**13페이지**

01 СНАРТER

컴파일러 개론

1.1 프로그래밍 언어

1.2 번역기와 컴파일러

1.3 컴파일러의 일반적 구조

1.4 컴파일러 자동화 도구

**14페이지**

고급 언어로 쓰여진 프로그램이 컴퓨터에서 실행되기 위해서는 해당하는 언어 번역기(language translator)가 있어야 한다. 컴파일러(compiler)란 언어 번역기 중 하나로 고급 언어 프로그램을 입력으로 받아 의미적으로 동등하면서 직접 기계에서 실행될 수 있는 형태로 번역하는 작업을 한다.

컴파일러를 효과적으로 제작하기 위하여 기능적으로 독립적인 각 단계로 나누어 다중 패스(multi-pass) 컴파일러로 구현하고 있다. 또한 프로그래밍 언어가 다양해지고 컴퓨터 하드웨어 기술이 발달함에 따라 많은 컴파일러가 필요하게 되었으며 컴파일러 자체를 자동적으로 제작하려는 연구가 활발히 진행되었다.

이 장에서는 프로그래밍 언어의 개념, 컴파일러의 구조, 그리고 컴파일러 자동화 도구에 관하여 살펴보기로 하자.

**15페이지**

section 1.1 프로그래밍 언어

언어는 의사 전달의 수단으로 자연 발생적인 자연 언어와 필요에 의해 인위적으로 만든 인공 언어로 구분할 수 있다. 이 인공 언어 중의 하나가 컴퓨터 언어이며 인간과 컴퓨터 사이를 연결해 준다. 컴퓨터 언어 중 어떤 일을 컴퓨터로 처리하기 위해서 일련의 과정을 기술할 때 사용되는 언어가 프로그래밍 언어이다.

프로그래밍 언어는 사용 목적에 따라 특수 목적 언어(special purpose programming language) 와 일반 범용 언어(general purpose programming language)로 나눌수 있다. 특수 목적 언어는 컴퓨터의 특정한 응용에 적합하도록 고안된 언어이다. 모의 실험(simulation)용 언어인 CSMP, GPSS 등과 DBMS에서 제공하는 질의 언어 (query language) 등이 이에 속한다. 또한, 특정한 소프트웨어 시스템에서 제공하는 사용자 접속 언어(user interface language)도 특수 목적 언어로 분류할 수 있다. 이에 비해, 일반 범용 언어는 모든 컴퓨터 응용에 적합하도록 고안된 언어이다. 우리가 흔히 사용하고 있는 대부분의 프로그래밍 언어는 모두 여기에 속한다.

프로그래밍 언어는 또한 형태와 기능에 따라 저급 언어(low-level language)와 고급 언어(high-level language)로 나눌 수 있다. 저급 언어에는 기계어와 어셈블리어가 있는데 초창기에 사용된 언어는 기계어였다. 기계어는 0과 1의 조합으로 실행의 의미를 표현하는 언어인데 이 언어는 상당히 복잡하고 어렵다는 단점을 가지고 있다. 이러한 단점을 약간이나마 보완하기 위해 기호(mnemonic)를 사용하여 프로그래밍을 할 수 있는 언어가 개발되었는데 이것이 어셈블리어이다. 어셈블리어는, 예를 들어 10110101과 같은 기계어를 ADD라는 명령 기호로 대치함으로써 기계어의 단점을 완화시켜 주었다. 그러나, 저급 언어들은 모두 컴퓨터 구조에 관한 지식을 요구하며 프로그래머의 생각을 자연스럽게 표현할 수 있는 언어 구조를 갖추고 있지 못하다. 저급 언어는 자기 자신에 대한 고유의 컴퓨터에서만 실행될 수 있는 특징을 지니고 있다. 고급 언어는 특정한 컴퓨터 구조에 대한 지식이 없어도 프로그래머의 생각은 자연스럽게 표현할 수 있도록 저급 언어에서는 제공되지 않은 다양한 언어의 기능을 갖추고 있다. 고급 언어는 자연 언어와 비슷한 형태로 설계되었으며 언어 번역기만

**16페이지**

갖추면 모든 기종에서 수행될 수 있는 특징을 갖고 있다.

고급 언어의 장점은 저급 언어에 비해 배우기가 쉽고 프로그래머의 생산성(productivity)을 높일 수 있을 뿐 아니라 한 기종에서 다른 기종으로 프로그램을 쉽게 가져갈 수 있는 이식성(portability)이 우수하다는 점이다. 그리고 고급 언어는 저급 언어보다 디버깅이 용이하며 기계 독립적이기 때문에 특정한 컴퓨터의 구조를 모르고도 프로그래밍을 할 수 있다는 것이다. 이와 같은 이유로 요사이는 특수한 경우를 제외하고는 모두 고급 언어를 사용하여 프로그래밍을 한다.

프로그래밍 언어에 대한 이론과 그에 따른 컴파일러 이론이 발달됨에 따라 프로그래머의 생산성을 높이고 쉽게 프로그래밍을 할 수 있는 많은 고급 언어들이 고안되고 설치되었다. 일반 범용 프로그래밍 언어의 흐름을 살펴보면, 1960년대의 코볼, 포트란을 시작으로 70년대 파스칼과 C 언어, 그리고 1980년대의 에이다 등으로 요약될 수 있으며 90년대에는 객체지향 언어(object-oriented language)가 그 대를 이어 받았다. 이 이외에도 많은 고급 언어들이 컴퓨터 회사 수준에서 개발되고 사용되었다.

여기서 몇 가지 고급 언어의 특징을 간략히 살펴보기로 하자.

코볼(Cobol)은 COmmon Business Oriented Language의 약자로 1960년대 초에 코다실(CODASYL) 위원회에서 고안하여 발표한 언어로 그 이름이 의미하듯이 주로 업무용으로 사용된다. 그후 1968년과 1974년에 걸친 표준화 작업으로 표준 코볼 (ANSI Cobol)이 제정되어 요즘도 업무용으로는 거의 표준 언어처럼 전산화 작업에 쓰이고 있다. 그러나, 코볼은 많은 약점을 갖고 있기 때문에 다른 대체 언어를 사용하는 것이 바람직하다.

포트란(Fortran)은 FORmula TRANslation의 약자로 1950년대 말부터 1960년대 초에 걸쳐 J. Backus를 중심으로 개발한 과학 계산용 언어이다. 수학적인 식을 직접 컴퓨터 프로그램에 표현할 수 있도록 설계된 언어이며, 표준화 과정을 거쳐 1977년 Fortran77이 발표되어 현재까지 널리 사용되고 있는 언어 중에 하나이다.

특히, 요사이 병렬 언어인 HPF(High Performance Fortran)가 발표되어 초고속 컴퓨팅에 새롭게 주목을 받는 언어가 되었다. 알골(Algol)은 ALGOrithmic Language의 약자로 1960년에 발표한 Algol60과 1968년에 수정 보완하여 IFIP의 WG2.1에서 발표한 Algol68이 있다. 알골은 수치 계산용 언어로 구문 구조를 형식 문법(formal grammar)으로 표현한 최초의 언어이며 언어의 구조와 의미가 명료하고 특히 제어 구문 구조가 우수하여 그 이후에 개발된 많은 프로그래밍 언어에 큰 영향을 주었다. 파스칼(Pascal)은 알골을 기반으로 1970년대. 초에 N. Wirth가 고안한 언어로서

**17페이지**

프로그래밍에 대한 근본적인 개념을 명확하고 자연스럽게 또한 체계적인 원리로 가르칠 수 있도록 프로그래밍 언어론적인 관점에서 설계된 언어이다. 특히 다양한 자료형을 제공하여 자료 구조, 알고리즘 설계 등 전산학적인 응용에 적합한 언어이다.

C 언어는 B 언어를 기반으로 D. M. Ritchie가 1974년경에 고안하여 설치한 언어로 시스템 프로그래밍을 위한 효율적인 언어로 개발되었다. 그후 C 언어의 기본 운영체제인 UNIX가 널리 확산됨에 따라 C 언어는 모든 응용에 사용하게 되었다.

1988년경에 이르러 표준 C(ANSI C)가 제정되어 현재 가장 많이 사용하는 언어가 되었다. 더욱이 C++ 언어가 C 언어를 기본 언어로 사용하고 있고 객체 지향 프로그래밍이 각광을 받기 시작하여 C 언어의 확고한 위치는 앞으로도 지속될 것이다.

에이다(Ada)는 1980년에 미 국방성(DoD)에서 발표한 언어로 실시간 응용(real-time applications)에 적합하도록 설계된 언어이다. 이 언어는 J. Ichbiah가 이끄는 그룹에서 제안한 언어를 최초의 프로그래머로 일컬어지는 August Ada의 이름을 따서 명명하였다. 에이다 언어의 개발 목적은 신뢰성(reliability), 단순성(simplicity), 모듈화(modularity), 효율성(efficiency) 등으로 요약될 수 있으며 에이다가 갖는 언어의 특징으로는 패키지(package), 포괄 구조(generic features), 분리 컴파일, 다중 처리(multi-tasking) 등을 들 수 있다. C++ 언어는 C 언어에 클래스 개념을 추가하여 객체 지향 프로그래밍이 가능하도록 1983년경에 B. Stroustrup이 개발한 언어이다. 객체 지향 프로그래밍 언어의 특징인 클래스(class), 상속성(inheritance), 다형성(polymorphism) 등을 구비하고 있지만 순차 언어인 C 언어에 기반을 두고 있어 진정한 의미의 객체 지향 프로그래밍 언어라고는 볼 수 없다. 그러나, 객체지향 개념이 정립되어감에 따라 C 언어와 자리바꿈을 하고 있는 중이다.

자바(Java)는 인터넷 및 분산 환경 시스템에서 효과적으로 응용 프로그램을 작성할 수 있도록 설계된 언어로써 객체지향 패러다임의 특성과 다양한 개발 환경을 지원하고 있다. 또한, 자바는 언어적인 측면에서 예외 처리와 스레드(thread)를 지원한다. 이 언어는 1996년 여름에 썬 마이크로시스템즈사의 제임스 고슬링(James Gosling)에 의해 발표되었으며 운영 제제 및 하드웨어 플랫폼에 독립적인 차세대 언어로써 각광을 받고 있다.

C#(C sharp)는 마이크로소프트사에서 2000년 중반에 발표한 범용 객체지향 언어로 .NET Framework에서 실행될 수 있는 응용프로그램을 효과적으로 작성한 수 이도로 설계되었다. 존 구우(John GOugh)를 중심으로 개발된 언어로 자바와 비슷한 개념과 언어 기능을 갖추고 있으나 실행은 JIT(Just In-Time Compilation) 방법을

**18페이지**

사용하여 자바의 단점을 보완하려 했다. 자바가 갖춘 대부분의 기능을 .NET 환경에서 효율적으로 구현하려는 의도로 간주할 수 있다. C# 언어에서 사용한 중간 언어는 MSIL(MicroSoft Intermediate Language)이고 환경은 CLR(Common Language Runtime)이다.

인터넷 프로그램이란 웹 브라우저에서 실행될 수 있는 프로그램을 의미하며 인터넷 프로그램을 작성하는 일을 인터넷 프로그래밍(internet programming)이라 정의할 수 있다. 이와 같은 언어는 크게 서버사이드 언어와 클라이언트사이드 언어로 구분 할 수 있으며, 전자에 속하는 언어로는 PERL, PHP, ASP, JSP 등이 있고 후자에 속하는 언어로는 HTML(Hyper Text Markup Language), DHTML, 자바스크립트, 자바 애플릿 등이 있다.

컴퓨터의 응용 범위가 넓어짐에 따라 수많은 프로그래밍 언어들이 창안되고 끊임없이 수정되고 개선되지만 좋은 프로그래밍 언어의 요건은 다음과 같이 요약될 수 있다.

① 언어의 개념이 명료해야한다. 문법적인 구조(Syntax)와 그에 따른 의미(semantic)가 일관성이 있으며 단순해야 한다.

② 프로그래머의 생각을 자연스럽게 표현할 수 있어야 한다.

③ 프로그램의 호환성, 신뢰성, 모듈화, 효율성 등이 좋아야 한다.

④ 언어의 확장성이 우수해야 한다.

⑤ 좋은 프로그래밍 환경을 갖고 있어야 한다.

고급 언어로 작성된 프로그램이 컴퓨터에서 실행되기 위해서는 컴퓨터가 직접 이해할 수 있는 언어로 바꾸어 주어야 한다. 이와 같은 일을 하는 프로그램을 컴파일러라 부르는데, 고급 언어로 작성된 프로그램을 실행하는 데 있어 그와 의미적으로 동등하며 실행될 수 있는 형태의 프로그램으로 바꾸어 실행한다. 다음절에서는 이와 같은 언어 번역기에 대해서 살펴보기로 하자.

**19페이지**

1.2 번역기와 컴파일러

번역기(Hanslator)란 한 프로그래밍 언어로 작성된 프로그램을 입력으로 받아 그와 동등한 의미를 갖는 다른 프로그래밍 언어로 된 프로그램을 출력하여 주는 시스템 프로그램을 말한다. 이때 입력되는 프로그램을 소스 프로그램이라 하고 이 프로그램을 기술한 언어를 소스 언어(source language)라 한다. 그리고 출력되는 프로그램을 목적 프로그램이라 하고 이 프로그램을 기술한 언어를 목적 언어(object language EE target language)라 한다.

만일 소스 언어가 파스칼이나 C언어와 같은 고급 언어이고 목적 언어가 .어셈블리어나 기계어일 경우, 이 번역기를 컴파일러라 한다. 소스 프로그램이 컴파일러를 통해 수행되는 과정은 [그림 1.1]과 같다. 소스 프로그램이 일단 실행 가능한 프로그램으로 바뀌면 소스 프로그램을 수정하지 않는 한 필요할 때마다 반복 실행할 수 있다.

그림 1.1 소스 프로그램이 수행되는 과정

로더(loader)의 출력인 실행 가능한 프로그램(executable program)을 일반적으로 로드 모듈(load module)이라 부르며 주 기억 장치의 사용자 영역에 적재되어 운영 체제와 실행 환경(run-time environment)의 도움을 받아 실행되어 결과를 출력한다.

크로스 컴파일러(cross-compiler)란 소스 프로그램을 컴파일러가 실행되고 있는 기계에 대한 기계어로 번역하는 것이 아니라 다른 기종에 대한 기계어로 번역하는 컴파일러를 말한다. [그림 1.2]는 A 컴퓨터에서 작동하는 컴파일러가 B 컴퓨터를 위한 실행 코드를 생성하는 경우를 보여준다.

**20페이지**

그림 1.2 크로스 컴파일러

예를 들어, 썬사의 워크스테이션(SPARC 프로세서)에서 수행되는 C 언어 컴파일러가 펜티엄 프로세서를 위한 목적 코드를 생성하는 경우이다. 크로스 컴파일러의 출력 프로그램을 실행하기 위해서는 그 기종으로 프로그램을 가져가서 실행시키거나 또는 그 기종에 대한 코드 인터프리터(또는 에뮬레이터)가 있어야 한다. 주로 새로운 기종에 필요한 컴파일러를 설치할 때 사용하는 기술이다.

이밖에 다른 번역기를 살펴보면 어셈블러(assembler), 인터프리터(interpreter), 그리고 프리프로세서(preprocessor) 등을 들 수 있다. 어셈블러는 어셈블리 언어로 쓰여진 프로그램을 입력으로 받아 기계어 프로그램으로 바꾸어 주는 번역기이다.

어떤 번역기들은 소스 언어를 특수한 형태의 중간 언어로 변환한다. 인터프리터는 이러한 중간 언어를 입력으로 받아 목적 언어로 변환하지 않고 직접 실행해서 그 결과를 출력하여 주는 프로그램이다.

그림 1,3 인터프리터의 기능

고급 언어로 쓰여진 프로그램의 의미를 실행하는데 있어서 컴파일러는 그와 동등한 의미를 갖는 실행 프로그램으로 바꾸어 그 프로그램을 실행함으로써 결과를 얻고 인터프리터는 소스 프로그램의 의미를 직접 실행하여 결과를 얻는다. 소스 프로그램의 수정 없이 계속 반복 실행하는 운용 시스템에서는 컴파일러가 효율적이며, 개발 시스템이나 교육용 시스템에서는 인터프리터가 더 능률적이다.

프리프로세서(또는 전처리기)는 프로그래밍 언어에 유용한 기능들을 추가시킴으로써 언어를 확장시켜 주는 역할을 한다. 이 경우에 프로그래밍 언어를 프리프로세서의

**21페이지**

기본 언어(base language)라 부른다. 프리프로세서의 지시문(directive)이 모두 처리되어 확장된 프로그램은 기본 언어에 대한 언어 번역기를 불러 번역된다. [그림 1.4]는 프리프로세서 지시문이 포함된 소스 프로그램이 처리되는 과정을 나타낸 것이다.

그림 1.4 전처리기의 수행 과정

프리프로세서는 기본 언어에 다음과 같은 기능들을 추가시킨다. 첫 번째 기능은 매크로 체제이다. 유사한 소스 코드를 매크로로 정의하고 필요할 때마다 확장하여 프로그래머의 생산성을 증가시킬 수 있다.

두 번째 기능은 컴파일 시간에 필요한 컴파일 시간 라이브러리(compile-time library)들을 포함하는 것이다. 이렇게 포함된 라이브러리들은 소스 프로그램과 함께 하나의 프로그램으로 컴파일된다.

프리프로세서의 세 번째 기능으로는 조건부 컴파일(conditional compile)을 들 수 있다. 조건부 컴파일이란 조건에 따라 소스 프로그램의 일부분을 선택적으로 삽입 또는 삭제할 수 있는 기능을 의미한다. 그러므로 서로 조금씩 다른 기능을 갖는 프로그램들을 하나의 파일에 프로그래밍 할 수 있다.

이상에서 언급한 번역기들의 기능을 요약하면 [그림 1.5]와 같다.

그림 1.5 여러 가지 번역기들의 기능

**22페이지**

1.3 컴파일러의 일반적 구조

컴파일러는 고급 언어로 쓰여진 프로그램을 어떤 특정한 컴퓨터에서 직접 실행 가능한 형태의 프로그램으로 번역해 주는 컴퓨터 프로그램이다. (A compiler is a computer program which translates other programs written in a particular high-level programming language into executable code for a specific target computer.)

컴파일러의 구조는 크게 전단부(front-end)와 후단부(back-end)로 나눌 수 있다. 전단부는 소스 언어에 관계되는 부분으로 소스 프로그램을 분석하고 중간 코드를 생성하는 부분이다. 이에 비해 후단부는 소스 언어보다는 목적 기계(target machine)에 의존적이며 전단부에서 생성한 중간 코드를 특정 기계를 위한 목적 코드로 번역하는 부분이다.

그림 1.6 컴파일러의 개략적 구조

전단부는 각 언어마다 하나씩 필요하며 후단부는 목적 기계 당 하나씩 필요하다. 전단부가 문법 이론(grammar theory)에 의해 잘 정립된 반면, 후단부는 아직도 경험적인 방법을 통하여 구현하며 계속적인 연구가 진행되고 있는 부분이다. 컴파일러를 기능별로 세분하면 [그림 1.7]과 같으며 전단부는 소스 프로그램을 읽어 들이는 어휘 분석 단계로부터 중간 코드를 생산하는 중간 코드 생성 단계까지이다. 따라서 전단부의 출력은 중간 코드가 된다. 후단부는 그 이하를 말하며 중진

코드에 대한 목적 프로그램이 생성된다.

**23페이지**

그림 1.7 컴파일러의 일반적 구조

컴파일 과정의 첫 번째 단계인 어휘 분석(lexical analysis)은 어휘 분석기(또는 스캐너(scanner))에 의해 이루어지며 이는 소스 프로그램을 읽어 들여 일련의 토큰(token)을 생성하는 일을 한다.

그림 1.8 어휘 분석기의 기능

토큰이란 문법적으로 의미를 갖는 최소의 단위로 프로그램은 토큰의 열로서 구성되어 있다고 생각할 수 있다. 토큰의 종류는 크게 특수 형태와 일반 형태로 분류할 수 있다. 특수 형태는 언어를 정의할 때 언어 설계자가 결정하는 지정어(keyword), 연산자 기호, 구분자(delimiter) 등이며 일반 형태는 프로그래머가 프로그래밍할 때 사용한 명칭과 상수들이다.

예를 들어, 다음과 같은 문장을 읽어 들였다고 하자.

a = b + 10;

어휘 분석 단계에서는 a, =, b, +, 10, 그리고 ;의 여섯 개의 토큰으로 위의 문장을 분리한다. 여기서, =, +, ;은 특수 형태의 토큰이며 a, b, 10은 일반 형태의 토큰에 해당된다. 어휘 분석 단계의 출력은 이러한 일련의 토큰들이다. 각 토큰들은 효율적인 처리를 위해서 고유의 내부 번호를 정수 형태로 갖는데 이것을 토큰 번호(token number)라 부른다. 생성된 토큰들은 다음 과정인 구문 분석 단계의 입력이

**24페이지**

된다. 프로그래머가 프로그램의 설명을 위해 쓴 주석(comment)은 이 과정에서 모두 처리된다. 어휘 분석 단계에 대해서는 이 책의 4장에서 자세히 다루기로 한다.

구문 분석기(syntax analyzer)는 파서(parser)라고도 하는데, 어휘 분석 단계의 출력인 토큰들을 받아 소스 프로그램에 대한 에러를 체크하고(error checking) 올바른 문장에 대해서는 구문 구조(syntactic structure)를 만든다.

그림 1.9 구문 분석기의 기능

즉, 소스 프로그램에 에러가 있으면 그에 해당하는 에러 메시지를 출력하고 그렇지 않으면 프로그램에 대한 구문 구조를 (트리/형태로 만들어 출력한다. 프로그램의 구문 구조를 트리 형태로 표현하는 방법에는 파스 트리(parse tree)와 추상 구문 트리 (abstract syntax tree)가 있으나 요즘 대부분의 컴파일러에서는 추상 구문 트리를 사용한다. 이와 같은 내용은 6.2절에 자세히 설명되어 있다.

예를 들어, 위 배정문에 대한 추상 구문 트리는 [그림 1.10]과 같다.

그림 1.10 배정문 a = b + 10;의 추상 구문 트리의 예

컴파일 과정의 세 번째 단계인 중간 코드 생성(intermediate code generation) 과정에서는 파서의 출력인 추상 구문 트리를 입력으로 받아 의미 검사(semanme checking)를 행하고 그에 해당하는 충간 코드를 생성한다.

그림 1.11 중간 코드 생성기의 기능

**25페이지**

컴파일러에 따라 독립된 의미 분석 단계를 가질 수 있다. 왜냐하면 컴파일러는 언어의 특성과 목적 기계, 그리고 컴파일러를 구현하는 사람(compicr impiementor)에 따라 다른 구조를 가질 수 있기 때문이다.

의미 분석 단계에서 하는 중요한 일 중에 하나는 형 검사(type checking)이다. 형 검사란 각 연산자(operator)가 소스 언어의 정의에 맞는 피연산자(operand)를 가지는 가를 검사하는 것이다. 예를 들어, 대부분의 프로그래밍 언어에서는 실수가 배열의 첨자로 사용되었을 때 에러로 간주한다. 또한, 어떤 언어에서는 실수와 정수의 혼합 연산을 허용하는데 이때는 연산을 수행하기 전에 정수를 실수로 바꾸어 주는 작업이 필요하다. 이와 같은 기능을 형 변환(type conversion)이라 부르며 모두 의미 분석 단계에서 처리한다.

중간 코드의 생성은 구문 분석 단계에서 만들어진 구문 구조를 이용하여, 코드를 생성한다. 구문 트리를 운행하면서 각 구문에 해당하는 중간 코드를 생성한다. 예를 들어, [그림 1.10]의 배정문은 다음과 같은 중간 코드로 변환된다.

즉, b의 값과 상수 10을 스택에 적재한 다음 덧셈을 해서 a에 저장한다. 여기서 사용한 중간 언어는 U-코드로 9.5.3 절을 참고하기 바란다.

중간 코드 생성 단계에서 생성된 중간 코드들은 다음 과정인 코드 최적화(code optimization) 단계의 입력으로 사용될 수 있다. 최적화 과정은 선택적인 단계(optional phase)로 생략되는. 경우도 있다. 그 기능은 같은 의미를 유지하면서 코드를 보다 더 효율적으로 만들어 코드 실행시 기억 공간이나 실행 시간을 절약하는데 있다. [그림 1.12]는 코드 최적화기의 기능을 보여준다.

그림 1.12 코드 최적화기의 기능

**26 페이지**

최적화는 최적화되는 관점에 따라 지역 최적화(local optimization 또는 peephole optimization)와 전역 최적화(global optimization)로 구분할 수 있다. 지역 최적화는 기본 블록(basic block)내에서 행해지며 부분적인 관점에서 일련의 비효율적인 코드들을 구분해 내고 좀더 효율적인 코드로 개선하는 방법이다.

지역 최적화 방법에 의해서 주로 (1) 컴파일 시간 상수 연산(constant folding), (2) 중복된 load, store 명령문 제거, (3) 식(expression)의 대수학적 간소화(algebraic simplification), (4) 연산 강도 경감(strength reduction), (5) 불필요한 일련의 코드 블록(null sequence) 삭제 등의 효과를 얻을 수 있다. 구체적인 내용은 11장을 참고하기 바란다. 실제로 지역 최적화기에 의해 상당한 수준의 코드 효율화를 취할 수 있다. 전역 최적화는 흐름 분석 기술(flow analysis technique)을 이용하여 기본 블록들 사이에 최적화를 행하는 것으로 여기에서 행할 수 있는 방법은 공통 부분식(common subexpression)의 축약, 루프 내에서 값이 변하지 않는 코드(loop invariant code)를 루프 밖으로 이동(code motion), 그리고 도달될 수 없는 코드(unreachable code)의 제거 등이 있다.

또한, 최적화는 그 위치에 따라 pfecode optimization과 postcode optimization으로 나눌 수 있다. Precode optimization은 중간 코드를 이용하여 최적화를 수행하며 그 위치는 목적 코드 생성 전에 행해진다. 그에 비해 postcode optimization은 목적 코드 생성 후에 목적 코드를 최적화하는 방법으로 기계 의존적인 최적화 방법이다.

그림 1.13 최적화기의 위치

목적 코드 생성기(target code generator)는 중간 코드를 입력으로 받아 그와 의미적으로 동등한 목적 기계(target machine)에 대한 코드를 생성하는 일을 한다.

그림 1.14 목적 코드 생성기의 기능

**27페이지**

목적 코드 생성기가 목적 코드를 생성하기 위하여 행하는 일을 크게 네 가지도 나눌 수 있다.

① 목적 코드 선택 및 생성

② 레지스터의 운영

③ 기억 장소 할당

④ 기계 의존적인 코드 최적화

각 변수들에 대한 기억 장소를 할당해야 하며 중간 코드의 의미와 일치하는 기계 명령어들을 효과적으로 선택하는 코드 생성 알고리즘을 사용해야 한다. 대부분의 컴퓨터들이 빠른 시간 내에 계산을 할 수 있는 소수의 고속 레지스터들을 가지고 있다. 그러므로 우수한 목적 코드 생성기는 이러한 레지스터들을 가능한 한 효율적으로 사용할 수 있어야 한다. 효율적인 레지스터의 사용은 임시 변수(temporary variable)의 사용을 줄일 수 있을 뿐만 아니라 전체적인 실행 속도를 빠르게 할 수 있다. 레지스터 할당(register allocation) 문제는 효율적인 코드 생성을 위해서 매우 중요한 문제이며 많은 알고리즘이 개발되어 있다.

목적 코드 생성기에서 행하는 기계 의존적인 최적화는 연속적인 명령어들을 의미적으로 동등한 하나의 명령어 또는 처리 속도가 빠른 명령어로 대체하여 기계어 코드의 성능을 향상시키는 방법이다. 일반적으로 코드 생성기에 의해 생성된 코드들은 잘 훈련된 어셈블리 프로그래머들이 손으로 작성한 코드들보다 훨씬 효율적이어야 한다.

컴파일러는 소스 프로그램에 나타난 모든 자료에 대한 정보들을 가지고 있어야 한다. 예를 들어, 어떤 변수가 정수를 나타내는지 실수를 나타내는지, 배열의 크기가 얼마나 되는지, 함수의 인수가 몇 개 필요한 지를 알아야 한다. 이런 자료에 대한 정보는 컴파일러의 앞부분, 즉 어휘 문석 단계와 구문 분석 단계에서 수집되어 심벌 테이블(symbol table)이라는, 곳에 저장된다. 테이블 관리는 이러한 정보들을 관리하는 부분이며 여기에서 사용되는 자료 구조를 심벌 테이블이라 한다. 심벌 테이블에 관한 내용은 이 책의 12장에 자세히 설명되어 있다.

컴파일러에서 중요한 기능의 한가지는 소스 프로그램의 에러를 발견하여 사용자에게 알려주는 것이다. 예를 들면, 어휘 문석 단계에서 소스 프로그램의 토큰이 철자가 틀리는 경우가 발생할 수 있고, 구문 분석 단계에서는 괄호가 빠지는 것과 같은 문법 규칙에 대한 에러가 발생할 수 있다. 이와 같은 에러에 대한 처리 기술은 컴파일러를 실제로 구현하는데 있어 매우 중요하다. 컴파일러에는 이런 에러들을

**28페이지**

처리하여 주는 루틴(error handling routine)을 갖고 있으며 에러 처리에 대해서는 이 책의 13장에서 살펴보기로 하겠다.

이상에서 우리는 컴파일러의 전반적인 과정을 살펴보았다. 컴파일러가 행하는 크고 복잡한 일을 기능적으로 독립적인(functionally independent) 여러 단계들로 나누어 쉽게 구현할 수 있게 했다. 컴파일러의 구현에 있어서 하나 이상의 단계들을 모아 패스(pass)라고 하는 하나의 모듈로 묶을 수 있다. 하나의 패스는 소스 프로그램이나 그전 패스의 출력을 읽어들여 그 패스를 이루고 있는 단계들의 기능에 따라 입력을 변환하여 중간 파일에 출력한다. 그러면 이 출력은 다음 패스의 입력이 된다.

만약 여러 단계가 모여 하나의 패스를 이룬다면, 서로간의 제어에 따라 각 단계들은 상호간의 수행을 할 수 있다. 어휘 분석 단계와 구문 분석 단계를 대표적인 예로 들 수 있는데, 어휘 분석기에서 무조건 토큰들을 생산하는 것이 아니라 구문 분석 단계에서 토큰이 필요할 때마다 어휘 분석 단계에게 토큰을 요구하게 하여 마치 부프로그램과 같은 역할을 할 수 있게 한다.

컴파일러의 컴파일 과정을 전단부와 후단부로 분리하여 작성하는데 초창기의 컴파일러는 이러한 구분 없이 전과정을 하나의 패스로 구성한 단일 패스 컴파일러(single pass compiler)였다. 컴파일러에서 패스의 개수는 사용하는 목적 기계나 프로그래밍 언어에 따라 결정된다. 여러 개의 패스로 구성된 컴파일러를 다중 패스 컴파일러(multi-pass compiler)라 부른다.

다중 패스 컴파일러는 기능에 따라 여러 패스로 구성되어 있기 때문에 컴파일러의 부분적인 기능을 개선하고 다른 기종으로 이전하기에도 편리하다. 또한, 다중 패스 컴파일러는 단일 패스 컴파일러보다 작은 기억 공간을 요구한다. 왜냐하면, 하나의 패스가 사용했던 공간을 다음 패스가 다시 사용할 수 있기 때문이다. 그러므로, 작은 기억 공간을 가진 컴퓨터에서는 다중 패스 컴파일러가 유리하다. 그 외에도 다중 패스 컴파일러는 단일 패스 컴파일러보다 많은 장점을 갖고 있어 요즈음 대부분의 컴파일러는 다중 패스로 구성된 컴파일러이다. 그러나, 다중 패스 컴파일러는

단일 패스 컴파일러보다 컴파일 속도가 느리다는 것이 단점이다.

**29페이지**

1.4 컴파일러 자동화 도구

컴퓨터 하드웨어 기술이 빠르게 발전하고 필요한 소프트웨어의 규모와 복잡성이 증가함에 따라 신뢰성이 있고 효율적으로 프로그래밍할 수 있는 프로그래밍 언어의 설계 및 이에 따른 컴파일러를 자동적으로 제작하는 것이 중요한 문제로 등장하였다.

다시 말해서, 프로그래밍 언어와 컴퓨터 구조가 다양해짐에 따라 많은 컴파일러가 필요하게 되었다. 즉, N개의 언어를 M개의 기계에 구현하려 할 때 N\*M개의 컴파일러가 필요하다. 이에 따라 컴파일러 제작을 도와주는 도구들이 필요하게 되었고 컴파일러의 전과정이나 각 단계들을 자동적으로 생성하는 도구들에 대한 연구가 활발히 진행되었다.

이러한 도구들을 총칭해서 컴파일러 생성기(compiler generator) 또는 컴파일러-컴파일러(compiler-compiler)라 부른다. 컴파일러-컴파일러는 특정한 프로그래밍 언어를 위한 언어 표현(language description)과 목적 기계에 대한 기계 표현(machine description)을 입력으로 받아 하나의 컴파일러를 출력한다. 이때 생성된 컴파일러는 언어 표현에 기술된 소스 프로그램을 입력으로 받아 목적 기계의 코드로 번역해 주는 역할을 수행한다.

그림 1.15 컴파일러-컴파일러의 기능

[그림 1.15]에서 제시된 컴파일러-컴파일러 모델은 이상적인 경우이고 실제적으

**30페이지**

로 요사이에 개발된 번역기 제작 시스템(translator writing system)은 어휘 분석, 구문 분석, 그리고 코드 생성과 같이 컴파일러의 한 단계(phase)만을 취급한다. 이렇게 만들어진 컴파일러의 단계는 다른 자동화 도구에 의해 생성된 단계와 통합하여 구현될 수 있다. 컴파일러의 각 단계를 생성하는 자동화 도구를 기능적인 면에서 살펴보면 [그림 1.16]과 같다.

그림 1.16 자동화 도구의 기능

생성기-생성기(generator-generator)는 생성될 단계의 기능을 묘사하는 메타 언어(meta language)를 입력으로 받아 각 단계가 사용하게 될 테이블(또는 프로그램)을 출력한다. 그러면, 각 구동기(driver routine)는 이 테이블을 이용하여 그 단계에서 수행해야 할 일을 처리한다. 따라서 필요한 컴파일러 단계를 구현하기 위하여 프로그램을 작성하는 것이 아니라 그 단계에서 처리하는 작업을 메타 언어로 기술하면 되는 것이다.

어휘 분석이나 구문 분석 단계에 대한 자동화 도구는 1970년대에 나타났으나 후단부 작성을 도와주는 도구는 최근에 이르러서야 발표되기 시작하였다. 그 이유는 최적화나 코드 생성에 대한 정형화(formalism)가 어려웠기 때문이다. 이 장에서 우리는 컴파일러의 각 단계별로 자동화 도구에 관하여 논하기로 하겠다.

1.4.1 어휘 분석기 생성기

어휘 분석기 생성기(Lexical Analyzer Generator)는 어휘 분석기를 자동으로 생성하는 도구이다. 토큰에 대한 표현을 입력으로 받아 기술된 형태의 토큰을 찾아내는 어휘 분석기를 만든다. 생성된 어휘 분석기는 입력 프로그램에서 토큰들을 구분해 내는 일을 한다.

**31페이지**

그림 1.17 어휘 분석기 생성기

일반적으로 토큰의 형태를 기술하는 토큰 표현(token description) 방법으로는 정규 표현(regular expression)을 사용한다. 정규 표현에 대한 자세한 내용은 3.2절을 참고하기 바란다.

어휘 분석기 생성기의 대표적인 예로는 UNIX 운영 체제하에서 수행되는 렉스(Lex)를 들 수 있다. 렉스는 1975년 벨 연구소(Bell Lab.)에 근무하는 레스크(M.E.Lesk)에 의해 개발된 소프트웨어 도구(software tool)로 토큰의 형태를 묘사한 정규 표현들과 각 정규 표현이 매칭되었을 때 처리를 나타내는 액션 코드(action code)로 구성된 입력을 받아 어휘 분석의 일을 처리하는 프로그램을 출력한다.

렉스의 기능을 그림으로 나타내면 [그림 1.18]과 같다.

그림 1.18 렉스의 기능

여기서, lex.yy.c가 어휘 분석 작업을 담당하도록 생성된 C 프로그램이다. 렉스에 대해서는 4.4절에서 자세히 설명하기로 한다.

1.4.2 파서 생성기

파서 생성기(parser generator)란 언어의 문법 표현(grammar description)으로부터 파서(또는 구문 분석기)를 자동으로 생성하는 도구를 말하며 그 기능은 [그림 1.19]와 같다.

**32페이지**

보통 파서 생성기를 파서 제작 시스템(PGS: Parser Generating System)이라 부른다.

그림 1.19 파서 생성기의 기능

언어의 문법 표현으로부터 파서 생성기는 파서를 제어하는 테이블을 생성한다. 파서는 이 테이블을 이용하여 주어진 문장에 대한 문법적인 검사를 하며 다음 단계에서 필요한 의미 정보(semantic information)를 만든다. 모든 언어에 대하여 파서 부분은 동일하고 테이블만 다르다. 그러므로, 새로운 언어에 대한 파서를 만들기 위해서는 단지 문법 표현만 바꾸면 된다. 일반적으로 문법 표현으로는 context-free 문법을 사용한다.

파서 생성기는 컴파일러를 구현하는데 있어서 꼭 필요한 도구이며 따라서 각 대학이나 연구소에는 고유의 기능을 갖는 파서 제작 시스템을 갖고 있다. 이 절에서는 파서 생성기의 대표격인 YACC를 살펴보도록 하자.

그림 1.20 Lex와 YACC의 기능

YACC(Yet Another Compiler-Compiler)는 1975년에 벨 연구소에 근무하는 존슨(S.C. Johnson)을 중심으로 개발된 파서 생성기로 렉스와 마찬가지로 UNIX 운영 체제하에서 수행되는 유틸리티(utility)이다. YACC는 문법 규칙과 그에 해당하는 액

**33페이지**

션 코드를 입력으로 받아 파싱을 담당하는 프로그램을 출력한다. 일반적으로 YACC는 파서만을 생성하고 따라서 필요한 스캐너는 전술한 렉스를 사용하여 생성 할 수 있다. 다음 [그림 1.20]은 렉스와 YACC를 이용하여 컴파일러의 전단부를 구현하는 예를 보여준다.

여기서, y.tab.c가 YACC로부터 생성된 프로그램으로 문법 규칙에 의해 기술된 언어의 문장에 대해 구문 검사(syntax checking)를 하며 문법 규칙과 결합된 액션 코드를 필요할 때마다 수행한다. 렉스와 YACC의 액션 코드는 일반적으로C 언어로 작성한다. YACC의 자세한 내용은 14장을 참조하기 바란다.

1.4.3 코드 생성의 자동화

코드 생성은 중간 언어를 목적 기계 언어로 바꾸는 컴파일러의 과정으로 기계 정형화(machine formalization)를 통하여 자동적으로 구성하려 한다. 목적 기계의 명령어들을 나타낸 테이블을 사용하여 기계 독립적인 코드 생성 알고리즘(CGA: Code Generation Algorithm)을 고안하는 방법이다. 코드 생성 과정을 기계 종속적인 (machine dependent) 부분과 독립적인(machine independent) 부분으로 나누어 정형화하는데 목적을 두고 있으며 주로 템플릿 매칭(template matching) 또는 테이블을 이용한 방법(table driven method)에 의해 코드를 생성한다.

코드 생성 과정을 자동화한다는 것은 [그림 1.21]에서 보는 것과 같이 기계 독립적인 CGA와 CGA가 사용하는 테이블을 기계 표현(machine description)으로부터 자동적으로 유도하는 것을 의미한다.

그림 1.21 코드 생성기 자동화 모델

코드 생성에 관한 과거의 연구는 두 분야로 나눌 수 있는데, 그 첫째는 패턴 매

**34 페이지**

칭 코드 생성(pattern matching code generation)이고, 다른 하나는 테이블을 이용한 코드 생성(table driven code generation)이다. 이들 연구는 모두 기계 종속적인 테이블(machine dependent table)과 코드 생성 알고리즘을 분리해서 코드 생성 과정을 정형화 하려는 의도이다.

그러나, 그들의 방법은 테이블 구성을 위해 적용되는 알고리즘과 기계 표현(machine description)의 일반성 면에서 서로 조금씩 다르다. 패턴 매칭 코드 생성에서는 목적 코드를 생성하기 위해 경험적인 검색 알고리즘(heuristic search algorithm)을 사용하고, 반면에 테이블을 이용한 코드 생성에서는 context-free 파싱 이론으로부터 유도된 결정적 파싱 알고리즘(deterministic parsing algorithm)을 사용한다.

1.4.4 컴파일러-컴파일러 시스템

컴파일러의 한 부분만을 자동화하면 그 나머지 부분은 컴파일러 설치자가 제공해야 한다. 그러므로 실질적인 컴파일러 제작 시스템이 되기 위해서는 전과정을 자동화해야 한다. 이와 같은 의도는 PQCC(Production-Quality Compiler Compiler)팀과 ACK(Amsterdam Compiler Kit)팀에 의하여 시도되었다. PQCC와 ACK는 컴파일러 개발 과정을 자동화하기 위한 도구들이다.

카네기 멜론(Carmegie-Mellon) 대학에서 울프(W. Wulf)를 중심으로 개발된 PQCC는 소스 언어와 목적 기계를 정형화하여·컴파일러를 생성하는 시스템이다. 컴파일러의 전단부가 잘 정립된 반면 후단부는 미비한 상태이다. 그러나, 실질적인 번역기 제작 시스템(translator writing system)을 구현하려면 체계적인 후단부가 필요하다. 이와 같이 실질적인 컴파일러-컴파일러 시스템을 구성하려는 연구가 PQCC이다.

PQCC 시스템의 전반적인 개요는 [그림 1.22]와 같다.

그림 1.22 PQCC 시스템

**35페이지**

PQCC에서는 트리 형태의 중간 언어인 TCOL(Tree Common Oriented language)를 사용하였으며 그 시스템의 기능은 언어와 목적 기계에 대한 정형화된 표현을 입력으로 받아 PQC(Production Quality Compiler)라고 부르는 기초 코드 생성기와 최적화기가 사용하는 테이블을 만들어 낸다. 그리고 PQC는 이 테이블을 이용하여 전단부에서 생성한 TCOL을 목적 코드로 번역하는 작업을 한다.

그림 1.23 ACK의 구성

ACK는 컴파일러의 후단부를 자동화하기 위한 도구의 하나로서 이식성과 재목적성이 매우 높은 컴파일러를 만들기 위한 실질적인 노구이다. 이것은 네덜란드에 소재하고 이는 브리제(Vrije) 대학의 타넨바움(Andrew S. lanenbaum)을 중심으로 개발되었다.

각 언어에 대한 전단부가 중간 언어인 EM(Encoding Machiné) 코드를 생성해 낸다. EM-코드는 가상적인 스택 기계에 근거를 둔 일종의 어셈블리어이다. 그리고 생성된 EM 코드에 대한 최적화 과정(peephole optimization & global optimization)

**36페이지**

을 거친 후에 후단부에 의해 목적 프로그램으로 변환된다.

ACK에서 작동하는 컴파일러는 다음과 같은 8개의 단계로 구성되어 있으며 ACK는 각 단계를 효과적으로 생성할 수 있는 방법을 제공해 주고 있다.

① 프리프로세서(preprocessor)

② 전단부(front-end)

③ 핍홀 최적화기(peephole optimizer)

④ 전역 최적화기(global optimizer)

⑤ 후단부(back-end)

⑥ 목적 기계 최적화기(target machine optimizer)

⑦ 범용 어셈블러와 링커(universal assembler/linker)

⑧ 응용 패키지(utility package)

전단부는 각 언어당 하나씩 존재하며 후단부는 각 기계당 하나씩 필요하다. 현재 ACK에서 제공한 전단부에는 Fortran, Algol, Pascal, BASIC, Modular-2, Occam, C, Ada, C++ 등이 있으며, 후단부는 Intel 8086/80286/80386, Motorola 68000/68020, Zilog Z80/Z8000, Vax 시스템, Sparc 프로세서, Sun2, Sun3 등이 있다.

PQCC와 ACK에 관한 자세한 내용은 14장을 참고하기 바란다.

**37 페이지**

chapter 01 연습문제

1.1 다음 괄호에 알맞은 말을 쓰시오.

(1) 한 프로그래밍 언어로 쓰여진 프로그램을 입력으로 받아 그와 동등한 의미를 갖는 다른 프로그래밍 언어로 된 프로그램을 출력하여 주는 시스템 프로그램을 ( )라 말한다.

(2) 좋은 프로그래밍 언어는 ( )와 Semantics가 명료해야 한다.

(3) 한 기종에서 작동하는 컴파일러를 다른 기종으로 그대로 이전하면 ( )가 된다.

(4) 컴파일러의 일반적인 구조는 어휘 분석, 구문 분석, 중간코드 생성, ( ) 그리고 목적코드 생성 과정으로 나눌 수 있다.

(5) 어휘 분석기를 간단히 ( )라 부르며 구문 분석기를 ( )라 부른다.

(6) 중간 코드 생성기의 입력은 ( )이고 출력은 중간 코드이다.

(7) 최적화는 그 위치에 따라( )와 ( )으로 나눌 수 있다.

(8) N개의 언어를 M개의 기종에 설치하려할 때, ( )개의 컴파일러가 필요하다.

(9) 언어 표현과 목적 기계에 대한 기계 표현을 입력으로 받아 하나의 컴파일러를 생성하는 도구를( )라 부른다.

(10) 어휘 분석기 생성기의 입력은 ( )이고 출력은 ( )이다.

(11) 파서 제작 시스템의 입력은 ( )이고 출력은 ( )이다.

(12) 생성기-생성기는 생성될 단계의 기능을 묘사하는 ( )를 입력으로 받아 각 단계가 사용하게 될 테이블(또는 프로그램)을 출력한다.

(13) 일반적으로 YACC는 파서만을 생성하기 때문에 스캐너는( )를 사용하여 생성해야만 한다.

(14) POCC에서 사용한 중간 언어는 ( )이고 ACK에서 사용한 중간 언어는 ( )이다.

(15) C# 언어의 중간 코드는 ( )이고 실행 환경은 ( )이다.

**38페이지**

1.2 다음 약어에 대한 원어를 쓰시오.

(1) PGS

(2) YACC

(3) PQCC, TCOL

(4) ACK, EM

(5) JIT

1.3 컴파일러의 일반적인 구조를 어휘 분석기, 구문 분석기, 중간 언어 생성기, 코드 최적화기, 그리고 목적 코드 생성기로 나누어 설명하고 특히, 각 단계의 입력과 출력을 설명하시오.

1.4 코드 최적화는 관점에 따라 지역 최적화와 전역 최적화로 구분할 수 있다. 지역 최적화(핍홀 최적화)에 의해 얻을 수 있는 효과를 열거하시오.

1.5 목적 코드 생성기가 목적 코드를 생성하기 위하여 행하는 작업을 크게 4가지로 나누어 설명하시오.

1.6 렉스와 YACC의 기능을 간단히 설명하시오. 그리고 렉스와 YACC를 이용하여 컴파일러의 전단부를 구현하는 방법을 설명하시오.

1.7 컴파일 과정을 어휘 분석, 구문 분석, 중간 코드 생성,/코드 최적화, 목적 코드 생성 단계로 나누어 설명하고 각 단계를 자동적으로 구성하기 위한 컴파일러 제작 도구에 관하여 논하시오.

**649페이지**

Index

가상 terminal 358

가상 기계 391

가상 기계 코드 392

간접 주소 명령어 398

간접(indirect) left-recursion 249

값 스택 428, 443

객체지향 언어 16

거듭제곱(power) 45

검색 연산 529

결정적 유한 오토마타 83

계승자(successor) 505

고급 언어 15

고울 룰 363

골격 프로그램 570, 584

곱(product) 44

공리 78

공통 부분식의 제거 508, 515

괄호 언어 60

구문 단계 에러 554

구문 분석 232

구문 분석 방법 233

구문 분석기 24, 232, 239

구문 에러 547

구문 트리 243

구문 모델 492

귀납 증명법 55

기계 종속적 최적화 518

기본 블록 503

기본 언어 576

노드 이름 432

논리식 479

논리적 에러 547

다중 패스 컴파일러 28

단말 노드 432

단순 매칭 언어 60

단일 명령어 396

단일 생성 규칙 74, 190

단일 패스 컴파일러 28

닫혀있다 104

대수학적 간소화 510

대입 182, 252

도달 불가능한 상태 94

도달될 수 없는 코드의 제거 516

동기화(synchronizing) 토큰 555

동등한 문법 64

동적 에러 549

동치 관계 101

**650페이지**

레벨 테이블 538

레지스터 배정 520

레지스터 할당 27, 520

렉스 31, 151, 570

렉스의 입력 151

렉스의 입력 명세 571

루프 불변 511

루프 언롤링 513

루프 융합 513

루프 최적화 511

링커 607

마크 심벌(mark symbol) 320

메타 심벌 196

메타 언어 425, 569

명세 파일 574

명칭 리스트 52

명칭의 인식 134

모순 증명법 55

모호성 177

모호한 문법 177, 350

목적 기계 최적화 605

목적 기계 최적화기 생성기 605

목적 기계 표현 테이블 605

목적 언어 19

목적 코드 생성기 26

무한 언어 44

문맥 자유 문법 62

문법 46

문법 심벌 46

문법 흐름도 199

문법-지시적 변환

문법-지시적 변환 시스템

문법을 고안한다 52

문장 45, 48

문장 부분 456

바이트코드 408

번역기 19

번역기 시스템 423

번역기 제작 시스템 30, 568

범용 어셈블러 607

범위 검사 명령어 398

복합문 478

본질적으로 모호하다 180

부분 집합 구성 100

분석 단계 526

블록 구조 언어 537

비결정적 유한 오토마타 88

비단말 노드 432

비정렬된 심벌 테이블 531

사상 함수 83

삽입 연산 529

상대 주소 457

상수 전파 509

상수 폴딩 509

상태 전이표 84

생성 규칙 46

생성기-생성기 30

선언 부분 456

선언문 456

**651페이지**

선행자(predecessor) 505

소스 언어 19

속성 526

수식문 479

순서 트리 175

순서쌍 132

스캐너 130, 569

스캐너 생성기 569

스택 구현 트리 구조 심벌 테이블 539

스택 구현 해시 구조 심벌 테이블 540

스택 기계 391

스택 심벌 테이블 538

스택 운영 명령어 397

스탠포드 파서 제작 시스템 583

스트링 41

스트링 상수의 인식 139

스트링 풀 528

스트링의 길이 42

스트링의 접속 42

시작 상태 83

시작 심벌 46, 171

식 462

실수 상수의 인식 137

실행 시간 에러 547, 549

심벌 테이블 27, 132, 526

알골 16

알파벳 41

액션 코드 156, 576

어휘 분석기 23, 130

어휘 분석기 생성기 30

언어 고안자 56

언어 번역기 14

언어 설계자 130

언어 정의 43

에러 메시지 550

에러 처리 28, 545

에이다 17

연산 강도 경감 509, 512

연산자 문자 154

연쇄법 535

외부 표현법 414

우문장 형태 173

우선형 문법 72

우측 결합 353

우측 유도 173

우파스 173, 239

위스콘신 파서 제작 시스템 588

유도 172

유도 트리 174

유일해 80

유한 언어 44

유한 오토마타 언어 103

응용 패키지 607

의미 규칙 425

의미 분석 25

의미 에러 547, 549, 563

의사 변수 577

이동 209

이진 명령어 396

인식기 61, 83

인터넷 프로그래밍 18

인터프리터 20

일반적인 top-down 방법

일반적인 표기법

입력 명세 573

**652페이지**

입력 심벌의 길이 296

자바 17

저급 언어 15

전단부 22, 598

전방법 535

전역 상수 폴딩과 전파 515

전역 최적화 26, 515, 602

전처리기 20

정규 문법 57, 72, 73

정규 언어 58, 72

정규 언어의 닫힘 성질 117

정규 언어의 속성 106

정규 표현의 연산자 76

정렬된 심벌 테이블 531

정수 변수 선언 203

정수 상수의 인식 135

제곱값 중간법 535

제산법 534

제어 흐름 명령어 398

제어문 479

종합 단계 526

좌문장 형태 173

좌선형 문법 72

좌우 대칭 언어 60

좌측 결합 353

좌측 유도 173

좌파스 173, 239

주석 140

중간 언어 376

중간 코드 생성 24

중간 코드 생산가422, 451, 452

중복 매칭 인어 60

중복된 load의 제거 518

증감 연산자 474

지역 최적화 26, 508

지정어 테이블 143

직접(direct) left-recursion 249

종결 상태 83

초기 형상 292, 317

촘스키 계층 57

촘스키 정규 형태 193

최단 거리 매칭 554

추가된 문법 320

추가된 생성 규칙 320

추상 구문 트리 24, 243, 385

축약된 유한 오토마타 103

컴파일 시간 에러 547

컴파일러 14, 22

컴파일러 구현자 130

컴파일러 생성기 29, 568

컴파일러-컴파일러 29, 568

컴파일러-컴파일러 시스템 34

코드 생성 알고리즘 591

코드 생성기604

코드 생성의 자동화 33, 591

코드 최적화 502

코드 최적화기 25

**653페이지**

코드-생성기 생성기 591, 604

코볼 16

크로스 컴파일러 19

테이블을 이용한 코드 생성 594

텍스트 문자 154

토큰 23, 130

토큰 값 131

토큰 번호 131

튜링 기계 61

트리 구조 심벌 테이블 532

파서 생성기 31, 572

파서 제작 시스템 318, 359

파스 트리 241

파스 트리를 구성하는 방법 261

파스칼 16

파싱 스택 259, 427

파일 형식 403

패턴 매칭 코드 생성 593

패턴 묘사 테이블 602

패턴-매칭 521

펌핑 렘마 119

포트란 16

폴딩법 535

푸시다운 리스트 208

푸시다운 오토마타 208

프레임 600

프로그래밍 언어 15

프로시저 284

프리프로세서 20, 598

필요 없는 생성 규칙 183

필요한 심벌 184

핍홀 521

핍홀 최적화 521, 601

함수 정의 488

함수 호출 487

해석적 코드 생성 592

해시 값 534

해시 심벌 테이블 534

해시 함수 534

형 변환 25, 471

형식 매개변수 489

형식 언어 40

확장 182

확장된 BNF 196

확장된 PDA 212

회문 언어 60

후단부 22, 604

후방법 535

흐름 그래프 505

힙(heap) 393

**654페이지**

accept 상태 261

accessible symbol 184

ACTION 테이블 315

ACK 35, 597

AST 385

AST 노드의 구조 437

AST 정보 432

AST의 구성 429

AST의 생성 438

AST의 설계 432

AVAIL 539

AVL 트리 533

bison 578

BNF 195

bottom marker 292

Bottom-up 방법 255

bottom-up 방식 233, 236

bubble.mc 615

buildNode() 440

buildTree() 441

C 언어 17

C#(C sharp) 17

C++ 언어 17

C0 323

C0 방법 340

C1 335

C1 방법 340

canonical collection 319

CLOSURE 321

closure 아이템 320

CLR 411

codeGen() 454

context-free 문법 170

context-free 언어 58, 215

context-sensitive 언어 58

cycle-free 192

dangling-else 355

DFA의 상태수 최소화 100

Diana 390

ЕСР 590

EGO 603

EM 35, 394, 599

embedded 생성 규칙 53

empty 스트링 42

end marker 292

FIRST 271

flex 578

FOLLOW 276

genLabel() 484

GNU 383, 578

GOTO 그래프 325

GOTO 테이블 315

**655페이지**

GOTO 함수 323

Greibach 정규 형태 194

handle 256

handle pruning 258

IR 379

L(G) 49

L(M) 86, 91

L(P) 210

L(a) 77

L\* 45

LALR 340

LALR(1) 문법 619

Le(P) 213

leader 503

left-recursive 249

lex.yy.c 152, 158

left-factoring 249

LL 문법 254

LL 조건 254, 278

LL(1) 문법 296

LL(1) 파싱 테이블 296

LL(k) 문법 303

LLG 72

LOOKAHEAD 281

lookahead 334

LR 구문 분석 314

LR 파서 314

LR 파싱 테이블 315

LR(0) 아이템 319

LR(1) 문법 337

LR(k) 문법 348

k-LOOKAHEAD 302

kernel 아이템 320

Mini C 612

minimum Hamming distance 551

MSIL 410

NFA에서 DFA로의 변환 92

nonterminal 심벌 46

nullable 272

nullable nonterminal 189

P-코드 392

pal.mc 617

panic mode 555, 558

Pascal-P 컴파일러 392

PDA P 209

PE 411

perfect.mc 616

postcode optimization 26

Postfix polish 표기법 378

PQCC 34, 596

precode optimization 26

**655페이지**

Predictive 파싱 테이블 290, 295

prefix 42, 44

Prefix polish 표기법 379

prime.mc 615

printNode() 444

printTree() 444

processOperator() 462

proper 192

Quadruple 381

Recursive-descent 파서 281, 289

recursively enumerable 언어 57

reduce 255

reduce 아이템 320

reduce-reduce 충돌 333, 350

retract 144

ring sum 272

RLG 72

RTL 383

rv\_emit() 467

SCANGEN 588

set/reset 연산 529

Shift 행동 259

shift-reduce 충돌 332, 350

start 지시선 86

strong LL 조건 283

Strong LL(k) 문법 301

suffix 42, 44

T\* 43

T+ 43, 45

TCOL 35, 387

terminal 심벌 46

terminating nonterminal 184

Top-down 방식 233, 246

triple 381

U-코드 395

U-코드 번역기 492

U-코드 인터프리터 495

UNCOL 597

Viable prefix 321

WR 43

YACC 32, 357, 572

yylex() 159

yytext 157

yywrap() 159

E-CLOSURE 98

e-free 188

E-free 문법 188

E-NFA 98

E-생성 규칙 74, 188

**657페이지**

e-이동 210

e-전이 97

.NET IL 410

3-주소 코드 81