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?

Beyond Syntax

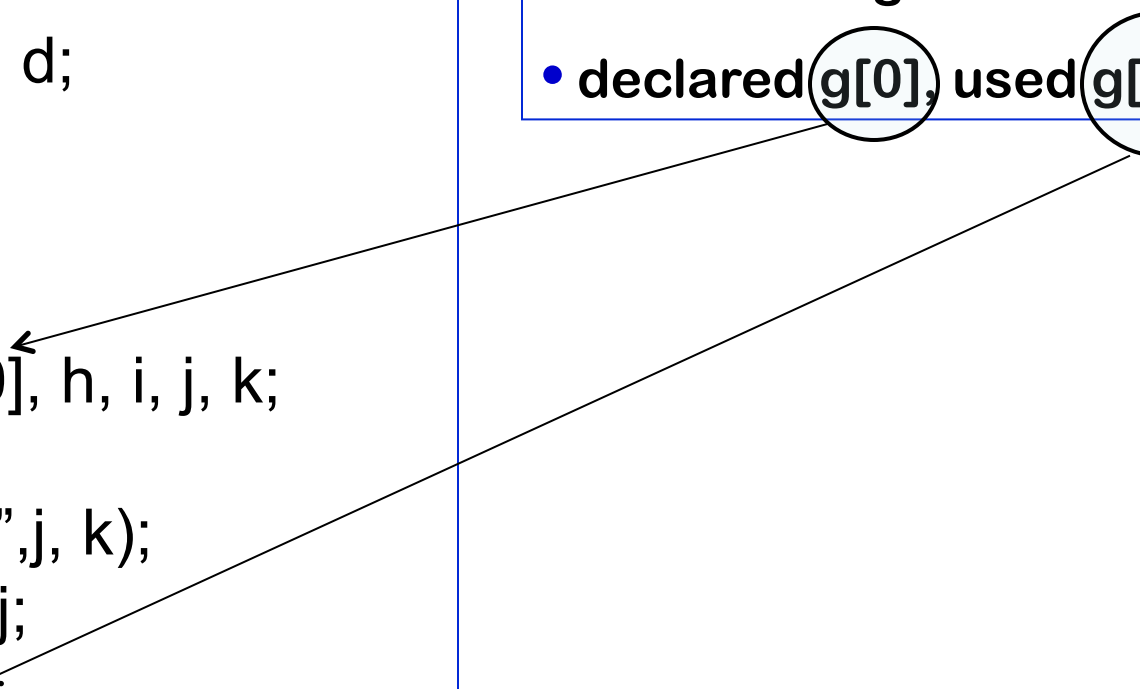
There is a level of correctness that is deeper than grammar

```
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?

- **declared g[0], used g[17]**



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?

- **declared g[0], used g[17]**
- **wrong number of args to fie()**

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?

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

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?

- **declared g[0], used g[17]**
- **wrong number of args to fie()**
- **“ab” is not an int**
- **wrong dimension on use of f**

Beyond Syntax

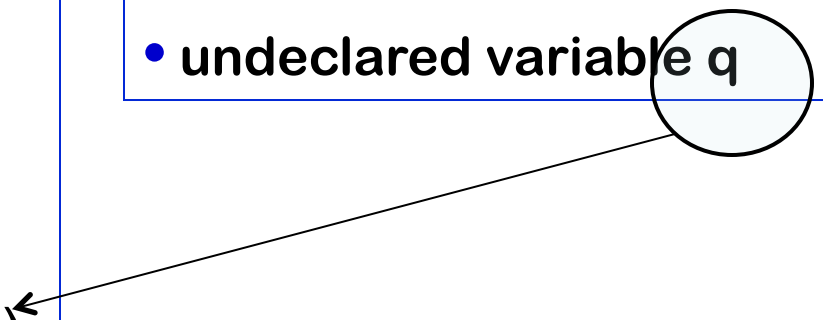
There is a level of correctness that is deeper than grammar

```
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?

- **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**



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?

- **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**

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?

- **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”

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?

These are beyond a context-free grammar

Beyond Syntax

To generate code, the compiler needs to answer many questions

- In "*a*[i,j,k]", does *a* have three dimensions?
- Where can "z" be stored? *(register, local, heap, etc.)*
- How many arguments does "fie()" take?
- Does "*p" reference the result of a "malloc()" ?
- Do "p" & "q" refer to same memory location?
- Is "x" defined before it is used?

These are beyond a context-free grammar

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

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

What is an Attribute Grammar?

- Context-free grammar augmented with 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*
- Rules specify how to compute a value for each attribute

What is an Attribute Grammar?

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

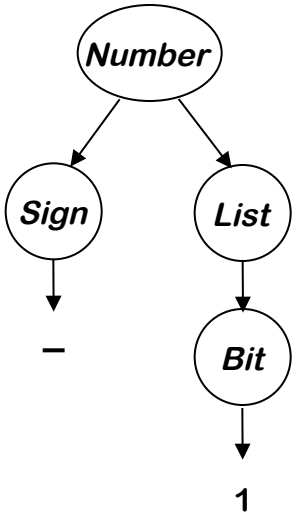
We would like to augment it with rules that compute the decimal value of each valid input string

Example: parse -101 and compute -5

Examples

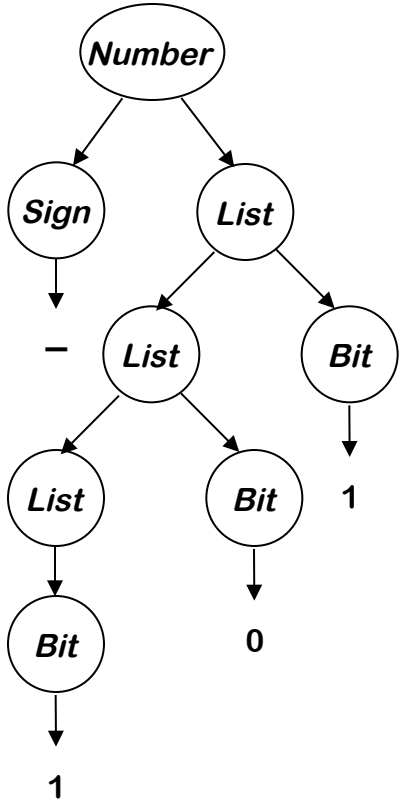
For "–1"

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

Grammar and its Attributes

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

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

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

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

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

LHS to the RHS

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

RHS to the LHS

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

Subscripts needed

Attribute Grammars

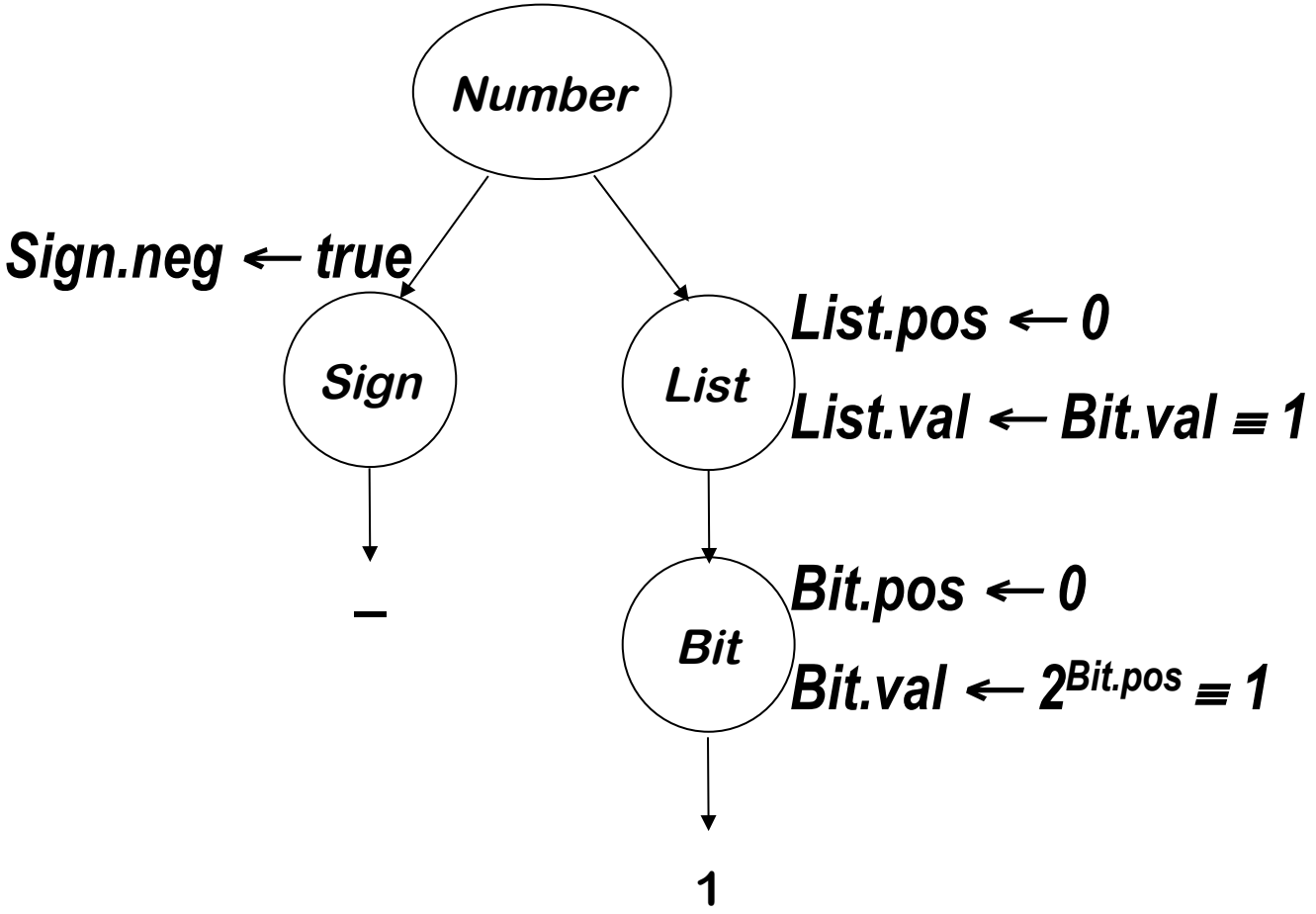
Add rules to compute the decimal value of a signed binary number

Subscripts needed

Bit	→	0	*Bit.val* ← *0*
	|	1	*Bit.val* ← $2^{Bit.pos}$

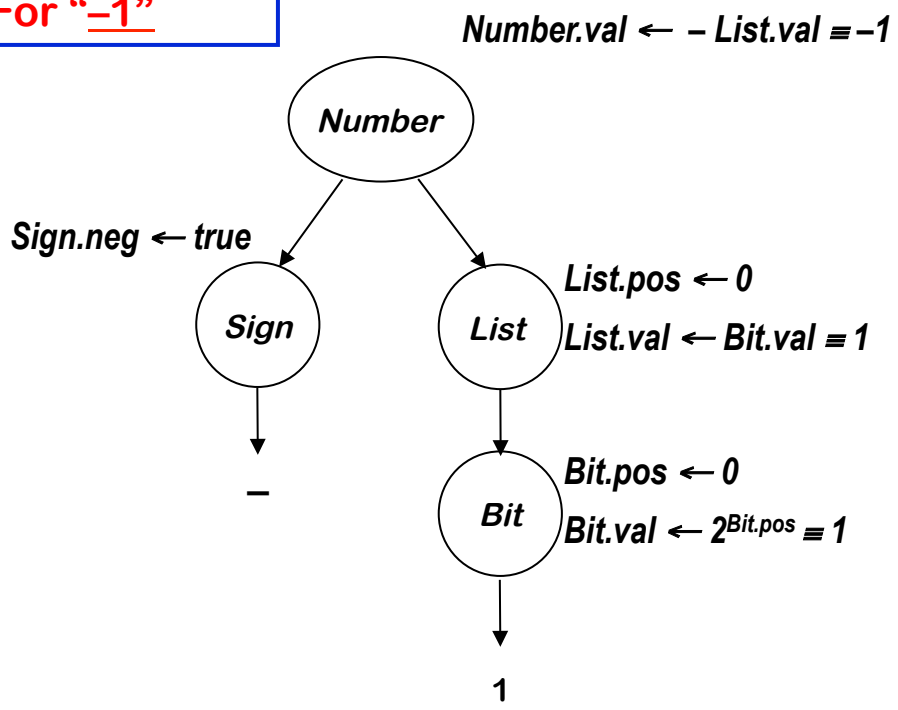
Back to the Examples

For "–1"



Back to the Examples

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

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
  - → Graph must be non-circular

This produces a high-level, functional specification

Two kinds of Attributes

- Synthesized attribute
  - → Upward flow of values
  - → Depends on values from children
- Inherited attribute
  - → Downward flow of values
  - → 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, …

Evaluation Methods

Dynamic, dependence-based methods

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

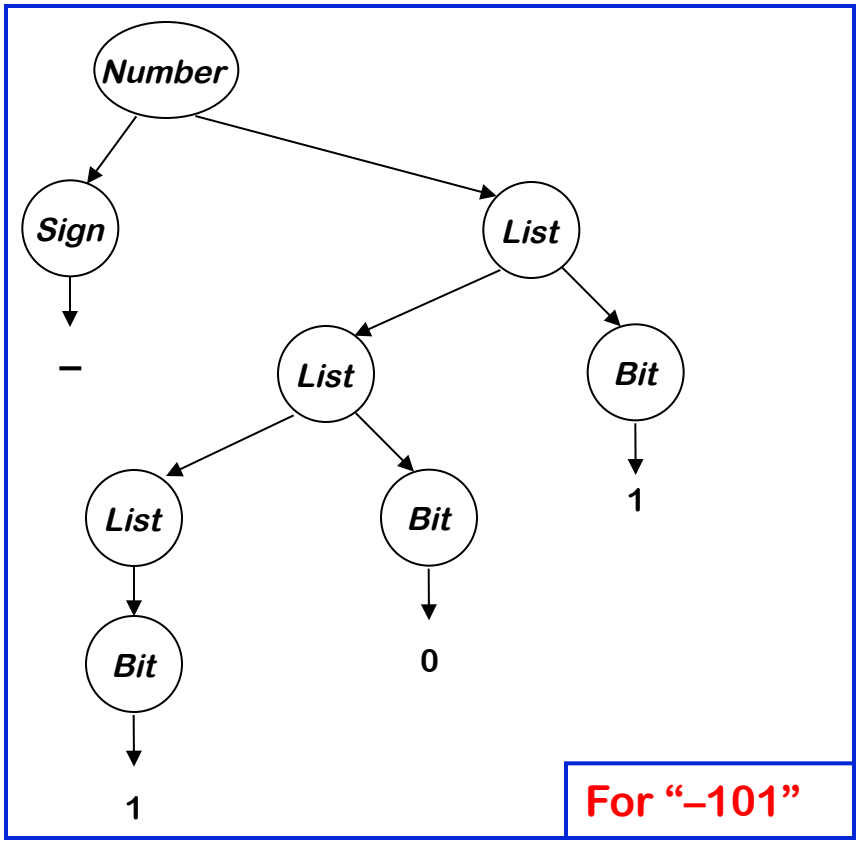
Rule-based methods *(treewalk)*

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

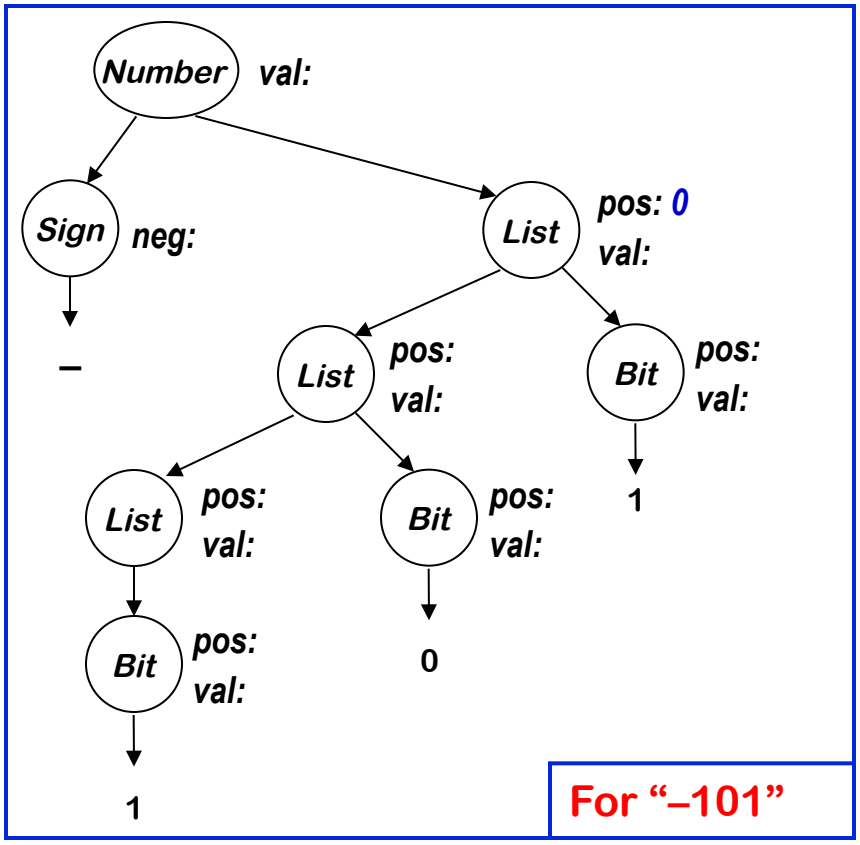
Oblivious methods *(passes, dataflow)*

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

Back to the Example

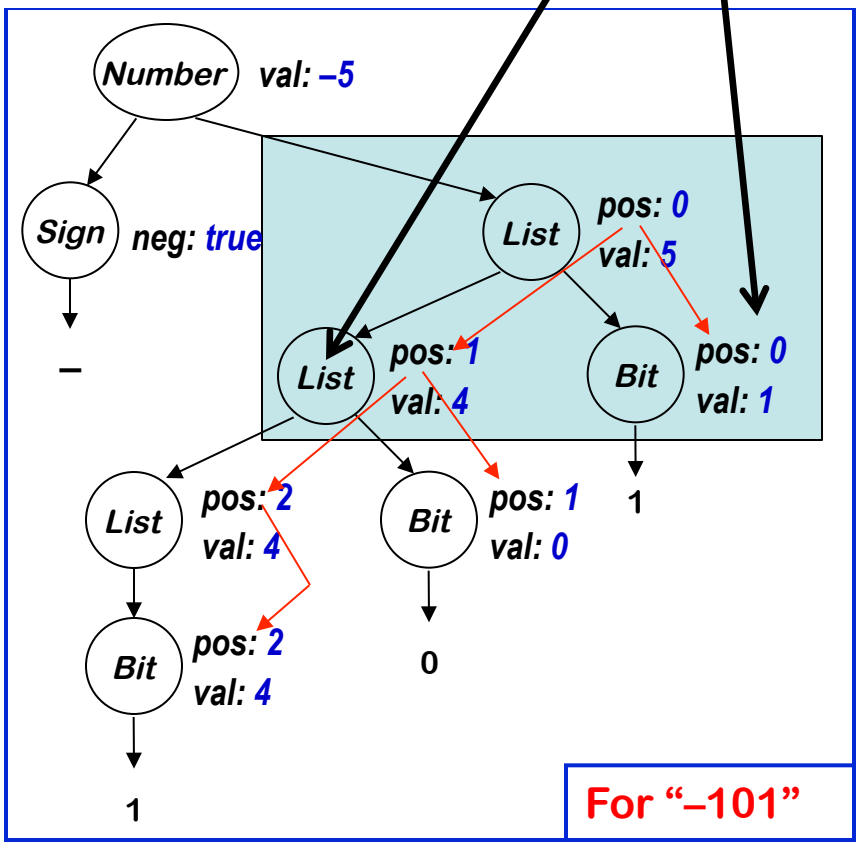


Back to the Example



Back to the Example

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

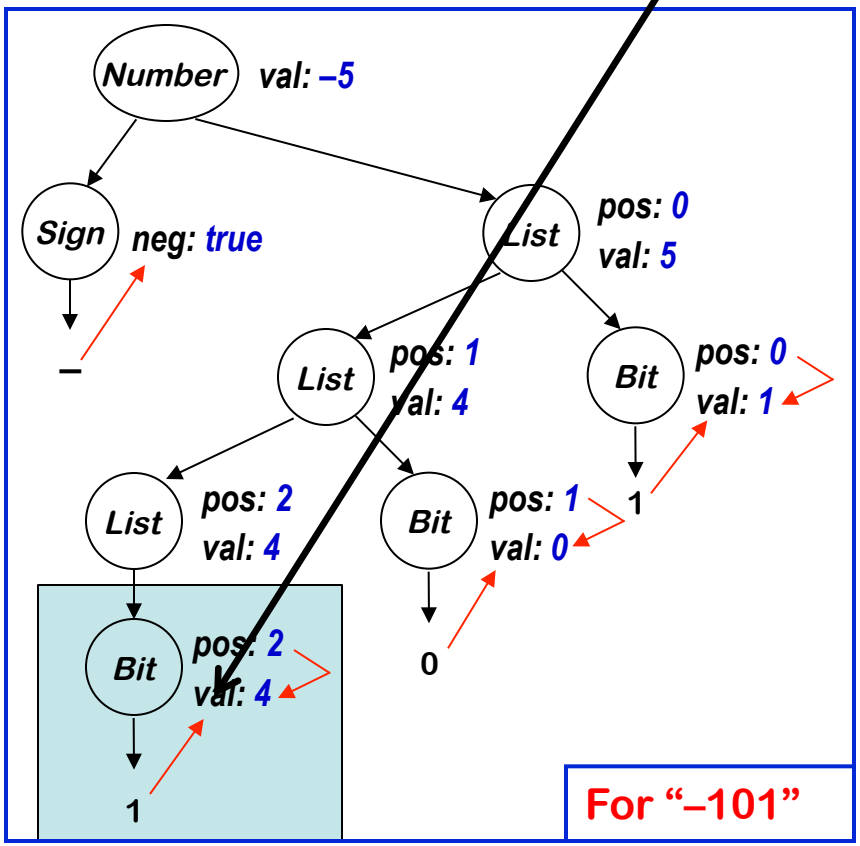


Inherited Attributes

Note: the downward (pointing arrows) flow of information

Back to the Example

Bit	→	0	*Bit.val* ← *0*
	|	1	*Bit.val* ← $2^{Bit.pos}$

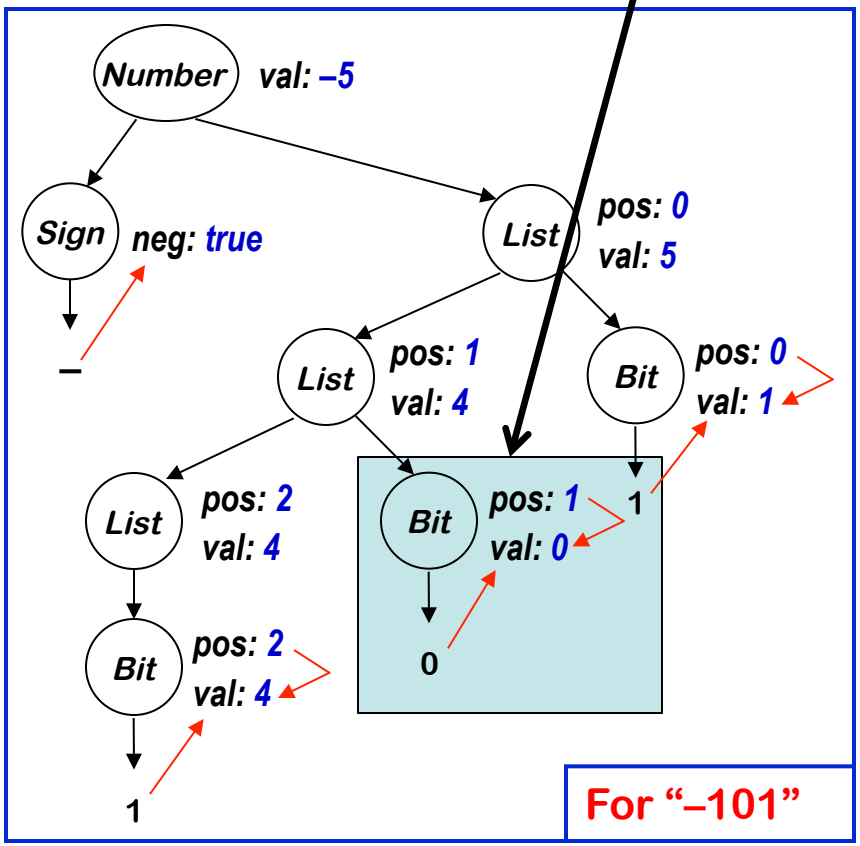


Synthesized attributes

Note: the upward (pointing arrows) flow of information and the flow from sibling attributes

Back to the Example

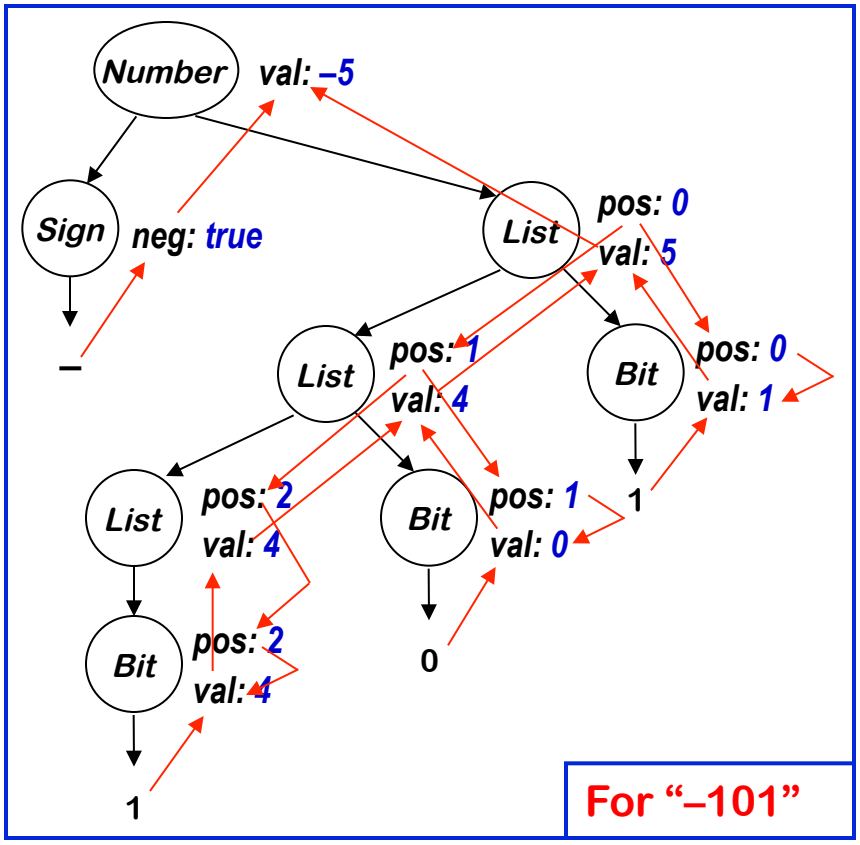
Bit	→ 0	*Bit.val* ← *0*
	| 1	*Bit.val* ← $2^{Bit.pos}$



Synthesized attributes

Note: the upward (pointing arrows) flow of information and the flow from sibling attributes

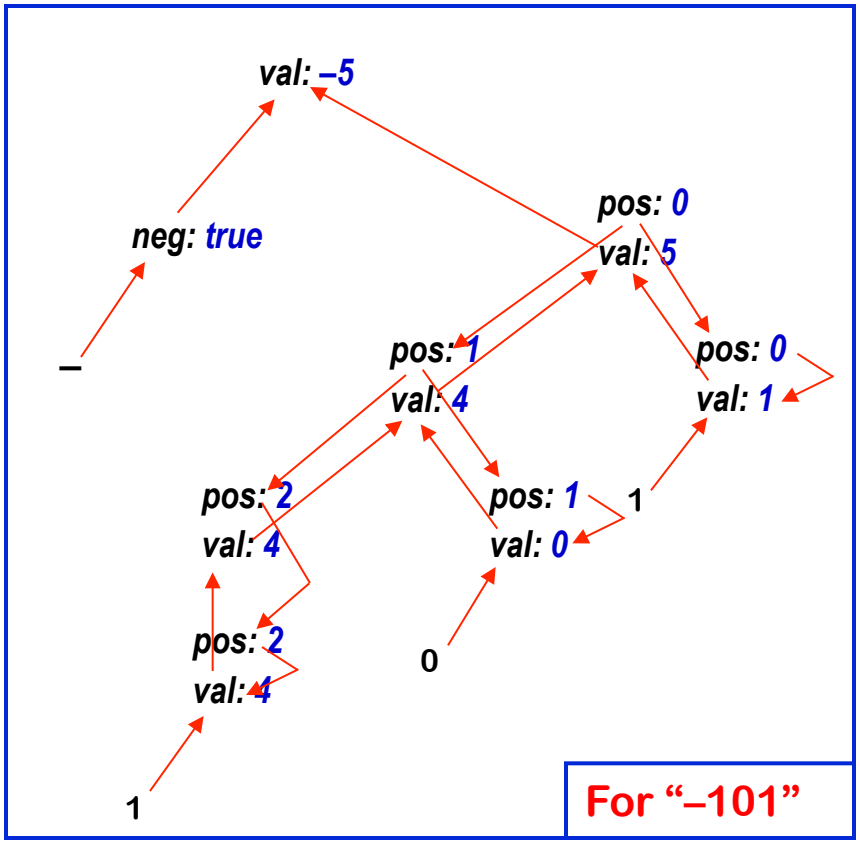
Back to the Example



If we show the computation …

then peel away the parse tree …

Back to the Example



The dependence graph must be acyclic (no cycles!)

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.

An Extended Example

Grammar for a basic block (§ 4.3.3)

$Block_0$	→	$Block_1$ *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*

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*