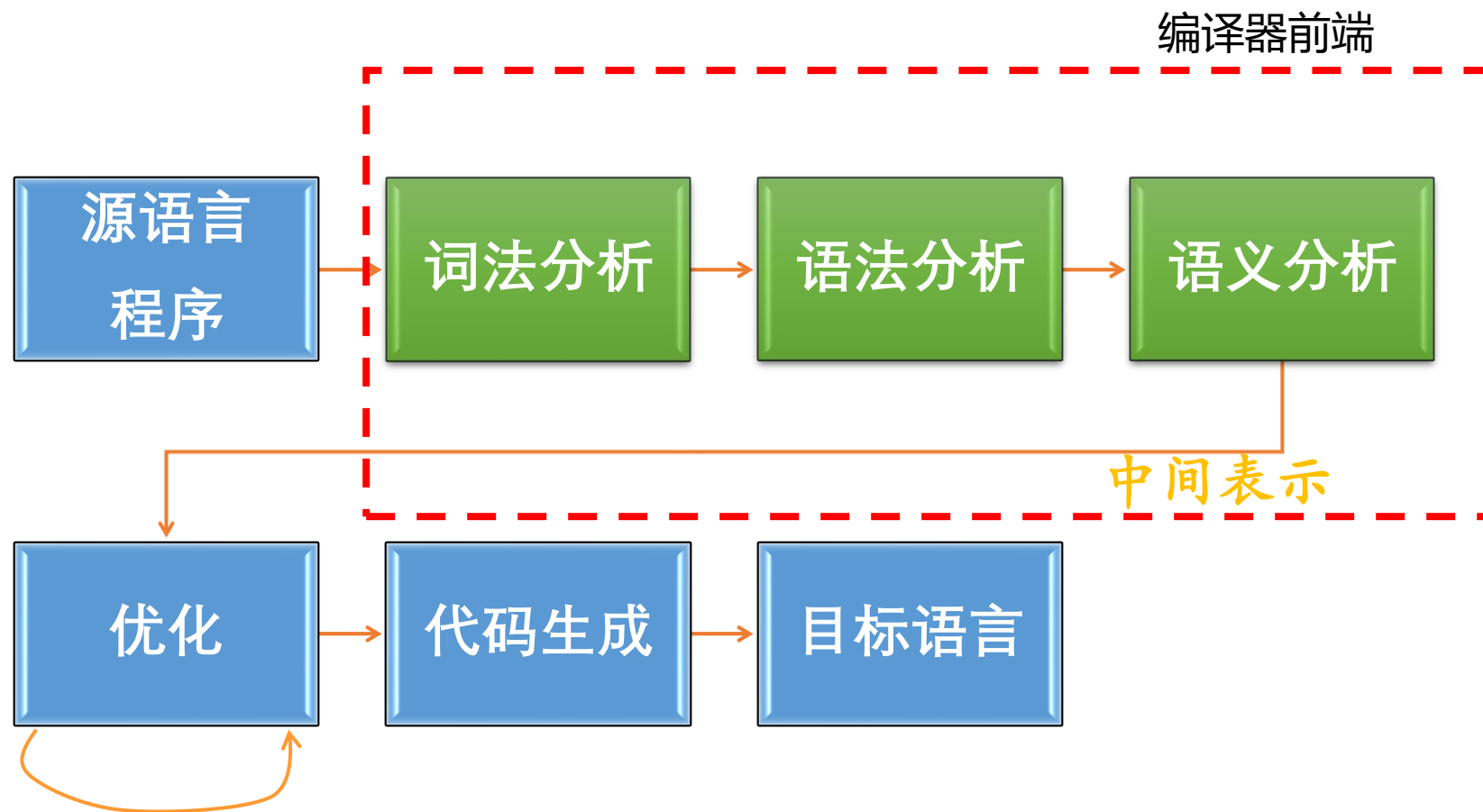
# 并行编译与优化 Parallel Compiler and Optimization

计算机研究所编译室

*Lecture Three Compiler Front:  
Semantic Analysis and Internal  
Representation*

**第三课 编译器前端：语义分析和  
中间表示**

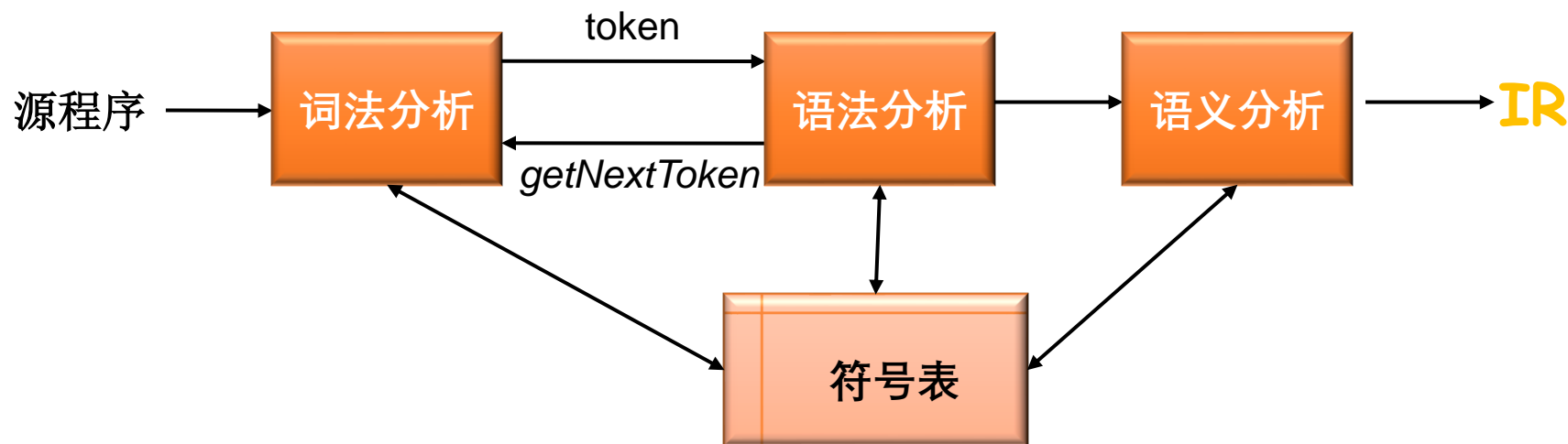
# 复习：编译器结构



# 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）



# 复习：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 $lex, flex, ANLTR$ 等工具来构建DFA，也可以手工进行

# 复习：语法分析器的构建

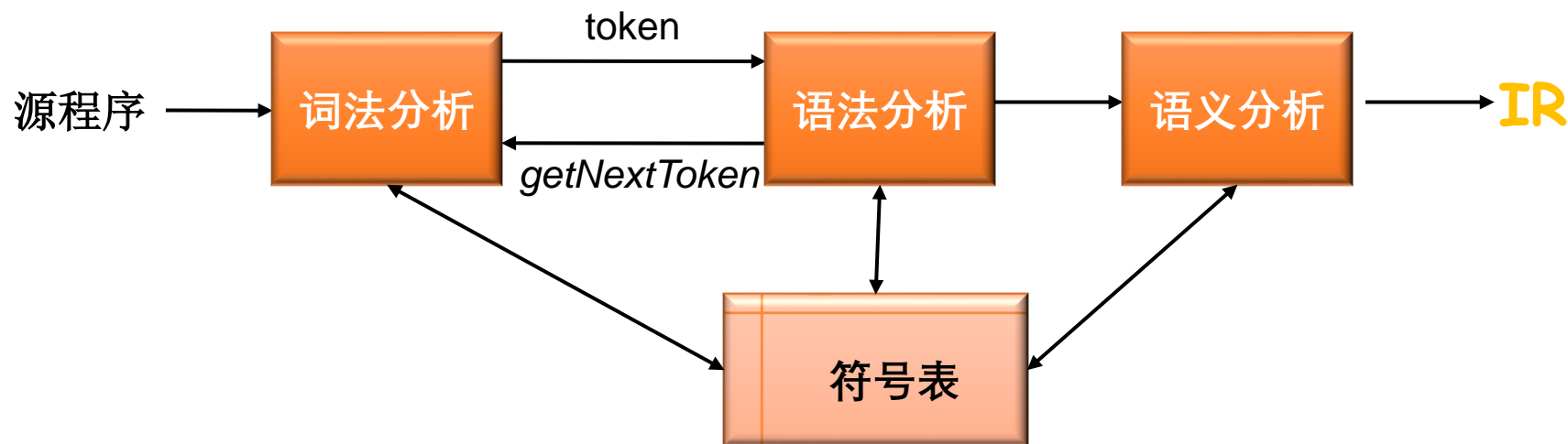
- 使用上下文无关文法定义语言的语法规则
- 选择解析器类型

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:yacc,ANTLR

# 内容

- 翻译成中间表示
- 语义分析
  - ⊕ 做什么?
  - ⊕ 怎么做?
- 中间表示



# 翻译成中间表示

## ■ 语法指导的翻译 (Syntax Directed Translation)

- ⊕ 语法分析器驱动的翻译：为文法中每个产生式配上一组语义规则，并且在语法分析的同时执行这些语义规则

## ■ 模型驱动的翻译 (Model-driven Translation)

- ⊕ 创建抽象语法树 (AST) 模型，遍历时进行翻译



# 1.1 语法指导的翻译

## ■ 语法指导的翻译 (Syntax Directed Translation)

### ⊕ 语法分析器驱动的翻译

```
%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;
}
```

## 1.1 语法指导的翻译

### ■ 语法指导的翻译 (Syntax Directed Translation)

⊕ 语法分析器驱动的翻译

### ■ 属性文法 (Attribute Grammars, 属性翻译文法)

⊕ 在上下文无关文法的基础上，为每个文法符号（终结符或非终结符）配备若干相关的“值”（属性）。

- 属性代表与文法符号相关信息，如类型、值、代码序列、符号表内容等
- 属性可以进行计算和传递
- **语义规则**：对于文法的每个产生式都配备了一组属性的计算规则

## 1.1 语法指导的翻译

### ■ 语法指导的翻译 (Syntax Directed Translation)

⊕ 语法分析器驱动的翻译

### ■ 属性文法 (Attribute Grammars, 属性翻译文法)

⊕ 在上下文无关文法的基础上，为每个文法符号（终结符或非终结符）配备若干相关的“值”（属性）。

- 属性代表与文法符号相关信息，如类型、值、代码序列、符号表内容等
- 属性可以进行计算和传递
- **语义规则**：对于文法的每个产生式都配备了一组属性的计算规则

## 1.1 语法指导的翻译

### ■ 语法指导的翻译 (Syntax Directed Translation)

⊕ 语法分析器驱动——为文法中每个产生式配上一组语义规则，并且在语法分析的同时执行这些语义规则

### ■ 语义规则被计算的时机

⊕ 在自上而下语法分析中，一个产生式匹配输入串成功时

⊕ 在自下而上分析中，当一个产生式被用于进行归约时

## 2.1 语义分析

- 词法分析和语法分析处理语法问题，源代码是否符合语言定义的语法规范
- 语义问题
  - ⊕ 程序需要完成的操作是什么？
  - ⊕ 如何等价地进行程序翻译

### 语法错误

```
int b 4;  
y x <;  
if w x y z;
```

### 语义错误

```
int b = "hi"; // 类型不匹配  
a = b + 1;    // a未声明
```

## 2.1 语义分析

- 使用语法树和符号表的信息，检查源程序与语言定义的语义一致性
  - ⊕ 类型检查 (type checking)
  - ⊕ 参数正确性
  - ⊕ 其他语言规范要求，例如声明位置和顺序
  - ⊕ 边界检查.....
- 这些检查上下文无关文法无法处理
- 一些语义分析无法在编译时静态完成
  - ⊕ 除0
  - ⊕ 越界
  - ⊕ 解析空指针.....

## 2.1 语义分析

### ■ 示例，对于C程序

检查赋值语句  $lhs = rhs$

- ⊕  $lhs$  是可赋值的
- ⊕  $lhs$  和  $rhs$  具有兼容的类型

检查变量  $id$

- ⊕  $id$  使用前被声明
- ⊕  $id$  作用域
- ⊕ 编译器是否为  $id$  指派了存储空间

## 2.1 语义分析

### ■ 语义分析/语义检查时机

- ⊕ 编译时静态分析
- ⊕ 动态分析

### ■ 除了编译器前端外，静态分析可能在编译器中端分析和优化阶段进行

- ⊕ 控制流分析/数据流分析
- ⊕ 别名分析
- ⊕ 依赖关系分析

### ■ 编译器生成代码，运行时检查


- ⊕ 越界检查



## 2.1 语义分析

### ■ 示例

```
b=1;  
a=2;  
if(b==a) printf("b = %d\n", b);
```



```
b=1;  
a=2;  
if(b=a)  
    printf("b = %d\n", b);
```

```
int *p,*q, *r;  
int a;  
p = (int*)malloc(1024);  
p[0] = 1;  
q=p;  
free(p);  
r = (int *)malloc(1024);  
r[0] = 3;  
a = q[0];  
printf("a = %d\n",a);
```

```
int *p;  
p = (int*)malloc(1024);  
p[256] = 3;
```

## 2.2 语义分析工具

### ■ 类型系统 (Type System)

- ⊕ 对编程语言中的各种成分（如数值、变量、表达式等）赋以类型属性的规则的集合
- ⊕ 类型检查
- ⊕ 在目标代码生成阶段，也需要使用类型检查阶段所收集的类型信息

### ■ 符号表

- ⊕ 记录标识符的各种语义信息

## 2.2.1 类型系统

### ■ 类型表达式

- ⊕ 类型表达式或者是基本类型，或者是由类型构造符（constructor）作用于其他类型表达式构成。
- ⊕ 基本类型和类型构造符都取决于具体的语言
  - ① 一个**基本类型**是一个类型表达式。
    - ⊕ 基本类型随程序语言的不同而不同，但基本都包含**布尔类型**（boolean）、**整型**（integer）、**浮点**（float）、**字符**（char）等常见数据类型。
    - ⊕ 基本类型**void**表示没有数据类型。
    - ⊕ 一个专用的基本类型**type-error**，用于类型检查过程中指示有类型错误

## 2.2.1 类型系统

### ■ 类型表达式

- ⊕ 类型表达式或者是基本类型，或者是由类型构造符（constructor）作用于其他类型表达式构成。
- ⊕ 基本类型和类型构造符都取决于具体的语言
  - ② 类型表达式可以命名，一个类型名是一个类型表达式。
  - ③ 用类型构造符作用于已有的类型表达式，可以得到新的类型表达式
    - ⊕ 例如，C语言中，类型构造符包括数组、指针、函数、结构、联合、枚举等。
  - ④ 类型表达式可以包含变量，变量的值是类型表达式

## 2.2.1 类型系统

### ■ 类型系统

- ⊕ 把类型表达式赋给语言各相关结构成分的规则的集合
  - 形成聚合类型的规则
  - 自动类型转换，等等
- ⊕ 同一种语言的编译程序，基于不同的实现，可以使用不同的类型系统，并可能进行扩充

## 2.2.1 类型系统

### ■ 类型检查

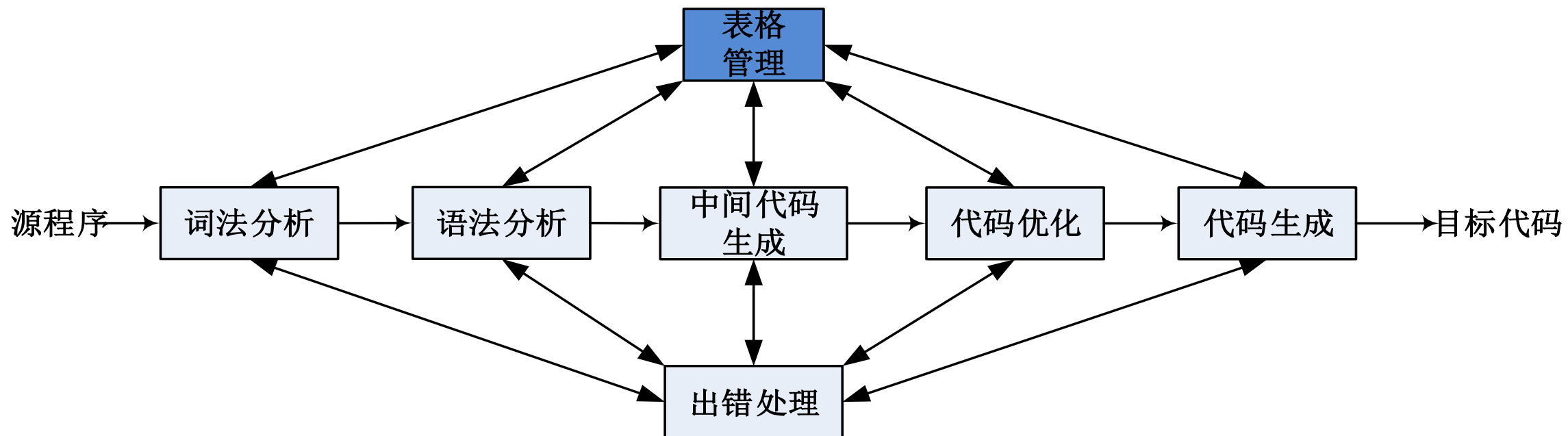
- ⊕ 执行类型系统推导规则的过程称之为类型检查
- ⊕ 编译时进行，则称之为“静态类型检查”
- ⊕ 运行时进行，则称之为“动态类型检查”

### ■ 良类型系统

- ⊕ 能够在编译时确定目标程序在运行时是否会发生类型错误。

- 如果一种语言称为“强类型语言”，那么该语言编写的程序在编译时通过，则在运行时不发生类型错误

## 2.2.2 符号表



### ■ 符号表的作用

- ⊕ 一致性检查和作用域分析;
- ⊕ 辅助代码生成

## 2.2.2 符号表

### ■ 符号表的每一项(入口)包含两大栏:

- ⊕ 名字栏, 也称主栏, 关键字栏
- ⊕ 信息栏, 记录相应的不同属性, 分为若干子栏

名字	信息
----	----

### ■ 对符号表的操作:

- ⊕ 填入名称
- ⊕ 查找名字
- ⊕ 访问信息
- ⊕ 填写修改信息
- ⊕ 删除



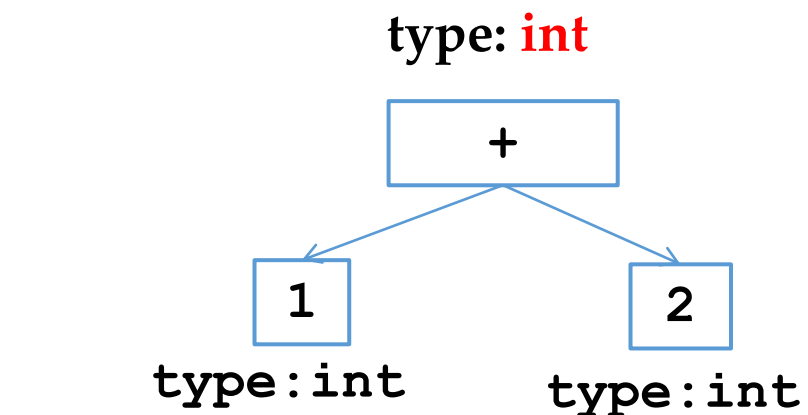
## 2.2.2 符号表

- 对符号表进行操作的时机
  - ⊕ 定义性出现
  - ⊕ 使用性出现
- 按名字的不同种属建立多张符号表，如常数表、变量名表、函数名表、...
- 符号表的信息栏中登记了每个名字的有关性质
  - ⊕ 类型：整、实或布尔等
  - ⊕ 种属：简单变量、数组、过程等
  - ⊕ 大小：长度，即所需的存储单元字数
  - ⊕ 相对数：指分配给该名字的存储单元的相对地址

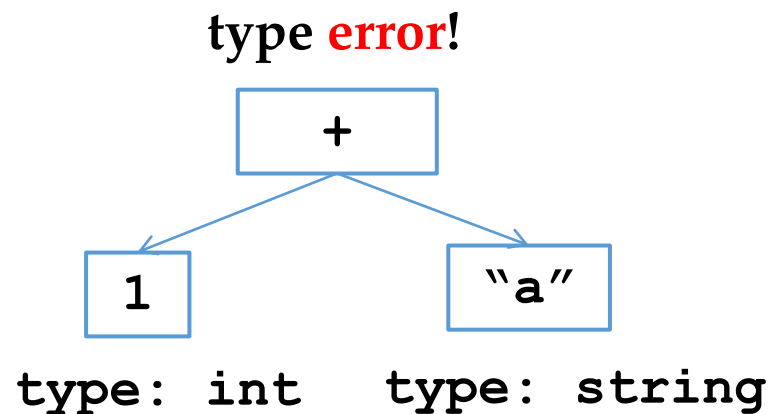
## 2.3 类型检查

### ■ 表达式的类型检查

- ⊕ 每个表达式都具有一个类型
- ⊕ 根据操作符检查操作数类型表达式，计算结果类型表达式



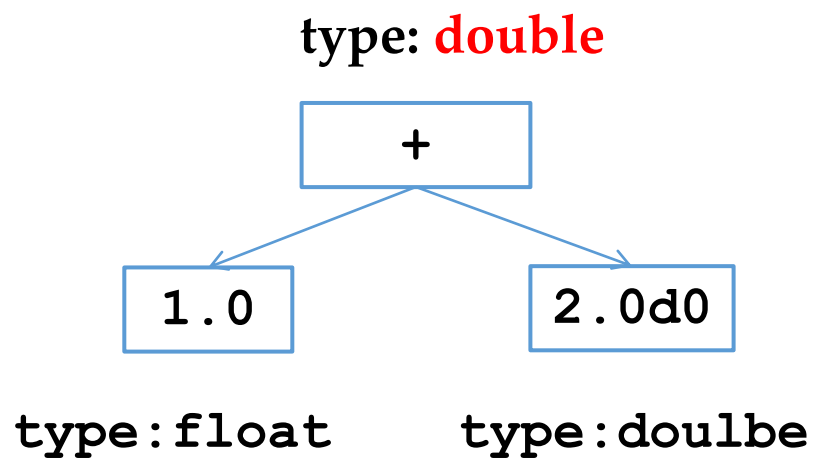
⊕ 递归检查



## 2.3 类型检查

### ■ 表达式的类型检查

⊕ 根据类型系统的规则进行类型转换



兼容的数据类型自动转换

## 2.3 类型检查

```
#include <stdio.h>
int main()
{
    int a=1.0;
    printf("a = %d\n",a);
    return 0;
}
```

语句类型检查，兼容的数据类型自动转换

## 2.3 类型检查

### ■ 声明的类型检查

- ⊕ 按照程序书写顺序检查
- ⊕ 变量
  - 查询符号表，判断变量在当前作用域是否已经被定义
  - 计算类型表达式
  - 变量-类型信息放入符号表中
- ⊕ 复合语句
  - 创建新的作用域
  - 处理声明（形参+局部变量），放入符号表
  - 对语句进行类型检查
- ⊕ 函数
  - 查询符号表，判断函数在当前作用域是否已经被定义
  - 计算结果和参数类型
  - 检查函数体复合语句的类型

## 2.3 名字空间

- 程序语言具有不同种类的标识符
  - ⊕ 变量、函数、标号、类型名、聚合类型的成员.....
- 存在不同的名字空间 (Name Spaces)
  - ⊕ 成员名局部于聚合类型
  - ⊕ 类型名不同于变量和函数名
- 实现
  - ⊕ 不同的名字空间具有不同的符号表
  - ⊕ 嵌套符号表

# 中间表示

## ■为什么使用中间表示

- ⊕ 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

## 3.1 中间表示

### ■ 中间表示

- ⊕ 有时又称中间语言或中间代码，是编译器和虚拟机内部使用的、能够表示源代码的数据结构或代码
- 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
- 也有些中间表示的设计是专门为了支持一种特定的语言，例如JVM的设计仅仅是为了支持Java语言
- 而大多数中间表示语言的设计是用来将不同的前端语言 and 后端平台连接起来。



## 3.1 中间表示

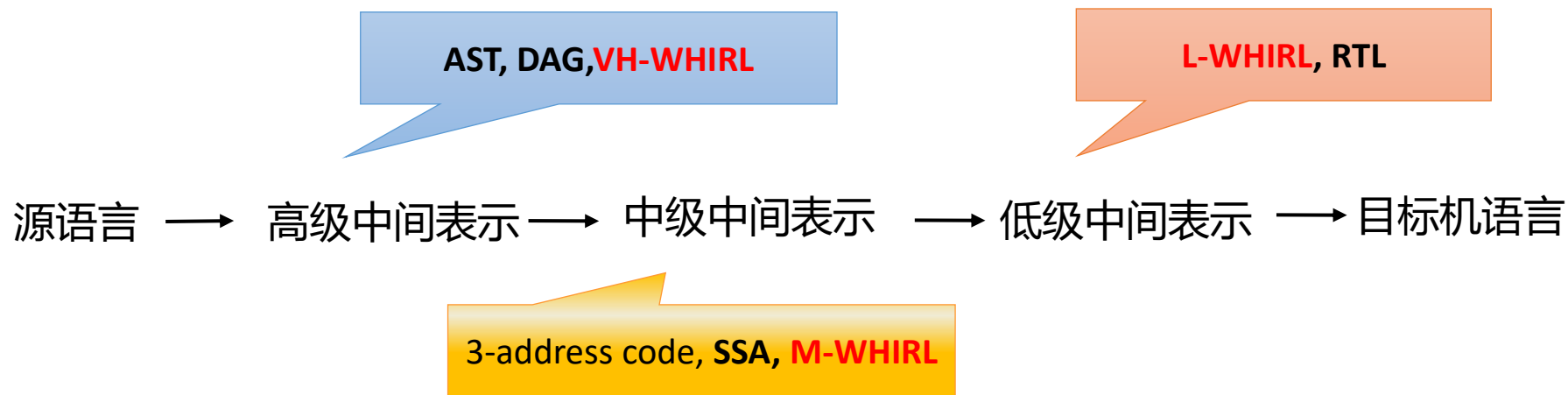
### ■好的中间表示

- ⊕ 易于生成
- ⊕ 便于操作，处理代价小
- ⊕ 精确地反映源代码信息
- ⊕ 独立于具体的语言和硬件

### ■中间表示是设计编译器架构的关键之一

## 3.1 中间表示

### ■ 不同抽象层次的中间表示



### ■ 编译器在编译过程中可以使用不同抽象层次的中间表示

⊕ GCC, open64等

## 3.2 中间表示的类型

### ■ 结构化的中间表示

- ⊕ 图
- ⊕ 精确地反映源代码信息
- ⊕ 抽象语法树、有向无环图

### ■ 线性中间表示

- ⊕ 具有简单、紧凑的数据结构，易于重排
- ⊕ 三地址代码

### ■ 混合

- ⊕ 图和线性代码混合
- ⊕ 例如：GCC使用的gimple中间表示

## 3.3 抽象语法树 (AST)

### ■ Abstract Syntax Tree (AST)

- ⊕ 抽象语法结构的树表示
- ⊕ 节点表示源代码中的一种语法结构
- ⊕ 但并不依赖于具体的文法

C代码:

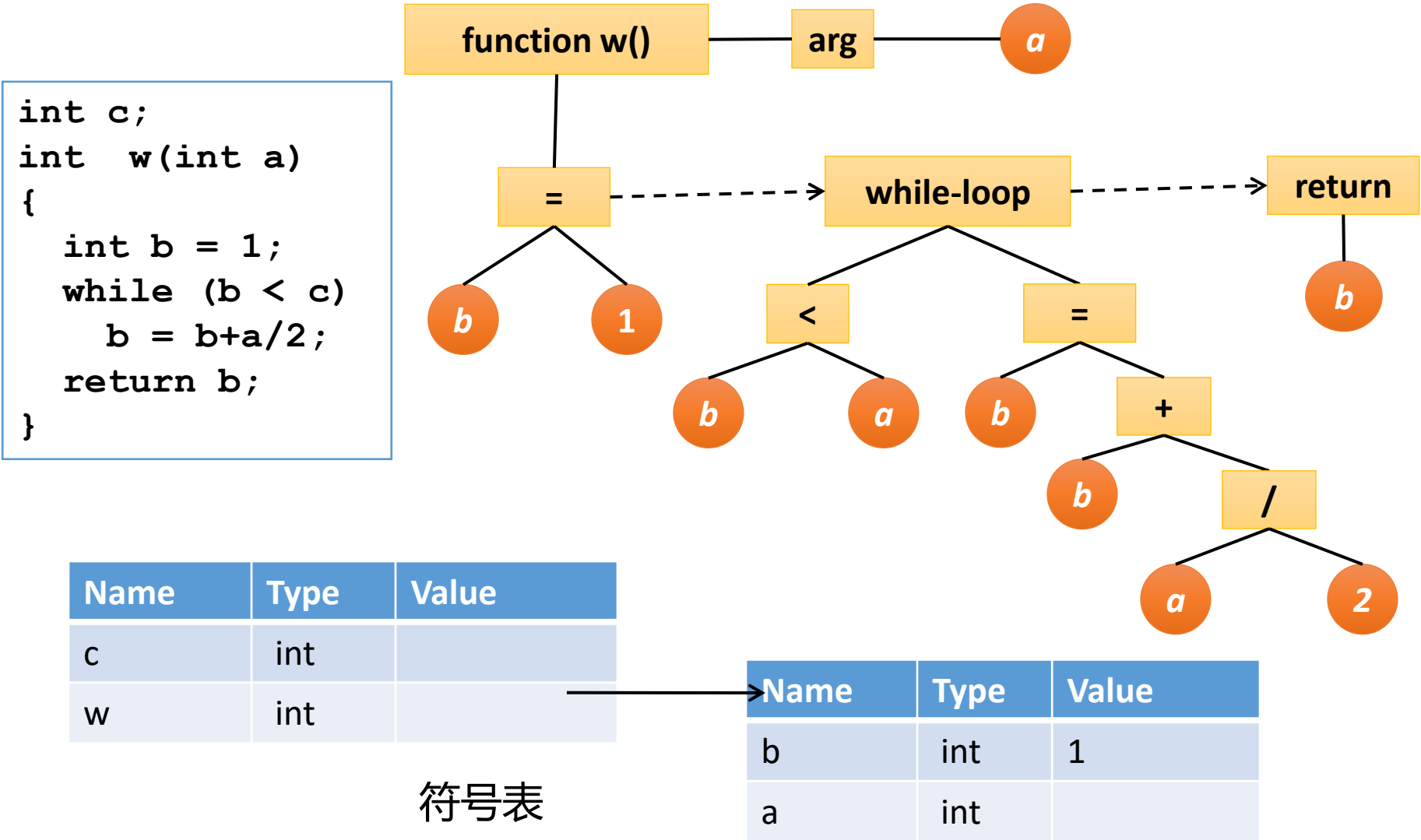
```
if (cond) {  
  }  
else {  
  }
```

Fortran代码:

```
if (cond)  
  then ...  
  else ...  
endif
```

使用上下文无关文法描述语法规则时，if语句描述不同的，但是构造AST时，都需要3个分支节点来表示：cond、then、else

# 3.3 抽象语法树 (AST)



Name	Type	Value
c	int	
w	int	

符号表

Name	Type	Value
b	int	1
a	int	

## 3.3 抽象语法树 (AST)

### ■ Abstract Syntax Tree (AST)

⊕ 尽管AST并不依赖于具体的文法，但是保留了源程序很多信息

### ■ 编译器中对AST的使用

⊕ 源到源编译器

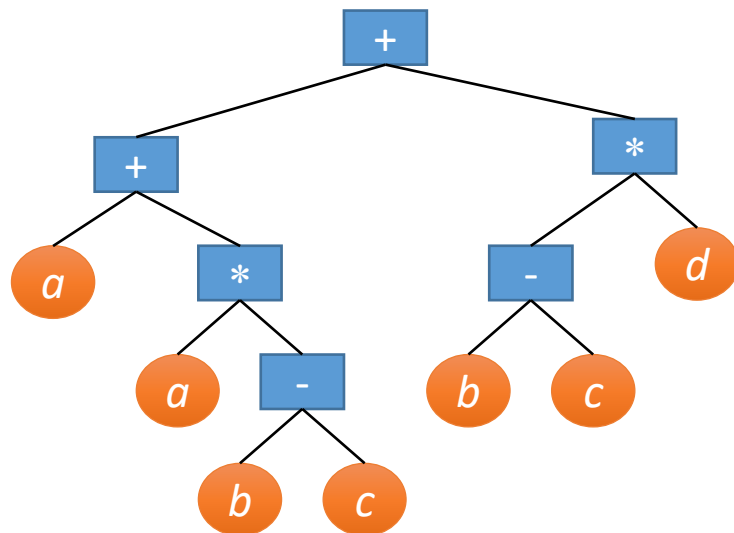
⊕ 过渡中间表示

## 3.4 有向无环图

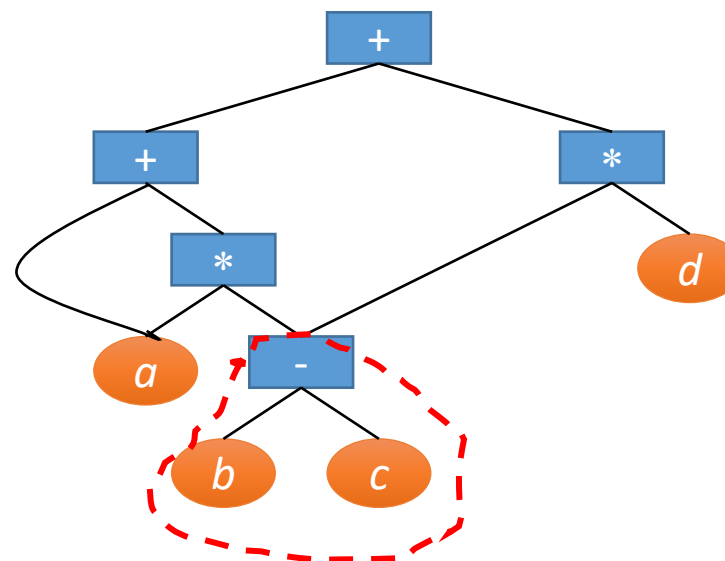
### ■ 有向无环图 (Directed Acyclic Graph, DAG)

- ⊕ AST的变体, 每个结点对应一个唯一值
- ⊕ N是一个公共子表达式, 则N可能有多个父结点

$$a + a * (b - c) + (b - c) * d$$



AST



DAG

## 3.5 三地址代码

- 三地址代码是由下面形式的语句构成的序列：

$x := y \text{ op } z$

- ⊕ 其中， $x$ 、 $y$ 、 $z$ 是名字、常数或者临时变量， $op$ 表示操作符
- ⊕ 每条语句的右边只能有一个操作符

- 语句 $a = x + y + z$ 需要被翻译成两条语句序列：

$t = x + y$

$a = t + z$

- ⊕ 其中， $t$ 是编译器生成的临时变量
- 可以看作AST或DAG的一种线性表示



## 3.5 三地址代码

### ■ 三地址代码的具体表示方法

- ⊕ 三元式

- ⊕ 间接三元式

- ⊕ 四元式

### ■ 线性的中间表示，简单紧凑，容易进行重排等操作

### ■ 目前大部分编译器采用三地址代码作为中间表示

## 3.5 三地址代码

### ■ 四元式

⊕ 一个带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result

$$a := b * (-c) + b * (-c)$$

	<u>op</u>	<u>arg1</u>	<u>arg2</u>	<u>result</u>
(0)	uminus	c		$T_1$
(1)	*	b	$T_1$	$T_2$
(2)	Uminus	c		$T_3$
(3)	*	b	$T_3$	$T_4$
(4)	+	$T_2$	$T_4$	$T_5$
(5)	:=	$T_5$		a

## 3.5 三地址代码

### ■ 三元式

- ⊕ 三个域：op、arg1和arg2
- ⊕ 通过计算临时变量值的语句的位置来引用这个临时变量

$$a := b * (-c) + b * (-c)$$

<u>op</u>		<u>arg1</u>	<u>arg2</u>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

## 3.5 三地址代码

### ■ 间接三元式

- ⊕ 为了便于优化，用 **三元式表**+**间接码表** 表示中间代码
- ⊕ 间接码表: 一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置。
- ⊕ 优点: 方便优化，节省空间

## 3.5 三地址代码

■ 例如，语句

$X := (A+B) * C;$

$Y := D - (A+B)$

间接代码

(1)

(2)

(3)

(1)

(4)

(5)

三元式表

	OP	ARG1	ARG2
(1)	+	A	B
(2)	*	(1)	C
(3)	:=	X	(2)
(4)	-	D	(1)
(5)	:=	Y	(4)

## 3.6 混合表示

```
int c;  
int w(int a)  
{  
    int b = 1;  
    while (b < c)  
        b = b+a/2;  
    return b;  
}
```

GCC中间表示: GIMPLE  
Tree  
表达式为三地址代码

```
w (int a)  
{  
    int D.1244;  
    int c.0;  
    int D.1246;  
    int b;  
  
    b = 1;  
    goto <D.1242>;  
    <D.1241>:  
    D.1244 = a / 2;  
    b = D.1244 + b;  
    <D.1242>:  
    c.0 = c;  
    if (b < c.0) goto <D.1241>; else goto <D.1243>;  
    <D.1243>:  
    D.1246 = b;  
    return D.1246;  
}
```

## 3.6混合表示

```
for (i = 0; i < N; i++)  
    Sum(&A[i], &P);
```

```
loop:                                ; preds = %bb0, %loop  
%i.1 = phi i32 [ 0, %bb0 ], [ %i.2, %loop ]  
%AiAddr = getelementptr float*, %A, i32 %i.1  
call void @Sum(float %AiAddr, %pair* %P)  
%i.2 = add i32 %i.1, 1  
%exitcond = icmp eq i32 %i.1, %N  
br i1 %exitcond, label %outloop, label %loop
```

LLVM IR

RISC-like 三地址代码

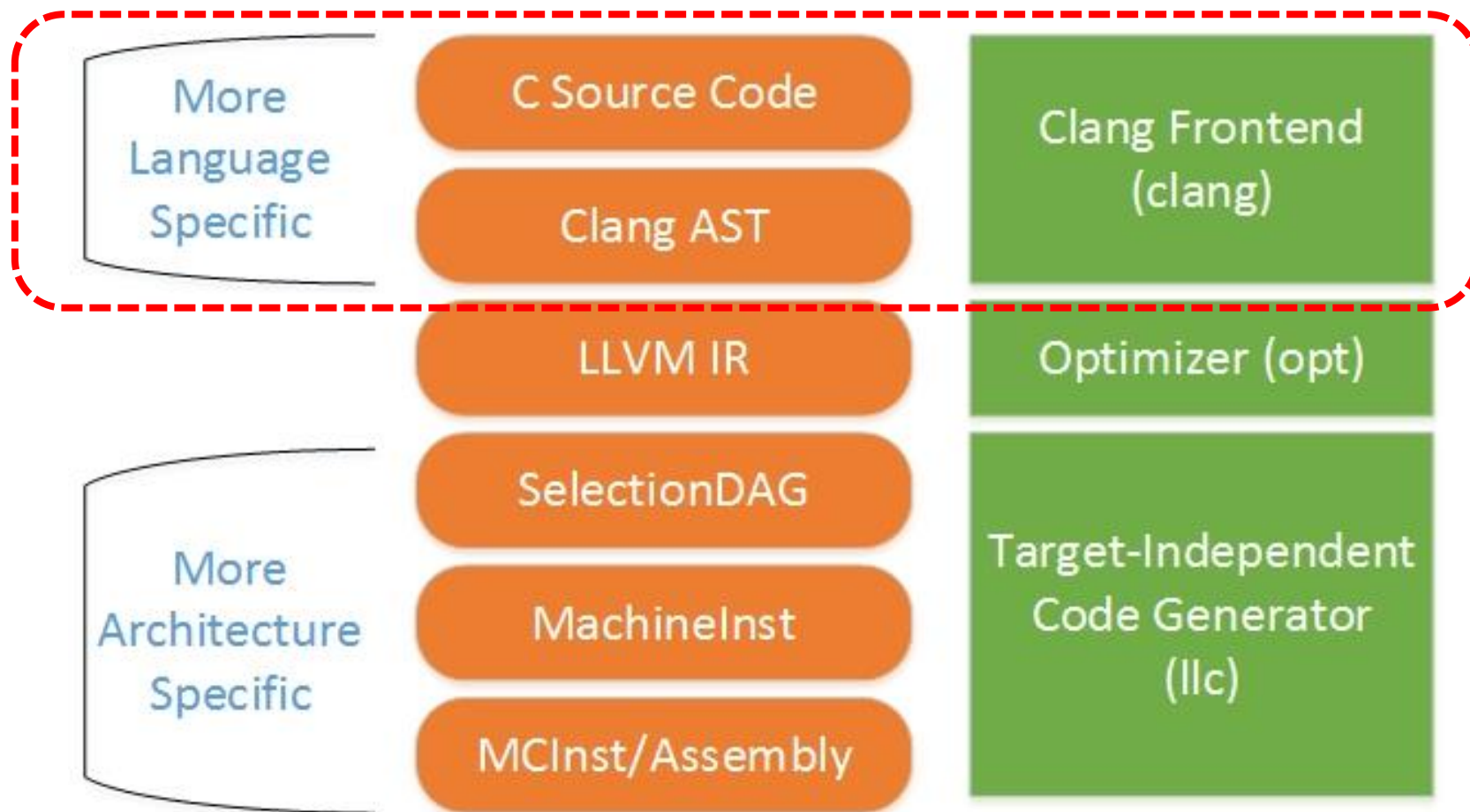
无限多寄存器集合

静态单一赋值

带有类型指针的Load/store指令

简单、低级的控制流结构

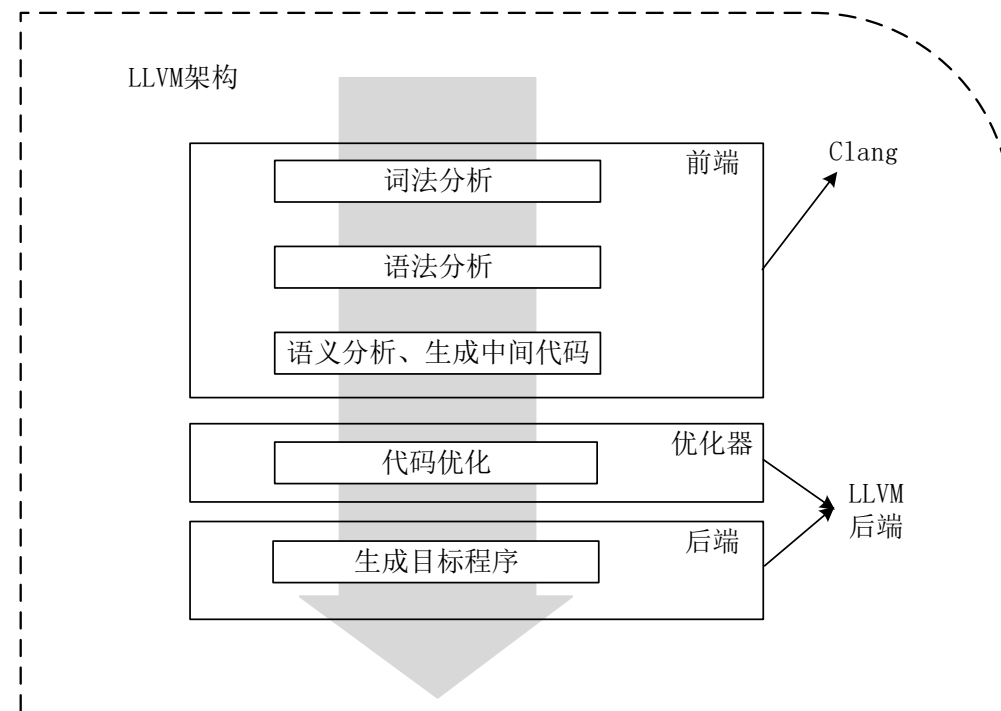
## 3.7 LLVM中间表示





## 4 编译前端实例—Clang

- Clang基于LLVM架构，为C语言族（包括C、C++、Objective-C、OpenCL、CUDA等）的语言提供语言前端和工具
- Clang提供了和GCC兼容的编译器驱动和MSVC兼容的编译器驱动



`$clang hello.c -o hello` (GCC兼容的编译器命令)

`D:\>clang-cl hello.c` (MSVC兼容的编译器命令)

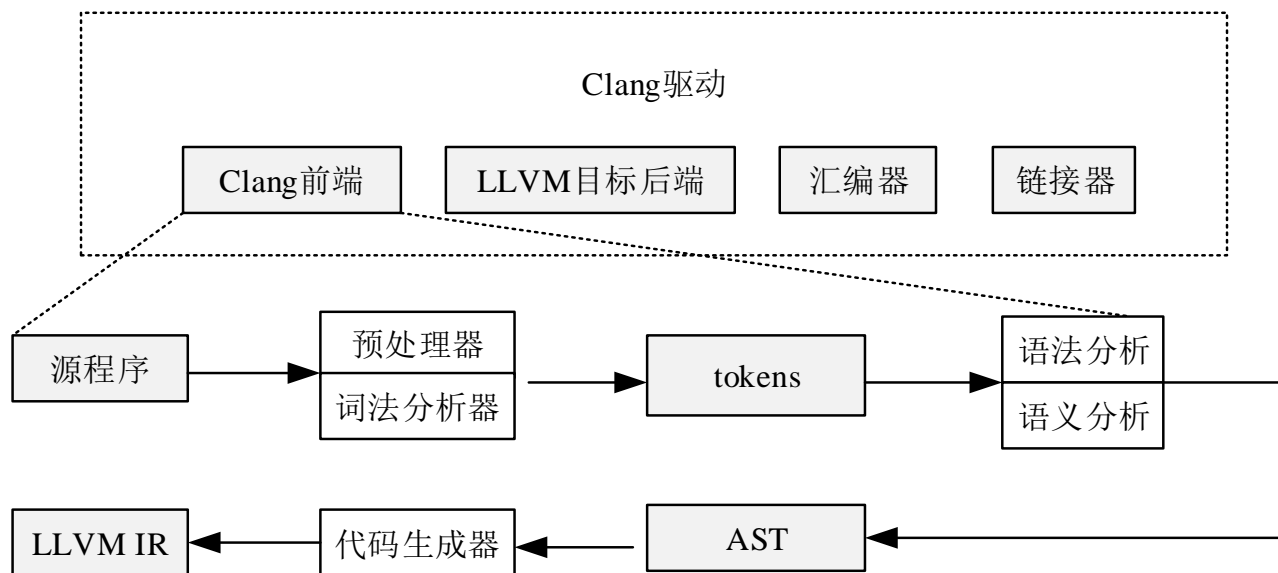
## 4.1 Clang优点

### ■ 优点

- ⊕ Clang编译速度快，占用内存小
- ⊕ 采用基于库的模块化设计，易于集成开发环境（IDE）集成或者用于开发其他的工具
- ⊕ 在编译过程中，Clang 创建并保留了大量详细的元数据，有利于调试信息的输出和错误报告。

- clang还提供了clang静态分析器和clang-tidy工具帮助程序员发现程序中隐含的错误

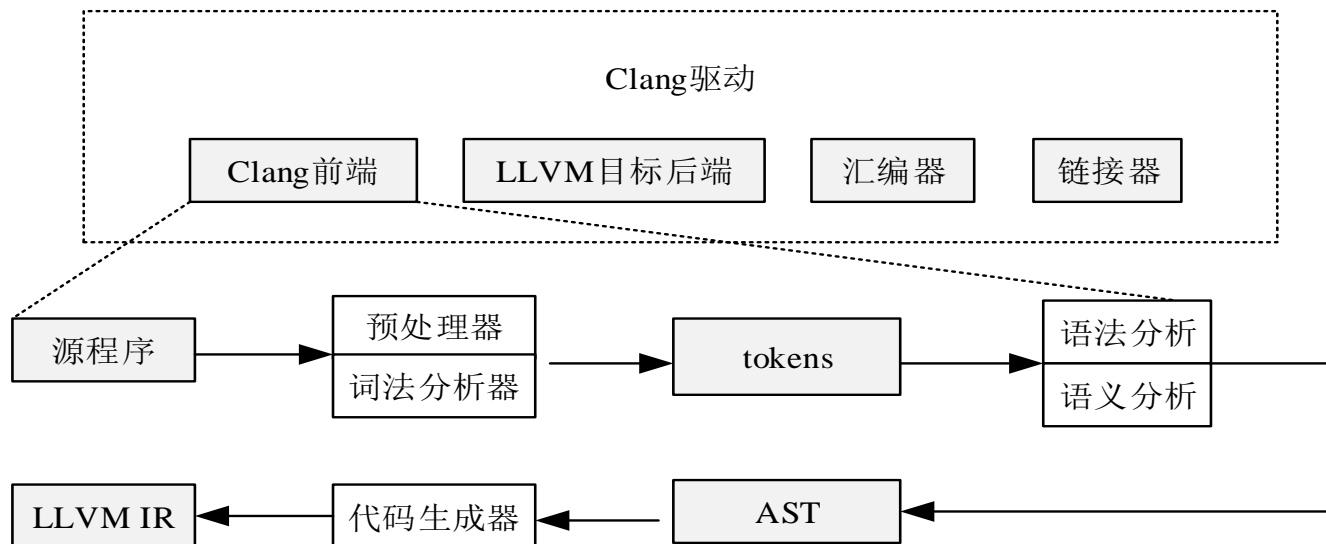
# 4.2 Clang处理流程



Clang语义分析并不是在语法分析完成之后再遍历AST进行，而是在AST节点生成的同时执行分析检查



## 4.2 Clang处理流程



- 词法分析输出词法单元，并生成相关的符号表
  - ⊕ Clang定义了词法单元类（Token）保存词法单元信息
- 采用递归下降的语法分析技术，分析词法单元组成的序列是否符合源语言语法，生成AST
- 语义分析利用符号表信息检查语义正确性
  - ⊕ Clang语义分析并不是在语法分析完成之后再遍历AST进行，而是在AST节点生成的同时执行分析检查

## 4.3 Clang抽象语法树

- AST是一种携带丰富语义信息的源代码语法结构的抽象表示，包括类型、表达式树和符号等
  - ⊕ 以树的形式表现编程语言的语法结构，树上的每个节点都表示源代码中的一种结构
  - ⊕ 和其他编译器使用的AST不同，括号表达式和编译时常量以完整的形式在Clang AST中保留。
- AST最基本的两类节点
  - ⊕ 声明 (Decl)
    - 包含不同类型的子类，用于标识不同的声明类型，例如函数声明类FunctionDecl，参数声明类ParmVarDecl
  - ⊕ 语句 (Stmt)
    - 不同的子类，对应不同的语句类型，例如 ReturnStmt类标识函数返回，CompoundStmt类标识复合语句
    - 表达式 (Expr) 也是AST中的一种语句

# 4.3 Clang抽象语法树

```
int f(){
    int a=(7+3)*7;
    int b = 5;
    return b-a;
}
```



```
TranslationUnitDecl 0x320d4ff8 <<invalid sloc>> <invalid sloc>
|-TypeDecl 0x320d5e40 <<invalid sloc>> <invalid sloc> implicit __int128_t __int128'
|-BuiltinType 0x320d55c0 'int128'
...此处省略clang内部类型说明...
`-FunctionDecl 0x3217d880 <test.c:1:1, line:6:1> line:1:5 f 'int ()'
  |-CompoundStmt 0x3217dc20 <line:2:1, line:6:1>
    |-DeclStmt 0x3217dab0 <line:3:9, col:24>
      |-VarDecl 0x3217d988 <col:9, col:23> col:13 used a 'int' cinit
      |-BinaryOperator 0x3217da90 <col:17, col:23> 'int' '*'
        |-ParenExpr 0x3217da50 <col:17, col:21> 'int'
          |-BinaryOperator 0x3217da30 <col:18, col:20> 'int' '+'
            |-IntegerLiteral 0x3217d9f0 <col:18> 'int' 7
            |-IntegerLiteral 0x3217da10 <col:20> 'int' 3
          |-IntegerLiteral 0x3217da70 <col:23> 'int' 7
        |-DeclStmt 0x3217db68 <line:4:9, col:16>
          |-VarDecl 0x3217dae0 <col:9, col:15> col:13 used b 'int' cinit
          |-IntegerLiteral 0x3217db48 <col:15> 'int' 5
        |-ReturnStmt 0x3217dc10 <line:5:9, col:18>
          |-BinaryOperator 0x3217dbf0 <col:16, col:18> 'int' '-'
            |-ImplicitCastExpr 0x3217dbc0 <col:16> 'int' <LValueToRValue>
              |-DeclRefExpr 0x3217db80 <col:16> 'int' lvalue Var 0x3217dae0 'b' 'int'
                |-ImplicitCastExpr 0x3217dbd8 <col:18> 'int' <LValueToRValue>
                  |-DeclRefExpr 0x3217dba0 <col:18> 'int' lvalue Var 0x3217d988 'a' 'int'
```

- 一个翻译单元中的顶层声明始终是 TranslationUnitDecl

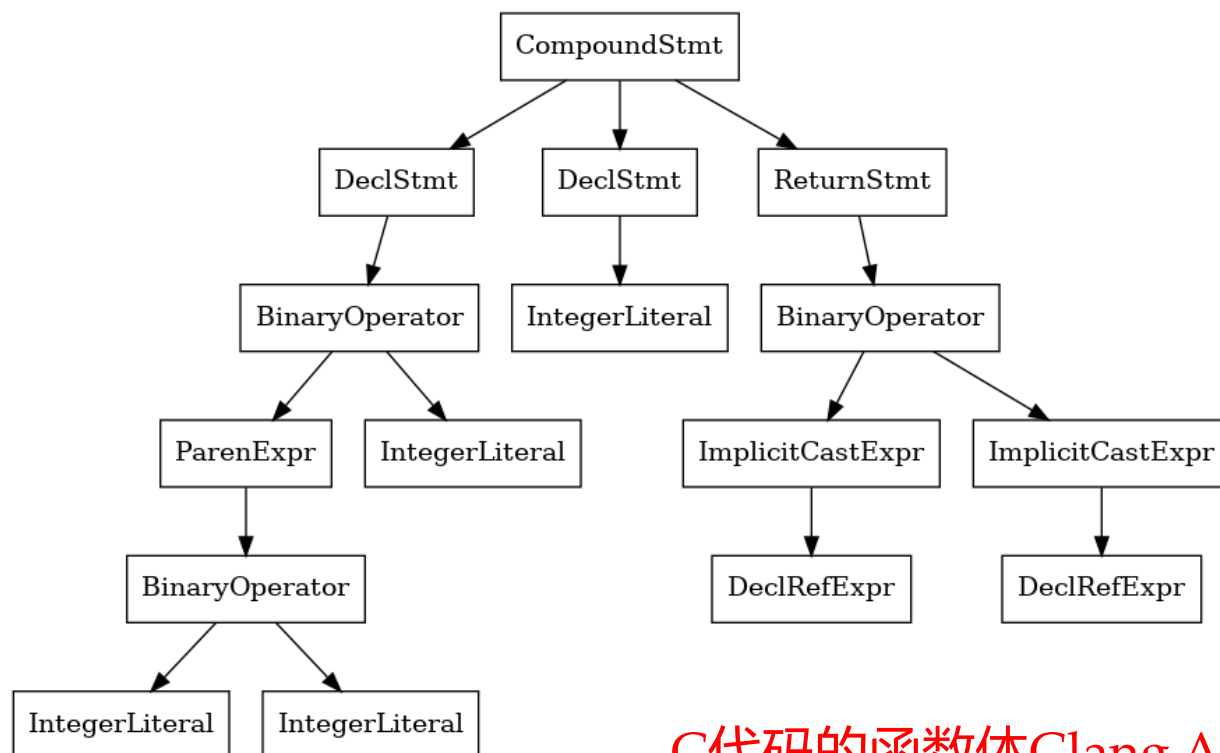
⊕ 函数 “f” 的 FunctionDecl

- “f” 的函数体是一个复合语句 (CompoundStmt)

⊕ 复合语句的子节点是2个 DeclStmt (声明a和b) 和一个ReturnStmt

## 4.3 Clang抽象语法树

- ASTContext类包含翻译单元的完整AST
- 利用ASTContext::getTranslationUnitDecl接口，从顶层TranslationUnitDecl开始，可以遍历任何一个AST节点



Clang提供了图形方法帮助用户查看AST

C代码的函数体Clang AST(`clang -cc1 -ast-view test.c`)

## 参考

- 《编译原理》
- Dragon: chap 2 ~ chap 6
- Tiger: chap 2 ~ chap 5



# 作业

1. 继续完成编译前端实验
2. 实验准备
  - 熟悉课程实验定义的中间表示