اصول طراحی کامپایلرها

حسین کارشناس

دانشکده مهندسی کامپیوتر

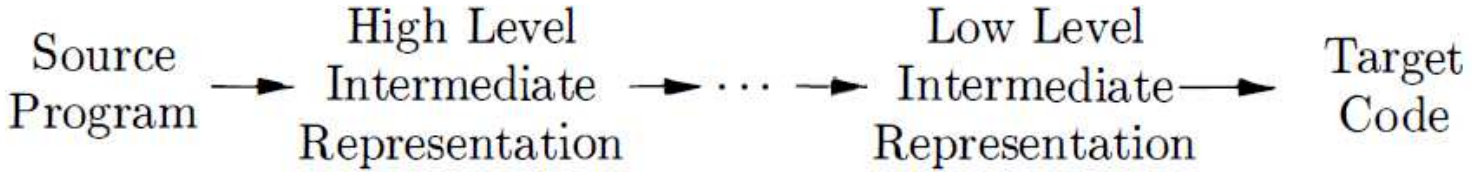
ترم اول ۹۸ – ۹۷

تولید کد میانی
(Intermediate Code Generation)

تولید کد میانی

- هدف: ترجمه برنامه ورودی به یک زبان میانی
  - ورودی: برنامه ورودی ارزیابی شده یا نمایشی از آن
    - مثلاً AST برنامه ورودی
  - خروجی: برنامه به زبان میانی
    - واسط بین قسمت‌های پیشین و پسین کامپایلر
  - تسهیل کننده فرآیند بهینه‌سازی و ترجمه نهایی به زبان هدف

- امکان استفاده از دنباله‌ای از نمایش‌های میانی

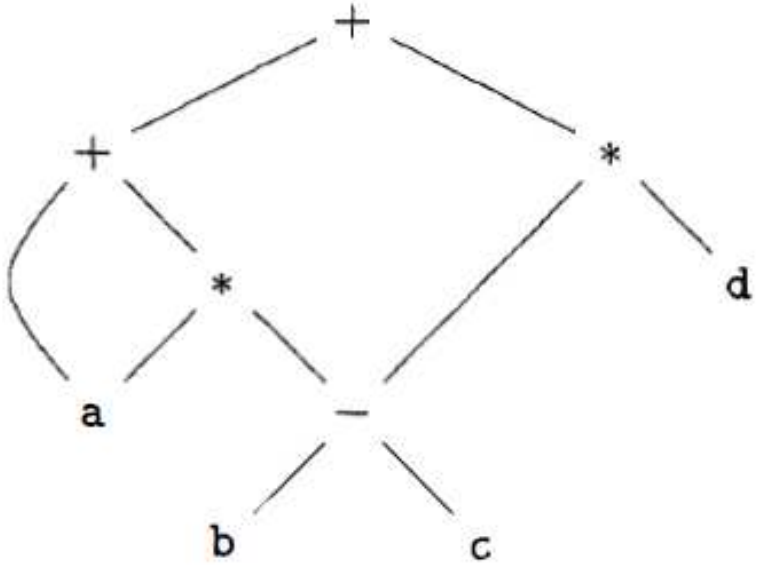


تولید کد میانی

- ویژگی‌های یک نمایش میانی خوب
  - به راحتی قابل ترجمه به زبان هدف باشد
  - برخی اطلاعات لازم در مورد ساختار برنامه ورودی را نشان دهد
    - لازم برای بهینه‌سازی برنامه
  - وابسته به معماری ماشین خاص نباشد
- کدهای سه آدرسه (three address code)
  - بر اساس دو مفهوم اصلی دستور (کد) و آدرس
  - مانند یک زبان اسمبلی سطح بالا
  - شکل کلی دستورات: x = y op z

تولید کد میانی

- ویژگی‌های کد سه آدرسه
  - هر دستور حاوی یک عملیات ساده
    - بسیاری از کدهای سه آدرسه مستقیماً به یک دستور در زبان هدف تبدیل می‌شوند
  - امکان استفاده از تعداد دلخواه متغیرهای موقتی
- مثالی از کد سه آدرسه برای عبارت a + a * (b – c) + (b – c) * d



$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

تولید کد میانی

- انواع آدرس‌های ممکن در کدهای سه آدرسه
  - نام‌ها (names)
    - نام متغیرها و توابع تعریف شده در برنامه ورودی
  - ثابت‌ها (constants)
  - متغیرهای موقت (temporaries) تولید شده توسط کامپایلر
    - برای نگهداری نتایج میانی عملیات
    - تسهیل جابجایی دستورات
      - مناسب برای بهینه‌سازی

تولید کد میانی

- انواع دستورات بکارگرفته شده در کدهای سه آدرسه
  - دستورات انتساب
    - عملگرهای دوتایی: $x = y\ op\ z$
    - عملگرهای تکی: $x = op\ y$
    - کپی ساده: $x = y$
    - کپی از یک خانه حافظه: $x = y[i]$
    - کپی به یک خانه حافظه: $x[i] = y$
    - انتساب آدرس: $x = \&y$
    - کپی از محتویات یک آدرس: $x = *y$
    - کپی محتویات به یک آدرس: $*x = y$

تولید کد میانی

- انواع دستورات بکارگرفته شده در کدهای سه آدرسه (ادامه)
  - دستورات پرش به یک برچسب
    - غیرشرطی: goto L
    - به شرط x صحیح: if x goto L
    - به شرط x غلط: ifFalse x goto L
    - به شرط برقراری رابطه *relop*: if x *relop* y goto L
  - دستورات فراخوانی توابع
    - تعیین پارامترها: param x
    - فرخوانی p با n پارامتر: call p, n
    - فرخوانی p با n پارامتر و مقدار بازگشتی: y = call p, n
    - بازگشت مقدار: return y

تولید کد میانی

- پیاده‌سازی تابع gen(e, t) برای تولید کد میانی هر دستور یا عبارت e
  - مثال: کد میانی برای دستور انتساب و عبارت‌های ریاضی

$S \rightarrow \mathbf{id} = E\ ;\ S$	
$\mid R = E\ ;\ S$	
$\mid \epsilon$	
$E \rightarrow E + E$	gen(E_1+E_2, t): { gen(E_1, t_1); gen(E_2, t_2); pr(t = t_1 + t_2); }
$\mid E \times E$	gen(E_1×E_2, t): { gen(E_1, t_1); gen(E_2, t_2); pr(t = t_1 × t_2); }
$\mid -E$	gen(-E_1, t): { gen(E_1, t_1); pr(t = neg t_1); }
$\mid \mathbf{id}$	gen(id, t): { pr(t = id.lexeme); }
$\mid \mathbf{num}$	gen(num, t): { pr(t = num.lexeme); }
$\mid R$	
$\mid (\ E\ )$	gen((E1), t): { gen(E_1, t); }
$R \rightarrow R\ [\ E\ ]$	
$\mid \mathbf{id}\ [\ E\ ]$	

تولید کد میانی

- مثال: یک SDT برای تولید کد میانی دستور انتساب و عبارات ریاضی

$$
\begin{array}{lll}
S \rightarrow \mathbf{id} = E\ ; & \{\ gen(\ top.get(\mathbf{id}.lexeme)\ '='\ E.addr);\ \} \\
\quad \mid\ L = E\ ; & \{\ gen(\ L.array.base\ '['\ L.addr\ ']'\ '='\ E.addr);\ \} \\
E \rightarrow E_1 + E_2 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ E_1.addr\ '+'\ E_2.addr);\ \} \\
\quad \mid\ - E_1 & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ '\mathbf{minus}'\ E_1.addr);\ \} \\
\quad \mid\ (\ E_1\ ) & \{\ E.addr = E_1.addr;\ \} \\
\quad \mid\ \mathbf{id} & \{\ E.addr = top.get(\mathbf{id}.lexeme);\ \} \\
\quad \mid\ L & \{\ E.addr = \mathbf{new}\ Temp\,();\ gen(E.addr\ '='\ L.array.base\ '['\ L.addr\ ']');\ \}
\end{array}
$$

- ویژگی addr دربردارنده آدرس مورد استفاده در کد سه آدرسه است
- تابع gen() یک کد سه آدرسه تولید می‌کند

تولید کد میانی

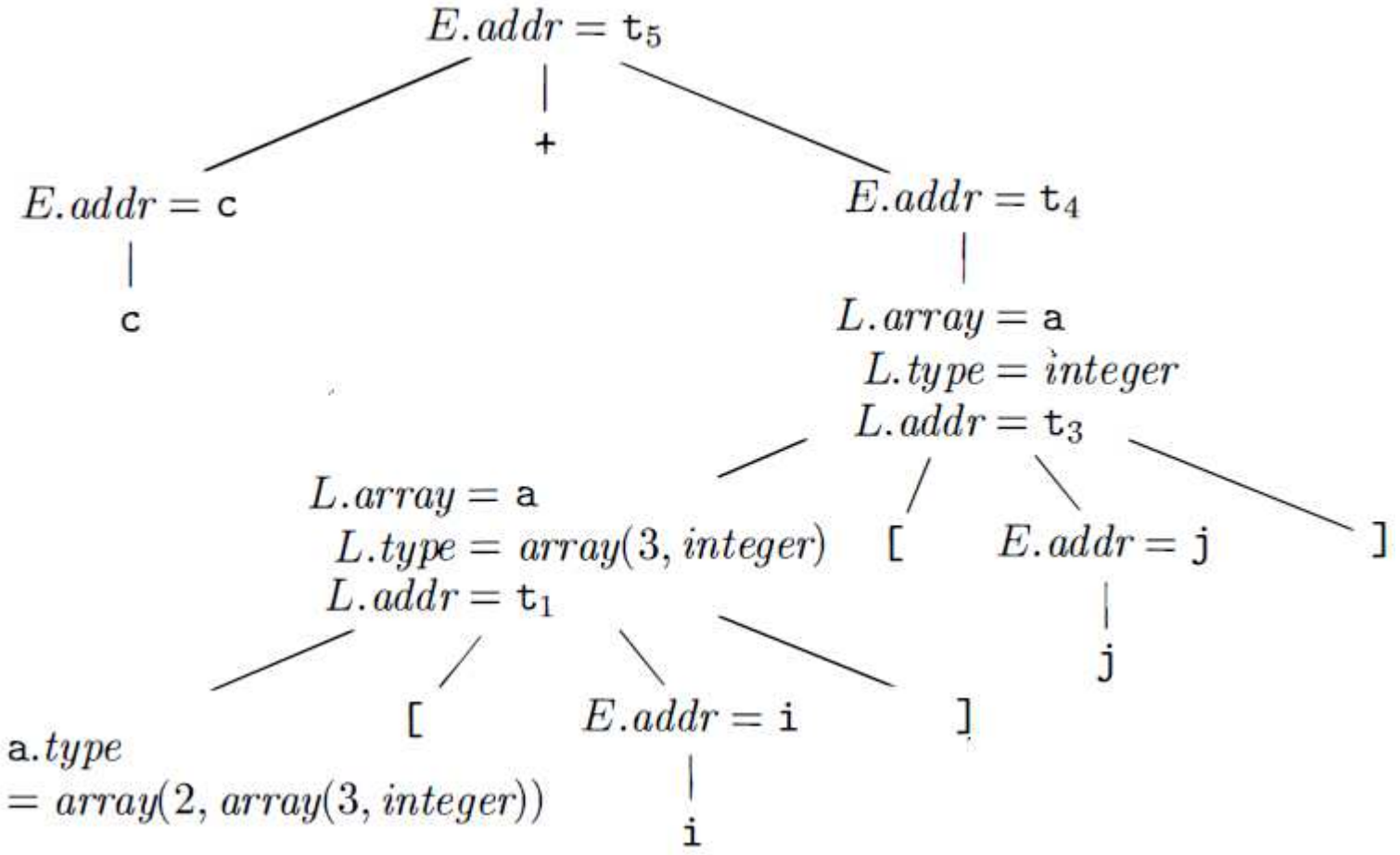
- مثال: یک SDT برای ترجمه آدرس‌دهی به آرایه

$$
\begin{array}{lll}
L \rightarrow \textbf{id}\ [\ E\ ] & \{ & L.array = top.get(\textbf{id}.lexeme); \\
 & & L.type = L.array.type.elem; \\
 & & L.addr = \textbf{new}\ Temp\,(); \\
 & & gen(L.addr\ '='\ E.addr\ '*'\ L.type.width);\ \} \\
\mid\ L_1\ [\ E\ ] & \{ & L.array = L_1.array; \\
 & & L.type = L_1.type.elem; \\
 & & t = \textbf{new}\ Temp\,(); \\
 & & L.addr = \textbf{new}\ Temp\,(); \\
 & & gen(t\ '='\ E.addr\ '*'\ L.type.width); \\
 & & gen(L.addr\ '='\ L_1.addr\ '+'\ t);\ \}
\end{array}
$$

- ویژگی array یک نشانه‌رو به سطر مربوط به آرایه در جدول نشانه‌ها است
  - فیلد elem نوع هر یک از ابعاد آرایه را به دست می‌دهد

تولید کد میانی

- مثال: درخت تجزیه (مشروح) برای عبارت `c+a[i][j]`



تولید کد میانی

- دستورات کنترل جریان اجرا
- تولید کدهای میانی با پیش‌فرض اجرای پشت سرهم (sequential) دستورات
- وجود ساختارهای برنامه‌نویسی ویژه برای تغییر جریان اجرا
  - مثال: دستورات شرطی (if) و تکرار (مانند while)
- تولید کد میانی این ساختارها با استفاده از دستورات پرش (jump)
  - دستور goto شرطی یا غیرشرطی
  - مقصد پرش توسط یک برچسب تعیین می‌شود
- برچسب‌گذاری (labeling) دستورات
  - تعیین آدرس کد هدف پس از تولید کد میانی تمام برنامه
- بیان شرط در پرش‌های شرطی با استفاده از عبارت‌های بولی

تولید کد میانی

- مثال: کد میانی معادل دستور زیر

```
do
      i = i + 1;
while (a[i] < v);
```

```
L:   t1 = i + 1
     i = t1
     t2 = i * 8
     t3 = a [ t2 ]
     if t3 < v goto L
```

کنترل جریان اجرا

- نیاز به تعیین نوع عبارت بولی با استفاده از بستر (context) آن
  - مثال: یک گرامر نمونه برای تولید عبارت‌های بولی

$$B \rightarrow B \,||\, B \mid B \,\&\&\, B \mid !\, B \mid (\, B \,) \mid E \;\mathbf{rel}\; E \mid \mathbf{true} \mid \mathbf{false}$$

  - عبارت‌های بولی معمولی
    - قابل ترجمه مانند عبارت‌های ریاضی
  - عبارت‌های بولی در دستورات کنترل جریان اجرا
    - تعیین کننده شرط پرش
  - تفاوت در کد میانی تولید شده برای دو نوع عبارت بولی
    - برای هر یک از مقادیر true و false:
      - تولید کد میانی برای دستورهای پرش و اختصاص برچسب برای مقصدهای پرش

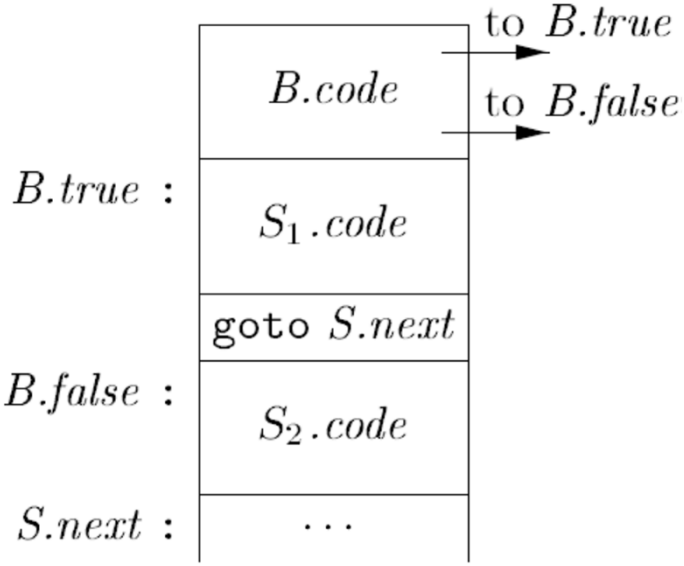
کنترل جریان اجرا

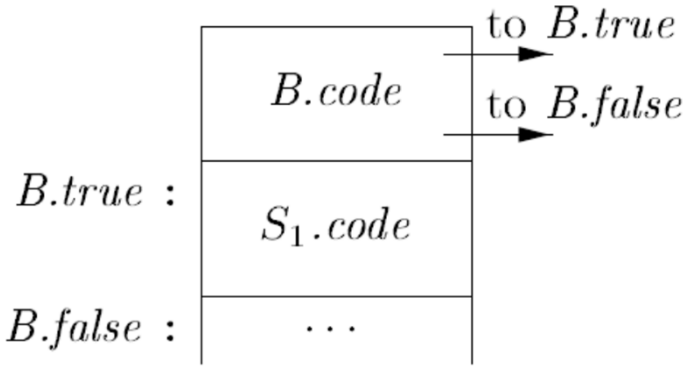
- ساختار کدهای میانی تولید شده

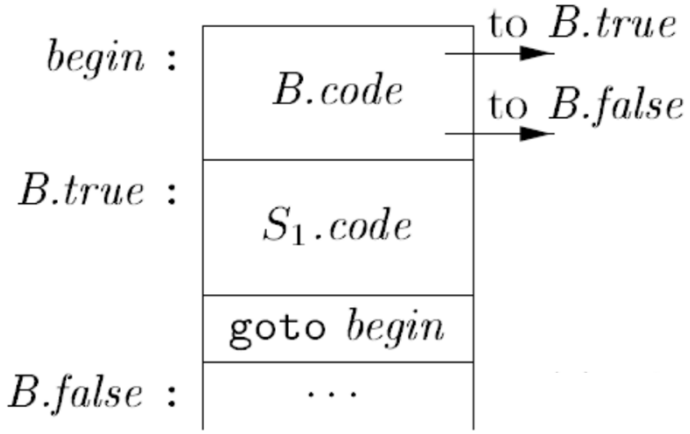
$S \rightarrow$ **if** (B) S_1

$S \rightarrow$ **if** (B) S_1 **else** S_2

$S \rightarrow$ **while** (B) S_1







کنترل جریان اجرا

- مثال: کد میانی برای دستورات کنترل اجرا

$S \rightarrow$ **if (** B **)** S_1

```
gen(S, -): {
    gen(B, t);
    L1 = newLabel();
    pr(ifFalse t goto L1)
    gen(S1, -);
L1:
}
```

| **if (** B **)** S_1 **else** S_2

```
gen(S, -): {
    gen(B, t);
    L1 = newLabel();
    pr(ifFalse t goto L1)
    gen(S1, -);
    L2 = newLabel();
    pr(goto L2)
L1: gen(S2, -);
L2:
}
```

| **while (** B **)** S_1

```
gen(S, -): {
    L1 = newLabel();
L1: gen(B, t);
    L2 = newLabel();
    pr(ifFalse t goto L2)
    gen(S1, -);
    pr(goto L1)
L2:
}
```

| **other**

تولید کد میانی

- مثال: یک SDD برای تولید برچسب در کد مربوط به دستور while
  - بکارگیری ویژگی‌های موروثی next، false و true
  - بکارگیری ویژگی ساختی code

$S \rightarrow \textbf{while}\ (\ C\ )\ S_1$	$L1 = new();$ $L2 = new();$ $S_1.next = L1;$ $C.false = S.next;$ $C.true = L2;$ $S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code$

- SDT معادل

$S \rightarrow \textbf{while}\ ($	$\{\ L1 = new();\ L2 = new();\ C.false = S.next;\ C.true = L2;\ \}$
$C\ )$	$\{\ S_1.next = L1;\ \}$
S_1	$\{\ S.code = \textbf{label} \parallel L1 \parallel C.code \parallel \textbf{label} \parallel L2 \parallel S_1.code;\ \}$

کنترل جریان اجرا

- مثال: کد میانی برای عملگرهای منطقی میانبر (shortcut)

$B \rightarrow B_1$ **||** B_2

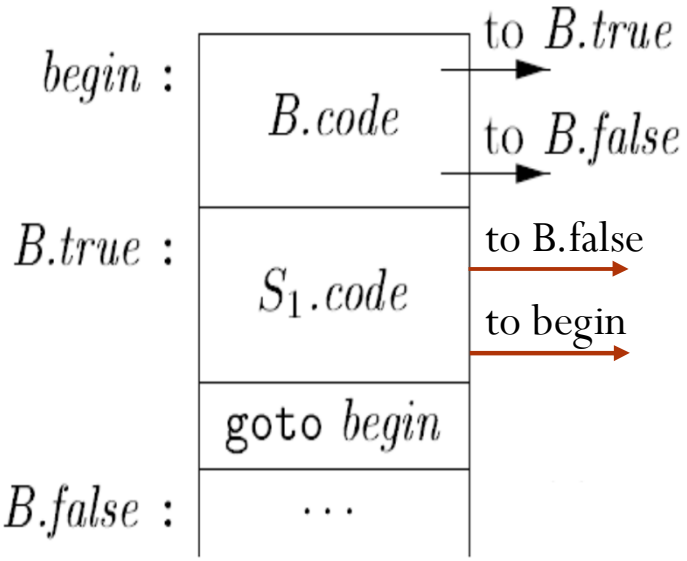
```
gen(B, t): {
    gen(B1, t1);
    L1 = newLabel();
    pr(if t1 goto L1)
    gen(B2, t2);
    L2 = newLabel();
    pr(ifFalse t2 goto L2)
L1: pr(t = 1)
    L3 = newLabel();
    pr(goto L3)
L2: pr(t = 0)
L3:
}
```

| B_1 **&&** B_2

```
gen(B, t): {
    gen(B1, t1);
    L1 = newLabel();
    pr(ifFalse t1 goto L1)
    gen(B2, t2);
    L2 = newLabel();
    pr(if t2 goto L2)
L1: pr(t = 0)
    L3 = newLabel();
    pr(goto L3)
L2: pr(t = 1)
L3:
}
```

کنترل جریان اجرا

- تولید کد میانی برای دستورات break و continue
  - مشخص کردن ساختار تکرار حاوی دستور
    - برای break: نگهداری برچسب دستور بعد از بدنه ساختار تکرار
    - برای continue: نگهداری برچسب اولین دستور شرط ساختار تکرار



- تولید کد پرش در محل مربوطه
  - برای break: کد پرش به بعد از دستور تکرار
  - برای continue: به ابتدای دستور تکرار

کنترل جریان اجرا

- دستور switch-case
  - یک انشعاب چندتایی (n-way branch)

- روند اجرا
  - ارزیابی عبارت انتخاب کننده (E)
  - پیدا کردن مقدار متناظر (V_i)
  - اجرای دستور مرتبط با مقدار پیدا شده (S_i)
  - در صورت عدم تطابق با تمام V_iها اجرای دستور S_n

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn−1: Sn−1
        default: Sn
}
```

- کد میانی تولید شده حاوی دنباله‌ای از پرش‌های شرطی است

کنترل جریان اجرا

- مثالی از کد میانی برای دستور switch-case

```
switch ( E ) {
        case V1: S1
        case V2: S2
                 ...
        case Vn-1: Sn-1
        default: Sn
}
```

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Jump Table

کنترل جریان اجرا

- راهکارهای مختلف پیاده‌سازی دنباله پرش‌ها
  - ایجاد یک جدول از جفت‌های <مقدار، برچسب>
    - بررسی تکراری مقدار عبارت انتخاب کننده (E) با هر یک از خانه‌های جدول
  - ایجاد یک جدول درهم (hashtable)
    - جستجوی مقدار عبارت انتخاب کننده (E) به عنوان کلید در جدول
    - خانه متناظر با هر کلید حاوی برچسب (L_i) دستور مرتبط (S_i) است
  - ایجاد یک آرایه حاوی برچسب‌های دستورات S_i
    - اندازه آرایه $[\max(V_i) - \min(V_i)]$
    - استفاده از عبارت انتخاب کننده (E) برای اندیس‌دهی به آرایه
    - اندیس‌های فاقد مقدار متناظر حاوی برچسب پیش‌فرض (default)

کنترل جریان اجرا

- فراخوانی توابع
  - تولید کد میانی برای محاسبه آرگومان‌ها
  - تولید کد میانی برای فراخوانی تابع
    - تعیین آرگومان‌ها
    - فراخوانی تابع

- مثال: کد میانی تولید شده برای فراخوانی یک تابع

```
n = f(a[i]);
```

```
t1 = i * 4
t2 = a [ t1 ]
param t2
t3 = call f, 1
n = t3
```

بهینه‌سازی کد

بهینه‌سازی کد

- هدف: بهبود بکارگیری منابع در برنامه
  - زمان اجرا
  - اندازه کد تولید شده
  - بکارگیری شبکه، مصرف انرژی، ...
- اهمیت صحت بهینه‌سازی
  - عملکرد برنامه نباید تغییر کند
- پیچیده‌ترین مرحله کامپایل
  - معمولاً بیشترین زمان کامپایل را به خود اختصاص می‌دهد
  - نیاز به در نظر گرفتن توجیه‌پذیری برخی از انواع بهینه‌سازی

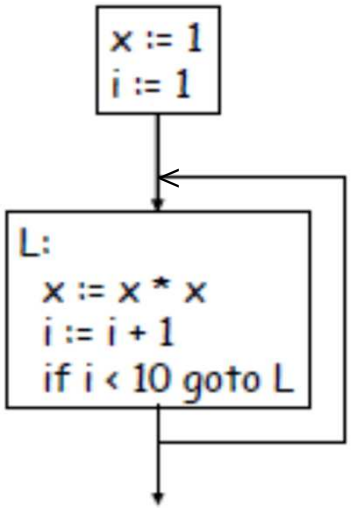
بهینه‌سازی کد

- بلاک پایه (basic block)
  - قطعه کد بیشینه‌ای که فقط دارای یک نقطه ورود و یک نقطه خروج باشد
    - دارای هیچ برچسبی بجز برای اولین دستور نباشد
    - دارای هیچ دستور پرشی بجز آخرین دستور نباشد
  - امکان پرش به یک بلاک پایه مگر در ابتدای آن وجود ندارد
  - امکان پرش به بیرون یک بلاک پایه مگر از انتهای آن وجود ندارد
  - مثال:

```
1.  L:
2.     t := 2 * x
3.     w := t + x
4.     if w > 0 goto L.
```

بهینه‌سازی کد

- گراف کنترل جریان اجرا (control-flow graph)
  - گره‌ها نشان دهنده بلاک‌های پایه
  - یال‌ها نشان دهنده انتقال جریان اجرا از یک بلاک به بلاک دیگر
    - بین بلاک A و B یال وجود دارد اگر و فقط اگر اولین دستور بلاک B بتواند بعد از آخرین دستور A بیاید
  - نمایش کد برنامه بصورت گراف‌های کنترل جریان اجرا
  - مثال:



بهینه‌سازی کد

- سطوح مختلف بهینه‌سازی
  - بهینه‌سازی محلی
    - بهینه‌سازی بلاک‌های پایه بصورت مستقل
  - بهینه‌سازی سراسری
    - بهینه‌سازی نحوه انتقال اجرا بین بلاک‌های پایه
      - بهینه‌سازی گراف‌های کنترل جریان اجرا بصورت مستقل
      - به ازای هر رویه (procedure) یک گراف کنترل جریان اجرا داریم
  - بهینه‌سازی بین رویه‌ای (inter-procedural)
    - بهینه‌سازی نحوه انتقال اجرا بین رویه‌های مختلف برنامه

- پیاده‌سازی برخی از انواع بهینه‌سازی به علت پیچیدگی توجیه‌پذیر نیست

بهینه‌سازی محلی

- در سطح یک بلاک پایه اعمال می‌شود

- ساده‌ترین نوع بهینه‌سازی کد
  - بکارگیری مجموعه‌ای از گام‌های ساده بهینه‌سازی
  - با اعمال هر گام معمولاً گام‌های بهینه‌سازی دیگری امکان‌پذیر می‌شود
- اکثر کامپایلرها این نوع بهینه‌سازی را انجام می‌دهند
  - بکارگیری تکراری بهینه‌سازی تا وقتی که گام بهینه سازی دیگری ممکن نباشد
  - ممکن است شرط توقف سقف زمانی برای بهینه‌سازی باشد

بهینه‌سازی محلی

- متداول‌ترین گام‌های بهینه‌سازی
  - ساده‌سازی جبری، یکی‌سازی ثابت‌ها
  - حذف عبارت‌های مشترک، حذف کد مرده
- ساده‌سازی جبری (algebraic simplification)
  - حذف و ساده‌سازی برخی از دستورات
  - مثال:

```
x := x + 0
x := x * 1
x := x * 0          ⇒  x := 0
y := y ** 2         ⇒  y := y * y
x := x * 8          ⇒  x := x << 3
x := x * 15         ⇒  t := x << 4; x := t - x
```

بهینه‌سازی محلی

- تبدیل به شکل انتساب تکی (single assignment)
  - هر نام فقط یکبار به عنوان مقصد دستورات قرار گیرد
  - باعث ساده‌سازی بعضی انواع بهینه‌سازی محلی می‌شود
  - مثال:

```
x := z + y          b := z + y
a := x      ⇒       a := b
x := 2 * x          x := 2 * b
```

- تبدیل انتشار بازنویسی (copy propagation)
  - استفاده از مبدأ انتساب اگر انتساب به صورت تکی باشد
  - مثال:

```
b := z + y          b := z + y
a := b      ⇒       a := b
x := 2 * a          x := 2 * b
```

بهینه‌سازی محلی

- یکی‌سازی ثابت‌ها (constant folding)
  - نتیجه عملیاتی که فقط شامل ثابت‌ها باشد در زمان کامپایل محاسبه می‌شود
  - مثال:

```
x := 2 + 2  ⇒ x := 4
```

  - مثال: حذف دستور

```
if 2 < 0 jump L
```

  - مثال: ترکیب تبدیل انتشار بازنویسی و یکی‌سازی ثابت‌ها

```
a := 5                a := 5
x := 2 * a      ⇒     x := 10
y := x + 6            y := 16
t := x * y            t := x << 4
```

بهینه‌سازی محلی

- حذف عبارت مشترک (common subexpression elimination)
  - مورد استفاده در دستورات انتساب با سمت راست یکسان
    - نام‌های موجود در دستور اول تا رسیدن به دستور دوم نباید مقصد انتساب دیگری باشند
  - مثال:

```
x := y + z          x := y + z
...          ⇒      ...
w := y + z          w := x
```

- حذف کد مرده (dead code elimination)
  - اگر مقصد یک دستور انتساب اصلاً استفاده نشود حذف می‌شود
  - مثال:

```
x := z + y          b := z + y          b := z + y
a := x       ⇒      a := b       ⇒      x := 2 * b
x := 2 * a          x := 2 * b
```

بهینه‌سازی محلی

- مثال: بهینه‌سازی محلی یک بلاک پایه

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

```
a := x ** 2
b := 3
c := x
d := c * c
e := b * 2
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := c * c
e := b << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 3 << 1
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := x * x
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + d
g := e * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

```
a := x * x
b := 3
c := x
d := a
e := 6
f := a + a
g := 6 * f
```

```
a := x * x


f := a + a
g := 6 * f
```