프로그래밍 원리/언어/도구

Programming Principles/Languages/Tools

이광근

서울대학교

컴퓨터공학부

kwangkeunyi.snu.ac.kr

차례

컴퓨터라는 도구

인류역사에 유례가 없던 도구

컴퓨터의 놀라운 점은 뭘까?

컴퓨터 v.s. 칼, 활, 바퀴, 고무밴드, 자동차, 냉장고, · · ·

컴퓨터 = "만능"

만능기계, 컴퓨터의 사용법

컴퓨터	다른 도구
글쓰기	힘쓰기
마음, 지혜	팔다리 근육

글쓰기 = “프로그램짜기”

사용 언어 = “프로그래밍 언어”

프로그래밍 언어의 두 기원

프로그램 오류 자동 검증: 타입 체킹

프로그램 실행 자원 자동 관리: 메모리 재활용

프로그래밍에 동원되는 생각방식 14선

프로그래밍 언어의 두 뿌리

- *튜링기계Turing Machine* (Alan Turing)
  - 상위로 상위로: 기계에 명령하는 언어
  - C, C++, C#, Java, JS, Objective C, Python, PHP, Scala, Swift, Rust 등
- *람다계산법Lambda Calculus* (Alonzo Church)
  - 상위로 상위로: 값을 계산하는 언어
  - OCaml, Haskell, F#, Scala, Lisp, Scheme, Racket, Clojure, Lua 등
- 최근 언어들은 두 방식 모두를 지원

기계의 중력 vs 람다의 중력

"기계의 중력"에서 프로그래밍

기계중심, 명령형 프로그래밍

- 명령하며 기계상태를 변화시키는 주문
- 물건/기계상태의 변화: 자연스러운/직관적인 현상
- 요리하기(냄비내용물), 장보기(장바구니), 공부하기(마음)

```
cup = water(100);
cup.heat(500); cup.add(tea); cup.sip();
cup = water(100);
cup.heat(100); cup.add(noodle); cup.chrup();
```

"람다의 중력"에서 프로그래밍

값중심, 계산형 프로그래밍

- 기계는 없다. 값을 계산하는 것 뿐.
- 값은 변하지 않는다. 새 값이 만들어 질 뿐. "1+2"
- 익숙했던 방식: "산수 스타일"
- 요리하기(냄비내용물), 장보기(장바구니), 공부하기(마음)

```
cup = water 100;
sip( (cup heat 500) add tea );
chrup( (cup heat 100) add noodle );
```

$$A \cup B, \quad \sum_{1}^{10}(x+1), \quad \int_{1}^{\infty} \frac{1}{x}dx, \quad \frac{d}{dx}(x^2+1)$$

$P \subset Q$ 이고 $P \neq Q$라고하자. $P^c \cup Q$와 $P \cap Q^c$ 와 $P \cup Q^c$의 합집합으로 ...

람다 중력권의 잇점

코딩력↑ × 안전성↑

- 신속한 프로그래밍
  - 기계염두 없이/기계보다 상위에서: 코딩력 +1
  - 메모리 자동 재활용: 코딩력 +1
  - 실행비용 기계중심언어와 비슷
- 편안한 프로그래밍
  - 2세대 오류 검증기술 장착: 타입체킹 *type checking*
  - 자동이고, 믿어도 되고, 귀찮게 안하고: 안전성 +2, 코딩력 +2
- 미래준비된 프로그래밍
  - 3세대 오류 검증기술과 밀접: Coq & OCaml
  - 람다중력권의 거울: 프로그램짜기 ⟷ 증명하기

왜 기계중력권의 언어들은 못하나?

- 선발주자의 덫, 기계중력권 언어들
  - 디지털 컴퓨터와 기계권 언어의 밀궁합: "첫끝발"
  - 상식적으로 언어만들기 관성
  - 축적된 연구결과가 나오기 이전. 나와도 관성탓
- 후발주자의 잇점, 람다중력권 언어들
  - 디지털 컴퓨터와 람다권 언어의 미궁합: "뒷끝발"
  - 축적된 성과를 가지고 언어만들기.
  - 축적된 연구결과 + 번역기술 + 실행기술. 후발의 장단점

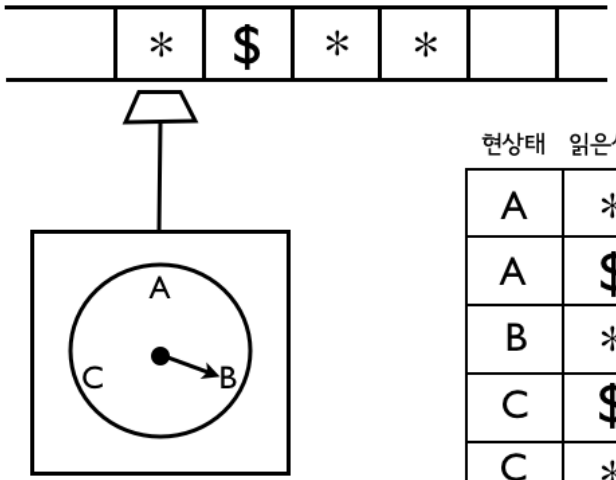
No War. 두 중력권을 모두 즐길 프로그래밍언어로 발전중

튜링기계(Turing Machine)

"기계적인"의 정의 = "튜링기계로 만들어 돌릴 수 있는" (1935년)

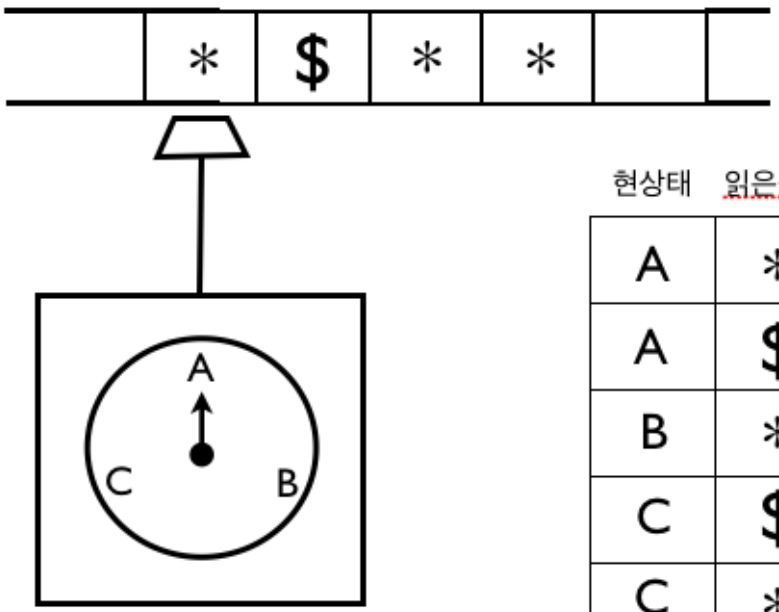
프로그램 하나	=	튜링기계 하나
컴퓨터	=	튜링기계 하나

튜링기계의 한 예:



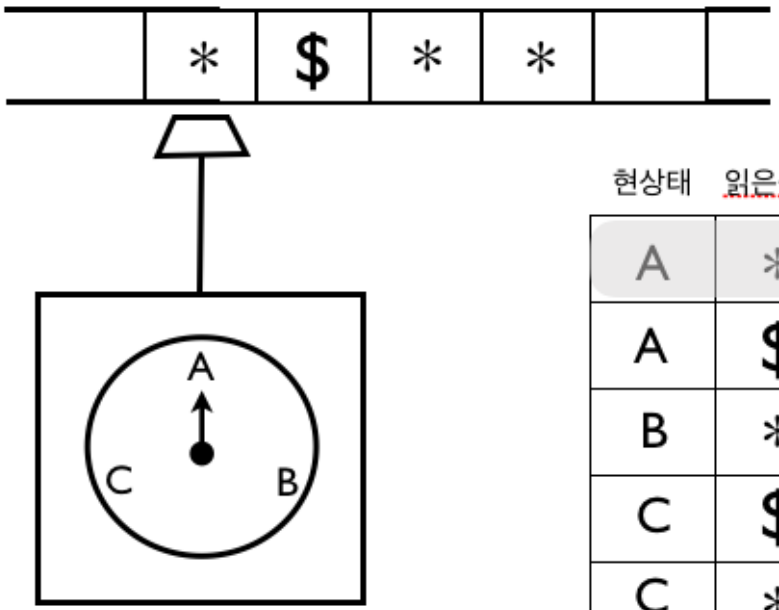
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



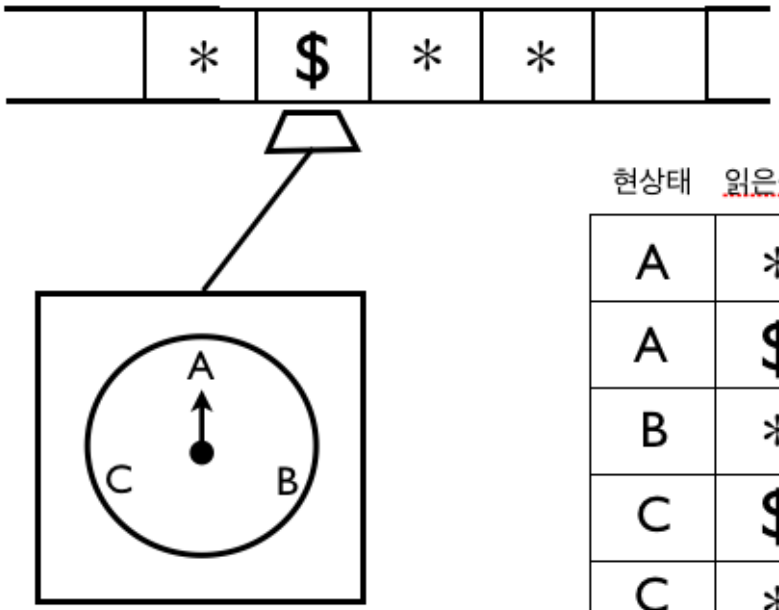
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



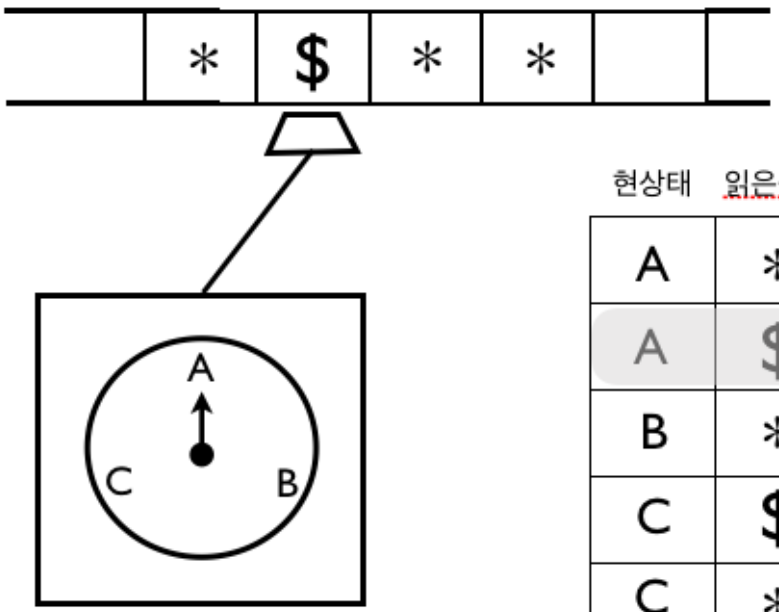
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



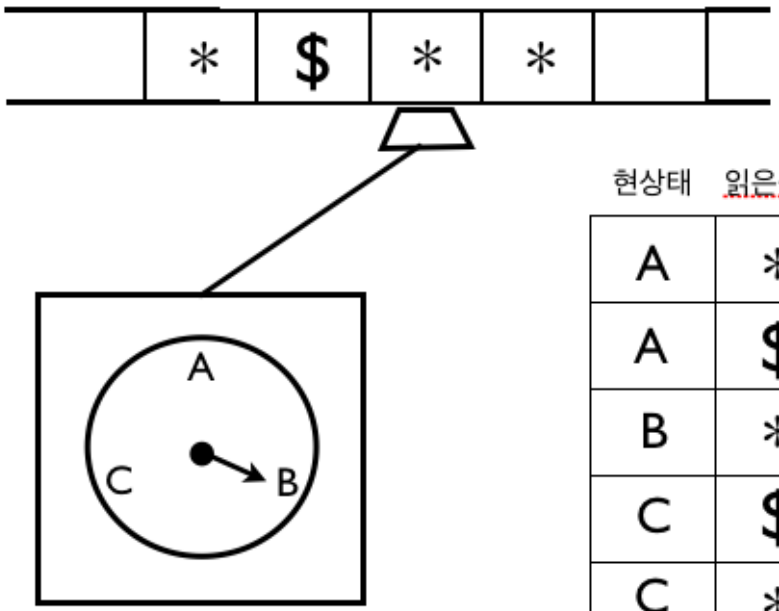
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



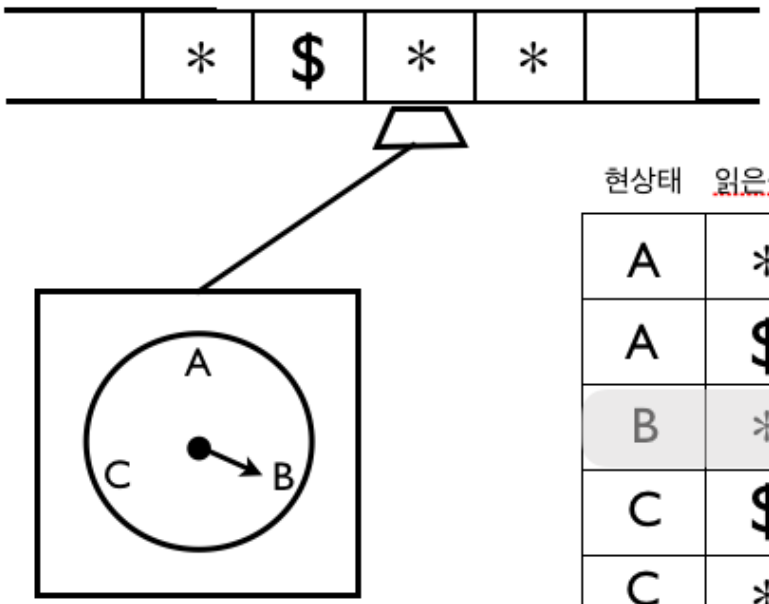
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



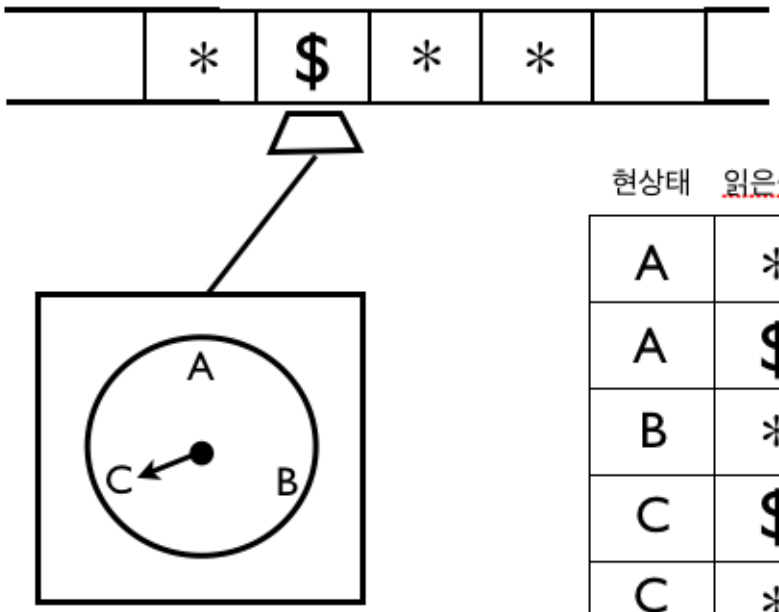
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



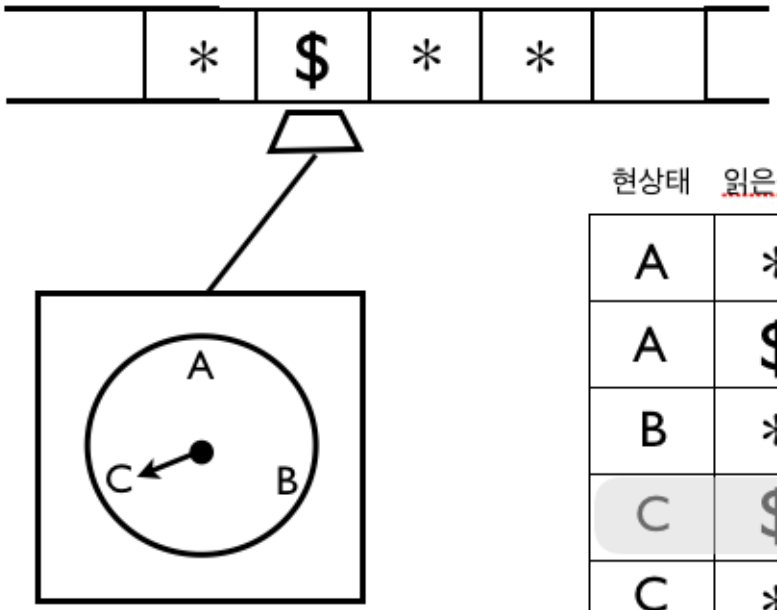
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



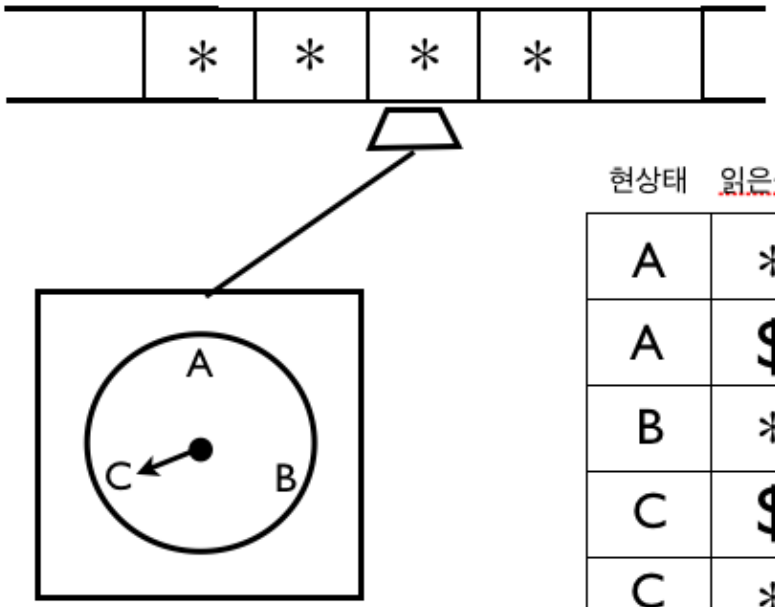
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



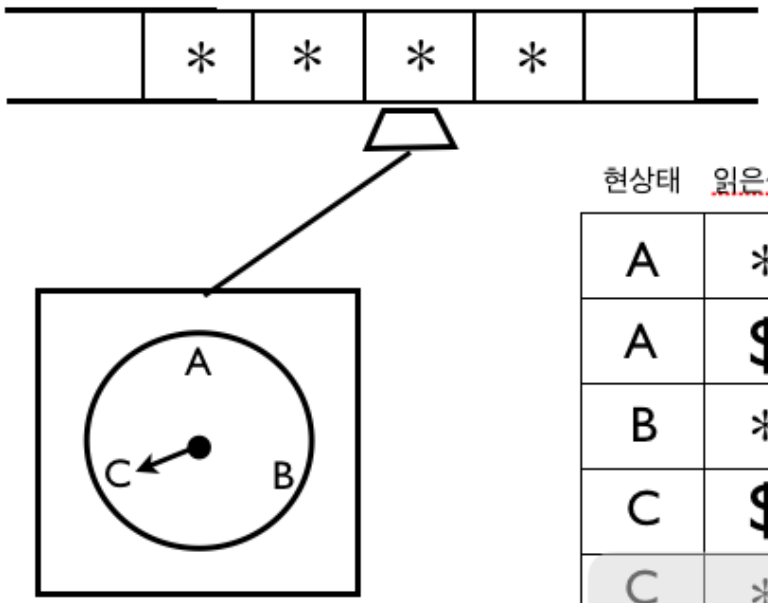
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



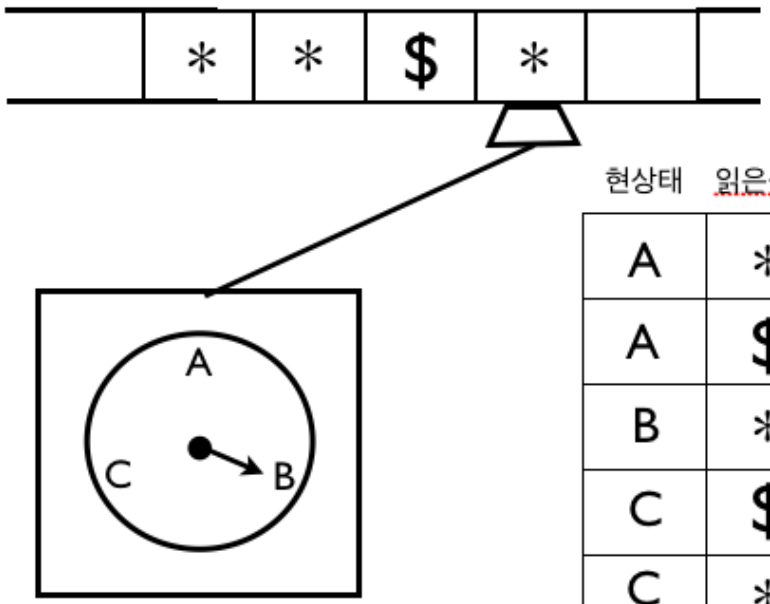
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계의 작동



현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

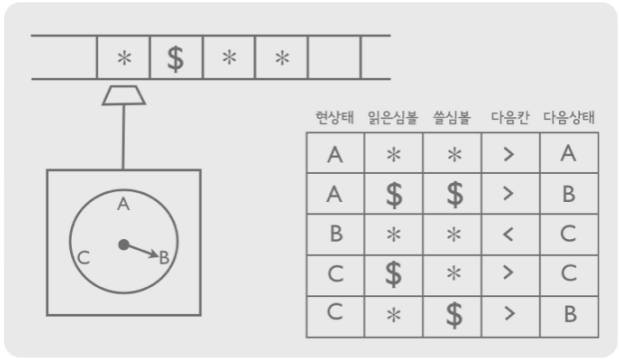
튜링기계의 작동



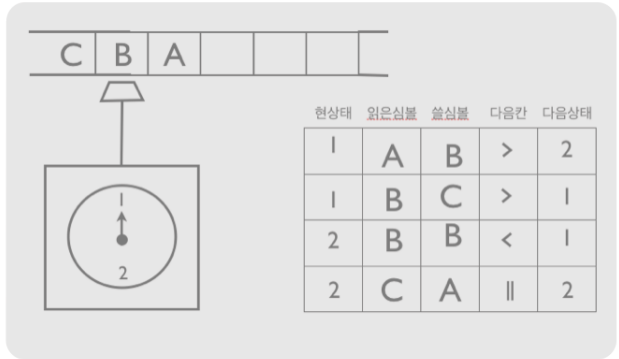
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

튜링기계 하나 = 프로그램 하나

더하기 튜링기계, 카톡 튜링기계, 유튜브 튜링기계, 등등



현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B



현상태	읽은심볼	쓸심볼	다음칸	다음상태
1	A	B	>	2
1	B	C	>	1
2	B	B	<	1
2	C	A	|	2

⋮

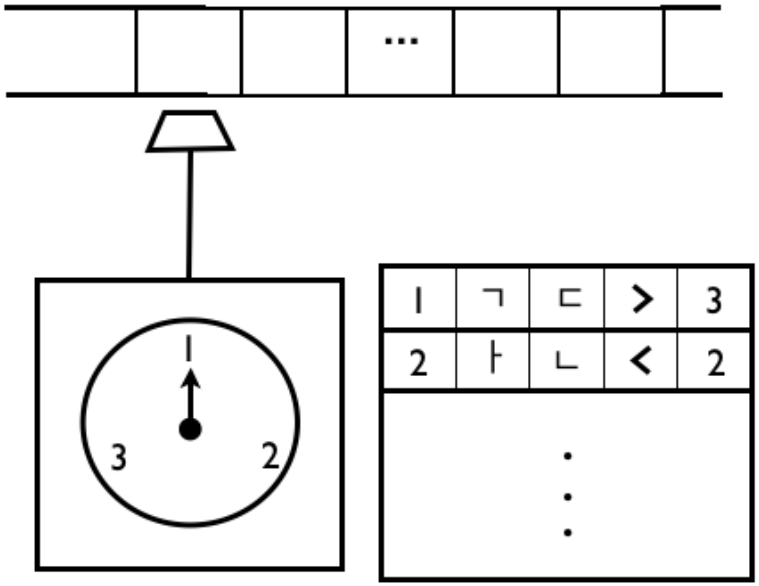
소품으로 등장한 특이한 튜링기계 하나

만능 튜링기계$_{universal\ machine}$
하나의 튜링기계, 그러나 "만능"

- 입력: 튜링기계를 글로 표현해서 테잎에 받는다
- 출력: 그 튜링기계의 작동을 그대로 따라한다

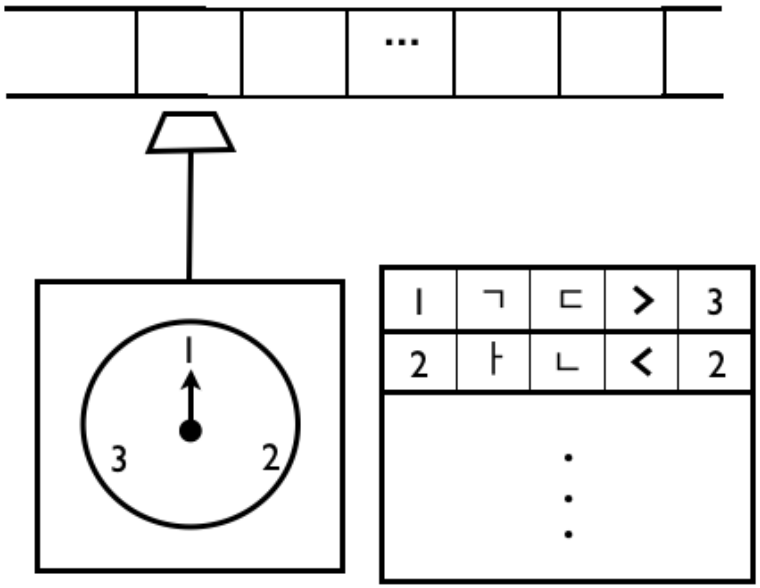
만능 튜링기계(universal machine)

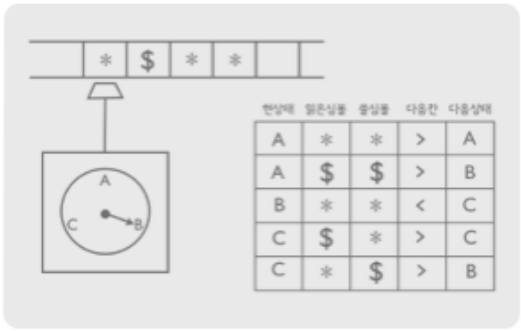
하나의 튜링기계, 그러나 "만능"

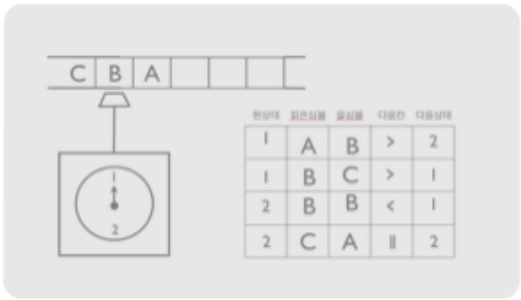


만능 튜링기계(universal machine)

하나의 튜링기계, 그러나 "만능"

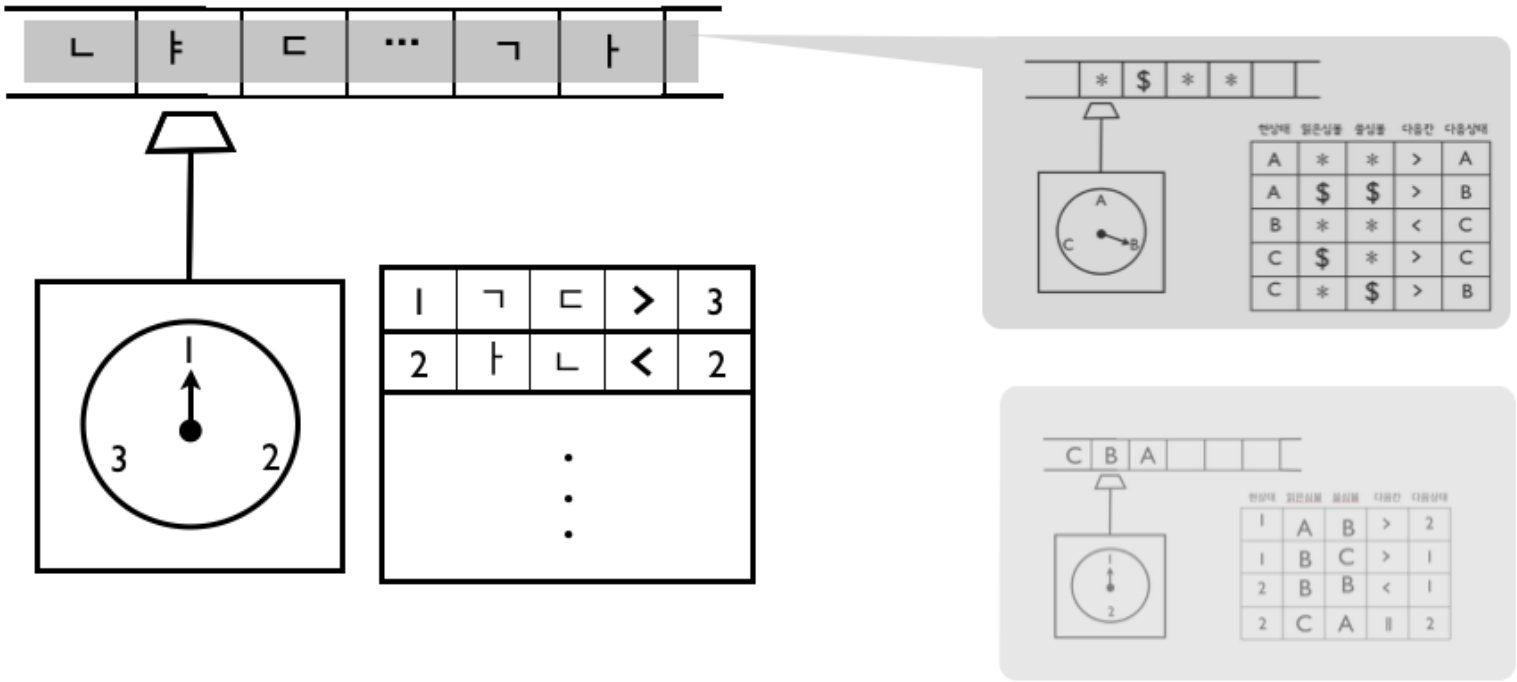






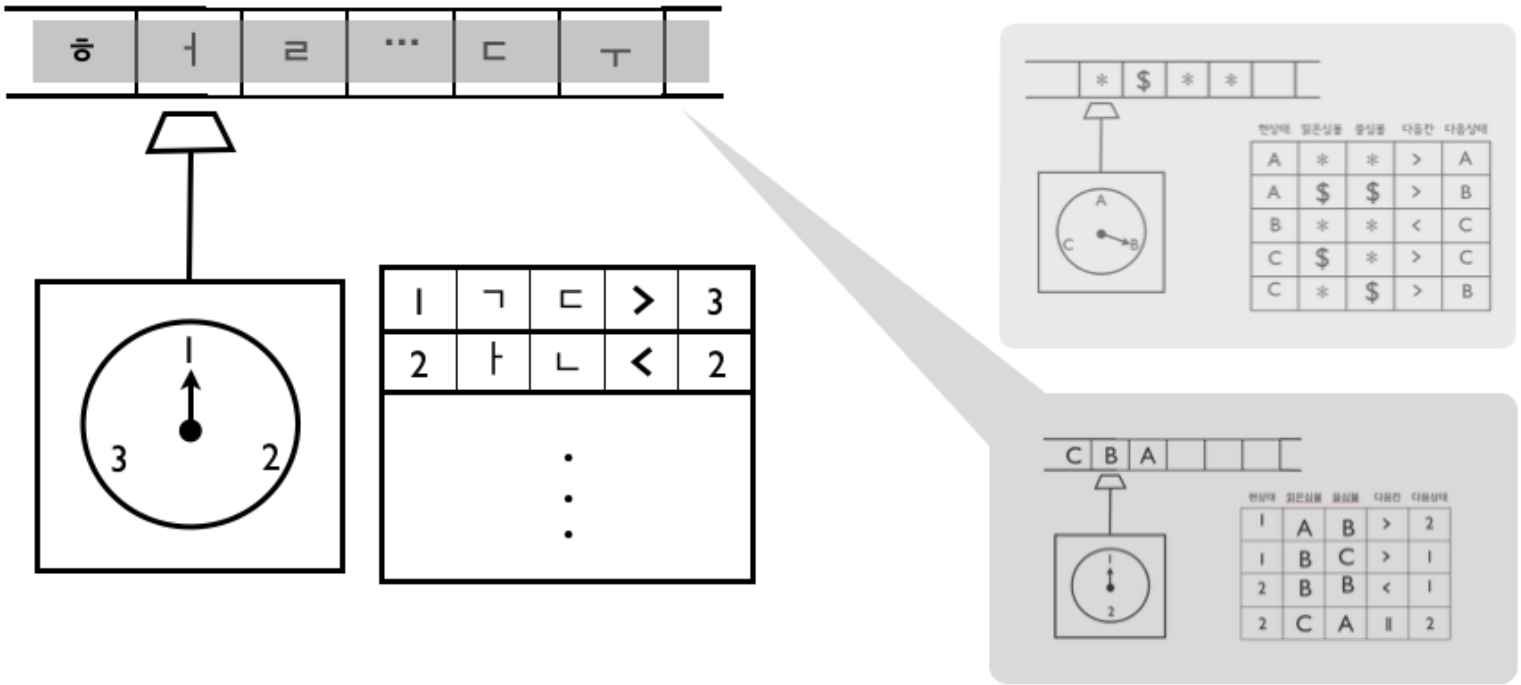
만능 튜링기계(universal machine)

하나의 튜링기계, 그러나 "만능"

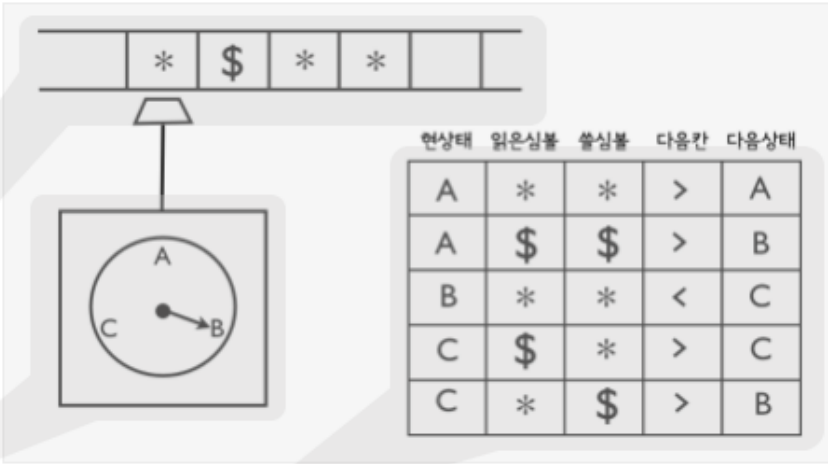


만능 튜링기계(universal machine)

하나의 튜링기계, 그러나 "만능"



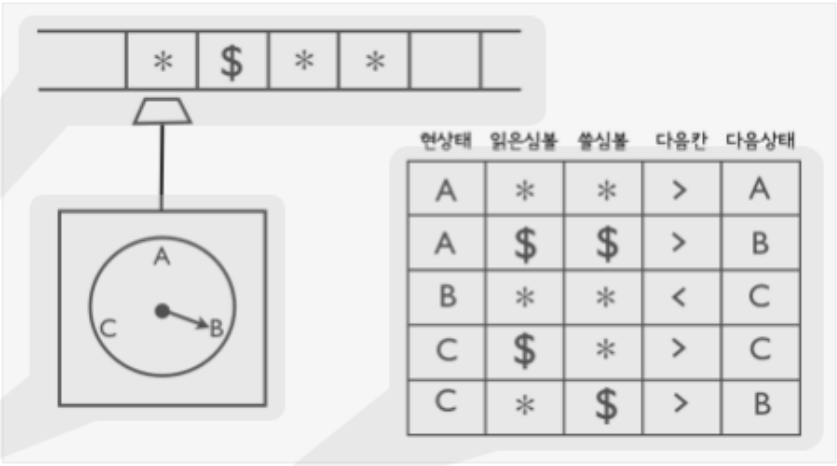
만능 튜링기계: 튜링기계를 입력받고 실행?



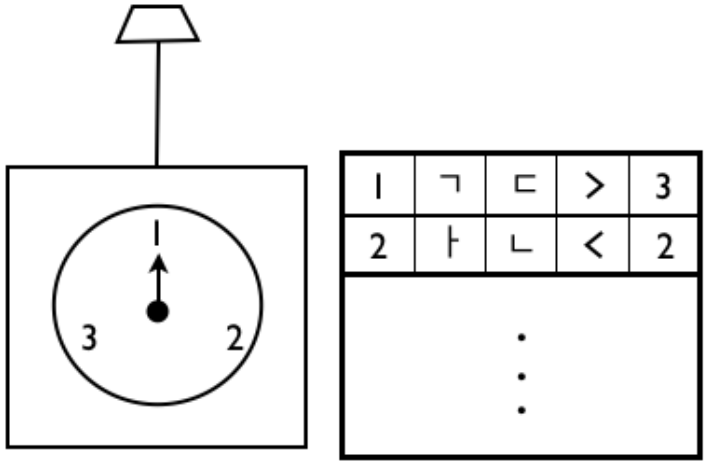
현상태	읽은심볼	쓸심볼	다음칸	다음상태
A	*	*	>	A
A	$	$	>	B
B	*	*	<	C
C	$	*	>	C
C	*	$	>	B

테입[ㅌ1]ㅌ2ㅌ1ㅌ1 상태ㅅ2 규칙ㅅ1ㅌ1ㅌ1>ㅅ1 규칙ㅅ1ㅌ2ㅌ2<ㅅ2 ...

만능 튜링기계: 튜링기계를 입력받고 실행?

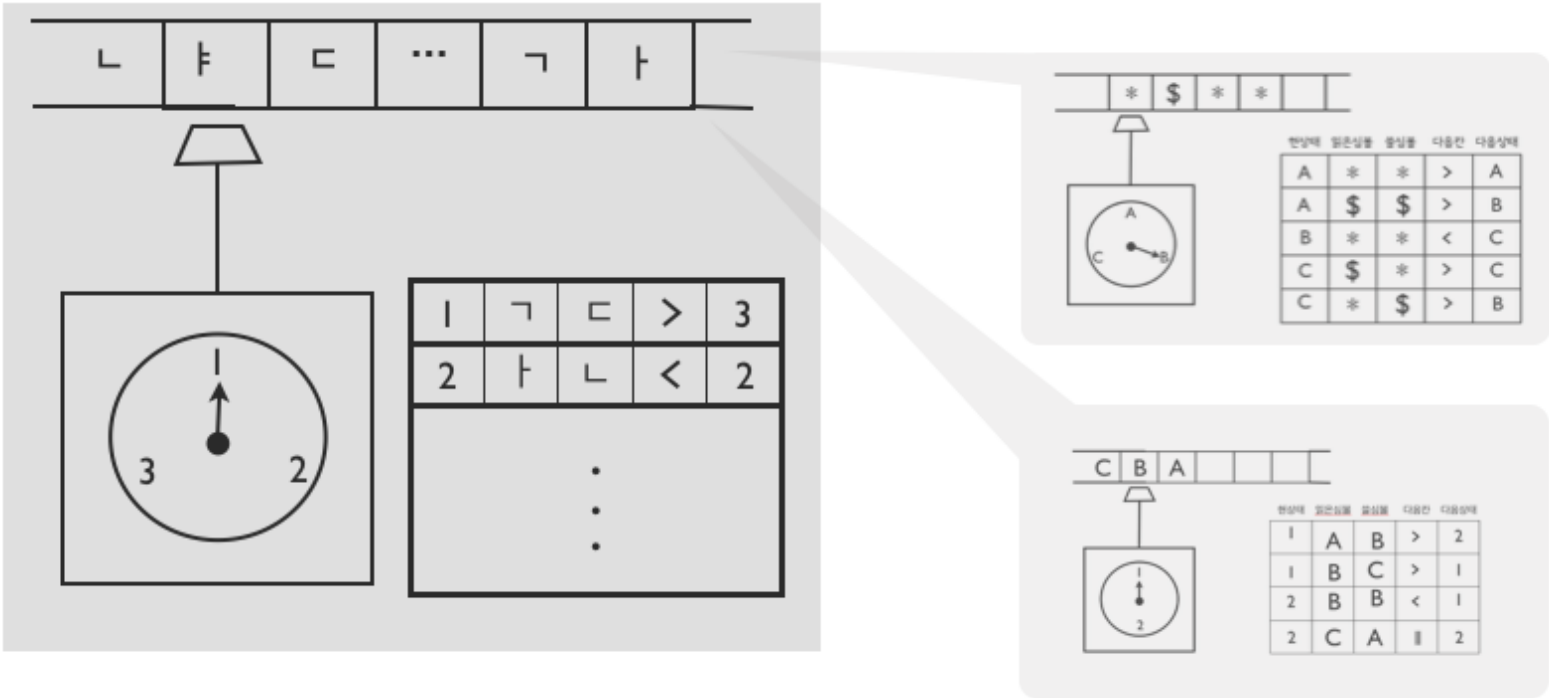


테입[ㅌ1]ㅌ2ㅌ1ㅌ1 상태ㅅ2 규칙ㅅ1ㅌ1ㅌ1>ㅅ1 규칙ㅅ1ㅌ2ㅌ2<ㅅ2 ...



컴퓨터 = 만능 튜링기계(universal machine)

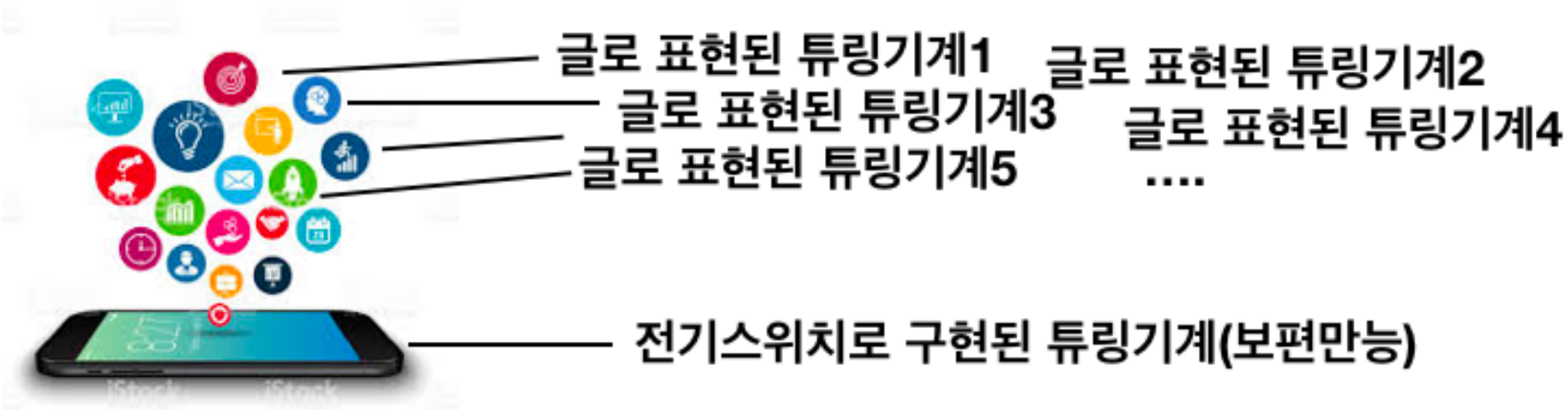
- 사용법: 임의의 튜링기계(소프트웨어) 테잎에 싣기
- 자동작동: 테잎에 있는 튜링기계 그대로 따라하기(실행)



(사진출처: Google)

컴퓨터 = 만능 튜링기계(universal machine)

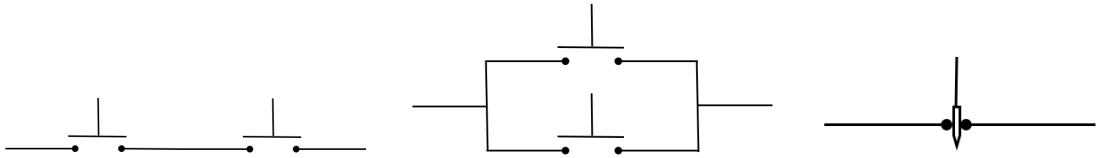
- 사용법: 임의의 튜링기계(소프트웨어) 테잎에 싣기
- 자동작동: 테잎에 있는 튜링기계 그대로 따라하기(실행)



(사진출처: Google)

언어기원 I: 튜링기계(Turing Machine)

- 컴퓨터의 실현 = 만능튜링기계가 전기스위치로 구현됨



- 소프트웨어/프로그램 짜기 = 튜링기계 만들기
- 프로그래밍 언어 = 명령형/기계중심/물건중심

개념모델	튜링기계 (and/or/not 스위치) ↓
상위	메모리와 cpu (+, -, move, store, jmp, etc.) ↓
상위	명령형언어(C, Java, etc.)

언어기원 I: 튜링기계(Turing Machine)

기계중심 프로그래밍

- 이름짓기 대상: 기계부품(메모리), etc.
- 프로그램실행 = 기계상태(메모리)/물건(메모리에 구현된)의 변화과정

언어기원 I: 튜링기계(Turing Machine)

기계중심 프로그래밍

- 이름짓기 대상: 기계부품(메모리), etc.
- 프로그램실행 = 기계상태(메모리)/물건(메모리에 구현된)의 변화과정

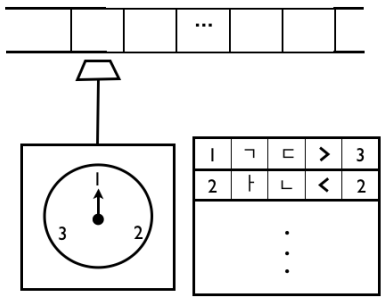
```
a = 1;
b = a+1;
if (a == b) then c = a+1 else c = a-1;
foo(a,b);
while (c < 100) a = a+b; c = c+1;
```

언어기원 I: 튜링기계(Turing Machine)

기계중심 프로그래밍

- 이름짓기 대상: 기계부품(메모리), etc.
- 프로그램실행 = 기계상태(메모리)/물건(메모리에 구현된)의 변화과정

```
a = 1;
b = a+1;
if (a == b) then c = a+1 else c
foo(a,b);
while (c < 100) a = a+b; c = c+
```



언어기원 I: 튜링기계(Turing Machine)

기계중심 프로그래밍

- 이름짓기 대상: 기계부품(메모리), etc.
- 프로그램실행 = 기계상태(메모리)/물건(메모리에 구현된)의 변화과정
- 그러나, 같은 이름이 그때그때 달라요
  - 변화무쌍
  - 오류잡기 어려움

```
a = 1;
a = a + 1;
b = a + 1;
```

언어기원 II: 람다계산법(Lambda Calculus)

- 기계적인 계산의 또 다른 정의
- 튜링기계와 표현력은 동일, 그러나
- "기계"(변화)를 염두에 둘 필요가 없는 과정
- 오직 값의 계산과정만 있는 과정

언어기원 II: 람다계산법(Lambda Calculus)

계산중심/값중심 프로그래밍

함수중심 프로그래밍

- 다양한 종류의 값
  - 기본(primitive): 숫자, 글자 등
  - 복합(compound): 짝, 함수 등
  - 모두가 자유롭게 다뤄지는 값
- 불변
  - 값은 변하지 않음
  - 있던 값으로 새 값이 계산되고, 예전 값이 변하는 것은 아니고

$$a + b, S \cup \{1\}$$

람다 계산법(lambda calculus)

- "기계적인"의 정의 = "람다식으로 계산할 수 있는" (1935년)
- 컴퓨터에 실리는 프로그램 하나 = 람다식 하나

람다식 E	는	x	이름
	또는	$\lambda x.E$	함수정의: 인자 x, 내용 E
	또는	$E\ E$	함수적용

람다 계산법(lambda calculus)

$$
\begin{aligned}
\underline{3} &= \lambda s.(\lambda z.\, \underbrace{s(s(s\, z))}_{3}) \\
\underline{+}\, n\, m &= \lambda s.(\lambda z.ns(msz)) \\
\underline{\times}\, n\, m &= \lambda s.(\lambda z.n(ms)z) \\
\underline{T} &= \lambda x.(\lambda y.x) \\
\underline{F} &= \lambda x.(\lambda y.y) \\
\underline{and}\, a\, b &= a\, b\, \underline{F} \\
\underline{\text{zero?}}\, n &= n\, (\lambda x.\underline{F})\, \underline{T} \\
\underline{\text{repeat}}\, E &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\, E
\end{aligned}
$$

언어기원 II: 람다 계산법(Lambda Calculus)

개념모델	튜링기계 (and/or/not 스위치) ↓	람다식 (var/lam/app 식) ↓
상위	메모리와 cpu (+, -, move, store, jmp, etc.) ↓	계산식 (+, -, 짝, 함수, 재귀, if, etc.) ↓
상위	명령형언어(C, Java)	계산형언어(OCaml, Haskell)

차이점: 기계중심 vs 계산중심 프로그래밍 (1/6)

```
a = 1;
b = 2;
c = a + b;
d = a + b;
```

- c와 d가 같은 것인가?
- a를 바꾸면 c도 바뀔까?
- d를 바꾸면 c도 바뀔까?
- e = c를 수행하려면 c를 복사해야 하는가?
- c 가지고 일 봤으면, 그 것을 없애도 되는가?

차이점: 기계중심 vs 계산중심 프로그래밍 (2/6)

```
a = set(1,2,3);
b = set(4,5,6);
c = setUnion(a,b);
d = setunion(a,b);
```

- c와 d가 같은 것인가?
- a를 바꾸면 c도 바뀔까?
- d를 바꾸면 c도 바뀔까?
- e = c를 수행하려면 c를 복사해야 하는가?
- c 가지고 일 봤으면, 그 것을 없애도 되는가?

차이점: 기계중심 vs 계산중심 프로그래밍 (3/6)

변하는 집합(물건) vs 불변하는 집합(값)

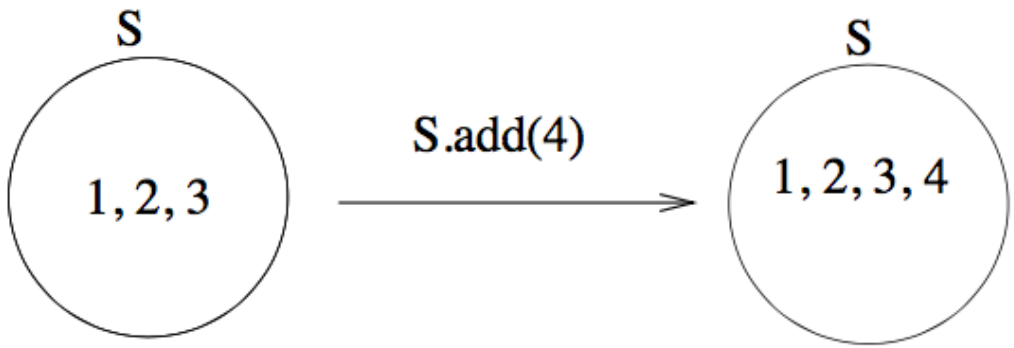
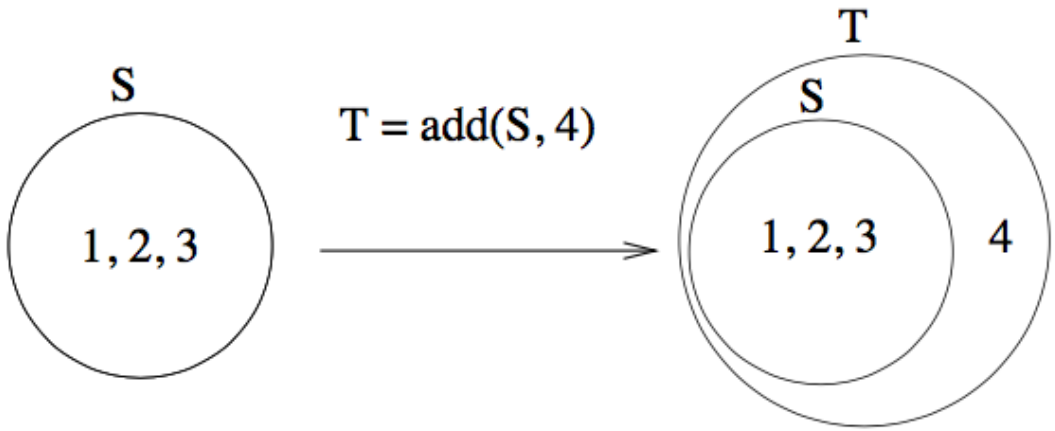


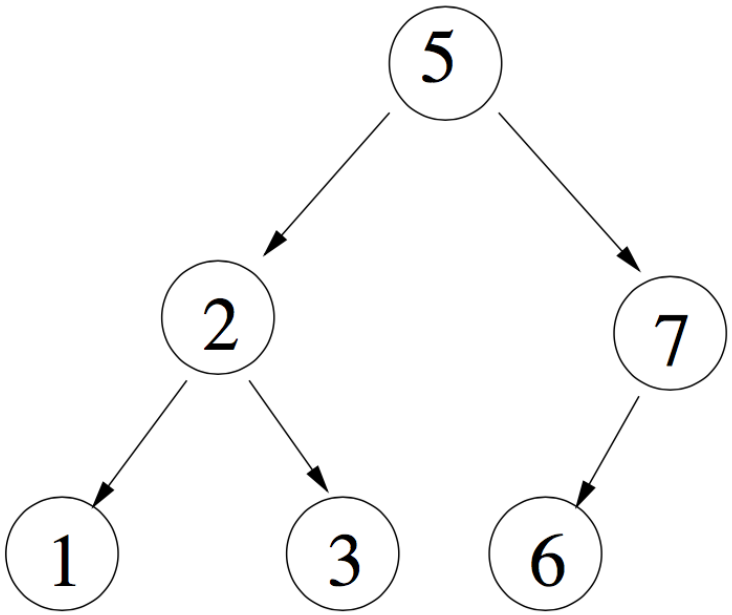
그림 5: 집합이라는 물건은 변한다



차이점: 기계중심 vs 계산중심 프로그래밍 (4/6)

변하는 집합(물건) vs 불변하는 집합(값)

집합 $\{1, 2, 3, 5, 6, 7\}$의 구현

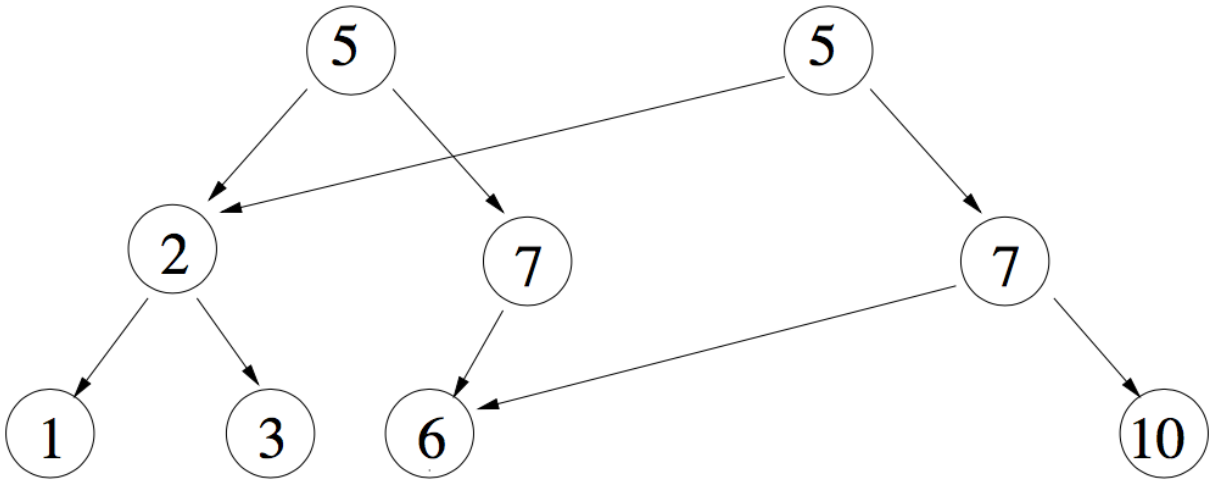


차이점: 기계중심 vs 계산중심 프로그래밍 (5/6)

변하는 집합(물건) vs 불변하는 집합(값)

$\{1, 2, 3, 5, 6, 7\} \cup \{10\}$ 의 구현

불변하면 맘껏 공유가능 vs 변하면 무조건 복사해야

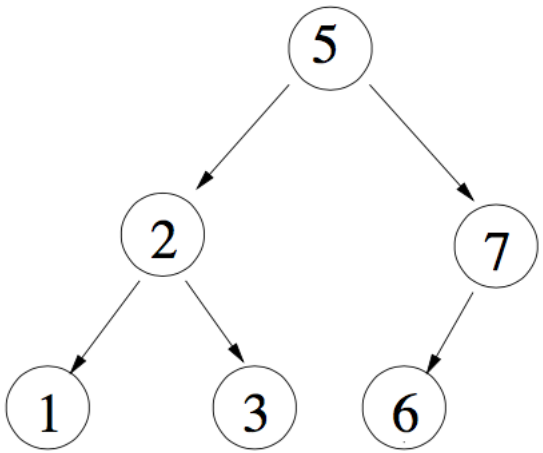


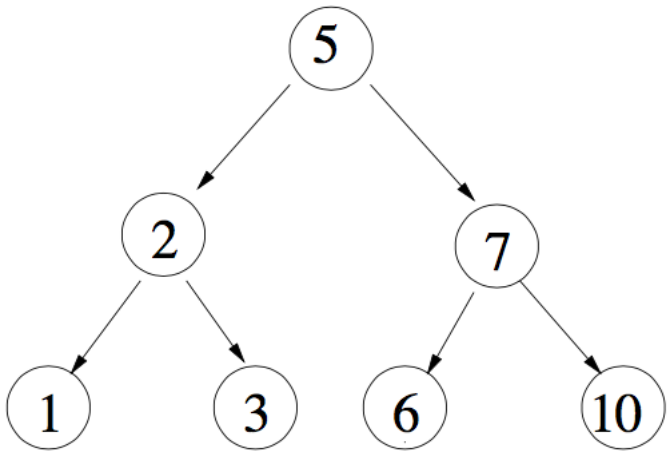
차이점: 기계중심 vs 계산중심 프로그래밍 (6/6)

변하는 집합(물건) vs 불변하는 집합(값)

$\{1, 2, 3, 5, 6, 7\} \cup \{10\}$ 의 구현

불변하면 맘껏 공유가능 vs 변하면 무조건 복사해야





정리: 프로그래밍 언어의 두 뿌리, 중력권, 열매들

튜링기계(Turing Machine)	람다계산법(Lambda Calculus)
명령형(imperative)	계산형(applicative)
물건중심(object-oriented)	값중심(value-oriented)
기계중심	값중심
기계작동 과정	계산 과정
상태가 변하는 과정	값이 계산되는 과정
	안전/자동/2세대 오류검증기술
C, C++, Java, JS, Rust, etc.	OCaml, Haskell, etc.

Scala, Python, C++18 etc.

프로그래밍 언어의 두 기원

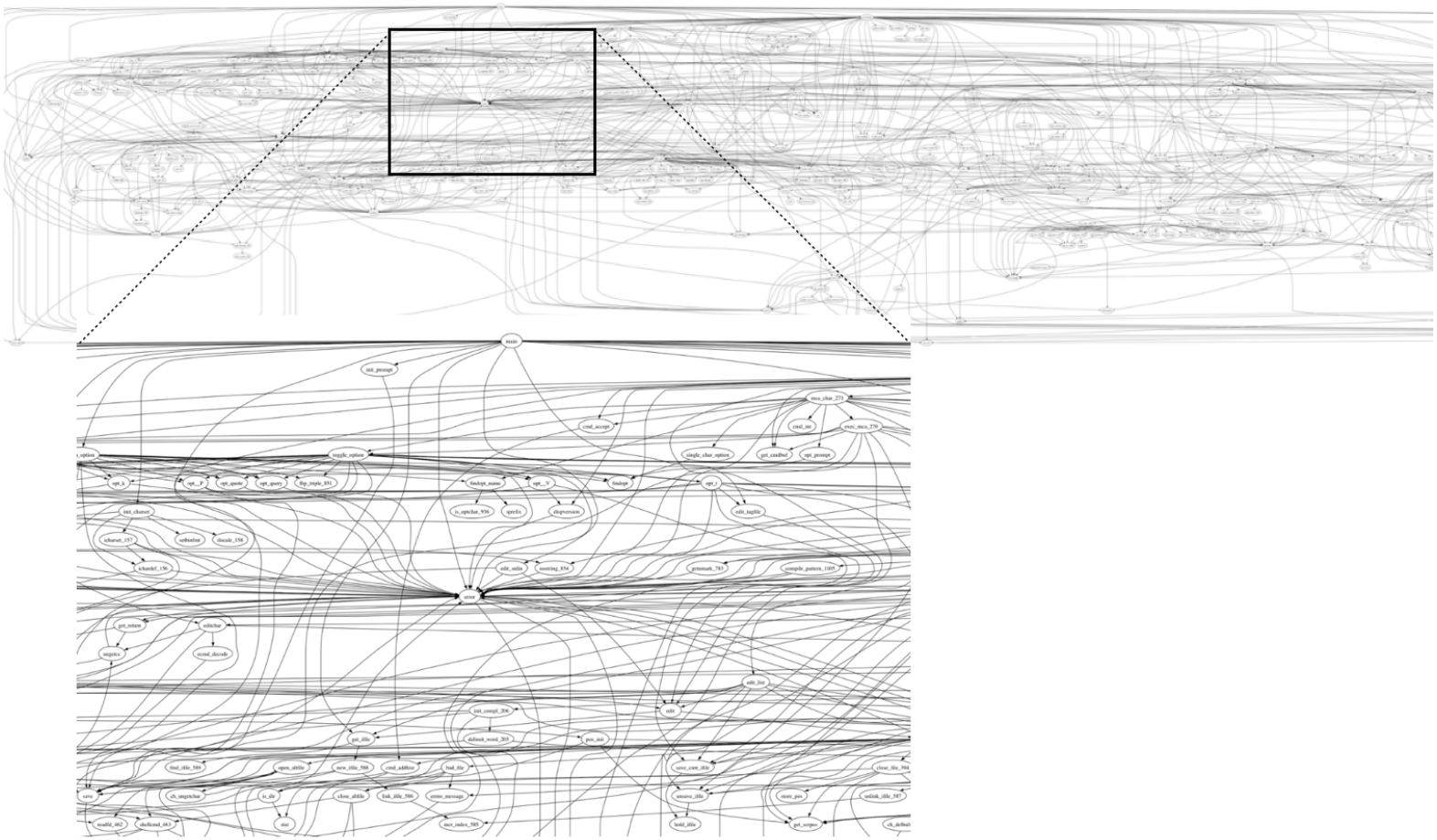
프로그램 오류 자동 검증: 타입 체킹

프로그램 실행 자원 자동 관리: 메모리 재활용

프로그래밍에 동원되는 생각방식 14선

컴퓨터 프로그램은 복잡하다

less-382(23,822 LoC)



컴퓨터 프로그램 제작의 어려움

- 프로그램은 기계가 자동으로 실행함
  - 기계는 우리가 바라는 바를 실행하지 않음
  - 기계는 sw에 적힌 바를 실행할 뿐
  - 모든 상황을 고려해야 × 사소한 실수가 없어야
- 프로그램의 실행을 "미리 정확히 알기"가 어려움
  - 모든 프로그램을 미리 정확히 자동으로는 불가능
  - 사람이 확인해가야: 다양한 도구로 비용절감
  - 현재의 기술수준: 걸음마 "3발짝", 초보수준

고백: 컴퓨터분야는 아직 미숙합니다

분야의 역사 $\sim$ 겨우 5-60년

그래서

- 미숙하기 때문에 $\Longrightarrow$ 기회가 많다
  - 많은 흥미로운 연구기회들: 불만/비평 시각만으로
  - 많은 치고나갈 사업기회들: 좋은 연구성과 선구안만으로 [1]
- 미숙하기 때문에 $\Longrightarrow$ 지금 “Newton”, “Galileo”, “Curie”, 와 같이 살고있는 수준

팩첵트: 컴퓨터분야 발전 속도가 빠르다? (1/3)

- 컴퓨터속도: 10^7배/50년 발전

10^3flops/cpu ENIAC(1945)

$\longrightarrow 2 \times 10^{10}$flops/core IBM Sequoia(2010)

팩첵트: 컴퓨터분야 발전 속도가 빠르다? (2/3)

다른 분야와 얼추 비슷: $10^7 \sim 10^9$배 발전/100년

- 에너지분야: 10^7배/100년 발전

  10^{-1}hp/hr (하인) $\longrightarrow$ 1.35×10^6 hp/hr (원전 1GW/hr)

- 교통분야: 10^9배/100년 발전

  10^2j (가마) $\longrightarrow$ 10^{11}j (Delta II)

팩첵트: 컴퓨터분야 발전 속도가 빠르다? (3/3)

- 하드웨어: 다른 분야와 비슷한 발전
- 소프트웨어: 다른 분야가 이루어 놓은 것을 아직 이루지 못했슴

무엇일까?

다른 분야의 현재 수준

- 제대로 작동할 지를 미리 검증할 수 없는 기계 설계는 없다.
- 제대로 작동할 지를 미리 검증할 수 없는 회로 설계는 없다.
- 제대로 작동할 지를 미리 검증할 수 없는 공장 설계는 없다.
- 제대로 서있을 지를 미리 검증할 수 없는 건축 설계는 없다.

뉴튼방정식, 미적분 방정식, 통계역학, 무슨무슨 방정식...

모든 기술의 질문

우리가 만든 것이
우리가 의도한대로 움직인다는 것을
어떻게 미리
확인할 수 있는가?

(일상에서도 잦은 질문과 답: 입학시험, 입사시험, 궁합, 클럽관리)

프로그램 분야에서의 이 질문

작성한 프로그램의 오류를
자동으로 미리 모두 찾아주거나,
없으면 없다고 확인해주는
기술들은 있는가?

그래서, 프로그램의 오류때문에 발생하는
개인/기업/국가/사회적 비용을
절감시켜주는 기술들은 있는가?

프로그램 오류(bug)

- 프로그램에 있는 오류
- 프로그램이 생각대로 실행되지 않는 것
- 사람이 프로그램을 잘못 만들었기 때문
  - 글 쓴 나의 실수
  - 천재지변이 아님

프로그램 오류의 예

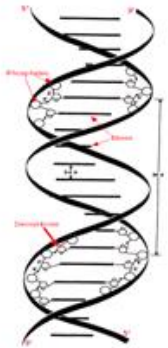
어머니가 전해준 슈퍼마켓 심부름 목록(프로그램)

1. 우유 1리터 2병
2. 우유가 없으면 오렌지쥬스 1리터 3병
3. 우유가 있으면 식빵 500그람 1봉
4. 쌀과 자두

만약에: 우유 1리터 1병만 있으면? 쌀과 자두? 쌀과자두?

프로그램 오류의 문제

DNA 오류



SW 오류

⇓

살다가 병이든다

사회적 비용

해결책 = 경제적기회

⇓

SW로 동작하는 제품이 고장난다

사회적 비용

해결책 = 경제적기회

프로그램 오류 검증 기술: 1세대

문법 검증기: 70년대에 완성된 기술. lexical analyzer & parser

오류 = 프로그램의 생김새가 틀린 것

1. 우유 1리터 2병
2. 우유가 없으면 오렌지ㅈㅅ 1리터 3병
3. 있으면 빵식 우유가 500그람 1봉
4. 쌀과 자두

프로그램 오류 검증 기술: 2세대. 타입체킹(type checking)

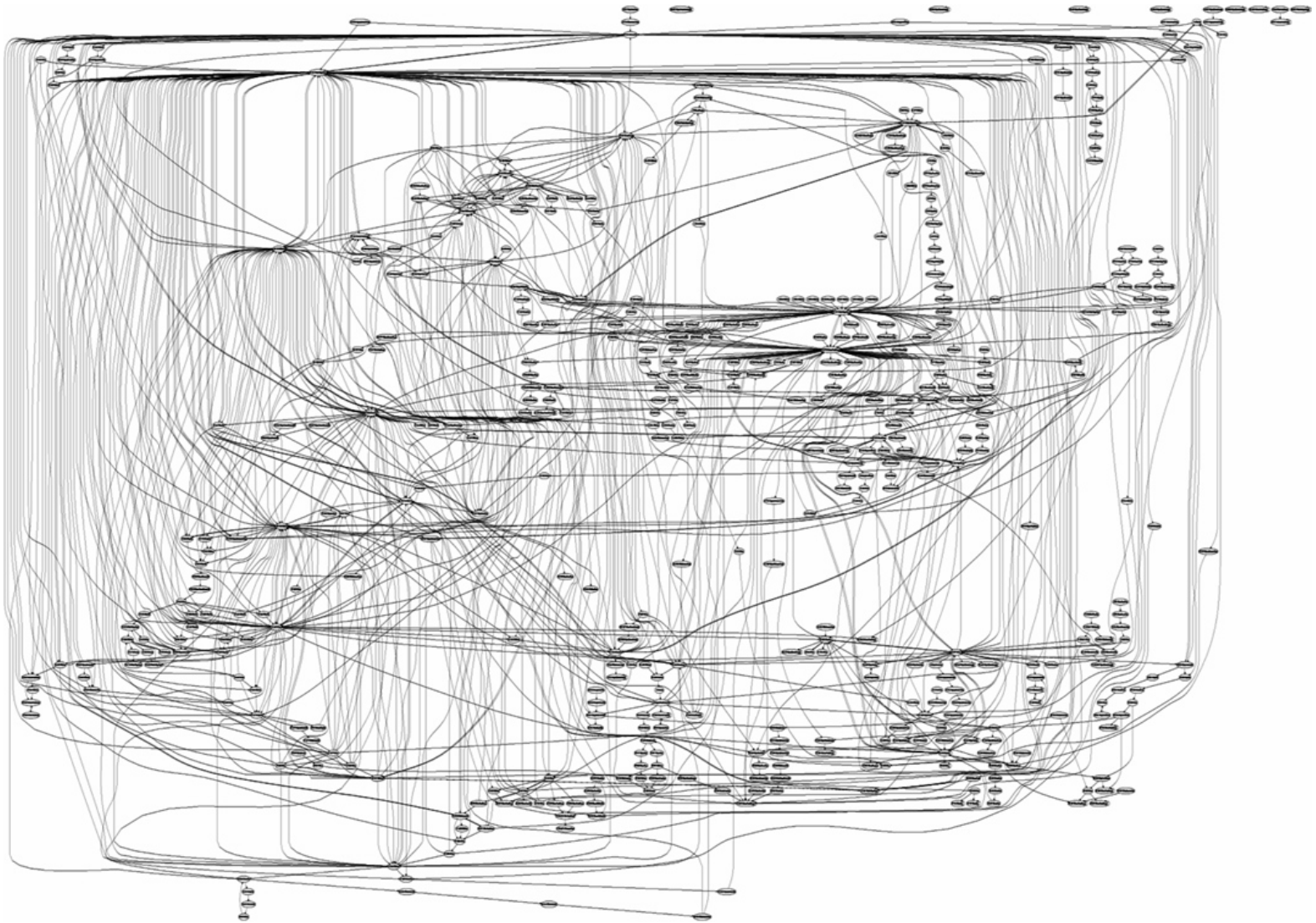
90년대에 완성.

(30년간 프로그래밍언어 분야의 대표 성과)

오류 = 생긴것은 멀쩡한데, 잘못된 종류의 값이 계산에 흘러드는 경우

1. 우유를 담은 얼음을 가스 불위에 올려놓고 데운다
2. 식빵을 유리접시에 담고 방아로 빻는다
3. 2의 접시에 1의 우유를 튀긴다
4. 남자와 소나무를 결혼시킨다

오류가능성: 잘못된 종류값이 흘러들 수 있는



프로그램 오류 검증 기술: 3세대

프로그램분석검증: 아직 미완성

(static analysis, verification, model checking, ...)

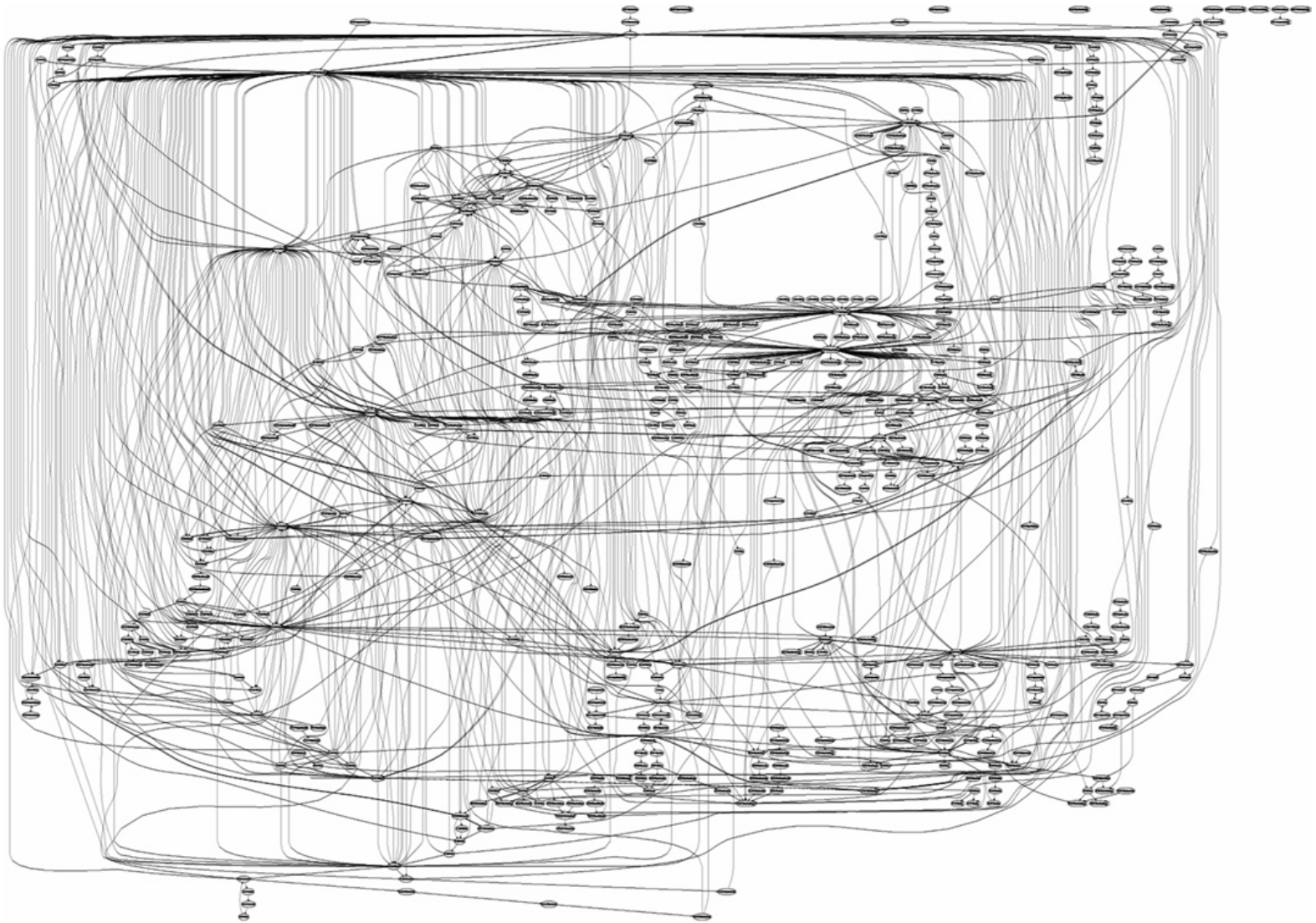
오류 = 생긴것도 멀쩡하고 타입도 맞지만, 바라는 바가 아닌 것

오류 = 필요한 조건을 만족시킬 수 없는 경우

1. 우유를 담은 냄비를 가스 불위에 데운다
2. 식빵을 스텐접시에 담고 방아로 빻는다
3. 남자1명과 여자1명을 결혼 시킨다

- 가스불이 포항제철 용광로가 된다면?
- 방아가 현대중공업의 프레스가 된다면?
- 남녀 나이의 차가 100살이된다면?

오류가능성: 잘못된 크기 값이 흘러들 수 있는



타입체킹(type checking): 람다 중력권의 혜택

- 람다 중력권의 혜택: 논리는 언어의 거울
- 거울

논리 세계		프로그래밍 세계
증명하기	$\longleftrightarrow$	프로그램짜기
증명된 결론	$\longleftrightarrow$	프로그램의 타입
증명 오류	$\longleftrightarrow$	프로그램 타입오류
증명 검산	$\longleftrightarrow$	프로그램 타입오류 검산
증명 빈칸메꾸기	$\longleftrightarrow$	프로그램 타입유추

- 거울의 효능
  - 자동이고, 믿어도 되고, 귀찮게 안하는 타입체킹(type checking)
  - "2세대 오류검증 기술"

람다 중력권의 혜택: 논리는 언어의 거울

람다 중력권 프로그래밍 세계에서 언어와 논리는 동전의 양면

증명하기	⟷	프로그램짜기
"논리적인 비약없이 새로운 사실을 확인해가는 과정."	⟷	공짜없이 새로운 데이터를 만들어가는 과정.
"참인 사실 혹은 사실이라고 가정한 것들로부터 시작."	⟷	기초적인 데이터에서 부터 시작. 기초적인 데이터는 정수, 문자, 참거짓 등.
"사실을 기반으로 해서 새로운 사실들을 만듬."	⟷	이미 만든 데이터를 가지고 새로운 데이터들을 만듬.
"만들어가는 과정은 근거없는 건너뛰기 없이, 논리적으로 누구나 수긍하는 추론의 징검다리를 밟고 가는 과정만 있다."	⟷	새 데이터를 만드는 과정은 공짜가 없다. 사용하는 프로그래밍 언어에서 제공하는 프로그램 조립방식을 써서 만든다.

논리 추론의 징검다리

$$\frac{A \quad B}{A \wedge B} \qquad \frac{A \wedge B}{A} \qquad \frac{A \wedge B}{B}$$

$$\frac{\begin{array}{c}\overline{A} \\ \vdots \\ B\end{array}}{A \Rightarrow B} \qquad \frac{A \Rightarrow B \quad A}{B}$$

$$\frac{A}{A \vee B} \qquad \frac{A}{B \vee A} \qquad \frac{A \vee B \quad \begin{array}{c}\overline{A} \\ \vdots \\ C\end{array} \quad \begin{array}{c}\overline{B} \\ \vdots \\ C\end{array}}{C}$$

논리 증명나무

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (A \Rightarrow C)}}{A \Rightarrow B} \quad \dfrac{}{A}}{B} \quad \dfrac{\dfrac{\dfrac{}{(A \Rightarrow B) \wedge (A \Rightarrow C)}}{A \Rightarrow C} \quad \dfrac{}{A}}{C}}{B \wedge C}}{A \Rightarrow (B \wedge C)}$$

$$(A \Rightarrow B) \wedge (A \Rightarrow C) \Rightarrow (A \Rightarrow (B \wedge C))$$

거울: 증명하기 = 프로그램짜기 (1/2)

$$\frac{A \quad B}{A \wedge B} \longleftrightarrow \text{데이터 뭉치 만들기}$$

$$\frac{A \wedge B}{A} \quad \frac{A \wedge B}{B} \longleftrightarrow \text{데이터 뭉치 사용하기}$$

$$\frac{\begin{array}{c}\overline{A} \\ \vdots \\ B\end{array}}{A \Rightarrow B} \longleftrightarrow \text{함수 만들기}$$

$$\frac{A \Rightarrow B \quad A}{B} \longleftrightarrow \text{함수 사용하기}$$

거울: 증명하기 = 프로그램짜기 (2/2)

$$\frac{A}{A \vee B} \qquad \frac{A}{B \vee A} \quad \longleftrightarrow \quad \text{데이터 뭉뚱그리기}$$

$$\frac{A \vee B \qquad \begin{array}{c}\overline{A}\\ \vdots\\ C\end{array} \qquad \begin{array}{c}\overline{B}\\ \vdots\\ C\end{array}}{C} \quad \longleftrightarrow \quad \text{뭉뚱그린 데이터 사용하기}$$

$$\overline{A} \quad \longleftrightarrow \quad \text{함수의 인자를 사용하기}$$

짠 프로그램이 무난히 작동할지 검진하기 (1/2)

프로그래머의 불안

- 제대로 작동하는 프로그램일까?
  - 타입에 맞게 실행중에 잘 작동할까? (2세대 오류)
- 복잡한 프로그램 실행 과정과 환경. 예상못한 타입오류.
- 프로그램 오류는 고스란히 드러나리라.

거울의 효능

- 프로그램 짜고나서: 짠 프로그램이 무난히 작동할지 살피기
- 무난히 작동 = 타입에 맞게 돈다
- 거울

논리 세계		프로그래밍 세계
증명하기	$\longleftrightarrow$	프로그램짜기
증명된 결론	$\longleftrightarrow$	프로그램의 타입
증명 오류	$\longleftrightarrow$	프로그램 타입오류
증명 검산	$\longleftrightarrow$	프로그램 타입오류 검산
증명 빈칸메꾸기	$\longleftrightarrow$	프로그램 타입유추

- 논리분야(거울 저 쪽)의 성과들
- 프로그램(거울 이쪽)에서도 가능하겠지
  - 아니라: 기계중심/물건중심/상태변화/변화무쌍의 프로그래밍 중력권에서가
  - 가능하려나보다!: 값계산만 있는 세계, 상위의 프로그래밍 중력권에서는

프로그램 검진 vs 다른 분야

만든것	프로그램	$\longleftrightarrow$	기계/건물/약물 디자인
실행기	컴퓨터	$\longleftrightarrow$	자연
바램	실행에대해 미리 확인	$\longleftrightarrow$	작동에대해 미리 확인
안심	“생각대로 돌것이다”	$\longleftrightarrow$	“생각대로 작동할것이다”
확인도구	컴퓨터과학의 성과	$\longleftrightarrow$	자연과학의 성과

짠 프로그램이 무난히 작동할지 검진하기 (2/2)

자동검진이 가능

- 과거: 개인의 기예에 의존하던 프로그램의 완성도
- 현재, 미래: 누구나의 기술로 한 발 전진
  - 자동이고, 믿어도 되고, 귀찮게 안하고
- 컴퓨터는 더 이상 맹목적이지 않다
- 프로그램을 검진해서 타입체킹 통과한 것만 실행

자동이고/믿어도 되고/귀찮게 안하는 타입체킹(1/6)

- 귀찮게 안하는

```
f(x) = x+1      v.s.      f(x: int): int = x+1
```

- 자동으로 타입을 유추한다(사람같이)

```
f(x) = x+1
sum(f) = f(0)+f(1)
weigh(춘향이)+ weigh(그네)
```

자동이고/믿어도 되고/귀찮게 안하는 타입체킹(2/6)

타입유추 = 연립방정식 세우고 풀기

- `f(x) = x+1`

  f의 타입 연립방정식?
- `f(1) + f(true)`

  위 식의 타입 연립방정식?
- `sum(f) = f(0)+f(1)`

  sum의 타입 연립방정식?
- `sum(lam x.x+1) + sum(lam x.block x)`

  위 식의 타입 연립방정식?
- `weigh(l) = if l=empty then 0 else 1+weigh(rest l)`

  weigh의 타입 연립방정식?
- `weigh(춘향이)+ weigh(그네)`

  위 식의 타입 연립방정식?

자동이고/믿어도 되고/귀찮게 안하는 타입체킹(3/6)

- 자동이고(자동으로 타입을 유추하며 체크하는 알고리즘이 고안됨)

증명됨: 알고리즘이 해를 찾아낸다 $\iff$ 타입 연립방정식의 해가 있다

- 믿어도 되고

증명됨: 타입 연립방정식의 해가 있다 $\implies$ 타입오류 없이 실행된다

- 아쉬움: 깐깐한 면이 있어요

증명됨: 타입 연립방정식의 해가 있다 $\not\impliedby$ 타입오류 없이 실행된다

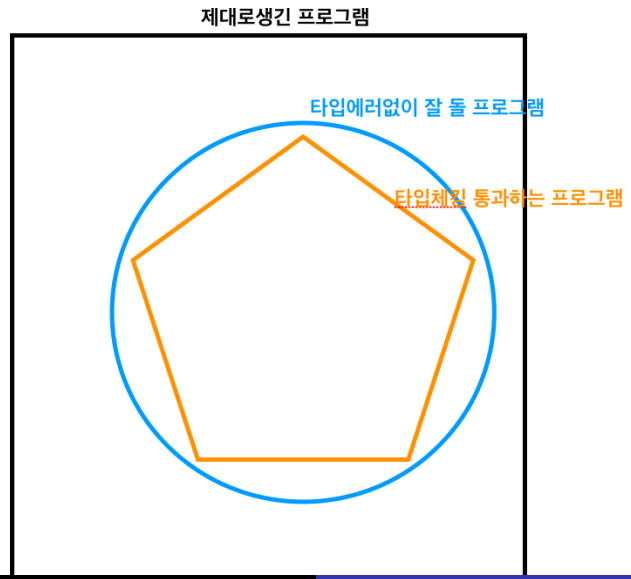
자동이고/믿어도 되고/귀찮게 안하는 타입체킹(4/6)

깐깐한 면이 있는게 아쉬워요:

안전(sound)하지만 완전(complete)하지는 않아요

안전: 타입 연립방정식의 해가 있다 $\Longrightarrow$ 타입오류 없이 실행된다

불완전: 타입 연립방정식의 해가 있다 $\not\Longleftarrow$ 타입오류 없이 실행된다



자동이고/믿어도 되고/귀찮게 안하는 타입체킹(5/6)

- 타입 연립방정식의 해가 있으면

  증명됨: 타입 연립방정식의 해가 있다 $\Longrightarrow$ 타입오류 없이 실행된다

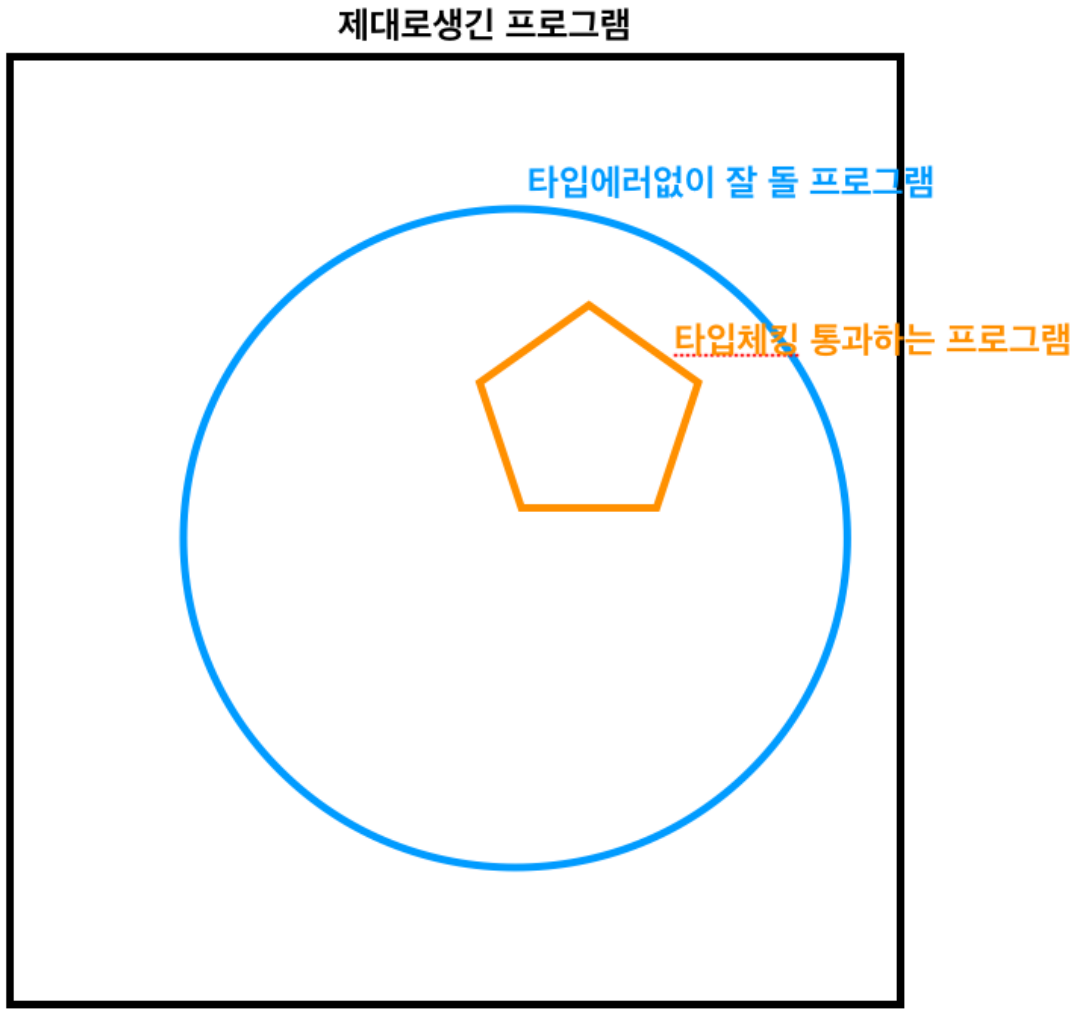
  - "문제없네요. 맘놓고 실행하세요."
  - "제 말 믿어도되요" (안전)
- 타입 연립방정식의 해가 없으면

  증명됨: 타입 연립방정식의 해가 있다 $\not\Longleftarrow$ 타입오류 없이 실행된다

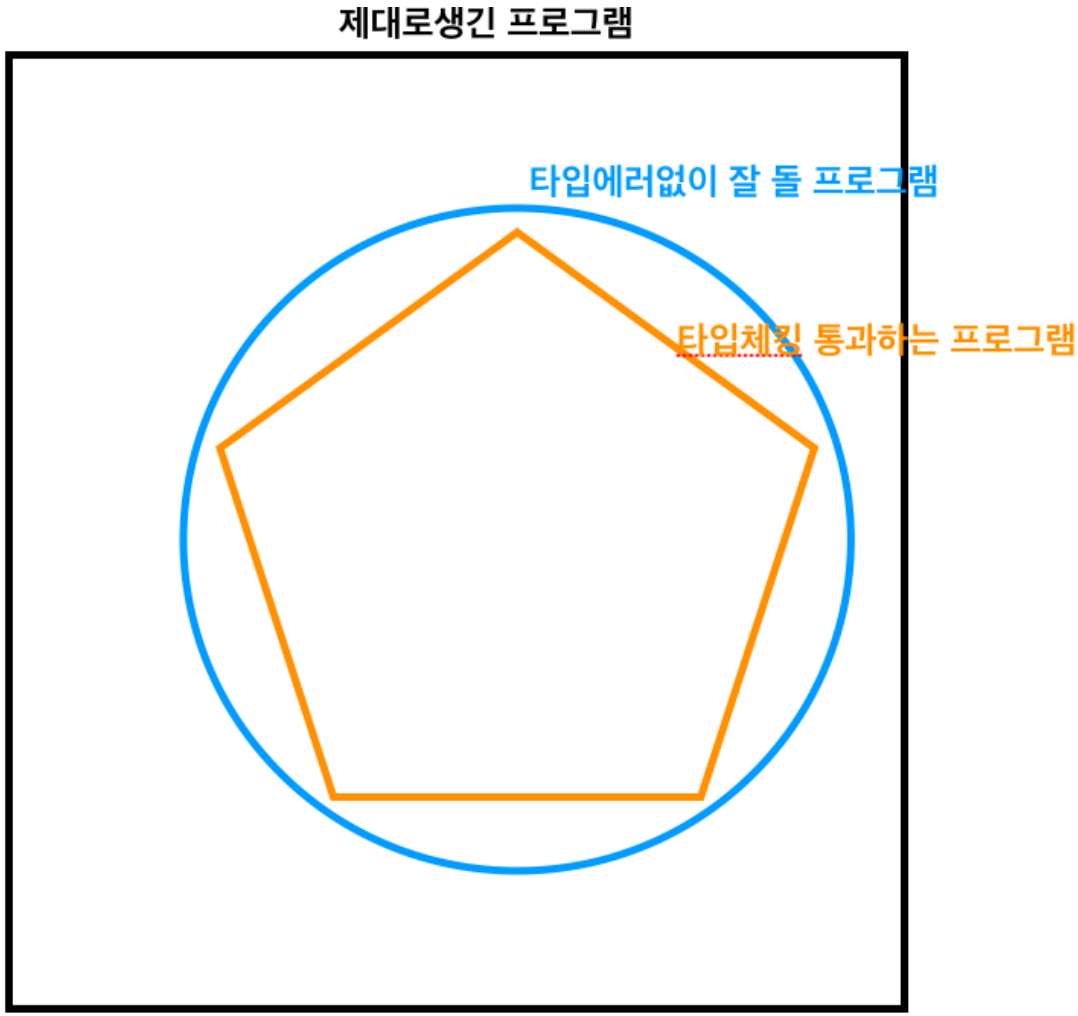
  - "여기 타입오류가 있는 것 같아요."
  - 하지만, 잘 도는 프로그램일 수 있다 (불완전)
  - 그런데, 그런 경우는 드물고, 쉽게 피할 수 있다

```
some(f) = f(1) + f(true); some(lam x.1);
```

믿어도되지만 깐깐한 경우가 적은 타입체킹(6/6)



믿어도되지만 깐깐한 경우가 적은 타입체킹(6/6)



타입 시스템의 정교한 정도

- 자동이고 믿어도되면서 깐깐한 정도가 덜해야
- let-다형타입시스템(let-polymorphic type system) (OCaml)
  - 함수정의를 보고, 인자타입을 몰라도 되네?
  - 그렇다면, 그 함수는 다형함수(polymorphic function).

```
I(x) = 1
length(l) = if l=empty then 0 else 1 + length(rest l)
```

- 왜 "let-다형"?
  - 함수정의를 볼때만 다형일 수 있는지 유추
- 왜 그때만?
  - 다른 때도 그러면(너무 덤비면) 믿을 수 없게됨

정리: 프로그램 오류 자동 검증, 타입 체킹

- 계산형/값중심 프로그래밍언어에 장착된
- 2세대 오류(타입오류) 잡는 자동 기술
- 자동으로/믿어도 되고/귀찮게 안하고
- 안전하지만 완전하진 않음
  - 믿어도 됨, 하지만 깐깐한 면이. 그러나 드뭄.
- 성과배경: 람다중력권에서 보게된 거울(증명 = 프로그램)
- 아직 2세대 그러나 든든한 성과: OCaml의 let-다형타입시스템

프로그래밍 언어의 두 기원

프로그램 오류 자동 검증: 타입 체킹

프로그램 실행 자원 자동 관리: 메모리 재활용

프로그래밍에 동원되는 생각방식 14선

프로그램 실행 자원 두 가지

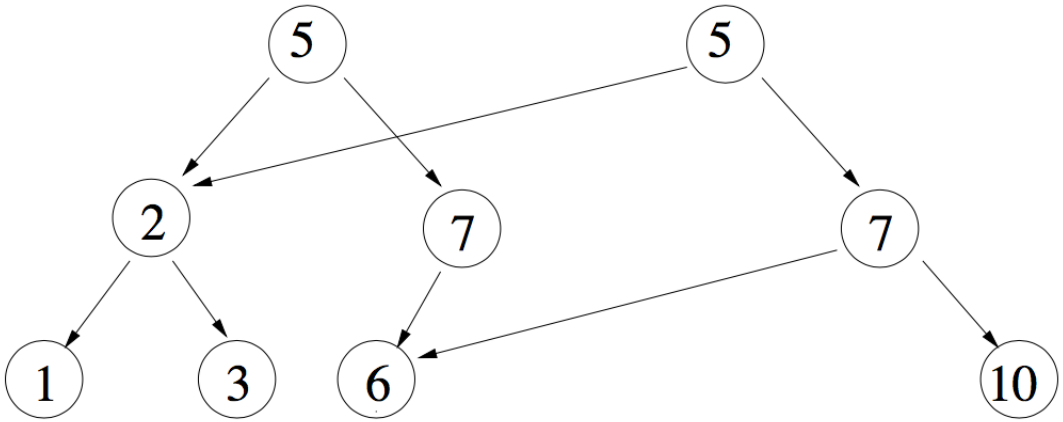
프로그램 실행은 시간과 메모리를 쓴다

- 시간
  - 무한: 그러나 한없이 쓸 수는 없다
- 메모리
  - 유한: 그래서 한없이 쓸 수 없다

프로그램 실행에 필요한 메모리(1/2)

- 값들을 구현하는 데 필요

```
a = set(1,2,3,5,6,7);
b = set(10);
c = setUnion(a,b);
d = setunion(a,b);
```



프로그램 실행에 필요한 메모리(2/2)

- 값 구현뿐 아니라, 기타 계산과정에서도 필요

$$
\begin{aligned}
&\texttt{fac(3)}\\
&= \boxed{3\times}\ \texttt{fac(2)}\\
&\qquad = \boxed{2\times}\ \texttt{fac(1)}\\
&\qquad\qquad = \boxed{1\times}\ \texttt{fac(0)}\\
&\qquad\qquad\qquad\ \ 1\\
&\qquad\qquad = 1\\
&\qquad = 2\\
&= 6
\end{aligned}
$$

메모리를 재활용해야

```
while (true)
  ... setUnion(a,b) ...

loop (1000000)
  ... link(chain, block) ...
```

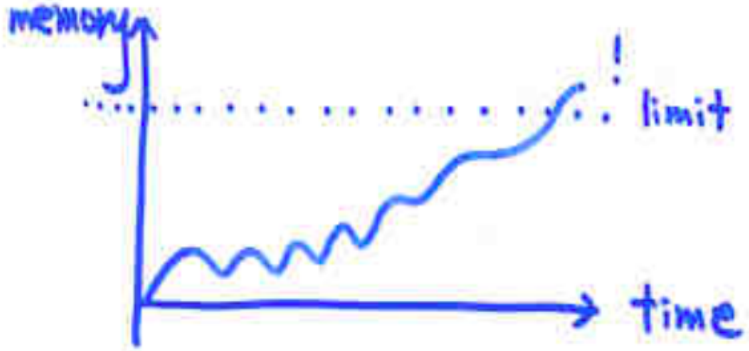
- 필요한 새 메모리가 늘 있을 수 없다!
- 답: 재활용

메모리 재활용을 누가하나? (1/3)

프로그래머에게 맡기자

- 그런데, 너무 어렵다
  - 너무 빨리해도 문제: "메모리 미아(dangling pointer)"

  - 너무 늦게해도 문제: "메모리 누수(memory leak)"



- 1990년대 초까지. 다른 대안(자동 재활용)기술이 덜 성숙.

메모리 재활용을 누가하나? (2/3)

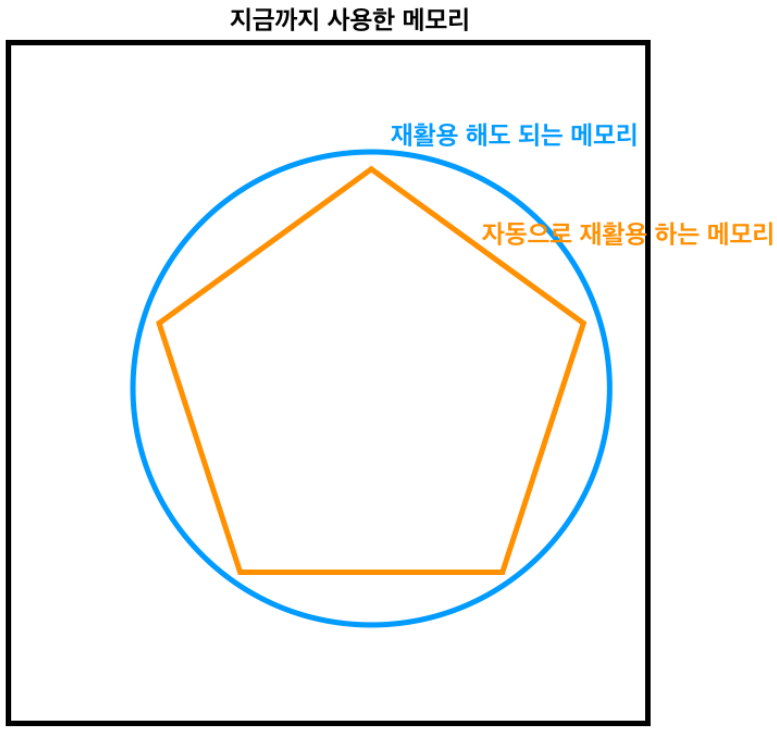
자동으로 하자

- 프로그램 실행중에 메모리 자동 재활용(garbage collection)
  - 실행중에 메모리 사용이 어느 수위를 넘기면 실행을 중단하고
  - 지금까지 사용한 메모리를 살펴서
  - 앞으로 사용하지 않을 메모리를 재활용
- 정확한 재활용은 불가능
  - 지금까지 사용한 메모리중에 미래에 사용하지 않을 메모리?
  - 정확하게 알아내는 프로그램은 불가능

메모리 재활용을 누가하나? (3/3)

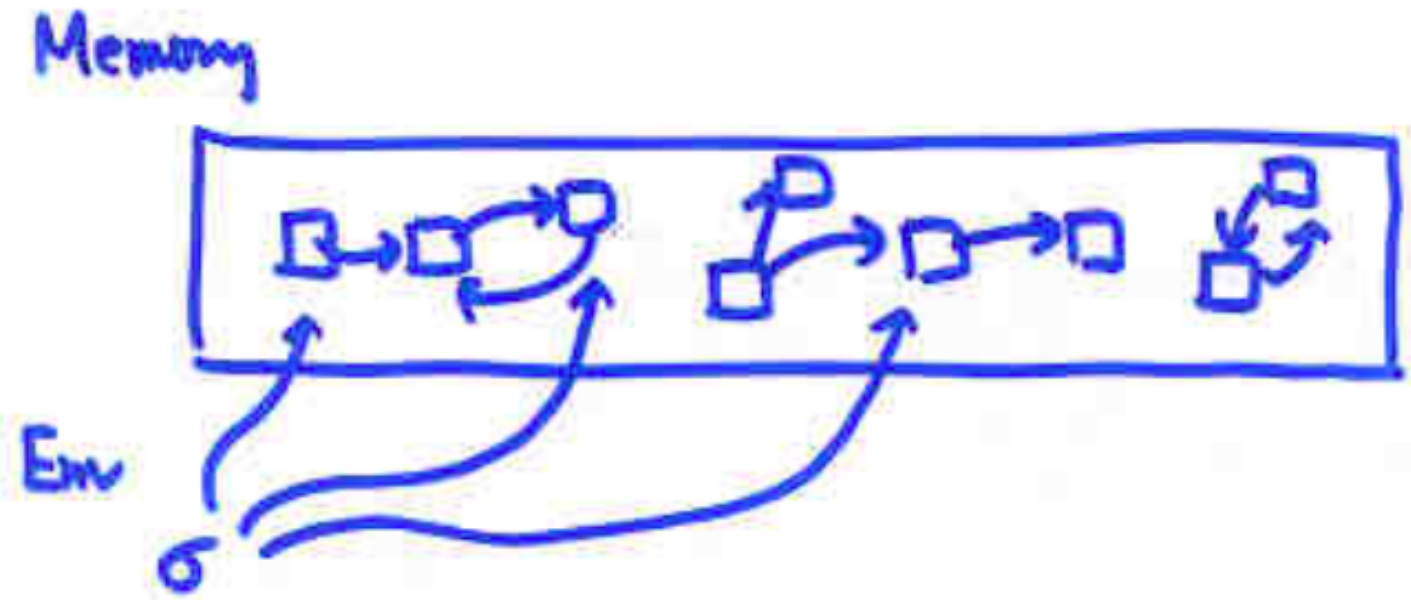
자동으로 정확히는 불가능. 그렇다면?

- 안전하게(sound)는 가능
- 완전(complete)하지는 못하다 (재활용 빠뜨리는 게 있다)

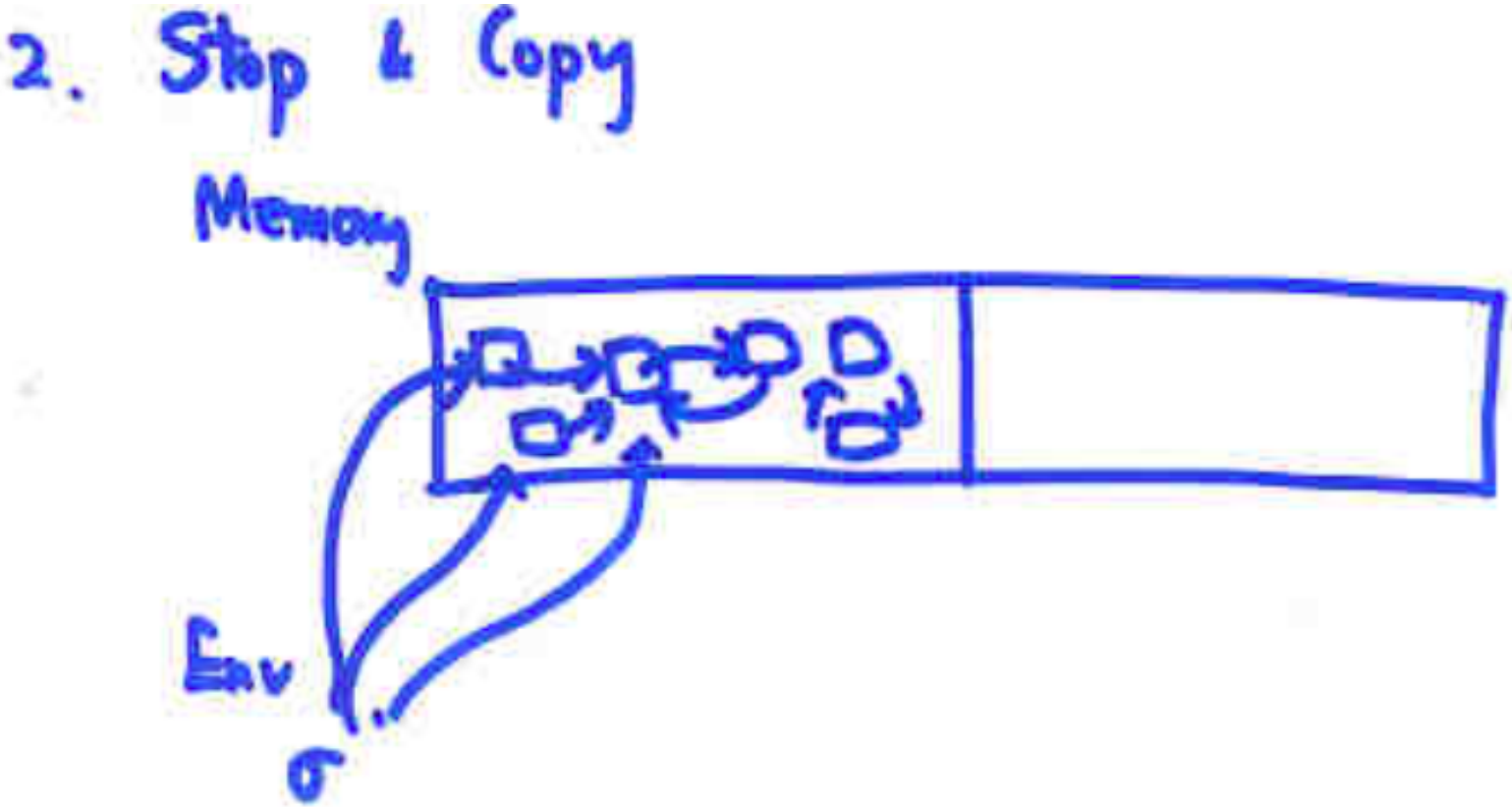


메모리 재활용 알고리즘 (1/3)

1. Mark & Sweep

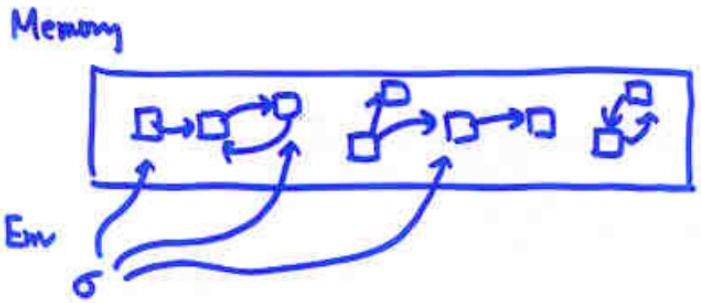


메모리 재활용 알고리즘 (2/3)

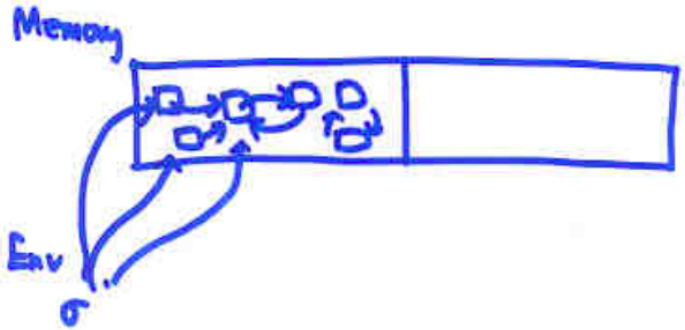


메모리 재활용 알고리즘 (3/3)

1. Mark & Sweep



2. Stop & Copy



- 딜레마: 도달할 수 있는 것을 모두 찾아가려면 메모리가 필요
- 딜레마: 메모리 재활용은 메모리 없을때 하는데
- "헨젤과 그레텔" 아이디어

정리: 자동 메모리 재활용

- 프로그래머에게 맡기면 희망이 없다. 너무 어렵기때문.
- 자동 메모리 재활용(garbage collection) 기술이 성숙
  - 주류 프로그래밍언어에 장착됨: Java, Scala, Ocaml, etc.
  - 안전하게 재활용
  - 완전하게는 못한다. 재활용 빠뜨리는게 있다. 그래서 메모리 누수(memory leak)이 가능. 그러나 드물다.

프로그래밍 언어의 두 기원

프로그램 오류 자동 검증: 타입 체킹

프로그램 실행 자원 자동 관리: 메모리 재활용

프로그래밍에 동원되는 생각방식 14선

프로그래밍에 동원되는 지혜 14선

일상의 지혜와 크게 다르지 않다

1. *데이터 정리하기*$_{data\ structure}$
2. *속내용 감추기/핵심 간추리기*$_{abstraction}$
3. *조립식으로 생각하기*$_{composition}$
4. *계층쌓기로 생각하기*$_{hierarchy}$
5. *끼리끼리 포장하기*$_{module}$
6. *반복으로 생각하기*$_{repetition}$
7. *재귀적으로 생각하기*$_{induction\ \&\ recursion}$
8. *순서로 생각하기*$_{sequentiality\ \&\ concurrency}$
9. *상태나 값으로 생각하기*$_{state\ \&\ value}$
10. *틀을 짜서 재사용하기*$_{framework}$
11. *실행비용 생각하기*$_{cost}$
12. *올바른지 확인하기*$_{correctness}$
13. *통밥 동원하기*$_{heuristics}$
14. *무작위 동원하기*$_{randomization}$

프로그래밍에 동원되는 지혜 14선

1. *데이터 정리하기*$_{data\ structure}$
   - 자료 정리하는 지혜, 정리한 모양새(구조)에 따라 일을 편하게 할 수 있다
   - 비유: 책방의 책정리, 식당줄 일렬
2. *속내용 감추기/핵심 간추리기*$_{abstraction}$
   - 복잡한것을 다루는 지혜, 외부에는 속구현을 알려주지말라
   - 비유: 자동차운전, 서비스계약

프로그래밍에 동원되는 지혜 14선

3. *조립식으로 생각하기*$_{composition}$
   - 부품을 조립해서 전체를 제작, 부분방법들의 모임이 전체방법
   - 비유: 인구조사, 수능공부, 대학
4. *계층쌓기로 생각하기*$_{hierarchy}$
   - 조립 과정이 여러 계층으로
   - 비유: 세포 $<$ 장기 $<$ 사람 $<$ 마을 $<$ 나라 $<$ 문명

프로그래밍에 동원되는 지혜 14선

5. *끼리끼리 포장하기*$_{module}$
   - 비슷한 것끼리 모아서 포장 정리해 놓기
   - 비유: 이삿짐 포장
6. *반복으로 생각하기*$_{repetition}$
   - 같은 작업을 반복하면서 일을 마치는 지혜
   - 비유: 밥 먹는 일, 이동하는 일

프로그래밍에 동원되는 지혜 14선

7. *재귀적으로 생각하기*$_{induction\ \&\ recursion}$
   - 전체와 부분이 같은 구성. X로 X만들기, 그런 X를 훑기
   - 비유: 일렬의 줄, 조직도, 전화번호부, 나무가지, 양파
8. *순서로 생각하기*$_{sequentiality\ \&\ concurrency}$
   - 일의 순서를 생각하는 지혜, 순서대로 혹은 동시에
   - 비유: 연극 진행, 요리

프로그래밍에 동원되는 지혜 14선

9. *상태나 값으로 생각하기*$_{state\ \&\ value}$
   - 변하는 상태를 생각하기, 변함없는 값을 생각하기
   - 비유: 장볼때 장바구니, 숙제 베끼기, 부서 메뉴얼 만들기
10. *틀을 짜서 재사용하기*$_{framework}$
   - 자주 쓰는 것을 재사용하기 쉽게 준비하기
   - 비유: 떡살, 공교육 시스템

프로그래밍에 동원되는 지혜 14선

11. *실행비용 생각하기cost*
    - 프로그램가 시간과 메모리를 얼마나 쓸지 어림잡는 지혜
    - 예: 1부터 100까지 그녀가 마음속에 가지고 있는 숫자 알아맞추기, n vs $\log_2 n$
12. *올바른지 확인하기correctness*
    - 프로그램에 오류가 없는지 확인/검산하는 지혜
    - 비유: 로켓을 잘못만들면 자연이 실행시키는 그 로켓은 상승중 폭발

프로그래밍에 동원되는 지혜 14선

13. *통밥 동원하기*$_{heuristics}$
    - 정확한 답은 너무 시간이 걸리고, 적당한 답으로 만족해야할 때, 통밥에 기댄다.
    - 예: 1000만 관객의 영화만들기. 모든 선택의 기로마다 모든 가능성을 따져보기? 너무 시간이 걸리는 넌센스.
14. *무작위 동원하기*$_{randomization}$
    - 널뛸 수 있는 프로그램성능을 진정시키는 알약이면서 + 방법을 모를 때 무작위 반복으로 어림잡는 지혜
    - 예: 군대 정렬 기준잡기, 오줌싸게가 필요한 소금량 구하기

강의리뷰

프로그래밍 언어의 두 기원

프로그램 오류 자동 검증: 타입 체킹

프로그램 실행 자원 자동 관리: 메모리 재활용

프로그래밍에 동원되는 생각방식 14선