Assembly

$D(R_b, R_i, S)$ = Mem[Reg[Rb] + S*Reg[Ri] + D]

movl	src, dest	dest = src
cmpl	src2, src1	cc src1 ● src2
jmp	label	
je	label	==
jne	label	≠
js	label	−
jns	label	+
jg	label	signed >
jge	label	signed ≥
jl	label	signed <
jle	label	signed ≤
ja	label	unsigned >
jb	label	unsigned <
pushl	src	%rsp = %rsp − 4
popl	dest	%rsp = %rsp + 4
call	label	push addr next, jmp
ret		pop rip, rsp +4
leal	src, dest	dest = addr of src
incl	dest	dest = dest + 1
decl	dest	dest = dest − 1
addl	src, dest	dest = dest + src
subl	src, dest	dest = dest − src
imull	src, dest	dest = dest * src
xorl	src, dest	dest = dest ^ src
orl	src, dest	dest = dest | src
andl	src, dest	dest = dest & src
negl	dest	dest = − dest
notl	dest	dest = ~ dest
sall	k, dest	dest = dest << k
sarl	k, dest	dest = dest >> k arith
shrl	k, dest	dest = dest >> k log

Analysis / Optimizer:

- CfgBuilder
- AvailableExpression DFA
- Backward DFA
- ConstantPropagation DFA
- Forward DFA
- LiveVariable DFA
- ReachingDefinition DFA
- AvailableExpression Opt
- BaseOpt
- ConstantExpression Opt
- ConstantPropagation Opt
- ForkOpt
- PreCalculateOperators Opt
- ReachinDefinition Opt
- RemoveUnused Opt
- RemoveUnusedVar Opt

Parser (after/during) Dynamic Undecidable
?-Program exec.

Compiler:

- All var/methods/symbol defined P
- All var initialized before use ?
- Program has no syntax error P
- Defined program start P
- Exceptions are caught ?
- No null-ptr access D
- No out-of-bound access for arrays, records/instances D
- Program type checks P
- No unused variable/fields P
- Return stmt reachable ?
- Execution reaches end of program ?
- Program terminates U
- All functions return value P
- No access to private/protected members unless legal P
- No cycle in inheritance graph P
- Interfaces/abstract functions implemented P
- Loops terminate U
- Program executes within a given time budget? ?/U
- Program stays within power budget ?/U
- All values computed can be represented ?

Frontend

- CurrentContext (in wich class+method?; correctReturn)
- SemanticAnalyzer (circular inheritance?; creates symbols)
- SemanticChecker (checks semantic failures)
- TypeManager (keeps track of type inheritance; assignable?)
- SymbolTable (store scopes for local, param, field lookup)

IR:

- ExprVisitor
- BasicBlock
- AstRewriteVisitor
- ControlFlowGraph
- DominatorTree Algorithm

Backend:

- AssemblyEmitter
- AstCodeGenerator
- ExprGenerator
- StmtGenerator
- RegisterManager
- VTable

Syscalls are link between I/O and Reg (OS & prog.)

Code (Flexibility, Reusability, Readability)

Correct program: behaves the same way
Incorrect program: behaves in a defined way

Constant propagation:

T

1. start with estimate for each stmt's output claim
2. determine each stmt's input claim from cf
3. apply transfer fct for all stmt's
4. continue with 2. until no more changes to output

Reaching definitions:

- $kill_B = \{d \mid d$ is killed in $B\}$
- $gen_B = \{d \mid d$ appears in B and no subseq. stmt in B kills
- $OUT[ENTRY] = \emptyset$
- init $OUT[B] = \emptyset \ \forall B \neq ENTRY$
- while (changes to any OUT(B)){
  for (B ≠ Entry) {
  $IN(B) = \bigcup_{B_i, B_i \text{ is predecessor of } B \text{ in cfg}} OUT(B_i)$
  $OUT(B) = gen_B \cup (IN(B) - kill_B)$ }}

Live Variables:

- $def_B = \{var \mid var$ is def. in B prior to any use of var in B$\}$
- $use_B = \{var \mid var$ is used in B prior to any defs of var in $B\}$
- $IN[EXIT] = \emptyset$
- init $IN[B] = \emptyset \ \forall B \neq EXIT$
- while (changes to any IN(B)) {
  for (B ≠ EXIT) {
  $OUT(B) = \bigcup_{B_i, B_i \text{ is successor of } B \text{ in cfg}} IN(B_i)$;
  $IN(B) = use_B \cup (OUT(B) - def_B)$ }}

Def-use chain: For each use of a var incl. in the list all defs that reach this use.

Use-def chain: For each def of a var incl. in the list all uses of this var.

Available expressions:

- $gen_B = \{expr \mid expr$ a+b is eval. in B, a nor b subseq. def in B$\}$
- $kill_B = \{expr \mid$ a or b defined in B, a+b not subseq. def in B$\}$
- U = set of all expr that appear in program
- $OUT[ENTRY] = \emptyset$
- init $OUT[B] = U \ \forall B \neq ENTRY$
- while (changes to any OUT(B)) {
  for (B ≠ ENTRY) {
  $IN(B) = \bigcap_{B_i, B_i \text{ is predecessor of } B \text{ in cfg}} OUT(B_i)$;
  $OUT(B) = gen_B \cup (IN(B) - kill_B)$ }}

Register allocation / Live ranges:

- Compute liveness information
- Compute reaching definitions
- Set L_p: virtual registers live at point P
- Graph coloring for detecting min amount of regs

DEP: Data Execution Protection
ASLR: Address Space Layout Randomization
Stack canary: Corruption of local stack only if canary val not overwritten
Tiered compilation: Combined benefits of interpreter, C1 & C2

Context-Free: $A \to xy$ ✓ $xA \to y$ ✗

Ambiguous / Mehrdeutig: Multiple Left-Most derivation

V Table: → V Table addr → super V Table → ...

V Table addr	super V Table
field 1	metod 1
field 2	method 2
⋮	⋮

Non-Context-Free: $\{a^n b^m c^n d^m\}$ - Matching parameters in Methods

Branch-less-conditionals: Compiler predicts a branch and takes it until realization

Design pattern:
- Control flow is always difficult to deal with.
- Raise lvl of programming (hide details, easier to read, simpler to write, easier to maintain/extend/understand)

Visitor:
- Coupling of data structure and access methods
- Powerful in connection with trees
- Object must be prepared to accept a visitor
- Good for i.e. arithmetic expressions
- Traverses the object structure
- Beneficial if: object structure contains objects with different (unrelated) functionality or object structure changes rarely but new operations are added often

Parsing Table (LL): Input (Terminal) Symbol

Non-Terminal	a	b	\$
S	$S \to AB$	Error	Error
A	$A \to \varepsilon$	Error	Error
B	Error	$B \to b$	Error
\$	Error	Error	Accept

Parsing Table (LSR):

	Action				Goto		
State	b	c	d	\$	S	X	Y
0	s1	-	-	-	1	-	-
1	-	-	r2	-	-	2	-
2	-	-	-	Acc	-	-	3
⋮	⋱						⋮

Stack	Input	Action
\$ <0>	bbbdc\$	Shift
\$ <0>b, <2>	bbdc \$	Reduce
		Acc

Regular Grammar: Left-Linear Xor Right-Linear

- If $[A \to \alpha \cdot a\beta] \in I_j$, $a \in T$ and $goto(I_j, a) = I_k$, $action(I_j, a) =$ shift a, new State $= I_k$
- If $[A \to \alpha \cdot] \in I_j$ and $t \in Follow(A)$, $action(I_j, t) =$ reduce $A \to \alpha$
- If $[S' \to S \cdot] \in I_j$, $action(I_j, \$) = accept$
- rest = error