CSE 582 – Compilers
Intermediate Representations
Hal Perkins
Autumn 2002

Agenda
- Parser Semantic Actions
- Intermediate Representations
  - Abstract Syntax Trees (ASTs)
  - Linear Representations
  - & more

Compiler Structure (review)
What’s a Parser to Do?

- Idea: at significant points in the parse perform a semantic action
  - Typically when a production is reduced (LR) or at a convenient point in the parse (LL)
- Typical semantic actions
  - Build (and return) a representation of the parsed chunk of the input (compiler)
  - Perform some sort of computation and return result (interpreter)

Intermediate Representations

- In most compilers, the parser builds an intermediate representation of the program
- Rest of the compiler transforms the IR for efficiency and eventually translates it to final code
  - May transform initial IR to a different IR at some point

IR Design

- Decisions affect speed and efficiency of the rest of the compiler
  - Desirable properties
    - Easy to generate
    - Easy to manipulate
    - Expressive
    - Appropriate level of abstraction
  - Different tradeoffs depending on compiler goals
    - Not necessarily the same in different parts of the same compiler
Types of IRs

- Three major categories
  - Structural
  - Linear
  - Hybrid

  Some basic examples now; more when we get to later phases of the compiler

Levels of Abstraction

- Key design decision: how much detail to expose
  - Affects possibility and profitability of various optimizations
  - Structural IRs are typically fairly high-level
  - Linear IRs are typically low-level
  - But this isn’t necessarily true

Example: Array Reference

\[ A[i,j] \]

- Load 1 => r1
- Sub rj, r1 => r2
- Load 10 => r3
- Mult r2, r3 => r4
- Sub ri, r1 => r5
- Add r4, r5 => r6
- Load @A => r7
- Add r7, r6 => r8
- Load r8 => r9
**Structural IRs**
- Typically reflect source (or other higher-level) language structure
- Tend to be large
- Examples: abstract syntax trees (ASTs), DAGs
- Particularly useful for source-to-source transformations

**Concrete Syntax Trees**
- Full grammar needed to guide parser, but contains many extraneous details
- Chain productions
- Rules that control precedence and associativity

**Syntax Tree Example**
- Concrete syntax for \( x = 2 \times (n + m) \);
Abstract Syntax Trees

- Want only essential structural information
- Omit extraneous junk
- Can be represented explicitly as a tree or in a linear form
  - Example: LISP/Scheme S-expressions are essentially ASTs

AST Example

- AST for \( x = 2^*(n+m); \)

Linear IRs

- Pseudo-code for an abstract machine
- Level of abstraction varies
- Simple, compact data structures
- Examples: stack machine code, three-address code
Stack Machine Code

- Originally used for stack-based computers (famous example: B5000)
- Now used for Java, C# (MSIL)

Advantages
- Compact; mostly 0-address opcodes
- Easy to generate
- Simple to translate to naïve machine code
- But need to do better in production compilers

Stack Code Example

Hypothetical code for \(x = 2 \times (n + m)\);
- pushaddr \(x\)
- pushconst 2
- pushval \(n\)
- pushval \(m\)
- add
- mult
- store

Three-Address code

- Many different representations
- General form: \(x \leftarrow y \text{op} z\)
  - One operator
  - Maximum of three names
- Example: \(x = 2 \times (n + m)\); becomes
  - \(t1 \leftarrow n + m\)
  - \(t2 \leftarrow 2 \times t1\)
  - \(x \leftarrow t2\)
Three Address Code (cont)

- Advantages
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Various representations
  - Quadruples, triples, SSA
  - Much more later...

Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Example: control-flow graph

What to Use?

- Common choice: all(!)
  - AST or other structural representation built by parser and used in early stages of the compiler
    - Closer to source code
    - Good for semantic analysis
    - Facilitates some higher-level optimizations
  - Flatten to linear IR for later stages of compiler
    - Closer to machine code
    - Exposes machine-related optimizations
    - Hybrid forms in optimization phases
Coming Attractions

- Representing ASTs
- Working with ASTs
  - Where do the algorithms go?
  - Is it really object-oriented?
  - Visitor pattern
- Then: semantic analysis, type checking, and symbol tables