**프로그램 언어론**

1단원 프로그래밍 언어 소개

프언을 배워야 하는 이유

-현재 사용하고 있는 언어의 이해증진

-유용한 프로그래밍을 구사할 수 있는 능력의 증대

-언어 선택의 능력 증대

-새로운 언어의 학습

-새로운 언어의 설계 용이

형식적인 언어의 정의

-컴퓨터와 의사 소통하기 위한 표현법(약간 부적절한 면 존재)

-1940년대 이전: 프로그래밍이 전선 연결(프로그래밍 언어?)

-1940년대 VON Neumann 방식(Program stored computer)

-CPU가 처리할 작업을 명령 코드로 작성

-프로그래밍 개념 시작

-어셈블리 언어 탄생: symbol 사용

->기계종속적

->저급 수준의 추상화

->읽고 쓰기 어려움

-고급 수준의 추상화 요구: 고급 언어 등장

->프로그램 간략, 이해 용이, 이식성 증가

->배정, 반복, 선택 개념

프로그래밍 언어의 정의: 프로그래밍 언어는 기계가 읽을 수 있고, 사람이 읽을 수 있는 형식으로 계산을 기술하는 표현 체계이다.

계산(computation)이란?

-수학적으로 형식적인 정의 가능

-컴퓨터가 처리할 수 있는 작업을 의미(일반적)

기계가 읽을 수 있는(machine-readable)

-효과적인 번역 가능(간결한 구조의 언어 요구)

-단순 번역 알고리즘 존재

-제한된 시간 내에 번역 가능(프로그램 크기 단순 비례 이상적 임)

-문맥 자유 언어(context free language) ->위 조건들 만족

사람이 읽을 수 있는(human-readable)

-기계 독립적인 추상성 제공 -> 자연어 형태

-프로그램 대형화 -> 판독성은 새로운 국면 -> 지역성 요구

-프로그래밍 언어 -> 소프트웨어 개발 환경에 포함되는 경향

프로그래밍 언어 고찰

-언어의 특징들에 대한 기본 개념

-특징들의 구현과 이 때 발생하는 문제점

-특징들의 설계에 대한 기본 개념

-실제 언어의 도입 예: 비교 분석

프로그래밍 언어를 고찰함으로써 얻는 장점

-사용하고 있는 언어의 이해와 효율적인 알고리즘 작성

🡪일부 특징 및 특성은 거의 사용 안되며 이해를 못함

이들이 어떻게 수행되고 구현되는지 표현 안됨(지침서)

프로그래머 시행 착오에 의존하여 이해

🡪특징들 사용의 효율성과 비효율성 이해

예: recursion, 동적 기억 장소 할당, 병행성

🡪효율적 알고리즘 작성

-유용한 프로그래밍 구사 능력

언어->사고

다양한 언어 구조 -> 알고리즘 작성 시 어휘 구사력 증가

어휘를 적재적소 사용

특징 구현 기술 -> 실제 적용 능력 제공(coroutine 등)

좋은 알고리즘 작성

-프로그래밍 언어 선택 능력

언어에 대한 다양한 지식 ->문제 해결에 적절한 언어 선택

새로운 프로그래밍 언어의 학습 능력

새로운 프로그래밍 언어 설계 능력

프로그래밍 언어에서의 추상화(Abstraction)

추상화 개념: 주어진 작업이나 객체를 속성들의 일부분을 가지고 필요한 만큼 묘사할 수 있는 방법을 지원하는 것 필수적인 속성만으로 주어진 것을 묘사하므로 나머지 속성들은 추상화, 은닉 또는 삭제됨

추상화 범주

-자료 추상화(Data Abstraction): 문자열, 수, tree 같은 계산될 자료의 특성을 추상화(string, number, tree등)

-제어 추상화(Algorithm Abstraction): 실행 순서를 제어하는 특성을 추상화

Ex) 반복문(Loop statements), 조건문(Condition statements), 프로시저 호출(Procedure call) 등

추상화에 포함된 정보의 양에 따른 분류

-기본 추상화(Basic Abstraction): 가장 지역적인 기계정보에 대한 추상화

-구조 추상화(Structured Abstraction): 보다 전역적인 정보인 프로그램의 구조에 대한 추상화

-단위 추상화(Unit Abstraction): 단위 프로그램 전체에 대한 정보의 추상화

자료 추상화

-컴퓨터 내부의 자료 표현을 추상화

-관련된 자료의 집합을 추상화

-전형적인 구조 추상화 예: 배열, 레코드 (구조형 자료)

-구조형에 새 이름 부여 기능(새 차원의 추상화)

-자료의 생성과 사용에 대한 정보를 한 장소에 모아두고, 자료의 세부사항에 대한 접근을 제한하는 도구

제어 주상화

-몇 개의 기계 명령어를 모아 이해하기 쉬운 추상 구문으로 만드는 것

-프로그램에서 어떤 검사된 값에 따라 분할된 명령어의 한 그룹을 수행

-프로시저의 집합을 추상화(관련된 프로시저 그룹 추상화), library 형태 지원

-그 외의 제어 추상화 EX)병행 프로그래밍(동기화, 통신 개념 지원): coroutine, task, process

추상화란: readability 증가 목적

계산 전형: 프로그래밍 언어는 컴퓨터의 계산을 모방하는 데서 비롯됨

-명령의 순차적 실행

-기억장소를 표시하는 변수의 사용

-값의 변경을 위한 배정문의 사용

-단점: 비결정적 계산이나 순서 비의존형 계산의 표현에는 부적절, 비효율

함수형 언어: 함수의 평가 및 호출 방법 제공 Ex)Modula-2, LISP

논리형 언어: 기호 논리학에 근거, 계산의 내용만 기술 Ex)Prolog

객체지향 언어: 기억장치 기억장소와 이 기억장소의 값을 변경할 수 있는 연산의 집합 EX)Simular 67, C++, Ada

언어 정의의 범주

-언어에 대한 정확한 형식 기술 방법이 필요

-프로그래밍 과정에서 프로그램 동작의 이해를 위해 필요

-구문론과 의미론으로 구분

구문론(Syntax): 언어 구문에 관한 정의는 형식화가 많이 되어 있음, 대부분의 언어가 문맥 자유(Context-free) 문법을 만족하며 BNF로 정의

🡪Pascal if 문의 context-free grammar의 표현 방법

의미론(Semantics): 의미는 프로그램의 실행 시 어떤 일이 발생하는가를 기술

🡪현재, 의미론에 대한 형식화가 잘 이루어져 있지 않음 Ex)if문의 조건이 거짓으로 판명되었을 때, else 문 이하가 존재하지 않을 경우

🡪사용되고 있는 형식적 정의 방법 Ex)denotational semantics, axiomatic semantics

프로그래밍 언어의 정확한 형식 정의가 필요한 이유

-언어의 구조와 의미의 정확한 기술 없이는 프로그래밍 불가능

-모호함 없는 언어 구현

-프로그래밍 과정에서 프로그램이 어떻게 동작할 것인가에 대한 문제 발생 시 유용

-프로그램 설계 과정을 규격화 하는 방법

디지털 컴퓨터 이전의 언어

최초의 알고리즘

• 찰흙 판, BC 1500 – 3000, 바빌론 근처 메소포타미아

• 60진법 사용 (현재 시, 분, 초 개념에 사용), 부동 소수점, 가감승제 가능

• 대수학 방정식 해결 (This “is a procedure is a procedure.” 표현 사용)

• if x <0 then branch 개념 없음 (음수, 0 개념 없음)

• 알고리즘 반복 작성, 우수한 표기법이었음

Euclid

• B. C. 300 B. C. 300년, 최대공약수 알고리즘 (모국어 사용)

• 수에서 0 개념 없음, 수 1이 제수로 인식되지 않았음

• 특수한 경우 반복시킴

• 바빌로니아로 사람들 보다 별로 진보되지 못함

Charles Babbage (1792 – 1871)

• Difference Engine

-계산 처리를 위한 기계

-유한 차(finite difference)의 원리를 기본으로 함

• Analytical Engine: 현 디지털 컴퓨터의 원리 포함

-현재 컴퓨터와 유사한 구조

-프로그램과 자료를 주기억장치에 저장시키는 저장 개념 없음 (연산카드, 변수카드 사용)

• Ada Augusta(Charles Babbage 조수)

-최초의 컴퓨터 프로그래머 (Analytical Engine 프로그래머)

-Ada의 ‘알고리즘 분석’에 대한 견해 “거의 모든 계산에서 성공적인 수행을 할 수 있는 매우 다양한 배치 방법들이 가능하다. 그리고 이들 배치 방법을 선택하는데 있어서 계산을 수행하는 기계 목적에 맞도록 여러 가지 고려해야 할 점들이 있다. 한 가지 중요한 고려 대상은 계산하는데 필요한 시간을 최소로 줄이는 배치 방법을 선택하는 것이다” 18 최소로 줄이는 배치 방법을 선택하는 것이다.

1930-1940년대

프로그래밍 표기에서 창조적인 것들이 급격히 증가

• Zuse의 Plan Calculus

• Turing의 Turing Machine

• Church의 Lambda Calculus

• Aiken의 Mark-I

• von Neumann von Neumann의 Flow Diagrams Flow Diagrams

디지털 컴퓨터의 초기 개발자에게 알려지지 않음

1950년대- 최초의 프로그래밍 언어

1950년대 초반

• 기계어 사용 (프로그램 저장 방식) 🡪 어셈블리 언어: 기계 의존적, 자연어와 상이한 구문

1950년대 중반

• FORTRAN (FORmula TRANslation)

-1954 ~57, John Backus에 의해 고안된 최초의 고급 언어

-Fortran II Fortran IV Fortran 66 Fortran 77 Fortran 90

-Fortran 컴파일러는 효율적인 기계어 코드를 생성 → 언어의 긴 생명 배열,

-반복 구조, 분기문, 부프로그램 등의 특징은 이후 언어에 큰 영향

1950년대 후반에 등장한 언어

• COBOL (Common Business-Oriented Language): 상업자료처리용 상업자료 처리용

• ALGOL 60 (ALGOrithmic Language 60: 알고리즘 기술, 범용

• LISP (LISt Processor): 기호연산, 리스트 처리용

• APL (A Programming Language): 배열처리용

ALGOL60(58~60년) : 알고리즘 기술을 위한 언어로 개발(연구 및 응용)

• 영향: Pascal, C, Modula-2, Ada 등 Algol 파생어에 영향

• 특징

-양식의 자유화(free-format), 구조적 명령문, 블록 개념 도입

-자료형 선언, Recursion, call-by-value 기법

-스택 기반의 실행 환경, 구문정의를 위한 Backus-Naur Forms Naur Forms의 사용

LISP(50년대 후반): J. McCarthy(MIT大)가 고안, Neumann방식 탈피

• 영향: MacLisp, UTLisp, Franz Lisp, Common Lisp, Scheme

• 특징

-리스트 구조와 함수의 응용을 기본, 인공지능 분야에서 사용

-획일적인 자료구조, S-expression, expression, 함수 응용이 기본 표기법

-run-time 기억장소 할당 제공을 위한 Heap/garbage collection 사용

APL(50년대 후반~60년대 초): K Iverson(Harvard): K. Iverson(Harvard大)이 고안

• 특징: 행렬을 포함하는 수학적 연산의 편리 제공

• 스타일이 기능적, 반복구조 자동 제공 연산자 채택

• 단점: 제어구조 不在, Greek 문자 사용, 프로그램 이해에 어려움

1960년대 – 프로그래밍 언어의 폭증

PL/I(63~64년): IBM Project(IBM 360 시리즈에 사용 계획)

• 새로운 개념 추가 (병행성, 기억 장소 할당, 예외처리)

• IBM의 막강한 지원이 있었으나 많은 문제점 내포

번역기가 크고 작성이 어려움/실행 효율 저하/배우기 어려움, 오류 발생 유도

언어 특성들의 상호 작용으로 인한 신뢰성 저하

Algol68(63~68년)

• Algol60 + 타 언어의 여러 기능 + 표현력 있고 일관된 구조

• 새로운 특성 독자 개발

• 이론적으로 일관성 갖는 구조

• 일반적인 형 시스템, 직교성 강조 (수식 위주)

• 언어 기술 → 새로운 용어 과다 사용

• 디자인 일관성, 파일 시스템, 실행 환경 → 우수

• 범용 컴퓨터 사용 불가능 (비판 → 소수 구현)

특기할 기타 언어

• Snobol (StriNg Oriented symBOlic Language)

-R. Griswold가 개발

-최초의 문자열 처리 언어, 패턴 매칭 기능 다양

• Simula67 (Object-Oriented Language)

-Nygaard & Dhal(Norway) 개발

-객체 지향 언어(클래스 개념 최초 도입)

-Simula I + Algol 60

-시뮬레이션 목적

• Basic (Beginners All-purpose Symbolic Instruction Code)

-1964년 J. Kemeny & T. Kurtz(Dartmouth Univ.) 개발

-시분할용의 간결한 언어 마이크로 컴퓨터 이식 (교육용, 사무처리용, 가정용)

-언어군 (ANSI 표준: 1978 ‘minimal Basic’, 1988 ‘Standard Basic’)

-후에 나온 Basic: 제어 구조, 변수 선언, 프로시저 추가

1970년대: 간결성, 추상화, 연구 사항

1970년대 초반 언어의 특징

-60년대의 혼돈 후 → 간결성과 일관성 추구

• Niklaus Wirth: Algol 68 디자인에 반발 (방대성)

• Wirth + Hoare: Algol W 제안

1970년대 초반에 등장한 언어

Pascal (1971년): N. Wirth(IBM 360 시리즈에 사용 계획)

-작고, 간결하고, 효율적이고, 구조적이며, 프로그래밍 교육용

-분리 컴파일, 유용한 문자열 조작, 입출력 기능 → 삭제 또는 축소 (성공적)

• C (1972년) (): Dennis Ritchie(Bell Lab.)

-Pascal 과는 다른 각도의 간결성

-수식 위주로 형 시스템과 실행환경 축소, 하드웨어 접근 용이

-중급 언어로 분류: 기계 접근성 강화 (고급 언어와 상반 개념)

-운영체제 프로그래밍용 (Bliss(1971), Forth(1971))

-C 언어로 작성된 UNIX 성공 → C 언어 대중화

• C, Pascal ⇒ 간결성, 디자인 일관성 (小group의 고안): 성공 요인

1970년대 중·후반 언어의 특징

• **자료 추상화, 병행성, 증명** 등의 메커니즘을 집중적으로 시도

• 1970년대 중, 후반에 등장한 언어

• CLU (1974-77, MIT의 Babara Liskov)

-추상화 기법을 위한 일관성 있는 접근 방식 - **자료 추상화, 제어 추상화, 예외처리**

-자료 추상화: cluster (Simula의 class와 유사)

-제어 구조: iterator iterator - 일반성

-예외처리: Ada와 비슷한 구조 제안

• Euclid (1976-77, 토론토(Toronto) 대학)

-Pascal을 개선한 언어

-Pascal의 단점인 이명(aliasing) 개선, 자료형의 추상화, 프로그램 증명 보조 추가

-프로그램의 형식적 검증을 목적으로 한 최초 언어

• Mesa (1976-79, Xerox사 Palo Alto 연구소)

-Pascal 구조에 모듈 구조, 예외처리기, 병행성, 병렬 프로그래밍의 개념 추가

-시스템 프로그래밍용 → Modula-2 메커니즘 고안에 지대한 영향

1980년대: 통합과 새로운 방향

1980년대 언어의 특징

• Ada 등장(개념 통합)

• 함수형 언어에 새로운 관심 부여 - 스킴(Scheme), ML(Meta Language) 개발

• 논리형 프로그래밍 언어 - Prolog 등장

• 객체 지향 언어에 대한 활발한 연구 및 개발

1980년대 등장한 언어

• Ada: J. Ichbiah Team (미 국방성의 공모작 수정)

-자료 추상화, 타입 메커니즘(package), 병행 처리(task), 예외처리기 도입

-디자인이 섬세하고 상세함, 사용 급증

-단점: 복잡 방대 (제2의 PL/I 가능성)

• Modula-2: 1982-88, Modula 언어를 기본, 운영체제 구축용(N. Wirth)

-Pascal의 디자인에 결점 보완(당시 실험적 언어)

-추상화, 부분적 동시처리 개념, 교육 목적으로 인기

-내장형 시스템 프로그래밍 목적: 하드웨어 접근이 용이한 다목적 언어

-가능한 한 소규모의 간결한 언어로 시도 → 예외처리 기능 등 제외

•Prolog (1972- , A. Colmerauer의 그룹)

-논리형 언어

-인공지능 분야에서 많이 사용

• SETL (뉴욕 대학(NYU)의 J. Schwartz)

-집합론을 프로그래밍 언어로 시 한도 것 (미적분, 수학의 교육 모델로 사용)

-구현 결핍

• Smalltalk (1972-80, Xerox사의 Alan Kay, Dan Ingalls)

-객체 지향 언어의 모범

• C++ (1980, Bjarne Stroustrup)

-C 언어를 확장한 객체지향 언어

• Eiffel (1980년대 중반, Bertrand Meyer)

-Pascal에 가까우면서 일관성 있는 객체지향 언어

1980년대 등장한 함수형 언어

• Scheme: 1975-78, MIT의 Gerald J. Sussman과 Guy L. Steele, Jr. 개발

-Lisp 언어 개정

-Lisp보다 획일적이고, lambda calculus에 가깝게 설계됨

• Common Lisp: 표준

• ML(Meta Language): 1978, 에딘버러(Edinburgh) 대학의 Robin Milner 개발

-기존 함수형 언어와 相異함

-Pascal과 유사하나 유연성 (문법, Type checking) Type checking)

• Miranda :1985-86, 맨체스터 대학의 David Turner David Turner가 개발

1990년대: World Wide Web 프로그래밍 (Java)

1990년대 언어의 특징

• 제4세대 언어 대두

-응용문제를 빠르게 구현

-실사용자가 직접 프로그래밍

-데이터베이스를 쉽게 처리

• HTML, Java 등장

1990년대 등장한 언어

•Java

-James Gosling 설계 팀장

-C++ 기반: 많은 구조 삭제, 일부 구조 변경, 일부 구조 추가

♣ C++의 강력함과 유연성 제공

♣ C++보다 규모는 작아지고, 더 간결하고, 신뢰성이 증가된 언어

-한 응용 분야 목표: 내장 시스템

- Java의 응용 분야가 확장됨 ⇒ Web 프로그래밍 (C/C++결점 보완)

Java 특징: 간결성과 신뢰성 제공 원칙

• 자료형, 클래스 제공

• 기본 자료형: 스칼라 형 제공

• 배열 – 미리 정의된 클래스 객체로 제공(C++와 구별)

• 포인터 삭제 ⇒ 참조형 제공

포인터 → 기억장소 지시, 참조형 → 객체 지시

• 논리형 제공 (산술 연산에 사용 불가: C/C++: C/C++와 구별),

• 부프로그램 제공 안함 ⇒ class method 제공

• 단일 상속 ⇒ 다중 상속 변칙 사용 (인터페이스 사용)

• 동시성 제공 (synchronize) - Thread • 쓰레기 수집

• 묵시적 형 변화 (implicit type conversion) - 확대형 변환(widening)

• 중간 언어 제공 (이식성 증가)

• Java 애플릿 - client에서 실행

• 프로그래머 - Java 선호 (C++기피)

프로그래밍 언어의 세대론과 미래

4세대 언어의 특징

• 소프트웨어의 위기 (software crisis)

• 4세대 언어들은 데이터베이스 시스템을 위해 만들어진 명령어로 출발 에디터(editor)나 디버거(debugger), 서류작성기나 제어용 유틸리티와 같은 개발도구들과 함께 디자인 환경에 포함

• 대표적 예: SQL(Structured Query Language), Lotus, Delphi 등

• 거대한 File 처리 응용 프로그램을 몇 개 안 되는 코드로 신속하게 작성

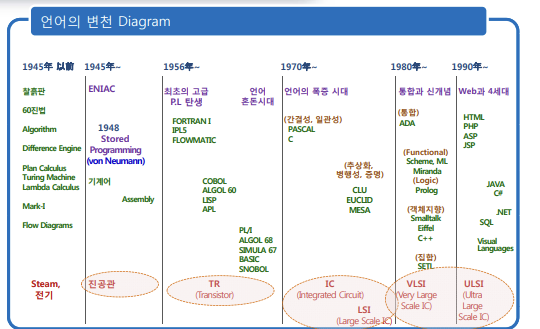
• 언어 설계 이 에론 공헌한 바는 작지만 실용성 큼

• 최근의 Visual Basic, Visual C++, Power Builder도 이 부류에 속함

명세 언어

• 사용자가 원하는 바를 서술하면 시스템이 요구 사항을 구현해 주는 언어

• 1979, Winograd의 명세 언어에 대한 견해 → 논리형 언어와 제5세대 언어가 시도하는 바를 기술

“고급 언어는 기계어 코드의 복잡함과 난해함을 탈피할 수 있게 하며, 고급 언어 시스템은 복잡한 시스템이나 구성 요 소를 이해하여 다룰 수 있게 한다. 이제는 알고리즘의 세부적인 명세로부터 떠나서 우리가 작성한 패키지나 객체의 성질을 기술하는데 집중시킬 필요가 있다. 즉, 새로운 세대의 프로그래밍 도구가 될 조건은 프로그래밍 시스템이 명령 이 아닌 선언인데 프로그램의 주 작업이 알고리즘을 수행하는 명령문들의 나열보다는 계산과정과 계산이 수행되는 객 “고급 언어는 기계어 코드의 복잡함과 난해함을 탈피할 수 있게 하며, 고급 언어 시스템은 복잡한 시스템이나 구성 요 소를 이해하여 다룰 수 있게 한다. 이제는 알고리즘의 세부적인 명세로부터 떠나서 우리가 작성한 패키지나 객체의 성질을 기술하는데 집중시킬 필요가 있다. 즉, 새로운 세대의 프로그래밍 도구가 될 조건은 프로그래밍 시스템이 명령 이 아닌 선언인데 프로그램의 주 작업이 알고리즘을 수행하는 명령문들의 나열보다는 계산과정과 계산이 수행되는 객 34 이 아닌 선언인데, 프로그램의 주 작업이 알고리즘을 수행하는 명령문들의 나열보다는 계산과정과 계산이 수행되는 객 체(object)를 묘사하는 작업을 수행하는 방향으로 가게 될 것이다.” 

프로그래밍 언어의 세대론과 미래

소프트웨어 위기 (software crisis)의 극복

• 언어적 기술보다는 조직적인 방법으로 해결

기존 코드의 재사용 증대

이식성 증대

문법 위주의 에디터를 사용하여 프로그래머의 생산성 높임

• 언어 설계의 미래

하드웨어와 컴퓨터 구조의 발달에 의해 영향 받음

새로운 언어의 출현보다는 기존 언어들을 갈고 닦아, 언어 설계에 대한 이해를 점진적으로 증가시키는 방향으로 예상됨

주요 언어의 설계 목적

⚫ Fortran – 실행의 효율성

⚫ Cobol – 영어와 유사한 문법 구조로 program의 easy readability

⚫ Algol60 – 블록 구조 제공으로 알고리즘 작성 용이

⚫ Pascal – 간단한 명령형 언어, 하향식 설계 증진

P.L 역사와 설계 기준

1950년대

• 초기 설계기준: Efficiency of execution (Fortran)

• Cobol과 Algol 60의 등장으로 효율성보다 일반적인 원칙 중시

-Algol 60: 블록구조, 재귀적 용법으로 논리적으로 간결 명료한 알고리즘 표현 용이

-Cobol: 프로그램 판독성 증가 (영어 유사 문법 구조)

1960년대

• 복잡성 제어(complexity control) 필요성 인식 - 추상화 기법, 언어 규칙과 제한의 감소 필요성

-Simula67: 강력한 추상화 기법 제공

⇒ Algol68: 일반성, 직교성(orthogonality) 제공으로 언어의 복잡성 감소

1970년대 - 1980년대 초 • 간결성(simplicity)과 추상화(abstraction)를 강조

예) Pascal, C, Euclid, Modula-2, Ada

• 언어 구성에 수학적 정의 도입

• 프로그램의 정확성 증명 기법을 갖춘 언어 제공 ⇒ 프로그램 신뢰성 증진

1980년대

• 언어에 논리 또는 수학 개념의 삽입 증진

• 논리를 프로그래밍 언어 자체에 포함

• 함수형 언어에 관심 (ML, Miranda, Scheme)

• 객체 지향 언어에 관심 증가 (C++)

언어 설계의 기본 원칙

• 효율성 (efficiency)

• 일반성 (generality)

• 직교성 (orthogonality)

• 획일성 (uniformity)

기타 설계 원칙

• 간결성 (simplicity)

• 안전성 (security)

• 표현력 (expressiveness)

• 기존 표기나 규칙과의 일관성

• 정확성 (preciseness)

• 확장성 (extensibility)

• 기계 독립성 (machine independence)

• 제약성 (retractability), 부분성 (subset)

효율성의 기준에 따른 분류

• 목적 코드의 효율성

-번역기의 효율적 실행 코드 생성 → 최적화 (code optimization)

-예) Pascal에서 상수는 수식으로 표현되지 않음. 상수 식별자는 번역과정시 배정된 값으로 대체

• 번역의 효율성

-적절한 크기의 번역기로 빠르게 번역할 수 있는 것

-예) 언어 번역의 단계 구성 문제 (Pascal: 1-pass, Modula-2 : 2-pass)

• 구현의 효율성

-번역기의 효율적 작성 가능 여부

-예) 번역기 구현의 어려움, 번역 수행 알고리즘의 충분치 못한 이해로 실패한 언어 (Algol 60)

• 프로그래밍 효율성

-프로그램 작성의 단순성, 용이성 문제

-언어의 표현성, 추상화 메커니즘과 관련

-이상적인 언어 - Lisp, Prolog

❖ 실제 효율성은 신뢰도(reliability)와 관계됨

❖ 생성된 Software의 효율성: readability와 maintainability에 좌우

일반성(Generality)

• 특별한 경우를 피하거나 밀접한 관련의 여러 개념들을 하나의 개념으로 결합

• 일반성이 부족한 경우의 예

프로시저

→ Pascal: 프로시저 선언과 매개 변수 허용, 프로시저형 변수 불허

→ Modula-2: 일반성 보유

→ Ada: 매개 변수에 프로시저 사용 못함

배열

→ Pascal: 가변 배열 불허

→ C, Ada: 가변 배열 허용

→ Modula-2, Fortran: 가변 배열 전달능력, 가변배열 선언 불허

동등 연산자, 배정 연산자(=,:=)

→ 대부분 언어: 배열, 레코드에 적용 불허

→ Ada: 배열, 레코드에 적용 허용

매개변수

→ Fortran: call by reference 만 허용

→ Algol 68, C, Java: call by value, 객체에 대한 포인터를 값으로 전달 가능

→ 일반성 제공 → Ada: 일반성 제공

• 일반성이 갖는 문제점

-언어의 간결성 저하

-언어의 판독성 저하

-언어의 신뢰성 저하

→ 예) C 언어의 포인터 (일반성 제공) - 문제점 제기, Java는 포인터 불허 : 신뢰성과 판독성 문제 해결 Pascal에서는 이명(aliasing)과 위험을 줄이기 위해 포인터가 본질적으로 제한

직교성(Orthogonality)

• 수학 개념에서 유래 “직각 또는 완전히 독립적인 방향”

• 언어의 구성자들이 각각의 의미를 가진 채 결합하는 성질

→ (구성자 간의 상호작용 또는 문맥의 사용이 예상 밖의 행위를 야기하지 않아야 함)

• 비직교성 – 문맥에 의존하는 제한

• 비일반성 – 문맥과 관계없는 제한

• 직교성이 부족한 경우의 예

-함수 반환 값 자료형

→ Pascal: 스칼라형, 포인터형만 허용

→ C: 배열형만 제외

→ Ada: 완벽한 직교성 제공 (모든 자료형 허용)

파일

→ Pascal: 화일형은 특별한 상태 취급 (화일을 프로시저 매개 변수로 전달 금지, 화일 변수는 배정 금지)

→ 대부분 언어: 화일을 라이브러리로 취급 (비직교성 탈피)

문자열

→ Modula-2: 문자열 배정 (작은 문자열 → 더 큰 문자열), 크기가 다른 객체에 대한 유일한 배정

매개변수 전달기법

→ C: 배열 - call by reference, 이외 모든 자료형 - call by value 방식

→ Ada: 모든 자료형 - call by value, result, value-result 허용(직교성 보장)

Algol 68의 중요 설계 목표 - 직교성 보장

획일성(Uniformity)

• 언어 구조들의 외모와 행동에서의 조화에 중점을 둠

• 유사한 것들은 유사하게 보이고 유사한 의미를 갖게 하며, 상이한 것들은 서로 다르게 보이고 서로 다르게 행동하여야 된다는 성질

• 획일성이 부족한 비조화의 예

-if문, while 문: begin-end 구조 요구, repeat 문: begin-end 구조 비 요구

-가변 레코드에서 case 문, case 제어문: 구문 상이 (Modula-2에서 해결)

-함수 값의 반환 방법 - 배정문과 유사 (타 언어 return문 사용으로 해결)

Pascal

→↑: 포인터 선언(↑integer)과 포인터 값(x↑)에 공용 (Modula-2는 POINTER TO로 해결)

→;: Modula-2, Pascal에서 문장 구분자와 선언 종결자로 사용 (C : 종결자로만 사용)

-비획일성은 특별한 문맥에서만 발생되고 구성자들 간의 상호작용으로 볼 수 있으므로 비직교성으로 간주될 수도 있다.

간결성(Simplicity)

• Pascal의 주된 설계 원칙은 간결성

• 직교성, 일반성, 획일성: 간결성 보장 못함 예) Algol 68

• 구성자의 수가 적다고 언어가 간결한 것은 아님

예) Lisp, Prolog: 적은 수의 구성자를 가지나 복잡한 실행시간과 시스템에 의존적

• 과다한 단순성 - 언어 사용에 방해, 표현력이 부족, 많은 제한 발생

표현력(Expressiveness)

• 복잡한 과정이나 구조를 표현하는데 용이함을 의미

• 표현력은 강하나 단순하지 않은 언어 - Lisp, Prolog, Algol 68

• 표현력이 강하면서 단순한 언어

- C 언어 예) while (\* s++ == \* t++);

정확성(Preciseness) ≡ 명확성(Definiteness)

• 언어에 대한 정확한 정의 → 언어의 행위가 예측 가능한 정의의 존재 여부

• 정확한 언어 정의 → 언어의 신뢰도, 번역기의 신뢰도에 영향

기계 독립성 (machine independence)

• 기계 독립적인 언어 정의를 통하여 보장 (호환성 제공)

• 기억 장소 할당과 기계 구조와 별개로 정의된 자료형 사용

안전성 (security)

• 프로그래밍 오류를 줄이고, 오류 발견 용이한 언어 목표

• 언어의 신뢰성과 정확성에 밀접한 관계

• 언어 설계 시 자료형, 형 검사, 변수 선언을 도입

기존 표기나 규칙과의 일관성

• 언어 설계 시 표준화된 특성과 개념을 갖도록 해야 함

• Algol 68 - 표준화된 표기를 잘 따르지 않은 언어

예) type 대신 mode 사용

성공적인 언어 설계를 위한 충고

• 신뢰성

-프로그램의 신뢰성 위해 진단 컴파일러 또는 점검 컴파일러 사용

-Cornell: PL/I diagnostic, C 언어 환경(debugger 포함)

-언어 구문의 과다한 간결성과 생략은 프로그램 판독성을 저하

-적절한 수준의 간결성은 프로그래머에게 좋은 훈련과 프로그램의 신뢰성을 증가

-짧은 프로그램 신뢰성 증진 (APL, 4세대 언어)

• 효율적인 번역

-초기 고급 언어 (Fortran, Cobol ...): 분리 컴파일 제공 → 효율적 번역 가능, 오류 유발

-Algol 68, Pascal(70년대 초반): 신뢰성 강조 → 통합 컴파일러

⇒ Ada: 조화 (분리 컴파일의 장점+통합 컴파일의 장점), specification part, body part로 해결

• 코드 최적화 (optimization)

-효율적인 목적 코드/컴파일링 비용 증가

-반복 수행부 등 일부분만 최적화 → 효과 큼

-실제 컴파일러: 여러 최적화 단계 제공

언어 구문 - 프로그래밍 언어의 기본 문자 집합

정합 순서(collating sequence)

• 문자 또는 문자열에 대한 일반적인 순서언어

예) 0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9 , A < B < C < ... < X < Y < Z

• 언어 구현 시에 결정, 일반적으로 문자의 bit 조합 표현 순서에 영향 (코드 체계 따름)

-프로그래머 정의 가능 (RPG, Snobol)

어휘 구조 용어

**• 어휘 토큰 (lexical token), 어휘 요소 (lexical element), 어휘 단위 (lexical unit)**

-기본 의미 단위를 표현하는 한 개 이상의 프로그래밍 언어의 알파벳 문자로 구성된 문자열

**• 언어 구성자**

-한 개 이상의 어휘 토큰으로 형성되며, 구문적으로 허용된 프로그램의 일부

**• 식별자(identifier), 미리 정의된 식별자, 예약어(reserved-word)**

-언어 어휘를 구성하는 단어나 기호 형태의 문자 알파벳, 변수 이름으로 사용할 수 없음

-장점: 프로그램 판독성 증가, 컴파일러가 기호 테이블을 빠른 시간에 탐색

-많은 예약어 → 언어 확장시 新 예약어와 이전에 사용된 프로그램의 식별자와 중복 우려

• 구분자, 분리자(separator)

BNF(Backus-Naur(Normal) Form) 표기법

• 구문(syntax) 형식을 정의하는 가장 보편적인 표기법

• 한 언어의 구문에 대한 BNF 정의

→ 언어의 문장을 생성하는 생성 규칙(production rule) 정의

• 생성 규칙

→ 생성 규칙의 왼쪽(정의될 대상), 오른쪽에는 그 대상에 대한 정의를 표현

• BNF 표기법에 의한 식별자(identifier) 정의 예

EBNF(Extended Backus-Naur Form) 표기법

• BNF 표기법을 확장하여 보다 읽기 쉽고, 간단하게 표현된 표기법

• BNF보다 추가된 특수한 의미를 갖는 EBNF의 메타 기호

→ 반복: { }, { }0 7 0번 이상 반복

→ 선택: [ ] 0 또는 1번 선택

• { }, [ ], |, ( ), ::=와 같은 메타 기호를 언어의 terminal로 사용하는 경우

→ ‘|’, ‘::=‘ 와 같이 인용부호로 묶어 표현

• sub-pascal 시작부에 대한 EBNF 표기

구문 도표

• 구문 도표는 그 형태가 순서도와 유사

• 구문 도표는 EBNF 와 일대일 대응

→ 다시 정의될 대상은 box로, terminal 기호는 원이나 타원형으로 표시

→ 이들 사이는 지시선으로 연결

텍스트, 시계이(가) 표시된 사진

자동 생성된 설명• 예:

Parse Tree

• 원시 프로그램의 문법 검사 과정에서 내부적으로 생성되는 트리 형태의 자료구조

• 문장 표현이 BNF에 의해 작성될 수 있는지 여부를 나타냄

• 예: 식별자에 대한 BNF를 통해 다음 TEST1에 대한 파스 트리 작성

구문(Syntax): 언어의 신뢰성에 영향

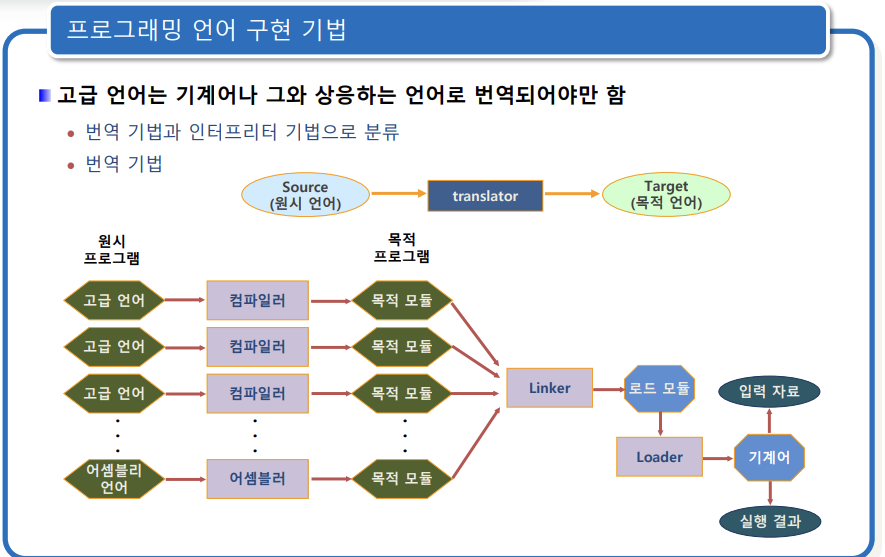
• 현수(dangling) else 문제

→ 중첩된 if 문에서 else는 어떤 if의 else인가?

• Algol 68

① if c1 then if c2 then S1 else S2 fi fi

② if c1 then if c2 then S1 fi else S2 fi



번역기법의 종류

① 컴파일러(compiler)

- 원시 언어: 고급 언어/목적 언어: 실제 기계 언어에 가까운 저급 언어

- 저급 언어: 준 기계어 형태 또는 어셈블리 언어

② 어셈블러(assembler)

- 원시 언어: 어셈블리 언어/목적 언어: 준 기계어 형태

③ 링케이지 에디터(linkage editor)

- 여러 개의 프로그램(재배치 형태 기계어)을 묶음 → 로드 모듈 생성

- 로드 모듈: 어느 정도 실행 가능한 하나의 기계어 프로그램

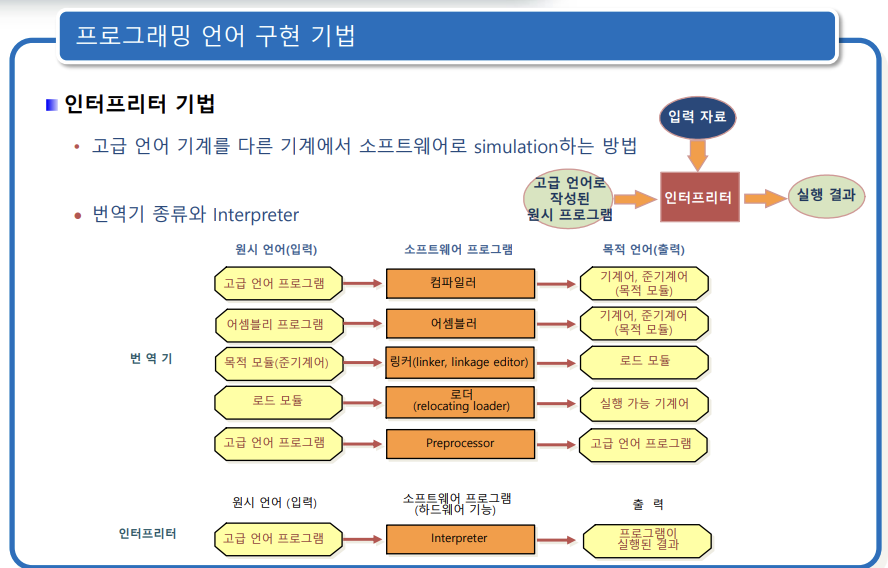
④ 로더(loader)

- 기계어 프로그램(로드 모듈)을 실제 실행 가능한 기계어로 번역해서 주 기억 장치에 적재

⑤ 프리프로세서(preprocessor)

- 원시 언어와 목적 언어가 모두 고급 언어인 번역기

- 고급 언어에 대한 언어를 확장하여 구현할 때 유용 (C++, concurrent C)



번역기

• 입력 프로그램과 동일한 의미의 목적 언어 프로그램 생성

인터프리터

• 직접 입력 프로그램을 실행하는 방법으로 처리

비교 분석

• 순수 번역 기법 (Assembly 등 저급 언어 가능)

• 순수 시뮬레이션 기법 (JCL, APL 등)

• 번역 효율적인 부분(반복 수행부와 수식 계산 등) 존재

• 원시 코드의 simulation이 효율적인 부분(I/O routine 등) 존재

• 순수 번역 기법이나 순수 시뮬레이션 기법은 실제로 거의 존재치 않음

텍스트, 장치이(가) 표시된 사진

자동 생성된 설명

번역 기법의 장단점

• 장점: 실행 시간 효율성 제공 (한번 decoding으로 반복 실행)

• 단점: 번역된 프로그램이 큰 기억 장치 요구 (I/O routine 등)

인터프리터 기법의 장단점

• 번역 기법과 장단점이 반대

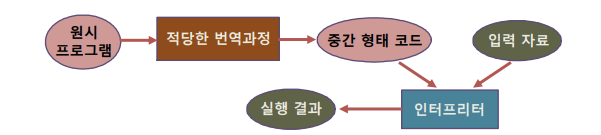
• 사용자 적응성(flexibility) 제공

하이브리드(hybrid) 기법

• 프로그램을 실행시키기 쉬운 형태로 번역한 후, 번역된 프로그램을 시뮬레이션으로 실행

• 현재 대부분의 인터프리터 언어가 이 방법을 따름

• 중간 형태 코드가 저급 언어이면 번역 기법으로 간주되기도 함



컴파일러 언어

• Fortran, Algol, PL/I, Pascal, Cobol, C, Ada

• 컴파일러 방법의 장점

Φ 기계어로 번역된 것을 하드웨어 인터프리터가 decoding하여 실행

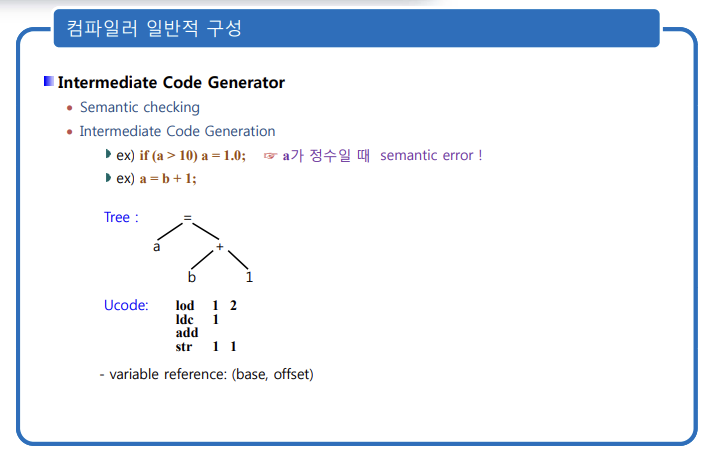
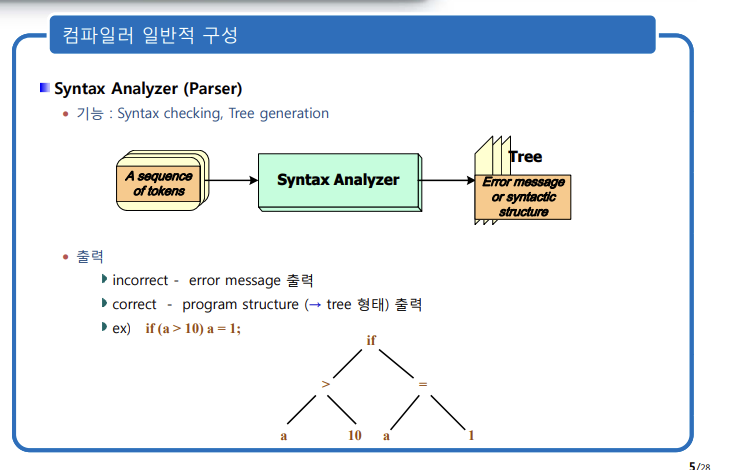
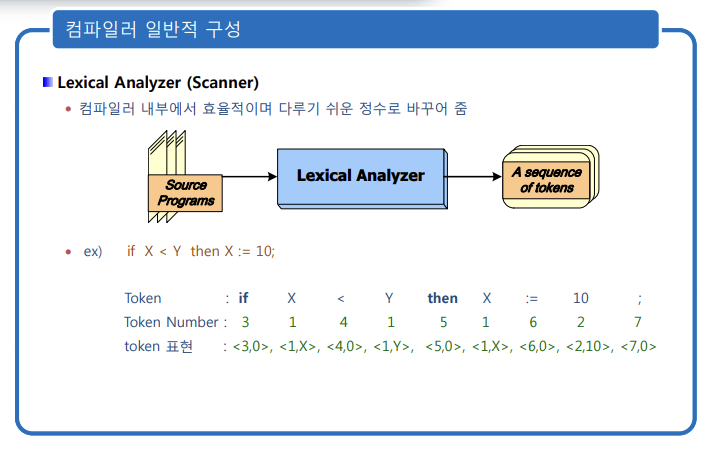
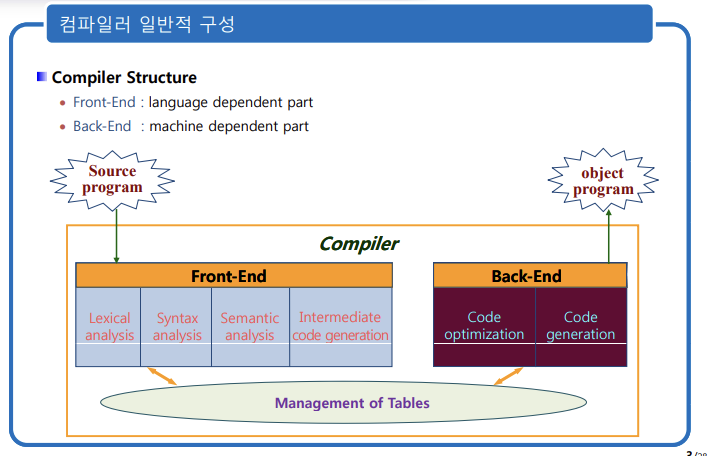
→ 빠른 프로그램 실행(효율성)

인터프리터 언어 • Lisp, SNOBOL4, APL, Prolog

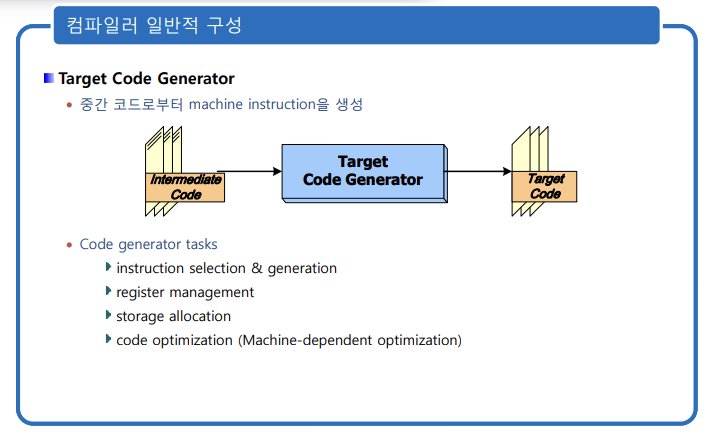
• 구현 방법 Φ 번역기가 중간언어를 생성 후, 이를 다시 소프트웨어 인터프리터로 실행 → hybrid 방법

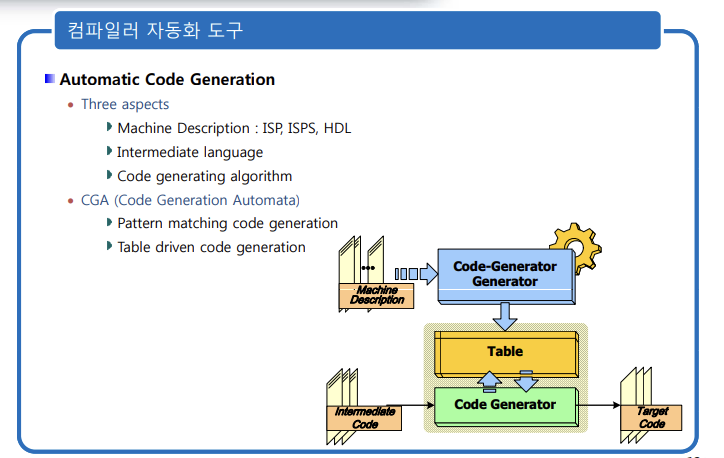
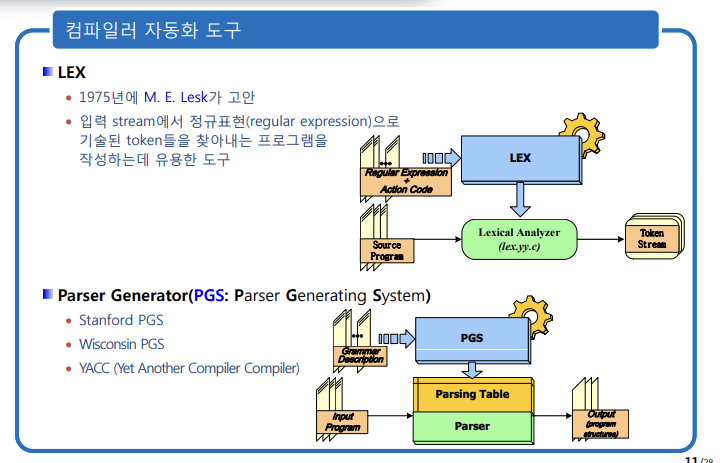
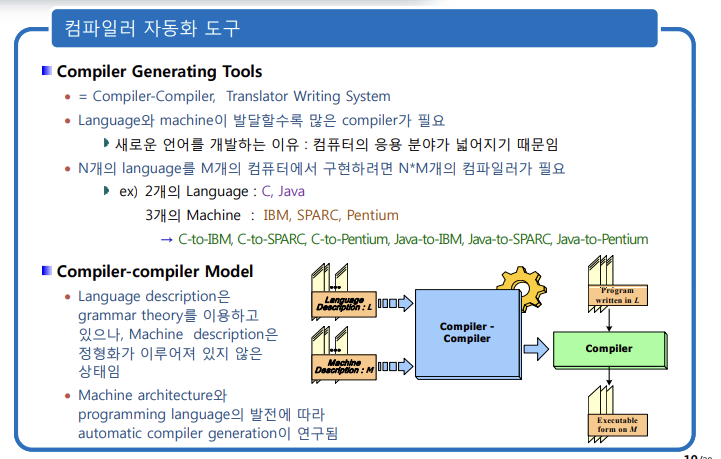
• 인터프리터 방법의 장점 Φ 컴파일러 방법보다 실행 시간이 비효율적이지만 사용자에게 flexibility를 제공

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

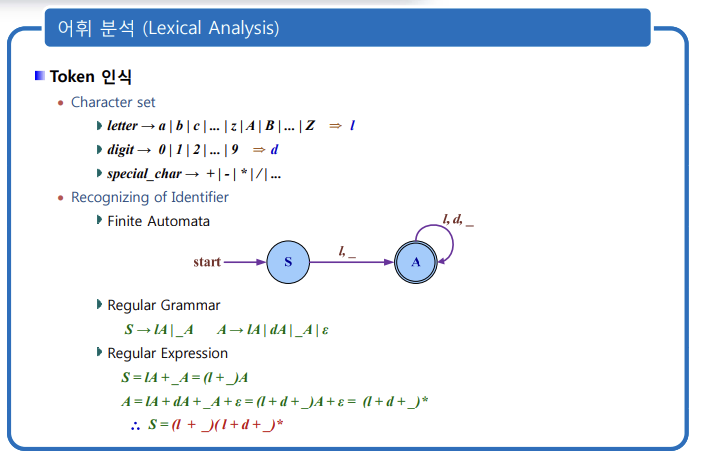
자동 생성된 설명

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

Management of table: backend front end 나눠진 기준

하드웨어의 따라 달라질 수 있는 기준 back end(하드웨어 중요)

frontend(하드웨어 성능 불필요 수학적 연산만 함)

Lexical analysis: 어휘 분석

Syntax analysis: 구문분석

Semantic analysis: 의미 분석

Intermediate code generation: 중간 코드 분석

Code optimization:

Code generation:

LEX: 어휘토큰을 자동화

PARSER Generator: 구문토큰 자동화 (PGS) 파일에 집어넣으면 문법이 알맞게...

Auto Code Generator:

-어휘분석

어휘 분석기: Scanner =Lexer 토큰은 문법적으로 의미있는 최소단위 정렬은 순서 쌍으로

Token Number는 Paser를 위해서 변수는 변수가 있는지 없는지가 중요

프로그래밍 언어에서 바인딩 시간이 끼치는 영향?

컴파일러 구성을 그리고 각 위치에 따른 간단한 설명?

프로그래밍 언어의 정의는 ??와 ??로 나뉜다

BNF가 주어지고 그것에 따른 구문 도표 작성

Ada언어에서 현수 else문제를 해결한 방법을 Ada언어의 사용으로 보여주고 어떤식으로 해결 했는지 서술