프로그래밍 언어론

1주차 강의

Introduction

Machine language = 0 과 1로만 이루어진 언어

Sequence of bits(0 과 1로만 이루어진 언어 = instruction code, set) that directly controls a processor(add, compare, move data from one place to another, and so on)

The next step was writing in assembly language(명령어와 1대1 mapping)

* Use mnemonic abbreviations(ex) 101110101 -> add(명령어))

어셈블리언어는 한가지만 있는 게 아닐 수 있다. -> assembly 언어를 기계어로 변환해주는 것을 assembler. 이것이 달라지면 언어가 달라 질 수 있다, cpu가 달라져도 그럴 수 있다. => 조금 더 사람에 가까워졌다.

Assembly language

* One-to-one mapping between mnemonics and machine language instructions translated by assembler ->mnemonics와 machine language와 1대1mapping, assembler에 의해 기계어로 변환.
* Assemblers are augmented with “**macro expansion**” to define parameterized abbreviations for common sequences of instructions -> 반복되는 어셈블러를 사용할 때 함수를 사용하는 것과 같이 비슷한 식으로 동일한 코드를 복사시키는 것임.

Easier than machine language, but still difficult and machine-centered

* System specific (not portable) -> 한시스템에서 배운 것이 다른 쪽에서 쓰일 수 없다.

Dreams about **machine-neutral programming languages**, with numerical computations capability -> Fortran -> 그래서 **machine-neutral programming languages**을 꿈꾸었다.

=> 시스템에 관계없이 하나의 코드를 작성에 다양한 시스템에서 작동하게 시켜주는 언어들을 이야기한다. Ex) c, java.

Catches up slowly, though

* Human could write faster programs than compiled code -> 사람이 작성한 코드가 컴파일 언어보다 빠르게 할 수 있다.

More high-level languages were introduced, such as Lisp and Algol -> 이런 언어가 있음을 알기

Translating from a high-level language to assembly or machine language is the job of a **compiler**

: 프로그래밍 언어 고급 어를 기계어나 어셈블리 언어로 번역해주는 역할이다.

Performance gap was narrowed and reversed eventually -> 사람이 만든 코드와 컴파일이 만든 코드 봤을 때 속도차이가 있다. 현재 좁혀지고 역전되고 있다.

Need to economize the programmer effort-> 프로그래밍을 하는 사람의 효율성 때문에 그렇다.

* the original construction of programs 처음에 프로그램을 만드는 비용이나 노력
* subsequent program maintenance – enhancement and correction 유지 보수하는 데의 노력
* Labor costs now heavily outweight the cost of computing hardware -> 사람의 인건비용이 컴퓨팅 하드웨어비용보다 비싸다.

이래서 high level programming에 관심을 갖는다.

* 1. The Art of Language Design

Why are there so many programming languages? 왜 많은 언어가 필요한가?

Evolution – we’ve learned better ways of doing things over time (진화나 발전)

* goto-based control flow (Fortran, Cobol, Basic: 초창기의 고급언어) jmp, jump명령어.
* Structured programming 구조화 프로그래밍, 반복문, 함수 사용(C언어, Algol, pascal)
* Object oriented programming (Smalltalk, C++, Eiffel, Java, C#) 객체 지향 프로그래밍 구조화 프로그래밍보다 재사용성에 목적을 둠.
* Scripting languages displace more traditional compiled languages, at least for rapid development (Python, Ruby) 바로 바로 실행시켜 결과를 볼 수 있는 언어.
* Special Purpose 특별한 목적

Lisp dialects – symbolic data and complex data structures를 다루기 위해 만들어짐.

icon and Awk are good for manipulating character strings

C – low-level systems programming을 하기 편하다.

Prolog – reasoning about logical relationships among data 인공지능에서 사용.

그 언어가 지향하는 분위기이다.

Personal Preference (사람의 취향)

* It may not be possible to develop a universally acceptable programming language

What makes a language successful? 언어의 성공요인

* Expressive power 표현력  
  어떤 언어가 다른 언어에 비해 파워 풀 하다.  
  프로그래밍을 할 수 있는 영향을 줄 수 있는 부분들을 가지고 있다.   
  명확하고 간단하게 그리고 유지보수 가능한 코드를 작성할 수 있도록 프로그래머들의 능력에 영향을 줄 수 있는 언어의 기능을 가지고 있다.
* Ease of use for the novice 초보자를 위해 만들어진 언어들  
  Basic, Pascal
* Ease of implementation 구현의 용이성  
  Basic, Forth, Squeak   
  Pascal
* Standardization 표준화|  
  플랫폼에서 호환성을 가진다.
* Open Source 오픈소스
* Excellent Compilers을 가졌다.
* Economics, Patronage, and Inertia 경제성, 지원, 관성의 법칙  
  기술적으로 좋다고 성공하는 것만은 아니다.  
  큰 회사가 밀어주면 된다.  
  이전에 쓰던 것 때문에 언어를 써야한다.

No one factor determines whether a language is “good” 어떠한 요소 1개가 언어를 좋다 나쁘다 할 수 없다, 여러가지가 맞아 떨어져야 한다, 운도 중요하다.

Need to consider issues from several points of view, specially both programmer and the language implementer 언어를 만들고 사용하는 입장에서 봐야한다. 언어를 만들 때에는 구현자와 사용자를 고려해야한다.

Conflicts and tradeoffs

* Features vs cost of their implementation (기능들 vs 구현의 비용.
  1. The Programming Language Spectrum 언어의 종류들

Many existing languages can be classified into families based on their model of computation  
어떠한 계산 모델을 사용 하느냐에 따라 구분된다.

* Declarative

The focus is on what the computer is to do 컴퓨터가 무엇을 하느냐에 대해서 focus

properties are declared, but no execution sequence is specified 속성들이 만들어지고 어떤 순서대로 실행되어야 하는 순서를 정하지 않는다.

higher level – more in tune with the programmer’s point of view, and less with the implementer's point of view 또다른 어떤 추상화 수준의 언어.

* Imperative (현재 언어 c, java)

focus is on how the computer should do it 컴퓨터가 어떻게 무엇을 처리해야 하는지에 대해서 focus

action oriented 실행하는 것이므로

computation is viewed as a sequence of actions computation을 action의 흐름으로 본다.

predominate mainly for performance reasons 성능적으로 빠르므로 사용

테이블이(가) 표시된 사진

자동 생성된 설명

Functional languages

employ a computational model based on the recursive definition of functions 재귀호출, 재귀적인 함수의 정의에 기초를 둔 computation 모델을 사용한다.  
함수= 입력-> 출력, 함수를 엮어가며 프로그래밍을 하는 방법.

based on **lambda calculus** developed in 1930s에 기반을 하였다.

a program is considered a function from inputs to outputs, defined in terms of simpler functions through a process of refinement 함수형 언어에서 프로그래밍은 input에서 output이 있는 그러한 함수를 고려해서 만든다. 그 함수를 단순한 형태의 함수로 쪼갠다. (함수의 결과를 다른 함수 또 다른 함수에 제공). 순수는 반복문의 개념 X.

Dataflow languages 데이터의 흐름.

Logic or constraint-based language 규칙을 만들고 어떻게 하면 어떻게 한다를 따라서 동작한다.

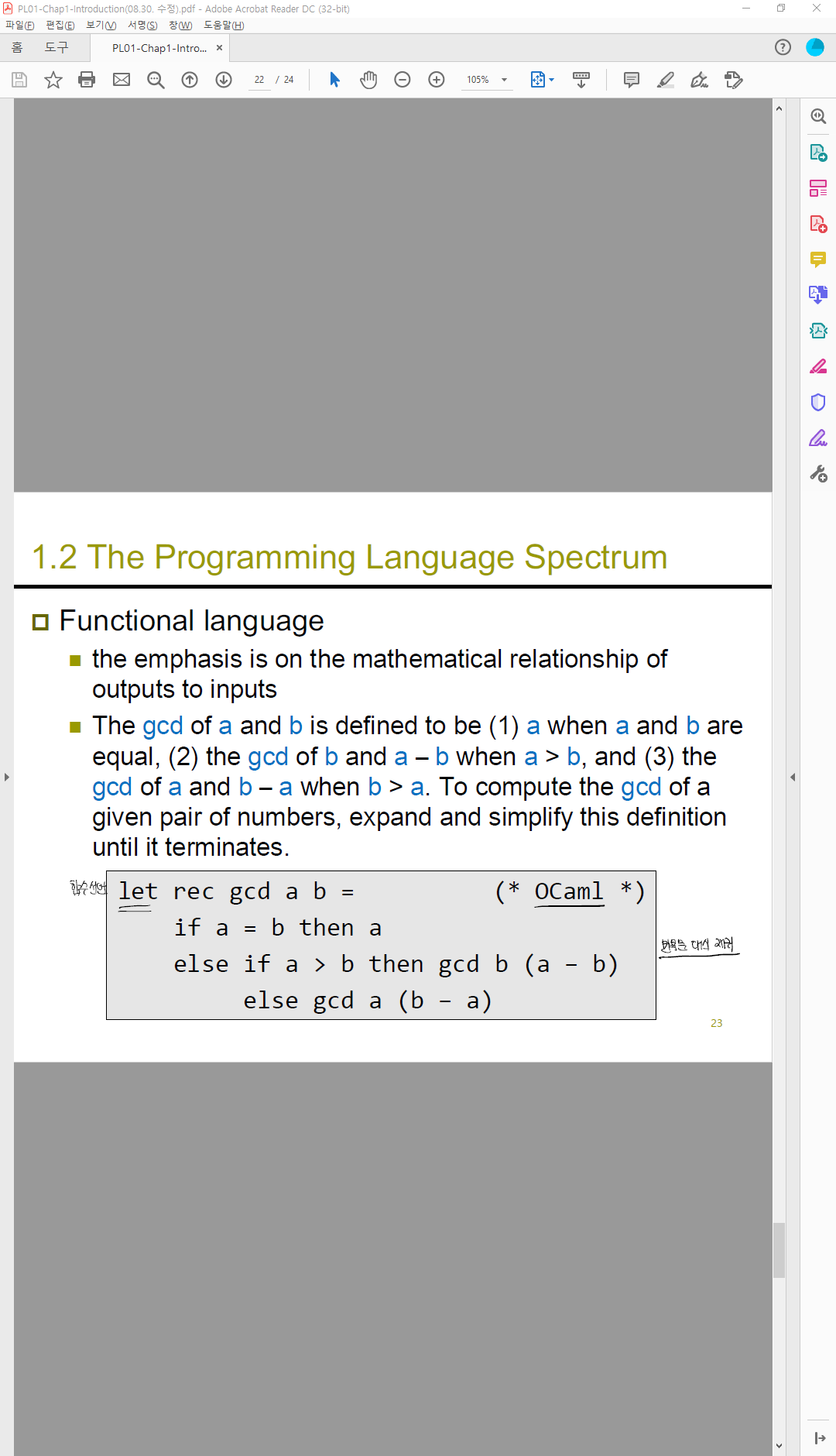
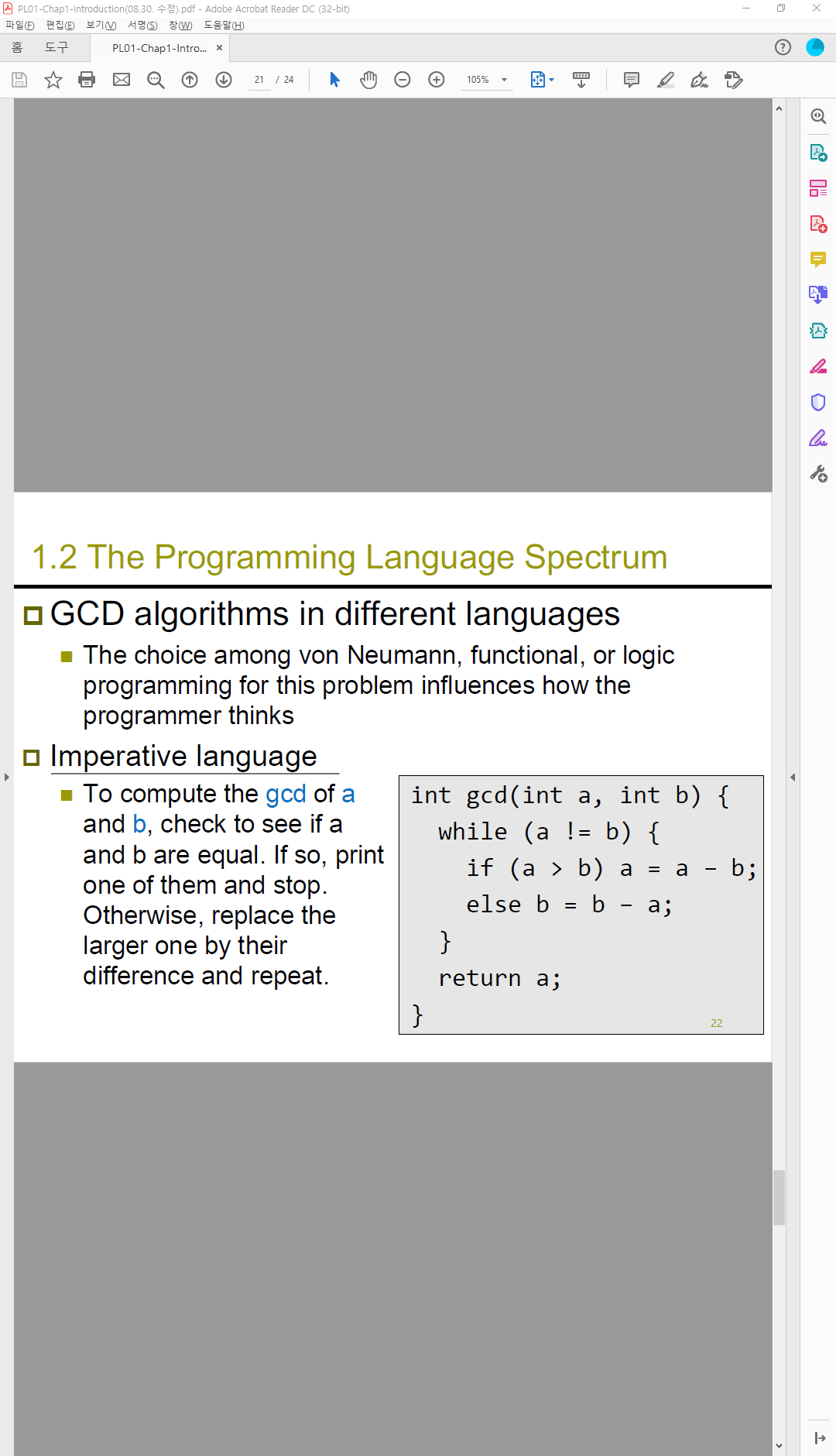
von Neumann languages 제일 많이 사용하고 접한 언어.

**Side effects** 부작용, 함수나 표현이 부작용을 발생시킴. Ex)함수의 input과 output이 다르다.  
이를 중요하게 사용하는 언어들을 이야기한다.

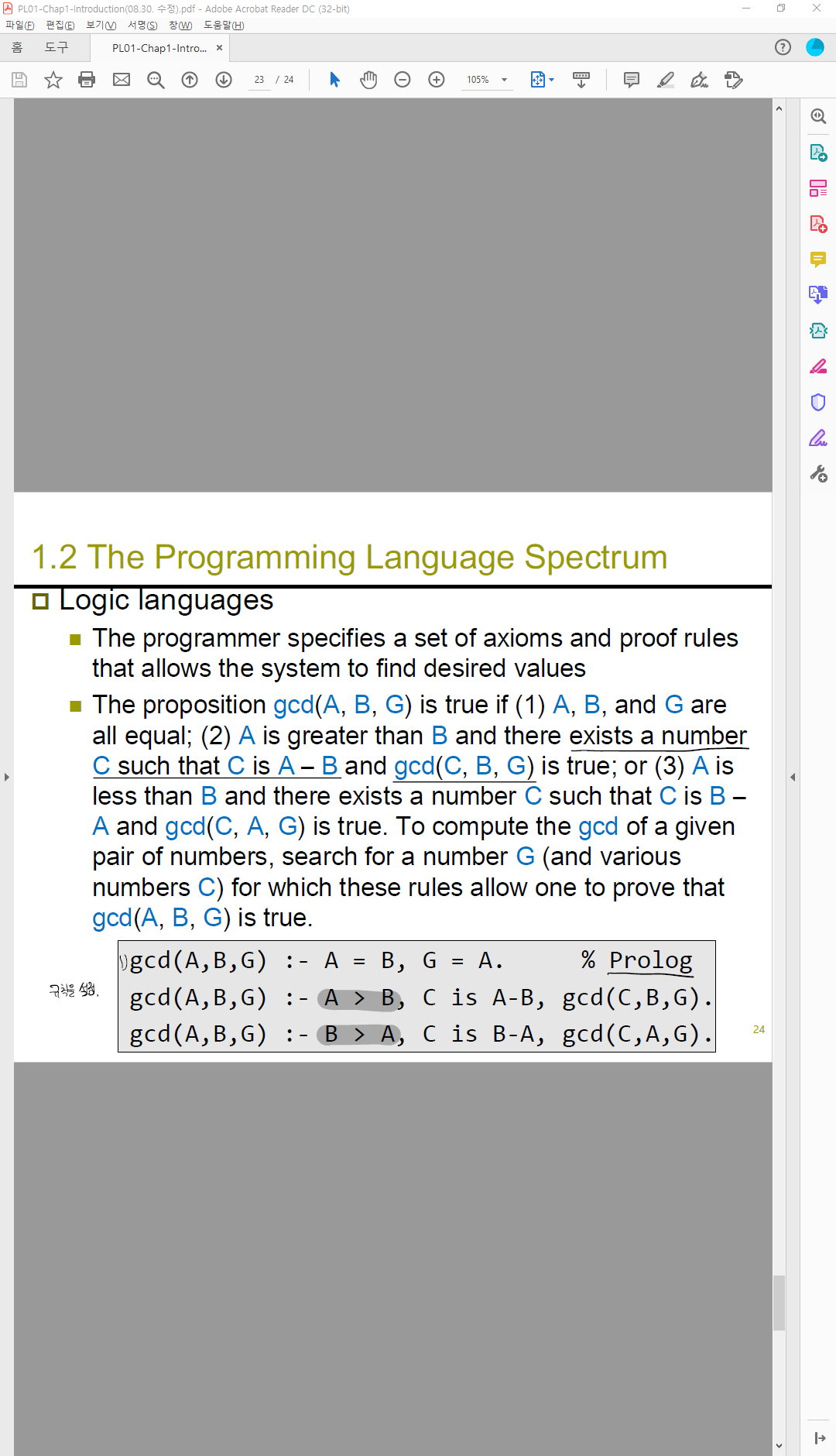
Object-oriented languages  
폰노이만 언어에 object를 붙임. // 언어끼리 서로 닮아간다.

Scripting Languages

* A subset of the von Neumann languages(특별한 성향을 가지지 않은 것들)
* Distinguished by their emphasis on coordinating or **“gluing together**” components drawn from some surrounding context 간단하게 쓰이기 위해 만들어졌다.
* Several scripting languages were developed for specific purposes 특정 목적을 가지고 만들어졌다.



반복문 대신 재귀호출

텍스트이(가) 표시된 사진

자동 생성된 설명

2주차 강의

Abstraction 추상화

* 초창기 언어들은 고수준의 언어이다. 이유는 그 언어들의 문법과 의미를 생각했을 때 하드웨어(어셈블리언어)에 비해 아주 많이 추상화 되어있다.
* Abstraction achieved two complementary goals   
  Machine independence 기계에 의존X  
  Easier to understand and programming for humans 사람이 이해할 때 쉽게 되어있다.
* primary reason of the design of modern languages
* A name: 문자열, 문자열 이름이 아닌 다른 것을 나타내는 l-value = 공간, r-value = 값 이렇게 나타날 수 있다. 이것도 추상화 작업이다.
* Names: 컴퓨터에서 언어는 프로그래밍언어이다. 프로그래밍 언어에서 이름은 식별자이다. -> 변수, 타입, 클래스의 이름 = 알파벳과 숫자의 조합.
* variables, constants, operations, types,은 저수준의 낮은 단계의 모양대신에 알아보기 쉬운 이름을 제공
* 추상화: 2가지를 이야기함에서 (사람이 이해할 때 쉽게 되어있다)이다.  
  프로그래밍 하는 사람들이 어떤 이름과 복잡한 코드 블록을 연동시키는 과정을 추상화라고 볼 수 있다.
* 추상화: 특정 컴퓨터의 자세한 구조의 자세한 내용으로부터 언어의 기능을 분리하는 목적: subroutine, function=> control 추상화
* Class는 data 추상화
* 추상화: 단순화, 복잡한 부분을 단순화 하게해준다. 그 부분에서 이름이 중요하다.

Binding

두개를 묶어주는 것임. => 프로그래밍에서는 이름과 인자를 묶는 것이다.

실행 중에서만 일어나는 것은 아니다.

Binding time binding이 일어난 시점.

일어나는 시점.

1. Program writing time  
   코드를 작성할 때  
   알고리즘, 데이터 구조, 이름들
2. Compile time  
   High-lever construct -> machine code로 결정  
   ex) if문이 machine code로 변경.
3. Link time  
   compilation(c언어의 link)  
   ex) c언어의 printf(“adc”)  
   subroutine(function)  
   link를 해주는 것은 linker  
   미리 만들어진 코드(standard subroutines)과 연결하는 것을 말함  
   어떠한 변수의 공간을 잡아야 한다는 것을 말한다.  
   어떠한 파일의 a가 다른 파일의 a라는 것을 연결.
4. Load time  
   메모리에 프로그램 코드, 데이터를 메모리에 올려서 실제 실행시킬 수 있게 해주는 작업  
   link address와 load address는 다르다.  
   link 해서 만들어진 공간은 0부터 시작 그러나 실제는 메모리에 그 공간에 올 수가 없다.
5. Run time  
   실행하는 시간을 의미 value와 variable을 연결.  
   elaboration time = 선어이 처음 보여질 때   
   **Elaboration of a declaration** refers to the **storage allocation and binding process이 될 때ex) int a(storage allocation)**시작부터 끝
6. Static (before runtime), dynamic(at runtime)  
   compile time link time

Early binding times → greater efficiency static(compile time)  
mostly compiled languages

Later binding times → greater flexibility dynamic(동적으로 이러나는 것)  
mostly interpreted languages ex)조건을 통해 어떤 것과 연동되는 것

Binding Time and Language Design

Semantic: 제대로 사용되었냐 의미가 있냐?

실행되어야 알 수 있는 경우에는 확인하기 어렵다.

Compile 언어는 만들기 어렵다. 함수호출이 되냐 안되냐는 뒤로 미뤘기에  
확인하는 것을 뒤로 미루면 유연하게 할 수 있다.

자바의 예제를 통해 late binding을 확인할 수 있다.

이유는 부모클래스 변수에 자식클래스 객체를 저장했을 때 중복되어있는 함수의 위치는 자식에서새로 만들어진 위치로 덮이기에 (binding이 runtime에서 결정)

Object Lifetime and Storage Managements

* 실행할 때를 이야기한다. Object(객체X) = 변수의 메모리 공간, 동적할당메모리공간, 클래스는 객체를 포함한다.
* 이름과 그 이름이 나타내는 object를 구별해야한다.
* 중요한 시점을 살펴봐야한다  
  object 생성, 소멸 (공간)  
  binding 생성, 소멸  
  binding이 deactivate(바인딩이 사라지는 것), reactivation(다시 살아나는 것).이는 임시로 사용할 수 없는 시간이 만들어짐.  
  참조를 하는 시점 = 실제 사용하는 시점.
* Lifetime = 만들어졌다가 사라지는 시점.
* Object나 binding은 일치하거나 일치하지 않을 수 도 있음.  
  해당 과정은 예제를 보기. 1. 짧은 것임.  
  pass by reference는 새로운 이름을 부여라고 생각하기.  
  이 &의 차이를 잘 봐두기.

텍스트이(가) 표시된 사진

자동 생성된 설명

* 2. 긴 경우-> bug.  
  t라는 이름으로 객체에 연결이 되어있었지만 &를 통해 넘겨주면서 새로운 이름 v를 붙였는데 delete를 통해 v를 제거하면서 공간이 사라져 t라는 것이 가르키는 곳이 사라졌다.== dangling point.

텍스트이(가) 표시된 사진

자동 생성된 설명

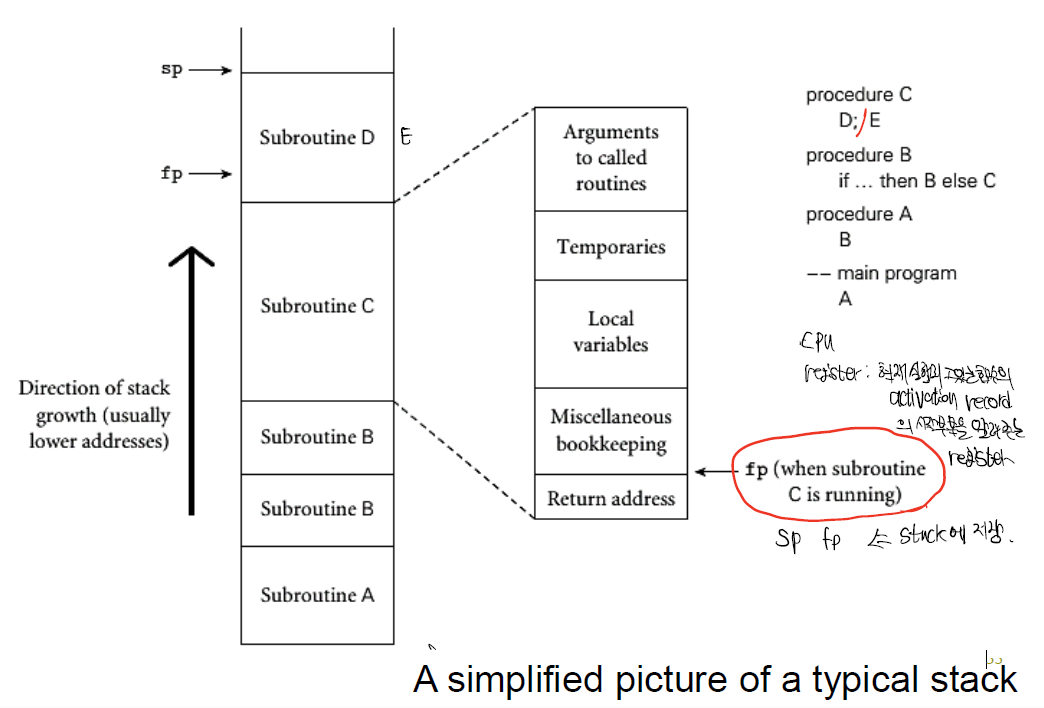
* Pointer와 reference와 비슷하다고 생각하기.  
  1. Object가 binding보다 오래 있으면 garbage  
  2. Object가 binding보다 짧게 살면 dangling reference
* Object lifetime은 storage allocation mechanism과 연관되어 있다.  
  1. Static object = 프로그램이 실행되는 동안에 남아있는 절대값의 주소(물리, 가상, 논리 주소가 아니다)= logical 주소의 절대값이다. == data의 영역에 해당(전역, 자바의 static변수)  
  2. Stack object = stack의 영역, 함수의 콜과 리턴  
  3. Heap object = heap의 영역 프로그래밍이 실행되는 도중에 할당 비할당가능.

Static allocation

* Static object: global전역번수, code역시 해당.
* 상수들도 해당한다.(numeric, string)
* Table을 만든다.(garbage 등 다양한 작업을 위한)
* 정적영역에서 고정되어 변하지 않는 값들을 이야기함. 리터럴 상수, 코드, 테이블  
  위에서 이야기한 것을 포괄적으로 이야기
* Local은 subroutine이 호출될 때 생성. 반환될때 종료 -> stack  
  (자기자신에 대한 호출 반복) recursion 이러한 생성 소멸 삭제를 하기위해 언어의 구현이 필요  
  recursion을 지원하지않는 언어도 있다. Fortran 90에서 지원을 안해서 local을 static에 할당.-> 장점: 다른 함수를 호출하는데 같은 영역 재사용 가능 반복호출이 아닐 시, 생성하고 관리하는 overhead 줄임. => 메모리를 효율적으로 사용가능
* 많은 종류의 언어들이 named constant는 compile time에 값이 결정되어야 한다고함.  
  ex) const int a = 2+3 Ok, contest int a = 2+b(변수) 불가능. -> 이미 결정된 값을 사용해야한다.
* 이러한 것들은 static하게 allocation, recursive의 local이라 하더라고 동일.
* c언어는 constant는 진짜 상수는 아님. 변수인데 값을 변경할 수 없는 변수이다.  
  실행 중까지 알려지지않은 값으로 설정될 수 있다. 의존적이다-> 아래 예제 보기.  
  파스칼은 진짜 상수이다. 위를 elavoration time constant라고함. 값이 변경될 수 있기에 stack에 가야한다.
* C#은 지정해서 사용한다.

Stack base allocation

* Recursion allocation  
  stack에서 변수들이 쌓인다 이 하나하나를 frame or record라고 한다. 영역은 다 다르다.  
  다만 이러한 공간을 activation record라고 한다.  
  **arguments** and **return values**, **local variables**, **temporaries**, and **bookkeeping information이 들어간다.**
* Bookkeeping은 subroutine이 어디로 돌아가야하는 가?



Activation record에 해당한다.\

* Stack의 관리는 calling sequence이다.
* Stack frame location은 compile time에서 알 수 없다.  
  실행이 안될 수 있기에 모른다.   
  frame안에 있는 object의 offset은 결정될 수 있다. Activation을 만들 때 얼만큼 떨어져있냐를 알 수 있음.
* Fp = cpu안에 있는 특별한 record 실행되고있는 함수의 activation record의 시작 부분을 알려주는 놈.  
  현재 실행하는 함수의activation record의 위치를 가르킴.  
  stack에 넣었다가 파악함.
* 지역변수는 offset을 알기에 얼만큼 떨어져있는 지 안다.

3주차 강의

stack frame이 만들어지는 것을 아는 것은 어렵다. Compile time에

하지만 stack frame 안에 있는 object들의 offset은 알 수 있다. Compile time에

* Accessing a local variable within the current frame can be done by adding a predetermined offset to the value in the frame pointer  
  상대적인 위치로 접근한다. Fp를 통해 접근도 가능하고 sp를 통해 -를 통한 접근도 가능하다.
* Stack은 일반적으로 높은 주소에 아래쪽으로 내려오는 주소로 만들어진다. 일반적인것임.
* 매개변수를 전달 받는 함수에서 이 매개변수는 부르는 쪽에 있는 것이 맞다. 일반적이다. 이게 편하다라고 해석하기
* 어셈블리 코드를 통해 본다면 offset을 통해 간접적으로 매개변수에 접근 또는 지역변수에 접근한다. 하지만 전역은 이미 알려져있으므로 직접 접근한다.
* Recursion이 없으면 static allocation을 사용한다. 지역변수를.  
   stack에 지역변수를 할당하는 것이 static보다 장점이 있을 수 있다.  
  대부분 모든 subroutine이 동시에 active(사용) X.
* 10~20%의 subroutine만 사용된다. 모두 올리면 공간낭비.
* Recursion이 없어도 stack을 사용하면 static보다 적은 메모리 사용할 수 있다.  
  recursion도 가능하다.

Heap-based allocation

* c언어의 malloc, 자바는 new  
  언제든지 allocate되고 deallocate가능한 영역을 heap  
  전체 영역이 아닌 sub영역.
* dynamically allocated data structures에서 사용한다.  
  fully general character strings일때도 사용한다. Java의 string class.
* 주의: stack이 함께 사용할 수 있음(함수안에 만들어진 변수는 stack, 넣는 문자열이 heap일 수도?(java hard coding은 data일 수도 있음)
* Heap을 관리하기 위해 전략적인 방법을 제공(speed, space)
* 공간을 생각할 때는 internal, external fragmentation (내부 외부 단편화)  
  internal은 특정크기 block일 때 발생, external은 다합치면 공간은 나오지만 떨어져 있어 연속적인 공간이 없으면 발생.  
  공간 낭비=internal.

Heap의 구현

* 비어 있는 공간을 linked list로 관리 = free list
* First fit: 맨 앞에 있는 것(가장 빠르다)
* Best-fit: 원하는 크기와 가장 크기가 비슷한 것 선택(공간낭비가 줄을 있다고 생각할 수 있음 -> 작은 자투리공간이 남는다.
* 공간을 받고 남는 공간을 다시 free list로 남는 부분이 작으면 internal fragmentation으로
* Best fit은 O(n)에 해당하는 검색이 필요. First fit O(1).
* External은 고정크기에서 발생X는 아니다.  
  heap에서 대부분 발생. 발생하다 보면 heap의 request를 만족하는 능력이 떨어진다.
* External 해결은 compact
* Deallocation은 언어마다 다르다.  
  c언어는 free를 해서 해야함. Explicit
* 최근은 implicit => garbage collection.
* Explicit의 장점: 구현이 간단, 빠르다.  
  단점: manual deallocation에서 버그가 많이 일어남.  
  너무 빨리 deallocation이 되면 dangling reference다.
* Garbage는 위의 반대이다.  
  단점은 type이 많으면 복잡.

이전까지는 데이터를 어떻게 생성하느냐 binding의 개념을 이야기함.

Scope rule

Scope= 변수나 함수의 사용범위.

* Binding이 유효한 textual의 영역을 scope임.

Statically scope(compile time의 영역)

* 컴파일 또는 코드상에서 결정(대부분의 언어가 해당)
* 코드를 보면 알 수 있다.

Dynamically scope(runtime)

* 코드만 봐서는 알 수 없고 실행해야 안다.
* Binding이 실행의 흐름에 따라 달라질 수 있다.

Scope

* 특별한 binding은 X.
* Binding이 변경되지않는 최대 code 영역을 말함.  
  적어도 그러한 binding이 없어지지 않는 한.

Referencing environment

* 프로그램 실행 중에 특정 상황, 시기의 유효한 binding을 말함.  
  static dynamic scope rule에 따라 다르게 나타난다.

C언어

* Subroutine의 entry의 시작될 때 새로운 scope영역이 만들어진다.
* 같은 이름을 가지는 global local이 있을 시 global은 deactivation.  
  예제들은 다시 정리하기.

Static scope

* Code를 보고 알 수 있다.
* 이름과 object의 사이의 관계가 compile time에 결정된다.
* 실행 중에 나타나는 제어흐름에 관계없이 우리가 code의 text만 보고 찾아낼 수 있다.
* Current binding의 개념은 주어진 위치에 가장 근접해 있는 block(surrounding block)에서 찾아내는 것  
  가까운 block이 없으면 다음 block.

초기 Basic은 단일 scope였음.  
Fortran 90 이전은 global과 local의 구분은 했음.

* local은 subroutine에 한정, 함수밖에서는 사용 불가.
* variable declarations are optional, 변수가 따로 선언 X시 현재 subroutine에 지역적으로 사용을 가정. 변수의 이름이 i-N이면 자동으로 integer로 가정, 아니면 real: double이라 생각.  
  이후 선택할 수 있게 하고 compile error을 발생.
* Programmers can override this rule by using an explicit save statement: 변수의 값을 보전하는 방법 c언어의 static과 동일.  
  프로그램 실행 중에 전체 life time을 가진다.  
  키워드는 달라도 제공된다.  
  object의 life time은 유효한데 binding은 선언되있는 곳에서만 실행가능 하다.  
  -> **함수가 여러 번 실행되도 그 지역변수의 값을 유지시키는데 사용된다.**

Nested subroutines

* subroutine안에 또다른 subroutines이 가능하다.  
  C에서 nested subroutines은 없다.
* 텍스트이(가) 표시된 사진

  자동 생성된 설명Closest nested scope rule: 가장 안쪽(가장 가까운 쪽)부터 scope를 확인하고 없으면 다음 scope  
  같은 이름이 있을 때 밖에 있는 것이 감춰진다.
* 많은 언어들이 미리 정의된 object를 제공  
  기본제공, I/O, math, predefine object는 가장 outer most scope에 있다.  
  가장 밖에 있다. = 전역 object를 말한다.

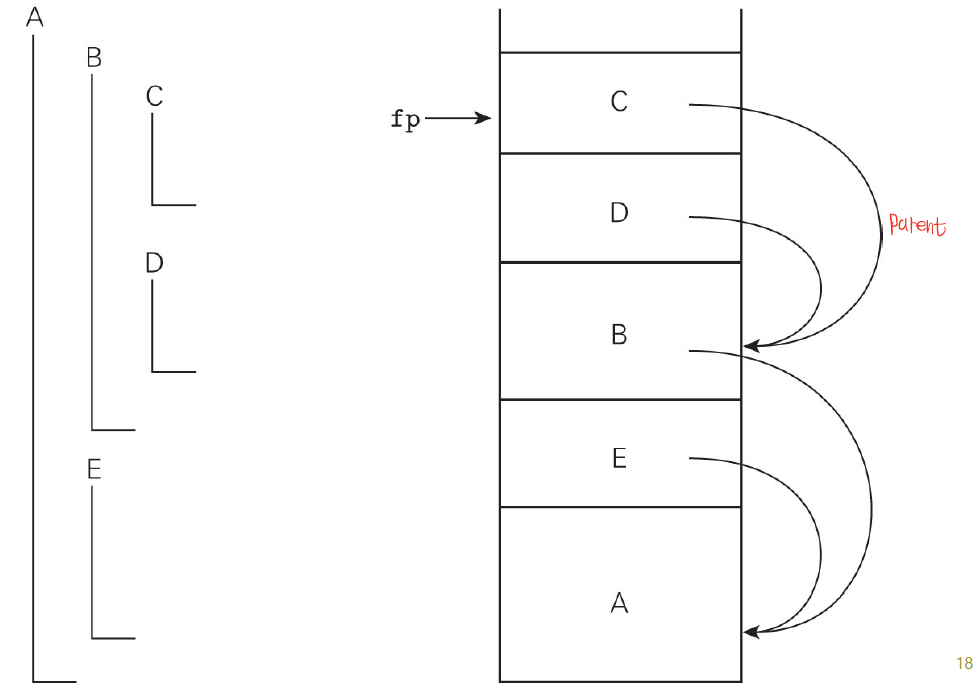
predefined object, global object 을 따로 정의하면 정의된 것이 사용 불가.

Procedure = void type의 반환을 하는 함수

Function은 type가 붙어있음 void가 아닌 type을 가지는 함수.

Begin end 는 함수 코드의 시작 , 끝을 나타낸다.

* A name-to-object binding that is hidden by a nested declaration of the same name is said to have a hole in its scope  
  같은 이름을 가지고 있는 변수에 의해 scope의 영역에 구멍이 뚤려있다.  
  일부언어에서 감춰진 영역의 변수는 달리 사용할 방법이 없다.  
  다른언어에서는 scope resolution, qualifier에 의해 감춰진 변수를 활용할 수 있게 해줄 수 있다.  
  ada = p.x이면 밖을 나타냄. C++은 ::을 쓰면 global을 호출.  
  이런식으로 밖의 변수를 접근하게 해주는 언어도 있다.
* Access to nonlocal object: 지역변수 아닌애들 접근하기.  
  local object 사용은 fp나 sp에서 offset을 통해 접근.  
  내부의 subroutine안에서 밖에 있는 변수를 사용할 때 전역은 data의 영역에 있으니 상관 X.  
  -> 해당 되는 stack frame을 찾아가야한다.-> static link를 만들어야한다. 각각의 pointer에서 == 각각의 frame마다 parent frame을 가리키는 pointer를 만든다.



* Parent frame에 대한 static link를 가진다.

Declaration order

* 변수가 선언된 순서들, scope에 어떤 영향을 미치냐?  
  -> 만약 변수 x가 선언된 그 block부터 scope인지 아니면 선언된 부분부터 scope인가?
* 초기언어는 모든 변수의 선언은 scope의 시작에서 하기로 했었음.  
  pascal은 이름이 사용되기전에 선언되어야 하지만 때에 따라 타입을 정의 전에 pointer를 정의해야하는 경우도 있음 -> forward reference(이런게있다라고 알리기 서로 재귀적으로 호출시)  
  pascal은 위의 문제에 전체 block이라고 답한다.
* whole-block scope(중간에 만들어도 전체 surrounding을 허용) and declare-before-use은 의외의 경우를 만든다.

텍스트이(가) 표시된 사진

자동 생성된 설명

해당 결과에서 오류가 발생 이유는 N이 내부에 있는 N을 사용 그리고 아직 값이 없기에 오류가 발생한다.

텍스트이(가) 표시된 사진

자동 생성된 설명

다음도 역시 오류가 발생 이유도 같다 N이 내부의 N으로 사용되고 const인 상수 값이 와야 하지만 결과론적으로 N은 변수이다.

* Pascal의 단점은 N이 제대로 사용되었는지 확인하려면 나머지 부분도 확인해야한다. 영역안에 있는 모든 것을 확인.
* C, C++, java는 해당 오류 발생 X -> 선언된 위치부터 유효하다.
* Java, C++은 같은 class안에만 있으면 어디에서 선언되어도 상관없다.  
  define before use를 완화.
* C#은 local은 whole block scope를 따라간다. Pascal과 같음.  
  **python도 같다.**
* recursive types and subroutines  
  이 둘은 사용되기전에 이름들이 declare 되어야한다.
* Mutual recursion: f에서 g를 부르고 g에서 f를 불러 다시 자기자신에게 오는 형태.  
  이당시에는 prototype를 적어준다.
* C에서는 declarer과 definition을 분리   
  함수의 원형을 적는 것은 정의, 함수의 코드 부분을 정해져있는 것을 정의한다고 함.  
  변수도 같은 형태가 나타난다.
* 즉 선언은 있다고 알려주고 정의는 값을 넣어서 하는 것이라고 생각하기.
* Declare가 부족하면 어딘 가에 해당 scope와 같은 곳에 있다.
* Nested block  
  ANSI C는 begin에서만 선언이 가능함.  
  대부분의 언어는 nested declaration이 outer declaration을 감춘다. -> 같은 이름으로 선언된 nested declaration의 경우 outer declaration을 감춘다. 그러나 java와 C#은 오류를 발생한다(그 outer declaration이 현재의 subroutine의 지역변수 인경우에 한해서).  
  **C와 java는 다르게 동작**
* Block을 생성하고 만드는 게 유용하다.  
  장점: interfere with another variable named temp  
  일반 지역변수와 같은 lifetime을 가진다.

Module

* Modularization: 모듈화(information hiding에서 중요한 역할을 한다).  
  client들에게 몰라도 되는 object나 함수, 알고리즘을 최대한 감추자(information hiding).  
  모듈화된 code는 최소한의 지식을 가지고 사용법을 익히면 된다. "cognitive load"
* Name conflict= 같은 이름
* Nested subroutine의 한계  
  Nested subroutine이 감춰저있는 그러한 subroutine과 life time이 같은 object만 information hiding이 가능하다.  
  -> 텍스트이(가) 표시된 사진

  자동 생성된 설명 f가 살아있을 때만 g가 살아있음.  
  해결법: save, static(life time은 전체)  
  단점: 이러한 것들이 장점은 있지만 그러한 값을 유지하는 게 1개이상으로 함수로 구성된 abstraction에서는 도움이 안된다. 1개에 함수에 대해서 값을 유지임.
* C에서 전역인데 static으로 쓰면 파일에서만 범위 제한.
* Module은 information hiding을 위해사용하고 함수와 함수사이에 넘나드는 state를 유지하고 싶을 때 module의 개념을 사용한다.
* 추상화된 타입의 module  
  collection of object가 가능.  
  모듈내의 변수내 함수들은 자유롭게 사용가능 안의 내용은 밖에서 보이면 안된다. export될때까지.
* Export-> 사용할 수 있다.  
  사용 쪽에서는 import해야함.  
  여기서는 접근성을 이야기하려고함.
* 현대에서는 python의 package.  
  java에는 module이있음.
* C++의 namespace의 binding은 namespace의 밖에서는 부분적으로 감춰지거나 전체가 감춰진다. ->그러나 사라지는 것은 아님.  
  scope나 일부로 감추는 것으로 봤을 때 안보일 거야라고 할 수 있지만 안보여서 사라지거나 lifetime이 없지않음.
* C++에서는 namespace는 반드시 outermost(전역보다 밖)에서만 존재 가능   
  seed라는 변수는 set\_seed나 randint와 관련없이 값을 유지.
* randmod밖에서 안에있는 함수를 사용가능? Ok rand\_mod**::**rand\_int로 사용가능.  
  모듈은 export, inport가 있어야함 C++은? -> export는 없어도 import는 조금 애매하다.  
  seed도 사용가능 그러나 좋은 아이디어는 아님.  
  rand\_mod:: == using namespace rand\_mod; 로 줄여서 쓸 수 있음.  
  이름 충돌(name conflict)이 일어나지않는 가정하에서(wholesale exposure of a module's names)
* variables may be exported read-only   
  types may be exported opaquely. -> 사용성의 일부제한을 거는 경우가 있음.
* 선택적으로 open 되어있을 수 있음
* Module은 abstraction을 생성하게 한다. 추상화된 타입의 실물을 생성하는 것을 허용한다.  
  단점: modula 2 에서의 예시에서 한 abstracion만 정의 가능하다.  
  해결은 module의 code복사해서 다른 이름을 만들기.  
  -> 그런 식 보다 manager의 역할로 만들어준다.  
  -> module을 또다른 타입으로 지원.(java)

Module Types and Classes

* Module을 타입으로  
  module type = class와 비슷  
  class는 inheritance(상속) and dynamic method(다형성)이 있어서 class는 module에서 진보된 개념을 가진다.
* Class가 module의 모든 기능을 지원하기 어렵다.  
  예제 게임에서 서버는 하나의 class로 구성 X, 엔진도 동일 그래서 class의 개념보다는 전통적인 module의 개념이 맞다.  
  그래서 java는 class를 아우르는 package 이를 아우리는 module의 개념이 생김.

자바의 모듈

* 자바는 class를 아우르는 것을 package라고 함.

텍스트이(가) 표시된 사진

자동 생성된 설명

의 문제가 발생함. Package를 알고 있는 사람이 모두다 사용함.

일부 package는 open 나머지는 hide.

Dynamic Scoping

* 실행할 때마다 결과가 달라진다.  
  실행 중에 프로그램이 어떻게 실행되느냐에 따라 달라진다.  
  Dynamic scope rules: 가장 현재의 binding은 어떤 변수의 이름이 주어질경우 그변수의 binding을 찾을려면 실행 중에 가장 최근에 마주친 것을 현재의 binding으로 봐야한다.  
  그러나 return되서 없어지지 않은 것.

텍스트이(가) 표시된 사진

자동 생성된 설명  
ada에서는 오류가 발생 이유는 max\_score가 double이므로 다이나믹을 사용하면. 문제가 발생 할 수 있다.

* Dynamic clash 때문에 Dynamic semantic checks를 해야 한다.
* 실행을 해야 문제가 있는 지 알 수 있다.

Implementing Scope

* Scope를 구현하는 방법: statically scoped은 compile time에서 이는 symbolic table을 만든다.   
  표에 각 사용되는 변수들이 있고 값을 나타냄: dic or map
* table에서 하는 일 2가지 1. 새로운 것을 mapping, 2. 값 찾기.  
  Static scope rules allows a given name to correspond to different objects (different information)= 지역변수와 전역변수생각해보기  
  이는 enter\_scope와 leave\_scope를 통해 stack에 집어넣는다.  
  look up은 위에서붙어 찾으면된다.
* Dynamic scope는 interpreter가 symbol table의 작업과 유사한 작업을 해야한다.

5주차 강의

The Meaning of Names within a Scope’

* 이전까지는 1to1 mapping을 이야기했었음.
* Aliase: 두개 이상의 이름이 같은 object를 가르키는 것.
* 한 개의 object에 여러 개의 이름이 붙어있는 경우: overloaded. == 다형성과 연결  
  같은 object에 여러 개의 이름: aliase.

Aliase

* Union은 메모리 공간을 공유한다.  
  같은 메모리 공간을 다른 이름으로 접근(접근 공간의 크기의 차이는 있을 수 있음)  
  일종의 aliase.
* C++의 reference type도 해당.
* 매개변수를 pass by reference 형태로 전달할 때 alias발생.  
  같은 메모리 공간에 다른 이름을 붙인다.

Overloading

* 이미 제공하고 있으나 객체지향 언어들이 제공하는 것과 다른 것과 다르게 제공(limited form)  
  c에서 +는 여러가지 기능을 한다.-> int, double, float등 다양한 타입이 들어올 수 있음.
* Overloading을 사용하면 symbol table을 유지해야한다.  
  lookup(변수의 이름을 찾는 것)에서는 request name에 대한 가능한 의미를 찾아 되돌려줄 것임. semantic analyzer(의미를 찾아가는 것)에서 context에 맞춰서 어떤 것을 선택할지 찾는다.
* Ada와 같이 잘 못 사용할 수 있는 경우를 막기위해 typename을 써야하는 경우도 있음.
* enumeration constant: 값을 나열하는 것. Enum  
  C에서는 구별되어야함.
* Ada and C++ can overload subroutine names 독립적인것.  
  java나 C#도 가능.  
  어떤한개의 주어진 이름이 개수가 정해지지 않은 subroutine을 나타내는 것이 가능한대 이렇게 할려면 각각은 개수가 다르거나 type이 다른 argument가 있어야함.
* 이미 만들어진 산술연산자를 overloading이 가능.  
  보통언어에서 prefix form이 정해져 overloading 해야하는데 이경우 prefix 형태가아닌 우리가 익숙한 infix형태로 만들어놓았다 -> 이래도 함수를 정의할 때 prefix형태로 해야한다.  
  🡪 overlading한 prefix 형태 ex) +”abc”, “casd” 를 🡺 “abc” + “casd” 도 가능하다.  
  이것이 지원이 가능해야한다.
* syntactic sugar = operator+(3,2) == 3+2로 할 수 있게 하는 것.  
  사용자가 익숙한 형태로 작성해줄 수 있게 하는 것.
* Hascal에서 Overloading한 연산자는 연산자의 우선순위, 결합 방식을 따라간다.
* Overloading의 개념을 coercion과 polymorphism과 구분해야 한다.  
  이 3가지 다 특정상황에서 한 개의 이름을 가지는 함수로 보이는 함수의 여러가지의 타입에 해당하는 argument들을 전달하는 것이 가능하게 보인다.  
  -> 한 개의 이름을 가진 함수에다가 내가 여러 개의 종류의 매개변수로 전달할 수 있는 것처럼 사용하는 case.
* Coercion: 컴파일러가 자동으로 다른 타입으로 변환  
  ex) c언어에서 f라는 함수를 double의 인자로 만들었을 때 f(2), f(2.1), f(2L)이 가능  
  다른 종류에 해당하는 타입을 저달-> complier가 변환을 해준다.  
  실질적으로 여러가지 형태가 있는 것은 아님(자동 형 변환 때문에 그렇게 보이는 것임)
* Polymorphism: 다형성,  
  java에서 함수를 만들 때 object 타입의 인자를 받는 함수를 만들면 여기에는 object로부터 상속된 class가능, 모든 class객체는 전달 가능.
* Print 예제에서 overloading은 함수를 여러 개 만들면 된다.  
  1. 없으면 오류를 내거나 2. 부모나 조상이 class타입을 받는 함수를 찾는다.  
  Coercion과 겹칠 수 있음.  
  자동 형변환을 할 수 있다면 호출 가능하면 호출.
* java에서 print 예제에서 tostring을 가지므로 print를 할 때 a가 override을 안하고 B가 tostring을 override했다면 a는 object의 tostring을 b는 자신의 tostring을 호출해 print를 한다.  
  == polymorphism: 부모 클래스로 자식클래스의 객체를 전달한 다음에 부모 클래스에 있는 함수를 호출하는 것처럼 보이지만 실제로는 자식이 override한 함수가 있다면 이 함수를 호출.
* Overloading: 같은 이름을 가진 여러 개의 object를 만드는 것.  
  어떤 type를 따라 적절한 형태 호출.  
  coercion은 자동 형 변환.  
  polymorphism은 argument multiple type에 대한 argument를 받는데 그게 어떤 타입이 support해주는 범위에서 사용.   
  tostring의 예제 생각.

The Binding of Referencing Environments

* 프로그램을 실행시키는 도중 멈췄을 때 그 당시에 reference environment를 정한다.  
  scope rule에 의해서  
  reference environment: 그 당시의 scope영역에 있는 것들.(변수, 함수)
* Static scope rule은 code를 보면 알 수 있다, 이름이 어디에 선언되어있는지 확인하면 어떤형태로 nest되어있는지 확인하면 실행중에 어떤 상황인지 알 수 있다.
* Dynamic scope rule은 runtime상에서 어떻게 실행되는지에 따라 달라진다.
* Reference to a subroutine을 만들어낼 수 있다.  
  함수가 다른 함수의 인자로 전달 되는 것.  
  함수를 전달하기 직전의 reference environment를 사용하게 할 것이냐? 아니면 실제로 넘겨진 함수를 호출하기 직전의 reference environment를 사용할 것이냐? 이것이 다를 수 있다.
* 보여준 예제에서는 후자를 택해야 된다.  
  전자일 경우는 line length가 main에서 전달 전에 정의 되어야한다.
* Shallow binding = late binding  
  parameter로 전달된 함수의 referencing environment가 late binding의 개념으로 들어가야한다. -> 호출되기 직전에 reference environment를 사용.  
  함수가 다른 함수의 인자로 전달, 그 전달된 함수안에서 매개변수로 전달된 함수를 사용할 때 그 함수가 호출되기 직전에 reference environment를 사용->shallow binding.  
  기본적으로 dynamic은 이를 지원.
* Threshold는 모를 수 있음-> predicate를 호출 할 때 Threshold없으니 문제가 없을 꺼같지만 print\_selected\_record가 어떻게 만들어졌는지 모른다면?  
  즉 안에서 threshold를 사용했다면? -> 문제가 발생 predicate를 호출할 때 threshold는 dynamic이므로 내부의 threshold를 사용한다.  
  이경우는 첫번째 경우에 해당.-> parameter로 전달된 함수에 대해서 특정 reference environment 함수를 전달하기 직전의 reference environment를 사용하면 좋겠음.  
  -> 이럴려면 reference environmentbind를 해야한다. -> deep binding.
* Shallow binding은 함수를 매개변수로 전달을 했을 때 그 매개변수로 전달된 함수가 실제로 호출되기 직전의 reference environment를 사용.
* Deep binding은 함수를 전달하기 직전에 reference environment를 그 함수가 사용했으면 하는 것.

Subroutine Closures

* Deep binding은 함수가 전달 될 때 보따리 싸듯이 함께 들고 간다고 생각.  
  함수가 전달될 때 reference environment가 묶여서 함께 간다.  
  shallow는 아님.
* 구현을 할려면: 그 당시의 reference environment의 set를 만들고 이것을 bundling을 해서 같이 보내야한다.  
  이런식으로 딸려보내는 것을 Closure이다.  
  -> reference environment를 가방에 넣은 것, 함수에 대한 reference자체를 closure라고함.
* closure안에 함수에 reference environment도 있고 함수에 대한 pointer도있음.  
  중요한 것은 함수자체와 함수에서 사용할 reference environment를 가방에 넣어놓은 것을 closure임.  
  이것이 subroutine closure이다.
* Subroutine을 인자로 넘길 때 subroutine과 reference environment를 한대 묶어서 전달.  
  주의: 그때의 reference environment는 그 당시에 사용할 수 있는 변수들이나 함수들을 이야기한다. 값을 나타내는 것은 아님. 어디에 있는 변수를 사용할 것이다.
* 대부분 shallow를 쓰는 대 특별한 경우에 deep을 써야할 경우도 있음.
* static에서 shallow는 안씀(의미가 없다) deep binding을 쓴다.
* static에서는 shallow는 의미 X,   
  static은 code의 위치를 보고함. deep도의미 x?  
  nesting에 level에 따라 달라 질 수 있다.  
  -> static scop일 때 binding rule의 언제 문제냐?  
  -> object가 local이거나 global이 아닐 때에서만 -> intermediate level of nesting== 만약 object가 현재 실행중인 subroutine에 local, 지역변수라면 그 서브루틴이 직접 호출이건 closure을 통한 호출이건 상관이 없다. 함수밖이 영향을 받지 안은 아님.  
  전역변수는 한 개이상의 instance는 있을 수 없다.-> main body의 program은 recursive X  
  object가 1개밖에 없으니 똑 같은 것을 유지할 수밖에 없다.  
  다만 어정하게 중간에있는 nested procedure는 문제가 될 수 있다.  
  근대 결론은 deep binding을 사용.

First-Class Values and Unlimited Extent

* 요소가 first-class인지 확인하려면 인자전달 가능, 함수에서 리턴 가능 , 변수에 전달 가능.
* Second-class는 인자전달가능, 함수에서 리턴 X OR 변수에 전달 X
* Third-class는 인자전달 X
* Subroutines are the most variation은 다르다.  
  언어마다 다름.
* Frist-class subroutine일 경우 복잡한 상황을 만들어 낼 수 있다.  
  서브루틴을 사용하는 함수 pointer가 함수가 실제 선언된 scope영역을 벗어나서 살아남을 수 있다.  
  -> 앞에서 closure의 예시와 같음.
* Local object가 없어진 뒤에 그 공간은 사라짐-> reference environment에서 long lived closure에 있는 예제의 lambda함수에 reference environment를 캡처해서 dangling reference에 대한 내용을 포함해 가지고 있다.  
  unlimited extent라 는 형태로 남는다.: their lifetimes continue indefinitely ( 더 길게 살아있다.)나중에 garbage collector가 쓸데 없다고 생각하면 없어짐.

6주차 강의

Object Closures

Dynamic scop에서 subroutine closure이 deep binding에서 사용.

Shallow binding은 subroutine closure이 필요 X.  
-> 최근의 변수를 사용하니까.

Static scope는 deep binding.

Nested subroutine을 pass하는 경우 closure의 reference environment는 일반적이지 않음.

the implementation of first-class subroutines is trivial in a language without nested subroutines  
->전역변수의 값이 바뀔 수 있지만 위치는 변경 X.

encapsulate our subroutine as a method of a simple object, and let the object's fields hold context for the method  
-> 객체를 만들고 객체에 nested subroutine에 있는 변수에 대한 개념을 변수에 넣고 그 다음 메소드를 포함해 객체를 넘긴다. -> 원래 호출 댔어야 했던 값을 가질 수 있다.

자바의 예시에서 closure는 함수로써 역할을 한다.  
이의 reference environment는 object enclosure or function object, functor의 3가지이름을 가진다.

자바의 예시에서 static은 제한적이다.  
이유는 2, 미리 정해준 값으로만 가능.

Lambda Expressions

익명함수를 lambda expression.

Delegate 역시 익명delegate이다.

Delegate가 =>가 되었음 해당 예제에서.

Lambda syntax에 관심을 가져야한다.

함수형 언어에서 함수를 값으로 처리할 수 있는 손쉬운 방법을 제공  
lambda expression을 잘 쓰는 새로운 함수를 실행 중에 만들어낼 수 있는 장점을 가짐.

함수를 값으로 처리하기 위해 쓰는 조작에 대해 imperative에서는 허용 X.-> 일반적 X.  
function에 대한 장점이 줄음. 잘 사용하면 코드 재사용 일반화 가능.

흔한 예가 call back함수. -> 필요할 때 함수를 만들어 제공.

-> 스트레티지 패턴 생각 compare만 갈아끼우기.

Call back에서 lambda가 유용하다.

Lambda의 왜 사용, 어떤 장점, 배웠던 reference environment와 무슨 관계에 대해 기억.

C++에서 function object는 ()를 override해서 함수처럼 호출.

Java는 lambda를 functional interface를 이용해 처리한다.

Java Arrays.sort는 static함수.

the compiler uses the lambda expression to create an anonymous class that implements the interface -> 자바 컴파일러는 lambda를 통해 익명클래스를 구현.  
functional interface를 구현하는 class를 만들고 그 클래스의 객체를 전달하는 경우에서 lambda 표현식을 사용이 가능하다.  
아니면 사용 불가. 극히 제한적이다. JVM에 의해서 그렇다.

자바 람다 표현식

한번보기.

Macro expansion

어셈블러: 어셈블리어를 컴파일해주는 것.

Macro expansion: code를 만들어놓으면 복사해 붙이기.

이의 문제: text를 바꿔서 치환하기에 문제가 발생.

최근은 macro를 빼고있다.

대신 named constant를 쓰고 있다. -> const int a = 3;

In-line subroutines -> macro처럼 동작, type을 검사해준다. 안전하다.  
속도는 비슷하다.

Control Flow 제어흐름.

exception handling and speculation-> 사용자가 입력을 제대로 할 것이라는 가정을 해서 프로그래밍을 하는데 조건이 만족을 안 할 때 handler 실행.

1. 핸들러 실행.
2. 해당 구간에서 종료.

Speculation: cpu가 미리 실행을 하는 것. 이문제는 심각 해진다.

Concurrency(1개의 cpu를 시간 나눠쓰기)(병행)와 pallor(여러 개 cpu)(병렬)은 다르게 취급할 수 있다.

Nondeterminacy

Statement나 expression에서 어떤 순서로 실행 먼저 실행할 것인지 정해져 있지 않다.

의도적으로 정하지 않은 것이다.  
같은 실행결과가 나올 것이다.

Expression Evaluation

인자를 가진 함수나 연산자를 표현식을 구성하는 요소이다.

연산자도 built-in function이다.

연산자도 함수이다.

연산자 2개는 infix, 1개는 prefix(ex +2, -2)

infix에서 괄호를 붙어야 할 경우도 있다.  
우선순위가 없을 때 괄호가 없다면 문제가 발생 가능.

Precedence(우선순위) and Associativity(연산순서)

Associativity언어마다 다르다.

Assignments

purely functional language  
-> 표현식이 프로그램을 구성하는 요소, computation이 값을 구하는 것으로 이뤄짐.

어떤 연산을 하고 이런 것을 함수에 전달해 값을 뽑는 것도 연산임  
그런 들이 전부 포함.

imperative language  
-> 메모리 값을 변경하면서 사용.

Side effect: 변수의 값이 변경.

함수에 값을 전달하고 함수에게 넘기는게 아니라면.

imperative language 표현식과 statement를 분리.

Expression = 값을 생성.  
statement는 아님.

References and values  
중요하다.

Reference to a value = l-value,  
named reference to a value는 r-value.

Value model과 reference model에 차이가 있다.

Reference는 가르킨다고 생각한다. (c의 pointer)

Value도 값이 있는 공간이 있는데 이 값을 복사해 저장한다 생각.

when it appears in a context that expects an r-value, it must be dereferenced to obtain the value to which it refers  
-> **아니면 모든 변수는 l-value로 취급.**

Reference와 value model을 사용하는데의 차이가 중요하게 작용하는 것이있음.  
가정1: 변수가 참조하고 있는 값을 그자리에서 변경가능하다 가정. (값 자체를 변경)  
가정2: 같은 값을 가지는 다른 object가 있을 수 있다. 가정.

이러면 같은 object를 가르키는지, 아니면 다른 object인데 같은 값인지를 구별해야 한다.

Java의 Arraylist는 object type을 받을 것이라 생각.

따라서 자동 boxing, 그리고 자동 unboxing해준다(object -> int,float)

Boxing -> 기본을 class형식으로, unboxing -> object, class ->기본 타입으로 변경.

Java는 1.5부터 boxing, unboxing, generic 제공.  
generic구현을 object를 사용해서 arraylist를 사용. -> 컴파일러가 data를 추가할 때 컴파일러가 확인하고 boxing을 거쳐야한다. 그러나 C#은 아니다.

Orthogonality -> 일관성? == 경계를 부시는 것이라 생각.

Assign에 대해서만 이야기한다.  
a를 배웠으면 같은 맥락b를 어떠할 것이라 생각.

Orthogonality: 언어의 기능들이 어떠한 조합으로도 사용가능 해야한다.-> 그 조합이 전부다 형식적으로 사용한다는 느낌을 주고 주어진 기능의 의미가 일관성 있어야한다. 다른 기능과 상관없이.

어떤 조합이 있든 일관성있게 의미하고 동작.

Expression-oriented => expression만 있음. Statement(문장)이라고 작성해야하는 context상에서 expression으로 표현가능.

-> if문을 표현식 안에서 사용가능???

A==B를 사용하고 싶은데 A = B를 사용하는 오류가 발생가능 C에서

Lack of Orthogonality in C

인자전달에서 구조체를 사용할 때 구조체 변수를 주는 것은 가능하나 {1,2}는 안된다.  
정수에선 가능했지만 구조체에서는 불가능했음 -> 일관성이 부족.

presuming the types match. C/C++ forbid arrays (and some forms of structs) to be assigned -> C에서 a[3] , b[3] 일 때 a=b불가능.

Assignments

b.c[3].d = b.c[3].d \* e; 일시 반복해서 쓰는게 불편.  
여기는 b.c.[3].d의 공간을 찾고 거기의 값과 e를 곱하고 다시 공간을 찾아야함.  
b라는 변수에서 -> c라는 구조체를 찾고 거기의 4번째의 주소를 가서 -> 그구조체의 d를 찾아야함. == redundant address calculation  
이때 +=을 제공.

Side-effect에서 redundant는 문제가 발생할 수 있음.

이를 위해 복합대입연산자를 제공.

배열이나 pointer를 사용할 때 increment,decrement를 사용하면 문제가 발생 할 수 있다.  
ex)a[b++] = c;

\*(t = p, p += 1, t) =>   
t= p  
p++  
\*t



this notation eliminates the asymmetry (nonorthogonality) of functions in most programming languages, which allow an arbitrary number of arguments but only a single return  
이는 함수입력값이 여러 개인데 리턴이 없거나 1개밖에 불가능한 상황을 해결해준다.

Initialization

초기값이 유용한 이유는 없으면 에러를 발생.

statically allocated but uninitialized variables are guaranteed to be represented in memory as if they had been initialized to zero  
C언어에서임 -> stack이나 heap은 아니다.

Definite assignment = 표현식에 사용되는 모든 변수에 값을 미리 assign 했어야 했다.  
conservation rule을 적용: 모든 것을 잡아내지 못함.

사람이 볼 때 괜찮은데 complier가 오류 발생.