204433 วิชาการแปลภาษาโปรแกรม

หัวใจของการทำ semantics analysis ก็คือการทำ type checking หรือการตรวจสอบความสอดคล้องกันของชนิดของข้อมูลในโปรแกรม ในวันนี้เราจะได้เรียนรู้พื้นฐานในการทำ type checking

ก่อนหน้านี้เราได้มีพูดถึง symbol table ที่เป็นองค์ประกอบที่คอมไพเลอร์ใช้ในการเก็บและจัดการข้อมูลที่เป็น ตัวแปร หรือ type หรือ ฟังก์ชั่นต่างๆ ในโปรแกรมเพื่อใช้ในการทำ semantics analysis และเราได้เห็นตัวอย่างของการสร้าง symbol table ในคอมไพเลอร์ของภาษา CSubset ซึ่งใช้โครงสร้างข้อมูลแบบ linked-list (ลองพิจารณาว่า symbol table แบบนี้มีข้อดีและข้อด้อยอย่างไร และมีโครงสร้างข้อมูลแบบใดที่เหมาะสมกับการสร้าง symbol table ได้อีกบ้าง) ก่อนที่เราจะอธิบายเรื่อง type checking เรามารู้จักกับคำศัพท์ที่เกี่ยวข้องกับ semantics analysis ที่ควรรู้ไว้

name คือชื่อที่เราตั้งให้กับวัตถุ (หรือสัญลักษณ์) ในโปรแกรมเพื่อแทนตัวแปร หรือ ฟังก์ชั่น หรือ type การตั้งชื่อให้กับวัตถุเหล่านี้เรียกว่าการทำ **binding** และผลลัพธ์จากการทำ binding เรียกว่า **declaration** ซึ่งอาจจะมี **scope** ที่จำกัดเฉพาะส่วนใดส่วนหนึ่งของโปรแกรม (นั่นคือ name ที่เกี่ยวข้องจะถูกมองเห็นเฉพาะในโปรแกรมส่วนนั้น) เรียกว่า **local declaration** หรือไม่จำกัด (นั่นคือ name ที่เกี่ยวข้องจะถูกมองเห็นได้ทั่วทั้งโปรแกรมในทุกๆส่วน) เรียกว่า **global declaration** ในกรณีที่มี declaration ของ name เดียวกันแต่ name นั้นอยู่คนละ scope เราจะให้ declaration ของ name ที่อยู่ใน scope ที่ใกล้กับการใช้งาน name นั้นมากที่สุด (closet) ชนะ พิจารณาส่วนของโปรแกรมด้านล่างที่มี declaration ที่เกี่ยวข้องกับ name คือตัวแปร x อยู่สอง declaration

```
{
    int x = 1;
    int y = 2;
    {
        double x = 3.14;
        y += (int)x;
    }
    y += x;
}
```

สำหรับประโยค y += (int)x เราจะได้ว่า x ในประโยคนี้เป็นปริมาณ double ที่มีค่าเริ่มต้น `3.14` ไม่ใช่ x ที่เป็นปริมาณ int ที่มีค่าเริ่มต้น = 1

ถ้าเราจะอธิบาย symbol table โดยใช้ศัพท์ที่เราได้อธิบายผ่านมา เราจะบอกว่า symbol table ก็คือองค์ประกอบในคอมไพเลอร์ที่ทำหน้าที่:

- เก็บข้อมูลของ name และการ binding ของ name เข้ากับวัตถุ (หรือสัญลักษณ์) ในโปรแกรม และ
- ให้ข้อมูลของ name ที่ถูกต้อง (นั่นคือจับคู่ name เข้ากับ declaration ที่ถูกต้อง) เมื่อใช้ name นั้นในส่วนใดๆของโปรแกรม

Type

การนิยาม type นั้นทำได้ยากเนื่องจากเป็นปริมาณพื้นฐานในภาษาโปรแกรมเช่นเดียวกับจุดที่เป็นปริมาณพื้นฐานในเรขาคณิต ดังนั้นเรามักจะพูดถึงลักษณะของ type มากกว่าโดยที่ถ้าเราพบสิ่งที่มีลักษณะดังต่อไปนี้ มันจะเข้าข่ายว่าเป็น **type** ในภาษาโปรแกรม

- เป็นเซ็ทของค่าของปริมาณใดๆ และ
- เซ็ทของ **operation** ต่างๆที่เป็นไปได้สำหรับค่าของปริมาณนั้น

ในภาษาที่มีการสนับสนุนการโปรแกรมเชิงวัตถุ **(Object-Oriented Programming)** ชื่อของ class สามารถนำมาระบุเป็น type ได้

การมี **type** จะช่วยกรองความผิดพลาดในการรันโปรแกรมลงได้มาก ลองพิจารณาคำสั่ง **MIPS assembly** ต่อไปนี้:
add $r1, $r2, $r3

คำสั่งนี้ทำการบวกข้อมูลที่อยู่ใน **$r2** และ $r3 และนำผลลัพธ์ที่ได้ไปเก็บไว้ที่ $r1 แต่ในระดับ assembly นี้ เราไม่สามารถบอกได้ว่าการบวกข้อมูลที่อยู่ใน $r2 และ $r3 เข้าด้วยกัน เป็นการกระทำที่สมเหตุสมผลหรือไม่ เช่นถ้า $r2 เก็บข้อมูลที่เป็น function pointer แต่ $r3 เก็บข้อมูลที่เป็น **integer** การนำสิ่งที่อยู่ใน **register** ทั้งสองมาบวกกันย่อมไม่สมเหตุสมผล และอาจจะเป็นผลมาจาก **bug** ในโปรแกรม ในทำนองตรงกันข้ามถ้าข้อมูลใน **register** ทั้ง **$r2** และ **$r3** เป็น **integer** การบวกกันย่อมนำมาซึ่งผลลัพธ์ที่เป็นเหตุเป็นผล ดังนั้นคอมไพเลอร์จะต้องพยายามกำจัดพฤติกรรมที่จะนำมาซึ่งการบวกข้อมูลที่เข้ากันไม่ได้ในกรณีแรกโดยการทำ **type checking** ซึ่งเป็นการตีกรอบให้ **operation** ที่เป็นไปได้สำหรับข้อมูล **type** หนึ่งๆ มีการใช้เฉพาะกับข้อมูล **type** นั้นๆเท่านั้น การทำ **type checking** สามารถทำได้ในสองลักษณะคือ static หรือ dynamic ภาษาโปรแกรมที่เป็นแบบ statically-typed จะทำ type checking ในช่วงที่ทำการคอมไพล์โปรแกรม ขณะที่แบบ **dynamically-typed** จะทำ **type checking** ในช่วงเวลาที่โปรแกรมกำลังทำงานอยู่ ทั้งสองแบบมีข้อดี (และข้อด้อย) ที่เราสามารถเปรียบเทียบกันได้ต่อไปนี้

ข้อดีของภาษาแบบ **statically-typed** คือ:

- สามารถตรวจจับความผิดพลาดหลายๆอย่างได้ในขณะทำการคอมไพล์โปรแกรม
- มีประสิทธิภาพดีกว่าแบบ **dynamically-typed** เพราะไม่ต้องแบกภาระในการทำ **type checking** ในขณะที่รันโปรแกรม

ส่วนข้อดีของภาษาแบบ **dynamically-typed** คือ:

- มีความยืดหยุ่นในการเขียนโปรแกรมมากกว่า (เพราะไม่ต้องกังวลถึงข้อจำกัดที่ตัวแปรนี้จะต้องเป็น **type** นั้นอยู่ตลอดเวลา)
- สามารถเขียนโปรแกรมและทำ **prototype** ได้เร็วกว่าแบบ **statically-typed** (เพราะไม่ต้องกังวลว่าจะต้องทำ **declaration** ให้ถูกต้องก่อนจะเริ่มลงมือเขียนตัวโปรแกรมที่ทำงานจริงๆ)

ภาษาโปรแกรมที่เป็นที่นิยมในปัจจุบันที่เป็นแบบ statically-typed เช่น จาวา มักจะมีลักษณะที่มีความเป็น dynamic typing ผนวกเข้าไปด้วยเช่นการทำ implicit cast หรือการทดสอบโดยใช้ instanceof

นอกจาก static กับ dynamic typing แล้ว เรายังสามารถจำแนกภาษาโปรแกรมจากมุมมองในเรื่อง type ได้อีกลักษณะหนึ่ง โดยพิจารณาว่าภาษาโปรแกรมนั้นเป็นแบบ strongly-typed หรือแบบ weakly-typed โดยภาษาโปรแกรมแบบแรกจะไม่ยอมให้มีพฤติกรรมที่ undefined เกิดขึ้นในขณะทำงานจริง ซึ่งตรงกันข้ามกับแบบหลังที่ยอมให้มีพฤติกรรมแบบ undefined เกิดขึ้นได้ พิจารณาส่วนของโปรแกรมในภาษาซีดังต่อไปนี้:

```
FILE *f;
f = fopen("test.txt", 'r');
```

จะเกิดอะไรขึ้นถ้าไฟล์ test.txt ไม่มีอยู่ใน directory ที่ทำการรันโปรแกรม ในข้อกำหนดของภาษาซีไม่ได้มีระบุไว้อย่างชัดเจน นั่นคือ undefined และให้อิสระกับผู้ที่จะสร้างคอมไพเลอร์และ environment ในการรันโปรแกรมภาษาซีเป็นผู้ define พฤติกรรมที่ควรจะเป็นเอง ดังนั้นซีจึงเป็นภาษาแบบ weakly-typed

เปรียบเทียบกับการจะเปิดไฟล์ในภาษาจาวาซึ่งคอมไพเลอร์จะบังคับให้การเรียกฟังก์ชั่นที่จะมีการเกี่ยวข้องกับปฏิบัติการแบบนี้ต้องอยู่ใน try block ซึ่งจะมี catch block ที่จะมาจับคู่กันเพื่อที่จะทำการจัดการกรณีเช่นไม่มีไฟล์ชื่อที่ต้องการเปิดใน directory ที่กำลังรันโปรแกรมอยู่ นั่นคือจาวา define อย่างชัดเจนว่าถ้าเกิดกรณีเช่นนี้แล้ว จะมี exception อะไรเกิดขึ้นได้ และผู้เขียนโปรแกรมจะต้องมีการ handle exception เหล่านี้ใน catch block ดังนั้นเรียกได้ว่าจาวาเป็นภาษาแบบ strongly-typed

ตารางด้านล่างแสดงการจำแนกภาษาโปรแกรมในแง่มุมที่เกี่ยวข้องกับระบบ type

	Strongly-typed	**Weakly-typed**
Statically-typed	Java, Ocaml	C/C++, PASCAL
Dynamically-typed	Python, PHP	Assembly

Type checking

เราใช้พื้นฐานทางคณิตศาสตร์และทฤษฎีทางคอมพิวเตอร์ในกระบวนการวิเคราะห์ของคอมไพเลอร์ในช่วง front-end เช่นการใช้ regular expression ในการทำ lexical analysis และการใช้ CFG ในการทำ syntactic analysis สำหรับการทำ type checking นั้นเราจะใช้พื้นฐานที่มาจากกฎการให้เหตุผลหรือ Rules of Inference ซึ่งมีรูปแบบหลักดังต่อไปนี้:

ถ้า สมมุติฐาน เป็นจริง แล้ว บทสรุป จะเป็นจริง (If Hypothesis is true, then Conclusion is true)

Inference rule สำหรับ type checking จะมีลักษณะทั่วไปดังต่อไปนี้

ถ้า E1 และ E2 ถูกจัดอยู่ใน type Tx หรือ Ty แล้ว E3 จะถูกจัดอยู่ใน type Tz โดย x หรือ y หรือ z เป็นการบ่ง type ชนิดใดๆที่เป็นไปได้ในภาษาโปรแกรมที่เรากำลังพิจารณา

เราใช้สัญลักษณ์ ^ แทนคำว่า “และ” (and) สัญลักษณ์ ⇒ แทนคำว่า “ถ้า แล้ว” (if-then) และ x : T หมายถึง “x มี type เป็น T” (x has type T) ดังนั้นประโยค ถ้า e1 มี type เป็น int และ e2 มี type เป็น int แล้ว e1 + e2 มี type เป็น int สามารถแทนด้วยสัญลักษณ์ที่เราได้กล่าวมาได้เป็น (e1 : int ^ e2 : int) ⇒ e1 + e2 : int

ประโยค (e1 : int ^ e2 : int) ⇒ e1 + e2 : int เป็นกรณีพิเศษของ inference rule โดยทั่วไปที่จะมีรูปแบบต่อไปนี้

Hypothesis_1 ^ Hypothesis_2 … ^ Hypothesis_n ⇒ Conclusion

โดยทั่วไปเราเขียน inference rule ในลักษณะดังต่อไปนี้

$$\frac{\vdash \text{Hypothesis} \ldots \vdash \text{Hypothesis}}{\vdash \text{Conclusion}}$$

โดยสัญลักษณ์ ⊢ แทนประโยค “เราสามารถพิสูจน์ได้ว่า” (It is provable that …)

Inference rule จะ sound (มีความถูกต้องตามหลักเหตุและผล) ก็ต่อเมื่อเราเริ่มจากการให้ว่าสมมุติฐานเป็นจริง และเราใช้หลักการอ้างเหตุผลที่ถูกต้องไปในแต่ละขั้นตอน (logical deduction) แล้วได้ข้อสรุปที่เป็นจริงด้วย

[ตัดไปที่สไลด์เรื่อง type rules]