프로그래밍언어(양승민)

Lee Jun Hyeok (wnsx0000@gmail.com)

June 7, 2025

목차

1. PL 1

PL(Programming Language)에 대해 알아보자.

1.1. 서론

1.1.1. 서론

1. PL Evaluation Criteria
PL에 대한 평가 기준은 여러 가지가 존재하는데, 아래와 같이 Readability, Writability, Reliability의 관점에서 분류할 수 있음.

	CRITERIA		
Characteristic	READABILITY	WRITABILITY	RELIABILITY
Simplicity	•	•	•
Orthogonality	•	•	•
Data types	•	•	•
Syntax design	•	•	•
Support for abstraction		•	•
Expressivity		•	•
Type checking			•
Exception handling			•
Restricted aliasing			•

여기에서 Orthogonality는 기본적인 것들만 존재하고, 나머지는 이를 활용하여 만들 수 있음을 의미함.

2. PL 종류
PL은 Imperative Language, Functional Language, Logic Language, Object Oriented Language로 총 4가지 종류로 분류할 수 있음.

obejct oriented language는 imperative language를 기반으로 구현되므로, 근본적으로는 imperative라고도 할 수 있음. 물론 imperative language와 관련이 없는 pure object oriented language도 존재한다고 함. 반면 functional language, logic language는 imperative language와는 관련이 없음.

functional language는 program을 함수의 조합으로 구성하는 language를 말함.

PL은 아니지만 최근에는 HTML 등의 Markup Language가 등장함. 물론 이런 language에도 programming 기능이 일부 포함되기도 하는데, 이런 것은 Markup-programming Hybrid language라고 함.

HTML은 hypertext를 위한 markup language임. 웹은 글 뿐만 아니라 소리, 이미지, 비디오 등을 포함하는 hypertext를 활용하고, 이를 위한 것이 HTML임. 이때 이미지는 픽셀로 나타내고, 소리는 quantization으로 나타내게 됨.

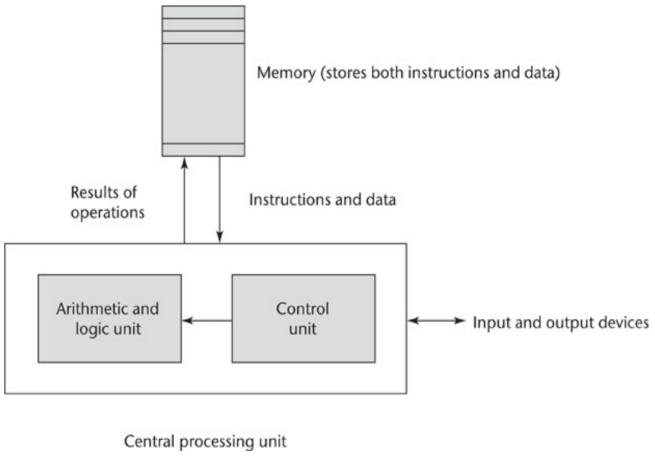
3. Programming Environment
Programming Environment는 소프트웨어 개발 도구의 집합임. 이는 단순히 file system, editor, linker, compiler만으로 구성될 수도 있고, 더 다양한 도구들을 포함할수도 있음.

UNIX, JBuilder(Java 관련), Visual Studio, NetBeans(Java 등)와 같은 programming environment들이 있음.

1.1.2. Von Neumann Architecture

1. Von Neumann Architecture
Von Neumann Architecture는 컴퓨터를 연산을 처리하는 cpu와, 데이터/program이 저장된 memory로 구하는 computer architecture로, 대부분의 상용 컴퓨터에서 사용해 온 computer architecture임. cpu와 memory는 bus로 연결되어 데이터/program을 주고받음.

물론 여기에는 모든 연산을 cpu에서 해야 한다는 단점도 존재하고, 이에 따라 cache 등을 활용함.



cpu와 memory가 구분되어 있고 memory에 데이터와 program이 저장되므로, 연산을 위해서는 memory의 instruction을 cpu로 올려(load) 수행하고, 그 결과를 다시 memory에 저장(store)해야 함. 이에 따라 von neumann arhitecture의 동작은 아래와 같이 다음으로 수행할 instruction을 가리키는 PC(Program Counter)를 활용한 Fetch-Execution(Instruction) Cycle의 반복임. 이는 stop instruction이 실행되거나, 제어(control)가 user로부터 os로 넘어갈 때까지 반복됨.

```
repeat forever
    fetch the instruction pointed to by the program counter
    increment the program counter to point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

이때 cpu는 control unit과 ALU로 구성됨. Control Unit 또는 Decoder는 opcode 등을 활용하여 instruction에 대한 decoding을 수행하는 부분이고, ALU(Arithmetic Logic Unit)은 실제 연산을 수행하는 부분임.

또한 cpu는 register를 포함함. register에는 PC, IR 등이 있음. IR(Instruction Register)은 instruction을 fetch에서 저장하는 register임.

instruction은 해당 instruction의 종류를 나타내는 opcode와 피연산자인 operand로 구성됨.

2. Imperative Language

PL은 computer architecture에 종속되어 설계 및 구현됨. von neumann architecture를 기반으로 설계되어, variable에 값을 할당하고 그 상태를 변경하는 것을 주 작업으로 하는 PL을 Imperative Language(명령형 언어)라고 함. 지금까지 설계된 대부분의 유명 PL는 imperative language임.

imperative language는 von neumann architecture에 대한 abstraction이라고 생각할 수 있음.

Functional Language(함수형 언어)는 주어진 parameter에 함수를 적용하는 작업을 주 작업으로 하는 PL임. 이 종류의 PL에서는 variable, assignment, iteration을 사용하지 않음. 물론 functional language가 여러 장점을 가지지만, von neumann architecture에서 이에 대해 충분히 효율적인 실행을 지원하지 못하므로 널리 활용되지 못했음.

예를 들어, 위와 같은 fetch-execution cycle에 따라 imperative language의 기본 문장이라고 할 수 있는 assignment(배정문) sum = sum + x - y를 기계어로 나타내면 아래와 같음. 즉, load와 연산, store의 반복임. 또한 추가로 제어문은 jump로 변환되는데, jump는 PC값을 바꾸는 연산으로 생각할 수 있음.

```
load r1, sum
load r2, x
add r1, r1, r2
load r2, y
sub r1, r1, r2
store sum, r1
```

1.1.3. program 실행 방법

PL로 작성된 program program을 실제로 실행하기 위한 방법으로는 아래와 같은 것들이 있음. 즉, 이것들은 high level language(ex. c, python, java 등)를 구현하는 방법임.

1. Compilation

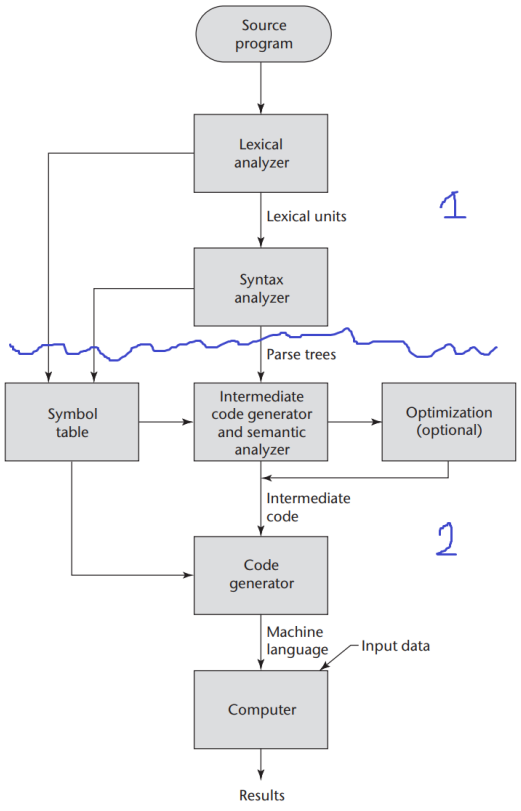
Compilation은 source language(PL)로 작성된 program을 컴퓨터에서 직접 실행시킬 수 있는 machine language로 변환(번역)하는 기법임. compilation을 수행하는 소프트웨어를 Compiler라고 함.

변환 이후에는 즉시 실행이 가능하므로 compilation은 실행 속도에서 이점이 있음.

compilation은 자연어의 번역과 대응되는 개념으로 이해할 수 있음. 번역은 문장 단위로 수행됨. 문장을 번역할 때는 우선 각 단어가 존재하는 어휘인지를 확인하고, 문법에 맞는지 검사하고, 이후 목표 언어로 변환함. PL에 대한 compilation도 동일한 과정을 거쳐 수행됨. 이때 PL에서는 문장을 Statement, 어휘를 Lexeme, 문법을 Syntax라고 함.

compiler의 동작 과정은 아래의 그림과 같음. 이는 lexical analyzer와 syntax analyzer를 포함하는 첫 번째 단계와, 그 이후 기계어를 생성하는 두 번째 단계로 구분할 수 있음.

더 구체적으로, 첫 번째 단계에서 lexical analyzer에서는 statement의 lexeme이 유효한지 검사하고, 이후 syntax analyzer에서는 statement가 syntax에 맞는지 확인함. 두 번째 단계에서 intermediate code generator는 intermediate code(중간 코드)를 생성함. 이는 machine language 수준의 코드이지만 machine(cpu)에 대해 독립적으로, machine에 종속적인 machine language와는 차이가 있음. 즉, cpu에 상관없이 intermediate code를 생성하고, 이후 code generator에서 cpu에 맞는 machine language를 생성함.



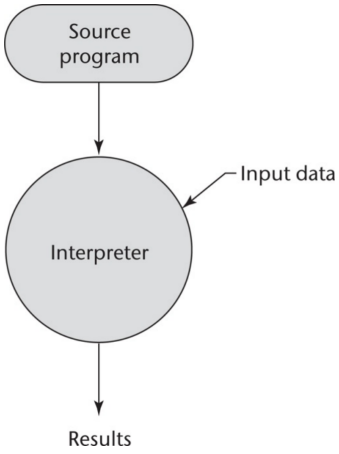
추가로, Preprocessor(전처리기)는 compile 직전에 실행되는 program임. preprocessor instruction은 program 내부에 존재하며, 기본적으로 매크로임.

2. Pure Interpretation

Pure Interpretation은 PL에 대한 번역 없이, Interpreter에 의해 interprete(이해 및 실행)되는 방식임. interpreter는 OS 위에서 실행되는 소프트웨어로, program과 cpu(정확히는 OS) 사이에서 동작하는

virtual machine으로 생각할 수 있음. 즉, 마치 fetch-execute cycle이 machine language에 대해서가 아니라, high level language에 대해서 수행되는 것처럼 동작함.

pure interpreter는 portability(이식성)가 높지만, compilation에 비해 훨씬 느림. 범용적으로 사용되는 language에 사용하기는 너무 느리지만, 최근에는 PHP, Javascript 등 web script language에서 많이 사용되고 있다고 함.

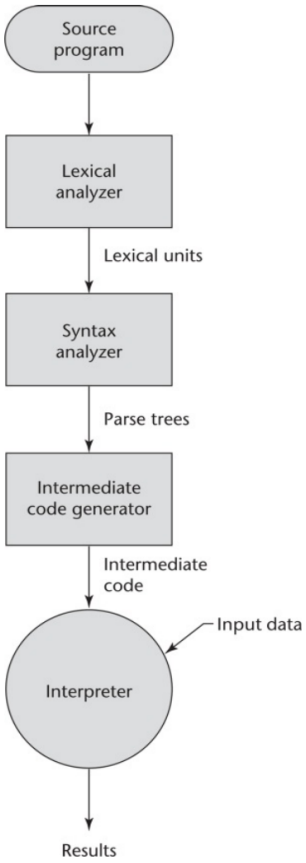


3. Hybrid Implementation

Hybrid Implementation은 compilation과 pure interpretation의 기법을 hybrid로 차용한 방식임. 즉, high level language를 intermediate code로 변환하고, 이를 interpretation에 활용함.

특히 Java가 이 방식을 사용함. java에서는 intermediate code를 byte code라고 하고, 이를 실행하는 byte code interpreter와 runtime system을 통틀어 JVM(Java Virtual Machine)이라고 함. 즉, software - byte code - JVM - OS - cpu의 구조로 실행됨.

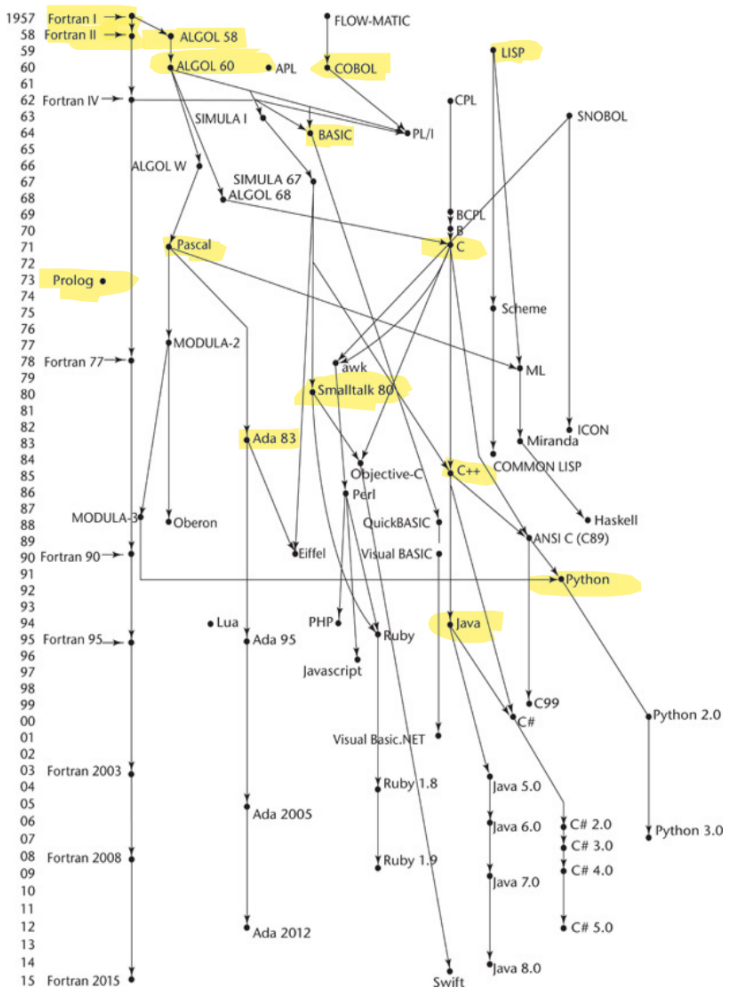
JIT(Just-In-Time)은 byte code를 바로 interprete하는 대신, 실행 시에 이를 machine language로 compile하여 활용하는 방식임. 이렇게 하면 더 빠르게 실행할 수 있다고 하며, java에서도 이를 사용함.



1.2. PL의 발전

1.2.1. PL의 발전

여러가지 PL들과 그 발전 과정을 알아보자. PL에 대한 genealogy(족보)는 아래와 같음.



1. FORTRAN

FORTRAN은 IBM에서 제시한 최초의 high level language임. 이는 IBM이 제시한 메인프레임 IBM 704를 기반으로 구현되었음.

FORTRAN은 수치 연산을 위한 language로, 간결한 문법을 가졌으며 compilation 기반임.

2. COBOL

COBOL은 business data 처리를 위한 language임.

FORTRAN은 수치 연산을 위한 것이었고, 이후 business 등에서도 데이터 처리에 대한 요구가 발생하여 만들어졌음.

3. ALGOL

ALGOL은 학문적 연구와 알고리즘 표현 등에 사용된 language임. ALGOL 자체는 성공하지 못했지만 이를 기반으로 C, Pascal 등이 등장함.

특히 ALGOL 68는 현대 범용 PL의 바탕임. ALGOL 68은 orthogonal design을 제공함. 즉, 최소의 구성 요소를 제공하고 이를 조합하여 사용하도록 했음. 모든 imperative language는 ALGOL 68 또는 ALGOL 60의 영향을 받음. C와 Pascal 모두 ALGOL 68로부터 비롯되었지만, Pascal이 좀 더 많이 닮음.

4. Basic

Basic은 단순성과 접근성에 집중한 interpreter 방식의 교육용 language임.

Basic은 이후 쉬운 GUI 개발을 제공한 VB(Visual Basic)으로 활용되었음. 또한 더 나아가 object oriented programming을 지원하는 VB.NET도 등장했음.

5. Pascal
Pascal은 단순함과 표현력에 집중한 ALGOL 기반의 교육용 language임.

C와 Pascal 모두 ALGOL에서 유래했는데, 한때 pascal이 더 우세했지만 Unix의 사용, 빠른 속도 등에 의해 2000년대부터는 c가 더 우세해짐.

6. Prolog
Prolog는 비절차적인 접근을 활용하는 logical programming language임.

7. Lisp
Lisp는 최초의 functional language로, AI 관련 연산 등을 위해 만들어졌음.

8. Smalltalk
Smalltalk는 pure obeject oriented language로, object oriented language의 시초임.

Smalltalk의 등장에 의해 C++, 이후 java가 등장 했음. 또한 c++과 Pascal을 기반으로 python이 등장함.

9. Ada
Ada는 DoD가 만든, 임베디드 시스템에서의 안전성과 신뢰성에 집중한 language임.

DoD(Department of Defense. 국방부.)에서 대부분의 program은 임베디드 program이었음. 이에 따라 DoD는 Pascal을 확장하여 임베디드 program을 위한 언어로 Ada를 제작함.

10. C++
C++은 imperative language(C)로서의 기능과 object oriented language(Smalltalk)로서의 기능을 결합한 language임.

11. Java
Java는 C++을 기반으로 만들어진 imperative-based object Oriented language임.

앞에 정리한 것처럼 Java는 byte code를 활용해 portability를 활용함. 이때 이렇게 portability와 interpretation에 따른 비용을 맞바꾼 것이 Java의 design에 따른 특징은 아님. 어떤 language도 intermediate code를 활용해 임의의 platform에서 실행시키도록 할 수 있음.

최초의 Java는 가전제품 소프트웨어를 위해 만들어졌지만, 이후 web 개발에 활용되면서 그 사용량이 급격히 증가했음.

12. Script Language
Script Language는 작업 자동화, 간단한 program 작성, 기존 소프트웨어 제어 등에 활용되는 progromming language임. 예를 들어, Javascript, PHP, Perl 등이 있음.

최초의 script language는 명령어의 집합인 Script에 대한 interpretation에 사용되었음. 예를 들어, UNIX 시스템에서 sh(Shell. command interpreter.)은 유틸리티 기능을 수행하는 script language임. sh를 사용하면 이미 존재하는 built-in program만이 아니라, 사용자가 정의한 내용을 실행할 수 있음. 이후 이런 간단한 형태에서 variable, flow control, function 등이 추가되어 programming language가 되었음.

13. Javascript
Javascript는 web 프로그래밍에서 동적인 HTML 문서를 만들 때 사용되는 script language임.

1990년대 web이 빠르게 발전하며 그래픽 기반의 브라우저들이 등장했음. 하지만 HTML 자체는 정적이므로, 이에 대한 추가적인 연산이 수행되어야 했음. CGI(Common Gateway Interface)를 사용하면 해당 연산을 서버에서 수행하고 그 결과를 반환받도록 할 수 있었지만, 대신 Javascript 등을 활용한 브라우저에서의 연산을 활용하게 되었음.

14. C샵
C#은 C++과 Java를 기반으로 하면서, VB 등으로부터 영향을 받은 language로, 모바일, web, 게임, 클라우드 등의 어플리케이션 개발에 사용됨.

C#은 주로 .NET과 함께 사용되는데, .NET은 마이크로소프트가 개발한 소프트웨어 개발 프레임워크로, 실행 환경과 라이브러리 등을 포함함.

1.3. Syntax and Semantics

1.3.1. Syntax Description

1. Syntax Description
Syntax Description은 PL로 작성된 문자열에 대해, 어떤 syntax 구조를 가지고 있는지 설명하고 또한 유효한지 검증하는 것임. 즉, token의 순서와 구조를 확인하여 formal하게 표현하는 것으로, 이는 ALGOL이 등장하면서 처음 시도되었음.

Syntax는 PL에 대한 문법으로, expression, statement, program unit에 대한 형식임. Semantics는 PL의 expression, statement, program unit이 가지는 의미임.

여기에서 description은 syntax나 semantics에 대해 형식 또는 의미를 설명하는 것을 말함. syntax에 비해 semantics에 대한 describe가 더 어려운데, 이는 syntax description에 대한 간결하고 널리 사용되는 notation은 존재하지만, semantics에 대한 것은 존재하기 않기 때문임.

grammer를 입력으로 제공하면 compiler에서 해당 language에 대한 syntax analysis를 수행하는 부분을 자동 생성하는 알고리즘이 예전부터 여러 개 존재해 왔음. 예를 들어, syntax analyzer generator(즉, compiler에 대한 compiler.)인 yacc은 1975년에 제시되었음. 이는 뒤에서 설명할 LR parsing table을 자동 생성함.

2. Lexeme과 Token
Lexeme은 PL에 대한 단어로, 가장 작은 단위의 syntatic unit임. 예를 들어, 각 numeric literal, 연산자, keyword(ex. if, while), identifier 등이 있음. 일반적으로 syntax description에서는 lexeme 자체의 형태에 대한 description을 수행하지 않고, 이는 lexical specification에서 수행됨.

Token은 PL에 대한 품사로, lexeme에 대한 category임. syntax description을 위해서는 token을 활용해야 함. 예를 들어, Identifier(식별자)는 variable, method, class 등을 식별하기 위한 이름을 나타내는 token의 한 종류임.

3. Grammer
Grammer는 syntax description에 사용되는 형식적인 규칙임. 이는 language generation mechanism으로도 이해할 수 있음.

노암 촘스키 등의 학자는 언어를 4단계로 나눴음. 그 중 context free는 가장 낮은 단계의 언어로, 문맥에 상관없이 단어의 의미가 하나만 존재하는 언어을 말함. 그 반대는 context sensitive로 문맥에 따라 의미가 여러 개인 언어를 말함. PL는 context-free로, 쉽게 번역할 수 있음.

예를 들어, index = 2 * count + 17;이라는 statement에서, 각 lexeme과 그 token은 아래와 같음.

Lexemes	*Tokens*
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

1.3.2. BNF

1. BNF
BNF(Backus-Naur Form)는 syntax description에 사용되는 natural notation임. 이는 rule의 집합인 context-free grammer로 이해할 수 있음. 또한 BNF는 PL에 대한 metalanguage(language를 describe

하는 languge)임.

BNF는 실제 문장이 주어지지 않았을 때 grammer를 활용해 해당 language에 속하는 모든 유효한 statement를 생성할 수 있고, 이를 해당 language에 대한 연구 및 이해에 활용할 수 있음. 실제 statement가 주어진 경우에는 해당 statement가 derivation에 의해 생성될 수 있는지를 확인하는 것으로 syntax description 및 검사를 수행할 수 있음.

BNF는 3가지 notation만 사용함.
1) -> : Rule을 정의함.
2) <> : Non-Terminal(pointed braket)을 나타냄.
3) | : Or의 기능을 함. 여러 가지가 존재하면 여러 번 쓰는 대신 |로 나누어서 표기함.

BNF는 아래와 같이 이런 notation을 활용해 syntatic structure를 표기함. 여기에서 화살표 왼쪽 부분인 LHS(Left-hand Side)는 해당 rule에서 정의되는 abstraction을 나타냄. 오른쪽 부분인 RHS(Right-hand Side)는 token, lexeme, 다른 abstraction에 대한 참조로 구성되어 해당 abstratcion을 정의함. LHS와 RHS로 구성된 정의를 Rule 또는 Production이라고 함.

정의된 abstraction을 Nonterminal Symbol이라고 하고, lexeme과 token을 Terminal Symbol이라고 함. extended BNF를 고려하지 않았을 때, BNF notation과 nonterminal을 제외한 나머지 symbol은 모두 terminal임.

```
<assign> -> <var> = <expression>
```

추가로, LHS의 nonterminal을 RHS에 작성하여 recursion을 구성할 수 있고, 종료 조건(Termination Condition)이 나올 때까지 임의의 길이를 가지는 작업을 처리할 수 있음.

또한 어떤 연산이 두 가지 nonterminal로 구성된 recursion에서 LHS의 nonterminal이 등장하는 위치에 따라 left recursive와 right recursive로 나눌 수 있는데, 이에 따라 연산자의 Precedence(우선순위)가 정해짐. Left Recursive는 LHS의 nonterminal이 왼쪽에 나오는 것이고, 왼쪽이 먼저 연산되므로 이는 해당 연산을 Left Associativity(좌결합)로 구현함. Right Recursive는 LHS의 nonterminal이 왼쪽에 나오는 것이고, 오른쪽이 먼저 연산되므로 이는 해당 연산을 Right Associativity(좌결합)로 구현함. 대부분의 연산자는 좌결합이지만 우결합하는 것(ex. 지수 연산)도 있으므로 그 종류에 따라 rule이 작성될 수 있음.

```
<ident_list> -> identifier | identifier, <ident_list>

<factor> -> <exp> ** <factor> | <exp>              (right recursive)
<factor> -> <factor> ** <exp> | <exp>              (left recursive)
```

2. Derivation
Derivation(유도)은 grammer의 rule들을 적용해 start symbol로부터 시작해 nonterminal들을 terminal로 변환하는 과정임. 또한 language 또는 parse tree를 생성하는 과정으로도 이해할 수 있음. program별로 grammer는 하나만 존재하므로, derivation은 단순히 자동화될 수 있음.

Start Symbol은 derivation의 시작점인 특별한 nonterminal로, 전체 program을 나타냄. 여기에서는 아래와 같이 <program>으로 나타냄.

```
<program> -> begin <stmt_list> end
```

derivation의 각 단계의 문자열을 Sentential Form이라고 함. derivation은 아래와 같이 sentential form에 nonterminal이 존재하지 않을 때까지 수행됨. 즉, nonterminal은 최종 결과에 드러나지 않음.

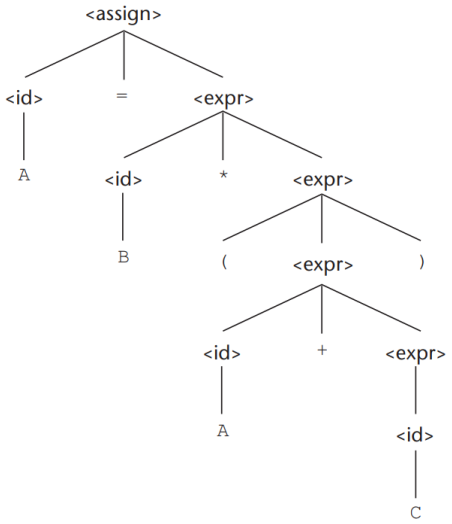
여기에서와 같이 여러 개의 nonterminal이 존재할 때 가장 왼쪽에 있는 것부터 변환하는 방식을 Leftmost Derivation이라고 함. leftmost 말고도 rightmost 등 여러 순서 규칙이 존재할 수 있는데, grammer에 의한 language 생성에는 영향을 미치지 않는다고 함. 여기에서는 leftmost를 기준으로 설명함.

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

3. Parse Tree

Parse Tree는 grammer를 활용한 derivation에 의해 생성되는 hierarchical syntactic structure로, 해당 language에서 정의되는 statement에 대한 derivation을 시각화한 것임.

이는 root인 start symbol로부터 시작하여 각 nonterminal을 rule에 의해 terminal과 nonterminal을 자식으로 갖도록 구성하며, nonterminal이 모두 사라질 때까지 이 과정을 반복함. 즉, leaf는 terminal임. 여기에서도 leftmost(pre-order)를 기본으로 설명함.



4. Ambiguity

Ambiguity는 어떤 grammer에서 하나의 문장에 대해 여러 개의 parse tree가 나오는 상황을 말함. 이와 같이 여러 의미로 해석될 수 있는 grammer는 불확실성이 존재하고, 유지보수가 어려울 수 있음.

*예를 들어, 아래와 같은 경우 <expr>은 첫 번째로도 recursion될 수 있고, 두 번째로도 recursion될 수 있음. 이에 따라 A=B+C*A와 같은 statement에 대해서 여러 개의 parse tree가 존재하게 됨.*

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

이는 아래와 같이 nonterminal을 하나 더 사용하여 abiguity를 제거할 수 있음.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

참고로 terminal은 말단이라는 뜻임.

또한 if-else에 대해서도 아래와 같이 rule이 구성되면 if가 중첩되는 경우 else가 어디에 붙어야 하는지에 대해 ambiguity가 존재함. else는 가장 가까운 if에 붙어야 함.

<if_stmt> → **if** (<logic_expr>) <stmt>
if (<logic_expr>) <stmt> **else** <stmt>

If we also have <stmt> → <if_stmt>, this grammar is ambiguous. The simplest sentential form that illustrates this ambiguity is

if (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>

1.3.3. EBNF

EBNF(Extended BNF)는 BNF를 더 편리하게 사용하기 위해 BNF를 확장한 것임. 이는 description power를 증가시키지는 않고, 단순히 readability와 writability를 향상시킴.

EBNF에서는 BNF에 추가로 아래와 같은 notation을 추가로 사용함. 물론 그 종류와 버전에 따라 제공되는 notation은 상이할 수 있음.

1) [] : zero or once. 즉, 해당 부분이 한 번 나타나거나, 나타나지 않음.

```
<if_stmt> -> if (<expression>) <statement> [else <statement>]
```

2) {} : zero or many. 즉, 해당 부분이 한 번 이상 나타나거나, 나타나지 않음.

```
<ident_list> -> <identifier> {, <identifier>}
```

3) + : one or many. 즉, 해당 부분이 한 번 이상 나타남. 즉, 아래의 두 문장은 동일함. 참고로, 여기에서 begin과 end는 c에서 {}에 해당되는 symbol임.

```
<compound> -> begin <stmt> {<stmt>} end

<compound> -> begin {<stmt>}+ end
```

4) (a|b) : a or b. 즉, 해당 부분 중 하나가 나타나야 함. 특히 동일한 형태의 연산자를 나타내는 rule에 유용함.

```
<expression> -> <term> {(+ | -) <term>}
```

EBNF를 사용하는 경우 아래와 같이 작성이 될 수 있는데, 이러면 associativity가 명확히 드러나지 않는다는 문제가 존재함.

```
<expression> -> <term> {+ <term>}
```

참고로, 앞에서는 자신이 자신을 호출하는 직접 재귀를 살펴봤지만, 아래와 같이 자기 자신을 직접 호출하는 것이 아니라 여러 가지 단계를 거쳐 재귀되는 간접 재귀도 존재함. (<expr>)가 나오면 재귀 호출이 수행되고, id가 나오면 끝남.

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>)
| id

1.4. Lexical/Syntax Analysis

compiler의 나머지 부분과 더 자세한 내용은 추후 컴파일러 수업에서 배우도록 하고, 여기에서는 그 앞쪽 부분인 lexical/syntax analysis에 대해 알아보자.

1.4.1. Lexical Analysis

1. Lexical Analysis
Lexical Analysis는 주어진 문자열(코드)에 대해 정해진 문자열과 일치하는지를 확인하는 pattern matching 과정임.

lexical analysis를 수행하는 부분(함수)을 Lexical Analyzer 또는 Lex라고 함. lex의 입력으로는 문자열이 주어짐. lex는 특정 문자열들을 논리적으로 grouping하고, 내부적인 code를 할당해 저장하고 있는데 이때 각 문자열이 앞에서 다룬 lexeme이고, group이 token임. 이때 lex는 syntax analyzer에 포함되어 front end로 기능함. 즉, lex는 parser에 의해 호출됨.

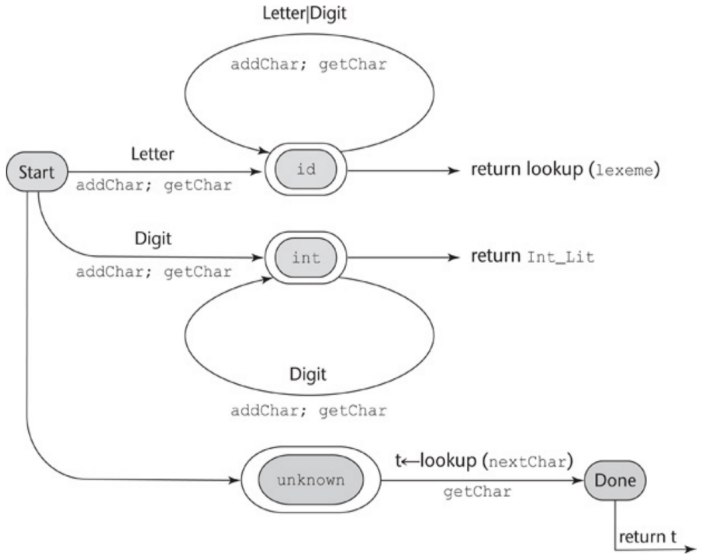
즉, lex는 아래와 같은 작업들을 수행함.
1) 문자열에서 주석과 white space를 제거함.
2) 문자열에서 lexeme을 찾음.
3) 유효한 lexeme인지를 검사함. 이는 lexeme을 찾는 작업에 의해 동시에 수행됨.
4) lexeme 중 user-defined name들을 compiler의 이후 단계에서 활용하도록 symbol table에 입력함.

lex는 찾은 lexeme에 token을 붙여서 parser로 반환함.

lex를 만들 때에는 token 패턴에 대한 formal description을 작성하여 이를 구현하거나, state diagram을 설계한 뒤 이를 구현할 수 있음.

2. State Diagram
State Diagram 또는 State Transition Diagram은 state(상태)를 나타낸 node와, state 변환을 유발하는 event와 그때 수행되는 action을 arc로 나타낸 방향 그래프임. 이를 활용해 lex의 동작을 정의 및 표현할 수 있음. 이때 처음 state를 Initial State, 마지막 state를 Final State라고 함.

state diagram으로 아래와 같이 lex의 동작을 나타낼 수 있음. 문자열의 각 문자가 event로 기능하고, 이는 switch문으로 간단히 구현이 가능함. 실제로는 letter와 digit 말고도 여러 가지로 나뉘지만, 여기선 간단하게 unknown으로 처리함. 언어별로 identifier 명명 규칙이 존재하는데, lex는 어떤 문자열이 identifier로 판단되면 해당 규칙에 맞지 않는 순간까지 문자를 읽어서 identifier를 하나 찾은 것으로 침.



해당 state diagram을 구현한 코드는 교재에 있으니 참고하자.

1.4.2. Syntax Analysis

Syntax Analysis 또는 Parsing은 lex으로부터 받은 정보를 활용해 parse tree를 생성하는 과정임. 이를 수행하는 부분을 Parser라고 함.

더 구체적으로, parsing은 아래와 같은 두 가지 목적을 가짐.

1) syntax에 부합하는지 검사함. 검사 도중 error가 발견되면 diagnotic message를 생성함. 이후 error를 recovery하고 계속해서 parsing을 수행하는데, 이는 compiler가 한 번의 analysis에서 program 전체에 대해서 검사를 수행하기 위함임.

2) syntax에 부합하는 입력에 대해서 온전한 parse tree를 생성. parsing은 parse tree를 생성하는 방향에 따라 top-down(root에서 leaf로.)과 bottom-up(leaf에서 root로.)으로 구분됨. 이때 parse tree 자체를 명시적으로 만드는 대신 관련 정보만 생성하기도 함.

parser 이후에 compiler는 중간 코드를 생성함. 실제로 parser는 중간 코드를 위한 사전 작업도 동시에 수행하는데, 여기에서 이것까지 다루지는 않는다.

1.4.3. Top-down Parsing

1. Top-down Parsing

Top-down Parsing은 start symbol로부터 시작하여 leftmost derivation을 수행하며 parse tree를 구성하는 방식임. 즉, 존재하는 모든 nonterminal을 pre-order(root, left, right 순으로 순회함.)로 해당 rule의 오른쪽 부분으로 변환하는 작업을 반복하며 parse tree를 구성함.

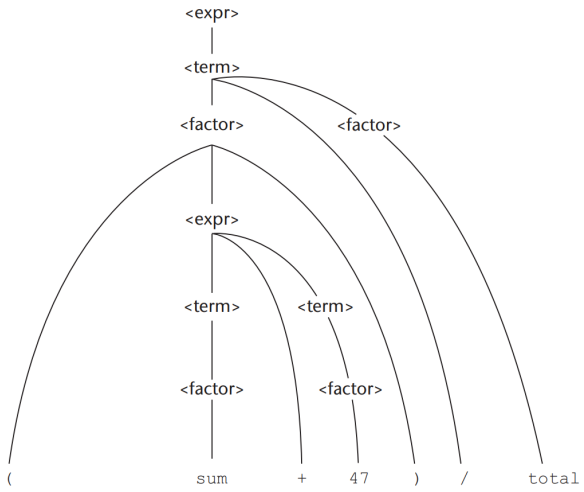
top-down parsing을 수행하는 알고리즘을 LL(Left-to-right and Leftmost derivation) 알고리즘이라고 함. 대표적인 LL 알고리즘에는 아래에서 설명할 recursive-descent parser와, BNF를 구현한 parsing table을 이용하는 방법이 있음.

LL 알고리즘은 leftmost derivation을 수행하므로, 기본적으로 left recursion이 존재하는 PL은 처리할 수 없음. left recursion의 경우 종료 조건이 오른쪽에 위치하는데, leftmost에서는 매번 왼쪽부터 처리하므로 종료 조건을 만나지 못해 무한 루프에 빠지게 됨. 이는 간접 재귀에서도 마찬가지임.

2. Recursive-descent Parser

Recursive-descent Parser는 BNF를 활용하여 단순 구현한 top-down parser임.

recursive-descent parser에서 각각의 nonterminal은 하나의 subprogram을 가짐. 입력 문자열이 들어왔을 때, 각 subprogram은 서로를 호출하며 해당 nonterminal을 root로 하면서 문자열과 일치하는 leaf를 가지는 parse tree를 찾음. 즉, nonterminal을 RHS를 그 내용으로 하는 일종의 함수로 취급함.



예를 들어, 아래와 같이 rule에 대해 subprogram(함수)이 정의될 수 있음.

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void  expr() {
  printf("Enter <expr>\n");

/* Parse the first term */
  term();

/* As long as the next token is + or -, get
   the next token and parse the next term */
  while  (nextToken == ADD_OP || nextToken == SUB_OP) {
    lex();
    term();
  }
  printf("Exit <expr>\n");
}  /* End of function expr */
```

subprogram은 함수로 이해할 수 있음.

1.4.4. Bottom-up Parsing

Bottom-up Parsing은 leaf로부터 시작하여 root까지 parse tree를 구성하는 방식임.

더 구체적으로는, leaf의 terminal로부터 시작해 reduce를 반복해 결국 start symbol을 얻는 과정임. 이 과정은 rightmost derivation의 반대 작업임. 여기에서 Reduce는 sentential form에서 RHS를 찾아 해당 rule의 LHS로 바꾸는 것을 말함. 또한 sentential form에는 하나 이상의 RHS가 존재할 수 있는데, 실제로 변환을 수행해야 하는 RHS를 Handle이라고 함. 즉, bottom-up parsing의 핵심은 handle을 찾는 것임.

bottom-up parsing을 수행하는 가장 대표적인 알고리즘이 LR(Left-to-right and Rightmost derivation) 알고리즘이고, 대부분의 기법들은 LR 알고리즘의 변형이라고 함. 최초의 LR 알고리즘은 Donald Knuth의 canonical LR인데 이는 많은 computation과 memory를 요구해 널리 쓰이지는 못했고, 이후의 LR 알고리즘들은 이를 개선했다고 함.

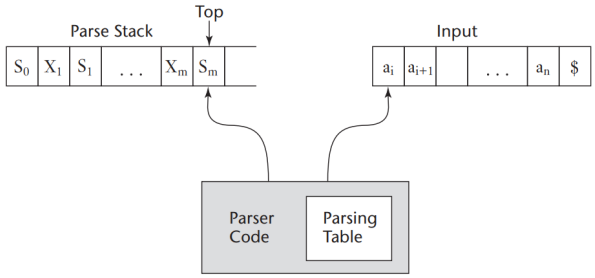
1.4.5. Bottom-up Parsing : LR Parser

1. LR Parser
LR Parser는 PL에 대한 LR parsing table과 parsing stack을 활용하는 bottom-up parser임. 각 input과 그에 따른 state(Parse State)를 관리하며 parsing을 수행함.

LR parser는 모든 PL에 대해 만들어질 수 있고, 함수 호출과 재귀가 없으므로 top-down보다 빠르면서,

오류를 찾아내기도 쉬움. 또한 LR parser는 left recursion이 포함된 PL도 처리할 수 있어, 처리할 수 있는 PL 집합이 LL parser의 superset임. LR parser의 유일한 단점은 LR parsing table을 직접 구성하기 까다롭다는 것인데, 현재는 grammer를 입력으로 넣으면 LR parsing table을 생성해주는 여러 program들(ex. yacc)이 존재해 문제되지 않음.

LR parser와 parsing table은 아래와 같은 구조를 가짐.



	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

아래와 같은 과정에 의해 parsing이 수행됨.

1) initial state(0)이 stack에 들어가 있는 상태로 시작함.
2) stack에서 맨 위의 state를 꺼냄. input과 해당 state에 따라 ACTION tabel에 따른 shift/reduce를 수행함.
3) $가 input으로 들어와 결국 accept될 때까지 2번 작업을 반복함.

2. Parse Stack

Parse Stack은 bottom-up parsing에 따른 변환 결과를 저장하는 stack으로, 입력된 terminal/nonterminal과 state를 순서대로 저장함(state를 마지막에 넣음.). 여기에서 stack의 맨 윗 값인 state는 현재 state를 의미함.

이때 Initial State(시작 상태)는 0임. 즉, 처음에는 stack state 0만 들어 있음.

3. LR Parsing Table

LR Parsing Table은 특정 PL에 대해 생성되어 LR parser에서 parsing을 수행하는 데 사용되는 table임. 이는 ACTION과 GOTO로 구성되며, row는 state를 나타냄.

ACTION은 parser의 action(동작)을 정의하는 핵심 부분으로, column을 해당 grammer의 terminal symbol로 함. 이때 $는 EOF로, 마지막임을 나타냄. input으로 들어온 다음 symbol(token)과 현재 state에 대응되는 action을 수행하도록 함. action에는 아래와 같이 S(shift)와 R(reduce)가 있음.

1) shift : state와 input을 순서대로 stack에 입력하는 연산임. 이는 S와 숫자로 표기하는데, 해당 숫자의 state를 stack에 넣는다는 것임. 즉, input을 받고 대응되는 state로 변환됨.

2) reduce : stack의 맨 위에 존재하는 handle을 LHS로 변환하는 연산임. 이는 R과 숫자로 표기하는데, 이는 grammer 중 해당 숫자의 rule에 대응되는 변환을 수행한다는 것임. 이후 변환에 의해 생성된 nonterminal과, GOTO에 의해 결정되는 state를 순서대로 stack에 넣음.

이때 handle에 해당하는 부분을 넣기 전에 stack 가장 위에 있는 state(ACTION에서 쓴 state와는 구분됨.)로 새로운 state를 결정함. 또한 여러 terminal/nonterminal로 구성된 rule을 활용해 변환하는 경우, stack에서 중간에 존재하는 state들은 무시함. 이때 당연하게도 reduce 시에는 변환한 nonterminal만을 stack에 넣고, input은 넣지 않음(input은 shift에서 들어감.).

GOTO는 reduce가 완료된 이후 변환된 LHS와 함께 stack에 넣어야 하는 state를 나타내며, column을 해당 grammer의 nonterminal로 함. 즉, reduce 연산에서는 rule만을 나타냈으므로 goto table을 사용해 state와 nonterminal에 따라 이후 어떤 상태로 갈지 정함.

예를 들어, 위의 table을 활용해 id+id*id$를 아래의 과정과 같이 parsing할 수 있음. 해당 과정에서 수행되는 reduce로 parse tree를 그릴 수 있음.

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

1.5. Name, Binding, Scope

variable에 대해 알아보자.

1.5.1. Name

Name 또는 Identifier는 program의 entity(object)를 구분하기 위해 사용하는 문자열임. 변수명, 함수명 등이 있음.

Reserved word(예약어)는 예약이 되어 있어 identifier로 사용이 불가능한 단어임. reserved word 중 Keyword는 특정 기능이나 역할을 수행하는 것임.

1.5.2. Variable

Variable(변수)은 memory cell 또는 memroy cell의 집합에 대한 abstraction임. variable은 총 6개의 attribute인 (name, address, value, type, lifetime, scope)로 정의 및 구분될 수 있음.

assignment에서의 위치에 따라 variable의 address는 l-value로, value는 r-value로 부르기도 함.

Alias는 하나 이상의 variable name이 동일한 memory로의 접근에 사용될 수 있는 경우 해당 variable들을 말함. alias에 의해 서로 다른 variable이 동일한 address를 가지는 경우가 발생할 수 있음. alias가 발생할 수 있는 상황으로는, union type을 사용하거나, 2개의 pointer를 사용하거나, call by reference를 활용하는 경우가 있음.

1.5.3. Binding

Binding은 entity(variable 등)와 attribute(type, storage, value 등)의 결합을 말함. binding이 생성되는 시점을 Binding Time이라고 함.

binding 중 runtime 이전에 생성되어 program 실행에 도중에 수정되지 않는 것을 Static Binding, runtime에 생성되어 program 실행 도중에 수정될 수 있는 것을 Dynamic Binding이라고 함.

여기에서는 type binding과 storage binding에 대해 살펴보자.

1. Type Binding
variable은 type binding이 수행되어야 reference가 가능함. variable에 대한 type binding은 static type binding과 dynamic type binding으로 나뉘는데, type이 결정되는 방식과 결정되는 시점의 측면에서 이해할 수 있음.

c, java 등 많은 언어들은 static binding을 기본으로 하고, interpreter 방식인 웹 기반 언어들은 타입에 대한 명시가 의미없는 경우가 많아 dynamic binding을 사용함. 특히 compiler 방식보다 pure interpreter 방식의 PL들에서 dynamic binding을 주로 구현해 사용한다고 함.

1) Static Type Binding
static type binding은 explicit declaration과 implicit declaration으로 생성될 수 있음. Explicit Declaration은 variable의 name과 type을 직접 명시하는 statement를 사용하는 방식이고, Implicit Declaration은 별도의 statement 없이 정해진 규칙(convention)에 따라 variable의 type이 정해지는 방식임.

2) Dynamic Type Binding dynamic type binding은 assignment에 의해 생성될 수 있음. 즉, 어떤 variable에 대한 type binding이 특정 값을 assign할 때 해당 값의 type에 따라 생성됨.

dynamic type binding은 program에 flexibility를 제공함. 하지만 error detection이 어려워 reliablity가 낮아지고, runtime에 임의의 type을 부여할 수 있어야 하므로 구현 비용이 높음.

2. Storage Binding
storage binding과 관련하여 Allocation은 가용한 memory pool에서 memory cell을 꺼내 binding에 사용하는 것을 의미하고, Deallocation은 해제된 binding의 memory cell을 memory pool에 반납하는 것을 의미함.

어떤 variable에 대한 Lifetime은 해당 variable이 특정 memory에 binding되어있는 시간을 의미함. 즉, binding된 시점부터 unbinding된 시점까지의 시간임.

storage binding에 따라 variable을 아래와 같이 4가지로 분류할 수 있음.

1) Static Variable
Static Variable은 static binding에 의해 memory에 binding되는 variable임. 예를 들어, global variable(전역변수)나, history sensitive한 local static variable(c에서의 static variable)에 사용됨.

History Sensitive는 함수가 종료되어도 값을 기억해야 하는 것을 말함. 반댓말은 Memoryless로, 함수가 종료되면 그 이후부터는 해당 값을 기억하지 않는 것을 말함.

2) Stack-Dynamic Variable
Stack-Dynamic Variable은 declaration 시에 storage binding이 생성되지만, type binding은 static하게 생성되는 variable임. 이렇게 runtime에 declaration에 의해 수행되는 allocation 및 binding 과정을 Elaboration이라고 함. 해당 memory는 runtime stack에서 allocation됨. 예를 들어, c에서의 지역변수가 있음.

3) Explicit Heap-Dynamic Variable
Explicit Heap-Dynamic Variable은 프로그래머가 explicit runtime instruction을 사용하여 memory를 allocation/dealloctation하는 variable임. 이는 name이 존재하지 않아 pointer나 reference variable로 reference해 사용함. 해당 memory는 heap에서 allocation됨. 예를 들어, c++에서는 new와 delete로 이를 구현함.

4) Implicit Heap-Dynamic Variable
Implicit Heap-Dynamic Variable은 해당 variable에 값을 assign할 때 storage binding이 생성되는 variable임. 해당 memory는 heap에서 allocation함.

3. Named Constant
Named Constant는 value에 대한 binding을 한 번만 가지는 variable임. 이를 활용하여 readability와 reliability를 확보할 수 있음. 예를 들어, 각 PL에서는 const, final, read-only 등의 키워드로 표기함.

1.5.4. Scope

Scope는 해당 variable이 visible한 statment의 범위임. Visible하다는 것은 해당 statement에서 reference 또는 assignment할 수 있다는 것을 말함.

Local Variable은 program unit 또는 block 내부에서 선언된 variable임. Nonlocal Variable은 program unit 또는 block 외부에서 선언되어 다른 program unit 또는 block에서도 visible한 variable임. Global Variable은 모든 program unit 또는 block에서 visible한 variable로, non-local variable의 부분집합임. 예를 들어, nesting이 존재하는 경우 부모 함수의 variable은 자식 함수에 의해 접근이 가능한데, 이는 global이 아니라 non-local임.

어떤 variable이 local/nonlocal/global인지는 해당 variable의 scope에 의해 결정됨. scoping 기법으로는 아래와 같이 static scoping과 dynamic scoping이 있음.

참고로, scope와 lifetime은 서로 독립적인 개념임.

1. Static Scope
Static Scoping은 nonlocal variable의 scope를 해당 variable의 코드 상 위치에 의해 static하게(runtime 이전에) 결정하는 방식임. 즉, 이 경우 visability가 코드의 구조에 의해 정적으로 결정됨. 이는 ALGOL 60에서 처음 제시된 것으로, 여러 PL에서 차용하고 있음.

static scoping은 subprogram의 nesting이 가능한지, 가능하지 않은지에 따라 두 가지 분류가 존재하는데, 여기에서는 nesting을 허용하는 분류에 대해서 설명함. 해당 분류에서는 어떤 variable에 대한 reference statement 등이 사용된 경우, 해당 subprogram 내에서 대응되는 declaration을 찾음. declaration을 찾지 못했다면 그 Static Parent(subprogram을 선언한 subprogram)로 가서 다시 찾는 과정을 반복함. 이때 가장 바깥쪽의 static parent는 Static Ancestor라고 함.

Block structured Language에서는 Block이라는 코드 영역을 사용하여, block에 대한 static scoping을 적용함. block 내의 stack dynamic variable은 제어가 block에 들어오면서 allocation되고, block이 종료되며 deallocation됨. 예를 들어, C에서는 nesting이 불가능하지만, block은 사용할 수 있음.

2. Dynamic Scoping
Dynamic Scoping은 nonlocal variable의 scope를 코드 상에서의 위치가 아니라, 각 subprogram의 호출 순서에 따라 dynamic하게(runtime에) 결정하는 방식임. 호출된 이후 종료되지 않은 중인 함수를 active된 함수라고 하는데, dynamic scope에서는 모든 active된 함수의 변수는 접근 가능한 것임.

dynamic scoping에서는 어떤 variable에 대한 reference statement 등이 사용된 경우, 해당 subprogram 내에서 대응되는 declaration을 찾음. declaration을 찾지 못했다면 해당 subprogram을 호출한 subprogram으로 넘어가 declaration을 찾는 과정을 반복함.

dynamic scoping을 쓰는 언어는 잘 없음.

3. Referencing Environment
어떤 statement에 대한 Referencing Environment는 해당 statement에서 visible한 전체 variable 집합을 말함. scoping 기법에 따라 visibility가 결정되므로, referencing environemnt도 scoping 기법에 따라 결정됨.

referencing environment는 static scoping에서는 해당 subprogram과 모든 static parent의 variable로 구성되고, dynamic scoping에서는 해당 subprogram과 모든 active한 함수의 variable로 구성됨.

1.6. Data Types

Data Type은 데이터에 대한 value와, 해당 value에 대한 operation(method)들의 정의임.

1.6.1. Primitive Data Types

Primitive Data Type은 다른 type을 활용해 정의되지 않는 기본 type임. primitive가 아닌 type은 primitive로 구성되거나, 그 subset임. PL은 여러 개의 primitive data type을 제공함.

primitive data type으로는 아래와 같은 것들이 있음. 일반적으로 integer, float, char, boolean으로 총 4가지의 primitive type이 제공됨.

1) Numeric
-> Integer : 정수. negative integer는 two's complement(one's complement를 구하고 1을 더함)를 활용해 저장함.
-> Floating Point : 실수.
-> Complex : 복소수.
-> Decimal : 십진수. BCD(Binary Coded Decimal)를 사용할 수 있음. 이는 십진수의 각 자리를 따로 이진수로 저장하는 방식임. 예를 들어, 12를 0001 0010와 같이 각 자리를 4비트를 활용해 이진수로 표현하는 것임. 이니수와 십진수 간의 변환이 없으므로 출력이 편리함.

2) Boolean
논리값. 양수이면 true, 양수가 아니면 false임.

3) Character
문자. 8비트를 사용해 영문자 등을 표현하는 ASCII를 주로 사용하고, 16비트를 사용해 전 세계 문자를 표현하는 Unicode도 사용함.

1.6.2. structured data types

1. Character String
Character String은 character의 sequence로 구성된 data type임. string에 대한 operation에는 assignment, catenation, substring reference, comparsion, pattern matching 등이 있음.

Substring Reference는 주어진 string에 대한 substring으로의 reference임. 즉, 일부분을 참조하는 것임. array에 대한 slice로 주로 사용됨.

*Pattern Matching은 string에서 특정 pattern과 일치하는 substring을 찾는 operation임. 여러 language들은 pattern matching을 위한 Regular Expression을 제공함. 이는 UNIX로부터 시작된 방식임. [A-Za-z]와 같이 작성하여 하나의 character에 대한 조건을 지정할 수도 있고, ?를 한 글자에 대한 wild card로, *로 임의의 길이 string에 대한 wild card로 사용할 수도 있음.*

PL에는 아래와 같이 string length에 대한 여러 option이 존재할 수 있음.

1) Static Length String : length가 static하게 결정되는 방식. 예를 들어, linked list 등으로 한 번의 allocation을 수행함.

2) Limited Dynamic Length String : 정의된 최대 length까지는 length가 dynamic하게 결정될 수 있는 방식. 예를 들어, pointer arrary으로 한 번의 allocation을 수행함.

3) Dynamic Length String : 임의의 length로 dynamic하게 결정될 수 있는 방식. 예를 들어, 공간이 부족할 때마다 allocation함.

2. Enumeration
Enumeration Type은 정의에서 가능한 모든 value(named constant)를 명시하는 type임. 이때의 named constant를 Enumeration Constant라고 함.

enumeration type은 readability와 reliability를 제공함.

예를 들어, C++에서는 아래와 같이 사용함.

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun}
```

3. Array
Array는 여러 유사한(homogeneous. type이 같음.) data element의 집합으로 구성된 data type으로, 각 element는 첫 번째 element에 대한 위치로 구분됨.

2차원 이상의 array는 그 형태에 따라 각 row의 길이가 모두 같은 Rectangular Array와, 각 row의 길이가 같지 않은 Jagged Array로 구분할 수 있음.

array는 FORTRAN에서 처음 등장했음.

4. Associative Array
Associative Array는 element별로 동일한 개수를 가지는 Key로 각 element를 구분하는 data type으로, element들은 정렬되어 있지 않음. 이때 element들의 자료형은 서로 다를(heterogeneous) 수 있음.

perl에서는 associative array를 hash function으로 구현하며, 이를 hash라고 부름. 아래와 같이 작성할 수 있음.

```
%salary = ("Gary" => 75000, "Perry" => 57000);
```

hash function은 어떤 입력에 대해 동일한 길이의 hash 값을 출력하는 function임. hash function을 이용한 hashing은 데이터를 저장하고 탐색하기 위한 방법으로, 기본적으로는 array로 구현됨. 동일한 hash 값을 가지는(collision) 요소들에 대해서는 linked list 등으로 구현함.

물론 element에 대한 search에 hasing이 아니라 linear search나 binary search(정렬 필요)를 사용할 수도 있는데, 데이터가 많이 바뀌는 경우 hasing을 사용함.

5. Record
Record는 name(field name)으로 구분되는 element(field)의 집합으로 구성된 data type임.

array에서 index로 각 element를 구분했지만, record에서는 name으로 각 element를 구분함.

record는 COBOL에서 처음 등장했음. 예를 들어, COBOL에서는 아래와 같이 record를 작성함. 이렇게 계층적인 구조를 가질 수 있음.

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST    PICTURE IS X(20).
        05  Middle   PICTURE IS X(10).
        05  LAST     PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.
```

계층적인 구조의 record에서 특정 field에 대한 reference 방법으로는, 해당 field까지의 모든 경로를 표기하는 Fully Qualified Reference와, 일부를 제외하고 표기하는 Elliptical Reference가 있음. 위의 예시에 대해서는 아래와 같이 작성될 수 있음.

```
*> Fully Qualified Reference
Employee_Record.Employee_Name.Middle

*> Elliptical Reference
First
FIRST OF EMPLOYEE-RECORD
FIRST OF EMPLOYEE-NAME
```

6. Tuple
Tuple은 record와 유사하지만 각 element가 name을 가지지 않는 data type임. element는 서로 다른 type을 가질(heterogeneous) 수 있고, index로 접근함.

예를 들어, phython에서는 아래와 같이 작성함.

```
myTuple = (3, 5, 2.4, 'apple')
```

7. List
List는 순서가 있는 데이터의 집합으로 구성된 data type임.

list는 최초의 funtional PL인 Lisp에서 처음 제시되었음.

8. Union
Union은 동일한 memory 공간에 대해 서로 다른 data type을 적용할 수 있도록 한 data type임. 이는 해당 type의 variable이 program 실행 도중 서로 다른 data type으로 value를 저장할 수 있도록 함.

예를 들어, c에서는 아래와 같이 작성함.

```
union flexType {
    int intEl;
    float floatEl;
};
union flexType el1;
float x;
...
el1.intEl = 27;
x = el1.floatEl;
```

일반적으로 자주 사용되지는 않지만 배타적으로 존재하는 field를 구성할 때 사용하기도 함.

1.6.3. Pointer and Reference Types

1. Pointer
Pointer는 memory address 또는 nil을 value로 가질 수 있는 data type임. nil(주로 0으로 구현됨.)은 유효한 address가 아니라, 해당 pointer가 특정 address를 referenece하고 있지 않음을 나타내는 value임.

pointer는 1. indirect addressing(call by reference)을 구현하거나, 2. heap으로부터 할당받은 dynamic storage를 관리하기 위해 사용됨. name이 존재하지 않아 variable을 Anonymous Variable이라고 하는데, heap-dynamic variable은 대체로 anonymous variable이므로 pointer를 활용해야 함.

2. Pointer의 문제점
pointer는 아래와 같은 문제점들을 가짐. garbage보다 dangling pointer가 더 큰 문제임.

1) Dangling Ponter/Reference : pointer가 deallocation된 heap address를 가지고 있는 경우. 이후 해당 address가 다른 variable에게 allocation된다면 문제가 더 커짐.

2) Lost Heap-dynamic Variable : allocation되었지만 address를 잃어버리는 등의 이유로 user가 사용하지 못하는 heap-dynamic variable. 이런 varibable를 garbage라고 부름. 해당 부분은 deallocation되지 않았으므로 memory를 낭비하게 됨.

garbage 문제에 대한 해결책은 아래와 같은 것들이 있음.

1) Eager Approach : reference counter를 사용하여, 해당 memory 영역을 reference하는 대상의 개수를 저장함. 이후 reference counter가 0이 되면 garbage로 인식되어 즉시 heap에 반환해 재활용함 (reclamation).

2) Lazy Approach : 사용 가능한 memory가 없는 상황이 되면, garbage로 인식되는 것들을 heap에 반환해 재활용함.

c/c++은 어셈블리어에서의 방식과 동일하게 pointer를 활용함. 즉, pointer에 대한 산술 연산이 가능하지만, pointer가 가진 문제점들이 발생할 수 있어 주의해야 함. java에서는 delete 기능을 제거하고 garbage collector를 사용하여 dangling pointer 문제를 해결함.

3. Reference Type
Reference Type은 다른 variable을 reference하는 데에 사용되는 data type임.

반면 다른 variable을 reference하는 것이 아니라, 자체적으로 value를 가지는 variable은 Scalar Variable이라고 하고, Value Type을 가진다고 함.

pointer가 address를 value로 가지는 data type이라면, reference type은 memory의 object 또는 value를 reference하는 data type임. pointer에 대한 산술 연산은 자연스럽지만, reference type에 대한 산술 연산은 의미가 없음.

참고로, indirect addressing은 c에서는 *로 구현되고, 어셈블리에서는 주로 아래와 같이 로 구현된다고 함.

```
jump 0x200      -> 0x200으로 점프.
jump @0x200     -> 0x200에 저장된 값으로 점프.
```

pointer는 goto와 그 위험성이 자주 비교됨. pointer와 goto 모두 프로그램의 어느 곳에도 갈 수 있고 그만큼 위험할 수 있음.

물론 goto는 wild goto와 structured goto로 나뉨. wild goto는 어셈블리의 jump와 유사하게 프로그램의 어디든 갈 수 있는 goto이고, structured goto는 break, continue 등과 같이 정해진 위치로만 갈 수 있는 goto임.

c에서는 아래와 같이 &와 *로 call by reference를 구현함. 이때 x는 a의 주소를 값으로 가지고, a는 자신의 값을 값으로 가지므로 x와 a는 alias가 아님. *x와 a가 alias임.

```
swap(&a, &b);

swap(*x, *y) {
 temp = *x;
 *x = *y;
 *y = temp;
}
```

1.6.4. Optional Type

Optional Type은 variable이 어떠한 value도 가지고 있지 않음을 나타내는 data type임.

variable이 value를 가지지 않는 상황은 0 등으로도 나타낼 수는 있지만, 이는 실제로 값이 0인 경우와 구분할 수 없음. 이에 따라 일부 PL들은 특정 값을 사용하는 optional type을 정의하여 사용함.

1.6.5. Type Checking

1. Type Checking
Type Checking은 operator에 대한 operand들이 서로 compatible(호환 가능)한 type인지를 검사하는 작업임. 이때 compatible하다는 것은 두 type이 operator에 대해 legal하거나, legal하도록 implicit하게 변환될 수 있는 것을 의미함. 이때 이런 자동 변환을 Coercion(형변환)이라고 함.

또한 runtime에 수행되는 type checking을 Dynamic Type Checking이라고 함.

2. Strong/Weak Typing
type mismatch에 대한 처리 방식에 따라 strong typing과 weak typing이 있음.

Strong Typing은 coercion 없이 type error가 항상 detection되는 것을 말함. strong typed PL에서는 compile time과 runtime 모두에서 각 operand에 대한 type checking이 가능해야 함.

Weak Typing은 ceorcion 등을 적용하여 type error를 덜 감지하는 것을 말함.

3. Type Equivalence
Type Equivalence는 두 type에 대해, 어떤 opperand의 type이 coercion 없이 서로 변환될 수 있는 것을 말함.

PL에서 variable에 대한 type equivalence에 대한 정의에는 아래와 같은 두 가지 approach가 존재함.

1) Name Type Equivalence : 두 variable이 동일한 declaration에서 선언되었거나, 동일한 type으로 선언된 경우 equivalent하다고 함.

2) Structure Type Equivalence : 두 variable의 type이 동일한 structure를 가지는 경우 equivalent하다고 함. name type euqivalence보다 더 유연한 기준임.

1.7. Data Abstraction

PL에서 제공하는 data에 대한 abstraction을 알아보자.

1.7.1. ADT

1. Abstraction

Abstraction(추상화)은 entity에 대해 중요한 attribute만을 포함하는 view 또는 representation임. 즉, abstraction은 중요하지 않은 부분은 제외하고 중요한 부분만을 고려하도록 해 programming process를 단순화함.

현재의 PL에서 제공하는 주요 abstration으로는 process abstraction과 data abstraction이 있음. Process Abstraction은 subprogram을 사용하는 것임. 즉, 구체적인 process를 몰라도 subprogram을 호출하는 것으로 작업을 수행할 수 있음. Data Abstraction은 ADT로 구현됨.

2. ADT

ADT(Abstraction Data Type)는 어떤 data type을 가지는 데이터와, 해당 데이터에 대한 subprogram의 묶음(enclosure)임. subprogram(operation)을 활용하여 해당 type에서 필요한 부분만을 사용하도록 하고, representation(구현. 실제 데이터 구성 등을 의미함.)은 노출되지 않도록 함. 이때 ADT의 instance를 Object라고 함.

함수가 어떤 명령어 집합을 가지도록 할 수 있지만, global variable이나 call by reference 등에 의해 다른 곳에서 declare된 variable을 건드리게 되는 functional side effect가 발생할 수 있음. 즉, 각 모듈이 독립적이지 않는 coupling의 문제가 발생할 가능성이 존재함. 이에 따라 ADT를 활용하여 encapsulation, information hiding을 구현함.

구체적으로는 아래의 조건을 만족시키는 것을 ADT라고 함.

1) 해당 type의 object가 가지는 representation은 그 object를 사용하는 program unit(client)에게 노출되지 않고, 제공된 operation(subprogram)으로만 접근이 가능함.

2) 해당 type에 대한 declaration과 interface(operation에 대한 protocol)가 하나의 syntax 단위 안에 포함되고, program unit은 해당 type에 대한 variable을 생성할 수 있음. 또한 이때 interface는 representation에 종속되지 않음.

ADT를 활용하면 해당 데이터에 제공된 operation으로만 접근이 가능하므로 reliability와 integrity를 확보할 수 있음.

interface 또는 specification, prototype은 서로 동일한 의미임.

예를 들어, stack에 대한 ADT는 아래와 같이 정의할 수 있음.

create(stack)	Creates and possibly initializes a stack object
destroy(stack)	Deallocates the storage for the stack
empty(stack)	A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise
push(stack, element)	Pushes the specified element on the specified stack
pop(stack)	Removes the top element from the specified stack
top(stack)	Returns a copy of the top element from the specified stack

c++에서는 private, public 등을 지정하고, constructor(초기화 시에 활용)와 destructor를 사용해 ADT를 구현할 수 있음. 또한 scope operator(::)를 사용해 특정 class를 활용할 수 있음.

java에서도 c++과 유사하게 ADT를 구현할 수 있지만, 모든 object가 heap에 생성되며 reference variable로 reference됨. 또한, 모든 method가 class내에 정의됨. 또한 사용하지 않는 object에 대해 garbage collector가 동작함.

1.8. OOP

1.8.1. OOP

1. OOP

OOP(Object-Oriented Programming)는 모든 것을 object로 보는 방법론임. OOP를 지원하는 PL은 3가지 핵심 특징으로 ADT, inheritance, dynamic binding을 포함함.

OOP에서 ADT는 Class, 그 instance는 Object라고 함. 또한 object에 대해 정의된 subprogram을 Method

라고 하고, method에 대한 호출을 Message라고 하며, class에 대한 method의 집합을 Message Protocol 또는 Message Interface라고 함. OOP에서의 작업은 interface를 통한 object 사이의 message 송수신으로 이해할 수 있음.

variable과 method는 그 소유자에 따라 instance와 class로 나눠짐. Instance Variable/Method는 각 object(instance)가 소유한 variable/method로, object를 생성해야 사용할 수 있음. 이때 variable은 각 object에 대해 생성되고, method는 모든 object가 하나의 코드 영역을 공유하여 사용함. Class Variable/Method는 class가 소유한 variable/method로, object를 생성하지 않아도 사용할 수 있음. 이때 각 obejct가 동일한 class variable에 접근하게 됨. java에서는 static을 붙여 class variable/method를 지정함.

ADT는 앞에서 다뤘으므로, 여기에서는 inheritance와 dynamic binding을 살펴보자.

2. Inheritance

Inheritance(상속)는 어떤 class가 기존 class의 데이터와 method를 상속받을 수 있도록 하고, 또한 상속받은 부분을 수정하거나 새로운 부분을 추가할 수 있도록 하는 기법임. 이는 효율적인 software reuse을 가능하게 해 productivity를 높임.

이때 어떤 class를 상속받은 class를 derived class, subclass, child class라고 하고, child class가 상속한 class를 base class, superclass, parent class라고 함.

child class가 parent class와 달라지는 경우는 아래와 같은 것들이 있음. child class는 기본적으로 parent class의 variable/method를 선택적으로 상속받을 수 없음.

1) 상속받은 부분에 새로운 variable/method 추가한 경우.
2) 상속받은 method를 override한 경우. 즉, method의 동작을 수정함.
3) parent class에서 variable/method를 private으로 지정하여 child class에서 visible하지 않은 경우.

inheritance는 parent class는 하나만 가질 수 있는 Single Inheritance와, 여러 parent class를 가질 수 있는 Multiple Inheritance로 나뉨. java에서는 single inheritance만을 지원함.

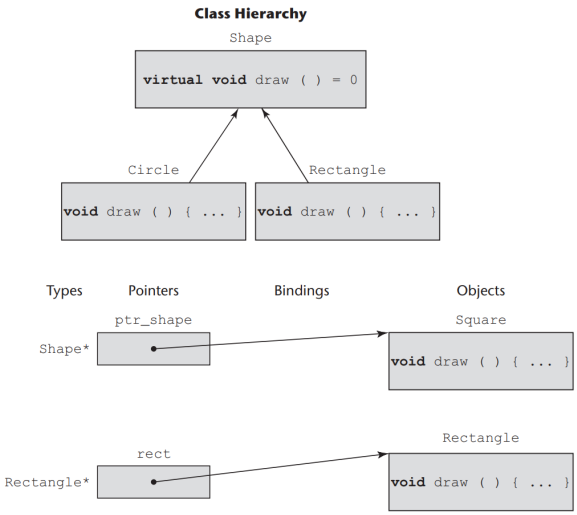
3. Dynamic Binding

OOP에서는 dynamic binding을 활용한 Polymorphism(다형성)을 지원함. 이는 Dynamic Dispatch라고도 부름. 즉, polymorphic reference가 가능하여 하나의 reference variable이 여러 class의 object를 reference할 수 있음.

또한 이에 따라 구현이 존재하지 않으며 그 자체로는 호출되거나 선언될 수 없는 Pure Virtual Function이 존재함. 각 virtual class를 상속받은 class는 pure virtual function을 override(구현)하여 활용함.

java에서는 overloading, overriding이 다형성을 구현함.

예를 들어, 아래와 같이 사용될 수 있음.



2. PL 2

2.1. Concurrency

우선 concurrency에 대해 알아보고, 각 PL에서 이를 어떻게 지원하는지를 알아보자.

2.1.1. Concurrency

1. Concurrency

Concurrency(동시성)는 시스템에서 여러 task나 process가 동시에 실행되는 것처럼 보이거나, 실제로 동시에 실행되는 것임.

concurrency 알고리즘은 scalable하고 portable해야 한 것이 이상적임. scalable하다는 것은 더 많은 processor(cpu, core)가 증가하면 실행 속도도 증가하는 것임. portable하다는 것은 하드웨어 또는 architecture 독립적으로 활용이 가능하다는 것임. 특히 하드웨어 수준에서의 발전에 따라 concurrency 알고리즘 설계 시에는 두 가지가 잘 고려되어야 함.

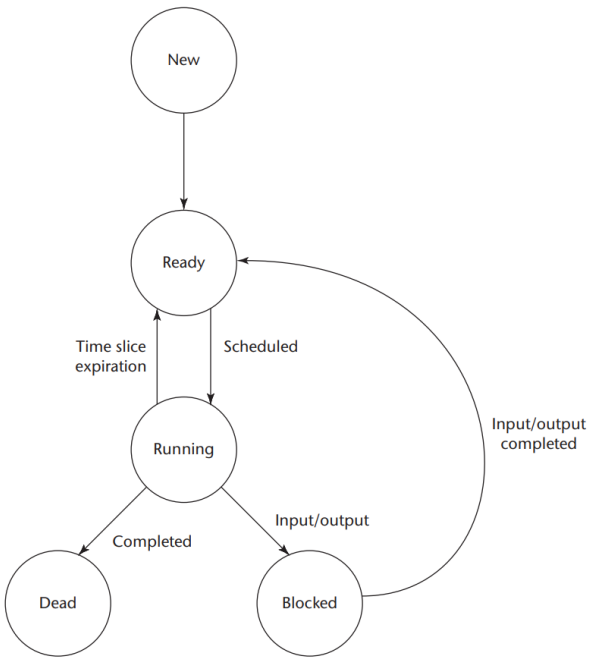
concurrency에 대해서는 아래와 같이 두 가지 형태의 처리를 고려할 수 있음.
1) parellel execution : 책상을 여러 명이서 나르는 것. 여러 대상이 각각 작업을 수행하는 경우.
2) pipeline : 각자 자리에 서서 책상을 넘겨 나르는 것. 한쪽에서 작업을 수행한 이후에 반대쪽에서 작업을 수행해야 하는 경우.

2. Physical vs. Logical Concurrency

concurrency는 physical concurrency와 logical concurrency로 나눌 수 있음. Physical Concurrency는 말 그대로 하드웨어에 의한 concurrency로, core가 여러개여서 실제로 각각 수행되는 것을 말함. Logical Concurrency는 하드웨어가 하나이거나 제한된 경우에도 적용 가능한 논리적인 수준의 concurrency로, 각 concurrency 단위들에 대한 timesharing으로 시간을 쪼개서 활용하는 것을 말함.

logical concurrency는 scheduler라고 하는 runtime system program에 의해 구현됨. scheduler는 각 time slice에 어떤 task가 실행될지를 결정하여 processor(cpu 또는 core)를 할당함.

task는 아래와 같은 state를 가질 수 있으며, 각 state에 대해서는 queue가 존재함. 아래의 그림을 Process(Task) State Transition Diagram이라고 함. 실행되던 task의 time slice가 끝나면(timeout) scheduler에 의해 ready 상태로 들어가 대기하게 되는데, 이렇게 줬던 자원을 회수하는 것을 Preemption 이라고 함. 이후 scheduler는 ready queue에서 다음으로 실행할 task를 꺼내 실행함.



2.1.2. Subprogram Level Concurrency

concurrency는 그 단위에 따라 subprogram, unit, statement 등 여러 측면에서 고려할 수 있는데, 여기에서는 여기에서는 subprogram(thread) 단위의 concurrency에서의 개념들을 살펴보자.

1. Task
Task는 동일한 프로그램 내의 다른 unit들과 concurrent하게 실행될 수 있는 프로그램 unit으로, subprogram과 유사함. task는 process라고도 부름. java 등의 언어에서는 threads라고 부르는 객체의 method가 task를 수행함.

task는 자신의 고유한 주소 공간에서 실행되는 Heavyweight Task와, 고유한 주소 공간을 가지지 않고 다른 unit과 동일한 주소에서 실행되는 Lightweight Task로 구분됨. Thread는 process 내의 존재하는 독립된 실행 단위로, lightweight task임.

어떤 task가 다른 task와 통신하거나 영향을 미치지 않으면 disjoint하다고 함.

2. Synchronization
Synchronization은 여러 task들에 대해 어떤 task가 먼저 실행될지에 대한 순서를 결정하는 방법임. 특히 이는 공유 자원이 존재하는 경우에 대한 것임. synchronization의 기본 원리는 공유 자원에 대한 lock-/unlock으로, 각 작업 앞뒤로 lock/unlock 처리를 함. synchronization은 이를 통해 mutual exclusion(상호배제)을 구현함.

데이터를 공유하는 경우, synchronization은 아래와 같이 2가지로 구분됨.

1) Cooperation Synchronization : 한 task가 수행되기 위해서는 다른 task가 완료되어야 하는 경우에 대한 synchronization.

cooperation synchronization은 Producer-Consumer Problem을 해결하는 것으로 생각할 수 있음. 이는 os의 개발과 관련해서 등장한 문제로, 한 program unit인 producer가 데이터를 생성해 buffer에 저장하고, 다른 program unit인 consumer가 buffer에서 데이터를 꺼내 활용하는 상황에 대한 것임. 이때 producer가 입력을 buffer에 너무 빨리 쓰면 overwrite될 수 있고, consumer가 buffer를 너무 빨리 읽으면 다시 읽게 될 수 있음.

2) Competition Synchronization : 동시에 사용될 수 없는 어떤 데이터에 대해 여러 task가 접근하려고 하는 경우에 대한 synchronization.

competition synchornization은 두 개 이상의 task가 동시에 공유 데이터에 접근하는 상황인 Race Condition을 방지하고, orderly execution을 수행하도록 함. race condition이 발생하는 경우 프로그램의 동작은 어떤 task가 먼저 해당 부분에 도착하는지에 따라 결정되어버리는 문제가 생기므로 이를 해결함.

3. Language Design for Concurrency
대부분의 PL에서는 concurrency를 지원함. 기본적으로 os(java에서는 JVM)가 thread를 지원하고, PL은 이를 활용함.

일부 언어에서는 concurrency를 지원하는 library가 존재함. c, c++, FORTRAN에서는 OpenMP, posix의 Pthread 등을 활용함. pthread는 posix이므로 UNIX 계열의 os에서 버전에 상관없이 해당 서비스를 활용할 수 있음.

synchronization 기법 중에 RW Lock이라는 것이 있음. RW는 read write은 data에 대해서는 reader와 writer가 존재할 수 있음. RW lock은 read는 임의로 수행할 수 있게 하고, write는 배타적으로 수행하도록 하는 기법임.

2.1.3. Semaphore

1. Semaphore
Semaphore는 하나의 정수와, task descriptor를 저장하는 queue를 포함하는 자료구조로, counting 하는 것으로 synchronization 처리를 함. Task Descriptor는 task에 대한 정보를 저장하는 자료구조임. 이는 dijkstra가 제시한 방법으로, 가장 널리 사용되는 synthronization 기법 중 하나임.

semaphore에는 wait/lock과 signal/release/unlock이 존재함. wait과 signal의 인자로는 semaphore 정수 s를 입력하는데, wait(s)는 s가 0보다 크면 s를 1 줄이고, 작으면 block함. signal(s)는 s를 1 늘림. wait과

signal을 race condition이 발생할 수 있는 지점의 앞뒤로 작성하여 synchronization 처리를 함. 아래는 wait과 signal의 스도코드임.

```
wait(aSemaphore)
if aSemaphore's counter > 0 then
     decrement aSemaphore's counter
else
    put the caller in aSemaphore's queue
    attempt to transfer control to some ready task
    (if the task-ready queue is empty, deadlock occurs)
end if

release(aSemaphore)
if aSemaphore's queue is empty (no task is waiting) then
     increment aSemaphore's counter
else
    put the calling task in the task-ready queue
    transfer control to a task from aSemaphore's queue
end
```

2. Synchronization 유형별 Semaphore

synchronization 유형별로 아래와 같이 semaphore를 적용할 수 있음.

1) cooperative synchronization

아래와 같이 2개의 semaphore를 사용하여 producer와 consumer가 사용하는 shared buffer에 대한 synchronization을 적용함.

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);     { wait for a space }
  DEPOSIT(VALUE);
  release(fullspots);   { increase filled spaces }
  end loop;
end producer;

task consumer;
   loop
   wait(fullspots);     { make sure it is not empty }
   FETCH(VALUE);
   release(emptyspots); { increase empty spaces }
   -- consume VALUE --
   end loop
end consumer;
```

2) competition synchronization

아래와 같이 1개의 semaphore를 사용하여 접근하려는 공유 자원 앞뒤로 lock과 unlock을 걸어 synchronization을 적용함.

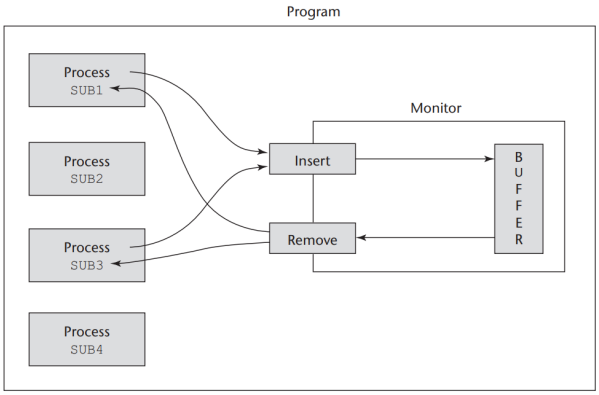
```
task producer;
  loop
  -- produce VALUE --
  wait(emptyspots);       { wait for a space }
  wait(access);           { wait for access }
  DEPOSIT(VALUE);
  release(access);        { relinquish access }
  release(fullspots);     { increase filled spaces }
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);       { make sure it is not empty }
  wait(access);          { wait for access }
  FETCH(VALUE);
  release(access);       { relinquish access }
  release(emptyspots);   { increase empty spaces }
  -- consume VALUE --
  end loop
end consumer;
```

semaphore에서는 wait과 signal을 프로그래머가 직접 지정해야 하는데, 이에 따라 당연하게도 코드를 빼먹거나 잘못 작성하는 경우에 의해 reliability가 낮다는 문제점이 존재함.

2.1.4. Monitor

Monitor는 공유 자원에 대한 class를 만들고, 특정 operation을 활용해 접근하도록 해서 synchronization을 구현하는 기법임. 즉, 아래와 같은 구조를 가짐.



monitor는 encapsulation을 통해 semaphore의 낮은 reliability를 해결함. 하지만 monitor와 semaphore는 동등한 concurrency power를 가지며, 서로가 서로의 구현에 활용될 수 있음.

2.1.5. Java에서의 Synchronization

1. Java Semaphore
java에서는 java.util.concurrent.Semaphore로 semaphore를 활용할 수 있음. 이때 Semaphore 객체는 counter가 존재하지만 queue는 존재하지 않음. 또한 wait과 release에 대응되는 메소드로 acquire()와 release()를 제공함.

2. Monitor
java에서는 synchronized 키워드를 메소드에 붙여 메소드 수준에서의 synchronization을 적용하여 monitor를 구현할 수 있음. synchronized 키워드를 붙이면 해당 메소드는 한 번에 하나만 호출될 수 있음.

3. Java Thread
java에서의 concurrent unit은 thread 관련 class의 메소드 run()임. 이는 다른 run() 메소드와, main() 메소드들과 concurrent하게 실행될 수 있음. 이때 run() 메소드가 실행하는 작업을 thread라고 함. java에서 thread는 lightweight task임.

java에서 thread 관련 class를 생성하는 방법은 아래와 같이 2가지가 존재함. 이후 해당 class의 객체를 생성하고 start() 메소드를 호출하면 run()에 해당하는 부분이 실행됨.

1) Thread class를 상속받고 run() 메소드를 오버라이드함.
2) Runnable 인터페이스를 implement하고 run()을 구현함. java에서는 하나의 class만 상속받을 수 있으므로, 다른 class를 상속받는 경우 인터페이스를 구현하도록 해야 함.

thread 관련 class의 메소드로는 sleep()과 join()도 있음. sleep()은 정수를 입력으로 받아, 해당 정수만큼의 ms 동안 thread가 block되도록 함. thread는 block이 끝나면 ready 상태가 됨. join()은 다른 thread의 run()이 완료될 때까지 실행을 연기하는 메소드임. 예를 들어, t1.join()을 호출하면 t1이 끝날 때까지 기다림. 또한 t1.join(2000)과 같이 호출하면 timeout을 걸어 해당 시간(ms)까지만 기다림.

thread의 lifetime은 start()부터 run()의 마지막 부분까지임.

프로그램이 여러 개의 thread를 가지는 경우, scheduler는 어떤 thread가 먼저 수행될 것인지를 결정함. 이때 환경에 따른 각 구현에서 scheduler의 동작이 동일하게 구현되어 있지는 않지만, 대체로 round-robin 방식으로 동일한 길이의 time slice를 각 ready thread에 분배하는 식으로 동작함. 이때 각 thread는 동일한 priority를 가지는 것으로 처리함.

4. Synchronization 유형별 처리 방법
synchronization 유형별 처리 방법은 아래와 같음.

1) cooperative synchronization
모든 class의 상위 class인 Object가 포함하는 wait(), notify(), notifyAll() 메소드를 사용해 synchronization을 적용할 수 있음. 이 세 메소드는 synchronized 메소드에서만 호출할 수 있음.

wait() 메소드는 현재 실행 중인 thread를 일시 중단하고(block 상태로 전환) 해당 객체의 wait list에 추가함(semaphore에서의 그것과는 다름.). notify()는 특정 event를 기다리는 waiting thread에게 event의 발생을 알려 깨움. 이때 어떤 event가 깨워지는지는 JVM에 의해 결정되고, 직접 지정할 수 없음. 이에 따라 notifyAll()을 사용할 수 있는데, notifyAll()은 해당 객체가 가진 wait list의 모든 thread를 깨움.

아래와 같이 queue를 처리할 때 wait()과 notifyAll()을 활용할 수 있음. deposit()과 fetch()를 각각 사용하는 producer와 consumer는 synchronization 처리됨.

```
  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
  }  //** end of Queue constructor

  public synchronized void deposit (int item)
         throws InterruptedException {
    try {
      while (filled == queSize)
        wait();
      que [nextIn] = item;
      nextIn = (nextIn % queSize) + 1;
      filled++;
      notifyAll();
    }  //** end of try clause
    catch(InterruptedException e) {}
  }  //** end of deposit method

  public synchronized int fetch()
      throws InterruptedException {
    int item = 0;
    try {
      while (filled == 0)
        wait();
      item = que [nextOut];
      nextOut = (nextOut % queSize) + 1;
      filled--;
      notifyAll();
    }  //** end of try clause
    catch(InterruptedException e) {}
    return item;
  }  //** end of fetch method
}  //** end of Queue class
```

2) competition synchronization
특정 공유 데이터에 접근하는 method에 synchronized 키워드를 붙여 synchronization을 적용할 수 있음.

2.2. Exception/Event Handling

exception handling과 event handling에 대해 알아보자. 이 둘은 서로 다른 개념이지만 둘 다 비결정적으로 발생하는 상황을 다루고, 또한 처리 과정이 유사함.

여기에서의 event handling은 java에서 제공하는 GUI event handling을 살펴봄.

2.2.1. Exception Handling

1. Exception
Exception은 하드웨어 또는 소프트웨어가 감지하여 처리할 수 있는 unusual event임. exception이 감지되었을 때 수행되는 특별한 처리를 Exception Handling이라고 하고, 이는 Exception Handler라고 하는 program unit 또는 segment임. 관련 event가 발생하여 exception이 등장하는 것을 Raised 또는 Thrown(c 계열 언어들)이라고 함.

대부분의 하드웨어 시스템은 runtime error(ex. segment fault)를 감지함. 초창기의 PL들은 이런 error를 감지하거나 처리하도록 설계 및 구현되지 않았고, 이에 따라 단순히 error가 발생할 때마다 프로그램이 종료되고 제어가 os에게 넘어가도록 되어 있었음. 최근의 PL에서는 이런 error를 program-detected event인 exception으로 처리할 수 있도록 함.

exception은 하드웨어 또는 os에 의해 자동으로 감지되어 발생하는 Implicit Exception과, 사용자 코드에

의해 직접적으로 발생하는 Explicit Exception으로 나눌 수 있음.

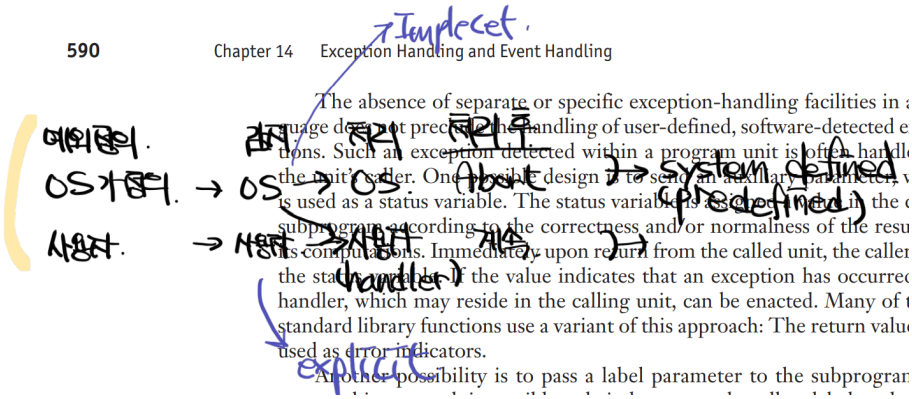
exception handling은 Ada, c++, java 등에 의해 구현되었음.

2. Exception Handling 과정

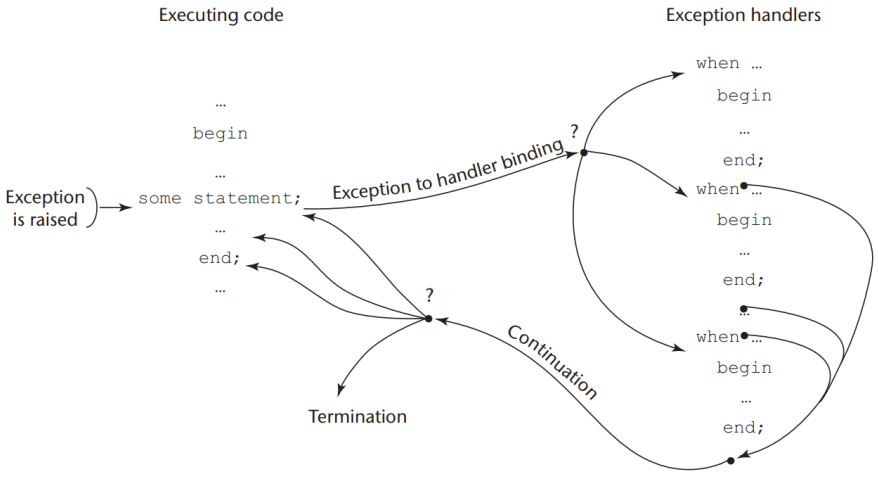
exception handling 과정은 아래와 같이 4가지 단계로 구성됨.

1) exception 정의
2) exception 감지
3) exception 처리
4) exception 처리 이후의 동작

이는 아래의 그림과 같이 os에 의해 정의 및 감지되는 implicit exception과, 사용자의 의해 정의 및 감지되는 explicit exception으로 구분해 고려할 수 있음. exception이 os에 의해 처리되면 프로그램은 abort되지만, 사용자가 정의한 handler에 의해 처리되면 종료되지 않도록 할 수 있음.



사용자에 의한 exception handling의 control flow는 아래와 같음.



terminate는 계획된 방식을 통한 프로그램의 종료를 의미하고, abort는 os에 의한 비정상적인 강제 종료를 의미함.

2.2.2. Exception Handling in C++

c++에서의 exception handling은 Ada, ML 등으로부터 영향을 받았으며 1990년에 표준에 포함되었음.

아래와 같이 try-catch와 throw를 활용해 exception handling을 할 수 있음.

```
try{
    ...
    throw NegativeInputException();
    ...
}
catch{
    ...
}
```

2.2.3. Exception Handling in Java

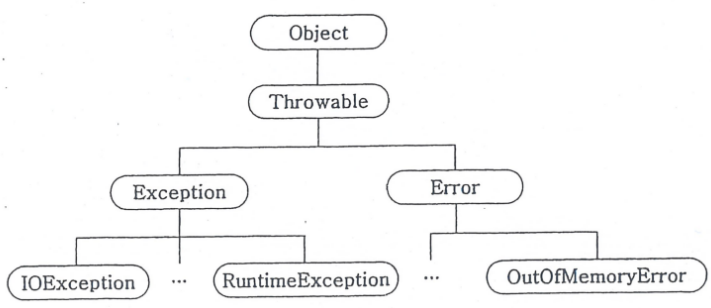
java에서의 exception handling은 c++에서의 그것과 유사하지만, 객체지향적으로(객체를 throw하도록.) 구현되었으며 JVM에서 정의하는 predefined exception을 활용함. 또한 finally, throws를 활용함.

1. Exception 관련 Class

모든 java exception은 Throwable class의 하위 class임. Throwable class는 Error와 Exception을 class로 가짐.

Error class는 JVM이 throw하는 error에 대한 class임. 이는 사용자에 의해서 throw될 수 없고, 사용자에 의해 처리되어서는 안 됨.

Exception class의 하위 class들은 RuntimeException과 그 하위 class, 그리고 나머지 class들로 구분할 수 있음.



Error와 그 하위 class들, RuntimeException과 그 하위 class들은 unchecked exception이고, 나머지 Throwable class들은 checked exception임. Checked Exception은 compiler가 그 처리를 확인하는 exception으로, 메소드에서 해당 exception을 try-catch로 handling하거나 throws로 호출자에게 넘기는 처리를 해야 함. Unchecked Exception은 compiler가 고려하지 않는 exception임.

checked exception은 try-catch로 처리하지 않았다면, 아래와 같이 메소드에 throws로 명시해야 함. 이에 따라 해당 메소드를 호출하는 쪽에서 이를 고려하여 처리하도록 함.

```
void buildDist() throws IOException {
    ...
}
```

RuntimeException와 그 하위 class도 throws에 작성할 수 있지만, 주로 작성하지 않는다고 함.

2. Exception Handling

java에서는 exception handling을 위해 try-catch문을 활용함. 당연하게도 catch문 parameter의 class는 Throwable의 하위 타입이어야 함. 이는 c++에서의 문법과 동일함.

throw된 exception 객체가 해당 메소드에서 try-catch로 처리되지 않았다면 그 메소드를 호출한 메소드로 객체가 전달됨. 이 과정이 반복되다가 결국 main에 도달하고, 여기에서도 처리되지 않았다면 JVM에게 전달되어 프로그램이 종료됨. 이를 exception의 Propagation이라고 함.

이때 catch문은 위에서부터 하나씩 검사하면 호환되는 class가 존재하면 해당 catch를 활용함.

```
try {
    // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (Exception1 e1) {
    // Exception1이 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
} catch (Exception2 e2) {
    // Exception2가 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
...
} catch (ExceptionN eN) {
    // ExceptionN이 발생했을 경우, 이를 처리하기 위한 문장을 적는다.
}
```

아래와 같이 exception 객체를 생성하고 throw 키워드를 사용해 exception을 발생시킬 수 있음. 이때 Exception class 또는 적절한 하위 class를 상속받아 사용자 정의 exception class를 생성해 활용할 수 있음. throw로 exception을 명시하는 것은 actual parameter 역할을 수행하는 것이고, catch문의 parameter는 formal parameter 역할을 수행하는 것임.

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}
...
throw new MyException("MyException occured!");
```

Exception 객체를 생성할 때 인자로 문자열을 넣어주면 해당 문자열이 exception 메시지로 활용됨. Exception 객체에 대해서는 getMessage(), printStackTrace()로 그 정보를 확인할 수 있고, 이는 특히 catch문에서 활용할 수 있음.

try-catch로 exception을 처리한 뒤 exception을 다시 발생시켜 해당 메소드와 그 메소드를 호출한 메소드 모두에서 exception을 처리하도록 할 수 있는데, 이를 Exception Rethrowing이라고 함.

```
try {
    ...
}
catch(Exception e) {
    ...
    throw e;
}
```

3. Finally문

Finally문은 try-catch에 의한 처리에도 불구하고 항상 수행해야 하는 작업을 위해 사용함. 예를 들어, 파일 닫기, 자원 반납 등을 수행함. 이는 아래와 같이 작성함. 이 경우 제어가 넘어가거나 프로그램이 종료되는 경우에도 finally는 항상 수행됨.

```
try {
    ...
}
catch(...) {
    ...
}
finally {
    ...
}
```

catch문 없이 try와 finally만 작성하기도 함. 이 경우 try문 안에서 break, continue, return에 의해 제어가 넘어가도 finally는 항상 실행됨.

```
try {
    ...
}
finally {
    ...
}
```

4. Assertion
Assertion은 지정한 조건이 참인지를 검사하는 문법으로, defensive programming을 위해 사용됨. 이는 아래와 같이 작성하고, 제어가 도달했을 때 condition이 참이면 그냥 넘어가고 거짓이면 AssertionError exception을 발생시킴. 이때 뒤에 expression을 작성하면 해당 값을 AssertionError 생성자의 인자로 넣어 message로 함.

```
assert condition;
asser condition : expression;
```

이는 주로 디버깅을 위해 사용되고, 프로그램에 대한 검증이 완료되면 제외함. 이때 코드를 전부 수정하는 대신 단순히 JVM에 옵션을 주는 것만으로 비활성화가 가능함.

2.2.4. Event Handling

Event는 어떤 사건이 발생했음을 알리는 알림으로, user action에 따라 runtime system이 생성한 객체로도 볼 수 있음. 이런 event를 handling하는 코드 부분을 Event Handler라고 함. event에 대해 적절한 작업이 수행되도록 하는 것을 Event Handling이라고 함.

event handling은 exception handling과 유사하게, event의 발생에 따라 Event Handler가 호출되는 식으로 동작함. 하지만 exception이 implicit/explict하게 발생할 수 있는 것과는 달리, event는 GUI에서의 사용자 상호작용 등 external action에 의해 발생함. 여기에서는 이런 GUI에서의 상호작용에 따른 event를 살펴봄.

또한 event는 exception과 달리, 처리하지 않으면 아무 일도 일어나지 않음.

상호작용이 발생하는 GUI에서의 graphical object/component를 Widget이라고 함(ex. button). widget에 대한 반응을 구현하는 것이 event handling의 가장 흔한 형태임.

2.2.5. Event Handling in Java

java에서의 GUI 처리에는 AWT와 Swing 등을 활용할 수 있음.

1. AWT
AWT는 java 최초의 GUI 툴킷임.

AWT에서 Event Listener는 event를 감지하고 처리하는 interface이고, 이를 구현한 것이 Event Handler임. 사용자는 widget(Event Source, Event Generator)을 생성하고, event 객체를 전달받는 event handler를 event listener로 등록(registration)함. 사용자가 widget과 상호작용하면 event 객체가 생성되고, event listener는 event를 처리함.

```
import java.awt.*;
import java.awt.event.*;

class FrameTest3 {
    public static void main(String args[]) {
        Frame f = new Frame("Login");     // Frame객체를 생성한다.
        f.setSize(300, 200);              // Frame의 크기를 설정한다.

        // EventHandler클래스의 객체를 생성해서 Frame의 WindowListener로 등록한다.
        f.addWindowListener(new EventHandler());
        f.setVisible(true);               // 생성한 Frame을 화면에 보이도록 한다.
    }
}

class EventHandler implements WindowListener
{
    public void windowOpened(WindowEvent e) {}
    public void windowClosing(WindowEvent e) { // Frame의 닫기 버튼을 눌렀을 때 호출된다.
        e.getWindow().setVisible(false);      // Frame을 화면에서 보이지 않도록 하고
        e.getWindow().dispose();              // 메모리에서 제거한다.
        System.exit(0);                       // 프로그램을 종료한다.
    }
    public void windowClosed(WindowEvent e) {} // 아무내용도 없는 메서드 구현
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
```

event와, 해당 event를 처리하는 event listener interface는 아래와 같음.

이벤트	인터페이스	메서드
ActionEvent	ActionListener	actionPeformed(ActonEvent ae)
ComponentEvent	ComponentListener	componentMoved(ActonEvent ae)
		componentHidden(ActonEvent ae)
		componentResized(ActonEvent ae)
		componentShown(ActonEvent ae)
MouseEvent	MouseMotionListener	mouseDragged(MouseEvent me)
		mouseMoved(MouseEvent me)
	MouseListener	mousePressed(MouseEvent me)
		mouseReleased(MouseEvent me)
		mouseEntered(MouseEvent me)
		mouseExited(MouseEvent me)
		mouseClicked(MouseEvent me)
MouseWheelEvent	MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)
KeyEvent	KeyListener	keyPressed(KeyEvent ke)
		keyReleased(KeyEvent ke)
		keyTyped(KeyEvent ke)
TextEvent	TextListener	textValueChanged(TextEvent te)
FocusEvent	FocusListener	focusGained(FocusEvent fe)
		focusLost(FocusEvent fe)
ItemEvent	ItemListener	itemStateChanged(ItemEvent ie)
AdjustmentEvent	AdjustmentListener	adjustmentValueChanged(AdjustmentEvent ae)
WindowEvent	WindowListener	windowClosing(WindowEvent we)
		windowOpened(WindowEvent we)
		windowIconified(WindowEvent we)
		windowDeiconified(WindowEvent we)
		windowClosed(WindowEvent we)
		windowActivated(WindowEvent we)
		windowDeactivated(WindowEvent we)
	WindowFocusListener	windowGainedFocus(WindowEvent e)
		windowLostFocus(WindowEvent e)
	WindowStateListener	windowStateChanged(WindowEvent e)
ContainerEvent	ContainerListener	componentAdded(ContainerEvent ce)
		componentRemoved(ContainerEvent ce)

[표13-24] Event의 종류와 관련 인터페이스

event 발생 시에, event 객체에 해당되는 listener에서 해당 동작에 대한 메소드가 호출됨. 메소드 목록은 아래와 같음.

메서드	호출시기
actionPeformed(ActonEvent ae)	Button을 클릭했을 때, Menu를 클릭했을 때, TextField에서 Enter키를 눌렀을 때, List의 Item하나를 선택하여 더블클릭했을 때
componentMoved(ActonEvent ae)	컴포넌트가 이동되었을 때
componentHidden(ActonEvent ae)	컴포넌트가 화면에 보이지 않게 되었을 때
componentResized(ActonEvent ae)	컴포넌트의 크기가 변경되었을 때
componentShown(ActonEvent ae)	컴포넌트가 화면에 보여 질 때
mouseDragged(MouseEvent me)	마우스 버튼을 누른 채로 마우스를 움직였을 때
mouseMoved(MouseEvent me)	마우스 포인터를 이동시킬 때
mousePressed(MouseEvent me)	마우스 버튼을 눌렀을 때
mouseReleased(MouseEvent me)	마우스 버튼을 떼었을 때
mouseEntered(MouseEvent me)	마우스 포인터가 이벤트 소스의 영역 안으로 들어왔을 때
mouseExited(MouseEvent me)	마우스 포인터가 이벤트 소스의 영역 안에서 밖으로 이동할 때
mouseClicked(MouseEvent me)	마우스 버튼을 눌렀다가 떼었을 때
mouseWheelMoved(MouseWheelEvent e)	마우스의 휠을 움직였을 때
keyPressed(KeyEvent ke)	키보드의 키를 눌렀을 때
keyReleased(KeyEvent ke)	키보드의 키를 떼었을 때
keyTyped(KeyEvent ke)	키보드의 키를 눌렀다 떼었을 때
textValueChanged(TextEvent te)	TextField 또는 TestArea의 내용이 변경되었을 때
focusGained(FocusEvent fe)	이벤트 소스로 focus가 이동했을 때
focusLost(FocusEvent fe)	이벤트 소스가 갖고 있던 focus가 다른 컴포넌트로 이동했을 때
itemStateChanged(ItemEvent ie)	Checkbox, CheckboxItem, List, Choice의 status가 변경되었을 때 (selected ↔ unselected)
adjustmentValueChanged(Adjustment Event ae)	Scrollbar의 값이 변경되었을 때
windowOpened(WindowEvent we)	윈도우가 열렸을 때
windowClosing(WindowEvent we)	윈도우가 닫힐 때(닫기 버튼을 눌렀을 때)
windowClosed(WindowEvent we)	윈도우가 닫혔을 때(dispose()가 호출 되었을 때)
windowIconified(WindowEvent we)	윈도우가 최소화(아이콘화) 되었을 때
windowDeiconified(WindowEvent we)	윈도우가 최소화 상태에서 다시 원래 크기로 되었을 때
windowActivated(WindowEvent we)	윈도우가 활성화 되었을 때
windowDeactivated(WindowEvent we)	윈도우가 비활성화 되었을 때
windowGainedFocus(WindowEvent e)	윈도우가 포커스를 얻을 때
windowLostFocus(WindowEvent e)	윈도우가 포커스를 잃었을 때
windowStateChanged(WindowEvent e)	윈도우의 상태가 변했을 때
componentAdded(ContainerEvent ce)	컨테이너에 컴포넌트가 추가되었을 때
componentRemoved(ContainerEvent ce)	컨테이너에 컴포넌트가 제거되었을 때

[표13-25] 이벤트 리스너를 Component에 추가/제거하는 메서드

또한 각 event source마다 등록할 수 있는 listner와, 해당 listener를 등록 및 제거할 때 사용하는 메소드가 정해져 있음.

Listener	Listener를 추가 / 제거 때 사용하는 메서드	이벤트 소스
ActionListener	void addActionListener(ActionListener l) void removeActionListener(ActionListener l)	Button,List, MenuItem, TextField
AdjustmentListener	void addAdjustmentListener(AdjustmentListener l) void removeAdjustmentListener(AdjustmentListener l)	Scrollbar
ComponentListener	void addComponentListener(ComponentListener) void removeComponentListener(ComponentListener l)	Component
ContainerListener	void addContainerListener(ContainerListener l) void removeContainerListener(ContainerListener l)	Container
FocusListener	void addFocusListener(FocusListener l) void removeFocusListener(FocusListener l)	Component
ItemListener	void addItemListener(ItemListener l) void removeItemListener(ItemListener l)	Checkbox, CheckboxMe uItem, List, Choice
KeyListener	void addKeyListener(KeyListener l) void removeKeyListener(KeyListener l)	Component
MouseListener	void addMouseListener(MouseListener) void removeMouseListener(MouseListener l)	Component
MouseMotionListener	void addMouseMotionListener(MouseMotionListener) void removeMouseMotionListener(MouseMotionListener l)	Component
TextListener	void addTextListener(TextListener l) void removeTextListener(TextListener l)	TextField, TextArea
WindowListener	void addWindowListener(WindowListener l) void removeWindowListener(WindowListener l)	Window
WindowFocusListener	void addWindowFocusListener(WindowFocusListener l) void removeWindowFocusListener(WindowFocusListener l)	Window
WindowStateListener	void addWindowStateListener(WindowStateListener l) void removeWindowStateListener(WindowStateListener l)	Window

[표13-27] 이벤트 리스너를 컴포넌트에 추가 또는 제거할 때 사용하는 메서드

또한 이때 event listener를 매번 직접 구현하는 경우 사용하지 않는 메소드에 대한 코드도 작성해야 한다는 번거로움이 존재하는데, 이에 따라 Adapter class를 상속받은 위 해당 메소드를 오버라이드하는 것으로 구현할 수 있음. adapter class는 lister interface를 단순히 빈 메소드로 구현해 놓은 것임. adapter class는 아래와 같은 것들이 존재함.

Adapter클래스	이벤트 리스너(Interface)
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

또한 당연하게도 아래와 같이 코드를 작성할 수 있음.

```
class KeyHandler implements KeyListener
{
  public void keyPressed(KeyEvent e) {
    System.out.println(e.getKeyChar());
  }
  public void keyReleased(KeyEvent e) {}
  public void keyTyped(KeyEvent e) {}
}
```

```
class KeyHandler extends KeyAdapter {
  public void keyPressed(KeyEvent e) {
    System.out.println(e.getKeyChar());
  }
}
```

2. Swing

Swing은 widget에 대한 class와 interface 등을 포함하는 툴킷으로, AWT를 확장하여 만든 것임. 객체를 생성하여 해당하는 widget을 만들 수 있음. AWT가 가진 component 이름에 j를 붙인 것으로 식별자가 구현되었다고 함.

2.3. Subprogram

2.3.1. Subprogram

1. Subprogram

Subprogram은 process abstraction의 한 종류임. 즉, subprogram을 활용하면 사용자가 구체적인 세부사항까지 지정하지 않아도 process를 실행할 수 있음.

일반적인 subprogram은 아래와 같은 특성을 가짐.

1) 각 subprogram은 하나의 entry point를 가짐.
2) 한 번에 하나의 subprogram만이 실행됨. 즉, subprogram이 호출되면 caller는 실행이 연기됨.
3) subprogram의 실행이 종료되면 제어가 caller에게 돌아옴.

Subprogram Definition은 해당 subprogram에 대한 interface와, 그 action을 나타내고, Subprogram Call은 특정 subprogram에 대한 실행 요청임. call된 이후로 종료되지 않은 subprogram은 Active되어 있다고 함. 예를 들어, main이 A를, A가 B를 call했고, 현재 제어가 B에 있다면 main, A, B 모두 active임.

subprogram은 abstraction과 재사용, information hiding 등에서 의의가 있음. 하지만 coupling이라는 side effect가 존재하는데, 이에 따라 data abstraction으로 객체지향이 등장했음.

2. Function vs. Procedure
subprogram은 수학에서의 함수를 그대로 구현하여 반환값이 존재하는 Function과, 반환값이 존재하지 않는 Procedure로 구분됨. c 등 많은 PL에서는 이를 키워드 등으로 구분하는 대신, 반환값의 유무에 따라 function과 procedure를 구현함.

procedure는 반환값이 존재하지 않으므로 1) formal parameter가 아니면서 해당 procedure와 caller 모두에서 visible한 variable이 존재하거나, 2) caller에게 데이터를 전송할 수 있는 formal parameter를 가진 경우 caller에 결과를 전달할 수 있음.

앞에서 다룬 것처럼, subprogram의 사용에 따라 functional side effect(coupling)가 존재함. 이를 방지하려면 parameter가 항상 in mode여야 하는데, Ada 등 일부 PL에서는 이를 강제함.

3. Parameter
non-method subprogram이 처리할 데이터에 접근하는 방식에는 1) visible한 nonlocal variable에 직접 접근하는 것과, 2) parameter passing을 활용하는 것이 있음.

subprogram의 Parameter Profile은 해당 subprogram이 가진 formal parameter의 number, order, type을 포함하는 개념임. 또한 subprogram의 Protocol은 parameter profile과 return type을 포함하는 개념임.

앞에서 다룬 것처럼, parameter에는 formal parameter와 actual parameter가 존재함.

대부분의 PL에서는 actual parameter와 formal parameter의 binding이 parameter의 순서에 의해 정해지는데, 이를 Positional Parameter라고 함. 반면 순서가 아니라 formal parameter의 이름을 명시해 binding하는 방식은 Keyword Parameter라고 함. 예를 들어, c 등은 positional parameter만을 지원하지만, python은 keyword parameter도 지원함. positional parameter는 parameter의 개수가 적을 때는 안전하고 효율적이지만, 개수가 많아지면 프로그래머가 헷갈릴 수 있으므로 keyword parameter가 더 유용할 수 있음.

python, c++ 등에서는 formal parameter에 default 값을 지정하여 acutal parameter가 지정되지 않는 경우에 해당 값을 활용하도록 지원함.

actual parameter로 작성한 variable에 대해 formal parameter와의 type checking을 수행할 것인지도 design issue 중 하나임. 물론 작은 typographical error에 의해 문제가 발생하기 쉬우므로, consistency 및 reliability 측면에서는 이를 적용하는 것이 좋음.

parameter로 subprogram을 전달할 수 있도록 지원하는 PL도 존재함.

PL에서의 가장 주요한 abstraction들로는 process abstraction과 data abstraction이 존재함. 초기 PL에서는 process abstraction만을 지원했지만, 이후 data abstraction의 중요성이 강조되면서 객체지향이 등장했음.

대부분의 imperative PL에서는 반환값으로 사용할 수 있는 자료형에 제한을 둠. 예를 들어, c에서는 arrary와 function을 반환할 수 없음. 물론 이는 반환값을 pointer로 해서 반환이 가능함.

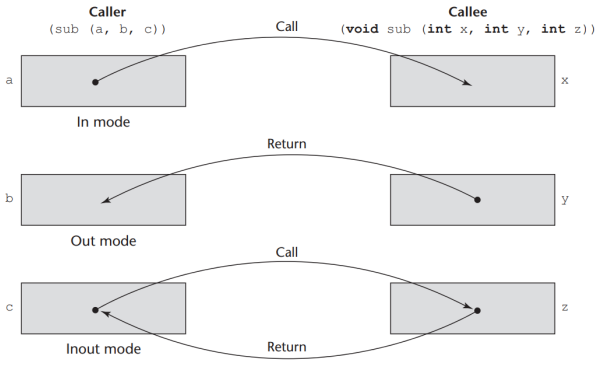
jump는 jp(jump)이고, call은 jr(jump and return)임. 즉, call은 이후 다시 돌아오는 연산이 수행됨.

2.3.2. Parameter Passing Method

1. Parameter Passing Method
Parameter Passing Method는 subprogram으로 parameter를 어떻게 전달할지, 그리고 parameter를 통해 subprogram으로부터 어떻게 값을 넘겨받을지에 대한 방법론임. 이는 subprogram design에서 가장 주요한 issue 중 하나임.

parameter passing에 대한 Semantic Model은 특정 formal parameter에서의 데이터 전달 방식임. semantic model은 아래와 같이 3가지가 존재하고, 이를 활용해 formal parameter를 구분할 수 있음.

1) In mode : 대응되는 actual parameter로부터 데이터를 받을 수 있는 경우.
2) Out mode : 대응되는 actual parameter로 데이터를 전달할 수 있는 경우.
3) Inout mode : 대응되는 actual parameter로부터 데이터를 받고, 전달할 수 있는 경우.



parameter passing method를 결정하는 주요 고려 사항으로는, 1) 효율성과 2) 필요한 데이터 전달이 one-way인지 two-way인지가 있음. 또한 subprogram 외부의 데이터를 조작하는 것은 최소화해야 함.

2. Implementation of Semantic Models
semantic model은 PL에서 실제로 어떤 형태로 구현되었는지 살펴보자. 대부분의 PL에서는 pass-by-value와 call-by-reference를 지원하지만, 다른 방식까지 세분화해서 지원하는 PL도 존재한다고 함.

1) Pass-by-Value(In)
Pass-by-Value 또는 Call-by-Value는 actual parameter의 값이 formal parameter를 초기화하는 데에 사용되는 방식임.

빠르지만, 추가적인 메모리 공간이 필요함. 또한 값을 복사하는 작업의 overhead가 클 수 있음.

2) Pass-by-Result(Out)
Pass-by-Result는 해당 formal parameter가 local variable처럼 사용되다가, subprogram이 종료되면 caller의 actual parameter로 그 값이 전달되는 방식임. 이 경우 formal parameter로의 값 전달은 수행되지 않음.

pass-by-value와 마찬가지로 추가적인 메모리 공간이 필요하고, copy에 따른 overhead가 존재할 수 있음. 또한 동일한 actual parameter가 여러 개 존재하는 경우 처리 순서를 결정해 줘야 함.

3) Pass-by-Value-Result(Inout)
Pass-by-Value-Result는 pass-by-value와 pass-by-result를 모두 활용하는 방식임. 즉, actual parameter의 값이 formal parameter를 초기화하는 데에 사용되고, subprogram 종류 이후 해당 값이 actual parameter로 전달됨.

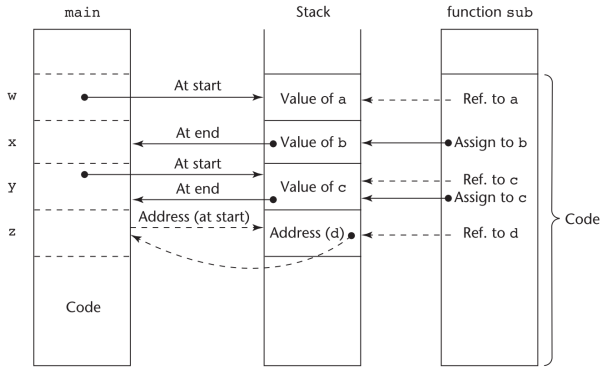
추가적인 메모리 공간이 필요하고, 값을 복사하는 작업이 두 번 발생하므로 그 overhead가 더 커짐.

Ada에서는 in out keyword를 사용해 이를 활용할 수 있음.

4) Pass-by-Reference(Inout)
Pass-by-Reference 또는 Call-by-Reference는 값이 아니라 access path(주로 address)를 전달하는 방식임.

값의 전달 과정이 효율적이지만, 더 느릴 수 있고 side effect가 발생할 수 있음.

5) Pass-by-Name(Inout)
Pass-by-Name은 subprogram 내에 존재하는 해당 formal parameter가 actual parameter로 textual substitution되는 방식임. 이때 formal parameter는 subprogram call 시에 access method에 binding되고, 값이나 주소에 대한 실제 binding은 할당 또는 참조가 발생할 때까지 지연됨.



Function header: **void** sub (**int** a, **int** b, **int** c, **int** d)
Function call in main: sub (w,x,y,z)
(pass w by value, x by result, y by value-result, z by reference)

3. Parameter Passing in Various Languages

c에서는 pass-by-value를 default로 사용하고, pointer에 의한 pass-by-reference를 지원함.

c++에서는 추가로 Reference Type이라는 특별한 pointer type을 지원함. 이는 implicit하게 dereference 되고, 실제로는 pass-by-reference로 동작함. 또한 c++에서는 reference type을 해당 subprogram에서 constant하게 사용하도록 할 수도 있음.

```
void fun(const int &p1, int p2, int &p3)
{
    ...
}
```

java에서는 c/c++처럼 pass-by-value가 default임. 하지만 object에 대해서는 pass-by-reference가 적용됨.

c#에서도 pass-by-value가 default이지만, ref keyword를 작성하여 pass-by-reference도 활용할 수 있음.

```
void sumer(ref int old, int new)
{
    ...
}
...
sumer(ref sum, newValue)
```

c#과 java에서 array는 object로, 각 element가 array일 수 있는 single-dimension임. 각 array는 길이를 저장하는 하나의 named constant를 가짐.

2.3.3. Local Referencing Environment

subprogram은 local referencing environment를 가짐. local referencing environment에 포함되는 local variable과 nested subprogram에 대해 알아보자.

1. Local Variable

subprogram 내부에서 정의된 variable은 대체로 scope가 해당 subprogram으로 제한되는 경우가 많으므로, 이를 Local variable이라고 함.

대부분의 PL에서 local variable은 stack dynamic인 것이 default임. stack dynamic local variable은 메모리 효율적일 수 있고, recursion의 구현에 활용할 수 있음.

static인 local variable은 stack dynamic에 비해 1) allocation과 deallocation에 따른 runtime overhead가 없고, 2) 직접 access하므로 명확하고 효율적이고, 3) 해당 subprogram이 history sensitive할 수 있도록 함. 하지만 static local variable은 recursion을 구현하지 못함.

c에서는 local에서 variable을 정의하면 default로 stack dynamic이고, static keyword를 붙여 정의하면 static이 됨.

2. Nested Subprogram
Nested Subprogram은 subprogram 안에서 정의된 subprogram임. 이에 따라 scope가 global, local, non-local로 구분되었음.

nested subprogram은 ALGOL 60으로부터 시작되었고, 이를 지원하는 PL은 ALGOL 60의 후손 PL들임. 반면 c의 후손 PL들은 이를 지원하지 않음.

2.3.4. Calling Subprograms Indirectly

어떤 subprogram이 call되어야 하는지가 runtime에 결정되는 상황 등에서는 pointer나 reference를 활용해 subprogram call이 indirect하게 수행되어야 함.

c/c++에서는 function pointer를 통해 indirect call을 지원함. 특히 c++에서는 function pointer의 타입이 protocol(반환값, paramter types)에 의해 결정되어, protocol이 같은 function들에 대한 indirect call을 구현할 수 있음. c/c++에서는 function/array의 이름이 해당 function/array의 주소이므로, 이를 pointer에 할당하여 활용할 수 있음.

c#에서 Delegate는 메소드에 대한 참조를 저장하는 객체임. 이를 통해 메소드를 변수처럼 활용할 수 있음. 아래와 같이 delegate 객체를 생성하고, 동일한 protocol을 가지는 메소드를 입력하여 활용할 수 있음. 이를 통해 indirect call이 구현됨.

```
public delegate int Change(int x);

Change chgfuc1 = new Change(myFun); // 방법1
Change chgfuc2 = myFun; // 방법2

chgfun2(12); // myFun 호출 가능
```

2.3.5. Overloading and Generic

1. Overloading
Overloaded Subprogram은 하나의 referencing environment에 동일한 name을 가지면서, 다른 protocol을 가지는 subprogram이 존재하는 subprogram임.

*Overloaded Operator는 operand에 의해 그 타입과 동작이 결정되는 operator임. 예를 들어, java에서 * 는 operand에 의해 integer 연산이 될 수도 있고, floating point 연산이 될 수도 있음.*

Ada, c++, python, Ruby 등에서는 사용자가 overloaded operator를 정의할 수 있음.

2. Generic
Polymorphic Subprogram은 각 call에서 서로 다른 타입의 parameter를 받을 수 있는 subprogram임.

특히 동일한 알고리즘으로 서로 다른 타입의 데이터를 다루는 경우 이를 활용하여 software productivity를 확보할 수 있음. 이런 polymorphism은 여러 형태가 존재할 수 있는데, 여기에서는 아래의 두 가지를 살펴보자.

1) Ad hoc polymorphic subprogram : 서로 다른 타입을 처리하는 subprogram들이 항상 유사하게 동작하는 것은 아닌 polymorphic subprogram. overloaded subprogram에서 이를 구현함.

2) Parametric polymorphic subprogram : 서로 다른 타입을 처리하는 subprogram들이 항상 유사하게 동작하는 polymorphic subprogram. generic parameter를 활용하는 subprogram인 Generic Subprogram에서 이를 구현하는데, 대체로 type expression에서 parameter의 type을 지정함.

c++, java, c# 등에서는 compile time에서의 parametric polymorphism을 지원함. 예를 들어, java에서는 아래와 같이 작성함.

```
generic_class<T>
...
// Comparable의 하위 타입만 T로 활용될 수 있음
public static <T extends Comparable> T doSomething() {
    ...
}
...
public void drawAll(ArrayList<? extends Shape>) things) {
    ...
}
```

c++에서는 generic을 template이라고 함.

2.3.6. Closure

Closure는 subprogram과, 정의된 시점에서의 referencing environment의 묶음으로, subprogram이 정의될 때의 variable 상태를 저장해 두는 문법임.

대부분의 functional PL, scripting language, c# 등 static-scope이고, nested subprogram을 지원하고, subprogram을 parameter로 전달할 수 있는 PL에서 closure를 지원함.

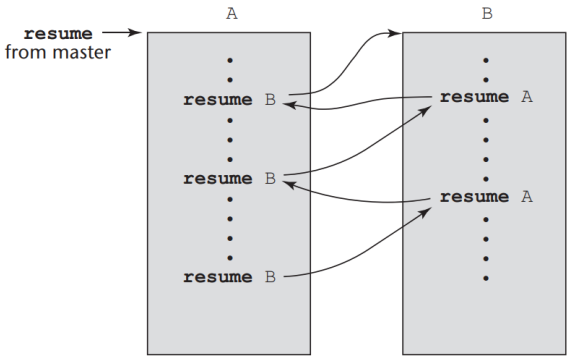
아래는 javascript의 closure 예시임.

```
function makeAdder(x) {
 return function(y) {return x + y;}
}
. . .
var add10 = makeAdder(10);
var add5 = makeAdder(5);
document.write("Add 10 to 20: " + add10(20) + "<br />");
document.write("Add 5 to 20: " + add5(20) + "<br />");
```

2.3.7. Coroutine

Coroutine은 subprogram을 일시 중단하고 실행할 수 있는 문법임. 일반적인 subprogram에서 caller와 subprogram이 master-slave 관계인 것과는 달리, caller와 coroutine은 비교적 동등하게 실행됨. coroutine의 제어 방식을 Symmetric Unit Control Model이라고도 함.

예를 들어, 아래와 같이 subprogram을 번갈아 실행할 수 있음.



2.4. Implementation of Subprogram

subprogram의 implementation에 대해 알아보자.

2.4.1. Implementation of Subprogram

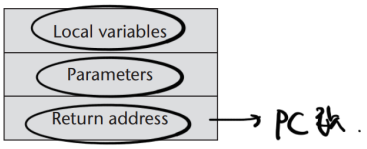
subprogram의 call과 return의 동작 과정을 통틀어 Subprogram Linkage라고 함. 이는 subprogram의 구현에 대한 주요 사항들을 포함함. subprogram의 구현은 call과 return의 동작을 중점으로 살펴볼 수 있음.

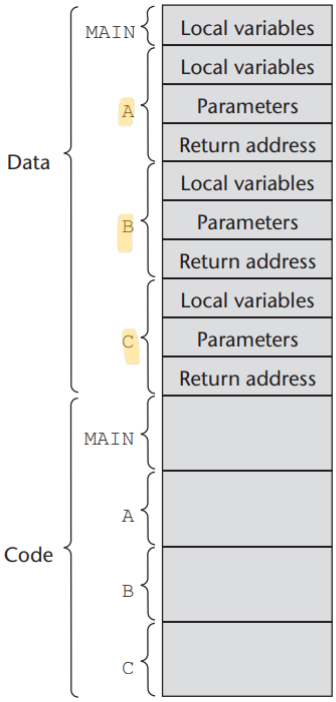
Activation Record는 subprogram의 noncode(data) part에 대한 format으로, 이는 static하게 정의됨. 어떤 subprogram이 실행될 때의 메모리 공간은 activation record을 따르고, 이를 Activation Record Instance라고 함.

이제 subprogram의 각 문법 요소에 따른 implementation을 알아보자. 즉, activation record가 어떤 구조로 되어 있는지를 봄.

1. Simple Subprogram
local variable이 static하게 할당되는 단순한 subprogram에서 activation record는 local variable, parameter, return address의 순서로 구성됨.

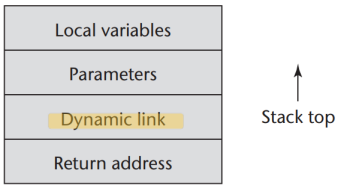
이때 변수는 코드에서 등장하는 순서대로 저장되며, parameter가 위치 상 먼저 등장하므로 local variable 보다 먼저 저장됨.





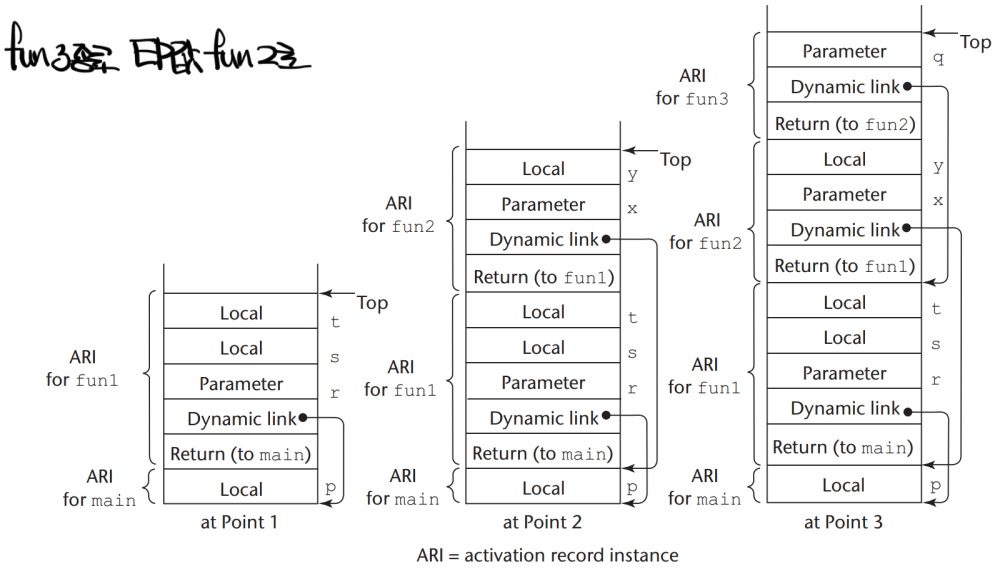
2. Stack-Dynamic Local Variable
local variable이 stack-dynamic인 경우에는 1) compiler가 implicit allocation을 수행해야 하고, 2) recursion에 따라 임의의 개수의 subprogram activation이 존재할 수 있으므로 처리가 더 복잡함. static과는 달리, 당연하게도 stack dynamic에서는 컴파일러가 컴파일 타임에 주소를 할당하지 못함.

이 경우 activation record에 dynamic link와 EP를 추가로 활용하고, activation record는 local variable, parameter, dynamic link, return address의 순서로 구성됨.



Dynamic Link는 caller의 activation record instance의 base를 가리키는 pointer임. subprogram 종료 이후 caller로 돌아갈 때, code에서는 PC를 사용하지만 data에서는 dynamic link를 사용함.

EP(Environment Pointer)는 현재 제어가 실행 중인 subprogram의 activation record instance의 base를 가리키는 register(pointer)임. 각 variable은 EP로부터의 Offset(Displacement, local_offset)을 가지고, EP와 offset을 활용해 주소가 계산됨. subprogram이 call되면 현재의 EP값은 새로운 subprogram의 activation record instance의 dynamic link로 저장되고, 새로운 activation record instance의 base 주소를 가지게 됨.



이때 특정 시점에 stack에 존재하는 dynamic link의 집합을 Dynamic Chain 또는 Call Chain이라고 함. 이는 현재 제어가 실행 중인 subprogram까지 어떤 실행 과정이 수행되었는지에 대한 dynamic history를 나타냄.

3. Nested Subprogram

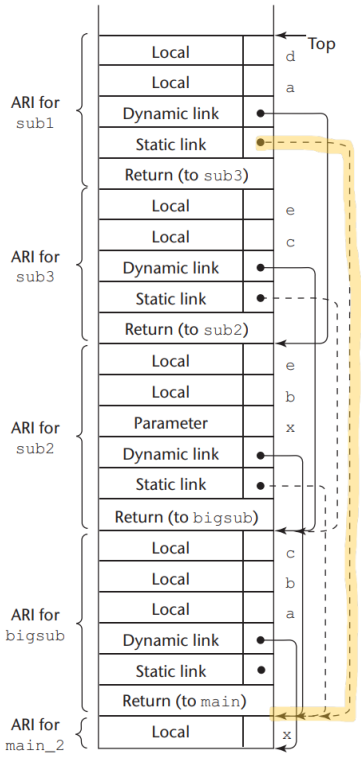
nested subprogram이 지원되는 경우에는 부모의(nonlocal) 변수도 참조할 수 있어야 하므로, 부모의 EP 값을 저장하고 있어야 함. 이에 따라 activation record에 추가로 Static Link 또는 Static Scope Pointer라고 부르는 field를 사용하여 static parent가 가지는 activation record instance의 base 값을 저장함.

참고로 앞에서 다룬 것처럼 static parent는 코드 상에서의 parent로, 해당 subprogram을 정의한 subprogram임.

Static Chain은 stack의 특정 static link들을 연결하는 chain임. 어떤 subprogram의 static link는 그 부모의 EP 값을, 그 부모는 또 그 부모의 EP 값을 가지게 됨. static chain을 활용해 static link를 따라가며 nonlocal 변수를 찾아낼 수 있음. static_depth는 연쇄적인 nesting에서 얼마나 깊이 nested되어 있는지를 나타내는 정수로, nested되지 않았다면 0이고, nested될 때마다 1씩 커짐.

chain_offset은 특정 변수에 도달하기 위해 거쳐야 하는 link의 개수임. nested subprogram이 존재하는 환경에서 어떤 변수를 특정할 때, (chain_offset, local_offset)으로 나타낼 수 있음.

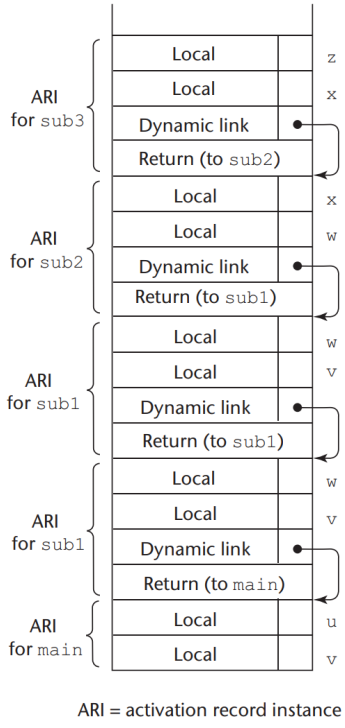
참고로 아래 그림에선 점선이 static link를 나타냄.



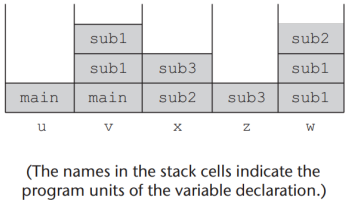
4. Dynamic Scoping

dynamic scoping을 구현하는 방법에는 deep access와 shallow access가 존재함. 이 둘은 구현 방식에는 차이가 있지만, 동작 결과는 동일함.

1) Deep Access : nested subprogram을 구현한 방식과 유사하게, dynamic link들의 집합인 Dynamic Chain을 활용해 nonlocal 변수에 접근하는 방법.



2) Shallow Access : 변수명마다 별도의 stack을 사용하고, subprogram이 호출될 때 특정 이름의 변수가 선언되면 해당 이름의 stack에 셀이 생성됨. 이후 어떤 변수가 호출되면 대응되는 stack의 가장 위에 있는 셀을 활용함.



(The names in the stack cells indicate the program units of the variable declaration.)

c-based language들에서는 Block을 지원하는데, 이 경우도 dynamic하게 할당이 가능함.

3. Functional/Logic PL

3.1. Functional/Logic PL

3.1.1. Functional PL

Functional PL은 수학적 함수를 기반으로 하는 PL로, 프로그램을 함수의 집합으로 봄. 이에 따라 imperative PL과는 다른 형태로 문제에 접근하게 됨.

(연산자 값1 값2) 형태가 functional PL의 기본 형태임.

Mathematical Function은 domain set의 원소와 range set의 원소 사이의 mapping임. mathematical function의 특징으로는 아래와 같은 것들이 있음.

1) iteration보다는 주로 recursion, condition에 의해 제어됨.
2) external value를 활용하지 못하므로 side effect가 존재하지 않음.

Lisp는 가장 오래되고 널리 쓰인 funtional PL임. 이후 scheme, common lisp 등의 아류가 만들어졌음.

functiona PL은 imperative PL에 비해 더 단순함. 또한 일부 functional PL 사용자들에 의하면 훨씬 빠르다고 함. 하지만 functional PL과 imperative PL 사이의 문법적 차이 때문에, 이미 imperative PL이 대부분을 점유하고 있는 상황에서 functional PL으로의 전환이 쉽지 않다고 함.

3.1.2. Logic PL

1. Logic PL
Logic PL은 symbolic logic을 사용하고 logical inference를 수행하도록 하는 PL임. 이는 declarative한 방식으로, 어떤 결과를 얻기 위한 과정을 작성하는 대신 그 결과에 대한 세부사항을 제공하는 식으로 동작함. logic PL은 fact와 rule로 구성되어 있으며, 질문이 들어오면 해당 fact와 rule로 적절한 결과를 도출함.

이는 주로 RDBMS, expert system, NLP 등에서 활용됨.

2. Proposition
Compound Term은 수학에서의 함수 하나를 나타내는 개념으로, function symbol인 Functor와 parameter의 ordered list로 구성됨. 가장 간단한 형태의 proposition(명제)인 Atomic Proposition은 compound term으로 구성됨.

Compound Proposition은 둘 이상의 atomic proposition들이 logical operator에 의해 복합적으로 구성된 proposition임. logical operator로는 아래와 같은 것들이 있음.

Name	*Symbol*	*Example*	*Meaning*
negation	¬	¬ a	not *a*
conjunction	⋂	*a* ⋂ *b*	*a* and *b*
disjunction	⋃	*a* ⋃ *b*	*a* or *b*
equivalence	≡	*a* ≡ *b*	*a* is equivalent to *b*
implication	⊃	*a* ⊃ *b*	*a* implies *b*
	⊂	*a* ⊂ *b*	*b* implies *a*

또한 아래와 같은 Quantifier가 존재하는데, compound proposition에서 variable은 quantifier가 존재해야 등장할 수 있음.

Name	*Example*	*Meaning*
universal	∀ *X.P*	For all *X*, *P* is true.
existential	∃ *X.P*	There exists a value of *X* such that *P* is true.

예를 들어, 아래와 같이 작성할 수 있음.

likes (bob, trout) ⊂ likes (bob, fish) ⋂ fish (trout)

proposition을 활용한 inference rule을 Resolution이라고 함.

3. Prolog

prolog은 가장 유명한 logic PL 중 하나임.

prolog에서는 하나의 문장을 Term이라고 함. term은 Constant(Atom 또는 integer), Variable, Structure(함수에 대응되는 개념)로 나뉨.

prolog에서 Fact는 아래와 같은 형식으로 씀. 예를 들어, female(shelley)이면 shelley가 female이라는 fact임.

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

Rule은 아래와 같이 :-로 구분하고 왼쪽은 consequence, 오른쪽은 antecedent_expression을 나타내는 Head Horn clause로 나타냄. 여기에서 A :- B이면 B implies A로 이해할 수 있다.

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
```

Theorm은 logci PL이 구성한 system이 prove하기를 기대하는 proposition으로, prolog에서는 이를 Goal 또는 Query라고 함. system은 yes로 prove되었음을, no로 prove되지 못했음을 나타냄. 이때 prove하는 과정을 Inference라고 함.

inference에는 bottom-up, top-down 등 여러 방식이 존재함.

4. 기타

4.1. TOY

4.1.1. TOY

1. TOY
TOY는 단순한 형태의 Lisp-like funcational language임. TOY는 statement로 구성된 다른 language와 달리, term(용어)으로 구성되어 있음.

TOY는 MINUS와 IF를 primitive function으로 가지고, 이를 활용하여 사용자가 함수를 정의할 수 있도록 함.

L은 모든 가능한 term의 집합으로, 아래와 같이 정의됨.

t1) variable v1, v2, ...는 term임.
t2) integer constant ..., -2, -1, 0, 1, 2, ...은 term임.
t3) t1과 t2가 term인 경우, (MINUS t1 t2)와 (IF t1 t2)는 term임.
t4) fn이 정의된 function name이고, s1, ..., sk가 term인 경우 (fn s1 ... sk)는 term임.

Z는 모든 integer에 대한 집합임. TOY에서는 이를 연산의 대상으로 함.

2. DEFUN
DEFUN은 TOY에서 함수를 정의하는 데에 사용하는 매크로임. 아래와 같이 형식으로 함수를 정의함. DEFUN은 각 argument를 검사하지 않고 단순히 정의를 생성함. DEFUN은 매크로이지만 실행된 경우 정의한 함수의 name을 반환함(이는 값을 얻는 것이 아니라 정의를 위해 사용됨.).

TOY에서는 함수가 항상 ()로 둘러싸여 있음.

```
(DEFUN <function name> (<parameter 1> < parameter 2> . . . < parameter n>)
<process description>)

Example: (DEFUN ADD (x y) (MINUS x (MINUS 0 y)))
```

3. TOY의 함수들
TOY는 기본적으로 아래의 함수들을 가짐. 이는 prefix가 기본임. 즉, TOY에서 MINUS와 IF는 공리, 나머지는 theorem임.

1) MINUS
MINUS는 아래와 같이 작성하며, a에서 b를 빼서 그 결과를 반환함. 빼기 연산이 있으면 사칙연산 전부를 구현할 수 있음.

```
(MINUS a b)
```

2) IF
IF는 아래와 같이 작성하며, a는 조건이고 b는 조건이 참이면 반환하는 값임. a가 양수이면 참이고, 0 또는 음수이면 거짓임. 거짓인 경우 b 대신 0이 반환됨.

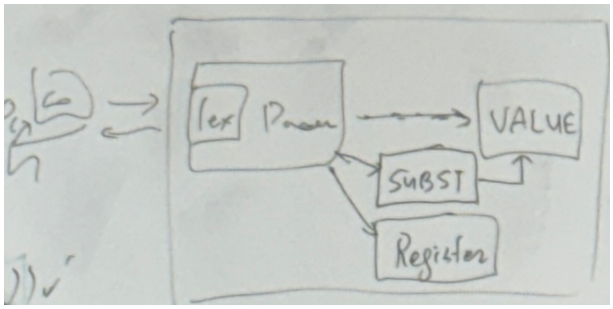
```
(IF a b)
```

반복문도 if로 구현됨. 물론 goto가 있어야 함.

BNF로 나타내 보자. terminal은 const id ( ) if minus이고, non-terminal은 <term>임.

<term> -> const | id | (M <term> <term>) | (if <term> <term>)

3. TOY Interpreter
TOY Interpreter는 interpreting term을 활용해 구현됨. Interpreting Term은 기본적인 term을 활용해 구성되는 higher-level expression임. 해당 interpreter의 앞 단은 lex와 parser로 구성되는 것으로 이해할 수 있음.



interpreting term는 아래와 같은 함수들로 정의 및 평가됨. 즉, 이 함수들을 사용해 TOY 함수를 나타냄.

1) VALUE<term> : term을 정수 값으로 매핑하는 함수.

2) SUBST<fn (n1 … nk)> : TOY 함수에 대한 정의를 활용하여, formal parameter를 actual parameter로 변환하여 term을 반환하는 함수.

3) APPLY<fn (n1 … nk)> : SUBST을 적용하고 VALUE로 그 값을 얻는 함수.

즉, 아래와 같이 구현됨.

(v1) VALUE< u > = undefined if u is a variable,
(v2) VALUE< n > = n if n is an integer,
(v3) VALUE< (MINUS t1 t2) >
= < t1 – t2 > if t1 and t2 are integers,
= VALUE< (MINUS VALUE<t1> VALUE<t2>) > otherwise.
VALUE< (IF t1 t2) >
= VALUE<t2> if t1 is positive integer and t2 has a value.
= 0 if t1 is 0 or a negative integer,
= VALUE< (IF VALUE<t1> t2) > otherwise
(v4) VALUE< (fn (s1 .. sk) >
= APPKY<fn (VALUE<s1> .. VALUE<sk>) > if fn is neither IF nor MUNUS
(a) APPLY< fn (n1 .. nk) >.
= VALUE< SUBST<fn (n1 ,. nk) > >.
Where the n1 .. nk are integers corresponding to the values of
VALUE<s1> .. VALUE<sk>.

TOY program은 (fn n1 … nk) 꼴이고, APPLY<fn (n1 … nk)>로 그 결과가 계산됨.

4. Recursive Term
함수를 직접 재귀 또는 간접 재귀를 사용해 recursive하게 구성할 수 있음. 이를 활용하면 곱셈을 구현할 수 있음.

TOY에서 MINUS와 IF를 활용해 아래와 같이 여러 함수를 구현할 수 있음.

```
(DEFUN ADD (x y) (MINUS x (MINUS 0 y) ))
(DEFUN EQUAL (x y) (MINUS (MINUS 1 (IF (MINUS x y) 1)) (IF (MINUS y x) 1)) )
(DEFUN POS (x) (IF x 1) )
(DEFUN ZERO (x) (EQUAL x 0) )
(DEFUN NEG (x) (IF (MINUS 0 x) 1) )
```

아래와 같이 IF/THEN/ELSE와 곱셈도 구현이 가능함.

Let IF/THEN/ELSE denote
(ADD (IF x y) (IF (MINUS 1 x) z))
The value of the term
(IF/THEN/ELSE x y z)
is VALUE<y> if VALUE<x> is positive,
is VALUE<z> if VALUE<x> is zero or negative,
is undefined otherwise.

Let TIMES name the term

(IF/THEN/ELSE x (ADD y (TIMES (MINUS x 1) y)) 0)

We will now prove, for non-negative n and all k, that

(TIMES n k)

yields the product of n with k.

참고로 함수란 domain(정의역)의 값과 range(치역)의 값 사이의 mapping임. TOY는 domain과 range 모두 Z임.

함수 정의에 존재하는 parameter를 Formal Parameter(가인수)라고 함. 함수 호출 시에 작성하는 parameter는 Actual Parameter(실인수)라고 함. formal parameter는 다른 변수로 바뀌어도 동일하지만, actual parameter는 그렇지 않음.