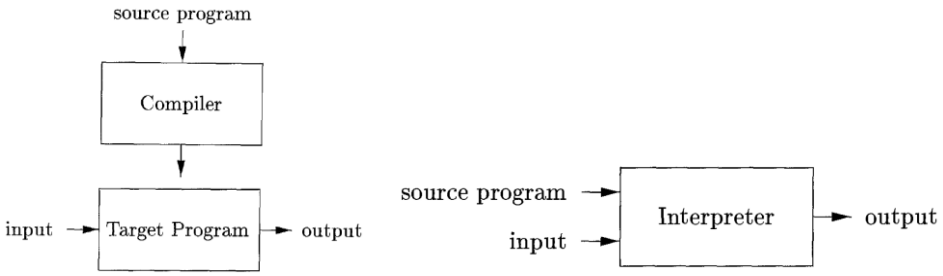
Compilers & Lab

Assignment #1 (Theory)

- **0 Score** for the corresponding assignment **for both the copy and source**.
- **Clearly indicate** the source if you get some from other places or if it is not your original idea or thoughts. **(Otherwise, the score will be cut by 10%)**

1. What is the difference between a compiler and an interpreter?



(좌: 컴파일러, 우: 인터프리터)

컴파일러는 원시 언어로 작성된 프로그램을 읽어서, 의미가 같은 목표 언어로 작성된 목표 프로그램을 만드는 프로그램인 반면, 인터프리터는 사용자가 제공한 입력에 대해서, 원시 언어로 기술된 연산을 직접 해석해서 실행하는 프로그램이다. 일반적으로 컴파일러의 목표 프로그램은 입력을 출력으로 사상하는 과정을 거쳐야 하는 인터프리터보다 훨씬 빠르고, 인터프리터는 문장 단위로 원시 프로그램을 실행시키기 때문에 컴파일러보다 상세한 오류 진단 메시지를 제공한다.

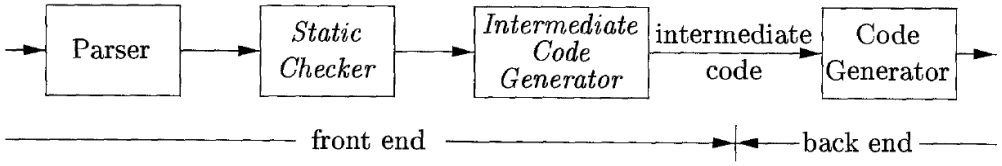
2. What are important qualities of compilers?

1. 정확성 (주어진 코드 의미를 훼손하지 않고 그대로 보존하는지)
2. 컴파일 속도
3. 컴파일 결과의 실행 속도
4. 컴파일 결과의 크기
5. 디버그 지원
6. 친숙한 Front-end (구문, 작성 검사)
7. 언어간 호출 지원 (인터페이스 호환성)
8. 정확하고 이해 가능한 최적화

3. Why are compilers commonly split into multiple passes?

컴파일 단위를 한번만 거쳐 컴파일하는 단일 패스 컴파일러와 달리, 컴파일 단위를 여러 번 거치는 다중패스로 구성하면 중간 코드 생성, 기계어 코드 최적화 등의 작업을 단위별로 더 쉽게 할 수 있고, 이러한 패스들은 독립적으로 작성되므로, 다른 언어의 컴파일, 다른 기계에서의 컴파일 과정 등에서 재활용하여 사용할 수 있는 장점이 있다. 이러한 이유 때문에 많은 컴파일러들이 다중패스 구조를 채택한다.

4. What are the typical responsibilities of the different parts of a modern compiler?



컴파일러의 분석 및 합성 모델에서 컴파일러는 크게 전단부(front-end), 후단부(back-end)의 2가지 부분으로 구성된다.

- 전단부(front-end)는 구문을 분석하는 파서, 정적 (타입) 검사기, 중간 코드 생성기로 이루어져 있으며 원시 프로그램을 분석한 뒤 중간 표현을 생성한다.
- 후단부(back-end)는 중간 코드 생성기로부터 생성된 중간 코드로부터 코드 생성기를 통해 목표 코드를 생성한다.

5. How are context-free grammars specified?

문맥-자유 문법은 다음 4개의 요소로 구성된다.

1. “토큰”이라 불리는 터미널(terminal) 기호의 집합. (터미널은 문법이 정의하는 언어의 기본적인 기호임)
2. “구문 변수”라고 불리는 논터미널(nonterminal)의 집합. 논 터미널들은 일반적으로 터미널 문자열들의 집합을 나타낸다.
3. “생성식”의 집합. 각 생성식은 논터미널과 터미널, 또는 논터미널들의 나열로 구성된다.
4. 시작 기호로 논터미널 중 하나를 지명한다.

BNF(Backus-Naur form)은 문맥 자유 문법을 표현하기 위해 사용되는 대표적인 예라고 할 수 있다.

6. What is “abstract” about an abstract syntax tree?

추상 구문 트리에서 "추상"의 의미는, 언어의 모든 문법을 하나하나 세세하게 반영하는 구체적인 구문 트리(Concrete syntax tree, Parse tree)와 달리 실제 구문에서 나타나는 모든 상세한 정보를 모두 나타내지는 않는다는 것을 의미한다.

7. What is intermediate representation and what is it for?

중간 표현이란 원시 프로그램을 목표 언어(또는 코드, 프로그램)으로 번역하는 과정에서 컴파일러가 내부적으로 사용하는 데이터 구조 또는 코드이다. 중간 표현은 컴파일러의 효율적인 번역을 위해 필요한, 최적화, 변환 등의 추가적인 후처리를 위해서 필요하며 크게 생산 용이성(원시 코드로부터 만들기 쉬워야 함)과, 번역 용이성(목표 코드, 프로그램으로 변환하기 쉬워야 함)의 두가지 특징을 가진다.

8. Why is optimization a separate activity?

일반적으로 컴파일러에서는 원본 소스 코드를 더 효율적으로 개선하고 나타내는 다양한 부분의 최적화 작업을 수행한다. 그런데 이런 최적화 작업은 원본 소스 언어의 많은 부분을 해석한 뒤에 진행하는 것이 더 효율적이고 수월하다. (가령 원본 소스 코드를 중간 언어로 나타내는 경우, 원본 소스 코드보다 더 작은 단위로, 더 기계어와 가까운 형태로 변환되기 때문에 이 단계에서 최적화를 적용하는 것이 더 수월하다.) 이러한 이유 때문에 최적화 과정은 보통 컴파일 후반 과정*에 별도의 절차로 적용되는 경우가 많다.

*. 예를 들어 후단부 패스에서 기계어로 중간 언어를 번역하는 과정에서는, 그 대상 머신에 대한 최적화 과정이 포함된다.

9. Is Java compiled or interpreted? What about Smalltalk? Ruby? PHP? Are you sure?

자바는 일반적으로 JVM 위에서 돌아가는 바이트코드로 컴파일 된 뒤, 실행 시간 JVM 런타임에서 바이트코드가 해당 머신의 기계어로 번역되어 실행되는 JIT(Just In Time) 컴파일 방법이 사용된다. 이 컴파일 방법은, 실행 시간에 바이트코드가 기계어로 변환되어서 실행된다는 점에서 인터프리터의 방법과 유사하다고 할 수 있지만, 기본적으로 바이트코드라는 변환물을 우선적으로 빌드해야 한다는 점에서는 컴파일러의 방법에 가깝다고 볼 수 있다. 한편 최근에 개발중인 자바의 JVM 중 하나인 GraalVM과 같은 가상머신은 AOT(Ahead Of Time) 컴파일 방식을 지원하는데 이 방식은 실행 시간에 머신에 종속되는 기계어 코드로 자바 코드를 컴파일한다. 이러한 두가지 방법이 공존하는 것을 보면, 자바는 전체적으로는 컴파일 언어이고, 부분적으로는 인터프리터를 사용하는 부분이 있다고 생각된다. Smalltalk 같은 경우에는 컴파일러와 인터프리터가 모두 존재한다고 알려져 있다. 또한 Ruby와 PHP는 실행 시간에 코드를 해석하는 인터프리터 구조를 사용하고 있다.

10. What are the key differences between modern compilers and compilers written in the 1970s?

컴퓨팅 성능이 지금과 같지 않았던 옛날 컴파일러들은, 대체로 느리고 현대 컴파일러에 비해 불편한 사용성을 가진 경우가 많았고, 지금과는 달리 정말 번역 기능에만 집중하여, 코드 최적화 기능을 많이 포함하지 않았다. 또한 대부분의 컴파일러가 크로스 플랫폼 작동을 지원하는 요즘과 달리, 1970년대의 컴파일러들은 특정한 운영체제의 종속되게끔 작성되는 경우가 많았고, 무료 컴파일러들을 얼마든지 구하는 것이 가능한 요즘과 달리 대체로 비싼 값(기계, 운영체제와 컴파일러를 묶어 파는 번들을 구매하거나, 단일 소프트웨어로)에 구매해서 사용해야 하는 경우가 많았다.

11. Why is it hard for compilers to generate good error messages?

사용자가 무엇을 작성하려고 했는지, 의도를 파악하는 것이 어렵기 때문이다. 가령 다음과 같은 자바 코드의 구문 오류를 생각해보면

```
System.out.println("Hello World");
// 정상적인 코드
```

```
System.out.println(Hello World);
// 구문  오류1
System.out.printllll(“Hello World”);
// 구문  오류2
```

구문 오류1과 같은 경우 System.out.println의 인자로 1개의 Object가 들어갈 수 있는데 구문을 해석하다 보면 Hello, World의 두개의 토큰이 존재하고 이들은 각각 아무런 변수도 아니므로 어디서 오류가 났는지에 대해서는 쉽게 파악을 할 수 있고, 아마 조금 더 똑똑한 컴파일러라면 스트링 리터럴 지시자 “”를 오류 메시지에 제안으로 표시할 수도 있을 것이다. 구문 오류2도 마찬가지로, System.out (PrintStream)이 가지고 있지 않은 함수인 printllll을 오류로 검출하고 유사성이 높은 print, println과 같은 함수를 오류 메시지에 같이 추천해줄 수도 있을 것이다. 그런데 이와 같은 좋은 오류 메시지는 추측에 기반한다. 실제로는 사용자가 Hello 변수를 출력하거나, World 변수를 출력하려고 했을 수도 있는 것이고, 이러한 모든 경우를 추측하는 것은 불가능에 가깝다. 컴파일 타임에 검출할 수 없는 논리 오류 같은 것을 생각하면 더욱 그렇다. 또한 언어의 명세나 구현 복잡성 등의 이유로도 좋은 오류 메시지를 출력하기 힘들 수도 있다.

12. What is “context-free” about a context-free grammar?

좌변에 1개 이상의 논터미널이 올 수 있는 문맥 의존 문법(Context-sensitive

grammar)과 무제한 문법(Unrestricted grammar)과는 달리 문맥 자유 문법의 생성 규칙 중 가장 주요한 특징은 좌변에는 항상 1개의 논터미널만 올 수 있다는 것(A[논터미널] → x[문자열])이다. 이는 하나의 논터미널만 고려하여 문자열을 생성하기 때문에 문맥에서 자유롭다는 의미를 가진다.

Reference

Compilers Principles Techniques and Tools - Chapter 1, 2, 3, 6

Wikipedia - Abstract syntax tree

Is Smalltalk a compiled or an interpreted language?

Wikipedia - Smalltalk

Wikipedia - Little Smalltalk

Why do C++ compilers have obscure error messages

INTRODUCTION TO COMPILERS

Wikipedia - Context-sensitive grammar

Wikipedia – Intermediate representation

<The End of the Assignment>