

CS 42I Lecture 20 – Dynamically-typed languages

- ▶ Static vs. dynamic typing
- ▶ Implementation
- ▶ Example: “XL”
 - ▶ Overview
 - ▶ Implementation – front-end and back-end
- ▶ Efficiency



Dynamically-typed languages

Compiler does not check programs for possible run-time type errors; they are detected at run time.

1960: Lisp, APL

1970's: Smalltalk

1990's: Perl, Python, JavaScript, Ruby

Used mainly for “scripting” – i.e. small programs

Advantage: convenience

Disadvantage: inefficiency; no type-checking



Static vs. dynamic typing

Static typing: Programs are checked before execution to determine the type of each expression. Thus, in an expression $e_1 \oplus e_2$, it is known before execution exactly what operation \oplus is. (E.g. “+” in Java)

Dynamic typing: Programs are not checked. At run time, values have associated type tags, and these are used to determine what operation is meant. (E.g. “+” in Python).

Important distinctions:

- Types of expressions vs. types of values
 - Static types vs. explicit types
-



Why static typing?

- Types are a “sanity check” on program.
- Since operations can be selected at compile time, more efficient.



Why dynamic typing?

- Convenience

- E.g. can use lists to represent all kinds of structured data, without needing additional type declarations.
- Most dynamically-typed languages have libraries – e.g. reg. expr.'s, graphics, I/O – that are much easier to use than typical APIs of statically-typed languages.



Example: LISP (“LISt-Processing”)

- Fully parenthesized notation
- Values: numbers, symbols, pairs, “nil”, lists
(Technically, lists not a separate type: they are defined as either nil or pairs whose second element is a list; special notation is provided: (a b ... c) => (a, (b, ... (c, nil)...)).)
- Regular control structures, but mainly use recursion.
- Note that lists are heterogeneous; this makes static type-checking impossible.



Example: Python

- Lisp w/o parentheses
- Data types: numbers, strings, pairs, (heterogeneous) lists
- Syntax: “Ordinary,” but uses indentation for statement grouping (instead of braces), e.g.

```
def find (x, lis):  
    if lis == []:  
        return False  
    elif x == lis[0]:  
        return True  
    else:  
        return find(x, lis[1:])
```

- Lots of helpful libraries
-



Implementation of programming languages

Three methods:

- Interpret (execute directly off AST)
- Compile to virtual machine and interpret VM code
- Compile to target machine code

Can use any method for any language, but *typically*:

- Compile to machine code for highest efficiency; sacrifice portability; used for conventional, statically-typed lang's
- Compile to VM code when efficiency is less important; lose efficiency; gain portability; used for dynamically-typed languages and "managed" languages (Java, C#)
- Interpret mainly for simplicity of implementation; lose efficiency

Dynamically-typed languages not used when efficiency is main goal, so compilation to VM most common.



Example: “XL”

Extension language – designed to write small scripts to control applications in the “Slice” system.

Still in development – “pre-alpha”


TL;DR: Dynamically-typed Java, w/ built-in notation for lists; w/o objects (only static methods)



Examples

```
void StrokeAdd () {
    currslide = StrokeNode.GetParent();
    laser = Root["LaserPointer"];
    if (laser == "On") {
        laserstroke = currslide.FindChildByAttribute("LaserStroke",
                                                    "True");

        if (laserstroke != null)
            laserstroke.Remove();
        StrokeNode["LaserStroke"] = "True";
    }
    StrokeNode["Author"] = Root["UserName"];
    StrokeNode["AuthorRole"] = Root["UserRole"];
}
```



Implementation

Written in Java (but easily transformable to C#)

Front end:

- Lexer and parser generated by hand-written generators

- Hand-written translation from concrete to abstract syntax

- No type-checking (at present), so no symbol table

Back end:

- Direct interpretation of AST



Front end – lexer generator

Lexer generator:

- Based on description of DFA
- DFA description about 100 lines; code-generating code about 50 lines
- Sample of DFA – part of comment section:

```
[slash, Slash, [[oneOf("*"), inCcomment], [oneOf("/"), inCPcomment],
                [oneOf("="), slasheq]]],
[slasheq, SlashEqual, []],
[inCcomment, Error, [[notOneOf("*"), inCcomment],
                    [oneOf("*"), inCcomment2]]],
[inCcomment2, Error, [[notOneOf("*/"), inCcomment],
                    [oneOf("*"), inCcomment2],
                    [oneOf("/"), endcomment]]],
[endcomment, Discard, []]
```



Front end – parser generator

Generates recursive-descent parser, using LL(1)

- LL(1) test and generator written in 200 lines of Python
- Grammar has ≈ 150 productions (simplified from Java)
- To handle two-symbol lookahead, when needed, can insert a predicate to help determine production to use.
- Sample from grammar: productions from non-terminal “statement”:



Front end – parser generator

```
[statement, [  
  [block],  
  [["toklis.peek(0) == tokens.Colon"], Identifier, Colon, statement],  
  [If, parExpression, statement, elseStmtOpt],  
  [For, Lparen, forstatement],  
  [While, parExpression, statement],  
  [Switch, parExpression, Lbrace, caseStmts, Rbrace],  
  [Return, expressionOpt, Semicolon],  
  [Throw, expression, Semicolon],  
  [Break, identifierOpt, Semicolon],  
  [Continue, identifierOpt, Semicolon],  
  [expression, Semicolon],  
  [Semicolon]  
]]
```



CST \Rightarrow AST

Hand-written translation.

AST is generic tree type, where each node has a name, i.e. abstract syntax operator, and then either a list of children or a token value (e.g. integer constant).

The AST operators are:

`compUnit, classDecl, method, formals, var, stmtlist, exprstmt, vardecl, ifstmt, whilestmt, returnstmt, throwstmt, breakstmt, continuestmt, switchstmt, casestmt, exprlist, ident, unarypreoptr, unarypostoptr, binaryop, condexpr, subexpr, listexpr, mappair, rangeexpr`



Back end

At run time, have three data structures:

AST:

Environment: Map from global variables to values; stack of maps from local variables to values.

Heap: Where all values reside



Values

Java class Value contains tagged values. There are 10 types of values.

```
public class Value {
    public enum valtype {IntV, FloatV, CharV, BooleanV,
        StringV, ListV, DictV, NullV, VoidV, ObjectV};

    public valtype thetype;
    public Object thevalue;

    // lots of constructors/destructors, e.g.

    public Value (String s) {
        // Only for boolean values
        thetype = valtype.StringV;
        thevalue = s;
    }
}
```



Execution

Class eval has fields representing global and local environments:

```
public SimpleEnv globalvars;  
public EnvList localenvs;    // EnvList = stack of SimpleEnvs
```

and methods:

```
void execute (AST stmt)    // execute stmt  
Value evaluate (AST exp)   // evaluate expression  
LHValue evaluateLHS (AST exp) // evaluates "left-hand value" of expr
```



Example: Execute if statement

```
case ifstmt:
```

```
    v1 = evaluate(children.get(0));
```

```
    if (v1.isTrue() == 1)
```

```
        execute(children.get(1));
```

```
    else
```

```
        execute(children.get(2));
```

```
    return Value.Void;
```



Example: Execute function call

```
AST method = find(fname, globals);
ASTList formals = method.getFormals();
ValueList actuals =
    evallist(children.get(1).getChildren());
SimpleEnv env = zipEnv(formals, actuals);
localenvs.addAtStart(env);
AST stmt = method.getBody();
try {
    v = execute(stmt);
    localenvs.removeHead();
}
catch (ReturnException re) {
    localenvs.removeHead();
    return re.returnval; }
```



Example: Evaluate variable reference

```
case ident:  
    int ii = exp.getVar().hashname();  
    if (locals.contains(ii))  
        v1 = locals.valueof(ii);  
    else  
        v1 = globalvars.valueof(ii);  
    return v1;
```



Example: Evaluate expression w/ binary optr

```
case binaryop:
    return applyBinop(exp.getOptr(), children.get(1),
                      children.get(2))

public Value applyBinop (tokens t, AST opnd1, AST opnd2) {
    Value v1, v2;
    if (isStrictOp(t)) {
        v1 = evaluate(opnd1);
        v2 = evaluate(opnd2);
    }
    . . .
    switch (Value.maxtype (v1.thetype, v2.thetype)) {
    case CharV:
        int i = v1.getCharvalue();
        int j = v2.getCharvalue();
        return new Value(doIntop(t, i, j));
```



Sources of inefficiency

- Minor: many inefficiencies in data representations
- Major
 - “Cost of interpretation”
 - Boxing/unboxing/tag-checking



What to do about them

- “Cost of interpretation”

- Boxing/unboxing/tag-checking



Summary

- Dynamic typing allows maximum flexibility in determining the semantics of operations – e.g. can convert values at run time in whatever way is likely to be most useful to the programmer.
- In practice, it is very difficult to mix dynamic and static typing.
- However, you really don't want to develop any large program in a dynamically typed language.
- Any of the three implementation methods can be used for any language, but dynamically-typed languages usually implemented by virtual machines or (less often) direct interpretation, because maximum efficiency is not the main goal, and these techniques are simpler and more portable.

