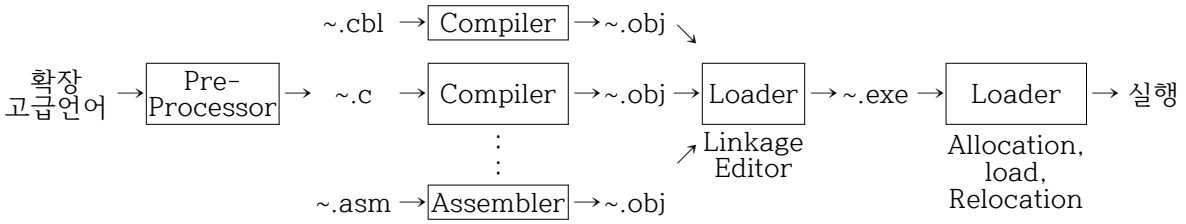
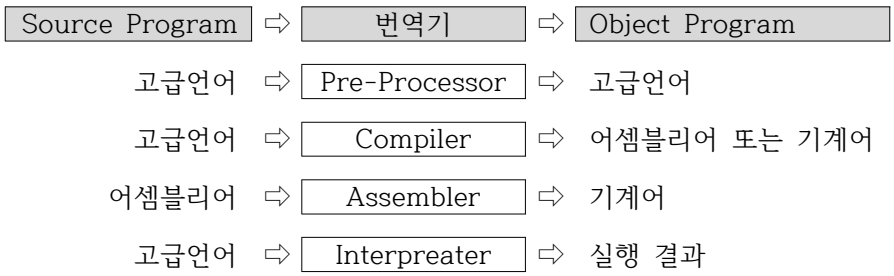
2 장 컴파일러의 개요

1. 번역기의 처리과정



[그림 1-1] 언어 번역기의 처리 과정

2. 컴파일러의 종류



[그림 1-2] 언어 번역기의 종류

(1) 전처리기 : 프로그래밍 언어에 유용한 기능들을 추가하여 언어를 확장시켜 주는 역할을 하는 것으로 확장 또는 추가된 기능들을 순수한 고급언어로 바꿔주는 번역기

* C언어가 제공하는 전처리기

표제파일(Header File) : File을 포함하여 언어를 확장

매크로 치환 : 자주 사용되는 상수값이나 식을 간단한 기호식 이름으로 대치하여 반복 사용하는 불편을 줄이고 프로그램의 의미와 관련된 기호를 부여함

조건번역 : 조건에 따라 실행부를 다르게 적용할 경우

```
#if SYSTEM == 1
        #include "unix.h"
#else SYSTEM == 2
        #include "dos.h"
#endif
```

(2) 컴파일러 : FORTRAN, COBOL, PASCAL, C 등의 고급언어로 작성된 프로그램을 어셈블리어나 기계어로 번역하는 번역기

(3) 어셈블러 : 기계어와 1:1로 대응되는 기호식 표현(Mnemonic)인 어셈블리어로 작성된 프로그램을 그에 대응하는 기계어로 번역해 주는 번역기

(4) 인터프리터 : 번역과 동시에 실행

(5) Cross-Compiler : 컴파일러가 실행되는 컴퓨터가 아닌 다른 컴퓨터의 실행코드를 생성

(6) Simulation : 다른 컴퓨터(상황) 등을 가상 Data로 실행하여 결과를 예측

3. 컴파일러와 인터프리터

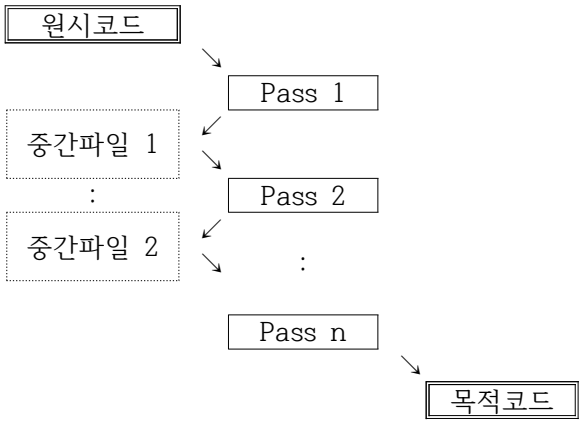
Compiler	구분	Interpreater
전체를 번역한 후 실행	처리방법	명령 단위로 번역 즉시 실행
소용량 (번역 단계별 실행)	주기억	대용량(원시코드, 번역기, 실행코드 동시 적재)
전체 오류를 한번에 수정	번역시간	오류 마다 처음부터 다시 번역
1회	번역회수	실행할 때 마다
빠르다 (실행코드 실행)	실행시간	느리다 (번역 후 실행)
숙련자	적 용	초보자

4. 컴파일러의 물리적 구조

- 주기억의 용량이 작은 경우에 이용된다.
- 전방참조를 해결하기 위하여 일반적으로 2 pass 이상의 단계를 거치게 된다.

  전방참조 : 소스코드 진행과정에서 앞으로 발생할 명칭을 미리 사용

  후방참조 : 소스코드의 앞부분에 이미 정의된 명칭을 사용



5. 컴파일러의 논리적 구조

(1) 어휘분석(Lexical Analysis)
- 원시코드를 기종마다 다른 고유의 문자열(Token)로 변환(이식성 향상)
  예약어(reserved word) : 명령어 등과 같이 언어 별로 규정된 단어
  리터럴(literal) : 문자나 숫자 상수(constant) 등을 literal pool에 저장
  연산자(operator) : 산술 및 논리 연산 등의 각종 기호
  식별자(identifier) : 변수명, 함수명, 프로시저 이름 등을 symbol table에 저장
  구분자(delimiter) : 빈칸, 괄호, 따옴표, 마침표 등
- 어휘분석은 scan 또는 scanning, 어휘분석기는 scanner라고도 한다.
- 최적화 단계를 제외한 컴파일 시간의 75% 소비
- 설계가 간단하다

(2) 구문분석(Syntax Analysis)
- Token을 입력하여 추상 구문 형태로 출력
- 추상구문 : Parse Tree에서 더 이상 필요없는 구두점 및 논터미널을 제거한 트리
- 구문분석을 parsing, 구문분석기를 parser라고도 한다.
- BNF/EBNF 문법 기반

(3) 의미분석(Semantic Analysis)
- 식별자의 선언 여부 검사
- 타입 검사
- 타입변환 연산자 삽입 (묵시적 타입 → 명시적 타입)

(4) 중간코드생성(Intermediate Code Generation)
- 부함수(sub routine) 형태의 코드로 변환
- Macro Processor의 입력과 같이 macro를 포함한 어셈블리어 형태

(5) 코드최적화(Code Opimization)
- 지역최적화와 전역최적화로 나누며,
- 지역최적화는 중복 load, store명령의 제거, 불필요한 코드의 삭제, 제어흐름의 최적화, 식(expression)의 대수적 간소화, 연산의 세기경감, 상수전파(constant folding/propagation), 복사전파(copy propagation), 공통부분식(common subexpression) 제거, 결합변형 등이 있다.
- 전역최적화는 코드이동(code motion), 귀납변수(induction variable) 최적화, 루프융합(loop fusion/jamming), 루프전개(loop unrolling) 등이 있다.

(6) 코드생성(Code Generation)
- 매크로 처리기와 어셈블러의 결합 형태로 처리
- 최적화된 중간코드를 기계어로 변환하는 마지막 단계
- 생성된 object code가 바로 실행될 수 없이 loader에 의하여 executable code로 변환되어야 한다.