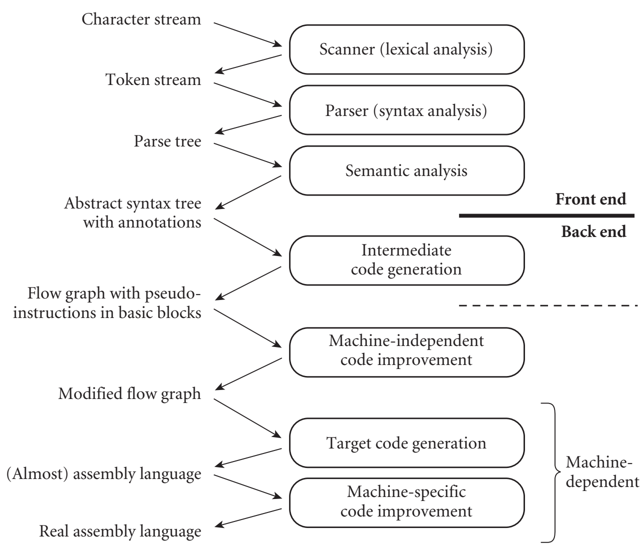
SI 413: Computers are good at running instructions. Not at reading your mind. – Donald Knuth

Professor Keith Sullivan

Back-end Structure



Intermediate Forms

- Intermediate form (IF) provides the connection between phases
- Classified based on levels:
  - **High level** IF based on trees or DAGs
  - **Medium level** IF: three address instructions for idealized machine with unlimited registers
  - **Low level** IF resemememble assembly
- Most compilers use a combination of IFs

3AC or TAC

Typicall of the following form

destination_addr = source_addr1 operation source_addr2

- Similar to assembly, but is machine independent
- Like assembly, no if/then block or looping

Python example

```
p = 2
while p*p <= n:
   if n % p == 0:
      break
   p += 1

if p*p > n:
   p = n

print(p)
```

```
p = 2
condition:
  temp = p * p
  check = temp <= n
  br check loop afterloop
loop:
  temp = n % p
  check = temp == 0
  br check afterloop update
update:
  p = p + 1
  br condition
afterloop:
  temp = p * p
  check = n < temp
  br check if2 afterif
if2:
  p = n
  br afterif
afterif:
  # Note: this is one plausible way a library
  # call could work, assuming some special
  # variable names arg1, arg2, etc.
  arg1 = "%d\n"
  arg2 = p
  call printf
```

Stack Based IF

- Stack based languages when brevity and simplicity are paramount
  - Embedded systems and printers
  - Postscript and PDF
- Medium level: pass code from a compiler to an interpreter/virtual machine
  - Java bytecode and CLI
- Difficult to optimize

Types

- Most programming languages have a notion of type
- Provide:
  - Implicit context
  - Limit the set of operations in a semantically valid program
  - Make code easier to read and understand
  - Can drive performance optimizations

Define Type

- Denotational: a set of values.
- Structural: either a built-in type, or composite type
- Abstraction: an interface consisting of a set of operations with well-defined and mutually consistent semantics

Typing

- Type checking
- Strongly typed: if the language the application of any operation to an object that is not intended to support the operation
- Statically typed: type information is known at compile time
- Dynamically typed: type information is checked at run-time

Classification of Types

- Most languages provide common built-in types
- Numeric types: typically implementation dependent
- Enumerations
- Subrange types
- Composite types

Type Equivalence

- Structural equivalence: content of the two type definitions
- Name equivalence: lexical occurrence of type definitions
  - Strict name equivalence: aliased types are distinct
  - Loose name equivalence: aliased types are equivalent

Type Casts

Converting type casts:

1. Types are structurally equivalent, but the language uses name equivalance
2. Types have different sets of values, but the intersecting ones are represented the same way
3. Types have different low-level representation but we can define some sort of correspondence between them

Nonconverting type casts: a change of type that doesn't alter the underlying bits

Inverse Square Root

- Alias argument to an integer as a way to approximate $\log_2 x$
- Use this approximation to compute an approximation of $\log_2 \frac{1}{\sqrt{x}} = -\frac{1}{2} \log_2 x$
- Alias back to float to compute an approximation of base-2 exponential
- Refine approximation with a single iteration of Newton's method

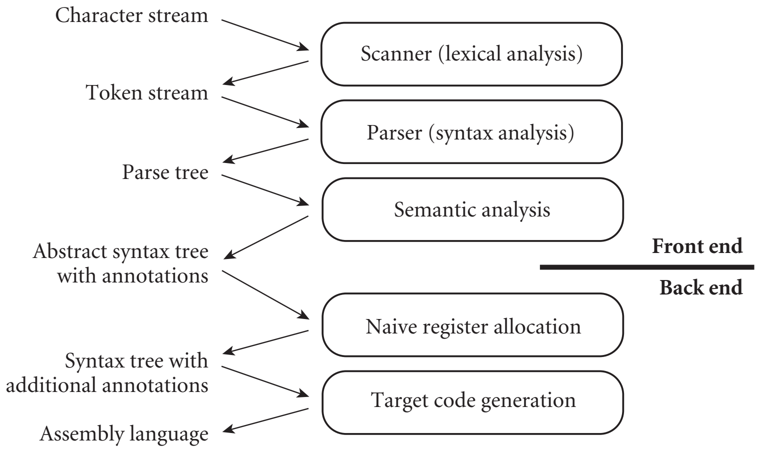
Nonconverting type example

```
float Q_rsqrt( float number )
{
  long i;
  float x2, y;
  const float threehalfs = 1.5F;

  x2 = number * 0.5F;
  y  = number;
  // evil floating point bit level hacking
  i  = * ( long * ) &y;

  i  = 0x5f3759df - ( i >> 1 ); // what the fuck?

  y  = * ( float * ) &i;

  // 1st iteration
  y  = y * ( threehalfs - ( x2 * y * y ) );

  return y;
}
```

Heron's Formula Example

push a	r2 := a	
push b	r3 := b	
push c	r4 := c	
add	r1 := r2 + r3	
add	r1 := r1 + r4	
push 2	r1 := r1 / 2	-- s
divide		
pop s		
push s		
push s	r2 := r1 − r2	-- s − a
push a		
subtract		
push s	r3 := r1 − r3	-- s − b
push b		
subtract		
push s	r4 := r1 − r4	-- s − c
push c		
subtract		
multiply	r3 := r3 × r4	
multiply	r2 := r2 × r3	
multiply	r1 := r1 × r2	
push sqrt	call sqrt	
call		

Code Generation

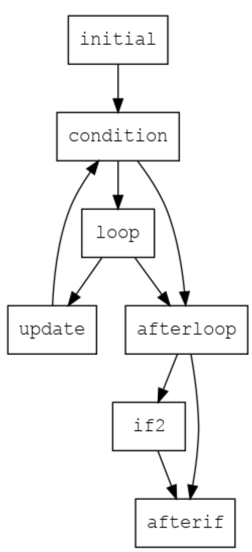


Basic Blocks and CFG

- Most IRs make it easier to analyze the control flow
- Basic blocks: sequential chunks of statements without branches
  - Start with a label
  - End with a control flow statement
- Control flow graph: possible execution paths between basic blocks

Example

```
initial:
  p = 2
  br condition
condition:
  temp = p * p
  check = temp <= n
  br check loop afterloop
loop:
  temp = n % p
  check = temp == 0
  br check afterloop update
update:
  p = p + 1
  br condition
afterloop:
  temp = p * p
  check = n < temp
  br check if2 afterif
if2:
  p = n
  br afterif
afterif:
  # Note: this is one plausible way a library
  # call could work, assuming some special
  # variable names arg1, arg2, etc.
  arg1 = "%d\n"
  arg2 = p
  call printf
```



Static Single Assignment

- Formal definition: A program in in SSA form is every variable in the program is assigned only once.
  - Any name can appear on the left hand side only once
  - Write once, read many
- Restriction on how IF assigns variable names and/or register names
- Makes later compiler optimizations easier
- Recall: referential transparency

SSA example

```
initial:
  p = 2
  br condition
condition:
  temp = p * p
  check = temp <= n
  br check loop afterloop
loop:
  temp = n % p
  check = temp == 0
  br check afterloop update
update:
  p = p + 1
  br condition
afterloop:
  temp = p * p
  check = n < temp
  br check if2 afterif
if2:
  p = n
  br afterif
afterif:
  arg1 = "%d\n"
  arg2 = p
  call printf
```

Typical Correction

```
initial:
  p1 = 2
  br condition
condition:
  t1 = p * p
  t2 = t1 <= n
  br check loop afterloop
loop:
  t3 = n % p
  t4 = temp == 0
  br t4 afterloop update
update:
  p2 = p + 1
  br condition
afterloop:
  t5 = p * p
  t6 = n < t5
  br t6 if2 afterif
if2:
  p3 = n
  br afterif
afterif:
  arg1 = "%d\n"
  arg2 = p
  call printf
```

Actual Solutions

- Use memory: store address on the stack
  - Slow compared to CPU registers
- ϕ-function: used to determine where value in a basic block came from based on CFG

```
p4 = phi(p1, p2)
t1 = p4 * p4      # proper SSA form
```

ϕ-function Example

```
initial:
    p1 = 2
    br condition
condition:
    p4 = phi(p1, p2)
    t1 = p4 * p4
    t2 = t1 <= n
    br t2 loop afterloop
loop:
    t3 = n % p4
    t4 = t3 == 0
    br t4 afterloop update
update:
    p2 = p4 + 1
    br condition
afterloop:
    t5 = p4 * p4
    t6 = n < t5
    br t6 if2 afterif
if2:
    p3 = n
    br afterif
afterif:
    p5 = phi(p3, p4)
    arg1 = "%d\n"
```

How to determine ϕ

If the variable was set earlier in same basic block, just use that value

```
y = 7 * 3
y = y + 1
temp = y < 10
```

becomes

```
y1 = 7 * 3
y2 = y1 + 1
temp = y2 < 10
```

How to determine ϕ

Trace backwards in CFG. If all paths reach the same basic block, use that variable name

```
one:
  x = 5
  x = x - 3
  br x two three
two:
  y = x * 10
  br three
three:
  y = x + 20
```

```
one:
  x1 = 5
  x2 = x1 - 3
  br x2 two three
two:
  y1 = x2 * 10
  br three
three:
  y2 = x2 + 20
```

How to determine ϕ

If both previous cases fail, need a ϕ-function

```
one:
  x = 5
  x = x - 3
  br x two three
two:
  x = x * 10
  br three
three:
  x = x + 20
```

```
one:
  x1 = 5
  x2 = x1 - 3
  br x2 two three
two:
  x3 = x2 * 10
  br three
three:
  x4 = phi(x2, x3)
  y2 = x4 + 20
```

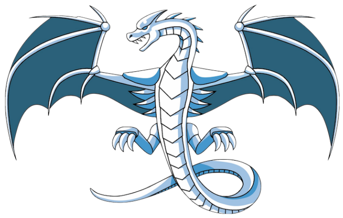
Why do all this?

Easier to perform code optimizations

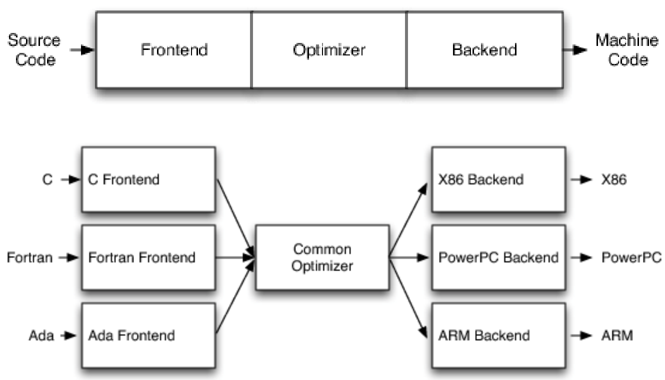
- Constant propagation
- Common subexpression elimination
- Dead code elimination
- Function inlining
- Loop unrolling
- Register allocation

LLVM

- A collection of modular and reusable compiler toolchain tech (e.g., assemblers, compilers, debuggers)
- Started as a research project in 2000
- Unique internal representation
- Clang



Compiler Design



Successes and Issues

Successess

- Java and .NET
- Translate input source into C
- GCC

Issues

- Monolithic

LLVM IR

- Platform independent assembly language with infinite registers
- Strongly typed reduced instruction set computing (RISC) instruction set
- First class
- Contents of LLVM IR assembly file is called a module
- Modules contain zero or more top-level entities such as global variables and functions
- Function declaration contains zero basic blocks; function definitions contains one or more basic blocks

Identifiers

- Global identifiers (functions, global variables) begin with @
- Local identifiers (register names, types) begin with %
- Why prefixes? No name clash with reserved words
- Local identifiers scoped to each function

Example: multiply %X by 8. The easy way:

```
%result = mul i32 %X, 8
```

and the hard way

```
%0 = add i32 %X, %X           ; yields i32:%0
%1 = add i32 %0, %0           ; yields i32:%1
%result = add i32 %1, %1
```

LLVM IR Example

```
int f(int a, int b) {
    return a + 2*b;
}

int main() {
    return f(10, 20);
}
```

```
define i32 @f(i32 %a, i32 %b) {
; <label>:0
    %1 = mul i32 2, %b
    %2 = add i32 %a, %1
    ret i32 %2
}

define i32 @main() {
; <label>:0
    %1 = call i32 @f(i32 10, i32 20)
    ret i32 %1
}
```

LLVM IR Example

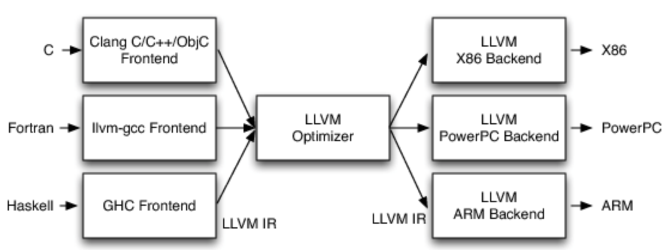
```
; Global variable initialized to the 32-bit integer value 21.
@foo = global i32 21

; f returns 42 if the condition cond is true, and 0 otherwise.
define i32 @f(i1 %cond) {

  entry:
    br i1 %cond, label %block_1, label %block_2

  block_1:
    %tmp = load i32, i32* @foo
    %result = mul i32 %tmp, 2
    ret i32 %result

  block_2:
    ret i32 0
}
```

SSA

- LLVM IR is SSA
- Recall ϕ function

```
define i32 @f(i32 %a) {
; <label>:0
    switch i32 %a, label %default [
        i32 42, label %case1
    ]

case1:
    %x.1 = mul i32 %a, 2
    br label %ret

default:
    %x.2 = mul i32 %a, 3
    br label %ret

ret:
    %x.0 = phi i32 [ %x.2, %default ], [ %x.1, %case1 ]
    ret i32 %x.0
}
```

LLVM Implementation



Attribute Grammars

1. $E_1 \longrightarrow E_2$ `+` T ▷ E$_1$.val := sum(E$_2$.val, T.val)
2. $E_1 \longrightarrow E_2$ `-` T ▷ E$_1$.val := difference(E$_2$.val, T.val)
3. $E \longrightarrow T$ ▷ E.val := T.val
4. $T_1 \longrightarrow T_2$ `*` F ▷ T$_1$.val := product(T$_2$.val, F.val)
5. $T_1 \longrightarrow T_2$ `/` F ▷ T$_1$.val := quotient(T$_2$.val, F.val)
6. $T \longrightarrow F$ ▷ T.val := F.val
7. $F_1 \longrightarrow$ `-` F_2 ▷ F$_1$.val := additive_inverse(F$_2$.val)
8. $F \longrightarrow$ `(` E `)` ▷ F.val := E.val
9. $F \longrightarrow$ `const` ▷ F.val := const.val

Attribute Grammars

reg_names : array [0..$k-1$] of register_name := ["r1", "r2", . . . , "rk"]
  −− ordered set of temporaries

program $\longrightarrow$ *stmt*
  ▷ stmt.next_free_reg := 0
  ▷ program.code := ["main:"] + stmt.code + ["goto exit"]

while : *stmt*$_1$ $\longrightarrow$ *expr stmt*$_2$ *stmt*$_3$
  ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_3$.next_free_reg := stmt$_1$.next_free
  ▷ L1 := new_label(); L2 := new_label()
  stmt$_1$.code := ["goto" L1] + [L2 ":"] + stmt$_2$.code + [L1 ":"] + expr.code
    + ["if" expr.reg "goto" L2] + stmt$_3$.code

if : *stmt*$_1$ $\longrightarrow$ *expr stmt*$_2$ *stmt*$_3$ *stmt*$_4$
  ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_3$.next_free_reg := stmt$_4$.next_free
    stmt$_1$.next_free_reg
  ▷ L1 := new_label(); L2 := new_label()
  stmt$_1$.code := expr.code + ["if" expr.reg "goto" L1] + stmt$_3$.code + ["goto" L2]
    + [L1 ":"] + stmt$_2$.code + [L2 ":"] + stmt$_4$.code

assign : *stmt*$_1$ $\longrightarrow$ *id expr stmt*$_2$
  ▷ expr.next_free_reg := stmt$_2$.next_free_reg := stmt$_1$.next_free_reg
  ▷ stmt$_1$.code := expr.code + [id.stp→name ":=" expr.reg] + stmt$_2$.code

read : *stmt*$_1$ $\longrightarrow$ *id*$_1$ *id*$_2$ *stmt*$_2$

Attribute Grammars

- Two tasks: determine registers for each subtree at runtime, and generate code
- Simple stack to allocate registers

Example: $(a + b) * (c - (d/e))$

r1 = a	push a
r2 = b	push b
r1 = r1 + r2	add
r2 = c	push c
r3 = d	push d
r4 = e	push e
r3 = r3 / r4	divide
r2 = r2 - r3	subtract
r1 = r1 * r2	multiply

Target Code

```
main:
    a1 := &input  -- "input" and "output" are file control blocks
                  -- located in a library, to be found by the linker
    call readint  -- "readint", "writeint", and "writeln" are library subroutines
    i := rv
    a1 := &input
    call readint
    j := rv
    goto L1
L2: r1 := i       -- body of while loop
    r2 := j
    r1 := r1 > r2
    if r1 goto L3
    r1 := j       -- "else" part
    r2 := i
    r1 := r1 − r2
    j := r1
    goto L4
L3: r1 := i       -- "then" part
    r2 := j
    r1 := r1 − r2
    i := r1
L4:
L1: r1 := i       -- test terminating condition
    r2 := j
    r1 := r1 ≠ r2
    if r1 goto L2
    a1 := &output
```

Address Space

- Assemblers, linkers, and loaders typically operate on a pair of related file formats
  - relocatable object code
  - executable object code
- Relocatable object code is acceptable as input to a linker
  - multiple files in this format can be combined to create an executable program
- Executable object code is acceptable as input to a loader:
  - it can be brought into memory and run

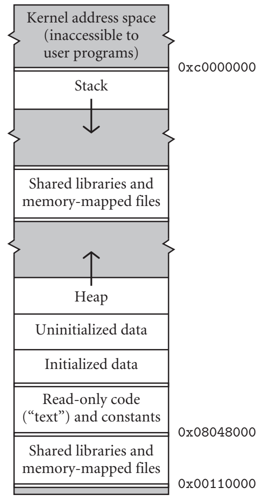
Relocatable Objects

- Import table
  - Instructions that refer to named locations whose addresses are unknown
  - Assume addresses will be in other files
- Relocation table
  - Instructions in the current file but are offset at runtime
- Export table
  - Lists names and addresses in current file that may be referred by other files

Contrast with executable objects: contains no references to external symbols

Memory Layout

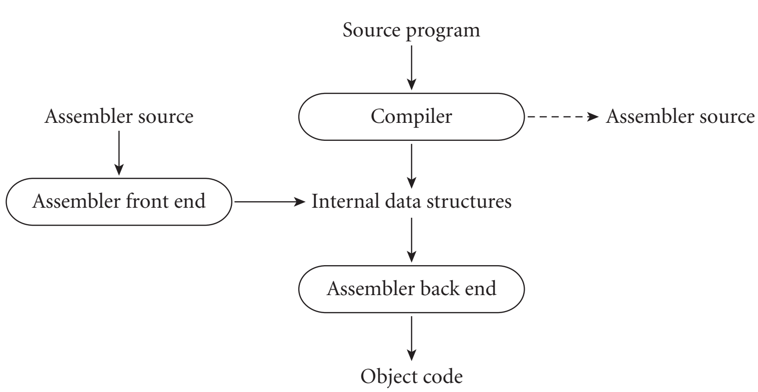
- Unitialized data: allocated at run-time or on demand. Usually zero-filled for repeatability and security
- Stack and heap: small initial allocation, OS then expands as needed
- Files: library that allows mapping of files into memory
- Dynamic libraries: shared code and linkage information



Assembler

- The compiler generates assembly code
- An assembler converts assembly to machine code (object file)
  - Replace opcodes and operands with machine language encodings
  - Replace symbolic names with actual addresses

Assembler



Emitting Instructions

- Basic task: translate symbolic representations to binary form
- Most assemblers make minor modifications to their input
  - `as`: GNU assembler
  - SPI's assembler for MIPS
- Directives: instructions to assembler to take some action or change a setting
  - Assemble code and data into specified sections
  - Reserve space in memory for uninitialized variables
  - Control the appearance of listings
  - Initialize memory
  - Assemble conditional blocks
  - Define global variables
  - Specify libraries from which the assembler can obtain macros
  - Examine symbolic debugging information

Directives

- Segment switching
  - `.text`: place instructions in code segment
  - `.date`: place instructions in initialized data segment
  - `.space n`: reserve n bytes in uninitialized data segment
- Data generation
  - `.byte, .hword, .word, .float` and `.double` place successive instructions in the current segment
  - `.ascii` places a single character in consecutive bytes
- Symbol identification
  - `.globl name` indicates name should be entered into the export table
- Alignment
  - `.align n` aligns subsequent output at an address evenly divisible by 2^n

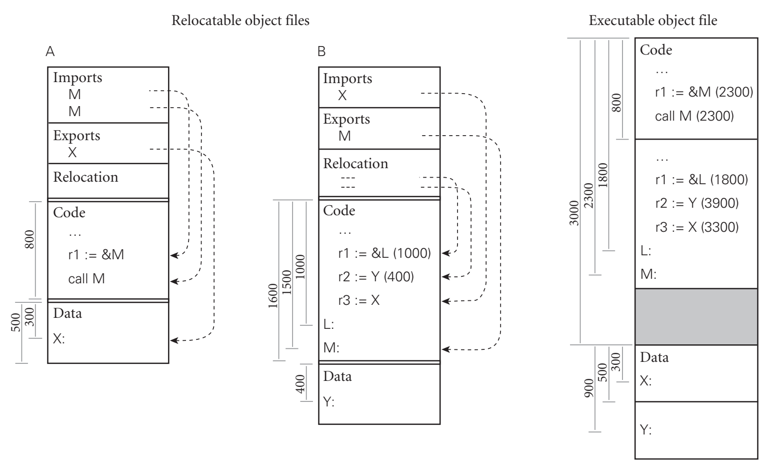
Assigning Address

- Assemblers work in multiple passes
  - Convert text to IR
  - Identify internal and external symbols, assigning address to all internal ones
  - Produce object code
- Within the object file
  - Any symbol in `.globl` goes into the table of exported symbols
  - Any symbol referred to, but not defined, must appear in table of imported symbols
  - Any symbol that depends on its placement in the current file goes in to the relocation table

Linking

- Separate compilation: fragments of the program can be compiled and assembled separately (compilation unit)
- The linker glues these fragments together
  - Each compilation unit must be relocatable
- Static linking: done prior to program execution
- Dynamic linking: done during program execution
- Two tasks: relocation and external symbol resolution
  - Virtual memory from the OS

Linking Example



Type Checking

- Within a compilation unit, semantic rules apply
- Across compilation units, header files are used
- Consider:
  - Module M's header makes promises re API
  - Compiler enforces those promises when compiling M
  - Problems?
- Create symbol to characterize M's header
  - Checksum
  - C and C++ ???
- Name mangling