اصول طراحی کامپایلر

درس ۲۳

تولید کد نهایی

Code Generation

کاظم فولادی قلعه
دانشکده مهندسی، پردیس فارابی
دانشگاه تهران

http://courses.fouladi.ir/compiler

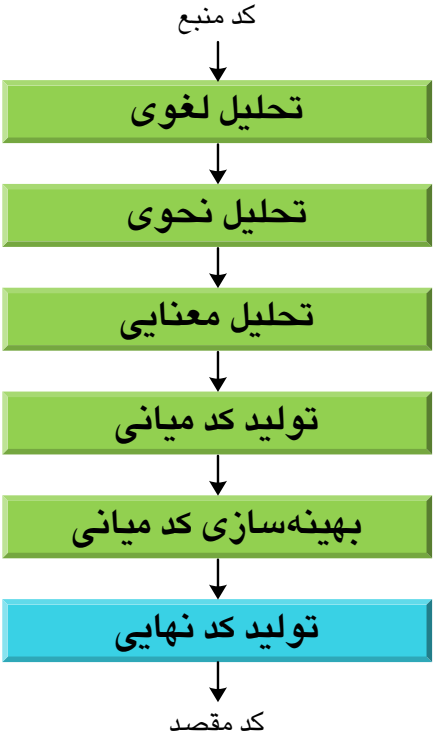
تولید کد نهایی

۱

مقدمه

مراحل فرآیند کامپایل در یک نگاه

تولید کد نهایی



تولید کد


دنباله‌ی اجزای کد میانی
sequence of interm. code parts
مولد کد
Code Generator
دنباله‌ی اجزای کد نهایی
optimal sequence of final code parts

تولید کد

وظایف مولد کد

- انتساب حافظه به متغیرها
- انتساب متغیرها به ثبات‌ها
- ترجمه به دستورالعمل‌های کد اسمبلی
- استفاده از مشخصات معماری سخت‌افزار
- ...

تولید کد

مثال


```
temp1 := id3 * 60.0
id1 := id2 + temp1
```

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

تولید کد نهایی

۲

مسائل تولید کد

تولید کد

مسائل اصلی در تولید کد مناسب

اساس تولید کد نهایی،
اعمال یک نگاشت مشخص از هر دستورالعمل کد میانی به دنباله‌ای از دستورالعمل‌های کد نهایی است.

مسائل اصلی در تولید کد مناسب

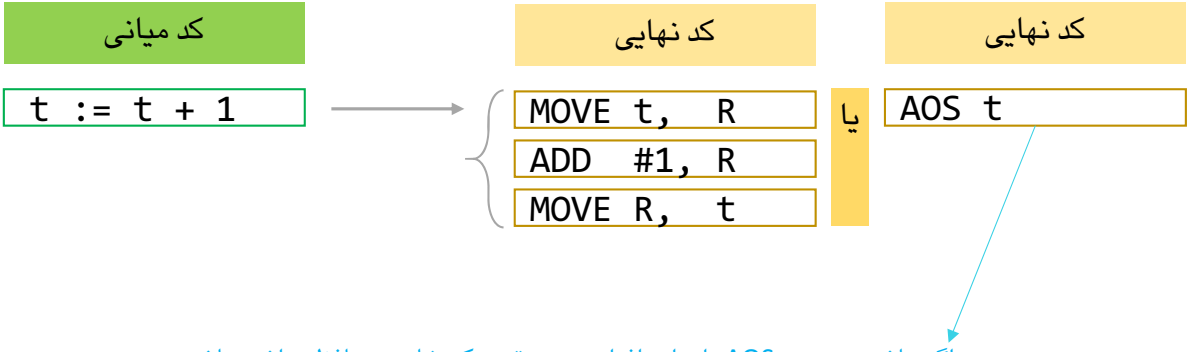
انتخاب کارآمدترین دستورالعمل‌ها	تصمیم‌گیری در مورد ترتیب یک محاسبه	تصمیم‌گیری در مورد ثبات‌های مورد استفاده
برای انجام محاسبات مشخص شده با دستورهای سه‌آدرسی	برای تولید کد کارآمدتر	برای استفاده‌ی بهینه از ثبات‌های محدود

مسائل اصلی در تولید کد مناسب

انتخاب کارآمدترین دستورالعمل‌ها

بیشتر ماشین‌ها این امکان را فراهم می‌کنند که یک محاسبه با بیش از یک روش انجام شود.

مثال:



اگر ماشین دستور AOS را برای افزایش مستقیم یک خانه‌ی حافظه داشته باشد

انتخاب اینکه کدام دنباله از دستورالعمل‌ها بهتر است، خود یک مسئله است:
این تصمیم‌گیری نیازمند دانشی وسیع‌تر در مورد **کد برنامه** و **ماشین مقصد** است.

مسائل اصلی در تولید کد مناسب

تصمیم‌گیری در مورد ترتیب محاسبات

برخی ترتیب‌های یک محاسبه به تعداد ثبات کمتری برای نگهداری نتایج میانی نیاز دارند.

مثال: محاسبه‌ی عبارت

```
a + b - (e - (c + d))
```

کد میانی

```
t1 := a + b
t2 := c + d
T3 := e - t2
t4 := t1 - t3
```

کد نهایی

```
MOVE a,  R1
MOVE b,  R2
ADD  R1, R2
MOVE c,  R1
MOVE d,  R3
ADD  R1, R3
MOVE e,  R1
SUB  R1, R3
SUB  R2, R3
MOVE R3, t
```

(a + b) - (e - (c + d))

یا

کد نهایی

```
MOVE a,  R1
MOVE b,  R2
ADD  R1, R2
MOVE e,  R1
SUB  R2, R1
MOVE c,  R2
ADD  R2, R1
MOVE d,  R2
ADD  R2, R1
MOVE R1, t
```

(((a + b) - e) + c) + d

تصمیم‌گیری در مورد بهترین ترتیب بسیار مشکل است.

مسائل اصلی در تولید کد مناسب

تصمیم‌گیری در مورد ثبات‌ها

تصمیم‌گیری در مورد اینکه کدام ثبات باید در محاسبه استفاده شود، مسئله‌ی دیگری است که در تولید کد خوب مؤثر است.

وقتی در یک ماشین، دستورالعمل‌ها از یک زوج ثبات استفاده می‌کنند که
یکی از آنها هم به‌عنوان عملوند و هم به عنوان نتیجه به‌کار می‌رود، مسئله‌ی انتخاب ثبات‌ها دشوارتر می‌شود.

مثال: دستورالعمل جمع در پردازنده‌ی 8086

```
ADD AX, BX
```

اصول طراحی کامپایلر

تولید کد نهایی

۳

مسئله‌ی تخصیص ثبات‌ها

مسئله‌ی تخصیص ثبات‌ها

مسئله‌ی تخصیص ثبات‌ها
Register Allocation Problem

اختصاص ثبات‌ها به متغیرهای موقتی برنامه
(بدون تغییر رفتار برنامه)

هدف: بازنویسی کد به‌گونه‌ای که
از تعداد کمتری متغیر موقتی نسبت به تعداد ثبات‌های ماشین مقصد استفاده شود.

مسئله‌ی تخصیص ثبات‌ها

مثال

با فرض اینکه متغیرهای a و e پس از استفاده می‌میرند

⇓

متغیر موقتی a می‌تواند پس از محاسبه‌ی a + b دوباره استفاده شود؛
متغیر موقتی e می‌تواند پس از محاسبه‌ی e - 1 دوباره استفاده شود.

⇓

هر سه متغیر a، e و f را می‌توان به یک ثبات r1 تخصیص داد.

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: ایده‌ی پایه

مقدار موجود در یک متغیر موقتی مرده، در ادامه‌ی محاسبات دیگر لازم نیست

⇓

می‌توان آن متغیر مرده را مجدداً استفاده کرد *(reuse)*.

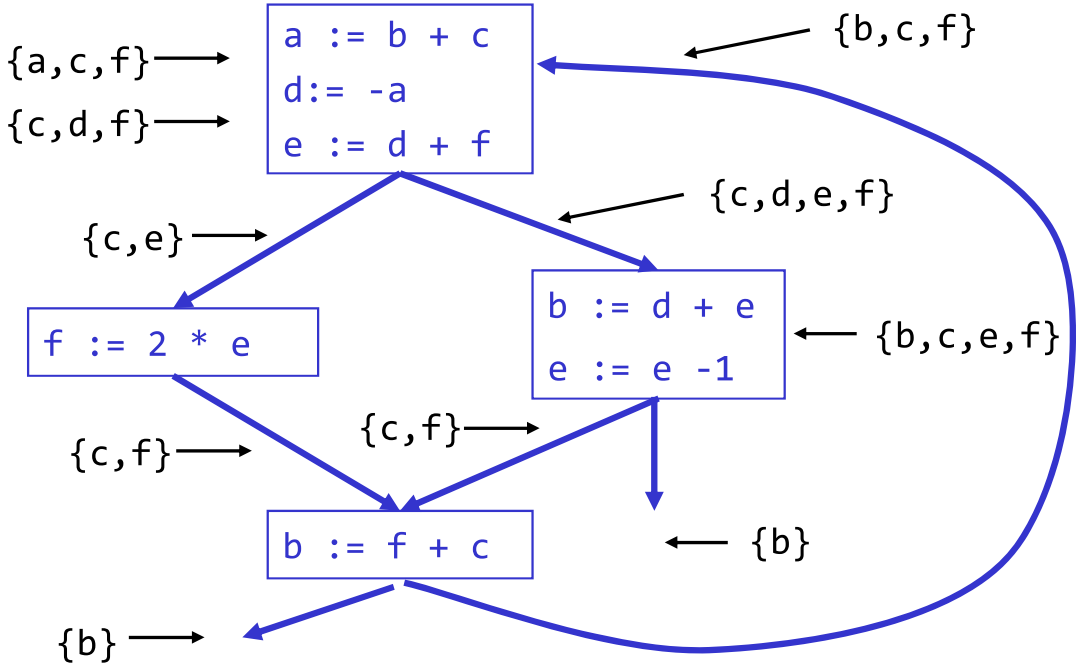
قاعده‌ی پایه

دو متغیر موقتی t_1 و t_2 می‌توانند در یک ثبات مشترک r قرار بگیرند،
اگر **در هر نقطه از برنامه، حداکثر یکی از آنها زنده باشد.**

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: مرحله‌ی تحلیل زنده بودن متغیرها

متغیرهای زنده در هر نقطه از برنامه را با بررسی **گراف جریان** مشخص می‌کنیم.



مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: گراف دخالت ثبات‌ها

دو متغیر موقتی که همزمان زنده هستند، نمی‌توانند به یک ثبات تخصیص داده شوند.

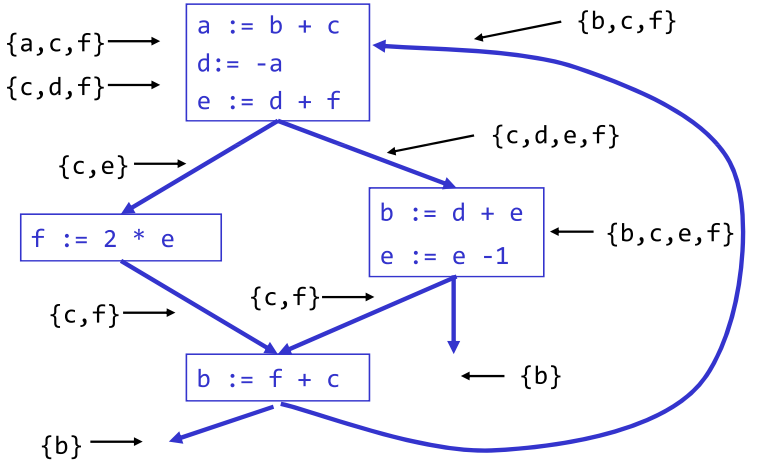
برای تشخیص همزمان زنده بودن، یک گراف می‌سازیم:

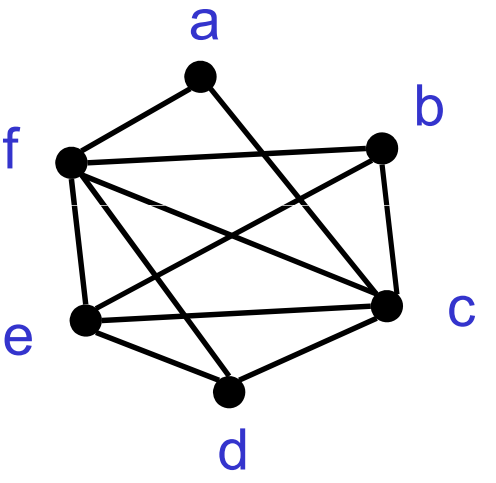
گراف دخالت ثبات‌ها *Register Interference Graph*	
گره‌ها *Nodes*	**یال‌ها** *Edges*
برای هر متغیر موقتی یک گره در نظر گرفته می‌شود.	بین گره‌ی متغیرهایی که به طور همزمان در یک نقطه از برنامه زنده هستند، یال رسم می‌شود.

دو متغیر موقتی می‌توانند به یک ثبات تخصیص داده شوند، اگر هیچ یالی آن دو را به هم متصل نکرده باشد.

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: گراف دخالت ثبات‌ها: مثال





b و c نمی‌توانند در یک ثبات قرار گیرند.
b و d نمی‌توانند در یک ثبات قرار گیرند.

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: گراف دخالت ثبات‌ها: خصوصیات

گراف دخالت ثبات‌ها
Register Interference Graph

این گراف اطلاعات لازم برای تشخیص تخصیص‌های مجاز ثبات‌ها را به طور کامل استخراج می‌کند.

این گراف یک نمای سراسری (روی کل گراف جریان برنامه) از نیازمندی‌های ثبات برنامه را به‌دست می‌دهد.

پس از ساخت این گراف، الگوریتم تخصیص ثبات، مستقل از معماری ماشین مقصد خواهد بود.

مسئله‌ی رنگ‌آمیزی گراف

رنگ‌آمیزی گراف
Graph Coloring

یک رنگ‌آمیزی گراف، یک انتساب از رنگ‌ها به گره‌های گراف است
به طوری که گره‌های مجاور (متصل با یک یال) همرنگ نباشند.

یک گراف k–رنگ‌پذیر است، اگر یک رنگ‌آمیزی با k رنگ داشته باشد.

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: تخصیص ثبات‌ها از طریق رنگ‌آمیزی گراف دخالت ثبات‌ها

مسئله‌ی k–رنگ‌پذیری گراف:

k = تعداد ثبات‌های ماشین

رنگ‌ها = نام ثبات‌های ماشین

تخصیص **رنگ** به هر **گره‌ی گراف**

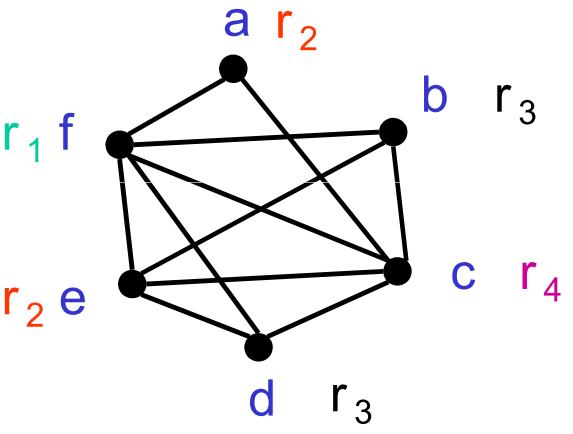
⇓

تخصیص **ثبات** به هر **متغیر موقتی**

اگر یک گراف k–رنگ‌پذیر باشد،
آن‌گاه یک تخصیص ثبات وجود خواهد داشت که بیش از k ثبات نیاز ندارد.

مسئله‌ی تخصیص ثبات‌ها

الگوریتم تخصیص ثبات: تخصیص ثبات‌ها از طریق رنگ‌آمیزی گراف دخالت ثبات‌ها: مثال



هیچ رنگ‌آمیزی‌ای با کمتر از **۴ رنگ** وجود ندارد

⇓

حداقل **۴ ثبات** نیاز داریم

(چندین رنگ‌آمیزی با ۴ رنگ وجود دارد)

اصول طراحی کامپایلر

تولید کد نهایی

۴

مدیریت حافظه‌ی نهان

کامپایلر و مدیریت حافظه‌های نهان/ثبات‌ها

کامپایلرها برای مدیریت ثبات‌ها
به‌خوبی عمل می‌کنند. (بهتر از برنامه‌نویس)

کامپایلرها برای مدیریت حافظه‌ی نهان (cache)
چندان خوب عمل نمی‌کنند.

این مسئله به‌عهده‌ی برنامه‌نویس گذاشته می‌شود.

+

بهینه‌سازی‌های ساده‌ی کامپایلر برای حافظه‌ی نهان

کامپایلر و مدیریت حافظه‌های نهان/ثبات‌ها

مثال: بهینه‌سازی حافظه‌ی نهان توسط کامپایلر

دو حلقه‌ی تودرتو که هر دو یک کار واحد را انجام می‌دهند:

کارآمدی حافظه‌ی نهان بالا

```
for (i = 1; i < 1000; i++)
    for (j = 1; j < 10; j++)
       a[i] *= b[i];
```

کارآمدی حافظه‌ی نهان پایین

```
for (j = 1; j < 10; j++)
    for (i = 1; i < 1000; i++)
       a[i] *= b[i];
```

کامپایلر می‌تواند نظیر این بهینه‌سازی را برای حافظه‌ی نهان انجام بدهد که «**تعویض حلقه‌ها**» نام دارد.

تولید کد نهایی

۵

بهینه‌سازی کد نهایی

بهنیه‌سازی کد نهایی

بر روی کد نهایی می‌توان بهینه‌سازی‌هایی را اعمال کرد.

کد نهایی با استراتژی دستور به دستور تولید شده است.

⇓

این کد حاوی دستورالعمل‌های افزونه و ساختارهای غیربهینه است.

⇓

برای بهبود کیفیت کد نهایی بهینه‌سازی لازم است.

تکنیک اصلی بهینه‌سازی کد نهایی
بهینه‌سازی تکه‌ای *Peephole Optimization*
بهبود کد نهایی با بررسی و جایگزینی محلی در هر مرحله، دنباله‌ی کوتاهی از دستورالعمل‌های کد مقصد بررسی می‌شود و در صورت امکان با دنباله‌ی سریع‌تری جایگزین می‌شود.

بهنیه‌سازی کد نهایی

نمونه‌هایی از بهینه‌سازی تکه‌ای

PEEPHOLE OPTIMIZATION

۱ حذف دسترسی‌های اضافی به حافظه

حذف خواندن و نوشتن‌های اضافی حافظه

```
MOV R, a
MOV a, R
```

۲ حذف پرش‌های چندتایی

یک پرش به پرش دیگر

```
      GOTO  L1
L1:   GOTO  L2
L2:
```

۳ حذف کد دسترس‌ناپذیر

حذف کدی که از دسترس جریان کنترل برنامه خارج است.

۴ ساده‌سازی‌های جبری

برای مثال،

```
a + 0 = a
a – a = 0
a * 1 = a
a * 0 = 0
```

۵ کاهش پیچیدگی

برای مثال،
استفاده از دستور شیفت به چپ به‌جای

```
x = x * 2
```

۶ استفاده از ویژگی‌های ماشین

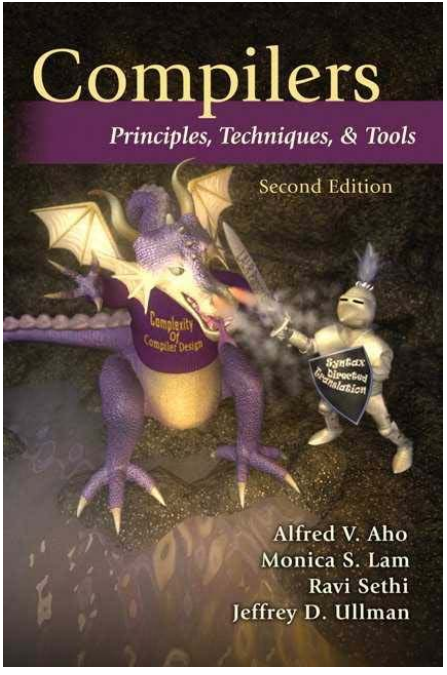
برای مثال،
وجود دستور افزایش یک برای خانه‌های حافظه

تولید کد نهایی

۶

منابع

منبع اصلی



A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman,
Compilers: Principles, Techniques and Tools,
Second Edition, Addison-Wesley, 2007.

Chapter 8 (8.1-8.4)