□ 기술해설 □

다중 패러다임 프로그래밍과 Leda 언어

동아대학교 김명호*

● 목 차 ●

1. 서 론

패러다임(paradigm)이란 문제를 이해하고 지식을 구성하기 위한 모든 이론, 표준 및 기법을 의미하며[1], 프로그래밍 패러다임은 컴퓨터에서 계산 수행의 의미와 실행될 작업들의 구성 방법에 관한 개념적인 수단을 의미한다[2].

언어학자들의 연구 결과에 의하면 에스키모 언어에는 눈을 의미하는 단어가 많이 있는 반면 아라비아 언어에는 낙타를 의미하는 단어가 수 없이 많이 있다. 이는 어떤 현상이나 문제를 이해하는 관점(패러다임)과 표현 수단 사이에는 극히 밀접한 관계가 있음을 증명하는 것이다.

이와 같은 현상은 프로그래밍에서도 자주 발견되는데 특정 패러다임을 지원하는 언어에 익숙할수록 문제를 이해하는 사고방식도 그 패러다임에 치우치게 된다. 예를 들어 Pascal과 같이 배정문, 선택문과 반복문이 자연스럽게 표현되는 언어에 익숙한 프로그래머는 어떤 문제를 만나더라도 배정과 선택, 반복의 조합으로 접근하는 경향이 있는 반면 APL과 같이 정렬이 대단히 쉬운 언어에 익숙한 프로그래머는 동일한 문제를 해결하더라도 정렬의 변형으로 주어진 문제를 이해하고 표현하려는 경향이 있다. 그러나 적용하고자 하는 패러다임이 해결하고자 하는 문제에 적합하지 않는 경우 프로그래밍이 불편하거나 작성된 프로그램이 극히 비효율적일 가능성이 높게 된다.

단일 패러다임만 사용하는 경우의 문제점을 해결하기 위하여 다중 패러다임 프로그래밍이 제안되었다[3]. 다중 패러다임 프로그래밍은 문제 해결을 위한 다양한 개념적 도구를 제공하여 임의의 문제를 해결할 때 하나의 사고를 강요하기 보다는 부분적인 문제들에 대하여 적합한 패러다임을 적용하고 이들을 결합하여 전체 문제를 효과적으로 해결하도록 하기 위한 것이다.

본 논문은 다중 패러다임 프로그래밍을 지원하는 언어에 관하여 고찰하기 위한 것이다. 먼저 대표적인 프로그래밍 패러다임인 명령형, 함수형, 논리형 및 객체지향 패러다임의 개념에 대하여 기술하며, 이들을 결합한 다중 패러

*비회원

다임 프로그래밍언어에 관하여 설명한다. 본 논문에서는 특히 Leda 언어[4]를 사용하여 다중 패러다임 프로그래밍의 실제적인 사례를 보였다. Leda는 명령형, 객체지향, 함수형 및 논리형 패러다임을 효과적으로 표현할 수 있으면서도 이들의 자유로운 결합이 가능하도록 설계된 대표적인 다중 패러다임 프로그래밍언어이다. Leda는 단순하고 작은 규모의 언어이면서도 다중 패러다임 특성으로 인하여 많은 문제에 대하여 효과적이고도 우아한 해결을 가능하게 하는 특징이 있다.

2. 패러다임과 프로그래밍언어

프로그래밍 문제를 해결하기 위한 패러다임은 다수 있지만 비교적 넓은 부류의 문제들에 대하여 효과적으로 적용할 수 있는 대표적인 패러다임으로는 명령형, 객체지향, 함수형, 논리형 패러다임을 들 수 있다. 여기에서는 각 패러다임에 대한 간략한 소개와 함께 factorial 계산을 위한 프로그램 작성의 예를 보인다.

2.1 명령형(imperative) 패러다임

명령형 패러다임은 von Neumann 구조의 컴퓨터에서 처리장치가 기억장소(상태)를 변경하는 것을 계산 수행의 모델로 사용한다. 그러므로 명령형 언어에서 계산 작업을 구성하는 수단은 상태 변경을 위한 명령이며 변수, 배정문, 선택이나 반복과 같은 제어구조가 기본적으로 제공된다.

다음은 factorial을 계산하기 위한 프로그램을 명령형 패러다임을 지원하는 대표적인 언어인 Pascal[5]로 작성한 것이다.

```
{writeln(factorial(5));}
function factorial(n : integer) : integer;
var i, result : integer;
begin
  result := 1;(* 배정문 *)
  for i := 1 to n do result := result * i;(* 반복문 *)
  factorial := result;
end (* factorial *);
```

명령형 프로그래밍은 하드웨어의 동작과 유사한 방법으로 프로그램을 표현하므로 효율성을 쉽게 얻을 수 있으나 프로그램의 실행 상태가 명령에 의해 동적으로 변경되어 이해를 어렵게 하고, 병행 수행이 어려우며, 문제의 복잡도를 해결하는 수단으로는 너무 저수준의 사고방식이라는 문제점이 있다.

명령형 패러다임을 지원하는 언어로는 Fortran, Cobol. Basic, Pascal, C, Ada 등을 들 수 있다.

2.2 객체지향(object-oriented) 패러다임

객체지향 패러다임은 객체들 사이에 메시지를 주고 받는 것을 계산 수행의 모델로 사용하며, 프로그램의 각 요소가 전체 프로그램의 구조와 기능을 모방하도록 하는 순환적 설계(recursive-design)를 효과적으로 지원한다.

객체지향 언어는 객체, 메시지(와 동적 바인딩), 상속성과 다형성 등을 기본적으로 제공한다[6].

다음은 factorial을 계산하기 위한 프로그램을 객체지향 패러다임을 지원하는 대표적인 언어인 Smalltalk[7]로 작성한 것이다.

```
"' (Factorial new ' 5) value' results in 120"
Object subclass : #Factorial
  instanceVariableNames : 'arg'
  classVariableNames : ''
  poolDictionaries : ''!
!Factorial class methods!
new : anArg
  ↑ (self new) arg : anArg! !
!Factorial methods!
arg : anArg
  arg := anArg!
value
  (arg = 0)
    ifTrue : [ ↑ 1 ]
    ifFalse : [ ↑ arg * (Factorial new : (arg-1)) value]! !
```

객체지향 패러다임에서는 동적 바인딩과 다형성을 활용하여 프로그램을 비파괴적으로 확장할 수 있고, 상속성을 사용하여 기존 프로그램의 구현을 효과적으로 재사용할 수 있다. 그러나 상속성의 무분별한 남용은 프로그램의 확장을 심각하게 저해할 수도 있다.

객체지향 패러다임을 지원하는 언어로는 Simula, Smalltalk, Eiffel, Sather, Beta, C++, Objective-C 등이 있다.

2.3 함수형(functional) 패러다임

함수형 패러다임은 함수를 값에 적용하는 것을 계산 수행의 모델로 사용한다. 함수는 상태

의 개념이 없으므로 동일한 함수를 동일한 값에 적용하면 항상 동일한 결과를 산출하는 특징(referential transparency)을 만족해야 한다. 이를 위하여 순수한 함수형 언어에서는 변수와 문장이 전혀 제공되지 않으며 식 만을 사용하여 프로그램을 작성한다. 함수형 언어에서는 함수의 일등급(first-class) 특성과 고차함수(higher-order function), 지연계산(lazy evaluation) 등으로 인하여 명령형 언어에 비해 보다 간결하면서도 이해도가 높은 프로그램을 용이하게 개발할 수 있다[8, 9].

다음은 factorial을 계산하기 위한 프로그램을 함수형 패러다임을 지원하는 대표적인 언어인 Haskell[10]로 작성한 것이다.

```
-- factorial 5 => 120
factorial n = foldr (*) 1 [1..n]
```

이처럼 간단한 표현이 가능한 이유는(데이타형 조건을 만족하는) 임의의 이항함수(예에서는 정수 곱셈을 위한 함수 "*")를 foldr 함수의 인수로 전달할 수 있기 때문이다. 리스트의 모든 원소와 전달된 어떤 값에 대하여 이항함수를 모두 적용한 결과를 얻기 위한 표준 고차함수 foldr은 사용자에 의해 다음과 같이 직접 작성될 수도 있다.

```
foldr f init [] = init
foldr f init (x : xs) = f x (foldr f init xs)
```

함수형 패러다임은 수학적인 특성을 만족하는 함수만 사용하므로 프로그램의 검증이 용이하며, 고차함수와 지연계산을 활용하여 재사용성이 극히 높은 프로그램 개발을 가능하게 한다.

함수형 패러다임을 지원하는 언어로는 Lisp, Scheme, FP, SML, Sasl/KRC/Miranda, Haskell 등을 들 수 있다.

2.4 논리형(logic) 패러다임

논리형 패러다임은 정리의 증명(theorem proving)을 계산 수행의 모델로 사용한다. 프로그램은 공리(사실과 규칙)로 구성되며, 프로그램의 실행은 미리 정해진 추론규칙을 사용하여 질의(정리)를 증명하는 과정에 대응한다. 논리형 언어는 선언적이고 비절차적인 특징이 있어서 문제 해결의 방법보다는 문제의 의도만 기술하여도 프로그램 개발이 가능한 경우가 많이 있다[11].

다음은 factorial을 계산하기 위한 프로그램을 논리형 패러다임을 지원하는 대표적인 언어인 Prolog[12]로 작성한 것이다.

```
% ?- factorial(5, R). => (R = 120, yes)
factorial(0, 1) :- !.
factorial(N, R) :- N1 is N-1, factorial(N1, R1), R is N*R1
```

논리형 언어의 수행 모델인 정리 증명은 탐색과 유사하므로 인공지능에서의 탐색 문제나 연역 데이타베이스 등에 활용될 수 있다. 논리형 언어의 선언적, 비절차적 특성은 프로그래밍을 쉽게 하는 요소이기도 하지만 효율적인 프로그램의 작성을 대단히 어렵게 하는 요인이 되기도 한다.

논리형 패러다임을 지원하는 언어로는 Prolog, CLP(Σ^*), Godel 등을 들 수 있다.

2.5 기타 패러다임들

지금까지 소개한 패러다임 이외에도 데이타 흐름, 액세스 지향, 제약조건형, 병렬, 시각 프로그래밍 등의 다양한 패러다임이 널리 알려져 있다.

그러나 이들은 앞서 소개한 4대 패러다임 만큼 사고의 차이로 보기는 어려우며, 이들의 변형이거나 보완하기 위한 목적으로 파생된 패러다임으로 간주할 수 있다.

3. 다중 패러다임과 프로그래밍언어

하나의 문제를 해결하기 위하여 다양한 패러다임을 사용하는 것을 다중 패러다임 프로그래밍이라고 하며, 여러개의 패러다임을 동시에 지원하는 언어를 다중 패러다임 언어라고 한다. 여기에서는 다중 패러다임 프로그래밍과 언어와의 관계를 설명하고, 대표적인 다중 패러다임 언어 Leda를 소개한다.

3.1 다중 패러다임 프로그래밍

수학자가 어떤 추측이나 정리를 증명하고자 할 때는 한가지 수단에만 의존하지 않고 대수, 기하, 논리, 위상 등의 다양한 수단에서 최적의 조합을 사용한다. 즉 정리 증명의 관점에서 본

수학자의 능력은 하나의 수단을 모든 정리 증명에 억지로 적용하기 보다는 여러 가지의 수단을 적절히 선택하여 부분 정리들을 증명하고, 전체 정리의 증명을 유도해 낼 수 있는 능력에 달려 있다. 여기에서 수학자를 프로그래머에, 정리 증명을 문제 해결을 위한 프로그래밍에, 정리 증명의 각 수단을 패러다임에 대응시켜 보면 다중 패러다임의 중요성은 쉽게 이해할 수 있을 것이다.

복잡한 문제를 해결하기 위한 프로그램을 작성하는 경우 부분 문제들에 대하여 적절한 패러다임을 적용하여 효과적으로 해결하고 이들을 결합하여 전체 프로그램을 완성하는 기법은 다중 패러다임 프로그래밍이 지원하는 대표적인 특징이다.

3.2 다중 패러다임 프로그래밍언어

다중 패러다임 프로그래밍을 지원하기 위한 기존의 연구는 어떤 언어로 작성된 부분 프로그램을 다른 언어로 작성된 부분들과 결합하기 위한 방법에 집중되어 있었다[13,14]. 이에 반해 다중 패러다임 언어는 단일 언어를 사용하여 다중 패러다임 프로그래밍을 적극적으로 지원하기 위한 것이다. 다중 패러다임 언어는 각 패러다임을 위한 기능을 무원칙하게 조합하기 보다는 유기적으로 결합하여 사용할 수 있도록 함으로써 "전체는 부분의 합보다 크다"[1]는 목적을 만족시킬 수 있다.

다중 패러다임 언어의 설계는 단일 패러다임만 지원하는 언어의 설계에 비해 훨씬 어려운 작업이다. 각 패러다임을 지원하는 요소는 작고도 필수적인 기능을 모두 제공하여야 하며, 이들이 빈틈없이 결합될 수 있도록 하여야 한다. 프로그램의 편이성 뿐만 아니라 적절한 효율성을 제공하는 구현이 가능해야 한다. 더구나 모든 유용한 패러다임과 이들의 자유로운 결합을 효과적으로 활용할 수 있도록 프로그래머를 교육하는 것은 보다 근본적인 문제이다.

1) 이 문장은 "M개의 요소를 가진 어떤 차원과 N개의 요소를 가진 다른 차원이 자유롭게 결합될 수 있으면 M×N개의 요소를 가진 차원을 직접 제공하는 것과 같은 효과를 얻을 수 있다"는 개념인 언어의 직교성(orthogonality)을 설명할 때 자주 사용된다.

이 모든 문제점에도 불구하고 다중 패러다임 언어를 개발하기 위한 시도는 여러 차례 있었다[15]. 대부분의 다중 패러다임 언어는 기본적으로 지원하는 패러다임 위에 다른 패러다임을 추가하는 방법으로 설계되었다. Leda[4]와 G[16]는 4대 패러다임을 모두 지원하는 진정한 의미에서의 다중 패러다임 언어이다.

3.3 다중 패러다임 언어 Leda

Leda는 Oregon State University의 Budd 교수에 의해 설계되고 다수의 학생들에 의해 구현된 다중 패러다임 언어이다. Leda는 특정 패러다임에 치우치지 않게 명령형, 객체지향, 함수형, 논리형 프로그래밍을 모두 지원하도록 설계되었다. Leda는 유연성과 표현력을 제공하면서도 매우 작은 규모의 언어로 설계되었으며, 문법적으로도 Pascal에 비해 그다지 복잡하지 않다. Leda를 사용하여 다중 패러다임을 교육하기 위한 교재도 개발된 바 있다[17].

명령형 패러다임을 위한 Leda의 기능은 Pascal의 경우와 유사하여 변수, 배정문, 제어문, 입출력문이 제공된다. 다음은 factorial 문제를 위한 명령형 Leda 프로그램이다.

```
{print(factorial(5));}
function factorial(n : integer)->integer;
var i, result : integer;
begin
  result := 1;
  for i := 1 to n do result := result * i;
  return result;
end;
```

객체지향 패러다임을 위한 Leda의 기능은 Smalltalk와 유사하게 모든 메소드가 동적 바인딩이 수행되고, 모든 객체가 Object라는 클래스에서 파생된 클래스의 객체로 간주되며, 포인터 의미론을 따르지만, 정적 형검사를 사용하는 점에서는 Smalltalk와 다르다. 다음은 factorial 문제를 위한 객체지향 Leda 프로그램이다.

```
{Factorial(5).value().print();}
class Factorial;
var arg : integer;
  function value()->integer;
  begin
    if arg = 0 then return 1;
    return arg * Factorial(arg-1).value();
  end;
end;
```

함수형 패러다임을 위한 Leda의 기능은 무한 수명을 가지는 일등급 프로시듀어(first-class closure)와 고차함수, 지연 계산 등을 제공한다. 다음은 factorial 문제를 위한 함수형 Leda 프로그램이다.

```
function foldr[X:object, Y:object](aList:List[X], ident:Y, f:
function(X,Y)->Y)->Y;
begin
  if empty[X](aList) then return ident;
  return f(head[X](aList), foldr[X, Y](tail[X](aList), ident,
f));
end;
{print(factorial(5));}
function factorial(n: integer)->integer;
begin
  return foldr[integer, integer](integer.times, 1, enum_integers
(1,n));
end;
```

예제에서의 foldr 함수는 Haskell의 경우와 마찬가지로 factorial 함수의 정의 뿐만 아니라 리스트의 모든 원소들을 어떤 기준 값과 이항 함수로 모두 결합하기 위한 목적으로 재사용할 수 있다.

논리형 패러다임을 위한 Leda의 기능은 논리형 함수 관계(relation), 논리적 배정문(<-), 단일화(unification), & 및 | 연산자, backtracking을 사용한 탐색 등을 들 수 있다. 다음은 factorial 문제를 위한 논리형 Leda 프로그램이다.

```
{ if (factorial(5, fr)) then print(fr); }
function factorial(n  integer, byRef result: integer)->relation,
var n1, r1  integer;
begin
  return (n = 0) & result <- 1 |
         (n > 0) & n1 <- (n-1) & factorial(n1, r1) & result
<- n*r1;
end;
```

Leda에서는 각 패러다임을 위한 기능의 임의의 조합이 가능하여 클래스의 메소드가 함수를 인수로 하거나 관계를 멤버로 가질 수 있고, 함수가 관계를 인수로 하여 임의의 결과(심지어는 관계)를 산출할 수 있으며, 관계가 함수를 인수로 하거나 명령형에 준하는 형식으로 구현될 수도 있다. 그 결과 앞서 예시한 논리형 factorial 프로그램을 다음과 같이 작성할 수도 있다.

```
function factorial(n: integer, byRef result: integer)->relation;
var r1: integer;
begin
  if (n = 0) then
    return result <- 1
  else if ((n > 0) & factorial(n-1, r1)) then
  return result <- n*r1
end;
```

이러한 기법을 작은 문제에 적용하는 것은 프로그래머의 취향에 따른 선택의 문제일 뿐이지만, 복잡한 문제의 이해와 해결을 위해서는 대단히 강력한 수단으로 작용할 수 있다.

4. Leda를 사용한 다중 패러다임 프로그래밍

다중 패러다임 프로그래밍언어는 복잡한 문제를 명령형 부분, 객체지향 부분, 함수형 부분, 논리형 부분 등으로 분해하는 높은 수준의 추상화뿐만 아니라, 개별적인 단위 프로그램의 구현에 이르기까지의 모든 과정을 지원할 수 있다. 여기에서는 Leda 언어를 사용하여 서로 다른 패러다임을 조화롭게 사용하여 추상적 데이타 형을 설계하고 구현하는 예를 보인다.

4.1 그래프 데이타 추상화

그래프는 노드의 집합과 노드들을 연결하는 에지의 집합으로 구성된 데이타 형이다. 그래프의 정의에 집합이 사용되므로 먼저 집합을 위한 데이타 추상화를 다음과 같이 정의하였다.

```
class Set[X: equality];
function plus(val: X)->Set[X]; .  {overloads "+"}
function minus(val: X)->Set[X]; .  {overloads "-"}
function includes(val: X)->boolean; ..
function items(byRef val: X)->relation; .
end;
```

예시한 프로그램 설계에서 집합 데이타 형의 전체적인 구조는 객체지향적 클래스로 구성되었으며, 원소 추가, 삭제, 조회를 위한 멤버 plus, minus, includes 등은 함수형으로[2], 원소 열거를 위한 멤버 items는 논리형으로 정의되었다. 이와 같이 단일 문제의 해결에 서로

2) 이 경우 멤버의 형태만으로는 함수형이라고 단정할 수 없지만 집합의 상태를 직접 변경하는 대신 변경된 집합을 결과치로 하는 함수로 정의한 것은 함수형 프로그래밍의 사고를 반영한 것이다.

다른 패러다임을 혼용할 수 있는 것이 Leda 언어의 주된 특징이다.

집합 데이타 형 사용자의 관점에서 보면 객체에 메시지를 전달함으로써 필요한 연산을 수행하는 객체지향적 사고와, 각 메시지를 위한 인터페이스가 요구하는 함수형, 논리형을 활용하여 각 패러다임의 장점을 쉽게 얻을 수 있다. 다음은 집합 데이타 추상화를 사용하는 예를 보인 것이다.

```
var s: Set[integer]; i: integer;
begin
  ...
  s := Set[integer]() + 1 + 2 + 3 + 4 + 5;
  s := s - 4;
  for i:items(s) do print(i), { results in 1, 2, 3, 5 }
end;
```

에에서 s를 객체(변수)로 선언하여 상태 변경(배정문)의 대상으로 사용한 것은 객체지향과 명령형 사고이다. 집합에 새로운 원소를 추가하는 과정을 "Set[integer]() + 1 + ..."와 같은 간결한 식으로 표현한 것은 함수형 사고이다. 한편 "for s.items(i) do ..."와 같은 탐색 기능을 사용하여 모든 원소를 열거한 것은 논리형 사고이다.

집합 데이타 추상화를 사용하여 그래프 데이타 추상화를 다음과 같이 정의할 수 있다.

```
class Graph;
  function addNode(v: integer)->Graph;
  function addEdge(e: Edge)->Graph;
  function removeNode(val: integer)->Graph;
  function removeEdge(e: Edge)->Graph;
  function includesNode(val:integer)->boolean;
  function includesEdge(e: Edge)->boolean;
  function node(byRef val: integer)->relation;
  function edge(byRef tail: integer, byRef head: integer)->
relation;
  end;
```

그래프도 집합의 경우와 유사하게 전체적인 구조는 객체지향적이며, 각각의 멤버들은 함수형, 혹은 논리형으로 정의되었다.

4.2 그래프 데이타 추상화의 구현

제3장에서 소개한 바 있는 Leda 언어의 4대 패러다임 지원 기능을 활용하여 그래프 데이타 추상화를 구현할 수 있다. 여기서는 대표적인 멤버들의 구현만 소개하기로 한다.

먼저 그래프의 구현에 사용되는 집합 데이타 추상화를 위한 데이타 구조는 리스트 구조와 유사한 방법으로 정의할 수 있다.

```
class Set[X equality],
  var value: X; next: Set[X];
  ... member functions are defined here ...
end;
```

집합에 새로운 원소를 추가하기 위한 멤버함수 Set[X].plus는 다음과 같이 구현할 수 있다.

```
function plus(val: X)->Set[X];
begin
if self.includes(val) then return self;
return Set[X](val, self);
end;
```

모든 원소의 열거를 위한 멤버 Set[X].items는 논리형 기능을 사용하여 효과적으로 구현할 수 있다.

```
function items(byRef val: X)->relation;
begin
  return unify[X](val, value) , defined(next) & next.items
(val);
end;
```

멤버 Set[X].items는 모든 원소의 열거 뿐만 아니라 특정한 값이 집합의 원소인가를 조사하기 위한 목적으로 사용될 수도 있다. 그러므로 이를 활용하여 Set[X].includes를 다음과 같이 구현할 수 있다.

```
function includes(val: X)->boolean; begin return self.items
(X) end;
```

그래프 데이타 추상화의 구현도 집합의 경우와 유사하다. 그래프 데이타 추상화를 위한 데이타 구조는 다음과 같다.

```
class Graph;
  var nodeSet: Set[integer], edgeSet: Set[Edge];
  ... member functions are defined here ...
end;
```

그래프에 새로운 에지를 추가하기 위한 멤버 Graph.addEdge는 집합 연산이 함수형으로 정의된 점을 활용하여 다음과 같이 간결하게 구현할 수 있다.[3]

```
function addEdge(e: Edge)->Graph;
begin
```

3) 여기에서는 어떤 에지가 그래프에 추가될 수 있으려면 시작과 끝을 의미하는 노드가 그래프에 이미 정의되어 있어야 하는 것으로 약속하였다.

```
    if includesNode(e.tail) & includesNode(e.head) then
      return Graph(nodeSet, edgeSet+e)
    return self;
  end;
```

그래프에서 노드를 제거할 때는 연관된 모든 에지까지 함께 제거하는 것으로 약속할 수 있다. 이 경우 어떤 노드를 제거하면 새로운 에지 집합을 생성한 후, 원래 그래프의 에지 집합 순회하여 Graph.addEdge를 호출하여 결과 그래프의 에지 집합을 얻을 수 있다. 다음은 이와 같은 아이디어를 사용하여 Graph.removeNode를 구현한 것이다.

```
function removeNode(val: integer)->Graph;
var g: Graph; e. Edge;
begin
  g := Graph(nodeSet-val, NIL);
  if defined(edgeSet) then
    for edgeSet.items(e) do g := g.addEdge(e);
  return g;
end.
```

주어진 구현에서는 edgeSet.items를 호출하여 모든 에지의 열거를 극히 단순하게 표현할 수 있었다.

그래프에서도 모든 노드와 에지를 열거할 수 있는 기능을 논리형 멤버로 정의하였다. Graph.edge를 위한 구현은 다음과 같다.

```
function edge(byRef tail: integer, byRef head. integer)->re
lation;
var e: Edge;
begin
  return defined(edgeSet) & edgeSet.items(e) &
    unify[integer](tail, e.tail) & unify[integer](head, e.head);
end;
```

4.3 그래프를 응용한 예제 프로그램

Leda가 제공하는 다중 패러다임 프로그래밍 기능은 그래프 데이타 추상화의 구현 뿐만 아니라 그래프를 위한 응용 프로그램 개발에도 활용될 수 있다. 예를 들어 그래프에서 어떤 노드에 도달하는 에지의 갯수를 구하기 위한 함수 inDegree는 명령형과 논리형 기능을 혼용하여 다음과 같이 간결하게 작성할 수 있다.

```
function inDegree(g. Graph, aNode: integer)->integer;
var count: integer,
begin
  count := 0,
  for g.edge(NIL, aNode) do count := count + 1:
  return count;
end,
```

그래프에서 두 노드 사이에 경로가 존재하는지의 여부를 검사하거나, 경로를 형성하는 모든 노드를 열거하기 위한 관계 path는 다음과 같이 작성할 수 있다.

```
function path(g: Graph, byRef i: integer, byRef j: integer)-
>relation;
var k: integer;
begin
  return g.node(i) & g.node(j) &
    ((i = j) | g.edge(i, j) | (g.edge(i, k) & g.removeNode(i).
path(k, j)));
end;
```

주어진 예에서는 논리형 멤버 Graph.node와 Graph.edge를 사용하여 주어진 그래프의 모든 노드와 에지를 열거하고, 함수형 멤버 Graph.removeNode를 사용하여 식을 작성하게 함으로써 간결하고도 이해도가 높은 구현을 유도할 수 있었다.

5. 결 론

보편적으로 사용되는 명령형, 객체지향, 함수형 및 논리형 패러다임 등은 적절히 사용되기만 하면 간결하고, 이해도가 높으며, 효율적인 프로그램을 유도해 낼 수 있다. 그러나 적용하고자 하는 패러다임이 문제 유형에 적절하지 않은 경우 프로그래밍 작업이 극히 어렵거나 비효율적인 프로그램이 되기 쉽다.

다중 패러다임 프로그래밍은 복잡한 문제를 구성하는 부분 문제들에 대하여 서로 다른 패러다임을 적용하고 이들을 결합하여 전체 문제를 해결하는 기법으로, 단일 패러다임만을 사용하여 전체 문제를 해결하는 경우에 비해 다양한 사고의 수단을 제공한다. 다중 패러다임 언어는 단일 언어를 사용하여 다중 패러다임 프로그래밍을 적극적으로 지원하는 언어이다.

다중 패러다임 프로그래밍 기법, 다중 패러다임 언어 설계 및 구현 등에 관한 연구는 아직 초기 단계에 있지만 단일 패러다임만 사용하는 경우의 문제점과 다중 패러다임의 상대적인 장점이 인식될수록 관심과 수요가 점차 늘어날 것으로 전망된다. 특히 Leda나 G와 같이 작고 효율적이면서도 여러 패러다임을 유기적으로 결합하여 강력한 표현력을 제공하는 언어의 출현에서 다중 패러다임 프로그래밍의 밝은

미래를 충분히 짐작할 수 있다.

감사의 글

본 논문을 위하여 Leda 인터프리터와 모든 예제 프로그램의 사용을 허락해 주신 Oregon State University의 Timothy A. Budd 교수님께 진심으로 감사를 드립니다.

참고문헌

[1] T.S. Kuhn, The Structure of Scientific Revolutions, Univ. of Chicago Press, 1962.

[2] R.W. Floyd, "The Paradigms of Programming," CACM, 22(8), pp.455-460, 1979.

[3] J. Placer, "Multiparadigm Research: A New Direction in Language Design," ACM SIGPLAN Notices, 26 (3), pp.9-17, 1991.

[4] R. Pandey, W. Pesch, J. Shur and M. Takikawa, "A Revised Leda Language Definition," Technical Report 93-60-02, Oregon State University, 1993.

[5] K. Jensen and N. Wirth, Pascal User Manual and Report (3rd ed.), ISO Pascal Standard, Springer-Verlag, 1985.

[6] T. Budd, An Introduction to Object-Oriented Programming, Addison-Wesley, 1991.

[7] A. Goldberg and D. Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, 1983.

[8] J. Backus, "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," CACM, 21(8), pp.613-641, 1978.

[9] P. Hudak, "Conception, Evolution, and Application of Functional Programming Languages," ACM Computing Surveys, 21(3), pp.359-411, 1989.

[10] P. Hudak and J. Fasel, "A Gentle Introduction to Haskell," ACM SIGPLAN Notices, 27(5), pp.1-53, 1992.

[11] C.J. Hogger, Introduction to Logic Programming, Academic Press, 1984.

[12] W. Clocksin and C. Mellish, Programming in Prolog (3rd ed.), Springer-Verlag, 1987.

[13] B. Hailpern, "Multiparadigm Languages and Environments," IEEE Software, 3(1), pp.6-9, 1986.

[14] R. Hayes and R.D. Schlicting, "Facilitating Mixed Language Programming in Distributed Systems," IEEE Trans. on Software Engineering, 13(12), pp. 1254-1264, 1987.

[15] B. Kinnersley (ed.), The Language List, available over WWW, http://cui_www.unige.ch/langlist, 1995.

[16] J. Placer, "The Multiparadigm Language G," Computer Language, 16 (3), pp.235-258, 1991.

[17] T.A. Budd, Multiparadigm Programming in Leda, Addison-Wesley, 1995.

김 명 호

1984 경북대학교 컴퓨터공학과 졸업(공학사)
1986 한국과학기술원 전산학과 졸업(공학석사)
1989 한국과학기술원 전산학과 졸업(공학박사)
1989~현재 동아대학교 컴퓨터공학과 부교수
관심분야 : 프로그래밍언어(프로그래밍 패러다임, 다중 패러디임 프로그래밍, 정형적 의미론)