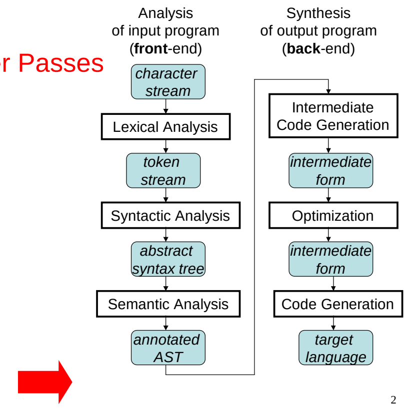
Building An Interpreter

After having done all of the analysis, it's possible to run the program directly rather than compile it … and it may be worth it

Compiler Passes



Implementing A Language

Given type-checked AST program representation:

- might want to run it
- might want to analyze program properties
- might want to display aspects of program on screen for user
- ...

To run program:

- can interpret AST directly
- can generate target program that is then run

Compilers vs. Interpreters

Interpreter

– A program that reads a source program and produces the results of executing that program

Compiler

– A program that translates a program from one language (the *source*) to another (the *target*)

Interpreter

- Interpreter
  - – Execution engine
  - – Program execution interleaved with analysis

```
running = true;
while (running) {
    analyze next statement;
    execute that statement;
}
```

  - – May involve repeated analysis of some statements (loops, functions)

Compiler

- Read and analyze entire program
- Translate to semantically equivalent program in another language
  - – Presumably easier to execute or more efficient
  - – Should "improve" the program in some fashion
- Offline process
  - – Tradeoff: compile time overhead (preprocessing step) vs execution performance

Typical Implementations

- Compilers
  - FORTRAN, C, C++, Java, COBOL, etc.
  - Strong need for optimization in many cases
- Interpreters
  - PERL, Python, Ruby, awk, sed, sh, csh, postscript printer, Scheme, Java VM
  - Effective if interpreter overhead is low relative to execution cost of individual statements

Pascal Compilers and P-code

Distribution consisted of 3 tools:

- Pascal to P-code compiler (written in Pascal)
- Pascal to P-code compiler (written in P-code)
- P-code interpreter, written in Pascal

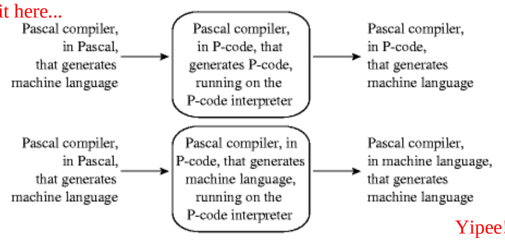
What to do?

1. Re-write the interpreter in machine code, then you can execute any P-code program using the interpreter!
   1. Run the version of the compiler written in P-code, to compile Pascal programs into P-code...
   2. Run the resulting P-code program on the interpreter!

Pascal Compilers and P-code

As an optimization, also re-write the version of the compiler written in Pascal to produce machine code instead of P-code.

Use it here...



Yipee!

Hybrid approaches

- Well-known example: Java
  - Compile Java source to byte codes – Java Virtual Machine language (.class files)
  - Execution
    - Interpret byte codes directly, or
    - Compile some or all byte codes to native code
      - Just-In-Time compiler (JIT) – detect hot spots & compile on the fly to native code
- Variation: .NET
  - Compilers generate MSIL
  - All IL compiled to native code before execution

Implementing Interpreters

Create **data structures** to represent run-time program state

- values manipulated by program
- **activation record** (a.k.a stack frame) for each called method
- **environment** to store local variable bindings
- pointer to lexically-enclosing activation record/environment (**static link**)
- pointer to calling activation record (**dynamic link**)

- EVAL loop executing AST nodes

Pros and Cons of Interpretation

\+ simple conceptually, easy to implement

- fast turnaround time
- good programming environments
- easy to support fancy language features

\- slow to execute

- data structure for value *vs.* direct value
- variable lookup *vs.* registers or direct access
- EVAL overhead *vs.* direct machine instructions
- no optimizations across AST nodes

Compilation

Divide interpreter work into two parts:
- compile-time
- run-time

Compile-time does preprocessing
- perform some computations at compile-time once
- produce an equivalent program that gets run many times

Only advantage over interpreters: faster running programs

13

Compile-time Processing

Decide representation of run-time data values

Decide where data will be stored
- registers
- format of stack frames
- global memory
- format of in-memory data structures (e.g. records, arrays)

Generate machine code to do basic operations
- just like interpreting expression, except generate code that will evaluate it later

Do optimizations across instructions if desired

14

Compile-time vs Run-time

Compile-time	**Run-time**
Procedure	Activation record/stack frame
Scope, symbol table	Environment (contents of stack frame)
Variable	Memory location or register
Lexically-enclosing scope	Static link
Calling Procedure	Dynamic link

15

An Interpreter for MiniJava

New Stuff Needed: Some Data Structures + some Code

Data Structures: In `Evaluator` subdirectory, two data structures:

1) Data structure to *represent* run-time values:

`Value` hierarchy
- analogous to `ResolvedType` hierarchy

```
Value
    IntValue
    BooleanValue
    ClassValue
    NullValue
```

16

MiniJava Interpreter [continued]

2) Data structure to *store* `Values` for each variable:

`Environment` hierarchy
- analogous to `Symbol Table` hierarchy

```
Environment
  GlobalEnvironment
  NestedEnvironment
    ClassEnvironment
    CodeEnvironment
      MethodEnvironment
```

And some Code:
- `evaluate` methods for each kind of AST class

17

Activation Records

Each call of a procedure allocates an activation record (instance of `Environment`)
- Activation record stores:
  - mapping from names to `Values`, for each formal and local variable in that scope (**environment**)
  - lexically enclosing activation record (**static link**)
- Method activation record: also
  - calling activation record (**dynamic link**)
- Class activation record: also
  - methods (to support run-time method lookup)
  - instance variable declarations, not values
  - values stored in class instances, i.e.,`ClassValues`

18

Activation Records vs Symbol Tables

For each method/nested block scope in a program:

- exactly one symbol table, storing **types** of names
- possibly many activation records, one per invocation, each storing **values** of names

For recursive procedures,

- can have several activation records for same procedure on stack simultaneously

All activation records have same "shape," described by single symbol table

Example

```
...

class Fac {
   public int ComputeFac(int num) {
      int numAux;
      if (num < 1) {
          numAux = 1;
      } else {
          numAux = num * this.ComputeFac(num-1);
      }
      return numAux;
   }
}
```


Generic Evaluation Algorithm

Parallels the generic typechecking algorithm

To evaluate a program,

- recursively evaluate each of the nodes in the program's AST, each in the context of the environment for its enclosing scope
- on the way down, create any nested environments & context needed
- recursively evaluate child subtrees
- on the way back up, compute the parent's result/effect from the children's results
- parent controls order of evaluation of children, whether to evaluate children

Each AST node class defines its own **`evaluate`** method, which fills in the specifics of this recursive algorithm

Generally:

- declaration AST nodes add *value* bindings to the current environment
- statement AST nodes evaluate (some of) their subtrees
- expression AST nodes evaluate their subtrees and compute & return a result value

Some Key AST Evaluation Operations

`void Program.evaluate() throws EvalCompilerExn;`

- evaluate the whole program:
  - evaluate each of the class declarations
  - invoke the main class's main method

`void ClassDecl.evaluateDecl(GlobalEnvironment) throws EvalCompilerExn;`

- evaluate a class declaration

`void Stmt.evaluate(CodeEnvironment) throws EvalCompilerExn;`

- evaluate a statement in the context of the givenenvironment

`Value Expr.evaluate(CodeEnvironment) throws EvalCompilerExn;`

- evaluate an expression in the context of the given environment, returning the result

An example evaluation operation

```
class IntLiteralExpr extends Expr {
  int value;

  Value evaluate(CodeEnvironment env)
          throws EvalCompilerException {
     return new IntValue(value);
  }
}
```


An example evaluation operation

```
class AddExpr extends Expr {
  Expr arg1;
  Expr arg2;

  Value evaluate(CodeEnvironment env)
            throws EvalCompilerException {
      Value arg1_value = arg1.evaluate(env);
      Value arg2_value = arg2.evaluate(env);
      return new IntValue(arg1_value.getIntValue()
                          +
                          arg2_value.getIntValue());
  }
}
```

`getIntValue` asserts that the value is an int and returns its value

An example evaluation operation

```
class VarDeclStmt extends Stmt {
  String name;
  Type type;

  void evaluate(CodeEnvironment env)
           throws EvalCompilerException {
      env.declareLocalVar(name);
  }
}
```

declareLocalVar adds a new uninitialized binding to the current environment

25

An example evaluation operation

```
class VarExpr extends Expr {
  String name;

  Value evaluate(CodeEnvironment env)
            throws EvalCompilerException {
      // (record var_iface during typechecking)
      return var_iface.lookupVar(env);
  }
}
```

lookupVar looks at the kind of variable being read, and does the right thing. For a local variable:

```
      return env.lookupLocalVar(name);
```

returns contents of binding for name in **env** (or enclosing env)

26

An example evaluation operation

```
class IfStmt extends Stmt {
  Expr test;
  Stmt then_stmt;
  Stmt else_stmt;

  void evaluate(CodeEnvironment env)
      throws EvalCompilerException {
      Value test_value = test.evaluate(env);
      if (test_value.getBooleanValue()) {
            then_stmt.evaluate(env);
      } else {
            else_stmt.evaluate(env);
      }
  }
}
```

getBooleanValue asserts that the value is a boolean and returns its value

Controls which substatement gets evaluated

27