Motivation:

1. Easier to construct compilers for different architectures (modular $\Rightarrow$only intermediate-code-to-machine-code step needs modification).

2. Optimization is machine independent.

Intermediate languages

3-address code: sequence of statements of the general form

```
x = y op z
```

where `x`, `y`, `z` – names, constants or compiler-generated temporaries

`op` – operator (arithmetic, logical, shift, etc.) that takes **at most two** operands

Example:

```
i = 2*j + k - 1;
```

⇓

```
t1 = 2 * j
t2 = t1 + k
i = t2 - 1
```

Instruction set

Assignment statements

```
x = y op z   x = op y   x = y
```

Array references

```
x = y[i]   x[i] = y
```

Pointer operations

```
x = &y      x = *y     *x = y
```

Jumps

```
goto L        if x relop y goto L
```

Procedure calls

```
param x1
param x2
⋮
param xn
call p, n
```

Implementation

1. **Quadruples:**
   - structure with 4 fields
     ```
     typedef struct {
       int op;
       SYM_TAB *arg1, *arg2, *result;
     } QUAD;
     ```
   - any unused field is left blank/NULL
   - Disadvantage: temporary names have to be entered into symbol table

Implementation

2. **Triples:**
   - avoids entering temporary names into symbol table
   - for a temporary, use serial number of statement computing its value
   - use record with three fields: operator, arg1, arg2
   - flag (separate field) specifies whether operand is pointer to symbol table entry or to triple
   - for assignments:

Instruction	Representation		
	operator	operand1	operand2
`a = t1`	ASSIGN	`a`	(n)
`x[i] = y`	(0) `[]=`	`x`	`i`
	(1) ASSIGN	(0)	`y`
`x = y[i]`	(0) `=[]`	`y`	`i`
	(1) ASSIGN	`x`	(0)

Assignment statements – I

Section 8.3

$S \rightarrow \mathbf{id} = E$
```
{ p = lookup(id.name);
  if (p != NULL)
     /* id = E.place */
     gen(ASSIGN, p, E.place);
  else error(); }
```

$E \rightarrow E_1 + E_2$
```
{ E.place = newtemp();
  /* E.place = E1.place + E2.place */
  gen(ADD,E.place,E1.place,E2.place); }
```

$E \rightarrow -E_1$
```
{ E.place = newtemp();
  /* E.place = - E1.place */
  gen(UMINUS, E.place, E1.place); }
```

$E \rightarrow (E_1)$
```
{ E.place = E1.place }
```

$E \rightarrow \mathbf{id}$
```
{ p = lookup(id.name);
  if (p != NULL) E.place = p;
  else error(); }
```

Assignment statements – I

Auxiliary functions:
lookup – returns pointer to symbol table entry for given argument
gen – generates a 3-address statement (prints to file or adds to array)
newtemp – generates a new temporary variable name

Assignment statements – I

Re-using temporary names:

- Bulk of temporaries are generated during translation of expressions, e.g.

  $E \rightarrow E_1 + E_2$ `{ E.place = newtemp();`

  `gen(ADD, E.place, E1.place, E2.place); }`

- $E_1.place,\ E_2.place$ not used elsewhere in the program $\Rightarrow$can reuse temporary names used for E_1, E_2

Method:

1. Initialize *count* to 0.
2. When a temporary is used as an operand, decrement *count*; when a new temporary is needed, create t_{count} and increment *count*.

Example: `x = a*b + c*d - e*f;`

Assignment statements – II

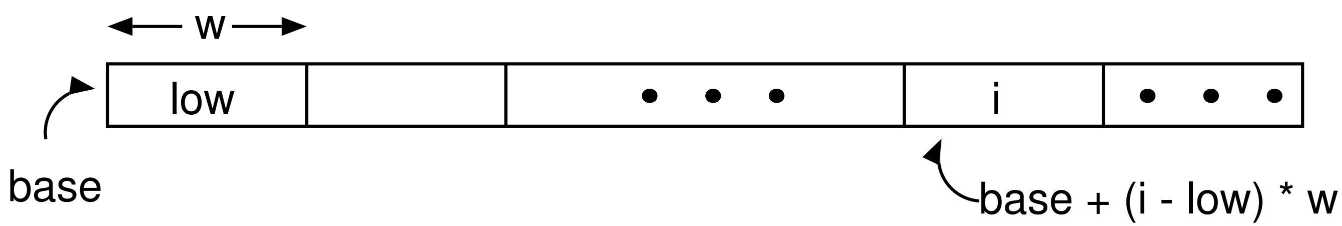
Aim: handle mixed-type expressions

$E \rightarrow E_1 + E_2$

```
{ E.place = newtemp();
  if E1.type==INT && E2.type==INT
     gen(ADDI,E.place,E1.place,E2.place)
     E.type = INT;
  else if E1.type==INT && E2.type==FLOAT
     u = newtemp();
     gen(ITOF, u, E1.place);
     gen(ADDF,E.place,u,E2.place);
     E.type = FLOAT;
  ...
```

Array variables

1-dimensional arrays:

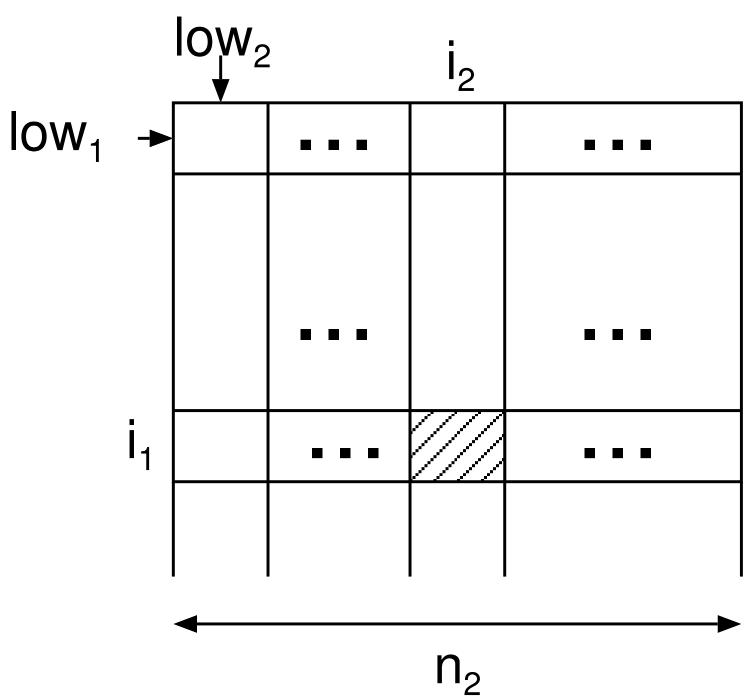


$$\text{Address of } A[i] = base + w \times (i - low)$$
$$= \underbrace{(base - w \times low)}_{c} + w \times i$$

c – constant that can be computed at compile time and stored in symbol table entry for A

2-dimensional arrays:

- Column major form (Fortran):
  $A[1,1] \quad A[2,1] \quad A[3,1] \quad \ldots \quad A[1,2] \quad A[2,2] \quad \ldots$
- Row major form (C, Pascal):
  $A[1,1] \quad A[1,2] \quad A[1,3] \quad \ldots \quad A[2,1] \quad A[2,2] \quad \ldots$



Address of $A[i_1, i_2]$

$$\begin{aligned}
&= base + \\
&\quad w \times (i_1 - low_1) \times n_2 + \\
&\quad w \times (i_2 - low_2) \\
&= w \times (i_1 \times n_2 + i_2) + \\
&\quad \underbrace{\overbrace{base - w \times}\ (low_1 \times n_2 + low_2)}
\end{aligned}$$

Array variables

General recurrence relation:

$$e_1 = i_1$$
$$e_m = e_{m-1} \times n_m + i_m$$

Modified grammar: replace **id** with L, where:

$$\begin{array}{lcl} L & \rightarrow & \mathbf{id}\ [\ Elist\ ]\ \mid\ \mathbf{id} \\ Elist & \rightarrow & Elist\ ,\ E\ \mid\ E \end{array} \quad \Rightarrow \quad \begin{array}{lcl} L & \rightarrow & Elist\ ]\ \mid\ \mathbf{id} \\ Elist & \rightarrow & Elist\ ,\ E\ \mid\ \mathbf{id}\ [\ E \end{array}$$

$S \rightarrow L = E$
```
{ if (L.offset == NULL) gen(ASSIGN, L.place, E.place);
  else gen([]=, L.place, L.offset, E.place); }
```

$E \rightarrow L$
```
{ if (L.offset == NULL) E.place = L.place;
  else { E.place = newtemp();
         gen(=[], E.place, L.place, L.offset); }}
```

$L \rightarrow \mathbf{id}$
```
{ L.place = lookup(id.name);   L.offset = NULL; }
```

Array variables

$L \rightarrow Elist\ ]$
```
{ L.place = newtemp();   L.offset = newtemp();
  gen(ASSIGN, L.place, const_part(Elist.array));
  gen(MULT, L.offset, Elist.place, w(Elist.array)); }
```

$Elist \rightarrow id\ [\ E$
```
{ Elist.array = lookup(id.name);
  Elist.place = E.place;
  Elist.ndim = 1; }
```

$Elist \rightarrow Elist_1, E$
```
{ t = newtemp();   m = Elist1.ndim + 1;
```
/* $e_m = e_{m-1} \times n_m + i_m$ */
```
  gen(MULT, t, Elist1.place, n(Elist.array, m));
  gen(ADD, t, t, E.place);
  Elist.place = t;
  Elist.array = Elist1.array;   Elist.ndim = m; }
```

Example:
```
int A[10,20];
x = A[y,z];
```

Boolean expressions

Section 8.4

Uses:

1. Computing logical values
2. Control flow

Translation issues:

Encoding: true - 1, false - 0, OR
true - non-zero values, false - 0 OR
true - positive values, false - zero or less

Form of evaluation: evaluate as numerical expression, OR
evaluate using flow of control

Extent of evaluation: complete or lazy

Numerical values + Complete evaluation

$E \to E_1$ **or** E_2	`{ E.place = newtemp();` `gen(E.place = E1.place OR E2.place); }`
$E \to E_1$ **and** E_2	`{ /* analogous */ }`
$E \to$ **not** E_1	`{ E.place = newtemp();` `gen(E.place = NOT E1.place); }`
$E \to (E_1)$	`{ E.place = E1.place; }`

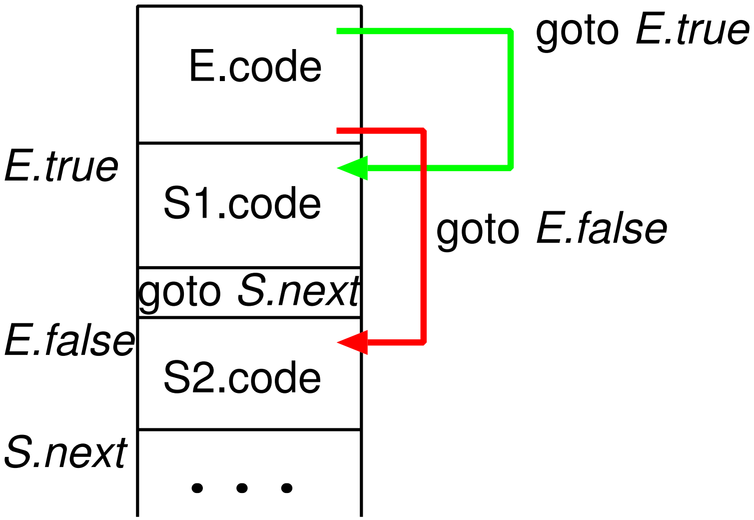
$E \to$ **id**$_1$ **relop id**$_2$	`{ E.place = newtemp();`
	`gen(if id1.place relop.op id2.place goto next +3);`
	`gen(E.place = 0);`
	`gen(goto next +2);`
	`gen(E.place = 1); }`

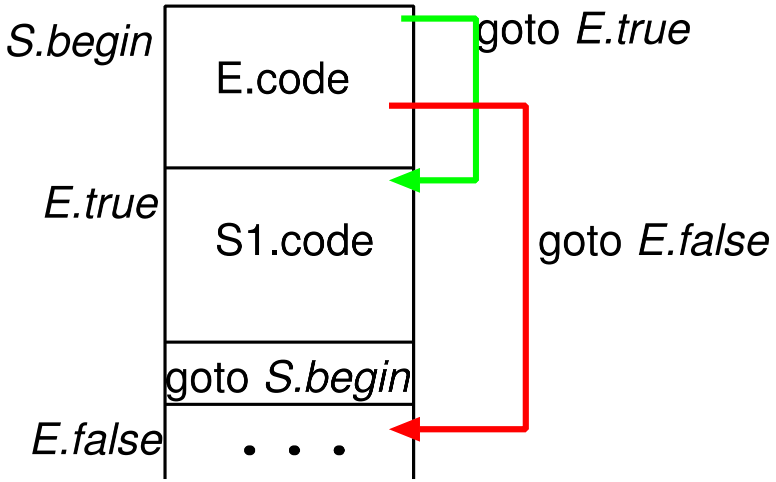
$E \to$ **true**	`{ E.place = newtemp();` `gen(E.place = 1); }`
$E \to$ **false**	`{ /* analogous */ }`

next - serial # of next instruction to be generated

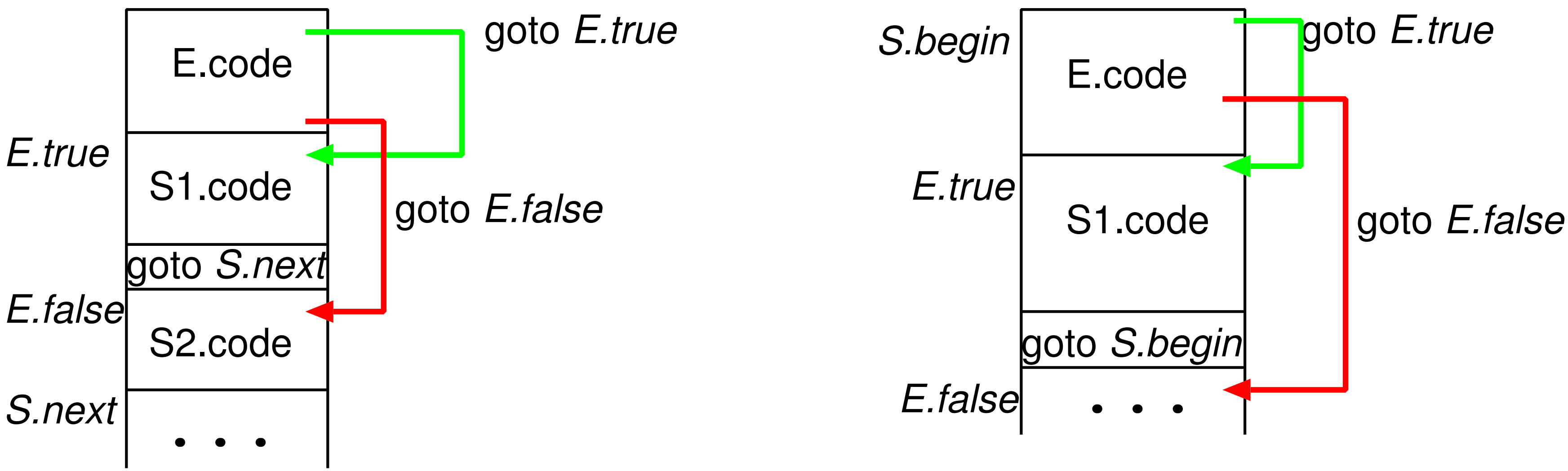
Example: `a<b or c>d and i==j`

Control flow + Lazy evaluation





Control flow + Lazy evaluation



Attributes:

- $E.true$, $E.false$ (inherited) – label of statement to which control should flow if E is true (false)
- $S.next$ (inherited) – label of first instruction to be executed after S
- $E.code$ (synthesized) – sequence of 3-address instructions corresponding to E

Control flow + Lazy evaluation

$S \rightarrow \textbf{if } E \textbf{ then } S_1$

```
{ E.true = newlabel(); E.false = S.next;
  S1.next = S.next;
  S.code = E.code +
           "E.true :" + S1.code; }
```

$S \rightarrow \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2$

```
{ E.true = newlabel(); E.false = newlabel();
  S1.next = S2.next = S.next;
  S.code = E.code +
           "E.true :" + S1.code +
           "goto S.next" +
           "E.false :" + S2.code; }
```

$S \rightarrow \textbf{while } E \textbf{ do } S_1$

```
{ E.true = newlabel(); E.false = S.next;
  S.begin = newlabel(); S1.next = S.begin;
  S.code = "S.begin :" E.code +
           "E.true :" S1.code +
           "goto S.begin"; }
```

Control flow + Lazy evaluation

$E \rightarrow E_1$ **or** E_2	`{ E1.true = E.true; E1.false = newlabel();`
	`E2.true = E.true; E2.false = E.false;`
	`E.code = E1.code + "E1.false :" E2.code; }`

$E \rightarrow E_1$ **and** E_2	`{ E1.true = newlabel(); E1.false = E.false;`
	`E2.true = E.true; E2.false = E.false;`
	`E.code = E1.code + "E1.true :" E2.code; }`

$E \rightarrow$ **not** E_1	`{ E1.true = E.false; E1.false = E.true;` `E.code = E1.code; }`
$E \rightarrow (E_1)$	`{ E1.true = E.true; E1.false = E.false;` `E.code = E1.code; }`

$E \rightarrow$ **id**$_1$ **relop id**$_2$	`{ E.code =`
	`"if id1.place relop.op id2.place goto E.true" +`
	`"goto E.false"; }`

$E \rightarrow$ **true**	`{ E.code = "goto E.true"; }`
$E \rightarrow$ **false**	`{ E.code = "goto E.false"; }`

Control flow + Lazy evaluation

Example:

```
while a < b do
  if i < N and c > d then
     i = i + 1
  else
     i = i - 1
```

Backpatching

Motivation: two passes required to replace symbolic addresses (labels) in jump instructions by actual addresses

Idea:

- all (forward) jump statements that have the same target are put on a list
- when the target address is known, fill in actual address for each statement on list

Section 8.6

Attributes:
$E.tlist$ – all jumps (conditional / unconditional) to $E.true$
$E.flist,\ S.nlist$ – analogous

Backpatching

$E \rightarrow \mathbf{true}$	`{ E.tlist = makelist(next); gen("goto -"); }`
$E \rightarrow \mathbf{false}$	`{ E.flist = makelist(next); gen("goto -"); }`

$E \rightarrow \mathbf{id}_1\ \mathbf{relop}\ \mathbf{id}_2$	`{ E.tlist = makelist(next);`
	`E.flist = makelist(next+1);`
	`gen("if id1.place relop.op id2.place goto -");`
	`gen("goto -"); }`

$E \rightarrow E_1\ \mathbf{or}\ M\ E_2$	`{ backpatch(E1.flist, M.quad);`
	`E.tlist = merge(E1.tlist, E2.tlist);`
	`E.flist = E2.flist; }`

$E \rightarrow E_1\ \mathbf{and}\ M\ E_2$	`{ backpatch(E1.tlist, M.quad);`
	`E.tlist = E2.tlist;`
	`E.flist = merge(E1.flist, E2.flist); }`

$E \rightarrow\ \mathbf{not}\ E_1$	`{ E.tlist = E1.flist; E.flist = E1.tlist; }`
$E \rightarrow (E_1)$	`{ E.tlist = E1.tlist; E.flist = E1.flist; }`
$M \rightarrow \varepsilon$	`{ M.quad = next; }`

Backpatching

$S \rightarrow$ **if** E **then** $M\ S_1$

```
{ backpatch(E.tlist, M.quad);
  S.nlist = merge(E.flist,S1.nlist); }
```

$S \rightarrow$ **if** E **then** $M_1\ S_1\ N$ **else** $M_2\ S_2$

```
{ backpatch(E.tlist, M1.quad);
  backpatch(E.flist, M2.quad);
  S.nlist = merge(S1.nlist,S2.nlist,
                  N.nlist); }
```

$S \rightarrow$ **while** $M_1\ E$ **do** $M_2\ S_1$

```
{ backpatch(S1.nlist, M1.quad);
  backpatch(E.tlist, M2.quad);
  S.nlist = E.flist;
  gen("goto M1.quad"); }
```

$\Leftarrow$

Backpatching

$N \rightarrow \epsilon$	`{ N.nlist = makelist(next);` `gen("goto -"); }`
$S \rightarrow \textbf{begin}\ L\ \textbf{end}$	`{ S.nlist = L.nlist; }`
$S \rightarrow A$	`{ S.nlist = NULL; }`
$L \rightarrow L_1\ ;\ M\ S$	`{ backpatch(L1.nlist, M.quad);` `L.nlist = S.nlist; }`
$L \rightarrow S$	`{ L.nlist = S.nlist; }`

Procedure calls

$S \rightarrow$ **call id** $(Elist)$
```
{ n = length(Elist.q);
  for each x on Elist.q
      gen("param x");
  gen("call id.place, n"); }
```

$Elist \rightarrow Elist_1 \ , \ E$
```
{ Elist.q = enqueue(Elist1.q,
                     E.place); }
```

$Elist \rightarrow E$
```
{ Elist.q = makequeue(E.place); }
```

Declarations

Aim: process declarations in a block to lay out storage for variables
(storage layout $\equiv$ determining the starting address (offset) of each variable within the data area)

Section 8.2

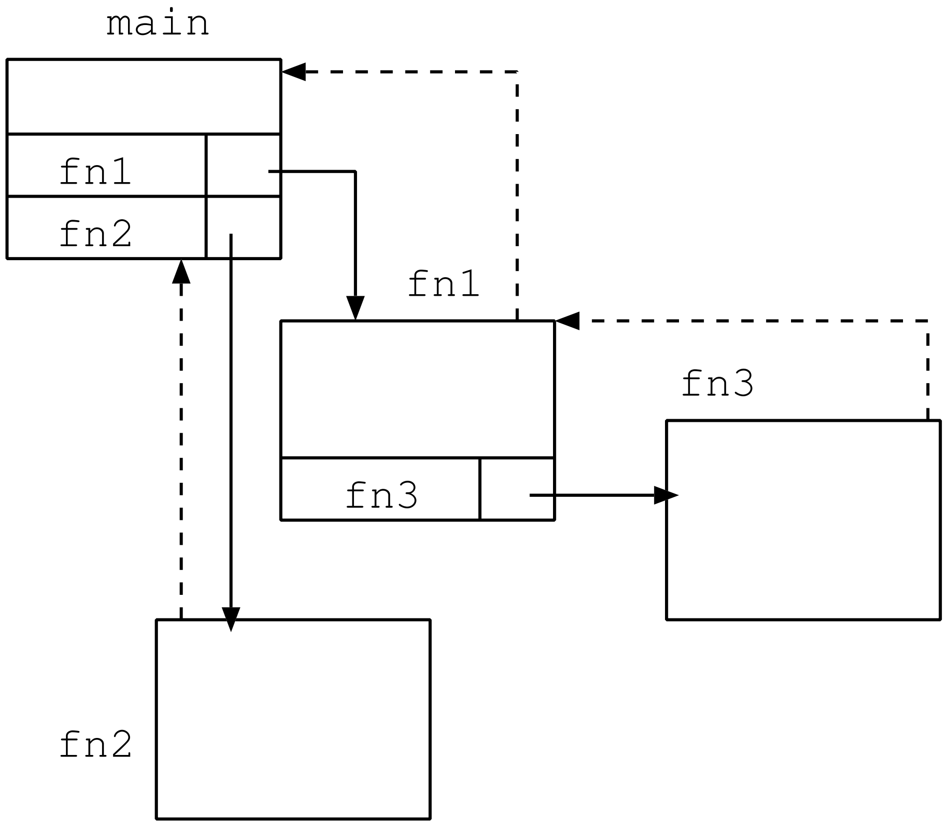
Production	Semantic action
$S \rightarrow Dlist$	`{ offset = 0; }`
$Dlist \rightarrow Dlist\ D \mid D$	
$D \rightarrow T\ L;$	`{ L.type = T.type; L.size = T.size; }`
$L \rightarrow L_1\ ,\ \mathbf{id}$	`{ enter(id.name,L.type,offset); L1.type = L.type; L1.size = L.size; offset += L.size; }`
$L \rightarrow \mathbf{id}$	`{ enter(id.name,L.type,offset); offset += L.size; }`
$T \rightarrow \mathbf{int}$	`{ T.type = INT; T.size = 4; }`

Nested procedures

```
int main()
{
  ...
  void fn1()
  {
    float fn3()
    { ... }
    ...
  }

  int fn2()
  { ... }

  ...
}
```



Declarations

$P \rightarrow M\ Dlist$

$M \rightarrow \varepsilon$
```
{ t = mktable(NULL);
  push(t,tstack); push(0,offset); }
```

$D \rightarrow$ **proc id** $N\ Dlist\ S$
```
{ t = top(tstack);
  setsize(t,top(offset));
  pop(tstack); pop(offset);
  enterproc(top(tstack),id.name,t); }
```

$N \rightarrow \varepsilon$
```
{ t = mktable(top(tstack));
  push(t,tstack); push(0,offset); }
```

$L \rightarrow L_1$, **id**
```
{ enter(top(tstack),id.name,L.type,
        top(offset));
  L1.type = L.type; L1.size = L.size;
  top(offset) += L.size; }
```

$L \rightarrow$ **id**
```
{ enter(top(tstack),id.name,L.type,
        top(offset));
  top(offset) += L.size; }
```