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
- Tokens
- Characters
- IR
- Middle (optimization)
- IR (maybe different)
- Code Gen
- Assembly or binary code
- Target
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 always hold
Examples: Array Reference

\[
A[i,j]
\]

or

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

\[
\text{loadI } 1 \Rightarrow r1
\]
\[
\text{sub } r_j, r1 \Rightarrow r2
\]
\[
\text{loadI } 10 \Rightarrow r3
\]
\[
\text{mult } r_2, r_3 \Rightarrow r4
\]
\[
\text{sub } r_i, r1 \Rightarrow r5
\]
\[
\text{add } r_4, r_5 \Rightarrow r6
\]
\[
\text{loadI } @A \Rightarrow r7
\]
\[
\text{add } r_7, r_6 \Rightarrow r8
\]
\[
\text{load } r_8 \Rightarrow r9
\]
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
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
Linear IRs

- Pseudo-code for some abstract machine
- Level of abstraction varies
- Simple, compact data structures
  - Commonly used: arrays, linked 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: $t1 \leftarrow a[i,j+2]$
IRs for a[i,j+2], cont.

- **Medium-level**
  - \( t_1 \leftarrow j + 2 \)
  - \( t_2 \leftarrow i \ast 20 \)
  - \( t_3 \leftarrow t_1 + t_2 \)
  - \( t_4 \leftarrow 4 \ast t_3 \)
  - \( t_5 \leftarrow \text{addr a} \)
  - \( t_6 \leftarrow t_5 + t_4 \)
  - \( t_7 \leftarrow \ast t_6 \)

- **Low-level**
  - \( r_1 \leftarrow [\text{fp-4}] \)
  - \( r_2 \leftarrow r_1 + 2 \)
  - \( r_3 \leftarrow [\text{fp-8}] \)
  - \( r_4 \leftarrow r_3 \ast 20 \)
  - \( r_5 \leftarrow r_4 + r_2 \)
  - \( r_6 \leftarrow 4 \ast r_5 \)
  - \( r_7 \leftarrow \text{fp} - 216 \)
  - \( f_1 \leftarrow [r_7 + r_6] \)
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 – in section
  - 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