Syntax ➔Simple Semantics

Lecture 11

What do we do while parsing

- What semantic actions are plausible?
  - Mere recognition
  - Immediate execution
  - Generate Abstract Parse Trees

What does a rule $E \rightarrow E+T$ mean? (I)

- $E_1 \rightarrow E_0+T$ could mean "I will happily exchange my E, +, and T for an E, if it means I can determine if this input string is a sentence."

- $(E \rightarrow E + T$ #'(lambda(E P T) 't))
- in this model the result of parsing 3+4 is "true"

What does a rule $E \rightarrow E+T$ mean? (II)

- $E_1 \rightarrow E_0+T$ could mean "E_1 means: the result of taking whatever E_0 means, and whatever T means, and adding them together."

- (E $\rightarrow$ E + T #'(lambda(E P T)(+ E T)))

- In this model, the result of parsing 3 + 4 is 7

What does a rule $E \rightarrow E+T$ mean? (III)

- $E_1 \rightarrow E_0+T$ could mean "whatever E_0 means, and whatever T means, make a tree of them with + at the root."

- (E → E + T #'(lambda(E P T)(list '+ E T)))
- In this model, parsing 3 + 4 produces (+ 3 4).
- [this happens also to be a lisp program, but you could imagine some other tree structure, e.g. Tree{root="plus", left=Tree{root= "3" left=nil…, right=Tree{ ….} }]

What else might a rule mean?

- Arbitrary other semantics could be attached to a rule, to be applied when reducing E+T to E.
- For example
  - If E_0 and T are of type integer then E_1 must also be of type integer
- A rule typedeclare $\rightarrow$ type id might set up some information in memory to help make sense of a program's variables. In principle might not be in the tree.

What else might a rule mean?

- We could have a grammar with augments that formatted mathematical formulas...
  - Fraction $\rightarrow$ Num / Denom

```
#'(lambda (N s D)
(makevertpile N
                (hline (max (width N)(width D))
                 D))
```

Building a parse tree

```
S      → E $
E      → T EP
EP     → ADDOP T EP
EP     →
ADDOP  → +  |  -
T      →  F TP
TP     →  MULOP F TP
TP     →
MULOP  → *  |  /
F      → id  |  num  |  (E)
```

Sample = id + id + num * (num / num) $

Building a parse tree

;; (id + id + num * |(| num / num |)| $

```
(S (E (T (F id) (TP))
      (EP (addop +) (T (F id) (TP))
          (EP (addop +)
              (T (F num)
                 (TP (mulop *)
                     (T (F |(|
                            (E (T (F num) (TP (mulop
  /) (T (F num) (TP)) (TP)))
                               (EP))
                            |)|)
                        (TP))
                     (TP)))
              (EP))))
   $)
```

Building a more useful tree

;; (id + id + num * |(| num / num |)| $

```
(+ id id (* num (/ num num))) ; recdec315s
… just the terminals!!! instead of the clumsy previous tree

(S (E (T (F id) (TP))
      (EP (addop +) (T (F id) (TP))
          (EP (addop +)
              ….  ;; all these pointless reductions in tree
```

What kind of tree should we build for a MiniJava program?

- Cheap trick: a tree that is an equivalent Lisp program.
- Cheap trick: a tree that if printed in infix order is a C program that could be compiled and run.
- Really cheap trick: a Java program ☺
- More typical treatment:
  - A symbol table structure listing all the types, variables, functions (in all their different scopes)
  - With all the function bodies or other executable components defined as trees
  - Use above data to do type checking.

Abstract Syntax Tree Example. Here is a MJ program (BinarySearch.java)

```
class BinarySearch{
   public static void main(String[] a){
     System.out.println(new BS().Start(20));
   }
}
// This class contains an array of integers and
// methods to initialize, print and search the array
// using Binary Search

class BS{
   int[] number ;
   int size ;

   // Invoke methods to initialize, print and search
   // for elements on the array
   public int Start(int sz){
     int aux01 ;
     int aux02 ;
```

Abstract Syntax Tree Example. Here is a MJ parser OUTPUT (BinarySearch.java)

```
(Program  ;; this has line numbers only, output of simple-
   ;;lex, simple-parse
 (MainClass (id BinarySearch 1) (id a 2)
  (Print
   (Call (NewObject (id BS 3)) (id Start 3) (ExpList
  (IntegerLiteral 20)))))
 (ClassDecls
  (ClassDecl (id BS 10) (extends nil)
   (VarDecls (VarDecl IntArrayType (id number 11))
    (VarDecl IntType (id size 12)))
   (MethodDecls
    (MethodDecl IntType (id Start 16) (Formals (Formal
  IntType (id sz 16)))
     (VarDecls (VarDecl IntType (id aux01 17)) ……
```

This is really rather clumsy to look at

- But at least we can look at it without writing another program (not like Java).
- We have removed line/column markers in many places, but left some in there so that we still can identify some errors by location.
- The next pass can be like our CS61a analyzing meta-circular evaluator (remember that ☺?)

Just as a preview

- If we see a binary arithmetic operator, we can hope to look up the declared types of the arguments to it (if ids). Or the computed types of the arguments to it (if expressions).
- For more ambitious languages, a<b is legal when a comparison is valid (e.g. strings, floats, ints and floats, single and double). Or a+b makes sense when one or both are strings.

- The primary issues:
  - knowing types of objects (in scopes..)
  - Knowing what's legal to do with the objects.