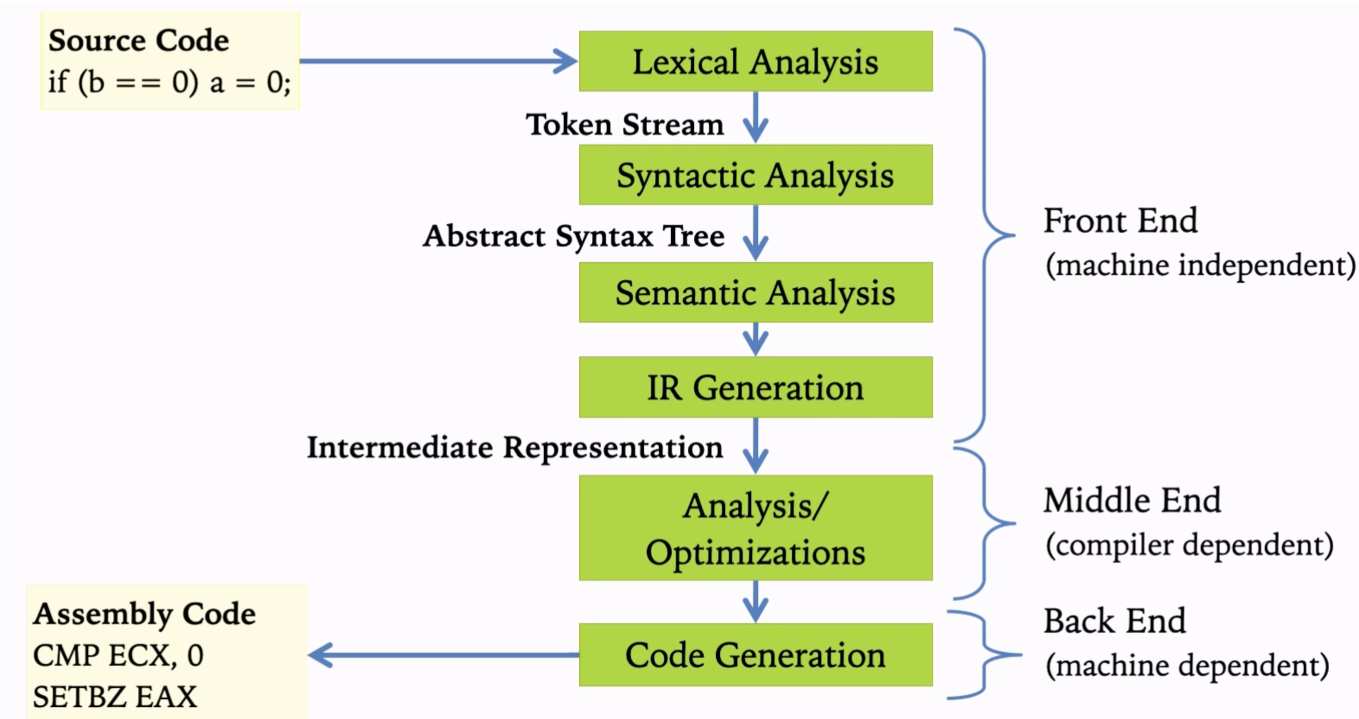
NOTES: Compilers

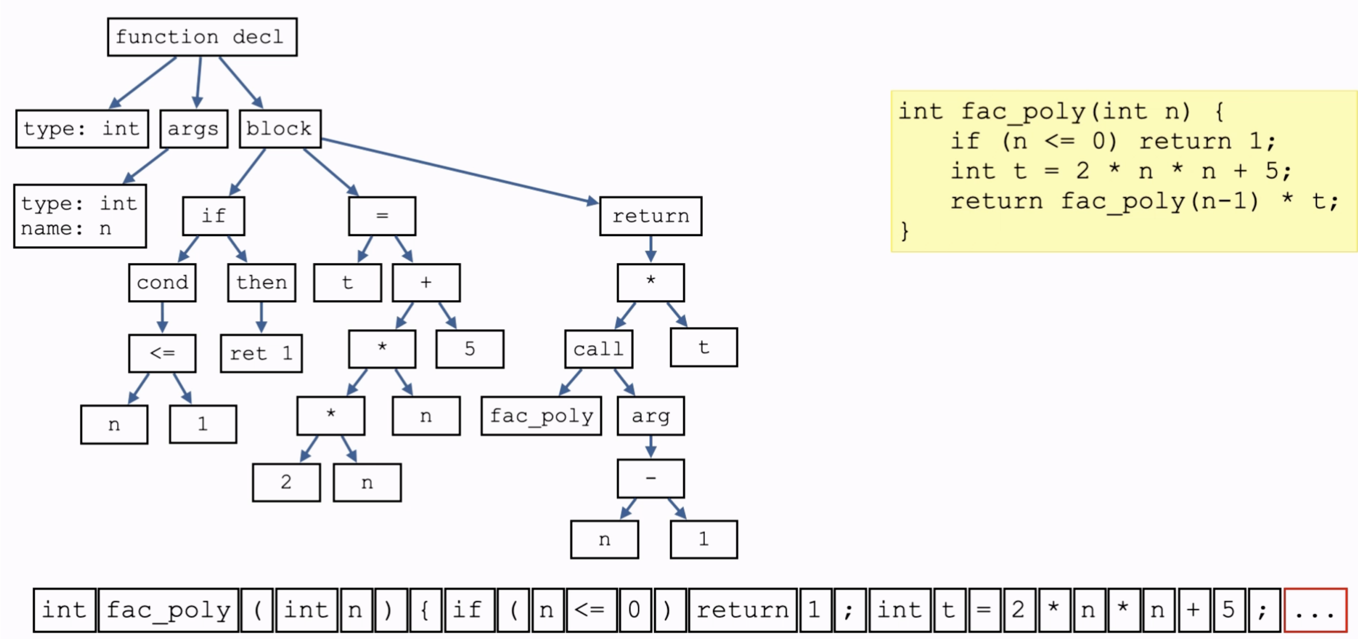
Compilers

- A compiler takes a source code from one language, and outputs the equivalent code in another language. (Usually from higher to lower level)
- Source code is expressive (matches human grammar), redundant (more info than needed), and abstract (not tied to a specific machine).
- For any program and any input, the result of interpreting the program should be the same as executing the compilation in the target langauge.
- "Self-hosted" compilers are compilers that are written in the language they compile.

Compiler Strategy



- Lexer generates token stream: identifies each token (word)
  - Regular expressions are used to identify tokens
- Parser generates abstract syntax tree: identifies the structure of the code ("sentences")

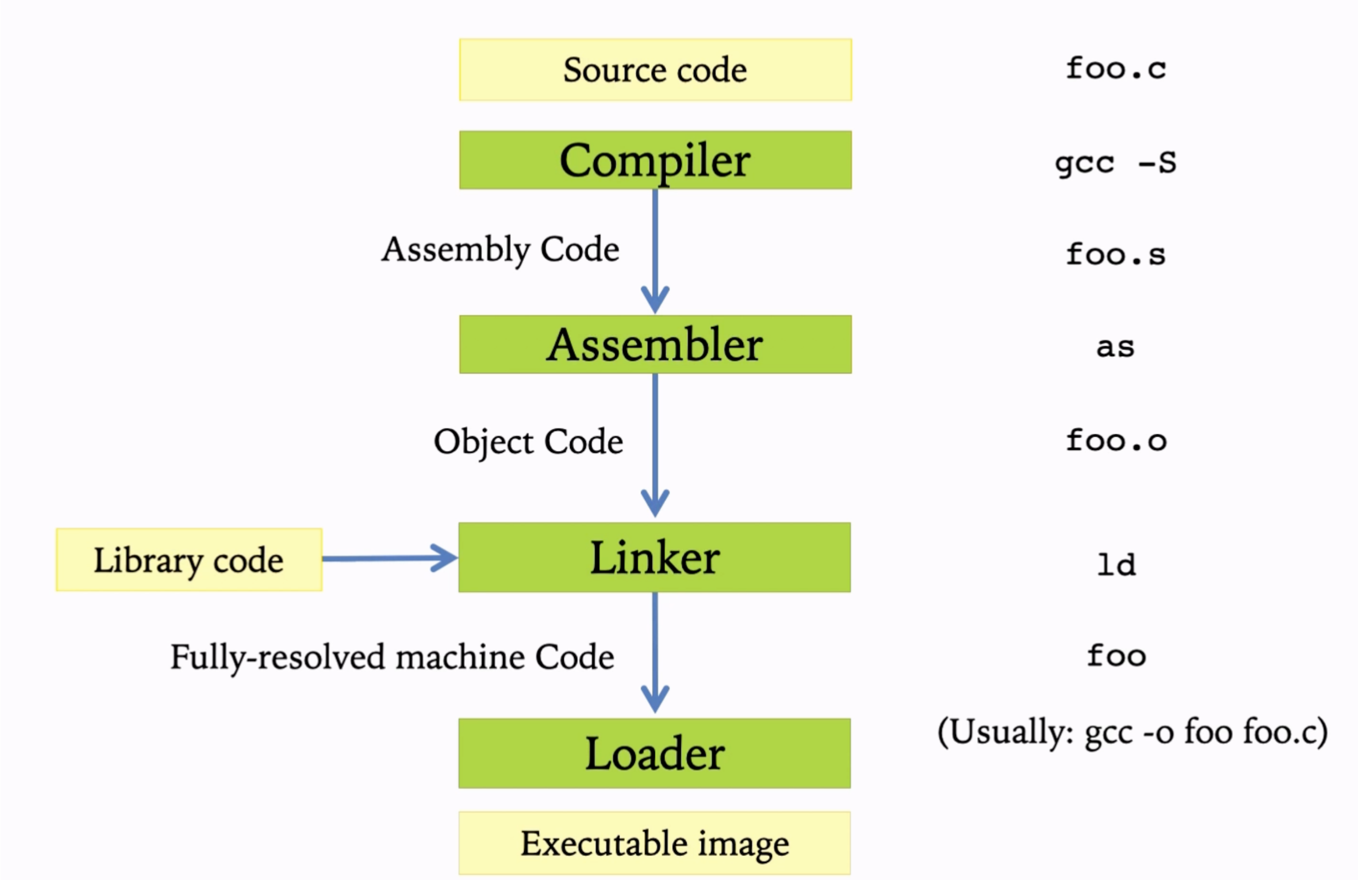


- Checks are performed
  - Scope checking (check if variables are declared)
  - Type checking
- Interprediate representation: analyze and make the code more concise (like IL for Java/C#)
  - Easy translation target, and easy to translate
  - Branches and labels replace loops and conditionals
  - IR language has infinite registers
  - IR follows the architecture's calling convention
- IR to ASM
  - All intermediate values are usually stored on the stack
  - Operands are stored into eax and ecx

Frontend and Backend

- Frontend compiles from language to IR
- Backend compiles from IR to machine code
  - Rust, C, etc all share the same backend: LLVM

Compilation and Execution Pipeline



Buffer Overflow Attacks

- A buffer overflow attack can occur when a malicious program is able to change the return address of a function to point to a different location in memory.
- This can be mitigated by the compiler by randomizing the location of the program's memory.

Optimizations

- Compilers can cut down on constant and multiplicative efficiency (i.e. $O(10n)$ to $O(n)$)
- Compilers cannot optimize algorithmic efficiency (i.e. $O(n^2)$ to $O(1)$)

Code Motion

- Ex. moving an often-computed loop constant out of the loop

```
// Old
for(int j = 0; j < n; j++) {
    a[n * i + j] = b[j];
}

// New
int ni = n * i;
for(int j = 0; j < n; j++) {
    a[ni + j] = b[j];
}
```

Strength Reduction

- Replace an expensive operation with a cheaper one
- Ex: multipliply by 2 with bit shift
- Ex: replace a sequence of products with additions

```
// Old
for(int i = 0; i < n; i++)
    int product = n * il;
    for(int j = 0; j < n; j++)
        a[product + j] = b[j];

// New
int product = 0;
for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++)
        a[product + j] = b[j];
    product += n;
```

Common Subexpressions

- You can extract a common expression out of repeated statements into a variable

```
// Old
int prev2 = val[i * n + j - 2];
int prev1 = val[i * n + j - 1];
int next1 = val[i * n + j + 1];
int next2 = val[i * n + j + 2];

// New
int inj = i * n + j;
int prev2 = val[inj - 2];
int prev1 = val[inj - 1];
int next1 = val[inj + 1];
int next2 = val[inj + 2];
```

Lea Optimizations

- The `lea` instruction can be used to perform arithmetic operations on the address of a memory location
- `lea` can be used to do arithmetic without changing flags

```
for(int i = 0; i < len; i++) {
    accum += a[i];
    accum += a[i];
}
```

```
movl    (%rdi,%rcx,4), %r8d   ; Load a[i] into r8d
leal (%rax,%r8,2), %eax       ; Multiply r8d by 2 and add to rax; store
result in eax
```

Optimization Blockers

- Procedure calls can block optimizations, since we do not know if the function will change the value of a variable
  - However, some common functions (like STDlib functions, like `strlen`) can be optimized. The compiler inserts a check to see if the string changed, and only computes it when it changes

```
// This cannot be optimized
for(int i = 0; i < len; i++) {
    accum += a[i];
    accum += myFun(a[i]);
    accum += a[i];
}
```

- Aliasing can block optimizations, since we do not know if the value of a variable will be changed by another variable
  - To get around this, you can optimize manually, or change the type of one variable (i.e. one is `int`, one is `long`) — this is called "strict aliasing"
    - This is why, for example, casting a float pointer to an int pointer is undefined behavior

```
// This cannot be optimized, since if a and b point to the same memory,
there is an edge case
for(int i = 0; i < n; i++) {
    b[i] = 0;
    for(int j = 0; j < n; j++)
        b[i] += a[i * n + j];
}
```