論 文

論文番號 12－2－4

KAIST CHILL 컴파일러

(KAIST CHILL Compiler)

이 동 길* 최 광 무** 전 길 남***
(Donggill Lee, Kwang-Moo Choe, Kilnam Chon)

요 약

본 연구에서는 ESS 개발환경을 고려하여 production quality CHILL compiler 개발의 환경조성을 위한 CHILL subset compiler를 설계 제작하였다. KAIST Parser Generating System 등을 이용하여 front end를 설계 제작하였으며 Amsterdam Compiler Kit로 부터 back end와 intermediate machine의 환경을 제공받아 CHILL subset을 위한 compiler를 설계 제작하였다

ABSTRACT

A compiler for the subset of CHILL (CCITT High Level Language) is designed and implemented, considering the developing environment of ESS (Electronic Switching Systems). The compiler developed provides a basis for developing the production quality CHILL compiler KAIST CHILL compiler is developed using the powerful compiler-compiler tools, such as KAIST Parser Generating System and Amsterdam Compiler Kits.

1. 서 론

전자교환기를 비롯한 통신분야에 있어서 SPC (Stored program control)기법을 이용함에 따라 대규모의 복잡한 software를 개발하는데 있어서 software 위기에 대한 인식이 점증하고 있다. 이 위기에 대처하기 위해서는 적절한 고급언어(high level language)를 사용한 software 개발환경의 조성이 필요하게 되었다. 이에 따라 CCITT는 1980년 11월 CC-

• 본 논문은 한국전기통신연구소의 부분적인 재정지원을 받았음.

*정 회 원 : 한국전기통신연구소
**종신회원 : 한국과학기술원 전산학과
***평 의 원 : 한국과학기술원 전산학과
접수일자 : 1985. 4 13

ITT Recommendation Z.200[1]으로 권장한 바 있으며 현재 ITT, NTT등은 CHILL을 채택하여 ITT 1240, NTT D-10, D-70 등의 전자교환기를 개발했으며 대다수의 선진국 회사들은 CHILL programming 환경을 조성하기 위해 작업을 진행하고 있다. CHILL은 concurrent processing, reliability, separate compilation 및 system programming의 기능을 갖추고 있어서 널리 사용될 것으로 보인다. 이에 우리나라에서도 CHILL에 근거한 programming 개발환경에 관한 연구를 수행하여 CHILL에 의한 software 개발환경을 조성하는 것이 시급한 문제로 되었다.

본 연구는 실질적인 규모의 software 개발에 사용 가능한 production quality CHILL compiler 개발의 기본환경 조성을 위하여 prototype CHILL를 KAIST Parser Generating System[10], Amsterdam Compiler Kit[7]등 각종 강력한 Compiler-Compiler 도구를 이용하여 설계 제작하였다.

2. 설계 고려 사항

2.1 Requirement

ESS(Electronic Switching System) software 개발환경은 복잡한 기능과 여러가지 다양한 특성을 가지며 개발환경 자체도 software development system, hardware development system, target system으로 구성되는 복잡한 구조를 가지고 있다. 따라서 ESS software 개발을 위한 CHILL program들은 software development system에서 compile되어 그 object code가 hardware development system으로 down loading되어 수행 및 debugging되거나, 혹은 software development system에 수행(simulation)되어 debugging 될 수 있어야 한다. 이에 따라 CHILL compiler는 CHILL program을 intermediate code로 변환하여주는 front end와 intermediate code를 target code로 변환하여주는 back end의 두부분으로 크게 나누어 설계 하였다. 중간단계로서 가상의 intermediate machine을 사용하는 이유는 다음과 같다.

(1) target machine의 변경에 따른 retargetability의 증가

(2) host machine(software development system)에서의 simulation 및 debugging의 용이

(3) Compiler 개발의 module화

Amsterdam Compiler Kit[7]는 intermediate machine으로서 stack machine의 일종인 EM virtual machine architecture를 제공하여주며 또한 intermediate code로 이용할 수 있는 EM code의 환경을 제공하여 주는데 EM code를 target code로 변환하는 backend 부분은 table driven code generation을 이용하여 target machine specification으로 부터 target code로 자동적으로 생성할 수 있다.

2.2 Compiler-Compiler tools

(1) KAIST Parser Generating System [11]

KAIST Parser Generating System 은 BNF로 정의된 language의 syntax specification을 input으로 읽어서 새로운 formalism[9]에 의거하여 LALR(1) parsing 및 Abstract Syntax Tree의 구성을 위한 table

을 작성하여 줌으로써 compiler의 syntax analysis 부분이 반자동적으로 이루어진다. 특히 Abstract Syntax Tree에서 구문법적 구조(syntactic category)를 표현해 주는 non-terminal 및 terminal symbol 들을 제거한 압축된 표현양식으로서 compilation의 다음단계인 semantic analysis 및 code generation을 용이하게 해준다. Abstract Syntax tree는 설계자가 지정하여주는 적당한 production rule 및 terminal symbol에 대하여 각각 interior node 및 leaf node를 형성하여 준다.

(2) Amsterdam Compiler Kit〔7〕

Amsterdam Compiler Kit는 compiler의 portability를 증가시켜주기 위하여 stack machine의 일종인 EM virtual machine 에서 사용되는 intermediate code를 생성하여주는 front end와 intermediate code인 EM code를 target code로 변환하는 back end로 구성되어 있다.

특히 back end 부분은 table driven code generation을 이용하여 Intel 8080/8086, Motolora 68000/6809, Zilog Z80/Z8000, DEC PDP-11 그리고 VAX등의 target code를 생성하여 준다. 그러나 Amsterdam Compiler Kit에는 front end의 개발을 위한 tool이 거의 제공되지 않으며 현재 Amsterdam Compiler Kit가 제공하는 front end는 pascal과 C가 있다.

따라서 본 CHILL compiler의 front end는 KAIST Parser Generating System을 이용하여 Amsterdam Compiler Kit의 EM code를 생성하도록 설계하였으며 back end는 Amsterdam Compiler Kit을 이용하여 target code로 변환하거나 interprete 한다.

2.3 CHILL Subset〔2〕

CHILL subset은 full CHILL과 compatible하고 가능한 한 full CHILL이 가지는 visibility와 능력을 가지는 것이 좋으나 개발상의 제약과 prototype compiler의 개발이므로 full CHILL의 몇몇 특성을 제거했다. CHILL subset의 결정에 고려되어야 할 주된 요소는 실용성과 구현의 용이성을 생각할수 있다. CHILL subset은 주로 일반 language의 기능과 concurrent 기능을 중심으로 선택되어 있으며 optional syntax와 alternative syntax가 제거의 주된 대상이었다. 그리고 실용성을 고려하고 다른 language의 기능과 full CHILL에 compatibility가 있도록 CHILL syntax을 제거하거나 간단히 하였다.

그리고 구현의 용이성을 고려하여 layout specification, exception handling 그리고 dynamic data structure 들을 제거하였다.

2.4 Abstract Syntax Tree

Abstract Syntax Tree는 parse tree의 compact한 형태로서 semantic analysis에 편리한 구조를 가지고 있다. Abstract Syntax Tree는 Abstract Syntax Tree를 생성하는 rule을 syntax grammar에서 함께 표시 함으로써 자동적으로 생성하게 되는데 이처럼 Abstract Syntax Tree을 formal 하게 표현함으로써 semantic analysis부분의 높은 validity를 제공하게 된다. 실질적으로 Compile하는데 필요한 모든 정보를 Abstract Syntax Tree로 얻게 되므로 Abstract Syntax Tree의 설계는 중요한 설계 factor 가 된다. Abstract Syntax Tree의 설계에서 고려되어야 할 점은 다음과 같다.

(1) Abstract Syntax Tree의 node의 수와 종류가 많을수록 정확하고 자세한 semantic을 추출할 수 있으나 traverse 해야할 node가 많을수록 Abstract Syntax Tree를 저장하기 위한 storage 및 traverse하기 위하여 execution time이 길어지게 된다. node의 종류는 Abstract Syntax Tree를 traverse 하면서 Symbol table을 만들고 code generation하는 program의 module의 complexity와 structure에 영향을 주게된다. 그러므로 효율적인 information을 가지는 node의 생성이 바람직하다.

(2) Abstract Syntax Tree를 설계하는데 있어서 intermediate code의 structure를 고려해야 한다. intermediate code generation의 phase에서는 Abstract Syntax Tree를 traverse 하면서 code generation을 하게 되는데 이때 필요한 정보는 현재 node에서 되도록 가까운 node에서 나타나도록 Abstract Syntax Tree의 구조를 설계하는것이 바람직하다.

(3) Abstract Syntax Tree의 구조는 input grammar에 의해 결정되는데 symbol table management와 code generation에 필요한 attribute가 traverse하기에 용이하도록 language syntax와는 관계없이 Abstract Syntax Tree node를 생성하기위한 chain rule 등을 추가할 수 있다.

(4) Abstract Syntax Tree를 traverse하기 위한 program module 수는 Abstract Syntax Tree의 subtree의 형태의 종류에 비례한다. 같은 형태의 subtree는 같은 program module을 이용하여 traverse 할 수 있으므로 서로 다른 subtree의 형태가 적을수록 traverse strategy을 implement 하기가 쉬워지며 traverse을 위한 program의 complexity와 size를 줄일 수 있다.

3.1 Front end의 설계 및 구현

front end 부분은 lexical analysis, syntax analysis, semantic analysis, inter-mediate code generation의 phase로 구성되며 back end 부분보다 훨씬 복잡하다.

3.1 Syntax Analysis 및 Abstract Syntax Tree의 구성

syntax analysis 부분을 위한 tool 로서는 YACC(LALR(1) parsing)과 Wisconsin Parser Generating System의 ECP(LALR(1)) 과 FMQ(LL(1)) 등의 여러가지가 있으나 본 연구에서는 KAIST Parser Generating System〔10〕를 이용하였다. KAIST Parser Generating System은 새로운 formalism〔9〕에 의거하여 LALR(1) parsing table을 생성하며 또한 abstract syntax tree〔8〕을 생성한다. 이 abstract syntax tree로 부터 compile 하는데 필요한 모든 정보를 얻게 되므로 abstract syntax tree의 설계는 중요한 설계 factor가 되는데 정보의 양, 속도, space을 고려하여야 하며 full CHILL 로의 확장이 쉬운 구조를 가져야 한다.

Example

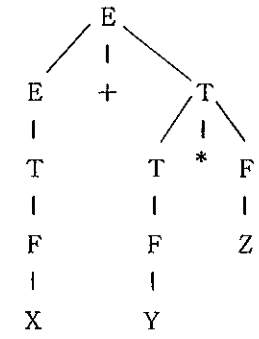
(input grammer)

```
lexicon 'id' ⇒ id:
syntax E
E → E '+' T ⇒ plus:
      T:
T → T '*' F ⇒ mul:
      F:
```

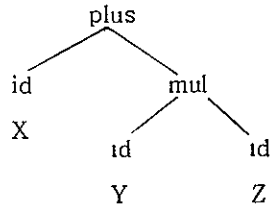
F→' id' :

(input) X+Y*Z

(parse tree)



(abstract syntax tree)



3.2 Semantic analysis

3.2.1 Symbol table management

Semantic analysis 단계에서는 Abstract Syntax Tree을 traverse하면서 필요한 정보를 모아서 symbol table을 구성한다. semantic analysis를 위한 symbol table을 구성하는 주요한 data structure로는 block descriptor, identifier descriptor, mode descriptor가 있다.

(1) block descriptor

identifier의 lifetime과 visibility를 제어하는 data structure로써 stack으로 구성된다.

(2) identifier descriptor

identifier가 declare될때 만들어지는 data structure로서 declare된 identifier에 대한 information이 저장되며 Abstract Syntax Tree의 terminal node에 연결된다. 또한 identifier의 scope를 check하기 위해 모든 identifier descriptor들은 identifier descriptor내의 block link field를 이용하여 연결되어 하나의 spool을 이룬다 CHILL subset에서 declare될 수 있는 identifier의 종류로는 syn name new mode name, structure 내의 field name, procedure name, begin end block의 name, grant identifier, declared identifier, seize identifier, module name, region name, parameter, set element, signal 등이 있다.

identifier descriptor를 구성하는 field는 다음과 같다.

- idclass : identifier의 종류
- name
- identifier가 가지는 mode : pointer
- mode name : new mode의 name에 대한 pointer
- location
- block number : visibility와 life time check에 이용
- nested : block의 nested depth
- block link
- value
- offset : field를 나타내는 identifier일 경우 structure 내에서의 relative position을 표현
- parameter number
- local data area size
- parameter counter
- 기타

(3) mode descriptor

각 identifier 가 가지는 mode 에 대한 information 을 가진다. mode 의 종류로는 integer mode, boolean mode, reference mode, array mode, set mode, structure mode, instance mode, event mode, buffer mode, character mode, new mode 등이 있다.

mode descriptor 를 구성하는 field는 다음과 같다.

- mode 의 종류
- mode 가 가지는 object 의 size
- array mode 에 대한 element 의 수, lower bound 와 upper bound 및 element 의 mode
- structure mode 에서의 field name buffer mode 와 event mode 에서의 element mode
- instance mode 일 경우 process identifier 에 대한 pointer
- 기타

(4) Symbol table 의 구조

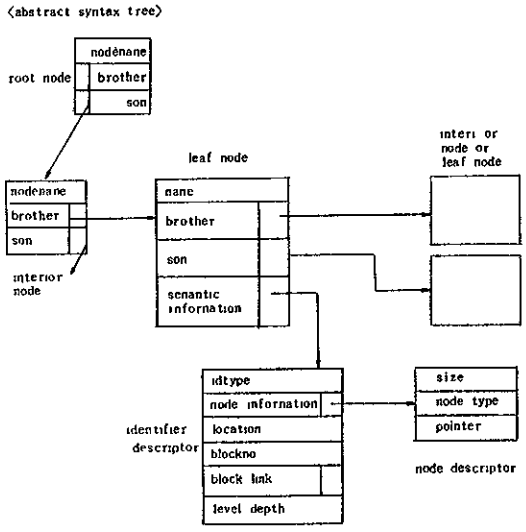


Fig 1 Symbol Table Organization

3.2.2 Visibility analysis

CHILL 에 있어서 module 은 단지 visibility 만을 control 할뿐 life time 은 block 에 의해 control 된다. inner module 과 outer module 은 grant statement 와 seize statement 을 사용하여 visibility 을 가질수 있으며 다른 module 에서 grant 한 것을 다시 seize 함으로써 같은 level 에 있는 module 사이에서도 communication 을 위한 visibility 을 가진다. 이는 region 에서도 동일하다. 임의의 한 module에서 GRANT와 SEIZE의 visibility 을 해결하기 위해서 먼저 current module 에서 선언된 identifier 에 대한 symbol table 을 작성하고 current module 에서 선언된 identifier 들중 Abstract Syntax Tree 를 이용하여 grant 되는 identifier 를 찾아 outer module 의 Symbol table에 grant 된 identifier 의 attribute 들을 outer module 의 Symbol table 에 있는 해당 identifier descriptor 에 copy 해준다. current module 에서 outer module 의 symbol table 을 이용하여 seize 된 identifier 의 name 과 그에 해당하는 모든 attribute 들을 copy 하여 current symbol table 에 insert 한뒤 current module 보다 1 level nested module 에서 Abstract Syntax Tree 를 이용하여 grant 되는 identifier 를 찾아서 identifier 의 name 만을 current Symbol table 에 insert 한다. 이와 같은 방법으로 Abstract Syntax Tree 을 traverse 하면서 지금까지 만들어진 symbol table 을 이용하여 GRANT와 SEIZE의 visibility 을 해결할 수 있다.

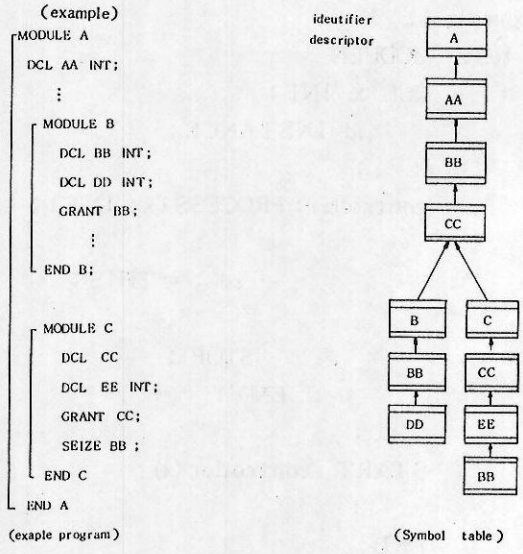


Fig.2 Identifier의 Visibility Visibility

3.3 Concurrent facility들을 위한 EM virtual machine의 확장

원칙적으로 EM virtual machine은 concurrent language를 위한 것이 아니기 때문에 이의 확장이 필요하다.

3.3.1 Process switching을 위한 EM virtual machine architecture의 확장

EM virtual machine은 instruction address space와 data address space을 가진다. data address area는 각각의 addressing mechanism에 따라 global data area, local data area, 및 heap area로, 나누어진다. 하나의 process가 start될때마다 process는 block의 자격이므로 start된 process을 위한 local data area가 allocate되어야 한다. Process는 delay 상태와 running 상태를 천이하면서 수행하게 되는데 수행 중인 process가 state 천이를 할때마다 local data area에 있는 모든 information들을 allocate하거나 deallocate해야 하므로 overhead가 커지게 된다. 따라서 본 연구에서는 각 process에 대해서 별도의 data area을 제공함으로서 이러한 copy-in, copy-out의 overhead을 줄인다. process data area는 frame들로 구성되는데 각 frame들은 start action에 의해 ;generae되고 return이나 stop action에 의해 remove된다. 한 frame은 process의 formal parameter area, 각종 register에 대한 save area, local variable에 대한 area, 그리고 operand stack으로 구성된다. process data area는 CHILL monitor [12]에 의해 경영된다. 또한 process data area을 access하기 위해 새로운 register인 process data area base address register(PDB)를 정의한다. CHILL monitor[12]는 PDB의 내용과 EM code에서 나타나는 displacement을 이용하여 process definition에서 선언된 local variable의 addressing을 가능하게 해야하며 각 process에 해당하는 PDB 및 register의 내용을 save하고 restore할수 있어야 한다.

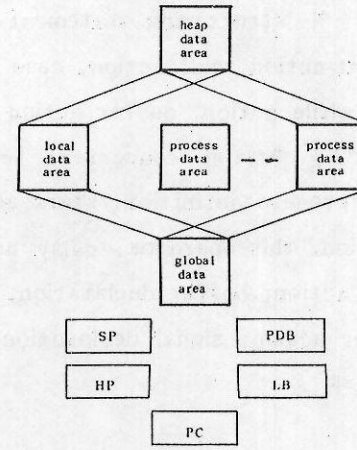


Fig.3 Structure of Data Area

3.3.2 EM Pseudo instruction의 확장

concurrent execution을 위해 CHILL monitor와 communicate할 수 있어야 하기 때문에 EM code을 이용하여 CHILL monitor[12]와 communicate하기 위한 instruction이 필요하다. EM code에서 정의된 MON instruction은 EM monitor 을 call하기 위한 instruction이나, 본 연구에서는 call code와 MON instruction을 이용하여 CHILL monitor와 communicate할수 있는 instruction을 정의했다. CHILL monitor call을 수행하기 위해 stack에 필요한 information을 넣고 다시 monitor call code를 push한뒤 MON instruction을 수행함으로서 실현한다. CHILL monitor에 대한 call code는 각각의 feature에 대해 고유의 number가 주어진다.

3.4 Code generation

symbol table과 Abstract Syntax Tree를 traverse하면서 EM code을 생성한다. module, region process, procedure, begin-end 등의 structuring statement와, assignment action, call action, case action, do while action, do for action , 및 simple I/O을 수행하며 concurrent feature로는 process definition, start action stop action, this operator, delay action continue action, buffer declaration, send buffer action, signal declaration 등을 처리한다.

Example 1.

```
test : MODULE
       DCL x INT;
           id INSTANCE;
           :
       controller: PROCESS (xx INT);
                       :
                   id := THIS;
                       :
                   STOP ;
               END ;
           :
       START  controller (x);
           :
       END;
  corresponding code
         :
enp $m-a-i-n
inp $-1
pro $-1, ?
mes 9, 2
    :
loc  67    ;call code number for THIS
mon        ;monitor call
ste 2      ;id := THIS
    :
loc 65     ;call number for stop
mon        ;monitor call
ret 0
end 2
    :
pro $m-a-i-n, ?
mes 9,0
loc 1      ;process name
loc 2      ;process data area size
loc 78     ;call code for
mon        ;process definition
loe 0      ;actual parameter x
loc 1      ;process name:controller
loc 64     ;call code number for start
mon        ;monitor call
asp 6      ;remove 6 bytes
    :
end 0
```

Example 2.

```
test : MODULE
       DCL a INT;
          :
       IF(a>1) THEN…;
       ELSEIF(a<1) THEN…;
       ELSE…;
     FI ;
        :
     END ;

   corresponding code

   corresponding code
          :
pro $m-a-1-n, ?
  :
loe 0       ;load a
loc 1       ;load 1
cmi 2       ;compare a with 1
tgt         ;true if a>1
zeg*1       ;branch if false
  :
bra*2
1
loe 0
loc 1
cmi 2
tlt         ;true if a<1
zeq*3
  :
bra*4
3           ;else
  :
4
2
  :
```

4. Conclusion

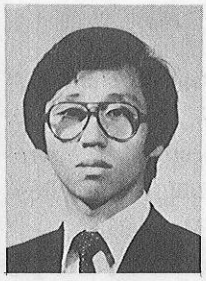
본 연구에서는 ESS 개발환경을 고려하여 CHILL subset compiler를 설계 제작하였다. KAIST parser generating system과 Amsterdam Compiler Kit 등의 compiler-compiler tool들을 이용하여 CHILL를 위한 prototype compiler[11]을 개발하였으며 CHILL의 concurrent facility들을 위한 CHILL monitor을 개발[12]하였다. concurrent processing의 efficiency를 위한 EM virtual machine architecture의 확장 및 error recovery, code optimization 등에 관한 연구가 계속 수행될 예정이며, 특히 program을 위한 simulator 등 software development tool의 개발에 집중적인 연구를 수행할 예정이다.

참 고 문 헌

1. CCITT, "CHILL Language Definition," CCITT Recommendation, Z. 200, Nov. 1980.
2. Beverly A. Butcher, "Selecting an Appropriate CHILL Subset," 2nd CHILL Conference, Illinois, pp.67-72, Mar. 1983.
3. J.L. Keedy, "The Concurrent Programming Features of the CCITT Language CHILL," A.T.R. Vol. 17, No. 1, pp.33-51, 1983.
4. Douglas Comer, Operating System Design The UNIX Approach, Prentice Hall, Bell Lab., 1984.
5. Dennis, Randal E. Bryant, and Jack B., "Concurrent Programming," Computing Structures Group Memo 148-1, MIT-LCS, June 1978.
6. T.A. Haque, "CHILL Programming Environments," Proc., of the IEEE 7th Int'l Conference, COMPSAC, pp.243-244, Nov. 1983.
7. A. Tanenbaum, et al, "A Practical Toolkit for Making Portable Compilers," CACAM, Vol. 26, No. 9, pp.654-660, Sept. 1983.
8. J. Hennesy, Stanford Parser Generator, University of Stanford.

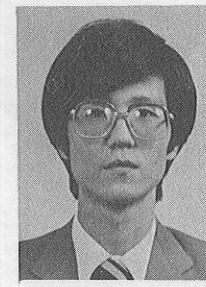
9. K.M. Choe, C.H. Park, and C.H. Chang, "A New Analysis of LALR Formalism," ACM TOPLAS, To be published in 1985.
10. K.M. Choe, "A New Analysis of LALR Formalism," Ph. D. Thesis, KAIST, Korea, Sept. 1983.
11. D.G. Lee, "Development of a Prototype Compiler of a CHILL Subset," M.S. Thesis, Seoul, KAIST, Feb. 1985.
12. Y.I. Yoon, "Development of CHILL Monitor in CHILL Programming Environments Under UNIX," M.S. Thesis, KAIST, Korea, Feb. 1985.

이 동 길



1983년 경북대학교 전자공학과를 졸업하고 1985년 한국과학기술원에서 전산학과 석사학위 취득 후 한국전기통신연구소에서 연구원으로 근무중이며 관심분야는 programming languages software engineering 및 distributed system임.

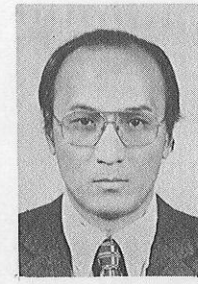
최 광 무



1976년 서울대학교 전자공학과를 졸업하고, 한국과학기술원 전산학과에서 1978년과 1984년에 각각 공학 석사 및 박사학위를 취득하였음.

현재 한국과학기술원 전산학과에서 조교수로 재직중이며, 주요 관심분야는 programming 언어론 및 compiler construction임.

전 길 남



12권 2호 참조