به نام خدا

تمرین کامپیوتری دوم درس طراحی کامپایلر

بهار ۱۴۰۴

فهرست مطالب

مقدمه

در فاز اول پروژه، تحلیل‌گر لغوی و نحوی (lexer and parser) زبان CPY را پیاده‌سازی کردید. در این فاز از پروژه از شما انتظار می‌رود یک name analyzer برای زبان طراحی کنید. با پیمایش AST، اطلاعات مربوط به توابع، pattern ها و متغیرها را در جدول علائم (Symbol Table) ذخیره کنید. برای این کار لازم است تا متد visit از Visitor interface در NameAnalyzer visitor، برای همه node ها override شود و بررسی‌های لازم برای هر node انجام شود.

بعد از آن باید مرحله‌ی بهینه‌سازی کد را انجام دهید. هدف این مرحله، شناسایی و حذف بخش‌هایی از کد است که بلااستفاده، مرده یا غیر ضروری هستند تا برنامه‌ای کاراتر، خواناتر و ساده‌تر تولید شود. بهینه‌سازی کد نه تنها باعث کاهش حجم برنامه و مصرف کمتر منابع می‌شود، بلکه تحلیل‌های بعدی روی کد (مانند بهینه‌سازی‌های زمان اجرا یا تولید کد ماشین) را نیز بسیار مؤثرتر و ساده‌تر می‌کند. همچنین کد بهینه شده، درک و نگهداری را برای توسعه‌دهندگان بعدی آسان‌تر می‌کند. می‌توانید بخشی از بهینه‌سازی را (یا در صورت صلاحدید همه آن را) در NameAnalyzer پیاده‌سازی کرده و بقیه آن را در کلاس visitor دیگری انجام دهید. توضیحات بیشتر در کارگاه داده خواهد شد.

فرضیات در تست‌ها

- در هنگام ویزیت کردن function call ها، هر تابعی با اسم و تعداد آرگومان‌های آن مشخص خواهد شد و نیازی به انجام type checking برای پیدا کردن توابع نیست.
- در خروجی نمونه sample ها در صورت پروژه، خطوط بعد از حذف شدن خط‌های اضافی به سمت بالا شیفت می‌شوند. در صورتی که در خروجی نهایی گیت‌هاب کلسروم این اتفاق نخواهد افتاد و صرفا برای نمایش بهتر به این شکل نمایش داده شده است.
- تست‌ها یا دارای خطاهای Name Analysis خواهند بود که در این صورت فقط خطاها را چاپ کنید، در غیر این صورت باید بهینه‌سازی‌ها را انجام داده و مانند فاز اول node ها را پرینت بگیرید.

خطاهای Name Analysis

1- استفاده از متغیر تعریف نشده

برای استفاده از هر متغیر، لازم است ابتدا آن متغیر در محدوده (Scope) مربوطه تعریف شده باشد. اگر متغیری قبل از استفاده تعریف نشده باشد، خطای زیر تولید می‌شود:

Line:<LineNumber>-> <Variable> not declared

```
1  void process() {
2      int a;
3      if(1){
4          if(1){
5              printf(a);
6              printf(b);
7              int b;
8          }
9          printf(b);
10     }
11 }
12 int main(){
13     int b;
14     printf(b + c);
15 }
```

```
Line:6-> b not declared
Line:9-> b not declared
Line:14-> c not declared
```

2- استفاده از تابع تعریف نشده

این خطا زمانی رخ می‌دهد که برنامه تلاش کند تابعی را فراخوانی کند که تعریف نشده است (توجه کنید که فانکشن‌ها یا توابعی که به طور معمول با استفاده از include اضافه می‌شوند را نباید در نظر بگیرید، مانند scanf, printf). در این صورت خطای زیر تولید می‌شود:

Line:<LineNumber>-> <Function> not declared

```c
void foo() {
    printf("heb");
}
void foo(int x) {
    printf("heb");
}
int foo(int x, char y) {
    printf("heb");
}
void moo(char x, char y) {
    printf("%s%s", x, y);
    moo(20);
}
void shoo() {
    int x;
    foo(foo(x), x);
    moo(10);
}
int main(){
    foo(3);
    foo(3, 5); // amdan ro in error nazashtam ke dar nazar begirid ba esm va tedad argoman bayad tashkhis bedid
    moo("Hello", "World");
}
```

```
Line:12-> moo not declared
Line:17-> moo not declared
```

بهینه‌سازی‌ها

توجه کنید که تمام این بهینه‌سازی‌ها به صورت ترکیبی هم می‌توانند در تست‌ها بیایند (این مورد رو در نظر بگیرید که sample ها منطقی هستند و حالت‌های بسیار پیچیده داده نخواهد شد). همچنین توجه کنید که می‌توانید قوانین بهینه کردن را قبل یا بعد از تحلیل دسترسی از main پیاده‌سازی کنید ولی در نهایت باید خروجی شما بهینه‌ترین خروجی باشد.

توجه: sample ها به زبان C نوشته شده‌اند ولی برای ورودی دادن به برنامه‌های خود، آن را به حالت CPY در بیاورید (برای دانشجویان مهندسی کامپیوتر).

1- متغیرها و پارامترهای بلااستفاده (فقط دانشجویان مهندسی کامپیوتر)

در بسیاری از برنامه‌های C، متغیرهایی در بدنه‌ی تابع تعریف می‌شوند یا پارامترهایی به عنوان ورودی به تابع داده می‌شوند که هرگز در محاسبات یا تصمیم‌گیری‌های برنامه استفاده نمی‌شوند. در فرآیند بهینه‌سازی، این موارد شناسایی شده و حذف می‌شوند تا کد سبک‌تر، خواناتر گردد. اگر پارامترهای یک تابع تغییر کند (مثلاً پارامتری حذف شود)، لازم است تمامی function call هایی که به این تابع اشاره دارند نیز متناسب با امضای جدید تابع (function signature) اصلاح شوند.

در این فرآیند بهینه‌سازی فرض می‌کنیم که تغییر تعداد پارامترها باعث تداخل با توابع دیگر نخواهد شد؛ یعنی پس از حذف آرگومان‌ها، همچنان امضای تابع با توابع دیگر تداخلی پیدا نمی‌کند. مثال:

```c
void process(int x, int unused_param) {
    int a = 5;
    int b = 10;
    int c = 15;
    printf("%d\n", x);
}
```

```c
void process(int x) {
    printf("%d\n", x);
}
```

۲- دستورات بعد از return

هر دستوری که پس از اجرای return نوشته شده باشد و در یک scope باشند، اجرا نمی‌شود و مرده محسوب می‌شود. در بهینه‌سازی، این دستورات باید بدون تغییر رفتار برنامه، حذف گردند. مثال:

```
int get_value() {
    return 42;
    int m = 3;
}
```

```
int get_value() {
    return 42;
}
```

۳- دستورات بی‌تأثیر (بدون اثر جانبی)

دستورات یا عباراتی که اجرا می‌شوند اما هم بر خروجی برنامه اثری ندارند و هم اینکه وضعیت متغیرها را تغییر نمی‌دهند، و هیچ عملی جانبی (Side Effect) ایجاد نمی‌کنند، باید حذف شوند. مثال:

```
void calculate() {
    5 + 7; // has no effect
    printf("Running...\n");
}
```

```
void calculate() {
    printf("Running...\n");
}
```

~~۴- دستورات Multiple Assignments~~

~~در صورتی که یک متغیر چندین بار مقداردهی شود و در طول این مسیر هیچ استفاده‌ای از مقدارهای قبلی نشود، می‌توان مقداردهی‌های قبلی را حذف کرد و فقط آخرین مقدار را نگه داشت. اما اگر بین مقداردهی‌ها از متغیر استفاده شده باشد (مثلا آن متغیر به عنوان آرگومان تابعی به کار رفته باشد و یا در محاسبه‌ای شرکت کرده باشد)، نباید مقداردهی قبلی را حذف کرد. مثال:~~

```c
int foo(int x) {
    return x;
}

void update() {
    int x = 5;
    foo(x);
    x = 10;
    x = 15;
    printf("%d\n", x);
}
```

```c
int foo(int x) {
    return x;
}

void update() {
    int x = 5;
    foo(x);
    x = 15;
    printf("%d\n", x);
}
```

۵- جایگزینی typedef ها و مقادیر ثابت (فقط مهندسی کامپیوتر)

در بسیاری از برنامه‌های C، برای ساده کردن نوشتار یا مدیریت مقادیر ثابت از typedef و const استفاده می‌شود. در فرآیند بهینه‌سازی، این نام‌های انتزاعی در صورت نیاز می‌توانند با نوع داده‌ی اصلی یا مقدار واقعی جایگزین شوند تا ساختار کد واضح‌تر شود یا تحلیل ایستا (Static Analysis) آسان‌تر انجام شود. مثال:

```c
typedef int myint;

#define SIZE 10

void example() {
    myint arr[SIZE];
    for (int i = 0; i < 10; i++) {
        arr[i] = i;
    }
}
```

```c
void example() {
    int arr[10];
    for (int i = 0; i < 10; i++) {
        arr[i] = i;
    }
}
```

تحلیل دسترسی از main

در فرآیند بهینه‌سازی کد، یکی از مهم‌ترین تکنیک‌ها حذف بخش‌هایی از برنامه است که هرگز در جریان واقعی اجرای برنامه مورد استفاده قرار نمی‌گیرند. در این پروژه، مبنای بهینه‌سازی بر اساس تحلیل دسترسی (Reachability) از تابع main تعریف شده است. به این صورت که فقط بخش‌هایی از برنامه (از جمله توابع) که با شروع پیمایش از main و از طریق زنجیره‌ای از فراخوانی‌های مستقیم یا غیرمستقیم قابل دسترسی باشند، در کد باقی خواهند ماند. در مقابل، هر تابع یا قطعه کدی که هیچ مسیری از main به آن وجود نداشته باشد، به عنوان کد مرده در نظر گرفته شده و باید حذف شود. مثال:

```c
void startEngine() {
    printf("Engine started.\n");
}

void checkOil() {
    printf("Oil level checked.\n");
}

void washCar() {
    printf("Car washed.\n");
}

int main() {
    startEngine();
}
```

```c
void startEngine() {
    printf("Engine started.\n");
}

int main() {
    startEngine();
}
```

در این پروژه لازم است تحلیل دسترسی با دقت و بر اساس پیمایش واقعی از main انجام شود. هدف این است که هیچ تابعی که در مسیر اجرای واقعی نقشی ندارد، در خروجی نهایی باقی نماند. نیازی به تحلیل داده‌ای یا پیش‌بینی رفتار زمان اجرای برنامه نیست و فقط مسیرهای موجود در کد مبنای تصمیم‌گیری قرار می‌گیرند. بنابراین نسخه‌ی بهینه‌ی نهایی تنها شامل بخش‌هایی خواهد بود که برای عملکرد واقعی برنامه ضروری هستند.

توجه کنید که حالات ترکیبی از این قسمت با قسمت بهینه‌سازی‌ها در Sample ها خواهد آمد.

موفق باشید.