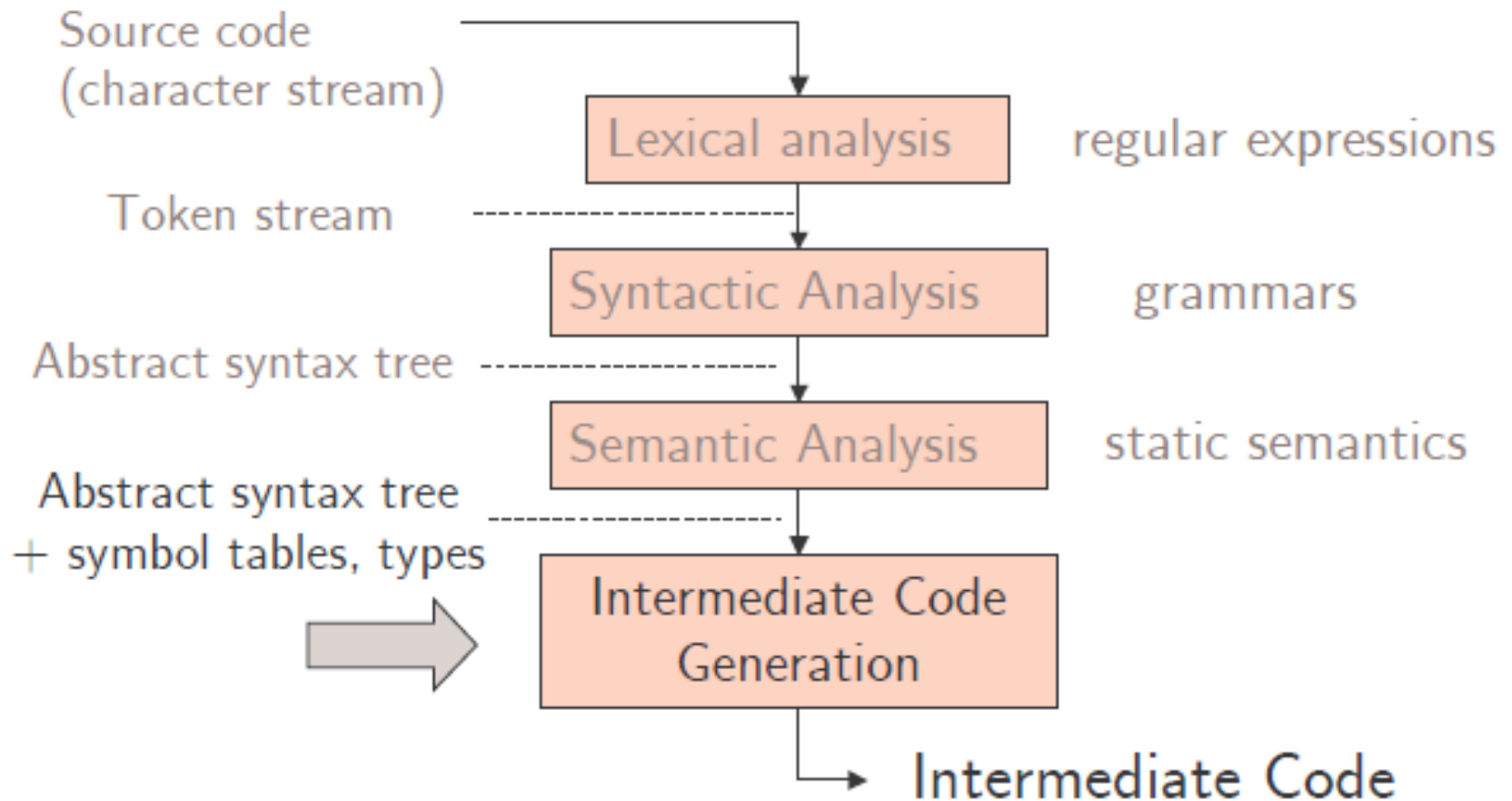


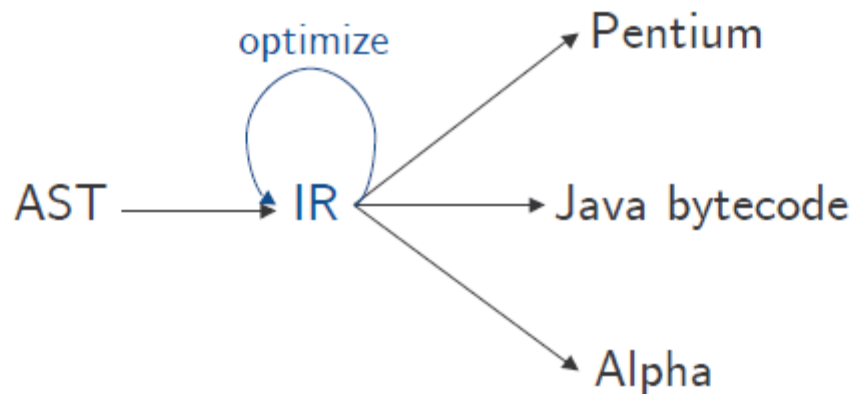
# Intermediate Representation (รูปแบบการแทนในระยะกลาง)

# เฟสต่างๆ ในคอมไพเลอร์ที่ผ่านมา



# Intermediate Representation (IR)

- จุดประสงค์หลักของการมี IR เพื่อจะทำให้การปรับปรุงโค้ด (optimization) และ การแปลงโค้ด (transformation) เป็นอิสระต่อรายละเอียดของฮาร์ดแวร์และของตัวภาษาระดับสูง (Language and Machine Independence)

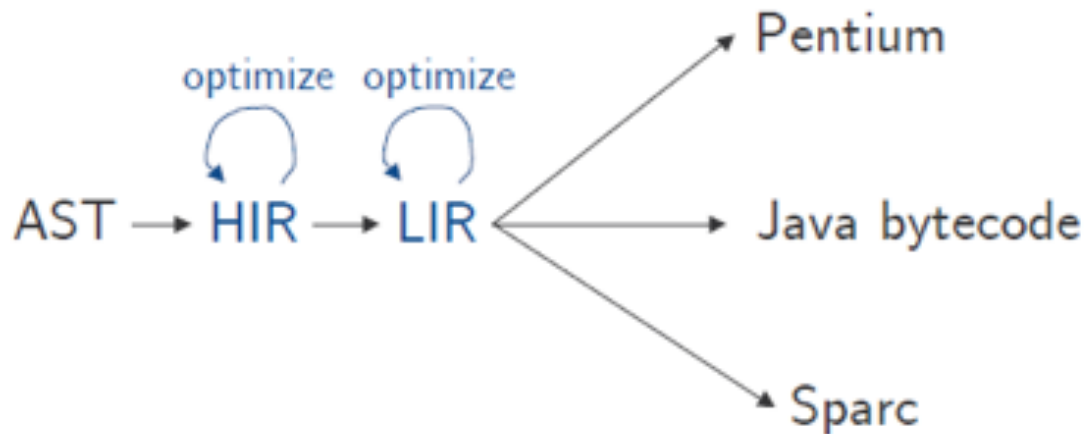


# คุณสมบัติของ IR ที่ดี

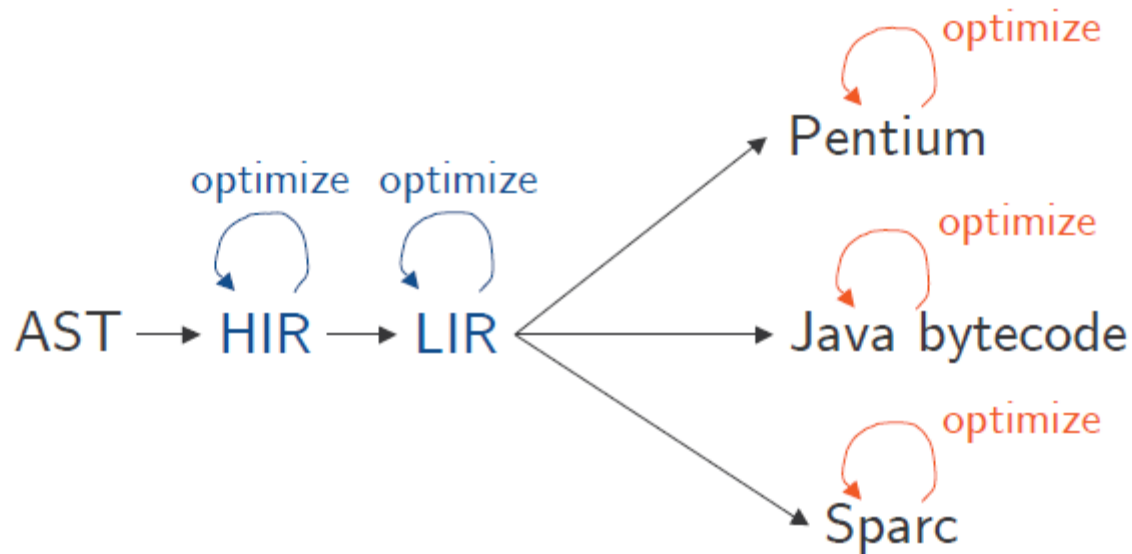
- แปลจาก AST ได้ง่าย
- แปลไปเป็น assembly ได้ง่าย
- เป็นตัวเชื่อมต่อประสานที่ลดช่องว่างระหว่าง AST กับ assembly โค้ด
  - Optimize ได้ง่าย
  - สามารถแปลงเป็น assembly ในหลากหลายรูปแบบได้ง่าย (easy to retarget)
- ขนาดของ node ในแต่ละเฟสของคอมไพเลอร์
  - AST > 40
  - IR ประมาณ 13
  - Pentium assembly มีมากกว่า 200 ชนิด

# การใช้ IR หลายลำดับชั้น

- การทำ optimization บางอย่าง ทำได้ดีและง่ายถ้ารูปแบบการแทนอยู่ในระดับที่ใกล้เคียงกับภาษาระดับสูง
- แต่ optimization บางอย่างเหมาะที่จะทำในรูปแบบการแทนในระดับล่าง
- ในกรณีเช่นนี้เราสามารถใช้การ IR หลายลำดับชั้นในการแก้ปัญหานี้

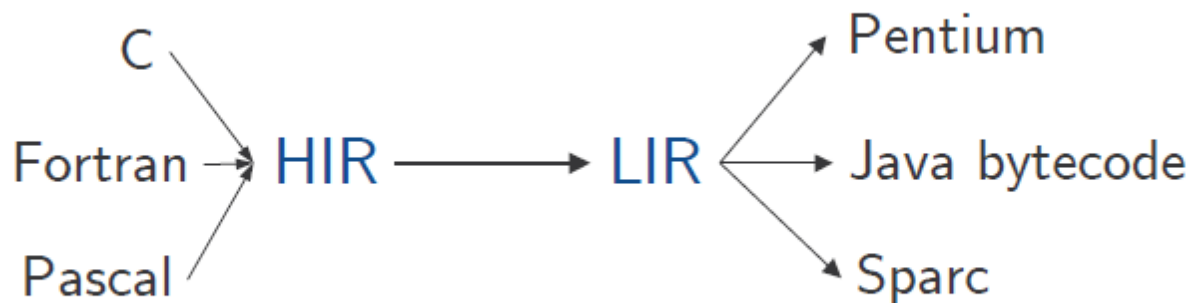


# การปรับปรุง โค้ด ในระดับล่าง



# IR หลายระดับ

- ส่วนใหญ่ทำกัน 2 ระดับ
  - High-Level IR ซึ่งไม่ขึ้นกับภาษาแต่จะใกล้เคียงกับภาษาที่จะทำการแปล
  - Low-Level IR ซึ่งไม่ขึ้นกับ CPU ฮาร์ดแวร์แต่จะใกล้เคียงกับภาษาแอสเซมบลีของ CPU
- การแปลจาก HIR ไปเป็น LIR ไม่ขึ้นกับ CPU และภาษาโปรแกรมตั้งต้น



# High-Level IR

- คือ AST นั่นเอง
  - การออกแบบ AST เราจะต้องให้มีความยืดหยุ่น ใช้งานได้กับภาษาโปรแกรมหลายๆรูปแบบ
- ยังรักษาโครงสร้างในภาษาระดับสูงอยู่
  - Struct หรือ array หรือ variable
  - if หรือ while หรือ assignment
- สามารถใช้ในการ optimize โค้ดในระดับบน
  - เช่นการ optimize nested loop



# Low-Level IR


- เป็นการแทน โค้ด ในรูปแบบของ abstract machine
- โครงสร้าง ใกล้เคียงกับ assembly
  - การ jump ไปในที่ใดๆของ โค้ด
  - คำสั่งที่ปฏิบัติการในระดับล่างเช่นการย้ายข้อมูล
- สามารถใช้ในการ optimize โค้ดในระดับล่าง
  - Register allocation
  - Branch prediction

# การออกแบบ low-level IR

- มีทางเลือกหลายรูปแบบ
  - Three-address code:  $a = b \text{ OP } c$
  - การแทนด้วย tree
  - Stack machine (เช่น Java bytecodes)
- ข้อเด่นของแต่ละรูปแบบ
  - Three-address code ง่ายต่อการทำ dataflow analysis เพื่อ optimize โค้ด
  - การแทนด้วย tree ง่ายต่อการผลิตโค้ดและเลือกคำสั่งในระดับล่าง
  - Stack machine ง่ายต่อการผลิตโค้ดสำหรับเครื่องที่มีสถาปัตยกรรมแบบ stack

# Three-Address Code

- เราจะเขียน IR แบบ three-address code:  $a = b \text{ OP } c$
- มี address ได้อย่างมากที่สุด 3 address ต่อหนึ่งคำสั่ง IR แต่อาจจะมีน้อยกว่านี้
- บางคนเรียกการแทนแบบนี้ว่า quadruple เพราะสามารถแทนได้ในลักษณะ  $(a, b, c, \text{OP})$
- ตัวอย่าง:

$a = (b+c) * (-e);$    $t1 = b + c$   
 $t2 = -e$   
 $a = t1 * t2$

# คำสั่ง arithmetic และ logic

- มีรูปแบบดังต่อไปนี้ โดย OP แทน operations ที่เป็นไปได้เช่นตัวอย่างด้านล่าง

$a = b \text{ OP } c$

$a = \text{OP } b$

- Arithmetic operations: ADD, SUB, DIV, MUL
- Logic operations: AND, OR, XOR
- Comparisons: EQ, NEQ, LE, LEQ, GE, GEQ
- Unary operations: MINUS, NEG

# คำสั่ง data movement

- Copy instruction: `a = b`

- Load/store instructions:

`a = *b`                      `*a = b`

– Models a load/store machine

- Address-of instruction (if language supports it):

`a = &b`

- Array accesses:

`a = b[i]`                      `a[i] = b`

- Field accesses:

`a = b.f`                      `a.f = b`

# คำสั่ง branch

- Label instruction:

`label L`

- Unconditional jump: go to statement after label L

`jump L`

- Conditional jump: test condition variable a; if true, jump to label L

`cjump a L`

- Alternative: two conditional jumps:

`tjump a L`      `fjump a L`

# คำสั่ง call

- มีรูปแบบที่เป็นทั้ง statement หรือ assignment
  - Statement: `call f(a1, ... , an)`
  - Assignment: `a = call f(a1, ... , an)`
- ในระดับ IR ยังไม่ลงรายละเอียดถึงการส่งผ่าน argument หรือการเซ็ท stack frame

# ตัวอย่างการแปล: while loop

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



```
n = 0  
label test  
t2 = n < 10  
t3 = not t2  
cjump t3 end  
label body  
n = n + 1  
jump test  
label end
```



# ตัวอย่างการแปล: if-else

```
m = 0;  
if (c == 0) {  
    m = m + n * n;  
} else {  
    m = m + n;  
}
```



```
m = 0  
t1 = c == 0  
cjump t1 trueb  
m = m + n  
jump end  
label trueb  
t2 = n * n  
m = m + t2  
label end
```

# การแปลงเป็น IR

- ใช้ syntax-directed translation
  - เริ่มจาก AST
  - สำหรับแต่ละ node ของ AST ให้มีนิยามการแปลงเป็น IR
  - จากนั้น traverse AST และแปลงเป็น IR โดยใช้ recursion
- การแปลงแบบนี้สามารถจัดการกับ nested construct เช่น nested if หรือ while ได้โดยธรรมชาติและตรงไปตรงมา

# สัญลัักษณ์ (Notation)

- $T[e]$  แทนโค้ด low-level IR โดย  $e$  แทน high-level IR (หรือ node ใน AST)
  - นั่นคือ  $T[e]$  เป็นชุดคำสั่งของ low-level IR
- ถ้า  $e$  เป็น expression  $t := T[e]$  แทนผลลัพธ์ของการ evaluate  $e$  หลังจากแปลงเป็นชุดคำสั่ง IR และนำผลลัพธ์นั้นเก็บไว้ที่ตัวแปร  $t$
- ถ้า  $v$  เป็นตัวแปร  $t := T[v]$  แทนคำสั่ง copy  $t = v$
- ใช้ temporary variable ในการเก็บผลลัพธ์ในระยะกลาง (intermediate value)

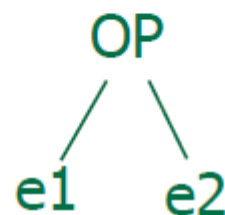
# การแปล expression

- Binary operations:  $t := T[ e1 \text{ OP } e2 ]$   
(arithmetic operations and comparisons)

$t1 := T[ e1 ]$

$t2 := T[ e2 ]$

$t = t1 \text{ OP } t2$



- Unary operations:  $t = T[ \text{OP } e ]$

$t1 := T[ e ]$

$t = \text{OP } t1$



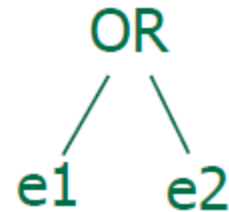
# การแปล Boolean expression

- $t := T[ e1 \text{ OR } e2 ]$

$t1 := T[ e1 ]$

$t2 := T[ e2 ]$

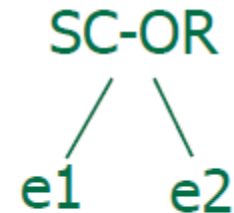
$t = t1 \text{ OR } t2$



# การแปล Boolean แบบ short-circuit

- Short-circuit OR:  $t := T[ e1 \text{ SC-OR } e2 ]$

```
t := T[ e1 ]  
tjump t Lend  
t := T[ e2 ]  
label Lend
```

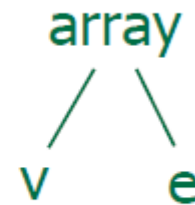


# การเข้าถึง (access) array หรือ field

- Array access:  $t := T[ v[e] ]$

$t1 := T[ e ]$

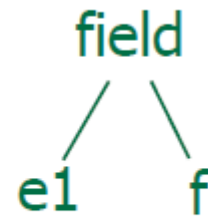
$t := v[t1]$



- Field access:  $t := T[ e1.f ]$

$t1 := T[ e1 ]$

$t := t1.f$



# การแปล statement เป็นชุด

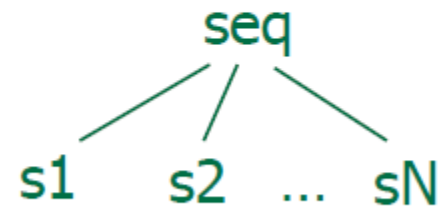
- Statement sequence:  $T[ s1; s2; \dots; sN ]$

$T[ s1 ]$

$T[ s2 ]$

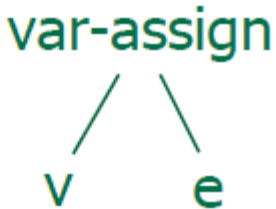
$\dots$

$T[ sN ]$

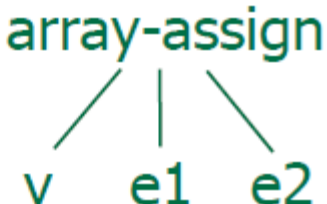




# การแปล assignment statement

- Variable assignment:  $T[ v = e ]$   
 $v := T[ e ]$   


```
graph TD; A[var-assign] --> B[v]; A --> C[e]
```

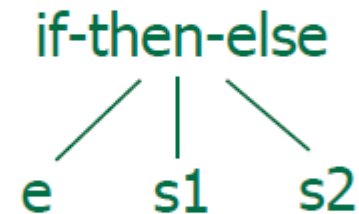
- Array assignment:  $T[ v[e1] = e2 ]$   
 $t1 := T[ e1 ]$   
 $t2 := T[ e2 ]$   
 $v[t1] = t2$   


```
graph TD; A[array-assign] --> B[v]; A --> C[e1]; A --> D[e2]
```

# การแปล if-then-else statement

- $T[ \text{if } (e) \{ s1 \} \text{ else } \{ s2 \} ]$

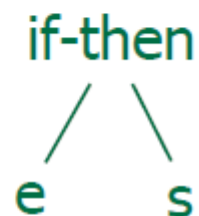
```
t1 := T[ e ]  
fjump t1 Lfalse  
T[ s1 ]  
jump Lend  
label Lfalse  
T[ s2 ]  
label Lend
```



# การแปล if-then statement

- `T[ if (e) { s } ]`

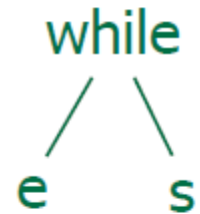
```
t1 := T[ e ]  
fjump t1 Lend  
T[ s ]  
label Lend
```



# การแปล while statement

- `T[ while (e) { s } ]`

```
label Ltest
t1 := T[ e ]
fjump t1 Lend
T[ s ]
jump Ltest
label Lend
```



# การแปล call และ return statement

- $T[ \text{call } f(e_1, e_2, \dots, e_N) ]$

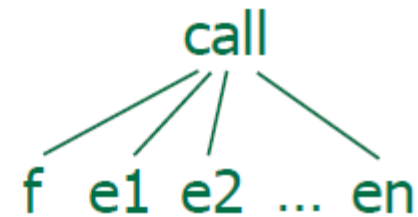
$t_1 := T[ e_1 ]$

$t_2 := T[ e_2 ]$

...

$t_n := T[ e_n ]$

$\text{call } f(t_1, t_2, \dots, t_n)$



- $T[ \text{return } e ]$

$t := T[ e ]$

$\text{return } t$

