Intermediate Representation (IR)

Compiler Design: Summer 2025

Promit Panja* (Based on lecture notes by Frank Pfenning, André Platzer, Rob Simmons, and Jan Hoffmann)

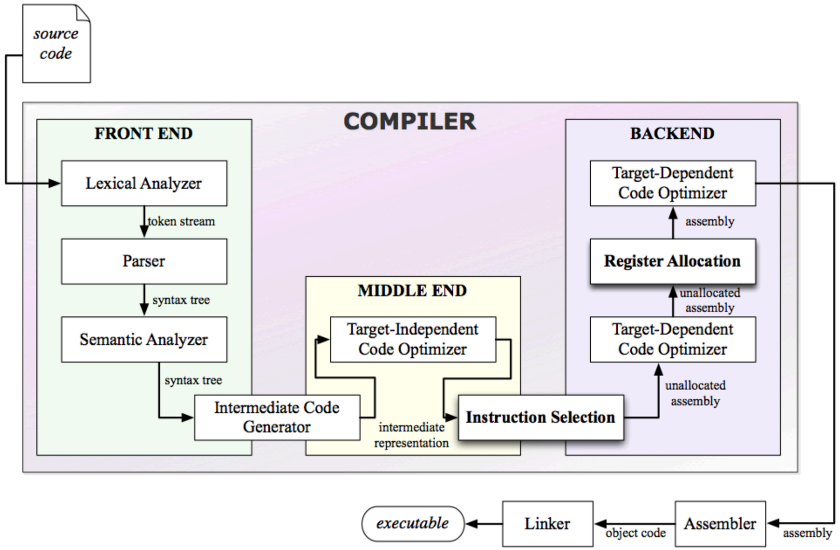
2025-05-26

KIT

What is IR?

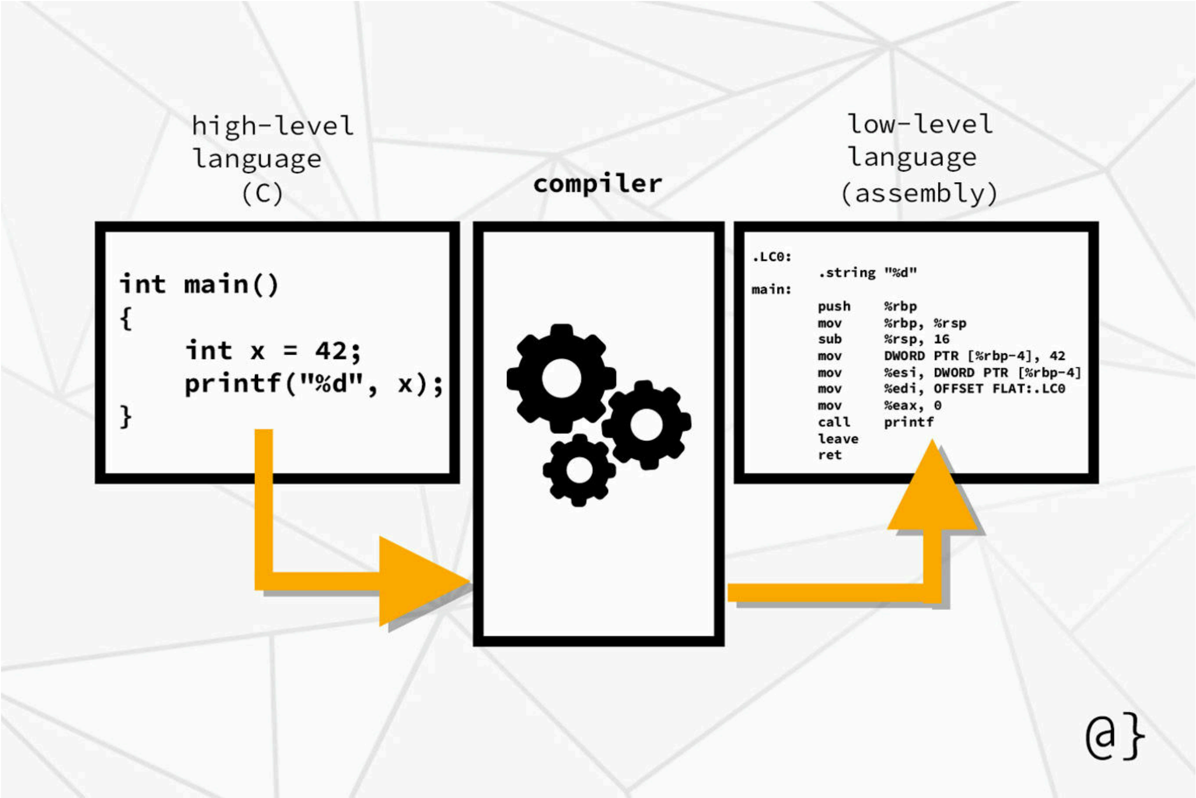
What is IR?

- Intermediate Representation (IR) is the **"middle end"** of the compiler.
- A bridge between the **source code** and **machine code**.



Why do we need IR?

Why do we need IR?



Why do we need IR?

Target-Independent Code Optimizations

Why do we need IR?

Target-Independent Code Optimizations

- Constant Propagation

Why do we need IR?

Target-Independent Code Optimizations

- Constant Propagation
- Dead Code Elimination

Why do we need IR?

Target-Independent Code Optimizations

- Constant Propagation
- Dead Code Elimination
- Loop Optimizations

Why do we need IR?

Target-Independent Code Optimizations

- Constant Propagation
- Dead Code Elimination
- Loop Optimizations
- Many more…

How many forms of IRs?

How many forms of IRs?

Number of IRs and their precise form depends on the context:

How many forms of IRs?

Number of IRs and their precise form depends on the context:

- The complexity and the form of the language.

How many forms of IRs?

Number of IRs and their precise form depends on the context:

- The complexity and the form of the language.
- Number of different machine architectures the compiler is designed to target.

How many forms of IRs?

Number of IRs and their precise form depends on the context:

- The complexity and the form of the language.
- Number of different machine architectures the compiler is designed to target.
- What kinds of optimizations are important for the implementation.

Common types of IRs

- IR Trees
- Static Single-Assignment form (SSA)
- LLVM IR
- Three-Address Instructions (Quads)
- Two-Address Instructions (Triples)

Abstract Syntax Trees (AST)

Abstract Syntax Trees (AST)

- After the source has been parsed and elaborated we obtain an **abstract syntax tree (AST)**.

Abstract Syntax Trees (AST)

- After the source has been parsed and elaborated we obtain an **abstract syntax tree (AST)**.
- We describe the ASTs in a BNF form (Backus-Naur Form).

Abstract Syntax Trees (AST)

- After the source has been parsed and elaborated we obtain an **abstract syntax tree (AST)**.
- We describe the ASTs in a BNF form (Backus-Naur Form).

$$\begin{aligned}\text{Expressions } e ::= {} & n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid f(e_1, \ldots, e_n) \\ & \mid e_1 ? e_2 \mid !e \mid e_1 \&\& e_2 \mid e_1 \parallel e_2\end{aligned}$$

Abstract Syntax Trees (AST)

- After the source has been parsed and elaborated we obtain an **abstract syntax tree (AST)**.
- We describe the ASTs in a BNF form (Backus-Naur Form).

$$\begin{aligned}\text{Expressions } e &::= n \mid x \mid e_1 \oplus e_2 \mid e_1 \oslash e_2 \mid f(e_1, \ldots, e_n) \\ &\mid e_1 ?\, e_2 \mid !e \mid e_1 \&\& e_2 \mid e_1 \parallel e_2\end{aligned}$$

$$\begin{aligned}\text{Statements } s &::= \text{declare}(x, \tau, s) \mid \text{assign}(x, e) \mid \text{if}(e, s_1, s_2) \\ &\mid \text{while}(e, s) \mid \text{return}(e) \mid \text{nop} \mid \text{seq}(s_1, s_2)\end{aligned}$$

Abstract Syntax Trees (AST)

```c
// Before Elaboration
for (int x = 4; x < 30; x++) { y = y + x }
```

Abstract Syntax Trees (AST)

```c
// Before Elaboration
for (int x = 4; x < 30; x++) { y = y + x }
```

↓

```c
// After Elaboration
{
  int x = 4;
  while (x < 30) { y = y + x; x += 1 }
}
```

Abstract Syntax Trees (AST)

```c
{
  int x = 4;
  while (x < 30) { y = y + x; x += 1 }
}
```

Abstract Syntax Trees (AST)

```c
{
  int x = 4;
  while (x < 30) { y = y + x; x += 1 }
}
```

$\downarrow$

```c
declare(x, int, seq(assign(x, 4),
                    while(x < 30,
                          seq(assign(y, y + x), assign(x, x + 1)))))
```

Abstract Syntax Trees (AST)

- We carry out various static analyses to see if the program is well-formed.

Abstract Syntax Trees (AST)

- We carry out various static analyses to see if the program is well-formed.
  - Type-checking.

Abstract Syntax Trees (AST)

- We carry out various static analyses to see if the program is well-formed.
  - Type-checking.
  - Every finite control flow path ends in a return statement.

Abstract Syntax Trees (AST)

- We carry out various static analyses to see if the program is well-formed.
  - Type-checking.
  - Every finite control flow path ends in a return statement.
  - Every variable is initialized before its use along every control flow path.

IR Trees

IR Trees

Through this translation we want to achieve several goals

IR Trees

Through this translation we want to achieve several goals

- Isolate **potentially effectful** expressions, making their order of execution explicit,
  - which simplifies instruction selection
  - remaining pure expressions can be optimized much more effectively.

IR Trees

Through this translation we want to achieve several goals

- Isolate **potentially effectful** expressions, making their order of execution explicit,
  - which simplifies instruction selection
  - remaining pure expressions can be optimized much more effectively.
- Make control flow explicit in the form of **conditional** or **unconditional** branches,
  - which is closer to the assembly language target
  - apply standard program analyses based on an explicit control flow graph

IR Trees

- We describe the IR through pure expressions p and commands c.

IR Trees

- We describe the IR through pure expressions p and commands c.
- Programs r are just sequences of commands.

IR Trees

- We describe the IR through pure expressions p and commands c.
- Programs r are just sequences of commands.
- An empty sequence of commands is denoted by $' \cdot '$

IR Trees

- We describe the IR through pure expressions p and commands c.
- Programs r are just sequences of commands.
- An empty sequence of commands is denoted by $' \cdot '$
- We write $r_1 ; r_2$ for the concatenation of two sequences of commands.

IR Trees

$$\text{Pure Expressions } p ::= n \mid x \mid p_1 \oplus p_2$$

IR Trees

$$\text{Pure Expressions } p ::= n \mid x \mid p_1 \oplus p_2$$

$$\begin{aligned} \text{Commands } c ::= \ & x \leftarrow p \mid x \leftarrow p_1 \oslash p_2 \mid x \leftarrow f(p_1, \cdots, p_n) \\ & \mid \text{if } (p_1 ? p_2) \text{ then } l_t \text{ else } l_f \mid \text{goto } l \\ & \mid l : \mid \text{return}(p) \end{aligned}$$

IR Trees

$$\text{Pure Expressions } p ::= n \mid x \mid p_1 \oplus p_2$$

$$\begin{aligned}\text{Commands } c ::= {} & x \leftarrow p \mid x \leftarrow p_1 \oslash p_2 \mid x \leftarrow f(p_1, \cdots, p_n) \\ & \mid \text{if } (p_1 \, ? \, p_2) \text{ then } l_t \text{ else } l_f \mid \text{goto } l \\ & \mid l : \mid \text{return}(p)\end{aligned}$$

$$\text{Programs } r ::= c_1 ; \cdots ; c_n$$

Translating Expressions

Translating Expressions

- The first idea may be to translate abstract syntax expressions → pure expressions

Translating Expressions

- The first idea may be to translate abstract syntax expressions $\rightarrow$ pure expressions
  - potentially effectful expressions have to be turned into commands,
  - and commands are **not** permitted inside pure expressions.

Translating Expressions

- The first idea may be to translate abstract syntax expressions $\rightarrow$ pure expressions
  - potentially effectful expressions have to be turned into commands,
  - and commands are **not** permitted inside pure expressions.
- Returning just a command, or a sequence of commands is also insufficient because
  - we somehow need to refer to the result of the translation as a pure expression so we can use it.

Translating Expressions

A solution is to translate from an expression e to a **pair** consisting of a sequence of commands r and a pure expression p. After executing r, the value of p will be the value of e.

Translating Expressions

A solution is to translate from an expression e to a **pair** consisting of a sequence of commands r and a pure expression p. After executing r, the value of p will be the value of e.

$$\operatorname{tr}(e) = \langle \check{e}, \hat{e} \rangle$$

- $\check{e}$ is a sequence of commands r that we need to write down to compute the effects of e.
- $\hat{e}$ is a pure expression p that we can use to compute the value of e back up.

Translating Expressions

- Constants and variables translate to themselves.

$$\mathrm{tr}(n) = \langle \cdot, n \rangle$$

$$\mathrm{tr}(x) = \langle \cdot, x \rangle$$

- It is possible that the sub-expressions of pure operation $e_1 \oplus e_2$ have effects,

$$\mathrm{tr}(e_1 \oplus e_2) = \langle (\check{e}_1 ; \check{e}_2), \hat{e}_1 \oplus \hat{e}_2 \rangle$$

Translating Expressions

- Effectful operations and function calls require us to introduce some commands and a fresh temporary variable to refer to the resulting value from the operation or call

$$\begin{aligned} \mathrm{tr}(e_1 \oslash e_2) &= \langle(\check{e}_1; \check{e}_2; t \leftarrow \hat{e}_1 \oslash \hat{e}_2), t\rangle && (t \text{ fresh}) \\ \mathrm{tr}(f(e_1, ..., e_n)) &= \langle(\check{e}_1; ...; \check{e}_n; t \leftarrow f(\hat{e}_1, ..., \hat{e}_n)), t\rangle && (t \text{ fresh}) \end{aligned}$$

Translating Statements

- Translating statements is somewhat simpler, because we only need to return a sequence of commands.

Translating Statements

- Translating statements is somewhat simpler, because we only need to return a sequence of commands.
- But is slightly more complicated in other ways, since we have to manage control flow via jumps and conditional branches.

Translating Statements

- Translating statements is somewhat simpler, because we only need to return a sequence of commands.
- But is slightly more complicated in other ways, since we have to manage control flow via jumps and conditional branches.

$$\mathrm{tr}(s) = \check{s},$$

where $\check{s}$ is a sequence of commands r.

Translating Statements

- Assignments and conditionals are simple.

$$
\begin{aligned}
\mathrm{tr}(\mathrm{assign}(x, e)) &= \check{e}; x \leftarrow \hat{e} \\
\mathrm{tr}(\mathrm{return}(e)) &= \check{e}; \mathrm{return}(\hat{e}) \\
\mathrm{tr}(\mathrm{nop}) &= \cdot \\
\mathrm{tr}(\mathrm{seq}(s_1, s_2)) &= \check{s}_1; \check{s}_2
\end{aligned}
$$

Translating Statements

- Conditionals require labels and jumps.

$$
\begin{aligned}
\mathrm{tr}(\mathrm{if}(e, s_1, s_2)) = \ & \check{e};\\
& \text{if } (\hat{e}! = 0) \text{ then } l_1 \text{ else } l_2;\\
& l_1 : \check{s}_1; \text{goto } l_3;\\
& l_2 : \check{s}_2; \text{goto } l_3;\\
& l_3 : \qquad\qquad\qquad\qquad (l_1, l_2, l_3 \text{ fresh})
\end{aligned}
$$

Translating Boolean Expressions

- The code with the previous translations does not take advantage of the way conditional branches work in x86 and x86_64.
- Where we can compare two values and then branch based on the outcome of the comparison by testing condition flags.

$$\mathrm{cp}(b, l, l') = r$$

- Where b is a boolean expression.

Translating Boolean Expressions

We define

$$
\begin{aligned}
\mathrm{cp}(e_1 \mathbin{?} e_2, l, l') &= \check{e}_1; \check{e}_2; \text{if } (\hat{e}_1 \mathbin{?} \hat{e}_2) \text{ then } l \text{ else } l' \\
\mathrm{cp}(!e, l, l') &= \mathrm{cp}(e, l', l) \\
\mathrm{cp}(e_1 \&\& e_2, l, l') &= \mathrm{cp}(e_1, l_2, l'); l_2 : \mathrm{cp}(e_2, l, l') \quad (l_2 \text{ fresh}) \\
\mathrm{cp}(0, l, l') &= \text{goto } l' \\
\mathrm{cp}(1, l, l') &= \text{goto } l \\
\mathrm{cp}(e, l, l') &= \check{e}; \text{if } (\hat{e} != 0) \text{ then } l \text{ else } l'
\end{aligned}
$$

Translating Boolean Expressions

This is then used to translate statements in a straightforward way

$$
\begin{aligned}
\mathrm{tr}(\mathrm{if}(b, s_1, s_2)) = \mathrm{cp}(b, l_1, l_2) \\
l_1 : \mathrm{tr}(s_1); \mathrm{goto}\ l_3 \\
l_2 : \mathrm{tr}(s_2); \mathrm{goto}\ l_3 \\
l_3 : \qquad\qquad\qquad (l_1, l_2, l_3\ \text{fresh})
\end{aligned}
$$

Extended Basic Blocks

- The translations we have seen so far translates code into basic block form.

Extended Basic Blocks

- The translations we have seen so far translates code into basic block form.
- One of the benefits of basic blocks is that they can be treated as a single independent unit in many analyses.

Extended Basic Blocks

- The translations we have seen so far translates code into basic block form.
- One of the benefits of basic blocks is that they can be treated as a single independent unit in many analyses.
- However, the translation to basic block results in large number of labels and jumps which can obscure the structure of the program to some degree.

Extended Basic Blocks

- The translations we have seen so far translates code into basic block form.
- One of the benefits of basic blocks is that they can be treated as a single independent unit in many analyses.
- However, the translation to basic block results in large number of labels and jumps which can obscure the structure of the program to some degree.
- An **extended basic block** is a collection of basic blocks with one label at the beginning and internal labels, each of which is the target of only one internal jump and no external jumps.

Extended Basic Blocks

```c
init fastpow(int b, int e)
//@requires e >= 0;
{
  int r = 1;
  while (e > 0)
    {
      if (e % 2 != 0)
        r = r * b;
      b = b * b;
      e = e / 2;
    }
  return r;
}
```

Extended Basic Blocks

```c
fastpow(b, e):
  r <- 1
  goto loop
loop:
  if (e > 0) then body else done
body:
  t0 <- e % 2
  if (t0 == 0) then mult else next
mult:
  r <- r * b
  goto next
next:
  b <- b * b
```

```
  e <- e / 2
  goto loop
done:
  return r
```

Extended Basic Blocks

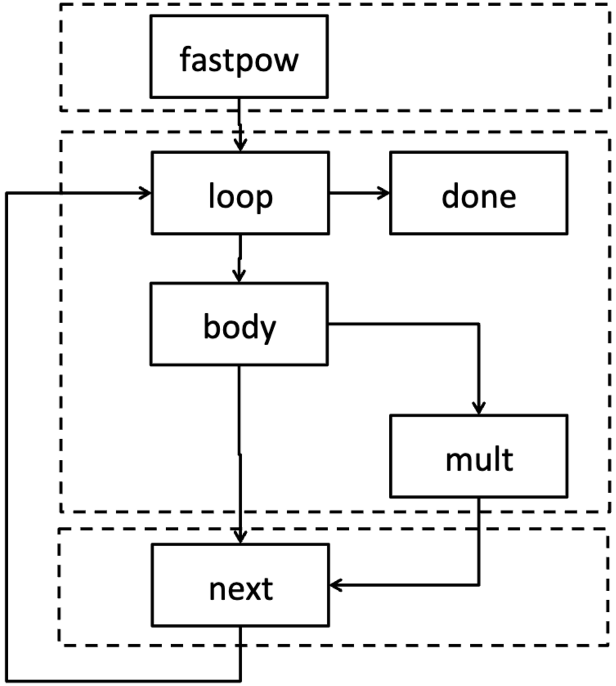


Figure 1: Control flow graph of the program.

Extended Basic Blocks

```c
fastpow(b, e):
  r <- 1
  goto loop
loop:
  if (e > 0)
  then
    t <- e % 2    // loop
    if (t0 == 0)
      r <- r * b  // mult
      goto next
    else
      goto next
  else
```

```
    return r      // done
next:
  b <- b * b
  e <- e / 2
  goto loop
```

What is LLVM?

What is LLVM?

“A collection of modular and reusable compiler and toolchain technologies”

What is LLVM?

"A collection of modular and reusable compiler and toolchain technologies"

- Implemented in C++
- LLVM was started by Vikram Adve and Chris Lattner at UIUC in 2000.
- Originally 'Low Level Virtual Machine' started as a research project on dynamic compilation.
- Evolved into an umbrella project for a lot of different things.

LLVM IR

LLVM IR

- Language- and Target-Independent
- One IR for analysis and optimization
  - IR must be able to support high and low-level optimization
- Optimize as much as early as possible
- Low-level and target-independent semantics
  - RISC-like three address code
  - Infinite virtual register set in SSA form
  - Load/store instructions with typed-pointers