Abstract Syntax: what and why

CIS 461/561, Fall 2018

Direct Interpretation

Many "little languages" embed an interpreter directly in parser actions

example: simplecalc example in Cup distribution

```
expr     ::= expr:e1 PLUS expr:e2
          {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
          |
          expr:e1 MINUS expr:e2
          {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
          |
          expr:e1 TIMES expr:e2
          {: RESULT = new Integer(e1.intValue() * e2.intValue()); :}
          |  ...
```

Parse tree as AST (almost)

Beaver expression tree example

```
%goal expr;


expr
     = expr.a MULT  expr.b           {: return new MultExpr (a, b); :}
     | expr.a DIV   expr.b           {: return new DivExpr  (a, b); :}
     | expr.a PLUS  expr.b           {: return new PlusExpr (a, b); :}
     | expr.a MINUS expr.b           {: return new MinusExpr(a, b); :}
     | MINUS expr.e @ UNARY_MINUS    {: return new NegExpr  (e);    :}
     | NUMBER.n                      {: return new NumExpr  (n);    :}
     | LPAREN pexpr.e RPAREN         {: return e;                   :}
     ;
```

Separate module “walks” the tree to evaluate it (with the visitor pattern)

Direct Translation

We could just as well embed code generation directly in parser actions

“Single pass compilation”

Fine for very simple transformations in which information is available in the right order.

- “Right order”? Looking at attribute grammars will help us talk about order of producing and using information; we’ll do that later.

Two (or more) passes:
building an intermediate representation

We want:

Simplicity (remove extraneous detail)

Regularity (simplify processing)

Suitability for next processing steps

- Binding, type checking, maybe code generation

Some Options …

- Build the parse tree (concrete syntax)
- Build a more "abstract" tree
- Build something else
  - "Target tree" (closer to machine)
  - Abstract machine code (linear)
  - Control flow graph
  - Single assignment form
  - …

Why not the parse tree?

- Messy
  - Unnecessary stuff from the concrete syntax, like parentheses
  - Unnecessary stuff added to concrete syntax to help parsing
    - (e.g., parentheses, “unit productions” for precedence)
- Irregular
  - More than one way to say something; complicates further processing

Aside: Unit productions

```
expr  =   expression PLUS term
       |  term
       ;

term =   term TIMES product
       |  product
       ;
```

A “unit” production is a production with a single non-terminal on the right

Abstract Syntax

Regularize:

```
int i, j;  ⇒  int i; int j;
```

Simplify:

```
3*(i+k) ⇒  Times(IntConst(3),
  Plus(i,k))
```

In Quack we further regularize this by "desugaring" Times and Plus to method calls.

Organize:

Use node types to prepare for binding, type checking

Attribute Grammars

A way of thinking about what can be done during parsing, and during each pass over the tree

Associate *attributes* with each terminal and non-terminal symbol in the grammar

- Attribute = any kind of information that may be produced during one production and used in another: Types, object code sequences, etc.

Associate *attribute equations* with productions

- Attribute equation = use of attributes of some symbols to compute attributes for other symbols

cf. Dragon book sec 5.1, 5.2, pp 304-...

Perspective

The next several slides are about attribute grammars, BUT
we are not actually going to use attribute grammars in a formal way.

Attribute grammar ideas are very useful for thinking about processing the AST, but attribute grammar tools have never gotten better than just writing code. So far.

A bit of history ...

Language syntax systematized

- with context-free languages, Backus-[Naur|Normal] form, and parsing algorithms

Syntax-directed compilation framework

- attempts to describe semantics and translation in the same framework (through grammar)
- Attribute grammars [Knuth 1968] describe types, values, other attribute through equations associated with grammar productions

Attributes and Denotation

Meaning through structure, e.g.,

[[A "+" B]] = [[A]] **+** [[B]]

- where "+" is in syntactic domain, **+** is in semantic domain

Read [[x]] as "the meaning of x"

Generalizes from values to other attributes

exp_1 ::= exp_2 "+" exp_3

{:

exp_1.type = addtype(exp_2.type, exp_3.type);

:}

Why Attribute Grammars?

Rules are local to productions

A way to organize our thinking and our code

We can reason about classes of grammars

Systematic creation of (hand-coded) evaluators

Creation of automatic attribute evaluators

- More popular in mid-80s than now; due for a resurgence?

Inherited and Synthesized

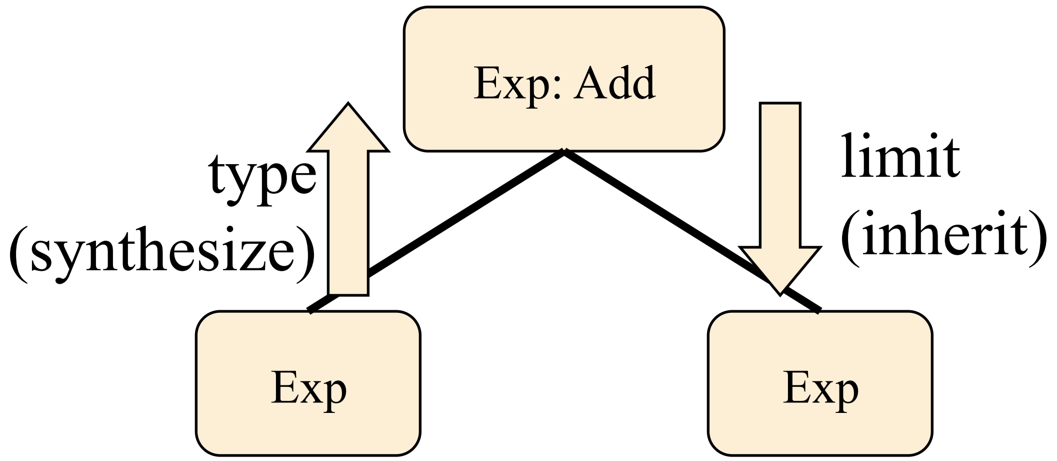
```
Exp0 ::= Exp1 “+” Exp2
{:
   Exp0. type = type_add(Exp1.type, Exp2.type);
   Exp1.limit = Exp0.limit;
:}
```

“Type” is synthesized attribute (bottom-up)

“Limit” is inherited attribute (top-down)

- More realistic examples require a mix in which inherited values depend on synthesized values

Inherited and Synthesized



Exp_0 ::= Exp_1 "+" Exp_2
{:
Exp_0. type = type_add(Exp_1.type, Exp_2.type);
Exp_1.limit = Exp_0.limit;
:}

Beyond Inherited & Synthesized

Exp_0 ::= "let" Exp_1 "in" Exp_2 "end"

{: Exp_0.value = ?? :}

Meaning of Exp_2 depends on Exp_1

- Value is neither inherited nor synthesized, but it can be evaluated left-to-right

This grammar is (or could be) *L-attributed*

- Allow synthesized attributes in lhs, left-to-right flow of attributes in rhs of production

L-attributed Grammar Properties

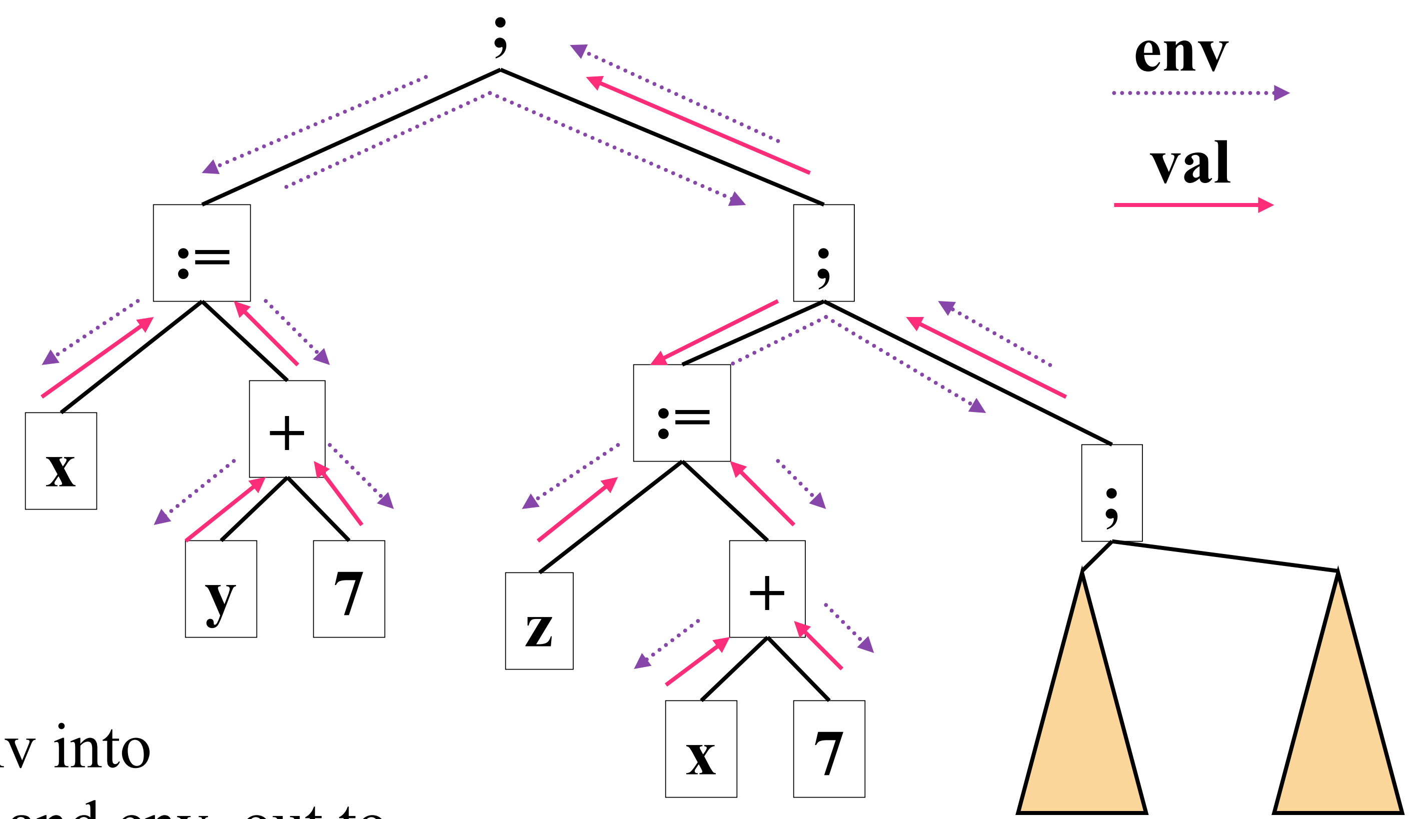
An L-attributed grammar can be evaluated on-the-fly during LL(1) parsing

- but not during shift-reduce parsing; why?

An L-attributed grammar can be evaluated during one depth-first traversal of an abstract syntax tree

- even if AST was built with an S-attribution, e.g., by CUP or Bison action routines

Evaluating L-Attribution



split env into
env_in and env_out to
get L-attribution

Designing Attribute Equations

Use synthesized attributes where possible

- Simplest, most uniform to evaluate, either on-the-fly or during AST traversal

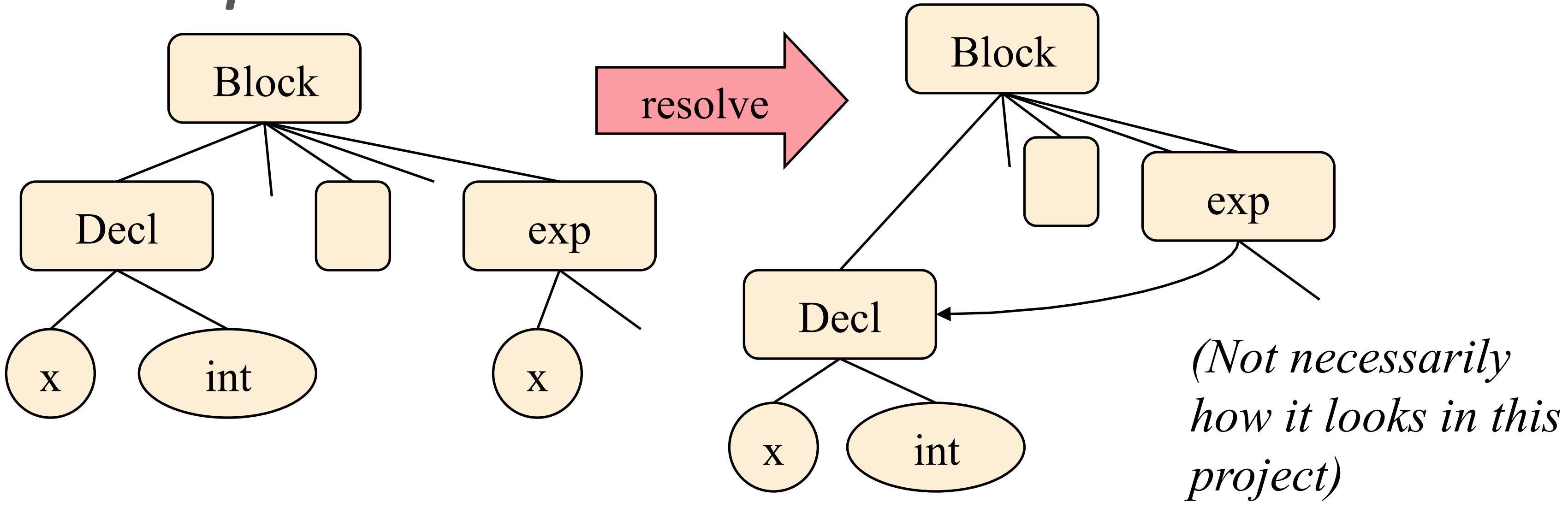
Use inherited or left-to-right evaluation where necessary

If L-attribution is not possible:

- Analyze attribute dependence for multi-pass evaluation

Even if the equations are entirely in your head or on paper. We use attribute grammars as a design tool even if we don't have attribute grammar tools.

Example: Name Resolution



The environment (symbol table) flows left to right; a reference is (often, not always) "resolved" to a link to the declaration sub-tree

This can be non-trivial when there is *overloading;* note binding may be partly or fully dynamic (e.g., OO dispatch)

Attribute Grammar as Design Step

Even without AG-based tools ...

First sketch evaluation as attribute grammar

- What can be done with S-attribution? What requires L-attribution? Any inherited attributes? Any cycles?
- Break cycles with multiple attributes

Translate attribute evaluation to code

  - In depth-first traversals of AST

Why No Tools? (1)

Historically: Theory => Tools

- CFGs and parsing theory gave us Yacc, etc.
- Attribute grammars gave us ???

Where are the tools?

- Yacc/Bison/CUP/Beaver/... action routines are bare-bones mechanism for on-the-fly evaluation
- Synthesizer Generator (mid-80s) ... died out with syntax-directed editors

Time for a resurgence?

- E.g., type checkers from type inference rules? We know how to do it, if we agreed on a representation.

Why No Tools? (2)

Whatever happened to attribute grammars?

- Introduced in 68, efficient tools produced in 80s; disappearing from compiler texts

Main obstacles

Memory (for multi-pass evaluation)

- Until recently, only academic toy compilers built a complete AST
- No longer an issue?

Ad hoc link between concrete and abstract syntax

Too much “outside” the formal description

Aside: What you should know

Distinguish synthesized from inherited attributes

Recognize S-attribution, L-Attribution

- pure bottom-up vs. left-to-right, depth-first

Recognize and break circularity

Organizing an AST

Problem: Two dimensions of modularity

- OO AST can capture one; hard to cut both ways

Node classification dimension

- e.g., "multiplication" is subclass of "binop"
- Nicely captured in OO structure

Compiler phase dimension

- Type checking, improvement, code generation
- Orthogonal to OO structure; even with "visitor" pattern, OO AST does not effectively localize change (e.g., adding an optimization pass).

What To Do?

Simple languages / compilers:

One AST structure, multiple walks over it

- Maybe with visitor pattern, maybe not

Bigger languages / compilers:

A sequence of representations:

- AST (for binding and type-checking), then
- Flow graphs OR register transfer lists OR ...
  - – Target-oriented representation of operational semantics, plus memory layout
  - – LLVM is an example target-oriented but machine-independent representation; generating LLVM code is much like generating machine code

Example AST construction

In my ASTNode.h

```
class Typecase : public ASTNode {
    ASTNode& expr_;
    Seq& cases_;
public:
    explicit Typecase(ASTNode& expr, Seq& cases) :
       expr_{expr}, cases_{cases} {};
    void json(std::ostream& out, AST_print_context& ctx) override;
```

In my quack.yxx

```
statement: TYPECASE expr '{' type_alternatives '}'
   { $$ = new AST::Typecase(*$2, *$4); }
   ;
```

AST with a little more A *(for "abstract")*

```
class If : public ASTNode {
   ASTNode &cond_;
   Seq &truepart_;
   Seq &falsepart_;
public:
   explicit If(ASTNode& cond, Seq& truepart, Seq& falsepart) :
      cond_{cond}, truepart_{truepart}, falsepart_{falsepart} { };
   void json(std::ostream& out, AST_print_context& ctx) override;
};
```

```
statement: IF expr statement_block  opt_elif_parts
        { $$ = new AST::If(*$2, *$3, *$4); } ;

opt_elif_parts:  ELIF expr statement_block  opt_elif_parts
          { $$ = new AST::Block();      $$->append(new AST::If(*$2, *$3, *$4)); }
          |   ELSE statement_block   { $$ = $2; }
          | /* empty */              { $$ = new AST::Block(); }
          ;
```

Desugaring

Syntactic sugar: Syntax for readability or convenience, without adding any new expressiveness to the language.

```
class Call : public ASTNode {
   ASTNode& receiver_;     /* Expression computing the receiver object */
   ASTNode& method_;       /* Identifier of the method */
   Seq& actuals_;          /* List of actual arguments */
public:
   explicit Call(ASTNode& receiver, ASTNode& method, Seq& actuals) :
     receiver_{receiver}, method_{method}, actuals_{actuals} {};
   static Call* binop(std::string opname, ASTNode& receiver, ASTNode& arg);
   void json(std::ostream& out, AST_print_context& ctx) override;
};
```

```
expr:  expr '*' expr   { $$ = AST::Call::binop("TIMES", *$1, *$3); }
   |  expr '/' expr     { $$ = AST::Call::binop("DIV", *$1, *$3); }
   |  expr '+' expr     { $$ = AST::Call::binop("PLUS", *$1, *$3); }
   |  expr '-' expr     { $$ = AST::Call::binop("MINUS", *$1, *$3); }
   |  '-' expr  %prec NEG  { auto zero = new AST::IntConst(0);
                             $$ = AST::Call::binop("MINUS", *zero, *$2);
                           }
```