**컴파일러**

**입문**

**제 1 장 컴파일러 개론**

차

**1.1 프로그래밍 언어**

목

**1.2 번역기와 컴파일러**

**1.3 컴파일러의 일반적인 구조**

**1.4 컴파일러 자동화 도구**

Introduction to Compiler Design Theory Page 2

**언어**

 좋은 프로그래밍 언어의 **요건**

 언어의 개념이 명확

 문법적인 구조(**syntax**)

 의미(**semantics**)

**프로그래밍**

 프로그래머의 생각을 자연스럽게 표현

 호환성(이식성), 신뢰성, 모듈화, 효율성

 언어의

**확장성**이 우수

 좋은 프로그래밍 환경

Introduction to Compiler Design Theory Page 3

 **COBOL**

 **CO**mmon **B**usiness **O**riented **L**anguage  1960년대 초, 코다실(CODASYL)에서 발표  주로 업무용으로 사용  1968 ~ 1974년에 ANSI COBOL을 제정

 **FORTRAN**

 **FOR**mula **TRAN**slation  1960년대 초, J.Backus를 중심으로 개발  과학 계산용 언어  1977년 FORTRAN77이 발표

Introduction to Compiler Design Theory Page 4

 **ALGOL**

 **ALGO**rithmic **L**anguage  ALGOL68 - 1968년 IFIP WG2.1에서 발표  구문구조를 형식 문법으로 표현  언어의 구조와 의미가 명료  제어 구문 구조가 우수  후에 개발된 많은 프로그래밍 언어에 큰 영향

 **Pascal**

 1970년대 초, N.Wirth가 고안  프로그래밍 언어론적인 관점에서 설계  전산학적인 응용에 적합 - 자료구조, 알고리즘

Introduction to Compiler Design Theory Page 5

 **Ada**

 1980년 에서 발표  reliability, simplicity, modularity, efficiency  package, generic features, multi-tasking  real-time application에 적합

 **C++**

 1983년 B.Stroustrup  **OOPL** : class, , polymorphism

Introduction to Compiler Design Theory Page 6

 **Java**

 James Gosling, Sun MicroSystems  OOPL, Exception,  Internet & Distributed Environment  Applet, Application

 **C#** John Gough, Microsoft

 Framework, OOPL(Object-Oriented Programming Language)  JIT (**J**ust **I**n-**T**ime Compilation)  MSIL (**M**icro**s**oft **I**ntermediate **L**anguage)

CLR (**C**ommon **L**anguage **R**untime)  Expressiveness and Simplicity

Introduction to Compiler Design Theory Page 7

**컴파일러**

 Compiler

“A compiler is a computer program which translates programs

written in a particular programming language into executable code for a specific target computer.”

ex) **C compiler on SPARC**

 C program을 입력으로 받아 에서 수행 가능한 코드를

출력한다.

Source **Compiler** Object Programs

Programs

(Assembly Language, Machine Language)

**번역기와**

Introduction to Compiler Design Theory Page 8

 **Compiler Structure**

Source Programs

**Front-End**

**IC**

**Back-End**

Object Programs

 **Front-End** : dependent part  **Back-End** : dependent part

Introduction to Compiler Design Theory Page 9

**Cross-Compiler**

“A cross-compiler is a program which is to run on machine A

and produce target code for another machine B.”

**Compiler on A machine** Source Programs  Execution : **down-loading** or interpretation

Target Code for machine B

Introduction to Compiler Design Theory Page 10

**Interpreter**

“An interpreter transforms a program directly into a sequence of machine actions and produces the results of the program.”

data **Interpreter** Source Programs

C:\...

Result

 **Compiler :** System  **Interpreter : Developing** System or System

Introduction to Compiler Design Theory Page 11

 **Preprocessor**

 for the language

Source **Preprocessor** Programs

 Macro substitution  Conditional compilation  Inclusion of files

Extended Source Programs

**Translator**

**Translator**

Target Programs

Target Programs

Target Programs

Introduction to Compiler Design Theory Page 12

**일반적인**

Source Programs

Object Programs

**컴파일러**

**구조**

**구조**

**Lexical Analyzer**

Token

Optimized Code **Target Code Generator**

Introduction to Compiler Design Theory Page 13

**Syntax Analyzer**

Tree **Intermediate Code Generator**

Intermediate Code

**Code Optimizer**

**1. Lexical Analyzer( )**

 컴파일러 내부에서 효율적이며 다루기 쉬운 정수로 바꾸어 줌.

Source Programs

**Lexical Analyzer** A sequence of tokens

ex) **if ( a > 10 )** ...

Token : **if ( a > 10 )** ...

Token Number : 32 7 4 25 5 8

Introduction to Compiler Design Theory Page 14

**2. Syntax Analyzer( )**

 **기능**: **Syntax checking**, **Tree generation**.

 **출력**: incorrect - error message 출력

correct - program structure (=> tree 형태) 출력

ex) if (a > 10) a = 1;

if

> =

a 10 a 1

**Tree** A sequence of tokens **Syntax Analyzer** Error message or syntactic structure

Introduction to Compiler Design Theory Page 15

**3. Intermediate Code Generator**

 Intermediate Code Generation

ex) if (a > 10) a = 1.0; ☞ a가 정수일 때 semantic error !

ex) a = b + 1;

Tree : =

a +

b 1

Ucode:lod 1 2

ldc 1 add str 1 1

- variable reference: (base, offset)

Introduction to Compiler Design Theory Page 16

**4. Code Optimizer**

 Optional phase  비효율적인 code를 구분해 내서 더 효율적인 code로 바꾸어 준다.  **Meaning of optimization**

 major part : improve running time  minor part : reduce code size

ex) LDC R1, 1

LDC R1, 1 (x)

 **Criteria for optimization**

 preserve the program meanings  speed up on average  be worth the effort

Introduction to Compiler Design Theory Page 17

 **Local optimization**

 local inspection을 통하여 inefficient한 code들을 구분해 내서

좀 더 efficient한 code들로 바꾸는 방법.

1. Constant folding 2. Eliminating redundant load, store instructions 3. Algebraic simplification 4. Strength reduction

 **Global optimization**

 flow analysis technique을 이용

1. Common subexpression

2. Moving loop invariants

3. Removing unreachable codes

Introduction to Compiler Design Theory Page 18

**5. Target Code Generator**

 중간 코드로부터 machine instruction을 생성한다.

**Target** Intermediate Code

**Code Generator** Target Code

 **Code generator tasks**

1. instruction selection & generation

2. register management

3. storage allocation

4. code optimization (Machine-dependent optimization)

Introduction to Compiler Design Theory Page 19

**6. Error Recovery**

Error recovery - error가 다른 문장에 영향을 미치지 않도록

수정하는 것 Error repair - error가 발생하면 복구해 주는 것

 **Error Handling**

 Error detection  Error recovery  Error reporting  Error repair

 **Error**

 Syntax Error  Semantic Error  Run-time Error

Introduction to Compiler Design Theory Page 20

**도구**

 Compiler Generating Tools

**컴파일러**

**자동화**

**자동화**

(= Compiler-Compiler, Translator Writing System)

 Language와 machine이 발달할 수록 많은 compiler가 필요.  새로운 언어를 개발하는 이유: 컴퓨터의 응용 분야가 넓어지므로.

 N개의 language를 M개의 컴퓨터에서 구현하려면 N\*M개의

컴파일러가 필요.

ex) 2개의 language : **C**, **Java**

3개의 Machine : **IBM**, **SPARC**, **Pentium**

C-to-IBM, C-to-SPARC, C-to-Pentium Java-to-IBM, Java-to-SPARC, Java-to-Pentium

Introduction to Compiler Design Theory Page 21

 Compiler-compiler **Model**

 Language description은 grammar theory를 이용하고 있으나, Machine description은 정형화가 이루어져 있지 않은 상태임.

 HDL : **H**ardware **D**escription **L**anguage

→ Computer Architecture를 design하는 데 사용.  Machine architecture와 programming language의 발전에 따라

automatic compiler generation이 연구됨.

**Program** Description Language : **L**

**written in *L***

**Compiler - Compiler**

**Compiler** Description Machine : **M**

**Executable form on *M***

Introduction to Compiler Design Theory Page 22

**1. Lexical Analyzer Generator**

 LEX : 1975년에 가 고안.

 입력 스트림에서 정규표현으로 기술된 토큰들을 찾아내는

프로그램을 작성하는데 유용한 도구.

**LEX** Regular Action Expression + Code

**Lexical Analyzer** Token Source Program

***(lex.yy.c)***

Stream

Introduction to Compiler Design Theory Page 23

**2. Parser Generator(PGS: P**arser **G**enerating **S**ystem**)**

**PGS**

Description Grammar **Parsing Table**

Input Program

Output **Parser** (program structures)

(1**) Stanford PGS**

 John Hennessy  파스칼 언어로 쓰여 있음 : 5000 lines  특징 : 구문 구조를 AST 형태로 얻음.  Output : Abstract Syntax Tree(AST)의 정보를 포함한 파싱 테이블을 출력.

Introduction to Compiler Design Theory Page 24

(2) Wisconsin PGS

 C.N. Fisher  파스칼 언어로 쓰여 있음.: 10000 lines  특징 : error recovery

(3) YACC(Yet Another Compiler Compiler)

 UNIX에서 수행.  C language로 쓰여 있음.

**LEX YACC** Source **lex.yy.c y.tab.c** Program

Regular Action Expression + Code

Grammar Action + Code Rule <Lexical Analysis> <Syntax Analysis>

Introduction to Compiler Design Theory Page 25

Result by Action Code

3. Automatic Code Generation

 Three aspects

1. Machine Description : ISP, ISPS, HDL 2. Intermediate language 3. Code generating algorithm  CGA

Pattern matching code generation Table driven code generation

Machine Description

Intermediate Code Target Code

Introduction to Compiler Design Theory Page 26

**Code-Generator Generator**

**Table**

**Code Generator**

4. Compiler Compiler System

(1) PQCC(Production Quality Compiler Compiler System)

 W.A. Wulf(Carnegie-Mellon University)  input으로 language description과 target machine description을 받아

PQC(Production Quality Compiler)와 table이 output됨.  중간 언어로 tree구조인 TCOL을 사용.  Pattern Matching Code Generation에 의해 code를 생성함.

(2) ACK(Amsterdam Compiler Kit)

 Vrije 대학의 Andrew S. Tanenbaum을 중심으로 개발된 Compiler의

Back-End 자동화 도구.

 UNCOL 개념에서 출발.

 EM이라는 Abstract Machine Code를 중간 언어로 사용.

 Portable Compiler를 만들기에 편리.

Introduction to Compiler Design Theory Page 27

Source Program

**Front-End**

TCOL

**PQC** Object Code

**PQC** Object Code

**PQC** Object Code

 **PQCC Model**

Introduction to Compiler Design Theory Page 28

Language Description + Machine Description

**PQCC**

**Table**

 **ACK Model**Source **Front-End EM**

**Back-End** Program ***FORTRAN***

***ALGOL PASCAL CADA***

**Interpreter**

Object Code

C:\...

Result

Introduction to Compiler Design Theory Page 29

***Intel 8080/8086/80386 Motorola 6800/6809/***

***68000/68020 Zilog Z80/Z8000 VAX SPARC***