CSE P 501 – Compilers

Intermediate Representations

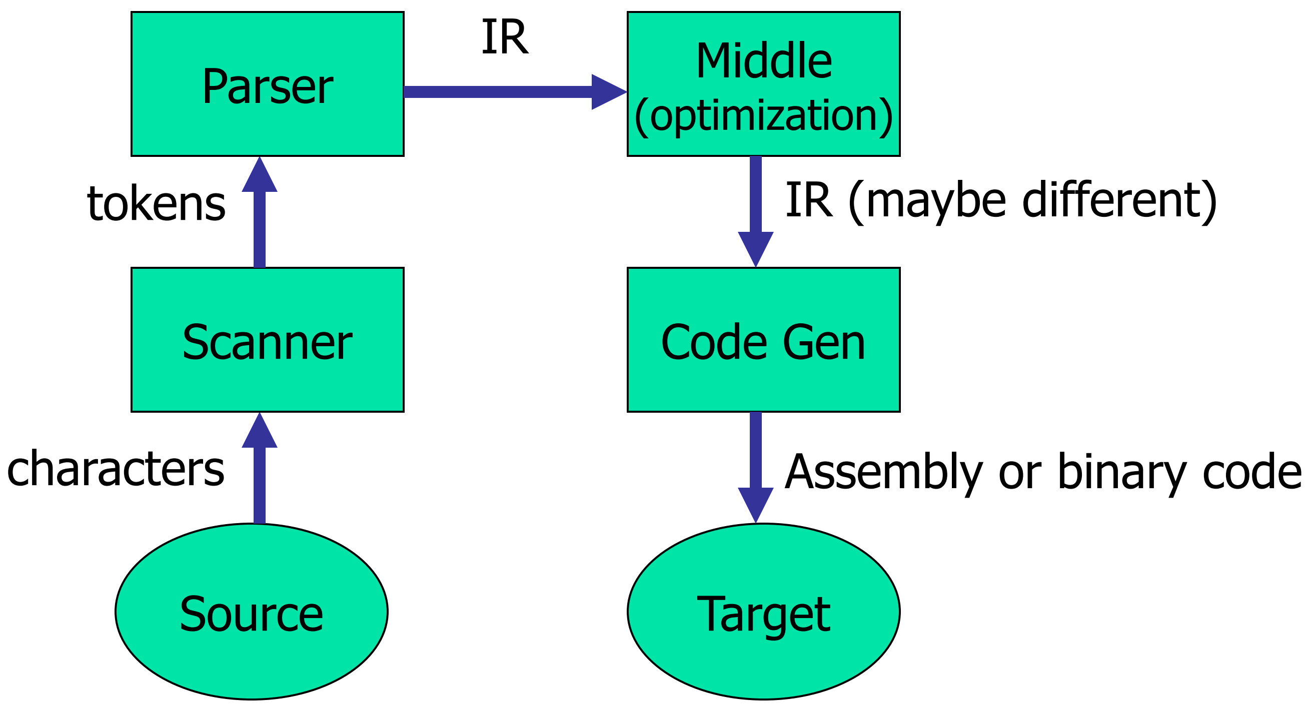
Hal Perkins

Autumn 2009

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 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

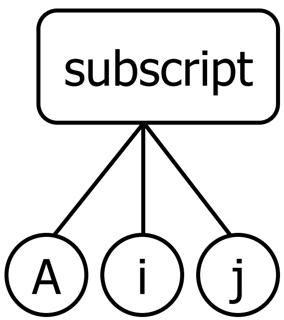
- Structure
  - Graphical (trees, DAGs, etc.)
  - Linear (code for some abstract machine)
  - Hybrids are common (e.g., control-flow graphs)
- Abstraction Level
  - High-level, near to source language
  - Low-level, closer to machine

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]



or

t1 ← 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
- 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

expr ::= *expr* + *term* | *expr* – *term* | *term*
term ::= *term* * *factor* | *term* / *factor* | *factor*
factor ::= *int* | *id* | (*expr*)

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
- Common output from parser; used for static semantics (type checking, etc.) and high-level optimizations
  - Usually lowered for later compiler phases

AST Example

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

Directed Acyclic Graphs

- DAGs are often used to identify common subexpressions
  - Not necessarily a primary representation, compiler might build dag then translate back after some code improvement
  - Leaves = operands
  - Interior nodes = operators

Expression DAG example

- DAG for $a + a * (b - c) + (b - c) * d$

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] (from Muchnick, sec. 4.2)

- High-level: $t1 \leftarrow a[i,j+1]$

IRs for a[i,j+2], cont.

- Medium-level

  t1 ← j + 2

  t2 ← i * 20

  t3 ← t1 + t2

  t4 ← 4 * t3

  t5 ← addr a

  t6 ← t5 + t4

  t7 ← *t6

- Low-level

  r1 ← [fp-4]

  r2 ← r1 + 2

  r3 ← [fp-8]

  r4 ← r3 * 20

  r5 ← r4 + r2

  r6 ← 4 * r5

  r7 ← fp – 216

  f1 ← [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

Three-Address code

- Usual form: $x \leftarrow y$ (op) z
  - One operator
  - Maximum of three names
- Example: x=2*(n+m); becomes

  $t1 \leftarrow n + m$

  $t2 \leftarrow 2 * t1$

  $x \leftarrow t2$

Three Address Code

- Advantages
  - Resembles code for actual machines
  - Explicitly names intermediate results
  - Compact
  - Often easy to rearrange
- Various representations
  - Quadruples, triples, SSA
  - We will see much more of this…

Stack Machine Code

- Originally used for stack-based computers (famous example: B5000)
- Now used for Java (.class files), C# (MSIL)
- Advantages
  - Very compact; mostly 0-address opcodes
  - Easy to generate
  - Simple to translate to machine code or interpret directly
    - And a good starting point for generating optimized code

Stack Code Example

- Hypothetical code for x=2*(n+m);

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

Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Most common example: control-flow graph
  - Nodes: basic blocks
  - Edge from B1 to B2 if execution can flow from B1 to B2

Basic Blocks

- Fundamental unit in IRs
- Definition: a *basic block* is a maximal sequence of instructions entered at the first instruction and exited at the last
  - i.e., if the first instruction is executed, all of them will be (modulo exceptions)

Identifying Basic Blocks

- Easy to do with a scan of the linear instruction stream
- A basic blocks begins at each instruction that is:
  - The beginning of a routine
  - The target of a branch
  - Immediately following a branch or return

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

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