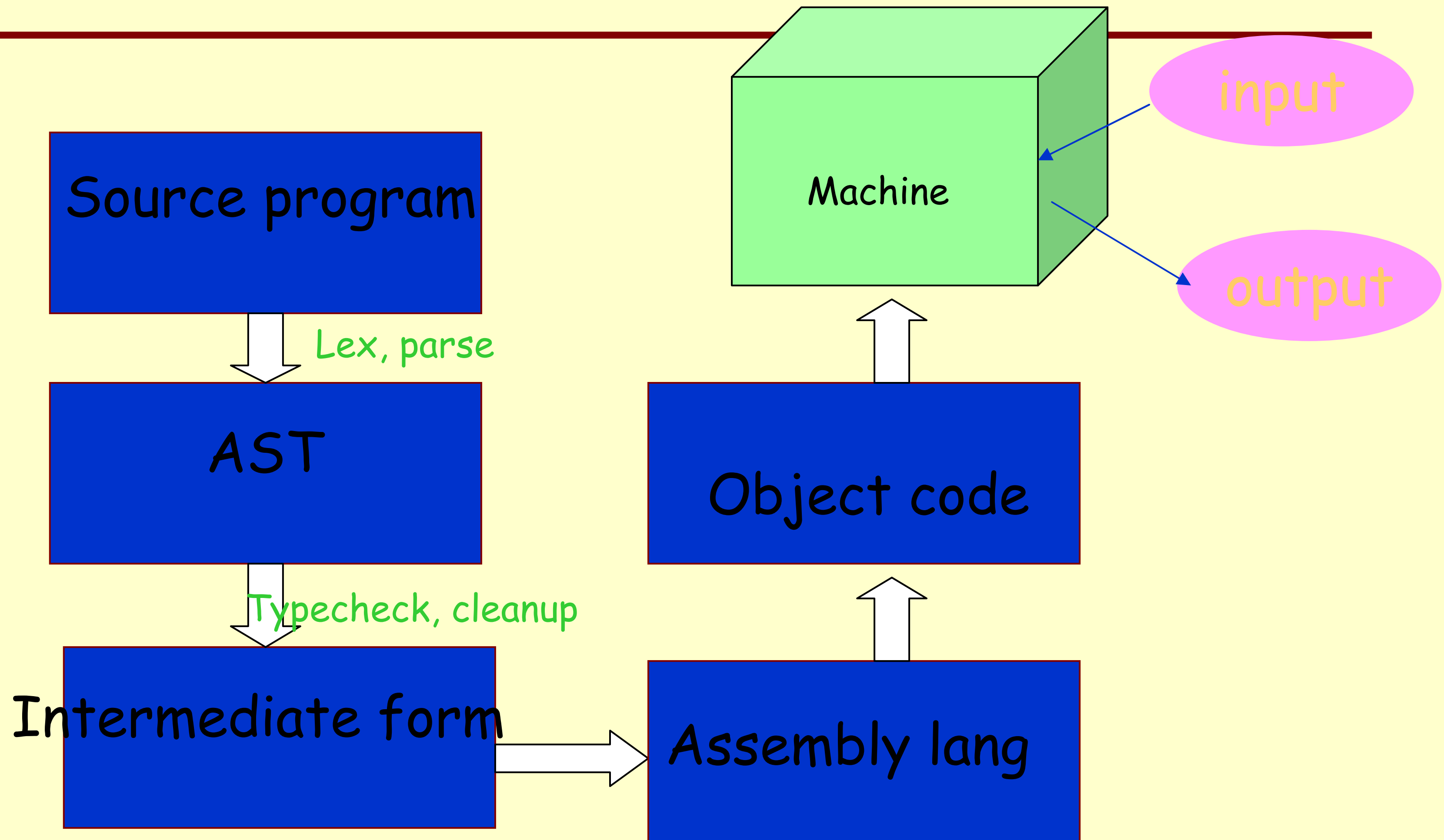
Language definition by interpreter

Lecture 14

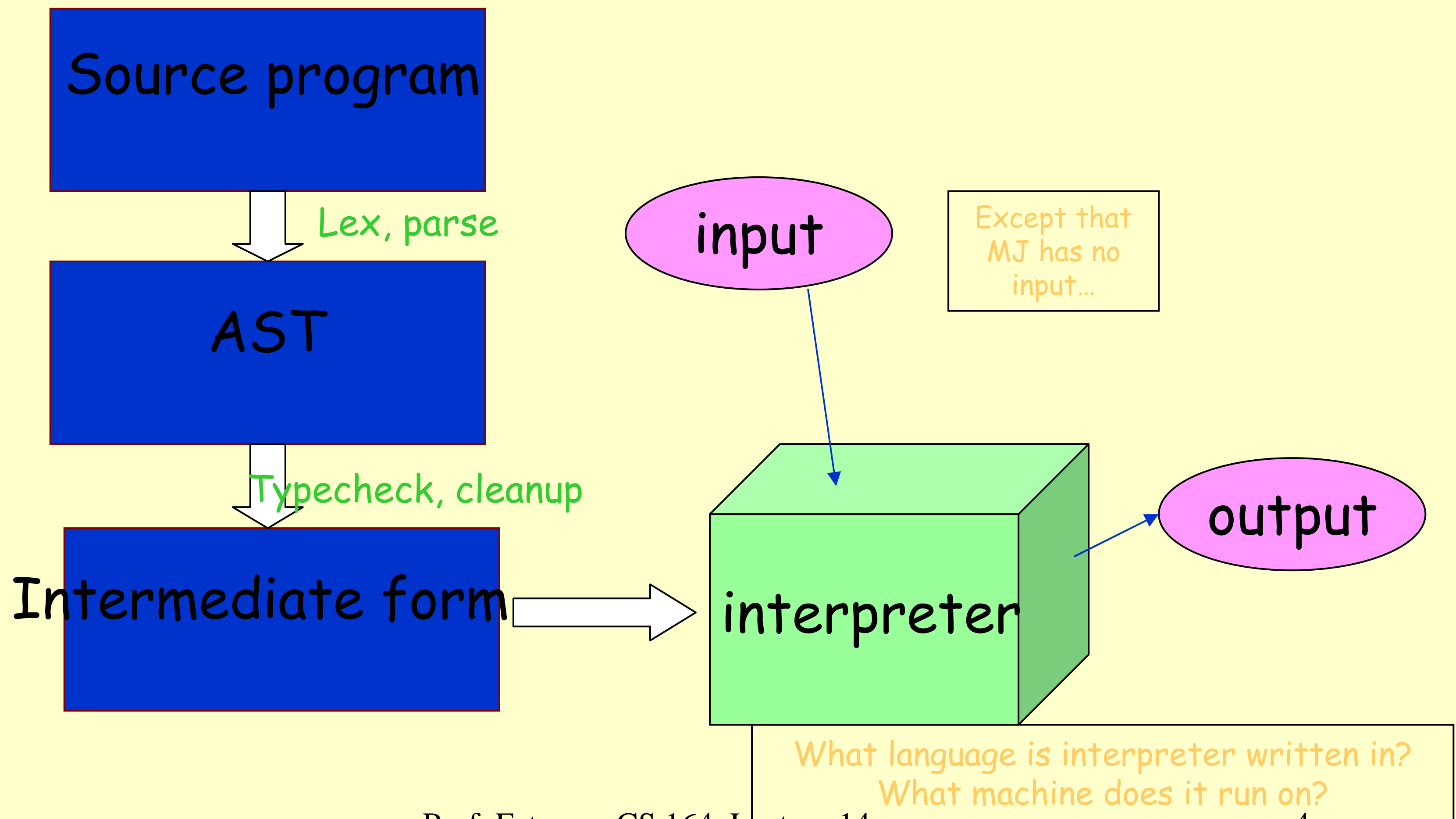
Routes to defining a language

- Formal mathematics
  - Context free grammar for syntax
  - Mathematical semantics (axioms, theorems, proofs)
  - Rarely used alone (the downfall of Algol 68)
  - Can be used to verify/prove properties (with difficulty)
- Informal textual
  - CFG + natural language (Algol 60, Java Language Spec)
  - Just natural language (Visual Basic for Dummies) -- examples
  - Almost universally used
- Operational
  - Here's a program that does the job
  - Metacircular evaluator for Scheme, Lisp
  - Evaluator/ interpreter for MiniJava

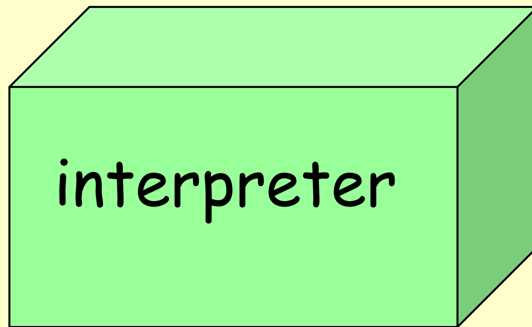
Typical compiler structure



"MiniJava run" structure

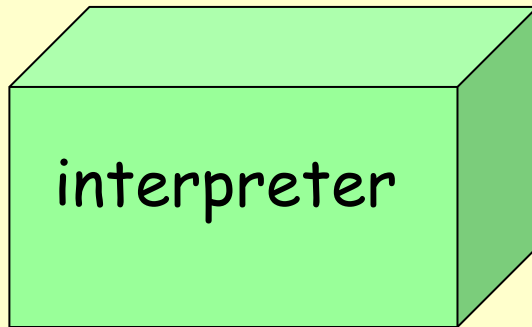


Interpreter structure: advantages



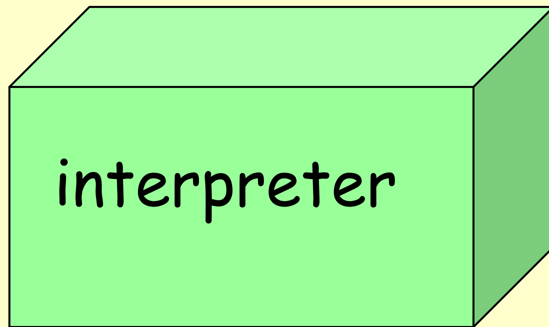
- Interpreter is written in a higher level language: source language derives semantics from interpreter program and the semantics of the language of the interpreter (e.g. whatever it is that "+" does in Lisp).
  - What does EACH STATEMENT MEAN?
    - Exhaustive case analysis
  - What are the boundaries of legal semantics?
  - What exactly is the model of scope, etc..

Interpreter structure: more advantages



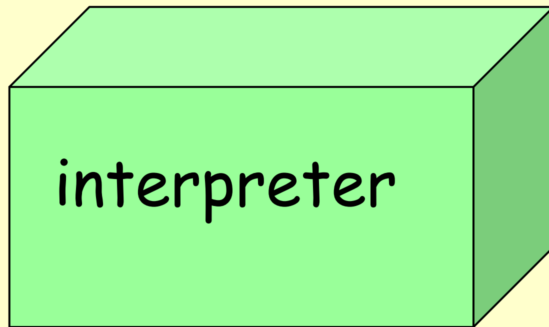
- Prototyping / debugging easier (compare to machine level)
- Portable intermediate form; here AST in Lisp as text; could be byte code
- intermediate form may be compact
- Security may be more easily enforced by restricting the interpreter or machine model [e.g. if Lisp were "safe", so would the interpreter be safe.]
- In modern scripting applications, most time is spent in library subroutines so speed is not usually an issue.

Interpreter structure: disadvantages



- Typically unable to reach full machine speed. Repeatedly checking stuff
- Difficult to transcend the limits of the underlying language implementation (not full access to machine: if interpreter is in Lisp, then "interrupts" are viewed through Lisp's eyes. If interpreter is in Java, then Java VM presents restrictions.)
- Code depends on presence of infrastructure (all of Lisp??) so even a tiny program starts out "big". [Historically, was more of an issue]
- (if meta-circular) bootstrapping… (digression on first PL)

An interpreter compromise (e.g. Java VM)



- "Compile" to a hypothetical byte-code stack machine appropriate for Java and maybe some other languages, easily simulated on any real machine.
- Implement this virtual byte-code stack machine on every machine of interest.
- When speed is an issue, try Just In Time compiling; convert sections of code to machine language for a specific machine. Or translate Java to C or other target.

IMPORTANT OBSERVATION

- Much of the work that you do interpreting has a corresponding kind of activity that you do in typechecking or compiling.
- This is why I prefer teaching CS164 by first showing a detailed interpreter for the target language.

Interpreter to TypeChecker / Static Analysis is a small step

- Modest modification of an interpreter program can result in a new program which is a typechecker. An interpreter has to figure out VALUES and COMPUTE with them. A typechecker has to figure out TYPES and check their validity.
- For example:
- Interpreter: To evaluate a sequence $\{s_1, s_2\}$, evaluate s_1 then evaluate s_2, returning the last of these.
- Typechecker: To TC a sequence $\{s_1, s_2\}$, TC s_1 then TC s_2, returning the type for s_2.
- Interpreter: To evaluate a sum (+ A B) evaluate A, evaluate B and add.
- Typechecker: To TC a sum, (+ A B) TC A to an int, TC B to an int, Then, return the type, i.e. Int.
- Program structure is a walk over the AST.

How large is MJ typechecker / semantics ?

- Environment setup code for MJ is about 318 lines of code.
- Simple interpreter, including all environment setup, is additional 290 lines of code, including comments.
- Add to this file the code needed for type checking and you end up with an extra 326 lines of code.
- Environment setup would be smaller if we didn't anticipate type checking.

Interpreter to Compiler is a small step

- Modest modification of an interpreter program can result in a new program which is a compiler.
- For example:
- Interpreter: To evaluate a sequence $\{s_1, s_2\}$, evaluate s_1 then evaluate s_2, returning the last of these.
- Compiler: To compile a sequence $\{s_1, s_2\}$, compile s_1 then compile s_2, returning the concatenation of the code for s_1 and the code for s_2.
- Interpreter: To evaluate a sum (+ A B) evaluate A, evaluate B and add.
- Compiler: To compile a sum, (+ A B) compile A, compile B, concatenate code-sequences. Then, compile + to "add the results of the two previous sections of code". And concatenate that.
- Program structure is a walk over the intermediate code AST.

AST for MJ program

- AST is a data structure presenting the Program and all subparts in a big tree reflecting ALL parsed constructs of the system.
- Here's fact.java's AST with some parts abbreviated with #.

```
(Program
 (MainClass (id Factorial 1) (id a 2)
  (Print (Call (NewObject #) (id ComputeFac 3)
   (ExpList #))))
 (ClassDecls
  (ClassDecl (id Fac 7) (extends nil) (VarDecls)
   (MethodDecls (MethodDecl IntType # # # # #)))))
```

Start of the interpreter

```
(defun mj-run (ast)
  "Interpret a MiniJava program from the AST"
  (mj-statement (fourth (second ast)) ;; the body
                      (setup-mj-env ast) ;; set up env.
))
```

```
(Program
 (MainClass (id Factorial 1) (id a 2)
  (Print (Call (NewObject #) (id ComputeFac 3) (ExpList #))))
 (ClassDecls
  (ClassDecl (id Fac 7) (extends nil) (VarDecls)
   (MethodDecls (MethodDecl IntType # # # #))))); define ComputeFac
```

MJ statements

```
(defun mj-statement (ast env)
  "Execute a MiniJava statement"
  ;; we do a few of these in-line, defer others to subroutines.  They could all be subroutines..
    (cond
      ((eq (car ast) 'If)         (if (eq (mj-exp-eval (cadr ast)env) 'true)
                                      (mj-statement (caddr ast) env) ;;then
                                      (mj-statement (cadddr ast) env))) ;;else
      ((eq (car ast) 'While)       (mj-while ast env))  ;; um, do this on another slide
      ((eq (car ast) 'Print)       (format t "~A~%" (mj-exp-eval (cadr ast))))
      ((eq (car ast) 'Assign)      (mj-set-value (id-name (second ast)) ;; look at this later
                                            (mj-exp-eval (caddr ast) env)))
      ((eq (car ast) 'ArrayAssign)    ;; etc
     ((eq (car ast)  'Block)       (dolist (s (cdadr ast)) (mj-statement s env)))
;;;;  add other statement types in here, if we extend MJ
      (t
       (pprint ast)
       (error "Unexpected statement")))))
```

What else is needed?

- All the functions (mj-while etc etc) [simple]
- All the supporting data structures (symbol table entries and their organization) [tricky]

if

COMPARE….

((eq (car ast) 'If)

```
(if (eq (mj-exp-eval (cadr ast)env) 'true)
                                  (mj-statement (caddr ast) env) ;;then
                                  (mj-statement (cadddr ast) env))) ;;else
```

```
(mj-if  (mj-exp-eval (cadr ast)env)
        (mj-statement (caddr ast) env) ;;then
        (mj-statement (cadddr ast) env))) ;;else

(defun mj-if(a b c)(if a b c))
```

2 problems: in Lisp, (if X Y Z) evaluates X. If X is non-nil, returns value of Y.
MJ's false is non-nil.
Also, in the call to mj-if, we have evaluated both branches. We lose.

Loops: While is very much like Lisp's while

```
((eq (car ast) 'While)
  (while (eq (mj-exp-eval (cadr ast) env) 'true)
     (mj-statement (caddr ast) env)))
```

Also part of the main interpreter: exp-eval

```
(defun mj-exp-eval (ast env)
 "Evaluate a Mini-Java expression subtree"
 (labels
     ((c  (v) (eq (car ast) v))  ;; some shorthands (DSB idea!)
      (e1 ()  (mj-exp-eval (second ast) env))
      (e2 ()  (mj-exp-eval (third  ast) env)))
   (cond
     ((eq ast 'this)       (mj-this env))
     ((atom ast)           (error "Unexpected atomic expression"))
     ((c 'Not)             (mj-not (e1)))
     ((c 'And)             (mj-and (second ast) (third ast) env))
     ((c 'Plus)            (mj-+   (e1) (e2))); also Times, Minus, LessThan
     ((c 'IntegerLiteral) (second ast)) ;also BooleanLiteral
     ((c 'ArrayLookup)     (elt (e1) (e2)))
     ((c 'ArrayLength)     (length (e1)))
     ((c 'NewArray)        (make-array `(,(e1)) :initial-element 0))
     ((c 'NewObject)       (mj-new-object (id-name (second (second ast)))
  env))
     ((c 'Call)            (mj-call ast env)) ;; REALLY IMPORTANT
     ((c 'IdentifierExp)   (mj-get-value (id-name (second ast)) env)))))
```

Revisit the statement interpreter

- (labels
- ((c (v) (eq (car ast) v))
- (e (i) (mj-exp-eval (nth i ast) env))
- (s (i) (mj-statement (nth i ast) env)))

```
 cond
     ((c 'Assign)     (mj-set-value (id-name (second ast))
                               (e 2) env))
     ((c 'ArrayAssign) (setf (elt (mj-get-value (id-name (second ast))
env)
                            (e 2))
                      (e 3)))
```

(looking at code for simple-interp)

- [no more slides of this]