Context-sensitive Analysis
Part II
Chapter 4 (up to Section 4.3)

Attribute Grammars

Add rules to compute the decimal value of a signed binary number

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

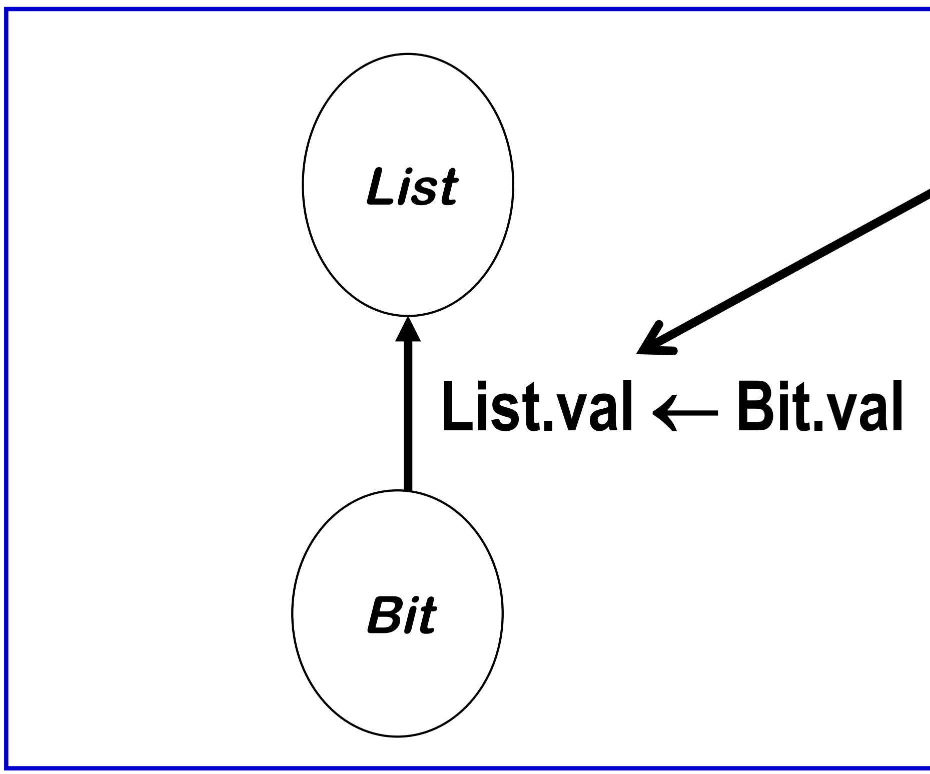
Two kinds of Attributes

- Synthesized attribute
  - → Bottom-Up flow of values
  - → Depends on values from the node itself, children, or constants
- Inherited attribute
  - → Top-down flow of values
  - → Depends on values from siblings, parent and constants

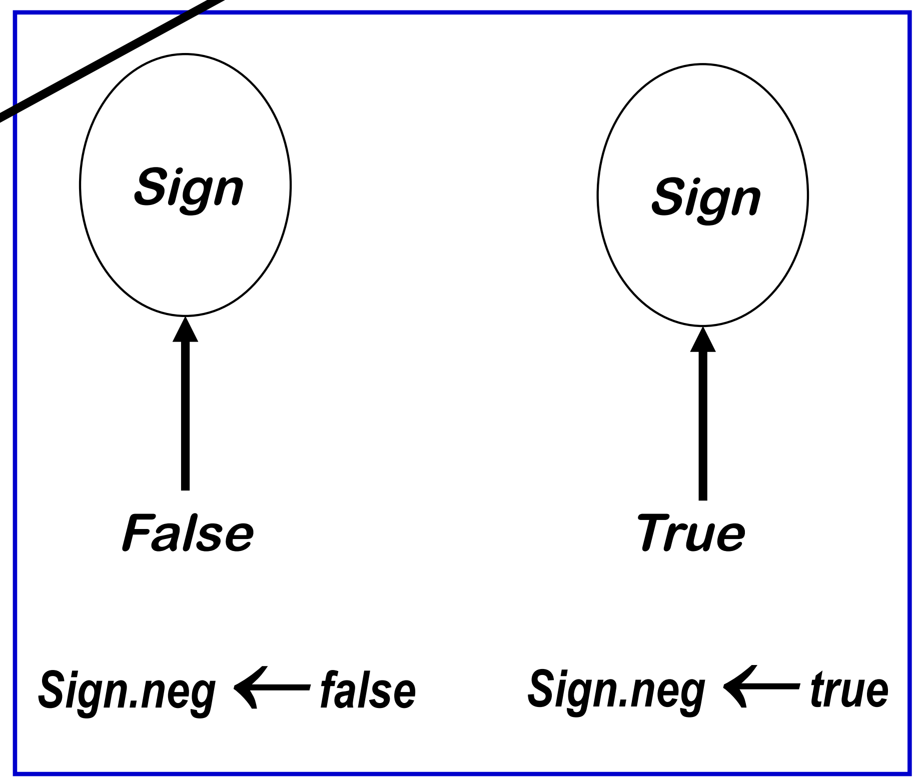
Synthesized Attributes

Depends on values from the node itself, children, or constants

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← − *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Bottom-Up flow (Children)

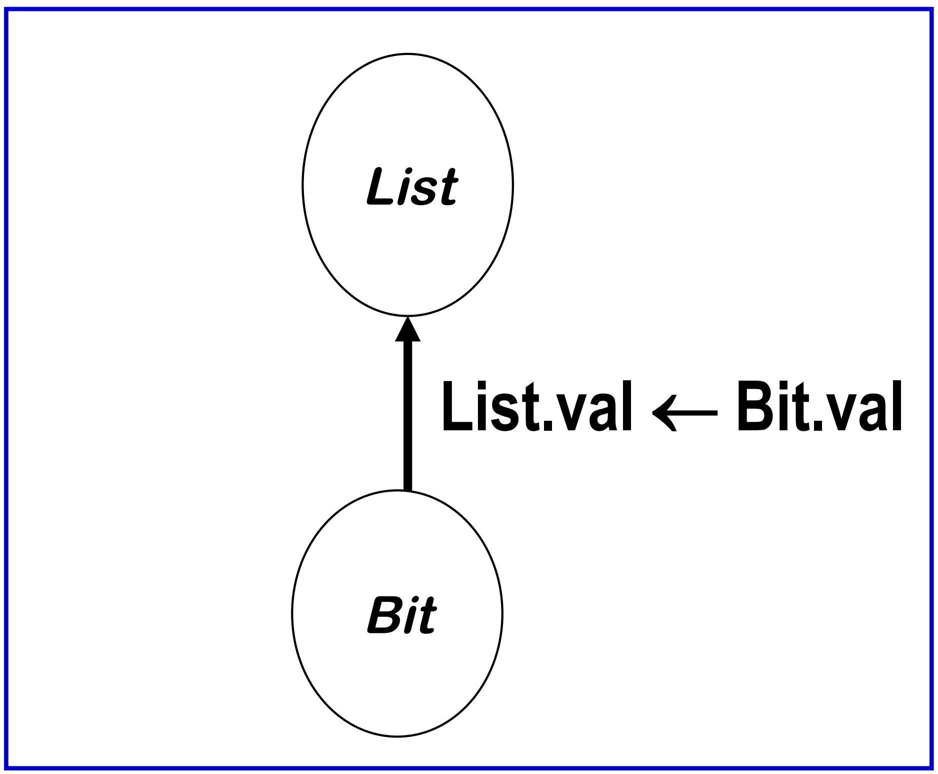


Constants

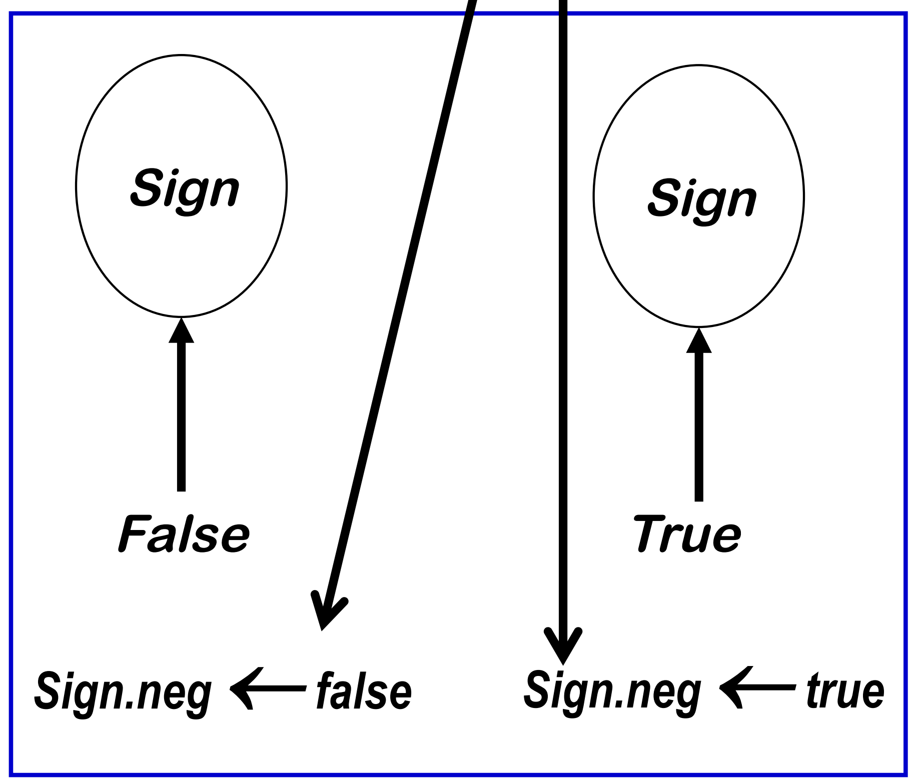
Synthesized Attributes

Depends on values from the node itself, children, or constants

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← − *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Bottom-Up flow (Children)

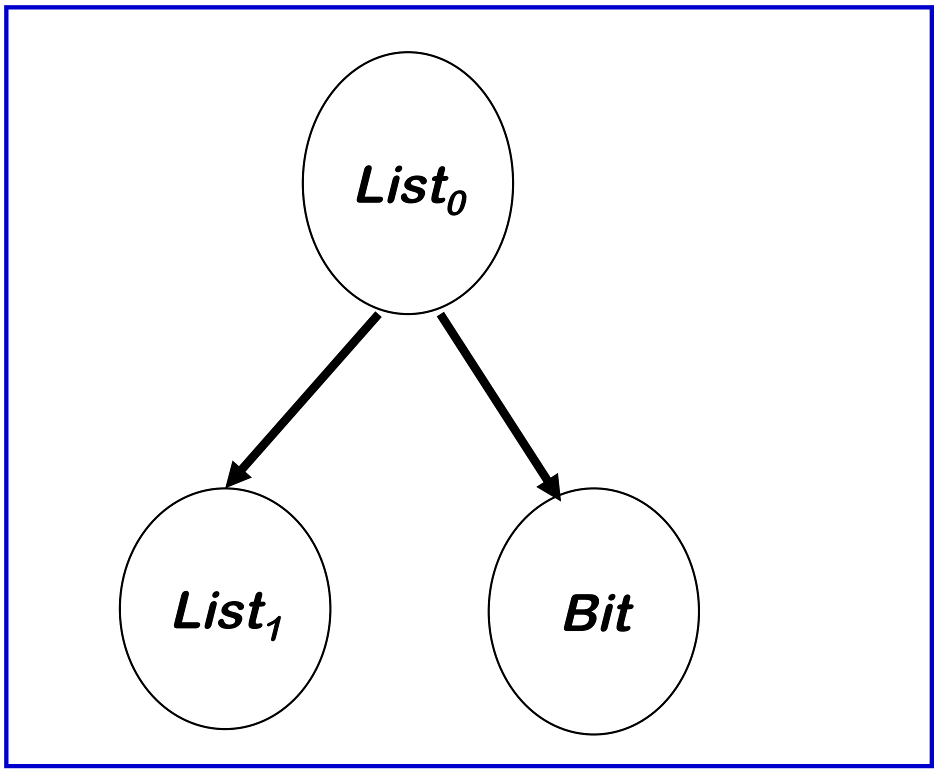


Constants

Inherited Attributes

Depends on values from siblings, parent and constants

Productions			Attribution Rules
Number	→	*Sign List*	$List.pos \leftarrow 0$ If $Sign.neg$ then $Number.val \leftarrow - List.val$ else $Number.val \leftarrow List.val$
Sign	→	+	$Sign.neg \leftarrow false$
	|	–	$Sign.neg \leftarrow true$
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
Bit	→	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Top-down flow

$List_1.pos \leftarrow List_0.pos + 1$

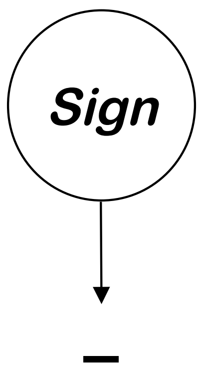
$Bit.pos \leftarrow List_0.pos$

Back to the Examples

For “<u>–1</u>”

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

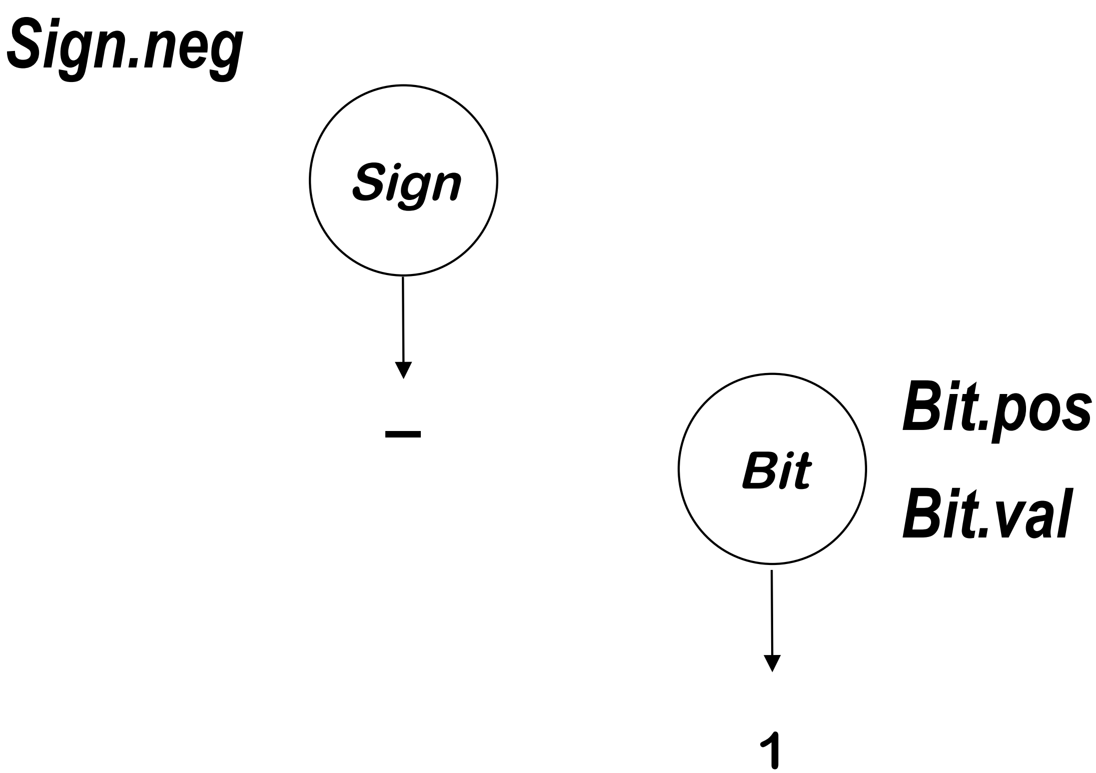
Sign.neg



Back to the Examples

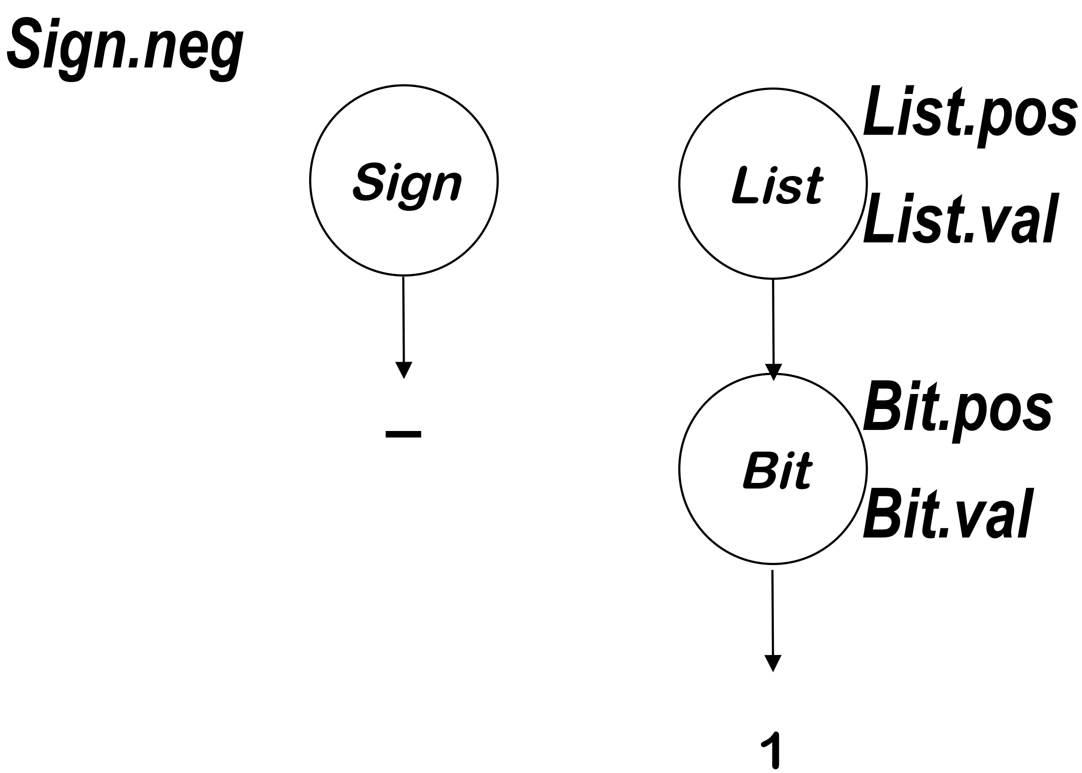
For "–1"

Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val



Back to the Examples

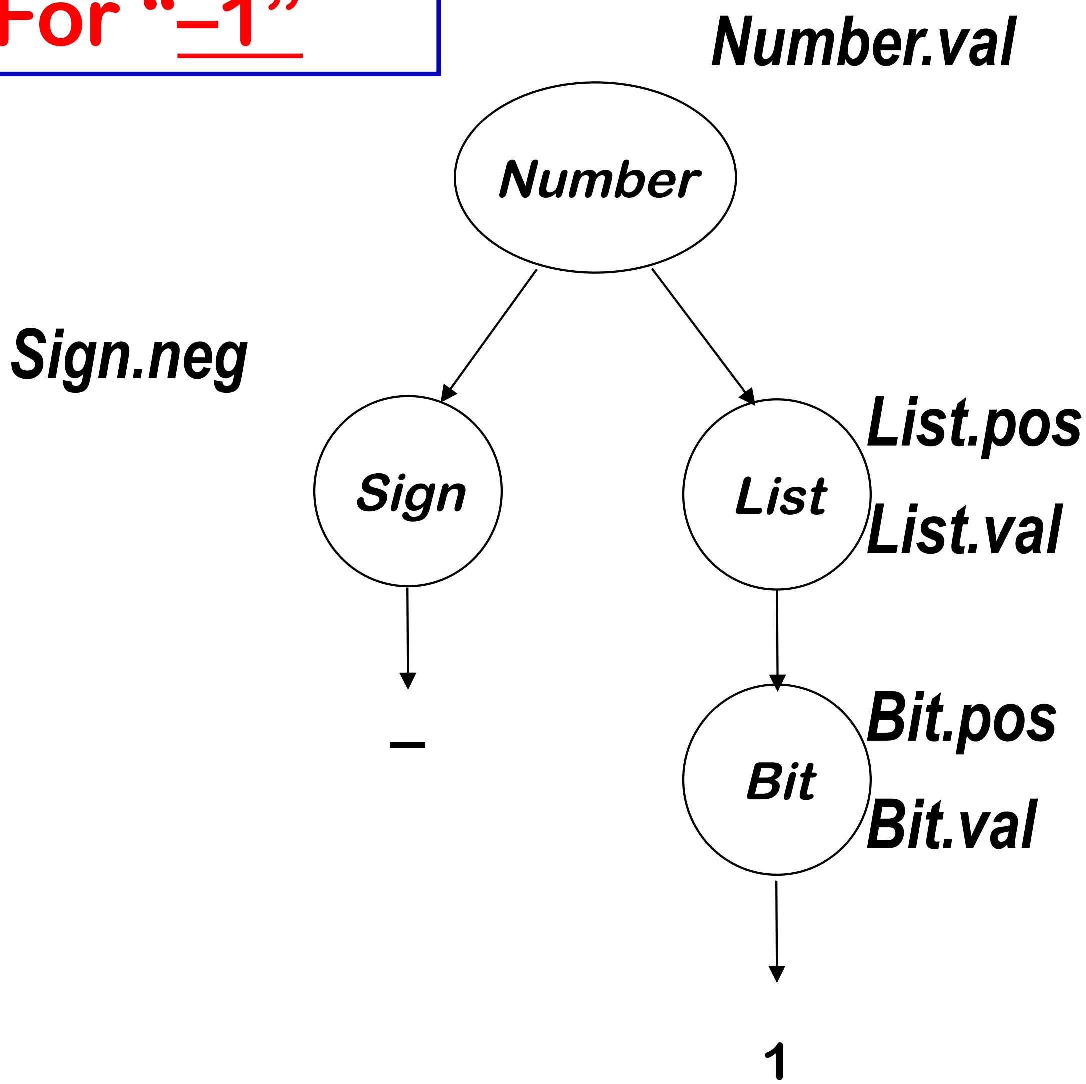
For "<u>–1</u>"



Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

Back to the Examples

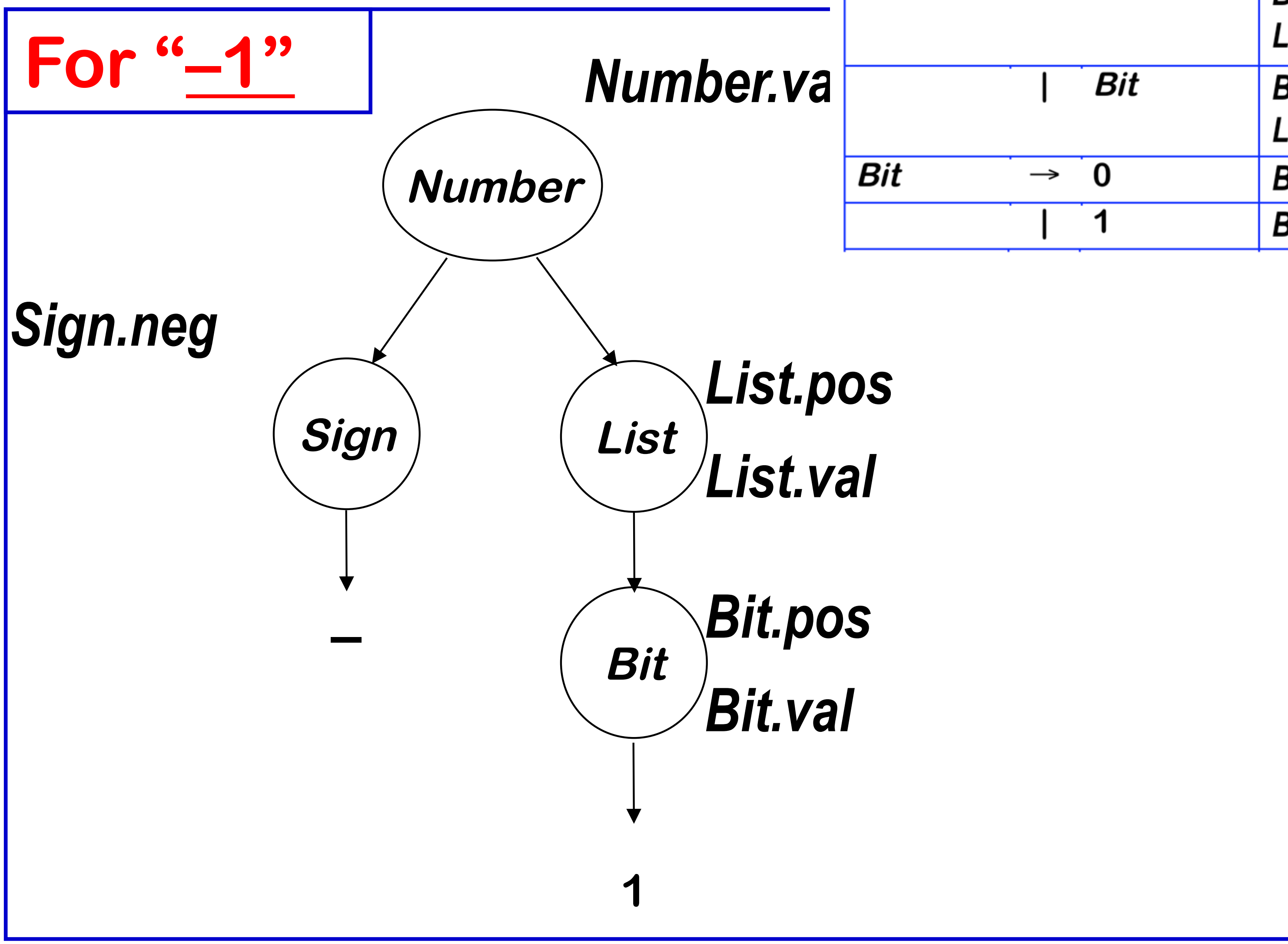
For "–1"



Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val

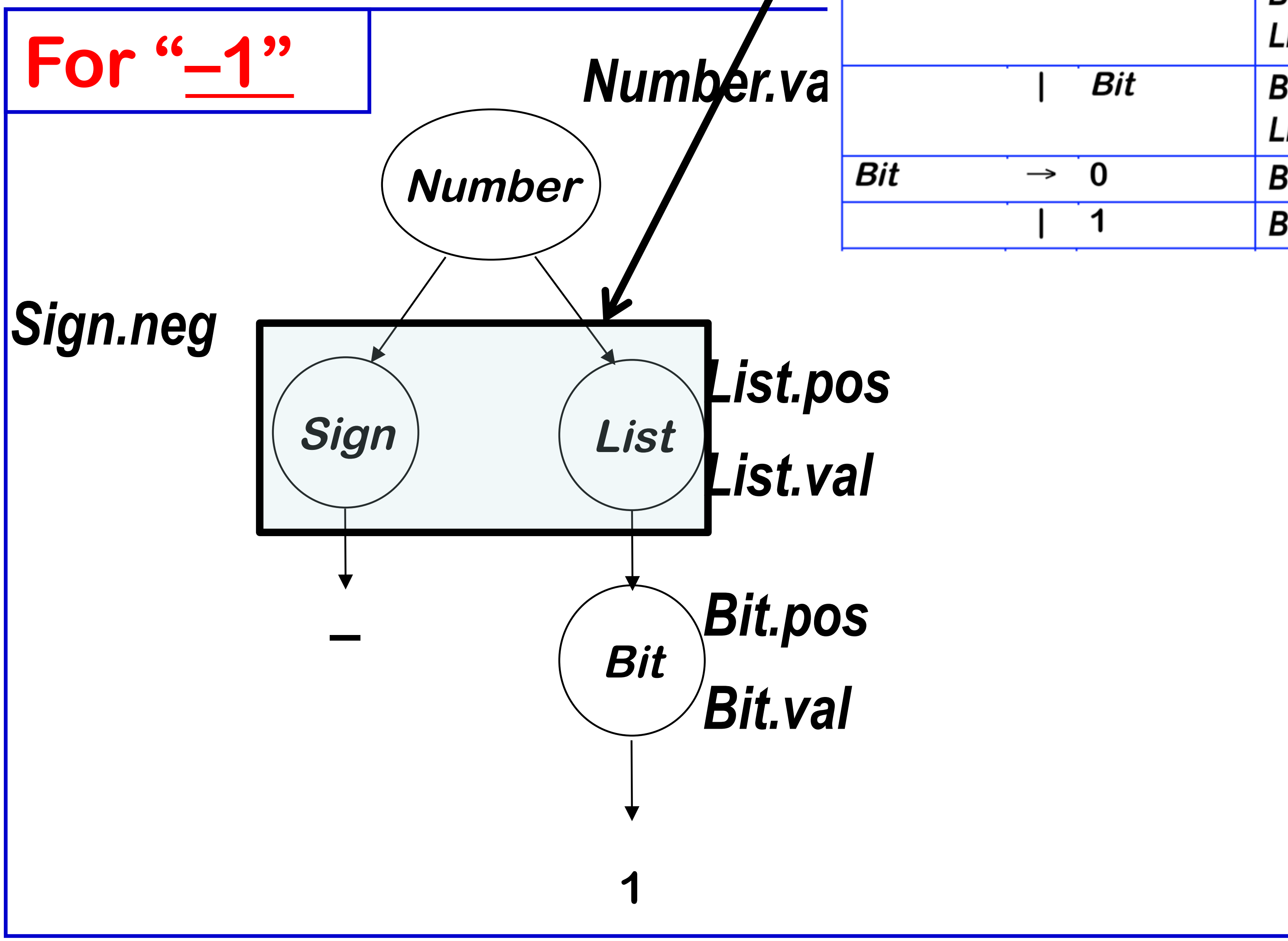
Back to the Examples

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Back to the Examples

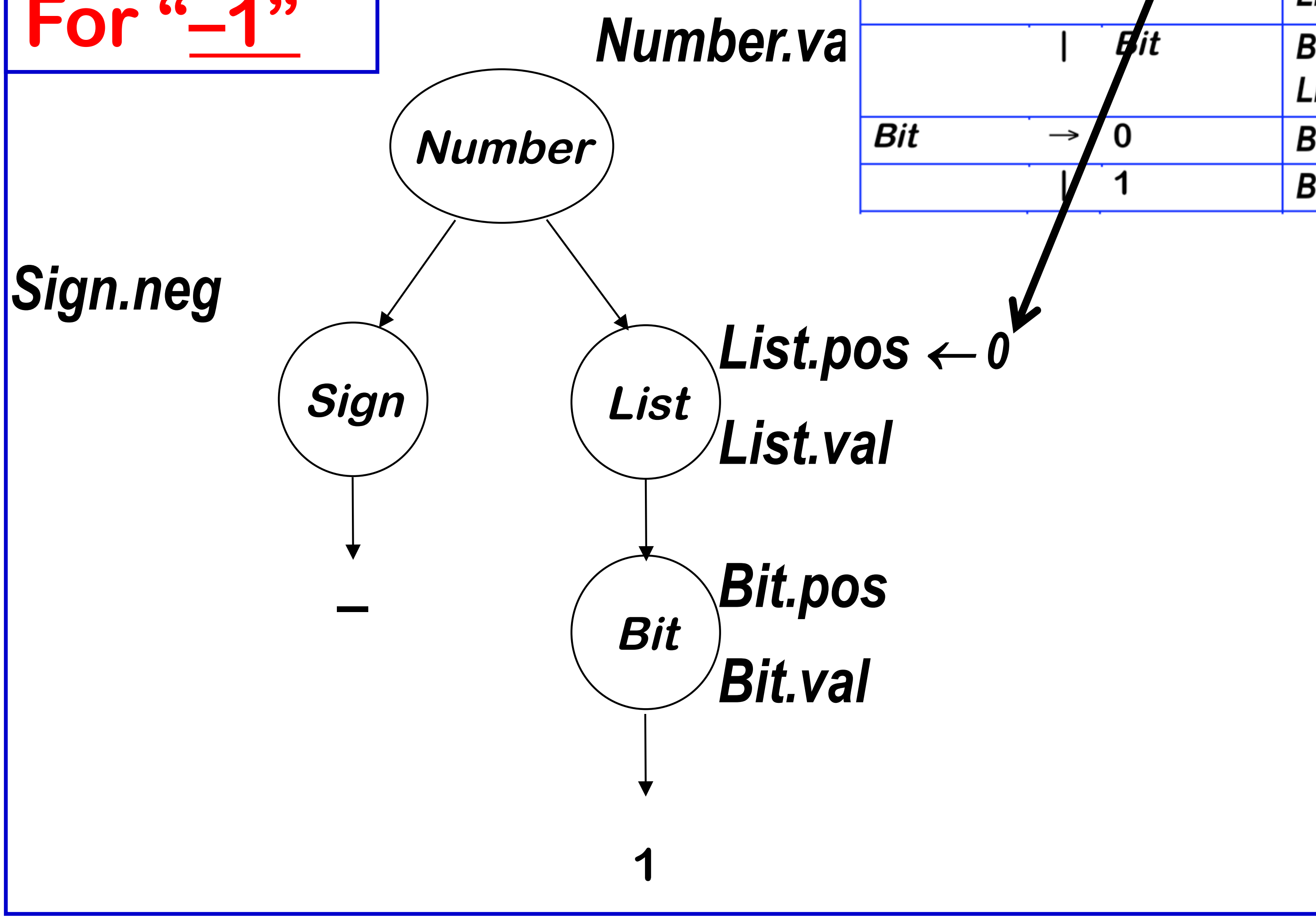
Productions			Attribution Rules
Number	→	Sign List	List.pos ← 0 If Sign.neg then Number.val ← – List.val else Number.val ← List.val
Sign	→	+	Sign.neg ← false
	|	–	Sign.neg ← true
$List_0$	→	$List_1$ Bit	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	Bit	Bit.pos ← List.pos List.val ← Bit.val
Bit	→	0	Bit.val ← 0
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Back to the Examples

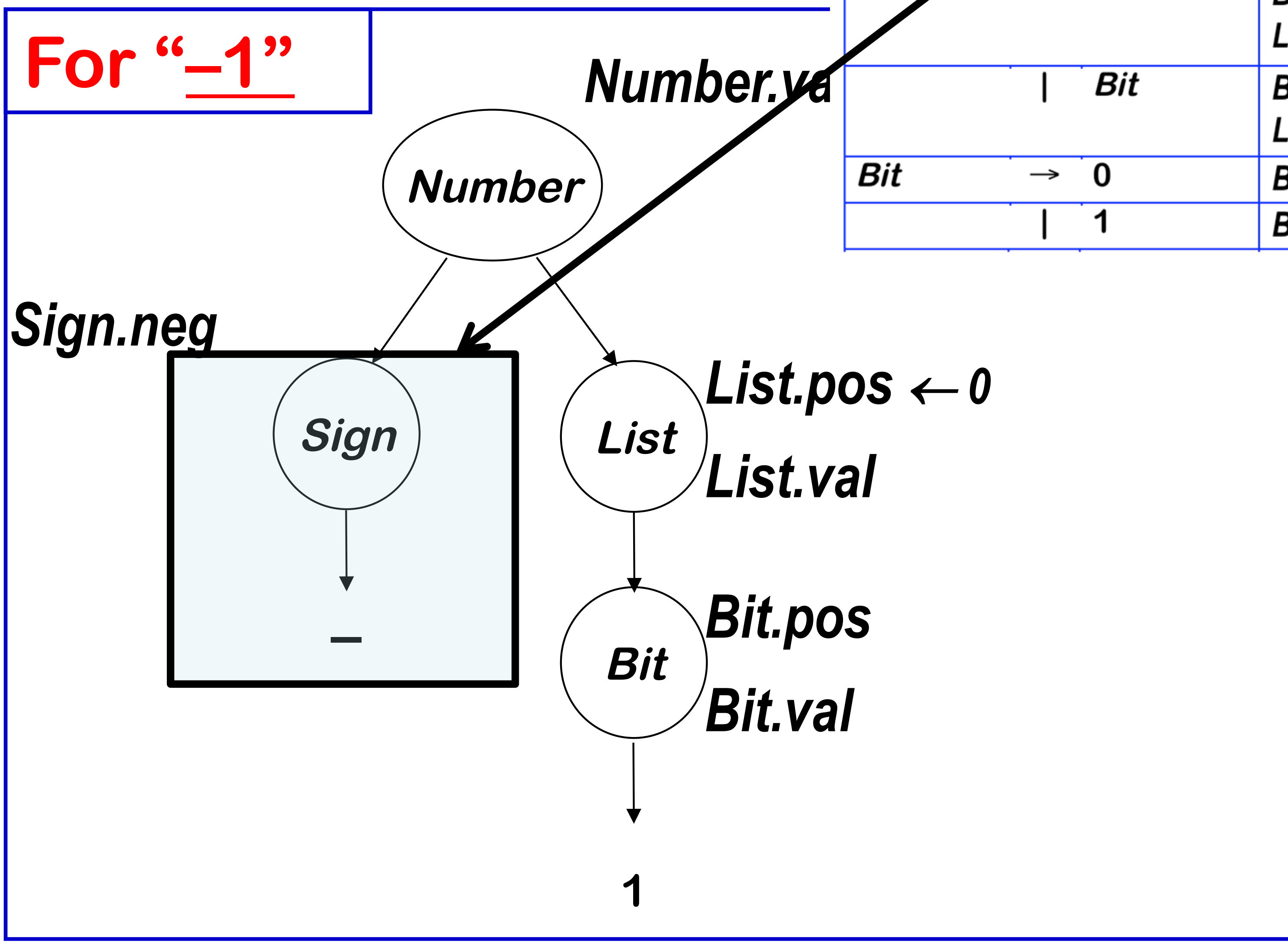
Productions			Attribution Rules
Number	→	Sign List	List.pos ← 0 If Sign.neg then Number.val ← – List.val else Number.val ← List.val
Sign	→	+	Sign.neg ← false
	|	–	Sign.neg ← true
$List_0$	→	$List_1$ Bit	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	Bit	Bit.pos ← List.pos List.val ← Bit.val
Bit	→	0	Bit.val ← 0
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

For "–1"



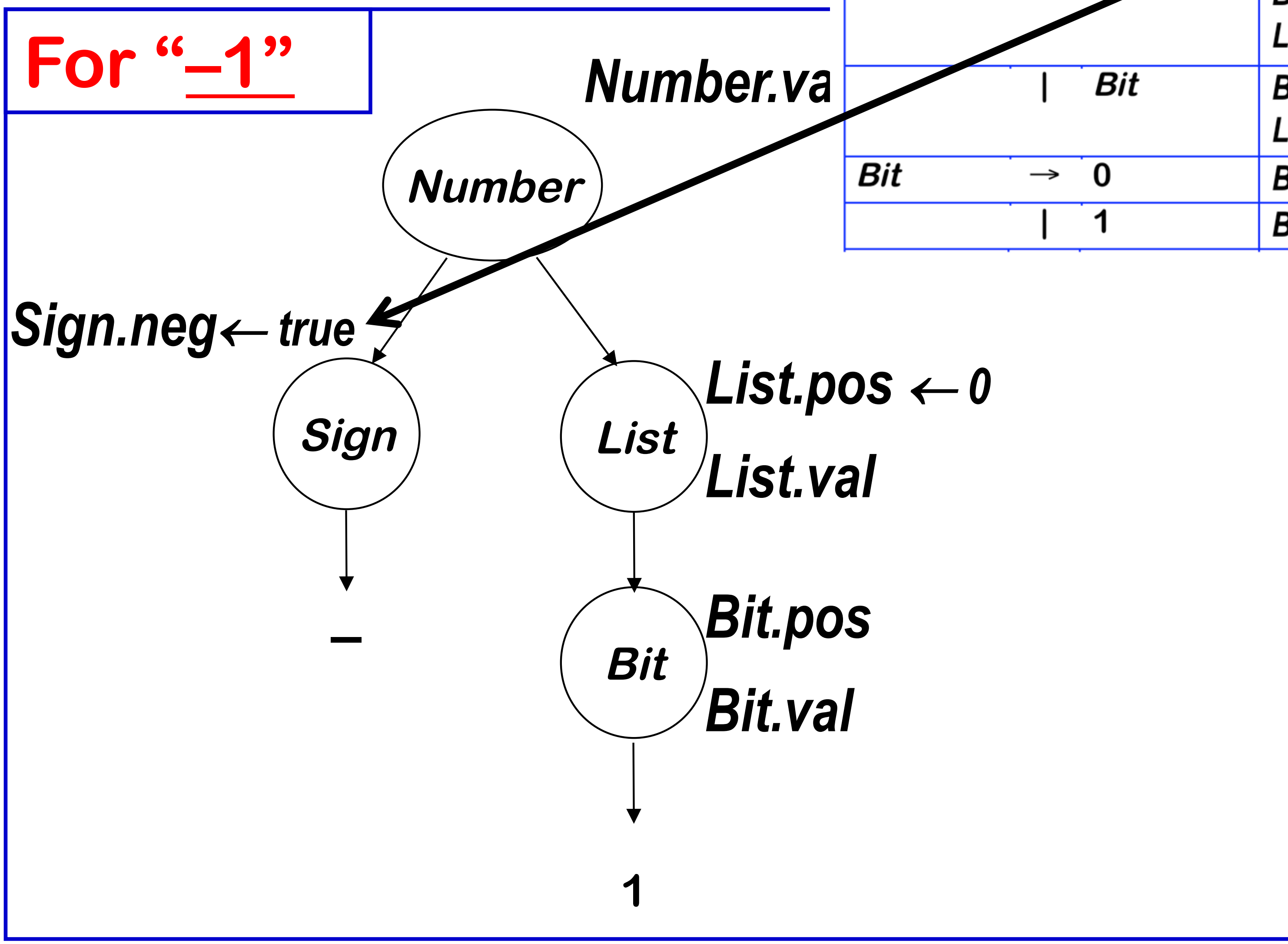
Back to the Examples

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Back to the Examples

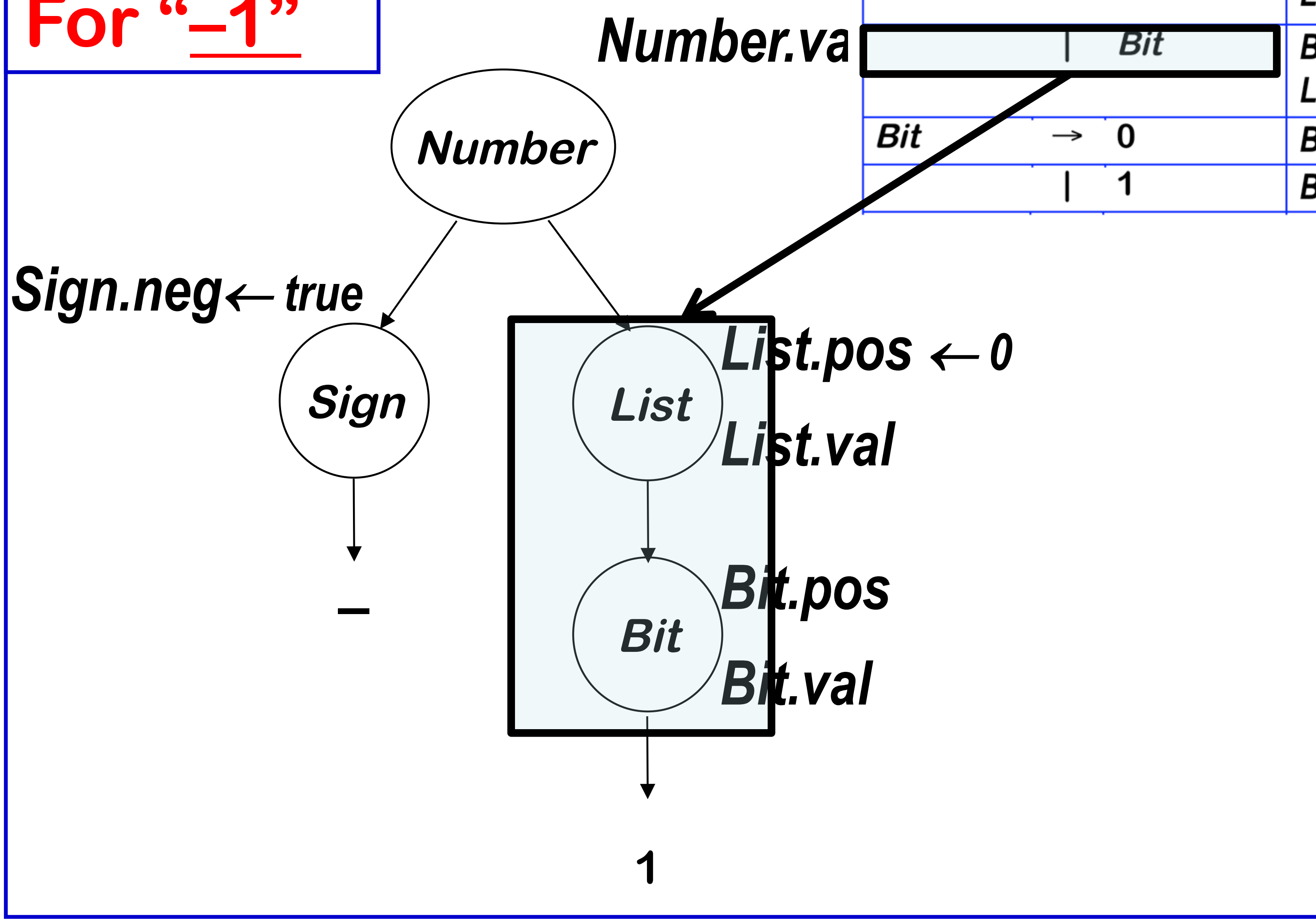
Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Back to the Examples

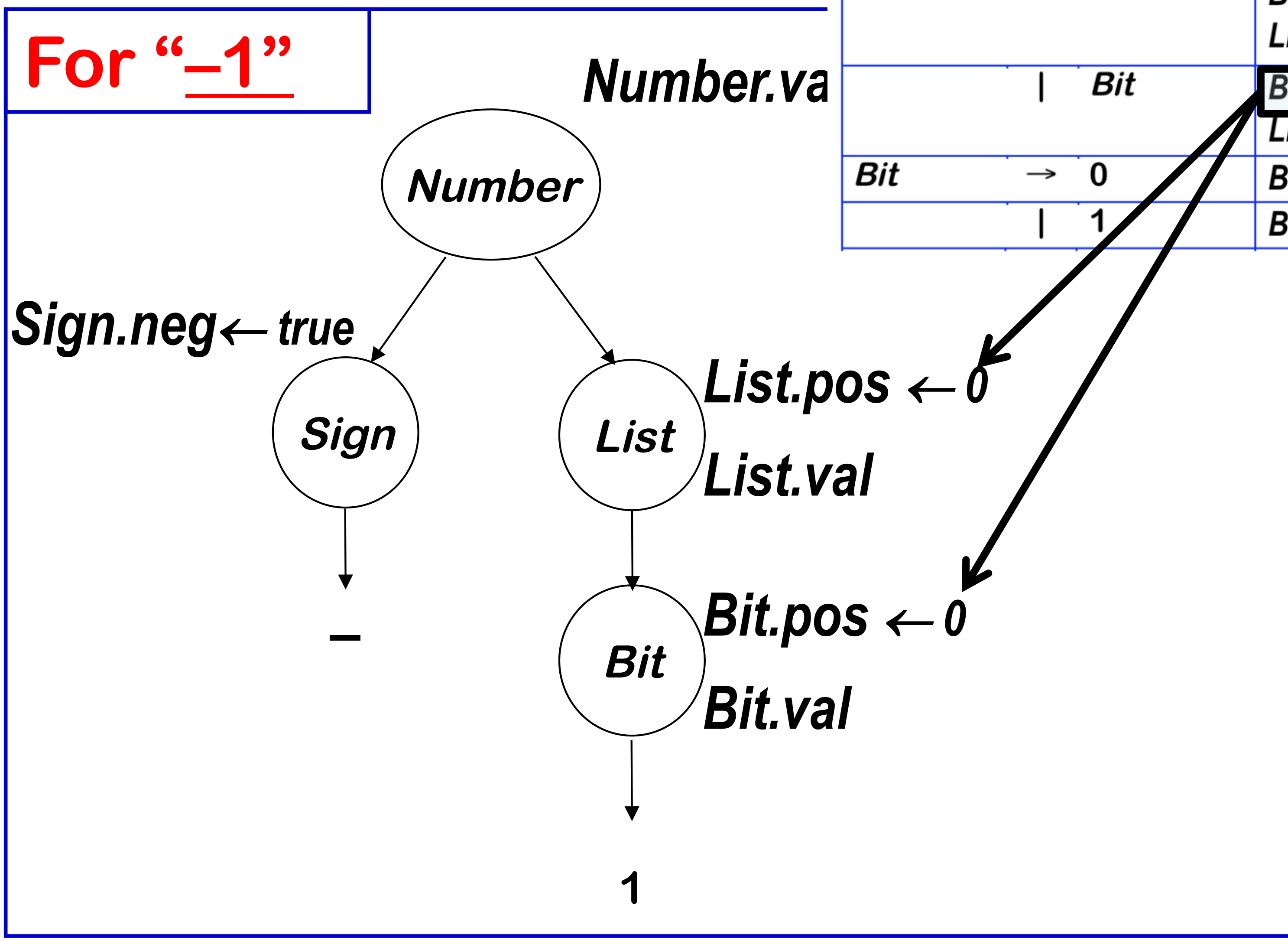
Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← 0 If *Sign.neg* then *Number.val* ← – *List.val* else *Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← false
	|	–	*Sign.neg* ← true
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← 0
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

For "–1"



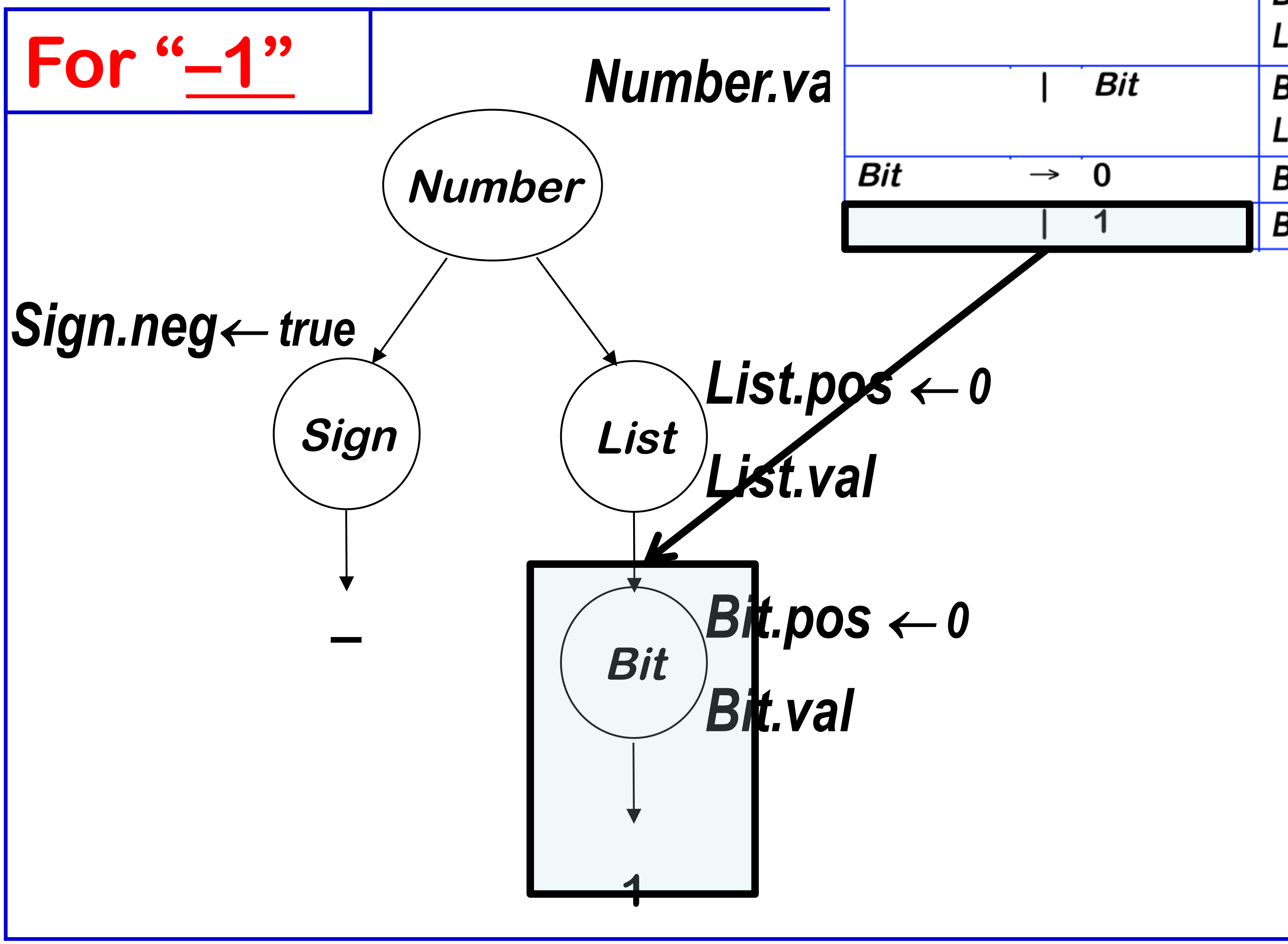
Back to the Examples

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	**+**	*Sign.neg* ← *false*
	|	**–**	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



Back to the Examples

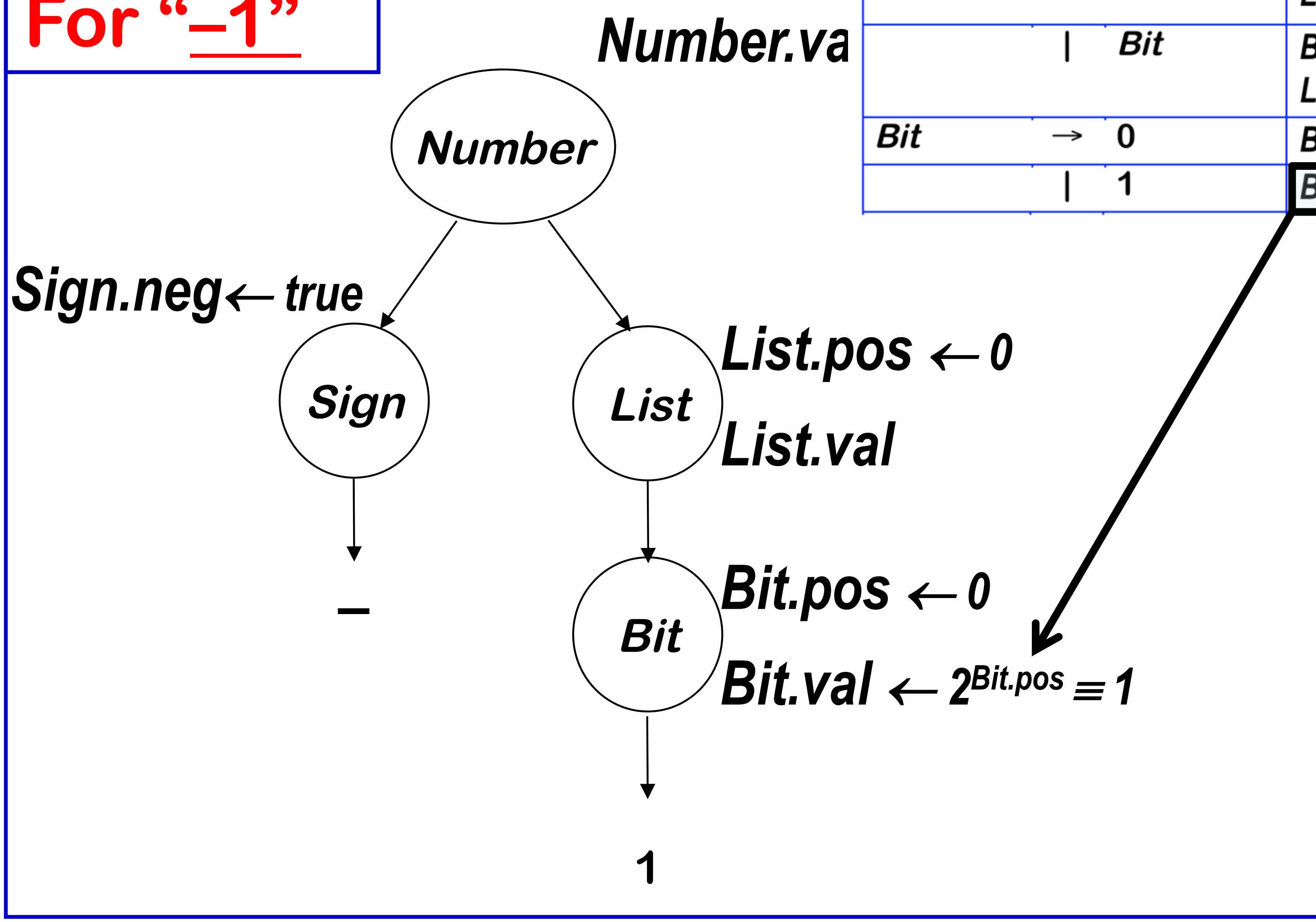
Productions			Attribution Rules
Number	→	*Sign List*	$List.pos \leftarrow 0$ If $Sign.neg$ then $Number.val \leftarrow -List.val$ else $Number.val \leftarrow List.val$
Sign	→	+	$Sign.neg \leftarrow$ false
	|	–	$Sign.neg \leftarrow$ true
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	$Bit.pos \leftarrow List.pos$ $List.val \leftarrow Bit.val$
Bit	→	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$



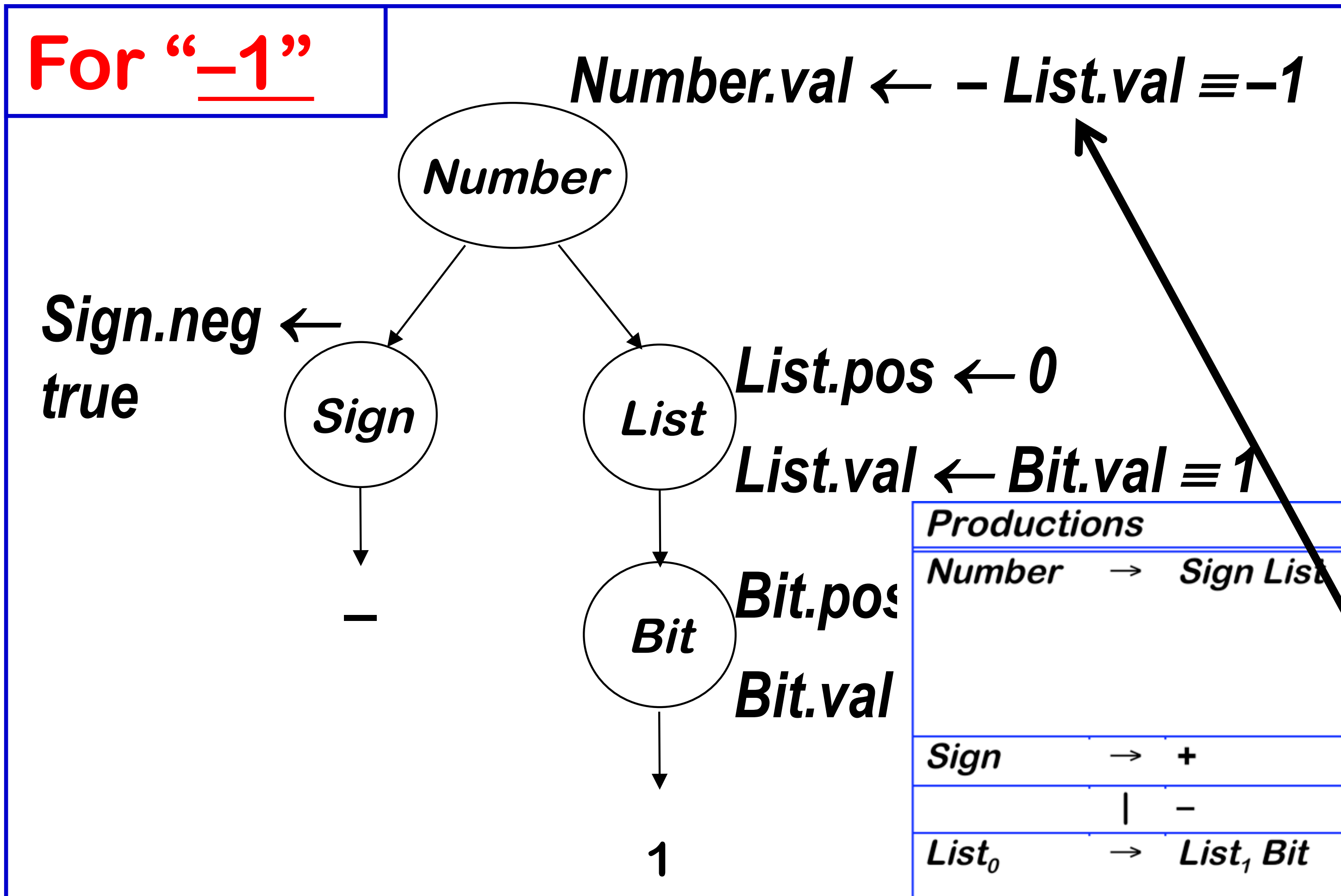
Back to the Examples

Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← 0 *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* ← *List.pos* *List.val* ← *Bit.val*
Bit	→	0	*Bit.val* ← 0
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

For "–1"



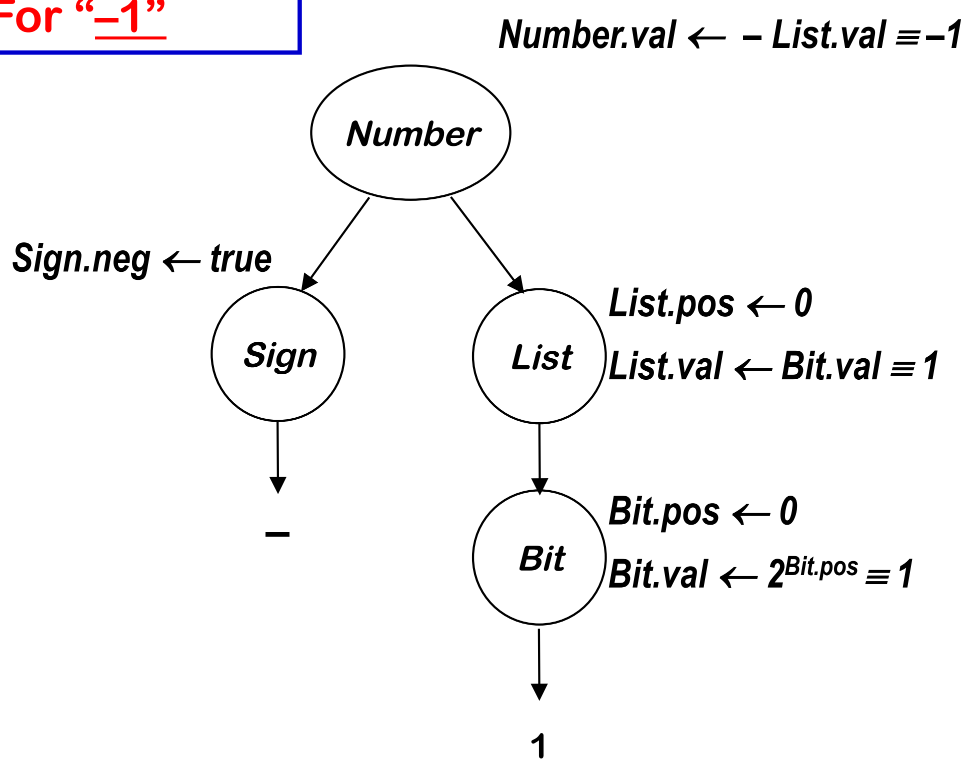
Back to the Examples



Productions			Attribution Rules
Number	→	*Sign List*	*List.pos ← 0* *If Sign.neg then Number.val ← – List.val* *else Number.val ← List.val*
Sign	→	+	*Sign.neg ← false*
	|	–	*Sign.neg ← true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos ← List.pos* *List.val ← Bit.val*
Bit	→	0	*Bit.val ← 0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Back to the Examples

For "–1"



Evaluation order must be consistent with the **attribute dependence graph**

One possible evaluation order:

1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

Other orders are possible

Attributes + Parse Tree

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Rules & parse tree define an attribute dependence graph
  - → Dependence graph must be non-circular (no cycles)

This produces a high-level, functional specification

Using Attribute Grammars

Attribute grammars can specify context-sensitive actions

- Take values from syntax
- Perform computations with values
- Insert type tests, type inference, logic, …

Evaluation Methods

Dynamic, dependence-based methods

- Build the parse tree
- Build the dependence graph
- Topological sort the dependence graph
- Define attributes in topological order

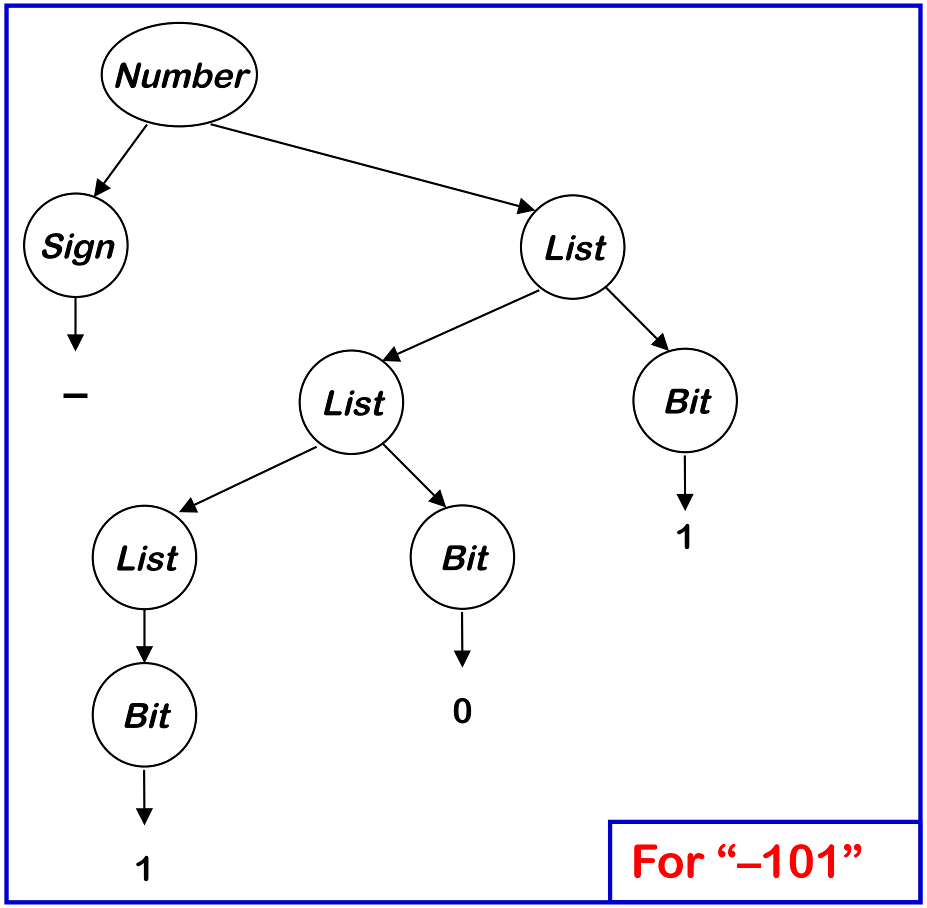
Rule-based methods *(treewalk)*

- Analyze rules at compiler-generation time
- Determine a fixed (static) ordering
- Evaluate nodes in that order

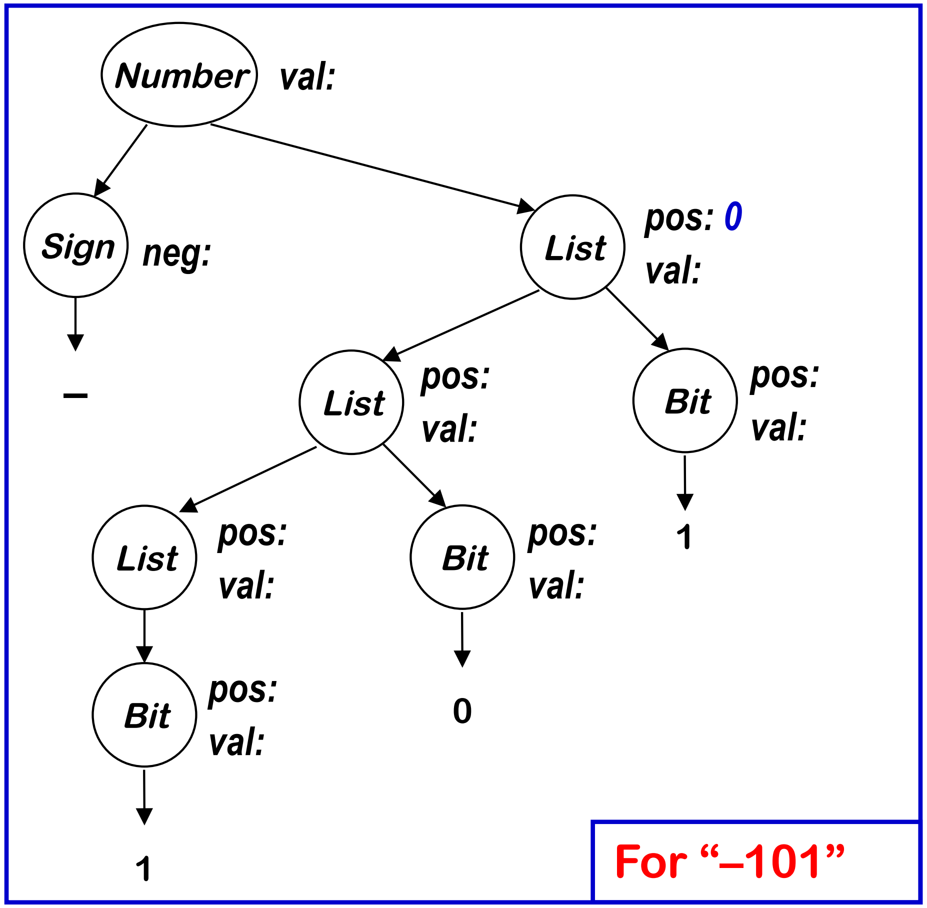
Oblivious methods *(passes, dataflow)*

- Ignore rules & parse tree
- Pick a convenient order (at design time) & use it

Back to the Example

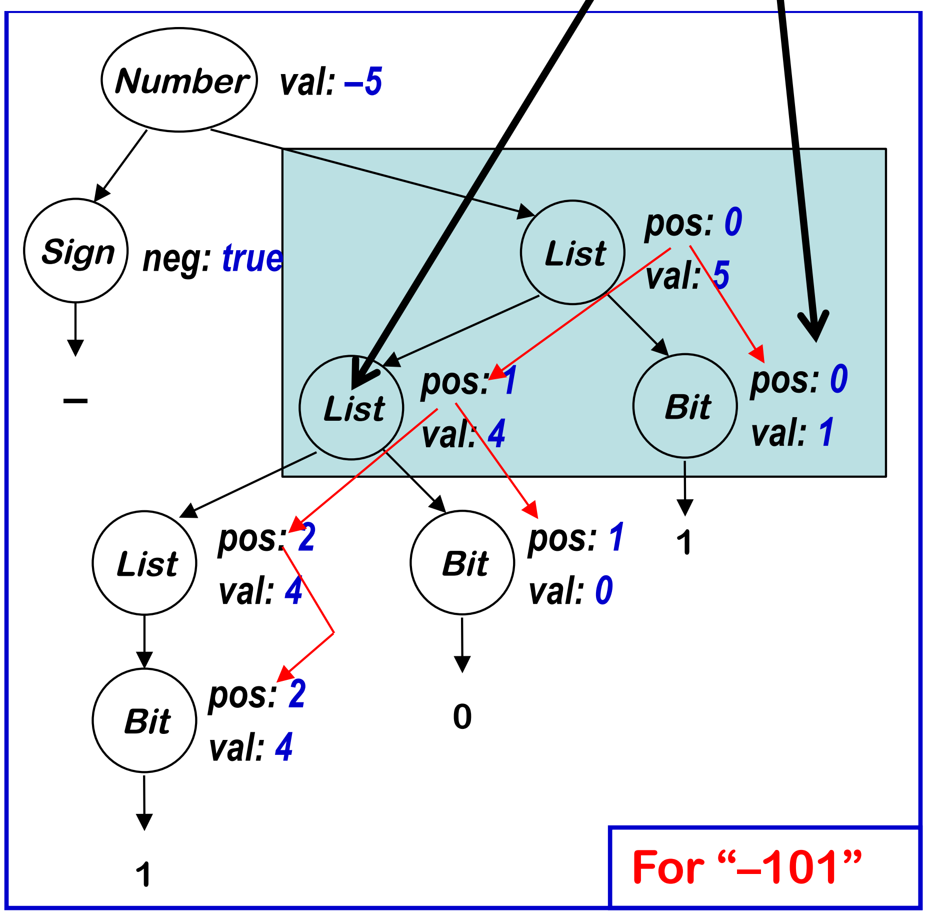


Back to the Example



Back to the Example

$List_0$	$\rightarrow$	$List_1\ Bit$	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$

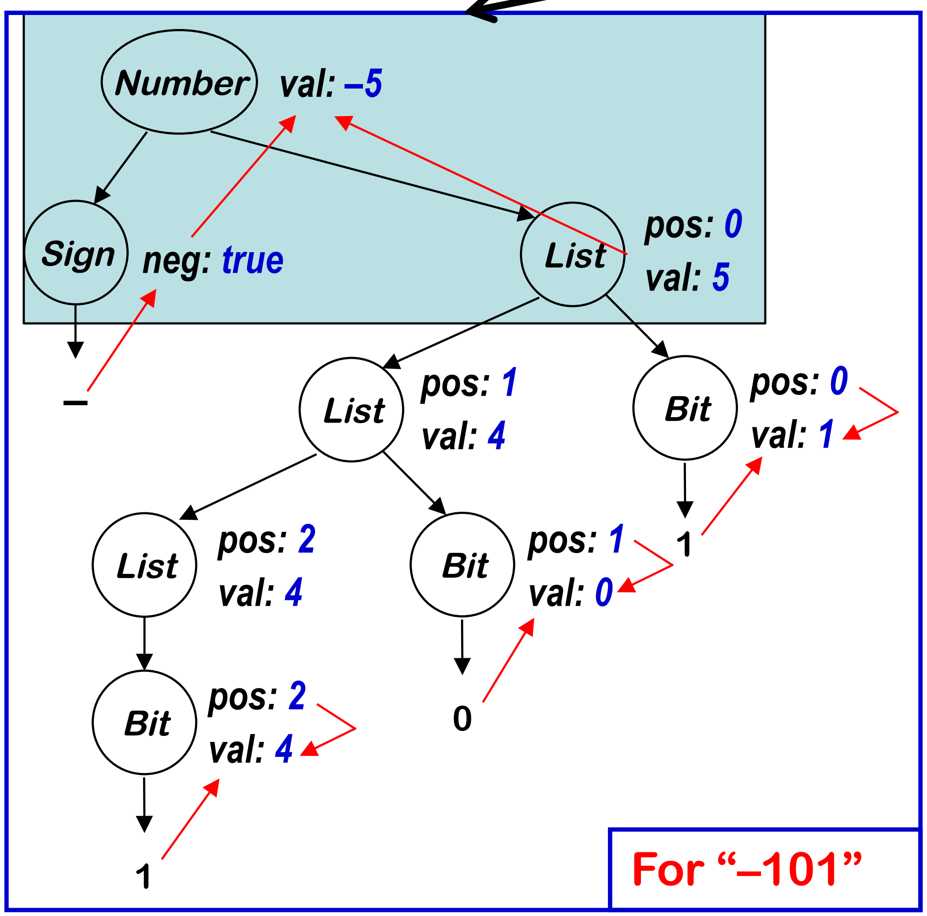


Inherited Attributes

Note: downward flow (pointing arrows) of information

Back to the Example

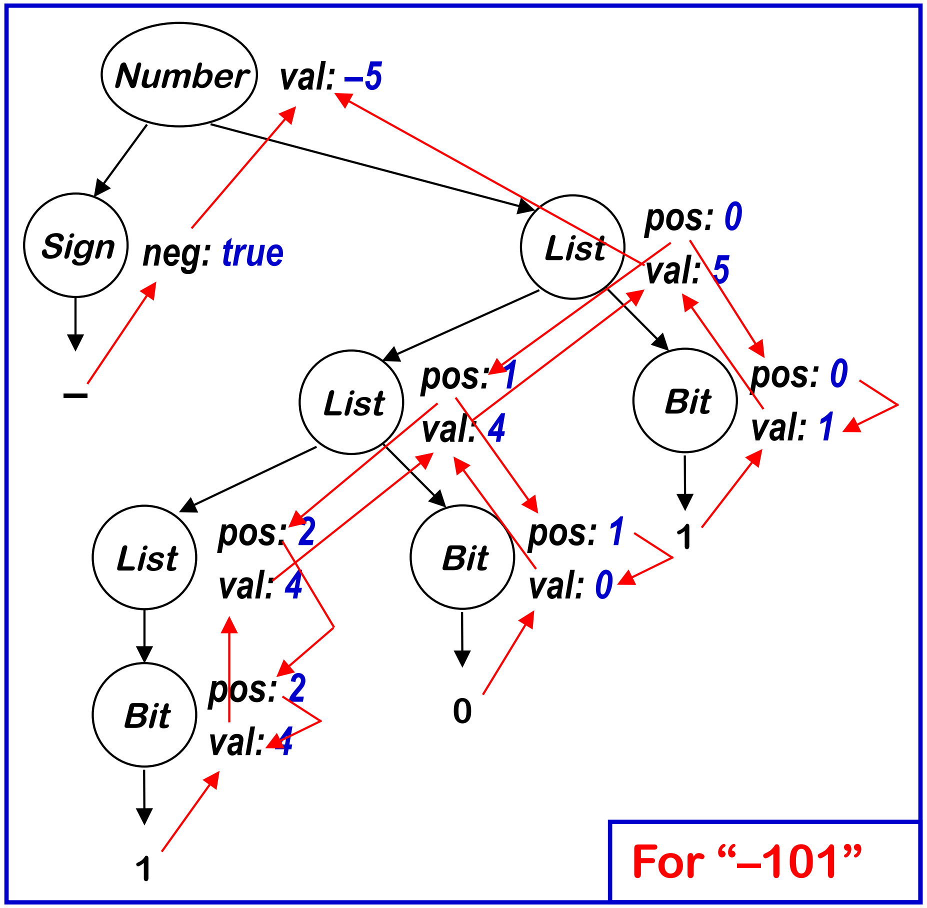
Number → *Sign List*		*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*



Synthesized attributes

Note: upward flow (pointing arrows) of information and the flow from node’s (self) attributes

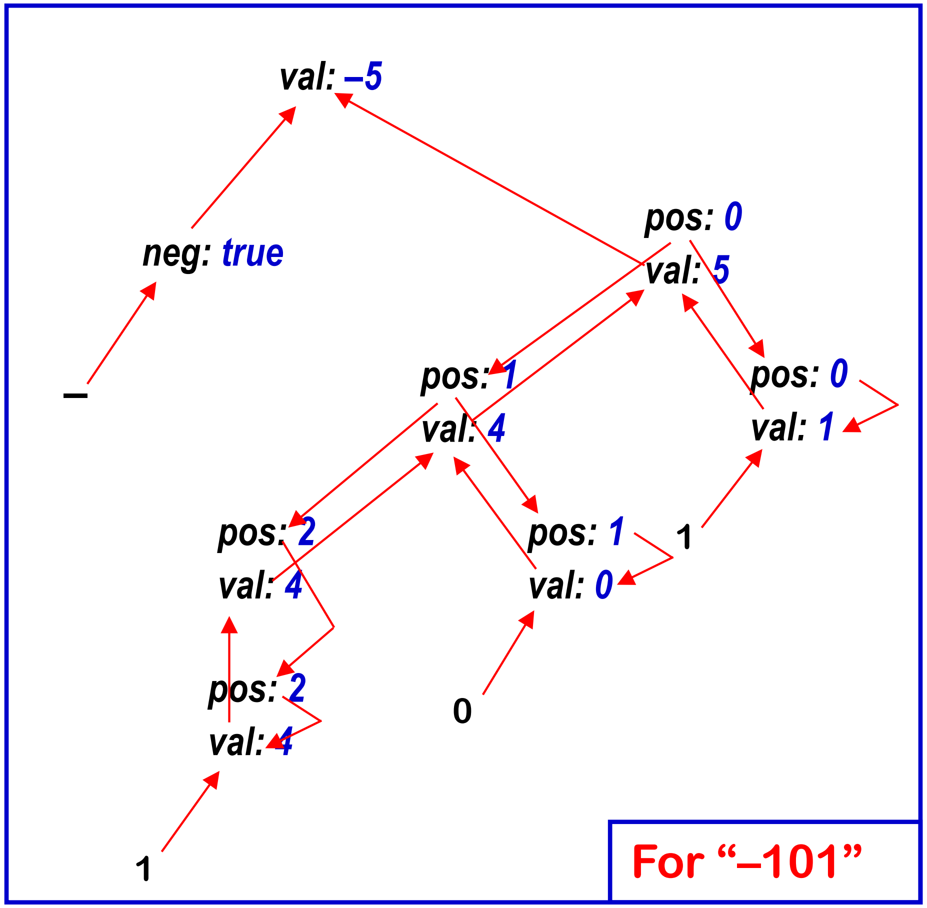
Back to the Example



If we show the computation …

then peel away the parse tree …

Back to the Example



All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The dependence graph must be acyclic (no cycles!)

An Extended Example

Grammar for a basic block

$Block_0$	→	$Block_1$ *Assign*
	|	*Assign*
Assign	→	*Ident* = *Expr* ;
$Expr_0$	→	$Expr_1$ + *Term*
	|	$Expr_1$ – *Term*
	|	*Term*
$Term_0$	→	$Term_1$ * *Factor*
	|	$Term_1$ / *Factor*
	|	*Factor*
Factor	→	(*Expr*)
	|	*Number*
	|	*Identifier*

An Extended Example

Grammar for a basic block

$Block_0$	$\rightarrow$	$Block_1$ *Assign*
	|	*Assign*
Assign	$\rightarrow$	*Ident* = *Expr* ;
$Expr_0$	$\rightarrow$	$Expr_1$ + *Term*
	|	$Expr_1$ – *Term*
	|	*Term*
$Term_0$	$\rightarrow$	$Term_1$ * *Factor*
	|	$Term_1$ / *Factor*
	|	*Factor*
Factor	$\rightarrow$	(*Expr*)
	|	*Number*
	|	*Identifier*

An Extended Example

Grammar for a basic block

$Block_0$	→	$Block_1$ $Assign$
	|	$Assign$
$Assign$	→	$Ident = Expr$;
$Expr_0$	→	$Expr_1 + Term$
	|	$Expr_1 - Term$
	|	$Term$
$Term_0$	→	$Term_1 * Factor$
	|	$Term_1 / Factor$
	|	$Factor$
$Factor$	→	($Expr$)
	|	$Number$
	|	$Identifier$

An Extended Example

Grammar for a basic block

$Block_0$	→	$Block_1$ Assign
	|	Assign
Assign	→	Ident = Expr ;
$Expr_0$	→	$Expr_1$ + Term
	|	$Expr_1$ – Term
	|	Term
$Term_0$	→	$Term_1$ * Factor
	|	$Term_1$ / Factor
	|	Factor
Factor	→	(Expr)
	|	Number
	|	Identifier

Example basic block

```
a = -5
b = a * 17
c = b / 2
d = a + b - c
```

How many clock cycles will this block take to execute?

An Extended Example

Grammar for a basic block

Block$_0$	→	*Block$_1$ Assign*
	|	*Assign*
Assign	→	*Ident = Expr ;*
Expr$_0$	→	*Expr$_1$ + Term*
	|	*Expr$_1$ – Term*
	|	*Term*
Term$_0$	→	*Term$_1$ * Factor*
	|	*Term$_1$ / Factor*
	|	*Factor*
Factor	→	*(Expr)*
	|	*Number*
	|	*Identifier*

Simple *Attribute Grammar*

Estimate cycle count for the block of instructions

- **Each operation has a COST**
- **Add them, bottom up**
- **Assume a load per value**
- **Assume no reuse**

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

Block₀	→	*Block₁ Assign*	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
	|	*Assign*	$Block_0.cost \leftarrow Assign.cost$
Assign	→	*Ident = Expr ;*	Assign.cost ← **COST**(store) + Expr.cost
Expr₀	→	*Expr₁ + Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*Expr₁ – Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*Term*	$Expr_0$.cost ← Term.cost
Term₀	→	*Term₁ * Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(mult) + Factor.cost
	|	*Term₁ / Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(div) +Factor.cost
	|	*Factor*	$Term_0$.cost ← Factor.cost
Factor	→	*(Expr)*	Factor.cost ← Expr.cost
	|	*Number*	Factor.cost ← **COST**(loadI)
	|	*Identifier*	Factor.cost ← **COST**(load)

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

$Block_0$	→	*$Block_1$ Assign*	$Block_0$.cost ← $Block_1$.cost + Assign.cost
	|	*Assign*	$Block_0$.cost ← Assign.cost
Assign	→	*Ident = Expr ;*	Assign.cost ← **COST**(store) + Expr.cost
$Expr_0$	→	*$Expr_1$ + Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*$Expr_1$ - Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*Term*	$Expr_0$.cost ← Term.cost
$Term_0$	→	*$Term_1$ * Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(mult) + Factor.cost
	|	*$Term_1$ / Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(div) +Factor.cost
	|	*Factor*	$Term_0$.cost ← Factor.cost
Factor	→	*(Expr)*	Factor.cost ← Expr.cost
	|	*Number*	Factor.cost ← **COST**(loadI)
	|	*Identifier*	Factor.cost ← **COST**(load)

An Extended Example (continued)

Adding attribution rules **All these attributes are synthesized!**

$Block_0$	→	*$Block_1$ Assign*	$Block_0$.cost ← $Block_1$.cost + Assign.cost
	|	*Assign*	$Block_0$.cost ← Assign.cost
Assign	→	*Ident = Expr ;*	Assign.cost ← **COST**(store) + Expr.cost
$Expr_0$	→	*$Expr_1$ + Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*$Expr_1$ - Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*Term*	$Expr_0$.cost ← Term.cost
$Term_0$	→	*$Term_1$ * Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(mult) + Factor.cost
	|	*$Term_1$ / Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(div) +Factor.cost
	|	*Factor*	$Term_0$.cost ← Factor.cost
Factor	→	*(Expr)*	Factor.cost ← Expr.cost
	|	*Number*	Factor.cost ← **COST**(loadI)
	|	*Identifier*	Factor.cost ← **COST**(load)

An Extended Example (continued)

Adding attribution rules

All these attributes are synthesized!

$Block_0$	→	*$Block_1$ Assign*	$Block_0$.cost ← $Block_1$.cost + Assign.cost
	|	*Assign*	$Block_0$.cost ← Assign.cost
Assign	→	*Ident = Expr ;*	Assign.cost ← **COST**(store) + Expr.cost
$Expr_0$	→	*$Expr_1$ + Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*$Expr_1$ - Term*	$Expr_0$.cost ← $Expr_1$.cost + **COST**(add) + Term.cost
	|	*Term*	$Expr_0$.cost ← Term.cost
$Term_0$	→	*$Term_1$ * Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(mult) + Factor.cost
	|	*$Term_1$ / Factor*	$Term_0$.cost ← $Term_1$.cost + **COST**(div) +Factor.cost
	|	*Factor*	$Term_0$.cost ← Factor.cost
Factor	→	*(Expr)*	Factor.cost ← Expr.cost
	|	*Number*	Factor.cost ← **COST**(loadI)
	|	*Identifier*	Factor.cost ← **COST**(load)

An Extended Example (continued)

Adding attribution rules

All these attributes are synthesized!

Production			Attribution rule
$Block_0$	→	$Block_1$ *Assign*	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
	|	*Assign*	$Block_0.cost \leftarrow Assign.cost$
Assign	→	*Ident* = *Expr* ;	$Assign.cost \leftarrow$ **COST**(store) + Expr.cost
$Expr_0$	→	$Expr_1$ + *Term*	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(add) + Term.cost
	|	$Expr_1$ - *Term*	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(add) + Term.cost
	|	*Term*	$Expr_0.cost \leftarrow Term.cost$
$Term_0$	→	$Term_1$ * *Factor*	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(mult) + Factor.cost
	|	$Term_1$ / *Factor*	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(div) +Factor.cost
	|	*Factor*	$Term_0.cost \leftarrow Factor.cost$
Factor	→	(*Expr*)	$Factor.cost \leftarrow Expr.cost$
	|	*Number*	$Factor.cost \leftarrow$ **COST**(loadI)
	|	*Identifier*	$Factor.cost \leftarrow$ **COST**(load)

An Extended Example

Properties of the example grammar

- All attributes are synthesized $\Rightarrow$ S-attributed grammar
- Rules can be evaluated bottom-up in a single pass
  - $\rightarrow$ Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement? (see backup slides)

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

Backup Slides

A Better Execution Model

Adding load tracking

- Need sets ***Before* and *After*** for each production
- Must be initialized, updated, and passed around the tree

Factor → (Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After
| Number	Factor.cost ← COST(loadi) ; Factor.After ← Factor.Before

| Identifier	If (Identifier.name ∉ Factor.Before)
	then
	Factor.cost ← COST(load);
	Factor.After ← Factor.Before ∪ Identifier.name
	else
	Factor.cost ← 0
	Factor.After ← Factor.Before

This looks more complex!

A Better Execution Model

Adding load tracking

- Need sets ***Before* and *After*** for each production
- Must be initialized, updated, and passed around the tree

Factor	→ (Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After
	| Number	Factor.cost ← COST(loadi) ; Factor.After ← Factor.Before

	| Identifier	If (Identifier.name ∉ Factor.Before)
		then
		Factor.cost ← COST(load);
		Factor.After ← Factor.Before ∪ Identifier.name
		else
		Factor.cost ← 0
		Factor.After ← Factor.Before

This looks more complex!

A Better Execution Model

- Load tracking adds complexity
- Every production needs rules to copy *Before* & *After*

A sample production

$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost + COST(add) + Term.cost$; $Expr_1.Before \leftarrow Expr_0.Before$; $Term.Before \leftarrow Expr_1.After$; $Expr_0.After \leftarrow Term.After$

Lots of work, lots of space, lots of rules to write

An Even Better Model

What about accounting for finite register sets?

- *Before* & *After* must be of limited size
- Adds complexity to *Factor→Identifier*
- Requires more complex initialization

Jump from tracking loads to tracking registers is small

- Copy rules are already in place
- Some local code to perform the allocation