7. Semantic Analysis

목차

7.1 의미 분석의 개요

- **의미 분석(semantic analysis) 이란** :
  - **구문 트리**와 **기호표**에 있는 정보를 이용하여 소스 프로그램이 **언어 정의에 의미적으로 일치**하는지를 검사하고, 다음 단계인 중간 코드 생성에 이용하기 위해 **자료형 정보를 수집**하여 구문 트리나 기호표에 저장하는 것
  - 이와 같은 일을 담당하는 도구를 **의미 분석기**(semantic analyzer)
  - 의미 분석에서는 파스 트리(혹은 구문 트리)가 입력되고, 출력으로는 의미 분석이 이뤄진 파스 트리(혹은 구문 트리)가 생성되거나 구문 지시적 번역(syntax directed translation)을 이용하면 직접 중간 코드를 얻을 수도 있다.
  - 그런데 자연 언어에서는 의미 분석이 의미가 있지만 형식 언어는 의미를 가지고 있지 않으므로 의미 분석에서 크게 중요하지 않다. 따라서 의미 분석 단계에서 이미 어휘 분석 단계부터 만들어진 기호표(symbol table)에 필요한 정보를 저장하거나 참조해야 하고, 식별자가 프로그램 내의 수식과 문장에서 해당 언어의 형 규칙에 따라 올바르게 사용되고 있는지를 검사하는 **형 검사**(type checking)를 수행한다.

7.1 의미 분석의 개요

- **형 검사란** 각 연산자가 소스 프로그램 규칙에 의해 허용 된 피연산자를 가졌는지 검사하는 것이다.
  - 예를 들면 어떤 프로그래밍 언어에서 배열의 첨자 는 정수형만을 허용하는데 배열의 첨자에 부동 소수형을 사용했다면 의미 분석기가 에러임을 지적한다.
- **의미 분석 과정에서 하는 일은**
  - 상수 정의(constant definition), 형 정의(type definition), 변수의 형 선언(variable type declaration) 등이다.
  - **상수 정의 과정**에서는 상수의 이름이 기호표에 들어가고 각 **상수는 정의된 값을 기호표**에 보관하여 이후 적당한 기회에 기억 공간을 배정받고 초기값을 갖는다.
  - **형 정의 과정**에서는 주어진 생성 규칙의 의미에 의해 형의 해당 자료 구조가 구성되며, **이런 자료 구조는 기호표**에 보관된다.
  - 형 정의 과정에서 구성된 자료 구조는 형 설명자(type descriptor)를 만들어서 구성된 형에 대한 자료 구조 크기나 성격 등을 보관하고, 이후 형들이 기억 공간을 배정받을 때 필요한 정보를 제공하게 한다. **변수의 형 선언 과정**에서는 선언되는 변수들이 선언된 형에 의해 설명자를 얻을 수 있어 **변수 이름과 형 설명자가 모두 같이 기호 표**에 들어가며, 이때 각 변수는 기억 공간을 배정받아 해당 형을 구성하여 해당 변수에 기억 공간을 배정하게 된다.

7.1 의미 분석의 개요

- 일반적으로 수식에 대해 연산을 할 때 연산은 크게 두 가지 종류가 있다.
  - 하나는 프로그래밍 언어에서 피연산자와 연산 결과의 형이 고정되어 있는 연산, 즉 **형 고정 연산**(type specific operation)이고, 또 하나는 프로그래밍 언어에서 피연산자와 연산 결과의 형이 변할 수 있는 연산, 즉 **일반적 연산**(generic operation)이다.
- 예를 들어 a + b와 같은 산술식을 생각해보자. a와 b의 자료형이 각각 int형과 float형이라면 형 고정 연산에서는 에러 메시지를 주지만, 일반적 연산에서는 a나 b의 형을 변환하여 연산을 허용한다.
  - **형 변환**(type conversion)은 주어진 자료형의 값을 다른 자료형의 값으로 변환하는 것을 말하며, 시스템에서 자동으로 형을 변환하는 **묵시적 형 변환**(implicit type conversion)과 프로그래머가 명시적으로 형 변환을 요구하는 **명시적 형 변환**(explicit type conversion)이 있다.
  - 명시적 형 변환은 프로그래머가 명령문으로 요구한 형 변환으로 **캐스트**(cast) 연산이라고 일컫는다. 그리고 묵시적 형 변환은 시스템에서 자동으로 형을 변환하는 것으로 **자동 변환**(automatic type conversion) 또는 **강제 형 변환**(coercion)이라고 한다.
- 자동 변환을 하려면 형을 변환해야 하는데, 이는 컴파일러가 판단해서 결정하는 것이 아니라 언어 정의(language definition) 시간에 이미 어떤 연산으로 할 것인지 결정해 놓는다.

7.2 기호표

- **식별자에 대해 기호표**에는 식별자를 나타내는 문자열, 자료의 형을 나타내는 속성, 메모리에서 식별자의 위치 등 속성을 나타낸다.
  - 첫째, 식별자 이름은 문자열로 나타낸다. 같은 이름이 여러 개의 블록이나 프로시저에서 사용된다면 이 식별자가 속해 있는 블록이나 프로시저를 나타내는 것이 있어야 한다.
  - 둘째, 식별자의 레벨, 형식 인자 formal parameter, 배열 등을 나타내준다.
  - 셋째, 배열의 차원 수, 각 차원의 상한과 하한을 나타내야 한다.
  - 넷째, 가능하다면 메모리에서 식별자의 위치를 나타내는 오프셋(offset) 등이 있어야 한다.
- 기호표는 특정 기호가 가진 속성을 포함하는 하나의 레코드로 구성되며, 전형적인 기호표는 다음과 같은 형태이다.

식별자 이름	형	레벨	오프셋	차원 수	기타 속성
A	2	0	0	0	
⋮	⋮	⋮	⋮	⋮	

7.2 기호표

- **기호표의 효과적인 구성은** 전체적인 컴파일 시간에 많은 영향을 준다. 여기서는 기호표를 구성하는 세 가지 방법으로 선형 리스트(linear list), 트리(tree), 해시표(hash table)를 이용하는 방법에 대해 알아본다.
- 선형 리스트 방법은 n개의 항목과 m개의 조회(inquiry)가 있을 때 구현이 간단하지만, 자료의 수가 커지면 이를 모두 조사하여 접근하는 방법을 택하기 때문에 비효율적이다. 이진 탐색 트리(binary search tree) 방법은 구현하기가 어렵지만 (n + m)log n의 횟수가 소요되므로 n이 큰 경우에 효율적이며, 해싱(hashing) 방법은 가장 효율적인 방식이다.
- **선형 리스트를 이용한 기호표 구성**
  - 기호표를 가장 쉽고 간단하게 구현할 수 있는 방법으로 [표 7-1]과 같다.

표 7-1 선형 리스트

이름 1	정보 1
이름 2	정보 2
⋮	⋮
이름 n	정보 n

7.2 기호표

- 이 구조에 새로운 식별자를 추가하려면 선형 리스트를 차례로 검조해야 하며, 새로운 식별자의 이름이 없는 경우에는 선형 리스트의 가장 끝에 새로운 식별자를 추가하고 그에 대한 속성을 기록한다.
- 즉 선형 리스트에 n개의 식별자가 저장되어 있을 때 새로운 식별자 하나를 삽입하려면 n개의 식별자를 검조한 다음에 삽입이 가능하므로 시간이 n에 비례한다. 또한 하나의 식별자를 찾는 데에도 평균적으로 n/2개의 식별자를 찾아야 할 것이다. 임의의 상수 c가 연산에 필요한 시간이라고 할 때, n개의 이름에 대한 삽입과 m개의 조회에 대한 모든 작업량은 cn(m + n) 이 되므로 n과 m이 증가함에 따라 효율이 더 떨어지게 된다. 따라서 선형 리스트 방법은 프로그램의 크기가 작은 작업의 컴파일에 사용된다. 또한 선형 리스트 구조의 장점은 최소한의 메모리를 사용한다는 것이다.

- **트리 구조를 이용한 기호표 구성**
  - 트리 구조 기호표는 각 기호표에 왼쪽과 오른쪽의 2개 링크 필드(link field)를 추가함으로써 기호표의 구조가 보다 효율적이 될 수 있다. 그리고 기호를 만나는 순서에 의해 이진 검색 개념을 이용한 이진 탐색 트리를 구성한다.
  - 즉 새로 추가될 노드의 기호는 트리의 루트 노드를 통해 진행되며, [그림 7-1]에서 보듯이 각 노드의 식별자를 나타내는 이름과 비교하여, 새로 정의된 기호가 이름보다 크면 그 노드의 오른쪽 링크 부분을 따라 다음 노드로 가고, 이름보다 작으면 왼쪽 링크 부분을 따라 다음 노드로 진행되는 과정을 널(null) 링크를 만날 때까지 되풀이한다. 그리고 그 링크 부분에 새로 정의된 기호가 들어 있는 노드를 연결한다.

7.2 기호표

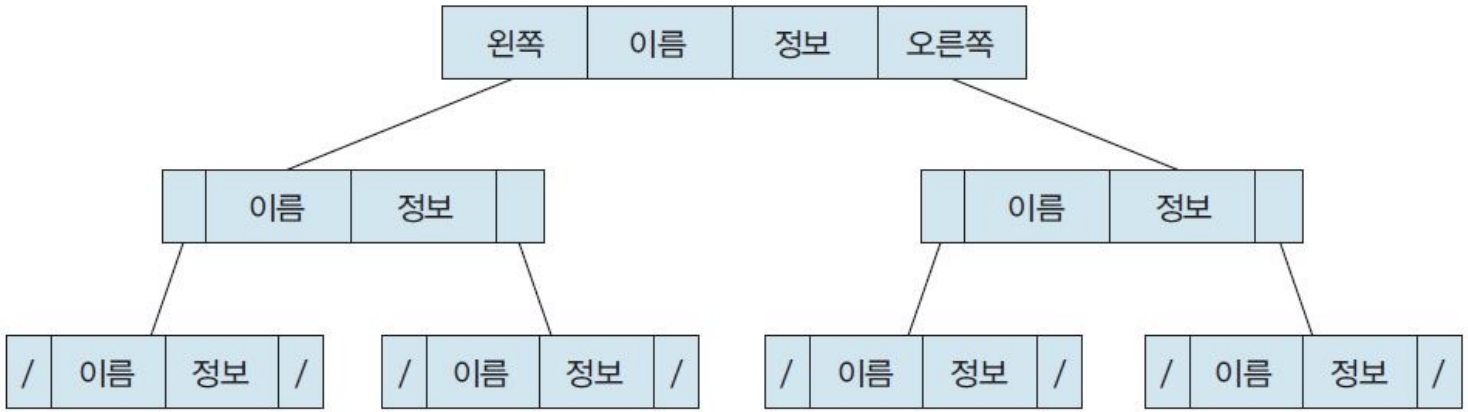


그림 7-1 이진트리 구조를 이용한 기호표

- 이진 트리 구조의 경우 식별자 이름이 임의 순서(random order)대로 나타나며, 트리에서 평균 높이는 식별자 이름의 수가 n이라면 log n에 비례한다. n개의 이름을 추가하는 데 필요한 시간은 n log n이고, m개의 조회를 하는 데 필요한 시간은 m log n이다. 그리고 전체적으로 필요한 시간은 (n + m) log n에 비례한다. 만약 n이 50보다 크면 선형 리스트보다 효율적이다.

7.2 기호표

- **해시표 구조를 이용한 기호표 구성**
  - 해시 함수 – 임의의 길이의 입력 메시지를 고정된 길이의 출력값으로 압축시켜 하나의 문자열을 보다 빨리 찾을 수 있도록 주소에 직접 접근할 수 있는 짧은 길의 값이나 키로 변환하는 알고리즘을 수식으로 표현한 것. 그래서 자료의 무결성 검증, 메시지 인증에 사용한다.
  - 해시표 구성 방법은 기호표 구조 중 가장 효율이 좋은 검색법을 가진 방법이다. 이 방법은 새로 정의된 기호에 대해 해시 함수(hash function)라는 함수를 적용하여 해시표에서 그 위치를 찾아 삽입하는 방법이다. 이 구성표에 대한 검색은 삽입하는 방법과 같이 해시 함수를 적용하여 찾는다.
  - **해시 함수**로는 **제산법(division method)**, **제곱값 중간법(mid-square method)**, **폴딩법(folding method)** 등이 사용된다. 해시표의 간단한 예는 [그림 7-2]와 같다.

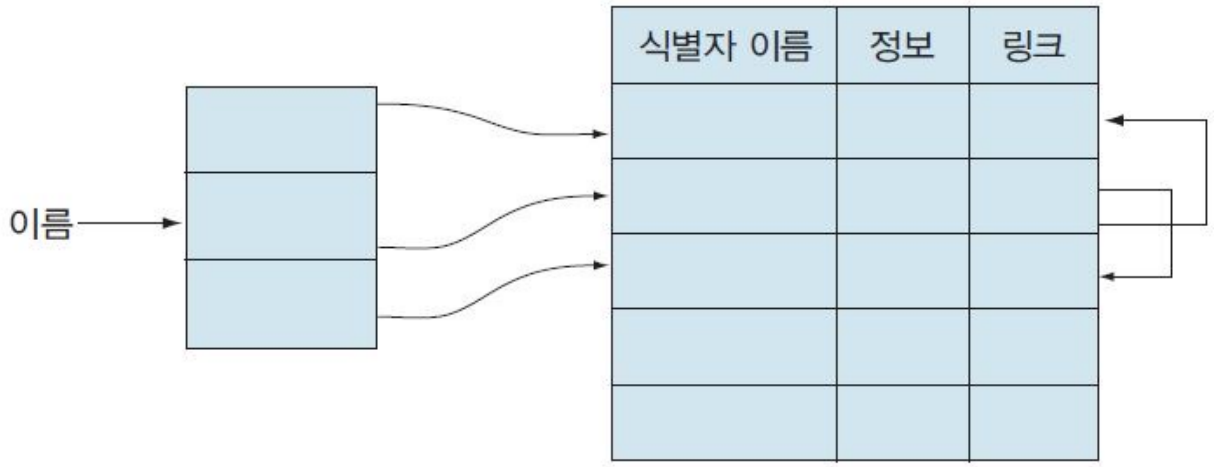


그림 7-2 해시 기호표

7.2 기호표

- 해시표에 n개의 식별자 추가와 m개의 조회가 있다면 전체적으로 필요한 시간은 n(m + n)/k 에 비례한다. 여기서 k는 해시표가 0, 1, …, k-1의 순서를 가진 k개의 단어로 구성되어 있다는 의미이다. 최대 k개의 식별자에 대해 다음과 같이 0부터 k-1까지의 번호를 부여하는 해시 함수를 기호표 검색 방법으로 사용한다.
- hash : name_set → {0, 1, …, k-1}
- 이제 해시 함수의 실제적인 구현 방법을 알아보자. 프로그램 텍스트에서 주어진 식별자의 최 대수 k를 정하고 기호표를 할당한다. 프로그램으로부터 식별자에 대한 토큰이 인식되면 식별자의 첫 두 문자를 아스키코드로 변환하여 이를 합하고, 두 문자의 값을 합한 정수값을 표 크기인 제수(나눗셈에서 나누는 수)로 나눈 후 나머지 값을 식별자에 대한 해시표의 주소로 결정한다.
- 이와 같은 방법으로 기호표의 상대적인 주소를 계산하여 식별자를 기억시키고, 이후에 검색 할 때도 동일한 방법으로 해시 값을 구하여 주소로 취한다. 그러나 서로 다른 식별자에 대해 해시 값이 동일할 수 있기 때문에 한 주소에 여러 개의 식별자가 중첩되는 충돌을 피하기 어렵다.
- 예를 들어 식별자의 첫 두 문자가 같은 경우에는 같은 해시 값을 얻으므로 서로 다른 식별자 들이 동일한 기억 공간을 차지하게 된다. 이러한 충돌을 보완하기 위해 해시 함수는 기호표에서 해시 값의 분산이 고르고 충돌이 최소가 되도록 정의되어야 한다. 이 문제는 제수를 어떤 값으로 정하느냐에 달려 있다.
- 제수를 정하는 방법은 여러 가지가 있으며, 제수를 크게 할수록 충돌이 줄어들지만 메모리를 낭비할 것이고 제수를 작게 할수록 충돌이 많아질 것이다.

7.3 속성 문법

- 속성 문법은 속성과 의미 규칙을 사용하여 문맥자유 문법을 확장한 것으로, 각각의 논터미널 기호 또는 터미널 기호의 속성을 결합하여 의미 분석에 이용한다. 이때 속성이란 어떤 프로그래밍 언어 구문에 관한 특성이다. 예를 들어 컴퓨터 분야가 아닌 다른 분야에서 어떤 사람이나 객체의 속성은 그 사람의 유머 감각이나 객체의 색깔처럼 그 사람이나 객체를 묘사하는 특성을 말한다.
- 속성은 프로그래밍 언어의 정의 시간으로부터 번역 시간 및 실행 시간까지 해당 특성이 결정 되는 시간에 따라 다양한 형태로 나타날 수 있다. 속성의 대표적인 예는 변수의 자료형, 식의 값, 메모리에서 변수의 위치, 어떤 숫자의 유효 자릿수 등이다. 그리고 이러한 속성을 확정하는 것을 **바인딩(binding)**이라 하고, 바인딩이 결정되는 시간을 **바인딩 시간(binding time)**이라 한다. 또한 바인딩 시간이 프로그램 실행 이전인 경우는 **정적 바인딩(static binding)**, 바인딩 시간이 프로그램 실행 중인 경우는 **동적 바인딩(dynamic binding)**이라 한다.
- **구문 지시적 정의**(syntax-directed definition)는 문맥자유 문법에 속성과 의미 규칙을 결합한 것이다. 속성은 문법 기호와 결합하고, 의미 규칙은 생성 규칙과 결합한다. 만약 X가 문법 기호이고 a가 속성이라면 X.a는 특정 파스 트리 노드 X에서의 a 값을 나타낸다.

7.3 속성 문법

- **[예제 7-1] 구문 지시적 정의의 생성 규칙에 대한 의미 이해하기**
- 다음은 간단한 계산기에 대한 구문 지시적 정의이다. 각 생성 규칙에 대한 의미를 설명해보자

생성 규칙	의미 규칙
① S → E $	print(E.val)
② E → E1 + T	E.val = E1.val + T.val
③ E → T	E.val = T.val
④ T → T1 × F	T.val = T1.val × F.val
⑤ T → F	T.val = F.val
⑥ F → (E)	F.val = E.val
⑦ F → **digit**	F.val = **digit**.lexval

7.3 속성 문법

- **[풀이]**
  - 이 구문 지시적 정의는 연산자 +와 *를 가진 산술식 문법을 바탕으로 한다. 산술식의 끝은 $로 끝난다. 여기서 각 논터미널 기호 E, T, F는 val이라 불리는 단 하나의 속성을 가진다. 또한 터미널 기호 digit는 속성 lexval을 가지는데 lexval의 값은 어휘 분석기가 내주는 정수 값이다.

① 생성 규칙 1 : S → E$에 대한 의미 규칙은 E.val을 출력하라는 뜻이다.

② 생성 규칙 2 : E1과 T 속성 값의 합으로 E의 속성 val을 계산한다.

③ 생성 규칙 3 : T의 속성 값은 E의 속성 val을 계산한다.

④ 생성 규칙 4 : T1과 F 속성 값의 곱으로 T의 속성 val을 계산한다.

⑤ 생성 규칙 5 : F의 속성 값은 T의 속성 val을 계산한다.

⑥ 생성 규칙 6 : E의 속성 값은 F의 속성 val을 계산한다.

⑦ 생성 규칙 7 : F.val에 하나의 숫자 값을 주는데 이것은 어휘 분석기가 내주는 토큰 digit의 수치 값이다.

7.3 속성 문법

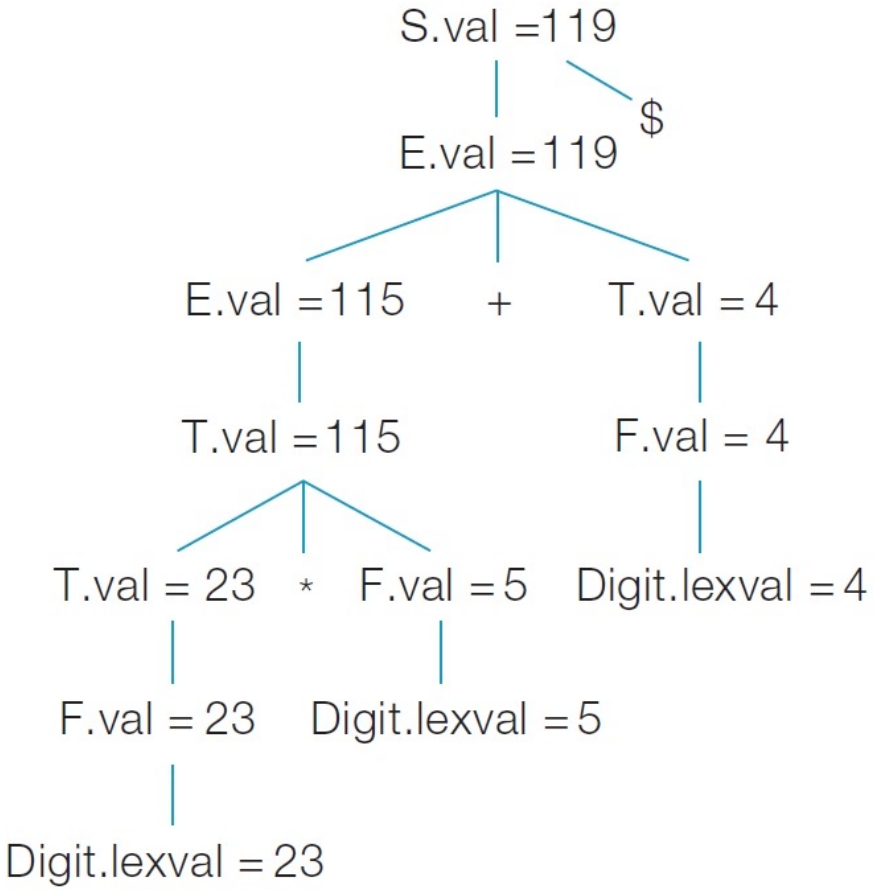
- 생성 규칙에 해당하는 의미 규칙을 기술할 때 논터미널 기호는 목적에 따라 문자열, 숫자, 자료형 등을 나타내기 위한 다양한 속성을 표현하고, 이런 속성을 점(.) 연산자를 이용하여 나타낸다. 예를 들어 논터미널 기호 A의 속성으로 값을 나타내는 value와 기억 공간을 나타내는 addr 속성을 표현하기 위해 A.value와 A.addr로 표기한다. 속성 값은 구하는 위치에 따라 **합성 속성(synthesized attribute)**과 **상속 속성(inherited attribute)**으로 구분된다.
- 합성 속성은 생성 규칙의 왼쪽에 있는 논터미널 기호의 속성이 생성 규칙의 오른쪽에 있는 함수에 의해 결정되는 것을 말한다. 즉 생성 규칙 X → ABC에 대해 다음과 같다.

  X.attribute ::= func(A.attribute, B.attribute, C.attribute)
- 반면에 상속 속성은 생성 규칙의 왼쪽에 있는 논터미널 기호의 속성을 이용하여 생성 규칙의 오른쪽에 있는 논터미널 기호의 속성이 결정되는 것을 말한다. 즉 생성 규칙 X → ABC에 대해 다음과 같다.

  A.attribute ::= func(X.attribute, B.attribute, C.attribute)
- 이때 각 노드에서 속성 값을 주석으로 보여주는 **주석 파스 트리(annotated parse tree)**를 만들 수 있다.

7.3 속성 문법

- **[예제 7-2] 산술식에 대한 주석 파스 트리 만들기**
- 산술식 23 * 5 + 4$에 대한 주석 파스 트리를 만들어보자
- **[풀이]** 산술식 23 * 5 + 4$에 대한 주석 파스 트리는 다음과 같다.



생성 규칙	의미 규칙
① S → E $	print(E.val)
② E → E1 + T	E.val = E1.val + T.val
③ E → T	E.val = T.val
④ T → T1 × F	T.val = T1.val × F.val
⑤ T → F	T.val = F.val
⑥ F → (E)	F.val = E.val
⑦ F → **digit**	F.val = **digit**.lexval

7.4 형 검사

- 만약 어떤 연산자가 부적합한 피연산자에 적용되는 경우에 컴파일러는 에러를 보고하는데 이를 **형 검사**라 한다.
- 형 검사를 수행하기 위해 컴파일러는 소스 프로그램의 각 요소에 **형 표현식(type expression)**을 부여한다. 그 다음 이러한 형 표현식이 어떤 논리 규칙의 집합에 부합하는지를 결정하는데, 이 논리 규칙의 집합을 원시 언어의 **형 시스템(type system)**이라고 부른다.
- **[정의 7-1] 형 표현식**
  - 기본형은 형 표현식이다. boolean, char, integer, real, void가 기본형이다.
  - 형 표현식은 이름을 가질 수 있으므로 형 이름도 형 표현식이다.
  - 형 표현식에 적용되는 형 생성자(type constructor)도 형 표현식이다.
  - 형 표현식을 값으로 가질 수 있는 변수도 형 표현식이다

7.4 형 검사

- **1. 자료형의 종류**
  - 프로그래밍 언어는 항상 int나 real과 같은 몇 개의 기본형을 가지고 있다. 이렇게 미리 정의 된 형은 기계어로 연산이 정의되어 있는 수치 자료형이나 boolean, char와 같이 구현하기 쉬운 특성을 가진 **기본 자료형(elementary data type)**을 기본형이라 한다. 기본형은 그 값이 내부적으로 명시적인 구조가 아니라는 점에서 단순형(simple type)이라고도 한다.
  - 정수를 표현하는 대표적인 방법은 2바이트나 4바이트를 사용한 2의 보수법이다. 실수 또는 부동 소수를 표현하는 대표적인 방법은 4바이트나 8바이트를 사용하여 부호 비트, 지수부(exponential part), 맨티사(mantissa)로 나누어 나타내는 것이다. 한편 문자를 표현하는 대표적인 방법은 1바이트를 사용하여 아스키코드로 나타내는 방법을 이용한다.
  - 어떤 언어에서는 새로운 자료형을 정의할 수도 있는데 대표적인 예로 부분 범위형(sub-range type)과 열거형(enumeration type)이 있다. 예를 들어 파스칼 언어에서는 0부터 9까지의 정수를 가진 부분 범위형을 다음과 같은 방법으로 선언한다.

```
type digit = 0..9;
```

  - 한편 C 언어에서는 red, blue, green 값을 가진 열거형을 다음과 같이 선언한다.

```
typedef enum{red, blue, green} color;
```

  - 부분 범위형과 열거형은 정수로 표현하거나 이러한 값을 나타내는 데 충분한 더 작은 메모리로 구현할 수 있다.

7.4 형 검사

- 미리 정의된 형의 집합이 주어졌을 때 배열, 레코드, 구조체(structure)와 같은 형 생성자를 이용하여 새로운 형을 생성할 수 있다. 이러한 생성자는 이미 존재하는 형을 매개변수로 받아 생성자에 의해 정의되는 새로운 구조의 형을 반환하는 함수로 볼 수 있다. 이를 **구조적 자료형(structured data type)** 혹은 구조적 형이라고 하며 배열, 레코드, 공용체, 포인터 등이 있다. 배열형 생성자는 첨자형(index type)과 원소형(component type)을 매개변수로 받아 새로운 배열형을 생성한다. 파스칼 언어와 같은 구문에서는 다음과 같이 표기한다.

array [index-type] of component-type

- 예를 들어 첨자형이 color이고 원소형이 char인 파스칼 언어에서의 배열형은 다음과 같다.

array [color] of char

- 레코드 또는 구조체형 생성자는 이름과 형의 목록을 받아 새로운 형을 생성한다. C 언어의 구조체형 생성자는 다음과 같다.

```
struct
{ double r;
  int i;
 }
```

7.4 형 검사

- 레코드는 이질형heterogeneous type의 원소가 묶일 수 있다는 점에서, 그리고 이름을 통해 각 원소를 참조한다는 점에서 배열과 다르다. 배열의 모든 원소는 동질형(homogeneous type)이고 첨자 를 통해 참조된다. 공용체형(union type)은 합집합 연산에 대응한다. C 언어에서는 다음과 같이 공용체를 사용할 수 있다.

```
union
{ double r;
  int i;
}
```

- 포인터형은 다른 형의 값을 가리키는 값으로 이뤄져 있다. 즉 포인터형의 값은 기본이 되는 형의 값이 저장된 메모리 위치의 주소가 된다.

7.4 형 검사

- **2. 형 시스템**
  - 형 시스템은 형 표현식을 프로그램의 여러 부분에 할당하는 규칙의 모임을 말한다. 형 검사기는 이러한 형 시스템을 구현한 것이다.
  - 컴파일러가 수행하는 검사는 정적인 반면에 목적 프로그램이 수행될 때 행해지는 검사는 동적이라고 한다. 정적 검사에는 **형 검사, 제어 흐름 검사(flow of control check), 유일성 검사(uniqueness check), 이름에 관계된 검사(name related check)** 등이 있다.
    - 흐름 제어 검사는 제어를 이동시키는 흐름을 검사하는 것이고, 유일성 검사는 어떤 객체가 정확하게 한 번만 정의되었는지를 검사하는 것이다. 또한 같은 이름이 두 번 이상 나타날 수 있는데 이를 검사하는 것이 이름에 관계 된 검사이다. 여기서는 형 검사에 대해서만 살펴보자.
  - 견고한 형 시스템(sound type system)은 형 에러를 동적으로 검사하지 않아도 되는 형 시스템을 말 한다. 만약 프로그램이 형 에러 없이 실행된다는 것을 컴파일러가 보장할 수 있다면 이 언어 의 구현은 강 타입(strongly typed)이라고 한다.

7.4 형 검사

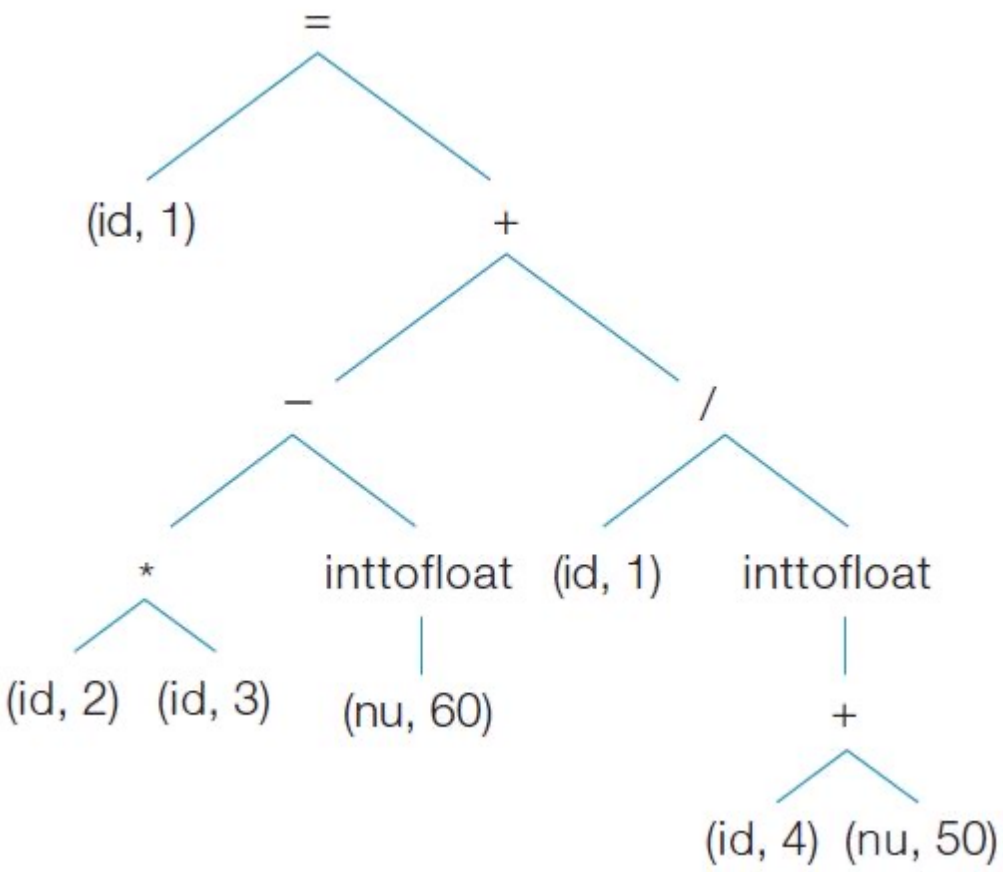
3. 형 변환

- 연산은 두 가지 형태로 구분된다. 첫 번째로 형 **고정 연산(type specific operation)**은 피연산자에 따라 연산 결과의 형이 고정되어 있는 연산이다. 파스칼 언어에서는 형 고정 연산을 허용하므로 피연산자의 형이 다르면 바로 에러 메시지를 내보낸다. 두 번째로 **일반적 연산(generic operation)**은 피연산자와 연산 결과의 형이 변할 수 있는 연산을 말한다. C 언어에서는 일반적 연산을 허용한다. 그렇다면 일반적 연산의 형 변환에 대해 살펴보자.
- x + i와 같은 수식을 고려해보자. 여기서 x는 실수형이고 i는 정수형이다. 정수와 실수는 컴퓨터 내부 표현이 다르고 정수 연산과 실수 연산에는 서로 다른 기계 명령어가 사용되기 때문에 컴파일러는 +의 피연산자 중에 하나를 변환하여 덧셈의 피연산자가 동일한 형이 되도록 한다. 형 변환 규칙은 언어마다 다양하다. 자바의 규칙은 정보를 보존하는 확장(widening) 변환과 정보를 잃을 수 있는 축소(narrowing) 변환을 구별한다. 확장 변환 규칙에서는 계층이 더 낮은 형이 더 높은 형으로 확장될 수 있다. 따라서 char는 int 또는 float로 확장될 수 있지만 short로는 확장될 수 없다. 그리고 축소 변환 규칙에서는 char, short, byte가 각각 서로에 대해 축소 변환이 가능하다.
- 하나의 형에서 다른 형으로의 변환을 컴파일러가 자동적으로 수행한다면 이 변환은 묵시적(implicit)이다. 묵시적 형 변환은 **강제 형 변환(coercion)**이라고도 하며 많은 언어의 확장 변환에서만 허용한다. 형 변환을 하기 위해 프로그래머가 무언가를 명백하게 서술해야 한다면 이 변환은 명시적(explicit)이다. 명시적 형 변환은 **캐스트(cast)**라고도 한다.

7.4 형 검사

- **[예제 7-4] 치환문의 형 검사하기**
  - 2장의 [예제 2-3]에서 의미 분석을 통해 형 변환을 했다. [예제 2-3]의 치환문 ni = ba * po - 60 + ni / (abc + 50);에서 ni, ba, po는 실수형, abc는 정수형이라고 하자. 일반적 연산을 허용하는 언어이며, 정수형과 실수형의 연산은 정수형을 실수형으로 변환하여 실수형 결과를 나타낸다고 가정하고 형 검사를 해보자
  - [풀이]
  - 첫 번째로 연산 ba * po는 자료형이 같기 때문에 문제가 없다. 다음으로 ba * po의 결과인 실수형과 정수형 60의 연산은 자료형이 다르기 때문에 정수형인 60을 실수형으로 변환해야 한다. abc와 50은 정수형이므로 연산을 한다. 그 다음에 ni와 (abc + 50)의 결과를 연산하는데, 자료형이 맞지 않으므로 정수형인 (abc + 50)을 실수형으로 변환하고 연산을 수행한다. 결과는 다음과 같다

7.4 형 검사



7.4 형 검사

- 형 검사와 관련된 중복 정의(overloading)와 다형형(polymorphic type)에 대해 간단히 살펴보자.
- 같은 연산자 이름이 2개의 다른 연산에 사용된 경우에 이 연산자는 중복 정의되었다고 한다. 중복 정의의 예로는 산술 연산자를 들 수 있다. 일반적으로 산술 연산자는 다른 종류의 수치 값에 대한 연산을 나타내는데, 예를 들어 2 + 3은 정수의 덧셈이지만 2.0 + 3.0은 부동 소수점의 덧셈이다. 이러한 덧셈은 내부적으로 다른 명령어로 구현되어야 한다.
- 한편 다형형이란 하나의 언어 구문에서 2개 이상의 형을 허용하는 것이다.