Symbol Tables in JastAdd

Andrew Casey

Review

- What is a symbol table used for?
  - Determining the origin and the properties of a given symbol

Review

- What goes in a symbol table?
  - Name
  - Definition site
  - Type
  - Other annotations/attributes

Attribute Grammars

- Introduced by Knuth as a way to specify programming language semantics

Attribute Grammars

- An attribute is a function from parse tree nodes to a domain of your choosing
- The value of an attribute at a given node is defined in terms of the values of attributes of neighbouring nodes in the parse tree

Synthetic Attributes

- Attributes that depend on values in the *descendants* of a node are called *synthetic attributes*
- *Synthetic attributes* pass information *up* the parse tree

Inherited Attributes

- Attributes that depend on values in the *ancestors* of a node are called *inherited attributes*
- *Inherited attributes* pass information *down* the parse tree

Example

- From Knuth's original paper
- Suppose we want to determine a value for the binary number XX...X.X...X, where X is a single bit

Grammar

$B \rightarrow 0$

$B \rightarrow 1$

$L \rightarrow B$

$L \rightarrow LB$

$N \rightarrow L$

$N \rightarrow L\,.\,L$

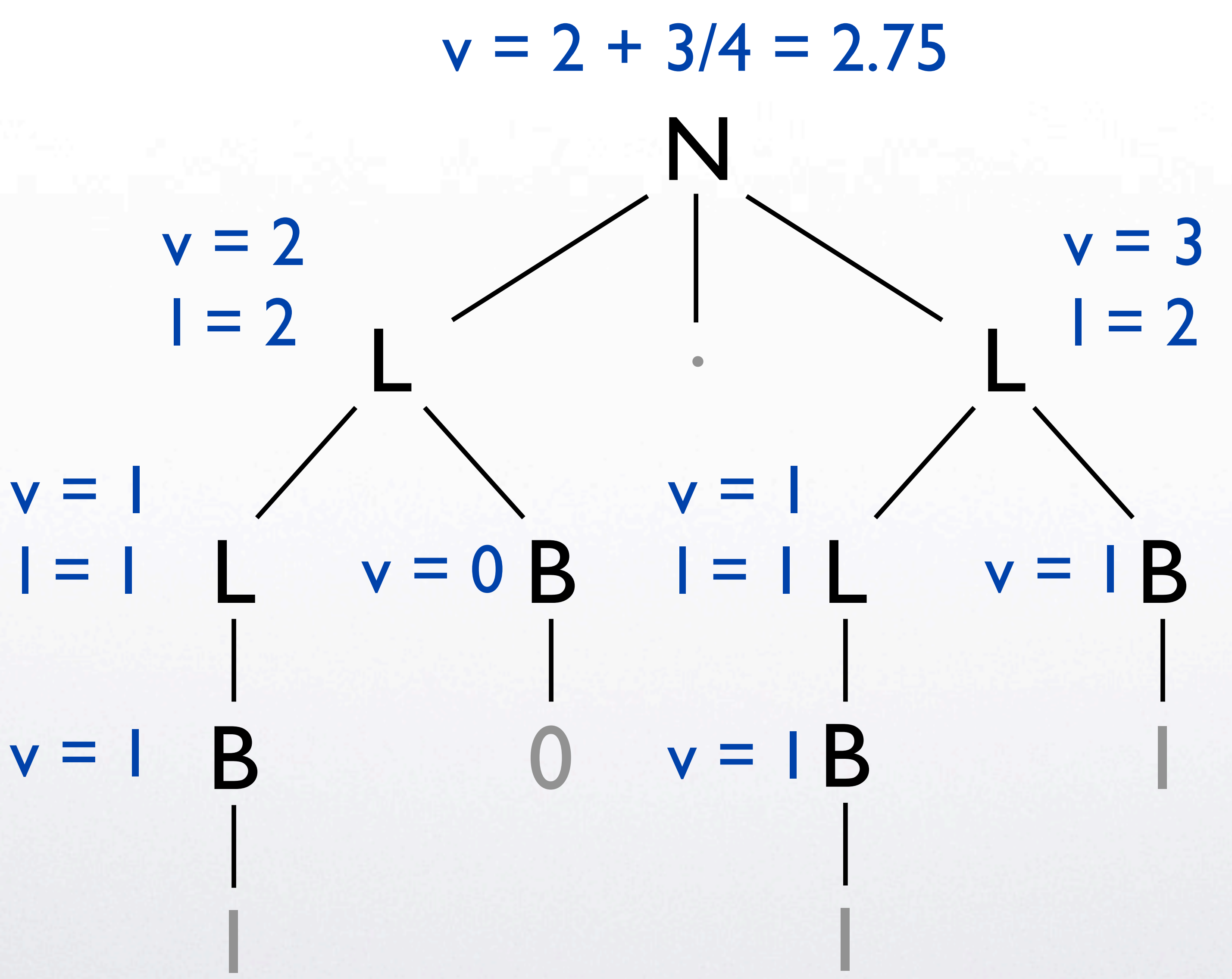
Approach 1

- Use only synthetic attributes
- Each subtree is independent

Approach 1

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 1$
$L \rightarrow B$	$v(L) = v(B); l(L) = 1$
$L_1 \rightarrow L_2B$	$v(L_1) = 2v(L_2) + v(B); l(L_1) = l(L_2) + 1$
$N \rightarrow L$	$v(N) = v(L)$
$N \rightarrow L_1 . L_2$	$v(N) \rightarrow v(L_1) + v(L_2)/2^{l(L_2)}$

Approach 1



Approach 2

- Use inherited attributes to make things more intuitive (i.e. close to our mental model of how binary numbers work)
- e.g. the ‘1’ in ‘100’ means ‘8’

Approach 2

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2^{s(B)}$
$L \rightarrow B$	$v(L) = v(B); l(L) = l(B); s(B) = s(L)$
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B); l(L_1) = l(L_2) + 1;$
	$s(L_2) = s(L_1) + 1; s(B) = s(L_1)$
$N \rightarrow L$	$v(N) = v(L); s(L) = 0$
$N \rightarrow L_1 . L_2$	$v(N) \rightarrow v(L_1) + v(L_2); s(L_1) = 0; s(L_2) = -l(L_2)$

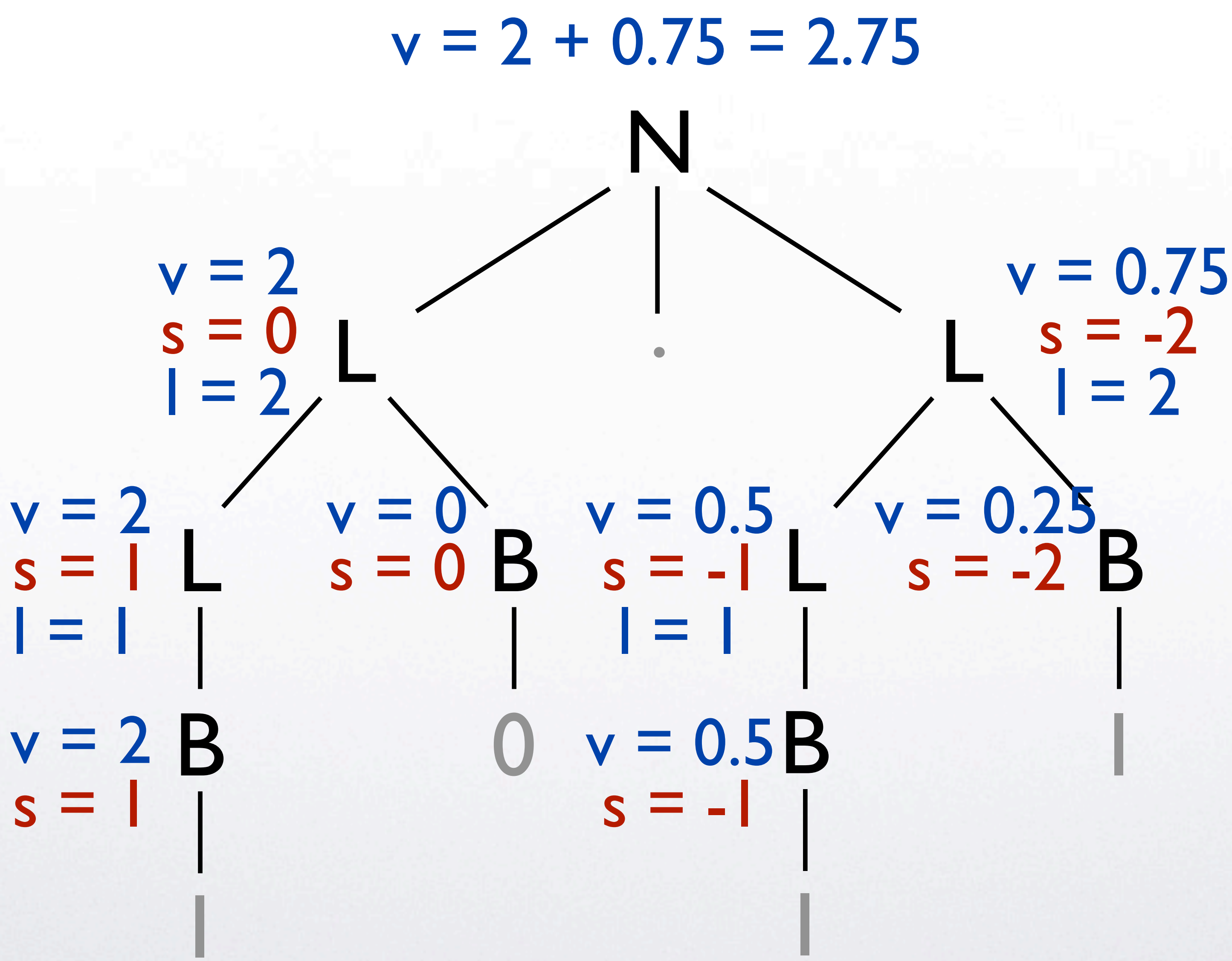
Approach 2

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2^{s(B)}$
$L \rightarrow B$	$v(L) = v(B); l(L) = l(B); s(B) = s(L)$
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B); l(L_1) = l(L_2) + 1;$
	$s(L_2) = s(L_1) + 1; s(B) = s(L_1)$
$N \rightarrow L$	$v(N) = v(L); s(L) = 0$
$N \rightarrow L_1 . L_2$	$v(N) \rightarrow v(L_1) + v(L_2); s(L_1) = 0; s(L_2) = -l(L_2)$

Approach 2

$B \rightarrow 0$	$v(B) = 0$
$B \rightarrow 1$	$v(B) = 2^{s(B)}$
$L \rightarrow B$	$v(L) = v(B); l(L) = l(B); s(B) = s(L)$
$L_1 \rightarrow L_2B$	$v(L_1) = v(L_2) + v(B); l(L_1) = l(L_2) + 1;$
	$s(L_2) = s(L_1) + 1; s(B) = s(L_1)$
$N \rightarrow L$	$v(N) = v(L); s(L) = 0$
$N \rightarrow L_1 . L_2$	$v(N) \rightarrow v(L_1) + v(L_2); s(L_1) = 0; s(L_2) = -l(L_2)$

Approach 2



v = 2 + 0.75 = 2.75
N
v = 2
s = 0
l = 2
L
.
L
v = 0.75
s = -2
l = 2
v = 2
s = 1
l = 1
L
v = 0
s = 0
B
v = 0.5
s = -1
l = 1
L
v = 0.25
s = -2
B
v = 2
s = 1
B
0
v = 0.5
s = -1
B
1
1
1

Practical Concerns

- If information can move both up and down the parse tree, then it is possible to define attributes cyclically!

Practical Concerns

- If we restrict ourselves to certain combinations of attributes, then we can compute their values more efficiently
- Synthetic-only: single post-order pass
- Inherited-only: single pre-order pass
- LR-attributed: single LR-parsing pass (i.e. by the time a node is created, all values it depends on have already been computed).

JastAdd

- An attribute grammar system for Java
- Primarily used for creating extensible compilers
- Primarily the work of Torbjörn Ekman (Oxford) & Görel Hedin (Lund)

JastAdd

- Where does it fit?

  1. You create a lexer and parser as usual (e.g. using flex and bison)
  2. You specify an AST structure in JastAdd
  3. You build the AST in the actions of your grammar
  4. You decorate the AST with attributes

Abstract Syntax

- Since attributes are so interwoven with the abstract syntax, you have to use the abstract syntax specification language provided by JastAdd to create your AST classes

Example

abstract Expr;

AddExpr : Expr ::= LHS:Expr RHS:Expr;

IDExpr : Expr ::= <Name>;

NumExpr : Expr ::= <Value:int>;

Attributes

- Attributes are defined in separate files (aspects) but are ultimately inserted into the generated AST node classes

Synthetic Attributes

- syn ReturnType NodeType.Attribute();
- eq NodeType.Attribute() = value;

Inherited Attributes

- inh ReturnType NodeType.Attribute();
- eq ParentNodeType.getChild().Attribute() = value;

Evaluated in the context of the parent node

Extra Features

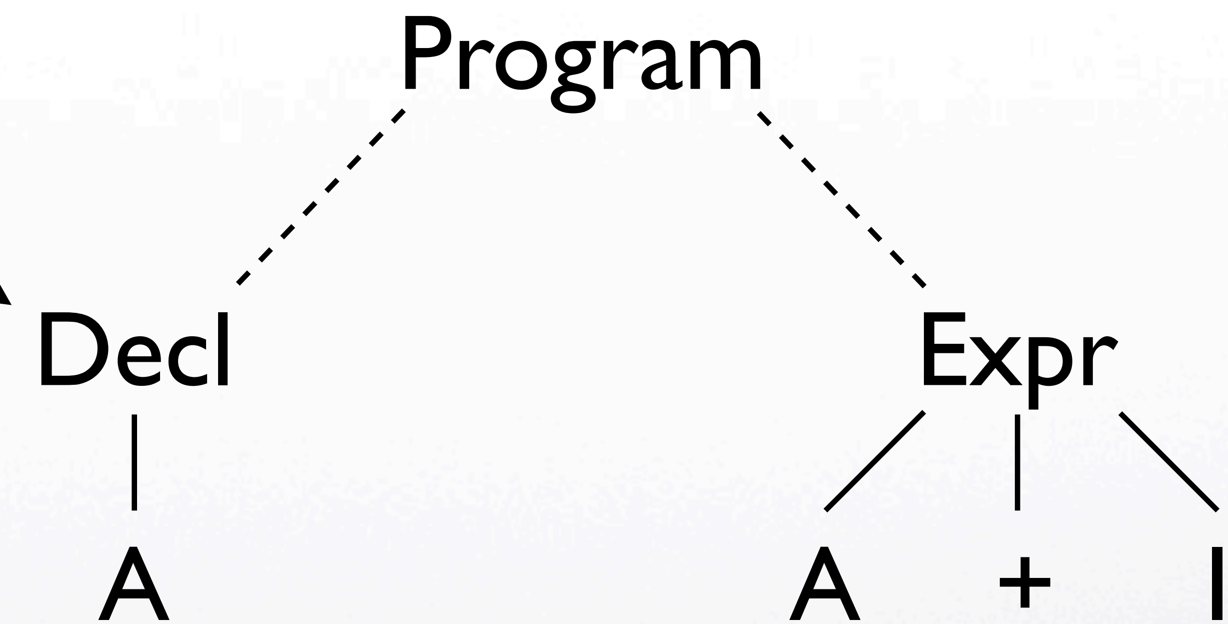
- Reference attributes
- Parameterized attributes
- Broadcast inherited attributes

JastAdd Symbol Tables

- Instead of building a separate symbol table, just decorate the parse tree nodes

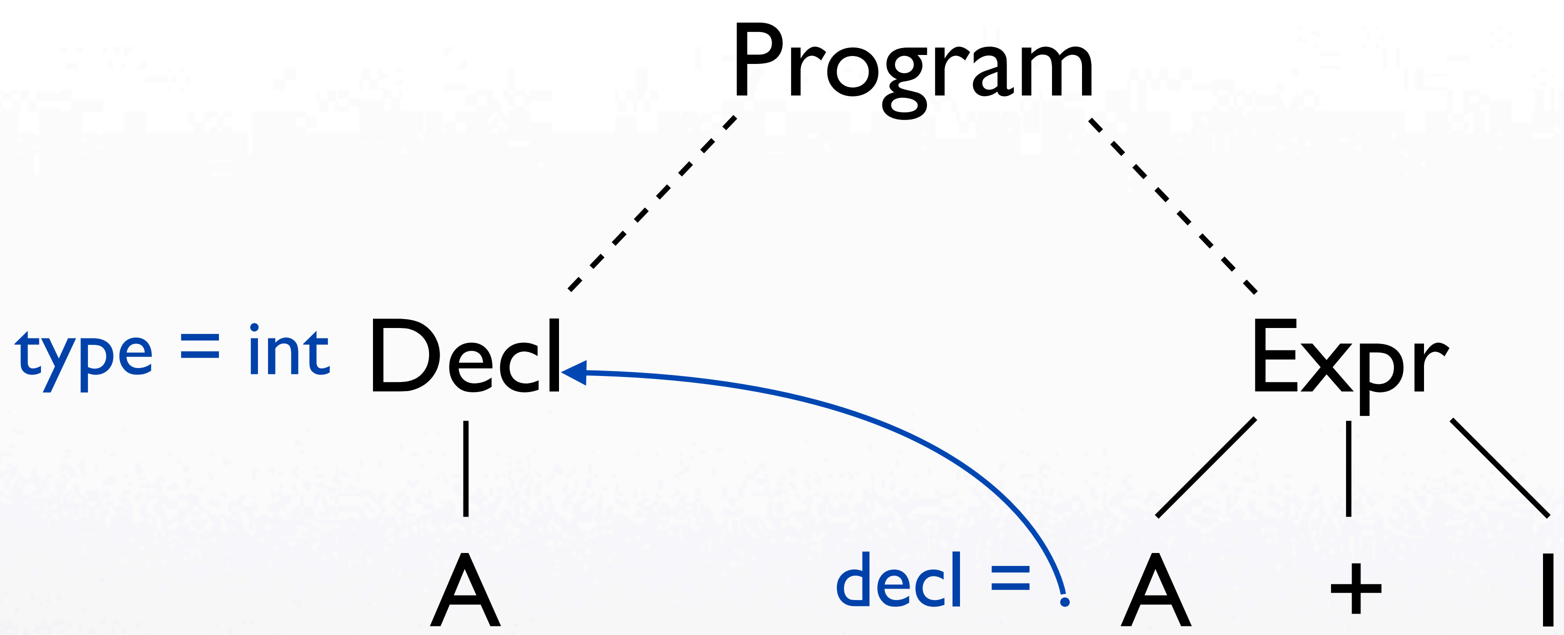
Example - Table

Symbol	Type	Decl
A	int	.
⋮	⋮	⋮



Lookup: table.lookup("A") returns a record

Example - Attributes



Lookup: A.getDecl() returns a Decl node

Detailed Example

- Synthetic attributes push declarations up to root node
- Inherited attributes push declarations down to use nodes

Detailed Example

Listing Declarations:

```
syn Set<Decl> ASTNode.listDecls();
eq ASTNode.listDecls() {
    Set<Decl> decls = new HashSet<Decl>();
    for(int i = 0; i < getNumChild(); i++) {
        decls.addAll(getChild(i).listDecls());
    }
    return decls;
}
eq Decl.listDecls() = Collections.singleton(this);
```

Detailed Example

Declaration Lookup:

```
syn Decl Program.lookupDecl(String name) {
    for(Decl d : listDecls()) {
        if(d.getName().equals(name)) {
            return d;
        }
    }
    return null;
}
```

Detailed Example

Lookup propagation:

```
inh Decl IDExpr.lookupDecl(String name);
eq Parent.getIDExpr().lookupDecl(String name) = lookupDecl(name);
```

Meta-Variable: substitute names of nodes with IDExpr children

Detailed Example

Link to Decl:

```
syn Decl IDExpr.getDecl() = lookupDecl(getName());
```

Advantages

- Modularity
- Extensibility
- Laziness

Disadvantages

- Definition of structure is less centralized
- May require more computation

Sources

- D. Knuth, Semantics of context-free languages, *Math. Sys. Theory*, 2: 1 (1968), 127–145.
- http://jastadd.org/