Compiler phases (simplified)

Source text
  ↓ Lexing
Token stream
  ↓ Parsing
Abstract syntax tree
  ↓ Translation
Intermediate representation
  ↓ Code generation
Assembly
  ← Optimization
Syntax-directed translation

- Compilation strategy in which *syntax* of the program drives code generation
  - Assembly code generated from abstract syntax tree, or even directly by the parser
  - No substantial code analysis or transformation
- Demo: `sdt.ml`

- Easy to implement, but:
  - Produces inefficient code
  - Can be difficult to implement some language features (e.g., first-class functions)
  - Difficult to re-target compiler to new architectures
Syntax-directed translation

- Compilation strategy in which *syntax* of the program drives code generation
  - Assembly code generated from abstract syntax tree, or even directly by the parser
  - No substantial code analysis or transformation
- Demo: *sdt.ml*
- Easy to implement, but:
  - produces inefficient code
  - can be difficult to implement some language features (e.g., first-class functions)
  - difficult to re-target compiler to new architectures
Intermediate Representations
An intermediate representation (IR) breaks code generation up into two phases:

1. Translation from source language into IR
2. Generating target code from IR

Diagram:
- Abstract syntax tree
  - Translation
  - Intermediate representation
    - Code generation
      - Assembly
An intermediate representation (IR) breaks code generation up into two phases:
1. Translation from source language into IR
2. Generating target code from IR

Good level of abstraction at which to perform optimization
A simple let-based IR (let.ml)

\[ x = 2*(x + y) - (z * z) \]

1. Makes evaluation order explicit (no nested expressions)
2. Names all intermediate values (∼ unboundedly many “virtual” registers)
3. Distinguish between variables & intermediate values
Why use an IR?

- Appropriate abstraction level for machine-independent optimization
  - Simpler, lower-level than source language
  - Retain (some) information from source language that’s helpful for analysis & optimization
    - E.g., types, distinguish between writes to memory & computation of intermediate values

- Safety: IR can enforce maintenance of invariants (e.g. types)
- Reusability: IR can mediate between many source & target languages
- Saves the work of reimplementing optimization & code generation passes
Why use an IR?

- Appropriate abstraction level for machine-independent optimization
  - Simpler, lower-level than source language
  - Retain (some) information from source language that's helpful for analysis & optimization
    - E.g., types, distinguish between writes to memory & computation of intermediate values
- Safety: IR can enforce maintenance of invariants (e.g. types)
Why use an IR?

- Appropriate abstraction level for machine-independent optimization
  - Simpler, lower-level than source language
  - Retain (some) information from source language that's helpful for analysis & optimization
    - E.g., types, distinguish between writes to memory & computation of intermediate values
- Safety: IR can enforce maintenance of invariants (e.g. types)
- Reusability
  - IR can mediate between many source & target languages
  - Saves the work of reimplementing optimization & code generation passes
Reusability
What makes a good IR?

1. Convenient to translate source language to IR
2. Convenient to generate assembly from IR
3. Convenient to manipulate IR during optimization
   - Narrow interface ⇒ fewer cases to consider
What makes a good IR?

1. Convenient to translate source language to IR
2. Convenient to generate assembly from IR
3. Convenient to manipulate IR during optimization
   - Narrow interface $\Rightarrow$ fewer cases to consider
   - E.g., static single assignment (SSA) form enforces that there is exactly one assignment to any temporary (as in the let IR)
     - Safe to reorder instructions as long as no read/write dependency
     - Dead code analysis is more powerful
Varieties of IR

- In practice, compilers often use several IRs
  - GCC: Source → GENERIC → GIMPLE → RTL → Target

- High-level
  - Preserves high-level structures, but may simplify (e.g., convert for to do/while) or elaborate
  - Some high-level optimizations (e.g., function inlining)

- Mid-level
  - “Abstract assembly language”
    - Still retains some high-level features (e.g., explicit functions, variables, structured data)
  - Machine-independent optimizations

- Low-level
  - Machine-dependent optimizations