بررسی پیاده‌سازی یک بهینه ساز کلّی *

امین عباسپور **علیرضا رضایی**

دانشکدهٔ مهندسی کامپیوتر
دانشگاه علم و صنعت ایران

چکیده

در این مقاله به بررسی پیاده سازی یک بهینه ساز کلّی[1] (غیر محلّی) می‌پردازیم. با توجه به رشد روز افزون زبان‌های برنامه سازی و بالا رفتن سطح انتزاع در آن‌ها و نیاز همیشگی به اجرای هر چه سریع‌تر نرم‌افزارها، مقوله کامپایل و بهینه‌سازی از اهمیّت خاصّی برخوردار است. از آنجایی که امروزه با توجه به رشد سخت افزار تا حد زیادی از مشکل پیچیدگی فضای برنامه‌ها کاسته شده، اکثراً هدف از بهینه سازی، افزایش سرعت می‌باشد. در این مقاله سعی شده است تا با تاکید بر روشهای عملی پیاده سازی یک بهینه‌ساز نمونه، زمینهٔ لازم برای تلاش‌های بعدی از جهت ابزار و متد پیاده‌سازی برای خواننده ایجاد شود.

۱ معرفی

بهینه‌سازی را می‌توان به دو گروه کلی تقسیم کرد:

محلّی که در داخل بلاک‌های اولیّه[2] صورت می‌گیرد و عموماً متاثر از ماشین و مجموعهُ دستورات آن است.

کلّی که بین بلاک‌های اولیّه صورت می‌گیرد و مستقل از ماشین است.
در اینجا تاکید ما بر بهینه سازی کلّی است.

۲ مراحل کار

کار از دو بخش مجزا تشکیل شده است، که بخش اوّل ورودی قسمت دوم(بهینه ساز) را تامین می‌کند. در بخش اوّل کار ورودی به صورت کدی شبیه پاسکال به کامپایلر داده می‌شود. خروجی این بخش کد ۳ آدرسه مستقل از ماشین است که بر اساس گرامر به کار رفته در کتاب [۱] می‌باشد. در زیر مروری بر این کد داریم.

*ارائه شده به عنوان پروژهٔ درس ظراحی کامپایلر پیشرفته ، فروردین ۱۳۸۳، دانشگاه علم و صنعت ایران، تهران.
[1] Global Optimizer
[2] Basic Blocks

۱.۲ بخش اوّل

۱.۱.۲ تولید کد ۳ آدرسه

عناصر این کد عبارتند از:

۱) دستورات تعریف نوع def type x. مقدار type می تواند مقادیری مانند char و int داشته باشد.

۲) دستورات مقدار دهی مانند x:=y op z که *op* یک عمل ۲ تایی ریاضی یا منطقی است.

۳) دستورات کپی x := y

۴) دستورات پرش غیر شرطی goto L

۵) دستورات پرش شرطی if x relop y goto L، که relop عبارت مقایسه ای مانند = است.

٦) دستورات فراخوانی توابع به صورت زیر

param x_1
param x_2
...
param x_n
call p, n

در زیر نمونه ای از سورس کد اولیه و حاصل خروجی بخش اوّلیه را می بینیم

```
var
       a,b,c:int;
       d,e:char;
       g:int;
begin
       a:=(b+2)-(3+4);
end.
```

↓

```
def    int     a
def    int     b
def    int     c
def    char    d
def    char    e
def    int     g
_t0    :=      b       +       2
_t1    :=      3       +       4
a      :=      _t0     -       _t1
```

۲.۱.۲ تولید گراف جریان برنامه

همراه با تولید کد میانی، گرافی از بلاک‌های اولّیه برنامه و ارتباط بین آن‌ها رسم می‌شود. گراف‌ها به کمک ابزار dot رسم می‌شوند. برای آشنایی بیشتر با این ابزار می‌توانید به آدرس انتهای مقاله مراجعه کنید.

۳.۱.۲ تولید درخت دودویی معادل برنامه

گراف دیگری که تولید می‌شود، مربوط به درخت خلاصهُ نحوی است. در این گراف label مربوط به هر عنصر نیز نمایش داده می‌شود.

۲.۲ بخش دوّم

در بخش دوّم، کد ۳ آدرسه به عنوان ورودی داده می‌شود و بهینه سازی روی این کد انجام می‌شود. از آنجایی که بخش اوّل و دوّم به طور مستقل پیاده سازی شده‌اند و تنها ارتباط آن‌ها از طریق کد ۳ آدرسه است لازم بود که در این بخش ورودی مجدّدا پارس شده و بلاک‌های اوّلیه مجدّداً تشخیص داده شوند.

۱.۲.۲ الگوریتم بهینه سازی

در ابتدا نیاز داریم که بلاک‌های اولیه را تشخیص دهیم. پس از آن برای هر بلاک متغیّرهای def و use شده را شناسایی می‌کنیم. برای این کار در هنگام تحلیل نحوی در کنار هر عبارت ، تعریف و استفاده را نیز تشخیص می‌دهیم. مانند عبارت زیر:

```
param_cmd: tPARAM variable \
        { $$ = mk_cmd2("param", $2); use($2); }
```

پس از یافتن تعریف و استفاده‌ها، تا انتهای تشخیص بلاک‌های اولیّه منتظر می‌مانیم و سپس بر اساس شمارهٔ خط هر تعریف واستفاده تشخیص می‌دهیم که هر یک به کدام بلاک اولیّه تعلّق دارد.
یافتن بلاک‌های اولیه و تعریف و استفاده‌ها در هر بلاک زمینه لازم برای اجرای الگوریتم‌های iterative را فراهم می‌آورد.
برای شروع الگوریتم با توجّه به گراف جریان کنترل (که در آن از هر بلاک یا به یک بلاک یا دو بلاک دیگر می‌رویم) پدر هر گره را شناسایی می‌کنیم.
پس از اتمام الگوریتم، مجموعه‌های In و Out هر بلاک تشخیص داده می‌شود.

۲.۲.۲ تشخیص تعاریف بلااستفاده

اکنون که مجموعه‌های kill،gen و out،in مربوط به هر بلاک را در اختیار داریم، از بلاک اول شروع به بررسی می‌کنیم.
در صورتی که متغیّری در این بلاک تعریف شده و در داخل همان بلاک مورد استفاده قرار نگرفته، کاندیدایی برای تعریف بلااستفاده[۳] است. اکنون در تمام بلاک‌هایی که این تعریف(متغیّر—خط) در مجموعهٔ In آن‌ها قرار دارد، بررسی می‌کنیم. در صورتی این تعریف در این بلاک نیز بلااستفاده است که یا در این بلاک اصلاً استفاده نشده باشد و یا قبل از استفاده از این متغیّر دوباره در بلاک مقداردهی شده باشد.
پس از بررسی تمام بلاک‌ها، اگر در تمام آن‌ها تعریف بلااستفاده باشد، می‌توان حکم کرد که تعریف در بلاک اوّل نیز بلااستفاده است.
به همین ترتیب سایر بلاک‌ها و تعریف‌های آن‌ها را مورد بررسی قرار می‌دهیم.
در زیر شبه‌کد فرایند را می‌بینیم.

```
1 for all b,d ∈ blocks, gen[b]−{localy used defs}
2   useful ← false
3   for all b' ∈ blocks − b
4       if d ∉ in[b'] continue
5       if d.variable used in b' before def
6           useful ← true
7           break
8   if usefull ≠ true
9       remove d from b
```

۳ راهنمای راه‌اندازی

پیاده‌سازی مجموعه فوق، تحت سیستم عامل Linux صورت گرفته است. برای تحلیل لغوی از flex[۴] و برای تحلیل نحوی از Bison[۵] استفاده شده است. مراحل کامپایل و اجرای برنامه‌های بخش اوّل و دوّم به شرح زیر است:

۱.۳ راه‌اندازی کامپایلر ساده پاسکال

برای کامپایل داریم

```
$ flex mesi.l <- generate lex.yy.c
$ yacc mesi.y <- generate y.tab.c
$ gcc -o mesi tree.c symbol.c y.tab.c -lfl
```

و برای اجرا

```
$ ./mesi < samples/sums.msi
$ dot -Tps tree.dot -o tree.ps
$ gv tree.ps
```

[۳] Redundant Definition
[۴] Free Lex
[۵] GNU Yacc Implementation

۲.۳ راه‌اندازی و اجرای بهینه‌ساز

برای کامپایل داریم

```
$ flex gopt.l <- generate lex.yy.c
$ yacc syntax.y <- generate y.tab.c
$ gcc -o gopt gopt.c y.tab.c tools.c \
        reachable.c available.c -lfl
```

و برای اجرا

```
$ ./gopt < code.3a
$ dot -Tps basic-blocks.dot \
      -o basic-blocks.ps
$ gv basic-blocks.ps
```

۳.۳ منابع

برای تهیهٔ سورس کدهای مثال و سایر نرم‌افزارهای مورد نیاز می‌توانید به آدرس زیر مراجعه کنید.

http://www.rasana.net/~amin/compiler.html

مراجع

[1] A. V. Aho, et al *Compilers Priciples, Techniques, and Tools*, Addison-Wesley, 1986.