

Compiling and Programming Systems

CS 502

Fall 2008

Lecture 1

8/26/08

Goals

- Understanding the interaction between language design and implementation.
 - Framework for understanding superficially disparate structures
 - How much can we understand about a program without executing it?
 - Program analysis
 - How can we transform a program in one shape to an equivalent program of another shape?
 - Program optimization
 - What are the organizational principles that underlie these concepts?
 - Compiler implementation
-

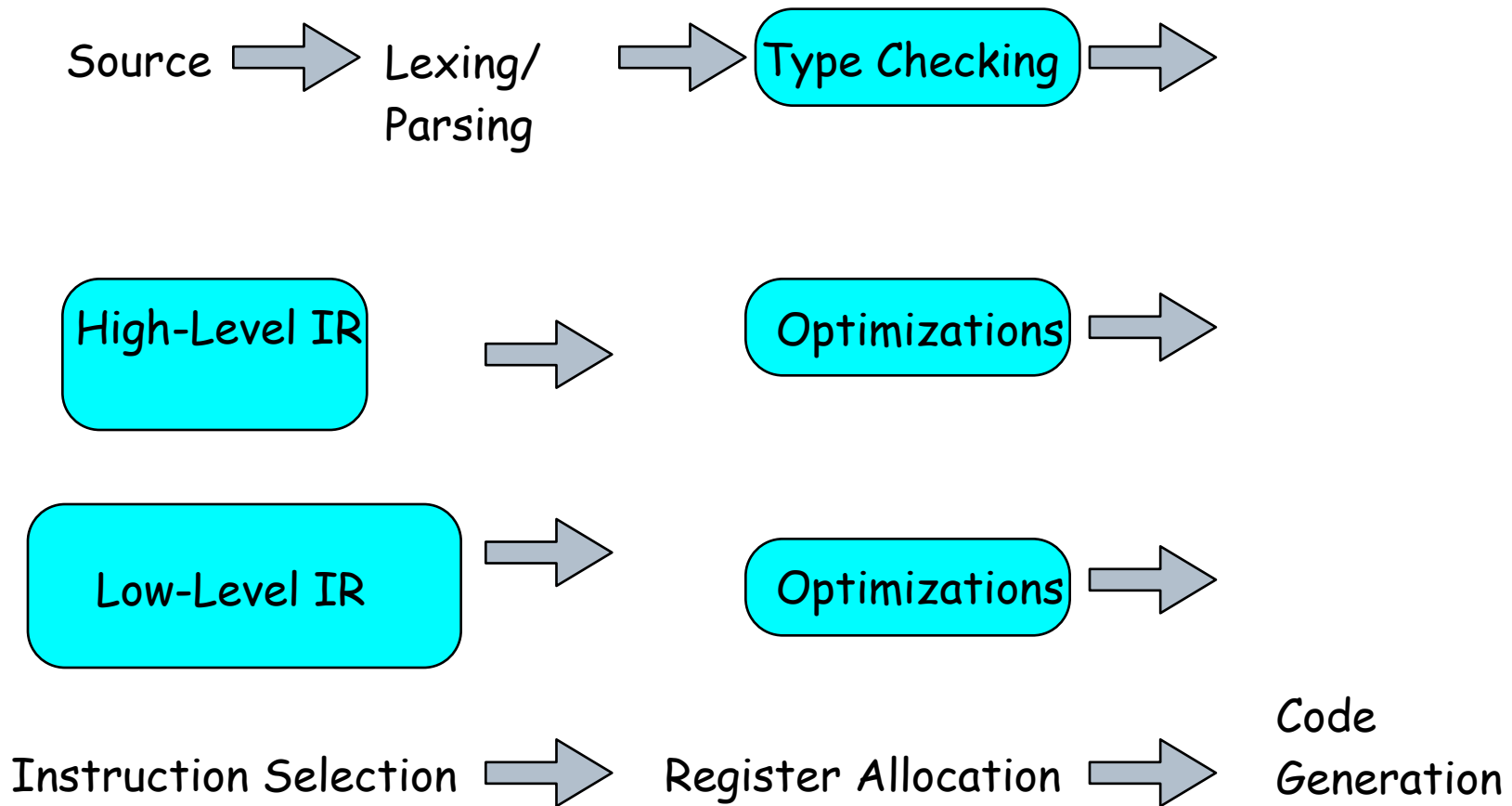
Essential Pre-requisites

- Some familiarity with programming language theory.
 - Sets, graphs, functions, relations ...
 - Must have taken an undergraduate compiler class or equivalent
 - E.g., must know how parsers work
 - Should be comfortable programming large projects.
 - We'll use Standard ML as the implementation language.
-

Organization

- Course divided into three parts (roughly):
 - Intermediate representations
 - Program optimizations
 - inter- and intra-procedural
 - Representation issues, virtual machine design, runtime support
 - Emphasis will be on the first two parts
 - Programming assignments for each part.
-

Structure of a Traditional Compiler



Phases

- Lexing: Turn source expressions into tokens; simple syntax checking.
 - Parsing: Convert input tokens into high-level intermediate representation; syntax and grammar checking.
 - Type-checking and static analysis
 - Error detection
 - High-level optimizations and transformations
-

Phases (cont)

- High-Level Intermediate Representation:
 - Explicit exposure of control and dataflow
 - Low-Level Intermediate Representation
 - More amenable to machine translation
 - Code generation
 - Compiler structure influenced both by language semantics and underlying VM/runtime organization
-

Interpreters vs. Compilers

- Question of degree
 - Number and kind of optimizations
 - Target language usually closer to source than machine architecture
 - Think of a spectrum of implementations:
 - JIT compilers for Java
-

Issues we'll look at ...

- What makes for a good intermediate representation?
 - The IR serves as the bridge between a high-level abstraction and its low-level incarnation
 - "High-level" --> good for humans, bad for machines
 - "Low-level" --> bad for humans, good for machines
 - How do we statically reason about a program?
 - What does a program mean?
 - Our interests are driven by pragmatics (e.g., optimization), not theory
 - How do we leverage this knowledge to improve implementation quality?
 - How do we map programs (i.e., an abstraction of an execution that typically assumes infinite resources) to a machine (i.e., a concretization of an execution with finite resources)
-

Transformational Compilation

- Compiler produces a series of semantics-preserving transformations on different intermediate languages.
 - The program associated with each IL must be semantically-equivalent to one another.
 - What does semantic equivalence mean?
 - How does this invariant limit the choice of IL, and the set of transformations that we perform?
-

Class project

- **Source language:** Mini-ML: a statically-typed higher-order mostly functional language that is a proper subset of ML
 - Provides integer lists, tuples, and records.
 - First-class procedures.
 - References (mutable locations)
 - **Goal:** Build a transformational optimizing compiler for mini-ML.
-

Implementation Language

Standard ML

- Strongly-typed mostly-functional language with rich module system
 - We'll provide module signatures(most of the time); you define implementations that satisfy these signatures
 - Modules correspond to:
 - Intermediate languages
 - Optimization phases
 - Supporting functionality
 - Hash tables, and other kinds of dictionaries
 - Error handling
 - I/O
-

Examples

Recursive Factorial:

```
fun fact(n:int): int =  
  if n = 0  
  then 1  
  else n * fact(n-1)
```

fact(5) (* returns 120 *)

Examples

Iterative factorial

```
fun fact(args: (int * int ref)): int =  
  (case args of  
    (0, a) => !a  
  | (n, x) => let val _ = x := (n * !x)  
              in fact(n-1,x)  
              end)
```

```
val n = ref 1  
val z = fact(5,n)
```

Examples

Yet another way to write factorial ...

```
fun fact(a:(int * (int -> int))):int =  
  (case a of  
    (0,k) => k 1  
  | (n,k) => let val k' = fn (z:int) => fact(z, fn (v:int) => k (n * v))  
              in k' (n - 1)  
              end)
```

```
val z = fact(5, fn (x:int) => x) (* returns 120 *)
```

What purpose does it serve to introduce these new internal functions?

Primary Goal

- Compile programs in these different styles.
 - Have generated code closely match (if not identical)
 - Key: Find good intermediate representations and transformations that unify superficially disparate programming styles and idioms
 - Result: More efficient, robust translator
-

Strategy

- Start with provided tokenizer and parser
 - Abstract syntax tree
 - Define a type checker
 - Apply simple transformations over abstract syntax:
 - Remove records
 - Eliminate patterns
 - Define control-flow representation
-

Strategy

- Build uniform representation for procedures and objects
 - Eliminate variable binding lookup
- Establish framework for understanding program optimizations
 - dataflow and control-flow
- Apply this framework
 - Jump elimination, inlining, dead-code elimination, common subexpression elimination, constant folding,

Strategy

- Translate program into form suitable for interpretation by an abstract virtual machine.
 - Apply low-level peephole optimizations
 - Register allocation
 - Code generation
-