اصول طراحی کامپایلر

درس‌نامه‌ی ۱۹

کاظم فولادی
http://kazim.fouladi.ir

ویراست اول: ۱۳۸۸
ویراست دوم: ۱۳۹۳

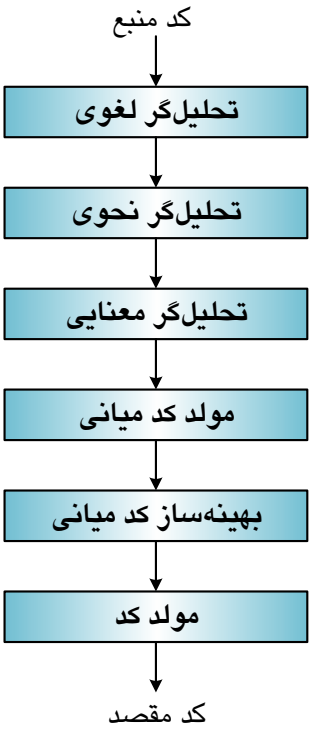
فهرست مطالب

۱۹ تولید کد

CODE GENERATION

۱۹-۱ مقدمه

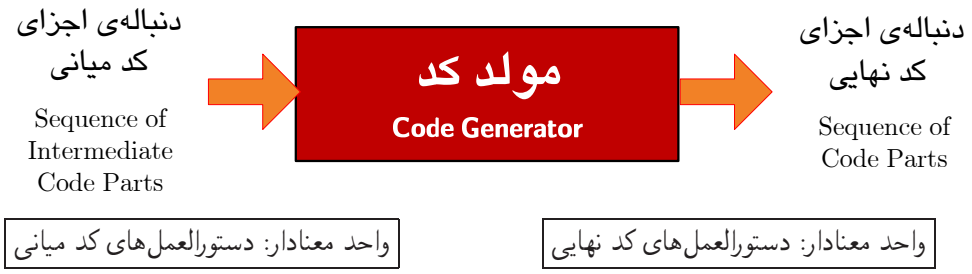
تولید کد، ششمین مرحله‌ی فرآیند کامپایل است.



شکل ۱۹-۱: تولید کد به عنوان ششمین مرحله‌ی فرآیند کامپایل

تولید کد نهایی در زبان مقصد با

- انتخاب مناسب مکان حافظه برای هر متغیر
- ترجمه‌ی هر دستورالعمل کد میانی به دنباله‌ای از دستورالعمل‌های اسمبلی کد مقصد
- انتساب مناسب متغیرها و مقادیر میانی به ثبات‌های حافظه



شکل ۱۹-۲: مرحله‌ی تولید کد نهایی

مثال

نمونه‌ای از تولید کد نهایی:

کد میانی سه‌آدرسی

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

↓

کد اسمبلی مقصد

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

۱۹-۲ 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، اما در این مرحله نیز مسائلی وجود دارد:

- انتخاب کارآمدترین دستورالعمل‌ها برای بازنمایی محاسبات مشخص شده با دستور سه آدرسه
- تصمیم‌گیری در مورد ترتیب یک محاسبه برای تولید کد کارآمدتر
- تصمیم‌گیری در مورد ثبات‌های مورد استفاده

۱۹-۲-۱ انتخاب کارآمدترین دستورالعمل‌ها

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

مثال

اگر ماشینی دستورالعمل AOS را داشته باشد که محتوای یک مکان حافظه را یک واحد افزایش می‌دهد، می‌توان برای تولید کد `t := t + 1`، به جای استفاده از دنباله دستورالعمل‌هایی شبیه

```
MOVE t, R
ADD #1, R
MOVE R, t
```

از دستورالعمل `AOS t` استفاده کرد.

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

۱۹-۲-۲ تصمیم‌گیری در مورد ترتیب محاسبات

برخی از ترتیب‌های یک محاسبه، به تعداد ثبات کمتری برای نگهداری نتایج میانی نیاز دارند. تصمیم‌گیری در مورد بهترین ترتیب بسیار مشکل است.

مثال

درکد زیر،

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

اگر به جای ترتیب فوق (t1-t2-t3-t4)، ترتیب (t2-t3-t1-t4) استفاده شود، تعداد ثبات‌های لازم برای نگهداری نتایج میانی کمتر خواهد بود.

۱۹-۲-۳ تصمیم‌گیری در مورد ثبات‌ها

تصمیم‌گیری در مورد اینکه کدام ثبات باید در محاسبه استفاده شود، مسئله‌ی دیگری است که در تولید یک کد خوب تاثیرگذار خواهد بود.

وقتی در یک ماشین، دستورالعمل‌ها از یک زوج ثبات استفاده می‌کنند که یکی از آنها هم به عنوان عملوند و هم به عنوان نتیجه به کار می‌رود، این مسئله دشوارتر می‌شود.

مسئله‌ی تخصیص ثبات‌ها (Register Allocation)

هدف از مسئله‌ی تخصیص ثبات‌ها، بازنویسی کد است، به طوری که از تعداد کمتری متغیر موقتی نسبت به تعداد ثبات‌های ماشین مقصد استفاده کند.

برای این منظور باید مقادیر موقتی را به ثبات‌ها نسبت داد، بدون اینکه رفتار برنامه تغییر پیدا کند.

مثال

تخصیص ثبات‌ها: مثال

```
a := c + d
e := a + b
f := e - 1
```

- فرض می‌کنیم متغیرهای a و e پس از استفاده می‌میرند.
- متغیر موقتی a می‌تواند پس از a + b مجدداً استفاده شود.
- همچنین متغیر موقتی e می‌تواند پس از e - 1 مجدداً استفاده شود.
- بنابراین می‌توان هر سه متغیر a، e و f را به یک ثبات r1 تخصیص داد:

```
r1 := c + d
r1 := r1 + b
r1 := r1 - 1
```

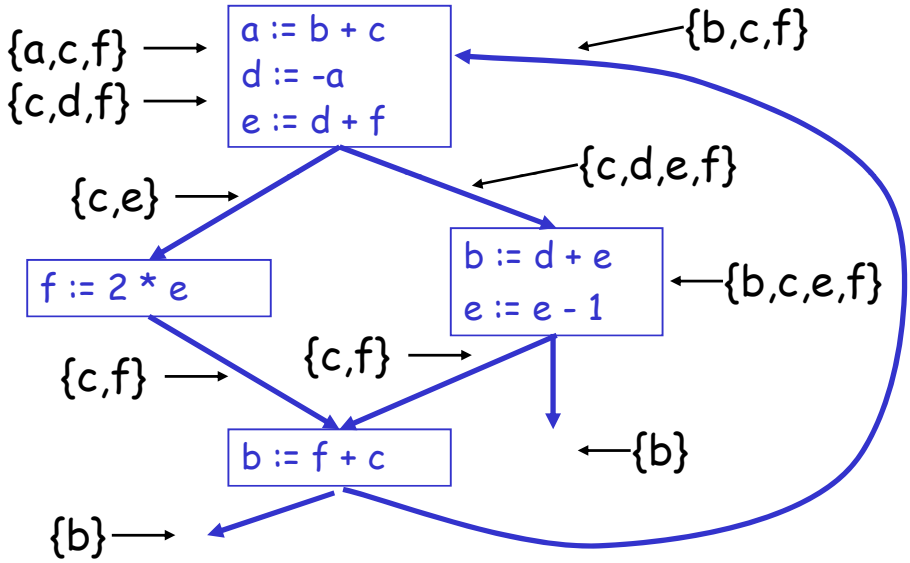
ایده‌ی پایه در تخصیص ثبات‌ها: مقدار موجود در یک متغیر موقتی مرده، در ادامه‌ی محاسبات دیگر لازم نیست، بنابراین می‌توان آن را مجدداً استفاده کرد (reuse).

قاعده‌ی پایه:

> متغیرهای موقتی t_1 و t_2 می‌توانند یک ثبات را در اشتراک داشته باشند، اگر **در هر نقطه از برنامه، حداکثر یکی از آن دو زنده باشد.**

الگوریتم تخصیص ثبات‌ها: مرحله‌ی تحلیل زنده بودن متغیرها

با رسم گراف جریان، متغیرهای زنده را در هر نقطه از برنامه تعیین می‌کنیم:



گراف دخالت ثبات‌ها

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

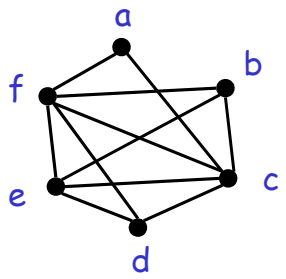
برای تشخیص این مورد، یک گراف بدون جهت می‌سازیم:

- برای هر متغیر موقتی یک گره در نظر می‌گیریم،
- یک یال بین گره‌های t_1 و t_2 رسم می‌شود، اگر این متغیرها به طور همزمان در یک نقطه از برنامه زنده باشند.

این گراف، گراف دخالت ثبات‌ها (register interference graph: RIG) نام دارد:
دو متغیر موقتی می‌توانند به یک ثبات تخصیص داده شوند، اگر هیچ یالی آن دو را به هم متصل نکرده باشد.

مثال

گراف دخالت ثبات‌ها:
برای مثال:



- b و c نمی‌توانند در یک ثبات قرار گیرند (بین آنها یال وجود دارد).
- b و d می‌توانند در یک ثبات گیرند (بین آنها یالی وجود ندارد).

خصوصیات گراف دخالت ثبات‌ها

- این گراف، اطلاعات لازم برای تشخیص انتساب‌های مجاز ثبات‌ها را به طورکامل استخراج می‌کند.
- این گراف، یک نمای سراسری (بر روی کل گراف کنترل برنامه) از نیازمندیهای ثبات را به دست می‌دهد.
- پس از ساخت این گراف، الگوریتم تخصیص ثبات مستقل از معماری ماشین مقصد خواهد بود.

تخصیص ثبات‌ها از طریق رنگ‌آمیزی گراف

مسئله‌ی رنگ‌آمیزی گراف
یک **رنگ‌آمیزی گراف**، یک انتساب از رنگ‌ها به گره‌های گراف است که گره‌های متصل شده با یک یال، همرنگ نباشند.

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

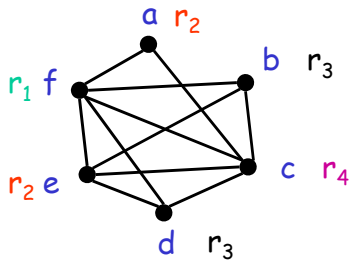
- رنگ‌ها = ثبات‌ها
- k = تعداد ثبات‌های ماشین

باید به هر گره‌ی گراف (متغیر موقتی) یک رنگ (ثبات) تخصیص دهیم.

اگر یک گراف تداخل ثبات، k-رنگ‌پذیر باشد، در این صورت یک انتساب ثبات وجود خواهد داشت که بیش از k ثبات نیاز ندارد.

مثال

گراف نمونه‌ی زیر را در نظر می‌گیریم:



- هیچ رنگ‌آمیزی‌ای با کمتر از ۴ رنگ وجود ندارد.
- چندین رنگ‌آمیزی با ۴ رنگ وجود دارد.

۳-۱۹ حافظه‌ی نهان (Cache)

- کامپایلرها برای مدیریت ثبات‌ها به خوبی عمل می‌کنند.
  - بهتر از آنچه یک برنامه‌نویس می‌تواند عمل کند.
- کامپایلرها برای مدیریت حافظه‌ی نهان، چندان خوب عمل نمی کنند.
  - این مسئله به عهده‌ی برنامه‌نویس گذاشته می‌شود.
- البته یک کامپایلر می‌تواند بهینه‌سازی‌های ساده‌ای را برای حافظه‌ی نهان انجام دهد.

مثال

حلقه‌های تودرتوی زیر را در نظر بگیرید:

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

کارایی حافظه‌ی نهان برای این برنامه پایین است، اما برنامه‌ی

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

همان کار را انجام می‌دهد ولی کارآیی حافظه‌ی نهان آن بهتر است (با توجه به اینکه از ترتیب سطری برای قرار دادن آرایه در حافظه استفاده می‌شود). کامپایلر می‌تواند چنین بهینه‌سازی‌ای را انجام دهد که به آن «تعویض حلقه‌ها» گفته می‌شود.

۴-۱۹ بهینه‌سازی کد نهایی

بر روی کد نهایی نیز می‌توان بهینه‌سازی‌هایی را اعمال کرد.

از آنجا که کد نهایی با استراتژی دستور به دستور تولید شده است، حاوی دستورالعمل‌های افزونه و ساختارهای غیربهینه خواهد بود. برای بهبود کیفیت کد مقصد، بهینه‌سازی لازم است.

تکنیک اصلی «بهینه‌سازی تکه‌ای» (Peephole optimization) نامیده می‌شود که به طور محلی کد مقصد را بهبود می‌دهد. در هر مرحله، دنباله‌ی کوتاهی از دستورالعمل‌های کد مقصد بررسی می‌شود و در صورت امکان با دنباله‌ی سریع‌تری جایگزین می‌گردد.

نمونه‌هایی از بهینه‌سازی تکه‌ای (Peephole optimization)

- حذف خواندن و نوشتن‌های اضافی حافظه

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

- حذف پرش‌های چندتایی (یک پرش به پرش دیگر)

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

- حذف کد غیرقابل دسترس
- ساده‌سازیهای جبری (`a = a + 0` `a = a * 1`)
- کاهش پیچیدگی (استفاده از دستور شیفت به جای `x = x * 2`)
- استفاده از ویژگی‌های ماشین (وجود دستور افزایش برای خانه‌های حافظه)