4세대 언어에서의 역공학 환경 구성

진 영 배 † 왕 창 종 ††

요 약

소프트웨어의 종류가 다양하고, 크기가 커짐에 따라서 유지 보수 문제는 더욱 복잡하고 어렵게 되었다. 따라서 소프트웨어의 유지 보수가 소프트웨어 생명 주기에서 가장 비용을 많이 차지하는 부분이 되었다. 또한 범용 소프트웨어를 개발하거나, 사용하기 위해 단말기 사용자에게도 쉽게 느껴지는 4세대 언어가 앞으로 많이 적용되는 개발 환경으로 될 것이다.이에 따라서 프로그램의 유지 보수 측면에서 많은 연구가 이루어진 COBOL, C, FORTRAN, PASCAL등과 같은 언어처럼 4세대 언어로 작성된 원시 코드에 대해서도 프로그램의 이해를 위한 분석 및 재사용 방법의 연구가 이루어져야 한다. 본 연구에서는 4세대 언어로 작성된 원시 코드로부터 메타 언어 형태로 변환하고 프로그램 이해를 위한 문서 생성기(Document Generator), 보고서 생성기(Report Generator), 모듈 분석기(Module Analyzer), 코드 번역기(Code Translator)를 설계하고 구현함으로서 4세대 언어에서의 역공학 도구를 제안하여, 프로그램의 이해 및 관리를 효율적으로 하는데 목적이 있다.

A Study on Construction of Reverse Engineering Environment in Forth Generation Language

Young Bea Jin † Chang Jong Wang ††

ABSTRACT

With the diversified and enlarged softwares, the issue of software maintenance became more complex and difficult consequently, the cost of software maintenance took up the highest portion in the software life cycle. At the same time, in order to develop or use software for general computers the development environment will be changed to incorporate user-friendly 4GL (Fourth generation Language). Therefore, it is required to take a closer look at the languages such as COBOL, C, FORTRAN, PASCAL which have source code comprised of 4GL and investigate the method of analysis and reuse for program understanding since a lot of research has been done with these languages in program maintenance. The purpose of this paper is to propose reverse engineering tool in 4GL and find an effective way of understanding and maintaining the program by transforming source code comprised of 4GL to meta language and designing and implementing Document Generator, Report Generator, Module Generator, Code Translator for program understanding.

1. 서 론

소프트웨어 공학의 주요 업무중의 하나가 새로운 소프트웨어 개발에서 기존의 소프트웨어 유지 보수로 변화하고 있다. 소프트웨어의 종류가 다양하고, 크기가 커짐에 따라서 유지 보수 문제는 더욱 복잡하고 어렵게 되고 프로그램 에러의 교정이나 요구 사항의 변경, 사용자에 대한 요구가 점차 증가되면서 소프트웨어의 유지 보수가 소프트웨어 생명 주기에서 가장 비용을 많이 차지하는 부분이 되었다. 새로운 시스템 개발에 많은 인력을 고용하는 것보다 기존 시스템 유지 보수에 더 많은 인력이 투입되어 실제로 소프트웨어의 유지 보수를 위해 투입되는 비용이 총 소프트웨어 예산의 2/3 정도가 되고, 날로 차지하는 비중이 커지고 있다[5].

소프트웨어 공학에 대한 접근 방법의 변화도 개발과 유지 보수를 포함하여 기존 시스템을 지

† 정 회 원:충청전문대학 사무자동화과 조교수
†† 정 회 원:인하대학교 전자계산공학과 교수
논문접수:1995년 5월 16일, 심사완료 : 1995년 6월 28일

속적으로 강화하는 소프트웨어 진화(software evolution)로 이어지고 있다. 실제로 현재 존재하는 많은 대규모 시스템의 형태는 수백만 라인의 코드와 프로그램 이해를 위한 관련된 문서의 부족, 부실한 문서화, 많은 수정, 특히 수정시 규칙보다는 예외 처리가 많은 것 등 유지 보수 작업이 상당히 어렵다. 소프트웨어 진화의 핵심은 원시 코드로부터 추상화된 시스템의 구조를 생성하여 프로그램을 이해하고 이에 대처하는 것이다 [10].

또한 소프트웨어 개발에 있어서의 경험 부족, 체계적이지 못한 문제 해결 방식, 효율적인 프로젝트 관리능력 결여, 소프트웨어 자체의 복잡성 증가 및 개발 도구의 부족등 소프트웨어 위기에 대한 대처의 방안으로 1980년대 후반부터 CASE와 4세대 언어가 등장하였다. CASE의 경우는 상당한 연구와 보고서 및 논문이 계속 발표되고 있으나 4세대 언어에 대해서는 특별한 대안이 없이 실제 업무 분야에서 상당히 많이 사용하고 있는 언어로 자리잡고 있다.

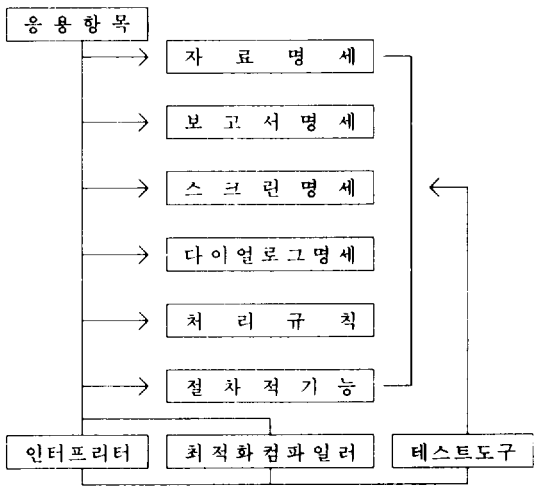
4세대 언어는 이용하는 작업에 따라 전통적인 소프트웨어 시스템 개발 수명 주기에서 프로그래밍(단위, 통합, 테스트 포함) 부분만이 개선되는 경향이 있으나 진정한 생산성 향상을 위해서는 명세 작성과 설계 단계에 관한 노력도 감소되어야 할 것이다. 즉, 4세대 언어는 단말사용을 통하여 응용 프로그램의 개발을 지원하는 프로그램 자동화의 측면을 갖고 있다. 범용 컴퓨터용 소프트웨어를 개발하거나 사용하기 위해 단말기 사용자에게도 쉽게 느껴지는 4세대 언어가 많이 사용되고 있고, 앞으로도 여러 분야에 적용되는 개발 환경으로 될 것이다.

본 논문의 구성은 2장에서 4세대 언어에 대한 고찰을 하고 3장에서는 역공학 도구의 필요성과 구조를 알아보고 4세대 언어에서의 역공학 형태를 제안한다. 4장에서는 4세대 언어에 대한 메타 언어의 설계와 변환 규칙을 작성하고, 5장에서는 역공학을 위한 환경을 구성하고 메타 언어와 파싱 정보(Parsing Information)를 이용하여 문서 생성기, 보고서 생성기, 모듈 분석기, 코드 번역기의 설계 및 구현을 한다.

2.4 세대 언어에 대한 고찰

과거보다 막대한 양과 서로 다른 형태의 과정을 거치는 프로그램에 있어서 좀더 쉽고 빨리 프로그램이 만들어져야 하기 때문에 컴퓨터 언어에 있어서의 혁명이 필요로 되고 있다. 일반적으로 응용 소프트웨어의 구성을 보면 전체의 시스템 중, 입출력이 차지하는 부분이 70−80%이며, 중요한 처리 내용은 파일 처리가 대부분이다. 유틸리티와 간이 언어로 기술할 수 있는 경우가 많고, 데이터의 규격화도 비교적 간단하게 할 수 있다. 그 때문에 시스템 전체의 규모는 상당히 작게 되고, 유지 변경에 유연한 시스템으로 할 수가 있다. 이에 따라 정보시스템 부문의 생산성이 향상되고, 이것들을 실현시키기 위하여 화일을 정규화하고 관계형 데이터베이스와 제4세대 언어 등의 기술에 대한 대응이 중요하다는 것을 알 수 있다.

4세대 언어의 목적은 원하는 형태의 정보를 가능한 한 쉽고, 빠르게 컴퓨터로부터 얻는 것이다. 일반적으로 4세대 언어에서 사용하는 도구들의 비절차적 구성 요소는 (그림 1)과 같다.



(그림 1) 4세대 언어의 일반적인 구성
(Fig. 1) Components of a 4GL

가능한 한 쉽게 프로그램을 구성하기 위해서 4세대 언어에서는 (그림 1)과 같이 다음 형태의 도구들을 기본으로 갖고 있다.

1) 데이터베이스 정의 (Database Definition)
2) 데이터베이스 동작 (Database Operation)
3) 절차 기술 (Procedure Descriptions)
4) 스크린 디자인
5) 다이얼로그 디자인
6) 보고서 생성기
7) 규칙 도구(Rules Tools)

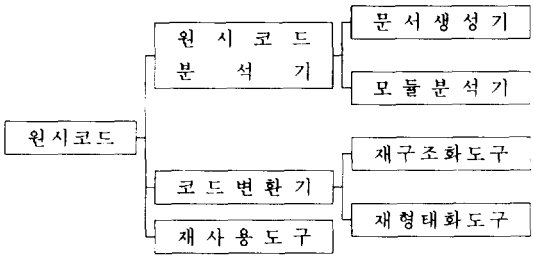
3. 역공학도구

3.1 역공학도구의 필요성과 구조

소프트웨어 생명 주기의 마지막 단계에서 얻어지는 프로그램이나 문서 등을 이용하여 생명 주기 초기 단계의 생성물에 해당하는 정보나 문서들을 생성하는 과정은 역공학(Reverse Engineering)으로서 순공학(Forward Engineering)과 구분된다. 순공학은 고수준 추상화와 논리적, 물리적 설계로부터 시스템을 개발하는 전통적인 과정이고, 역공학은 시스템의 구성 요소들과 구성 요소들 간의 상호 관계를 식별하여 좀더 높은 추상화 수준으로 시스템을 표현하도록 시스템을 분석하는 과정이다[7]. 상위 수준에서 유지 보수하는 이유는 보다 더 함축된 표현으로 알고리즘의 방법이 좀 더 응용 영역과 가깝게 연결되기 때문이다. 역공학은 프로그램을 수정하거나 이해하는 데 도움을 줄 수 있는 프로그램 요구 사항이나 설계 명세서의 복구 혹은 식별을 포함한다[9]. 이러한 역공학의 목적은 품질의 관점(quality issues), 운영의 관점(management issues), 기술적인 관점(technical issues)에 목표를 둔다. 이것은 복잡한 소프트웨어를 단순화시키거나 에러를 포함하고 있는 소프트웨어의 품질을 개선하고, 소프트웨어의 부작용(side effect)을 제거하고, 프로그래밍 표준을 시행하고 보다 더 나은 소프트웨어 유지 보수 기술을 이용하면서, 소프트웨어에서 중요한 변경이 실행되도록 허용한다. 일반적으로 프로그램 원시 코드로부터 역공학을 이용한 예를 보면 그 필요성은 다음과 같다.

1) 판독성 향상.
2) 여러 가지 존재하는 시스템의 종합.
3) 시스템의 유지 보수.
4) 현존하는 시스템의 재설계

역공학과 관련된 연구는 미시간 대학의 PRISE(Program for Research in Information System Engineering) 계획과 프로그램에서 조사하려는 프로그램의 특정 변수나 특정 시험 경로를 가진 실행문을 모아 놓은 프로그램 구성 요소들의 조각화(slicing) 개념 및 프로그램의 구문 분석에 의하여 프로그램 구성 요소들의 관계를 쉽게 파악할 수 있도록 상호 참조표(cross reference list), 호출 그래프(call graph)들을 보여줌으로서 프로그램의 이해를 도우려는 노력이 있어 왔다[18]. 다음 그림은 일반적으로 사용되는 역공학 도구의 형태들이다.



(그림 2) 역공학도구의 일반적인 형태
(Fig. 2) Reverse Engineering tools

원시 코드 분석기는 문서 생성기와 모듈 분석기로 구분한다. 현존하는 소프트웨어 시스템의 유지 보수에 연관된 주요한 문제 중의 하나는 문서의 부족이다. 문서화 도구는 프로그램 명세및 설계등 유지 보수를 돕는 것을 목표로 하며, 주로 역공학 기술들을 사용한다. 모듈 분석기는 프로그래머들이 실제로 프로그램 코드를 바꾸기 전에, 프로그래머들이 코드를 바꾸기 위해 많은 시간을 소비한다는 것에 바탕을 두고 있다. 즉 원시 코드를 분석하여 모듈도를 생성하여 프로그램에 대한 이해를 돕는다.

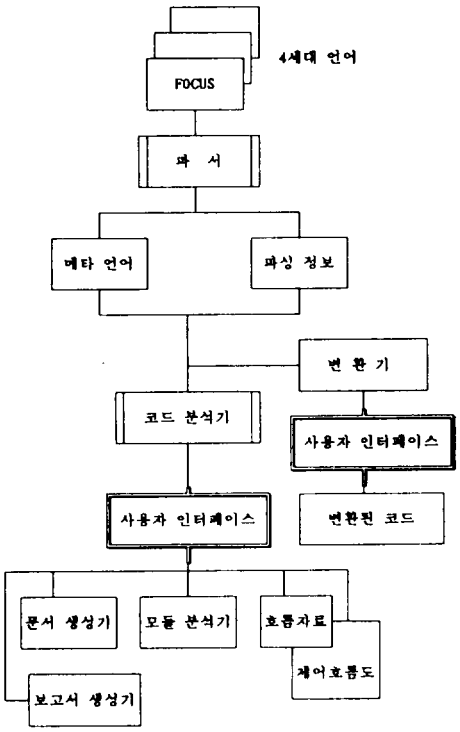
코드 변환기(Code Translator)는 한 언어로부터 다른 언어로 바꿔 주는 도구로써 재구조화 도구와 리포맷터로 나누어진다. 재구조화 도구는 코드 재구조 도구의 사용에 바탕을 둔 이론으로 유럽의 프로그램 60%가 비구조적 형태이며, 이것은 재구조화 도구가 필수적이라 할 수 있다. 리포맷터는 코드를 향상시키는 것이 아니라, 프

로그램의 읽기를 쉽게 하고 이해를 도모하기 위해 인덴테이션을 사용함으로써, 코드 레이아웃을 향상시킨다.

3.2 4세대 언어에서의 역공학 형태

본 논문에서는 일반적으로 가장 많이 사용하고 있는 FOCUS와 DBASEⅢ의 프로그램을 파서(PARSER)를 통하여 변형시킨 후, 이들을 수용할 수 있는 메타 언어를 정의하고, 메타언어와 파싱 정보를 통하여 역공학 도구를 생성하도록 한다. 또한, 다른 4세대 언어에 대해서도 변환 가능하도록 메타 언어를 설계한다. 제안된 각각의 역공학 도구는 (그림 3)에서와 같이 문서 생성기, 보고서 생성기, 모듈 분석기, 코드 번역기들이다.

전체 시스템 구성도에서 4세대 언어(FOCUS, DBASEⅢ, FOXPRO,…)를 파서로 분석하여 메타언어와 그에 따른 파싱 정보를 생성한다. 이 메타 언어와 정보를 이용하여 코드 분석기에 의해서 문서, 리포트, 모듈도를 생성하고,번역기는 메타 언어의 코드와 그에 따른 정보를 이용하여 새로운 4세대 언어를 구성하다.



(그림 3) 전체 시스템의 구성
(Fig. 3) The structure of overall system

4. 메타 언어의 구성

4세대 언어는 빠른 시간에 사용자의 욕구를 충족시킬 수 있는 응용 프로그램 작성에 매우 유리하다. 그래서 이러한 4세대 언어로 작성된 프로그램을 가지고 역공학을 하는 작업은 시스템 설계에 검증 및 효율적 유지 보수를 위해서 필수적이라 할 수 있다. 그러나 폭주하는 4세대 언어의 개발과 수많은 도구들을 사례별로 역공학 방법으로 적용하는 것은 의미 없는 작업일 것이다. 그래서 이러한 4세대 언어들을 충분히 다시 표현할 수 있는 공통의 언어를 설계하여 역공학 시스템의 내부 표현으로 사용함으로써 수많은 4세대 언어와 도구들에 대하여 능동적인 적응성을 갖도록 해야 한다.

4.1 메타언어의 설계

메타언어는 4세대 언어로 작성된 프로그램을 충분히 만족시킬 수 있는 구문을 갖고 있어야 할 뿐만 아니라 역공학 작업에 필요한 정보를 갖고 있어야 한다. 그래서 본 논문에서 제안하는 메타언어는 기존의 3세대 언어의 구문과 데이터 베이스를 관리하는데 필요한 구문으로 구성된다. 메타언어가 가급적 프로그래머나 시스템 관리자에게 친숙할 필요성이 있으므로 3세대 언어 부분은 C-형태의 구문을 사용하고 데이터 베이스 관리 부분은 SQL-형태의 구문을 적용하였다.

스키마 정의를 위한 구문은 함수 선언이나 자료 선언문과 동등하게 취급한다. 그리고 스키마 정의에 필수적인 기초 테이블 정의(base table

(표 1) 데이터베이스 정의를 위한 메타언어 EBNF
(Table 1) Meta Language EBNF for Database definition

```
<external-def>  ::=<fun-def> | <data-def> | <schema-def>
<schema-def>    ::=create schema <schema-element>
<schema-element>::=<base-table-def> | <view-def> | <privilege-def>
<base-table-def>::=create table <base-table-name>
                   {<base-table-def>[,]}*}
<view-def>      ::=create view  <view-name> [{{<colum> [,]*}] as
                   <query-spec> [with check option]
<privilege-def> ::=grant  <privileges>  on  <table-name>  to
                   {<grantee> [,]}*
```

definition), 뷰 정의(view definition,) 프리빌리지 정의(privilege definition)를 허용하였다. 〈표 1〉은 데이터베이스 정의를 위한 메타언어의 EBNF이다.

또한 데이터베이스를 조작할 수 있는 구문은 C-형태 구문의 명령문과 동등하게 취급하였다. 데이터베이스의 기본 조작 명령인 open, close, insert, delete, update, select, fetch, commit, rollback을 위한 명령문을 허용하였고, 이러한 데이터베이스 조작 명령문들이 다른 명령어와 쉽게 구별될 수 있도록 각 키워드 앞에 'db'라는 키워드를 접속시켜 사용하도록 하였다. 이것은 메타언어 CSQL이 변환 중의 내부 형태로만 사용되는 것이 아니라 메타언어 자체로도 충분히 우수한 정보를 갖게 하기 위해서이다. 〈표 2〉는 데이터베이스 조작을 위한 EBNF를 나타내고 있다.

〈표 2〉 데이터베이스 조작을 위한 EBNF
〈Table 2〉 EBNF for Database manipulate

```
<statement> ::=
                .
                .
            <db-state>:
                .
                .
<db-state>          ::=<close-statement>
                       <commit-statement>
                       <delete-state-pos>
                       <delete-state-sear>
                       <fetch-statement>
                       <insert-statement>
                       <open-statement>
                       <rollback-statement>
                       <select-statement>
                       <update-state-pos>
                       <update-state-sear>
<close-statement>   ::=dbclose <cursor>
<commit-statement>  ::=dbcommit work
<delete-state-pos>  ::=dbdelete from <table-name> where
                       current of <cursor>
<delete-state-sear> ::=dbdelete from <table-name>
                       [where <search-cond>]
<fetch-statement>   ::=dbfetch <cursor> into {<parameter>[,]}*
<insert-statement>  ::=dbinsert into <table-name>
                       [{{<column>[,]}*}] {values
                       {{<insert-atom>[,]}*} <query-spec>}
<open-statement>    ::=dbopen <cursor>
<rollback-statement>::=dbrollback work
<select-statement>  ::= dbselect [all | distinct]<selection>
                       into{<parameter>[,]}* <table-exp>
<update-state-pos>  ::=dbupdate <table-name> set {<assign>[,]}*
                       where current of <cursor>
<update-state-sear> ::= dbupdate <table-name> set {<assign>[,]}*
                       [where <search-cond>]
```

마지막으로 질의(query)에 대한 표현은 기존의 표현식에 첨가하여 활용하도록 했다. 그래서 〈expression〉은 c-형태 구문의 〈general-exp〉과 데이터베이스 질의 표현식인 〈query-exp〉의 통합 형태이다. 〈query-exp〉은 스칼라 연산과 테이블 연산을 모두 갖추고 있다. 스칼라 연산은 기존의 산술 연산과 MAX, MIN, SUM등의 데이터베이스 시스템이 갖고 있는 모든 스칼라 연산이 가능하고 테이블 연산은 from, where, group-by, having등의 clause로 구성되어 테이블 연산을 가능토록 한다. 〈표 3〉이 질의에 대한 EBNF이다.

〈표 3〉 질의에 대한 EBNF
〈Table 3〉 EBNF for Query

```
<expression>      ::=<general-exp> | <query-exp>
<query-exp>       ::=<query-term>
                     <query-exp> union [all] <query-term>
<query-term>      ::=<query-spec> | {<query-exp>}
<selection>       ::={<scalar-exp>[,]}* | *
<table-exp>       ::=<from-clause> [<where-clause>]
                     [<group-by-clause>] [<having-clause>]
<from-clause>     ::=from {<table> [<range-variable>][,]}*
<where-clause>    ::=where <search-condition>
<group-by-clause>::=group by {<column-ref> [,]}*
<having-clause>   ::=having <search-cond>
```

메타언어 CSQL은 기존의 프로그래머나 단말 사용자들에게 낯설지 않고 쉽게 접근할 수 있도록 가장 광범위하게 사용되는 언어를 기반으로 정의한 언어다. 그러나 이 결합 형태의 언어인 CSQL은 4세대 언어를 포함한 오늘날 사용되는 모든 시스템들에서 처리되는 원시 코드를 충분히 대치할 수 있다.

4.2 메타 언어의 변환 규칙

4세대 언어로 작성된 프로그램을 메타 언어로 변환하는 작업은 4세대 언어의 함축적인 의미를 풀어 헤지는 것과 거의 같다. 그래서 4세대 언어의 함축적인 의미들이 각각 장문의 메타 언어로 변환되어야 한다. 이들 변환의 모든 사례와 여러 가지의 4세대 언어를 다 소개할 수는 없기 때문에 4세대 언어가 갖고 있는 문법적 특성에서 앞 절에서 정의된 메타 언어로의 변환만을 설명한다.

첫 번째로 데이터베이스를 정의하는 부분들에 대한 변환이다. 4세대 언어에서는 데이터베이스와 데이터 모델의 일부분을 서술할 수 있어야 하기 때문에 project, select, sort, join과 같은 문장이 가능할 것이고 이러한 문장이 메타 언어로 변환되어야 한다. T1은 4세대 언어 1의 문장들이 메타 언어 구문으로 변환됨을 의미한다.

{Database Definition} $\xrightarrow{T_1}$ {⟨db-state⟩}

물론 4.1에서 보았듯이 ⟨db-state⟩는 4세대 언어의 각각의 관계 연산에 해당되는 구문들로 변환 가능하다. 예를 들어, 4세대 언어에서 "select"에 대한 문장이 있었다면 메타 언어 ⟨select-statement⟩로 변환이 되어 다음과 같은 형식의 문장이 될 것이다.

dbselect all * into sum, total from s-table

두 번째로 데이터베이스의 질의 표현이나 갱신에 대한 구문 변환이 필요하다.

{Database Operation} $\xrightarrow{T_2}$ {⟨query-exp⟩}

⟨query-exp⟩는 데이터베이스의 간단한 질의 표현 식이나 자료의 수정과 같은 연산을 표현해 주는 구문이다. 그러므로 4세대 언어에서 제의된 모든 연산이 대치 가능하다.

세 번째는 스크린 디자인 및 Dialogue 디자인이다. Dialogue 디자인은 스크린 디자인의 확장 형태이므로 사실상 스크린을 디자인하는데 필요한 모든 구문만 변환하면 된다.

screen design $\xrightarrow{T_3}$ {⟨write&position-state⟩}

4.1절에서는 생략되어 있지만 위치 이동과 형식화된 출력 문장의 조합으로 조합으로 스크린 디자인 문장은 대치 가능하다.

마지막은 리포트 제너레이션 문장의 변환이다. 스크린 및 다이얼로그 디자인은 화면상에서 출력 위치를 변화해 가면서 자유롭게 출력 가능하지만 리포트 제너레이션은 한 방향 출력만 가능하므로 버퍼에 데이터 집합을 옮긴 후 출력하는 작업이 반복된다.

{Report Generation} $\xrightarrow{T_4}$ {⟨buffering&write-state⟩}

이외에도 일반적인 절차적 기술에 대한 문장들이 있는데 이러한 문장들은 메타 언어 각각의 구문들로 충분히 표현 가능하다. 왜냐하면 메타 언어의 나머지 문장들은 3GL의 절차적 구문을 그대로 수용하기 때문이다.

{another procedural statements} $\xrightarrow{T_5}$ {⟨other s statements⟩}

지금까지의 모든 변환 규칙들은 현재 상용화된 대부분의 4세대 언어를 충분히 변환시킬 수 있다. 그리고 변환된 메타 언어 프로그램은 역공학 도구를 구성하는데 충분한 정보를 유지하고 있으며 내부적 표현뿐만이 아니라 독립된 하나의 언어로서 의미 전달이 가능하다.

4.3 메타 언어의 변환 예

4세대 언어들은 각 4세대 언어 파서에 의하여 공통의 메타 언어로의 변환되는 규칙을 4.2에서 서술하였으며, 여기서는 변환 규칙을 이용하여 4세대 언어들이 메타 언어로 변환되는 과정을 설명하기로 하겠다. dBASE III PLUS 프로그램을 메타 언어로 변환되는 과정을 보이기 위해 다음과 같이 사원-부서 데이터베이스를 정의한다. 사원 테이블은 사번, 성명, 직책, 입사일자, 봉급, 부서 번호의 필드로 구성되며, 부서 테이블은 부서번호, 부서명, 총인원, 부서위치의 필드로 구성된다. 두 테이블의 관계성은 부서 번호로 연결이 된다.

⟨표 4⟩ 사원, 부서 데이터베이스
⟨Table 4⟩ EMP-DEPT Database

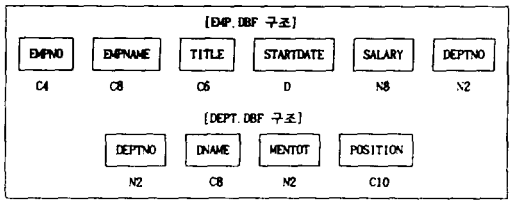
(a) 사원 테이블
(a) EMP Table

사번	성 명	직책	입사 일자	봉 급	부서번호
7369	홍길동	사원	1980/12/17	800000	20
7499	이일삼	대리	1981/09/20	1600000	30
7521	김이사	대리	1981/06/22	1250000	40
.	.	.	.	.	.
.	.	.	.	.	.

(b) 부서 테이블
(b) DEPT Table

부서번호	부서명	총인원	부서위치
10	영업부	3	김천
20	연구소	4	대전
.	.	.	.

⟨표 4⟩의 사원,부서 데이터베이스를 dBASE III PLUS 화일 구조도로 표현하면 다음과 같이 구성된다. 사원과 부서 테이블은 각각 EMP와 DEPT 화일로 구성되며, 각 필드들은 필드 명과 데이터형이 다음과 같이 구성된다.



(그림 4) EMP-DETP 화일 구조도
(Fig. 4) The structure of EMP-DEPT File

EMP.DBF화일과 DEPT.DBF화일을 부서명이 연구소인 사원의 사번, 성명, 직책, 입사일자, 봉급을 구하고자 하는 프로그램을 작성하면 다음과 같다.

```
. SELECT 1
. USE EMP
. SELECT 2
. USE DEPT
. JOIN WITH EMP TO EMPDEPT FOR  DEPTNO=EMP->DEPTNO
. USE EMPDEPT
. LIST FOR DNAME='연구소'  FIELDS EMPNO, ENAME, TITLE, STARTDATE, SALARY
```

(그림 5) dBASE III PLUS 프로그램 예
(Fig. 5) Example of dBASE III PLUS

(그림 5)는 EMP.DBF와 DEPT.DBF 데이터 베이스 화일을 죠인하여 새로운 화일을 임시로 작성하여 원하는 결과를 구하는 dBASEIII PLUS 프로그램이다. 이 프로그램을 4.2에 정의된 메타 언어 변환 규칙에 의하여 단계별로 변환하면, EMP와 DEPT 테이블을 죠인하여 새로운 임시 화일은 죠인뷰를 생성하여 처리되며, 이 뷰를 이용하여 원하는 결과를 검색할 수 있다. 이를 CSQL문으로 표현하면 (그림 6)이 생성된다.

```
CREATE VIEW EMPDEPT
AS SELECT *
   FROM   EMP, DEPT
   WHERE  EMP.DEPTNO = DEPT.DEPTNO;

SELECT  EMPNO, ENAME, TITLE, STARTDATE, SALARY
FROM EMPDEPT
WHERE DNAME='연구소'
DROP VIEW EMPDEPT
```

(그림 6) CSQL 언어
(Fig. 6) CSQL Language

또한, (그림 5)의 프로그램을 FOCUS 프로그램으로 작성하면 다음과 같이 생성된다.

```
JOIN DEPTNO IN EMP TO DEPTNO IN DEPT AS EMPDEPT
TABLE FILE EMP
PRINT EMPNO ENAME TITLE STARTDATE SALARY
WHERE DNAME EQ '연구소';
END
```

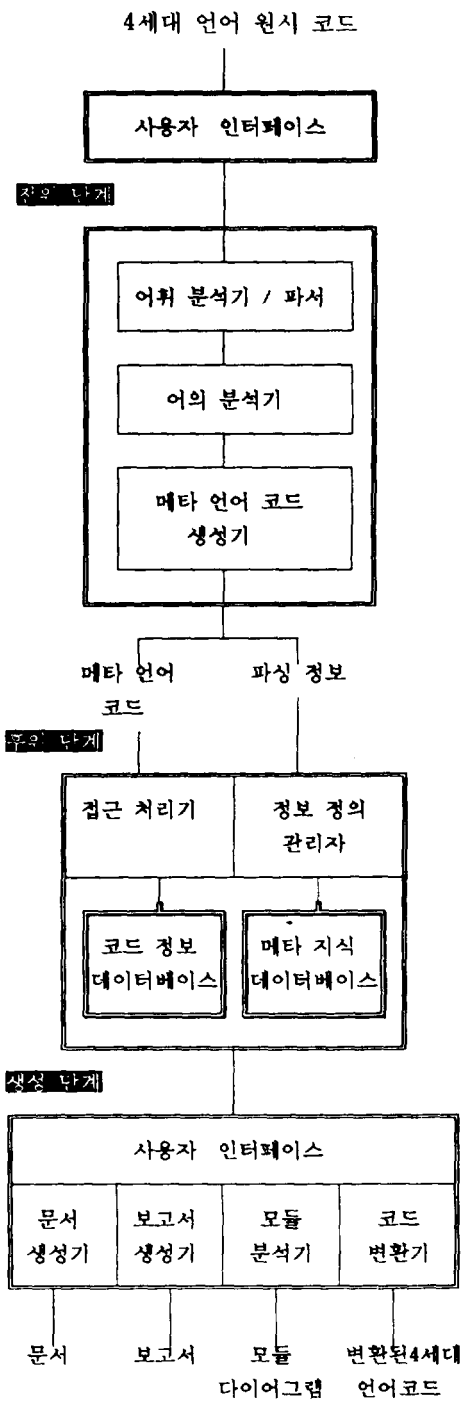
(그림 7) FOCUS 프로그램의 예
(Fig. 7) Example of FOCUS Program

(그림 7)의 프로그램 역시 메타 언어 변환 규칙에 의하여 (그림 6)과 같은 CSQL문을 생성하게 된다. 각 4세대 언어의 특성에 따라 동일한 결과를 원하는 프로그램을 작성하더라도 CSQL은 서로 상이할 수도 있다. 그러나 메타 언어는 각 4세대 언어의 특성들을 포함하고 있어 역공학도구의 설계에는 아무런 영향을 미치지 않는다.

5. 역공학도구의 설계 및 구현

4세대 언어로 작성된 원시 코드로부터 메타 언어 형태로 변환하고 이를 이용한 역공학도구의 논리적인 형태를 3장의 시스템 구성도에서 제안하였다. 제안된 시스템을 도구화하기 위하여 물리적인 관점에서의 전반적인 시스템 구성을 (그림 8)에 나타내었다.

4세대 언어로부터의 역공학 도구를 산출해내기위한 시스템의 구조는 전위 단계(Front-end), 후위 단계(Back-end), 생성 단계(Generation-set)로 구성하였다. 전위 단계 시스템은 일종의 번역기 역할을 하는 프로그램으로서 기본적으로 어휘 분석기(lexical analyzer)와 파서(parser), 어의 분석기(semantic analyzer), 메타언어 코드 생성기(Metalanguage code generator) 의 세 부분으로 구성된다. 이 그림에서 운용되는 프로그램은 필터와 유사한 형태를 갖는 전위 단계를 두어 원시 코드를 입력받는다. 즉, 전위 단계는 4세대 언어로 작성된 프로그램을 입력으로 받아들여 4.2 절에서 기술된 변환 관계에 따라 본 연구에 의하여 정의된 메타 정의에 합당한 데이터들을 추출하는 것이다. 후위 단계는 전위 단계에서 생성된 메타언어 문장과 파싱정보를 데이터베이스에 미리 정의된 논리적 데이터 베이스 구조 형태로 저장한다. 메타 언어 문장과 파싱 정보들은 접근처리기(Access Processor)에 의하여 데이터 베이스에 저장되는데, 이들 데이터가 저장되는 논리적 데이터 베이스 구조는 정보 정의 관리자 언어(Information Definition Manager Language)에 의하여 기술된다. 이렇게 기술된 데이터 베이스를 메타 지식 데이터베이스(Meta knowledge DB)로 구축한다.



(그림 8) 전반적인 시스템 구조
(Fig. 8) System Structure

이러한 후위 단계 과정이 끝나면 사용자 인터페이스 환경 하에서 사용 가능한 소프트웨어 도구들인 문서 생성기, 보고서 생성기, 모듈 분석기, 코드 번역기들에 의하여 필요한 역공학도구의 산출물들이 추출된다.

5.1 문서 생성기

소프트웨어의 보수 유지를 위해서는 시스템에 대한 문서들이 각종 보수 유지 작업이 기초 자료로 사용된다. 이러한 문서의 생성 및 관리 업무를 수작업에 기반을 두어 수행하게 되면 실제 시스템의 구조 변경으로 인한 문서와 시스템간의 일치성을 보장받기가 어렵다. 이에 필요한 것이 자동문서 추출 시스템(automatic documents extracting system)인데, 이러한 시스템은 원시 프로그램으로부터 문서화에 필요한 데이터를 직접 추출하여 그 정보를 데이터베이스에 저장하여 필요한 문서들을 생성하게 하여 주므로 수작업에 대한 문서화 과정의 부하를 극소화시켜 주므로 시스템에 대한 문서화 작업을 효율적으로 수행할 수 있게 한다. 전위 단계에 의하여 수행되어야 하는 작업은 메타 정의에 의하여 구축된 논리적 데이터베이스 구조에 적합한 개별적인 데이터들을 4세대 언어로부터 추출하는 것이다.

따라서 이러한 추출 작업을 위한 4세대 언어로 작성된 프로그램과 목적 언어 문장 사이의 변환 관계에 대하여 기술하도록 한다. 그런데 이러한 변환 관계를 형식적(formal)으로 서술하는 것은 매우 어렵기 때문에 입력 형식은 전위 단계와 후위 단계를 거쳐 생성되는 C-형태의 메타 언어로 사용하였다.

(1) Return-relationship
INPUT : ⟨type⟩⟨fun_id⟩(⟨para-list⟩);
OUTPUT : 함수 ⟨fun_id⟩는 함수 ⟨type⟩의 연산 결과를 돌려준다.

(2) Call-relationship
INPUT : ⟨fun_id⟩(⟨para-list⟩);
OUTPUT : 함수 ⟨fun_id⟩는 함수 ⟨caller⟩에 의해 호출됨
여기에서 ⟨caller⟩는 실제로 함수 ⟨fun_id⟩를 호출하는 함수를 말한다.

(3) Have-id-relationship;
INPUT : 〈storage class〉 int 〈name〉;
OUTPUT : 정수형 변수 〈name〉은 〈storage class〉범주에 속하는 변수

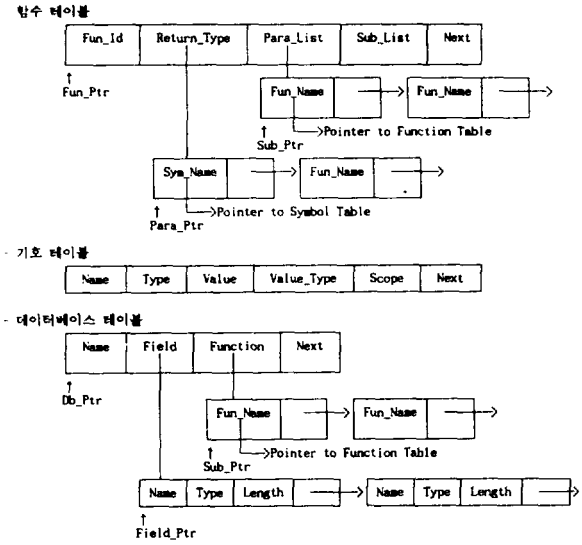
(4) Subpart-relationship
구조적 데이터를 구성하는 데이터 형들에 대해서는 4세대 언어의 다양한 변수 선언방식을 전부 다룰 수 없었기 때문에 SUB-TYPE 속성을 이용하여 전반적인 정보를 제공하였다.
INPUT : 〈storage class〉 STRUCT 〈identifier〉{ 〈struct-dec〉 }
OUTPUT : 구조체 〈identifier〉는 〈struct dec〉들의 부분 타입으로 이루어진 〈storage class〉범주의 변수이다.

(5) Value-para-relationship
INPUT : 〈declarator〉(〈type-spec〉〈identifier〉);
OUTPUT : 〈type spec〉형의 변수 〈identifier〉는 함수 〈declarator〉에서 값 전달 인수이다.

(6) Address-para-relationship
INPUT : 〈declarator〉(〈type-spec〉 {&} 〈identifier〉);
OUTPUT : 〈type spec〉형의 변수 〈identifier〉는 함수 〈declarator〉에서 주소 전달 인수이다.

이와 같은 규칙에 대한 개괄적인 알고리즘은 다음과 같다.

```
[알고리즘1] Document Generator
Input  : Meta-language program
Output : Documentation contents
Method :
Document_Gen()
{
open(meta-language program);
while ((stmt ← Next_Record()) ≠ EOF)
{
   stmt_type ← Analysis_Stmt(stmt); /* 입력문장 분석 후 유형별로 정수값(1~7) 반환 */

   switch(stmt_type)
   {  case 1 :     /* <type> <function_id>(<type> <identifier>); */
         Function_Relation(stmt, fun_type, fun_id, id_type, id);
         print("함수 ", fun_id, " 는 ", fun_type, "형태의 연산결과를 돌려준다");
         print(id_type, "형 변수" id, "는 함수", fun_id, "에 대한 값전달 인수이다");

      case 2 :     /* <type> <function_id>(<type> &<identifier>); */
         Function_Relation(stmt, fun_type, fun_id, id_type, id);
         print("함수 ", fun_id, " 는 ", fun_type, "형태의 연산 결과를 돌려준다");
         print(id_type, "형 변수" id, "는 함수", fun_id, "에 대한 값전달 인수이다");

      case 3 :     /* <function_id> (<parameter_list>); */
         Call_Relation(stmt, callee_id);
         print("함수 ", callee_id, " 는 함수 ", fun_id, "에 의해 호출된다");

      case 4 :     /* <storage_class> int <identifier>; */
         Have_Id_Relation(stmt, st_class, id);
         print("정수형 변수", id, "는", st_class, "범주에 속하는 변수이다");

      case 5 :     /* <storage_class> STRUCT <tag> { <struct_dec> }; */
         Subpart_Relation(stmt, st_class, tag, struct_dec);
         print("구조체 ", tag, " 는 ", struct_dec, "들의 부분형으로 구성된 ",
               st_class, " 범주의 변수이다");
   }
}
}
```



(그림 9) 파싱 정보의 자료 구조
(Fig. 9) Data structure of Parsing Information

5.2 보고서 생성기

보고서 생성기는 5장의 (그림 8)과 같은 과정에 의해 4세대 언어의 원시 프로그램을 메타언어로 변환할 때 생성하는 파싱 정보를 이용하여 프로그램 상에서 사용된 전역 및 지역 변수 정보, 함수 정보, 데이터베이스 화일에 대한 정보, 입출력 관련 정보 등을 추출해 내는 생성기이다. 보고서 생성기가 필요한 정보를 생성하기 위하여 사용하는 파싱 정보는 아래의 (그림 9)와 같이 함수와 관련된 정보를 관리하는 함수 테이블(Function Table), 심볼과 관련된 정보를 관리하는 기호 테이블(Symbol Table) 및 데이터베이스 관련 정보를 유지하는 데이터베이스 테이블(Database Table)로 구성된다. 이러한 파싱 정보로부터 필요한 정보를 생성하는 보고서 생성기를 알고리즘으로 표현하면 [알고리즘2]와 같다.

[알고리즘2] Report Generator
Input : Meta-language statements, Parsing information(Symbol, Function, Database table)
Output : Variable list, Function list, Database list, Input./Output value
Method :

```
struct Function_Table_Tag {              /* 함수 테이블의 구성 */
    char    *Fun_Id;                          /* 함수명                */
    char    *Return_Type;                     /* 함수의 반환값 유형    */
    struct Para_List_Tag    *Para_List;       /* 인자 리스트           */
    struct Fun_List_Tag     *Sub_List;        /* 부프로그램 리스트     */
    struct Function_Table_Tag    *Next; };

struct Symbol_Table_Tag {                /* 기호 테이블의 구성 */
    char    *Name;                            /* 명칭                  */
    char    *Type;                            /* 자료형                */
    int     Value;                            /* 자료값                */
    char    Value_Type;                       /* 자료값 종류(값/주소)  */
    char    *Scope;                           /* 선언한 함수명         */
    struct Symbol_Table_Tag    *Next;   }

struct Db_Table_Tag {                    /* 데이터베이스 테이블의 구성 */
    char *Name;                               /* 데이터베이스 화일 명칭      */
    struct Field_List_Tag    *Field;          /* 데이터베이스의 필드리스트 */
    struct Fun_List_Tag      *Function;       /* 사용한 함수명 리스트       */
    struct Db_Table_Tag    *Next;   }

struct Fun_List_Tag {                    /* 부프로그램 리스트를 위한 자료구조 */
    struct Function_Table_Tag    *Fun_Name;   /* 함수 테이블에 대한 포인터 */
    struct Fun_List_Tag          *Next;   };

struct Para_List_Tag {                   /* 인자 리스트를 위한 자료구조 */
    struct Symbol_Table_Tag    *Sym_Name;     /* 기호 테이블에 대한 포인터 */
    struct Para_List_Tag        *Next;   };
```

```
struct Field_List_Tag {                    /* Database의 필드리스트를 위한 자료구조 */
    char   *Name;                                  /* 필드명칭          */
    char   *Type;                                  /* 필드자료형        */
    int     Length;                                /* 필드자료형의 크기 */
    struct Field_List_Tag   *Next;  }

struct Symbol_Table_Tag   Symbol_Table, *Sym_Ptr;
struct Function_Table_Tag   Function_Table, *Fun_Ptr;
struct Db_Table_Tag   Db_Table, *Db_Ptr;
struct Fun_List_Tag   *Sub_Ptr;
struct Para_List_Tag   *Para_Ptr;
struct Field_List_Tag   *Field_Ptr;

Report_Gen()
{
    /* program에서 사용된 전역 변수 명칭 출력 */
    print("----- Global variable -----");
    Sym_Ptr ← Symbol_Table;
    while(Sym_Ptr ≠ NULL)
    {
        if (Sym_Ptr->Scope = "global")   then
            print("   Symbol name : ", Sym_Ptr->Name);
        Sym_ptr ← Sym_Ptr->Next;
    }
    /* program에서 사용된 지역 변수 정보(변수명 및 선언 함수명) 출력 */
    print("----- Local variable -----");
    Sym_ptr ← Symbol_Table;
    while(Sym_ptr ≠ NULL)
    {
        if (Sym_Ptr->Scope ≠ "global")   then
            print("   Symbol name : ", Sym_ptr->Name, "Scope : ", Sym_ptr->Scope);
        Sym_ptr ← Sym_ptr->Next;
    }

    /* program에서 사용된 함수 정보(함수명, 반환 자료형, 인자, 인자 자료형) 출력 */
    Fun_ptr ← Function_Table;
    while(Fun_ptr ≠ NULL)
    {                          /* 함수명, 반환 자료형 출력 */
        print("Function Name : ", Fun_Ptr->Fun_Id, "return type :",
              Fun_Ptr->Return_Type);
        Para_Ptr ← Fun_Ptr->Para_List;
        while(Para_Ptr ≠ NULL)
        {                      /* 인자 리스트 및 자료형 출력 */
            print("   parameter variable : ", (Para_Ptr->Sym_name)->Name);
            print("              type    : ", (Para_Ptr->Sym_name)->Type);
            Para_Ptr ← Para_Ptr->Next;
        }
        Fun_Ptr ← Fun_Ptr->Next;
    }

    /* program에서 사용된 데이터베이스정보                                    */
    /*       (데이터베이스명, 필드명, 필드형, 필드크기, 사용한 함수명) 출력 */
    Db_Ptr ← Db_Table;
    while(Db_Ptr ≠ NULL)
    {
        print("Database file : ", Db_Ptr->Name);   /* 데이터베이스명칭 */
```

```
            Field_Ptr ← Db_Ptr->Field;
            while(Field_Ptr ≠ NULL)
            {
                print("   Field name  : ", Field_Ptr->Name);     /* 필드명  */
                print("        type   : ", Field_Ptr->Type);     /* 필드형  */
                print("        length : ", Field_Ptr->Length);   /* 필드크기 */
                Field_Ptr ← Field_Ptr->Next;
            }

            print("   Openning Function : ");
            Sub_Ptr ← Db_Ptr->Function;
            while(Sub_Ptr ≠ NULL)
            {                                  /* 해당 database를 사용한 함수명 */
                print("      Function name : ", (Sub_Ptr->FunName)->Fun_Id);
                Sub_Ptr ← Sub_Ptr->Next;
            }
            Db_Ptr ← Db_Ptr->Next;
        }

        /* program상의 입.출력 정보 출력 */
        open(meta-language program);
        while((stmt ← Next_Record()) ≠ EOF)
        {
        stmt_type ← Analysis_Stmt(stmt);   /* 입력문장 분석 후 유형별로 정수값 반환 */
                                           /*  1 or 2 : 함수 정의 문장    */
                                           /*  6 : 입력문,  7 : 출력문    */
        switch(stmt_type)
        {
            case  1 or 2 :  /* I/O문을 사용한 함수명 출력 */
                Function_Relation(stmt, fun_type, fun_id, id_type, id);
                print("I/O information of function : ", fun_id);
            case  6 :      /* 입력 문장 분석 결과 출력 */
                IO_Relation(stmt, io_value_list);   /* IO_Relation() : I/O문 분석 후 */
                                                    /* 변수 리스트(io_value_list) 반환 */
                while(io_value_list ≠ NULL)
                {     print("   Input variable : ", io_value_list->name);
                      io_value_list ← io_value_list->next);
                };
            case  7 :      /* 출력 문장 분석 결과 출력 */
                IO_Relation(stmt, io_value_list);
                while(io_value_list ≠ NULL)
                {     print("   Input variable : ", io_value_list->name);
                      io_value_list ← io_value_list->next);
                };
        }
    }
}
```

5.3 모듈 분석기

모듈 분석기는 프로그램 상에서 사용된 함수들 간의 호출/피호출 관련 정보를 추출하는 생성기이다. 모듈 분석기에 의해 생성되는 함수들 사이의 상호 참조 정보는 유지 보수자가 프로그램을 분석하는데 있어서 상당히 유용한 도구로 활용될 것으로 기대된다. 함수들 사이의 상호 참조 정보를 추출하기 위하여 모듈 분석기는 [알고리즘3]과 같이 5.2절에서 기술한 함수 테이블과 기호 테이블 정보를 활용한다.

```
[알고리즘3] Module Generator
Input  : Parsing information(함수 테이블, 기호 테이블)
Output : Cross reference information(Calling function : Called function)
Method :
struct Function_Table_Tag {              /* 함수 테이블의 구성 */
    char   *Fun_Id;                          /* 함수명                */
    char   *Return_Type;                     /* 함수의 반환값 유형    */
    struct Para_List_Tag   *Para_List;       /* 인자 리스트           */
    struct Fun_List_Tag    *Sub_List;        /* 부프로그램 리스트     */
    struct Function_Table_Tag   *Next; };

struct Symbol_Table_Tag {                /* 기호 테이블의 구성 */
    char   *Name;                            /* 명칭                  */
    char   *Type;                            /* 자료형                */
    int    Value;                            /* 자료값                */
    char   Value_Type;                       /* 자료값 종류(값/주소)  */
    char   *Scope;                           /* 선언한 함수명         */
    struct Symbol_Table_Tag   *Next;  }

struct Fun_List_Tag {                    /* 부프로그램 리스트를 위한 자료구조 */
    struct Function_Table_Tag   *Fun_Name;   /* 함수 테이블에 대한 포인터 */
    struct Fun_List_Tag         *Next;  };

struct Para_List_Tag {                   /* 인자 리스트를 위한 자료구조 */
    struct Symbol_Table_Tag   *Sym_Name;     /* 기호 테이블에 대한 포인터 */
    struct Para_List_Tag       *Next;  };

struct Function_Table_Tag   Function_Table, *Fun_Ptr;
struct Fun_List_Tag   *Sub_Ptr;
struct Para_List_Tag   *Para_Ptr;

Module_Gen()
{    /* 상호 참조(cross reference) 정보(주프로그램명 : 부프로그램명) 출력 */
    Fun_Ptr ← Function_Table;
    while(Fun_Ptr->Fun_Id ≠ NULL)
    {
        print("Main-program : ", Fun_Ptr->Fun_Id);  /* 주프로그램명 출력 */
        Sub_Ptr ← Fun_Ptr->Sub_List;
        while(Sub_Ptr ≠ NULL)
        {              /* 주프로그램에 대한 각 부프로그램명 출력 */
            print("Sub-program : ", (Sub_Ptr->Fun_Name)->Fun_Id);
            Sub_Ptr ← Sub_Ptr->Next;
        }
        Fun_Ptr ← Fun_Ptr->Next;
    }
}
```

5.4 코드 번역기

4장에서 제안된 메타언어는 거의 모든 4세대 언어를 변환할 수 있도록 구문을 갖추고 있다.그러므로 이 메타언어로 구성된 문장들을 4세대의 다른 언어로 변환하는 것은 그리 어렵지 않다. 메타언어의 문장들이 해당 4세대 언어의 문장으로 일대일 혹은 다대일로 변환되면 된다. 이것은 변환 과정 Ti의 역변환 과정이므로 T_i^{-1} 로 표시하고 다음과 같은 변환 규칙을 얻을 수 있다.

{⟨db-state⟩} $\xrightarrow{T}$ {Database Definition}
{⟨query-exp⟩} $\xrightarrow{T}$ {Database Operation}
{⟨write&position-state⟩} $\xrightarrow{T}$ {screen de-

sign}
{⟨buffering&write-state⟩} $\xrightarrow{T_1}$ {Report Generation}
{⟨others statements⟩} $\xrightarrow{T_2}$ {another procedural statements}

그런데 어떤 원시 코드의 메타언어로의 변환 후, 다시 본래의 4세대 언어로 변환을 하면 처음의 원시 코드가 생성되지 않을 수도 있다. 즉,

$S \xrightarrow{T_1} S' \xrightarrow{T_2} S''$ (S, S', S'' : 어떤 코드들)

일 때 S'와 S''는 동일할 수도 있고 그렇지 않을 수도 있다. 그러나 이것은 전혀 중요하지 않다.

왜냐하면 어떤 문장 S의 의미(semantic)를 함수 S로 표기할 때 다음 식은 언제나 성립한다.

$$\lceil S \rfloor = \lceil S'' \rfloor$$

6. 결 론

소프트웨어 생명주기중 소프트웨어 유지 보수를 위해 사용되는 비용이 많은 부분을 차지하고, 소프트웨어 유지보수중 원시 코드를 이해하는 데 소요되는 시간이 반 이상을 차지한다. 그러므로 이 부분의 자동화가 절실히 요구되고 있다. 또한 소프트웨어 개발에 있어서의 경험 부족, 소프트웨어 자체의 복잡성 증가 및 개발 도구의 부족등 소프트 위기에 대한 대처의 방안으로 1980년대 후반부터 CASE와 4세대 언어가 등장하였다. 4세대 언어는 특히 범용 소프트웨어를 개발하거나 사용하기 위해 단말기 사용자에게도 쉽게 느껴지는 환경을 구축하고 있고, 그러한 이유로 인하여 상당한 분야에서 많이 사용하고 있는 언어로 자리잡고 있다.

본 연구에서는 4세대 언어에서의 역공학을 위한 환경을 구성하였다. 전위 단계를 통하여 4세대 언어로 작성된 원시 코드로부터 메타 언어 형태로 변환하고, 메타언어 명령어와 파싱 정보를 구성하여 이를 데이터베이스로 구축하여 프로그램 이해를 위한 문서 생성기,보고서 생성기, 모듈 분석기, 코드 번역기를 설계하고 구현함으로서 4세대 언어에서의 역공학 도구를 제안하여, 프로그램의 이해및 관리를 효율적으로 하는데 목적이 있다.

앞으로의 연구 과제는 변형된 메타언어와 파싱 정보로부터 모듈을 재구성하고 재사용 가능한 모듈을 찾아 정보 저장소에 등록과 검색을 자동화 시킴으로서 체계적이고 적극적인 프로그램 이해 및 재사용 도구로도 활용하도록 연구가 지속되어야 할 것이다.

참 고 문 헌

[1] Alan R. Simon, "The Integrated CASE Tools Handbook," VAN NOSTRAND REINHOLD, pp. 190-196, 1993.

[2] Chapin, "Software Maintenance Life Cycle," Proceedings of Conference on Software Maintenance, pp. 6-13, 1988.

[3] F.P. Brooks, "No Silver Bullet : Essencs and Accidents of Software Engineering," IEEE Computer, pp. 61-70, Feb. 1991.

[4] Helen M. Edwards and Malcom Munro, "RECAST : Reverse Engineering from COBOL to SSADM Specification," Centre for Software Maintenance, University of Durham, UK, IEEE 1993.

[5] Ian Sommerville, "Software Engineering," Addison-Wesley Publishing Company, pp. 536-539, 1989.

[6] P.A.V.Hall, "Software Reuse and Reverse Engineering in Practice," CHAPMAN &HALL, pp. 214-220, 1992.

[7] Marijana Tomic "A possible Approach to object-oriented Reengineering of Cobol programs," ACM SIGSOFT, Apr 1994.

[8] FOCUS Manual Information Builders, Inc, 1994.

[9] Hongji Yang "The Supporting Environment for A Reverse Engineering System -The Maintainer's Assistant," IEEE Conference on Software Maintenance, 1991.

[10] Scott R. Tilley "Domain-Retargetable

Reverse Engineering," IEEE Department of Computer Science, University of Victoria, 1993.

[11] ROBERT S. ARNOLD, "Software Restructuring," IEEE, Vol. 77, No. 4, pp. 607-617 April 1989.

[12] Harry M. Sneed & Gabor Jandrasics, "Software Recycling," Conf. on Software Maintenance, pp. 82-90, 1987.

[13] Richard C. Linger, "Software Maintenance as an Engineering Discipline," Conference on Software Maintenance, pp. 292-97, 1988.

[14] Eric J. Byrne, "Software Reverse Engineering: A Case Atudy," Software Practice And Experience, Vol. 21, pp. 1349-1364, December 1991.

[15] Richard C. Waters, "Program Translation via Abstraction and Reimplementation," IEEE Transactions on Software Engineering, Vol. SE-14, No. 8, pp. 1207-1228, Aug 1988.

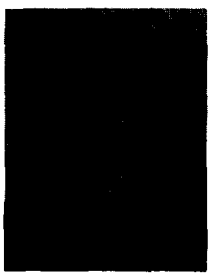
[16] 우치수, "역공학에 관한연구", 과학기술처 최종연구보고서 1987년

[17] 진영배, 김남용, 왕창종, "소프트웨어 유지보수및 재사용을 위한 역공학 도구의 재사용에 관한 연구", 한국정보과학회 가을 학술발표논문집 제21권 2호, 1994.

[18] 최은만, 이금석, 홍영식, "프로그램의 이해를 지원하는 소프트웨어 유지보수 도구 세트 개발", 정보과학회논문지 제21권 제5호, 1994. 5.

진 영 배

1980년 인하대학교 자원공학과 (학사)
1985년 인하대학교 전자계산학과 (이학석사)
1989년 인하대학교 전자계산전공 박사과정수료
1985년~93년 인천기능대학 전자과 조교수
1993년~현재 충청전문대학 사무자동화과 조교수
관심분야 : CASE, Object-Oriented Methodology, 역공학



왕 창 종

1964년 고려대학교 물리학과 (학사)
1975년 성균관대학교(경영학석사)
1981년~90년 인하대학교 전자계산소장
1992년~93년 정보과학회 부회장, 전산교육연구회 위원장
1979년~현재 인하대학교 전자계산공학과 교수
관심분야 : Software Engineering, CASE, 역공학, Expert System, Intelligent Tutoring System