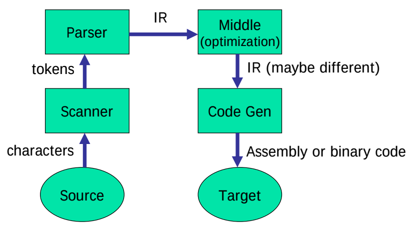
CSE P 501 – Compilers

Intermediate Representations
Hal Perkins
Summer 2004

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
  - Often will transform initial IR to one or more different IRs along the way

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
- Different tradeoffs in different parts of the same compiler

Types of IRs

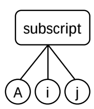
- Three major categories
  - Structural
  - Linear
  - Hybrid
- Some basic examples now; more when we get to later phases of the compiler

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 these generalizations aren't always true

Example: Array Reference

A[i,j]




```
loadI  1   => r1
sub  rj,r1  => r2
loadI  10  => r3
mult r2,r3 => r4
sub  ri,r1  => r5
add  r4,r5 => r6
loadI @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: syntax trees, DAGs
- Particularly useful for source-to-source transformations

Concrete Syntax Trees

- The full grammar is needed to guide the parser, but contains many extraneous details
  - Chain productions
  - Rules that control precedence and associativity
- Typically the full syntax tree does not need to be used explicitly

Syntax Tree Example

- Concrete syntax for x=2*(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 (.class files), 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*(n+m);

```
pushaddr   x
pushconst  2
pushval    n
pushval    m
add
mult
store
```


Three-Address code

- Many different representations
- General form: x <- y (op) z
  - One operator
  - Maximum of three names
- Example: x=2*(n+m); becomes

```
t1 <- n + m
t2 <- 2 * t1
x <- 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