CSE 401 – Compilers

Intermediate Representations
Hal Perkins
Winter 2010
Agenda

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

- We’re going to skip past LL parsing for the moment to keep the project on track.
Compiler Structure (review)

- **Source** → **Scanner** → **Parser** → **Middle (optimization)** → **Code Gen** → **Target**
  - Source: characters
  - Scanner: tokens
  - Parser: IR
  - Middle (optimization): IR (maybe different)
  - Code Gen: Assembly or binary code
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 to “improve” (optimize) it and eventually translates it to final code.
  - Often will transform initial IR to one or more different IRs along the way.
- Some general examples now; specific examples as we cover later topics.
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
IR Design Taxonomy

- Structure
  - Graphical (trees, DAGs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs)

- Abstraction Level
  - High-level, near to source language
  - Low-level, closer to machine
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 don’t necessarily hold
Examples: Array Reference

\[ A[i,j] \]

- \( \text{loadl } 1 \Rightarrow r1 \)
- \( \text{sub } rj, r1 \Rightarrow r2 \)
- \( \text{loadl } 10 \Rightarrow r3 \)
- \( \text{mult } r2, r3 \Rightarrow r4 \)
- \( \text{sub } ri, r1 \Rightarrow r5 \)
- \( \text{add } r4, r5 \Rightarrow r6 \)
- \( \text{loadl } @A \Rightarrow r7 \)
- \( \text{add } r7, r6 \Rightarrow r8 \)
- \( \text{load } r8 \Rightarrow r9 \)

or

\[ t1 \leftarrow A[i,j] \]
Structural IRs

- Typically reflect source (or other higher-level) language structure
- Tend to be large
- Examples: syntax trees, DAGs
- Generally used in early phases of compilers
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
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
- Common output from parser; used for static semantics (type checking, etc.) and high-level optimizations
  - Usually lowered for later compiler phases
ASTs in Java

- Basic idea is simple: use small classes as records (or structs) for nodes in the AST
  - Simple data structures, not too smart
- But also use a bit of inheritance so we can treat related nodes polymorphically
  - E.g., abstract AST class; extend to get generic classes for statements and expressions; extend those to get node types for specific kinds of statements and expressions
- Project details and survey of MiniJava AST classes in sections
To produce useful error messages, it’s helpful to record the source program location corresponding to a node in that node.

- Most scanner/parser generators have a hook for this, usually storing source position information in tokens.
- Included in the MiniJava starter code we distributed – useful to take advantage of it in your code.
AST Generation

- Idea: each time the parser recognizes a complete production, it produces as its result an AST node (with links to the subtrees that are the components of the production in its instance variables)

- When we finish parsing, the result of the goal symbol is the complete AST for the program
AST Generation in YACC/CUP

- A result type can be specified for each item in the grammar specification.
- Each parser rule can be annotated with a semantic action, which is just a piece of Java code that returns a value of the result type.
- The semantic action is executed when the rule is reduced.
Integrated tools like these can generate syntax trees automatically

- Advantage: saves work, don’t need to define AST classes and write semantic actions
- Disadvantage: generated trees might not have the right level of abstraction for what you want to do

For our project, do-it-yourself with CUP

- The starter code contains the AST classes from the minijava web site
Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
- Examples: three-address code, stack machine code
Abstraction Levels in Linear IR

- Linear IRs can also be close to the source language, very low-level, or somewhere in between.

- Example: Linear IRs for C array reference $a[i][j+2]$

  - High-level: $t1 \leftarrow a[i,j+1]$
IRs for a[i,j+2], cont.

- **Medium-level**
  - \( t1 \leftarrow j + 2 \)
  - \( t2 \leftarrow i \times 20 \)
  - \( t3 \leftarrow t1 + t2 \)
  - \( t4 \leftarrow 4 \times t3 \)
  - \( t5 \leftarrow \text{addr a} \)
  - \( t6 \leftarrow t5 + t4 \)
  - \( t7 \leftarrow *t6 \)

- **Low-level**
  - \( r1 \leftarrow [fp-4] \)
  - \( r2 \leftarrow r1 + 2 \)
  - \( r3 \leftarrow [fp-8] \)
  - \( r4 \leftarrow r3 \times 20 \)
  - \( r5 \leftarrow r4 + r2 \)
  - \( r6 \leftarrow 4 \times r5 \)
  - \( r7 \leftarrow \text{fp} - 216 \)
  - \( f1 \leftarrow [r7+r6] \)
Abstraction Level Tradeoffs

- High-level: good for source optimizations, semantic checking
- Low-level: need for good code generation and resource utilization in back end; many optimizing compilers work at this level for middle/back ends
- Medium-level: fine for optimization and most other middle/back-end purposes
Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Most common example: control-flow graph
  - Nodes: basic blocks – uninterrupted linear sequences of instructions
  - Edge from B1 to B2 if execution can flow from B1 to B2
  - More later when we survey optimization
What IR 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
  - Lower to linear IR for later stages of compiler
    - Closer to machine code
    - Exposes machine-related optimizations
    - Use to build control-flow graph
Coming Attractions

- Working with ASTs
  - Where do the algorithms go?
  - Is it really object-oriented? *(Does it matter?)*
  - Visitor pattern
- Then: Go back and look at LL (top-down) parsing
- After that: semantic analysis, type checking, and symbol tables