Administration

• Assignment 1
  – Due Wednesday January 17\textsuperscript{th} (tomorrow) at midnight

• Assignment 2
  – Do assignment 1 again, with visitors this time
  – Due Wednesday January 24\textsuperscript{th} at midnight
Visitors

• We are going to write a number of “passes” over a shared data structure (the parse tree)
  – Printing it
  – Finding definitions of identifiers
  – Type checking
  – IR code generation
There are two styles

• Add a method to each parse tree node for each pass
  – The book: object-oriented style
  – This keeps everything about the node in one place
  – But scatters the implementation of a pass all over the parse tree node classes

• Encode each pass in a class
  – The book: syntax separate from interpretation style
  – The traversal methods examine the class of each parse tree node and decide what to do
    
    ```java
    if n instanceof IntegerLiteral then
    ...
    else if n instanceof IdentifierExp then
    ...
    ```
We prefer passes in classes

- Gives us simple parse tree nodes
- Passes are all in one place
- And we can use the Visitor design pattern to avoid `instanceof`

- Each parse tree node implements
  - `R accept(Visitor v)`
    - which just calls `visit(v)`

- Each pass implements
  - `R visit(Node n)`
    - for every node type `Node`
    - where `R` is a common return type for every visit
Abusing Java generics

• And then we use Java generics to allow each visitor to return a different type

• Assignment 2 gives you a chance to implement this design pattern on the ABE interpreter from assignment 1
Practical Scanning and Parsing

• What do I really need to know about scanning and parsing to make it work?
  – Scanning?
    • Nothing
    • It “Just Works”
  – Parsing?
    • A few things
    • Limitations of the LL parsing technique
Not all grammars are LL(1)

• There are two common reasons why Top-down Recursive-descent, Predictive parsing won’t work for a particular grammar
  – Left recursion
  – Common prefixes
Left recursion

• If the first symbol on the RHS is the same as the symbol on the LHS this is called left recursion and is bad (for top down parsing)
  – \( E ::= T \mid E + T \)

• Why is it bad?

• How to fix it?

• Show tongue loading cup/lr-exp.cup
Common prefixes

• If two RHS for the same LHS have a common prefix, this is bad (for top-down parsing)
  – $S ::= \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S$
  – $P ::= \text{identifier} \mid \text{literal} \mid (E) \mid \text{identifier} (E)$

• Why is it bad?

• How to fix it?

• Show tongue loading cup/if-the-else.cup

• Show tongue loading cup/lr-exp-2.cup
LR parsing
Context-sensitive checking

- There are two kinds of context-sensitive checking
  - Use of identifiers
  - Type checking
Use of identifiers (symbols)

• Depending on the rules for your language, identifier checking takes either one pass or two
  – Expression language can be done in one pass
  – C can be done in one pass
  – Java requires two, why?
Symbol tables

• Symbol tables associate names found in programs with the thing named
  – In Expressions you can only name constants
    • All you need is the type of the constant and the location of the value (when it is computed)
  – In Java, you can name more things
    • ?

• We have to worry about multiple occurrences of the same name that name different things
  – But not in Expressions!
Symbol tables in the book

• Functional vs. imperative style symbol tables
  – This is essentially a religious argument
  – Both have their advantages and disadvantages
• The book worries about efficiency of lookup
• The “right” thing to do is probably:
  – create a separate symbol table for every scope (rather than for every binding)
  – have lookup traverse a tree of symbol tables