INTERMEDIATE CODE GENERATION
Part II

Based on Chapter 6 of Aho, Lam, Sethi, Ullman:

Compilers: Principles, Techniques, & Tools

2nd Ed, Addison Wesley, 2007

Table of Contents

- Introduction
- Intermediate Representation
- Assignment Statement
- Addressing Array Elements
- Type Checking
- Boolean Expressions
- Flow of Control Statements
- Backpatching
- Procedure Calls

Motivation

- While statement

S->while E do S1

S.begin = newlabel

S.after = newlabel

S.code = gen(S.begin ‘:’) | |

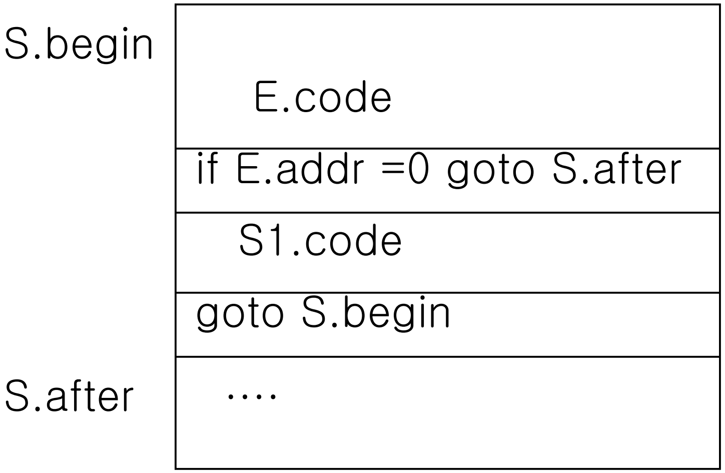
E.code | |

gen(‘if’ E.addr ‘=’ ‘0’ goto S.after) | |

S1.code | |

gen(‘goto’ S.begin) | |

gen(S.after ‘:’)



S.begin

E.code

if E.addr =0 goto S.after

S1.code

goto S.begin

S.after

….

Boolean Expressions

- Two purposes
  - to compute logical values
  - used as conditional expressions in the flow of control statements
- Grammar

  E -> E | | E | E && E | ! E | (E) | E **relop** E | **true** | **false**
- Two Methods of Translating Expressions
  - Number representation
    - to encode true and false numerically e.g. true to 1 and false to 0 and to evaluate a boolean expression analogously to an arithmetic expression
  - Control-flow translation
    - to represent the value of a boolean expression by a position reached in the code
    - convenient in implementing the boolean expression in flow-of-control statements

Boolean Expressions

- Numerical Representation

```
E -> E1 || E2        { E.addr = newtemp;
                       gen(E.addr ‘=‘ E1.addr ‘or’ E2.addr);  }
E -> E1 && E2        { E.addr = newtemp;
                       gen(E. addr ‘=‘ E1. addr ‘and’ E2. addr);  }
E ->    ! E1         { E. addr = newtemp;
                       gen (E. addr ‘=‘ ‘not’ E1. addr); }
E -> (E1)            { E. addr = E1. addr; }
E -> E1 relop E2     { E. addr = newtemp;
                         gen (‘if’ E1. addr relop.op  E2. addr ‘goto’ nextinstr + 2);
                         gen (E. addr ‘=‘ ‘0’);
                         gen (‘goto’ nextinstr + 1);
                         gen (E. addr ‘=‘ ‘1’);  }
E -> true            { E.place = newtemp;
                       gen (E. addr ‘=‘ ‘1’); }
E -> false           { E.place = newtemp;
                       gen (E. addr ‘=‘ ‘0’); }
```

Boolean Expressions

- Example(Numerical Representation)

```
Ex1:  a | | b && ! c
=>
t1 = not c
t2 = b and t1
t3 = a or t2


Ex2:  a < b
=>
100: if a < b goto 103
101: t = 0
102: goto 104
103: t = 1
104:
```

```
Ex3:  a < b || c < d && e < f
=>
100: if a < b goto 103
101: t1 = 0
102: goto 104
103: t1 = 1
104: if c < d goto 107
105: t2 = 0
106: goto 108
107: t2 = 1
108: if e < f goto 111
109: t3 = 0
110: goto 112
111: t3 = 1
112: t4 = t2 and t3
113: t5 = t1 or t4
```

Flow-of-Control Statements

- Grammar

  S -> if (E) S1
  | if (E) S1 else S2
  | while (E) S1
  | S1 S2
  | **assign**

- A boolean expression E is associated with two labels:
  - E.true : the label to which control flows if E is true
  - E.false: the label to which control flows if E is true

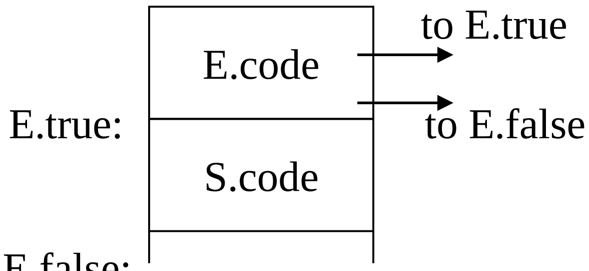
inherited attributes

- The semantic rule of control statement S allow control to flow from within S.code to the instruction immediately following S.code
  - S.next : a label of the first instruction to be executed after the code for S

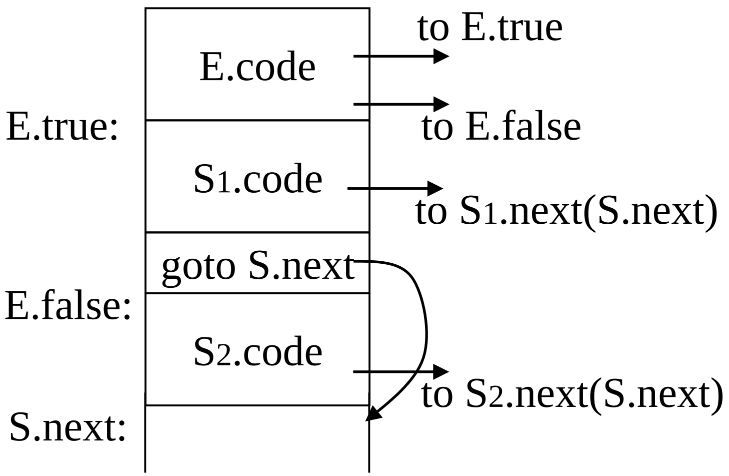
inherited attributes

Flow-of-Control Statements

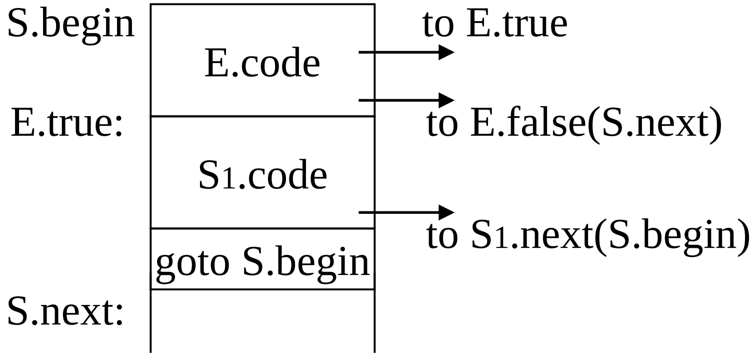
- Code Structures



if-then



if-then-else



while statement

Flow-of-Control Statements

- Syntax-directed definition

S-> if (E) S_1	E.true = newlabel
	E.false = S.next
	S_1.next = S.next
	S.code = E.code || gen(E.true ’:’) || S_1.code

S -> if (E) S_1 else S_2	E.true = newlabel E.false = newlabel S_1.next = S.next S_2.next = S.next S.code = E.code || gen(E.true ‘:’) || S_1.code gen(‘goto’ S.next) || gen(E.false ‘:’) || S_2.code

S -> while (E) S_1	S.begin = newlabel
	E.true = newlabel
	E.false = S.next
	S_1.next = S.begin
	S.code = gen(S.begin ‘:’) ||E.code ||gen(E.true ‘:’) ||
	S_1.code || gen(‘goto’ S.begin)

Flow-of-Control Statements

- Syntax-directed definition

P->S	S.next = newlabel
	P.code = S.code || gen(S.next ’:’)

S -> **assign**	S.code = **assign**.code

S -> S1 S2	S1.next = newlabel
	S2.next = S.next
	S.code = S1.code || gen(S1.next ‘:’) || S2.code

Control-Flow Translation of Boolean Exp.

- E is translated into a sequence of instructions that evaluates E as a sequence of conditional and unconditional jumps to one of two locations : E.true and E.false
  - Basic idea

Ex 1. a < b

```
if a < b goto E.true
goto E.false
```

Ex2. E_1 || E_2

short circuit evaluation

if E_1 is true, E itself is true => E_1.true = E.true
else E_2 must be evaluated, so E_1.false be the label of the first instruction in the code of E_2.

if E_2 is true, E itself is true => E_2.true = E.true
else E_2 is false, E itself is false => E_2.false=E.false

Control-Flow Translation of Boolean Exp.

- Syntax-directed definition

```
E -> E1 || E2        E1.true = E.ture;         E1.false = newlabel
                     E2.true = E.true;         E2.false = E.false
                     E.code = E1.code || gen(E1.false ':') || E2.code

E-> E1 && E2         E1.true = newlabel;       E1.false = E.false
                     E2.true = E.true;         E2.false = E.false
                     E.code = E1.code || gen(E1.true ':') || E2.code

E -> ! E1            E1.true = E.false;        E1.false = E.true
                     E.code = E1.code
E -> ( E1 )          E1.true = E.true;         E1.false = E.false;  E.code = E1.code

E-> E1 relop E2      E.code = E1.code || E2.code
                           || gen('if' E1.addr relop.op E2.addr 'goto' E.true) ||
                              gen('goto' E.false)
E -> true            E.code = gen('goto' E.true)
E -> false           E.code = gen('goto' E.false)
```

Control-Flow Translation of Boolean Exp.

- Example

```
a < b || c < d && e < f

        if a < b goto Ltrue
        goto L1
    L1: if c < d goto L2
        goto Lfalse
    L2: if e < f goto Ltrue
         goto Lfalse
```

Ltrue:
  true exit for the exp
Lfalse:
  false exit for the exp

```
while ( a < b )
    if ( c < d )
        x = y + z
    else
        x = y - z

    L1: if a < b goto L2
        goto Lnext
    L2: if c < d goto L3
        goto L4
    L3: t1 = y + z
         x = t1
         goto L1
    L4: t2 = y - z
         x = t1
         goto L1
    Lnext:
```

Lnext:
  while문의 S.next임

if의 S.next는 while의 S1의 next이고
이는 while의 S.begin(L1)임

Mixed Mode Boolean Expressions

- Sample Grammar

  E -> E + E | E && E | E **relop** E | **id**

  - E **relop** E produces boolean values
  - E && E requires both arguments to be boolean
  - E + E and E **relop** E take either type of arguments(arithmetic or boolean)

  - To determine type of expression, we can use a synthesized attribute E.type

  - Example: a + (b < c)

```
        if b < c goto Ltrue
        goto Lfalse
Ltrue: t1 = a + 1
        goto Lnext
Lfalse: t1 = a
Lnext:
```

Mixed Mode Boolean Expressions

```
E -> E1 + E2          E.type = arith
                      if E1.type = arith and E2.type = arith then
                         E.addr = newtemp;
                         E.code = E1.code || E2.code ||
                                     gen(E.addr ‘=‘ E1.addr ‘+’ E2.addr)

                      else if E1.type = arith and E2.type = bool then
                         E.addr = newtemp;
                         E2.true = newlabel;
                         E2.false = newlabel;
                         E.code = E1.code || E2.code ||
                                     gen(E2.true ‘:’ E.addr ‘=‘ E1.addr ‘+’ ‘1’) ||
                                     gen(‘goto’ nextinstr + 1) ||
                                     gen(E2.false ‘:’ E.addr ‘=‘ E1.addr)
                      ….
```

Backpatching

- How to implement the syntax directed definition of boolean expressions and control flow statements
  - in two passes
    1. construct a syntax tree
    2. translate it walking in depth-first order
  - in single passes
    - use backpatching
- Backpatching
  - the targets of the jumps temporarily left unspecified
  - put on a list of goto statements whose labels will be filled in when the proper label can be determined
- Three Functions used in Backpatching
  - makelist(i) : creates a new list containing only i, an index to an instruction
  - merge(p, q) : merges two lists p and q
  - backpatch(p, i): inserts i as the target label for each statement on the list p

Boolean Expressions

- Grammar

  E -> E1 | | M E2

  | E1 && M E2

  | ! E1

  | (E1)

  | id1 **relop** id2

  | **true**

  | **false**

  M -> ε // E1의 true시 또는 false 시 goto 해야 할 위치 계산

- Synthesized Attributes
  - E.truelist : E.true 가 발견되면 backpatching되어야할 명령들
  - E.falselist: E.false 가 발견되면 backpatching되어야할 명령들
  - M.instr : 다음 명령어의 위치

Boolean Expressions

- E -> E_1 && M E_2
  - E_1이 false이면 E 도 false가 된다. 즉, E.false가 발견되면 backpatch될 리스트(E.falselist)에 E_1.falselist 가 추가해야한다.
  - E_1이 true이면 E_2 코드의 시작(M.instr)로 jump해야한다. 즉, E_1의 truelist를 M.instr 로 backpatching해야한다.
  - E_2가 true이면 E도 역시 true이다.(E_1이 true인경우에만 테스트됨) 따라서 E.truelist에 E_2.truelist도 추가해야한다.
  - E_2가 false이면 E도 역시 false이다. 따라서 E.falselist에 E_2.falselist도 추가되어야 한다.

```
E -> E1 && M E2  {      backpatch(E1.truelist, M.instr);
                        E.truelist = E2.truelist;
                        E.falselist = merge(E1.falselist, E2.falselist);
                 }
```

Boolean Expressions

- Translation of Boolean Expressions

```
E -> E1 | | M E2            { backpatch(E1.falselist, M.instr);
                             E.truelist = merge(E1.turelist, E2.truelist);
                              E.falselist = E2.falselist; }
E -> ! E1                   { E.truelist = E1.falselist;
                              E.falselist = E1.truelist; }
E -> ( E1 )                 { E.truelist = E1.truelist;
                              E.falselist=E1.falselist; }
E -> id1 relop id2          { E.truelist = makelist(nextinstr);
                              E.falselist = makelist(nextinstr + 1);
                             emit('if' id1.addr relop.op id2.addr 'goto_');
                             emit('goto _');  }
E -> true                   { E.truelist = makelist(nextinstr);
                              emit('goto _'); }
E -> false                  { E.falselist = makelist(nextinstr);
                              emit('goto _'); }
M -> ε                      {M.instr = nextinstr; }
```

Boolean Expression

- Example : a < b || c < d && e < f

E1-> a < b	100: if a < b goto _	E1.truelist = {100}
	101: goto _	E1.falselist = {101}
M1 -> ε		M1.instr = 102
E2-> c<d	102: if c < d goto _	E2.truelist = {102}
	103: goto _	E2.falselist = {103}
M2 -> ε		M2. instr= 104
E3 -> e<f	104: if e < f goto _	E3.truelist = {104}
	105: goto _	E3.falselist = {105}
E4 -> E2 && M E3		backpatch({102}, 104)
		E4.truelist = {104 }
	100: if a < b goto _	E4.falselist = {103, 105}
	101: goto _	
	102: if c < d goto 104	
	103: goto _	
	104: if e < f goto _	
	105: goto _	

Boolean Expression

- Example : a < b || c < d && e < f (cont.)

$E \rightarrow E_1 \ || \ M \ E_4$

backpatch({101}, 102)

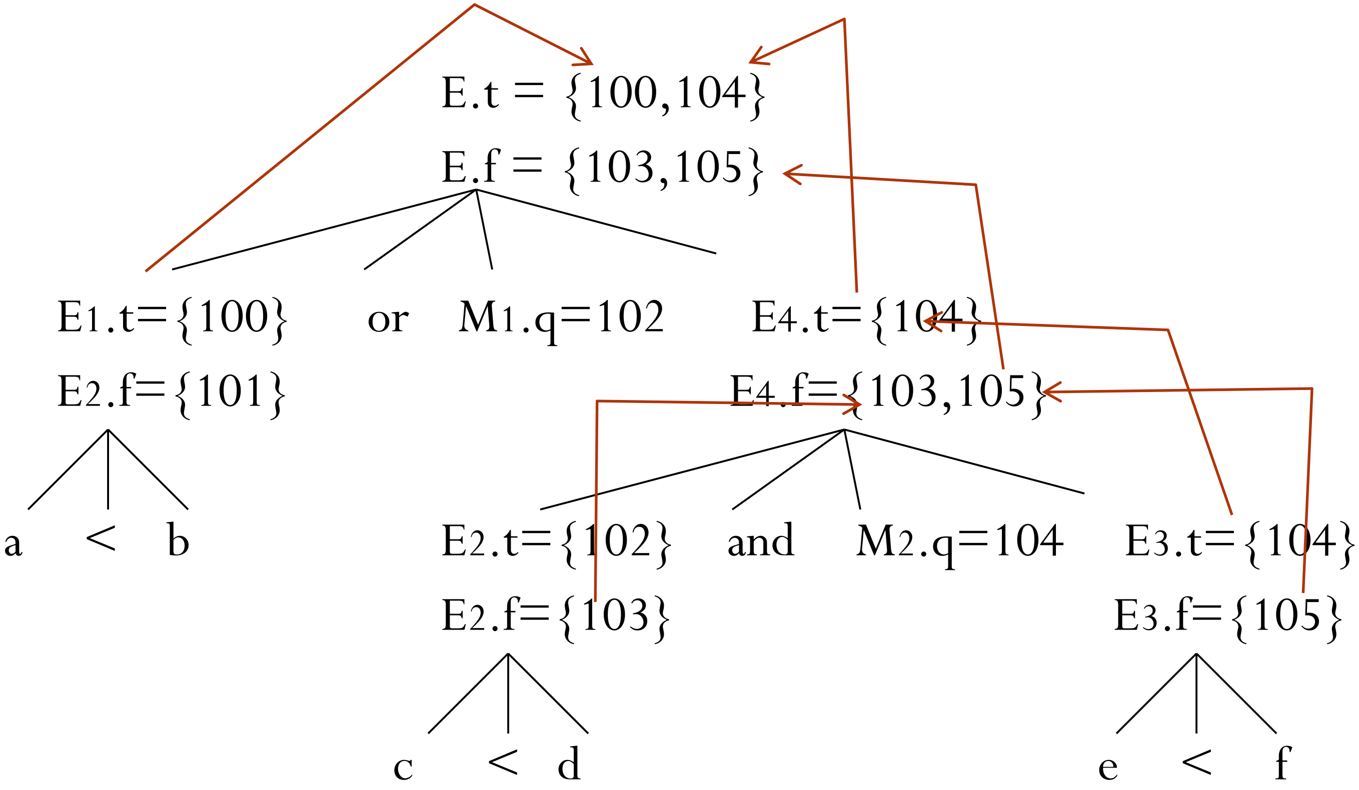
E.truelist = {100, 104 }

E.falselist = {103, 105}

```
100: if a < b goto _
101: goto 102
102: if c < d goto 104
103: goto _
104: if e < f goto _
105: goto _
```

Boolean Expression

- Example : a < b || c < d && e < f (cont.)



E.t = {100,104}
E.f = {103,105}

E1.t={100}
E2.f={101}

or

M1.q=102

E4.t={104}
E4.f={103,105}

a < b

E2.t={102}
E2.f={103}

and

M2.q=104

E3.t={104}
E3.f={105}

c < d

e < f

Flow-of-Control Statements

- Grammar

```
S -> if ( E ) S
    |  if ( E ) S else S
    |  while ( E ) S
    |  { L }
    |  A
L -> L  S
    |  S
```

- Attributes
  - E.truelist and E.falselist
  - S.nextlist : S의 다음 명령어로 jump할 명령어들
  - L.nextlist : L의 다음 명령어로 jump할 명령어들

Flow-of-Control Statements

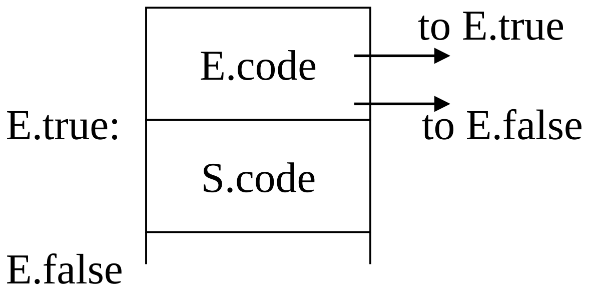
- if -then문

  S -> if (E) S_1

  - E.true의 위치를 기록하기 위한 marker 필요

  S-> if (E) M S_1

  - E.truelist를 M.instr로 backpatch
  - S_1.nextlist를 S.nextlist에 추가
  - E.falselist를 S.nextlist에 추가

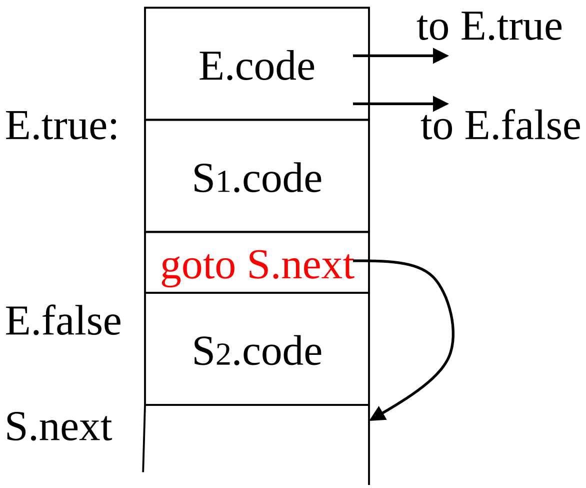


Flow-of-Control Statements

- if -then-else문

  S -> if (E) S_1 else S_2

  - E.true, E.false의 위치를 기록하기 위한 marker 필요
  - goto S.next 명령을 생성할 marker 필요



E.true:
E.false
S.next
E.code
S_1.code
goto S.next
S_2.code
to E.true
to E.false

  S-> if (E) M_1 S_1 N else M_2 S_2

  - E.truelist를 M_1.instr로 backpatch
  - E.falselist를 M_2.instr로 backpatch
  - S_1.nextlist를 S.nextlist에 추가
  - S_2.nextlist 를 S.nextlist에 추가
  - N.nextlist(goto S.next 명령 위치)를 S.nextlist에 추가

Flow-of-Control Statements

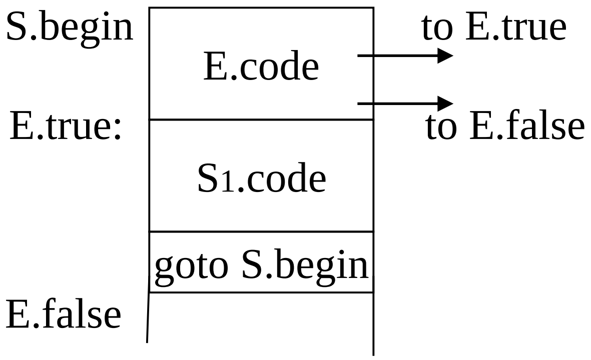
- While 문

  S -> while (E) S_1

  - S.begin과 E.true의 위치를 기록하기 위한 marker 필요

  S -> while M_1 (E) M_2 S_1

  - E.truelist를 M_2.instr로 backpatch
  - S_1.nextlist를 M_1.instr로 backpatch
  - E.falselist를 S.nextlist에 추가한다.



Flow-of-Control Statements

- Translation Scheme

```
S -> if ( E ) M S1                { backpatch(E.truelist, M.instr);
                                    S.nextlist = merge(E.falselist, S1.nextlist); }
M -> ε                            { M.quad = nextquad; }
S -> if ( E ) M1 S1 N else M2 S2
                                  { backpatch(E.truelist, M1.instr);
                                    backpatch(E.falselist, M2.instr);
                                    S.nextlist=merge(S1.nextlist, N.nextlist,
                                                         S2.nextlist); }
N -> ε                            { N.nextlist = makelist(nextinstr);
                                    emit('goto _'); }
S -> while M1 ( E ) M2 S1
                                  { backpatch(S1.nextlist, M1.instr);
                                    backpatch(E.truelist, M2.instr);
                                    S.nextlist = E.falselist;
                                    emit('goto' M1.instr); }
```

Flow-of-Control Statements

- Translation Scheme (cont.)

S -> { L }	{ S.nextlist = L.nextlist; }
S -> **A**	{ S.nextlist = null; }

L -> L$_1$ M S	{ backpatch(L$_1$.nextlist, M.instr);
	L.nextlist = S.nextlist; }

L -> S	{ L.nextlist = S.nextlist; }

Function Call

- Grammar

E -> id (Elist)

Elist -> Elist, E

| E

- Elist의 각 E 코드를 모두 생성한 후 호출 명령을 생성
  - Elist의 E를 위한 E.addr를 queue에 저장한 후
  - E-> id(Elist)를 reduce할 때 queue에서 차례대로 꺼내어
  - param E.addr 명령 생성

E_1.code

E_2.code

....

param E_1.addr

param E_2.addr

...

E.addr = call id.addr

Function Call

- 배열의 Elist->Elist , E와 구분

  E -> Elist)

  Elist -> Elist , E

  | id (E

  id의 type에따라(함수, 배열)

  Elist -> Elist , E에서 해야할일을

  달리 할 수 있다.

- Recursion

  fun-decl -> type id (para-list) body => body parsing 때 function definition 발견안됨

  =>

  fun-decl -> header body

  header -> type id (para-list) { enter function type }