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)*
- 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 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? ***(attributed grammars?)***
- 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
  - → Knowledge is power
  - → Information is immunization

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*
- The rules specify how to compute a value for each attribute

Example grammar

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

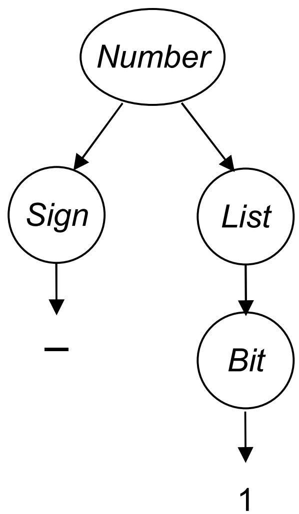
This grammar describes signed binary numbers

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

Examples

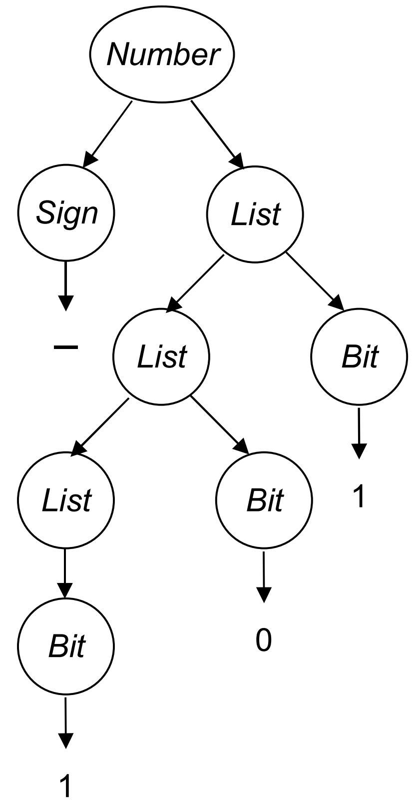
For "–1"

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

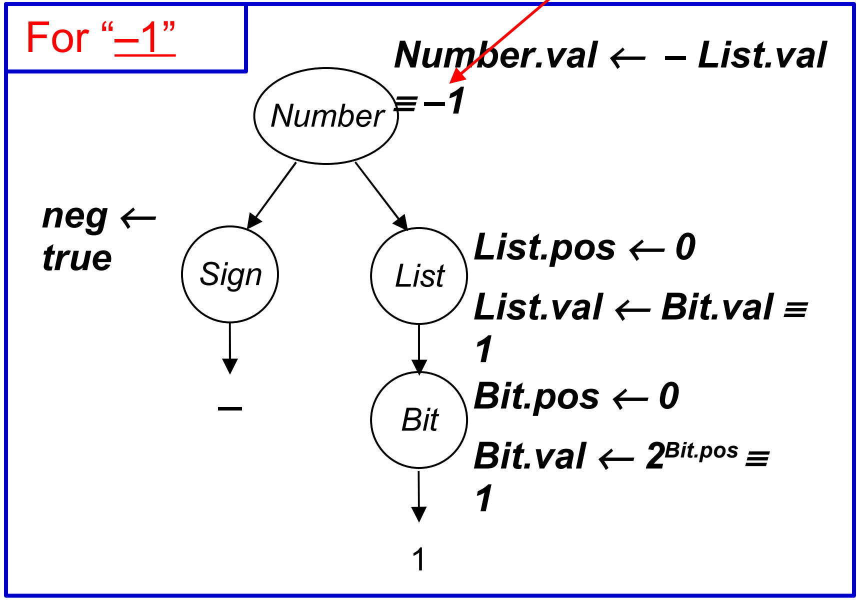
Add rules to compute the decimal value of a signed binary number

Productions	Attribution Rules
Number $\rightarrow$ *Sign List*	***List.pos* $\leftarrow$ *0*** ***If Sign.neg*** ***then Number.val* $\leftarrow$ *– List.val*** ***else Number.val* $\leftarrow$ *List.val***
Sign $\rightarrow$ <u>+</u>	***Sign.neg* $\leftarrow$ *false***
| <u>–</u>	***Sign.neg* $\leftarrow$ *true***
$List_0 \rightarrow List_1\ Bit$	$\mathbf{List_1.pos} \leftarrow List_0.pos + 1$ $\mathbf{Bit.pos} \leftarrow List_0.pos$ $\mathbf{List_0.val} \leftarrow List_1.val + Bit.val$
| *Bit*	***Bit.pos* $\leftarrow$ *List.pos*** ***List.val* $\leftarrow$ *Bit.val***
Bit $\rightarrow$ 0	***Bit.val* $\leftarrow$ *0***
| 1	***Bit.val*** $\leftarrow \mathbf{2^{Bit.pos}}$

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

Back to the Examples

Rules + parse tree imply an attribute dependence graph



One possible evaluation order:

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

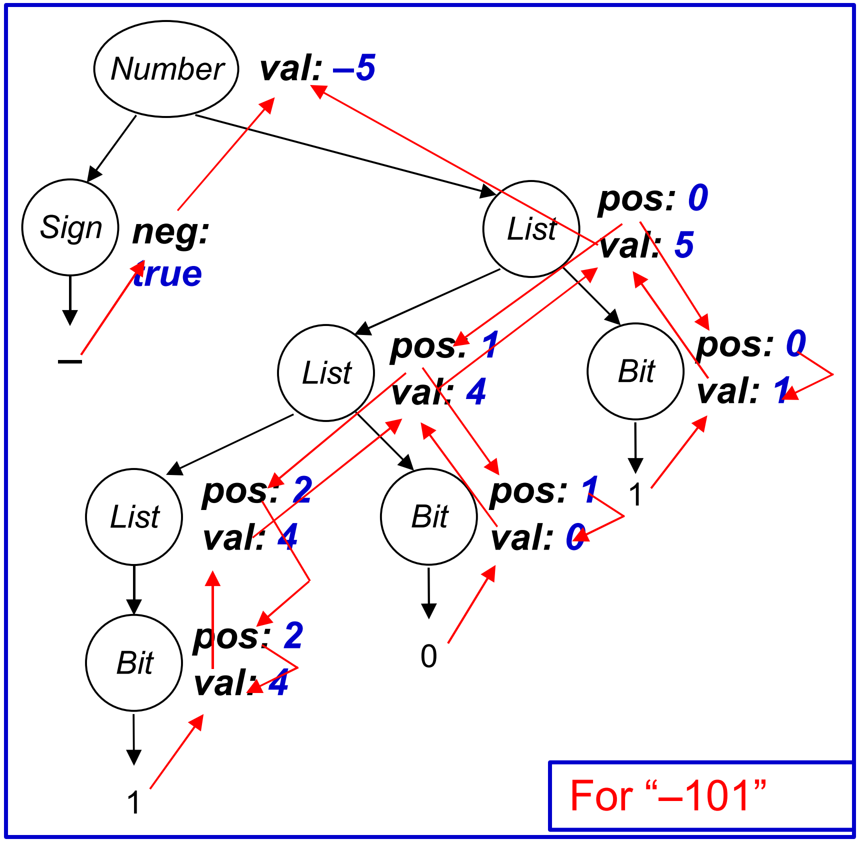
Other orders are possible

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



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

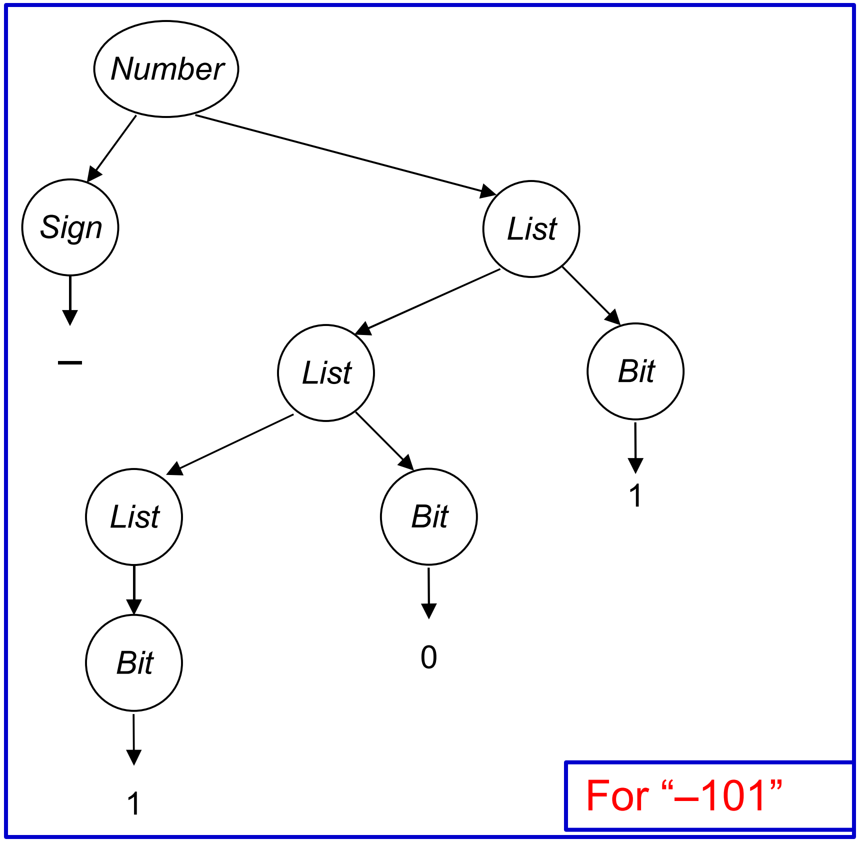
Rule-based methods *(treewalk)*

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

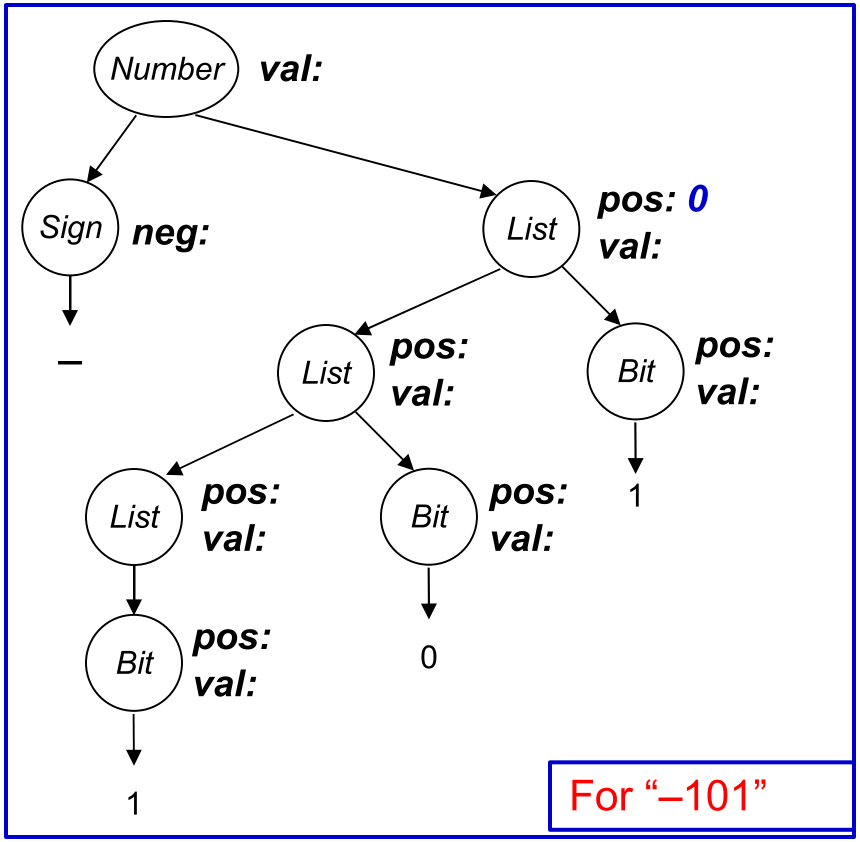
Oblivious methods *(passes, dataflow)*

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

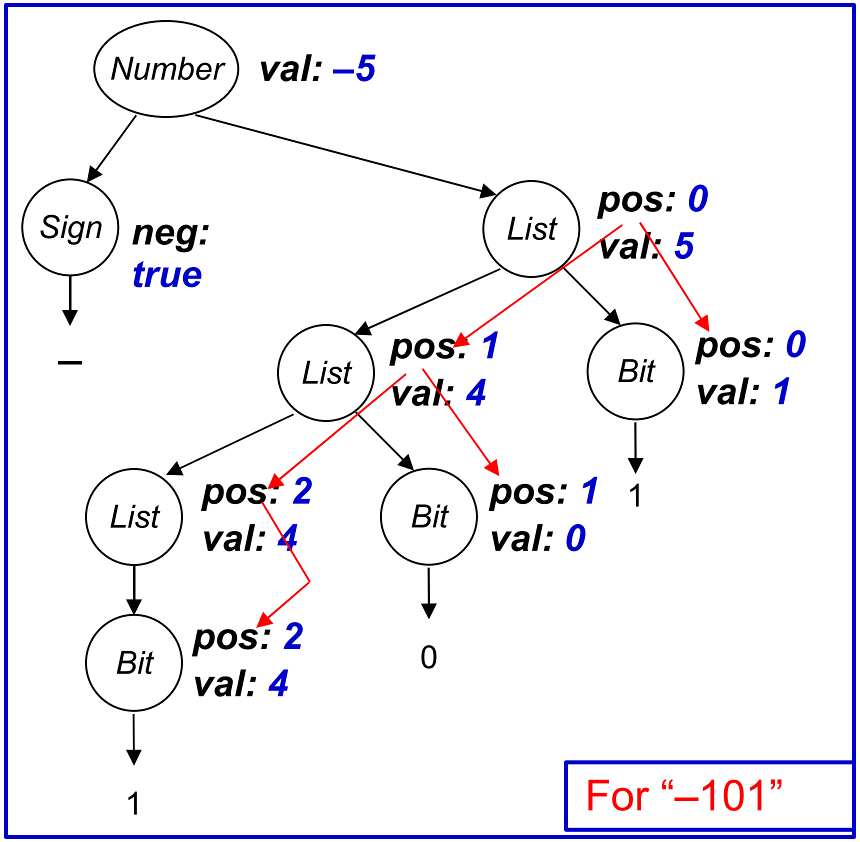
Back to the Example



Back to the Example

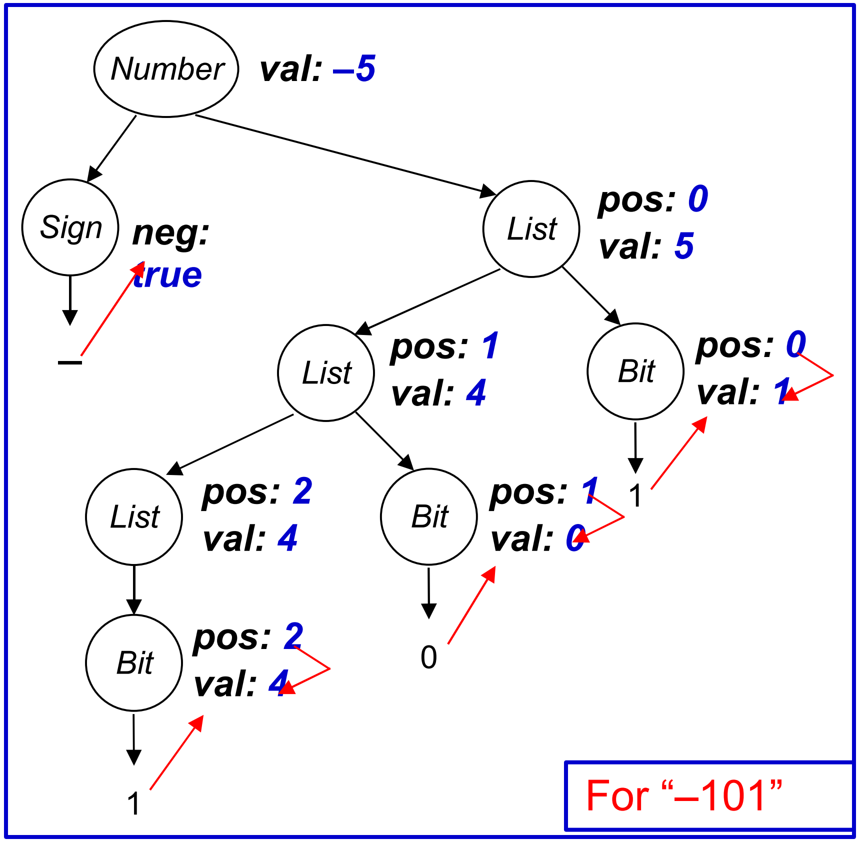


Back to the Example



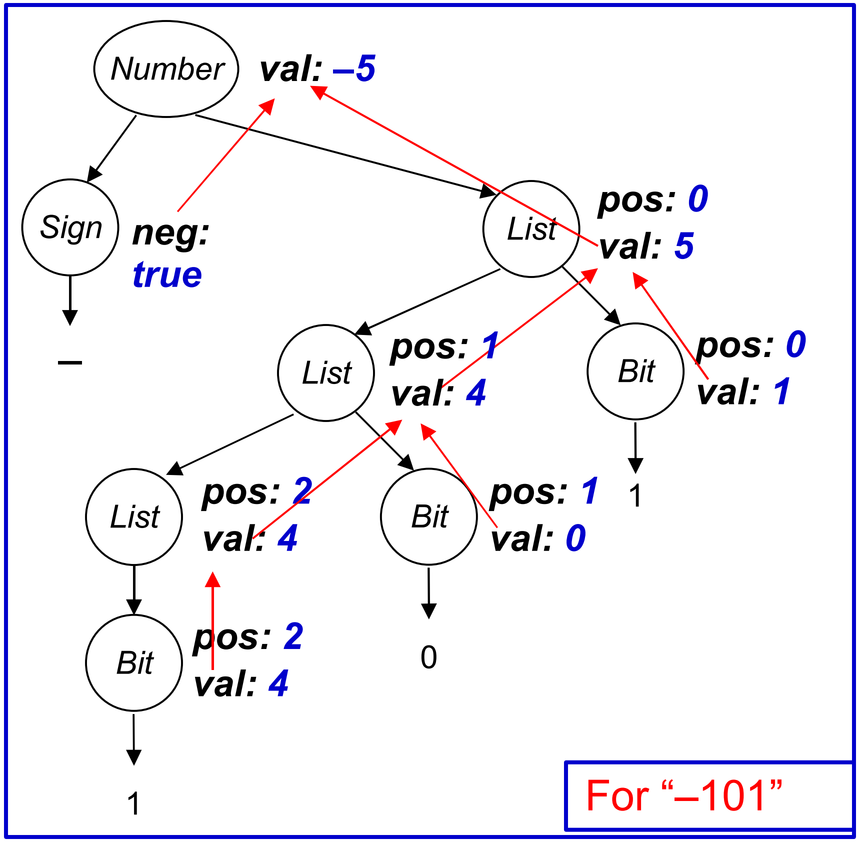
Inherited Attributes

Back to the Example



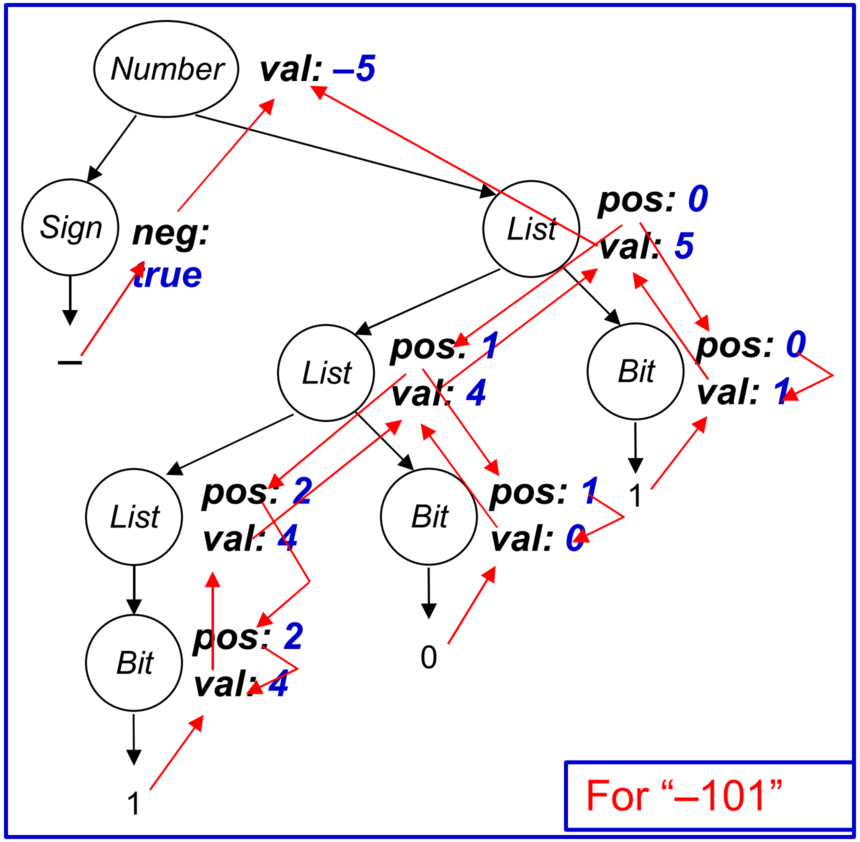
Synthesized attributes

Back to the Example



Synthesized attributes

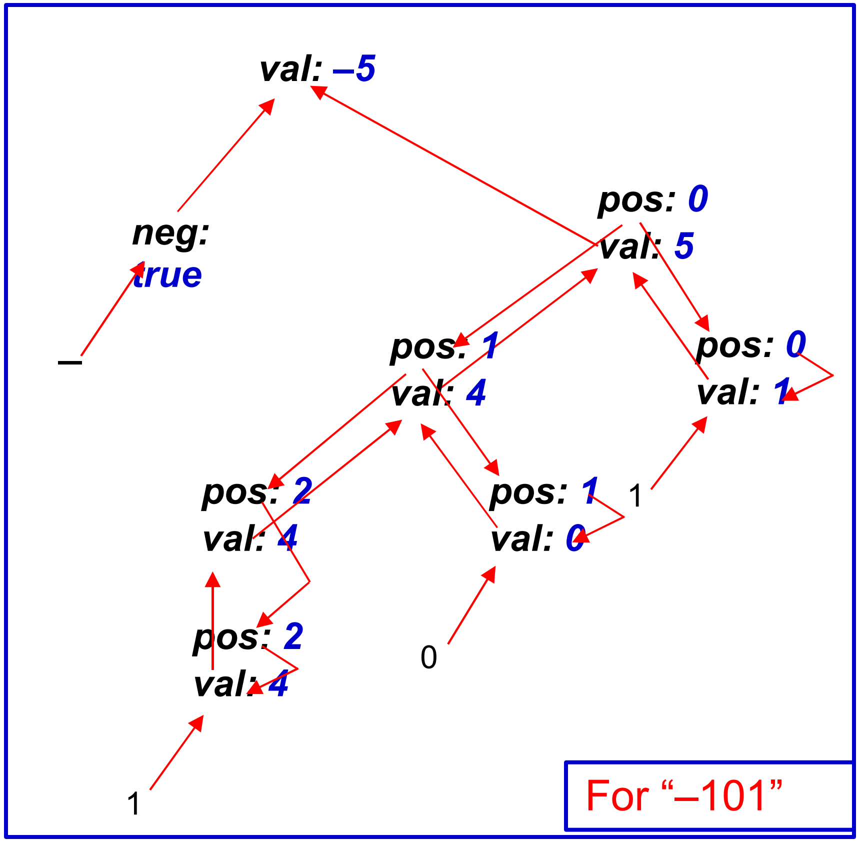
Back to the Example



If we show the computation ...

& then peel away the parse tree ...

Back to the Example



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

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate a snippet of code with each production
- At each reduction, the corresponding snippet runs
- Allowing arbitrary code provides complete flexibility
  - → Includes ability to do tasteless & bad things

To make this work

- Need names for attributes of each symbol on *lhs* & *rhs*
  - → Typically, one attribute passed through parser + arbitrary code (structures, globals, statics, …)
  - → Yacc introduced **$$**, **$1**, **$2**, … **$n**, left to right
- Need an evaluation scheme
  - → Fits nicely into **LR(1)** parsing algorithm

Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc-like notation

Typical Uses

- Building a symbol table
  - → Enter declaration information as processed
  - → At end of declaration syntax, do some post processing
  - → Use table to check errors as parsing progresses

assumes table is global

- Simple error checking/type checking
  - → Define before use → lookup on reference
  - → Dimension, type, ... → check as encountered
  - → Type conformability of expression → bottom-up walk
  - → Procedure interfaces are harder
    - ♦ Build a representation for parameter list & types
    - ♦ Create list of sites to check
    - ♦ Check offline, or handle the cases for arbitrary orderings

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (Somewhat) abstract names for symbols

Differences

- Actions applied as a unit; not true for AG rules
- Anything goes in *ad-hoc* actions; AG rules are functional
- AG rules are higher level than *ad-hoc* actions

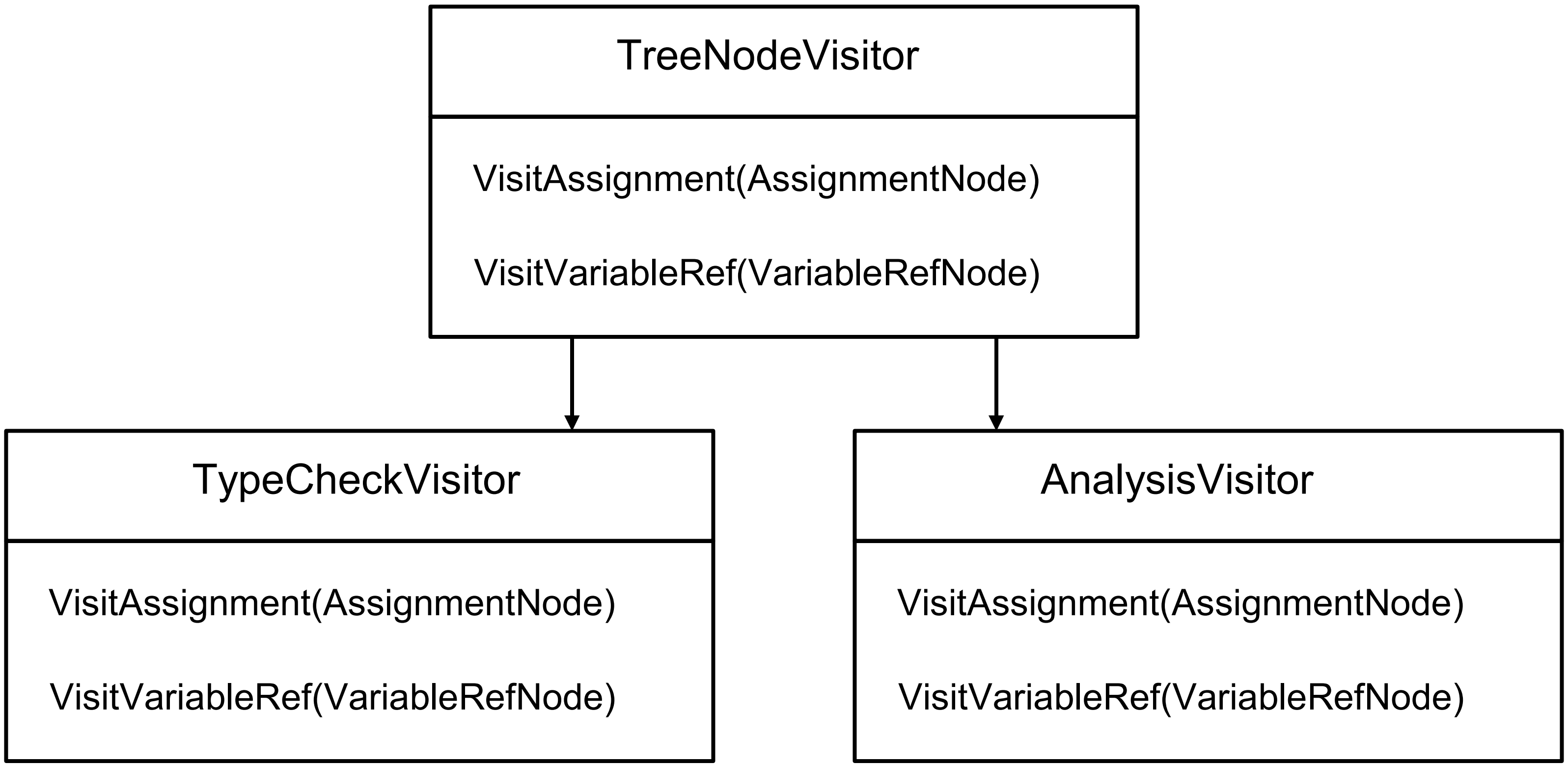
Limitations

- Forced to evaluate in a given order: *postorder*
  - → Left to right only
  - → Bottom up only

- Implications
  - → Declarations before uses
  - → Context information cannot be passed down
    - ♦ How do you know what rule you are called from within?
    - ♦ Example: cannot pass bit position from right down
  - → Could you use globals?
    - ▪ In this case we could get the position from the left, which is not much help (and it requires initialization)

Alternative Strategy

- Build Abstract Syntax Tree
  - → Use tree walk routines
  - → Use "visitor" design pattern to add functionality



Summary: Strategies for Context-Sensitive Analysis

- Attribute Grammars
  - → Pros: Formal, powerful, can deal with propagation strategies
  - → Cons: Too many copy rules, no global tables, works on parse tree
- Postorder Code Execution
  - → Pros: Simple and functional, can be specified in grammar (Yacc) but does not require parse tree
  - → Cons: Rigid evaluation order, no context inheritance
- Generalized Tree Walk
  - → Pros: Full power and generality, operates on abstract syntax tree (using Visitor pattern)
  - → Cons: Requires specific code for each tree node type, more complicated