

Lecture 13: Intermediate representations

David Hovemeyer

October 19, 2025

601.428/628 Compilers and Interpreters



Agenda

- ▶ Purpose of intermediate representations
- ▶ ASTs
- ▶ Linear IRs
- ▶ Control-flow graphs
- ▶ IR forms and compiler phases
- ▶ Are IRs necessary?

Purpose of IRs

What is an intermediate representation (IR)?

- ▶ “Intermediate representation” is a general term for any data structure which represents the program (or part of the program) being translated by the compiler
- ▶ Compilers typically have various *phases* for which different forms of IR are appropriate
- ▶ Examples:
 - ▶ Abstract Syntax Tree (AST)
 - ▶ Three-address code (a.k.a. “quads”)
 - ▶ Control-flow graph

Facts, annotation of IR

- ▶ IRs represent *facts* about the program
 - ▶ An IR can be *annotated* by these facts to make them available for compiler phases that need them
- ▶ Kinds of facts:
 - ▶ Facts that are directly embodied by the source code
 - ▶ Facts *inferred* from the program's syntax and semantics
 - ▶ Facts that are created/chosen by the compiler (decisions made to enable translation to the target language)

Examples of facts

- ▶ “Fact” = “something that is true at a particular program location”
- ▶ Examples:
 - ▶ The name “a” refers to a variable of type `int` (embodied by the source code)
 - ▶ The variable “a” contains the value 123 (inferred fact)
 - ▶ The storage for variable “a” is located at offset 12 in the stack frame (created fact)

Abstract Syntax Trees

Abstract Syntax Trees

- ▶ ASTs are a “condensed” form of the parse tree based on the derivation found by the parser based on the source language’s syntax rules
 - ▶ Nodes are labeled to identify what kind of source construct they represent (function def, variable declaration, etc.)
- ▶ AST nodes can be annotated with useful information
 - ▶ Pointer to symbol table entry
 - ▶ Type
 - ▶ Whether or not an expression is an lvalue
 - ▶ Etc.

Example C program

```
int main(int argc, char **argv) {  
    return argc + 1;  
}
```

AST of example C program

```
AST_UNIT
+--AST_FUNCTION_DEFINITION
  +--AST_BASIC_TYPE
    |  +--TOK_INT[int]
  +--TOK_IDENT[main]
+--AST_FUNCTION_PARAMETER_LIST
  |  +--AST_FUNCTION_PARAMETER
  |  |  +--AST_BASIC_TYPE
  |  |  |  +--TOK_INT[int]
  |  |  +--AST_NAMED_DECLARATOR
  |  |    +--TOK_IDENT[argc]
  |  +--AST_FUNCTION_PARAMETER
  |    +--AST_BASIC_TYPE
  |    |  +--TOK_CHAR[char]
  |    +--AST_POINTER_DECLARATOR
  |      +--AST_POINTER_DECLARATOR
  |      +--AST_NAMED_DECLARATOR
  |        +--TOK_IDENT[argv]
+--AST_STATEMENT_LIST
  +--AST_RETURN_EXPRESSION_STATEMENT
    +--AST_BINARY_EXPRESSION
      +--TOK_PLUS[+]
      +--AST_VARIABLE_REF
        |  +--TOK_IDENT[argc]
      +--AST_LITERAL_VALUE
        +--TOK_INT_LIT[1]
```

AST implementation

- ▶ AST is just a tree
- ▶ Each node labeled with “tag” indicating meaning of construct
- ▶ Add member variables as needed to store annotations in nodes

NodeBase class

- ▶ In Assignments 3–5, the Node class inherits from NodeBase
- ▶ You can modify NodeBase to add member variables, member functions
- ▶ That way, if we needed to give you an updated version of Node, you wouldn't lose the things you added

Things to put in NodeBase

```
class NodeBase {  
private:  
    Symbol *m_symbol;  
    std::shared_ptr<Type> m_type;  
    bool m_is_lvalue;  
    // etc...
```

Note that the pointer to `Symbol` (a symbol table entry object) should be a “dumb” pointer because `Symbol` objects are owned by the `SymbolTable` object in which they reside

Aside: source to source translation

- ▶ A compiler's target language doesn't need to be assembly language: it could be *source code*
- ▶ The target language could even be the same as (or similar to) the source language
- ▶ In a source-to-source translator, the syntax tree representing the original source code should contain enough information to reproduce it precisely
 - ▶ Which means you would need to avoid simplifications that would lose information

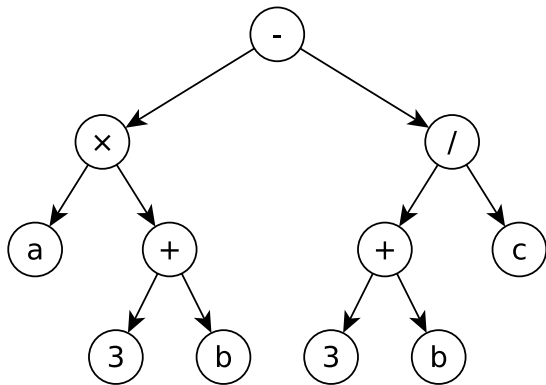
Aside: DAGs

- ▶ DAGs (Directed Acyclic Graphs) can be useful for recognizing and avoiding redundancy in computations
- ▶ Idea is to represent repeated computations with a single representation

DAG example

Computation: $a \times (3 + b) - (3 + b)/c$

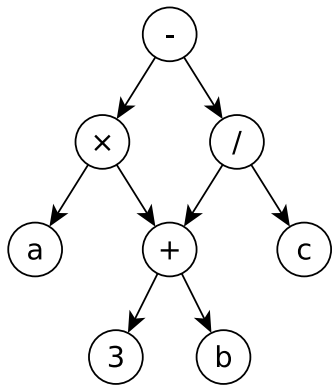
As AST:



DAG example

Computation: $a \times (3 + b) - (3 + b)/c$

As DAG:



DAGs: worthwhile?

- ▶ Redundancies made explicit with DAGs could enable generation of more efficient code
- ▶ E.g., a recursive treewalk, when visiting a previously-visited portion of the DAG, could make use of the result of the previously-emitted code
- ▶ However, there are good techniques to find and eliminate redundant computations in linear IRs (e.g., value numbering)
- ▶ Personal opinion: I'm a bit skeptical whether tree-based optimizations are worth the effort

Linear IRs

Linear IRs

- ▶ A *linear IR* is an intermediate representation a sequence of instructions
- ▶ Reasons why this is useful:
 - ▶ Is “closer” to target code than AST
 - ▶ Concisely represents the operations the program needs to perform when it is executed
 - ▶ Can be fairly convenient for analysis and optimization
- ▶ Disadvantage: doesn't naturally capture control flow (because execution is not sequential when there is a branch)
 - ▶ *Control-flow graphs* extend linear IRs to naturally represent control flow

Quads

- ▶ *Quads* are a common approach to implementing a linear IR
- ▶ Each instruction consists of
 - ▶ A single operation
 - ▶ Between 0 and 3 operands
 - ▶ Most instructions: one destination operand, two source operands
- ▶ An operand could represent
 1. A literal (immediate) value
 2. A “register”
 3. A memory reference via a pointer stored in a register
- ▶ Other forms of operands could be represented, but the ones above are sufficient in general

Quads (format, examples)

General format of a quad:

Opcode	Dest	Source ₁	Source ₂
--------	------	---------------------	---------------------

Example:

add_q	vr10	vr11	vr12
-------	------	------	------

Expressed as “assembly language”:

add_q vr10, vr11, vr12

Note that some opcodes might not require all three operands:

mov_q	vr13	vr15	
-------	------	------	--

Are three operands enough?

- ▶ Quads with (up to) three operands are sufficient to describe computations with one or two source operands and one destination operand
- ▶ This is completely sufficient to describe “ordinary” computations

One kind of operation that could require more than three operands is a *Phi node* in an SSA (Static Single Assignment) representation

The `Instruction` class in Assignments 3–5 allow an arbitrary number of operands, in case you want to add Phi nodes to your linear IR

“High-level” linear IR

- ▶ Many compilers have a “high-level” linear IR form (usually implemented as quads) which is not directly related to the target language
- ▶ Typically, the high-level IR
 - ▶ Is RISC-like (ALU operations have two source operands, one destination operand)
 - ▶ Has an “infinite” number of registers
 - ▶ Has various opcodes to represent computations on values, loads from/stores to memory, comparisons, control flow
- ▶ *Why is this kind of representation useful?*

Why high-level linear IR is useful

- ▶ A high-level linear IR represents the operations that the source program should perform
- ▶ Because it's "RISC-like", computed values are given explicit names
 - ▶ For example, in `add_q vr10, vr11, vr12`, the sum of the 64-bit values in `vr11` and `vr12` is given the name `vr10`
 - ▶ An important class of optimizations involves detecting when a value that is needed has been computed previously
 - ▶ If each computation places its result in a location (register) with a distinct name, it maximizes the extent to which computed values are available
 - ▶ Because high-level instructions don't modify the contents of source operands, they don't "destroy" computed values which might be useful

Why not use an IR based on the target language?

- ▶ Since the eventual goal of the compiler is to produce a translation of the source language to the target language, why not have an IR based on the target language?
- ▶ Answer: we *will* need an IR based on the target language; we'll refer to this as the “low-level” linear IR
- ▶ However, before creating the low-level IR, the compiler will first produce a translation as high-level IR
 - ▶ It should be reasonably straightforward to translate the high-level linear IR to equivalent low-level linear IR

Benefits of a high-level IR

- ▶ Compiler can have multiple “back ends” which translate to different target languages (e.g., x86-64 assembly and ARM assembly)
- ▶ Optimizations can be implemented on the high-level IR (which is designed to be amenable to analysis)
 - ▶ Optimizations on low-level code are also possible and useful, but optimizations on high-level IR are inherently shared between back ends)
- ▶ The target language may have features which make analysis and optimization more difficult
 - ▶ E.g., x86 instructions generally make one operand both a source and a destination

Instruction implementation

```
class Instruction {
private:
    int m_opcode;
    unsigned m_num_operands;
    std::vector<Operand> m_operands;
    // ...

public:
    // ...
    unsigned get_num_operands() const;
    const Operand &get_operand(unsigned index) const;
    void set_operand(unsigned index, const Operand &operand);
    // ...
};
```

Operand implementation

```
class Operand {
public:
    enum Kind { /* ...members... */ };
private:
    Kind m_kind;
    int m_basereg, m_index_reg;
    long m_imm_ival; // also used for offset and scale
    std::string m_label;
public:
    // ...
    Kind get_kind() const;
    int get_base_reg() const;
    int get_index_reg() const;
    long get_imm_ival() const;
    long get_offset() const;
    long get_scale() const;
    // ...
};
```

Idea: an operand represents a register, memory reference via a pointer in a register, an immediate integer value, or a label

A memory reference can optionally have an offset, index, and/or scaling factor

Instruction sequence implementation

```
class InstructionSequence {
private:
    struct Slot {
        std::string label;
        Instruction *ins;
    };
    std::vector<Slot> m_instructions;
    std::string m_next_label;

public:
    // ...
    void append(Instruction *ins);
    unsigned get_length() const;
    Instruction *get_instruction(unsigned index) const;
    void define_label(const std::string &label);
    std::string get_label_at_index(unsigned index) const;
    unsigned get_index_of_labeled_instruction(const std::string &label) const;
    // ...
};
```

Idea: an InstructionSequence is a sequence of Instruction objects (quads)

Each Instruction may (optionally) have a label (to allow it to be a control flow target)

Aside: memory use

- ▶ Since an IR is an in-memory representation of a program (or part of a program), the amount of memory it occupies can be significant
- ▶ For an ahead-of-time compiler on a modern system with a large amount of main memory, might not be a huge concern
- ▶ For a just-in-time (JIT) compiler, size of IR could be *very* significant

Aside: level of detail in high-level IR

- ▶ In designing a high-level linear IR, there is a question concerning how “detailed” the instructions should be
- ▶ More specifically, how close are the high-level IR instructions to operations in the source program?

Example: assigning to an array element

Consider the following C code:

```
a[i] = x;
```

Assume `vr10` is a pointer to the first element of `a`, `vr11` is the variable `i`, `vr12` is the variable `x`, and the elements of `a` are 8 bytes in size.

How to translate this statement into high-level IR?

Option 1: high-level

Translating `a[i] = x;`

```
mov_q      (vr10,vr11,8), vr12
```

This translation assumes the high-level IR has an indexed and scaled addressing mode for memory references.

Option 2: low-level

Translating `a[i] = x;`

```
mul_q    vr13, vr11, $8      /* compute element offset */
add_q    vr14, vr10, vr13    /* add offset to base address */
mov_q    (vr14), vr12
```

This translation does an explicit computation of the memory address of the element at index `i` (note that `vr13` and `vr14` are “temp registers” allocated to store partial results of the address computation)

Which approach is better?

Opinion: the low-level approach is better.

Making the high-level IR more complicated will make it more complicated to analyze and transform.

- ▶ It will also make the IR larger (in memory)

Techniques such as peephole optimization can be *very* effective for replacing explicit address computations with “fancy” addressing modes supported by the target language.

More generally, “simple and explicit” is good for earlier (higher-level) IR forms.

Control-flow graphs

Labels and control flow

The Instructions in an InstructionSequence can have labels, which can be referenced by control flow instructions.

Example C function:

```
int min(int a, int b) {  
    if (a < b)  
        return a;  
    else  
        return b;  
}
```

High-level IR code for min function

```
    enter    $0
    mov_l    vr10, vr1
    mov_l    vr11, vr2
    cmplt_l  vr12, vr10, vr11
    cjmp_f   vr12, .L1
    mov_l    vr0, vr10
    jmp      .Lmin_return
.L1:        mov_l    vr0, vr11
    jmp      .Lmin_return
.Lmin_return: leave    $0
    ret
```

Note: vr0 is the return value register, and vr1 and vr2 are argument registers

Observation

Analyzing and optimizing a sequence of instructions is complicated if there is control flow.

Idea: it's easier to analyze and transform “straight line” sequences of instructions

Control-flow graphs

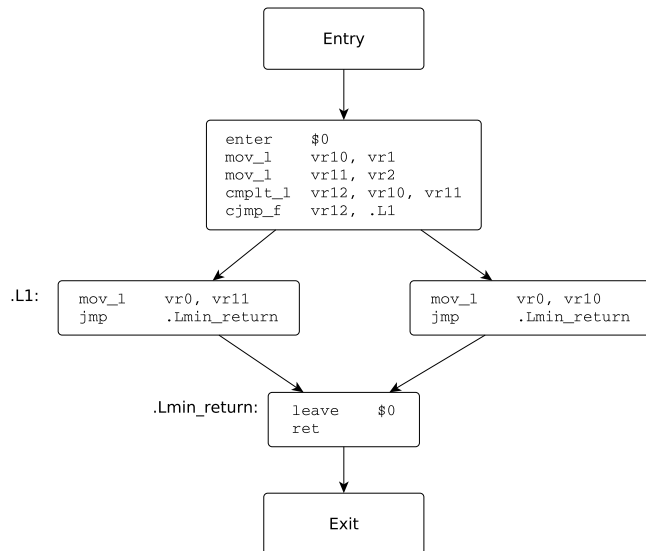
A *control-flow graph* is a graph of *basic blocks* representing one function in the program

A *basic block* is a sequence of instructions (e.g., quads) such that if there is a branch, it is the last instruction in the sequence

A control-flow graph should have a single *entry node* and a single *exit node*

- ▶ If the function has multiple return statements, each basic block ending with a return implicitly jumps to the common exit block

Example control-flow graph (min function)



Role of control-flow graphs

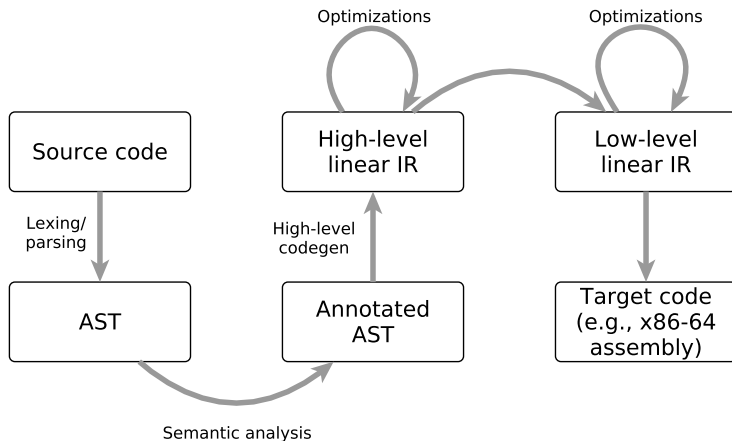
- ▶ Control-flow graphs allow for analyses and transformations which take control-flow into account
 - ▶ Especially: *dataflow* analyses
- ▶ A linear IR can be freely converted to and from a control-flow graph as necessary
- ▶ We'll have much more to say about control-flow graphs later on

IR forms and compiler phases

IR forms and compiler phases

- ▶ Different IR forms are appropriate at different points in the overall transformation from source code to target code
- ▶ The computations to make progress in the transformation are sometimes organized into “phases”
 - ▶ The process of moving towards the eventual target code representation is sometimes called “lowering”
- ▶ What these phases are called and what they do varies significantly from compiler to compiler

Possible organization



Note that optimizations will convert between linear IR and control-flow graphs as necessary

Are IRs necessary?

Utility and cost of IRs

- ▶ Intermediate representations are useful to allow analysis and transformation of code so that the quality of the generated code can be improved
- ▶ However, IRs can require significant memory
- ▶ If we're not too concerned about the absolute efficiency of the generated code, we could just generate it “on the fly”

“On the fly” translation

- ▶ Examples of on the fly codegen:
 - ▶ Tiny C compiler (tcc): <https://bellard.org/tcc/>
 - ▶ Some language virtual machines work this way when generating the initial translation of a function (e.g., JikesRVM's baseline compiler)
- ▶ *This approach can make sense if the goal is to generate code quickly*