编译原理

7. 中间代码生成

rainoftime.github.io

浙江大学

计算机科学与技术学院

Content

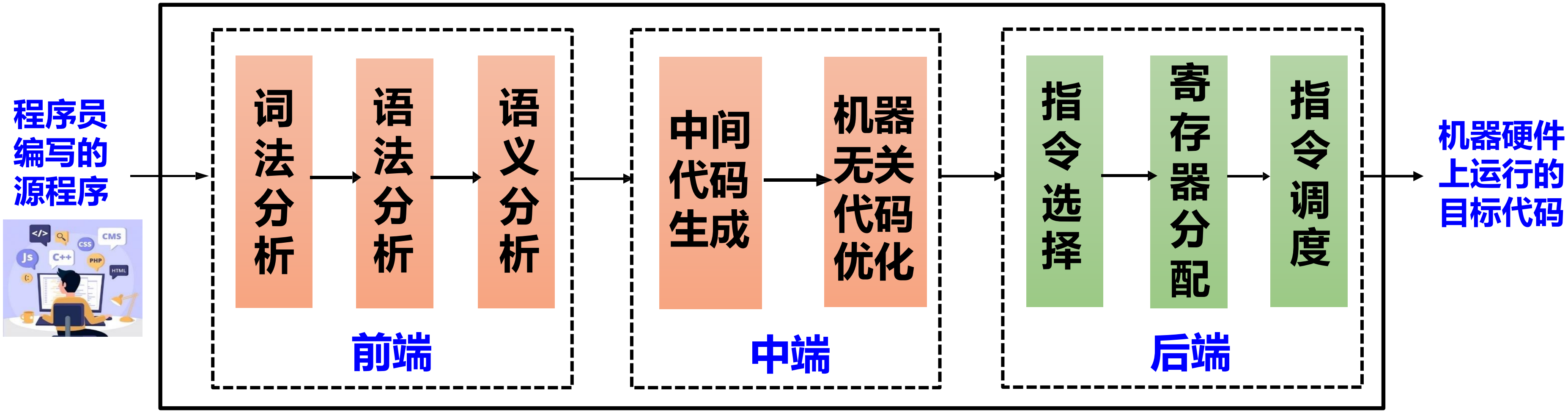
本讲内容

1 中间表示概述

2 IR Tree中间表示

3 IR Tree的生成

基本概念回顾



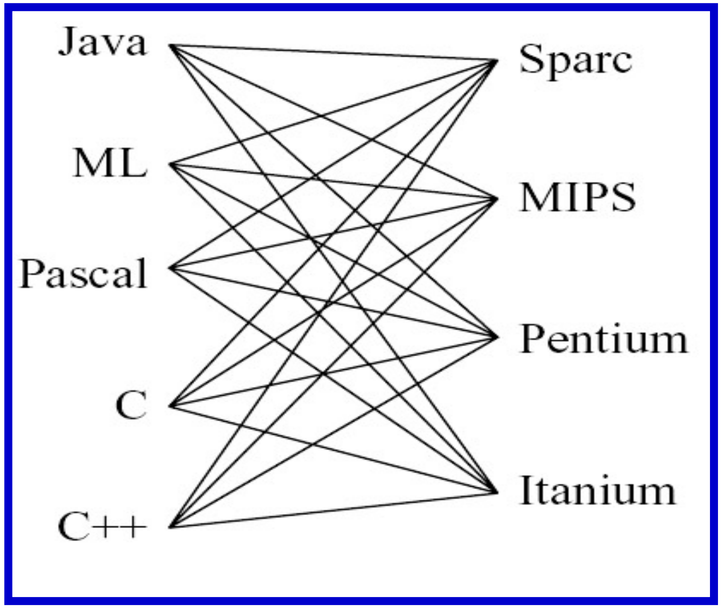
前端 – 从源码到IR生成
中端 – 基于IR的分析与变换（可能生成新IR）
后端 – (机器相关优化)；从IR 到目标代码

1. 中间表示概述

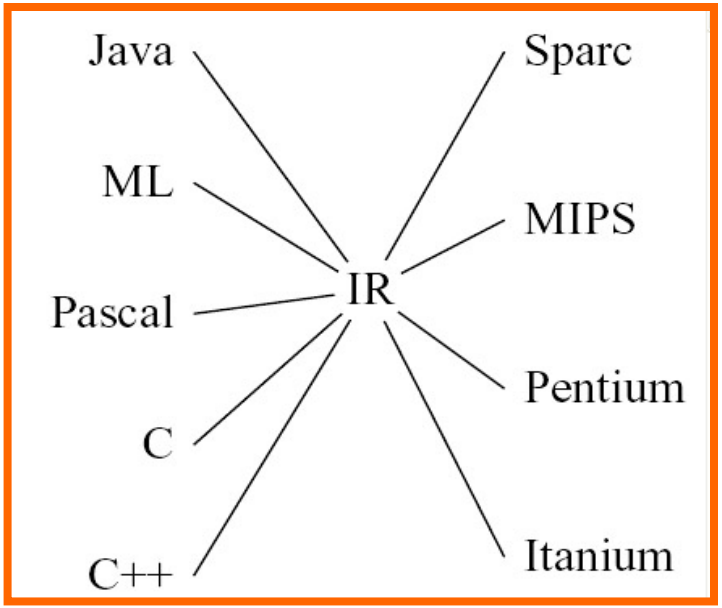
- □ 为什么需要中间表示(IR)
- □ (不要求掌握)IR分类: 根据抽象层次
- □ (不要求掌握)IR分类: 根据结构特征
- □ 三地址码

为什么需要中间语言/表示

- How about translating directly to real machine code (e.g., via semantic actions?)
  - hinders **modularity**
  - hinders **portability**



N * M



N + M

中间表示

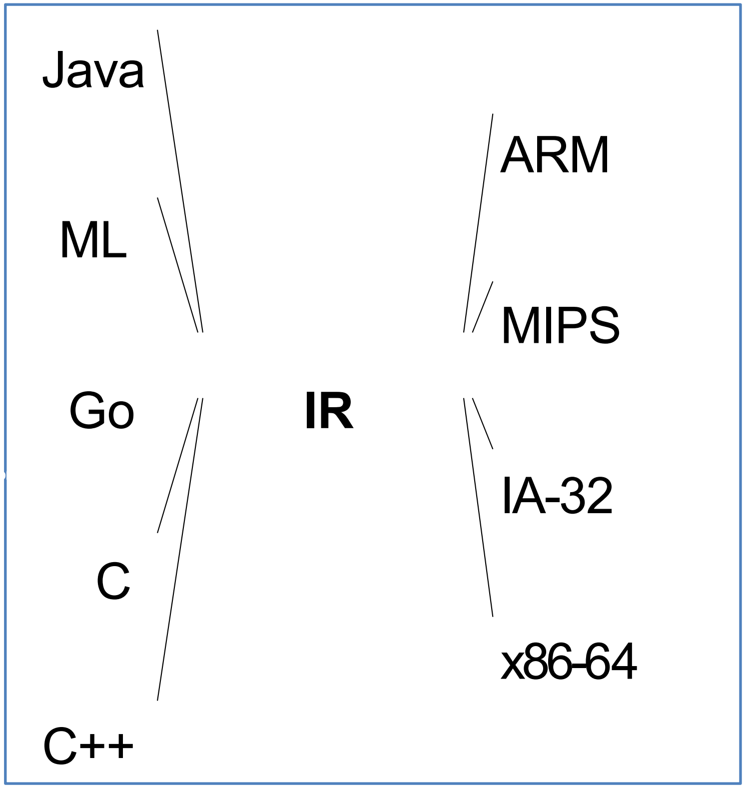
- **Intermediate representation (IR)**
  - A kind of abstract machine language
  - Able to express the target-machine operations
  - Without committing to too much machine-specific details
- **Design goal**
  - Portable compilers for different source languages and different target machines
- **IR should be simple**
  - Chunky piece of AST must be translated into IR
  - Groups of IR must be clumped together to form “real” machine instructions

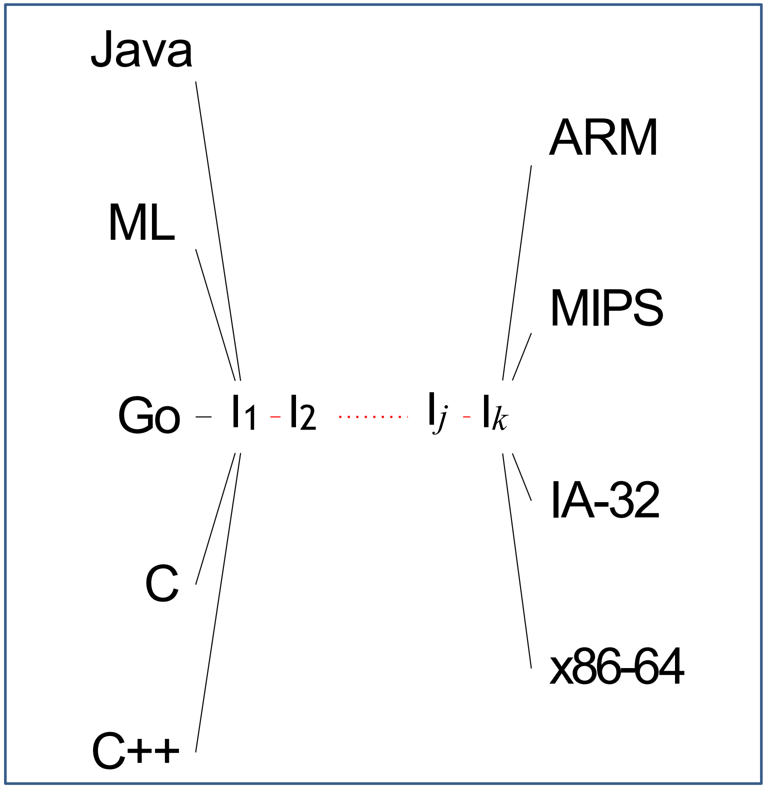
1. 中间表示概述

- 为什么需要中间表示(IR)
- (不要求掌握)IR分类: 根据抽象层次
- (不要求掌握)IR分类: 根据结构特征
- 三地址码

中间表示分类: 抽象层次

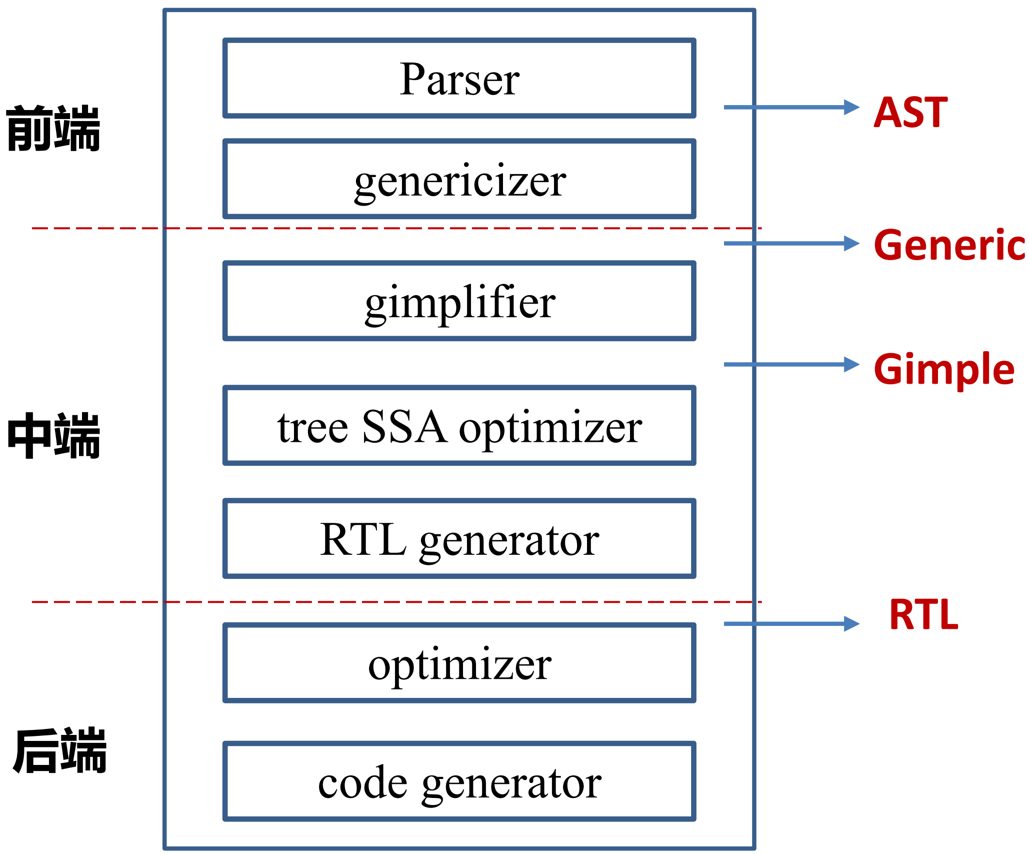
- 实际编译器可能采用多层IR
  - 支持不同层次的分析和变换

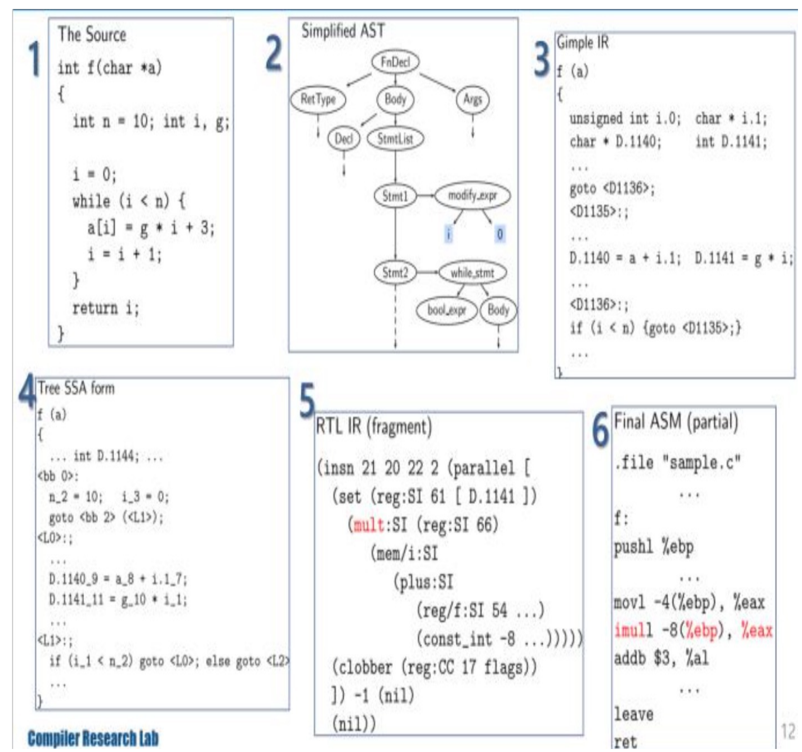




中间表示分类: 抽象层次

- 实际编译器可能采用**多层IR**
  - 支持不同层次的**分析**和**变换**

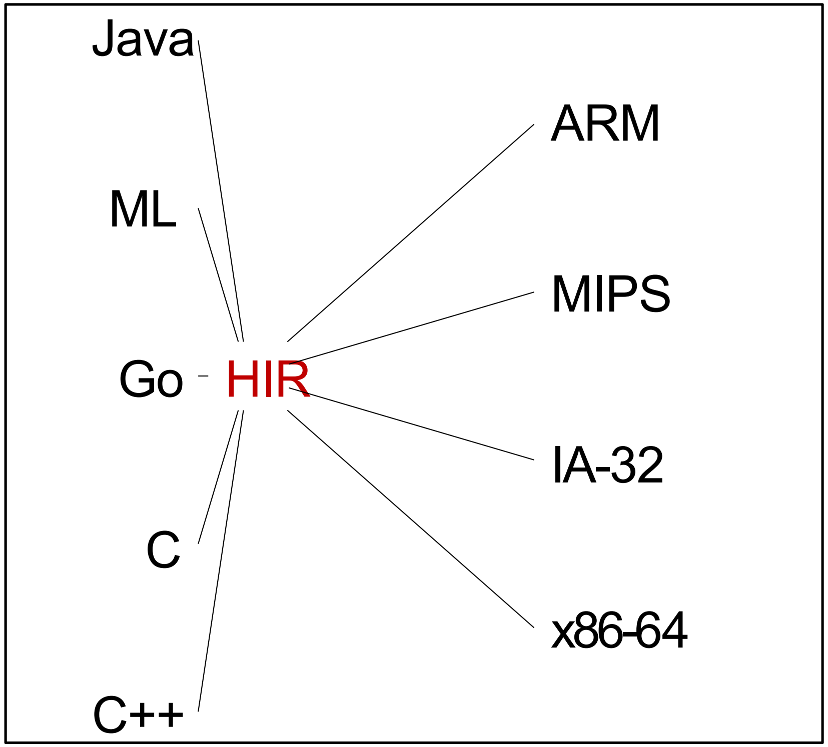




GCC编译器的多层IR

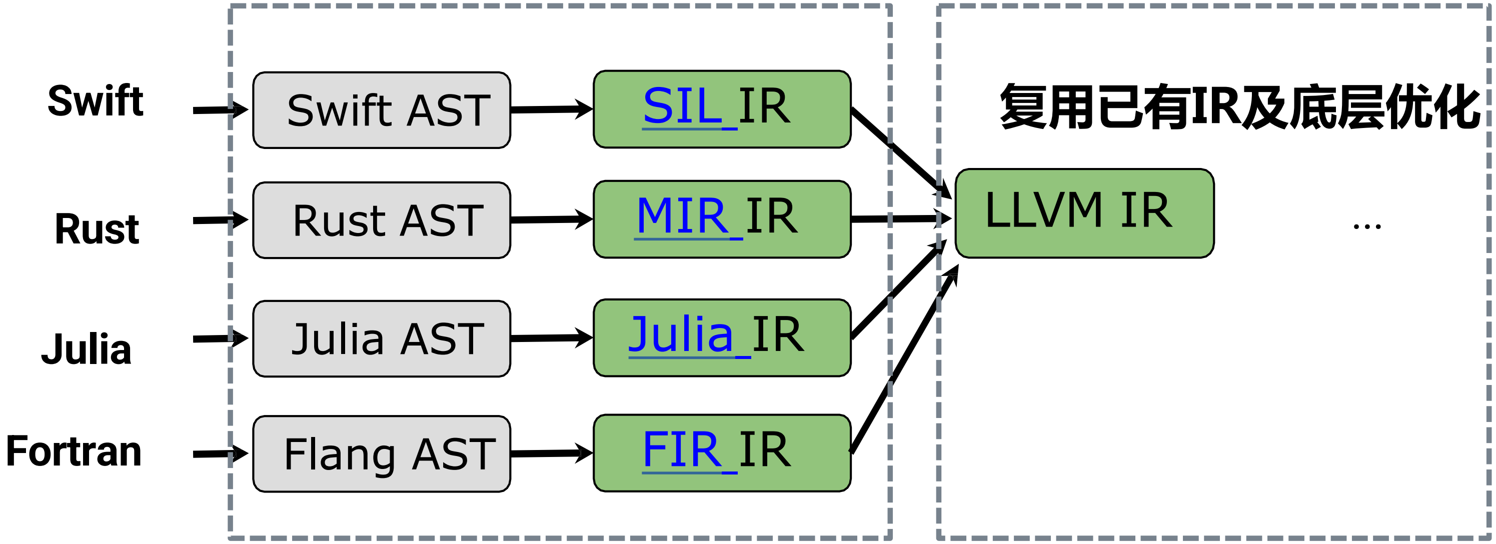
中间表示分类: 抽象层次

- **高层中间表示 High-level IR**
  - 贴近输入语言，方便由前端生成
- **低层中间表示 Low-level IR**
- **中层中间表示 Middle-Level IR**



例: 贴近输入语言的“高层IRs”

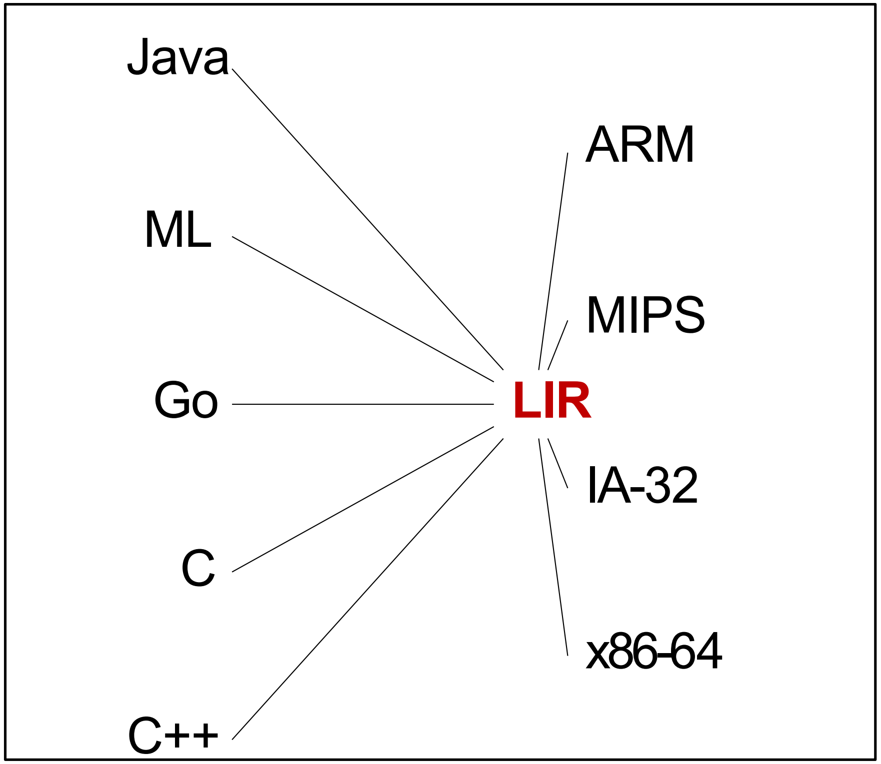
- **现代编程语言不断推出高级中间表示**



- **“高级”类型检查：borrow检查**
- **从高级编程抽象逐步"lowering"**

中间表示分类: 抽象层次

- **高层中间表示 High-level IR**
- **低层中间表示 Low-level IR**
  - 贴近目标语言，方面目标代码生成
- **中层中间表示 Middle-Level IR**



例: 贴近目标语言的“底层IRs”

- GCC编译器的RTL中间表示

```
(note 45 44 46 ("gcd.c") 11)
(note 46 45 47 NOTE_INSN_DELETED)
(note 47 46 49 NOTE_INSN_DELETED)
(insn 49 47 51 (set (reg:SI 64)
        (mem/f:SI (reg/f:SI 53 virtual-incoming-args) [0 a+0 S4 A32]))
    (nil))
(insn 51 49 52 (set (reg:SI 58)
        (reg:SI 64)) -1 (nil)
    (nil))
(jump_insn 52 51 53 (set (pc)
        (label_ref 56)) -1 (nil)
    (nil))
(barrier 53 52 54)
(note 54 53 55 NOTE_INSN_FUNCTION_END)
(note 55 54 59 ("gcd.c") 12)
(insn 59 55 60 (clobber (reg/i:SI 0 eax)) -1 (nil)
    (nil))
(insn 60 59 56 (clobber (reg:SI 58)) -1 (nil)
    (nil))
```

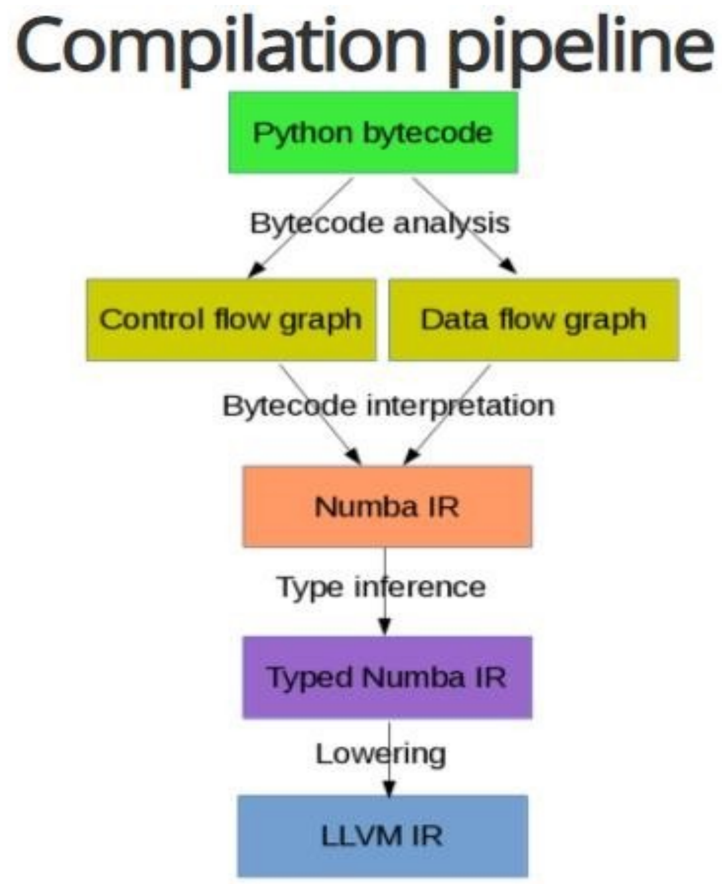
中间表示分类: 抽象层次

- **高层中间表示 High-level IR**
- **低层中间表示 Low-level IR**
- **中层中间表示 Middle-Level IR**

注意: 确切地说，三者之间没有严格的界限
(很难严格定义给定IR是HIR, MIR还是LIR

例: 各种在研的IRs

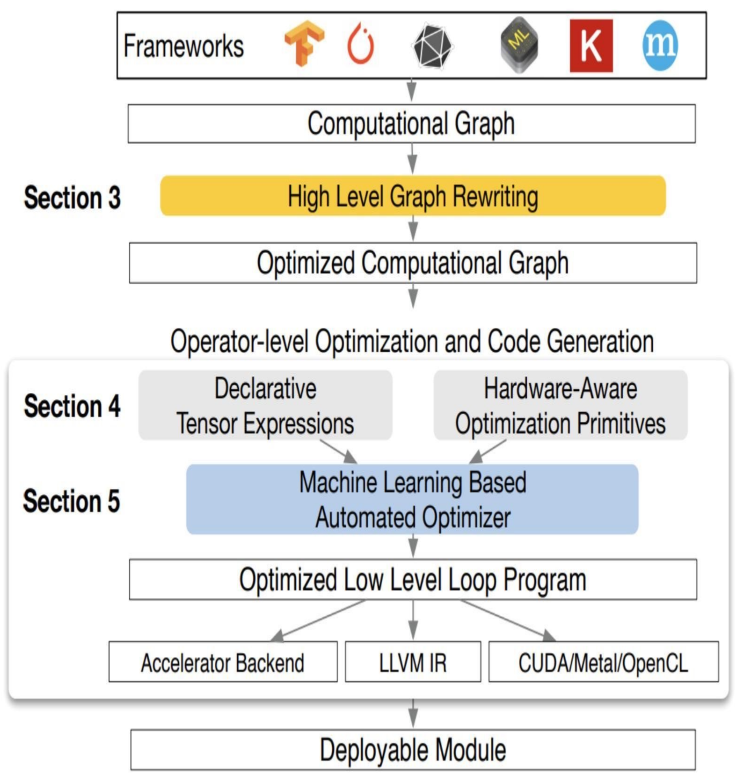
Numba: Accelerate Python Functions



https://numba.pydata.org/

TVM : Tensor Virtual Machine

面向深度学习和异构设备的优化



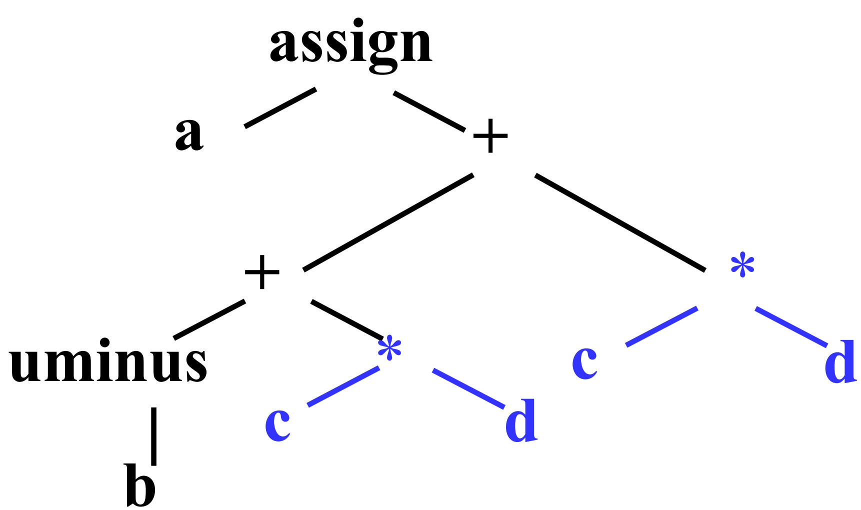
https://tvm.apache.org/

1. 中间表示概述

- 为什么需要中间表示(IR)
- (不要求掌握)IR分类: 根据抽象层次
- (不要求掌握)IR分类: 根据结构特征
- 三地址码

中间表示分类: 结构特征

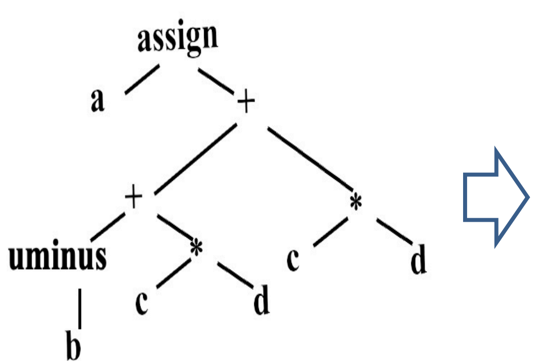
- **结构化表示 Structural**
  - Graphically oriented (e.g., tree, DAG,...)
  - Heavily used in source-to-source translators
- **线性表示 Linear**
- **混合表示 Hybrid**



$\boxed{a = (-b + c*d) + c*d}$

中间表示分类: 结构特征

- **结构化表示 Structural**
- **线性表示 Linear**
  - 线性表示: 存储布局是线性的
- **混合表示Hybrid**



$t_1 = -b$

$t_2 = c * d$

$t_3 = t_1 + t_2$

$t_4 = c * d$

$t_5 = t_3 + t_4$

$a = t_5$

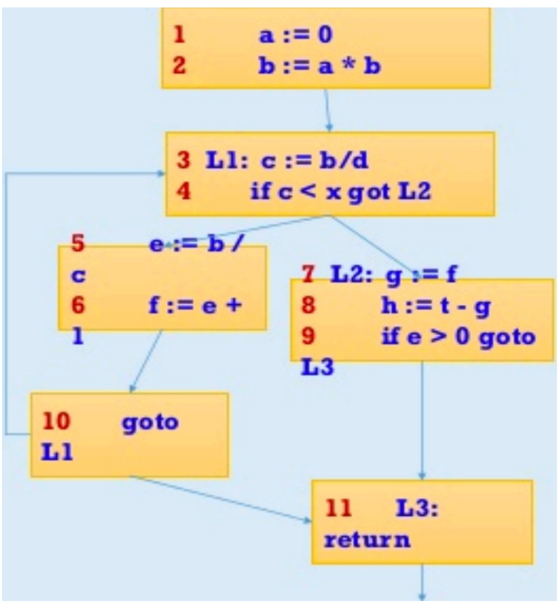
(a)三地址码three address code

```
% gcj-3.3 -c gcd.java
% jcf-dump-3.3 -c gcd
...
Method name:"gcd" public static
Signature: 5=(int,int)int  Attribute
"Code", length:66,  max_stack:2,
max_locals:2,  code_length:26
   0: iload_0
   1: iload_1
   2: if_icmpeq 24
   5: iload_0
   6: iload_1
   7: if_icmple 17
```

（b) stack machine code

中间表示分类: 结构特征

- **结构化表示 Structural**
- **线性表示Linear**
- **混合表示 Hybrid**
  - Combination of graphs and linear code



- 节点内的语句是线性表示
- 节点间构成图形化表示

例: 控制流图 Control-flow graph

1. 中间表示概述

- 为什么需要中间表示(IR)
- (不要求掌握)IR分类: 根据抽象层次
- (不要求掌握)IR分类: 根据结构特征
- 三地址码 (Three-Address Code)

三地址码 (Three-Address Code)

- **目标**
  - 接近大多数目标机器的执行模型 (机器码)
  - 支持大多数目标机器提供的数据类型和操作
  - 提供有限度的、高于机器码的抽象表达能力，更容易表达出大多数(命令式) 高级语言的特性

- **特征**
  - 以指令为单位
  - 每条指令只有有限数量的操作码

三地址码 (Three-Address Code)

- **三地址码的一般形式：**$x = y\ op\ z$
  - 每个“指令”最多1个算符，最多3个操作数(三地址)
  - 例: 表达式 $x + y * z$ 翻译成的三地址语句序列是

$$t_1 = y * z \quad t_2 = x + t_1$$

序号	指令类型	指令形式
1	赋值指令	$x = y$ **op** z $x =$ **op** y
2	复制指令	$x = y$
3	条件跳转	if x **relop** y goto n
4	非条件跳转	**goto** n
5	参数传递	**param** x
6	过程调用	**call** p, n
7	过程返回	**return** x
…	…	…

“地址”可以具有如下形式之一

- **源程序中的名字**(*name*)
- **常量** (*constant*)
- **编译器生成的临时变量**(*temporary*)

例: 三地址码

High-level language

```
read x ;  { input an integer }
if 0 < x then { don't compute if x <= 0 }
   fact:=1;
repeat
   fact:=fact*x;
   x:=x-1
until x=0;
write fact { output factorial of x }
end
```

Three-address code

```
read x
t1=x>0
if_false t1 goto L1
fact=1
label   L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4= x= =0
if_false t4 goto  L2
write fact
label   L1
halt
```

Three-Address Code - Implementation

- The entire sequence of three-address instructions is implemented as an array of linked list
- The most common implementation is to implement three-address code as **quadruples:**
  - one field for the operation
  - three fields for the addresses
- For those instructions that need fewer than three addresses, one or more of the address fields is given a null or "empty" value.

t1=x>0	**(gt, x, 0, t1)**
if_false t1 goto L1	**(if_f, t1, L1, _)**
fact=1	**(asn, 1, fact, _)**
label L2	**(lab, L2, _, _)**

- Other implementation: **triples**, **indirect triples**.

静态单赋值 (SSA) (实验需要，但不考)

- Static Single Assignment (SSA) 是一种特殊的三地址代码，其所有变量在代码中只被赋值一次

$$
\begin{array}{ll}
x = a; & x_1 = a; \\
y = x + 1; & y = x_1 + 1; \\
x = b; & x_2 = b; \\
z = x * 2; & z = x_2 * 2;
\end{array}
$$

静态单赋值 (SSA)的构造 (实验需要，但不考)

- 基本构造思路
  - 为每个变量维护一个计数器
  - 从入口开始遍历函数体
  - 遇到变量赋值时，为其生成新名字，并替换
  - 将新变量名传播到后续相应的使用处，并替换

$x = a;$	$x_1 = a;$
$y = x + 1;$	$y = x_1 + 1;$
$x = b;$	$x_2 = b;$
$z = x * 2;$	$z = x_2 * 2;$

- 通常只针对函数内的变量 (即局部变量) 计算SSA
- 全局变量的SSA在实际当中难以计算

SSA的作用(实验不需要，也不考)

- 方便了编译器中的很多分析和优化
  - 查询def-use信息(某些分析的子过程)
  - 加速现有算法（基于SSA的稀疏分析）
  - 严格依赖SSA的算法(ssapre, new gvn,…)
  - 利用SSA的变体(memory SSA, gated SSA…)

$x = 0;$	$x_1 = 0;$
…	…
$foo(x);$	$foo(x_1);$
…	…
$x = 1;$	$x_2 = 1;$
…	…
$\text{if } (x < 0)$	$\text{if } (x_2 < 0)$

广泛使用于现代编译器中(如LLVM)

2. Intermediate Representation Trees

Intermediate Representations in Tiger

- Some modern compilers use several IRs -- each IR in later phase is a little closer (to the machine code) than the previous phase

AST ==> IR1 ==> IR2 ... ==> IRk ==> machine code

- The Tiger compiler uses one IR only --- the Intermediate Representation (IR) Tree

AST => IR Tree => assembly => machine code

IR Tree: stay in the middle of AST and assembly!

IR Tree: A Low Level Tree Representation

- 一种特殊的树型中间语言/中间表示

BNF形式的文法描述

```
⟨Exp⟩ ::= "CONST" int
        | "NAME" ⟨Label⟩
        | "TEMP" ⟨Temp⟩
        | "BINOP" ⟨Oper⟩ ⟨Exp⟩ ⟨Exp⟩
        | "MEM" ⟨Exp⟩
        | "CALL" ⟨Exp⟩ [{⟨Exp⟩}] "call end"
        | "ESEQ" ⟨Stm⟩ ⟨Exp⟩

⟨Stm⟩ ::= "MOVE" ⟨Exp⟩ ⟨Exp⟩
        | "EXP" ⟨Exp⟩
        | "JUMP" ⟨Exp⟩ [{⟨Label⟩}]
        | "CJUMP" ⟨Relop⟩ ⟨Exp⟩ ⟨Exp⟩ ⟨Label⟩ ⟨Label⟩
        | "SEQ" [{⟨Stm⟩}] "seq end"
        | "LABEL" ⟨Label⟩

⟨Oper⟩  ::= "ADD" | "SUB" | "MUL" | "DIV" | "MOD"
⟨Relop⟩ ::= "EQ" | "NE" | "LT" | "GT" | "LE" | "GE"
```

IR Tree: Expressions

Expressions stand for the computation of some value, possibly with side effects

CONST(i)	The integer constant i
NAME(n)	The symbolic constant n (e.g. label) 比如goto .L中这个.L就是NAME
TEMP(t)	Temporary t. virtual register，不考虑寄存器个数
BINOP(o, e1, e2)	The application of binary operator o to operands e1, e2. The integer arithmetic operators are **PLUS,MINUS, MUL, DIV**; the integer bitwise logical operators are **AND, OR, XOR**; the integer logical shift operators are **LSHIFT, RSHIFT**; the integer arithmetic right-shift is **ARSHIFT.**
MEM(e)	The contents of wordSize bytes of memory starting at address e. When MEM is used as the left child of a MOVE, it means "store", but anywhere else it means "fetch". 比如x86里的[rax] = 100, rbx = [rax]
CALL(f, l)	A procedure call: the application of function f to argument list l.
ESEQ(s, e)	Statement s is evaluated for side effects, then e is evaluated for a result. 先计算 stm s,再根据 stm s 计算 exp e,得出结果

IR Tree: The Statements

Statements performs side-effects and control flow - no return value!

MOVE(TEMP t, e)	Evaluate e and move it into temporary t.
MOVE(MEM(e_1) e_2)	Evaluate e_1, yielding address a. Then evaluate e_2, and store the result into wordSize bytes of memory starting at **a**. 计算 e1,得到地址 a,再计算 e2,放入地址 a
EXP(e)	Evaluate e and discard the result.
JUMP(e, labs)	Transfer control (jump) to address e. The destination e may be a literal label, as in **NAME(lab)**, or it may be an address calculated by any other kind of expression.
CJUMP(o, e_1, e_2, t, f)	Evaluate e_1, e_2 in that order, yielding values a, b. Then compare a, b using the relational operator o. If the result is **true**, jump to t; otherwise jump to f.
SEQ(s_1, s_2)	The statement s_1 followed by s_2.
LABEL(n)	Define the constant value of name n to be the current machine code address.

注意Label和Expression中的Name的区别。Name是使用这个symbol，比如goto .L，而Label是定义这个symbol，即.L:

Discussion: 关于ESEQ的理解 (重要)

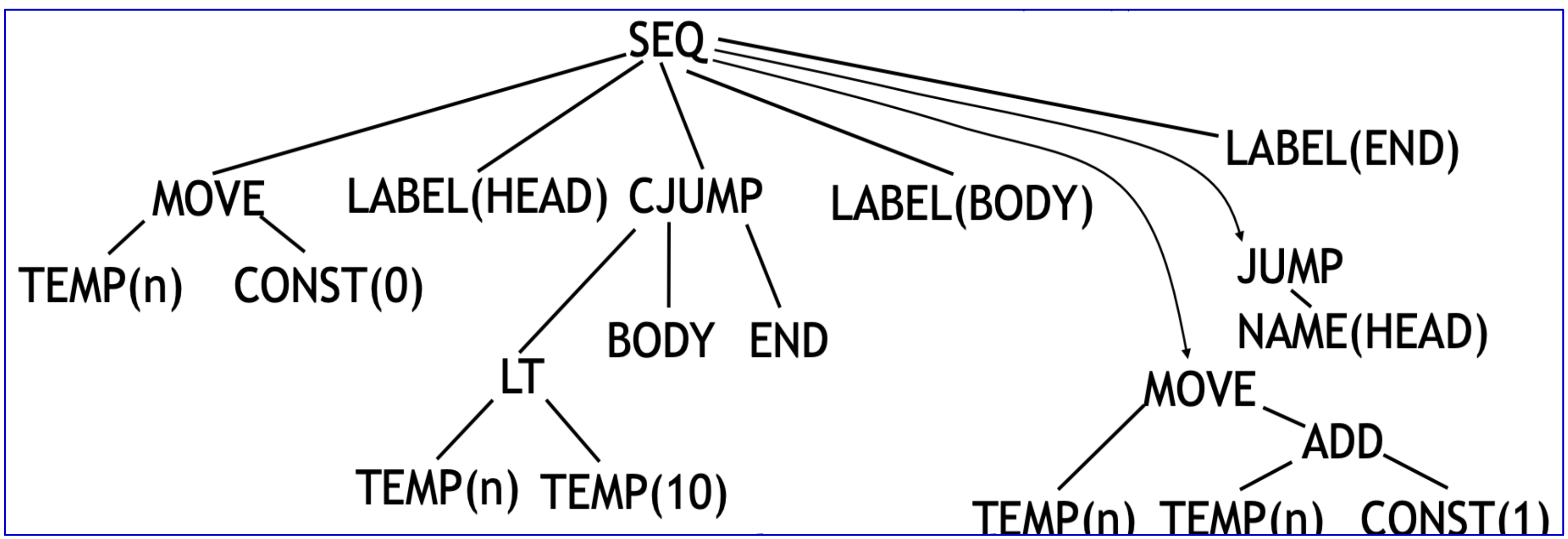
- **ESEQ(s, e)**：The statement s is evaluated for side effects, then e is evaluated for a result.
  - 假设s是statement a=5, e是expression a+5
  - Statement (如a=5)不返回值,但是有副作用
  - ESEQ(a=5, a + 5)最终的结果是10

- 关于副作用(Side effects) (重要)
  - **Side-effects** means **updating** the contents of a **memory cell** or a **temporary register**

Example: IR Tree

```
n = 0;
while (n < 10) {
   n = n + 1
}
```

```
SEQ(MOVE(TEMP(n), CONST(0)),
    LABEL(HEAD),
    CJUMP(LT(TEMP(n), CONST(10)),
          BODY, END),
    LABEL(BODY),
    MOVE(TEMP(n), ADD(TEMP(n),
                      CONST(1))),
    JUMP(NAME(HEAD)),
    LABEL(END))
```



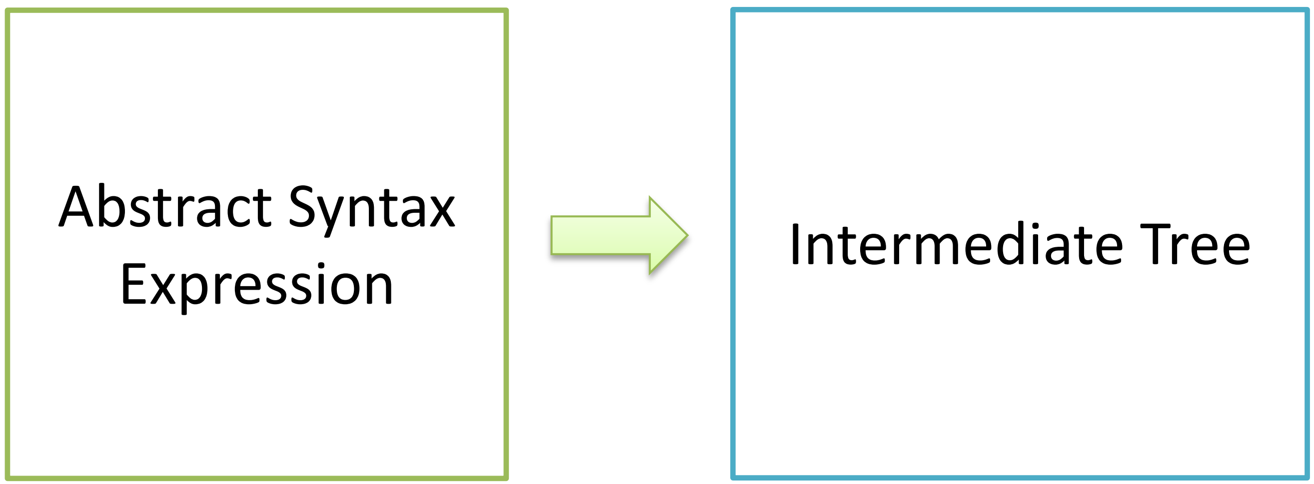
Example: IR Tree VS. X86-64 Instructions

Intel syntax	IR equivalent
17	CONST(17)
rax	TEMP(rax)
[rax]	MEM(TEMP(rax))
[rbx + 32]	MEM(ADD(TEMP(rax), CONST(32)))
[rax + rbx*8]	MEM(ADD(TEMP(rax), MUL(CONST(4), TEMP(rbx))))

- 注: 以上并非“IR Tree翻译到机器指令的规则”

3. 翻译到IR Trees

Translating Tiger AST to IR Trees



- Tree representation is also used in compilers such as GCC (called RTL)

Review: The Tiger Language

- A tiger program consists of
  - **Expressions**
  - **Declarations**
    - Variable
    - Function
    - Type

```
exp = VarExp of var
    | NilExp
    | IntExp of int
    | StringExp of string * pos
    | AppExp of {func: Symbol.symbol, args: exp list, pos: pos}
    | OpExp of {left: exp, oper: oper, right: exp, pos: pos}
    | RecordExp of {typ: Symbol.symbol, pos: pos,
                    fields: (Symbol.symbol * exp * pos) list}
    | SeqExp of (exp * pos) list
    | AssignExp of {var: var, exp: exp, pos: pos}
    | IfExp of {test: exp, then': exp, else': exp option, pos: pos}
    | WhileExp of {test: exp, body: exp, pos: pos}
    | ForExp of {var: Symbol.symbol, lo: exp, hi: exp,
                 body: exp, pos: pos}
    | BreakExp of pos
    | LetExp of {decs: dec list, body: exp, pos: pos}
    | ArrayExp of {typ: Symbol.symbol, size: exp,
                   init: exp, pos: pos}

dec = VarDec of {var: Symbol.symbol,init: exp, pos : pos,
                 typ: (Symbol.symbol * pos) option}
    | FunctionDec of fundec list
    | TypeDec of {name: Symbol.symbol, ty: ty, pos: pos} list
```

用Standard ML语言定义的Tiger抽象语法

Outline of the Translation

- **Translation of expressions**
  - **Overview**
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Three Kinds of Tiger AST Expressions

- What should the representation of an abstract syntax expression A_exp be in the tree language?
  - **Expressions with return values**
  - **Expressions that return no value** (such as while expressions)
  - **Expressions with Boolean values**, such as a > b
    - a conditional jump

- (Tiger does not distinguish “expressions” and “statements”)

Mapping Tiger AST Expressions to IR Tree

- To represent the three kinds of AST expressions
  - **Ex**: expressions that compute values (*Tree expression*)
  - **Nx**: expressions that compute no values (*Tree statement*)
  - **Cx**: conditional jump, a *Tree statement* that may jump to a true-label or false-label

```
typedef struct Tr_exp_ *Tr_exp;
struct Cx { patchList trues; patchList falses; T_stm stm;};
struct Tr_exp_ {
  enum { Tr_exp, Tr_nx, Tr_cx } kind;
  union {
    T_exp ex;
    T_stm nx;
    struct Cx cx; } u;
};
static Tr_exp Tr_Ex ( T_exp ex);
static Tr_exp Tr_Nx ( T_exp nx);
static Tr_exp Tr_Cx ( patchList trues, patchList falses,T_stm stm);
```

Mapping AST Expressions to IR Tree

- Things are Not That Easy
- Consider the translations *flag := (a>b | c<d)*
  - Requires the conversion of a Cx into an Ex *!*
  - (因为我们需要给flag赋值，而只有Ex才有返回值)
- **The Problem:** Need **utility functions** for conversion among **`Ex`**, **`Nx`**, and **`Cx`** expressions
  - Expressions, statements, and conditions: one needs to be converted to the other in various **contexts**

Mapping AST Expressions to IR Tree

- **unEx, unNx, unCx**: Utility functions for conversions among **Ex**, **Nx**, and **Cx**
  - For different kinds of output expressions, we use different conversion functions.
  - Tr_exp means the **input expression can be of any kind**

```
static T_exp unEx(Tr_exp e);
static T_stm unNx(Tr_exp e);
static struct Cx unCx(Tr_exp e);
```

flag := (a>b | c<d) ⟶

```
e = Tr_Cx (trues, falses, stm)
MOVE (TEMP(flag), unEx(e))
```

需要给flag赋值，而只有Ex(T_exp)才有返回值。
因此用unEx函数将e转成了Ex (类似强制类型转换)

Mapping AST Expressions to IR Tree

- **unEx, unNx, unCx:** Utility functions for conversions among **`Ex`**, **`Nx`**, and **`Cx`**
  - For different kinds of output expressions, we use different conversion functions.

- 为什么需要unEx, unNx, unCx这几个辅助函数?
  - 需要考虑到a>b被使用的“上下文”
  - IR翻译是context-dependent问题(难以用CFG刻画, 但是可以用属性文法、semantic actions等方式)

Outline of the Translation

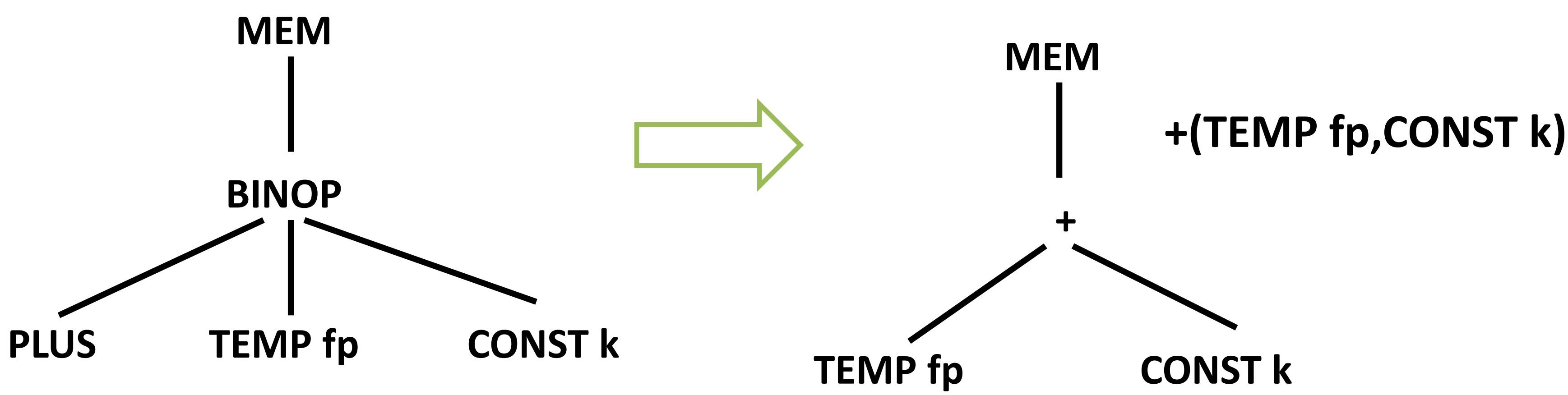
- **Translation of expressions**
  - Overview
  - **Simple Variables**
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Simple Variable

- A simple variable v declared in the current procedure's stack frame
- Accessing a **local variable v** at offset `k` (the frame pointer is `fp`)

MEM(BINOP(PLUS,TEMP fp, CONST k))

  - TEMP fp is the frame-pointer register
  - *k* is the offset of *v* within the frame
  - For Tiger, all variables are the same size – word size



We can abbreviate **BINOP(PLUS, e1, e2)** as **+ (e1, e2)**

Simple Variables

Example

- An access *a* is InFrame(k):

$$\boxed{\texttt{F\_Exp(a, T\_Temp(F\_FP())}}$$

Returns

$$\boxed{\texttt{MEM(BINOP(PLUS,TEMP FP,CONST(}k\texttt{)))}}$$

- An access *a* is InReg(t_{832})
  - Simply return TEMP t_{832}

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - **Array Variables**
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Array Variables

- Different programming languages treat arrays differently
- In **Pascal**: An array variable stands for the contents of the array
  - The following snippet copies the contents of b into a (all 12 ints)

```
var a,b : array[1..12] of integer;
begin
  a:=b
end.
```

- In **C**: arrays are like “pointer constants”

```
{ int a[12], b[12];
  a=b;
}
```

Illegal

assignment on array variables are illegal

```
{ int a[12], *b;
  b=a;
}
```

legal

b points to the beginning of the array a

Array Variables

- In **Tiger and ML**, array variables behave like pointers
  - has no named array constants as in C
  - new array values are created (and initialized) by the construct t_a[n] of i
    - t_a is the name of an array type
    - n is the number of elements
    - i is the initial value of each element

```
let
  type intArray = array of int
  var a:= intArray[12] of 0
  var b:= intArray[12] of 7
in a:= b
end
```

- *a* ends up pointing to the same 12 sevens as the variable *b*;
- the original 12 zeros allocated for *a* are discarded.

Array Variables

- Tiger record values are also pointers.
  - Record assignment, like array assignment, is pointer assignment and does not copy all the fields.

- How to translate array accesses, e.g., arr[2], arr[x]?

We will continue talking about this in the "Subscripting and Field Selection" part

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - **Structured L-values**
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

R-Values and L-Values

- **R-value**: appear on the right of an assignment
  - a+3 or f(x)
  - r-value does not denote an assignable location
- **L-value**: the result of an expression that can occur on the left of an assignment statement, such as x and a[i+2]
  - Denotes a location that can be assigned to
  - Can occur on the right of an assignment statement (In such cases, it means the contents of the location)

Structured *L*-Values

- An integer or pointer value is a "scalar" (It has only one component)
  - **All the variables and L-values in Tiger are scalar**
  - A Tiger array or record variable is really a pointer
- In C or Pascal there are structured L-values
  - Structs in C, arrays and records in Pascal
  - They are not scalar.

Structured *L*-Values

- An integer or pointer value is a "scalar" (It has only one component)
- In C or Pascal there are structured L-values
- For structured L-values, an address calculation should be MEM(+(TEMP fp, CONST K), S)

```
T_exp T_Mem(T_exp, int size);
```

Mem(+(TEMP fp,CONST k_n), S)

- *S* indicates the size of the object to be fetched or stored

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - **Subscripting and Field Selection**
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Subscripting and Field Selection

- To compute the address of a[i]:

$$(i - l) \times s + a$$

  - l: the lower bound of the index range
  - s: the size (in bytes) of each array element
  - a: the base address of the array elements

- If a is global, with a compile-time constant address, the substraction $a - s \times l$ can be done at compile time.

- T calculate the address of the field f of a record a

$$\mathbf{offset(f) + a}$$

Subscripting and Field Selection

- In the **Tiger** language, all record and array values are really pointers to record and array structures
  - There is no structural l-value

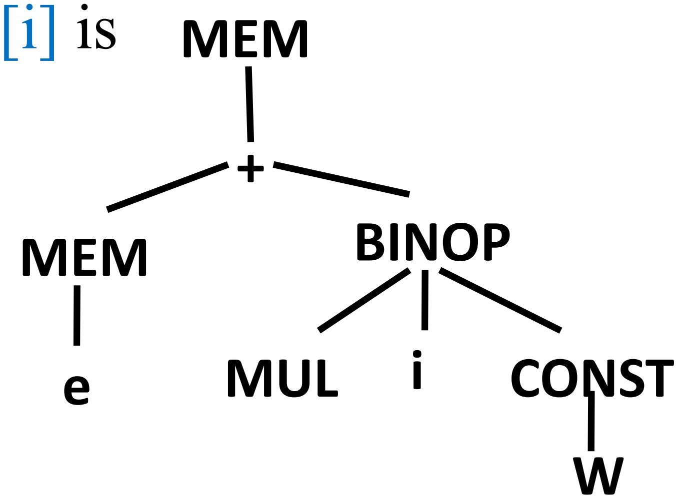
- The "base address" of the array is really the contents of a pointer variable, so MEM is required

- In the Tree IR, MEM means both store (when used as the left child of a MOVE) and fetch (when used elsewhere)

Subscripting and Field Selection

- Technically, an l-value should be represented as an address (without the top MEM node)
  - Converting an l-value to an r-value: fetching from that address
  - Assigning to an l-value: storing to that address

- IR tree of a[i] is



- MEM(e) represents a
- W is the word size

MEM(+(MEM(e),BINOP(MUL, i, CONST W)))

Example: Record Access

```
type rectype = {f1:int, f2:int, f3:int}
                        |        |        |
          offset:       0        1        2

var a:rectype := rectype{f1=4, f2=5, f3=6}
```

- Let e be IR tree for a:

```
a.f3:
  MEM(BINOP(PLUS, e, BINOP(MUL, CONST(3), CONST(w))))
```

- Compiler can emit code to check whether a is nil

A Sermon of (Memory) Safety

- Memory bugs are very common
- We can “insert” some additional instructions for dynamic check
  - Array bound check
  - Null check

- Related tool: AddressSanitizer (ASAN)

  https://clang.llvm.org/docs/AddressSanitizer.html

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - **Arithmetic**
  - Conditionals
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Arithmetic

- In Tiger it is easy
- Binaries are straightforward
  - Each arithmetic operator corresponds to a Tree operator (e.g., BINOP(o, e1, e2)
- No unary
  - Unary negation can be implemented as subtraction from zero
  - Unary complement can be implemented as XOR with all ones
- No floating point

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - **Conditionals**
  - While Loops
  - For Loops
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Conditionals

- The result of a comparison operator will be a Cx expression
  - A statement (T_stm) s that will jump to any true-destination and false-destination
- Conditional expressions can be combined easily with Tiger operators & and |
  - e.g., $a > b \mid c < d$
- E.g., an expression such as $x < 5$ will be translated as a Cx with:

stm = CJUMP (LT, x, CONST(5), $NULL_t$, $NULL_f$)
trues = {t}
falses = {f}

Conditionals

- How to handle if-expression?

  if e_1 then e_2 else e_3

- The most straightforward thing:
  - *e1* : Cx expression;
  - *e2* and *e3* : Ex expressions
  - Apply unCx to e1
  - Apply unEx to e2 and e3
  - Make two labels t and f for the conditional
  - Allocate a temporary r
  - After label t, move e2 to r
  - After label f, move e3 to r
  - Both branches finish by jumping to a newly created "join" label

```
unCx(e1)
LABEL t
r = unEx(e2)
JUMP join
LABEL f
r = unEx(e3)
JUMP join
…
LABEL join
...
```

Correct but not very efficient

Conditionals

if e_1 then e_2 else e_3

- If e2 and e3 are both “statements” (expressions that return no value), unEx will work, but it might be better to recognize this case specially.
- If e2 or e3 is a Cx expression, unEx will yield a horrible tangle of jumps and labels.
  - recognize this case specially

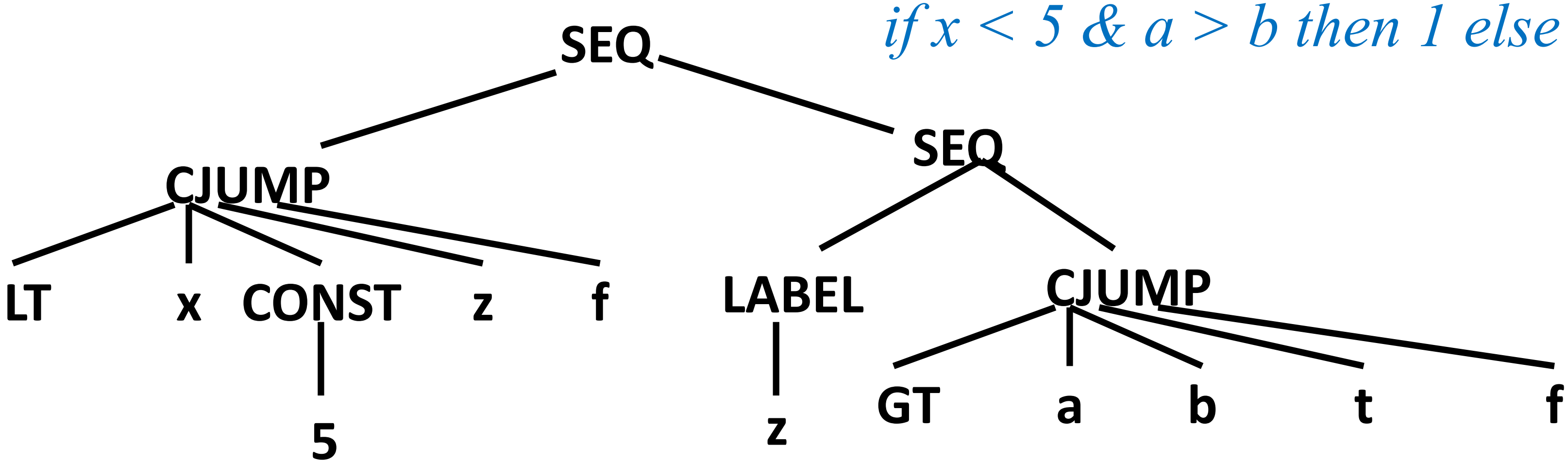
```
    case Tr_cx: {
      Temp_temp r = Temp_newtemp( );
      Temp_label t = Temp_newlabel( ), f= Temp_newlabel( );
      doPatch(e->u.cx.trues, t);
      doPatch(e->u.cx.falses, f);
      return T_Eseq(T_move(T_Temp(r),T_Const(1)),
               T_Eseq(e->u.cx.stm,
                                  T_Eseq(T_Label(f),
                   T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                     T_Eseq(T_Label(t), T_Temp(r))))));
    }
```

Example: Conditionals

Consider: ***if x < 5 then a > b else 0***

- x < 5 translates into Cx(s1)
- a > b will be translated as Cx(s2)

Similar to
if x < 5 & a > b then 1 else 0



SEQ(S1(z,f), SEQ(LABEL Z, s2(t,f)))

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - **While Loops**
  - For Loops
  - Function Call
- Translation of declarations
  - Variable Definition
  - Function Definition

While Loops

- The general layout of a while loop is:

```
test:
    if not(condition) goto done
    body
    goto test
done:
```

- If a break statement occurs within the body (and not nested within any interior while statements), the translation is simply a JUMP to *done*
  - How to know the done label?
  - Translation of break statements (the transExp function) needs to have a new formal parameter *break*, which is set to the *done* label of the nearest enclosing loop

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - **For Loops**
  - Function Call
- **Translation of declarations**
  - Variable Definition
  - Function Definition

For Loops

- A straightforward approach to translate for statements
  - rewrite the *abstract syntax* into the abstract syntax of the let/while expression shown.

```
for i:= lo to hi
do body
```

```
let var i := lo
    var limit := hi
in while i <= limit
   do (body; i:= i+1)
end
```

For Loops

- A straightforward approach to translate for statements

```
for i:= lo to hi
do body
```

```
let var i := lo
    var limit := hi
in while i <= limit
   do (body; i:= i+1)
end
```

- **Problem:**
  - When *limit=maxint*, *i+1* will overflow
- How to solve this problem?

```
if lo > hi goto done
i := lo
limit := hi
test:
        body
        if i >= limit goto done
        i := i+1
        goto test
done:
```

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - **Function Calls**
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Function Call

- Translating a function call $f(a_1, \ldots a_n)$ is simple, except that the static link must be added as an implicit extra argument:

$$\text{CALL}(\text{NAME } l_f, [sl, e_1, e_2, \ldots, e_n])$$

- l_f is the label for f ,
- sl is the static link --- *it is just a pointer to **f** 's **parent level***

Both the level of f and the level of the function calling f are required to calculate sl

Outline of the Translation

- **Translation of expressions**
  - Overview
  - Simple Variables
  - Array Variables
  - Structured L-values
  - Subscripting and Field Selection
  - Arithmetic
  - Conditionals
  - While Loops
  - For Loops
  - Function Calls
- **Translation of declarations**
  - Variable Definition
  - Function Definition

Declarations

- **Variable declaration**

  Need to figure out the **offset** in the frame, then **`move`** the expression on the r.h.s. to the proper **slot** in the **frame**.

- **Type declaration**

  No need to generate any IR tree code !

- **Function declaration**

```
        _global name
name:    ......                    prologue
        assembly code for body
        ......                     epilogue
```

Variable Declarations

- Recall that in semantic analysis, the *transDec* function updates the value t and type environment for the body of a *let* expression

- For IR translation, *transDec* should return an extra result as initialization of a variable
  - Translate initializations to assignment expressions
  - That must be put just before the body of the *let*.

- If *transDec* is applied to function and type declarations, the result will be a "no-op" expression, such as Ex(CONST(0)).

Function Declarations

- Function is translated into "assembly language segment" with three components
  - **Prologue**
  - **Body**
  - **Epilogue**

```
         _global name
name:    ......                  prologue
         assembly code for body
         ......                  epilogue
```

Function Declarations: Prologue

- A **prologue** contains:
  1. Pseudo-instructions to mark the beginning of a function (needed in the particular assembly language)
  2. A label definition of the function name
  3. An instruction to adjust the stack pointer (to allocate a new frame)
  4. Instructions to save "escaping" arguments (including the static link) into the frame, and to move nonescaping arguments into fresh temporary registers
  5. Store instructions to save any callee-save registers (including the return address register) used within the function

Function Declarations: Body and Epilogue

- The **body** of a Tiger function is an expression
  6. The translated expression

- An **epilogue** contains:
  7. An instruction to move the return value (result of the function) to the register
  8. Load instructions to restore the callee-save registers
  9. An instruction to reset the stack pointer (to deallocate the frame)
  10. A return instruction (JUMP to the return address)
  11. Pseudo-instructions, as needed, to announce the end of a function

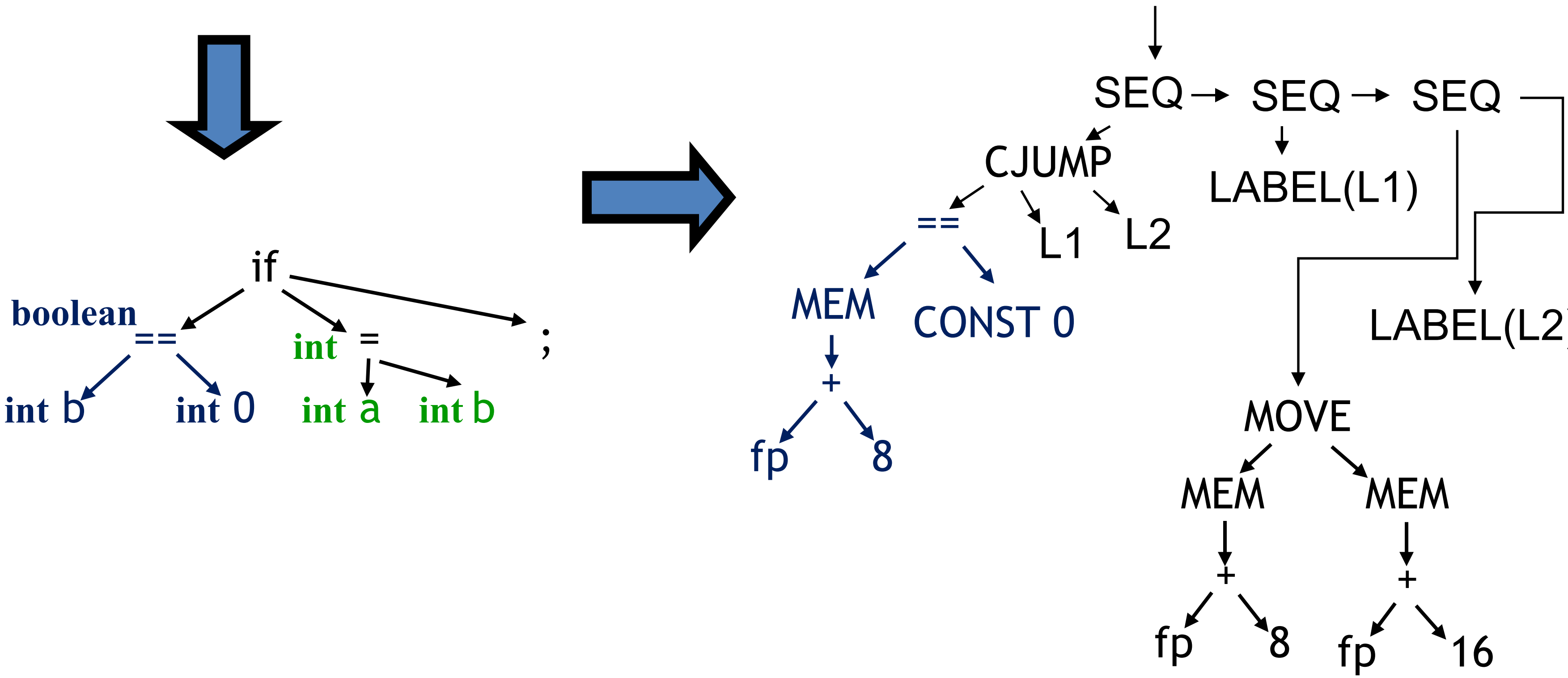
Function Declarations

– Steps 1, 3, 9, 11 depend on exact size of stack frame.
– These are generated late (after register allocation).

1. Pseudo-instructions to mark the beginning of a function (needed in the particular assembly language)
2. A label definition of the function name
3. An instruction to adjust the stack pointer (to allocate a new frame)
4. Instructions to save "escaping" arguments (including the static link) into the frame, and to move nonescaping arguments into fresh temporary registers
5. Store instructions to save any callee-save registers (including the return address register) used within the function
6. The translated expression for body
7. An instruction to move the return value (result of the function) to the register
8. Load instructions to restore the callee-save registers
9. An instruction to reset the stack pointer (to *deallocate the frame*)
10. A return instruction (JUMP to the return address)
11. Pseudo-instructions, as needed, to announce the end of a function

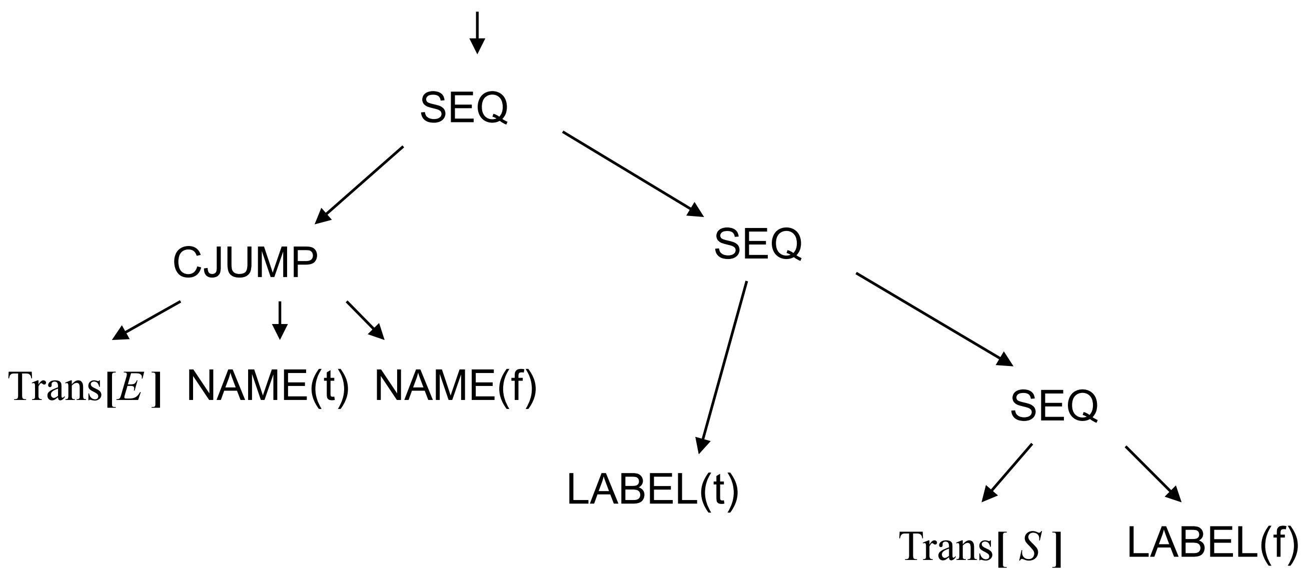
Example: Tiger AST to IR Tree

if (b==0) a = b;



Example: General Case

- Trans [if (E) S] = ?




```
= SEQ(CJUMP(Trans[E], NAME(t), NAME(f)),
            SEQ(LABEL(t),
                    SEQ(Trans[S], LABEL(f));
```

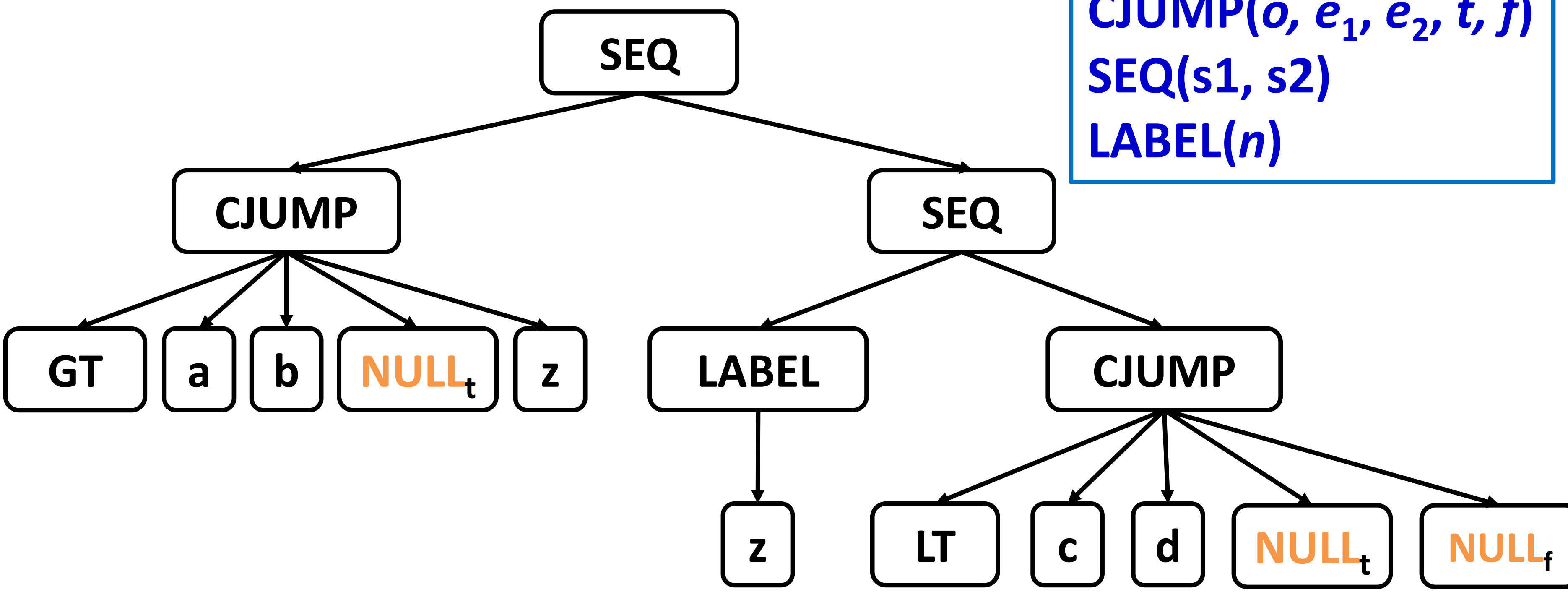
Thank you all for your attention

Kinds of Expressions

a>b | c<d

Temp_label z = Temp_newlabel ( );
T_stm s1 = T_Seq(T_Cjump(T_gt,a,b, $NULL_t$, z),
T_Seq (T_Label (z),
T_Cjump (T_lt,c,d, $NULL_t$, $NULL_f$)));

CJUMP(o, e_1, e_2, t, f)
SEQ(s1, s2)
LABEL(n)



Kinds of Expressions

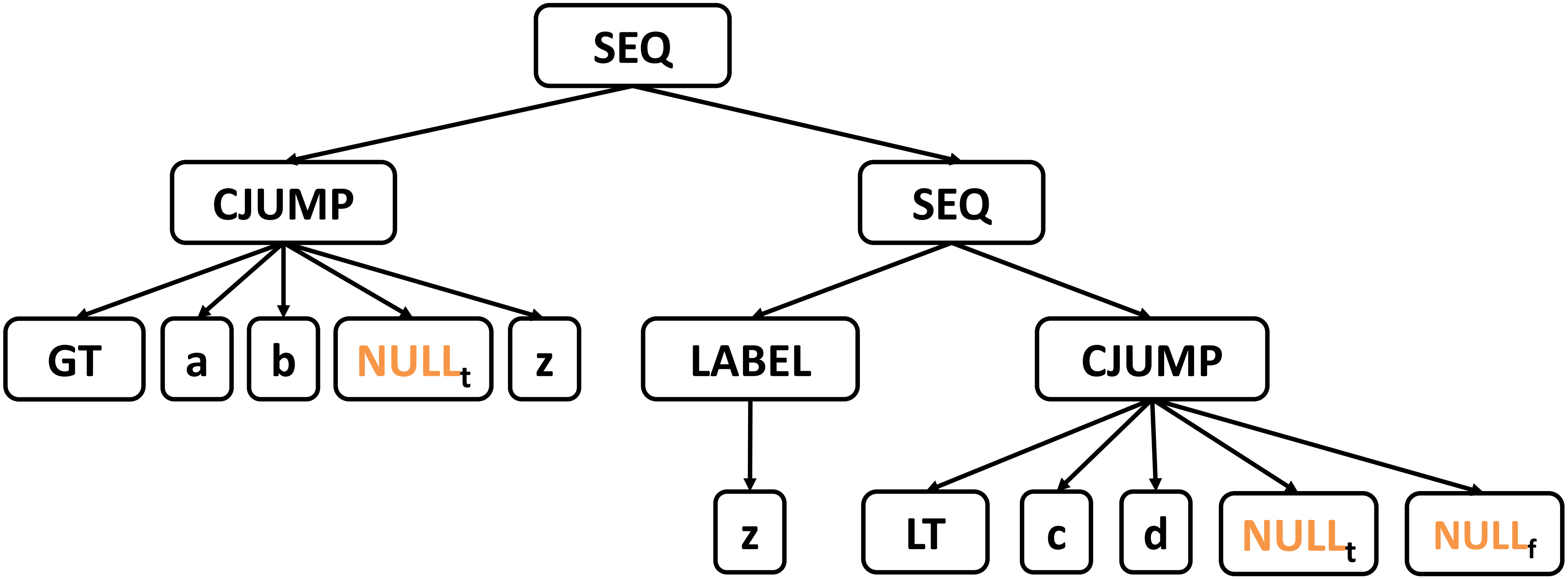
- We won't know the true-destination and false-destination until much later.
- We make a list of places which are now filled in with NULL and need to be filled in with t when t is known, and another list of all the places that need to be filled in with f.
  - True patch list and false patch list

```
typedef struct patchList_ * patchList;
struct patchList_ { Temp_label *head; patchList tail; };
static patchList patchList(Temp_label *head, patchList tail);
```

Kinds of Expressions

```
patchList trues = PatchList(
  &s1->u.SEQ.left->u.CJUMP.true,
    PatchList(&s1->u.SEQ.right->u.SEQ.right->u.CJUMP.true,
NULL));
patchList falses = PatchList(
  &s1->u.SEQ.right->u.SEQ.right->U.CJUMP.false, NULL);

Tr_exp e1 = Tr_Cx (trues, falses, s1);
```



关于unEx, unNx, unCx的实现

```
static T_exp unEx(Tr_exp e) {
switch (e->kind) {
  case Tr_ex:
    return e->u.ex;
  case Tr_cx: {
    Temp_temp r = Temp_newtemp( );
    Temp_label t = Temp_newlabel( ), f= Temp_newlabel( );
    doPatch(e->u.cx.trues, t);
    doPatch(e->u.cx.falses, f);
    return T_Eseq(T_move(T_Temp(r),T_Const(1)),
             T_Eseq(e->u.cx.stm, T_Eseq(T_Label(f),
               T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                 T_Eseq(T_Label(t), T_Temp(r))))));
  }
  case Tr_nx:
    return T_Eseq(e->u.nx, T_Const(0));
}
assert(0);
}
```

关于unEx, unNx, unCx的实现

```
static T_exp unEx(Tr_exp e) {
switch (e->kind) {
  case Tr_ex:
    return e->u.ex;
  case Tr_cx: {
    Temp_temp r = Temp_newtemp( );
    Temp_label t = Temp_newlabel( ), f= Temp_newlabel( );
    doPatch(e->u.cx.trues, t);
    doPatch(e->u.cx.falses, f);
    return T_Eseq(T_move(T_Temp(r),T_Const(1)),
             T_Eseq(e->u.cx.stm, T_Eseq(T_Label(f),
               T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                 T_Eseq(T_Label(t), T_Temp(r))))));
  }
  case Tr_nx:
    return T_Eseq(e->u.nx, T_Const(0));
}
assert(0);
}
```

关于unEx, unNx, unCx的实现

if Cx
    return 1
else
    return 0

MOVE(TEMP r, 1)
e
LABEL(f)
MOVE(TEMP r, 0)
LABEL(t)
TEMP(r)

```
static T_exp unEx(Tr_exp e) {
  switch (e->kind) {
    case Tr_ex:
      return e->u.ex;
    case Tr_cx: {
      Temp_temp r = Temp_newtemp( );
      Temp_label t = Temp_newlabel( ), f= Temp_newlabel( );
      doPatch(e->u.cx.trues, t);
      doPatch(e->u.cx.falses, f);
      return T_Eseq(T_move(T_Temp(r),T_Const(1)),
                T_Eseq(e->u.cx.stm,
                                   T_Eseq(T_Label(f),
                    T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                      T_Eseq(T_Label(t), T_Temp(r))))));
    }
    case Tr_nx:
      return T_Eseq(e->u.nx, T_Const(0));
  }
  assert(0);}
```

关于unEx, unNx, unCx的实现

```
MOVE(TEMP r, 1)
e
LABEL(f)
MOVE(TEMP r, 0)
LABEL(t)
TEMP(r)
```

e = *(a > b) | (c < d)*

```
MOVE(TEMP r, 1)
CJUMP(GT, a, b, t, z)
LABEL(z)
CJUMP(LT, c, d, t, f)
LABEL(f)
MOVE(TEMP r, 0)
LABEL(t)
TEMP(r)
```

关于unEx, unNx, unCx的实现

```
void doPatch (patchList tList, Temp_label label) {
  for ( ; tList; tList = tList->tail)
    *(tList->head) = label;
}

patchList joinPatch (patchList first, patchList second) {
  if (!first) return second;
  for (; first->tail; first = first->tail);
  first->tail = second;
  return first;
}
```