CS322 Languages and Compiler Design II

Spring 2012

Lecture 2

INTERPRETERS

An **interpreter** for a language L is a program $P_{L'}$ that given

- a description of a program Q_L (written in L), and
- an input I

behaves like Q_L on I.

L is the **source language**.

L', in which the interpreter is written, is the **implementation language**.

There are many possibilities for L' (including L itself!), but typically it will be a high-level language like C, Lisp, etc.

Important point is that $P_{L'}$ is **generic**: it should work for **any** possible program Q_L.

Examples

(Note: any language **may** be interpreted, but some **usually** are.)

BASIC, Pascal (PCODE), Java (bytecode), scripting languages, etc.

PROS AND CONS OF INTERPRETERS

$+$ **Easier** to write than a compiler; can take advantage of high-level features of implementation language.

$+$ **No compilation time overhead** for users of source language; code-test-debug can be much quicker.

$+$ **Portable**, assuming implementation language is.

$+$ Provides a **semantics**, relative to implementation language.

$-$ Interpretation is **slower** than running compiled code, mainly because decoding and dispatch are done in software, and because (ordinarily) very little optimization is done.

INTEPRETATION VS. COMPILATION

There is a continuum of possibilities between source interpretation and translation to machine code:

• Many systems translate source language to some **intermediate language** and then interpret that.

• Some systems include “just in time” compilers that generate machine code on the fly and then immediately execute it.

• Hardware processors can be viewed as “interpreting” machine code instructions (esp. if hardware is microcoded).

• Can build-special purpose hardware processors for specific languages (e.g., LISP machines, Java chips).

Defining Interpreters Using Attribute Grammars

Like other language processing, convenient to define interpreters using grammatical syntax framework.

As first step, can define interpreter using an attribute grammar.

Approach similar to semantics definitions, but instead of computing a **translation** (to machine code, functions, etc.), actually compute the **value** of the program within the grammar.

Thus we use a.g. formalism as our “implementation language.”

(Next step will be to encode the a.g. into a “real” language, such as Java or C.)

Need to use both **synthesized** and **inherited** attributes.

Review of Attribute Grammars

Attribute Grammars (a.k.a. "syntax-directed definitions") allow convenient, concise definition of calculations on recursive structures.

Calculations are specified by describing their **local** behavior at parse tree nodes; structure of parse tree defines **global** shape of computation.

• Attach **rules** (a.k.a. "attribute equations") to grammar productions.

• Rules compute **attribute** values at corresponding parse tree node based on attribute values at

- parent nodes (**inherited** attributes), and/or

- child nodes (**synthesized** attributes)

Example:

```
A := B C     ↑ A.syn := ... B. .... C. ....
             ↓ B.inh := ... A. ...
             ↓ C.inh := ... A. ....
```

• Terminals may have "built-in" attributes (think of them as being synthesized automatically).

A.G. DEFINITIONS ARE "SELF-CHECKING"

Must remember to define **all** needed attributes.

• If `S.inh` is an **inherited** attribute, it must be defined **each** time `S` appears in **any** grammar production right-hand side.

$$
\begin{array}{ll}
\mathtt{T} := \mathtt{S}_1\ \mathtt{S}_2 & \downarrow \mathtt{S}_1\mathtt{.inh} := \ldots\ \mathtt{T}\ \ldots \\
 & \downarrow \mathtt{S}_2\mathtt{.inh} := \ldots\ \mathtt{T}\ \ldots
\end{array}
$$

• If `S.syn` is a **synthesized** attribute, it must be defined **each** time `S` appears as a grammar production left-hand side.

$$
\begin{array}{ll}
\mathtt{S} := \mathtt{T}_1\ \mathtt{T}_2 & \uparrow \mathtt{S.syn} = \ldots\ \mathtt{T}_1\ \ldots\ \mathtt{T}_2\ \ldots \\
\mathtt{S} := \mathtt{U}_1\ \mathtt{U}_2\ \mathtt{U}_3 & \uparrow \mathtt{S.syn} = \ldots\ \mathtt{U}_1\ \ldots\ \mathtt{U}_2\ \ldots\ \mathtt{U}_3\ \ldots
\end{array}
$$

Purely Functional Attribute Grammars

Life is much nicer if we restrict the right-hand sides of attribute rules to be **pure functions**, i.e., calculations with no side-effects, because then

- The "result" of evaluation is just the value of the root node's synthesized attributes.

- Evaluation can occur in **any** order consistent with **data dependencies** among attribute rules.

Must avoid **circularities** in rules, e.g.:

```
A := B C     ↑ A.x = B.x + 10
             ↓ B.y = A.x - 5

B := D E     ↑ B.x = if D.flag then
                            B.y + 2
                         else E.z
```

Precise definition of circularity can be subtle.

Simple Expression Language with Local Binding

```
prog := exp
exp := NUM
exp := VAR
exp := exp1 '+' exp2
exp := exp1 '*' exp2
exp := LET VAR '=' exp1 IN exp2 END
```

Example:

```
let a = 2 + 5
in 14 + let b = a * 3
        in b + 7
        end
end
```

$\Rightarrow$ 42

Attribute Grammar for Interpretation

```
prog := exp
        ↓ exp.env := empty
        ↑ prog.val := exp.val

exp := NUM
        ↑ exp.val := NUM.num

exp := VAR
        ↑ exp.val := lookup(exp.env,VAR.var)

exp := exp1 '+' exp2
        ↓ exp1.env := exp.env
        ↓ exp2.env := exp.env
        ↑ exp.val := exp1.val + exp2.val
```

`exp := exp1 '*' exp2` *is similar*

```
exp := LET VAR '=' exp1 IN exp2 END
        ↓ exp1.env := exp.env
        ↓ exp2.env := extend(exp.env, VAR.var, exp1.val)
        ↑ exp.val := exp2.val
```

ATTRIBUTE GRAMMAR (CONT.)

Attributes:

Terminal `NUM` has `.num` attribute (number)

Terminal `VAR` has `.var` attribute (string)

Terminals `'+'`,`'*'`,`let`,`'='`,`IN`,`END` have no attributes.

Non-terminal `exp` has

- **inherited** `env` attribute (dictionary)

- **synthesized** `val` attribute (number)

A **dictionary** is a (functional) abstract data type supporting the following primitives:

```
empty: dictionary
lookup: dictionary × string → number
extend: dictionary × string × number → dictionary
```

Imperative Evaluation Strategies

Functional attribute grammars have nice properties, but can make it awkward to deal with **imperative** features of languages, such as input/output and assignment statements.

Alternative: **fix** the evaluation order of attributes, so that we can safely include imperative statements (**side effects**) in the “attribute equations” section.

Default order: depth-first, left-to-right, **but** must obey data dependencies.

- First evaluate children’s inherited attributes.

- Then recursively evaluate children, obtaining their synthesized attributes.

- Finally evaluate own synthesized attributes.

- Can perform side-effects at any point specified (no standard way to express this, though)

IMPERATIVE A.G. EXAMPLE

Add variable assignment (via an `update` primitive for the dictionary ADT) and printing (via a `write` primitive).

All the previous rules plus...

```
exp := PRINT exp₁
        ↓ exp₁.env := exp.env
        • write(exp₁.val)
        ↑ exp.val := exp₁.val

exp := VAR ':=' exp₁
        ↓ exp₁.env := exp.env
        • update(exp.env,VAR.var,exp₁.val)
        ↑ exp.val := exp₁.val
```

Implementing Imperative Attribute Grammars

It is easy to turn imperative attribute grammars into imperative **recursive descent** programs that process tree data structures. Programs could be in C or Java. (Type-checker was one example.)

- Each **nonterminal** N gets corresponding **function** `N`.
- **Inherited** attributes of N become extra **arguments** to the function `N`
- **Synthesized** attributes of N become **return values** from the function `N`.
- Follow evaluation order described previously.
- **Side effects** are executed wherever encountered.

C VERSION OF EXAMPLE

First the data structure:

```
typedef char* id;
typedef struct ExpS *Exp;
struct ExpS {
  enum
    {Num, Var, Plus, Print, Let, Assign} kind;
  union {
    struct { int n; } num;
    struct { id v; } var;
    struct { Exp e1, e2; } plus;
    struct { Exp e; } print;
    struct { id v; Exp e1, e2; } let;
    struct { id v; Exp e; } assign;
  } u;
};
```

C Version (continued)

Assume suitable operations on environments and I/O:

```
typedef ... Env;
static Env empty;
int lookup(Env, id);
Env extend(Env,id,int);
void update(Env,id,int);

void write(int);
```

And the actual evaluation code is...

C VERSION (CONTINUED)

```
int eval(Env env,Exp exp) {
  switch (exp->kind) {
  case Num : return exp->u.num.n;
  case Var : return lookup(env,exp->u.var.v);
  case Plus : {int v1 = eval(env,exp->u.plus.e1);
               int v2 = eval(env,exp->u.plus.e2);
               return (v1 + v2);}
  case Print : {int v = eval(env,exp->u.print.e);
                write (v);
                return v;}
  case Let : {int v1 = eval(env,exp->u.let.e1);
              Env env1 = extend(env,exp->u.let.v,v1);
              int v2 = eval(env1,exp->u.let.e2);
              return v2;}
  case Assign : {int v = eval(env,exp->u.assign.e);
                 update(env,exp->u.assign.v,v);
                 return v;}
  }
}
```

First Java Version of Example

```
abstract class Exp {
  abstract int eval(Env env);
}

class NumExp extends Exp {
  int n;
  NumExp(int n) { this.n = n; }
  int eval(Env env) { return n; }
}

class VarExp extends Exp {
  String v;
  VarExp(String v) { this.v = v; }
  int eval(Env env) { return Env.lookup(env,v); }
}

class PlusExp extends Exp {
  Exp e1, e2;
  PlusExp(Exp e1, Exp e2) {this.e1=e1;this.e2=e2;}
  int eval(Env env) { return e1.eval(env) + e2.eval(env); }
}
```

```
class PrintExp extends Exp {
  Exp e;
  PrintExp(Exp e) { this.e = e; }
  int eval(Env env) {
    int v = e.eval(env);
    System.out.println(v);
    return v;
  }
}

class LetExp extends Exp {
  String v;
  Exp e1, e2;
  LetExp (Exp e1, Exp e2) {this.e1=e1;this.e2=e2;}
  int eval(Env env) {
    int v1 = e1.eval(env);
    Env env1 = Env.extend(env,v,v1);
    return e2.eval(env1);
  }
}
```

```
class AssignExp extends Exp {
  String v;
  Exp e;
  AssignExp(String v, Exp e) {this.v=v;this.e=e;}
  int eval(Env env) {
    int v1 = e.eval(env);
    Env.update(env,v,v1);
    return v1;
  }
}
```

assuming these supporting operations on environments:

```
class Env{
  static Env empty = new Env();
  static int lookup(Env env, String s) { ... }
  static Env extend(Env env, String s, int v) { ... }
  static Env update(Env env, String s, int v) { ... }
}
```

Second Java Version using Visitors

```
abstract class Exp {
  abstract Object accept(ExpVisitor v);
}
interface ExpVisitor {
  Object visit(NumExp e);
  Object visit(VarExp e);
  Object visit(PlusExp e);
  Object visit(PrintExp e);
  Object visit(LetExp e);
  Object visit(AssignExp e);
}
class NumExp extends Exp {
  int n;
  NumExp(int n) { this.n = n; }
  Object accept(ExpVisitor v) { return v.visit(this); }
}
class VarExp extends Exp {
  String v;
  VarExp(String v) { this.v = v; }
  Object accept(ExpVisitor v) { return v.visit(this); }
}
```
and similarly for the other classes

```
class Eval {
  static int eval(final Env env,Exp e) {
    class EvalExpVisitor implements ExpVisitor {
      public Object visit(NumExp e) {
        return new Integer(e.n);
      }
      public Object visit(VarExp e) {
        return new Integer(Env.lookup(env,e.v));
      }
      public Object visit(PlusExp e) {
        int v1 = eval(env,e.e1);
        int v2 = eval(env,e.e2);
        return new Integer(v1 + v2);
      }
      public Object visit(PrintExp e) {
        int v = eval(env,e.e);
        System.out.println(v);
        return new Integer(v);
      }
```

```
    public Object visit(LetExp e) {
      int v1 = eval(env,e.e1);
      Env env1 = Env.extend(env,e.v,v1);
      return new Integer(eval(env1,e.e2));
    }
    public Object visit(AssignExp e) {
      int v1 = eval(env,e.e);
      Env.update(env,e.v,v1);
      return new Integer(v1);
    }
  }
  ExpVisitor visitor = new EvalExpVisitor();
  Integer r = (Integer) e.accept(visitor);
  return r.intValue();
}}
```