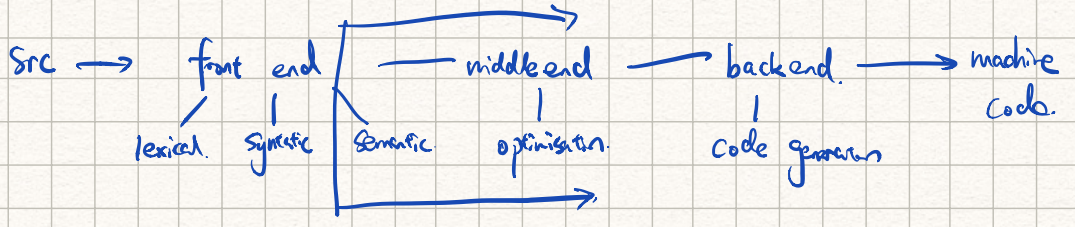
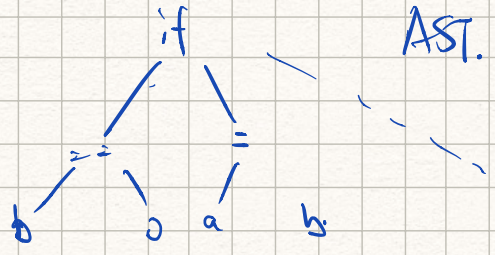
Compilers Back-End Notes

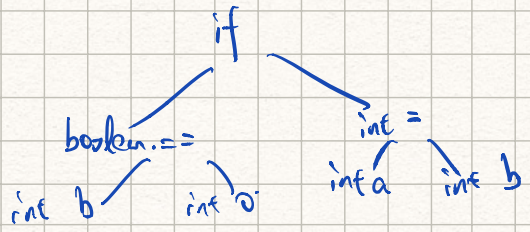


From front-end, we are here.



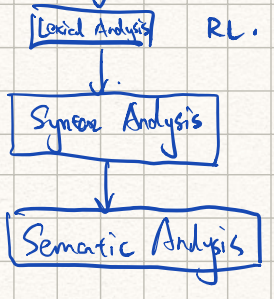
AST.

↓ Semantic analysis



decorated AST.

The Language Hierarchy



All syntactically correct programs are CFG.
Compilable programs are not.

CFG

→ regardless of the context of a nonterminal
Scope/ types are irrelevant to the parser.

Context-Sensitive Grammar.

$G = (N, \Sigma, P, S)$ ← start symbol

N: nonterm, Σ: terminals, P: production rule,

is context sensitive if all $p \in P$ are.

$$\alpha A \beta \to \alpha \gamma \beta$$

$A \in N$ $\quad \alpha, \beta \in (N \cup \Sigma)^*$ $\quad \gamma \in (N \cup \Sigma)^+$

Limitations of CFGs.

We can add context in CFG.

but very Complicated!

If we need grammar we need. Type 0 grammar.

Unrestricted — Turn.

Context (Semantic) Analysis

Outside of grammar check.

* undefined symbols
* type check errors
* scope errors.
* classes and method ambiguities
* Inheritance.

— Look at Scoping Rules / Typing Rules.

Scoping Rules.

Declaration.

- explicit var i : integer;
- implicit any variable start with I is a integer.

Maybe independent Src name declarations.

Scope rules determine which declaration applies in the context.

Binding

an assc name ⟷ things

```
boolen foo (int n, m) {
    tmp;
    if (n<m) tmp=true
      else ---false
    return tmp
}.
```

foo |→ int × int. → bool.

n |→ int.

m |→ int.

tmp |→ bool.

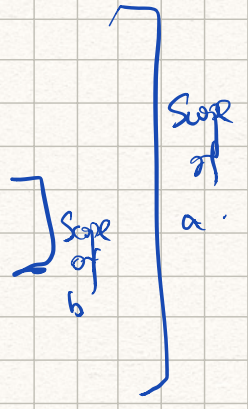
Scope. textual region. where a binding is active.

Static Scope.

Dynamic Scope.

Scope of vars.

```
{ int a)
    { int b)
    }
}
```



Scope of arguments of func.

```
public int facorid (int n) {. Scope of n
    ---
}.
```

OOP makes things + Complex.

Class defines a scope. that covers those of cls. methods.

```
class A {
    private x
    public void g();
}
class B extends A {
    public void h() { g(); }
}
```

scope of x.

Scope of B

Static Scope.

Common choice.

- lexical scope. → Scoping info in the src.
- Scope determined at compile time.
- typically, most recent encounter. like following src.

simplest design. — one global scope.

two scope. — global / local to a function

nested function / more complications

Hole and Qualifier.

* a binding hidden by a same-name nested declaration.

Hole.

```
var x = 2.
if x > 0
    {
        int x = 5   } hidden (Hole).
        x = 1.
    }
```

* Some language still allows access to outer / more global var by applying qualifier or Scope operator

```
int cout = 42;
std::cout << cout << endl;
```

Qualifier — function / not int.

Dynamic Scope.

binding depends on execution flow.

* whatever declaration recently take place
* binding expires when execution leaves lexical scope

In general, flow of execution is not predictable at compile time.

Dynamic scoping now implemented in interpreted languages

Dynamic Scoping

* can be dangerous
* easier to implement.

Referencing environment.

* scoping rules → where to bind.
* Referencing Environment → when to bind.

↓

use it (seq. of scopes to search to find the current binding of a given name)

Shallow deep binding.

(dynamic). Shallow binding happens when the function is actually called.

deep binding happens when the function is first referred to

Most use deep binding.

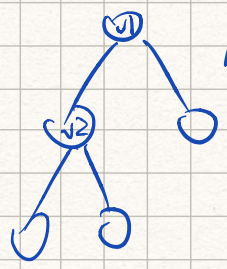
Implementation.

Symbol Tables

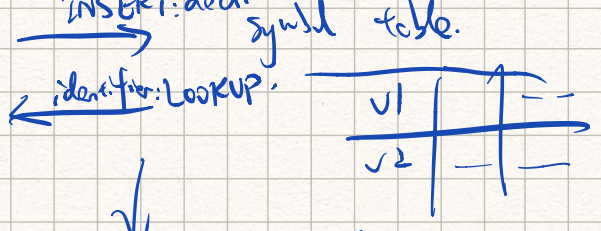
focus on static Scoping (deep binding rules)

Example.

foo method [int] x [int] → bool.
m arg. int.
n arg int
tmp var bool

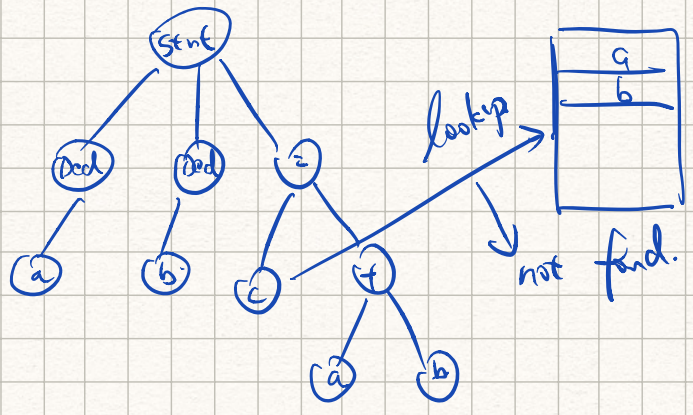


AST



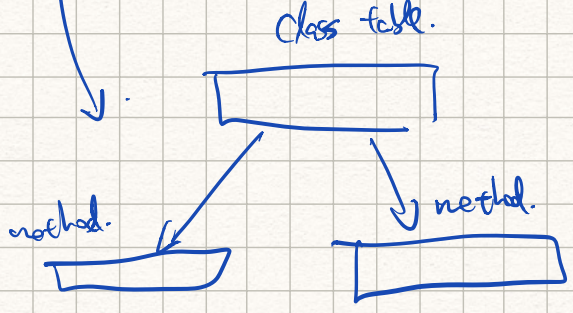
if cannot find id. in Symbol table. "unidentified id."

Example: check for unideclared identifiers



Single table is not sufficient.

- Nested scopes — inner binding hide outer bindings
- Forward references. — use when not declared. (e.g. funcs)



find which declaration is active.

* start from current.
* go up until find it.
* don't go downwards.

The hierarchy solves the name collision.

found in deeper tables.

Forward Reference

* construct tree hierarchy of tables first.
* then, traverse the Symbol table tree.

Implementation.

Two ways: * one table, add a colum with scope information.

* multiple tables: stack/ tree of tables

Symbol Table Op.

- Create a new empty symbol table with a given parent table.
- Insert a new id to a table.

- Lookup

Implementation: Individual Tables

* Linked list.
* BST. - good balance.
* Hashtable. - good hashing

Type checking.

e.g. has x been used according to its type
are parameters passed in of expected types

Type System: Define type expressions and rules

↓ ↓

all possible types constraints on types

Type expressions.

Basic types (primitive types)

int, double, boolean.

Build types from basic types using.

- type constructs.

(array — int[] int[10]
struct.
pointer.

- function types.

* class types

OOP

class A {
int i; public float f;
}.

struct {
int a;
float b;
}.

int * x.

Type. $T_1 \times T_2 \times T \cdots \times T_n \to T_r$.

foo: int x double → double.

Example Type Expressions.

public A foo (int i, B[]b, double[] d.)

foo : i:int, b: array(B), d: array(double) → A.

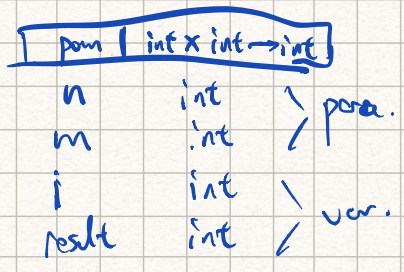
public class A {
int [10]a;
boolean b;
C[] c;
}.

{ a: array(int, 10), b: boolean.
c: array(C)}.

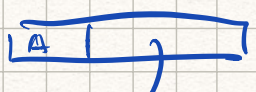
Type Info in Symbol table.

name ⟷ type.
parameter ⟷ type.
method ⟷ para, result.
class ⟷ instance var / method decl.

```
int pow (int n, int m) {
    int i=0
    int result=1;
    {
    |
    }.
```



class A.



private void m1 - -		public?
public double m2 i -	int	FALSE
j	int	TRUE
m1	void → void	FALSE
m2	int × double → double	TRUE

Type rules

int a;

int a; → int

double b; → double.

int c;

c = a+b → (a+b) → : int + double
= double + double.
= double.

for each assign stmt. LHS must be same type.
c.
of the expr on RHS
a+b

Comparison. should have same type.
> == <

result is boolean.

int n = pow(i+j, j)

type of (pow(i+j, j)) = int x int
→ int.

matches type of (n) = int.

Implementation.

Construct Symbol table.
* traverse AST.
* store type info in the table.

Scope and type checks
* After. STable built., visit to AST.
and access to
Symbol table.
* Ensure scope/type. are respected.

If no forward referencing, 1 single visit is enough

Error Handling.

when error found → report error and
continue.

also may need to create right type.
if not went a cascading errors

Semantic Analyser.

1. process decl.
   add. to STable.
   report multiple decl.
2. process stmt.
   undeclared var.
   update guesses of ID.
   update ID nodes of AST to STable.
3. Go through all stmt. again.
   check for type errors.

Intermediate Representation.

LLVM.

High Level IR.

extract orgind src intent easier.

Low Lvl. IR.

generate machine code easier

IR should.

independent of source or target lang.
convenient to produce during semtic analysis
- translate to multiple targets

AST → IR → Machine Code

instruction for a VM

abstract

GIMPLE / LLVM / Java Bytecode

Ideal for Optimisation.

AST about 40 different node types

IR a dozen instructions

much easier.

IR for 2 types of machines.

Register Machines

2+3

Load R1 2.
Load R2 3
ADD R1, R2
↓
5 goes to R.

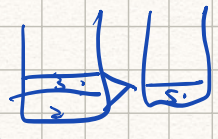
Identifier based temporary vars.

TAC (Three address Code)

Stack Machines

2+3

PUSH 2
PUSH 3
ADD



pop 2
pop 3
5 → result
push result

JVM.

Performed on stack.

data variety:

Data: Store vals. / referenced by addr or ID

Constant pool

Instruction: Code.

Register Machine.

TAC. has at most three addr (or less).

$a = (b+c)*(-d)$

$t1 = b+c \quad t2 = -d \quad a = t1 * t2.$

Type of TAC.

Binary a = b OP c.

Arithmetic: ADD, SUB, DIV, MUL, MOD
Logical : AND, OR, XOR.
Comparisons: EQ, NEQ, LE, LEQ - - -

Unary a OP c.
MINUS, NEG

COPY : a = b.

Array access; a=b[i] a[i]=b

Field access.: a=b.f b.f=a.

Control flow:

labelling label L.

Unconditional jmp. jump L.

Conditional jump tjump a L.
fjump a L

Function calls

param pi

call f

a= call f

return

return y.

Temporary Vars

operands can be.

vars.

consts.

temp vars

use tmp vars to store intermediate values

TAC Example.

```
n=0
while (n<10) {
  n=n+1
}
```

```
n=0.
label L
  t2= n LE 10
  fjump t2 END
  n= n ADD 1.
  jump L.
label END
```

Translation to TAC.

Need an algo to do so.

* Start from AST
* define rules for each AST node type
* Recursively apply templates

Given e.

T[e] is a translated low level IR.

t=T[e] means t stores the result of low level IR e.

t=T[e] $\Leftrightarrow$ t=e.

Unary operators.

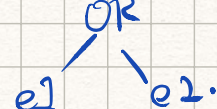
t=T[e] t= OP t1.

Binary Ops t1=T[e1].

t2=T[e2]

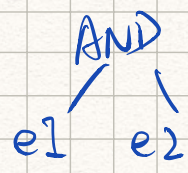
t= t1 OP t2.

Short-circuit OR.



t =T[e1]
fjump t END ← if e1 true must be
t =T[e2]

short-circuit AND.




```
t = T[e1]
tjump t NEXT
jump END
label NEXT
  t = T[e2]
label END
```

← if e1 true then evaluate e2 otherwise must false.

Array access

$t = T[v[e]]$

```
t] = T[e]
t = v[t]
```

Field access

$t = T[e1.e2]$

```
t] = T[e1]
t2 = T[e2]
t = t].t2.
```

List of statements

$T[s1; s2; \dots]$

```
T[s1]
T[s2]
```

Concat of IR instructions

Assignments

$T[v = e]$.

```
v = T[e].
```

$T[v[e1] = e2]$

```
t] = T[e1]
t2 = T[e2].
v[t] = t2.
```

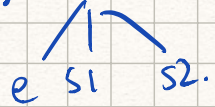
Control Flow.

If-then.

$T[\text{if}(e) \text{ then } s]$ <=>

```
t] = T(e)
fjump t] END
T[s].
END
```

If then else.



T[if(e) then s1 else s2]

⇔ t1 = T(e)

fjump t1 ELSE

T[s1]

jump END

ELSE

T[s2]

END

Function calls

T[[call f(e1, e2 .. en)]]

param T[e1]

param T[e2]

⋮

param T[en]

call f.

return

t = T[e]

return t.

Example.

```
if (c) {
    if (d) {
        a = b
    }
}
print(x)
```

T[- - -]

↓

T[if(c) ...]

T[print(x)]

t1 = T(c)

fjump t1 END

T[if(d) ...]

label END

param x.

call print

↓

⋮

TAC Exercise.

```
t1 = n==0
if n==0 return 0
else { n2=1. if n=1
       return 1.
     else {
          t3 = n-1
          BAR(n-1)
          t4 = n-2.
          BAR(n-2)
          return. BAR(n-1)+(n-2)
```

calc Fibonacci Seq.

n is de input

Implementing TAC.

* Many ops don't use arg 2.

* jups put labels in result.

Triples : results are primarily used for temps

Problem:

Optimisation often moves code around.

break triples (indexes no longer preserved)

* Indirect Triples

addr	inst
45	(20)
46	(21)
47	(22)
⋮	⋮

+ b [20].

* quadruples

* easier to optimise.

* redundet temporaries

* triples

* moving stat. chage references.

* Indirect triples.

* moving stmt chage. the referencing.
no change to triples

* save space if temp. is reused.

Stack Machines

* Operands results in the stack
* Register allocation not a issue
* ASM IR Java ByteCode

Java Bytecode. (Const. pooled at. beging of class)

* load and store.
* arithmetic and logic.
* type conversion.
* object creation and manipulation.
* operand stack mngment.
* Control flow transfer.
* method invocation and return.
* exception/ sync and etc.

Load/store.
a: reference
i: integer
l: long

Pros: Stack-based IR

* compact code. 1 param.

e.g.: iload #5

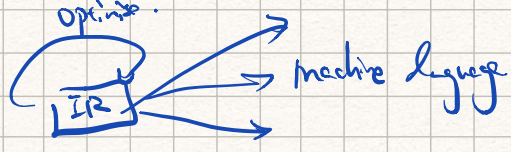
* simpler compiler / code gen is simpler. independent from other code.
* simpler interpreter. / mem access centralised less variants of instructions

Cons:

* more mem references (no tmp registers)
* common subexpression are hard to factor out.
* less flexible (hard to move around) when optimising.

Optimisation. outline.

* High level lvl. optimisation.
* Time-Space Trade off.
* Specific Optimisation Techniques
* Peephole-vs-Super optimisation.



+ persone semantics
+ faster.
+ small footprint.
+ little Space

Semantic preserving changes improves time/space characteristics

optimisation → improvement.
only for "avg. expected cases".

Lvl of Optimisation.

High lvl. src → AST.

Intermediate lvl. AST → IR.

Low lvl. IR → machine code.

Different lvl gives different possibilities.

Lvl of Optimisation

High lvl.
* type info.

Intermediate lvl.
* control flow / data flow analysis

Low lvl.
* Target code selection
* Register allocation

Time vs. Space.

Small is fast as well.

current data small L1 cache.

Common Techniques

Common subexpression

a1 := b2 * 42 + c + d.
a2 := b2 + 42 + c + d.

Algebric Identities ! caveat.

Computer arithmetic does not follow mathematics (floating point)

Dead code elimination.

* code that will not be executed.
* will never have any effect
* Analyse on control and data flow

Constant Folding

evaluate sub expression at compile time.
reduce total number of instructions

Propagation.

can be propagated through multiple variables

```
x = 10          10
y = x + 1       ↓ 11
x = x + 2       ↓ 12.
```

Constant Folding : Caveat.

Be careful about semantics / precisions

Compile time precision should match run time precision.

Strength reduction. use cheaper operators than expensive ones

Comparisons
add / subtract / bit ops
mull.
divise / mod.

faster
Int
float
slower.

e.g. $5 * x \Rightarrow x << 2 + x$.

$x^2 \Rightarrow x * x$.

common tricks

```
i = 10.
while (i > 0) {
    i = i - 1;
    t = 4 * i;
    a[i] = b[t]
}.
```

```
i = 10
t = 4 * 10
while (i > 0) {
    i -= 1
    t -= 4
    a[i] = b[t]
}.
```

Inlining

avoid function call by copy and pasting function in when called.

Caveats! : should be small funcs.
careful with recursion.

Loop optimisation.

* unrolling
  - repeat the loop if ops are known
    less jmps but grows code size.

* Fusion / Fission

fusion ⇒ put loops together fewer jmps

```
for (int i|~100)
    a[i]=i
    b[i]=i+1;
}.
```

fission ⇒ split loops up. multicore take a loop each

```
for (int i |~100)
    a[i]=i

for (  ——— )
    b[i]= i+1.
```

Code Motion. (Move loop independent code out of it)

Pro: less instructions cont.

Cons: result in register spill.

Tiling (Blocking).

Create smaller loop around data to prevent register spill.

(Cache for time).

```
for (i=0 to i<N i++)
    {
```

```
for (j=0  j<N  j+=B)
  for (i=j, i<min(N, j+B); i++) {
    ...
  }
```

Principle of Locality

* Temporal locality - reuse of data/resources within small time interval.
* Spatial locality - use. ——— within close storage location.
* seq - - - use - ——— in a linear order.

Inversion.

convert a while loop into a do-while loop

reduces goto jumps.

if --- initial check

```
do
  {
    ...
  }
while ( )
```

← change. / no need to jump back and check

check here.

Interchange.

change inner/outer loop for better locality.

```
for (int i ---)        for (j ———)
  for (int j ——).  ⇒     for (i ——)
```

Unswitching.

Take if-branching out by taking if outside of loops

Peephole Optimisation.

* only works on small portions of code.
* use better machine - - -

* keep scanning until no more optimisations.
* one peephole may bring another.

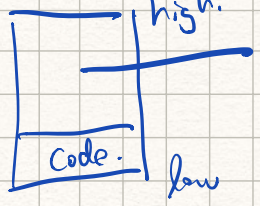
Find Result dependent on order applied.
best order is hard to find.

Super-optimisation.
exhaustive search over the space of all possible programs
to find the shortest program
→ used to find performance-critical inner loop optimisations.
→ used to find peephole optimisations

Runtime Organisation.
Machine Code is a set of instructions understandable by CPUs
ASM: Readable low level.
Corresponds 1-1 machine code
Easily transferrable by assembler.

Runtime organisation.
* manage runtime resources
* correspondance between Static (compile) – dynamic (run-time) structure
* storage organisation.

OS runs the program.
allocates the memory



1. Procedure Activation record.
2. Local data.
3. Global data.
4. Dynamic allocated data

An invocation of P is an activation of P

lifetime of an activation.
* all the steps to execute P
steps in procedure P calls

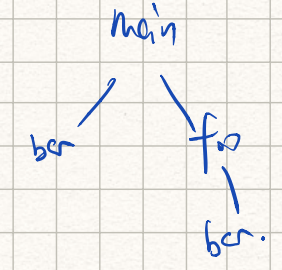
Similarly
lifetime of var.
is the portion of execution in which x is defined (dynamic)

Activation Tree.

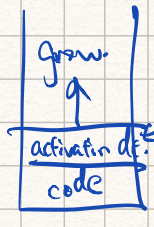
P → Q. Q should return before P
calls

lifetime of activation should be nested.
↓
tree.

```
foo() { bar() }
bar() {       }
main() {
    bar()
    foo()
}
```



Activation tree dependent on runtime behavior.
may be different for diff. inputs.
can use a stack to track the active procedures



We need activation record/call frame. to execute correctly

F()
G() ← call G.
F paused.
G's AR contains param -- info to execute G.

Goes back to F() with return.

General AR.

* Return / Results
* Params
* Control link.
* Return addr. (where to go back to)
* Temporaries (Tmp values)
* Local vars.
* Access link. (additional data needed by procedure)

Example of AR design.

* Store enough info to execute correctly
* compilers try to hold as much of frames as possible

Global data

Heap structure.

no global var in an AR

fixed addr assigned by compiler.

statically allocated.

← Global data/objects with fixed addr

a object. that outlives the procedure

(cannot use AR)

S.t. Stored in heap structure. (malloc free).

Stack and heap.

both grow.
must make sure not grow into.

if grow into (out of memory)

Memory management.

garbage = value that will not be used anymore.
what is garbage? hard

Solutions

C / C++ style.
programer deal with it.
memory leaks
dangling references
double free bugs free (freed again)

GC.

GC. avoid above issues.
but. cause additional resource / performance.

Code Gen.

Conversion of IR → machine instructions

Input : IR.

Output : absolute machine language.

relocatable machine language. .o file

assembly language. - S.

Code gen = Instruction Selection + scheduling
+ register allocation.
+ debug data generation.

Three Main Architecture.

* Register.
* Stack ← operation between top of stack and other items
* Accumulator. ← where calc occur.

Instruction set.

Software
ISA. ——→ defines valid executions of hardware.
Hardware.

Architecture.

native DT. ——→ RISC. Reduced IS Computer
instructions CISC Complex - --
registers
addr modes
mem. architecture
interrupt & exception.
external I/O

x: y + z

CISC
MOV y R0
ADD z R0
MOV R0, x

RISC
MOV y, R0
MOV z, R1.
ADD R0, R1.
MOV R1, x

IR → Machine Need to know to is of the architecture.

Goal: gen code that stores into regs as much as possible
much quicker than memory.

Regs.

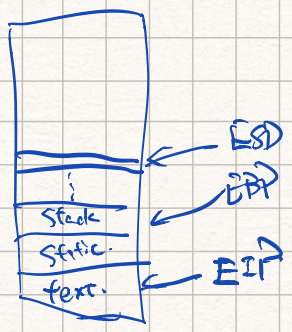
general registers — data
ptr
index.
segment registers
Control registers

X86 Register Simple Model

Data Reg. EAX Accumulator EBX Base.
ECX Count. EDX Data.

Ptr. Reg. Stack Ptr. ESP
Base Ptr. EBP
Instruction Ptr. EIP

Index Reg. Source Index ESI
Dest. Index EDI



X86 ASM Instructions

two operands.
AT&T syntax (GCC/GAS)
Intel syntax.

op src dst

mov $1, %eax

op dst src.

mov %eax $1

X86 ASM Instructions

Arithmetic add sub inc mod idiv imul ---.

at3. movl %edx %eax (data, accumulator)
addl %ecx %eax
addl %ecx %eax

Logic and or not xor.

Data mov : mov. movl $5 (%rax)

Stack pushq. %rax popq %rax.

Comparison. : comp / test

Control : jmp :: jg, jl, jge, jle, je.

DTs.

Suffix

b.	byte	byte 8 bit
s	short	16 bit
w	word	16 bit
l	long	32 bit 4 byte
q	quad	64 bit 8 byte

4 byte offset.

movl 4(%rbp) %rax

pushq %rbp

make 5 a quad.

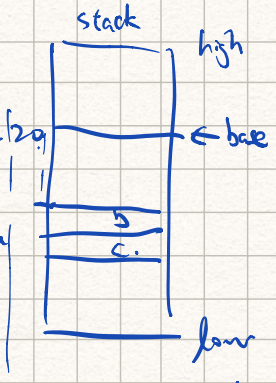
Argument Prefixes

% reg	Register.
offset (%reg)	offset in bytes
$nnn.	immediate.
label.	Label.

$a = (2+b) * c$.

```
movl  -20(%rbp)  %eax   [2]
addl  $2         %eax.
imull -24(%rbp), %eax
```

↑ integer multiplication



why don't we simply addl. offset(p-locator). offset(q--). ✗

need a register to write to

```
movl  16(%ebp)  %eax
addl  8(%ebp)   %eax
```

Function calls. : call ret.

push argument to the stack. → invoke the function.

function looks for args on stack.

return in a register.

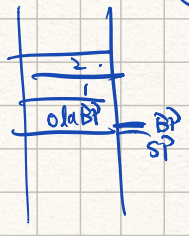
Caller pops the arg. off stack

```
foo (b, c) { callee
  int a=4
  a = (a+b)*c
  return a
}

main () {
  foo(1, 2);
}
```

Caller

```
push $2
push $1
call foo        foo(1,2)
```




```
pushq %rbp
movq  %rsp  %rbp
push  $4
movl  8(%rbp), %eax
      ⋮
movq. %rsp %rbp
popq  %rbp
```

ret .

Access Static Global Data.

Static variables not in stack.

have fixed addresses throughout.

directly refer. using Symbolic names

```
int g=5;
foo { int a=1+g;
```

```
. globl -g.
. data.
-g: . long 5
movl. -g % edx.
addl. $1, %edx.
```

Simple Code Gen.

Basic Block.

Seq. of consecutive statements where flow ↓ ↑ without branching

TAC of x=y+z → defs x reference y & z

1. determine leaders
   - R1. : First statement is a leader.
   - R2 : target of a jump is a leader.
   - R3 : immediately follows is a leader
2. for a leader a basic block is all cmds to the next

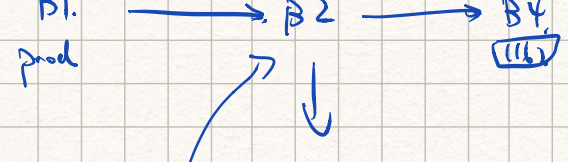
Example.

```
prod=0
i=1.
```

```
t1=i LEQ 20
t2=NOT t1.
cjmp t2 (b)
```

```
t3=i
t4=a[t3]
```

Flow Graphs. Rep. of basic blocks



```
t5 = i
t6 = b[t5]
t7 = t4 * t6
t8 = prod + t7
prod = t8
t9 = i + 1
i = t9
jmp (3)
```

```
(  B3
   t3
   |
   jmp B2.
```

Next use analysis.

Generate code for a basic block.

* For each in a block. want to know if the value will be used again.
  * Yes. keep the register (now is live)
  * No deallocate. — dead

We don't know how we arrived

cannot assume var in registers (reload them!)

don't know — go.

store them in memory!

Next Use Algorithm: Computing next Uses

Init assgn.:
all non temp are live
all temp are dead.

Scan over from beginning → end.
end → beginning

Suppose we find a TAC (i) x = y op z during back scan.

attach (i) the info abt nextuse of x y z.

x = not live.

y z = live nextuse = (i).

Simple Code Gen algo.

from TAC.

Inside each block. one stmt → machine code.

register descriptor (what's in a register)
new register needed.

address descriptor (where is a name)
decide access method.

$d = (a-b) + (a-c) + (a-c)$

```
t1 = a - b
t2 = a - c
t3 = t1 + t2
d = t3 + t2.
```

```
MOV a R0      SUB b R0
MOV a R1      SUB c R1.
ADD R0 R1.
ADD R1 R0
MOV R0 d.
```

R0 → $eax
R1 → $ebx

a = -4$(rbp)
b = -8$(rbp).
c = -16$(rbp)
d = -20$(rbp).