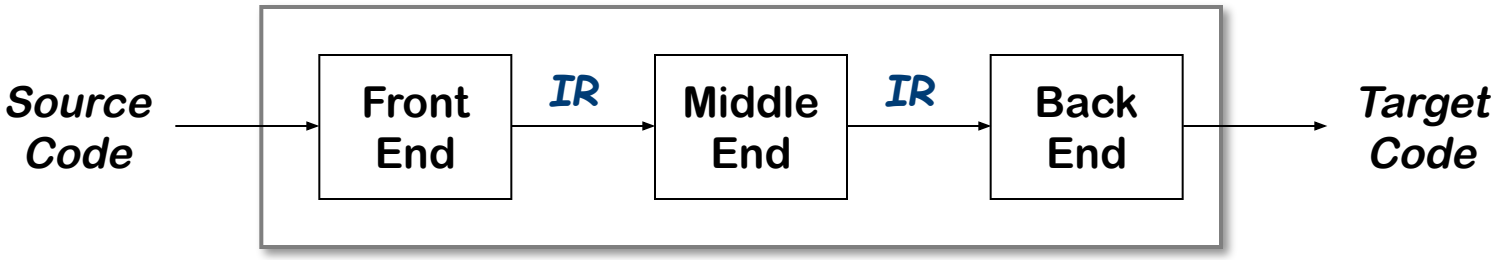
CENG513 Compiler Design and Construction
Intermediate Representations

Note by Işıl ÖZ:
Our slides are adapted from Cooper and Torczon's slides that are prepared for COMP 412 at Rice.

Intermediate Representations



- Front end - produces an intermediate representation (*IR*)
- Middle end - transforms the *IR* into an equivalent *IR* that runs more efficiently
- Back end - transforms the *IR* into native code

- *IR* encodes the compiler's knowledge of the program
- Middle end usually consists of several passes

Intermediate Representations

- Records all of the useful facts that might be passed between passes of the compiler
- During translation, the compiler derives facts that have no representation in the source code - like the addresses of variables and procedures
- The compiler augments IR with a set of tables that record additional information, which can be considered to be part of the IR

Intermediate Representations

- Decisions in *IR* design affect the speed and efficiency of the compiler

- Some important *IR* properties
  - Ease of generation
  - Ease of manipulation
  - Procedure size
  - Freedom of expression
  - Level of abstraction

- The importance of different properties varies between compilers
  - Selecting an appropriate *IR* for a compiler is critical
  - Source to source translator keeps its internal information in a form quite close to the source
  - A compiler that produces assembly code for a microcontroller might use an internal form close in form to the target machine's instruction set

Types of Intermediate Representations

Three major categories

- Structural
  - Graphically oriented
  - Heavily used in source-to-source translators
  - Tend to be large

  Examples:
  Parse trees,
  DAGs

- Linear
  - Pseudo-code for an abstract machine
  - Level of abstraction varies
  - Simple, compact data structures
  - Easier to rearrange

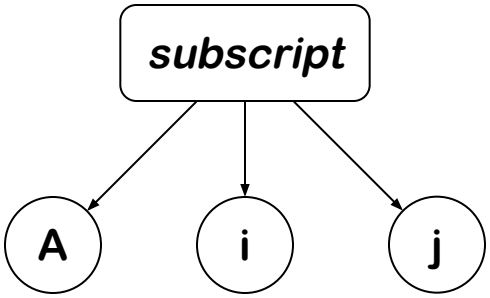
  Examples:
  3 address code
  Stack machine code
  ILOC

- Hybrid
  - Combination of graphs and linear code
  - Example: control-flow graph

  Example:
  Control-flow graph

Level of Abstraction

- The level of detail exposed in an *IR* influences the profitability and feasibility of different optimizations
- Two different representations of an array reference (A[i,j], two dimensional array with ten elements per dimension):



High level AST:
Good for memory disambiguation

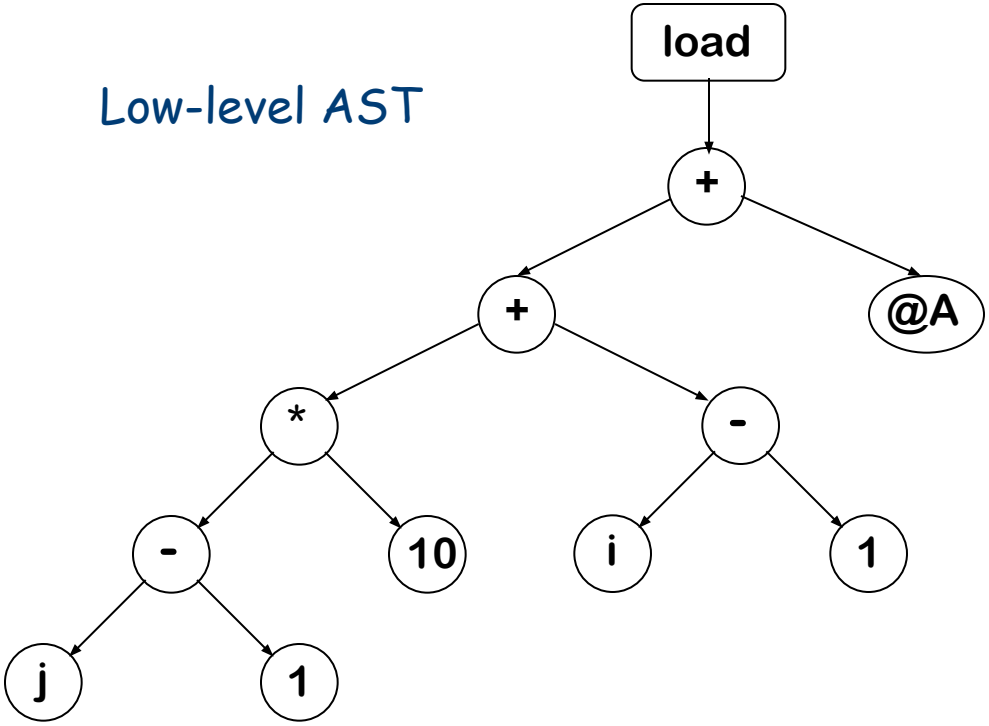
```
subI  r_i,1  ⇒ r_1
multI r_1,10 ⇒ r_2
subI  r_j,1  ⇒ r_3
add   r_2,r_3 ⇒ r_4
multI r_4,4  ⇒ r_5
loadI @A     ⇒ r_6
add   r_5,r_6 ⇒ r_7
load  r_7    ⇒ r_Aij
```

Low level linear code:
Good for address calculation

Level of Abstraction

- Structural *IRs* are usually considered high-level
- Linear *IRs* are usually considered low-level
- Not necessarily true:

Low-level AST

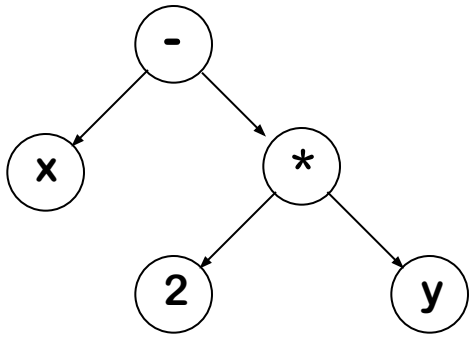



```
loadArray A,i,j
```

High-level linear code

Abstract Syntax Tree

An abstract syntax tree is the procedure's parse tree with the nodes for most non-terminal nodes removed



x - 2 * y

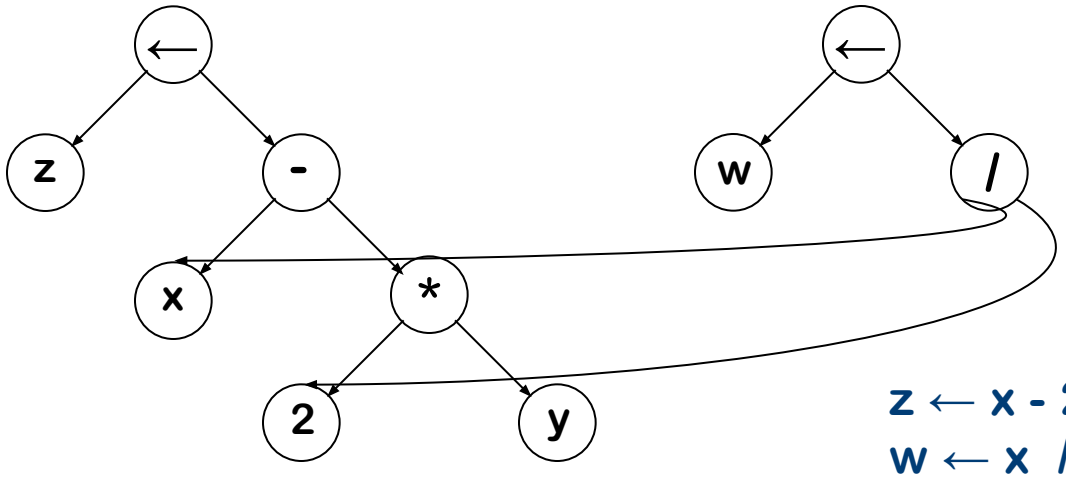
- Can use linearized form of the tree
  - Easier to manipulate than pointers

    `x 2 y * -` in postfix form

    `- * 2 y x` in prefix form

Directed Acyclic Graph

A directed acyclic graph (DAG) is an AST with a unique node for each value



z ← x - 2 * y
w ← x / 2

- Makes sharing explicit
- Encodes redundancy

With two copies of the same expression, the compiler might be able to arrange the code to evaluate it only once.

Graphical IR

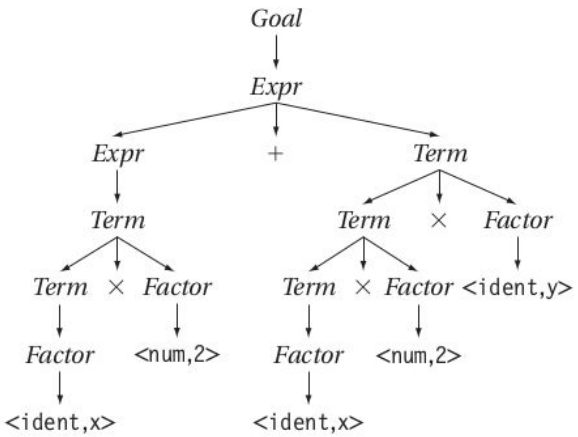
Parse tree is a graphical representation for the derivation, or parse, that corresponds to the input program

AST retains the essential structure of the parse tree but eliminates the extraneous nodes

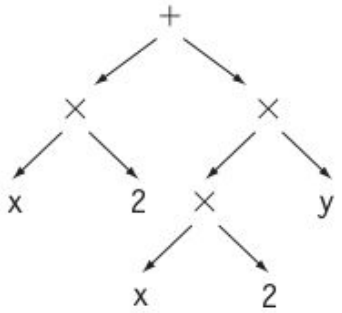
DAG avoids duplication of distinct copies of the expression

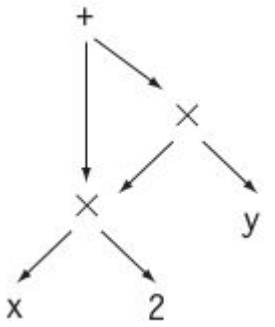
Goal	→	*Expr*
Expr	→	*Expr* + *Term*
	|	*Expr* − *Term*
	|	*Term*
Term	→	*Term* × *Factor*
	|	*Term* ÷ *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	num
	|	ident

Classic Expression Grammar



Parse Tree for x × 2 + x × 2 × y





Three Address Code

Several different representations of three address code

- In general, three address code has statements of the form:

  $x \leftarrow y \ op \ z$

  With 1 operator (*op*) and, at most, 3 names (x, y, & z)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t ← 2 * y
z ← x - t
```

Advantages:

- Resembles many real machines
- Introduces a new set of names (t)
- Compact form

Three Address Code: Quadruples

Naïve representation of three address code

- Table of k * 4 small integers
- Simple record structure
- Easy to reorder
- Explicit names

The original FORTRAN compiler used "quads"

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

RISC assembly code

load	1	y	
loadi	2	2	
mult	3	2	1
load	4	x	
sub	5	4	3

Quadruples

Three Address Code: Triples

- Index used as implicit name
- 25% less space consumed than quads
- Much harder to reorder

```
load  r1, y
loadI r2, 2
mult  r3, r2, r1
load  r4, x
sub   r5, r4, r3
```

(1)	load	y	
(2)	loadI	2	
(3)	mult	(1)	(2)
(4)	load	x	
(5)	sub	(4)	(3)

Implicit names occupy no space

Two Address Code

- Allows statements of the form

  $x \leftarrow x$ *op* y

  Has 1 operator (*op*) and, at most, 2 names (x and y)

Example:

$z \leftarrow x - 2 * y$ becomes

```
t1 ← 2
t2 ← load y
t2 ← t2 * t1
z  ← load x
z  ← z - t2
```

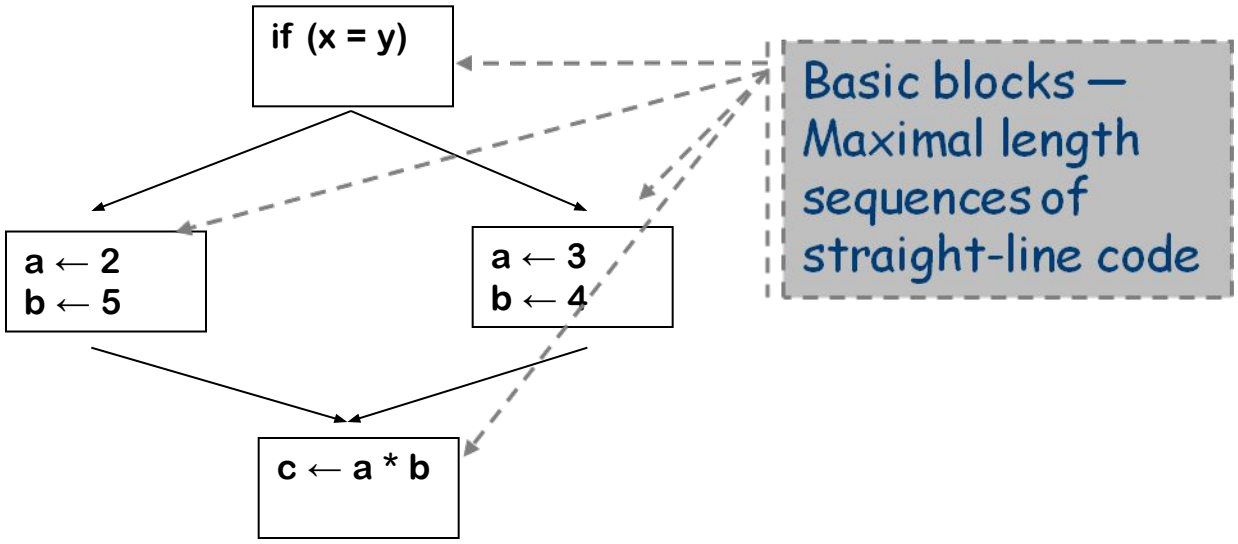
- Can be very compact

Control-flow Graph

Models the transfer of control in the procedure

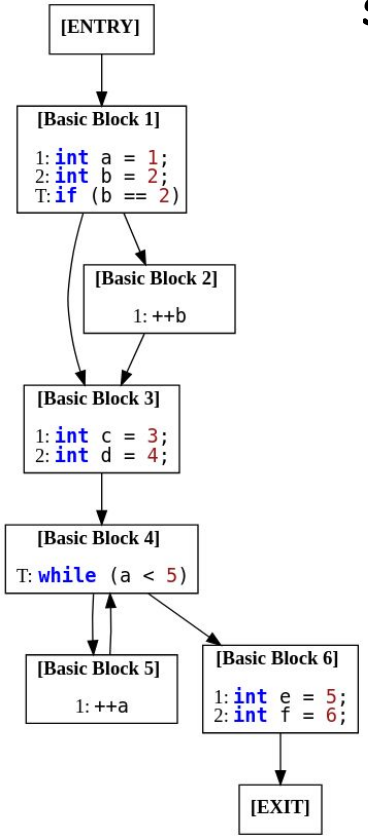
- Nodes in the graph are basic blocks
  - Can be represented with quads or any other linear representation
- Edges in the graph represent control flow

Example



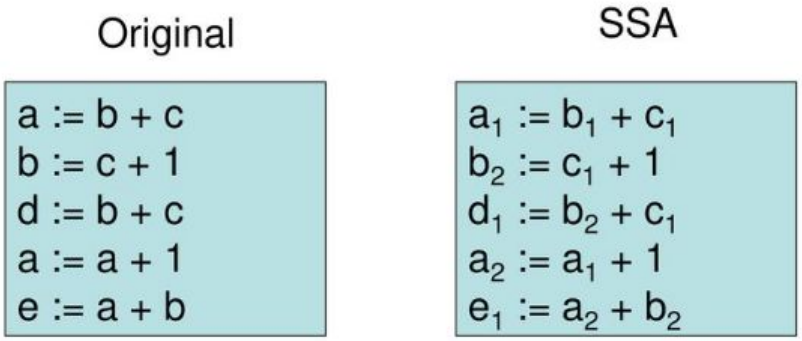
Basic Block

- A basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
- Whenever the first instruction in a basic block is executed, the rest of the instruct[illegible]sarily executed exactly once and in orde[illegible]



Static Single Assignment Form

- Each assignment has a unique name, each use refers to a single definition (adds information about both control flow and data flow)



- Easier for optimizations
- To make it work, introduce φ-functions at points, where control-flow paths merge

Static Single Assignment Form - φ-functions

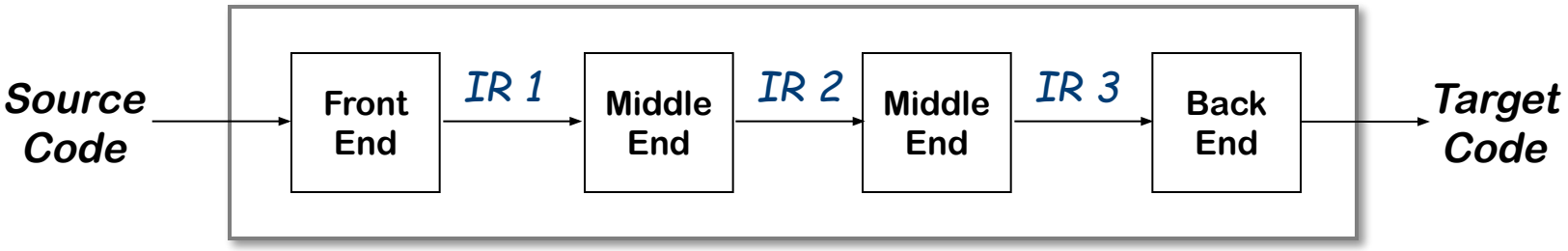
```
x ← ···
y ← ···
while(x < 100)
  x ← x + 1
  y ← y + x
```

Original Code

```
     x0 ← ···
     y0 ← ···
     if (x0 ≥ 100) goto next
loop: x1 ← φ(x0,x2)
     y1 ← φ(y0,y2)
     x2 ← x1 + 1
     y2 ← y1 + x2
     if (x2 < 100) goto loop
next:x3 ← φ(x0,x2)
     y3 ← φ(y0,y2)
```

Its SSA Form

Using Multiple Representations



- Repeatedly lower the level of the intermediate representation
  - Each intermediate representation is suited towards certain optimizations
- Example: the Open64 compiler
  - WHIRL intermediate format
    - → Consists of 5 different *IR*s that are progressively more detailed and less abstract

Memory Models

Two major models

- Register-to-register model
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores
- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers directly before they are used
  - Compiler back-end can remove loads and stores

Compilers for RISC machines usually use register-to-register

- Reflects programming model
- Easier to determine when registers are used

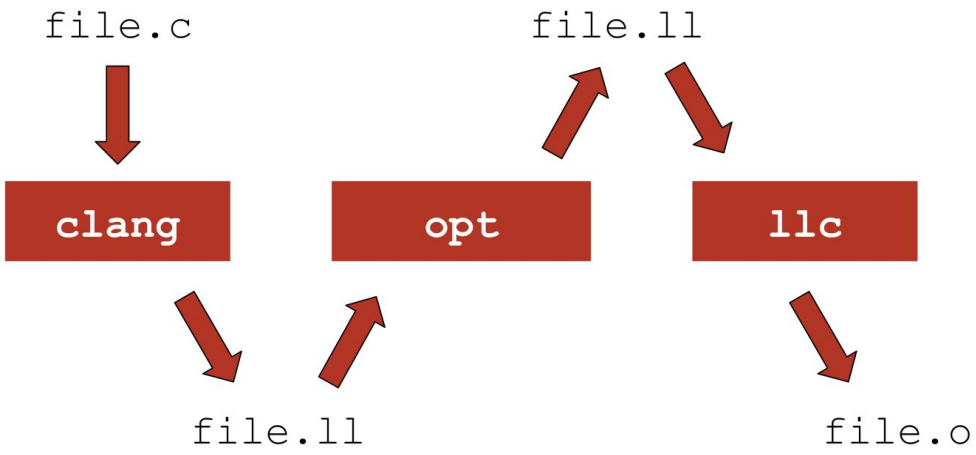
Symbol Tables

Data structure to store information about the various entities that the program being translated manipulates

- For a variable: a data type, its storage class, the name and lexical level of its declaring procedure, and a base address and offset in memory
- For an array: the number of dimensions and the upper and lower bounds for each dimension
- For functions and procedures: the number of parameters and their types, as well as the types of any returned values

- Can be recorded directly in the IR
- Central repository for these facts →symbol table

LLVM - Compilation Infrastructure



clang - Compile into X86 Assembly

```
.text
.file   "file.c"
.globl  prefix_sum
.p2align        4, 0x90
.type
prefix_sum,@function
prefix_sum:
  .cfi_startproc
# %bb.0:
  pushq %rbp
  .cfi_def_cfa_offset 16
  .cfi_offset %rbp, -16
  movq  %rsp, %rbp
  .cfi_def_cfa_register
%rbp
  xorl  %eax, %eax
  movq  %rdi, -8(%rbp)
  movq  %rsi, -16(%rbp)
  movl  %edx, -20(%rbp)
  cmpl  -20(%rbp), %eax
  jge   .LBB0_10
# %bb.1:
  movl  $0, -24(%rbp)
  ...
```

```
$ clang -S file.c -o file.ll
```

```
void prefix_sum(int *src, int *dst, int N) {
   if (0 < N) {
       int i = 0;
       do {
           int tmp = 0;
           int j = 0;
           if (j < i) {
               do {
                   tmp += src[j];
                   j++;
               } while (j < i);
               dst[i] = tmp;
           }
           i++;
       } while (i < N);
   }
}
```

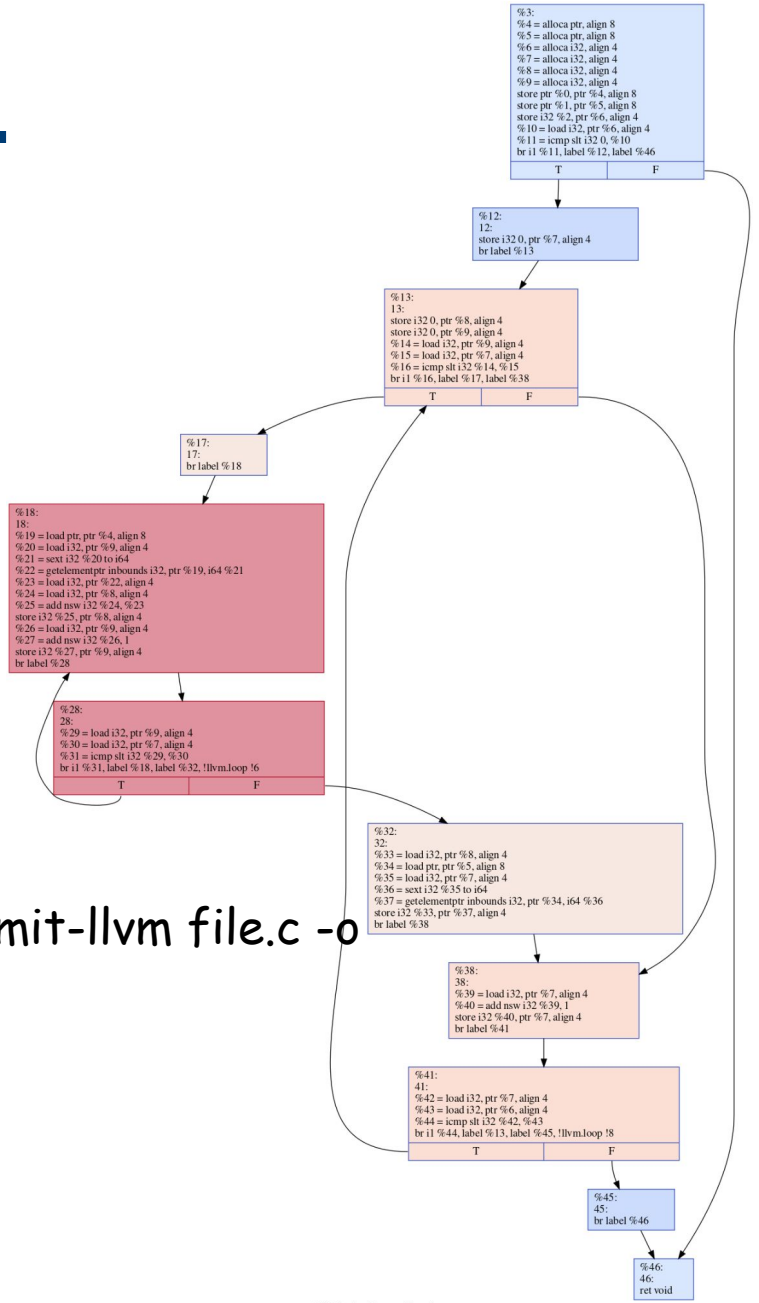
clang - Compile into LLVM IR

```
$ clang -S -emit-llvm file.c -o file.ll
```

```
define dso_local
void @prefix_sum(i32* %src, i32* %dst, i32 %N)
#0 {
entry:
  %src.addr = alloca i32*, align 8
  %dst.addr = alloca i32*, align 8
  %N.addr = alloca i32, align 4
  %i = alloca i32, align 4
  %tmp = alloca i32, align 4
  ...
```

opt - Control Flow Graph

LLVM IR to CFG

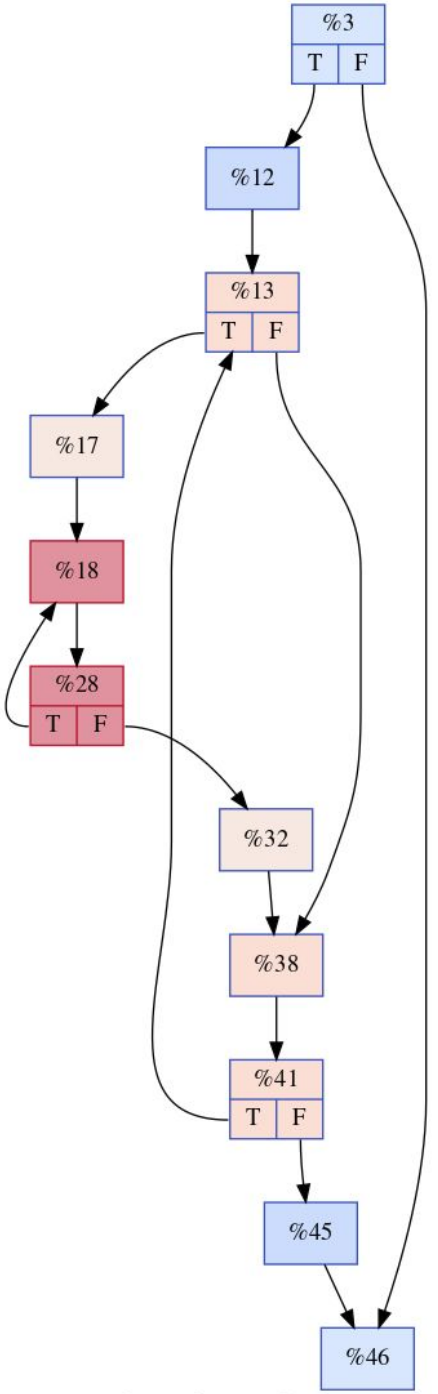



```
$ clang -Xclang -disable-O0-optnone -S -emit-llvm file.c -o file.ll
$ opt -p dot-cfg file.ll -S
$ dot -Tsvg .prefix_sum.dot > output.svg
```

opt - Control Flow Graph

LLVM IR to CFG

```
$ opt -p dot-cfg-only file.ll -S
$ dot -Tsvg .prefix_sum.dot >
output.svg
```



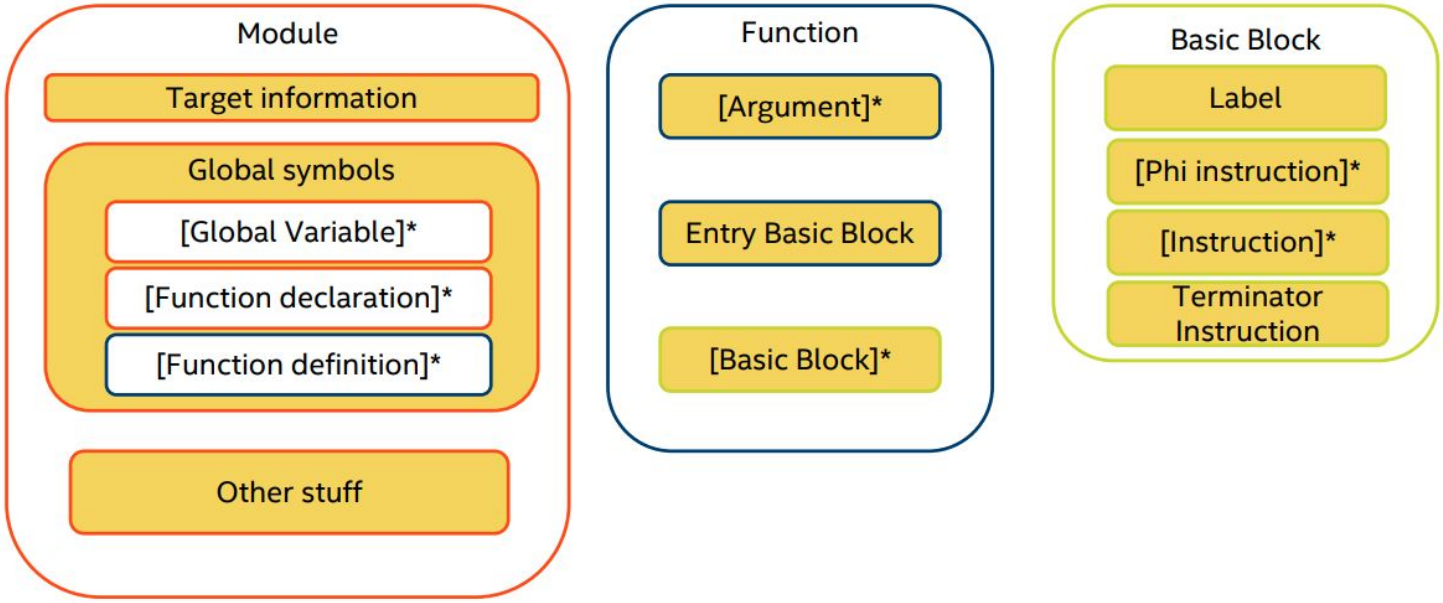
CFG for 'prefix_sum' function

LLVM Intermediate Representation

Is a low level programming language

… while being able to represent high-level ideas

Enables efficient code optimization



Hand-Written IR

```c
int factorial(int val);

int main(int argc, char** argv)
{
  return factorial(2) * 7 == 42;
}
```

```
declare i32 @factorial(i32)

define i32 @main(i32 %argc, i8** %argv) {
  %1 = call i32 @factorial(i32 2)
  %2 = mul i32 %1, 7
  %3 = icmp eq i32 %2, 42
  %result = zext i1 %3 to i32
  ret i32 %result
}
```

Hand-Written IR

```c
// Precondition: val is non-negative.
int factorial(int val) {
  if (val == 0)
    return 1;
  return val * factorial(val - 1);
}
```

```
; Precondition: %val is non-negative.
define i32 @factorial(i32 %val) {
  %is_base_case = icmp eq i32 %val, 0
  br i1 %is_base_case, label %base_case, label %recursive_case
base_case:
  ret i32 1
recursive_case:
  %1 = add i32 -1, %val
  %2 = call i32 @factorial(i32 %0)
  %3 = mul i32 %val, %1
  ret i32 %2
}
```

References

Chapter sections from the book:

- 5.2, 5.3, 5.4, 5.5

LLVM Language Reference Manual

- https://llvm.org/docs/LangRef.html

LLVM IR Tutorial, 2019 EuroLLVM Developers' Meeting

- https://www.youtube.com/watch?v=m8G_S5LwlTo

Kaleidoscope Code Generation

- https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl03.html