Today

- Type checking (cont)
- Interpretation (intro)

Assignments

- R-5, HW-5 out (due Thurs)
- R-6, HW-6 out (due next Thurs)
Type Checking ... HW 5

The goal of type checking is to:

- Detect errors due to type issues, e.g.:
  
  ```
  x = 0 + "1"; # assuming + requires two numbers
  if 42 <= true then x = 1; end # int and bool comparison
  ```

- Also includes “use before def” errors, e.g.:
  
  ```
  x = 42 + y; # y not defined
  if x < 10 then
    y = x + 1;
  else
    x = y; # y not defined in this block
  end
  ```

Type errors are based on a set of typing rules

- lots of possibilities, different languages have different rules
- we define a set of “strict” typing rules in HW4 (informally)
Basic Approach

For HW 5 ...

• navigate the AST using the Visitor pattern
• as we go, we infer types and look for type errors
• keep track of last inferred type using a “current type” variable ...

Given this code ...

```plaintext
x = 3;
if x < 25 then
    println(x);
end
```

1. Type check the assignment statement
   (a) Type check the RHS (current type assigned to INT)
   (b) Since x isn’t defined, assign INT as x’s type

2. Type check the if statement
   (a) Type check the conditional (both operands are of type INT)
   (b) Ensure each statement in the body is type safe

Note we have to keep track of variables and their types!
• we do this using a “symbol table” (var -> type mappings)
Complex Expression Example ...

AST fragment for the complex expression $3 + 4 \times 5$

```
c1: ComplexExpr
   s1: SimpleExpr
      t1: Token
         lexeme = "3"
         tokentype = INT
   t2: Token
      lexeme = "+
      tokentype = PLUS
c2: ComplexExpr
   s2: SimpleExpr
      t3: Token
         lexeme = "4"
         tokentype = INT
c3: ComplexExpr
   t4: Token
      lexeme = "5"
      tokentype = INT
```

The basic steps:

1. `visit_complex_expr(c1)` is called
2. call `visit_complex_expr(c2)`, which sets `current_type` to `INT`
3. store `current_type` in temporary var `right_hand_type`
4. call `visit_simple_expr(s1)`, which sets `current_type` to `INT`
5. check that `current_type == right_hand_type`
6. set `current_type` to `INT` (really, leave unchanged)

* Note that `visit()` functions called indirectly through `accept()`

Exercise: What happens in `visit_complex_expr(c2)` call?
Simple Assignment Example ...

AST fragment for the simple assignment $x = 3$;

The basic steps:
1. visit\_assign\_stmt(a1) is called
2. call visit\_simple\_expr(s1), which sets current\_type to INT
3. check if $x$ is in sym\_table
4. if not, add $x \rightarrow \text{INT}$ to sym\_table
5. otherwise, get $x$’s type from symbol table and check if equal to current\_type

* Note that visit() functions called indirectly through accept()
Interpretation ... HW 6

Writing a “pure interpreter” for MyPL:

- Overall very similar to type checking ... in fact, “easier”
- We’ll again use the visitor pattern ... over AST nodes
- We’ll also use the symbol table ... this time, var -> value
- Instead of a current type, we’ll keep track of the current value
Complex Expression Example ...

AST fragment for the complex expression $3 + 4 \times 5$

The basic steps:

1. visit_complex_expr(c1) is called
2. call visit_complex_expr(c2), which sets current_value to 20
3. store current_value in temporary var right_hand_value
4. call visit_simple_expr(s1), which sets current_value to 3
5. use math_rel to compute new value 23
6. set current_value to 23

* Note that visit() functions called indirectly through accept()

Exercise: What happens in visit_complex_expr(c2) call?
Simple Assignment Example ...

AST fragment for the simple assignment $x = 3$;

The basic steps:

1. visit_assign_stmt(a1) is called
2. call visit_simple_expr(s1), which sets current_value to 3
3. add $x$ to sym_table if not already there
4. set $x$'s value in sym_table to current_value

* Note that visit() functions called indirectly through accept()