Context-sensitive Analysis

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
    int a, b, c, 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 …)

Beyond Syntax

There is a level of correctness that is deeper than grammar

```
fie(a,b,c,d)
    int a, b, c, 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 …)

- **declared g[0], used g[17]**
- **wrong number of args to fie()**
- **“ab” is not an int**
- **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 each 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)*
- 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 a CFG

Beyond Syntax

These questions are part of context-sensitive analysis

- Questions & answers involve non-local information
- Answers may involve computation

How can we answer these questions?

- Use formal methods
  - → Attribute grammars?
    - Also known as attributed CFG or syntax-directed definitions
- Use *ad-hoc* techniques
  - → Symbol tables
  - → *Ad-hoc* code **(action routines)**

In scanning & parsing, formalism won; different story here.

Beyond Syntax

Telling the story

- The **attribute grammar** formalism is important
  - → Succinctly makes many points clear
  - → Sets the stage for actual, ***ad-hoc*** practice
- The problems with **attribute grammars** motivate practice
  - → Non-local computation
  - → Need for centralized information
- Some folks still argue for **attribute grammars**
  - → In practice, **ad-hoc** techniques used

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 has a set of values, or *attributes*
  - → *X.a denotes the value of* ***a*** *at a particular parse-tree node labeled X*
- The rules specify how to compute a value for each attribute

Example grammar

Number	→	Sign List
Sign	→	+
	|	-
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

This grammar describes signed binary numbers: +101, -11, +10101, but not 101

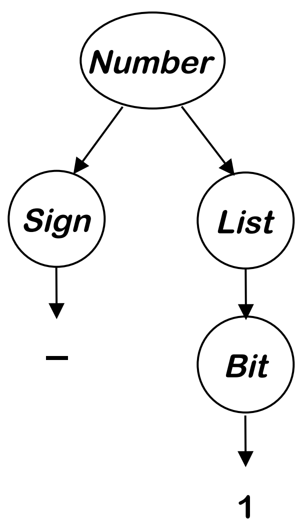
We would like to augment it with rules that compute the decimal value of each valid input string

Example: parse -101 and compute -5

Examples

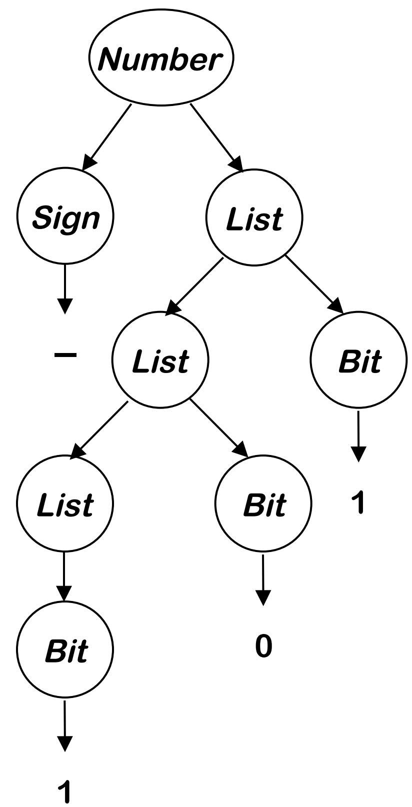
For "–1"

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



For "–101"

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



We will use these two throughout the lecture

Attribute Grammars

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

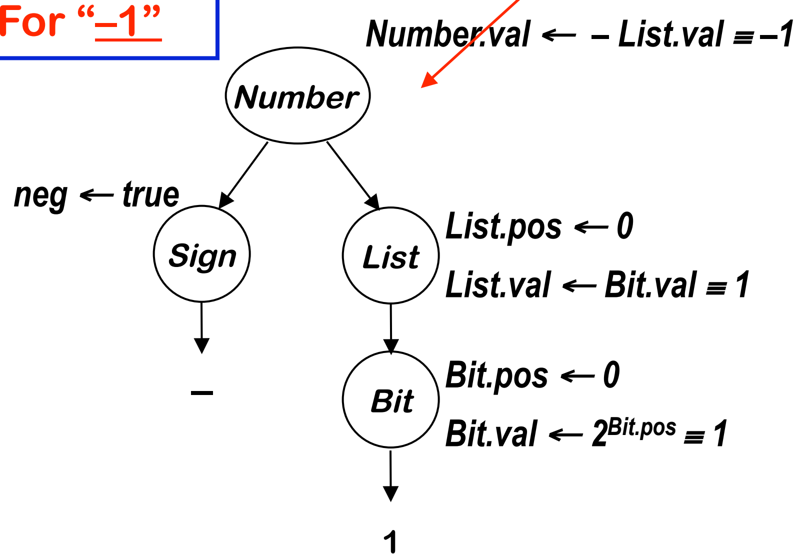
Productions			Attribution Rules
Number	→	*Sign List*	*List.pos* ← *0* *If Sign.neg* *then Number.val* ← – *List.val* *else Number.val* ← *List.val*
Sign	→	+	*Sign.neg* ← *false*
	|	–	*Sign.neg* ← *true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val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

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

Back to the Examples

Rules + parse tree imply an attribute dependence graph

For “–1”



One possible evaluation order:

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

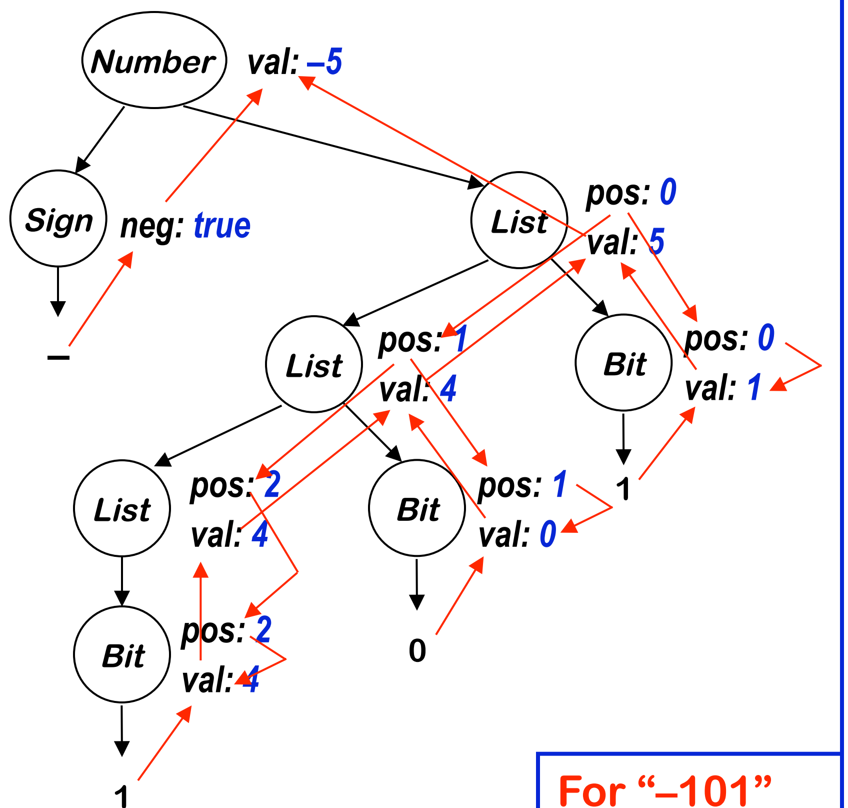
Other orders are possible

Evaluation order must be consistent with the attribute dependence graph

Knuth suggested a data-flow model for evaluation

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

Back to the Examples



For "–101"

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

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 attribute

Evaluation Methods

Dynamic, dependence-based methods

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

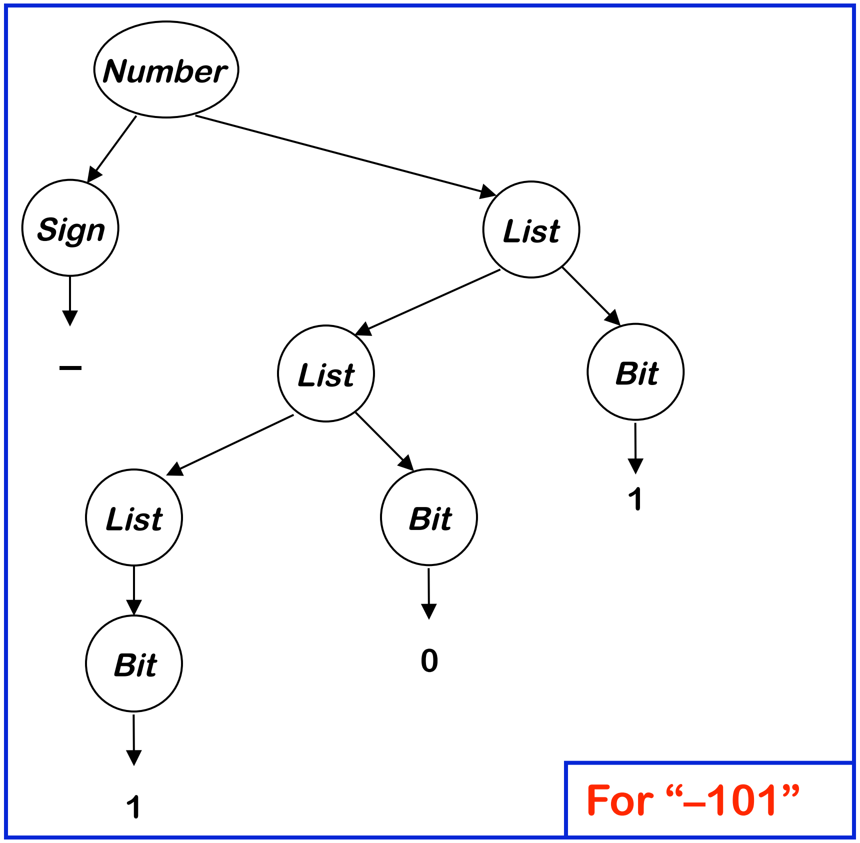
Rule-based methods *(treewalk)*

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

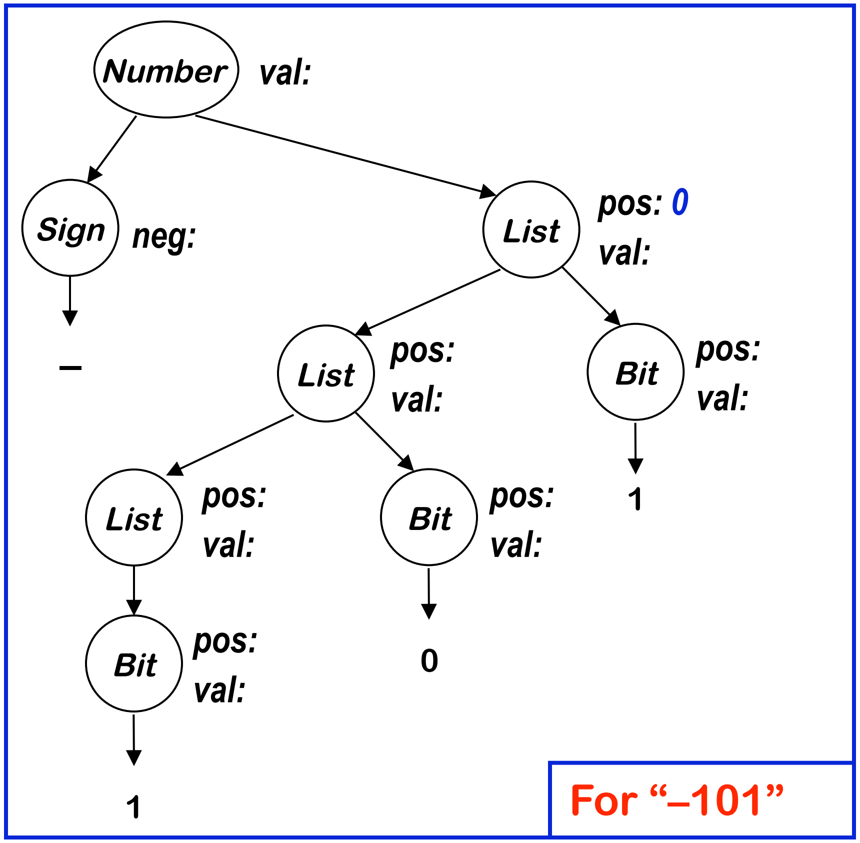
Oblivious methods *(passes, dataflow)*

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

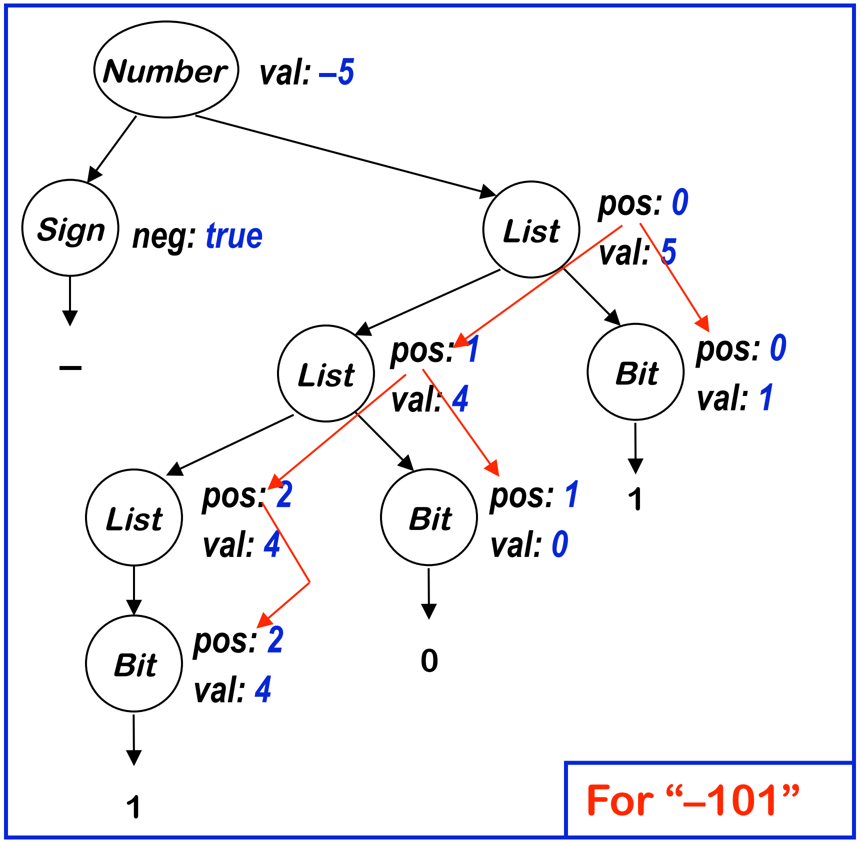
Back to the Example



Back to the Example

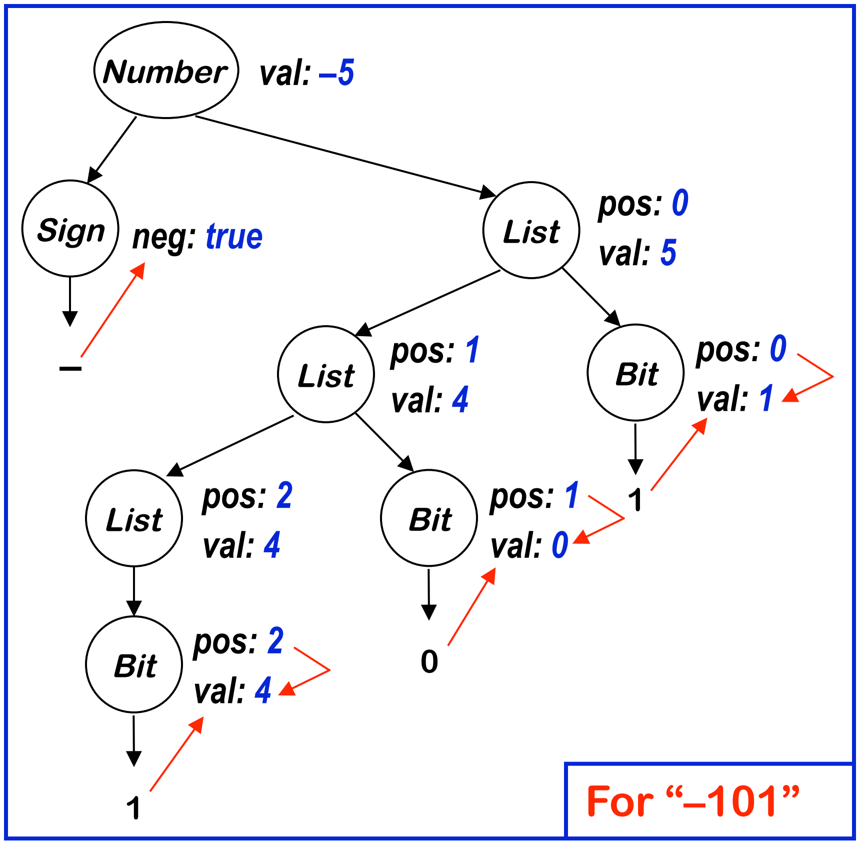


Back to the Example



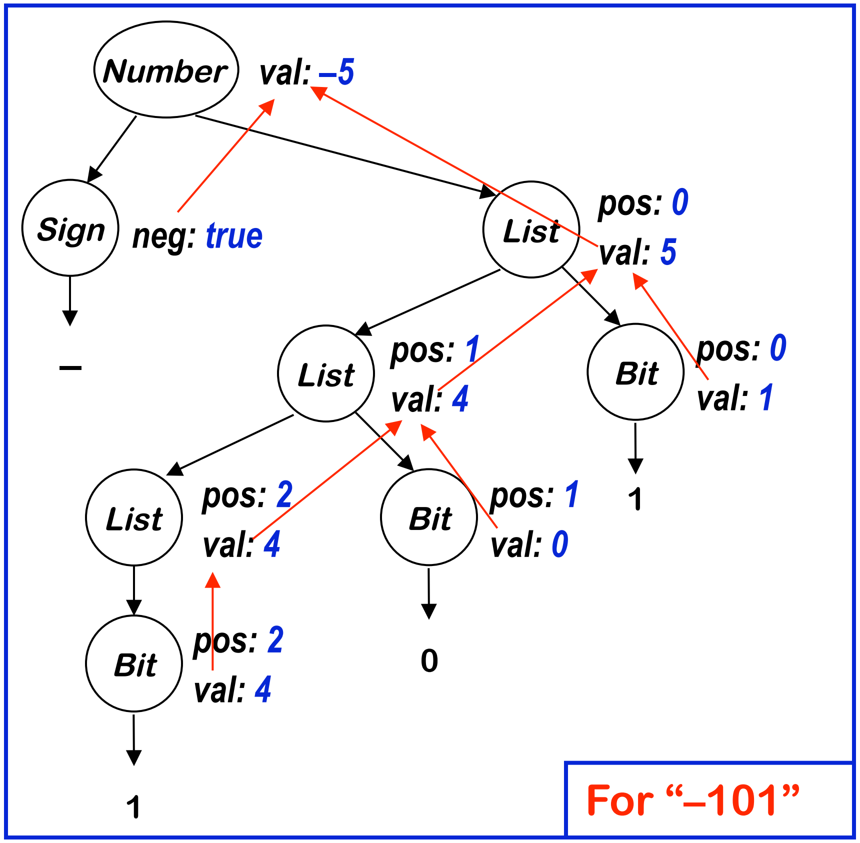
Inherited Attributes

Back to the Example



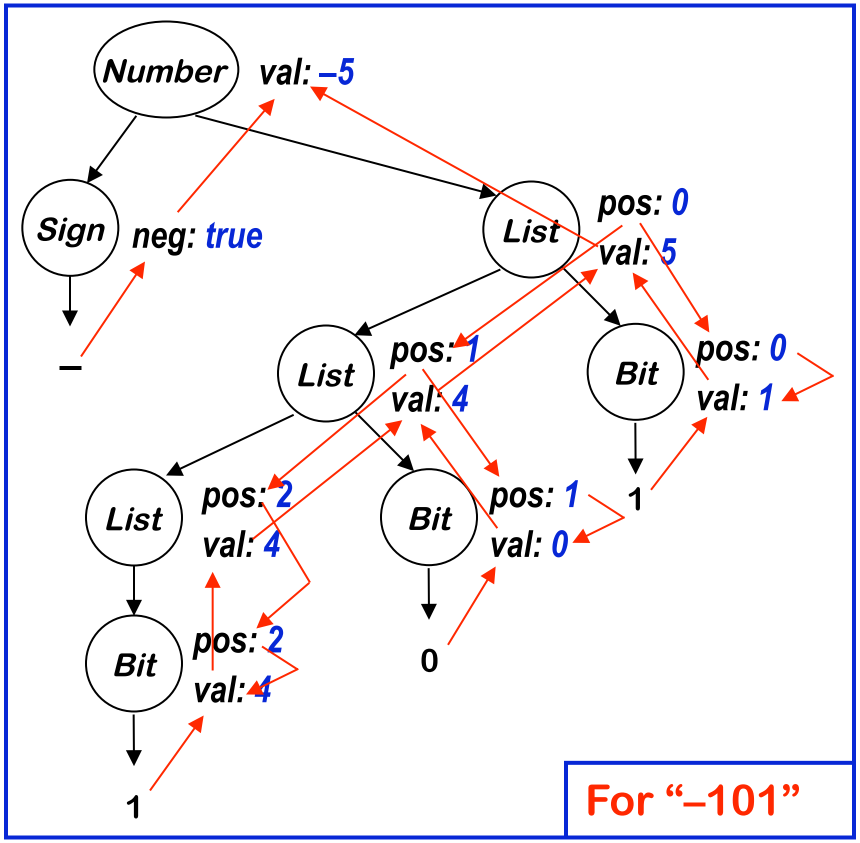
Synthesized attributes

Back to the Example



Synthesized attributes

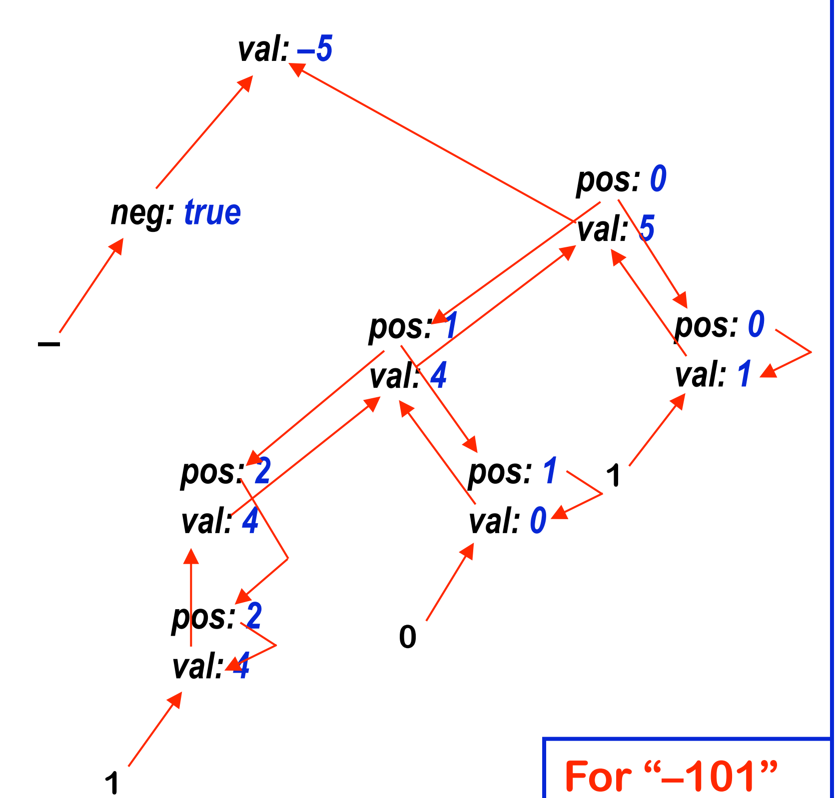
Back to the Example



If we show the computation ...

& then peel away the parse tree ...

Back to the Example



For "–101"

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

Circularity

We can only evaluate acyclic instances

- We can prove that some grammars can only generate instances with acyclic dependence graphs
- Largest such class is "strongly non-circular" grammars (*SNC*)
- *SNC* grammars can be tested in polynomial time
- Failing the *SNC* test is <u>not</u> conclusive

Many evaluation methods discover circularity dynamically

$\Rightarrow$ Bad property for a compiler to have

SNC grammars were first defined by Kennedy & Warren

An Extended Example

Grammar for a basic block *(§ 4.3.3)*

$Block_0$	$\rightarrow$	*$Block_1$ 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*

Let's estimate cycle counts

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

Simple problem for an *AG*

An Extended Example (continued)

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

$Block_0$	$\rightarrow$	$Block_1$ $Assign$	$Block_0.cost \leftarrow Block_1.cost + Assign.cost$
	$\mid$	$Assign$	$Block_0.cost \leftarrow Assign.cost$
$Assign$	$\rightarrow$	$Ident = Expr$;	$Assign.cost \leftarrow$ **COST**(store) + Expr.cost
$Expr_0$	$\rightarrow$	$Expr_1 + Term$	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(add) + Term.cost
	$\mid$	$Expr_1 - Term$	$Expr_0.cost \leftarrow Expr_1.cost +$ **COST**(add) + Term.cost
	$\mid$	$Term$	$Expr_0.cost \leftarrow Term.cost$
$Term_0$	$\rightarrow$	$Term_1 * Factor$	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(mult) + Factor.cost
	$\mid$	$Term_1 / Factor$	$Term_0.cost \leftarrow Term_1.cost +$ **COST**(div) +Factor.cost
	$\mid$	$Factor$	$Term_0.cost \leftarrow Factor.cost$
$Factor$	$\rightarrow$	($Expr$)	$Factor.cost \leftarrow Expr.cost$
	$\mid$	$Number$	$Factor.cost \leftarrow$ **COST**(loadI)
	$\mid$	$Identifier$	$Factor.cost \leftarrow$ **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
  - → Good fit to bottom-up, shift/reduce parser
- Easily understood solution
- Seems to fit the problem well

What about an improvement?

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

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
- But, most of it is in the "copy rules"
- Every production needs rules to copy *Before* & *After*

A sample production

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

These copy rules multiply rapidly

Each creates an instance of the set

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

Next class

⇒ Curing these problems with *ad-hoc* syntax-directed translation