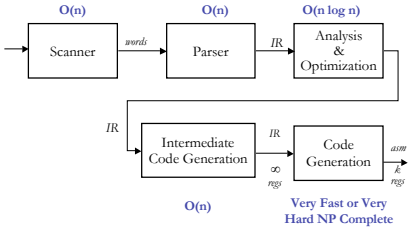
Intermediate Code Generation

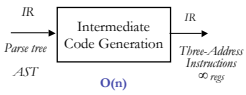
Introduction

Structure of a Compiler



A compiler is a lot of fast stuff followed by some hard problems
- The hard stuff is mostly in Code Generation and Optimization
- For super-scalars, its Allocation & Scheduling that counts

Intermediate Code Generation



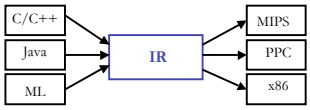
- Direct Translation
  - Using SDT scheme
  - Parse tree to Three-Address Instructions
  - Can be done while parsing in a single pass
  - Needs to be able to deal with Syntactic Errors and Recovery
- Indirect Translation
  - First validate parsing constructing of AST
  - Uses SDT scheme to build AST
  - Traverse the AST and generate Three Address Instructions

This Lecture

Same but Easier

Intermediate Representations

- Basic Idea:
  - Is target Machine independent
  - Common to many input languages



- What are the desired characteristics of an IR?
  - Represent and preserve the semantics of the input program
  - Amenable to Analysis and Transformations

Multi-Level IR?

- A Single IR is seldom the best solution for all uses…

- Multi-Level IR
  - High-Level IR close to AST
  - Low-Level IR close to machine instructions
  - Typically flows One-Way
- High-Level IR facilitates:
  - Source-to-Source Transformations
  - High-Level Program Analysis and Transformations (*e.g.,* loop unrolling)
- Low-Level IR facilitates:
  - Target Architecture Transformations & Analysis (*e.g.* register windows, predication, speculation)

Three-Address Instructions IR

- Construct mapped to Three-Address Instructions
  - Register-based IR for Expression Evaluation
  - Infinite Number of Virtual Registers
  - Still Independent of Target Architecture
  - Parameter Passing Discipline either on Stack or via Registers
- Addresses and Instructions
  - Symbolic Names are addresses of the corresponding source-level variable.
  - Various constants, such as numeric and offsets (known at compile time)
- Generic Instruction Format:

  `Label: x = y op z` or `if exp goto L`
  - Statements can have Symbolic Labels
  - Compiler inserts Temporary Variables (any variable with t prefix)
  - Type and Conversions dealt in other Phases of the Code Generation

Three-Address Instructions

- Assignments:
  - `x = y op z` (binary operator)
  - `x = op y` (unary)
  - `x = y` (copy)
  - `x = y[i]` and `x[i] = y` (array indexing assignments)
  - `x = phi y z` (Static Single Assignment instruction)
- Memory Operations:
  - `x = &y; x = *y` and `*x = y;` for assignments via pointer variables.

Three-Address Instructions

- Control Transfer and Function Calls:
  - `goto L` (unconditional);
  - `if (a relop b) goto L` (conditional) where `relop` is a relational operator consistent with the type of the variables `a` and `b`;
  - `y = call p, n` for a function or procedure call instruction to the name or variable p
    - p might be a variable holding a set of possible symbolic names (a function pointer)
    - the value n specifies that before this call there were n put`param` instructions to load the values of the arguments.
    - the `param x` instruction specifies a specific value in reverse order (i.e, the `param` instruction closest to the call is the first argument value.
    - Later we will talk about parameter passing disciplines (Run-Time Env.)

Function Call Example

Source Code	Three Address Instructions
⋮	⋮
`y = p(a, b+1)`	`t1 = a`
⋮	`t2 = b + 1`
`int p(x,z){`	`putparam t1`
`  return x+z;`	`putparam t2`
`}`	`y = call p, 2`
	⋮
	`p: getparam z`
	`  getparam x`
	`  t3 = x + z`
	`  return t3`
	`  return`

Loop Example

Source Code	Three Address Instructions
`do`	`L: t1 = i + 1`
`  i = i + 1;`	`  i = t1`
`while (a[i] < v);`	`  t2 = i * 8`
	`  t3 = a[t2]`
	`  if t3 < v goto L`

Loop Example

Source Code	Three Address Instructions
`do`	`L: t1 = i + 1`
`  i = i + 1;`	`  i = t1`
`while (a[i] < v);`	`  t2 = i * 8`
	`  t3 = a[t2]`
	`  if t3 < v goto L`

Where did this come from ?