울산대학교 연구논문집 제18권(자연과학·공학편) 제2호 pp. 107~114. 1987
UOU Report Vol.18. (Natural Science and Engineeriog) No.2. pp.107~114. 1987
University of of Ulsan

Prolog언어를 사용한 Compiler구현

朴 佑 昶
전자계산학과
(1987. 4. 30 접수)

＜要 約＞

본 논문에서는 Compiler를 구현하는 여러 언어가 있지만 논리프로그래밍 언어인 Prolog로서 Compiler를 구현할 수 있다는 것을 보였다. BNF로 표기되는 간단한 언어를 정의하고 이 언어로 작성된 프로그램을 문장분석, 코드생성, 어셈블리 과정을 거쳐 목적 프로그램으로 번역하였다. Prolog언어로서 문제해결은 구현과정과 명세과정이 밀접한 연관성이 있다는 점으로부터 Compiler를 작성할 때 얻을 수 있는 장점을 논했다.

On the Implementation of a Compiler with Prolog language

Park, U-Chang
Dept. of Computer Science
(Received April 30,1987)

〈**Abstract**〉

This paper shows that, like other languages, Prolog, as a logic programing language, can be used for the implementation of a compiler. A simple language, expressed by BNF, is defined and a sample program is written according to this form. This program is parsed and translated to target language by the compiler written with Prolog. From the point that the specification is very closely related to the implementation when a problem is solved with Prolog, resulting various advantages are discussed.

Ⅰ. 序 論

Prolog는 논리프로그래밍 언어로서 인공지능, 자연어처리, 데이타베이스 등 여러 방면에 사용되고 있다. Prolog는 그 구현방법이 기존언어들에서의 alogorithm구성과는 달리 문제에서 얻어나는 관계와 대상들에 대해 진위를 표현함으로서 해결한다. 이런 점에서 Prolog언어는 명령적(prescriptive)이면서도 기술적(descriptive)이다. Prolog가 원래 언어의 parsing을 위해 시작되었다는 점을 생각하면 위의 특징과 결합하여 프로그램언어의 번역과정인 compiling을 구현함으로서 장점을 얻을 수 있다.

II. 本 論

compile과정은 다섯단계로 이루어진다. (Figure. 1)

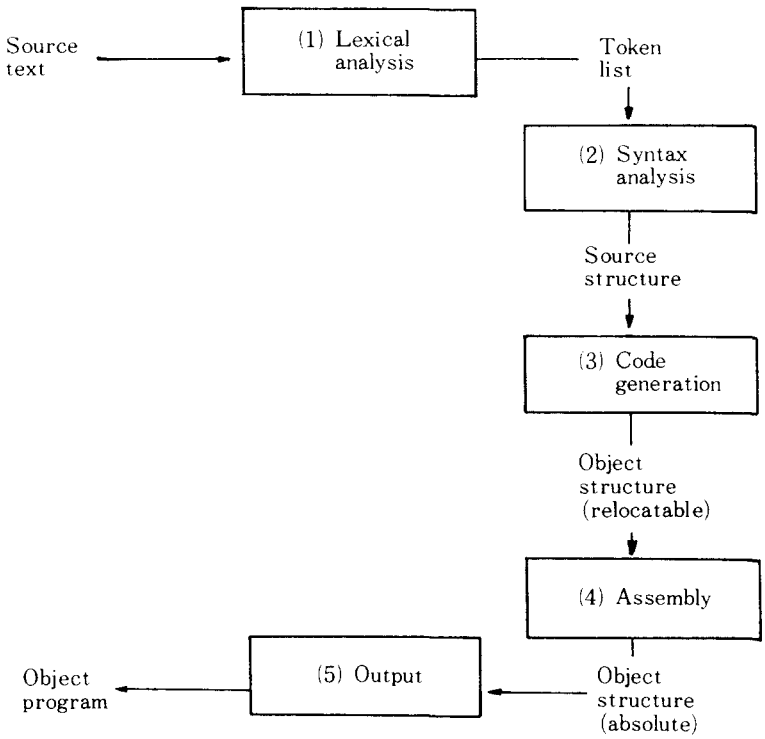

Fig 1 Compile의 단계

첫단계는 구문분석 과정(lexical analysis)으로 source프로그램의 문자들을 분석하여 token들로 나누는 과정이다.

둘째단계는 문장분석과정(syntax analysis)으로 token의 list로부터 parsing을 하는 과정이다. 둘째 단계의 결과로 program의 구조를 인식하여 이 구조에 이름을 붙인다.

예를 들면,

count : =count + 1은
assign(name(count),expression(+,name(count), cons(1)) 로 parsing된다.

세째 단계는 목적생성과정(code generation)으로 object program의 전단계를 만들어낸다.

마지막단계는 에셈블리 과정(assembly)으로 symbol에 대하여 값을 채운다.

compile 과정의 source language는 BNF로 표현되는 간단한 언어 SIMPLE을 정의하였고 source program의 오류는 존재치 않는다고 가정 compiler의 input이 되는 source language의 BNF는 Figure2에 나타나 있다. source language에서는 치환문, if문, while문, 간단한 I/O문을 정의하였고, 산술식의 연산은 사칙연산으로 제한하였다. program은 block을 구성할 수 있도록 begin-end문을 정의 하였다.

```
BNF for a SIMPLE language

<program>     ::= program <identifier>; <statements>
<statement>   ::= begin <statement> <rest>
<statement>   ::= <identifier> := <expression>
<statement>   ::= if <test> then <statement> else <statement
<statement>   ::= while <test> do <statement>
<statement>   ::= readln <identifier> | writeln <expression>
<rest>        ::= ; <statement> <rest>
<rest>        ::= end
<expression>  ::= <constant>
<expression>  ::= <constant> <arithmeticop> <expression>
<test>        ::= <expression> <comparisonop> <expression>
<constant>    ::= <identifier> | <integer>
<arithmeticop> ::= + | - | * | /
<comparisonop> ::= < | > | = | =< | => | <>
```

Fig 2 SiMPLE 언어의 BNF

compiler의 output이 되는 object code는 1 값이다. instruction의 형과 종류는 Figure3에 표시되 operand machine을 가정하고 명령어는(in 어 있다. 기계어명령어는 4칙연산과 load, stone, struction, operand)의 형태로 표시한다. ope fump 그리고 간단한 I/O명령어로 구성되어 있다. rand는 instruction의 형에 따라 주소값이거나 측치

ARITHMETIC		CONTROL	I/O etc
literal	read		
addc	add	jumpeq	readln
subc	sub	jumpne	writeln
multc	mult	jumplt	halt
divc	div	jumpgt	
loadc	load	jumple	
store		jumpge	
		jump	

Fig 3 목적어의 명령어 집합

Figure 2의 BNF에 의하여 1부터 value까지의 factorial값을 구하는 program을 test program으로서 사용하였고 Figure 4에 나타나있다.

```
program factorial;
  begin
    readln value;
    count := 1;
    result := 1;
    while count < value do
      begin
        count := count + 1;
        result := result * count
      end;
    writeln result
  end
```

Fig 4 Factorial을 구하는 Program

구현을 위한 Prolog언어는 IBM PC와 그 호환기종에서 사용되는 TURBO Prolog 로서 typed Prolog compiler이다.[3]

compile을 위한 clause는 다음과 같이 표현된다.

```
compile(Source, Object) :-
        lexana(Source, Tokenlist),
        parse(Tokenlist,Parsedstructure),
        codegen(Parsedstructure,D,Code),
        assemble(Code,D,Object).
```

Figure 4의 프로그램은 처음 source와 unify되어 lexical analyser인 lexana를 호출한다. 변수 Tokenlist는 lexana의 결과 source program을 token의 list로 분류하여 갖고 있다.

parse program은 이 token list를 parsing하여 output으로 Parsedstructure를 생성해 낸다. parsing은 difference list기법으로 각 clause는 input과 두개의 list를 인자로 하여 구성되어 있다.

difference list기법은 list X_1을 $X-X_2$로 표기하는 방법으로서 예를 들면 [a,b]=[a,b,Y]-[Y]로 표시할 수 있다. 이 기법은 program상의 append clause를 호출해야 하는 경우 list대신 difference list를 사용함으로서 효율을 얻을 수 있다.[2]

parse program과 그 결과는 program 1과 Figure 5에 나타나 있다.

parse program은 Figure 2에 표시된 BNF와 매우 비슷한 형태를 취하고 있다는 것을 알 수 있다. 즉 여기서 Prolog program이 가지는 명세와 구현의 밀접성을 발견할 수 있다.

```
/*****************************************/
parse(Source,Structure) :-
                openwrite(parseout,"parse.out"),
                writedevice(parseout),
                parseprogram(Structure,Source,[]),
                write(Structure),
                closefile(parseout).
parseprogram(St,[program,X,";"|Ss],So) :-
    identifier(X),statement(St,Ss,So).
statement(statement(St,Sts),[begin|Ss],So) :-
    statement(St,Ss,S1), rest(Sts,S1,So).
statement(assign(X,V),[X,":","="|Ss],So) :-
    identifier(X),expression(V,Ss,So).
statement(iif(T,Sa,Sb),[iif|IIF],So) :-
    test(T,IIF,S2),then(Sa,S2,S1),else(Sb,S1,So).
statement(while(T,S),[while|While],So):-
    test(T,While,S1),do(S,S1,So).
statement(readln(X),[readln,X|Ss],Ss) :-
    identifier(X).
statement(writeln(X),[writeln| Ss],So) :-
    expression(X,Ss,So).
rest(void,[end|So],So).
rest(statement(St,Sts),[";"|Ss],So) :-
    statement(St,Ss,S1),rest(Sts,S1,So).
expression(X,Ss,So) :- parseconstant(X,Ss,So).
expression(expr(Op,X,Y),Ss,So) :-
    parseconstant(X,Ss,S2), arith_op(Op,S2,S1),
    expression(Y,S1,So).
test(compare(Op,X,Y),[Exp1,Op,Exp2|So],So) :-
```

```
    expression(X,[Exp1],[]),comparisonop(Op),
    expression(Y,[Exp2],[]).
then(Sa,[then|Then],So) :- statement(Sa,Then,So).
else(Sb,[else|Else],So) :- statement(Sb,Else,So).
do(S,[do|S1],So) :- statement(S,S1,So).
arith_op("+",["+"|So],So).
arith_op("-",["-"|So],So).
arith_op("*",["*"|So],So).
arith_op("/",["/"|So],So).
parseconstant(name(X),[X|So],So) :- identifier(X).
parseconstant(number(N),[X|So],So) :- parseinteger(X,N)
identifier(X) :- isname(X).
parseinteger(X,N) :- str_int(X,N).
comparisonop("=").
comparisonop("<").
comparisonop(">").
comparisonop("<>").
comparisonop("=<").
comparisonop("=>").
```

Program I Parse

```
statement(readln("value"),
statement(assign("count",number(1)),
statement(assign("result",number(1)),
statement(while(
            compare("<",name("count"),name("value")),
  statement(assign("count",
              expr("+",name("count"),number(1))),
  statement(assign("result",
              expr("*",name("result"),name("count"))),
  void))),
statement(writeln(name("result")),
void))))
```

Fig 5 Parse의 Output

parse program의 결과 변수 Parsedstructure는 Figure 5와 unify되어 code생성단계인 codegen의 input으로 들어간다. codegen의 두번째 인자인 D는 불완전 자료구조로서 초기에는 무명변수이다가 codegen이 끝날 때 program 중 변수와 레이블들의 이름을 사전식 순서로 기억하고 있는 이진 tree구조이다.

D는 (key,value,Left,Right)의 구조로 된 recursive한 data로서 key값보다 더 적은 key를 갖는 data는 Left에 더 큰 key값을 갖는 data는 Right에 기억된다. 이 이진 tree구조는 lookup clause에 의해 search가 진행되며 만약 search가 실패하면 insert가 된다.

예를 들면 (x, 13), (y, 14)의 data는 t(x,13,−,t(y, 14,−,−)처럼 기억하게 된다. 여기서 −는 무명변수이다. 만약 (z, 9)라는 date를 lookup 시키면 t(x, 13,−,t(y,14,−,t(z,9,−,−)))로 insert가 된다.

codegen program과 그 결과는 program 2와 Figure 6에 나타나 있다.

```
                instr(readln,"value"),instr(loadc,"1"),
                instr(store,"count"),instr(loadc,"1"),
                instr(store,"result"),
  label("L1"),instr(load,"count"),instr(sub,"value"),
                instr(jumpge,"L2"),instr(load,"count"),
                instr(addc,"1"),instr(store,"count"),
                instr(load,"result"),instr(mult,"count"),
                instr(store,"result"),no_op,
                instr(jump,"L1"),
  label("L2"),instr(load,"result"),instr(writeln,"0"),
                no_op]
```

Fig 6 Codegen의 output

```
/*****************************************/
codegen(X,D,Y) :-
        open(codegenout,"codegen.out"),
        writedevice(codegenout),
        encode(X,D,Y),
        write(Y),
        closefile(codegenout).
encode(statement(X,Xs),D,Z) :-
     encode(X,D,Y), encode(Xs,D,Ys), append(Y,Ys,Z).
encode(void,D,[no_op]).
encode(assign(Name,E),D,Z) :-
     lookup(Name,Address,D),
     encode_expression(E,D,Code),
     append(Code,[instr(store,Name)],Z).
encode(if(Test,Then,Else),D,Z) :-
     encode_test(Test,"L1",D,TestCode),
     encode(Then,D,ThenCode),
     encode(Else,D,ElseCode),
     append(TestCode,ThenCode,Z1),
     append(Z1,[instr(jump,"L2"),label("L1")],Z2),
     append(Z2,ElseCode,Z3),
     append([label("L2")],Z3,Z).
encode(while(Test,Do),D,Z) :-
     encode_test(Test,"L2",D,TestCode),
     encode(Do,D,DoCode),
     append([label("L1")],TestCode,Z1),
     append(Z1,DoCode,Z2),
     append(Z2,[instr(jump,"L1"),label("L2")],Z).
```

```
encode(readln(X),D,[instr(readln,X)]):-
     lookup(X,Address,D).
encode(writeln(E),D,Z):-
     encode_expression(E,D,Code),
     append(Code,[instr(writeln,"0")],Z).
encode_expression(number(C),D,[instr(loadc,X)]) :-
     str_int(X,C).
encode_expression(name(X),D,[instr(load,X)]) :-
     lookup(X,Address,D).
encode_expression(expr(Op,E1,E2),D,Z) :-
     single_instruction(Op,E2,D,Instruction),
     encode_expression(E1,D,Load),
     append(Load,Instruction,Z).
encode_expression(expr(Op,E1,E2),D,Code) :-
     not (single_instruction(Op,E2,_,_)),
     encode_expression(E2,D,E2Code),
     single_operation(Op,E1,D,E2Code,Code).
single_instruction(Op,number(C),D,[instr(OpCode,X)]) :-
     literal_op(Op,OpCode),str_int(X,C).
single_instruction(Op,name(X),D,[instr(OpCode,X)]) :-
     memory_op(Op,OpCode), lookup(X,A,D).
single_operation(Op,E,D,Code,Z) :- commutative(Op),
     single_instruction(Op,E,D,Instruction),
     append(Code,Instruction,Z).
single_operation(Op,E,D,Code,Z) :-
     not (commutative(Op)),
     lookup(temp,Address,D),
     encode_expression(E,D,Load),
     op_code(Op,E,OpCode),
     append(Code,[instr(store,temp)],Z1),
     append(Z1,Load,Z2),
     append(Z2,[instr(OpCode,temp)],Z).
op_code(Op,number(C),OpCode) :-
     literal_op(Op,OpCode).
op_code(Op,name(X),OpCode):-
     memory_op(Op,OpCode).
literal_op("+",addc).     literal_op("-",subc).
literal_op("*",multc).    literal_op("/",divc).
memory_op("+",add).       memory_op("-",sub).
memory_op("*",mult).      memory_op("/",div).
commutative("+").         commutative("*").
encode_test(compare(Op,E1,E2),Label,D,Z) :-
     comparison_op(Op,OpCode),
     encode_expression(expr("-",E1,E2),D,Code),
     append(Code,[instr(OpCode,Label)],Z).
comparison_op("=",jumpne).    comparison_op("\=",jumpeq).
comparison_op(">",jumple).    comparison_op("<",jumpge).
```

```
lookup(ID,VAL,t(ID,VAL,_,_)) :- !.
lookup(ID,VAL,t(ID1,_,TREE,_)) :-
    ID < ID1,!,lookup(ID,VAL,TREE).
lookup(ID,VAL,t(_,_,_,TREE)) :- lookup(ID,VAL,TREE).
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

Program2 Codegen

assemble과정은 *codegen*의 결과인 Figure 6과 unify된 변수 Code와 symbol table의 역할을 하는 변수 D를 input으로하여 각 변수들에 대하여 주소를 할당하고 절대 *code*인 *object*를 만들어낸다. *Object*는 Figure 7에 나타나있다. Object code는 절대주소 0을 기준으로 번역되며 block(3)은 assembly언어의 define storage(DS)역할을 한다.

```
[code(readln,21),code(loadc,1),code(store,19),
 code(loadc,1),code(store,20),code(load,19),
 code(load,21),code(jumpge,16),code(load,19),
 code(loadc,1),code(store,19),code(load,20),
 code(null,17),code(store,20),code(jump,6),
 code(load,20),code(writeln,0),code(halt,0),
 _|_]
```

Fig 7 Compile이 끝난 Object Code

III. 결 론

Prolog언어로서 compiler구현은 최종목표인 compiler자체보다도 그 중간의 design과정, coding 과정 그리고 유지, 보수과정까지 같이 생각해 보는 것이 좋을 것같다. compiler을 구성하는 과정은 compiler의 기능에 대한 명세 (specificationg)가 되는데 이 과정은 바로 구현(implementation)과 밀접한 관계가 있다. 즉, 문제에 대한 명세가 구현과정이 되는 것이다. 이러한 명세와 구현의 밀접한 연관성은 구현된 code가 더 읽기 쉽고 자기 설명적이며 증명이 쉬워진다. 또 compiler의 수정이나 source language의 확장이 용이하다. 위의 장점외에 더 적은 시간과 노력으로 compiler를 구성할 수 있고 오류가 적으며 유지와 보수가 쉽다.

Prolog에 의한 compiler구현은 효율성에 있어서 저급언어에는 미치지 못하지만 구현하는 Prolog언어 자체의 효율성에 많이 의존한다.

參 考 文 獻

1) Clocksin, W. F and Mellish, C. S., "Programming in Prolog", 2nd Edition Springer—Verlag, New York, 1984.
2) Sterling, Leon and Shapiro, Ehud, "The Art of Prolog", MIT Press, 1986.
3) TURBO PROLOG, Borland International Inc, 1986.
4) Warren, David H. D., "Logic programming and compiler writing", SOFTWARE —Practice and Experience, Vol. 10, Number II, 1980.