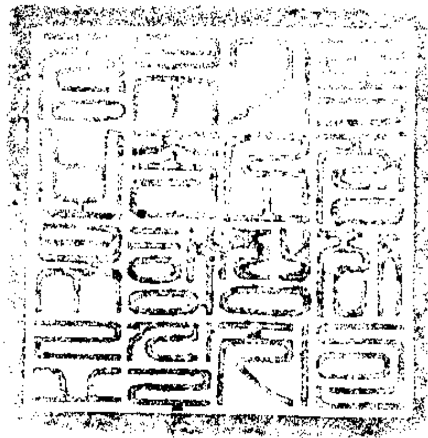
'89국책연구개발사업과제
"인공지능기술"사업의
세부과제

156

복합형 인공지능 개발시스템에 관한 연구

A Study on Hybrid Artificial Intelligence Development System

연구기관
한 국 과 학 기 술 원



과 학 기 술 처

제 출 문

과학기술처장관 귀하

본 보고서를 "인공지능 기술" 사업의 세부과제 "복합형 인공지능 개발시스템에 관한 연구" 사업의 최종 보고서로 제출합니다.

1990. 7

주관 연구기관 : 한 국 과 학 기 술 원

연 구 기 관 : 한 국 과 학 기 술 원

총괄 책임자 : 김 진형 (한국과학기술원 전산학과 부교수)

연 구 원 : 이 종만 (한국과학기술원 전산학과 박사과정)

송 종수 (한국과학기술원 전산학과 박사과정)

김 석원 (한국과학기술원 전산학과 박사과정)

조 원규 (한국과학기술원 전산학과 박사과정)

연 구 조 원 : 임 선배 (한국과학기술원 전산학과 석사과정)

이 동현 (한국과학기술원 전산학과 석사과정)

요 약 문

I. 제목

복합형 인공지능 개발시스템에 관한 연구

II. 연구개발의 목적 및 중요성

전문가 시스템과 같은 지식기반 시스템을 개발하는데 드는 시간과 노력을 줄이고 품질 높은 시스템을 개발하기 위해서는 다양한 개발환경을 제공하는 소프트웨어 도구, 즉 인공지능 개발시스템이 필요하다. 초기의 인공지능 개발시스템들은 대부분 하나의 지식표현 방법만을 제공하는 단순한 시스템이었으나 인공지능 시스템이 점점 복잡하여짐에 따라 다양한 지식표현 방법과 강력한 개발환경을 제공하는 복합형 인공지능 개발시스템이 사용되는 추세이다. 국내에서도 인공지능이 소개되면서 여러 기관에서 인공지능 시스템을 개발하거나 이를 계획하고 있다. 하지만 대부분이 이를 위하여 외국에서 상업적으로 개발된 시스템들을 도입하여 사용하고 있는 실정이다. 그런데 이러한 시스템들은 대부분 고가이며, 원시코드 등이 제공되지 않으므로 수정 보완 등이 쉽지 않다. 이러한 상황을 고려할 때 국산 워크스테이션에 적재하여 사용될 수 있는 복합형 인공지능 개발시스템의 개발이 시기적으로 매우 필요하다. 본 연구과제에서는 국내에서 생산되는 워크스테이션에 탑재할 수 있으며 LISP 개발환경, 규칙기반 시스템, 객체지향적 프로그래밍 환경, 한글개발환경 등을 제공하는 복합형 인공지능 개발시스템, HyKET (Hybrid Knowledge Engineering Environment Tool)을 개발한다.

III. 연구개발의 내용 및 범위

1. LISP 개발환경의 개선

- Common LISP에서 Foreign Language Interface 구현
- KCL에서의 Garbage Collection 방법의 개선에 관한 연구
- AKCL과 GNU-Emacs 환경 구축

2. 규칙기반 시스템

- Window를 이용한 KOPS 개발
- LISP과 규칙의 혼용방법 구현
- 규칙 기반 시스템에서의 진리 유지 기능 구현
- 불확실성 처리 기법 구현

3. 객체 지향적 프로그래밍 환경

- CLOS 분석 및 이식
- CLOS full spec.에 따른 개선

4. 사용자 인터페이스

- X Window 환경 구현
- LISP/X Interface
- CLOS Browser 설계

5. HyKET 시스템의 기능 통합

- Common Lisp 환경, 외부 언어 인터페이스
- 규칙 기반 시스템, Common Lisp Object System
- GNU-Emacs, X-Window System

- CLX

6. HyKET을 이용한 Demo System 구현 : PC 고장진단 전문가 시스템

IV. 연구개발 결과 및 활용에 대한 건의

o 연구개발 결과

- Common LISP에서 FLI 구현
- KCL에서의 GC 방법의 개선에 관한 연구
- AKCL과 GNU-Emacs 환경 구축
- Window를 이용한 KOPS 개발
- LISP과 규칙의 혼용방법 구현
- 규칙 기반 시스템에서의 진리 유지 기능 구현
- 불확실성 처리 기법 구현
- CLOS 분석 및 이식
- CLOS full spec.에 따른 개선
- X Window 환경 구현
- LISP/X Interface
- CLOS Browser 설계
- 여러가지 기능을 통합한 HyKET 시스템 구현
- HyKET을 이용한 PC 고장진단 전문가 시스템 개발

1, 2차년도 연구 결과에 이어 3차년도에 위와 같은 시스템들을 완성함으로써 복합형 인공지능 개발시스템 HyKET의 기본 골격과 구성요소를 갖추게 되었다. 이들 연구 결과들을 하나의 시스템으로 통합하여 전체 시스템을 구성하였고, 이를 이용하여 PC 고장진단 전문가 시스템을 구현하였다.

SUMMARY

Expert systems have rapidly evolved into the most visible application of artificial intellegence to the real world problems. Increasing demand of expert systems has led to the development tools for the rapid construction of expert systems.

The early Artificial Intelligence Development Systems have utilized only one knowledge representation methodology, such as frames, rules, or logic programming. As the Artificial Intelligence Systems become more complex, hybrid artificial intellegence development systems are required, which integrate several knowledge representation methodologies into a single system providing multi-paradigm knowledge representation schemes.

In this project, we develop a hybrid tool, called HyKET(Hybrid Knowledge Engineering Tool) integrating a common LISP environment, a rule-based system, an object-oriented knowledge representation system, and user interface features. In the 3rd project year, we have developed and extended the components of HyKET including Foreign Language Interface, integration of LISP and rule system, CLOS object-oriented language, X Window environment, and integrated these components into one system.

CONTENTS

Chapter 1. Introduction 13

Section 1. Introduction to AI Development System 13
Section 2. Structure of HyKET 16

Chapter 2. An Integration of Rule-Based System as an Extension of Common LISP 20

Section 1. Introduction 20
Section 2. Rule-Based System on Lisp Environment 23
Section 3. Analysis of KCL and CLIPS 24
Section 4. Design of KCLIPS 33
Section 5. Implementation and Evaluation of KCLIPS 44
Section 6. Conclusion 58
Reference 60

Chapter 3. A Portable Implementation of Common Lisp Object System .. 73

Section 1. Introduction 73
Section 2. Characteristics of Common Lisp Object System 75
Section 3. Implementation of Common Lisp Object System 94
Section 4. Evaluation and Expansion 123
Section 5. Conclusion 125
Reference 127

Chapter 4. Integration of HyKET System 129

Section 1. Common Lisp Environment 129
Section 2. Foreign Language Interface 130
Section 3. Rule-Based System 132
Section 4. Common Lisp Object System 133
Section 5. GNU Emacs 135
Section 6. X Window System 135
Section 7. CLX 136
Section 8. Demo Package 137

Chapter 5. PC Diagnosis Expert System Using HyKET 141

Section 1. Introduction 141

Section 2. Kinds of Fault Diagnosis Methods 142
Section 3. Structure of PC Diagnosis Expert System 143
Section 4. Development of PC Diagnosis Expert System on HyKET .149
Section 5. Conclusion 149
Reference 151

Chapter 6. Conclusion 152

목 차

제 1 장	서 론	13
	제 1 절 인공지능 개발 시스템의 개요	13
	제 2 절 HyKET의 구조	16
제 2 장	Common Lisp의 확장 개념으로서 Rule-based System의 통합	20
	제 1 절 서론	20
	제 2 절 Lisp 환경하에서의 Rule-based System의 필요성	23
	제 3 절 KCL과 CLIPS의 분석	24
	제 4 절 통합 시스템 KCLIPS의 설계	33
	제 5 절 KCLIPS의 구현 및 시험	44
	제 6 절 결론	58
	참고 문헌	60
제 3 장	이식성을 고려한 Common Lisp Object System의 구현	73
	제 1 절 서론	73
	제 2 절 Common Lisp Object System의 특징	75
	제 3 절 Common Lisp Object System의 구현	94
	제 4 절 평가와 확장	123
	제 5 절 결론	125
	참고 문헌	127
제 4 장	HyKET 시스템의 기능 통합	129
	제 1 절 Common Lisp 환경	129
	제 2 절 외부 언어 인터페이스	130

제 3 절 규칙 기반 시스템 132
제 4 절 Common Lisp Object System 133
제 5 절 GNU-Emacs 135
제 6 절 X-Window System 135
제 7 절 CLX 136
제 8 절 Demo Package 137

제 5 장 HyKET을 이용한 PC 고장진단 전문가 시스템의 구현 141
제 1 절 서론 141
제 2 절 고장진단 방법의 종류 142
제 3 절 PC 고장진단 시스템의 구조 143
제 4 절 HyKET에서의 PC 고장진단 시스템 개발 149
제 5 절 결론 149
참고 문헌 151

제 6 장 결 론 152

제 1 장 서론

기존의 알고리즘으로 해결하기가 어렵던 문제들에 대해 인공지능 기법을 적용하여 할 수 있다는 사실이 입증되었으며, 이중 특정 분야에 대한 문제 해결 시스템으로서 전문가 시스템이 실험실과 현장에서 크게 성공하고 있다. 전문가 시스템은 좁은 분야의 문제를 해결하기 위하여 그 분야 전문가의 경험적 지식을 표현하고, 이를 이용하여 추론과정을 거쳐 원하는 결과를 얻어내는 시스템이다. 전문가 시스템 개발시 그 분야에 대한 전문 지식을 수집, 정리하고 이를 바탕으로 추론할 수 있는 시스템을 개발하는 두가지 작업이 필요하므로 개발 노력과 시간이 많이 걸리는 단점이 있다. 따라서, 일반적인 목적으로 사용할 수 있는 전문가 시스템 개발도구를 만들어 놓은 다음 해결하고자 하는 분야에 대한 전문 지식을 이 개발 도구에 접합시킴으로서 쉽게 전문가 시스템을 개발할 수 있다.

본 장에서는 인공지능 개발 시스템의 개요에 관해 알아보고, 본 과제에서 개발하고 있는 HyKET(Hybrid Knowledge Engineering Tool) 시스템에 대해 설명한다.

제 1 절 인공지능 개발시스템의 개요

인공지능을 이용한 응용분야 중 가장 성공한 분야가 지식기반 시스템이라 불리우는 전문가 시스템이다. 전문가 시스템은 특정 분야의 전문가들이 가지고 있는 전문지식을 컴퓨터내에 구현한 후에 이 지식을 바탕으로 추론하여, 일반 사용자들에게 전문적인 도움을 제공하는 대표적인 인공지능 프로그램으로서, 인간 전문가가 지니는 인간적인 약점을 방지할 수 있을 뿐만 아니라 인간 전문가와

같은 수준의 전문지식과 추론능력을 발휘할 수 있어서 의학진단, 재정계획, 광물 탐사, 법률문제, 고장진단, 조립계획 등 다양한 분야에 걸쳐서 상당한 성공을 거두고 있다. 전문가 시스템은 지식을 컴퓨터에 입력하고 이를 이용하여 추론 과정을 거친 후 주어진 문제를 해결한다는 점에 있어서 지금까지 개발된 기존의 프로그램과는 큰 차이가 있으며 개발방법도 다르다. 전문가 시스템의 특성 중 두드러진 몇가지를 나열하면 다음과 같다. 첫째, 전문가의 경험적인 지식을 기반으로 하여 문제를 해결하여야한다. 전문가 시스템은 일반적인 전문지식뿐 아니라 문제를 해결하는데 중요한 역할을 담당하는 전문가의 경험적인 지식(heuristics)을 이용하여 불충분하고 상호 상충되며, 불확실한 입력자료를 근거로하여 믿을 만하고 타당성 있는 결론을 이끌어낼 수 있다. 이는 기존의 시스템이 알고리즘만을 이용했던 것과 비교될 수 있으며, 따라서 지식표현 시스템이 큰 비중을 차지하게 된다. 둘째, 프로그램 제어가 불확실하다. 기존의 프로그래밍 방식에서는 대개 머리속으로 해당문제의 해를 완전히 구한 뒤에 단순히 연산만을 컴퓨터에 맡기는데 반해 전문가 시스템에서는 문제해결 방식의 결정까지도 시스템 자체에 맡긴다. 세째, 기존의 많은 프로그램들이 수치적 연산을 이용한 방법으로 문제를 푸는데 반하여 전문가 시스템에서는 기호를 이용하여 문제를 표현하고 해결한다. 즉 주어진 문제를 기호를 이용하여 표현하고 컴퓨터에 미리 인식시켜둔 패턴과 비교분석함으로써 결론을 도출한다. 실제로 인간이 실세계의 문제를 해결할 때 이러한 문제 해결방식을 사용한다는 것이 심리적인 연구를 통하여 발견되었다.

이와같이 전문가 시스템은 분명히 기존의 프로그램과 그 성격이 분명히 다르며 이로 인하여 기존의 프로그램 개발방법과는 다른 개발방법을 택하고 있다. 기존의 프로그램은 프로그램을 하기 전에 벌써 그 해결방법을 알고 있는 문제들을 다루고 있다. 즉 미리 알고있는 문제해결 절차나 알고리즘을 프로그램으로 바꾸는 것이다. 따라서 기존의 소프트웨어 개발은 주로 top-down 개발방법을 쓰기

에 적합하다. top-down 개발방법은 프로그램 개발과정 중에서 발생하는 설계의 변경을 최소화 하는데 그 의미가 있다. 하지만 프로그램 개발도중이나 개발된 후 설계상의 오류로 인하여 변경이 필요한 경우가 발생하면 매우 많은 시간과 경비를 낭비하게 된다. 반면에 전문가 시스템을 개발하는 경우에는 top-down 개발방법이 부적합하다. 전문가 시스템을 개발할 때에는 기호, 상호관계, 전략 등을 포함하는 경험적 지식을 시스템내에 표현해야 하며 이러한 개념들을 기존의 PASCAL이나 C와 같은 프로그래밍 언어를 이용하여 표현하는 것은 힘든 일이다. 더우기 전문가 자신도 인간 전문가가 어떻게 결론을 이끌어내는지에 대해 정확하게 알고 있지 못하다. 따라서 전문가 시스템의 개발은 여러번의 반복작업을 통하여 점점 향상된 시스템으로 변화시켜가는 개발방법을 사용해야한다.

전문가 시스템은 전문가의 지식을 간직하고 있는 지식베이스와 이러한 지식을 효과적으로 적용할 수 있는 추론기관을 이용하여 불확실한 사실로부터 결과를 추론할 수 있는 능력을 가지고있다. 특정한 응용분야와 관련되는 전문가의 경험으로부터 얻을 수 있는 사실, 규칙과 이들 간의 상호관계를 포함하고 있는 지식베이스는 기존의 프로그래밍 기법으로는 구축하기가 매우 힘이 들고 개발과정의 어려움만 증가시키게 된다. 따라서 전문가 시스템을 개발할 때 이러한 지식을 어떻게 용이하게 프로그램 할 수 있느냐가 중요한 문제가 되며 이에 대한 많은 도구들이 개발되어 사용되고 있다. 전문가 시스템 개발은 반복적인 과정을 거쳐 시스템을 점진적으로 향상시키는 개발 방법을 사용해야 한다. 그리고 이와 같은 개발기법들을 적용하기 위해 기존의 개발도구와는 다른 전문가 시스템 개발도구가 필요하게 되었다. 초창기의 전문가 시스템 개발자들은 LISP과 같은 인공지능 프로그래밍 언어를 사용하였으며 이러한 노력의 결과 LISP 환경위에 전문가 시스템을 용이하게 개발할 수 있는 특별한 도구들이 제작 되었다. 이러한 개발도구들은 전문가의 지식을 편리하게 표현할 수 있고 표현된 지식을 이용하여 결론을

추론할 수 있는 기본적인 골격을 제공한다.

제 2 절 HyKET (Hybrid Knowledge Engineering Tool)의 구조

국내에서도 인공지능에 대한 열의와 관심이 고조되고 있고 인공지능시스템의 필요성이 학교 연구소 기업체 등에서 증대되고 있으며 이에 대한 연구와 투자가 늘고 있는 실정이다. 하지만 현재 국내의 인공지능개발환경은 아주 미비한 상태이며 이러한 환경구축을 위해 고가의 하드웨어 및 소프트웨어들을 수입해야 하고 또 이의 사용이 어려워서 쉽고 저렴한 가격의 국산 인공지능개발시스템이 절실이 요구되고 있다.

이에 따라 본 과제에서는 복합형 인공지능개발시스템 HyKET를 개발 중에 있다. HyKET는 그림 1-1과 같이 프레임을 기반으로 한 객체지향적 프로그래밍 환경, 규칙기반 추론시스템, 사용자 인터페이스, 강력한 윈도우시스템, Common LISP 프로그래밍 환경과 같은 인공지능 개발기술을 하나의 시스템 형태로 묶어 놓은 복합형 인공지능 개발시스템이다.

1. HyKET의 구성

가. LISP 개발환경

HyKET는 기호처리기능이 가능하고 개발환경이 뛰어난 LISP 프로그래밍 언어를 기본적인 프로그래밍 언어로 채택하며 모든 환경을 LISP 프로그래밍 환경 위에 구축한다.

LISP은 인공지능 연구 및 인공지능 시스템 개발에 가장 많이 사용되고 있는 프로그래밍 언어로서 객체(object)의 개념을 실현하기가 용이하고 절차적 지식

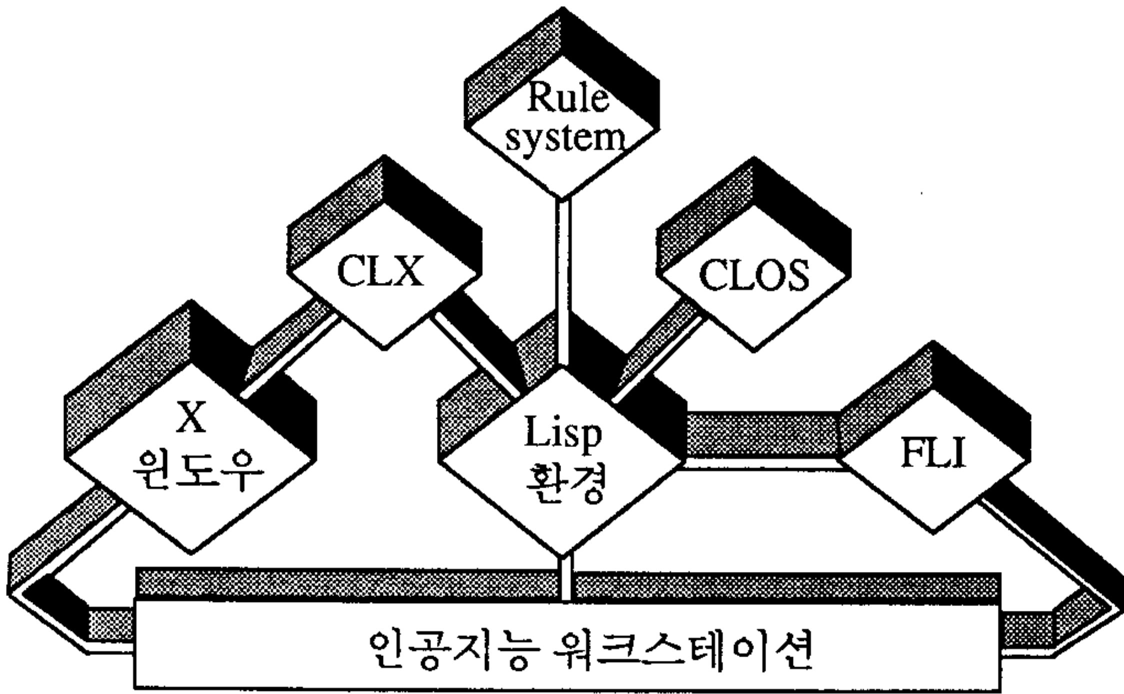


그림 1-1 복합형 인공지능 개발 시스템의 구성도

을 표현하기에도 적합하다. HyKET에서는 현재 LISP의 표준안으로 채택되고 있는 Common LISP을 사용하고 편집기, 디버거, 인스펙터등으로 구성된 LISP 개발환경을 구축한다. 그리고 한글 기호처리를 위한 한글 LISP환경을 구축하고 Common LISP으로는 KCL(Kyoto Common LISP)을 기본적으로 사용하고 있다. 이에 따라 KCL의 원시코드를 분석하여 내부구조를 이해하였으며 이를 바탕으로 KCL의 단점을 보완 개선하여 HyKET의 Common LISP 개발환경을 구축중에 있다. 그리고 한글처리 환경을 구성하였으며 입력편집기, 한글 GMACS 등을 개발하였고 KCL과 GNU-EMACS를 결합하였으며 KCL과 외부 언어와의 인터페이스인 Foreign Language Interface 방법을 구축하였고, LISP의 Garbage Collection 방법의 개선에 대해 연구하였다.

나. 객체지향적 프로그래밍 환경

프레임을 기반으로한 객체지향적 프로그래밍 환경을 HyKET에 구축한다. 이 환경에서는 여러가지 객체들을 표현할 수 있고 이러한 객체들이 계층적인 구조로 연결되어 상속관계를 갖는다. 또한 규칙내에서도 객체의 슬롯값이나 속성값들을 사용할 수 있으며 그래픽 인터페이스도 프레임을 기반으로 구축하고 윈도우 시스템과도 연결을 시킨다. 결국 LISP머신의 Flavor 시스템과 유사한 객체지향적 환경을 구축한다. 이를 위해 Common LISP에서의 표준 객체 지향 언어인 CLOS (Common LISP Object System)을 HyKET의 객체 지향 언어로 채택하였다. 현재의 버전은 CLOS의 full spec.에 비해 기능이 제한되어 있어 이를 개선하여 CLOS의 기능을 확장시켰다.

다. 규칙 기반 시스템

LISP 프로그래밍환경과 객체지향적 프로그래밍 환경을 바탕으로 하여 규칙기반 시스템은 일반적인 규칙기반 시스템의 특성을 지니면서 프레임과 결합되어 다

양한 지식표현 방법을 제공한다. 역방향 추론기능과 순방향 추론기능을 제공하며 규칙의 패턴으로 instance를 가질 수 있다. 또한 규칙의 각 문장에서 논리연산자를 표현할 수 있고 임의의 LISP 함수를 호출할 수 있다. 이의 일환으로 KAPS (KAIST Production System)를 개발하였다. 한편 일반적인 규칙 시스템인 OPS5의 기능을 개선하였으며 한글처리가 가능하게 하였고, PC위의 GCLISP상에 이식하여 사용하게 하였다. 또한, LISP에서 C 언어로 쓰여진 CLIPS의 코드를 혼합 사용할 수 있도록 KCLIPS를 개발하였다. 규칙기반시스템에 진리유지 기능으로 논리기반 진리유지 기능을 KCL로 HAPS2위에 구축하였다.

라. 윈도우시스템

기본적인 윈도우시스템의 기능을 가지며 규칙 시스템, LISP 환경과 결합되어 원활한 사용자 인터페이스 역할을 수행하고 효율적인 프로그래밍 개발환경을 제공하는 윈도우시스템을 구성한다. 현재 윈도우시스템은 세계적으로 X 윈도우로 표준화되고 있으며 특히 Common LISP 환경위의 윈도우시스템으로 CLX (Common LISP X Window)가 표준화 되고 있다. HyKET의 윈도우시스템은 CLX를 기본으로 KCL위에 구현하였다. 한편 PC상에서 프로그래머가 편리하게 윈도우를 생성, 관리할 수 있는 윈도우 시스템을 개발하였다.

제 2 장 Common LISP의 확장 개념으로서 Rule-based System의 통합

제 1 절 서 론

인공 지능(Artificial Intelligence)은 인간과 같은 지능을 가진 컴퓨터 프로그램을 개발하는 것으로, 1970년대에 들어와 전문가 시스템(expert system) 분야를 중심으로 활발한 연구가 진행되었다. 전문가 시스템은 초창기 인공 지능 연구자들이 일반 목적의 문제 해결 프로그램을 개발하려는 것과는 달리, 좁은 분야의 문제를 해결하는데 필요한 경험자의 지식과 추론이라는 방법을 이용하여, 실제 전문가와 비슷한 성능으로 문제를 푸는 프로그램이다. 현재 전문가 시스템은 진단, 예측, 해석, 제어, 설계, 구성 등의 문제 분야에서 폭 넓게 개발되고 있는데, 그 중에서도 가장 성공적인 사례로 DEC에서 개발한, 컴퓨터 시스템을 사용자의 요구에 적합하게 효율적으로 구성하는 전문가 시스템인, R1을 들 수 있다 [HAY 83].

전문가 시스템은 기존의 컴퓨터 프로그램과는 달리 중요한 특징이 있다. 첫째, 전문 지식(expertise)을 가진다는 점이다. 전문가 시스템 목적인 실제 전문가만이 보일 수 있는 높은 수준의 결과를 얻기 위해서는 좁은 분야의 경험적 지식이 필요하다. 둘째, 심볼 조작 동작(symbol manipulation operation)을 이용하여 주어진 문제를 푼다는 것이다. 시스템이 이용하는 지식이 거의 모두 심볼로 표현되며, 이를 이용한 추론 방법도 심볼 조작 동작으로 이루어진다. 셋째, 전문가 시스템은 문제 해결의 열쇠가 되는 지능(intelligence)을 가지고 있다는 점이다. 이 지능은 추론에 기반을 두고 있는데, 전문가 시스템은 맹목적인 탐색(blind

search)이 아니라 경험적인 탐색(heuristic search) 방법을 이용한 추론을 이용한다. 넷째, 전문가 시스템에는 설명 기능(explanation)이 있다. 전문가 시스템은 자신이 추론한 과정을 이용하여 시스템이 결정한 결과에 대하여 그 이유를 설명할 수 있다.

이러한 특징을 갖는 전문가 시스템을 성공적으로 개발하는 것은 쉬운 일이 아니다. 1970년대 초창기 전문가 시스템인 MYCIN, HEARSAY, CASNET, DENDRAL 등은 시스템 구축을 위하여 일반 프로그램 언어를 이용하거나 심볼 조작에 편리한 LISP 언어를 사용하는 것이 고작이었다. 그 이후 1970년대 말부터 전문가의 지식만 수집하여 제공하면 전문가 시스템을 구축할 수 있도록 개발된 전문가 시스템 도구(EMYCIN, OPS5, EXPERT 등)가 나오기 시작하여 전문가 시스템 개발의 황금기를 맞았다. 그러나 초창기 전문가 시스템 구축 도구들은 대부분 LISP 언어로 작성되어, 지식 베이스의 크기가 큰 실제 문제를 풀 때 속도가 매우 느려서 인기를 잃고 있다. 이러한 문제를 해결하기 위하여 일차 개발에 성공한 전문가 시스템은 상용화를 위하여 기존의 프로그램 언어인 Fortran 혹은 C 언어로 다시 프로그램되고 있는 실정이며, 최근 들어서는 전문가 시스템 구축 도구도 실용화를 의식하여 기존 프로그램 언어로 개발하는 추세이다. 그러나 LISP 언어와 기존 프로그램 언어가 서로 장단점을 가지고 있기 때문에, 둘 중 어느 방법을 선택하더라도 상대방의 장점을 얻기는 힘들다.

본 연구에서는 Kyoto Common LISP(KCL)을 이용하여 심볼 처리 언어인 LISP의 장점과 기존의 프로그램 언어인 C 언어로 작성되어 빠른 수행 속도를 제공하는 C Language Production System(CLIPS)의 장점을 모두 제공하는 전문가 시스템 구축 도구를 개발하려고 한다. 본 연구에서 시도하려는 방법은 원천 코드가 제공되는 CLIPS 프로그램을 수정하여 LISP 환경에서 CLIPS를 사용할 수 있게 KCL과 CLIPS를 통합하는 것이다. 이렇게 새로 개발된 도구인 KCLIPS

(KCL + CLIPS)는 기존의 전문가 시스템 도구에서는 동시에 제공하기 힘든 다음과 같은 장점을 제공한다. 첫째, 전문가 시스템 개발시 이 도구의 톱(top) 레벨에서 LISP의 다양한 함수와 CLIPS에서 정의된 함수를 통일된 하나의 방법으로 동시에 사용할 수 있다. 이 방법은 CLIPS만을 사용하여 전문가 시스템을 개발할 때 보다 더 좋은 개발 환경이 되며, LISP으로 작성된 전문가 시스템 도구만이 제공할 수 있는 장점이다. 둘째, CLIPS의 함수 내부에서 LISP 함수를 불러 사용할 수 있다. 이와같이 전문가 시스템을 개발할 때 규칙의 내부에서 LISP으로 작성된 함수를 쉽게 불러 사용할 수 있다는 것은 매우 큰 장점으로, CLIPS만을 이용하여 전문가 시스템을 개발 할 경우에는 불가능한 일이다. 셋째, 전문가 시스템에서 속도 병목 현상이 일어나는 부분인 추론 기관이 기존 언어인 C 언어로 작성되어 있어 속도가 빠르므로 큰 지식 베이스(규칙이 수천 개 이상)를 가지는 실용적인 전문가 시스템 개발에도 사용할 수 있다. 이는 LISP으로 작성된 전문가 시스템은 제공하기 힘든 기능이다. 넷째, 전문가 시스템의 기능을 쉽게 확장할 수 있다. 예를 들어 전문가 시스템이 외부 프로그램과 대화적으로 인터페이스하면서 수행되는 경우 LISP 언어를 이용하여 이러한 기능을 쉽게 프로그램하여 넣을 수 있다. CLIPS에서는 C, Fortran등과 같은 언어와의 외부 프로그램 기능은 제공하나 매우 불편하며, LISP언어와의 인터페이스 기능은 제공하지 않는다.

본 연구의 2절에서는 LISP 환경하에서의 rule-based system의 필요성을 설명하였고, 3절에서는 KCL의 구조를 분석하여 그 특징을 설명하였으며, CLIPS의 개요 및 구조적인 특징을 기술하였고, KCL과 CLIPS의 통합을 위한 방법론에 대하여 기술하였다. 4절에서는 KCL과 CLIPS의 통합 시스템인 KCLIPS의 설계를 위하여 KCL과 CLIPS의 차이점을 분석하여 기술하였으며, KCL과 CLIPS의 통합에 필요한 기능 및 메케니즘에 대하여 기술하였다. 5절에서는 실제 구현 및

사용 보기를 통한 시험 결과를 기술하였고, 6절에서는 결론을 기술하였다.

제 2 절 LISP 환경하에서의 Rule-based System의 필요성

1. 필요성

LISP언어는 심볼 조작에 편리한 언어로서 많은 장점을 지니고 있으나, 이 언어를 이용하여 전문가 시스템과 같은 응용 프로그램을 작성한다는 것은 매우 불편한 일이다. 반면 규칙을 기반으로하는(rule-based) 전문가 시스템 구축도구들은 전문가 시스템과 같은 프로그램 개발에는 편리하나, 심볼조작과 같은 기능을 제공하지 않아 이러한 기능이 필요할 경우에는 프로그램 확장이 어려워지는 단점이 있다. 그러므로 입출력과 같이 심볼처리가 많이 발생하는 경우에는 LISP기능을 사용하고, 의사결정이 필요한 경우에는 규칙기반 시스템의 추론기능을 사용한다면 보다 편리하게 다양한 기능을 갖는 프로그램을 작성할 수 있을 것이다.

이와같이 LISP의 심볼처리 기능과 규칙기반 시스템의 추론기능을 동시에 제공하게 되면 다음과 같은 장점을 제공할 수 있을 것이다. 첫째, 전문가 시스템 개발시 LISP의 다양한 함수와 전문가 시스템 구축도구의 함수를 동시에 사용할 수 있다. 둘째, 전문가 시스템 구축도구가 제공하는 규칙 내부에서 LISP 함수를 불러 사용할 수 있다. 셋째, 추론기능이 필요한 경우에는 쉽게 전문가 시스템 구축도구를 사용할 수 있다. 넷째, LISP의 기능을 이용하여 전문가 시스템의 기능을 쉽게 확장할 수 있다.

이와 같은 장점들은 LISP만을 사용하거나, 규칙을 기반으로 하는 전문가 시스템 구축도구만을 사용하는 경우 보다 훨씬 다양하고 강력한 프로그램을 손 쉽게 작성할 수 있게 하여준다. 이와 같이 LISP 환경하에서의 규칙기반 시스템의

필요성은 매우 크다고 할 수 있다.

제 3 절 KCL과 CLIPS의 분석

1. KCL의 구조

가. KCL의 구조

Kyoto Common LISP(KCL)은 표준화된 Common LISP의 스펙을 만족하는 LISP이다. 핵심 부분은 대부분 C 언어로 구현되어 있어서 C 언어와의 인터페이스 기능을 가지며 여러 기계에서 수행되는 호환성을 가진다. 보통 컴파일된 화일을 인터프리터(interpreter)가 수행하며 기본 동작원리는 Read-Eval-Print의 루프(loop)로 동작하며 사용자가 제공하는 입력을 받아 분석하고 계산하여 결과를 돌려주고 다시 입력을 기다린다.

KCL은 Value 스택을 이용하여 계산(evaluation)을 수행한다. 입력 스트림 버퍼에서 읽은 자료는 전역변수에 기억되고 이 변수가 가지는 내용을 파싱(parsing) 하여 의미있는 자료로 만든후 Value 스택의 vs_base, vs_top의 두 포인터 변수를 가지고 동적인 연산을 한다. Evaluation을 위한 기능으로 함수가 폐쇄(closure) 구조를 가지면 렉시칼 환경(lexical environment)은 Value 스택에 저장하고 폐쇄 구조를 가지지 않으면 Value 스택에 3개의 공간만 잡아준다. 렉시칼 환경은 변수의 바인딩, 지역 함수나 마크로의 정의, 그리고 택(tag)이나 블록(block)의 바인딩을 가지는 관련 리스트(association list)이다.

KCL은 22 종류의 객체를 갖고 있으며 이들은 공통적으로 타입 부분과 마크(mark) 부분을 갖고 보디(body)는 그 셀(cell)이 자유로울때는 포인터로 쓰이고 그렇지 않을때는 해당 정보를 간직하고 있다. 동적 수행시에 필요한 정보와 자료

는 Value 스택, 바인딩 스택, Invocation History 스택, 그리고 C 언어 제어 스택을 이용하여 수행한다.

LISP에서의 입출력은 스트림 버퍼를 통하여 일어난다. 스트림 버퍼는 기존 C 언어에서 사용하는 화일 구조와 추가적 정보를 갖는 상위 레벨의 구조로 이루어져 있다. KCL이 사용할 수 있는 스트림 버퍼는 기존과 같이 제공 되는 것과 사용자가 필요시 정의하여 사용할 수 있는 것이 있다.

나. KCL에서의 C 언어와의 인터페이스

KCL은 C 언어로 구성되어 있으며, 일반 LISP 프로그램을 컴파일 시키면 일단 C 언어 원시코드로 바꾸어주고 이 C 프로그램을 C 언어 컴파일러로 컴파일 시키고 있다. 때문에 C 언어로 된 프로그램은 적절한 마크로를 이용하여 LISP 환경에서 사용할 수 있도록 되어 있다.

KCL을 C 언어와 인터페이스 시키기 위하여는 CLINES 마크로와 DEFCFUN 함수를 사용하고있다. CLINES 마크로는 마크로 본체에 있는 스트링을 그대로 C 언어 화일에 옮기도록 한다. 이 C 언어 화일은 다시 컴파일 되며 LISP에서 사용할 때는 이 C 함수를 호출하는 함수를 만들어 주어야 하며 이를 defcentry LISP 함수가 처리해준다.

DEFCFUN 함수는 C 언어를 LISP 언어의 함수와 함께 사용할 수 있게 하여준다. DEFCFUN 함수에서는 함수본체에서 스트링이 오면 이를 C 언어로 생각하여 원시코드 그대로 C 언어 화일에 옮기고 리스트 형태가 오면 이를 LISP의 s-expression으로 생각하여 적절한 C 언어 코드로 바꾸어 C 언어 화일에 쓰게된다. 따라서 DEFCFUN 함수의 본체는 스트링과 리스트들이 섞인 형태가 된다. 그러나 이 리스트 형태는 LISP 언어의 형태와 같지 않고 약간의 차이점이 있어 불편한 점이 있다. 또한 이 함수에서 변수를 LISP 환경에 저장할 필요가 있을

경우 그 변수의 수량을 명시해 주어야 한다.

이와같이 KCL에서의 C 언어와의 인터페이스는 원시코드를 대상으로 하며 이는 KCL의 크로스 컴파일(cross-compile) 기능을 확장한 것으로 라이브러리 화일이나 오브젝트화일의 적재는 불가능하며, 전달할 수 있는 자료형은 정수와 문자형의 기본형만이 가능하다.

이러한 단점을 해결하기 위하여 한국과학기술원 인공지능연구실에서는 기존 KCL을 확장하여 다음과 같은 기능을 추가하였다 [YSO89].

- o LISP 언어에서 외부언어함수를 호출할 수 있는 기능
- o 외부언어에서 LISP 함수를 호출하는 기능
- o 외부언어 함수화일의 적재및 관리기능
- o 외부언어 자료형과 LISP 자료형의 접속기능

본 연구에서는 이와 같이 확장된 KCL을 사용하였다.

2. CLIPS의 구조

가. CLIPS의 개요

C Language Production System(CLIPS)는 NASA/Johnson space center의 인공지능 부서에서 개발한 리트 알고리즘(rete algorithm)에 바탕을 둔 전진추론(forward chaining) production 시스템이다 [CLP87]. 이 시스템은 범용 컴퓨터에서의 LISP의 부족한 기능을 보완하고 LISP으로 개발된 전문가 시스템 구축도구에 비해 보다 싼 가격으로 보다 빠른 전문가 시스템 개발용 도구를 제공하기 위한 목적으로 개발되었다.

CLIPS는 기본적으로 사실들(facts)을 저장하는 전역(global) 기억 장치와, 모

든 규칙들(rules)을 저장하는 지식기반(knowledge base), 그리고 규칙의 실행을 담당하는 추론기관(inference engine) 으로 구성 되어있다. 이러한 CLIPS의 특성을 요약 하면 표 2-1과 같다 [CKC89].

CLIPS를 이용하여 전문가 시스템을 개발하는 과정은 사실들(facts)을 정적(static) 지식으로 표현하고 이를 조절하기위한 규칙(rule)들을 설정함으로써 특정 분야의 문제를 지적으로 해결할 수 있는 프로그램을 작성하는 일이다.

추론과정에서 정적인 지식은 워킹 메모리(working memory)에 보관된다. 워킹 메모리는 CLIPS에서 사용하는 사실들의 집합으로 이루어지며 이러한 사실에 대하여 규칙이 적용된다. 워킹 메모리에 어떤 사실 하나를 추가하는 명령은 "assert" 이며, 제거하는 명령은 "retract fact-number" 이다.

```
(assert (The duck said "Quack"))
(retract 3)
```

지식기반(knowledge base)에는 CLIPS에서 사용하는 규칙들이 정의되며, 규칙은 다음과 같은 형태로 정의된다.

```
(defrule rule-name   "optional-comment"
  (pattern-1)
  (pattern-2)
       .
       .
  (pattern-n)
=>
  (action-1)
  (action-2)
       .
       .
  (action-m))
```

여기에서 defrule은 규칙을 정의하는 명령어이고 rule-name은 규칙의 이름을 나타낸다. 심볼 =>의 앞부분을 LHS(left hand side)라고 하며, LHS에는 임의

<table>
<tr><td rowspan="4">Knowledge
Base</td><td>Facts</td><td>A-V</td></tr>
<tr><td rowspan="2">Relation</td><td>rule</td></tr>
<tr><td>variable</td></tr>
<tr><td>Uncertainty</td><td>not available</td></tr>
<tr><td rowspan="5">Interface</td><td colspan="2">Recognize-Act cycle</td></tr>
<tr><td colspan="2">Forward Chaining</td></tr>
<tr><td colspan="2">Depth First Search</td></tr>
<tr><td colspan="2">Breadth First Search</td></tr>
<tr><td colspan="2">Meta-rule, Agenda</td></tr>
<tr><td rowspan="5">User-
Interface</td><td colspan="2">Edit .... Micro EMACS editor</td></tr>
<tr><td colspan="2">Trace</td></tr>
<tr><td colspan="2">Explanation</td></tr>
<tr><td colspan="2">Windows</td></tr>
<tr><td colspan="2">Other language interface</td></tr>
</table>

표 2-1 CLIPS의 특성

의 리스트 형태 또는 변수를 포함하는 패턴들로 구성된다. RHS에는 워킹 메모리에 있는 패턴과 매치(match)되는 사실이 있으면 수행되는 액션들이 존재한다. 이러한 액션에는 새로운 사실의 추가, 기존 사실의 제거, 결과의 출력, 입력자료의 요구등이 포함될 수 있다. 이와 같이 일단 지식기반이 만들어지고 사실 리스트 (fact list)가 만들어 지면 CLIPS는 규칙을 실행할 준비를 한다. CLIPS에서의 기본적인 실행 사이클(execution cycle)은 다음과 같다.

- o 규칙의 조건이 매치(match) 되었는지를 알아보기 위하여 지식 기반을 검토한다.
- o 규칙에 매치된 모든 규칙은 스택(stack)에 들어가게 된다. 이때 새로운 규칙의 우선순위(priority)가 현재 스택의 제일 위에 있는 규칙 보다 낮으면 우선순위가 새로운 규칙보다 낮은 규칙이 나올때 까지 스택을 조정한다.
- o 스택의 제일위에 있는 규칙을 수행 시킨다.
- o 규칙의 수행 결과로 새로운 조건을 만족 시키는 규칙이 발생하면 위의 과정을 되풀이 한다.

이와 같은 것을 사용한 간단한 프로그램 예는 아래와 같다.

```
(defrule duck
  (animal-is duck)
=>
  (fprintout t "quack" crlf))

(assert (animal-is duck))
(run)
```

이 프로그램의 결과는 아래와 같다.

```
quack
1 rules fired
```

나. CLIPS의 구조

CLIPS는 기본적으로 read-eval의 무한 루프(loop)를 수행하고있는 인터프리터로서 터미널 또는 화일로부터 입력을 읽어들여 이를 파싱(parsing)하고 계산(evaluation)을 수행하는 일을 반복하고 있다. CLIPS는 크게 CLIPS 파서(parser)와 C 함수 부분으로 되어 있으며 이 두 부분은 같은 자료구조를 공유한다. 전체적인 CLIPS의 구조는 그림 2-1과 같다.

CLIPS 파서는 다시 다음과 같이 여러가지로 구분 파서로 나뉘어진다.

o parse-deffacts

o parse-defrules

o if-parse

o while-parse

o assert-parse

o retract-parse

o 그외 공통 parser

이들 파서 각각은 정해진 키 워드(key word)로 되어 있는 프로그램 소스 코드를 분석하여 하나의 프레임(frame) 형태의 구조를(structure) 만든다. defrule로 정의된 규칙은 parse-defrule에서 파싱되어 규칙 기억장소인 *rule-list*에 추가되고 작업이 종료된다. 그외의 것은 파서에서 만든 자료구조와 이에 대한 포인터들이 generic_compute(*pointer)와 define_function 함수를 통하여 실제 실행부분인 C 함수 부분으로 넘겨진다. 한 예로 (assert (obj ?att value))의 CLIPS 코드가 수행되는 과정은 그림 2-2와 같다.

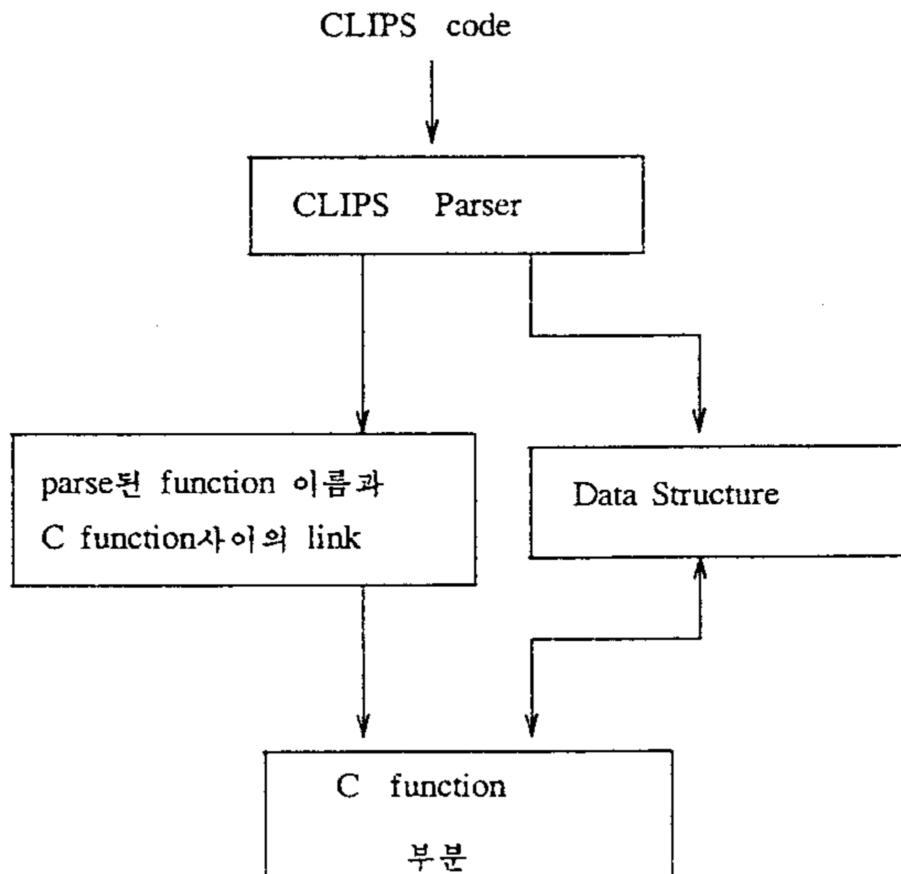


그림 2-1 CLIPS의 구조

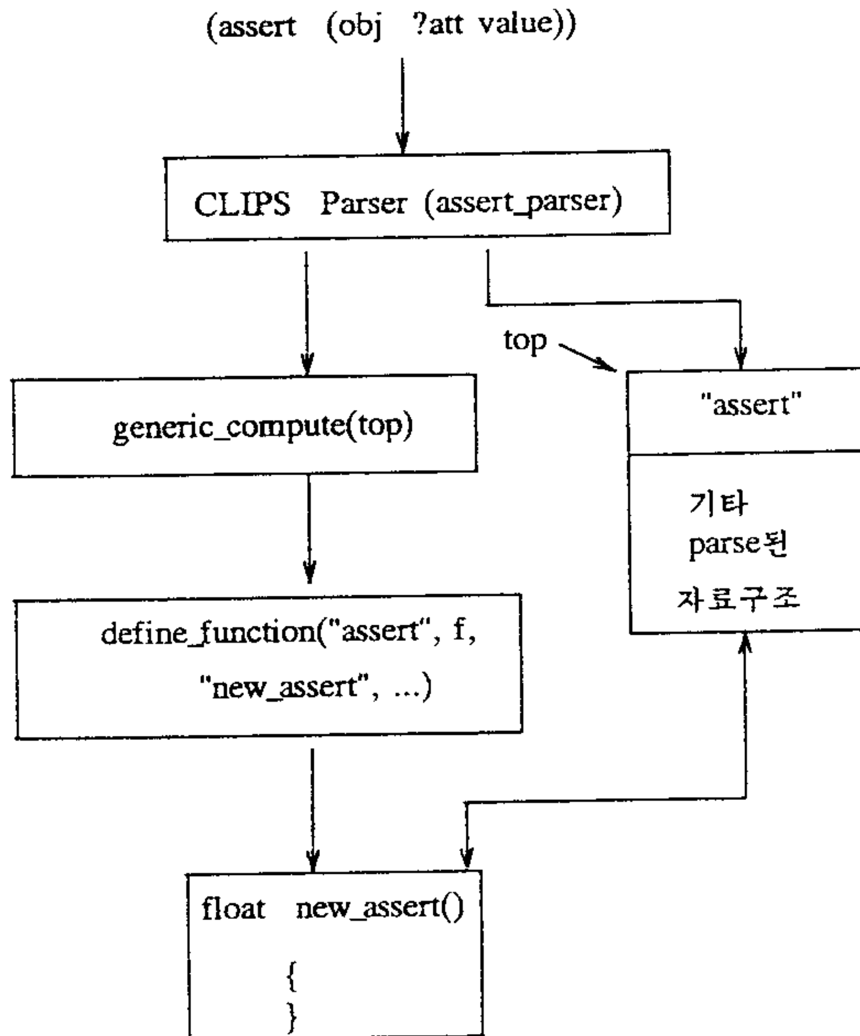


그림 2-2 CLIPS 프로그램의 수행과정

제 4 절 통합 시스템 KCLIPS의 설계

1. KCL과 CLIPS의 통합 방법

KCL과 CLIPS를 통합하여 전문가 시스템 도구를 만들 때, 두 시스템의 통합 정도에 따라 사용자에게 제공되는 기능 및 제약점에 차이가 생길 수 있다. 여기서는 KCL과 CLIPS의 통합 정도를 세개의 단계로 나누어 정의하고 이들 각각의 기능과 제약점을 보기를 들어 설명하였다.

가. KCL과 CLIPS 사이에 바인딩이 일어나지 않는 경우

이 경우 사용자는 새로 만들어진 프로그램의 top level에서 KCL과 CLIPS를 동시에 사용할 수 있지만, 하나의 S-expression내에 KCL과 CLIPS를 섞어서 사용할 수 없다. 즉 KCL을 사용하여 그 결과를 얻고난 뒤 다시 CLIPS를 사용하여야 한다. 다음 보기1은 이 통합의 예이다.

```
% kclips
> (setq number (+ 2 3))
5
> (assert (I have 5 books))
t
>
```

<보기 1> KCL과 CLIPS 사이에 바인딩이 일어나지 않는 경우

이 경우는 KCL과 CLIPS를 통합하여 하나의 프로그램화 하였지만, 그 통합 정도가 너무 낮아 KCL 수행의 결과를 CLIPS에게 바인딩 시킬 수 없고, CLIPS의 결과도 KCL에 바인딩 시킬 수 없다. 이 통합 방법은 제일 기초적인 통합이라고 할 수 있다.

이 통합 방법의 구현은 KCL과 CLIPS를 서로 독립적으로 수행시키면서, KCL과

CLIPS를 사용자의 요구에 따라 선택하여 불러서 그 결과를 돌려주는 스위처 (switcher)를 만들어줌으로써 가능하다.

나. KCL에서 CLIPS를 사용할 때 초기 바인딩이 일어나는 경우

이 경우는 KCL 함수 부름 내부에 CLIPS 문을 넣을 수도 있고, CLIPS 문 내부에서 KCL 함수를 불러 사용할 수 있다. KCL 함수 부름 내부에 CLIPS 문을 넣었을 경우 CLIPS를 수행한 후 그 결과를 KCL에 넘겨주며, CLIPS 내부에서 KCL을 사용했을 경우는 KCL의 리턴값이 CLIPS로 전달된다. 이때 CLIPS에서 본 KCL 리턴값의 전달은 초기 바인딩이다. 다음 보기2는 이 방법으로 통합된 시스템의 사용 예이다.

```
% kclips
 > (setq tom-book 2)
 2
 > (setq mary-book 3)
 3
 > (assert (they have #lisp(+ tom-book mary-book) books))
 t
 > (facts)
 f-1        (they have 5 books)
 t
 > (defrule rule1
   (startfact yes)
     =>
   (assert (they have #lisp(+ tom-book mary-book) books)))
 t
 > (pprule )
    (defrule rule1
   (startfact yes)
     =>
   (assert (they have 5 books)))
 t
   >
```

<보기 2> KCL의 수행 결과를 CLIPS에 바인딩시키는 경우

이 방법은 처음의 통합 방법에 비하여 두 시스템이 깊게 통합된 경우로서 두 언어가 하나의 언어처럼 사용될 수 있다는 장점이 있다. 그러나 초기 바인딩의 제약 때문에 CLIPS 규칙의 수행 시간에 KCL의 함수가 동작되지 않기 때문에 규칙 내부에 KCL을 추가시킨다는 것에 대한 의미가 적다.

이 방법을 사용하여 통합하는 경우 KCL과 CLIPS가 한 언어처럼 사용되어야 하기 때문에, 이들 두 언어를 하나로 볼 수 있고 또한 KCL 함수와 CLIPS 함수를 구분할 수 있는 문법이 새로 정의되어야 한다. 새로운 문법은 CLIPS 보다는 KCL에 유사하게 정의되는 것이 바람직한데 그 이유는 CLIPS 보다 KCL이 문법적으로 잘 정의되어 있기 때문이다. 또한 KCL 문 내부에 CLIPS 문을 사용한 경우 KCL은 CLIPS의 리턴값을 기다린다. 그런데 CLIPS에는 원래 리턴값이란 것이 존재하지 않으므로 이 경우 CLIPS 각 함수의 리턴값을 새로 정의하여야 한다. 그리고 KCL에서 CLIPS의 수행 결과를 가져오기 위하여 KCL에서 CLIPS 함수를 불러주는 메캐니즘이 있어야 한다.

다. KCL에서 CLIPS를 사용할 때 런 타임 바인딩이 일어나는 경우

이 경우는 두번째 통합 방법과 마찬가지로 KCL 함수 부름 내부에 CLIPS 문을 넣을 수도 있고, CLIPS 문 내부에서 KCL 함수를 불러 사용할 수 있다. 다만 CLIPS 함수 내에서 KCL을 불러 쓸 때, CLIPS의 함수가 수행되는 런 타임 때에 KCL이 수행된다. 이 경우의 예는 보기3과 같다.

```
% kclips
> (setq tom-book 2)
2
> (setq mary-book 3)
3
> (assert (they have #lisp(+ tom-book mary-book) books))
t
> (facts)
```

```
f-1        (they have 5 books)
t
> (defrule rule1
  (startfact yes)
    =>
  (assert (they have $lisp(+ tom-book mary-book) books)))
t
> (pprule )
    (defrule rule1
  (startfact yes)
    =>
  (assert (they have $lisp(+ tom-book mary-book) books)))
t
>
```

<보기 3> KCL에서 CLIPS를 사용할 때 런 타임 바인딩이 일어나는 경우

이 방법은 두번째 방법보다 더 유용하게 사용될 수 있다. 왜냐하면 CLIPS 규칙 내부에 포함된 KCL 함수가 CLIPS 규칙의 매치(match)시 수행되기 때문이다.

이 방법의 구현을 위하여서는 초기 바인딩 방법에서 구현되어야 하는 것 이외에도 CLIPS에서 KCL 함수를 불러서 그 결과를 가져올 수 있는 메케니즘을 추가하여야 한다. 그런데 이 방법에는 CLIPS 문 내부에서 KCL 함수를 부를 때 KCL 함수가 파라메터로 KCL의 global 변수만 가지게 하는 경우, CLIPS의 변수만 가지게 하는 경우, 이를 섞어서 사용할 수 있게 하는 경우가 있을 수 있다.

2. 초기 바인딩 메케니즘의 설계

기본적으로 KCL은 무한 루프(loop)를 수행하는 인터프리터(interpreter)로서 read-eval-print의 3가지 주요기능을 수행하고 있다. CLIPS 또한 무한 루프를 돌고있는 인터프리터로서 read-eval 2가지 주요기능을 수행하고 있다. 그러므로 서

로 독립적인 이 두가지 인터프리터를 통합하기 위해서는 어느 한쪽 루프를 없애 버려야한다. 본 연구에서는 그림 2-3과 같은 방법을 사용하였다.

그러나 위와같이 통합하기 위하여는 KCL과 CLIPS가 기본적으로 어떻게 다르며 이와같이 다른점들을 어떻게 해결하여야 하는가를 검토 하여야 한다. KCL과 CLIPS가 다른점은 아래와 같다.

o 서로 다른 자료구조(data structure)를 사용하고 있다.

o KCL에서는 return value가 있으나 CLIPS에는 return value가 없다.

o KCL에서는 quote(')를 사용하나 CLIPS에서는 quote를 사용하지 않는다.

예) KCL : (setq mylist '(father John Tom))

CLIPS : (assert (father John Tom))

o KCL에서는 return된 값을 print 하여주는 루틴(routine)이 있으나, CLIPS에서는 return값 자체가 없으며 print 하여주는 루틴도 없다

o KCL에서는 영문자의 대문자 소문자의 구별이 없으나, CLIPS에서는 영문자의 대문자와 소문자를 구별한다. 즉 case sensitive 하다.

이와같은 차이점을 해결하기 위하여 다음과 같은 해결방법을 고려하였다.

o 자료구조는 각각 고유한것을 사용하며 두개의 자료구조 사이의 자료전송(data transfer)은 버퍼(buffer)를 사용한다.

o CLIPS에 return기능을 추가하며 그 결과를 LISP의 print루틴을 사용하여 출력한다.

o LISP 프로그램내에 규칙기반 프로그램을 포함하는 환경을 고려하기 위하여 CLIPS문 내에 LISP 함수가 사용될 경우 이를 나타내기 위하여 LISP 함수 앞에 #lisp 이라고 표시한다.

예) (assert (The area is #lisp(square x) square meters))

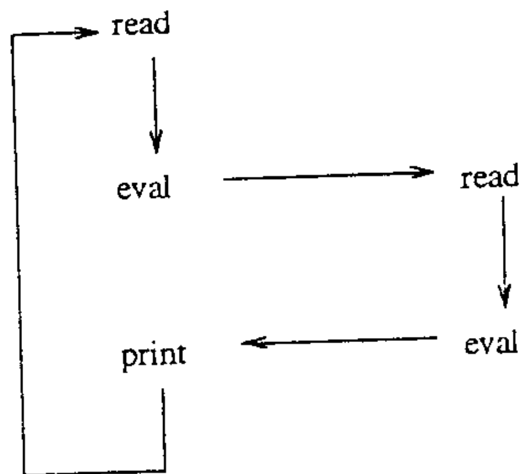


그림 2-3 KCLIPS의 수행 루프

o print 루틴은 KCL의 것을 사용한다.

o CLIPS에서도 KCL과 같이 소문자 대문자의 구별을 하지 않는다.

즉, CLIPS에서는 주로 소문자를 사용한다.

o KCL과 CLIPS에 같은 함수이름이 존재할 경우 그 문장이 KCL문이면 KCL함수로 CLIPS문이면 CLIPS 함수로 간주한다.

이와 같은 가정하에서 다음과 같이 CLIPS 문이 LISP 프로그램내에 포함되어 있는 경우를 생각하여 보자.

```
(setq a 3)
(setq b 2)

(assert (I have #lisp(+ a b) books))
```

세번째 문장을 수행하기 위하여 KCL에서는 #lisp(+ a b)를 수행하여 세번째 문장을 (assert (I have 5 books))로 바꾸어 놓은 다음 "assert" 라는 LISP 함수를 수행하려 할 것이다. 그러나 "assert" 는 LISP 함수가 아니라 CLIPS 함수이므로 이의 수행이 불가능 하다. 이와 같이 수행이 불가능한 "assert" 함수를 수행시키기 위하여 "assert"가 CLIPS의 함수인 것을 판단하여 CLIPS에게 넘겨준 후 이의 수행 결과를 받아 오는 방법을 고려하여야 할 것이다. 이것을 그림으로 나타내면 그림 2-4와 같다.

즉, KCL에서 "assert" 문장이 CLIPS의 함수인 것을 판단하여 "assert" 문장을 *trans-buffer* 라는 버퍼(buffer)로 옮긴 다음 CLIPS의 main 함수를 호출(call) 하는 기능을 수행한다. 이때 호출된 CLIPS의 main 함수는 *trans-buffer* 로부터 CLIPS 함수인 "assert" 문장을 읽어내어 수행하고 그 결과를 *result-buffer* 에 넣어 둔다. 그러면 KCL에서는 이 결과를 읽어서 print 루틴을 수행하게 된다.

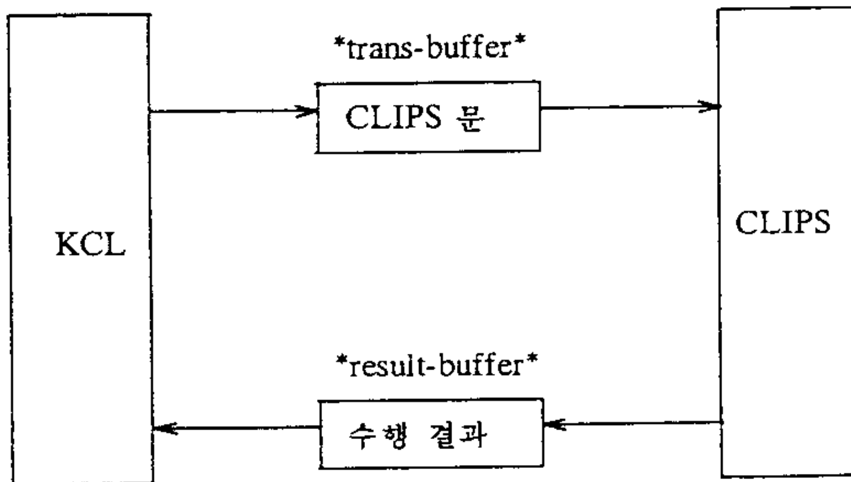


그림 2-4 KCLIPS의 기본 개념

이러한 개념을 설계하는데는 다음과 같은 메케니즘이 필요하다.

첫째, LISP 프로그램내에서 CLIPS 명령어를 판단하고, 해당 CLIPS 문장을 *trans-buffer*에 넣는 방법

둘째, LISP 프로그램 내에서 CLIPS의 main 함수를 호출할 수 있는 방법,

셋째, CLIPS에서 *trans-buffer*에 있는 내용을 읽어서 수행할 수 있는 방법,

넷째, CLIPS에서 *trans-buffer*에 있는 내용을 읽어서 수행한 후 그 결과를 *result-buffer*에 넣어 두는 방법

첫번째 문제를 해결하기 위하여는 CLIPS의 최상위 명령어 30개 모두를 인식하고 해당 CLIPS 문장을 *trans-buffer*에 넣어주는 에뮬레이션(emulation)프로그램이 필요하다. 이 프로그램은 LISP의 사용자정의 함수(user-defined function)로 미리 정의될 수 있으며 KCL과 같이 항상 사용할 수 있도록 준비 되어야 한다. 특히 이 프로그램은 KCL과 CLIPS의 원만한 인터페이스를 위하여 필요시 자료의 타입을 바꾸기도 하여야 한다.

두번째 문제와 네번째 문제를 해결하기 위하여는 한국과학기술원 인공지능연구실에서 확장한 KCL의 foreign language interface를 사용하기로 한다 [YSO89]. 그러나 CLIPS는 여러곳에서 수행결과를 돌려주므로 필요시 CLIPS의 많은 부분을 수정 하여야 한다.

세번째 문제를 해결하기 위하여는 CLIPS내에서 *trans-buffer*로 부터 입력을 읽어 들일 수 있는 system library call 기능을 개발하여야 한다. 이들 각각에 대한 구현은 다음 장에서 자세히 설명하기로 한다.

3. 런 타임 바인딩의 메케니즘의 설계

런 타입 바인딩은 프로그램이 입력되는 시점에 각 변수 값이 evaluation되는 초기 바인딩과는 달리 실행시 각 변수의 값이 evaluation되는 메케니즘이므로 초기 바인딩 메케니즘 이외에도 다음과 같은 추가 기능이 필요하다.

o CLIPS에서 LISP 함수내에 LISP 변수를 사용할 경우

```
(예) (defrule rule1
  (LHS conditions)
   .
      =>
  (bind ?x $lisp(twotimes a))
  (RHS actions)
   .
      ))
```

위의 예에서 $lisp(twotimes a)는 CLIPS 문 내에서 사용되는 LISP 함수로 LISP 변수 a를 사용하고 있다. 이러한 문장을 정확히 evaluation 하기위하여는 이 규칙이 조건을 만족하여 실행될 때 변수 값이 바인딩되어야 한다. 그러므로 CLIPS에서는 KCL에게 LISP 변수의 값을 물어 이 함수를 수행할 수 있는 기능을 추가 하여야 한다.

o CLIPS에서 LISP 함수내에 CLISP 변수를 사용할 경우

```
(예) (defrule '(rule2
  (LHS conditions)
   .
      =>
  (bind ?y (twotimes ?x))
  (RHS actions)
   .
      ))
```

위의 예에서 $lisp(twotimes ?x)는 CLIPS 문 내에서 사용되는 LISP 함수가 CLIPS 변수를 사용하고 있는 예를 보여주고 있다. 이러한 경우 먼저 CLIPS 변

수값을 바인딩 시킨후 KCL에 LISP 함수의 evaluation을 요구하여야 한다. 이와 같이 run-time 바인딩을 위하여는 다음과 같은 메케니즘이 필요하다.

첫째, LISP 함수와 CLIPS 함수가 섞여 사용될 경우 CLIPS 내에서 run-time 바인딩이 요구되는 LISP 문장을 알아내는 방법

둘째, CLIPS 내에서 LISP 함수가 사용될 경우 LISP 함수내에 CLIPS 변수가 사용되면 CLIPS 변수를 바인딩 시키고 LISP 변수가 사용되면 LISP 변수를 바인딩 시켜 KCL에 evaluation을 요구하는 방법

셋째, CLIPS에서 evaluation을 요구한 LISP 함수를 KCL에서 받아 처리하는 방법

넷째, KCL에서 evaluation한 결과를 CLIPS에서 받아가는 방법

이러한 방법하에 CLIPS내의 LISP 함수를 처리하는 절차를 살펴보면 다음과 같다.

초기 바인딩 메케니즘에 의해 KCL과 CLIPS가 서로 통합되어 심볼 테이블을 공유하므로, CLIPS에서 임의의 함수를 처리할 때 자체 함수 테이블을 찾아서 없으면 오류로 처리하지 않고 바로 KCL의 eval 함수를 호출한다. 이때 매개변수 값은 CLIPS에서 사용하는 LISP 함수 리스트가 되고 이 리스트의 자료구조는 LISP과 같은 형태로 만들어 준다. 물론 이때 CLIPS 변수가 사용되면 먼저 바인딩 시킨후 처리해야 된다.

CLIPS 내의 LISP 변수를 처리하는 과정은 똑같이 할 수 있다. 단지 eval 함수를 호출할 때 매개변수의 값이 심볼 이름이 된다. 여기서 주의할 점은 CLIPS 내에서 사용되는 LISP 함수의 리스트내에 다시 CLIPS 함수를 사용할 수 없다는 점이다. 이러한 제약을 두지 않으면 구현이 복잡해지고 궁극적으로는 KCL과 CLIPS의 자료구조를 공유해야 하는 형태가 된다.

제 5 절 KCLIPS의 구현 및 시험

1. KCLIPS의 구현

현재 KCLIPS(KCL + CLIPS) 환경은 초기 바인딩 메케니즘까지만 구현되어 있으며 런 타임 바인딩 메케니즘은 추후 구현될 예정이다. 이장에서는 초기 바인딩 메케니즘의 구현에 관하여 상세하게 기술하고자 한다.

가. 구현 개요

CLIPS를 KCL에 통합시키기 위해 가장 먼저 고려할 사항이 통합 방법에 관한 것이다. 즉, CLIPS의 메인 함수만을 인터페이스를 위한 유일한 연결점(entry point)으로 사용하느냐 아니면 CLIPS의 각종 명령어에 대한 함수를 각각 따로 연결점으로 사용하느냐를 결정해야 한다. 전자의 경우, 모든 명령어의 처리에 일관성이 있고 오류가 발생했을 때의 처리가 용이하며 CLIPS의 거의 모든 기능을 사용할 수 있는 장점이 있는 반면 명령어의 종류에 따라 필요없는 과정을 거치는 부담이 있고 명령어에 따른 리턴(return)값을 각각의 특성에 맞게 처리할 수 없다는 단점이 있다. 한편 후자의 경우, 명령어를 바로 연결하므로 필요없는 과정을 거치지 않아도 되고 각각의 리턴값을 원하는 형태로 정할 수 있으므로 KCL에서 사용하기가 용이하다는 장점이 있는 반면 CLIPS 각 명령어마다 파싱 과정을 거쳐야하는데 이 과정이 약간씩 다르기 때문에 CLIPS 인터프리터의 많은 부분을 수정하여야 하며 사용할 수 있는 명령어의 수 및 그 기능이 한정된다[CLA87].

본 연구에서는 전자의 방법, 즉 CLIPS의 메인 함수만을 KCL 인터페이스에 대한 연결점으로 하고 CLIPS의 모든 명령어를 이 연결점을 통하여 KCL과 연결하는 방법을 사용하였다. KCL에서는 CLIPS 명령어를 사용할 때 입력되는 문장

을 문자열로 바꾸어 그 주소와 크기를 매개 변수로 하여 CLIPS로 넘기게 하였으며, CLIPS에서는 표준 입력 스트림(stdin)에서 받아들이는 입력을 KCL로부터 전달되는 매개 변수에서 받아들이게 하고 표준 출력 스트림(stdout)으로 나가는 출력을 문자열로 바꾸어 그 주소 값을 KCL로 리턴하였다.

이러한 구현 방법을 위하여 KCL에서 추가한 사항, CLIPS에서 수정한 사항, 설치할 때의 문제점 및 고려사항 등을 대하여 살펴본다.

나. KCL에서의 추가사항

(1). KCL과 CLIPS의 인터페이스 정의

확장된 KCL에 의하면 외부 언어 인터페이스의 형식은 아래와 같다.

(load-foreign foreign-file {(entry-list)*}

*entry-list : (lisp-fn (return-type foreign-fn) (arg-type-list))

이 함수의 기능을 살펴보면, 첫째로 하나 또는 여러개의 화일을 인자로 받아 필요한 경우 해당 언어의 컴파일러를 호출하여 오브젝트 화일로 만들어 준 후 자체 로더로 기억장소에 적재시킨다. foreign-file은 외부 언어 화일의 이름을 나타내는데, 단일 화일인 경우는 문자열이며 여러개의 화일인 경우에는 문자열의 리스트로 표현된다. 화일명 내의 extention이 ".c"나 ".p"인 경우는 원시코드인 경우이므로 해당 컴파일러를 호출해 주며 ".o"인 경우는 오브젝트 화일로 간주한다. 둘째로, entry-list 안에 있는 lisp-fn과 foreign-fn을 연결 시킨다. lisp-fn은 KCL에서 외부함수를 호출할 때 사용하는 이름이고 foreign-fn은 외부함수의 실제 이름이다. foreign-fn은 문자열로 표시되고 return-type은 KCL의 자료형이어야 한다. arg-type-list는 매개 변수의 자료형의 리스트이다. 마지막으로, :lib-dir, :library, :language는 사용되는 라이브러리의 경로, 이름 및 언어의 종류를 나타낸다. 각각

의 초기값은 "c", nil, "c"이다.

CLIPS의 메인 루틴의 이름은 "clipsmain"이고 이는 "clipsmain.c"라는 화일에 있다. 이 화일을 컴파일한 "clipsmain.o"를 로드시킬 화일로 하고, main 루틴인 "clipsmain"으로부터 호출되는 각종 루틴이 들어 있는 화일들을 컴파일하여 /usr/lib/libclips.a에 아카이브 형태로 넣어둔다. 한편 CLIPS의 모든 명령어에대한 KCL의 연결점은 메인 루틴 하나이므로 이러한 인터페이스를 위의 형식에 따라 정의하면 다음과 같다.

```
(load-foreign "clipsmain.o"
              (clips-interface (string "clipsmain") (string int))
              :library ("clips" "m"))
```

여기서 clips-interface는 KCL에서 사용하는 함수 이름이고 "m"은 CLIPS에서 원래 사용하는 /usr/lib/libm.a 라이브러리이다.

(2). KCL에서 사용하는 CLIPS 함수 정의

앞에서 살펴본 바와같이 clips-interface를 정의하였지만 이러한 인터페이스를 통하여 KCL에서 CLIPS 명령어를 사용하기 위해서는 KCL에 CLIPS의 각 명령어에 해당하는 에뮬레이션 함수를 정의하여야 한다. 이 함수에서는, CLIPS로 연결될 때 입력으로 들어갈 문자열과 그 문자열의 크기를 정하여 clips-interface 함수를 호출하게 되며, clips-interface의 리턴값인 문자열을 각 명령어에 맞게 처리하게 된다. 예를 들어 CLIPS 명령어인 assert를 KCL에서 사용하기 위하여 아래와 같은 함수를 KCL에 정의하였다.

```
(defmacro assert (in-string)
    '(setq *result-buffer*
         ,(clips-interface
              (princ-to-string '(assert ,in-string))
              (length princ-to-string '(assert ,in-string)))))
```

CLIPS에서 사용자의 입력을 기다리는 명령어나 다음 동작을 위해서 사용자

에게 메세지를 보내는 명령어 등을 제외하고는 표준 출력 스트림으로 출력되는 모든 것을 문자열화 하여 리턴값으로 KCL에 보내도록 구현되어 있다. 그러므로 KCL에 정의된 각 함수에서 *result-buffer*를 적절히 이용하여 원하는 형태의 출력을 얻어야 한다. 현재 이부분을 KCL에서 표준출력장치로 출력하는 기능 정도로 구현하였지만 실제로 이용할 때는 각 사용자의 환경에 맞게 수정해서 사용하면 될 것이다. 예를 들면 assert 명령어에서 리턴값이 null이면 true로 하고 아니면 nil로 하는 방법도 있다. 그러나 이런 변화는 사용자의 요구에 따라 달라질 수 있으므로 본 연구에서는 취급하지 않았다.

현재 위와같이 정의하여 설치된 함수의 이름은 다음과 같다[부록1].

defrule, reset, clear, run, save, load, save-fact, load-fact, help, facts, rules, agenda, pprule, excise, undeffacts, batch, system, deffacts, dribble-off, memused, release-mem, conserve-mem, assert, retract, list-deffacts, ppdeffacts, watch, unwatch, match, dribble-on

다. CLIPS의 수정

(1). KCL과의 연결 부분

KCL에서 CLIPS의 명령어를 사용할 때 항상 호출되는 루틴이 CLIPS의 메인 루틴인 clipsmain이다. 이 루틴을 간단히 기술하면 아래와 같다.

```
clipsmain ()
        { init-clips(); version print; command_loop(); }
```

여기서 제일 먼저 수정해야될 부분이 KCL에서 넘어오는 매개 변수를 처리하는 부분이다. CLIPS는 인터프리터 형태이기 때문에 입력을 표준 입력 스트림으로부터 받아들이고 있다. 그러나 KCL과 연결하면 KCL로부터 받는 매개 변수

에 입력이 들어있으므로 이를 다른 루틴에서 사용할 수 있게끔 표준 입력 스트림과 자료 구조가 같은 FILE이라는 스트림 형태로 만들어야 한다. 한편 KCL로 넘어가는 리턴값도 CLIPS가 표준 출력 스트림으로 보내는 것을 넘기게 되므로 이를 위한 스트림 구조도 만들어야 한다. 이를 위해 FILE 형태로 lispinbuf와 lispoutbuf를 만든 뒤 매개 변수로 넘어온 문자열과 정수를 연결하고 리턴값으로 lispoutbuf의 문자열 포인터(pointer)를 넘겼다. lispoutbuf의 문자열이 nil이거나 포인터 값이 0일 경우 정상적으로 작업이 끝난 경우이므로 "T"를 리턴하였다.

문자열로 넘어오는 입력 자료가 KCL에서 처리된 것이기 때문에 전부 대문자로 표시되어 있어 CLIPS에서는 사용할 수가 없다. 이를 처리하기 위해 대문자를 소문자로 변경하는 부분을 추가하였는데 이 때문에 대문자 자료는 사용할 수 없게 되었다. format이나 printout 명령어에서 사용하는 문자 ' " '를 문자열 속에 포함시켜 전달하기 위하여 KCL에서 ' " '앞에 ' \ '를 추가하여 ' \" ' 형태로 입력해야 하므로 ' \ ' 문자를 제거하는 부분을 추가하였다. 한편, CLIPS에서는 입력의 제일 마지막 문자가 반드시 ' \n '이어야 하는데 전달되는 문자열에는 마지막에 ' \n ' 문자가 없으므로 이를 추가하였다.

init_clips 루틴은 CLIPS가 로드될 때 한번만 수행되는 루틴인데 그대로 두면 KCL에서 CLIPS 명령어를 사용할 때마다 수행되므로 이를 맨 처음에만 수행되게끔 수정하였으며, CLIPS 버전을 출력하는 부분은 제거하였다. 수정된 후의 clipsmain 루틴을 간단히 기술하면 아래와 같다.

```
static int init;
FILE lispinbuf, lispoutbuf, *sstdin, *sstdout;
char *clipsmain(p, cc)
   char *p; int cc;
   { lispinbuf_cnt = cc; lispinbuf._ptr = p;
      if (isupper) lispinbuf._ptr[i] += 'a' - 'A';
      remove ' \ ';
      strcat(lispinbuf._ptr, " \n ");
      if (init == 0) { init_clips(); init = 1 }
```

```
        command_loop();
        if (lispoutbuf._ptr == NULL || *lispoutbuf._ptr == "")
            *lispoutbuf._ptr = "T";
        return(lispoutbuf._ptr)
    }
```

여기서 *sstdin과 *sstdout은 CLIPS 명령어 중 입출력 관련 명령어인 read, readline, format, printout, fprintout을 처리하기 위한 임시 버퍼이다. 위에서 FILE 구조로 선언한 스트림들은 헤드 화일인 clips.h에 extern 변수로 선언하여 공동으로 이용하게 된다.

한편, clipsmain에서 호출하는 command_loop 루틴은 간단히 기술하면 아래와 같다.

```
command_loop()
  { while(TRUE)
        { 초기값 set; prompt print;
          gettoken("stdin")
            { if error { errprint; flush_term();}
                route_command() }
        }
  }
```

CLIPS에서는 명령어 단위로 CLIPS 문장을 처리한 후 KCL로 리턴해야 되므로 while loop은 필요없게 되며 따라서 프롬프트를 출력하는 부분도 필요없게 된다. 초기값은 명령어를 수행할 때마다 필요한 부분이므로 그대로 둔다. 처음의 토큰이 에러 상황일 때 flush_term()을 하여 표준 입력 스트림에 남아 있는 자료를 지우게 되는데 KCL과 연결되면 표준 입력 스트림이 아니고 lispinbuf이므로 이 버퍼를 지워야 된다. 그러나 flush_term()이 다른 곳에서 사용될 때는 표준 입력 스트림을 지워야 되는 경우도 있으므로 이를 구분해 줄 필요가 있다. 원래 flush_term()은 매개 변수가 없는데 이를 수정하여 지워야 될 버퍼를 매개 변수로 받을 수 있게 하였다. 이렇게 해서 수정된 command_loop을 간단히 기술하면 다음과 같다.

```
command_loop()
   { 초기값 set;
      gettoken("stdin")
         { if error { errprint; flush_term("stdin");}
            route_command() }
   }
```

(2). 입출력 처리

CLIPS에서 제공하는 입출력 시스템은 아주 유연성이 있다. 원래 강한 이식성을 목표로 만든 시스템이기 때문에 여러가지 복잡한 입출력 장치에 대해 독립적으로 수행할 수 있게끔 되어 있다. CLIPS 내에서 외부 입출력 장치와는 상관없이 호출할 수 있는 표준 입출력 함수를 두어 일관성을 유지하고 이들 입출력 함수는 다시 환경에 따라 연결함수(router)에 의해 실제 입출력 장치와 관련이 있는 함수와 연결된다.

CLIPS에서 사용하는 표준 입출력 함수는

```
cl_print(log_name, str), cl_getc(log_name), cl_ungetc(ch, log_name), cl_exit(num)
char *log_name, *str;       int ch, num;
```

이고 연결함수는

```
add_router(router_name, priority, query_fun, print_fun, getc_fun, ungetc_fun, exit_fun)
```

이다.

add_router는 실제 사용되는 입출력 장치의 종류에 따라 혹은 특수하게 처리하고자 하는 입출력이 있으면 그에 맞는 함수를 묶어 연결함수 리스트에 그 정보를 넣어두는 역할을 한다. 실제 CLIPS에서 add_router를 통해 연결함수 리스트에 들어가는 정보는 다음과 같다.

("fileio", 0, findfile, fileprint, filegetc, fileungetc, fileexit)

("whelp", 10, find_help, print_help, getc_help, ungetc_help, NULL)

("batch", 20, find_batch, NULL, getc_batch, ungetc_batch, exit_batch)

("trace", 20, find_trace, print_trace, getc_trace, ungetc_trace, exit_trace)

("str_io", 90, str_fnd, NULL, str_getc,str_ungetc, NULL)

이러한 연결함수 리스트의 정보와 표준 입출력 함수는 논리이름(logical name)에 의해 서로 연결된다. 논리이름의 종류는 다음과 같다.

"wclips" : CLIPS prompt를 출력할 때 사용

"wdialog" : CLIPS가 사용자에게 보내는 메세지 출력할 때 사용

"wdisplay" : 사용자가 요청한 정보를 출력할 때 사용

"werror" : error 메세지 출력할 때 사용

"wtrace" : 모든 trace(watch) 관련 정보를 출력할 때 사용

"wagenda" : agenda를 출력할 때 사용

"whelp" : help 기능에 의한 정보를 출력할 때 사용

"stdin" : 사용자 입력을 받아들일 때 사용

"stdout" : printout, fprintout, format를 처리할 때 사용

위에서 언급한 표준 입출력 함수, 연결함수, 논리이름 등에 대한 이해를 돕기위해 한가지 예를 들어 보기로 한다. CLIPS 수행중 에러 상황이 발생하여 그 메세지를 사용자에게 전달하고 싶으면 메세지 내용을 문자열 str에 넣고 표준 입출력 함수 cl_print("werror", str)을 호출한다. cl_print 함수는 연결함수 리스트를 우선순위 순서대로 찾아 각각의 query_fun(findfile, find_help 등)을 수행한다. 각각의 query_fun은 자기가 속한 연결함수가 "werror"을 처리할 수 있는가를 판

단하여 TRUE 또는 FALSE를 리턴한다. 즉 제일 우선순위가 높은 "fileio" 연결 함수의 query_fun인 findfile 함수부터 "werror"에 관한 처리 여부를 조사하는데 실제로 이 함수에서 TRUE가 리턴된다. 만약 FALSE이면 다음 연결함수인 " whelp"의 find_help가 수행된다. 이런 식으로 TRUE가 리턴될 때까지 수행된다. 이렇게 해서 "werror"을 처리할 연결함수가 "fileio"임을 알게되면 cl_print에서는 찾은 연결함수의 print_fun인 fileprint를 수행시킨다. fileprint는 "werror"를 수행시키기 위한 실제 출력 함수와 실제 출력 스트림을 찾아 출력을 하게된다. 이 경우 출력 함수는 시스템 라이브러리인 printf이고 출력 스트림은 표준 출력 스트림인 stdout이 된다. CLIPS의 입출력 시스템에 대한 상세한 사항은 참고 문헌 [CLA87]을 참조하기 바란다.

이제 이러한 CLIPS의 입출력 시스템을 KCL과 연결시키기 위해서 어떻게 수정하였는지를 입력과 출력 부분으로 나누어 살펴보기로 한다.

(가). 입력 관련 부분 수정

입력은 크게 두 곳으로부터 오게 되는데 그 하나는 표준 입력 스트림인 stdin이고 다른 하나는 화일이다. 이 중에서 화일일 경우에는 아무런 문제가 없으므로 stdin인 경우에만 수정하면 된다. KCL과 연결되면서 입력이 stdin이 아닌 버퍼로부터 들어오므로 stdin으로부터 입력을 받아들이는 부분을 버퍼로 받아들이게 수정해야 된다. 그러나 read나 readline과 같은 명령어는 사용자의 입력을 요구하는데 이 경우의 입력은 KCL로부터 넘어온 버퍼가 아닌 표준 입력 스트림으로부터 들어오게 된다. 버퍼로 대치해야 하는 stdin의 경우와 이 경우를 구별하기 위하여 새로운 논리이름 "sstdin"을 만들어 사용하였다. 이러한 수정 사항을 부분별로 좀더 상세히 기술하면 다음과 같다.

o 연결함수(router) 관련 부분

- query_fun(findfile, find_help, find_trace 등)에 새로운 논리이름 "sstdin"을 추가한다. 그리고 실제적인 스트림의 종류를 알려주는 find_fptr 함수에도 "sstdin"을 추가한다.
- getc_fun 중 str_getc는 문자열을 취급하고 getc_batch는 화일을 취급하므로 수정할 필요가 없다.(화일이 stdin인 경우는 없는 것으로 하였다.) getc_help와 getc_trace는 표준 입력 함수인 cl_getc를 이용하고 cl_getc는 결국 filegetc를 이용하므로 수정하지 않아도 된다. 단지 getc_help에서 논리이름 "stdin"은 "sstdin"으로 대치해야 된다. help 기능에서 입력이 필요한 경우란, 사용자가 원하는 정보에 관한 index를 입력하면 관련된 정보를 보여 주는 경우이거나, 출력해야 되는 정보가 한 화면보다 많아서 사용자로 하여금 입력을 받아서 (일반적으로 space나 return key 임) 다음 정보를 출력하는 경우이므로 이러한 입력은 버퍼가 아닌 실제 표준 입력 스트림으로부터 받아야 한다. filegetc에서는 시스템 라이브러리 함수인 getc를 이용해서 입력을 받아들이는데 이를 KCL로부터 넘어온 버퍼로부터 입력을 받아들이게끔 새로운 함수 buf_getc를 만들어 대치하였다. 단 "sstdin"인 경우에는 기존의 getc를 그대로 사용하였다.
- ungetc_fun도 getc_fun과 같은 방법으로 수정하였다.

o read, readline, help 명령어와 관련된 루틴에서 "stdin"인 경우 "sstdin"으로 대치하였다.

o parser 부분에서 스트림 종류를 조사하는 곳에 "sstdin"을 추가하였다.

o flush_term 함수는 에러 상황일 때 입력 스트림을 지워 다음 입력을 받아들이는 준비를 하는 함수인데 KCL과 연결될 때 버퍼로 대치되는 기존의 "stdin"의 경우는 지워야 할 필요가 없고, "sstdin"인 경우만 지워야 하므로 이를 위해 수정을 하였다.

(나). 출력 관련 부분 수정

출력은 표준 출력 스트림(stdout)으로 나가는 것이 있고 화일로 나가는 것이 있다. 화일로 나가는 출력인 경우에는 수정할 필요가 없고 표준 출력 스트림으로 나가는 것은 버퍼로 받아 그 주소를 KCL로 넘겨야 한다. 표준 출력 스트림으로 나가는 출력에도 여러 종류가 있다. 앞에서 언급한 논리 이름에 따르면 "wdialog", "wdisplay", "werror", "wtrace", "wagenda", "whelp", "stdout" 등이 모두 표준 출력 스트림으로 출력하기 위한 것이다. 이 중에서 "stdout"은 printout, fprintout, format 등의 함수를 처리할 때 사용되는데 이경우의 출력은 사용자와의 대화를 위한 경우가 많으므로 버퍼에 넣어 KCL로 전달하기보다 바로 표준 출력 스트림으로 내보내야 한다. "whelp"의 경우도 마찬가지로 표준 출력 스트림으로 출력해야 한다. "werror"의 경우에 에라 메세지를 KCL로 넘길것이냐의 판단은 사용환경에 따라 달라질 수 있다. 본 연구에서는 버퍼에 넣어 KCL로 넘기게 구현하였다.

CLIPS의 수행이 완전히 끝나버리는 exit 함수는 CLIPS 명령어인 exit를 사용할 경우나 시스템 오류에 의한 경우에 호출된다. exit 명령어에 의한 경우는 KCL에서 CLIPS의 exit 명령어를 제공하지 않으므로 발생하지 않지만 시스템 오류에 의한 exit는 막을 수가 없다. 이러한 경우 KCL까지 끝나버리게 된다. "werror"에 의한 출력 처리를 버퍼로 하므로 exit에 의해 갑자기 KCL까지 끝나버리면 exit 이유를 알 수 없게 된다. 이를 방지하기 위하여 exit하기 전에 "werror"에 의해 나가는 출력은 버퍼가 아닌 표준 출력 스트림으로 나가게 하였다.

이러한 출력에 관한 수정 사항을 부분별로 좀더 상세히 기술하면 다음과 같다.

o 연결함수 부분

- query_fun인 findfile, find_trace 등에 새로운 논리 이름 "sstdout"을 넣어 표준 출력 스트림으로 출력하는데 사용한다. 스트림의 종류를 알려주는 find_fptr 함수에도 "sstdout"을 추가한다.
- print_fun 중 print_help와 print_trace는 다시 표준 출력 함인 cl_print를 사용하는데 이 때 "stdout"을 "sstdout"으로 수정한다. fileprint에서는 시스템 라이브러리 함수인 fprintf를 이용해서 출력을 한다. 출력 스트림이 stdout인 경우 출력할 문자열을 버퍼에 넣어 KCL로 보내기 위해 새로운 함수인 bufprintf를 만들어 대치하였다. 출력 스트림이 sstdout인 경우는 표준 출력 장치로 출력하게 였다.

o printout, fprintout, format 명령어와 관련된루틴에서 "stdout"인 경우 "sstdout"으로 대치하였다.

o help 명령어와 관련된 루틴에서도 "stdout"인 경우 "sstdout"으로 대치하였다.

o 시스템 오류에 의해 exit될 때 그 원인을 알려 주는 에러 메시지는 "sstdout"을 이용하여 표준 출력 스트림으로 출력되게 하였다.

라. KCLIPS의 설치

시험 시스템인 SUN3 워크스테이션에 확장된 KCL과 CLIPS를 통합하여 KCLIPS를 구현하였다. KCL에 CLIPS를 통합시키는 과정에 CLIPS의 init_hash, format, kill 함수의 이름이 KCL과 동일하여 에러가 발생하였기 때문에 이들의 이름을 각각 iinit_hash, fformat, kkill로 수정하였고, CLIPS에서 external 변수로 사용하는 token이 KCL에서도 동일한 이름으로 사용되고 있기 때문에 CLIPS의 token을 ttoken으로 수정하였다. 한편, CLIPS에서 제공하는 편집기 기능은 KCL과 연결될 경우 필요없기 때문에 제거하였다. 이를 위해 헤더 화일인 clips.h의 CLP_

EDIT를 0으로 하고 편집기 관련 화일을 링크시키지 않았다.

CLIPS의 메인 루틴인 clipsmain.c 화일은 컴파일하여 사용자 디렉토리에 넣어두고 나머지 화일은 컴파일하여 아카이브로 만들어 이를 /usr/lib/libclips.a로 하였다. KCL에서 정의하여야 하는 CLIPS 관련 함수와 load-foreign 함수는 /usr/local/kclips/kclipsinit에 넣어 두었다. 이렇게 한 뒤에 사용자가 KCL을 수행시키고 (load "/usr/local/kclips/kclipsinit")하면 KCLIPS 환경이 된다. CLIPS의 메인 루틴인 clipsmain.o도 /usr/local/kclips 디렉토리에 넣어두고 load-foreign 함수에서 그 경로 이름을 사용하였다

2. KCLIPS의 시험

이상과 같이 구현된 KCLIPS의 검증을 위하여 여러가지의 KCL-CLIPS 혼용 프로그램을 작성하여 수행시킨 결과 아주 빠른 속도로 실행되었다. 아래 보기에 LISP 문장과 CLIPS 문장이 뒤섞여 있는 전형적인 프로그램예를 제시하였다. 이중 앞의 두 프로그램은 KCL 수행시 초기 바인딩이 되어도 수행가능한 예이며, 뒤의 두 프로그램은 규칙이 수행될때 run-time 바인딩이 필요한 경우이다. 현재까지의 구현은 KCL 수행시 행하여지는 초기 바인딩만 되어있다.

```
% kclips
(setq    a    3)
(setq    b    2)

(assert  '(add ,a 3))
(assert  '(add a 3))

(assert  '(I have ,(+ a b) books))
(assert  '(my brother has ,(- a 2) books))
(assert  '( ,(+ a 10)))

(defrule  '(how-many-books
              (mine ?my)
```

```
    =>
              (bind   ?b  (-  ?my  ,a))
              (bind   ?s  ,(+  a  b))
              (fprintout  t  \"brothers  =  \"?b \"sisters  =  \"?s  crlf))

(assert   '(mine  10))
(run    a)
```

```
%kclips
(setq    family1    '(lim     lim1     park1))
(setq    family2    '(lim1    kim1     lee1))
(setq    family3    '(park1   kang     cho))

(setq    limit     5)

(defrule   '(find-ancesters
               ?start  <-  (startfact  ?name)
               (?name      ?father   ?mother)
               =>
               (retract   ?start)
               (format    t  "%s  and  %s  are  ancesters  of  %s%n"    ?father
                ?mother   ?name)
               (assert    (startfact     ?father))
               (assert    (startfact     ?mother)))

(assert   family1)
(assert   family2)
(assert   family3)

(defun     ancesters    (name)
             (assert    '(startfact  ,name))
             (run       limit))

(defun     print-working-memory    ()
             (print    (facts))

(print-working-memory)
(ancestor      'lim)
```

```
%kclips
(defun    myfn(x)
```

```
(if  (< x 0) (- x 20) (+ x 10)))

(defrule '(rule1
  (test-value ?y)
 =>
  (bind ?r (myfn ?y))
  (fprintout t \"result is \" ?r))

(assert (test-value 30))
(assert (test-value -10))

(run)
```

```
%kclips
(defun next-in-straight (v2 v1)
  (or (= v2 (+ v1 1))
    (and (= v2 2) (= v1 14))))

(defrule '(rule2
  (card ?suit1 ?face1 ?value1)
  (card ?suit1 ?face1 ?value2
 =>
  (..............)
              .
  ( . )))
```

제 6 절 결 론

본 연구는 전문가 시스템 개발에 편리한 새로운 구축도구(KCLIPS) 개발에 대하여 기술하였다. 이 도구는 전문가 시스템 개발시 LISP의 다양한 함수와 CLIPS의 함수를 동시에 통일된 방법으로 사용할 수 있게 하여주며 CLIPS내부에서 LISP 함수를 사용할 수 있게 하여준다. 또한 KCLIPS는 속도에 영향을 주는 추론 부분은 C 언어로 작성된 CLIPS를 사용하여 빠른 속도를 제공하고 있으며, 필요한

기능을 쉽게 확장할 수 있도록 되어있어 개발시의 편리성과 사용시의 실용성을 모두 제공하고 있다.

이러한 환경구축을 위하여 KCL과 CLIPS의 차이점을 조사하였고, KCL과 CLIPS를 연결시킬때 고려하여야 할 사항, 수정되어야 할 부분, 추가되어야 할 부분등을 조사 분석하였으며, CLIPS의 소스코드를 분석 수정하였다. 현재 KCLIPS는 SUN3 워크스테이션에 구현되어 사용되고 있으며 검증을 위하여 LISP과 CLIPS의 혼합문으로 구성되어있는 프로그램을 작성하여 시험하였다.

현재 구현되어 있는 KCLIPS에서는 CLIPS의 수행결과가 스트링 타입으로 KCL에 그대로 전달되도록 되어있어 LISP에서 이 결과를 이용하여 심볼조작을 원할 경우에는 적절한 조치를 취하여야 한다. 이를 위하여 인터페이스 프로그램을 미리 작성하여 언제나 사용자가 수정할 수 있도록 따로 제공하고 있다. 또한 지금까지 구현되어 있는 KCLIPS는 KCL문장을 수행시킬 때 바인딩이 되는 초기 바인딩이므로 규칙의 조건이 만족되어 수행될때 이루어 지는 런 타입 바인딩은 계속적으로 좀더 많은 시간을 들여 구현하여야 할 것으로 생각된다.

참 고 문 헌

[CKC89] **최 창근, 인공지능을 이용한 고도의 구조해석/설계용 전문가 시스템의 개발, 건설분야 '88특정연구 요약집, 건설부, 과학기술처**, 1989.

[CLA87] CLIPS Advanced Programming Guide, Artificial Intelligence Section, NASA/Johnson Space Center, October 14, 1987.

[CLB87] CLIPS Basic Programming Guide, Artificial Intelligence Section, NASA/ Johnson Space Center, October 14, 1987.

[CLP87] CLIPS User's Guide, Artificial Intelligence Section, NASA/Johnson Space Center, October 14, 1987.

[ECL88] Enhanced Common LISP Production System User's Guide and Reference, First Edition, IBM, August 1988.

[GUY84] Guy L. Steele, Common Lisp : The Language, Digital Press, 1984.

[HAY83] Frederick Hays-Roth, Building Expert Systems, Addison Welsley, 1983.

[KCL84] Taiichi Yuasa and Masami Hagiya, Kyoto Common Lisp Report, 1985.

[OPS85] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin, Programming Expert Systems in OPS5, Addison-Wesley, 1985.

[YSO89] Yongseo Oh, An Implementation of Foreign Language Interface on Common LISP, Master's Thesis, Department of Computer Science,

KAIST, 1989.

[EKS88] Eunkang Song, A Study on Implementation of Object Oriented Language and the Design of Rule-based System, Master's Thesis, Department of Computer Science, KAIST, 1989.

[WAT86] D. A. Waterman, A Guide to Expert System, Addison-Welsley, 1986.

부록 1 : KCL과 CLIPS의 인터페이스 프로그램(kclipsinit)

```
(load "/usr/local/kcld/unixport/alien2.lsp")
(load-foreign "/usr/local/kclips/clipsmain.o"
              (clips-interface (string "clipsmain") (string int))
              :library ("clips" "m"))

(defun clips-lisp-call (stream sunchar char)
  (declare (ignore subchar char))
  (eval (read stream t nil t)))

(set-dispatch-macro-character #\# #\! #'clips-lisp-call)

(defmacro assert (in-string)
  `(setq *result-buffer*
  ,(clips-interface
    (princ-to-string `(assert ,in-string))
    (length (princ-to-string `(assert ,in-string))))))

(defmacro facts ()
  `(setq *result-buffer*
  ,(clips-interface
       (princ-to-string `(facts))
       (length (princ-to-string `(facts))))))

(defmacro defrule (&body all)
  `(setq *result-buffer*
  ,(clips-interface
       (princ-to-string `(defrule ,@all))
       (length (princ-to-string `(defrule ,@all))))))

(defmacro clear ()
  `(setq *result-buffer*
  ,(clips-interface
       (princ-to-string `(clear))
       (length (princ-to-string `(clear))))))

(defmacro reset ()
  `(setq *result-buffer*
  ,(clips-interface
       (princ-to-string `(reset))
       (length (princ-to-string `(reset))))))

 (defmacro run (&optional in)
    (if (eq in nil)
       `(setq *result-buffer*
```

```
            ,(clips-interface
               (princ-to-string '(run))
               (length (princ-to-string '(run)))))
         '(setq *result-buffer*
            ,(clips-interface
               (princ-to-string '(run ,in))
               (length (princ-to-string '(run ,in)))))))

(defmacro save-facts (in-string)
   '(setq *result-buffer*
   ,(clips-interface
      (princ-to-string '(save-facts ,in-string))
      (length (princ-to-string '(save-facts ,in-string))))))

(defmacro load-facts (in-string)
   '(setq *result-buffer*
   ,(clips-interface
      (princ-to-string '(load-facts ,in-string))
      (length (princ-to-string '(load-facts ,in-string))))))

(defmacro rules ()
   '(setq *result-buffer*
   ,(clips-interface
         (princ-to-string '(rules))
         (length (princ-to-string '(rules))))))

(defmacro agenda ()
   '(setq *result-buffer*
   ,(clips-interface
         (princ-to-string '(agenda))
         (length (princ-to-string '(agenda))))))

(defmacro pprule (in-string)
   '(setq *result-buffer*
   ,(clips-interface
      (princ-to-string '(pprule ,in-string))
      (length (princ-to-string '(pprule ,in-string))))))

(defmacro list-deffacts ()
   '(setq *result-buffer*
   ,(clips-interface
         (princ-to-string '(list-deffacts))
         (length (princ-to-string '(list-deffacts))))))

(defmacro ppdeffact (in-string)
   '(setq *result-buffer*
   ,(clips-interface
```

```
      (princ-to-string `(ppdeffact ,in-string))
      (length (princ-to-string `(ppdeffact ,in-string))))))

(defmacro watch (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(watch ,in-string))
      (length (princ-to-string `(watch ,in-string))))))

(defmacro unwatch (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(unwatch ,in-string))
      (length (princ-to-string `(unwatch ,in-string))))))

(defmacro matches (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(matches ,in-string))
      (length (princ-to-string `(matches ,in-string))))))

(defmacro dribble-on (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(dribble-on ,in-string))
      (length (princ-to-string `(dribble-on ,in-string))))))

(defmacro dribble-off ()
   `(setq *result-buffer*
   ,(clips-interface
           (princ-to-string `(dribble-off))
           (length (princ-to-string `(dribble-off))))))

(defmacro mem-used ()
   `(setq *result-buffer*
   ,(clips-interface
           (princ-to-string `(mem-used))
           (length (princ-to-string `(mem-used))))))

(defmacro release-mem ()
   `(setq *result-buffer*
   ,(clips-interface
           (princ-to-string `(release-mem))
           (length (princ-to-string `(release-mem))))))

(defmacro conserve-mem (in-string)
   `(setq *result-buffer*
```

```
   ,(clips-interface
      (princ-to-string `(conserve-mem ,in-string))
      (length (princ-to-string `(conserve-mem ,in-string))))))

(defmacro retract (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(retract ,in-string))
      (length (princ-to-string `(retract ,in-string))))))

(defmacro excise (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(excise ,in-string))
      (length (princ-to-string `(excise ,in-string))))))

(defmacro undeffacts (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(undeffacts ,in-string))
      (length (princ-to-string `(undeffacts ,in-string))))))

(defmacro batch (in-string)
   `(setq *result-buffer*
   ,(clips-interface
      (princ-to-string `(batch ,in-string))
      (length (princ-to-string `(batch ,in-string))))))

(defmacro deffacts (&body all)
   `(setq *result-buffer*
   ,(clips-interface
           (princ-to-string `(deffacts ,@all))
           (length (princ-to-string `(deffacts ,@all))))))
```

부록 2 : 수정된 CLIPS의 메인 루틴 프로그램(clipsmain.c)

```
#include <stdio.h>
#include "clips.h"
#include <ctype.h>

extern char *strcat();
static int sblim_init;
FILE lispbuf;
FILE lispoutbuf;
FILE *sstdin;
FILE *sstdout;
char *clipsmain(sblim_p, sblim_cc)
 char *sblim_p;
 int sblim_cc;
  {
    int i, j=0;
    char *sblim_q;

/* for lisp, upper case char => lower case char */
/* delete char '\' from input                    */
    for (i = 0; i < sblim_cc; i++)
        {
           if (isupper(sblim_p[i])) sblim_p[i] += 'a' - 'A';
           if (sblim_p[i] == '\') j++;
               else sblim_p[i - j] = sblim_p[i];
        }
    sblim_cc = sblim_cc -j;

/* for lisp, insert '\n' in the end of input */
    if ((sblim_q = (char *)malloc(sblim_cc + 1)) == NULL)
          exit("malloc error\n");
    bzero(sblim_q, sblim_cc + 1);
    strncpy(sblim_q, sblim_p, sblim_cc);
    strcat(sblim_q, "\n");

    lispbuf._cnt = sblim_cc + 1;
    lispbuf._ptr = sblim_q;
    lispbuf._base = sblim_q + sblim_cc + 1;
    lispoutbuf._cnt = 0;
    lispoutbuf._ptr= NULL;
    if (sblim_init == 0)
       {
        init_clips();
        sblim_init = 1;
       }
```

```
 command_loop();
 free(sblim_q);
 if (lispoutbuf._ptr == NULL || *lispoutbuf._ptr == "") return("T");
 return(lispoutbuf._ptr);
}
```

```
#include <stdio.h>
#include "clips.h"
#include "ctype.h"

extern  char *malloc();
extern  int   free();
extern  char *strcat();

buf_getc()
{
 register int c;

 if(lispbuf._cnt <= 0) {
  c = -1;
  lispbuf._cnt = 0;
  lispbuf._ptr = lispbuf._base = NULL;
 } else {
  c = *lispbuf._ptr++ & 0377;
  if((--lispbuf._cnt)<=0) {
   lispbuf._ptr = NULL;
   lispbuf._base = NULL;
   lispbuf._cnt = 0;
   }
  }
 return(c);
}

buf_fgetc()
{
 return(buf_getc());
}

char *
buf_fgets(s, n)
char *s;
{
 register c;
 register char *cs;

 cs = s;
 while (--n>0 && (c = buf_getc()) != EOF) {
  *cs++ = c;
  if(c=='\n')
```

```
    break;
  }
  if(c == EOF && cs==s)
    return(NULL);
  *cs++ = '\0';
  return(s);
}


buf_ungetc(c)
{
  if(c == EOF ||
        lispbuf._ptr == NULL || lispbuf._base == NULL)
    return (EOF);

  if(lispbuf._ptr == lispbuf._base)
    if(lispbuf._cnt == 0)
      lispbuf._ptr++;
    else
      return (EOF);

  lispbuf._cnt++;
  *--lispbuf._ptr = c;

  return (c);
}

bufprintf(sblim_str)
char *sblim_str;
{
    char *sblim_p;
      int   i, j;
            j = strlen(sblim_str);
   if (lispoutbuf._ptr == NULL)
                  { if((sblim_p = (char *)malloc(j + 1)) == NULL)
                           exit("malloc error");
                     bzero(sblim_p, j + 1);
                     strcpy(sblim_p, sblim_str);
                  lispoutbuf._ptr = sblim_p;
         lispoutbuf._cnt = j;
         return(1); }
            i = strlen(lispoutbuf._ptr);
   if((sblim_p = (char *)malloc(j + i + 1)) == NULL)
     exit("malloc error");
   bzero(sblim_p, j + i + 1);
   strcpy(sblim_p, lispoutbuf._ptr);
   strcat(sblim_p, sblim_str);
```

```
            free(lispoutbuf._ptr);
  lispoutbuf._ptr  =  sblim_p;
  lispoutbuf._cnt  +=  strlen(sblim_str);
}
```

부록 4 : KCLIPS Manual

1. KCLIPS 환경 구축

- o CLIPS source를 compile한 각종 object file을 /usr/lib/libclip.a에 archive 형태로 넣는다.

  ar r /usr/lib/libclips.a /usr/local/clips/source/*.o

- o ranlib /usr/lib/libclips.a 수행
- o clipsmain.o를 /usr/local/KCLIPS에 넣는다.
- o /usr/local/KCLIPS에 kclipsinit file을 넣어둔다.
- o KCL 하에서 (load "/usr/local/KCLIPS/kclipsinit") 를 수행한다. (1 - 2 분 정도 소요)

2. KCLIPS에서의 사용 방법

KCLIPS에서 CLIPS 명령어를 사용하는 방법은 원칙적으로 원래의 CLIPS 명령어 사용 방법과 동일하다. 단 CLIPS 문장 중에 KCL의 함수나 심볼을 사용하는 경우 초기 바인딩 혹은 런 타입 바인딩에 따라 사용 방법이 달라진다.

초기 바인딩의 경우 CLIPS 문장 중에 KCL 함수나 심볼을 사용할 때 반드시 ' #! '를 붙인다. 예를 들어

```
(setq a 3)
(setq b 5)
(assert (aa #!(+ 3 5))))
```

하면 (assert (aa 8))과 같은 결과이다.
런 타입 바인딩의 경우

```
(defrule testrule
```

```
           (assert (this is test))
    =>
           (assert (aa ($lisp (+ a b)))))     * a, b는 KCL의 심볼임
```

하면 나중에 실제로 rule이 fire되어 RHS(right hand side)가 수행될 때, 그 때의 KCL 심볼 a, b 값이 바인딩된다.

* $lisp(run-time binding)에 대한 기능은 아직 implement되지 않음

3. 주의점

o 문자열로 이루어진 자료 내에서도 대, 소문자를 구별하여 사용할 수 없다.

o CLIPS 명령어 중에 문자열을 사용하는 경우 " 앞에 \를 사용하여 \" 형태로 입력하여야 한다.

o 문자 "와 \는 데이타로 사용할 수 없다.

제 3 장 이식성을 고려한 Common Lisp Object System의 구현

제 1 절 서론

객체 중심 프로그래밍(Object Oriented Programming)은 지난 십여년간 다방면에 걸쳐서 연구되어 왔으며, 시뮬레이션 프로그래이나 시스템 프로그래밍, 그래픽스, 인공 지능 프로그래밍 등에서 이미 널리 사용되고 있다[MD86].

객체 중심 프로그래밍 형태는 70년대의 절차 중심 프로그래밍 형태(procedure-oriented programming style)에 이어 새로운 프로그래밍 형태로서 프로그래밍 언어 분야에서 많이 연구되고 있다[Pa86]. 객체 중심 프로그래밍을 위하여 Actor나 SIMULA, Smalltalk등과 같은 많은 객체 중심 언어들이 새로 고안되었으며, Flavor, C++, Object Pascal등과 같이 기존의 언어들이 객체 중심 프로그래밍의 지원을 위하여 확장 되어졌다[Sa89].

객체 중심 프로그래밍를 위한 여러 시스템들은 프로그래머에게 정보 은폐, 자료 추상화, 동적 결속(Dynamic Binding), 속성 상속등의 기능을 제공하므로서 프로그램의 신뢰도, 단위성(modulity), 유연성, 재사용성등을 증가 시켜준다[Pa86].

객체 중심 프로그래밍 시스템이 제공하는 정보 은폐의 기능은 인터페이스와 구현을 분리 시키고 단순화된 프로토콜만을 제공해 줌으로서 프로그램의 단위성과 신뢰도를 높여 준다. 사용할 객체들을 개념적으로 추상화하여 사용가능하게 하는 자료 추상화 기능은 프로그래머의 생산성을 높여 주며, 동적 결속은 프로그램이 수행중에 여러 형태의 객체들을 자유롭게 다룰 수 있게 해 줌으로서 프로그램의 유연성을 높여주며, 프로그램을 단계적이고 점진적으로 개발해 나갈수 있게 해준다. 객체들간의 계층구조 속에서 객체의 여러 속성들을 조합해주는 상속

기능은 프로그램의 중복성을 줄임으로서 전체 프로그램의 크기를 줄여 주며, 코드 세분화을 가능케하여 소프트 웨어 재사용의 측면에서도 큰 장점을 보여 준다.

이와 같이 많은 장점과 특징을 지닌 객체 중심 시스템은 특히 인공지능 분야에서 논리 언어나 기호 계산 언어등과 결합되어 전문가 시스템, 정보 검색, 지식 처리, 사용자 인터페이스등 여러 분야에서 중요한 위치를 차지하고 있다.

인공지능 분야에서 널리 사용되고 있는 LISP 언어에서도 많은 종류의 객체 중심 시스템들이 개발되어 왔고 현재도 개발및 개선 중에 있다. LISP 언어를 기반으로 하는 객체 중심 시스템들은 FLAVORS(MIT/Symbolics 1979)로 부터 LOOPS(Xerox PARC 1981), CommonLOOPS(Xerox PARC, 1985), Common Objects (Synder and Alen, 1986), ExperCommonLISP(1985), GLISP(Standford, G. S. Norvak, 1981), Object LISP(LMI), STROBE(Schlumberger-Doll Research), XLISP(David Bets), CommonORBIT등이 있다. 이러한 각 시스템들은 나름대로의 장단점을 보유하고 있고, 이미 많은 사람들이 사용하고 있으며, 이것을 사용하여 개발된 소프트 웨어 들이 많이 있다. 그러나 이러한 시스템들은 좋은 특성을 가짐에도 불구하고 LISP 언어 내에서 공유되지 못하고 있으므로 각각의 좋은 특징들을 포함하는 표준화의 필요성이 등장하게 되었다.

CLOS(Common Lisp Object System)은 이러한 필요성에 의해 제안되었으며 LISP의 표준화를 위해 만들어진 Common LISP의 한 부분으로 확립되었다. CLOS는 LISP을 기반으로 하는 객체 중심 시스템 중에서 특징적인 장점을 많이 가지고 있는 CommonLOOPS, Common Object, Object LISP의 피쳐들을 토대로 ANSI STANDARD COMMON LISP의 표준화를 제안하는 X3J13 위원회에서 연구 개발되어 1988년 6월에 Common LISP의 한 부분으로 확정 발표 되었다.

이와같이 LISP의 표준화를 위해 만들어진 Common LISP의 표준 객체 중심 시스템으로서, CLOS는 객체 중심 프로그래밍의 장점을 최대한 포함하고 있을

뿐만 아니라 사려깊게 연구되어 가능한 좋은 기능들을 일관성 있게 조합 하였으므로, 매우 우수하고 상세히 정의되어 있고, 효율적으로 구현가능하게 만들어져 있다. 또한 CLOS는 Common LISP의 타입 계층구조, 구문형태등과 상호 잘 부합 되도록 만들어졌기 때문에 기존의 Common LISP 프로그래머들에게 생소하지 않은 구문형태를 제공하고 있고 개념적으로 일관성을 유지시켜 준다.

이와같은 CLOS의 구현은 타입 계층구조의 부분적 확장등을 필요로 하므로 KCL이나 AKCL과 같은 기존의 Common LISP에서는 인터프리터의 수정을 요구한다. 이러한 일은 필요한 일이긴 하나 이미 구현된 많은 시스템들을 수정하기가 어렵고, 또한 거의 대부분의 CLOS 피쳐가 수정 없이 구현 가능 하며 객체중심 프로그래밍의 의미상 수행에 지장을 초래하지는 않으므로, 본 연구에서는 기존 인터프리터의 수정 없이 Common LISP 만을 사용하여 CLOS를 구현 하였다. 그러므로 이미 구현된 기존의 어떠한 Common LISP 시스템에서든지 사용 가능하며, 또한 인터프리터의 수정시에도 간단한 수정으로 접합시켜 확장 시킬수 있도록 구현 하였다. 또한 스페시피케이션이 확정되어 발표된지 얼마되지 않아 개발 단계의 몇몇 구현을 제외하고 아직 완벽한 구현이 없는 상태에서 대부분의 피쳐를 포함하는 CLOS를 구현하는 것은 CLOS의 빠른 실용화에도 큰 의의가 있다고 하겠다.

본 연구에서는 스페시피케이션에 따른 CLOS의 의미상의 분석과 여러 구현 알고리듬의 비교 분석 그리고 효율적 수행을 위한 내부 설계등에 중점을 두었으며 스페시피케이션의 미비점을 지적하고 새로운 피쳐등을 제안한다.

제 2 절 Common Lisp Object System의 특징

객체 중심 프로그램 개념은 근 30년에 걸쳐서 연구되어 이미 많은 부분들이 실용화 되어 왔다. 그러나 객체 중심 프로그램은 아직 연구되어야 할 많은 부분을 담고 있다. CLOS는 그 동안 연구되어온 객체 중심 프로그래밍 개념의 장점들을 비교적 일관성 있고 폭넓게 수용하고 있다.

1. 객체 중심 시스템의 구성요소

객체(object)는 자료(data)와 행동양식(behavior)이 결속된 실체(entity)로서 객체 중심 프로그램은 객체의 집합과 객체간의 통신(communication)으로서 구성된다. 객체 중심 시스템은 객체 중심 프로그래밍이 가능하도록, 객체의 정의, 객체간의 통신, 객체 관리등의 기능을 제공해주는 시스템이다. 이와같은 객체 중심 시스템의 기능은 대표적인 다음 네가지 기능으로 이루어 진다.

* 정보 은폐
* 자료 추상화
* 동적 결속
* 속성 상속

이러한 각 기능은 독립적으로 제공되지 않고 서로 간에 깊은 연관성을 가지고 종합적으로 제공 된다.

가. 정보 은폐

절차 중심 프로그래밍의 문제점 중 하나는, 한 변수를 의미상 관계 없는 여러 다른 프로시쥬어에서 수정 하는 것을 막지 못하므로서, 변수들의 상태 변화 추적이 모호하고 불안정하다는데 있다. 객체 중심 프로그래밍에서는 객체가 상태

변수(state variable), 혹은 상태 변수의 집합을 가지고 있고, 이의 변화를 배타적으로 주도하는 행동양식들과 깊이 관련되어 있어서, 외부에서는 객체내의 상태를 직접 변화 시킬수 없게 하여 준다. 그러므로 외부에서 그 객체의 상태를 참조하거나 수정하려 할 경우에는 메세지 전달 방식(message passing)등을 사용하게 하여 객체내의 상태가 부여 받은 의미를 일관되게 유지 하도록 하여 준다. 메세지 전달 방식은 그림 3-1에서 보는 바와 같이 객체를 하나의 활동적(active)인 실체로서 보고 원하는 조작(operation)을 메세지로서 전달하면 객체내의 행동양식 선택자(behavior selector)가 그 메세지에 해당하는 행동 양식을 골라 수행 시키도록 하는 것이다.

이러한 기능을 위해서는 객체 중심 시스템은 관련된 상태 변수들과 행동 양식들을 포장(encapsulation)하여, 객체와 객체간의 독립성을 철저하게 유지시켜 주어야 한다. 외부에서는 상태 변수를 직접 수정하지 못하도록 하고, 외부의 요구사항을 전달 받아 요구사항이 올바른것인지 검토하여 정당한 행동 양식에 의해 수행되도록 해 주어야 한다.

나. 자료 추상화

객체 중심 프로그램이 수행될때 객체는 기억 장소의 실제적인 영역을 차지하고있는 실현 객체(instance)로서 역할하게 된다. 실현 객체는 수행시 행동 양식에 의해 실제적으로 참조, 수정되는 변수의 값을 가지고 있다. 이러한 실현 객체들은 의미상 비슷한 종류끼리 모을 수 있는데, 이러한 집합을 객체 부류(class), 혹은 실현 객체의 구조를 대표한다는 의미에서 형판(template)라고 한다(그림 3-2). 이와같이 비슷한 실현 객체들의 집합을 객체 부류로 정의하고, 이 객체 부류에 대한 조작들을 기술할수 있게하는 것을 자료 추상화라고 한다.

하나의 객체 부류에 속하는 실현 객체들은 모두 같은 형태의 상태 변수 집

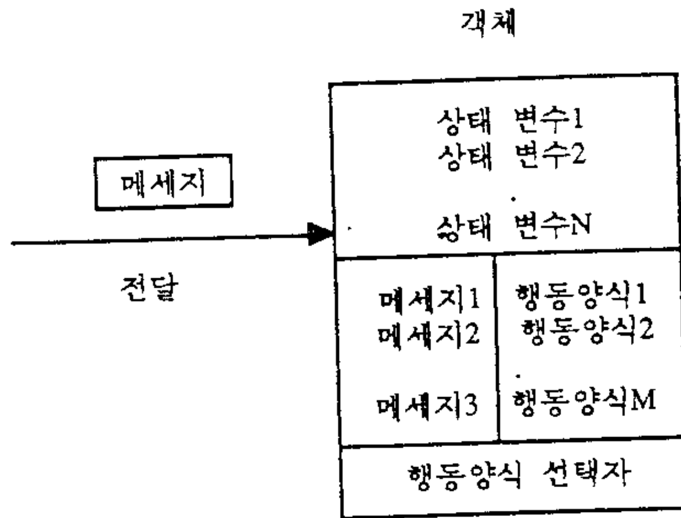


그림 3-1 객체와 메세지 전달

상태 변수	a b c d
행동 양식	behavior1 behavior2 . behaviorN

객체 부류

변수 이름	값
a b c d	'A' 'B' 'C' 'D'

실현 객체

그림 3-2 객체 부류와 실현 객체

합을 가지며 같은 행동 양식들을 사용한다. 프로그래머는 사용할 객체의 성질에 따라 객체 부류와 그에 따른 행동 양식을 정의하므로서 객체 중심 프로그래밍을 할 수 있게 된다. 정의된 객체 부류에 따라 생성된 실현 객체들은 서로간의 메세지 전달 방식등을 통해 목적하는 일을 수행하게 된다.

하나의 객체 부류는 기존의 객체 부류들을 포함하도록 정의 될 수 있다. 이때 포함된 객체 부류를, 포함한 객체 부류의 상위 객체 부류(superclass)라 하고, 포함한 객체 부류를, 포함된 객체의 하위 객체 부류(subclass)라 한다. 또한 인근한 상하위 객체 부류를 직접적 상위, 혹은 직접적 하위 객체 부류(direct super/subclass)라 한다. 하위 객체 부류는 상위 객체 부류로 부터 그 객체 부류의 성격을 확장 시켜 만들어 지게 되며 상위 객체 부류의 여러가지 성질들을 상속받게 된다. 이와같은 객체 부류의 계층구조는 고도의 자료 추상화를 가능하게 한다.

객체 중심 시스템은 프로그래머에게 객체 부류의 정의가 가능하도록 하며, 정의된 객체부류의 계층구조를 유지하고, 계층 구조에 따라 실현 객체의 생성이 가능하도록 해주며, 상속이 이루어 지도록 해야 한다.

다. 동적 결속

객체 중심 프로그램의 대표적인 조작(operation) 수행 방법은 메세지 전달 방식이다. 메세지 전달 방식은 어느 객체에 대하여 외부에서 조작을 필요로 할때는 필요한 조작을 메세지로서 그 객체에 전달하게끔 하는 것이다. 메세지를 전달받은 객체는 자신의 행동 양식 중에 적당한 것을 선택하여 수행 시킨다. 이때 하나의 메세지는 서로 다른 객체 부류의 실현객체들에게는 다른 의미를 가질수 있어서, 서로 다른 객체에서 서로 다른 행동 양식이 선택될 수 있다. 이와같은 것을 다형 허용성(polymorphism)이라 한다. 이 다형 허용성은 같은 형태의 행동

양식들이 서로 다른 객체 부류에 속할때 일관된 조작 형태를 제공해 줌으로 매우 편리하다.

하나의 실현 객체는 어느 특정 변수에 결속될 수 있고, 이 변수가 메세지를 전달 받는 객체로 사용되었다고 하면, 컴파일 시간에는 수행되어야 할 행동 양식을 알수 없다. 이것을 해결하기 위해서는 동적 결속이 필요하다.

일반적으로 객체 중심 프로그래밍에서 메세지 전달 방식은 다음과 같은 형태로 사용된다.

(send obj1 msg1)

send에 의해서 메세지 msg1은 객체 obj1에 전달된다. obj1은 전달된 메세지의 내용에 따라 적절한 행동 양식을 선택하여 수행한다(그림 3-1). 이렇게 선택되어 수행될 수 있는 행동양식을 단위 방법(method)라 한다.

위와 같은 메세지 전달 방식은 여러개의 객체에게 메세지를 보내는 다중 조작 (multiary operation)으로 확장 시킬수 있다(그림 3-3).

(send obj1 obj2 ... objn msg1)

이와 같은 것은 각각의 객체가 가지는 숫자를 모두 더한다던가 하는 경우에 매우 편리하다. 다중 조작의 수행시 문제점은 개별 단위 방법은 하나의 객체에만 연결 되었기 때문에 "더하기"와 같은 문제는 메세지 "더하기"를 받은 개별적인 단위방법들이 다음 단위방법의 순서를 알고 있어야 하거나 send의 기능을 확장 시켜야 한다는 것이다.

이러한 문제를 해결하는 방법은 빗질하기(currying), 대표화(delegation), 분배 (distribution), 다기능 함수(generic function)등 여러가지가 있다[Ga89]. 그중에 한 방법은 그림 3-4에서 보이는 바와 같이 여러개의 객체에 보내는 메세지를 대

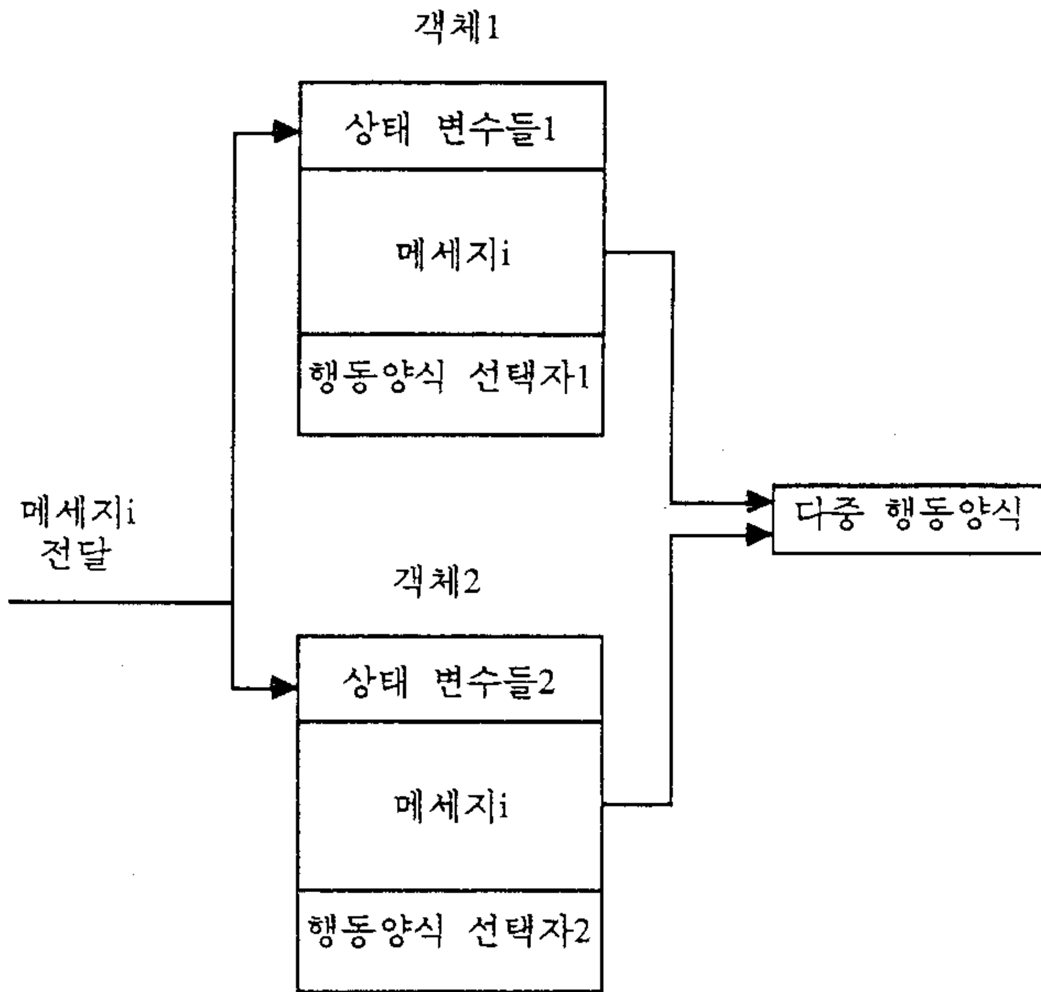


그림 3-3 다중 조작의 개념적 구성

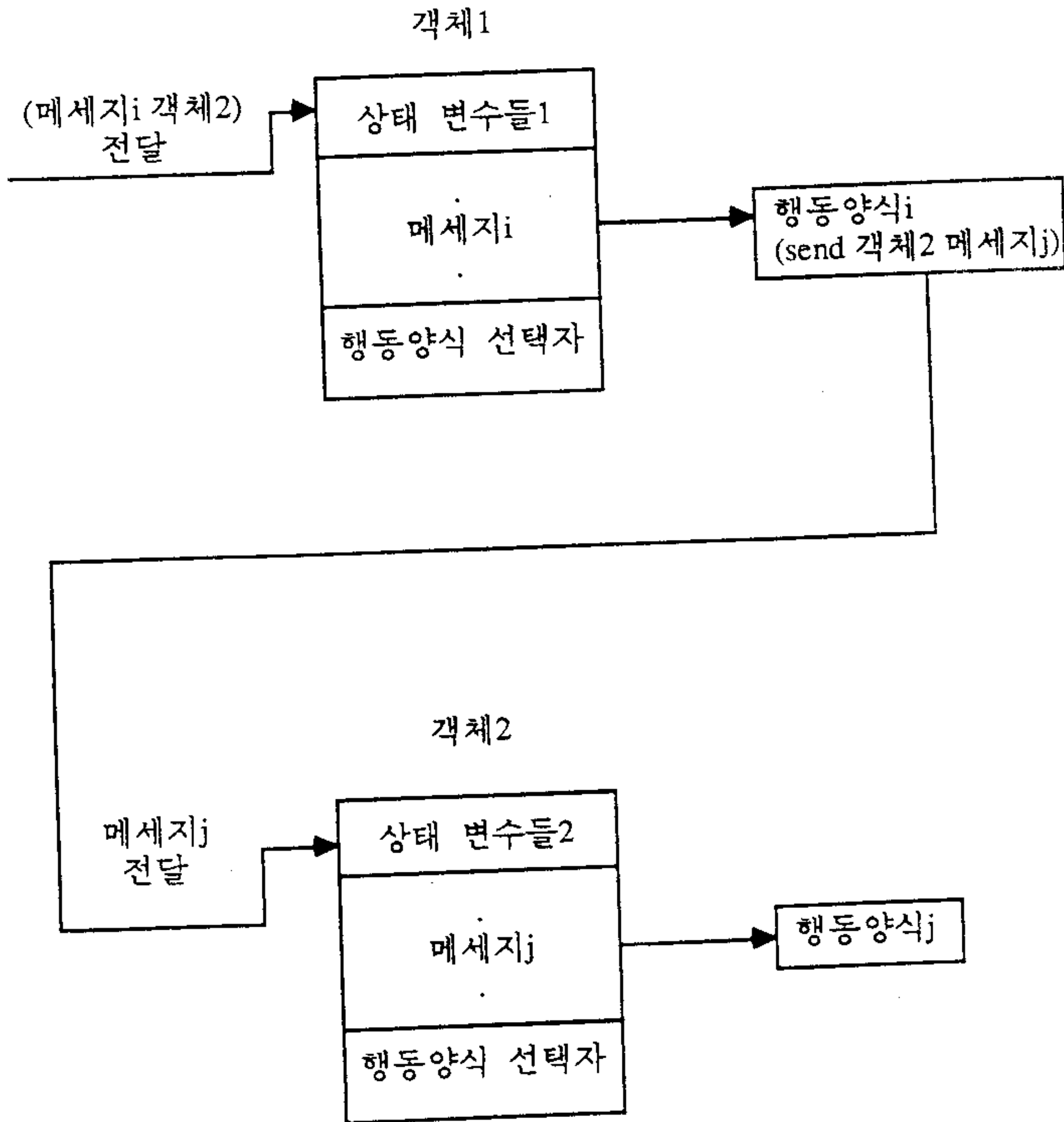
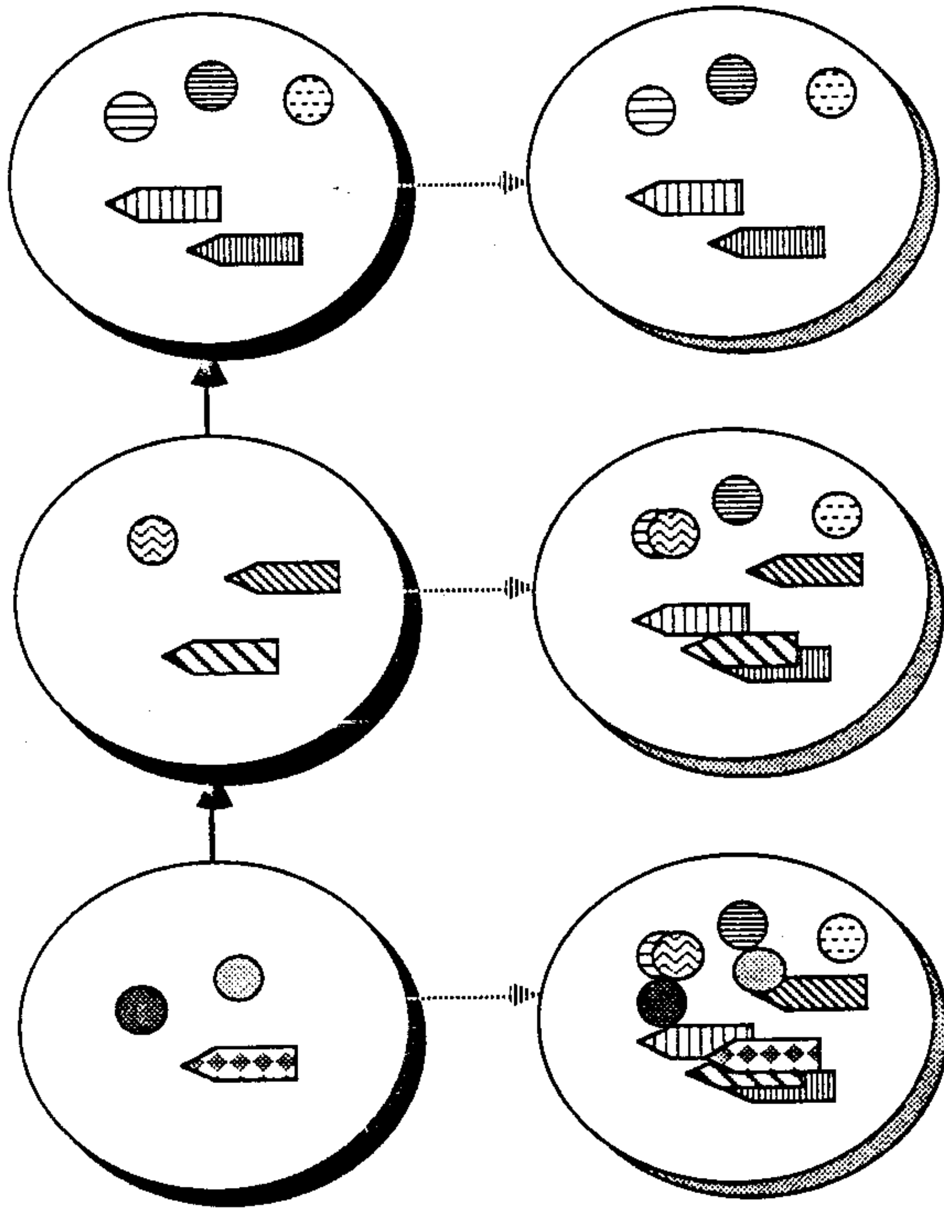


그림 3-4 행동양식을 객체에서 선택할때 다중조작의 한 방법

표하는 한 객체에 보내어 순차적으로 수행 시키는 방법이다. 이 방법은 프로그래머가 이러한 순서를 구성하여 서로 관련 있게 단위 방법들을 만들어 주어야 하는 단점이 있다. 이것은 행동 양식을 객체에서 선택하게 되므로서 발생하는 문제이다.

라. 속성 상속

하위 객체 부류는 상위 객체 부류의 속성을 상속 받는다고 한다. 객체 부류는 상태 변수들의 집합과 행동양식으로 되어 있으므로 하위 객체 부류는 상위 객체 부류의 상태 변수들과 행동양식들을 상속 받게 된다(그림 3-5). 그러므로 하위 객체 부류의 실현 객체는 곧 그 상위 객체부류의 실현 객체이기도 하며, 상위 객체 부류의 행동 양식인 단위방법들이 사용되어 질수 있다. 이와 같은 속성 상속은 객체 부류의 계층 구조의 형태에 따라 이루어지는데, 이러한 객체 부류의 계층 구조는 간단하게는 하나의 직접적 상위 객체 부류만을 허락하는 나무 구조의 형태를 취할수 있고, 더 확장을 시키면 여러개의 직접적 상위 객체 부류를 허락하는 비순환 그래프(acyclic graph) 구조를 취할 수 있다. 이 비순환 그래프 구조의 계층 구조에서 상속 기능은 몇가지 문제점을 가지고 있다. 비순환 그래프 구조에서는 상위 계층 부류간에 같은 속성이 있을 경우 상속시 충돌이 생기는 것이다. 이와같은 충돌의 해결 방법은 가리기 (shadow), 금지(error signaling), 결합(combination), 선택(selection)등의 방법이 있다[Ga89]. 가리기는 비순환 그래프 구조의 객체 부류간에 일련의 순서를 정하고, 그 순서에 의해서 우선 순위를 주고, 우선 순위가 앞서는 객체 부류의 속성을 상속 시키는 방법이다. 금지는 이와 같은 충돌을 허락하지않고, 프로그램에 이와 같은 상황이 발견되면 에러(error)를 출력하는 방법이다. 결합은 속성을 결합시켜 상속 시키는 방



객체 부류

상속 받은 상태

객체 부류의 상하위 관계

그림 3-5 객체 부류의 관계와 상속

법으로, 예를 들어 충돌이 일어나는 속성이 상태 변수의 타입(type)일때 두 타입을 모두 만족하는 새로운 타입을 상속시키는 방법이다. 선택은 충돌이 발생했을 때 프로그래머로 하여금 선택할수 있도록 하는 방법이다.

2 Common Lisp Object System

일반적인 객체 중심 시스템의 요소 중에서 CLOS의 특징을 살펴 보면 다음과 같다.

* 메세지 전달 형태의 조작을 사용하지 않는다.
* 다중 조작을 허용한다.
* 다중 상속을 허용한다.
* 상속 충돌의 해결을 위하여 결합과 가리기 방법을 사용한다.
* 여러가지 방법 조합을 제공하고 있다.
* Common LISP의 구문 형태와 일관성이 있다.
* 초객체 부류(metaclass) 개념을 사용하여 확장성이 있다.

CLOS는 초객체 부류 개념을 도입하고 있어서 CLOS 자체가 객체 중심 프로그래밍의 개념을 사용하여 작성되어 있다. 이것은 CLOS가 실제적으로 객체 중심 프로그래밍 언어를 통하여 작성 되어 있지는 않았지만 CLOS의 기본적인 초객체들을 정의하고 이 초객체에 대한 행동양식들을 정의한 형태로 구성되어 있다. CLOS의 초객체와 객체 부류, 실현 객체 간의 관계가 그림 3-6에 나타나 있다. 그러나 CLOS의 초객체 프로토콜은 아직 표준화 되어 있지 않아서 정확하게 기술되어 있지 않으므로 CLOS에서 다루고 있는 다음의 기본적인 객체를 중심으로 CLOS를 기술한다.

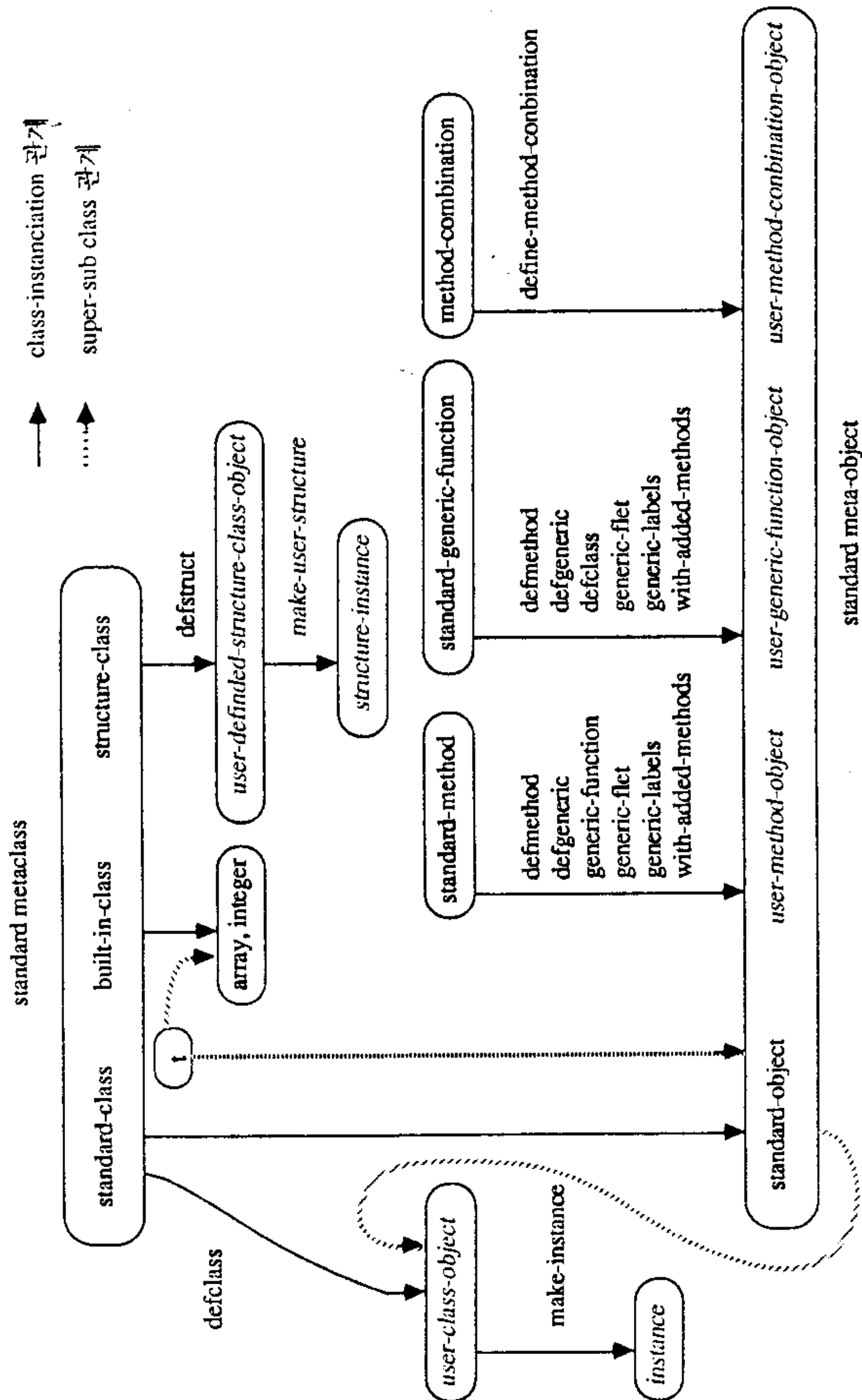


그림 3-6 CLOS의 meta-object와 class, instance간의 관계

* 객체 부류(class)
* 실현 객체(instance)
* 단위 방법(method)
* 다기능 함수(generic function)
* 방법 조합(method combination)

위의 기본적인 5개의 객체들은 그림 3-6에서 user-class-object, instance, user-method-object, user-generic-function-object, user-method-combination-object에 해당한다. 위의 각 객체들은 모두 각각의 객체 부류들을 가진다. 위의 각 객체들의 객체 부류에는 기본적인 실현 객체 생성 함수, 참조 함수, 수정 함수들을 가지고 있다.

가. 객체 부류와 실현 객체

객체 부류는 실현 객체의 구조를 정의하며 실현 객체의 타입을 제공한다. 실현 객체의 구조는 지역변수(local slot)들의 집합으로 구성되는데, 실현 객체의 각 지역변수에 해당하는 칸(slot)의 수와 타입을 객체 부류에서 정의한다. 사용자는 defclass 매크로를 사용하여 객체 부류를 정의하고, 이 객체 부류에 따르는 실현 객체를 다기능 함수인 make-instance를 사용하여 만들 수 있다. 객체 부류 existence를 정의하고 그에 대한 실현 객체를 생성하는 예가 아래에 나타나 있다.

```
>(defclass position () (x y))
#<existence>
>(make-instance 'position)
#<123>
```

이때 객체 부류 existence에는 x, y라는 두개의 칸이 있으며, 실현 객체 #

<123>은 x, y에 해당하는 저장소를 갖는다.

CLOS에서는 하나의 객체 부류가 여러개의 직접 상위 객체 부류를 가질 수 있다. 그림 3-7에서 보이는 것처럼 life라는 객체 부류는 variable-object와 position이라는 두개의 객체 부류를 상위 객체 부류로서 포함한다.

```
>(defclass life (position variable-object) (age-limit life-degree))
#<life>
```

이와 같은 형태는 객체 부류간에 비순환 그래프 형태의 계층 구조를 만들어 주며, 이 계층 구조를 통하여 다중 상속이 이루어 진다. 이 다중 상속 기능도 CLOS의 특징으로서 smalltalk와 같은 경우는 이러한 기능이 없다. 다중 상속시 발생하는 속성간의 충돌 해결은 결합과 가리기 두가지 방법을 사용한다. CLOS에서 상속은 객체 부류의 선택 조항(class option)과 칸의 선택 조항(slot option), 그리고 기본방법의 3가지로 나누어 볼수 있다. 각각의 상속은 객체 부류간의 계층 구조에 따라 이루어 지는데, 대부분의 상속은 객체 부류간의 우선 순위에 따라 우선되는 속성 만을 상속 시키는 가리기 방법을 사용하나, 칸의 타입의 경우는 결합의 방법을 제공한다. 타입의 충돌이 발생했을 경우는 충돌한 두개의 타입을 모두 허용하는 새로운 타입을 상속 시키게 된다. 위의 예에서 life는 position과 variable-object의 칸들을 상속 받아서 life가 가지는 칸들은 age, x, y, age-limit, life-degree가 된다.

나. 다기능 함수, 단위 방법, 방법 조합

CLOS에서는 객체에 대한 행동 양식의 수행을 메세지 전달 방식이 아닌 다기능 함수 형태로서 행하고 있다. 그림 3-7에서 객체 부류 variable-object와

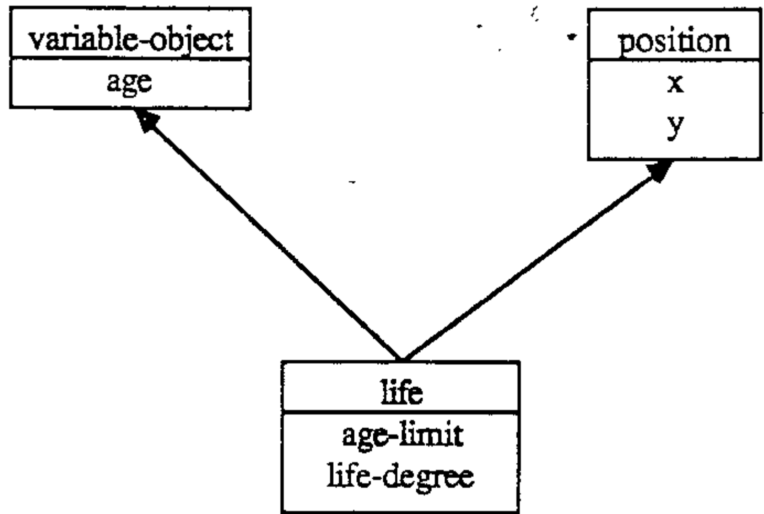


그림 3-7 두개의 상위 객체 부류를 갖는 객체 부류 계층 구조

print-state	x
variable-object position	(print (slot-value x 'age)) (print (slot-value x 'x)) (print (slot-value x 'y))

그림 3-8 두개의 단위 방법을 가지는 다기능 함수

position에 대하여 각각 print-state라는 단위 방법을 아래와 같이 정의하면 print-state라는 다기능 함수가 만들어지며 이 다기능 함수에 각각의 객체 부류에 대한 두개의 단위 방법들이 만들어 진다(그림 3-8).

```
>(defmethod print-state ((x variable-object))
  (print (slot-value x 'age))
print-state
>(defmethod print-state ((x position))
  (print (slot-value x 'x))
  (print (slot-value x 'y)))
```

이와같이 생성된 다기능 함수 객체는 사용자에의해서 일반적인 Common LISP 함수 호출 구문과 동일하게 호출 된다.

```
>(setq ex (make-instance 'variable-object))
#<224>
>(setf (slot-value ex 'age) 100)
100
>(print-state ex)
100
```

다기능 함수의 수행은 사용자가 지정한 전달값(argument)인 실현 객체에 대해서 그 전달값의 객체 부류를 얻어 자신의 단위 방법들 중에 그 객체 부류에 해당하는 것을 수행시켜 주는 것이다. 이때 적절한 단위 방법들을 찾고 그 방법들간의 수행 순서를 만드는 것등이 방법 조합 객체에서 할일이다. 이와 같은 형태의 다기능 함수는 일단 구문 형태가 Common LISP과 일관성이 있어서 기존

의 프로그래머들에게 익숙하며 그 사용상의 의미가 유사한 방법들을 모아 다기능 함수가 관리 함으로서 다중 조작의 구현이 쉽다. 그림 3-3에 나타난 개념적인 다중 조작은 객체가 방법을 선택해 주었을 때는 여러 객체에 동시에 관련된 방법을 다루기 어렵지만(그림 3-4), 그림 3-9 에서 보인 것처럼 메세지 전달 방법에서 같은 메세지가 호출한 단위 방법들을 모아서 방법을 선택해 주는 것은 쉽게 구현되며 이해 하기가 쉽다. 이와같이 CLOS의 메세지 전달 방법은 같은 메세지에의해 호출되는 기본 방법들의 집합을 다기능 함수에서 관리한다.

방법 역시 객체 부류의 계층구조에 따라서 상속되는 것으로, 이 방법의 상속에의해서 한 다기능 함수가 호출되면 그에 따라 여러개의 방법들이 수행 가능한것으로 선택될 수 있다. 사용자가 다기능 함수가 호출되어을 때 수행될 단위 방법의 순서를 정하는 것은 두가지 방법이 있다. 그것은 선언적 방법(declarative technique)과 필수적 방법(imperative technique)으로, 선언적 방법은 미리 방법이 조합되는 형태가 정의되어 있는 방법 조합을 사용하는 것이고, 필수적 방법은 call-next-method 함수를 사용하여 사용자가 적절히 수행 방법을 조합하는 것이다.

CLOS에서는 미리 정의된 많은 방법 조합들을 제공하고 있다. 이 방법 조합 객체가 하는 일은 다기능 함수의 단위 방법들이 가질 수 있는 자격자(qualifier)들을 정의하고, 다기능 함수의 수행시 순서화된 단위 방법들을 전달 받아, 이것을 자격자들에 따라서 묶고, 각 자격자에 따른 다른형태의 수행 순서를 조합하여 다기능 함수내에서 수행될 수 있는 형태로 출력하는 것이다. 이것은 객체내의 행동 양식 선택자의 역활과 유사하다.

만약 사용자가 다기능 함수의 방법 조합을 특별히 지정하지않는 한, 대표적인 방법 조합인 standard 방법 조합이 자동으로 지정 된다. 이 standard 방법 조합은 4가지의 자격자를 정의하고 있다. 이것은 around, before, after, primary로서, 각 자격자에 해당하는 방법들은 서로 다른 수행 방식을 취하고 있다. 그림

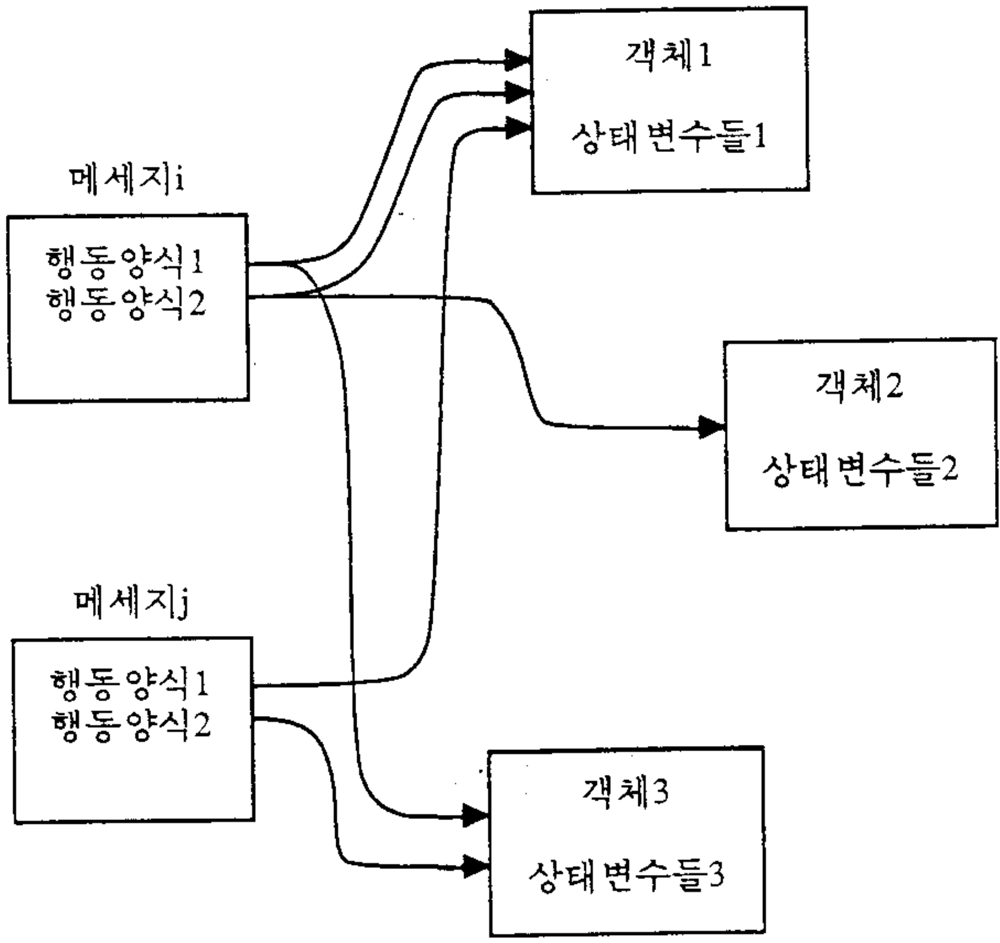


그림 3-9 다기능 함수 형태의 다중 조작

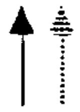


그림 3-10 standard 방법 조합의 단위 방법 순서화 형태

3-10에서 보이는 것처럼 선택된 방법에 around 단위 방법이 있으면 around 단위 방법 중 가장 우선된 단위 방법이 수행 된다. 단위 방법들의 우선 순위는 역시 객체 부류의 우선 순위 열에 따른다. around 단위 방법 내에서 call-next-method가 사용되어 있으면 그 다음 순위의 around 방법이 사용되고 만약 call-next-method 함수의 호출이 있었는데 더이상 around 단위 방법이 없거나, 선택된 단위 방법들 중에 around 단위 방법이 없었을 경우는 before 수행 방법이 수행된다. before 단위 방법은 선택된 모든 before 단위 방법이 우선 순위에 따라 순차적으로 다 수행 된다. before 단위 방법의 수행이 모두 끝나면 primary 단위 방법이 수행된다. primary 단위 방법은 around 단위 방법처럼 가장 우선하는 한 단위 방법이 수행 되고 만약 call-next-method 함수를 사용하고 있으면 그 다음 primary 단위 방법이 수행된다. primary 단위 방법의 수행이 끝나면 after 수행 방법들이 before 수행 방법들 처럼 순차적으로 전부 수행된다. 그러나 after 수행 방법들은 가장 우선 순위가 낮은 방법 부터 시작하여 가장 우선 순위가 높은 방법이 마지막이 되도록 수행된다.

제 3 절 Common Lisp Object System의 구현

본 연구에서는 기존의 Common LISP 시스템에서 시스템의 수정 없이 CLOS를 사용할 수 있도록 구현 하였으며 가능한 모든 기능들을 제공하여 효율적으로 수행하도록 구현하였다. CLOS의 스페시피케이션은 아직 초객체 프로토콜 부분이 미완이며 또한 표준화된 나머지 스페시피케이션도 부분적으로 잘못 기술되거나 미흡한 부분이 있어 완벽한 구현에 어려움이 많으나 여러가지 문헌과 기존의 시스템들을 통해 타당하다고 생각되는 점들로 보충하여 거의 대부분의 기능들을 의

미상에 어긋남이 없이 구현 하였다.

CLOS 구현시 효율에 가장 많은 영향을 미치는 부분들은 다음과 같다.

* 객체 부류 우선 순위 열(Class Precedence List)의 계산
* 상속되는 칸의 집합 계산
* 객체 부류의 우선 순위 찾기와 실현 객체에서 칸의 위치 찾기
* 적용 가능한 단위 방법의 집합과 그 순서화(sorting)

위와 같은 부분들은 각각 적절한 방법에 의해 효율적으로 구현되었으며, 효율의 개선을 위하여 모든 부분에서 다음과 같은 방법을 기본적으로 사용하였다.

* 모든 계산은 실제로 사용될 때 한다.
* 계산된 결과는 보관되어 환경(environment)이 바뀌기 전에는 다시 계산하지 않는다.
* 기호(symbol)을 사용한 찾기(search)는 모두 해쉬 표(hash-table)로 구성하였다.
* 한 객체에서 다른 객체의 정보를 유지할 때에는 내용을 복사하지 않고 지시자만을 유지한다.

효율을 개선하기 위하여 본 연구에서는 다음과 같은 가정을 수용하고 있다. 대부분의 객체 중심 프로그램은 자료 추상화의 목적으로 객체 부류 계층 구조를 유지하며 상속 기능을 사용할 뿐, 수행시 사용되는 실현 객체의 생성은 대부분 하위 객체 부류들에 대하여 행해진다는 것이다. 이와같은 가정은 실험을 통하여 확인되지는 않았으나 많은 프로그래머들이 경험을 통하여 수긍하리라고 생각한다. 이 가정은 객체 부류 우선 순위 열의 계산과 유지, 그리고 단위 방법의 순서화와 관리에 사용되고 있다.

1. 이식성의 고려

Common LISP의 새로운 기능으로서 추가된 CLOS는, 이미 널리 사용되고 있고 여러가지 제품이 나와 있는 기존의 Common LISP 시스템에서 빠른 시일내에 사용되도록 하여야 할 것이다. 그러나 CLOS는 그 자체에 이전의 Common LISP 스페시피케이션을 수정 해야만하는 부분들을 담고 있으므로 CLOS의 구현은 기존 시스템의 인터프리터의 수정을 필요로 한다. 그러나 이미 나와 있는 많은 제품의 인터프리터의 수정은 상당한 시간을 요하는 작업이며, 각 구현마다 서로 다른 부분이다. 또한 아직까지 CLOS의 스페시피케이션은 부분적으로 미완이어서 좀더 보완해야 할 부분이 있으므로 인터프리터의 수정을 완벽하게 제시하지 못하고 있다.

그러므로 본 연구에서는 CLOS의 스페시피케이션에서 인터프리터의 수정을 필요로 하는 부분을 분석하여, 기존의 Common LISP을 사용하여 우회적으로 처리하여 인터프리터의 수정 없이 CLOS의 사용이 가능하도록 하였다.

* 객체 부류의 타입
* 다기능 함수 객체의 타입
* Common LISP 타입 계층 구조 내에서의 상속 기능
* 초 객체 부류

CLOS에서는 객체 부류가 정의 되었을 때 그 객체 부류에 속하는 실현 객체의 타입을 제공한다. 이것은 Common LISP의 타입 계층 구조에 부합하는 것으로, 여러가지 타입에 관련된 조작을 가할 수 있다. 그러나 CLOS에서는 객체 부류의 이름 뿐만 아니라 객체 부류 객체 자체가 그 실현 객체의 타입 기술자 (type specifier)가 되도록 요구하고 있다. 그러나 이것은 기존의 Common LISP이

타입 기술자로서 기호 만을 취하고 있으므로 본 구현의 객체 부류와 같은 structure는 절대 타입 기술자가 될 수 없다. 이것은 인터프리터의 수정을 요하고 있다. 그러므로 본 구현에서는 이 부분을 제외 하였다. 그러나 객체 부류의 이름은 타입 기술자로서 제공 되고 있으므로 기존의 Common LISP 개념에서 벗어나지 않으며, CLOS의 사용에서도 결정적인 문제는 발생 하지 않는다.

또한 Common LISP에서 함수는 함수 타입이 정해져 있어서 새로운 함수 타입을 정의 할 수 없으므로, 다기능 함수 객체가 함수로서 그냥 수행되도록 할 수가 없다. 그러므로 본 구현에서는 다기능 함수 객체를 기호의 성질(property)로서 결속하고 기호에 다기능 함수 객체의 함수 부분만을 결속 시켜 다기능 함수의 호출이 가능 하도록 하였다. 그러므로 만약 다기능 함수 객체를 기호의 성질에서 제외 시키면 다기능 함수는 제대로 수행 되지 않는다. 그러나 이것은 정상적인 사용에서는 문제를 일으키지 않는다.

CLOS의 객체 부류 계층 구조는 대응하는 타입의 계층 구조를 만들며 또한 Common LISP의 타입도 객체 부류의 계층 구조와 같이 상속이 이루어지도록 제안하고 있다. 그러나 Common LISP의 타입에서 상속은 각 타입 간에 우선 순위 열이 마련되어야 한다. 이 타입의 우선 순위 열을 만들어 주는 것은 마찬가지로 기존의 Common LISP 시스템을 수정해야 하는 것이므로 제외 하였다.

CLOS의 초 객체 부류와 관련되 초 객체 프로토콜은 아직 Common LISP의 기능으로 확정되지 않았고 스페시피케이션도 확정되지 않아 제외 시켰다. 이상의 부분을 제외 하고는 모두 구현되어 있으므로 본 구현은 객체 중심 프로그래밍 시스템으로서 손색 없이 구현 되었다고 할 수 있다.

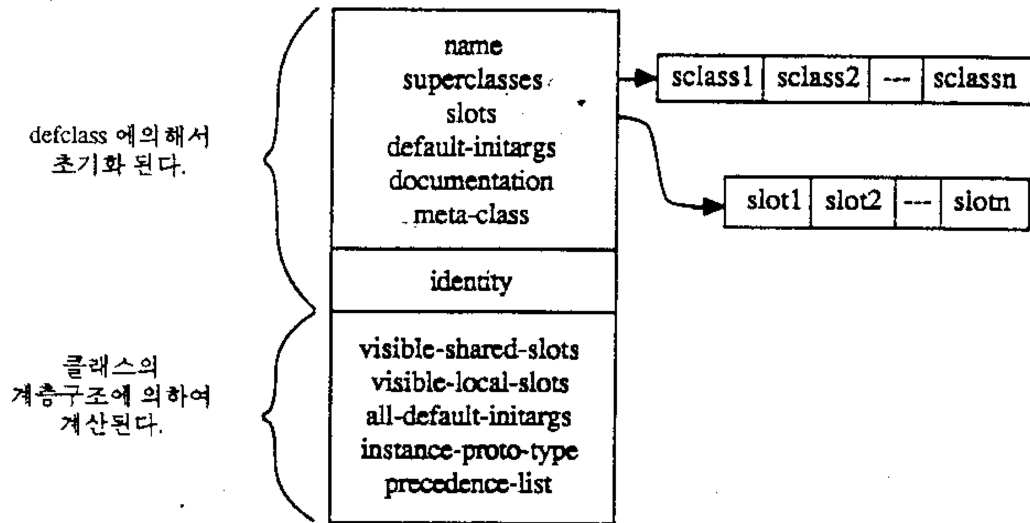
2. CLOS의 내부 설계

CLOS의 기본적인 객체인 객체 부류, 실현 객체, 다기능 함수, 단위 방법, 방법 조합들은 사용자가 정의한 여러가지 정보를 보관하는 각각의 구조들을 가진다.

사용자는 각각의 객체들을 정의하고 생성하며 각각의 객체에 조작을 가한다. 사용자가 일반적으로 CLOS를 사용하여 객체 중심 프로그래밍을 하는 순서는 다음과 같다.

1. 객체 부류를 정의하여 객체 부류 계층 구조를 만든다.
2. 각 객체 부류에 대한 방법들을 정의한다.
3. 실현 객체를 생성한다.
4. 실현 객체를 전달값으로 다기능 함수를 호출한다.
5. 필요에 따라 객체 부류, 다기능 함수, 방법들을 재정의한다.

이러한 순서에의해서 정의, 생성되는 객체들의 구조는 그림 3-11, 3-12, 3-13, 3-14에 나타나있다. 그림 3-11의 객체 부류의 구조는 사용자가 defclass 매크로를 사용하여 정의한 내용을 가지는 부분과 이후에 상위 객체 부류로 부터 상속 받는 정보를 보관하는 부분으로 나뉘어 진다. 객체 부류의 칸은 객체 부류에 속하여 모든 실현 객체에서 공통으로 참조하는 공통 칸(shared slot)과 각 실현 객체마다 다른 값을 갖게 되는 지역 칸(local slot)이 있다. 각 종류의 칸은 서로 다른 타입으로서 객체 부류에서 관리된다. identity는 기존의 객체부류가 재정의 되었을 때를 위한 것으로 그 객체가 재정의 될 때마다 새로운 값을 가지게 된다. 한번 생성된 객체 부류는 그 객체 부류의 정보에 관련되어 생성되는 실현 객체 등에서 참조되고 있는데, 이것은 객체 부류가 재정의 되더라도 그 객체 부류에 대한 지시자(pointer)는 변하면 안되므로 자신이 참조 했던 객체 부류의 identity를 보관하여 관리한다.



객체 부류

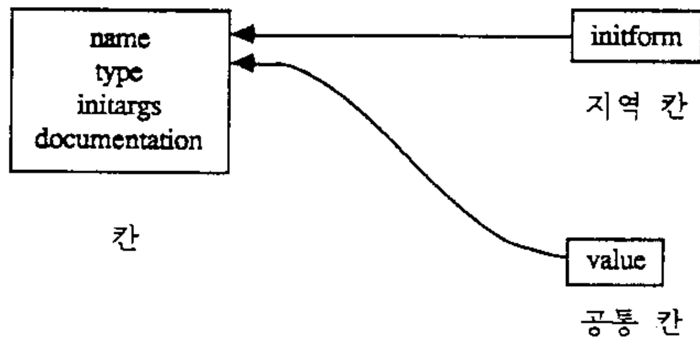


그림 3-11 객체 부류와 칸의 구조

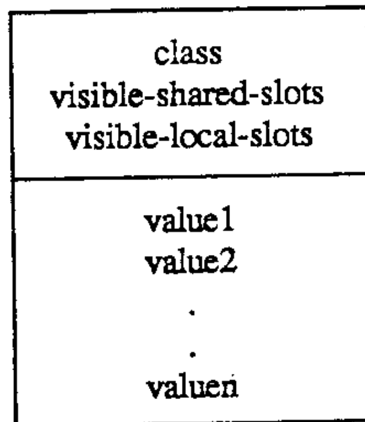


그림 3-12 실현 객체의 구조

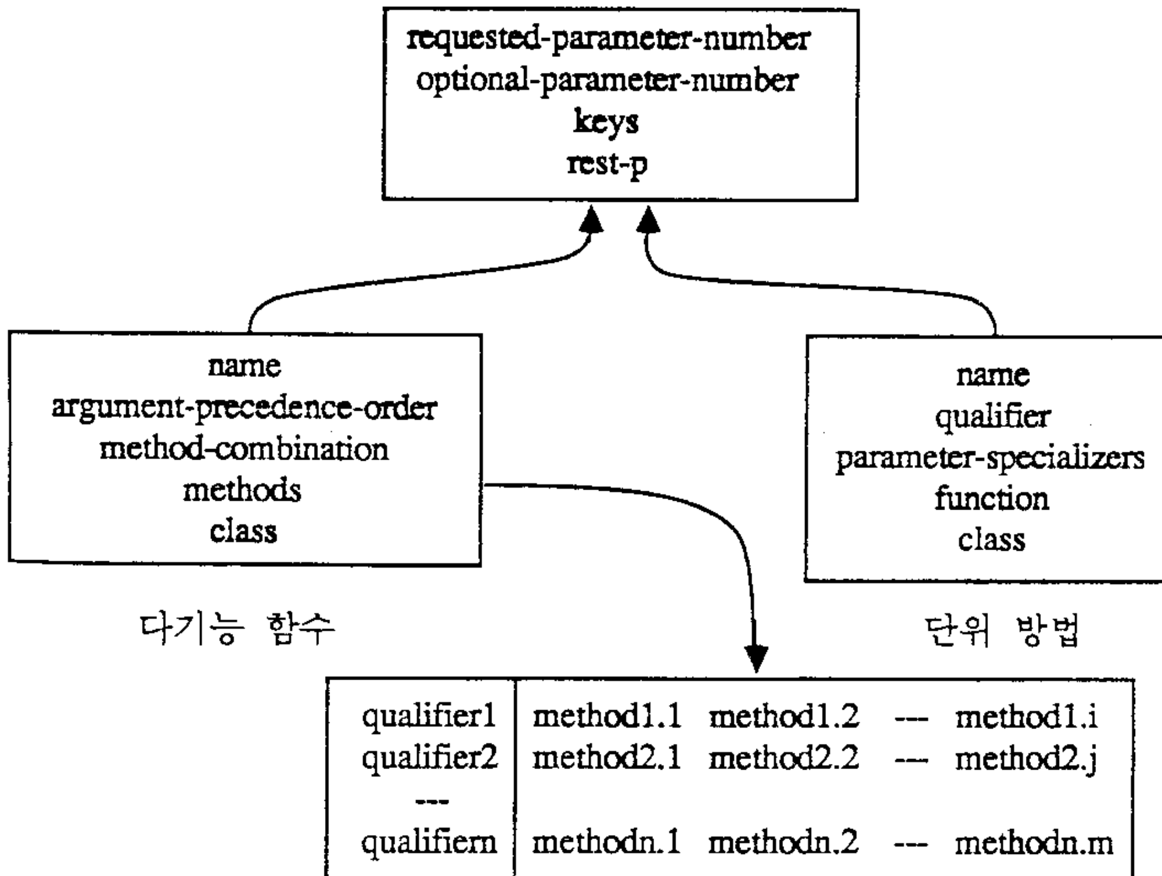


그림 3-13 다기능 함수와 단위 방법의 구조

그림 3-12에 보이는 실현 객체의 구조는 전적으로 해당 객체 부류에 의존하며, 객체 부류의 visible-shared-slots에 들어 있는 칸들에 해당하는 크기의 저장소(storage)를 갖는다. 또한 기본적으로 해당 객체 부류를 가르키며 객체 부류가 재정의 되었을 때 구조의 변형을 위하여 해당 객체 부류의 visible-shared-slots와 visible-local-slots에 대한 정보를 가지고 있다.

그림 3-13의 다기능 함수와 방법의 구조를 보면 다기능 함수는 자신의 방법들을 보관하며, 방법 조합을 가지고 있다. 방법은 자신이 관련된 객체 부류를 매개 변수 특화자(parameter specializer)에 가지고 있다. 이것들은 모두 같은 객체에 대한 지시자(pointer)를 유지하는 것으로 객체의 복사본을 유지하는 것은 아니다. 각 방법들은 자신의 역활이 지정되어 있는데, 이 역활자 (qualifier)에 따라서 다기능 함수에 분류 되어 있다.

그림 3-14의 방법 조합 객체는 그 방법 조합에서 허용되는 역활자와 그 방법 조합이 수행되는 함수를 가지고 있다. 방법 조합이 수행하는 함수는 다기능 함수 내의 방법들로 부터 일정한 형태의 순서를 만들어 수행 가능한 형태로 출력해 주는 것이다.

이상과 같은 구조는 다음에 설명할 상속과 다기능 함수의 방법 조합, 객체 부류 재정의의 관리등에서 사용될 내용들을 효율적으로 관리 할수 있도록 구성되어 있다. 각 부분의 자세한 설명은 각 해당 항목에서 서술한다.

3. 객체 부류의 계층 구조와 우선 순위 열의 계산

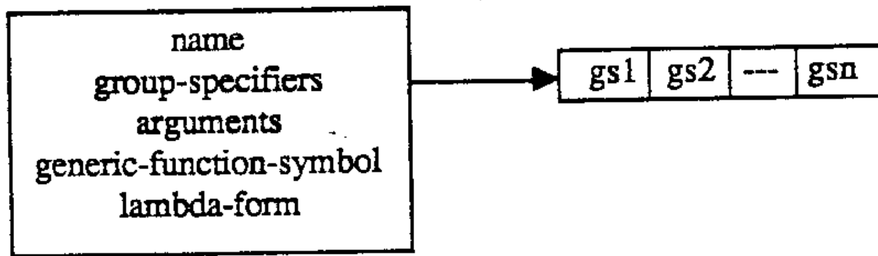
객체 중심 프로그램에서 상속은 객체 부류의 계층 구조에 따른다. 계층 구조가 단순한 나무 구조일 경우는 하위 객체 부류로 부터 최상위 객체 부류까지의 우선 순위 열이 단순한 일렬의 통로(path)를 순회 함으로써 간단히 계산 되

지만, CLOS에서 허용하는 비순환 그래프 형태를 취하게 되면 우선 순위 열의 계산이 간단하지 않다. 비순환 그래프 형태는 위상적 순서화(topological sorting)의 계산을 필요로 한다. CLOS에서 상속은 크게, 방법의 상속, 칸의 선택 조항 상속, 객체 부류의 선택 조항 상속으로 볼수 있다. 이 각각의 상속은 모두 객체 부류 우선 순위 열(class precedence list)에 의존하고 있으므로 객체 부류 우선 순위 열의 계산은 매우 중요하다.

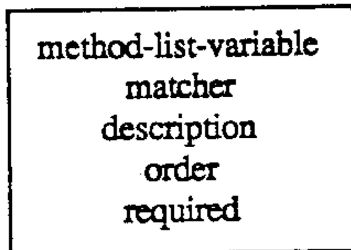
아래의 객체 부류 정의에 대한 계층 구조가 그림 3-15에 나타나 있다.

```
(defclass a ())
(defclass b (a))
(defclass c (a))
(defclass d (b))
(defclass e (b c))
(defclass f (d))
(defclass g (e))
(defclass h (e c))
(defclass i (f g h))
```

그림 3-15에 나타난 계층 구조 그래프가 일반적인 비순환 그래프와 틀린점은 직접적 상위 그래프로 가는 지향선(arc)간에 순서가 존재한다는 것이다. 즉 지향선 FI가 지향선 GI의 왼쪽에 나타난 것은 FI가 GI의 오른쪽에 나타난 것과는 다른 그래프이다. 그림 3-15에 나타난 것과 같은 비순환적 객체 부류의 계층 구조에서 각 객체 부류의 우선 순위 열은 다음 3가지 규칙에 의해서 유일하게 결정 된다.



방법 조합



단체 특화자

그림 3-14 방법 조합 객체와 단체 특화자의 구조

1. 한 객체 부류는 그것의 모든 상위 객체 부류 보다 모든 우선 순위 열에서 우선한다.
2. 객체 부류 C의 직접적 상위 객체 부류들 중 왼쪽에 나타난 상위 객체 부류 C1은 오른쪽에 나타난 상위 객체 부류 C2보다 객체 부류 C의 우선 순위 열에서 우선한다.
3. 만약 객체 부류 C1이 하나의 직접적 상위 객체 부류 C2를 가지며, C2 역시 하나의 직접적 하위 객체 부류 C1을 가지면 모든 우선 순위열에서 C1은 C2의 바로 앞에 온다.

이때 위의 규칙 상에서 주의 할 점은 어느 한 객체 부류가 다른 객체 부류에 우선 하는가는 전체 계층구조 상에서 항상 절대적이지는 않다는 것이다. 어떤 객체 부류 C1, C2가 있을 때, C1이 C2의 상위 객체 부류가 아니며, C2도 C1의 상위 객체 부류가 아니라면 C1과 C2의 상위 객체 부류인 C3, C4는 C1의 객체 부류 우선 순위 열과 C2의 객체 부류 우선 순위 열 상에서 순서가 서로 바뀔 수 있다. 그러므로 우선 순위 열은 전체 객체 부류에 대하여 한번 계산하여 공통적으로 사용할 수 없다.

위의 우선 순위 규칙에 따라 그림 3-15의 객체 부류 I의 우선 순위열을 계산하면 다음과 같다.

I, F, D, G, H, E, B, C, A

위와 같은 우선 순위 열을 계산하기위한 알고리즘은 그림 3-16에 나와 있다. 그림 3-16의 알고리즘은 위상적 순서화 알고리즘[Kn73]을 단계 2에서 동점 처리(tie breaking)를 위한 규칙 3을 적용 시켜서 확장한 것이다. 여기서 관계 집합 R이 위상적 순서화 알고리즘에 직접 사용되었는데 그림 3-15의 계층구조에 대하

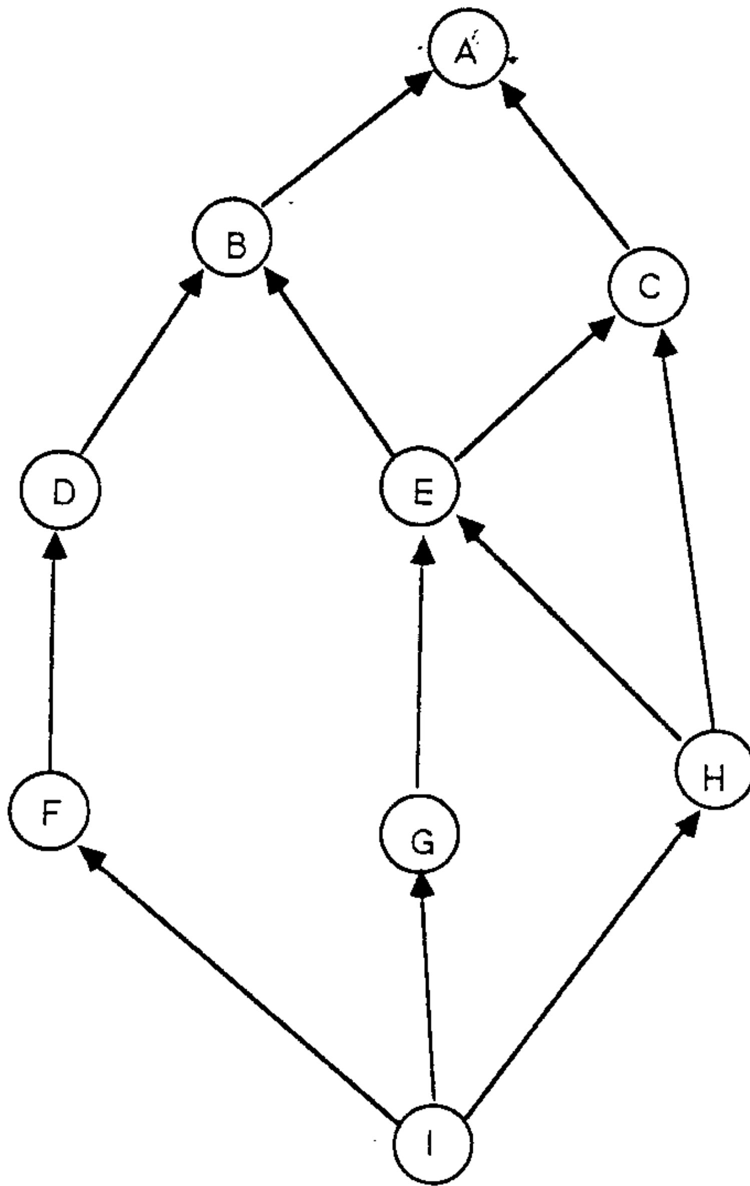


그림 3-15 비순환적 객체 부류 계층 구조의 한 형태

$Rc = \{ (C, C1)\ (C1, C2) \ldots (Cn\text{-}1, Cn) \}$

C1, C2 ... Cn은 C의 직접적 상위 객체 부류 .

C의 직접적 상위 객체 부류 열에서 $i<j$ 이면 Ci는 Cj에 우선 한다.

$Sc = \{ Ci \mid$ Ci는 객체 부류 C거나 C의 상위 객체 부류 $\}$

$$R = \bigcup_{c \in Sc} Rc$$

단계 0 : $CPL \leftarrow ()$, $Sc \leftarrow \{\}$, $Ct \leftarrow \{\}$

단계 1 : $Ct \leftarrow Ct \cup \{ x \mid (x,y) \in R,\ (z, x) \notin R,\ x, y, z \in Sc \}$

단계 2 : $C \leftarrow x \mid x \in Ct$, Ct의 원소 중 직접적 하위 객체 부류가
CPL에서 가장 오른 쪽에 있는 x

단계 3 : 만약 그러한 x가 없으면 잘못 된 계층 구조
아니면 CPL의 맨 뒤에 x를 첨가 한다.

단계 4 : Sc에서 C를 제거한다.
R에서 $(C, X) \mid X \in Sc$ 관계를 제거 한다.

단계 5 : 만약 Sc가 $\varnothing$ 이면 끝
아니면 단계 1

그림 3-16 객체 부류 우선 순위 열을 계산하는 알고리즘

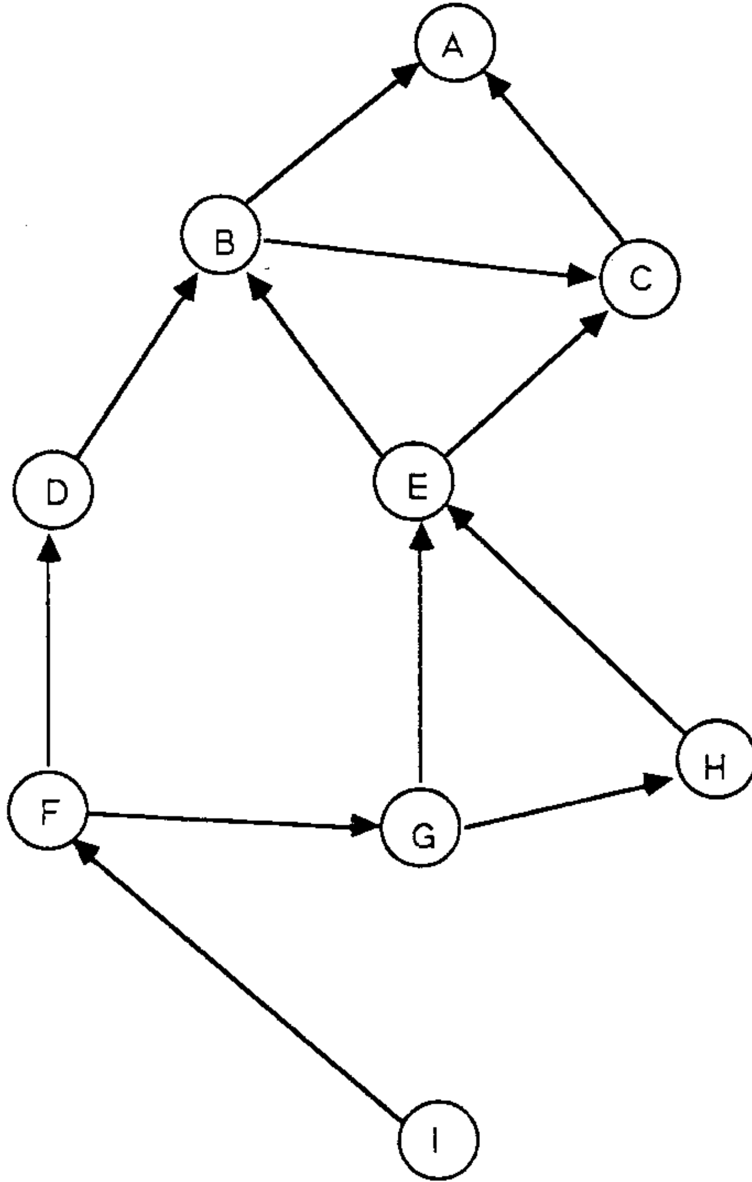
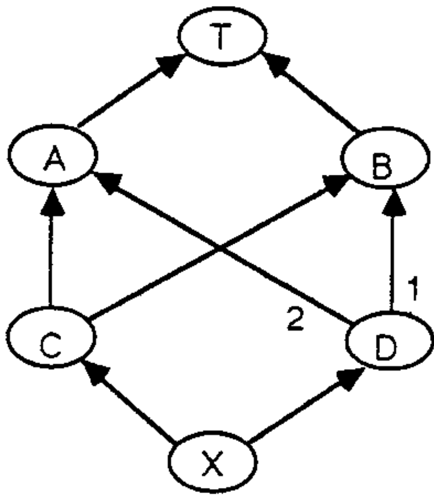


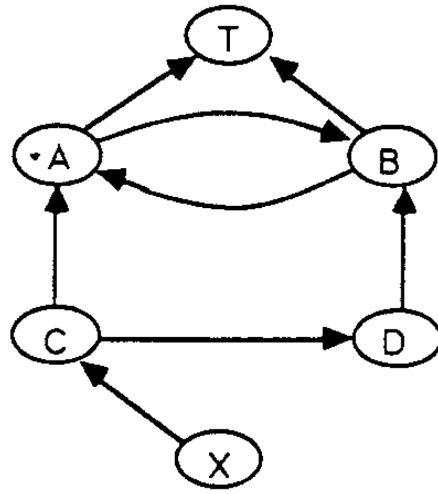
그림 3-17 객체 부류 I를 기준으로 변환된 객체 부류 계층 구조

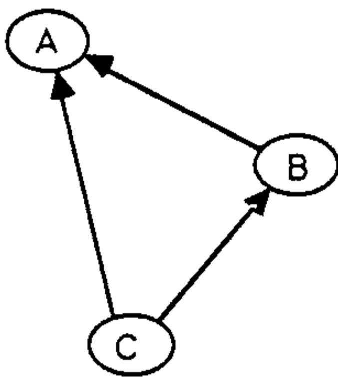
여 관계 집합 R을 구하고, 이 관계 집합을 지향 그래프(directed graph)로 나타내면 그림 3-17과 같다. 이 변형된 그래프에서는 지향선 간에 순서가 없는 비순환 그래프가 된다. 이 변형된 비 순환 그래프에 대하여 그림 3-16의 알고이즘을 적용 시키면 객체 부류 I에 대한 객체 부류 우선 순위 열이 계산 된다. 계층 구조 그래프가 비 순환 그래프가 아닐 경우는 잘못된 계층 구조 판정이 나는데, 이 계층 구조 그래프가 비 순환일 경우도 잘못된 계층 구조일 수가 있다. 그러나 변형된 계층 구조 그래프가 비순환일 경우는 아무 문제가 없는 계층 구조이다. 잘못된 계층 구조와 그 변형된 계층 구조 그래프가 그림 3-18에 나타나 있다. 일반적인 위상적 순서화 알고리즘의 시간 복잡도(time complexity)는 O(n+e)가 걸린다(n: node수, e: edge 수). 그러나 위의 알고리즘은 동점 처리를 위한 단계2에서 대체로 지향선 수에 비례하는 시간이 더 걸리므로 O(n+2e)정도가 수행된다. 그러나 이 구현에서는 객체 우선 순위 열은 그 객체에 대한 첫번째 실현 객체가 만들어지기 전까지는 계산되지 않는다. 이것은 일반적으로 상위 여러 객체들이 대부분 개념적인 추상화를 위하여 사용되며 실제적인 실현 객체를 만들지 않는 경우가 많으므로 효율적이라 하겠다. 이에 대한 상세한 설명은 재정의 관리에서 계속한다.

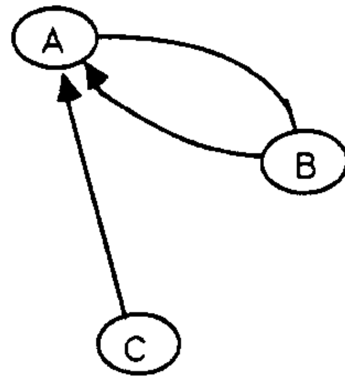
4. 다기능 함수와 단위 방법의 조합

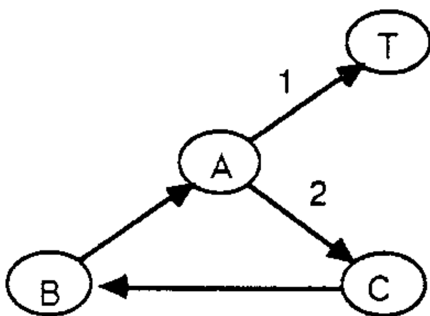
다기능 함수는 의미상 유사한 기능을 하는 단위 방법들을 가지고 있다. 다기능 함수 내에서 관리되고 있는 단위 방법들은 사용자가 이 단위 방법들이 서로 다른 객체에 대하여 비슷한 목적으로 수행됨을 정의한것으로 볼 수 있다. 그러므로 한 다기능 함수 내의 단위 방법들은 의미상 서로 관련을 가지고 있다. 이와같은 의미상의 관련은 이 단위 방법들에 공통되는 규칙이 존재 함을 말한다.











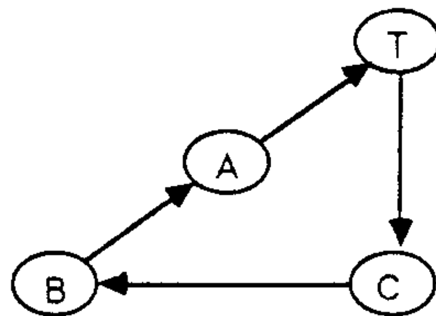


그림 3-18 잘못된 계층 구조와 그 변형된 그래프의 예

CLOS에서는 다기능 함수와 이 다기능 함수의 단위 방법들은 매개 변수의 수와 형식에서 서로 공통되는 규칙을 준수 하여야 한다. 각 단위 방법들은 서로 다른 객체들에 관련되어 있으나 각 단위 방법이 취급하는 객체의 수는 일정하여야 한다. 이러한 조건은 각 단위 방법들의 상속시 우선 순위를 계산하는 것과 관계가 있다.

하나의 단위 방법은 그림 3-13에서 보이는 바와 같이 하나의 자격자와 그 부류의 단위 방법들간에 공통된 몇개의 매개 변수 특화자를 가지고 있다. 단위 방법들은 자격자에 따라 묶이게 되며 매개 변수 특화자들에 따라 선택되고 순서화된다.

사용자가 다기능 함수를 호출 할때 호출 된 다기능 함수는 그림 3-19에서 보이는 단계 대로 수행 된다. 먼저 사용자는 정해진 수의 실현 객체 전달 값들과 다른 필요한 값들을 다기능 함수에 전달하고 호출한다. 호출된 다기능 함수는 전달 받은 실현 객체들의 객체 부류를 얻는다. 이 객체 부류들은 단위 방법들을 선택하고 순서화 하는데 중요한 역활을 한다. 이 객체 부류들을 방법 선택 객체 부류 열이라 하자. 방법 선택 객체 부류 열에서 각 객체 부류는 객체 부류 우선 순위 열을 갖는데 이 우선 순위 열들로 부터 단위 방법의 상속과 순서화가 이루어 진다. 우선 방법 선택 객체 부류 열은 적용 가능한 단위 방법들을 선택하는 부분에서 사용되어, 단위 방법의 전달 변수 특화자가 가지는 객체 부류가, 방법 선택 객체 부류 열의 대응하는 객체 부류의 상위 객체 부류인 단위 방법은 적용 가능한 단위 방법으로 선택 된다. 이렇게 선택된 단위 방법들은 다음 단계인 단위 방법 순서화 부분에서 방법 선택 객체 부류 열의 우선 순위 열들에 의해서 사전식 순서화(lexical sorting)가 된다. 그림 3-20에서 보이는 것처럼 방법 선택 객체 부류 열의 각각의 객체 부류에 대한 우선 순위열은 Ci에 해당하는 단위 방법의 전달 변수 특화자의 객체 부류들의 순서를 정해 준다. 이때

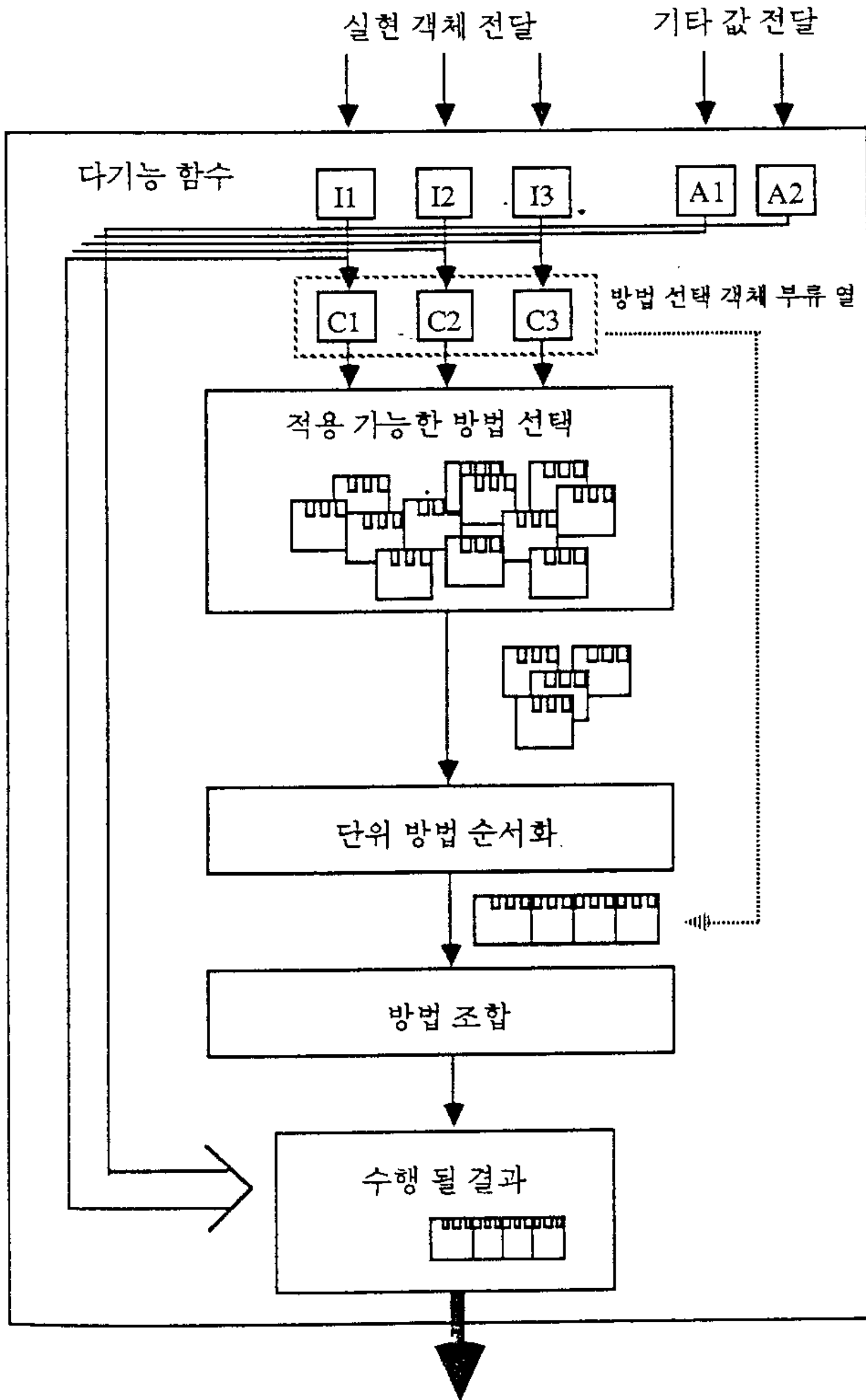
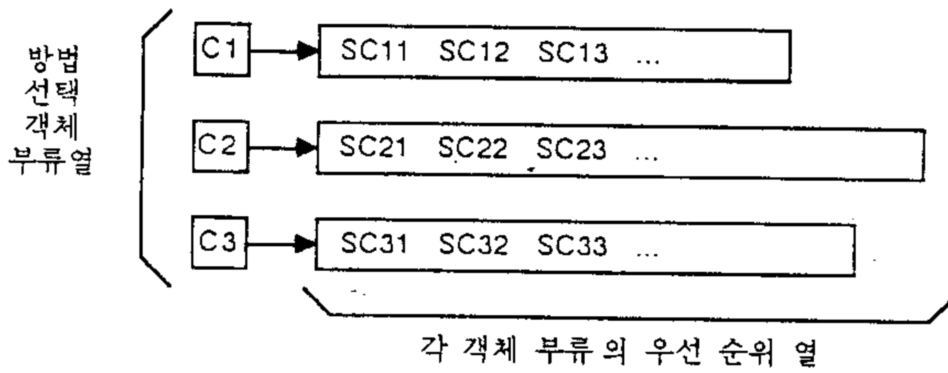
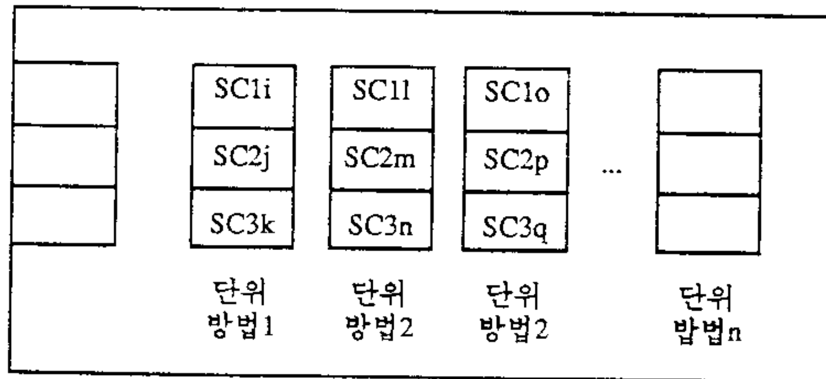


그림 3-19 다기능 함수의 수행





다기능 함수

그림 3-20 방법 선택 객체 부류 열과 단위 방법

C1에 해당하는 것이 제일 우선이며, C2, C3의 순으로 우선 순위를 가진다. 이와 같이 우선 순위열들에 따라 사전식 순서화가 된 단위 방법들은 방법 조합 함수에 전달 된다. 방법 조합 함수는 이로 부터 수행 가능한 형태의 조합을 만들어 주고 이 결과가 수행되어 다기능 함수의 수행 결과와 부대 효과(side effect)를 만든다.

이와같은 형태로 수행되는 다기능 함수는 정작 목적하는 방법 조합 결과의 수행 이전에 적용 가능한 방법 선택과 그 순서화에 많은 시간이 걸린다. 그러나 방법 조합의 결과는 전적으로 방법 선택 객체 부류에 따르는 것이므로 본 구현에서는 다기능 함수의 호출 시 한번 나타난 방법 선택 객체 부류 열에 대하여서는 그에따라 순서화된 단위 방법들을, 방법 선택 객체 부류 열을 색인(index)으로, 해쉬 표에 저장하여 관리 한다. 그러므로서 같은 방법 선택 객체 부류 열이 계산되는 다기능 함수가 호출 되었을 경우 중간 계산 없이 해쉬 표의 참조로 즉시 수행되도록 하는 것이다. 그러나 이것은 이론적으로는 해쉬 표가 방법 선택 객체 부류 열의 각각의 우선 순위 열에 있는 객체 부류의 수를 모두 곱한 크기 만큼 저장해야 하나 앞에서 말한 실현 객체 생성과 사용에 대한 가정에 따라 해쉬 표에 보관 해야 될 단위 방법들이 많지 않다고 할 수 있다.

이렇게 한번 해쉬 표에 저장된 단위 방법들은 이후에 단위 방법의 추가, 수정, 제거시 그상태로 관리된다. 또한 해쉬 표에 저장된 단위 방법들 역시 객체의 복사본이 아닌 지시자를 유지하고 있다.

CLOS에서는 Common LISP의 setf method의 정의 기능을 간편하고 일관성 있게 확장하여 다기능 함수의 setf method가 역시 다기능 함수로서 간단히 정의 될 수 있는 기능을 제공하고 있다.* 다기능 함수의 정의시 다기능 함수의 이름을 "(setf function-name)"의 형태로 정의하여 이것이 "(setf (function-name ...) ...)" 의 형태로 사용 가능하게 해 주는 것이 CLOS의 setf method 정의 기능

이다. 이것은 스페시피케이션에서 설명이 빠진 부분으로서 스페시피케이션의 보완을 필요로 한다. 이 부분의 구현은 내부적으로 function-name에 대응 하는 일정한 형태의 새로운 이름의 다기능 함수를 정의하여 Common LISP의 define-setf-method 매크로를 사용하여 구현 하였다.

5. CLOS 객체 재정의의 관리

CLOS의 객체는 수행 중에 재정의 될수 있으며, 재정의 관리 기능에의하여 일관성있게 유지 된다. 이 재정의 기능은 CLOS의 사용에 유연성을 더해 준다. CLOS의 기본적인 각 객체는 모두 수행중에 재정의 될 수 있으며, 각 객체의 재정의는 관련된 객체에 영향을 준다. 그중에서도 가장 중요한 것은 객체 부류의 재정의로서 가장 많은 영향을 미치게 된다. 그림 3-21은 CLOS의 각 부분의 수행 순서를 보이고 있다. 이 부분간 수행 순서는 생성된 각 객체의 의존성에도 관계가 있어서 상위 부분의 재정의는 하위 부분에 영향을 주어서 하위 부분의 수정을 뒤따르게 한다. CLOS의 기본적인 객체를 정의 하는 순서는 다음과 같다 [Ke89].

* 객체 부류는 서로 간에 정의되는 순서의 제약을 받지 않는다.
* 단위 방법들 간이나 다기능 함수와 단위 방법간에도 정의되는 순서의 제약이 없다.
* 실현 객체가 생성 되기 전 까지 그 실현 객체의 객체 부류와 그 상위 객체 부류가 이미 정의 되어 있어야 한다.
* 단위 방법이 생성되기 전 까지 그 단위 방법에 관련된 객체 부류들은 정의 되어 있어야 한다.

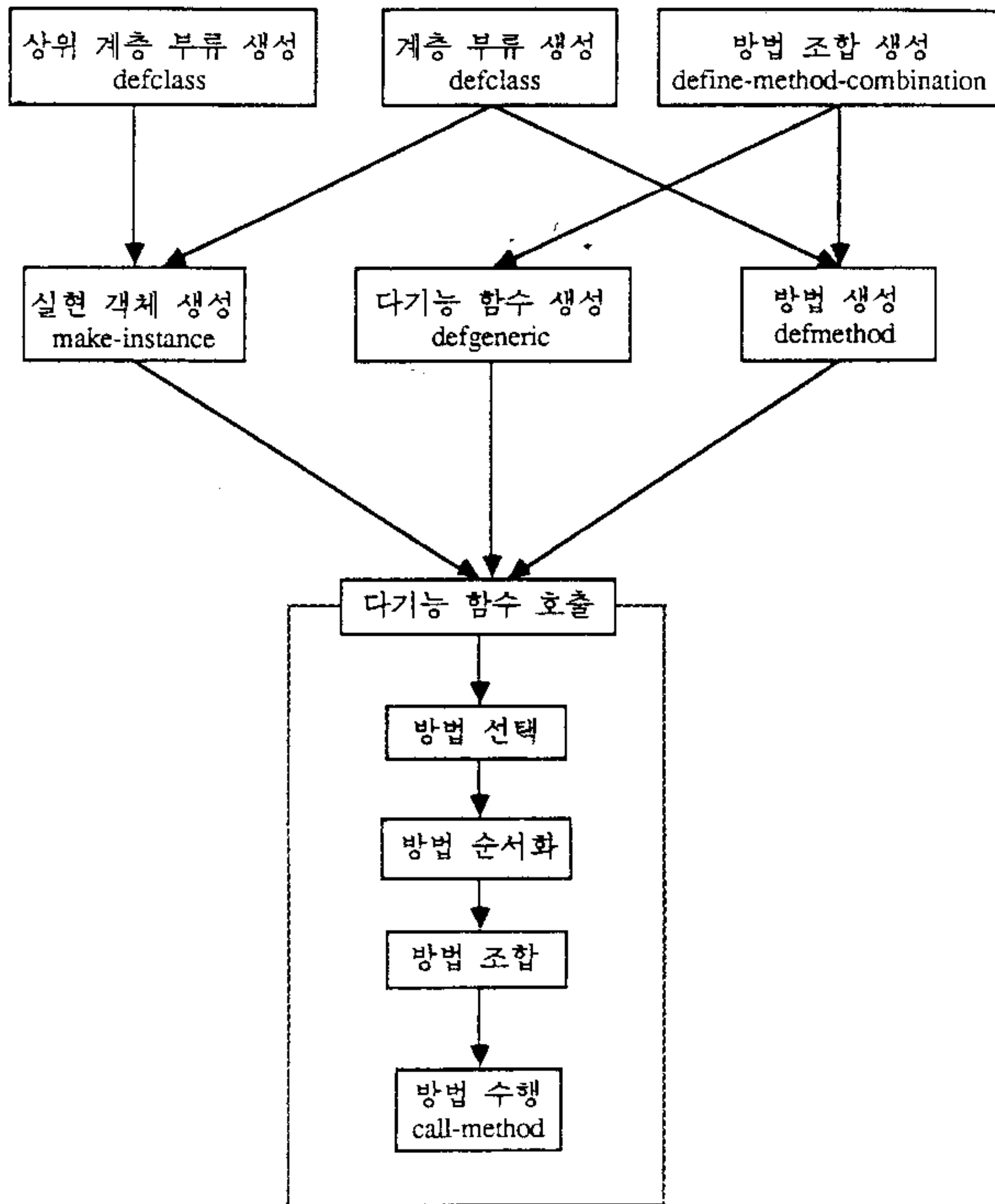


그림 3-21 CLOS의 각 부분 수행 순서

하나의 객체가 재정의 되었을 때 그에 영향을 받는 객체들을 수정하는 것은 두가지 방법이 있을 수 있다. 첫째는 그 객체가 재정의된 직후에 영향을 받는 모든 객체를 수정하는 방법이고, 두번째는 그 객체의 수정 후, 영향을 받는 객체들이 참조 될 때에만 수정하는 방법이다. 각 방법에 대한 단점은 다음과 같다.

* 객체 재정의시 수정의 단점
  - 그 객체에 영향을 받는 객체를 해당 객체에서 유지 관리하고 있어야 한다.
  - 그러므로 더이상 참조되지 않는 객체도 유지하게 되므로 쓰레기 수집(garbage collection)이 되지 않는다.
  - 또한 더이상 참조되지 않는 객체를 반복적으로 수정하게 된다.

* 참조시 수정의 단점
  - 재정의된 객체는 재정의 되었음을 표시해야 하고
  - 그 객체를 참조하는 객체는 자신의 필요에 따라 지난 번 객체에 대한 정보를 유지 해야 하며
  - 수정될 객체는 첫번째 참조 되는 시점에서 수정되는 시간이 걸린다.

위의 두 방법들은 장단점을 가지고 있어서 본 구현에서는 경우에 따라 두 방법을 선택 사용하였다. 객체 재정의시의 수정은 반복되는 수정이 많으나 영향 받는 객체의 관리가 쉽고 나중에 수행하는데 따로 시간이 걸리지 않는다. 그러나 쓰레기 수집이 안되는 것은 경우에 따라서 치명적인 단점이 될 수 있다. 참조시 수정은 반복하여 수정되지 않으므로 전체적인 관점에서 수행이 빠를 수 있다. 각 경우의 사용은 아래에서 개별적인 객체의 재정의 시 관리에서 상세히 설명한다.

가. 객체 부류의 재정의

객체 부류가 재정의 되면 그 객체 부류를 참조하고 있던 모든 객체들의 수정이 필요하다. 한 객체 부류가 재정의 되었을 때 수정을 필요로 하는 객체들은 다음과 같다.

* 그 객체 부류의 실현 객체와 단위 방법
* 그 객체 부류의 하위 객체 부류와 하위 객체 부류의 실현 객체, 단위 방법

그러므로 수정이 필요한 것은 실현 객체와 단위 방법들이다. 여기서 실현 객체와 단위 방법은 그 성격이 다르므로 서로 다른 방법을 사용하여 수정한다. 먼저 실현 객체의 경우는 임의로 생성된 실현 객체는 그 객체가 계속 사용되는지 아닌지는 전적으로 사용자에게 달려 있으므로, 시스템에서는 생성된 모든 실현 객체를 항상 관리 해야 할 필요가 없고, 또한 관리 하여서도 안된다. 그러므로 하나의 객체 부류가 재정의 되었을 때 그 객체 부류에 속하는 실현 객체를 알 수가 없다. 그러므로 모든 실현 객체는 그 실현 객체가 참조 될때 수정하게 된다. 그러기위해서는 실현 객체의 참조 시점에서 그 실현 객체의 객체 부류와 그 상위 객체 부류를 참조하여 재정의 된 것이 있는가를 살펴 보아야 한다.

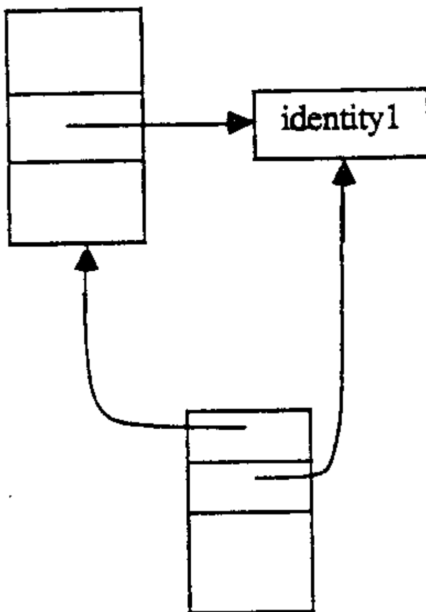
모든 실현 객체는 slot-value, slot-exist-p, slot-makunbound, slot-boundp, 이 4개의 함수를 통하여 참조 수정 된다. 그러므로 이 4개의 함수는 수행 될때 먼저 그 실현 객체의 객체 부류와 상위 객체의 재정의 여부를 검사하고 만약 재정의 되었으면 그 실현 객체를 수정 한다.

객체 부류의 재정의 여부는 객체 부류 내의 identity를 사용하여 관리 된다. 객체 부류 내의 identity는 객체 부류가 재정의 될때 마다 변하게 되어, 그 객체 부류를 참조하는 하위 객체 부류나 실현 객체는 그 실현 객체와 identity에 대한

지시자를 유지하여, 그것의 비교 검사로 자신이 참조하는 실현 객체가 재정의 되었음을 알게된다(그림 3-22).

하나의 실현 객체가 참조되었을 때 수정이 필요하다고 판단되면 그 실현 객체는 표 1에서 보이는 것처럼 이전의 객체 부류의 칸들로 부터 새로운 객체 부류의 칸을 가지도록 수정된다. 그러기위해서는 실현 객체는 객체 부류들에서 가지고 있는 칸들을 지시하여 보관 할 필요가 있다. 그림 3-11에서 보이는 실현 객체의 구조는 그러한 필요에의해 객체 부류와 그 객체 부류의 visible-local-slots, visible-shared-slots를 지시하고 있다.

단위 방법의 경우는 역시 실현 객체와 마찬가지로 재정의 되는 객체 부류에 해당하는 단위 방법과 그 하위 객체 부류의 단위 방법의 수정이 필요하다. 그러나 스페시피케이션에서는 이것을 사용자에게 맡겨 두고 있다. 오직 defclass를 사용하여 사용자가 기술 했던 단위 방법에 대하여서만 재정의 시에 수정을 한다. 이것은 간단히 이전의 객체 부류에서 만든 단위 방법을 제거하고 새롭게 기술된 방법들을 첨가 하는 것이다. 그러나 이것은 실제적으로 큰 의미가 없다. defclass의 사용시 기술하여 만들어진 단위 방법들은 사용자가 defmethod를 사용하여 만든 단위 방법들과 차이가 없으므로 해당 객체 부류의 관련 단위 방법들은 어떤 식으로든 수정이 되어야 의미가 있다. 객체 부류의 재정의에 따른 단위 방법의 수정은 실제적으로 단위 방법이 이전의 객체 부류는 가지고 있었으나 재정의된 객체 부류가 가지고 있지 않는 칸을 참조 할때 문제가 있으므로 필요한 것이다. 그러나 존재하지 않은 칸을 참조하는 단위 방법을 시스템이 적절히 수정해 줄 수는 없다. 단지 그러한 단위 방법이 있으면 제거 하는 것으로 족하다고 본다. 그러나 이전의 객체 부류에 대하여 만들어진 단위 방법이라도 참조 하고 있는 칸들이 수정 후에도 남아 있는 것이라면 제거 할 필요가 없다. 이것을 시스템에서 자동으로 처리 하려면 단위 방법의 정의시 참조되는 객체 부류에 참조되는



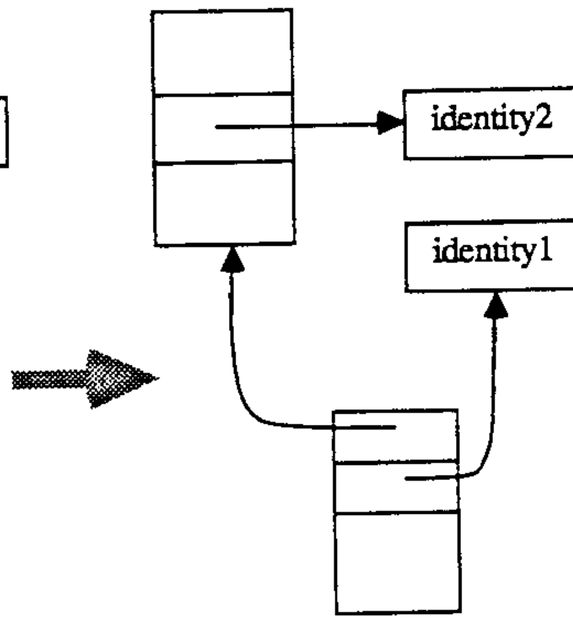
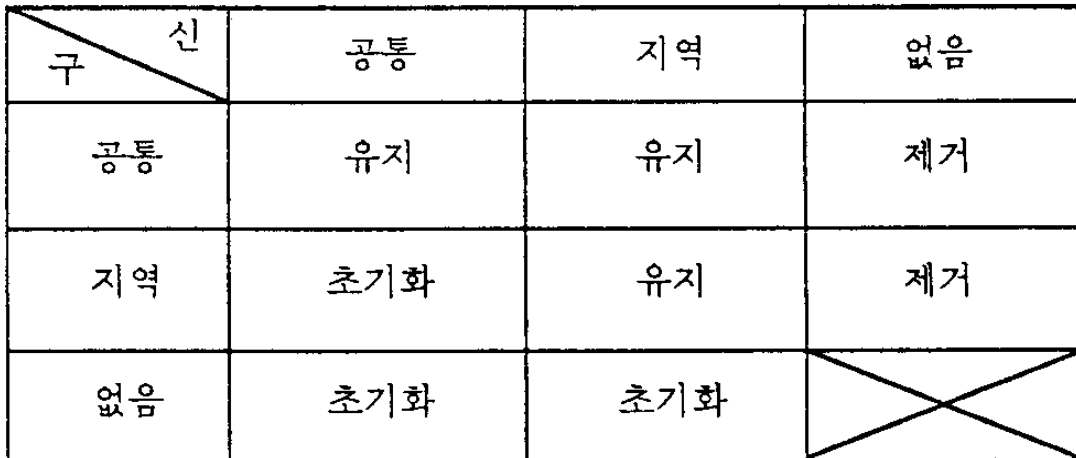


그림 3-22 객체 부류 재정의와 검사



구 \ 신	공통	지역	없음
공통	유지	유지	제거
지역	초기화	유지	제거
없음	초기화	초기화	

표 3-1 객체 부류 재정의에 따른 실현 객체의 수정

칸 들을 같이 기술하여야 한다. 이것은 CLOS의 구문을 확장하여야 하는 것으로 임의로 할 수 없다. 본 구현에서는 일단 스페시피케이션에서 제시한 한도에서 구현 하였다.

나. 다기능 함수의 재정의

다기능 함수는 그림 3-13에서 보는 것처럼 자신의 매개 변수 형태와 단위 방법들 그리고 그 조합을 수행할 방법 조합에 대한 정보를 가지고 있다. 이 각각의 정보는 수행 중에 수정 될 수 있다. 여기에서 방법 조합의 경우 이미 정의 되어 사용되는 다기능 함수의 방법 조합이 새로운 방법 조합으로 바뀌었거나, 사용되던 방법 조합이 재정의 되었을 경우는 어떻게 처리 되어야 하는가가 스페시피케이션에는 서술되어 있지 않다. 이것 역시 스페시피케이션의 문제점으로 방법 조합의 재정의에는 다음과 같은 문제가 있다. 먼저 방법 조합은 다기능 함수의 단위 방법들이 가질 수 있는 자격자들을 정의 하고 있다. 그러므로 방법 조합이 중간에 변하지 않는다면 어떠한 형태로든 최적화(optimization)을 도모 할 수가 있다. 그러나 방법 조합이 재정의 되거나 하면 최적화를 위한 기존의 정보들을 수정 해야 한다. 이때 체계적인 수정을 위하여 본 구현에서는 다기능 함수의 방법 조합은 수행 중에 변할 수 없도록 구현 되었다.

앞에서 서술 했던 단위 방법들의 첨가와 제거는 두가지 부분으로 나뉜다. 먼저 다기능 함수는 자신의 단위 방법들을 모두 가지고 있어서 이것에 대한 첨가와 제거는 간단히 이루어 진다. 그러나 다기능 함수의 빠른 수행을 위하여 만들어진 해쉬 표내의 단위 방법들에 대하여서도 조사하여 첨가와 제거가 되도록 하여야 한다. 그러므로 다기능 함수의 호출시 빠른 수행을 위해 단위 방법의 첨가와 제거시 약간의 작업이 더 필요해진다.

제 4 절 평가와 확장

본 연구에서는 이식성을 위한 일부분을 제외하고는 모두 구현되어 충분한 CLOS의 사용을 뒷받침하고 있다. Texas Instrument 사에서 CLOS의 개발과 같이 만들었던 CLOS는 개발 중의 것으로 하나의 직접적 상위 객체 부류만을 허용하여 나무 구조의 계층 구조만이 가능하고 before나 after, around와 같은 단위 방법 자격자가 허용되지 않는 등 가장 기본적인 기능만을 제공하고 있다.

아직 CLOS는 본격적으로 구현되지 않은 상태이며 많은 사용자등에게 검증받지 못한 것이므로 완전하다고 할 수 없다. 더우기 CLOS의 스페시피케이션은 일부분이 미완인 체로 나머지 부분만이 Common LISP의 기능으로 받아들여졌다. 그러므로 CLOS는 아직 더 개발의 여지가 있을 뿐 아니라 보완의 필요가 있다.

CLOS에서 객체 부류의 계층 구조는 비순환 그래프의 구조를 가지므로 상속 기능은 비순환 그래프 형태를 하나의 열로서 순서화 시킨 객체 부류 우선 순위 열에 따른다. 그러나 객체 부류 우선 순위 열에서 서로간에 상하위 관계가 없는 객체 부류간의 우선 순위가 시스템에의해 지정 되는 것은 의미가 없다. 그림 3-23에서 보이는 계층 구조에서 CLOS의 규칙에 의해서는 E, C, A, D, B, T의 우선 순위열을 가지게 되지만 객체 부류 A가 D나 B보다 꼭 앞에 와야할 이유는 없다. 그러나 이러한 구조상에서 CLOS에서는 객체 부류 E의 우선 순위 열 상에서, 사용자의 필요가 있더라도 D, A, B의 순이 되도록 할 방법이 없다.

이것을 위해서 한 객체 부류는 추가의 우선 순위를 정의 할 수 있도록 하는 것을 제안한다. 이것은 그림 3-23의 변형된 그래프에서 점선과 같은 우선 순위를 객체 부류 E에서 추가로 정의 할 수 있도록 하여 이 관계에 의하여 D,

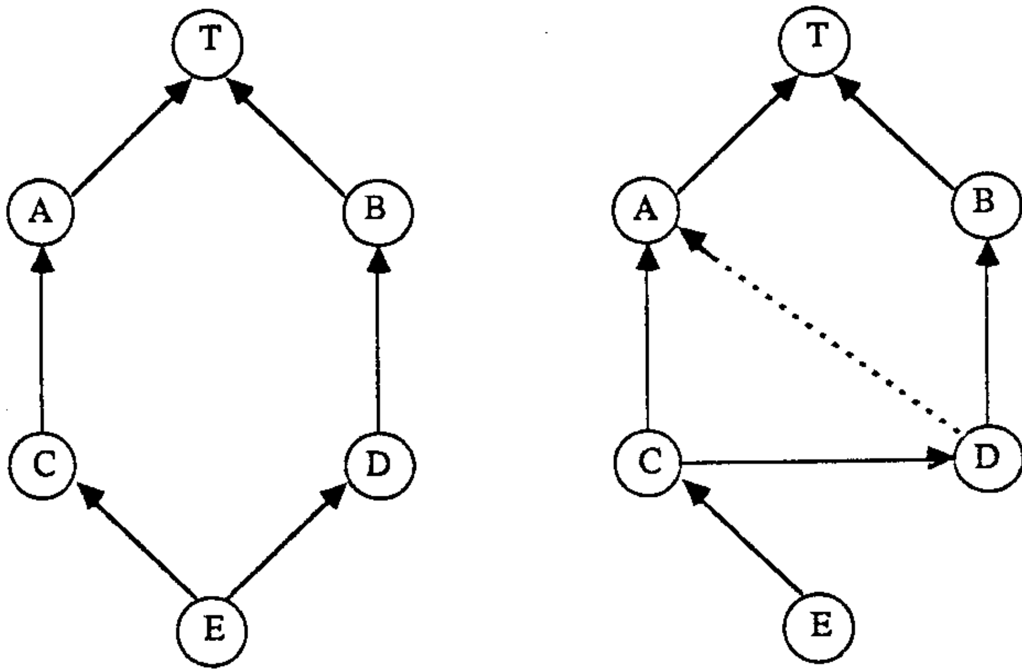


그림 3-23 보조 우선 순위를 첨가한 병형된 계층 구조

A, B 순의 우선 순위 열이 계산될 수 있도록 하는 것이다. 이 새롭게 첨가된 관계는 CLOS의 우선 순위열의 계산을 위한 3가지 규칙 중 동점 처리를 위한 마지막 규칙 보다 상위하는 규칙으로 적용 시켜 사용자가 특기 하였을 때에는 우선 하도록 하는 것이다.

또한 CLOS의 단위 방법은 정의시 매개 변수 특화자를 통하여 특별한 실현 객체나 객체 부류와 관련 맺을 수 있고, 관련 맺어진 실현 객체나 객체 부류에 의해 방법 조합내의 순서가 정해 진다. 이때 매개 변수 특화자의 범위를 객체 부류의 or를 첨가 하여 확장 시키는 것도 고려 해 볼 수 있다. 단위 방법이 수행 될 때 전달 받은 실현 객체를 몇개의 서로 다른 객체 부류의 것을 허용하도록 하고자 할때 객체 부류의 or 관계로 특화자의 범위를 확장 시키는 것이다. 이것은 단위 방법을 순서화 할때 그 단위 방법을 or로 연결된 각 객체 부류들의 단위 방법의 맨 마지막에 위치지우는 것으로 가능하다고 보여진다.

이와 같은 확장은 앞으로 더 자세한 검토가 필요하며 이외에도 많은 부분들이 좀더 풍부한 CLOS를 위하여 연구되어져야 할 것이다.

제 5 절 결론

Common LISP상에서 객체 중심 프로그래밍을 위한 표준 객체 시스템으로서 CLOS는 앞으로 많이 구현될 것이며 많은 응용 프로그램에 사용 될 것이다.

CLOS 구현에 관한 본 연구에서는 기존의 Common LISP 시스템 상에서 인터 프리터의 수정 없이 사용될 수 있는 CLOS를 시스템에 의존적인 일부를 제외하고 모든 기능을 구현하였으므로 기존의 어느 Common LISP 시스템에서든지 CLOS를 사용할 수 있게 하였으며, CLOS의 여러 부분에서 효율성을 고려하

여 빠른 수행을 가능하게 하였다.

본 구현에서는 CLOS의 효율을 저해 하는 계산들을 반드시 필요한 시점에서 수행하여, 관계된 객체들의 재정의가 있기 전에는 다시 계산하지 않도록 관리 하였으며, 객체들의 재정의 역시 영향 받는 객체들의 참조가 있기 전까지는 객체들을 수정하지 않으므로서 불필요하게 반복 수정 되는 일이 없게 하였다. 또한 단위 방법들의 조합은 호출되었을 때 계산된 조합을 해쉬 표에 보관하여 수정하는 관리를 통하여 다기능 함수의 수행이 빠르도록 구현 하였다.

CLOS는 아직 부분적으로 미완된 부분도 있으며 확정된 스페시피케이션도 부분적으로 설명이 미흡하거나 결여된 부분이 있어서 여러가지 보완을 필요로 한다. CLOS의 계층 구조에서 우선 순서열의 계산은 사용자가 의도 대로 조작 할 수 없는 부분이 있어 이 부분에 대한 연구가 더욱 필요 하다. 또한 단위 방법의 매개 변수 특화자의 확장은 사용자에게 더 많은 편리를 줄수 있을 것이다.

본 연구에서 구현된 시스템은 기존의 시스템의 타입 계층 구조의 수정과 다기능 함수 타입의 설정을 통하여 결합 될 수 있고, 더욱 효율적으로 수행 시킬 수 있다. 일관성 있게 작성된 CLOS는 종합적으로 만들어진 Common LISP에 비하여 매우 체계적으로 작성 되었으므로 CLOS를 중심으로 Common LISP의 여러 피처들을 개선할 필요가 있다로 본다.

참 고 문 헌

[BDGKKM88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon, "Common LISP Object System Specificatin X3J13 Document 88-002R," SIGPLAN Notices, V23, Sep, 1988.

[BKKMSZ86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M.Stefik, and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," Proceedings, OOPSLA, 1986, pp 17-29.

[Bo86] Alexander Borgida, "Exceptions in Object-Oriented Languages," SIGPLAN Notices, V21, #10, Oct, 1986, pp. 107-119.

[Bo89] Nick Bourbaki, "Lisp: The Quick and the Dead," AI Expert, Aug, 1989, pp. 23-28.

[Co86] Brad J. Cox, "Object-Oriented Programming," Addison-Wesley, 1986.

[Ga89] Richard P. Gabriel, "The Common Lisp Object System," AI Expert, Mar, 1989, pp. 54-65.

[Gr88] Paul Graham, "Common Lisp Macros," AI Expert, Mar, 1988, pp. 42-53.

[He86] James Hendler "Enhancement for multiple-inheritance," SIGPLAN Notices, V21, #10, Oct, 1986, pp. 98-106.

[Ke89] Sonya E. Keene, "Object-Oriented Programming in Common Lisp," Addison-Wesley, 1989.

[Kn73] Donald E. Knuth, Fundamental Algorithms, V1, Addison-Wesley, 1973, pp. 258-264.

[Ko89] Ray Kolbe, "A Preview of the Common Lisp Object System," JOOP, Jul/Aug 1989, pp. 39-40.

[LC85] Lannar Ledbetter and Brad Cox, "Software-ICs," BYTE, Jun, 1985.

[MR87] Linda deMichiel and Richard Gabriel, "The Common Lisp Object System : An Overview," Proceedings, ECOOP-87, 1987, pp. 201-220.

[Ma82] B. J. MacLennan, "Values and Objects in Programming Languages," SIGPLAN Notice, V17, #12, Dec, 1982, pp. 70-79.

[Mo86] David A. Moon, "Object-Oriented Programming with Flabors," Proceedings, OOPSLA, 1986.

[Ny86] Kristen Nygarard, "Basic Concepts in Object Oriented Programming," SIGPLAN Notices, V21, #10, Oct, 1986.

[Pa86] Geofrey A. Pascoe, "Elements of Object-Oriented Programming," BYTE, Aug, 1986.

[Ro81] David Robson, "Object-Oriented Software Systems," BYTE, Aug, 1981, pp. 74-86

[Sa89] John H. Saunders, "A Survey of Object-Oriented Programming Languages," JOOP, V1, #6, Mar/Apr, 1989, pp. 5-11.

[SB86] Mark Stefik and Daniel G. Bobrow, "Object-Oriented Programming: Themes and Variations," AI Magazine, V6, #4, Win, 1986, pp 40-62.

[Sn86] Alan Snyder, "Common Objects: An Overview," SIGPLAN Notices, V21, #10, Oct, 1986, pp. 19-28.

[St84] Guy L. Steele JR. Common Lips Reference Manual, Digital Press, 1984.

[St86] Rob Stronn "A Comparison of the Object-Oriented and Process Paradigms," SIGPLAN Notices, V21, #10, Oct, 1986, pp. 88-97.

[Sy89] "Getting Ready for CLOS," Symbolics, Jun, 1989.

[WH89] P. H. Winston and B. K. P. Horn, Lisp, 3rd ed, Addison-Weseley, 1989.

[WJ89] Walter Olthoff and James Kempf, "An Algebraic Specification of Method Combination for the Common Lisp Object System," Lisp and Symbolic Computation, V2, 1989, pp. 115-152.

[We86] Peter Wegner, "Classification in Object-Oriented Systems," SIGPLAN Notices, V21, #10, Oct, 1986.

제 4 장 HyKET 시스템의 각 기능통합

HyKET을 구성하는 각 부분 - Lisp 환경, 객체지향프로그래밍 환경, 규칙기반 시스템, 및 사용자 인터페이스- 들은 각각 독립된 기계및 환경에서 구축되었다. 이렇게 독립적으로 개발한 이유는 작업을 최대한 병행시킬 수 있고, 각 부분의 시제품을 다른 부분에 상관없이 빠른 시알내에 얻을 수 있기 때문이었다. 그렇지만 각 부분의 개발에 공통적으로 포함되어 있던 한가지 전제는 Common Lisp을 기반으로 한다는 것이었다. 이것은 후에 각 부분을 한 시스템으로 통합할 때 최소한의 노력을 들여 완성할 수 있도록 하기 위한 고려였다. 이번 3차년도의 연구 내용은 이런 병행성 및 공통전제에 입각한 개발을 통해 구해진 각 부분을 목표 기계인 PC 386상에 통합하여 하나의 통일된 시스템으로 완성하는 것이며, 이 완성된 시스템을 기반으로 데모용 패키지를 개발하는 것이다.

이에 따라 386기계에 HyKET이 통합되었으나 결과적으로 속도의 측면에서 실용적이지 않음을 알 수 있었다. 이런 결과에 따라 실제 데모용 패키지는 보다 빠른 68030 CPU를 사용하는 워크스테이션 상에서 개발되었다. 현재 워크스테이션 시장의 성장추세를 감안할 때 이제 워크스테이션은 결코 낯설은 기계가 아니며 HyKET의 가치는 그에 따라 의미를 갖게된다고 보인다. 본 장에서는 HyKET을 이루는 각 부분의 통합에 관한 내용과 그에 따른 데모 패키지의 구성에 관해 서술하겠다.

제 1 절 Common Lisp 환경

본 프로젝트에서 사용한 Common Lisp은 Unix상에서 운영되고 공개된 소프

트웨어인 Kyoto Common Lisp(KCL)이다. KCL은 원시코드가 공개되어 있기 때문에 쉽게 구할 수 있고 개발 후의 분배에 특별한 허가가 필요하지 않으므로 국내의 여러 사용자에게 분배할 수 있는 시스템을 구축하려는 본 프로젝트의 기본 조건을 만족하고 있다. HyKET의 전체환경을 통합하려면 일단 기반이 되는 Common Lisp 환경을 목표 기계에서 동작하도록 하는 porting 작업이 필요하다. 그러나 KCL의 386 porting은 삼성종합기술원에서 완료한 내용이 있으므로 이를 그대로 사용하기로 하였다.

제 2 절 외부 언어 인터페이스 (FLI: Foreign Language Interface)

Kyoto Common Lisp(KCL)은 Common Lisp의 full-implementation으로 C 언어와의 크로스 컴파일(Cross-compile) 기능으로 인해 주목을 받았다. KCL 컴파일러는 Lisp 프로그램을 컴파일시키면 일단 C 언어 원시 코드로 생성하여 이 C 프로그램을 C 언어 컴파일러로 컴파일 시킨다. 이후에 C 언어 오브젝트 모듈을 Lisp 주소 영역에 적재하기 위한 정보 자료와 함께 Lisp 주소 영역에 적재시키기 때문에 중간 과정으로 C 언어가 생성되는 기능을 크로스 컴파일이라고 한다[그림 2-2]. KCL은 C 언어로 작성된 커널(kernel) 부분과 Lisp 언어로 작성된 라이브러리 함수 및 컴파일러로 구성되어 있다. 따라서 KCL은 Lisp 환경서 비교적 C 언어로의 변환이 용이한 편이라 할 수 있고 C 언어로 커널 부분을 변경하기가 쉽다.

Common Lisp 사용자가 시스템 환경을 활용하기 위한 방법으로 FLI가 제공된다. 여러 Lisp 제품이 FLI를 제공하지만 기능면에서 부족한 점이 있기 때문

에 시스템과의 결합을 쉽게 하기 위해 필요한 FLI의 기능을 분석하고 이를 위한 FLI 함수를 정의하였다. 제안된 함수 및 기능을 사용자의 요구와 편의성에 따라 정리하고 Kyoto Common Lisp 상에 구현하였다.

구현된 기능을 검증하기 위해 전형적인 FLI 요구를 나타내는 실험 프로그램들을 사용하였다. 각 프로그램은 그래픽 처리, 라이브러리 사용 및 계산량 측정 등으로 FLI가 사용자에게 부담을 주는 사항이 없는가를 점검하고 기능이 옳바르게 수행되는가를 확인하였다.

FLI를 여러 시스템에서 수행되는 Kyoto Common Lisp 상에 구현함으로써 운영체제 및 시스템의 자원을 충분히 활용할 수 있게 되었고 특히 라이브러리나 패키지 형태로 이용되고 있는 터미날 입출력 응용 패키지나 윈도우 시스템을 Lisp에서 사용할 수 있게 되어 사용자 인터페이스 부분에 큰 기능을 발휘할 수 있게 되었다. Common Lisp만으로는 이들 기능을 처리하는 프로그램을 개발하는데는 난점이 있었다. 또한 개발 가능할 지라도 많은 시간과 경비가 소모되나 FLI를 통해 이러한 문제점을 해결할 수 있게 되었다.

구현된 시스템은 다양한 계층의 Lisp사용자를 통해 보다 많은 종류의 라이브러리나 패키지 등과의 접속을 시도함으로써 구문 및 기능의 변환이 개선되어야 할 것이다. 여러 종류의 실험 결과에서는 FLI가 오류없이 진행됨을 확인하였으나, 이 시스템의 완전한 검증을 위해서는 대규모의 입력 편집기나 영상처리 환경등을 실제로 구현하여야 기능의 장단점을 분류할 수 있다고 본다.

이 FLI를 전체에 통합시키려면 먼저 고려해 주어야 할 사항이 있다. FLI는 이미 동작중인 LISP에 외부의 오브젝트 코드를 적재해야하므로 incremental한 로딩을 할 수 있는 기능이 필요하다. kcl은 incremental한 loading을 사용하여 ~kcl/c/unixfasl.c 화일에 정의된 si:faslink를 구현하고 있으며 FLI에서 이 함수를 사용한다. 따라서 이것을 해결하는 것이 필요하다. 이 기능은 운영체제에 따라서 각

각 다르게 해결된다.

1) BSD O/S를 사용하는 경우

BSD에서는 시스템의 linker(ld 명령)에서 incremental loading 기능을 제공하므로 이를 이용하면 특별한 문제가 발생하지 않는다.

2) System V를 O/S로 사용하는 경우

System V에서는 시스템 loader인 ld에서 incremental loading을 제공하지 않기 때문에 si:faslink가 동작되지 않는다. 이를 위해 원래의 kcl에서는 자체에서 간단하게 만든 ild(incremental loader)를 제공하고 있으나 이것은 Main Frame에서만 사용이 가능하다. 386상에서는 앞서 언급한 삼성종합기술원의 작업결과를 사용하는 것이 좋다. 거기에서 porting한 sfasl.c란 화일은 원래 Austin KCL에서 제공한 code를 386에 맞도록 수정한 것이다.

본 통합과정에서 목표기계인 386에는 이 sfasl 화일을 이용하여 linking을 해결하였다. 또 데모를 위한 워크스테이션에서는 BSD 4.3을 사용하고 있으므로 시스템이 제공하는 linker를 사용하여 incremental loading을 해결하였다.

제 3 절 규칙 기반 시스템

HyKET의 규칙기반 시스템으로, KCL에 CLIPS(C Language Production System)을 결합하였다. CLIPS는 원래 LISP언어와 연결되어 있지 않고 C 프로그래밍언어로 구현되어 있으며 좋은 인공지능용 프로그램 개발환경을 제공할 수 있다. 규칙 기반 시스템과 기존의 인공지능용 프로그래밍 언어인 LISP 언어와의 연결은 기존의 인공지능용 프로그램에 쉽게 규칙기반 전문가 시스템을 포함 시킬

수 있게 되어 좋은 인공지능용 프로그램 환경을 제공할 수 있다는 점에서 매우 중요하다 할 수 있다. 즉 LISP언어와 규칙기반 시스템으로 하여금 같은 환경을 공유하게 함으로써, LISP의 장점인 다양하고 융통성있는 기능과 규칙기반 시스템의 장점인 규칙에 근거한 추론 기능을 한 프로그램내에서 사용할 수 있게 되어진다. 이러한 요구를 만족시키기 위하여 C 프로그래밍 언어로 구현되어 있는 KCL (Kyoto Common Lisp)과 CLIPS를 연결하여 새로운 인공지능용 프로그램 개발환경(KCLiPS)을 SUN workstation상에 구현하였다. 이러한 환경은 기존의 환경에 비하여 편리할 뿐만 아니라, C 프로그래밍 언어로 구현되어 있어 속도가 빠르며, 보다 값싸게 보다 좋은 프로그램 개발환경을 제공하는 장점을 갖는다.

KCLIPS는 전체 코드가 C 언어로 되어 있으므로 매우 portable하다. 따라서 KCL의 Foreign Language Interface만 동작한다면 어느 기계에나 쉽게 올릴 수 있다. Porting을 위해서는 일단 KCLIPS를 LISP와 상관없이 원하는 기계에 install 한다. Install이 제대로 이루어 졌으면 FLI가 그 기계에서 제대로 동작하고 있는지 확인한다. 이 두 가지만 제대로 이루어 진다면 kclips의 Porting은 '88년 보고서에 있는대로 kclips의 몇 라인을 수정해서 Lisp 상에서 FLI를 통해 접근할 수 있도록 해 주면 된다.[88년 보고서]

제 4 절 Common Lisp Object System [제3장 참조]

인공지능 분야에서 널리 사용되고 있는 LISP 언어에서도 많은 종류의 객체중심 시스템들이 개발되어 왔고 현재도 개발 및 개선 중에 있다. Lisp 언어를 기반으로 하는 객체 중심 시스템들은 FLAVORS(MIT/Symbolics 1979) 로 부터 LOOPS(Xerox PARC 1981), CommonLOOPS(Xerox PARC, 1985), Common

Objects (Synder and Alen, 1986), ExperCommonLISP(1985), GLISP(Stanford, G. S. Norval, 1981), Object LISP(LMI), STROBE(Schlumberger-Doll Research), XLISP(David Bets), CommonORBIT등이 있다. 이러한 각 시스템들은 나름대로의 장단점을 보유하고 이미 많은 사람들이 사용하고 있으며, 이것을 사용하여 개발된 소프트웨어가 많이 있다. 그러나 이러한 시스템들은 좋은 특성을 가짐에도 불구하고, Lisp 언어내에서 공유되지 못하고 있으므로 각각의 좋은 특징을 포함하는 표준화의 필요성이 등장하게 되었다.

CLOS(Common LISP Object System)는 이러한 필요성에 의해 제안되었으며 LISP의 표준화를 위해 만들어진 Common LISP의 한 부분으로 확립되었다. CLOS는 LISP를 기반으로 하는 객체 중심 시스템 중에서 특징적인 장점을 많이 가지고 있는 CommonLOOPS, Common Object, Object LISP의 장점들을 토대로 ANSI STANDARD COMMON LISP의 표준화를 제안하는 X3J13 위원회에서 연구 개발되어 1988년 6월에 Common LISP의 한 부분으로 확정 발표 되었다.

이와같이 Lisp의 표준화를 위해 만들어진 Common LISP의 표준 객체 중심 시스템으로서, CLOS는 객체 중심 프로그래밍의 장점을 최대한 포함하고 있을 뿐만 아니라 사려깊게 연구되어 가능한 좋은 기능들을 일관성있게 조합하였으므로, 매우 우수하고 상세히 정의되고 있고, 효율적으로 구현가능하게 만들어져 있다. 또한 CLOS는 Common LISP의 타입 계층구조, 구문형태등과 상호 잘 부합되도록 만들어져 있기 때문에 기존의 Common Lisp 프로그래머들에게 생소하지 않은 구문형태를 제공하고 있고 개념적으로 일관성을 유지시켜 준다.

이와같은 CLOS의 구현은 타입 계층구조의 부분적 확장등을 필요로 하므로 KCL이나 AKCL과 같은 기존의 Common LISP에서는 인터프리터의 수정을 요구한다. 이러한 일은 필요한 일이긴 하나, 이미 구현된 많은 시스템들을 수정하

기가 어렵고, 또한 거의 대부분의 CLOS 기능이 수정없이 구현 가능하며, 객체 중심 프로그래밍의 의미상 수행에 지장을 초래하지는 않으므로, 본 연구에서는 기존 인터프리터의 수정없이 Common LISP 만을 사용하여 CLOS를 구현하였다. 그러므로 이미 구현된 기존의 어떠한 Common LISP 시스템에서든지 사용가능하며, 또한 인터프리터의 수정시에도 간단한 수정으로 접합시켜 확장시킬 수 있도록 구현하였다. 또한 스페시피케이션이 확정되어 발표된지 얼마되지 않아 개발단계의 몇몇 구현을 제외하고 아직 완벽한 구현이 없는 상태에서 대부분의 피쳐를 포함하는 CLOS를 구현하는 것은 CLOS의 빠른 실용화에도 큰 의의가 있다고 하겠다.

이상과 같이 CLOS는 오직 Common LISP의 함수만을 사용하여 구현되었고, KCL은 Common LISP의 전체를 구현해 놓았으므로 KCL이 동작하는 곳에서는 항상 CLOS가 동작하게 된다. 이 CLOS를 이용하여 구현한 간단한 sample Demo Package는 뒤에서 설명하기로 한다.

제 5 절 GNU-Emacs

HyKET에서 제공하고 있는 Editor는 GNU-Emacs이다. GNU 환경은 MIT에서 개발하여 계속 그 범위를 넓혀 가고 있는 공개 소프트웨어인데 현재의 System에서는 가장 최근 버전인 18.55를 사용하고 있다. GNU 환경에 대한 install 및 Customization은 패키지 자체에 포함되어 있으므로 여기에서는 설명을 생략한다.

제 6 절 X-Window System

X Window는 널리 알려진 Portable한 Window System이다. 이것은 MIT의 Athena Project 수행을 통해 개발되었으며 Workstation상에서 Network-transparent하고 Machine-independent한 Window System 환경을 제공해 준다. 이런 두가지 특징은 X를 Window System의 표준으로 사용하는 경향을 야기했으며, 이를 이용한 다양한 응용 프로그램들이 공개되어있다.

HyKET에서는 이런 세계적 추세에 맞추어 X를 기반으로 하는 윈도우 시스템을 사용하고 있다. 현재 사용중인 X의 버전은 386의 경우에 X11 Release 2 이며, 데모용 워크스테이션에서는 X11 Release 3를 사용하고 있다.

이상과 같은 component를 종합한 HyKET의 구조는 그림과 같다.

제 7 절 CLX (Common Lisp X Interface)

CLX는 X Window에서 제공하는 여러가지 기능들을 Common Lisp에서 사용할 수 있도록 Texas Instruments Inc. 에서 개발한 서브루틴 라이브러리이다. 여기에는 Xlib의 함수들이 함수나 매크로의 형태로 포함되어 있으며, 이를 통하여 Lisp 사용자들이 사용자 인터페이스를 구성하는데 있어서 표준화된 GUI (Graphic User Interface)를 사용할 수 있게되었고, X Window의 장점들 - Network Transparency, Machine Independency - 을 수용할 수 있게되었다.

CLX로 작성된 프로그램은 X client로서 수행되며 X 서버와의 통신은 socket을 이용해서 스트림을 연결함으로써 이루어진다. CLX는 X Window의 기본적인 라이브러리인 Xlib와의 인터페이스이므로 매우 낮은 수준의 기능만을 사용할 수가 있다. 예를 들어 메뉴나 OSF/Motif의 Widget등은 사용할 수가 없고 Xlib의 기본요소인 window를 이용해야만 한다. 그러나, CLUE(Common Lisp User-

interface Environment)와 같이 CLX 위에서 작성된 패키지를 사용한다면 좀더 쉽게 사용자 인터페이스를 구축할 수가 있다.

CLX는 Common Lisp와 X Window가 수행되는 환경이면 어디서든지 수행이 가능하며, HyKET에서는 다양한 기종 및 OS를 지원하는 KCL(Kyoto Common Lisp) 과 X11 Release 3를 사용하였다.

제 8 절 Demo Package

1. FLI Demo

Foreign Language Interface, 즉 Lisp에서 외부 언어의 Object Code를 사용하는 것을 보여주기 위하여 네가지의 경우를 소개하기로 한다.

Unix 사용자가 사용하는 몇개의 대표적인 라이브러리는 다음과 같다.

- Mathematical Library : libm.a
- Curses
- Network Service Library : libnsl_s.a

이 중 위의 두가지 경우에 대해 Demo Program을 작성하였다.

한편 X Window System의 응용 프로그램 중 하나인 xwud (X Window Undump)를 Lisp과 연결하여 호출할 수 있도록 하여 X 상의 어떠한 application도 Lisp과 연결할 수 있음을 보였다.

Math Library

BSD Unix의 매뉴얼에 따르면 30여개의 Math 라이브러리 함수를 제공하고

있다. 그 중 Common LISP에서도 제공해 주는 sqrt(Square Root)를 제외한 20여개의 함수를 FLI를 써서 연결하였다. Interface의 화일 내용은 그림과 같다.

Curses and Termcap Library

curses 라이브러리는 Unix에서 제공되는 대표적인 terminal independent, 화일 관리 패키지이다. 이것은 termcap화일(System V의 경우에는 terminfo)를 참조하여 동작하므로 여기에 기록만 되어 있으면 어느 terminal이나 동작하며 이 점이 curses를 널리 사용하도록 유도한 동기가 되었다.

데모용 프로그램은 curses package의 대표적인 예제 프로그램인 twinkle을 Lisp에서 사용하도록 한다. twinkle은 terminal 상에 여러가지 pattern을 프린트해 줌으로써 화면상에 원하는 문자를 프린트 할 수 있음을 보여주는 Demo 프로그램이다. Interface 화일의 내용은 다음과 같다.

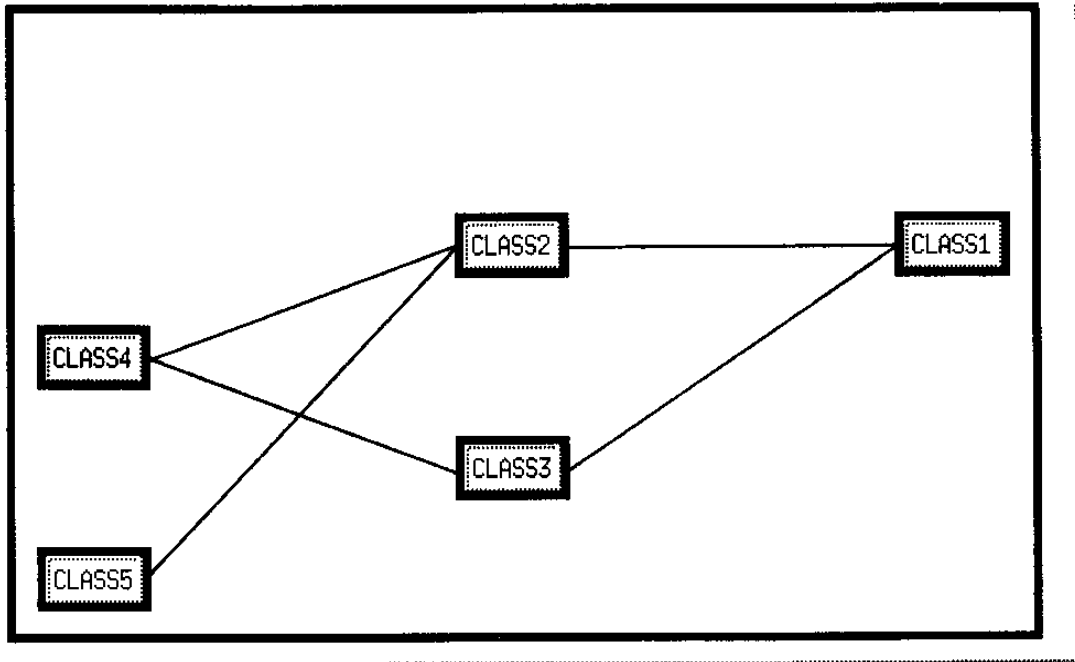
xwud

X Window는 화면에 디스플레이되는 이미지의 형식을 다양하게 제공하고 있으며 서로 다른 이미지 형식간에 변환시켜주는 프로그램도 많이 개발되어 있다. 이중 사용자가 가장 쉽게 취급할 수 있는 것이 xwd 형식이며, 같은 이름의 xwd란 프로그램은 임의의 윈도우에 대해 그 윈도우가 가지고 있는 이미지를 xwd 형식으로 화일에 저장하는 일을 한다. xwud 프로그램은 이렇게 저장된 화일을 다시 이미지로 재생시켜주는 기능을 한다. xwd나 xwud는 C로 개발되어 있으나 이를 FLI로를 이용하여 Lisp에서 호출할 수 있게 하였다. Interface화일의 내용은 다음과 같다.

2. CLOS Demo

CLOS의 Demo Package는 두가지가 제공되고 있다. 하나는 5개 정도의 간단한 class를 정의하고 그 각각의 method를 정의한 뒤, 각 클라스에 해당하는 인스턴스를 정의한다. 그 뒤에 각 인스턴스에 대해 정의된 함수를 호출하여 어떤 순서로 실행되는가를 보여주는 demo이다.

두번째 demo는 stream의 hierachy를 정의하고 있다. 이것은 가장 low level의 primitive function을 정의해 주면 곧장 사용할 수 있는 실제의 hierarchy이다. CLOS Demo의 예는 그림 4-1에 나타나 있다.



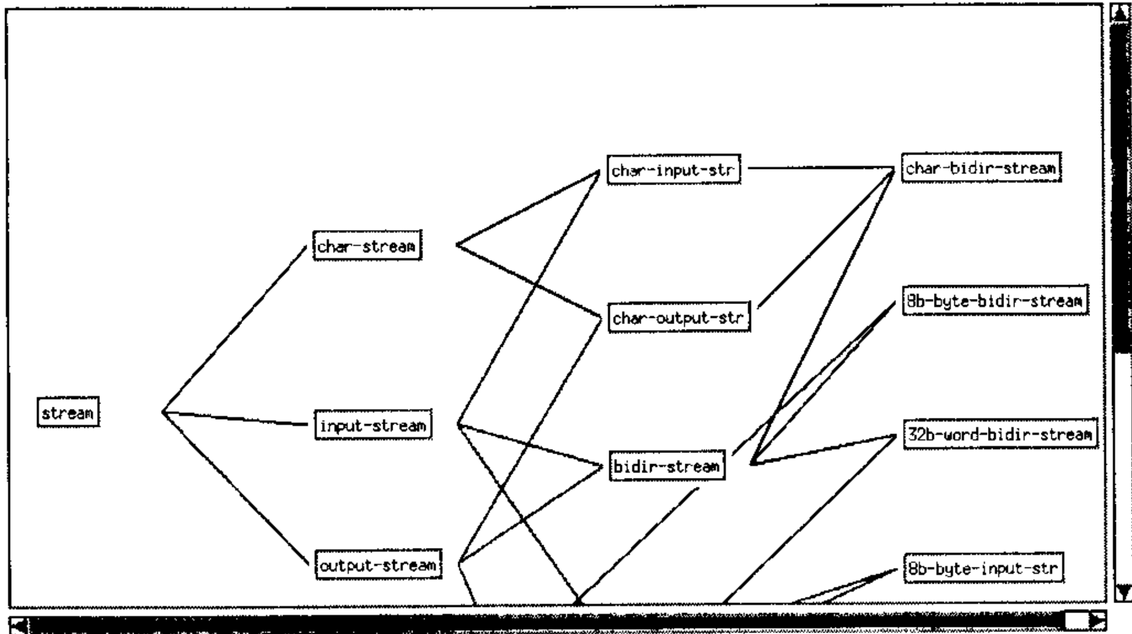


그림 4-1 CLOS Demo

제 5 장 HyKET을 이용한 PC 고장진단 전문가 시스템의 구현

제 1 절 서론

PC(Personal Computer)의 기능이 확장되고 다양화되어짐에 따라 사무용, 연구 및 개발용으로 여러 분야에 널리 이용되고 있다. PC의 종류에는 XT, AT, 386 머신등의 여러가지가 있다. 같은 종류의 PC에서도 기능 향상 및 시스템의 에러를 고치기 위한 목적등으로 인하여 부품 또는 보드의 사양이 서로 다르고, 내부에 사용된 부품의 종류가 서로 다를 수 있다. 이와 같이 PC 시스템의 구성 및 기능등이 빠른 속도로 변화한다는 점이 달른 분야와 비교하여 큰 특징이라 할 수 있다. 따라서 PC의 유지 및 관리는 매우 중요한 문제가 되고 있다.

PC의 고장은 그 원인이 다양하며 그 종류가 많아, 대부분의 고장은 기술자의 경험적 지식에 의해 진단되고 있다. 이러한 고장 진단의 지식은 이전, 전수 및 정리하기가 어렵다.

대부분의 컴퓨터 회사에서는 신입 A/S 요원의 교육을 위한 자료나 방법이 없어 실제 고장을 해결하는 현장에 나가 고장 진단에 관한 경험을 쌓도록 하고 있다. 또한 PC를 사용하고 있는 고객으로부터의 고장 신고 내용은 비교적 간단한 경우가 많아 사용자에게 간단한 고장 처리 능력을 부여함으로써 A/S 요원의 부담을 덜어줄 수 있는 경우도 있다. 특히 지방에는 매우 적은수의 A/S 요원들이 한곳에서 근무하기 때문에 서로 의문점을 상의할 수가 없어서 약간의 문제가 있어도 본사로 우송하는등 많은 부담을 주고 있다. 그러나, 여러 전문가들이 제시한 종합된 고장 진단 지식을 이들에게 제공함으로써 자세한 전문 지식이 없는 대리점이나 지방근무 A/S 요원들도 효율적으로 쉽게 고장을 고칠 수 있다. 따라

서, 위의 여러 경우에 사용될 수 있는 시스템으로서, PC의 고장 증상에 따라 상세한 고장 원인 및 고장 부분을 알려주어 고장 수리를 도와주는 기능을 갖는 PC 고장 진단 전문가 시스템을 개발하였다.

이 시스템이 가지는 기능은, 고장 진단시 고장난 PC의 증상에 따라 조치해야할 내용을 지시해 주거나, 고장의 원인이 되어 교체해야할 Board나 부품 및 고장의 발생 원인등을 지적해 준다. 또한, 고장의 원인에 따라 조치해야할 조치 사항도 제시해 준다.

제 2 절 고장 진단 방법의 종류

고장 진단 방법은 일반적으로 크게 2가지로 구분된다[1]. 첫째는, 시스템 각 부품의 기능과 연관관계를 기술해준 후, 관측된 시스템의 동작 상태에 대한 자료를 바탕으로 고장을 진단하는 방법이다[2, 3]. 시스템내 각 부품이 정상적으로 작동시 기대되는 값과 현재 관측된 자료 사이에 불일치하는 사항이 발생시, 이를 가장 잘 설명할 수 있는 고장난 부품을 찾아내는 방법이다. 두번째 방법은 경험적 지식을 이용하는 방법으로서, 일반적인 현상, 통계적인 직관, 전문가의 과거 경험등을 규칙으로 바꾸어 진단에 이용한다. 시스템의 구조나 각 부품의 기능등에 대한 자세한 표현이 요구되지 않으며, 다만 경험적 지식만이 중요한 역할을 한다. 이 방식의 대표적인 시스템에는 MYCIN[4]이 있다. 대부분의 고장 진단 시스템은 이러한 경험적 지식을 이용하는 방법이다. 이 연구에서 사용한 방법은 후자의 경험적 지식을 이용하는 방법이다.

진단 분야에서 사용되는 일반적인 진단 방법으로서 가설을 위주로 진단하는 방법은 다음과 같다[5, 6, 7]. 초기에 고장을 일으켰을 가능성이 있는 부품 가설

들의 집합을 만든다음 주어진 판단 근거에 따라 고려할 가설의 수를 줄이고, 고려할 가설들을 가능성의 순서로 서열을 정한다. 다음은 고려할 가설의 집합에서 하나의 가설을 선택하여 이 가설이 고장 원인이 될 수 있는지를 확인하기 위하여 이 가설과 관련된 징후들을 조사한다. 필요한 경우 시스템에서 검사할 곳을 지정해 준다. 수집된 모든 징후들을 이용하여 가설을 평가하여, 이 평가 결과에 따라 가설을 accept하거나 reject한다. 본 연구에서 개발한 시스템도 이러한 일반적인 고장 진단 방법을 기초로 하였다.

제 3 절 PC 고장 진단 시스템의 구조

시스템은 고장 진단기, 지식 베이스, 고장 진단 지식 습득 시스템, 설명 기능, 사용자 인터페이스등으로 구성된다. 본 시스템에서도 편리한 사용자 인터페이스를 위해, 윈도우 시스템과 메뉴 시스템을 기초로 하여 개발하였다. 그림 5-1은 고장 진단 전문가 시스템의 구성도이다.

1. 지식 베이스

고장 진단 전문가 시스템의 지식베이스는 고장 진단 지식베이스와 조치사항 지식베이스로 구성되어 있다.

가. 고장 진단 지식베이스

고장 진단 지식베이스에는

(1) 고장 진단 사항의 ID와 이름,

(2) 고장을 나타내는 가설들의 계층구조로서 상위 가설과 부가설들이 표현되며,

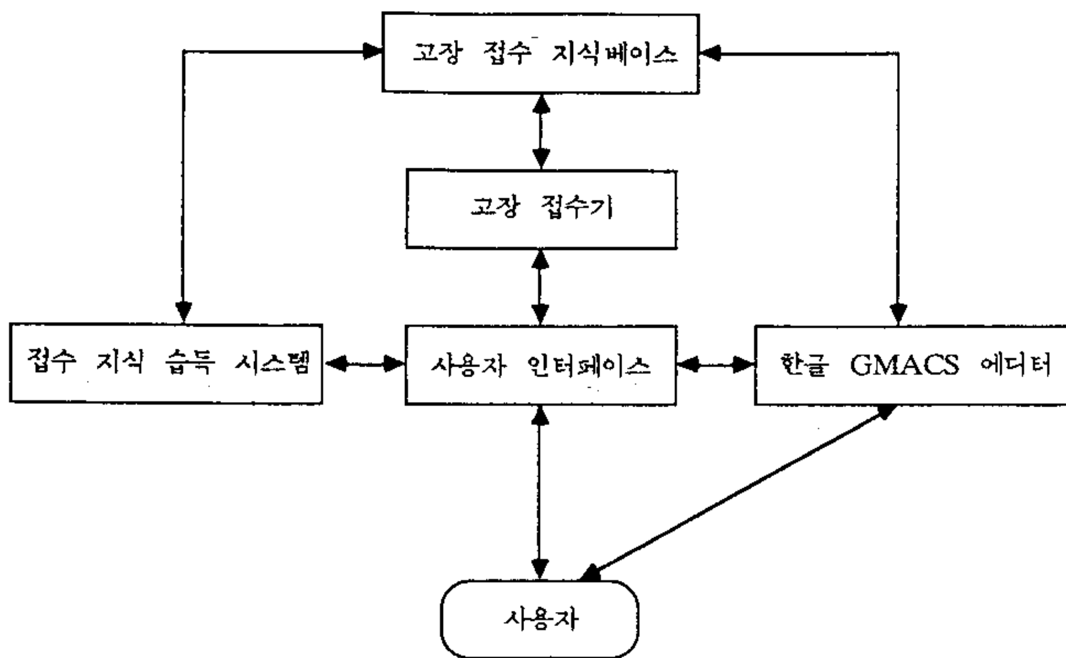


그림 5-1 고장 진단 전문가 시스템의 구성도

(3) 고장이 발생했다고 간주할 수 있는 징후들(triggers)

(4) 고장 발생시 반드시 나타나는 현상, 증상들(must-satisfy)

(5) 해당 가설이 고장을 일으킨 원인이라고 판단될 때 취할 조치사항을 나타내는 ID등이 표현된다.

나. 조치사항 지식베이스

조치사항 지식베이스에는,

(1) 조치사항의 ID와 이름,

(2) 조치할 조치사항의 제시와 그에 따른 행동을 취했을 때 얻은 결과를 묻는 말

(3) 왜 이런 조치사항을 취하라고 하는지에 대한 보다 상세한 설명

(4) 조치사항에 따라 행동을 취했을 때,

(가) 고장을 고친경우 취할 다음 행동으로서 고장 진단을 종료하거나 다른 고장 원인을 계속 찾도록 한다.

(나) 고장을 못 고친경우 취할 다음 행동으로서 다른 조치사항을 제시해 주거나, 일단 현재 고려중인 가설을 무시하고 다른 고장 가능성이 있는 가설을 가설 집합에서 선택하여 다시 검증을 반복하게 한다.

2. 고장 진단기

고장 진단기의 기능은 다음과 같다.

가. 메뉴를 통하여 눈으로 관측 가능한 증상들을 입력받는다.

나. 입력된 증상들을 이용하여 고려할 가설의 집합을 만든다. 입력된 징후가 가설의 triggers부분에 있는 모든 가설을 고려할 가설 집합에 포함시킨다.

다. 하나의 가설을 가설 집합에서 선택한다. 가설 집합에서 가설을 선택하는

기준은, 고장이 발생하는 빈도수가 많은 것이 먼저 선택되도록 하였다.

라. 선택된 가설의 must-satisfy 부분이 만족되는지를 확인한다. 이때, 테스트할 지점을 제시하고 그점에서의 시스템 상태가 원하는 것인지를 테스트한다.

마. 위의 must-satisfy 부분이 만족되는 경우 필요한 조치사항을 제시한다. 이와 함께 조치사항을 적용했을 때의 결과를 묻는다. 이때 must-satisfy 부분이 만족되지 않는 가설은 무시된다.

바. 조치사항의 지시사항에 따라 행동을 취했을 때,

(1) 고장을 고쳤으면 사용자의 의도에 따라 다른 고장 원인을 찾거나 모든 동작을 끝내고,

(2) 고장을 고치지 못했다면, 이 가설은 무시하고 고려할 가설 집합내의 다른 가설에 대해 단계 3부터의 진단을 반복한다.

(3) 더이상 고려할 가설이 없으면 진단을 종료한다.

위의 고장 진단 과정을 그림으로 나타내면 그림 5-2와 같다.

이와같이 고장 진단이 수행되는 과정의 한 예는 그림 5-3와 같다.

3. 설명 기능

설명 기능은 고장 진단 수행시 현재 하고 있는 질문에 대한 자세한 설명을 해 주는 기능이다. 설명 기능은 고장 진단시 가설의 must-satisfy부분들을 검증할 때와, 적용할 조치사항을 제시하는 경우에 설명 기능을 이용할 수 있다. 가설의 must-satisfy부분을 검증할 때, 고려중인 가설이 선택되기 위해 triggers로 작용한 증상과 고려중인 고장의 가설에 대한 정보등 현재 하고 있는 질문에 대한 설명을 해준다. 적용할 조치사항을 제시하는 경우의 설명기능에서는 위의 경우에 대한 설명과 함께 현재 제시한 조치사항에 대한 설명을 해준다. 또한, 각 질문에

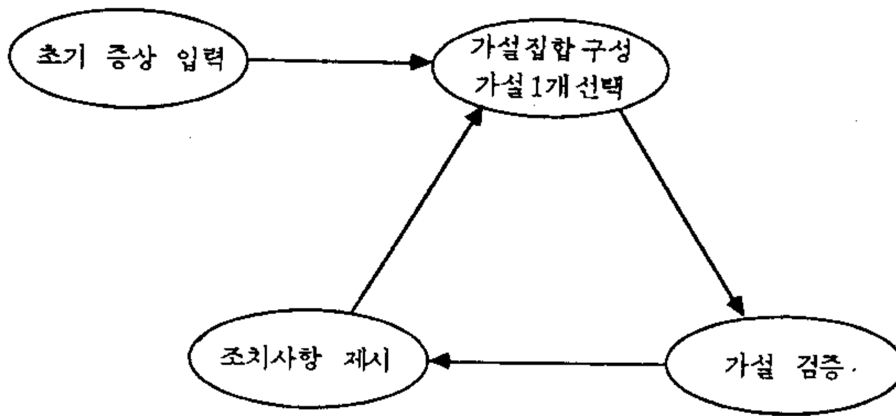


그림 5-2 고장 진단 과정

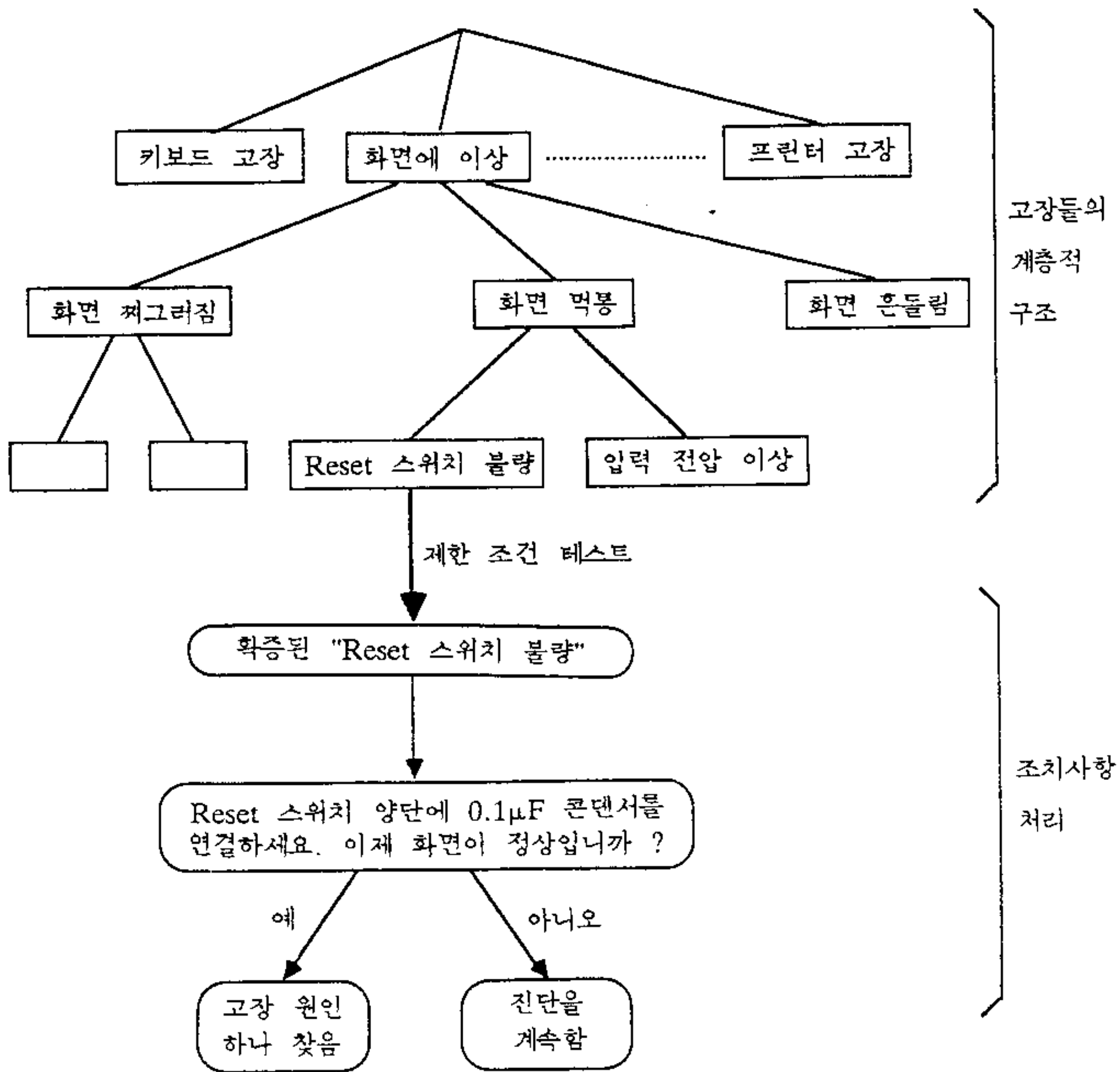


그림 5-3 고장 진단 수행의 한 예

대해 보다 상세한 형태의 질문 내용도 메뉴의 "자세한 질문"을 통하여 제공되고 있다. 설명기능은 각 질문에 대한 응답시 키보드를 통하여 "w"를 입력하거나, 메뉴의 "설명 기능"을 선택함으로써 사용할 수 있다.

제 4 절 HyKET에서의 PC 고장 진단 시스템 개발

PC 고장 진단 시스템은 IBM PC/AT에서 Golden Common LISP으로 구현되었으나 본 연구에서는 이 시스템을 기본으로 몇 가지 기능을 추가하고 지식베이스를 확장하여 HyKET에서 다시 구현하였다. 구현 언어는 KCL(Kyoto Common LISP)을 사용하였고 Sony News Workstation에서 수행된다. 고장 진단 수행중, 모든 상황에서 현재 고려 중인 요소에 대한 위치를 알려 주는 윈도우를 추가하여 사용자의 이해를 돕도록 구현하였다. 구성요소의 위치를 알려 주는 윈도우는 2개로서, 상세한 수준에서 구성요소의 위치를 지적해 주는 윈도우와, 상위수준의 구성요소중 어느부분에 속하는지를 지적해주는 윈도우가 있다. 각 윈도우에는 회로 또는 보드의 Block Diagram과 scanner로 받은 영상, 또는 카메라로 받은 영상을 디스플레이 할 수 있으며 , 특정요소를 지적하는 기능으로서 영상에서는 화살표로, Block diagram에서는 해당 블럭을 빨강 색깔로 칠하여 표시하도록 하였다.

제 5 절 결론

본 연구에서는 고장 증상에 따라 테스트할 곳을 지정해 주고 수집된 징후들을 이용하여 고장 원인을 찾아주며 적절한 조치사항을 제시해 주는 PC 고장 진

단 전문가 시스템에 대해 기술하였다.

이 전문가 시스템은 HyKET 환경에서 KCL로 구현하였고, 시스템 및 지식 베이스에는 한글을 사용하였다. 이 시스템의 개발을 통하여 고장 진단의 접수를 도와주며, 고장 진단 지식을 기록, 수정 및 확장할 수 있는 기능등을 제공함으로써 고장 진단을 자동화할 수 있는 계기를 제공하였다. PC 고장 진단 분야에 전문가 시스템을 이용함으로써 전문가 시스템이 여러 분야에 사용될 수 있는 가능성이 있음을 보여주었다. 또한 HyKEY을 이용하여 여러분야에 대한 전문가 시스템을 구축할수 있는 가능성을 보여주었다.

참 고 문 헌

[1] Larner, D. L., A Recursive Expert Troubleshooting System Utilizing General and Specific Knowledge, in: *Proceedings The Second Conference on Artificial Intelligence Applications*, (1985) 34-41.

[2] Reiter, R., A Theory of Diagnosis from First Principles, *Artificial Intelligence* 32 (1987) 57-95.

[3] Davis, R., Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence* 24 (1984) 347-410.

[4] Buchanan, B. G. and Shortliffe, E. H.(Eds.), *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project* (Addison-Wesley, Reading, MA, 1984).

[5] Horvitz, E. J., Heckerman, D. E., Nathwani, B. N. and Fagan, L. M., Diagnostic Strategies in the Hypothesis-Directed PATHFINDER System, in: *Proceedings The First Conference on Artificial Intelligence Applications*, (1984) 630-636.

[6] Merritt, B., Anatomy of a Diagnostic System, *AI Expert* 2(9) (1987) 52-63.

[7] Kahn, G. and McDermott, J., The MUD System, in: *Proceedings The First Conference on Artificial Intelligence Applications* (1984) 116-122.

제 6 장 결론

국내에 인공지능이 소개되면서 각 학교, 연구소 및 기업체에서 인공지능 시스템 개발에 관심을 기울이기 시작하였으며 이로 인하여 인공지능 시스템 개발에 필요한 개발도구들이 필요하게 되었다. 특히 다양한 지식 표현방법과 강력한 개발환경을 제공하는 복합형 인공지능 개발시스템이 국내에서 개발될 필요성이 대두되었다.

본 연구과제에서는 국내에서 생산될 인공지능 워크스테이션에서 사용할 수 있는 복합형 인공지능 개발 시스템 HyKET을 개발하였다. HyKET은 LISP 개발환경을 기반으로 하여 한글환경, 객체지향적 프로그래밍 환경, 규칙기반 시스템 및 사용자 인터페이스를 통합한 인공지능 개발시스템으로서 규칙, 객체, LISP 함수 등의 여러 형태로 지식을 표현할 수 있는 복합형 인공지능 시스템이다.

3차년도 연구에서는 1,2차년도에 이어 LISP개발 환경의 개선으로 Window를 이용하는 KOPS를 개발하였으며, CLIPS 규칙 기반 시스템과 KCL을 결합하여 LISP 코드와 CLIPS의 규칙을 혼용하여 사용할 수 있도록 하였다. 규칙 기반 시스템에서 불확실성 처리방법을 구현하였고 진리유지 기능을 구현하였다. 객체지향언어로 CLOS 시스템을 선택하여 확장중이다. 또한, 편리한 사용자 인터페이스 구축을 위해 X Window 환경을 구축하였으며, CLOS Browser를 설계, 구현하였다.

1,2차년도에 개발된 각 시스템들은 각기 독자적으로 사용되어 인공지능 시스템 개발에 사용될 수 있다. 하지만 이들이 하나의 시스템으로 통합되어 단일 시스템인 HyKET이 구성되기 위해서는 1,2차년도에 연구된 구성요소를 기반으로 하여 통합 작업이 필요하다. 이를 위해 타겟 머신으로 선정된 PC 386을 기반으로, LISP 환경과 객체 지향적 프로그래밍 환경인 CLOS, 규칙기반 시스템, 사용

자 인터페이스등을 통합하였고, Sony Workstation에 HyKET 환경을 구축하였다. 각 구성요소별로 상세 설계를 완료하였고, PC 고장 전문가 시스템 개발을 통하여 HyKET의 사용 가능성을 입증하였다.

주 의

1. 이 보고서는 과학기술처에서 시행한 특정연구개발 사업의 연구보고서이다.
2. 이 연구개발내용을 대외적으로 발표할 때에는 반드시 과학기술처에서 시행한 특정연구개발사업의 연구결과임을 밝혀야한다.