اصول طراحی کامپایلرها

حسین کارشناس

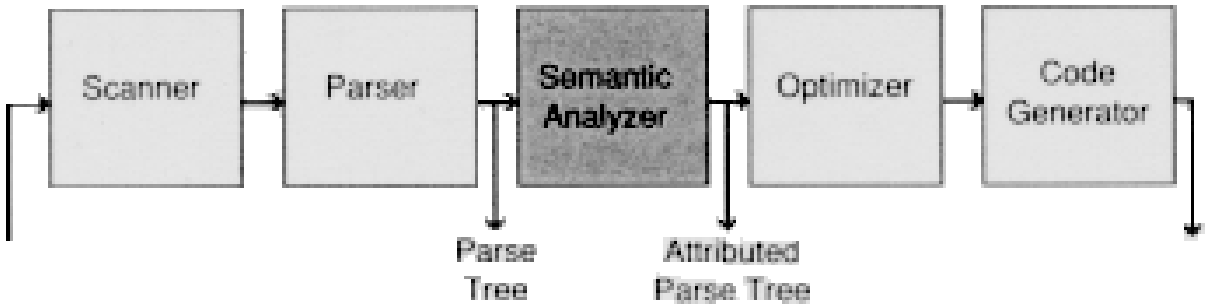
دانشکده مهندسی کامپیوتر

ترم اول ۹۸ – ۹۷

تحلیل معنایی

تحلیل معنایی

- هدف: بررسی نهایی صحت برنامه ورودی
  - شناسایی خطاهای واژه‌ای و دستوری در مراحل تحلیل واژه‌ای و دستوری
  - هر نوع خطاهای دیگری در برنامه ورودی باید در این مرحله شناسایی شود
    - خطاهایی که در مراحل قبلی کامپایل قابل شناسایی نیستند
- بعد از این مرحله برنامه ورودی به زبان دیگری ترجمه می‌شود



- امکان انجام همزمان مراحل باقیمانده از بخش پیشین کامپایلر

تحلیل معنایی

- محدودیت زبان‌های مستقل از متن
  - عدم امکان توصیف برخی از ساختارها با گرامرهای مستقل از متن

- مثال: بررسی اینکه آیا شناسه‌ها قبل از استفاده تعریف شده‌اند
  - مورد نیاز در اکثر زبان‌های مطرح برنامه‌نویسی مانند C و Java
  - مشابه زبان $\{wcw \mid w \text{ is in } (\mathbf{a}|\mathbf{b})^*\}$
  - در نظر گرفتن نماد یکسان برای تمام شناسه‌ها در تحلیل دستوری

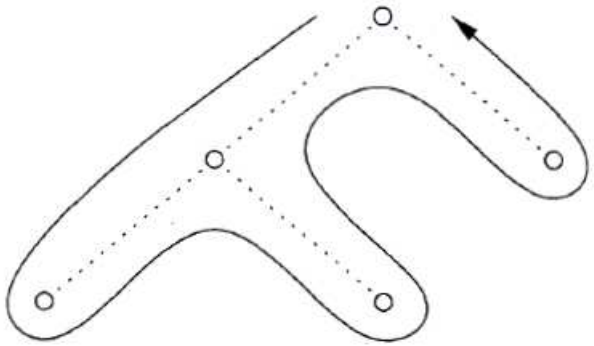
- مثال: تطابق تعداد پارامترهای توابع در هنگام تعریف و فراخوانی
  - مشابه زبان $\{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$

تحلیل معنایی

- برخی از بررسی‌های صورت گرفته در این مرحله
  - تمام شناسه‌ها باید قبل از استفاده تعریف شده باشند
  - بررسی نوع (type checking)
  - قرارگیری دستورهای خاص مانند break و continue در ساختارهای بخصوص
  - تطابق تعداد تعریف شناسه‌ها با محدودیت‌های زبان
  - تطابق تعداد پارامترهای توابع در هنگام تعریف و فراخوانی
  - درستی رابطه‌های وراثت (inheritance)
  - ...
- نوع بررسی‌های صورت گرفته در زبان‌های مختلف فرق دارد

تحلیل معنایی

- راهکار کلی: تحلیل معنایی با پویش AST یا درخت تجزیه
  - پردازش گره‌های درخت بصورت بازگشتی
    - پیش‌پردازش گره پیش از پردازش گره‌های فرزند
    - پردازش هر یک از گره‌های فرزند
    - پس‌پردازش گره پس از پردازش گره‌های فرزند



- راهکار دیگر: امکان انجام تحلیل معنایی (و نیز تولید کد میانی) در حین تجزیه
  - **ترجمه دستور گرا (syntax-directed translation)**
- لزومی به ساخت عینی درخت تجزیه یا AST نیست
- امکان در نظر گرفتن گراف دستور انتزاعی بجای AST برای فشرده‌سازی

حوزه‌ها (scopes)

- حوزه یک شناسه
  - محدوده‌ای از برنامه که آن شناسه قابل دسترسی باشد
  - معمولاً بر اساس محل تعریف (declaration) شناسه تعیین می‌شود
  - امکان وجود بیش از یک تعریف برای یک شناسه خاص (نام یکسان)
    - وجود حوزه‌های مختلف غیرهمپوشان (non-overlapping) برای یک شناسه

```
...
int i;                  /* global i        */
...
void f(···) {
      int i;            /* local i         */
      ...
      i = 3;            /* use of local i  */
      ...
}
...
      x = i + 1;        /* use of global i */
```

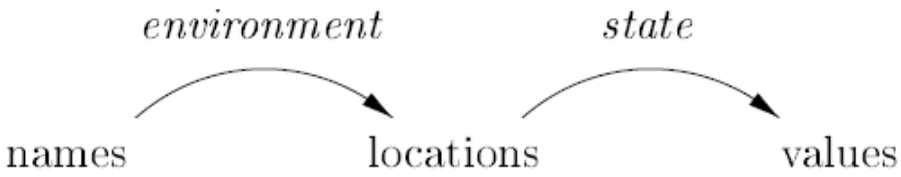
حوزه‌ها

- انواع حوزه‌ها
  - حوزه ایستا (static)
    - حوزه شناسه از کد برنامه بدست می‌آید
  - حوزه پویا (dynamic)
    - حوزه شناسه با توجه به رفتار برنامه در زمان اجرا تعیین می‌شود
    - کاربرد شناسه به یکی از تعاریف مختلف آن اشاره می‌کند
    - مثال:

```
#define a (x+1)
int x = 2;
void b() { int x = 1; printf("%d\n", a); }
void c() { printf("%d\n", a); }
void main() { b(); c(); }
```

حوزه‌ها

- محیط‌ها (environments) و حالت‌ها (states)



- محیط: یک نگاشت از شناسه به مکان ذخیره‌سازی
  - تغییر محیط (مکان ذخیره‌سازی) یک شناسه با توجه به حوزه‌های مختلف آن
- حالت: یک نگاشت از مکان ذخیره‌سازی به مقادیر ممکن
  - اختصاص یک مقدار به یک شناسه
- محیط‌ها و حالت‌ها می‌توانند ایستا باشند یا بصورت پویا تغییر کنند
  - اگر حوزه پویا باشد محیط یک شناسه در زمان اجرای برنامه تغییر می‌کند
  - اگر حوزه ایستا باشد محیط یک شناسه در زمان کامپایل مشخص می‌شود

حوزه‌ها

- تعیین حوزه‌های ایستا با توجه به ساختار قطعه‌ای (block) برنامه
  - ساختار قطعه‌ای به معنی قرارگیری قطعه‌ها به صورت تو در تو (nested) است
  - تعریف D به قطعه B تعلق دارد اگر B داخلی‌ترین قطعه حاوی D باشد
  - پیوند شناسه با نزدیکترین تعریف
    - حوزه شناسه x تمام قطعه‌ای است که در آن قرار دارد مگر قطعه‌هایی درونی که شناسه x را دوباره تعریف کنند
    - اگر $B_1, \ldots, B_k$ قطعه‌هایی تو در تو باشند (B_1 بیرونی‌ترین قطعه) که یک کاربرد از شناسه x را در بر می‌گیرند، و B_i داخلی‌ترین قطعه‌ای باشد که تعریف x به آن متعلق است، آنگاه کاربرد متغیر x در حوزه تعریفی است که متعلق به B_i است
- حوزه‌های ایستا لزوماً همیشه به این صورت تعیین نمی‌شوند
  - مثال: اسامی کلاس‌ها، فیلدهای یک کلاس

حوزه‌ها

- مثال از تعیین حوزه‌های ایستای با استفاده از ساختار قطعه‌ای برنامه

```
main() {
    int a = 1;                                   B1
    int b = 1;
    {
        int b = 2;                               B2
        {
            int a = 3;                           B3
            cout << a << b;
        }
        {
            int b = 4;                           B4
            cout << a << b;
        }
        cout << a << b;
    }
    cout << a << b;
}
```

DECLARATION	SCOPE
int a = 1;	$B_1 - B_3$
int b = 1;	$B_1 - B_2$
int b = 2;	$B_2 - B_4$
int a = 3;	B_3
int b = 4;	B_4

حوزه‌ها

- سوال: مقادیر اختصاص یافته به متغیرهای w، x، y و z پس از اجرای قطعه کد زیر چیست؟

```
int w, x, y, z;
int i = 4; int j = 5;
{   int j = 7;
    i = 6;
    w = i + j;
}
x = i + j;
{   int i = 8;
    y = i + j;
}
z = i + j;
```

حوزه‌ها

- سوال: حوزه هر یک از تعاریف زیر کدام است؟

```
{    int w, x, y, z;        /* Block B1 */
     {    int x, z;         /* Block B2 */
          {    int w, x;    /* Block B3 */ }
     }
     {    int w, x;         /* Block B4 */
          {    int y, z;    /* Block B5 */ }
     }
}
```

حوزه‌ها

- نوع‌های جدید تعریف شده در برنامه
  - مفهوم کلاس (class) در زبان‌های شیءگرا
  - استفاده از کنترل آشکار دسترسی (explicit access control)
    - حوزه شناسه‌های عضو یک کلاس تمام کلاس و زیرکلاس‌ها هستند
      - مگر آنکه در زیرکلاس‌ها جداگانه تعریف شده باشند
    - استفاده از کلیدواژه‌های مخصوص برای کنترل دسترسی (encapsulation)
      - protected ،private ،public
- توابع
  - امکان بازتعریف توابع با امضاء متفاوت (بارگذاری مضاعف – overloading)
  - تأثیر روش‌های ارسال پارامترها در حوزه متغیرهای یک تابع
    - فراخوانی با مقدار (call by value) یا با ارجاع (call by reference)

حوزه‌ها در تحلیل معنایی

- نیاز به تعیین نشانه‌های در دسترس در زمان پردازش هر گره از AST
  - در گره‌های مختلف حوزه‌های متفاوتی وجود دارند
  - در برخی از گره‌ها نشانه‌های جدیدی تعریف می‌شوند
- راهکار: جدول‌های نشانه‌های (ST) جداگانه برای حوزه‌های متفاوت
  - در هر گره اطلاعات آخرین جدول نشانه مورد استفاده قرار می‌گیرد
  - در برخی گره‌ها اطلاعات جدیدی به آخرین جدول افزوده می‌شود
  - در برخی گره‌ها حوزه جدیدی ایجاد می‌شود
  - در برخی گره‌ها باید به حوزه قبلی برگردیم
- امکان نیاز به بیش از یک عبور (pass) برای تعیین نشانه در دسترس

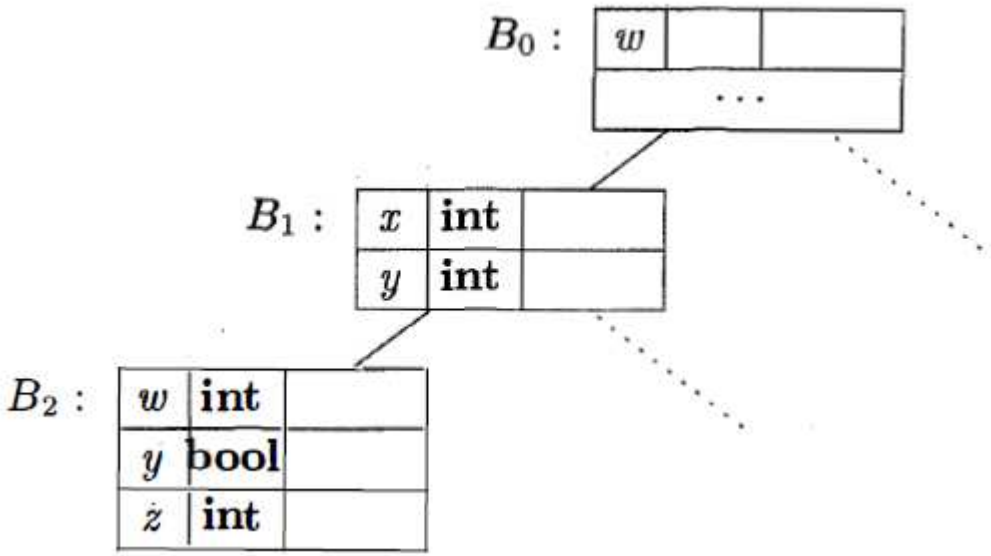
حوزه‌ها در تحلیل معنایی

- برخی روش‌های پیاده‌سازی جدول‌های جداگانه
  - استفاده از پشته
    - اضافه کردن جدول نشانه‌های جدید به بالای پشته با ورود به حوزه جدید
    - با تعریف نشانه‌های جدید اطلاعات آنها به آخرین جدول اضافه می‌شود
    - جستجوی جدول‌های نشانه‌ها از بالا به پایین پشته برای یافتن حوزه یک نشانه
    - حذف آخرین جدول نشانه‌ها از بالای پشته با پایان یک حوزه
  - استفاده از درخت
    - جدول نشانه‌های جدید به عنوان فرزند جدول نشانه‌های قبلی در نظر گرفته می‌شود
    - جستجوی جدول‌های نشانه‌ها از جدول فعلی به سمت ریشه برای یافتن حوزه یک نشانه
    - پشتیبانی ساده‌تر از حوزه‌های تو در تو

حوزه‌ها در تحلیل معنایی

- مثال: جدول‌های نشانه‌های متفاوت برای قطعه‌های تو در تو

```
1)  {   int x1; int y1;
2)      {   int w2; bool y2; int z2;
3)          ··· w2 ···; ··· x1 ···; ··· y2 ···; ··· z2 ···;
4)      }
5)      ··· w0 ···; ··· x1 ···; ··· y1 ···;
6)  }
```



بررسی نوع (Type Checking)

بررسی نوع

- مفهوم نوع
  - مجموعه‌ای از مقادیر و عملگرهای تعریف شده بر روی آنها
    - کلاس نمونه‌ای جدید از مفهوم نوع

- ضرورت وجود نوع
  - برای هر نوع از مقادیر عملیات خاصی تعریف شده است
  - امکان نمایش مشابه مقادیری که ماهیت متفاوتی دارند
    - مثلاً در زبان اسمبلی امکان اعمال عملیات روی مقادیر ماهیتاً متفاوت وجود دارد

```
add $r1, $r2, $r3
```

    - ممکن است این عملیات مفهومی نداشته باشد

بررسی نوع

- هدف بررسی نوع اطمینان از بکارگیری عملیات سازگار با نوع‌ها
  - تضمین تفسیر مورد نظر (صحیح) برای مقادیر
  - بکارگیری یک *سامانه نوع* برای هر زبان

- سامانه نوع (type system)
  - تعیین می‌کند که چه عملیاتی برای هر نوع معتبر است
  - توسط مجموعه‌ای از قوانین برای بررسی و استنتاج نوع بیان می‌شود
    - روش رسمی توصیف معنایی زبان‌های برنامه‌نویسی
  - الگوهایی برای تعیین نوع عبارت‌ها و ساختارهای مختلف برنامه

بررسی نوع

- استنتاج نوع (type inference)
  - تعیین نوع یک عبارت یا ساختار از برنامه ورودی
  - بر اساس نوع مؤلفه‌های تشکیل دهنده یا نحوه کاربرد تعیین می‌شود
- زبان با نوع قوی (strongly typed)
  - برنامه ورودی تأیید شده فاقد هرگونه خطای نوع در زمان اجرا باشد
- سامانه نوع بی‌عیب (sound)
  - تمام قوانین صحیح و منطبق با ویژگی‌های زبان برنامه‌نویسی باشند

بررسی نوع

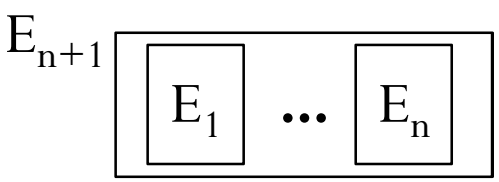
- دسته‌بندی زبان‌ها بر اساس سامانه نوع مورد استفاده
  - با نوع ایستا (statically typed)
    - بررسی نوع بخشی از فرآیند کامپایل (تصمیم‌گیری در زمان کامپایل)
      - زبان‌هایی مانند C/C++ و Java
  - با نوع پویا (dynamically typed)
    - بررسی نوع بخشی از اجرای برنامه (تصمیم‌گیری در زمان اجرا)
      - زبان‌هایی مانند Lisp، Perl، Python
  - ترکیبی
  - بدون نوع (untyped)
    - زبان اسمبلی

بررسی نوع

- زبان با نوع ایستا یا پویا؟
  - زبان‌های با نوع ایستا
    - امکان شناسایی بسیاری از خطاها در زمان کامپایل
    - جلوگیری از سربار محاسباتی برای بررسی نوع در زمان اجرای برنامه
    - محدودیت‌های بیشتر در نوشتن برنامه‌ها
      - در نظر گرفتن یک راهکار فرار: امکان تبدیل نوع (type cast)
  - زبان‌های با نوع پویا
    - امکان پیاده‌سازی سریع برنامه‌های اولیه (prototypes)
      - درگیر نشدن در جزئیات پیاده‌سازی مانند نوع مناسب برای متغیرها
    - افزایش زمان اجرای برنامه به دلیل نیاز به بررسی نوع
      - افزودن تدریجی ویژگی‌های ایستا به زبان با فراگیرتر شدن آن

بررسی نوع

- قوانین مورد استفاده در سامانه نوع
  - شکل کلی:



If E_1 has type T_1 and . . . and E_n has type T_n, Then E_{n+1} has type T_{n+1}

  - نمایش ساده‌تر: $E_1 : T_1 \wedge \ldots \wedge E_n : T_n \Rightarrow E_{n+1} : T_{n+1}$
  - مثال: $(e_1 : \text{Int} \wedge e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}$
  - مثال‌های بیشتر

i is an integer literal ⇒ i : int

e_1 : int ∧ e_2 : int ⇒ $e_1 < e_2$: bool

false : bool

new T : T

e_1 : bool ∧ e_2 : T_1 ⇒ while e_1 do e_2 : T_2

بررسی نوع

- محیط‌ها (جدول نشانه‌ها) در قوانین نوع
  - بستر مورد استفاده در بررسی نوع توسط قوانین نوع را مشخص می‌کند
    - مثال: نوع در نظر گرفته شده برای نشانه‌های مورد استفاده در هر عبارت
    - مثال: کلاس فعلی که عبارت در آن نوشته شده است
  - با تعریف نشانه‌های جدید، اطلاعات مرتبط به محیط افزوده می‌شود
    - با خروج از حوزه یک تعریف، اطلاعات محیط بروز می‌شود
  - مثال: محیط مورد استفاده در یک قانون نوع

$$\frac{O \vdash e_0 : T_0 \qquad O[T_0/x] \vdash e_1 : T_1}{O \vdash \{T_0\ x = e_0;\ \text{return}\ e_1(x);\} : T_1}$$

بررسی نوع

- بکارگیری زیرنوع‌ها (subtypes)
  - برای پشتیبانی از وراثت در شیءگرایی
  - امکان بکارگیری قوانین نوع کلی‌تر و استفاده از زیرنوع بجای یک نوع
  - مفهوم رابطه زیرنوع روی نوع‌ها

$$T_1 \leq T_1$$
$$T_2 \leq T_1 \text{ if } T_2 \text{ inherits from } T_1$$
$$T_3 \leq T_2 \text{ if } T_2 \leq T_1 \text{ and } T_3 \leq T_1$$

- مثال

$$\frac{O \vdash e_0 : T_0 \quad O[T/x] \vdash e_1 : T_1 \quad T_0 \leq T}{O \vdash \{T\ x = e_0;\ \text{return } e_1(x);\} : T_1}$$

بررسی نوع

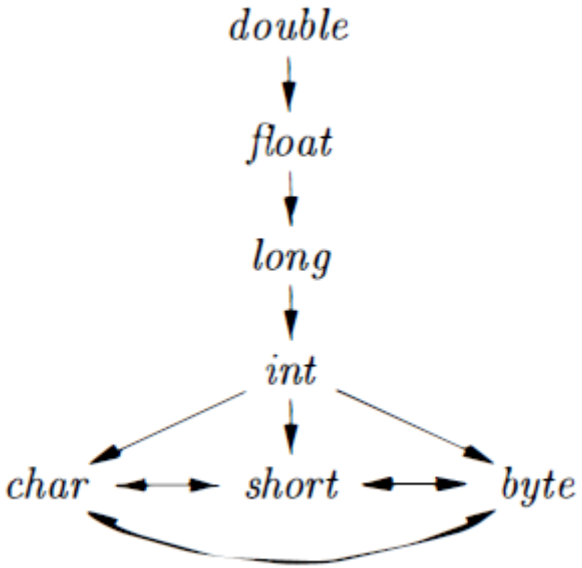
- بکارگیری زیرنوع‌ها (ادامه)
  - تعیین پایین‌ترین ابرنوع (supertype) مشترک میان دو نوع
    - بر اساس سلسله‌مراتب نوع‌ها
    - مثال: نوع دستور شرطی if-else

$$\frac{O \vdash e_0 : \mathbf{bool} \quad O \vdash e_1 : T_1 \quad O \vdash e_2 : T_2}{O \vdash \{\ \mathbf{if}\ (e_0)\ e_1;\ \mathbf{else}\ e_2;\ \} : \mathbf{lub}(T_1, T_2)}$$

  - $\mathrm{lub}(T_1, T_2)$ پایین‌ترین ابرنوع مشترک در درخت سلسله‌مراتب نوع را نشان می‌دهد
    - نوع دستور switch-case؟

بررسی نوع

- بکارگیری تبدیل نوع (type conversion)
  - در کنار رابطه زیرنوع باعث افزایش شمولیت قوانین نوع می‌شود
  - تبدیلات عریض کننده (widening) و تنگ کننده (narrowing)
    - در تبدیلات تنگ کننده ممکن است اطلاعات از دست برود
  - تبدیل نوع ضمنی (implicit) و آشکار (explicit)
    - توسط خود کامپایلر (اجبار – coercion)
    - توسط برنامه‌نویس (type casts)



- مثال: تبدیل تنگ کننده نوع در زبان Java

بررسی نوع

- نوع در عبارات حاوی فراخوانی توابع (dispatch)

$$e_0.f(e_1, \dots, e_n): ?$$

- نیاز به نوع، تعداد و ترتیب پارامترهای ورودی (signature) تابع
  - در هنگام تعریف تابع این اطلاعات در جدول نشانه‌ها نگهداری می‌شود

$$f(x_1:T_1,\dots,x_n:T_n): T_{n+1}$$

- برای متدهای کلاس آدرس توابع در جدول توابع (dispatch) نگهداری می‌شود
- پشتیبانی از توابع با بارگذاری مضاعف (overloaded)

بررسی نوع

- مهفوم بررسی نوع
  - بررسی تطابق برنامه ورودی با قوانین سامانه نوع
- امکان نیاز به استنتاج نوع
  - اثبات (proof) نوع یک مؤلفه از برنامه بر اساس قوانین نوع
  - استفاده از AST برای پیشبرد استنتاج با قوانین نوع
    - تشکیل، بهنگام‌سازی و انتقال محیط نوع‌ها در پویش بالا به پایین درخت
    - محاسبه و بررسی نوع عبارت‌ها در بازگشت از برگ‌ها به سمت ریشه
  - عملیات مکمل بررسی نوع

بررسی نوع

- پیاده‌سازی بررسی نوع
  - هر قانون در یک گره از AST بکار گرفته می‌شود
    - فرزندان یک گره دارای نوع‌هایی منطبق با نوع فرض‌های قانون بکارگرفته شده
    - خود گره دارای نوعی منطبق با نوع نتیجه قانون بکار گرفته شده
  - مثال

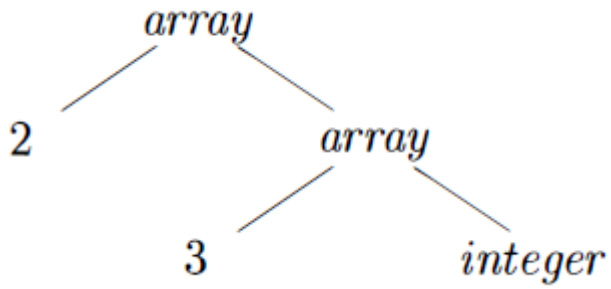
$$\frac{O \vdash e_1 : \text{Int} \quad O \vdash e_2 : \text{Int}}{O \vdash e_1 + e_2 : \text{Int}}$$

```
TypeCheck(Environment, e1 + e2) = {
    T1 = TypeCheck(Environment, e1);
    T2 = TypeCheck(Environment, e2);
    Check T1 == T2 == Int;
    return Int; }
```

بررسی نوع

- عبارت‌های نوع (type expressions)
  - نشان دهنده ساختار نوع‌های مورد استفاده در بررسی نوع
  - مثال: عبارت نوع برای int[2][3]

`int [2] [3]` ⟹ $array(2, array(3, integer))$



- دو مؤلفه اصلی در عبارت‌های نوع
  - انواع پایه (basic types) مانند: bool، char، integer، float، void
  - سازنده‌های نوع (type constructors): عملگرهای قابل اعمال به عبارت‌های نوع

بررسی نوع

- برخی از انواع سازنده‌های نوع
  - آرایه: array(number, T)
    - لیستی از عناصر هم‌نوع: list(T)
  - رکوردی از فیلدها: record($T_1, T_2, \ldots$)
  - ضرب دکارتی: $T_1 \times T_2$
  - تابع: $T_1 \rightarrow T_2$
- امکان نام‌گذاری عبارت‌های نوع با نام‌های نوع (type names)
  - مثال: برای نشان دادن عبارت نوع مربوط به یک کلاس با یک نام نوع
- امکان استفاده از متغیرهای نوع (type variables)
  - مقدار آن برابر با یک عبارت نوع است

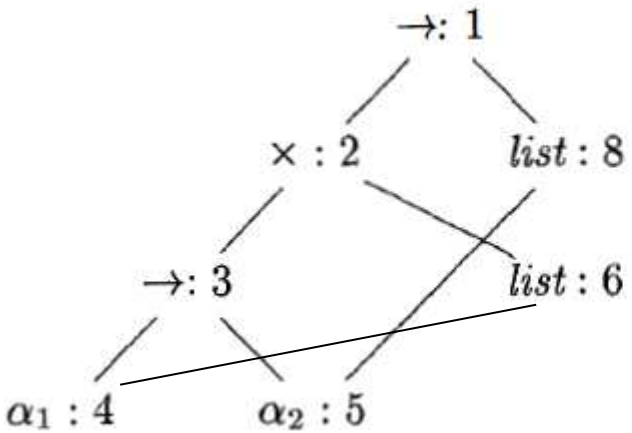
بررسی نوع

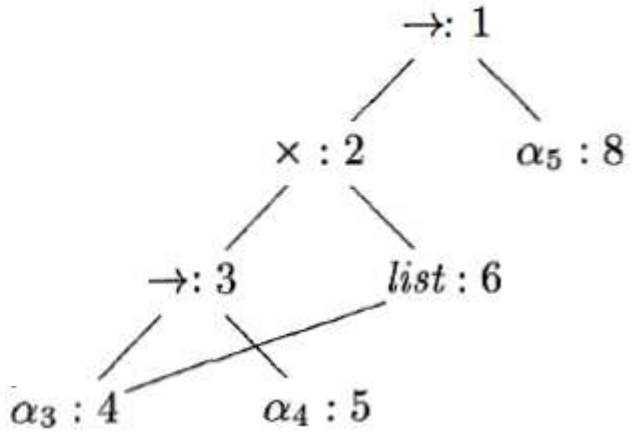
- برابری نوع (type equivalence)
  - بر اساس بررسی برابری ساختاری (structural equivalence) دو عبارت نوع
    - هر دو نوع‌های پایه یکسان باشند
    - هر دو نتیجه اعمال سازنده نوع یکسان به دو عبارت نوع با برابری ساختاری باشند
    - یکی از آنها نام نوعی برای نامگذاری دیگری باشد
  - برابری نام (name equivalence) دو عبارت نوع
    - ساختار کاملاً یکسان دو عبارت نوع
    - دو عبارت نوع با برابری ساختاری که هیچ نام نوعی در یک عبارت برای نامگذاری بخشی از عبارت نوع دیگر استفاده نشده باشد

بررسی نوع

- بررسی برابری ساختاری دو عبارت نوع
  - سعی در یکسان‌سازی دو عبارت نوع (مثلاً در استنتاج نوع)
    - جایگزینی (substitution) متغیرهای نوع با عبارت‌های نوع مناسب
  - مثال:

$$((\alpha_1 \to \alpha_2) \times list(\alpha_3)) \to list(\alpha_2)$$
$$((\alpha_3 \to \alpha_4) \times list(\alpha_3)) \to \alpha_5$$





- گره‌های برابر دارای شماره یکسان هستند

بررسی نوع

- الگوریتم یکسان‌سازی (unification) دو عبارت نوع

```
boolean unify(Node m, Node n) {
    s = find(m); t = find(n);
    if ( s = t ) return true;
    else if ( nodes s and t represent the same basic type ) return true;
    else if (s is an op-node with children s1 and s2 and
             t is an op-node with children t1 and t2) {
        union(s, t);
        return unify(s1, t1) and unify(s2, t2);
    }
    else if (s or t represents a variable) {
        union(s, t);
        return true;
    }
    else return false;
}
```

تابع find، عبارت نوع یک گره را برمی‌گرداند

سازنده نوع یکسان

تابع union، عبارت نوع دو گره را برابر قرار می‌دهد