Context-sensitive Analysis
or
Semantic Elaboration

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(int a, int b,int c,int d) {
    …
}
fee() {
    int f[3],g[0], h, i, j, k;
    char *p;
    fie(h,i,“ab”,j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”,p,q);
    p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

- number of args to fie()
- declared g[0], used g[17]
- “ab” is not an <u>int</u>
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are
“deeper than syntax”

To generate code, we need to understand its meaning !

Beyond Syntax

To generate code, the compiler needs to answer many questions

- Is “x” a scalar, an array, or a function? Is “x” declared?
- Are there names that are not declared? Declared but not used?
- Which declaration of “x” does a given use reference?
- Is the expression “x * y + z” type-consistent?
- In “a[i,j,k]”, does a have three dimensions?
- Where can “z” be stored? (register, local, global, heap, static)
- In “f ← 15”, how should 15 be represented?
- How many arguments does “fie()” take? What about “printf ()” ?
- Does “*p” reference the result of a “malloc()” ?
- Do “p” & “q” refer to the same memory location?
- Is “x” defined before it is used?

These are beyond the expressive power of a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on values, not parts of speech
- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars?
  - Attribute grammars
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code (action routines)

In context-sensitive analysis, ad-hoc techniques dominate practice.

Beyond Syntax

Telling the story

- We will study the formalism — an attribute grammar
  - Clarify many issues in a succinct and immediate way
  - Separate analysis problems from their implementations

- We will see that the problems with attribute grammars motivate actual, ad-hoc practice
  - Non-local computation
  - Need for centralised information

We will cover attribute grammars, then move on to ad-hoc ideas

Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of rules
- Each symbol in the derivation (or parse tree) has a set of named values, or attributes
- The rules specify how to compute a value for each attribute
  - Attribution rules are functional; they uniquely define the value

Example grammar

1	*Number*	$\rightarrow$	*Sign List*
2	*Sign*	$\rightarrow$	+
3		|	-
4	*List*	$\rightarrow$	*List Bit*
5		|	*Bit*
6	*Bit*	$\rightarrow$	0
7		|	1

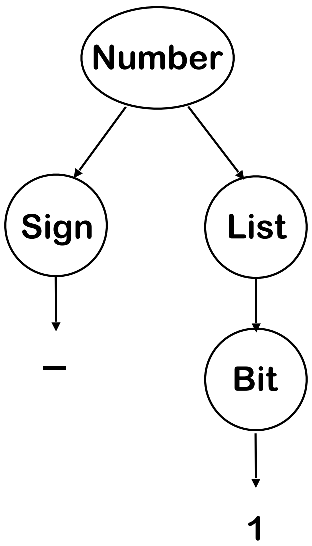
This grammar describes signed binary numbers

We would like to augment it with rules that compute the decimal value of each valid input string

Examples

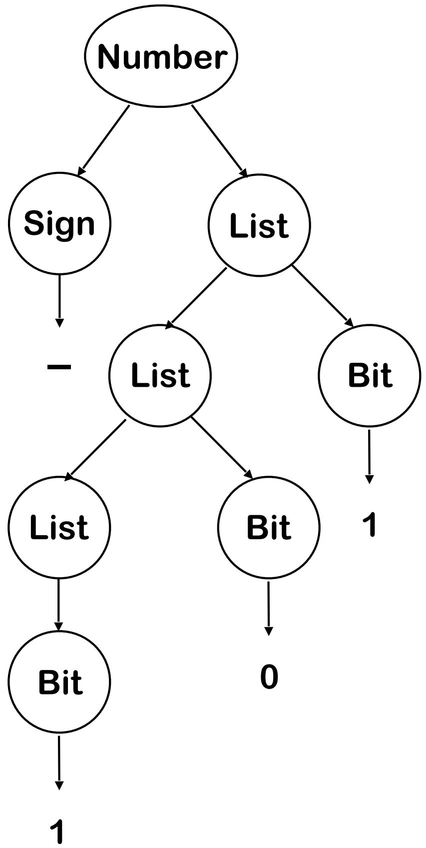
For "–1"

Number → Sign List
→ Sign Bit
→ Sign 1
→ – 1



For "–101"

Number → Sign List
→ Sign List Bit
→ Sign List 1
→ Sign List Bit 1
→ Sign List 0 1
→ Sign Bit 0 1
→ Sign 1 0 1
→ – 101



We will use these two examples throughout the lecture

Attribute Grammars

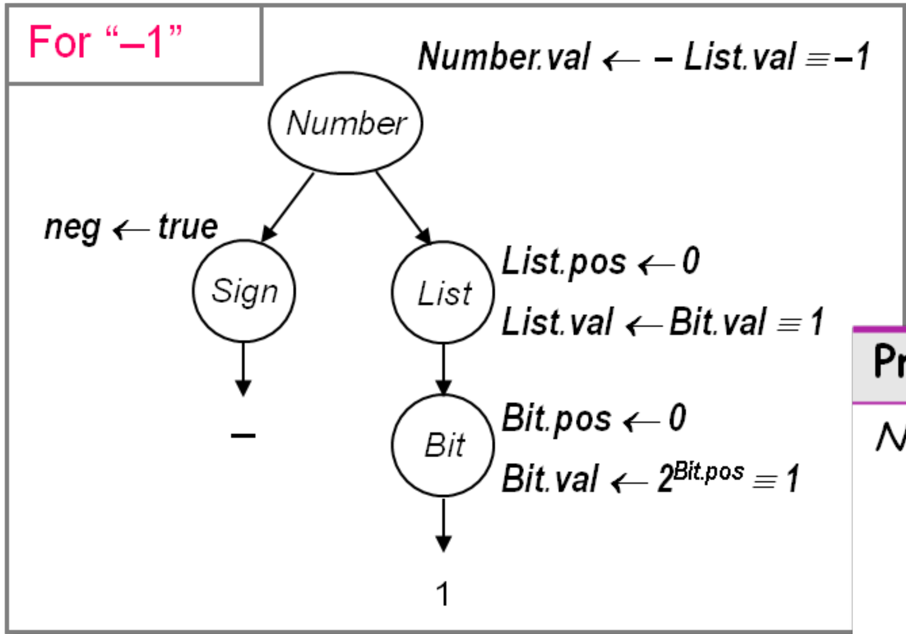
Add rules to compute the decimal value of a signed binary number

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

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

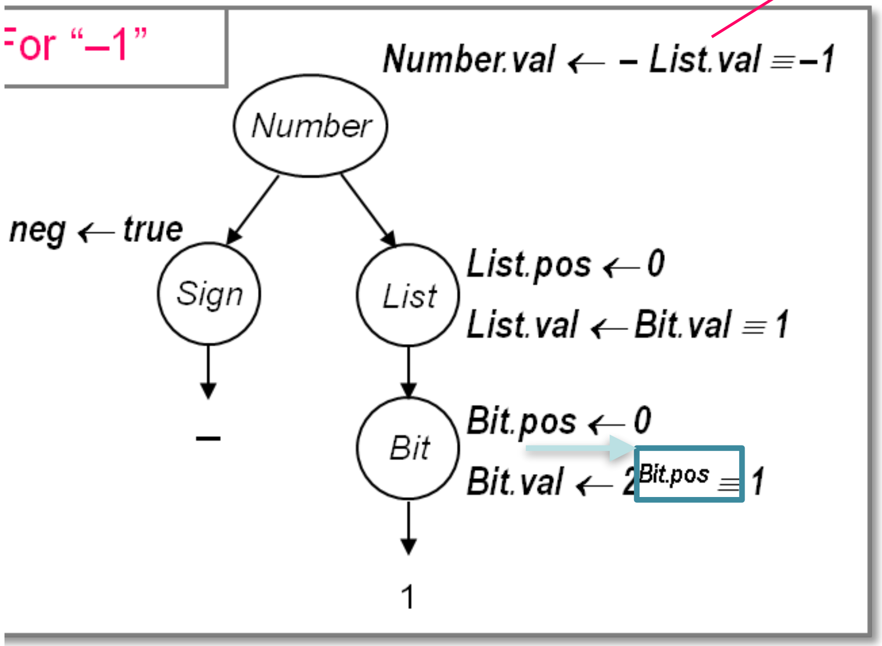
Symbol	Attributes
Number	val
Sign	neg
List	pos, val
Bit	pos, val



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	$Bit.val \leftarrow 2^{Bit.pos}$

Evaluation order

Rules + parse tree imply an 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

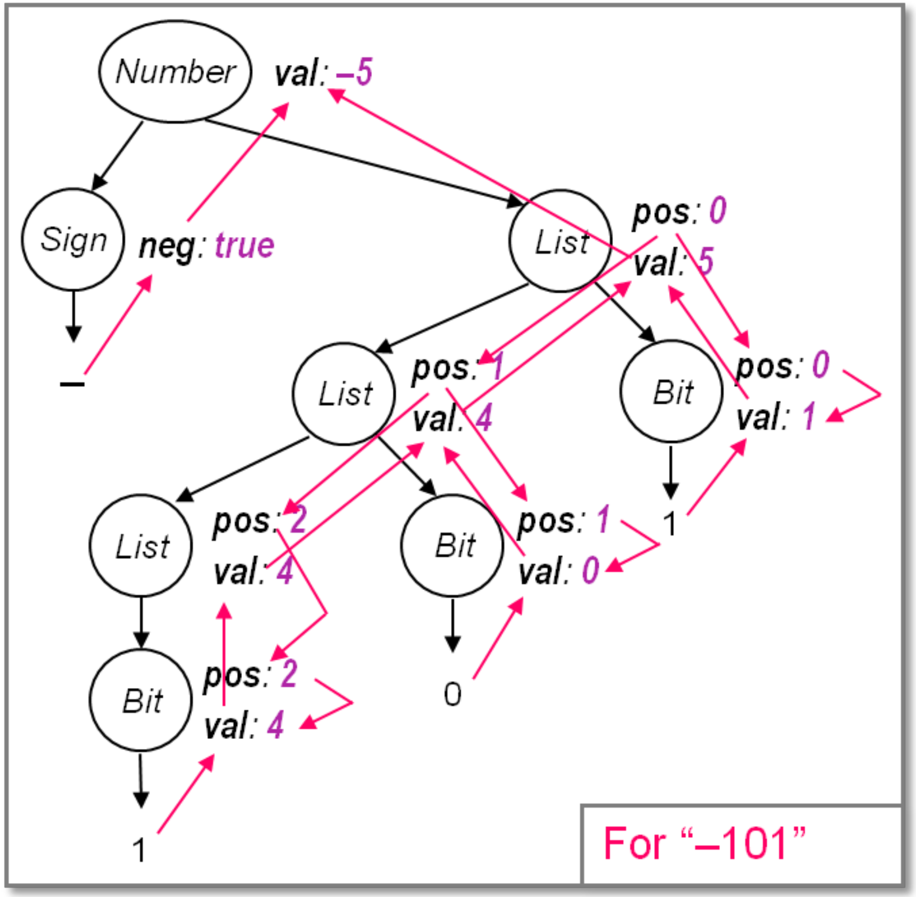
Other orders are possible

Knuth suggested a data-flow model for evaluation

- Independent attributes first
- Others in order as input values become available

Evaluation order must be consistent with the attribute dependence graph

Back to the Examples



This is the complete attribute dependence graph for "–101".

It shows the flow of all attribute values in the example.

Some flow downward

→ inherited attributes

Some flow upward

→ synthesized attributes

A rule may use attributes in the parent, children, or siblings of a node

The Rules of the Game

- Attributes associated with nodes in parse tree
- Rules are value assignments associated with productions
- Attribute is defined once, using local information
- Label identical terms in production for uniqueness
- Rules & parse tree define an attribute dependence graph
  - Graph must be non-circular

This produces a high-level, functional specification

N.B.: *AG is a specification for the computation, not an algorithm*

Synthesized attribute
- Depends on values from children

Inherited attribute
- Depends on values from siblings & parent

Using Attribute Grammars

Attribute grammars can specify context-sensitive actions

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

Synthesized Attributes

- Use values from children & from constants
- S-attributed grammars
- Evaluate in a single bottom-up pass

Good match to LR parsing

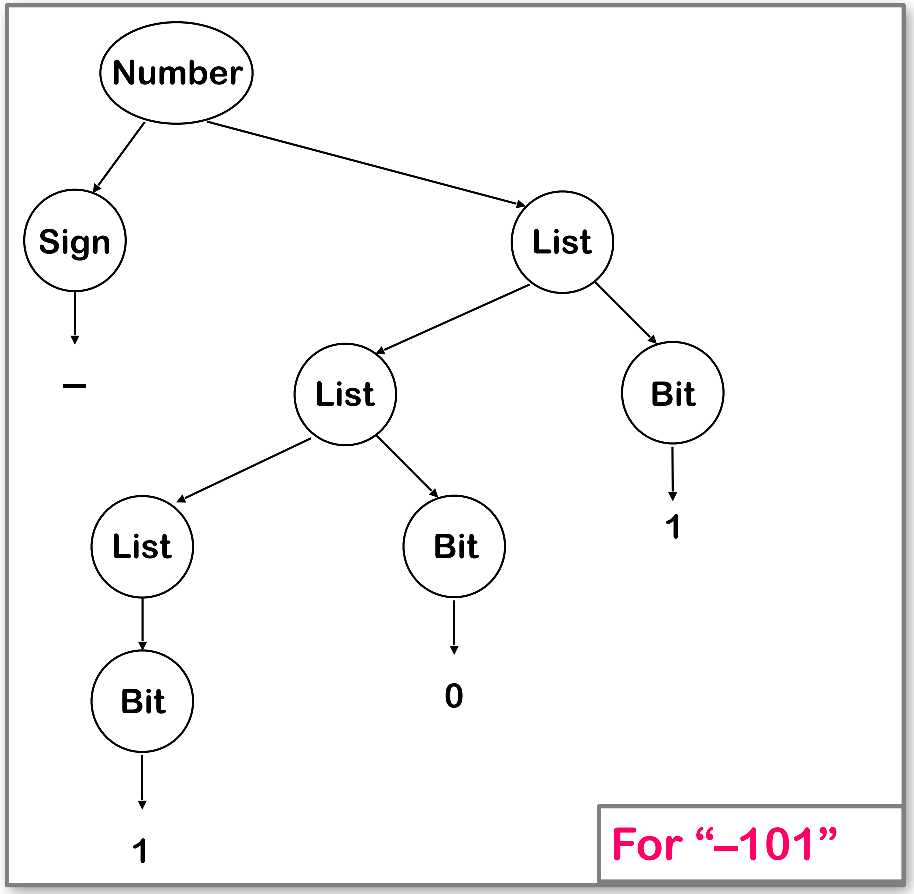
Inherited Attributes

- Use values from parent, constants, & siblings
- Directly express context
- Can rewrite to avoid them
- Thought to be more natural

Not easily done at parse time

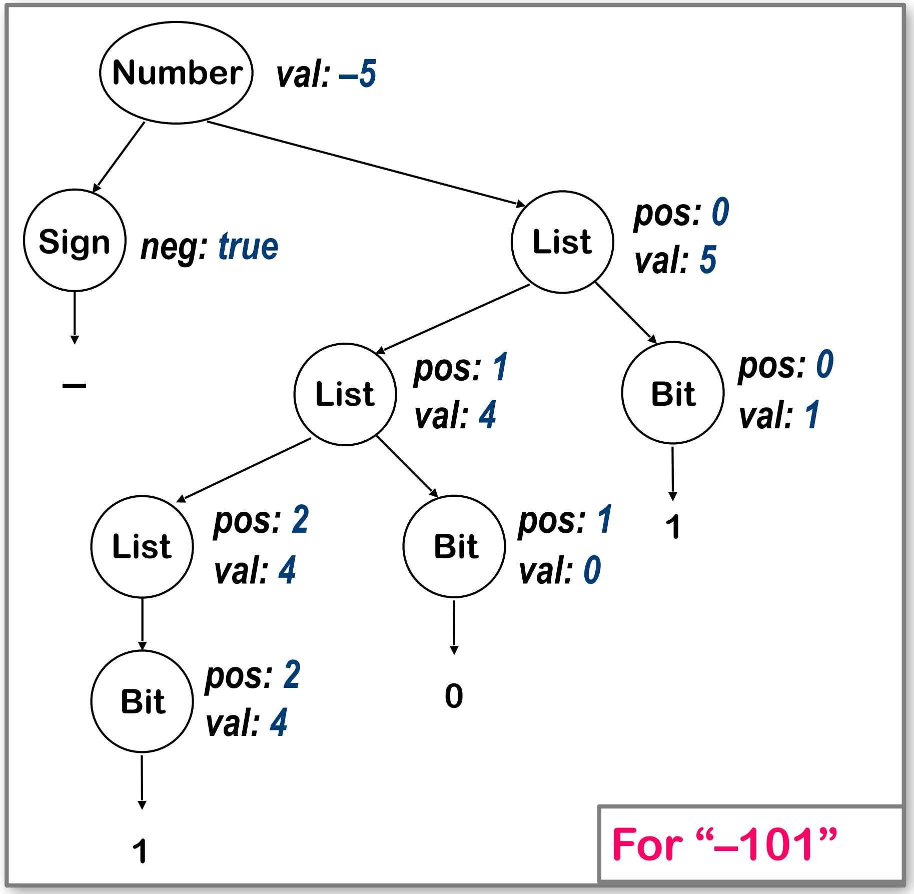
We want to use both kinds of attributes

Back to the Example



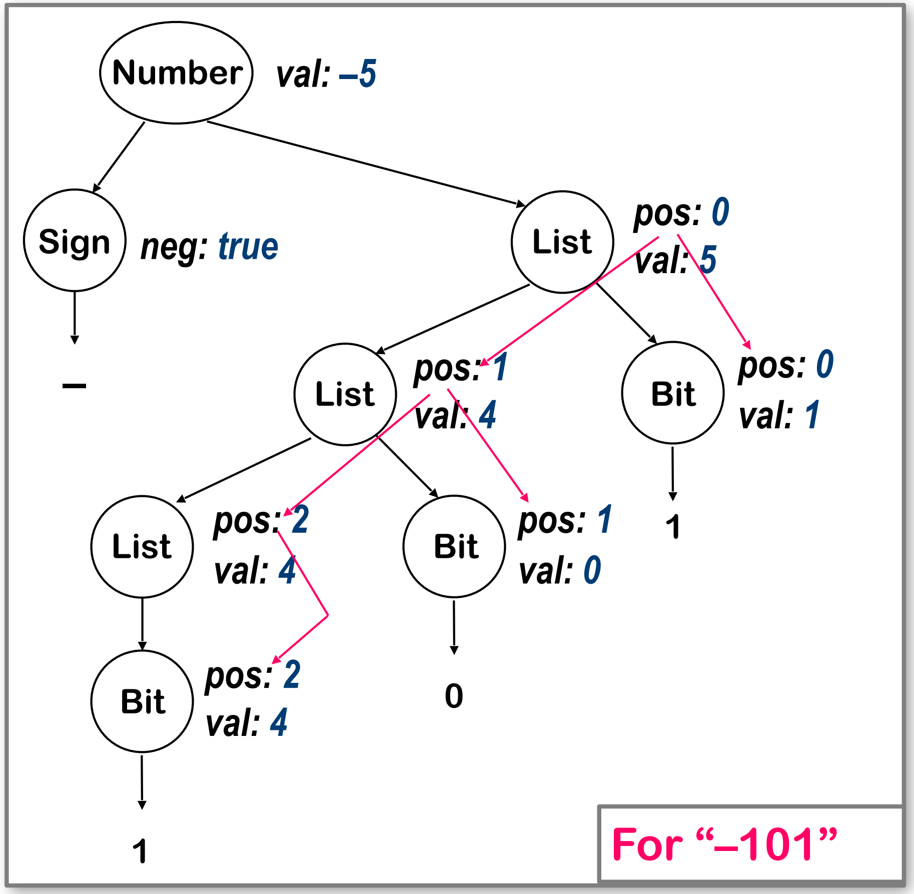
Syntax Tree

Back to the Example



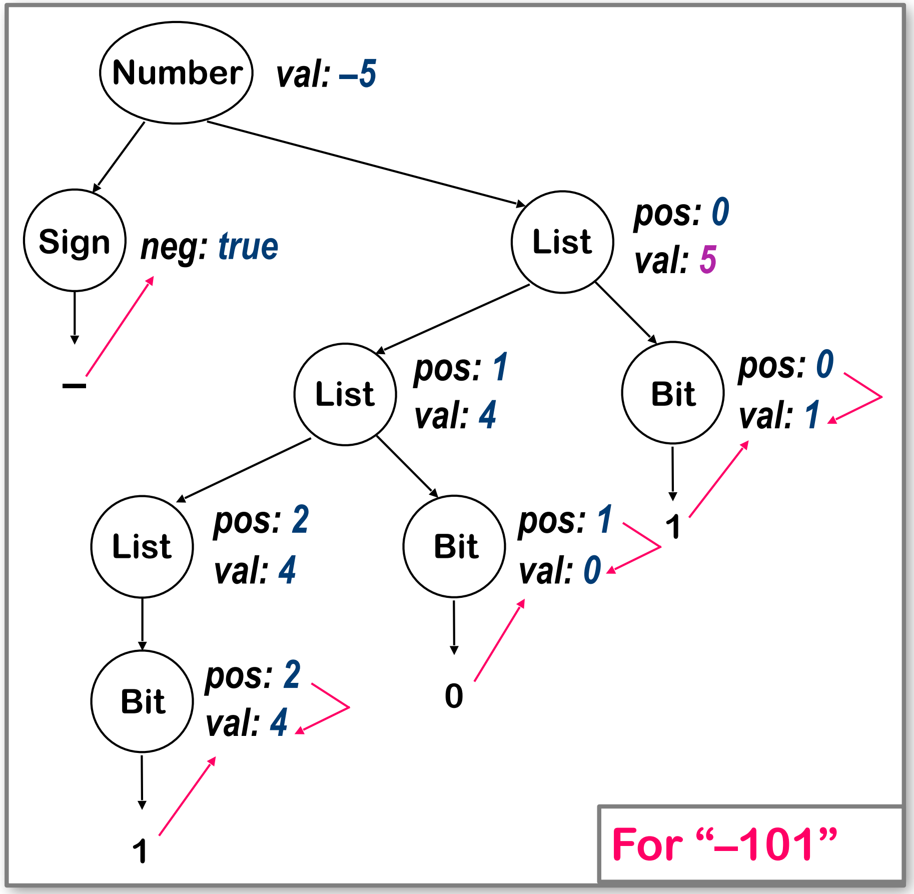
Attributed Syntax Tree

Back to the Example



Inherited Attributes

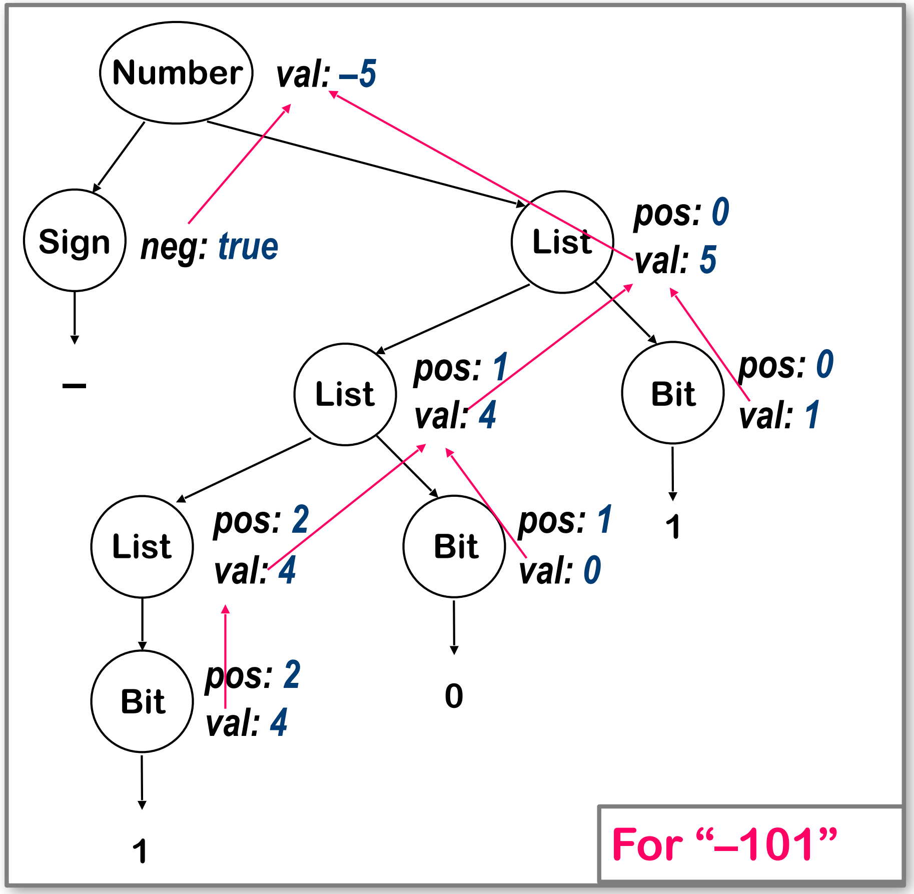
Back to the Example



Synthesized attributes

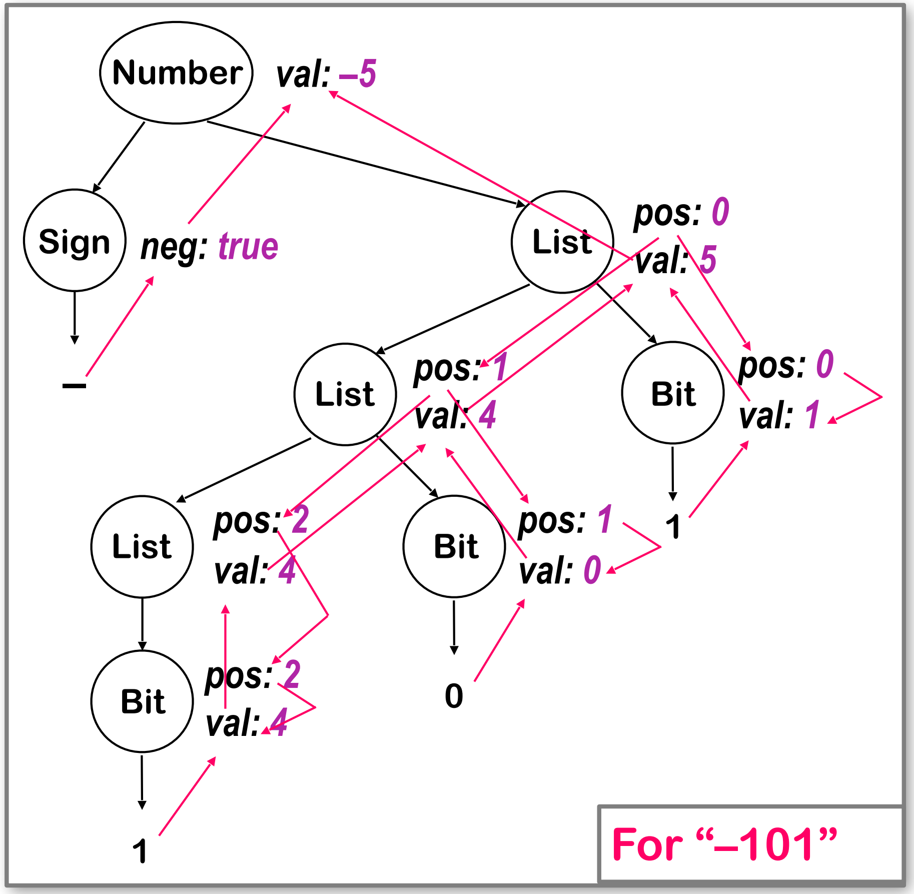
Val draws from children & the same node.

Back to the Example



More Synthesized attributes

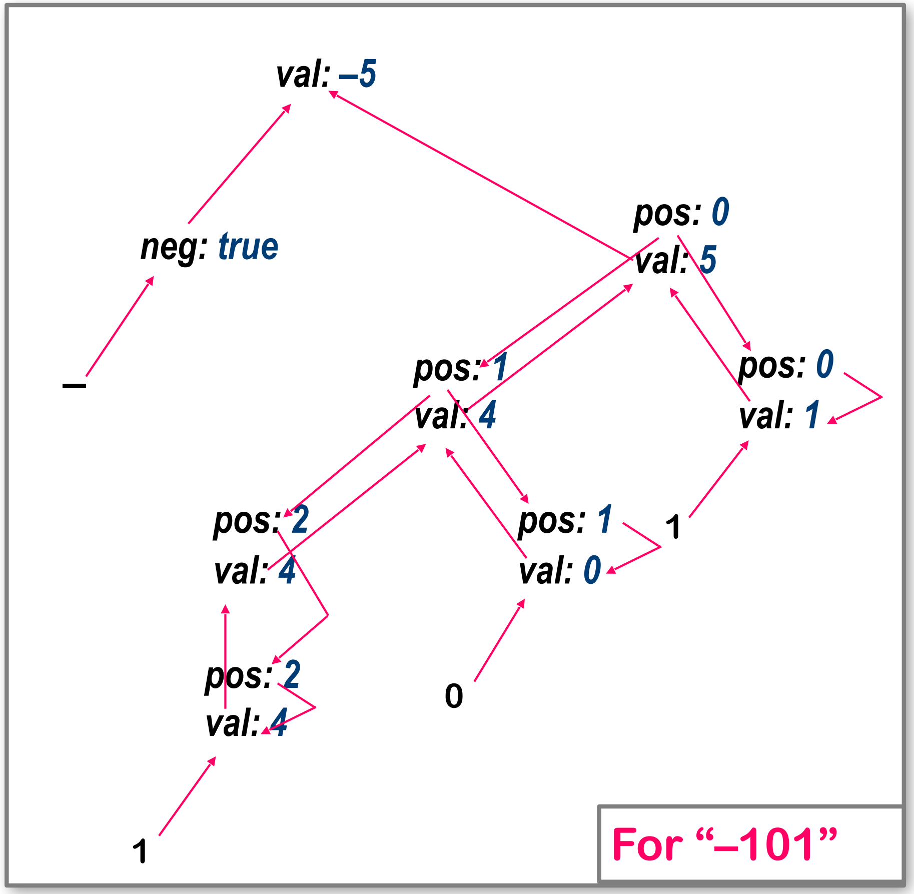
Back to the Example



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 dynamic methods sort this graph to find independent values, then work along graph edges.

The rule-based methods try to discover "good" orders by analyzing the rules.

The oblivious methods ignore the structure of this graph.

The dependence graph must be acyclic