04/14/2025. Quiz 7.

Ch4. Intermediate Representation.

* project. * symbol table. int i, j;
double d;
i = j+3;
type checking i = d + 2.0;

* AST.

4.1. introduction.

IR: collection of data structures representing the facts of the program that compiler discovers.

IR selection: trade off between efficient access and expressive power

4.2. classication of IR.

1. structure organization.

control flow.

graphical IRs (4.3) < Graphs eg CFG / trees e.g. AST

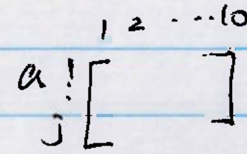
linear IRs. (4.4)

Hybrid IR. { linear IR for each block. / grahical IR for control flow among block

2. level of abstraction.

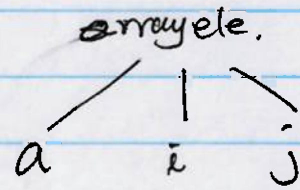
near source form.

lower level form



e.g. a[i, j]

// compute. $a + 4[(i-1)*10 + j-1]$

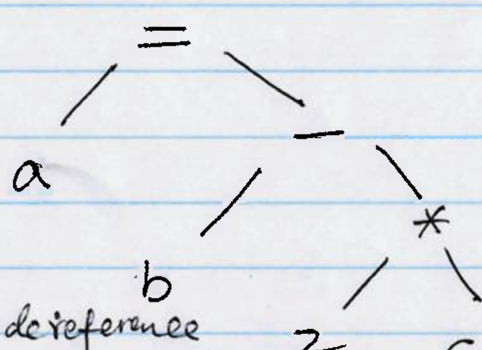


vs.

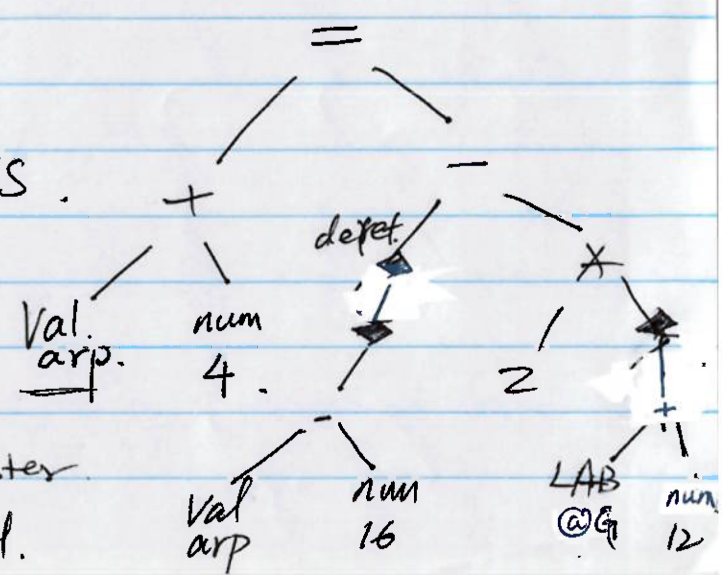
```
subI  ri, 1   => r1
mulI  r1, 10  => r2
subI  rj, 1   => r3
add   r2, r3  => r4
multI r4, 4   => r5
load  @a      => r6
add   r5, r6  => r7
load  r7      => r8
```

ex: $a \leftarrow b - 2 * c$

AST.



VS.



dereference.

Val node: val. already in register.

Num node: known const

LAB node: assembly level label.

3. naming

namespace of IR .

$a := b - 2 * c$

more names. help optimaton

too many names. can block . some of the datastru,

higher level linear code.

*	2	c	$\underline{t}$
−	b	$\underline{t}$	a

lower level linear code.

$t_0 \leftarrow rarp - 16$

$t_1 \leftarrow * t_0$

$t_2 \leftarrow * t_1$

$t_3 \leftarrow @G$.

$t_4 \leftarrow t_3 + 12$.

$t_5 \leftarrow * t_4$

$t_6 \leftarrow t_5 \times 2$

$t_7 \leftarrow t_2 - t_6$

$t_8 \leftarrow rarp + 4$

$* t_8 \leftarrow t_7$

4.5 symbol table

names discovered by the parser.

scalar variable : name. type. size. storage location.

functions: name, type for each parameter.
type for ret. vale.
function entry. point.

aggregate varrable: layout of member. property.
relative location within structure

compiler uses a set of symbol tables to represent different kinds of info. about different type of names.

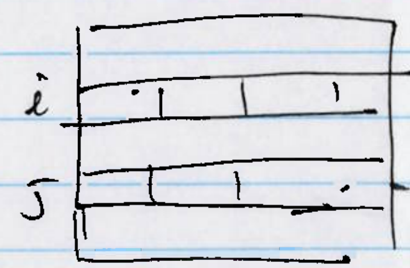
symbol table has two components.

① map from textual name to an index of a repository

name resolution.

② repository of name's property.

```
int i;
int j;
i = j + 1
```



1. name resolution.

name ⟶ unique index of the symbol table.

scope: the region of a program where a given name can be accessed.

04/16/2025

{ }   block scope

procedure also defines a new scope

```
void f(int x)
{
}
```

Scope can nest

lexical scope

```
{ int n, m;
    {
      int n;
      n = m+1;
    }
  n = n+1
}
```

outer.

inner most scope

Inheritane hierachies.

```
parent { int a;
         void f( );
       }

child { int b

        void g( ) { a = a * 2; }
```

2. table implementation.

* storage. should be contiguous to reduce allocation / access cost

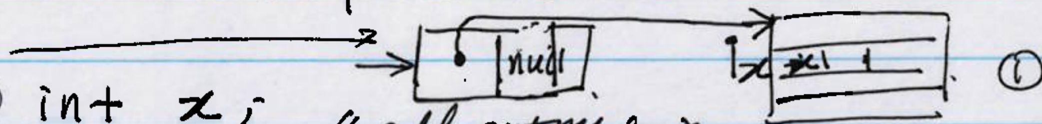
scalability

support chang to the search path

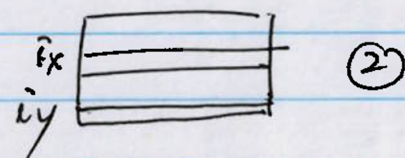
scoped symbol table.

- one table for each scope. names defined in the scope go into the scope table.
- search path links the table together.
- on block entry: creat a new table for the scope. add it to the front of the search path.

- on block exit: disconnect the table for the scope. discard it.



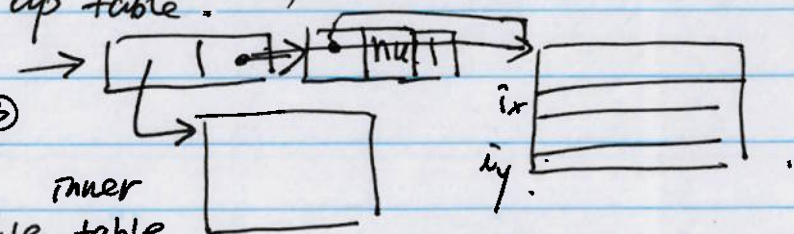
① int x; // add entry for x



② double y; // add entry for y

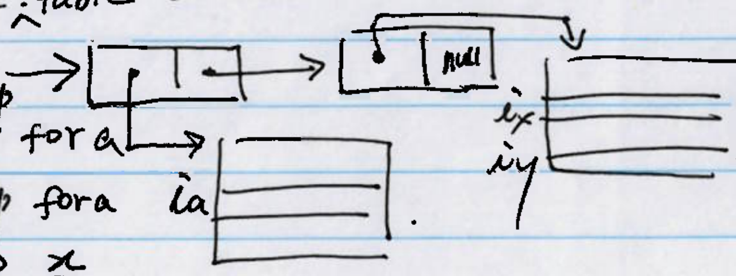
x = 2. //.look up table.

y = 5.0 //



③ {

int a; // update inner table

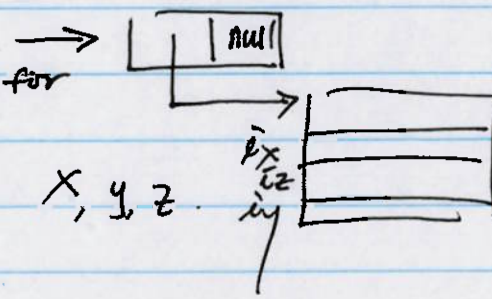


④ a = 1; // loopup table! for a

x = x+a; lookup for a

lookup x.

}



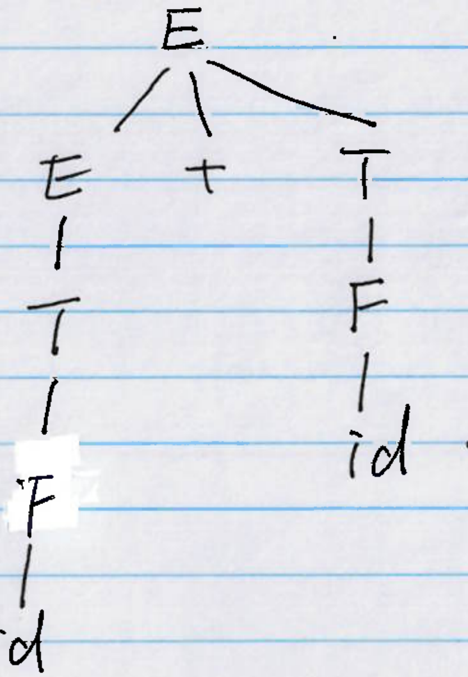
⑤ double z; // update for z

z = y + x; // lookup x, y, z.

4.3.

1. tyee. parse tree.

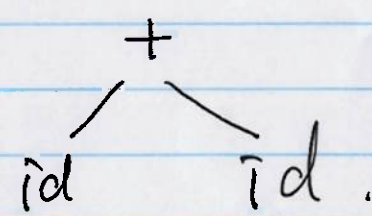
id+id.



AST: abstract syntax tree.

remove extranuous node from the parse.

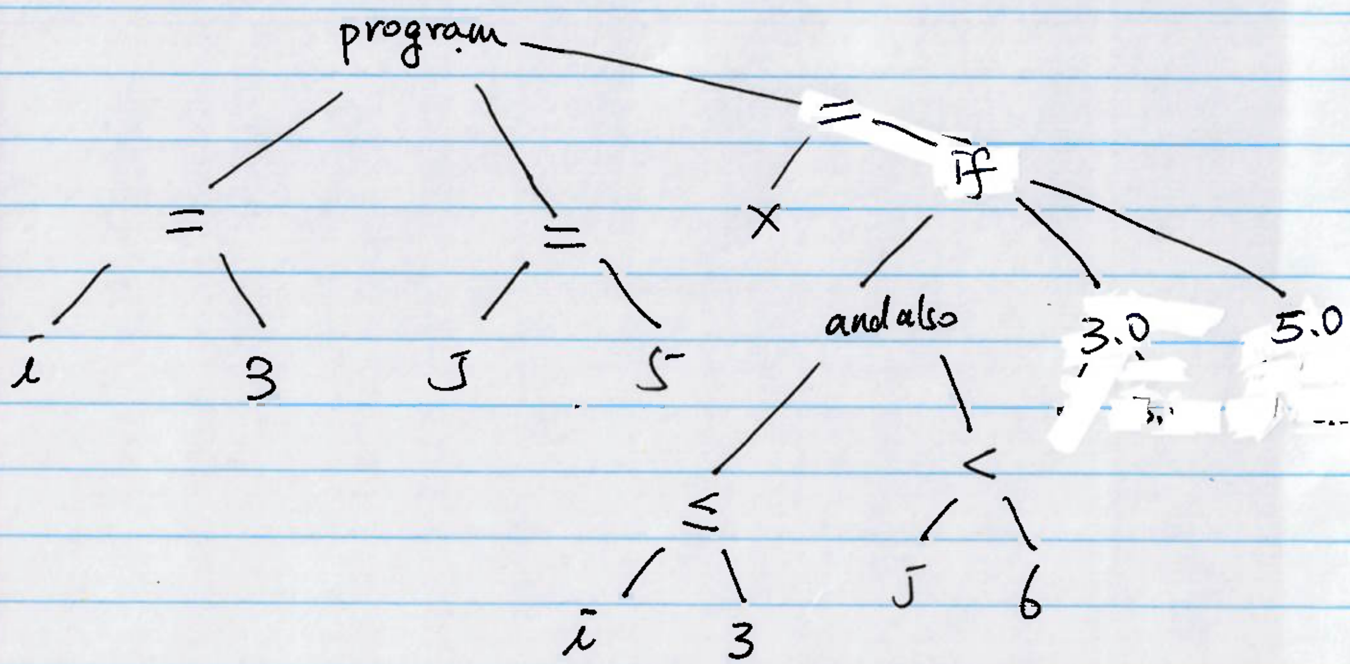
id + id



```
int i, j;
i = 3;
j = 5;
double x;
```

assume

x= if (i ≤ 3 andalso j < 6) then 3.0 else 5.0



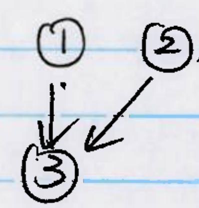
2. *control flow graph

*depence graph.

① $a+b \rightarrow t_1$

② $c-d \rightarrow t_2$

③ $t_1 * t_2 \rightarrow t_3$



call graph.

4.4. linear IR

ordered series of op. including jump and conditional branching op.

* one-address code } popular
* three-address code } popular

two-address code.

1. one-address code: Stack machine code.

$a - 2 * b$

push b // copy b from mem. to stack

push 2

multiply // pop 2 operands from stack
// multiply
// push result onto stack

push a

subtract //

* compact code. implicit name space
  eliminate many names from IR.

JVM : instruction set.

byte code : name derives from its limited size. many operations use only a single byte.

(2) three-address code.

$a - 2 * b$

high level.

$2 \rightarrow t_1$

$b \rightarrow t_2$

$t_1 * t_2 \rightarrow t_3$

$a \rightarrow t_4$

$t_4 - t_3 \rightarrow t_3$

lower level. (ILOC). assembly code for a simple abstract machine with unlimited # of register

LoadI $c \Rightarrow r_x$	$c \rightarrow r_x$
Load $r_x \Rightarrow r_y$	$mem(r_x) \rightarrow r_y$
LoadAI $r_x, c_y \Rightarrow r_z$	$mem(r_x + c_y) \rightarrow r_z$
LoadAO $r_x, r_y \Rightarrow r_z$	$mem(r_x + r_y) \rightarrow r_z$

store $r_x \Rightarrow r_y$. $r_x \rightarrow Mem(r_y)$

storeAI $r_x \Rightarrow r_y, c_z$ $r_x \rightarrow mem(r_y + c_z)$.

storeAO $r_x \rightarrow r_y, r_z$ $r_x \rightarrow mem(r_y + r_z)$.

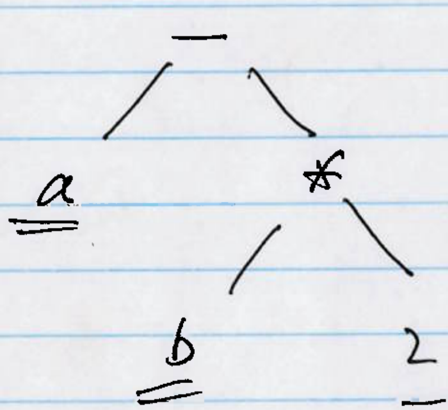
addI $r_x, c_y \Rightarrow r_z$ $r_x + c_y \rightarrow r_z$

add $r_x, r_y \Rightarrow r_z$ $r_x + r_y \rightarrow r_z$.

⋮

* . reasonable compact
* modern RISC. machines use 3-address code
* separates name for operands and result.
  give complile ~~room~~ freedom for optimization.
* . support a wide range of operations.

a − b*2 . ILOC



LoadI r_{arp}, @a $\Rightarrow r_0$

LoadI r_{arp}, @b $\Rightarrow r_1$

multi r_2, 2 $\Rightarrow r_2$

sub $r_0, r_2 \Rightarrow r_3$

4.6. name space — skipped.

4.7. placement of values in memory.

1. memory modes..

   ①. memory-to-memory.

   values are stored in memory.

   either [a.] IR. support mem-to-mem op.

   or. [b.] op. move active value into register.

   inactive value go back to memory.

   $a+b \Rightarrow c$

   [a.]. add. @a, @b ⇒ @c.

   [b]. Load @a ⇒ Vra.

   Load @b ⇒ Vra.

   add Vra, Vrb ⇒ Vrc

   store Vrc ⇒ @c.

② register-to-register.

whenever possible. (unambiguous) values are stored in virtual register.

$a + b \Rightarrow c$. add $Vra, Vrb \Rightarrow Vrc$.

ambiguity

$x = \&y$
$y{+}{+}$
$(*x){+}{+}$

value should be in memory.

③. stack model.

value have their primary home in memory

compiler. moves value onto/off stack

$a+b \Rightarrow c$.

push @a
push @b
add
pop @c.

2. assign values to <u>data areas</u>.