1 장
소개

강대기

프로그래밍 언어란 무엇인가?
프로그래밍 언어에서의 추상화
연산 패러다임
언어 정의, 구현 및 설계

원리

- 왜 프로그래밍 언어를 배우는가?
  - 언어가 사고를 지배함 – 언어는 사고의 감옥
  - 프로그래밍 언어는 우리가 컴퓨터에 대해 생각하는 바에 영향을 줌
- 컴퓨터가 하는 일은?
  - 연산
  - 정보 처리
  - 컴퓨터≈ 대략 사고하는 기계
  - 연산의 상한: 튜링 머신

프로그래밍 언어란?

- 프로그래밍 언어의 정의 1
  - 컴퓨터에게 우리가 원하는 일을 시키기 위한 표현 체계
    - 인간 대 기계 간의 통신 모델
- 프로그래밍 언어의 정의 2
  - 연산을 기계가 읽을 수 있고 사람이 읽을 수 있도록 기술하는 표현 체계
    - 사람 대 기계 통신 모델
    - 또한 사람 대 사람 통신 모델

연산이란?

- 튜링 머신 **(중요!)**
  - 컴퓨터의 수학적 모델
- 처치의 설정 (Church's Thesis)
  - 튜링 머신보다 더 강력한 기계를 만드는 것은 근본적으로 불가능하다.
- 연산
  - 컴퓨터에 의해 수행될 수 있는 모든 처리 과정
- 튜링 완전성
  - 튜링 머신이 수행할 수 있는 어떤 연산이라도 기술할 수 있는 프로그래밍 언어는 튜링 완전 (Turing complete)하다

기계 가독성(기계가 읽을 수 있음)

- 프로그래밍 언어는 기계가 읽을 수 있는 코드로 효율적으로 번역할 수 있는 정도로 단순해야 함
  - 번역이 가능해야 함
    - 번역하는 알고리즘이 존재해야 함
  - 번역은 충분히 쉬어야(easy) 함
    - 컴파일 시간이 컴파일된 프로그램의 실제 실행 시간보다 크면 안됨
    - 컴파일 시간이 선형 시간에 수행 가능하게 한다면, 문법은 컨텍스트 프리 문법(context-free grammar)에 의해 기술되어야 함

인간 가독성(사람이 읽기 쉬워야)

- 추상화
  - 프로그램 언어는 컴퓨터 하드웨어의 추상화를 제공해야 함
  - 프로그래밍 언어는 컴퓨터 기계의 상세한 부분은 감출 수 있어야 함
- 복잡도 제어
  - 큰 프로그램을 읽기 힘들다
  - 연산의 효과가 나타내는 범위를 작게 한다면 가독성이 높아짐
  - 큰 프로그램의 복잡성을 제어하기 위한 방법 중 하나로 모듈화 접근 방법이 있음

추상화

- 추상화가 가진 두 가지 측면
  - 상세함을 감춘다
  - 중요한 부분은 보인다
- 몬드리안(Mondrian) 나무들



사람이 용이하게 쓰도록(write) 하는 점은 어떤가

- 프로그래밍 언어는 프로그램을 *읽기* 쉽게 하는 게 아니라 *쓰기* 쉽게 하는 거 아니던가요?
- 프로그램을 쓰기 쉽게 하는 건, 전문가나 해커들이 좋아하는 목표임
  - Perl은 매우 쓰기 쉬우나 읽기는 어려움
- 가독성(Readability)이야말로 진정한 목표
  - 프로그래머가 프로그램을 짜면 그 프로그램은 후에 많은 사람들이 읽게 됨
- 유지 보수성(maintainabilty)은 어떤가?
  - 유지보수성은 프로그래밍 언어의 가독성(readability)과 쓰기 편함(writability)과 직접적으로 연관되어 있음

프로그래밍에서의 추상화

- 추상화 범주
  - 데이터 추상화
  - 제어 추상화
- 추상화 레벨
  - 기본 추상화: 지역화된 데이터/정보의 추상화
  - 구조화된 추상화: 프로그램 구조의 추상화
  - 단위 추상화: 전체 프로그램의 추상화

추상화의 예

	데이터 추상화	제어 추상화
기본 추상화	기본적인 값, 변수, 데이터 형	표현식(expression) 평가, goto, 배정문(assignment)
구조화된 추상화	데이터 구조, 구조화된 형	구조화된 제어, 서브프로그램
단위 추상화	추상화 데이터 형, 소프트웨어 컴포넌트, 라이브러리	

연산 패러다임

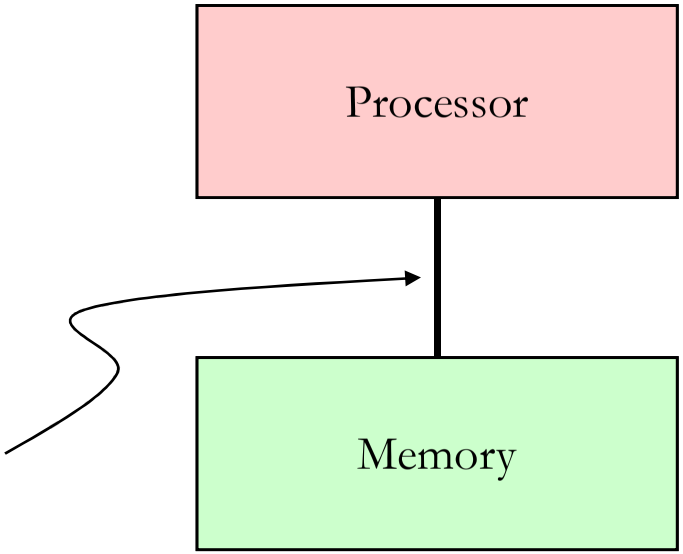
- 패러다임이란?
  - 토마스 쿤
  - 스타일, 패턴, 전형적인 예들
  - 문제 해결을 위한 일반적인 모델
- 연산 패러다임에 영향을 주는 요소
  - 컴퓨터 하드웨어
  - 수학적인 연산 모델

프로그래밍 패러다임 (중요!)

- http://en.wikipedia.org/wiki/Programming_paradigm
- 임퍼러티브 패러다임(Imperative Paradigm)
  - 폰 노이만 모델(von Neumann model)
  - 내장 프로그램에 의해 프로세서가 움직인다
- 함수 패러다임(Functional Paradigm)
  - 함수의 추상적인 개념에 근거
  - 람다 칼큘러스 또는 람다 계산법(lambda calculus) – 주로 함수 표현에 기반하는 형식 시스템(쉽게 말해 프로그래밍 언어)
- 논리 패러다임(Logic Paradigm)
  - 기호 논리에 근거함
  - 수학적인 연역법(deduction)
- 객체지향 패러다임(Object-Oriented Paradigm)
  - 실세계, 객체들 간의 상호작용에 근거
  - 임페러디브 패러다임의 확장

임퍼러티브 프로그래밍

- 폰 노이만 모델 (von Neumann Model)
  - 프로세서는 내장된 컴퓨터 프로그램에 의해 움직인다
  - 컴퓨터 메모리 안에 데이터 뿐만 아니라 프로그램도 들어간다 (너무 당연해 보이지만 그 당시에는 획기적이었음)
  - fetch-decode-execute 사이클
- 폰 노이만 병목현상 (bottle neck)
- 대표적 언어: C



임퍼러티브 프로그래밍 예

```
function gcd (u, v: in integer) return integer is
  y, t, z: integer;
begin
  z := u;
  y := v;
  loop
    exit when y = 0;
    t := y;
    y := z mod y;
    z := t;
  end loop;
  return z;
end gcd;
```

객체 지향 프로그래밍

- 객체
  - 메모리 위치 + 적용가능한 오퍼레이션
- 클래스
  - 동일한 특성을 가진 모든 객체들을 기술함
  - 객체들의 행동을 정의함
  - 클래스의 인스턴스가 객체
- 메시지와 메쏘드
  - 1 + 2
  - 임퍼러티브한 관점: + 는 데이터 1,2에 적용됨
  - 객체 지향 관점: 메시지 (+ 2)가 객체 1에 보내짐, 이 때 +는 객체 1의 메쏘드

객체 지향 프로그래밍 예 (1/2)

```
import java.io.*;
class IntWithGcd
{ public IntWithGcd( int val ) { value = val; }
  public int getValue() { return value; }
  public int gcd ( int v )
  { int z = value; /* "imperative" version */
    int y = v;
    while ( y != 0 )
    { int t = y; y = z % y; z = t;
    }
    return z;
  }
  private int value;
}
```

객체 지향 프로그래밍 예 (2/2)

```
class GcdProg /* driver */
{ public static void main (String args[])
  { System.out.println("Input two integers:");
    BufferedReader in = new BufferedReader(
          new InputStreamReader(System.in));
    try /* must handle I/O exceptions */
    { IntWithGcd x = /* create an object */
      new IntWithGcd(Integer.parseInt(in.readLine()));
      int y = Integer.parseInt(in.readLine());
      System.out.print("The gcd of " + x.getValue()
                       + " and " + y + " is ");
      System.out.println(x.gcd(y)); }
    catch ( Exception e)
    { System.out.println(e); System.exit(1); }
  }
}
```

함수 프로그래밍

- 연산이란 함수의 평가이다.
  - 연산을 함수의 평가라는 관점에서 응용한 언어들
- 함수 언어
  - 값 기반
  - 변수나 배정문은 사용 불가
  - 루프도 사용 불가
- 재귀 함수 이론에 근거한 이론적인 특성들

  튜링 완전성은 다음을 통해 얻을 수 있음
  - 정수 값들
  - 수학 함수들
  - 이미 가지고 있는 함수, 선택, 및 재귀를 통해 새로운 함수를 정의할 수 있는 메카니즘

함수 프로그래밍 예

```
(define (gcd u v)
   (if (= v 0) u
       (gcd v (modulo u v))))
(define (euclid) ; sequential!
   (display "enter two integers:")
   (newline) ; goes to next line on screen
   (let ((u (read)) (v (read)))
      (display "the gcd of ")
      (display u)
      (display " and ")
      (display v)
      (display " is ")
      (display (gcd u v))
      (newline)))
```

function **gcd**

gcd driver

논리 프로그래밍

- 연산은 증명이다
  - 선언적인 언어들
- 논리 언어들
  - 프로그램은 참인 주장(assertion)들의 집합
  - 하부 구조에 내재된 시스템들이 제어 흐름을 결정
  - 아주 고차원적인 언어들

논리 프로그래밍 예

```
gcd(U, V, U) :- V = 0.
gcd(U, V, X) :- not(V = 0),
                Y is U mod V,
                gcd(V, Y, X).
```

다중 패러다임 언어

- 순수하게 하나의 패러다임만 가지고 있는 언어는 거의 없음
- 현재의 임퍼러티브 언어들은 재귀 호출을 지원함
  - 다음 슬라이드의 C 프로그램
- 어떤 객체 지향 언어들은 기본형 데이터와 객체들을 구별함
  - 자바 언어는 두 개의 정수 데이터형이 있음: 기본 정수(**`int`**), 그리고 객체 정수(**`Integer`**)
- 대부분의 함수 언어들은 시퀀싱 (순서대로 나열) 기능이 있음
  - Scheme 언어에서는 입출력을 위해 시퀀싱 사용

함수 스타일 C 프로그램

```
#include <stdio.h>
int gcd(int u, int v) /* "functional" version */
{  if (v == 0) return u;
   else return gcd (v, u % v); /* "tail" recursion */
}
main() /* I/O driver */
{ int x, y;
  printf("Input two integers:\n");
  scanf("%d%d",&x,&y);
  printf("The gcd of %d and %d is %d\n",
                              x,y,gcd(x,y));
  return 0;
}
```

일반 재귀 호출(Recursion)

```
int sum(int n)
{
 if(n == 1)
   return 1;
 return n + sum(n-1);
}
```

```
sum(3) =>
 3 + sum(2)          <-- (1)
   3 + (2 + sum(1))
     3 + (2 + (1))
   3 + (3)           <-- (2)
 6
```

꼬리 재귀 호출(Tail Recursion)

```
int sum(int n, int acc)
{
 if(n == 1)
   return (acc+1);
 return sum(n-1, acc+n);
}

sum(3, 0) =>
 sum(2, 3)            <-- (3)
   sum(1, 5)
     6                <-- (4)
   6
 6
```

언어의 정의

- 언어를 왜 정의해야 하는가?
  1. 주어진 프로그램에 대해 어떤 연산이 실제 수행되는가를 알기 위해
  2. 프로그램에 대해 수학적으로 사고해 내기 위해
  3. 언어의 구현을 정확히 하기 위해 (기계에 독립적으로)
  4. 프로그래머가 부딪히는 어려운 문제를 풀기 위해
  5. 언어의 설계 과정에서 이론적 원리를 만들고 제공하기 위해
- 언어의 정의가 필요한 사람은?
  - 프로그래머: 1, 2, 4
  - 언어를 구현하는 사람: 3
  - 언어를 설계하는 사람: 5

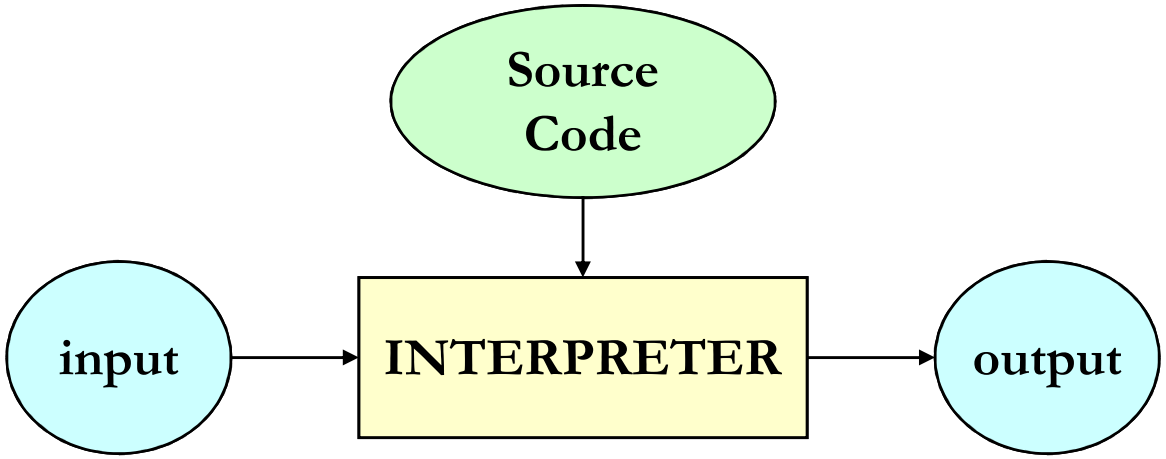
언어를 정의하는 방법

- 언어는 다음으로 정의됨
  - 신택스(syntax): 프로그램의 구조
  - 시멘틱스(semantics): 프로그램의 의미 또는 실행
    - 참고: 시멘틱 웹
- 신택스의 정의
  - 우리가 쓰는 자연어
  - 컨택스트 프리 언어(context-free grammar) (4 장 참고)
- 의미의 정의
  - 자연어: 정확하지 않음
  - 형식적 방법 (formal methods): 너무 어려움(13 장)

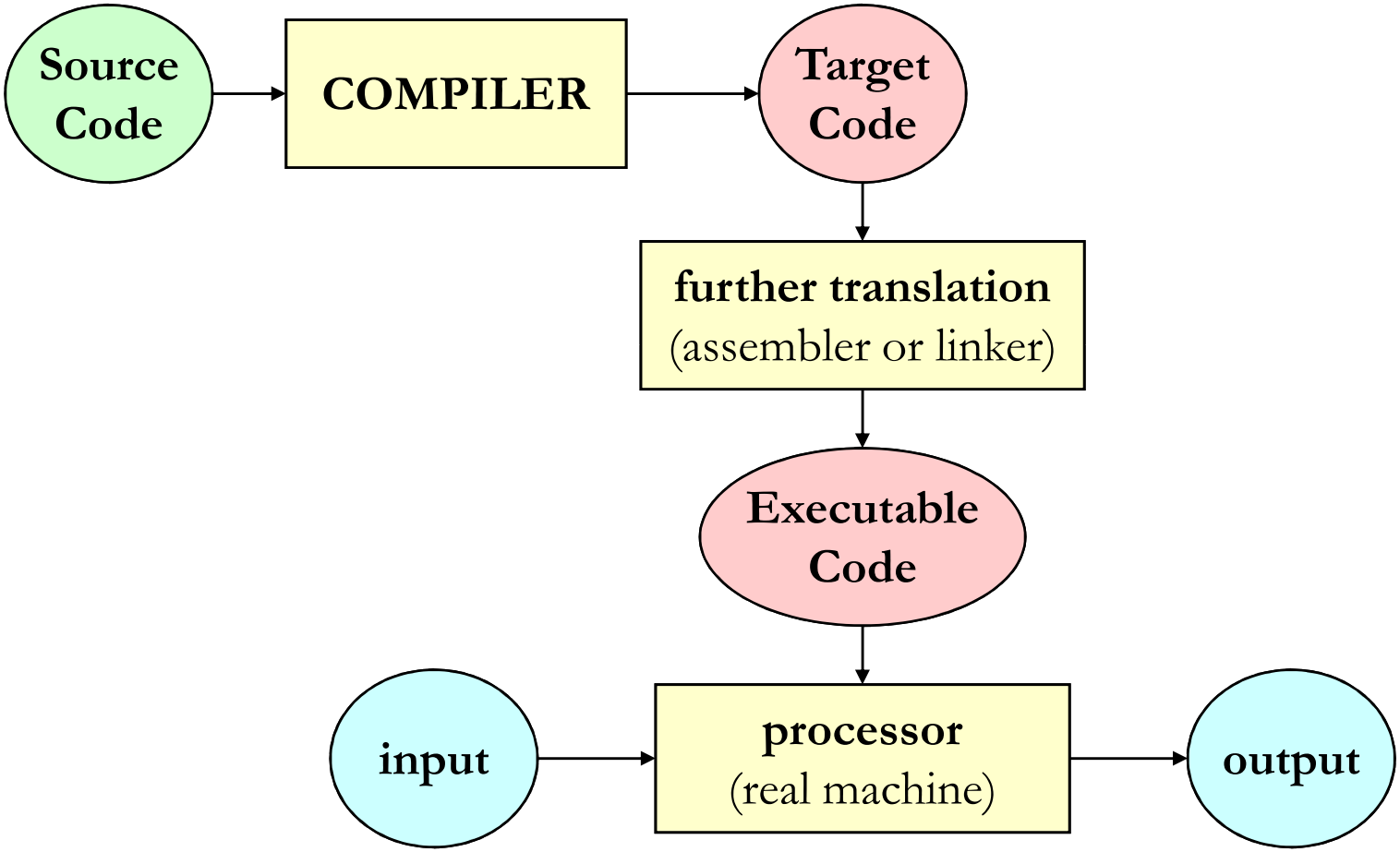
언어 번역기

- 언어 번역기
  - 언어로 쓰여진 프로그램을 받아서 실행함
  - 언어 처리기라고도 함
- **언어 번역기의 두 가지 형태 (중요!)**
  - 인터프리터(interpreter) - 통역
  - 컴파일러(compiler) - 번역
  - 의사 인터프리터 또는 하이브리드 번역기 (pseudo-interpreter, or hybrid translator)

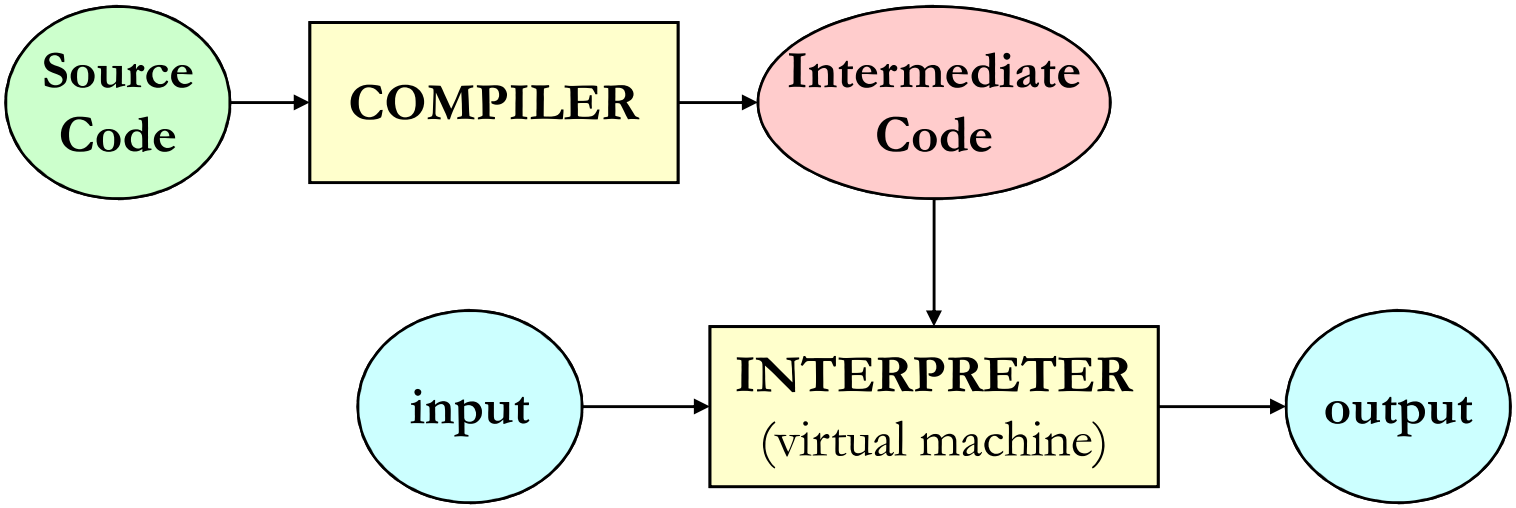
인터프리터



컴파일러


Source Code
COMPILER
Target Code
further translation
(assembler or linker)
Executable Code
input
processor
(real machine)
output

의사 인터프리터


Source Code
COMPILER
Intermediate Code
input
INTERPRETER
(virtual machine)
output

언어 번역의 중요 이슈들

- 비표준적인 특성들
  - 언어 ⊂ 번역기가 허용하는 언어
  - 즉, 번역기가 비 표준적인 특성을 가질 수 있다는 것임 → 구현에 의존하는 프로그램
  - 비표준적인 특성들은 사용하지 않는 것이 좋음
- 언어 특성들의 분류
  - 정적 특성들: 프로그램 실행 이전에 결정됨
  - 동적 특성들: 프로그램 실행 중에 결정됨
  - 언어 특성들은 구현 방법과 그에 상응하는 실행 환경을 결정함

번역의 효율성

	번역 비용	실행 속도
컴파일	높음	빠름
인터프리트	없음	느림
하이브리드	중간	중간

에러 처리

- 에러의 종류
  - 신택스 에러 (구문 에러): 프로그램이 제대로 표현되지 않았음, 다른 말로 폼이 안좋음 (not well-formed)
  - 시멘틱 에러 (의미 에러)
    - 정적 시멘틱 에러: 컴파일 시간에 잡을 수 있음 (예를 들어, 호환되지 않는 데이터 타입)
    - 동적 시멘틱 에러: 실행 시간에 잡을 수 있음 (예를 들어, 0으로 나누는 것)
  - 논리 에러: 프로그래머 만이 알고 있는 에러
- 대부분의 번역기들은 디버깅 툴들을 지원함

에러의 예 (Java)

```
public int gcd ( int v# ) // lexical error
{ int z = value     // syntax error - missing ;
  y = v;            // static semantic error -
                    //     y undefined
  while ( y >= 0 ) // dynamic semantic -
                    //     division by zero
  { int t = y; y = z % y; z = t;
  }
  return y;         // logic error - should return t
}
```

언어 설계

- 추상화 메커니즘
  - 사람이 읽을 수 있어야 하는 데, 중요한 요소
  - 훌륭하고 일관성 있는 추상화 집합이 제공
- 킬러 응용프로그램 (크게 히트치는 프로그램)
  - C: Unix
  - Java: Internet
  - C++: 가장 효율적인 객체지향언어
- 언어의 실용성
  - 다른 언어나 시스템과 인터페이스가 용이함
  - 좋은 API 라이브러리