# CS 4240: Compilers

Lecture 24: Attribute Grammars, Type Checking

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

April 17, 2019

# ANNOUNCEMENTS & REMINDERS

- **Last class is next Monday, April 22nd**
  - **Course review. Practice Final Exam will also be released that day.**
- **Project 3 due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Tuesday, April 30th**
  - 10% of course grade
- **Homework 3 due by 11:59pm on Tuesday, April 23rd**
  - **Automatic penalty-free extension until 11:59pm on Friday, April 26th**
  - 5% of course grade
- **FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm**
  - 30% of course grade

# Worksheet # 23 Solution

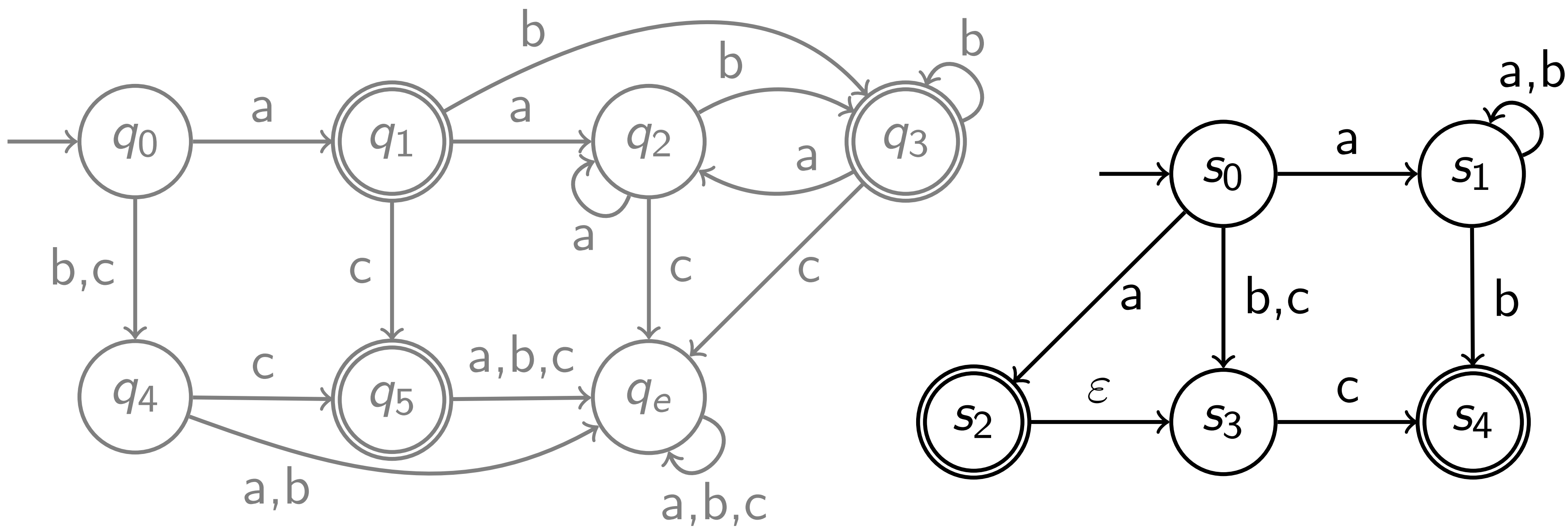
(From Lecture #23 given on 4/15/2019)

***(Recap)***

*NOTE: for convenience, a missing transition indicates an error if that symbol is encountered (in lieu of creating an explicit error state like $q_e$)*

## DFAs:

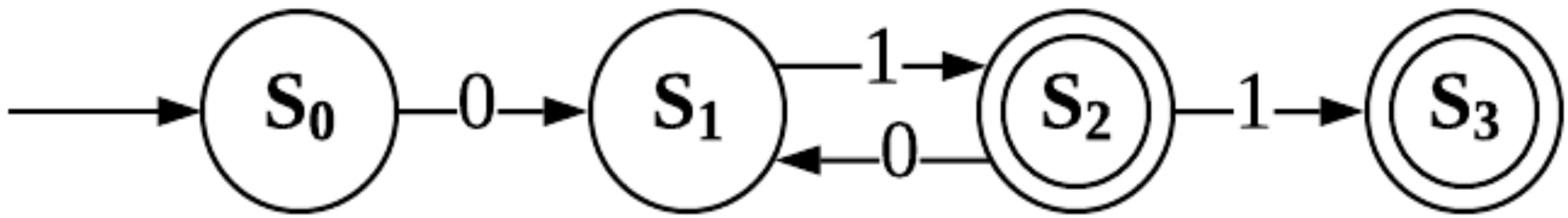
- *at most* *one* outgoing transition per $a \in \Sigma$
- decide acceptance directly




## NFAs:

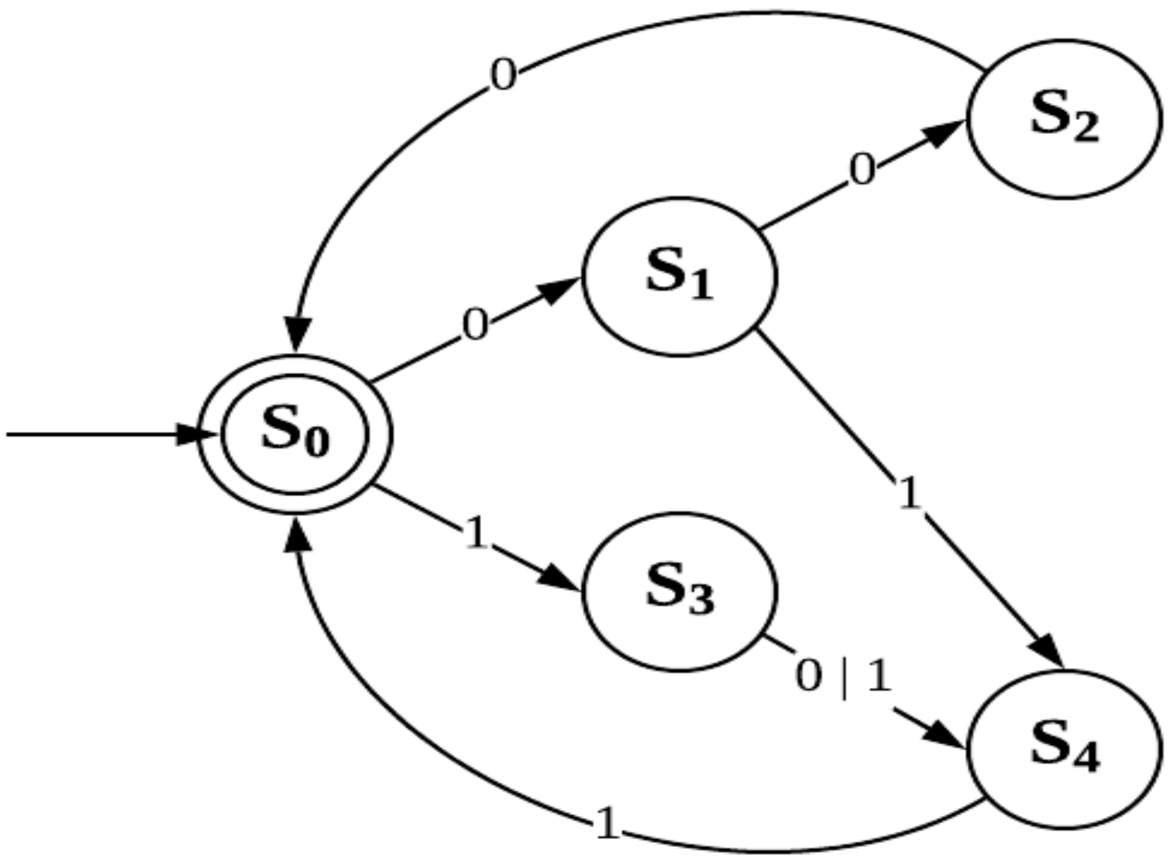
- *zero or more* transitions per $a \in \Sigma \cup \{\varepsilon\}$
- acceptance: exists an accepting run
- rejection: all possible runs reject

- Draw a DFA which accepts the same strings as the following regular expression : **$(01)^+(1?)$**




- The most popular answer from students was the DFA above. There can be other possible answers.
- **One of the differences between NFA & DFA :**
  **For every state,**
  - **DFA : at most one outgoing transition per**
  - **NFA : zero or more transitions per**

- Draw a DFA for the following language or explain why no such DFA exists: **The language of binary strings of length 3k for k>=0 such that every third bit is the value yielded by performing a logical 'or' on the preceding two bits.**



# 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 …)

# To generate code, we need to understand its meaning !

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 <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"

# 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 cannot be expressed in a Regular Expression or Context Free Grammar

# 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 parsing, formalism won; for semantic analysis, ad-hoc techniques dominate actual practice

# 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

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

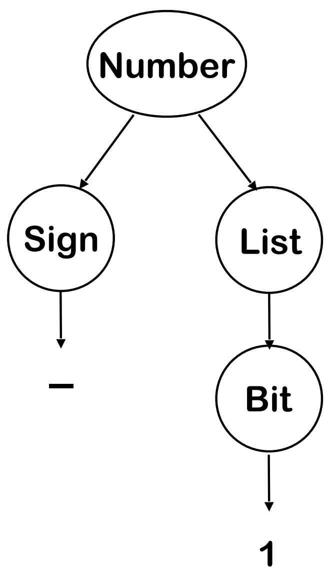
This grammar describes signed binary numbers

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

# Examples

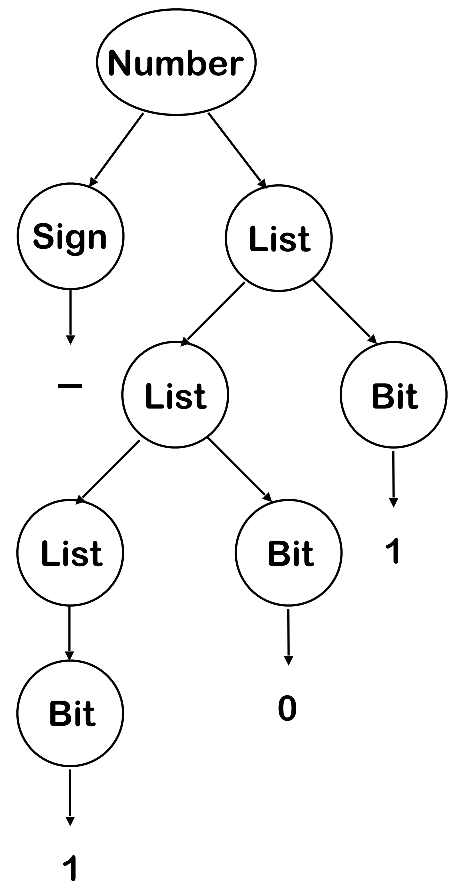
**For "–1"**

Number → Sign List
→ 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 throughout the lecture

# Attribute Grammars

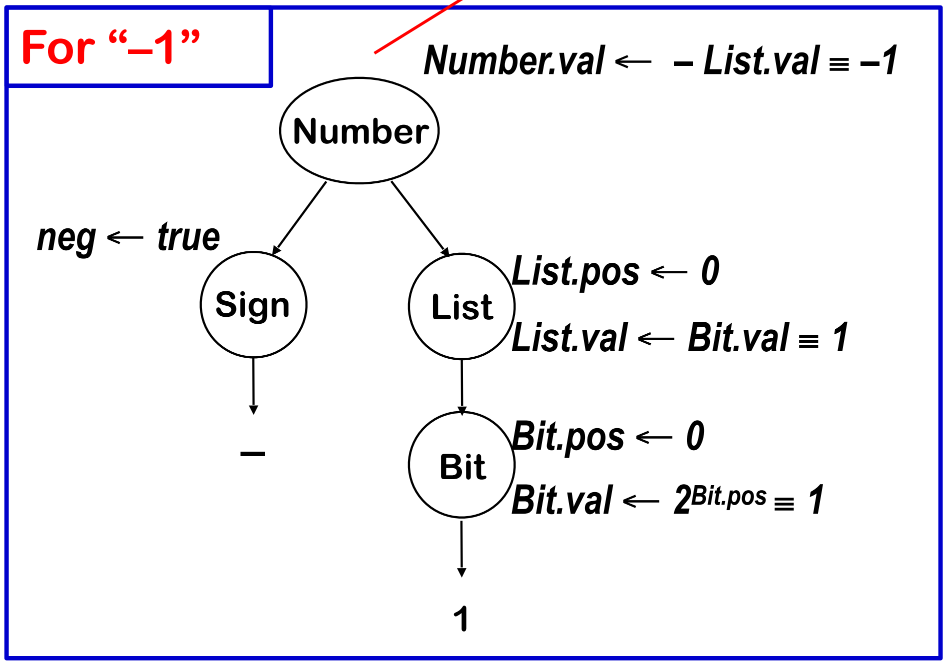
Add attributes & rules to compute the decimal value of a signed binary number

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




One possible evaluation order:

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

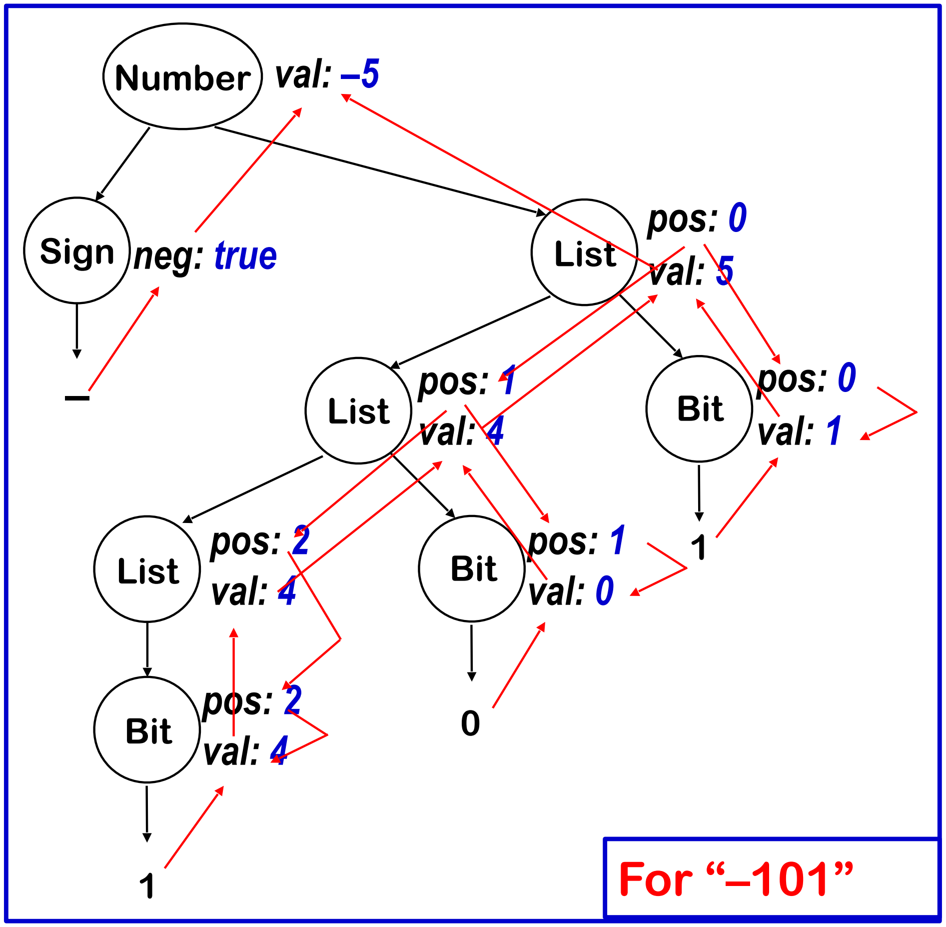
Other orders are possible

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

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 | Inherited Attributes |
|---|---|
| • Use values from children & from constants | • Use values from parent, constants, & siblings |
| • S-attributed grammars | • directly express context |
| • Evaluate in a single bottom-up pass | • can rewrite to avoid them |
| | • Thought to be more natural |
| Good match to LR parsing | Not easily done at parse time |

We want to use both kinds of attributes

# Evaluation Methods

## Dynamic, dependence-based methods

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

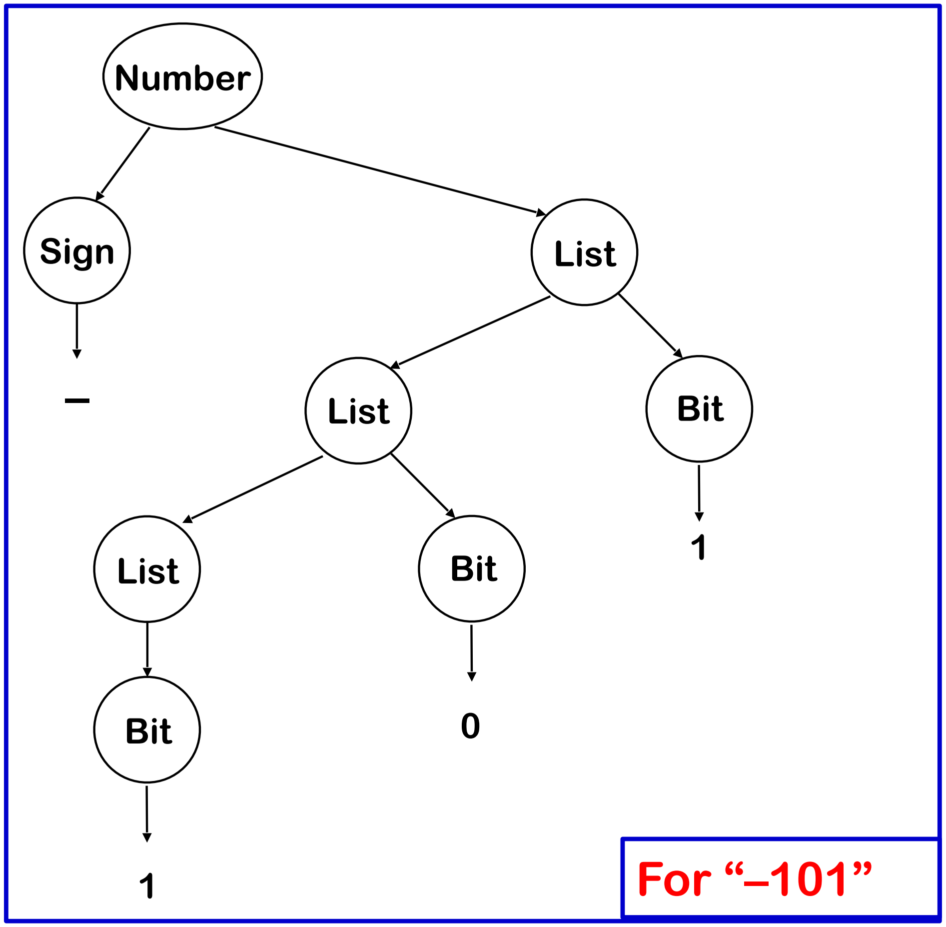
## Rule-based methods (treewalk)

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

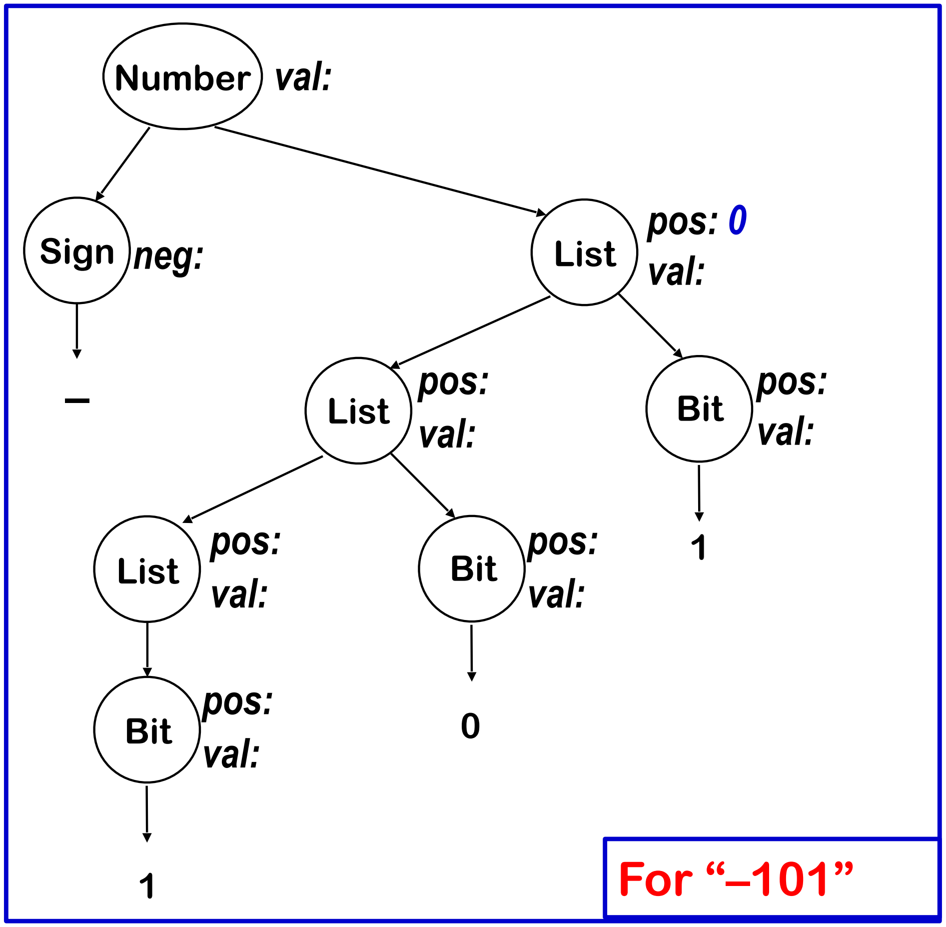
## Oblivious methods (passes, dataflow)

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

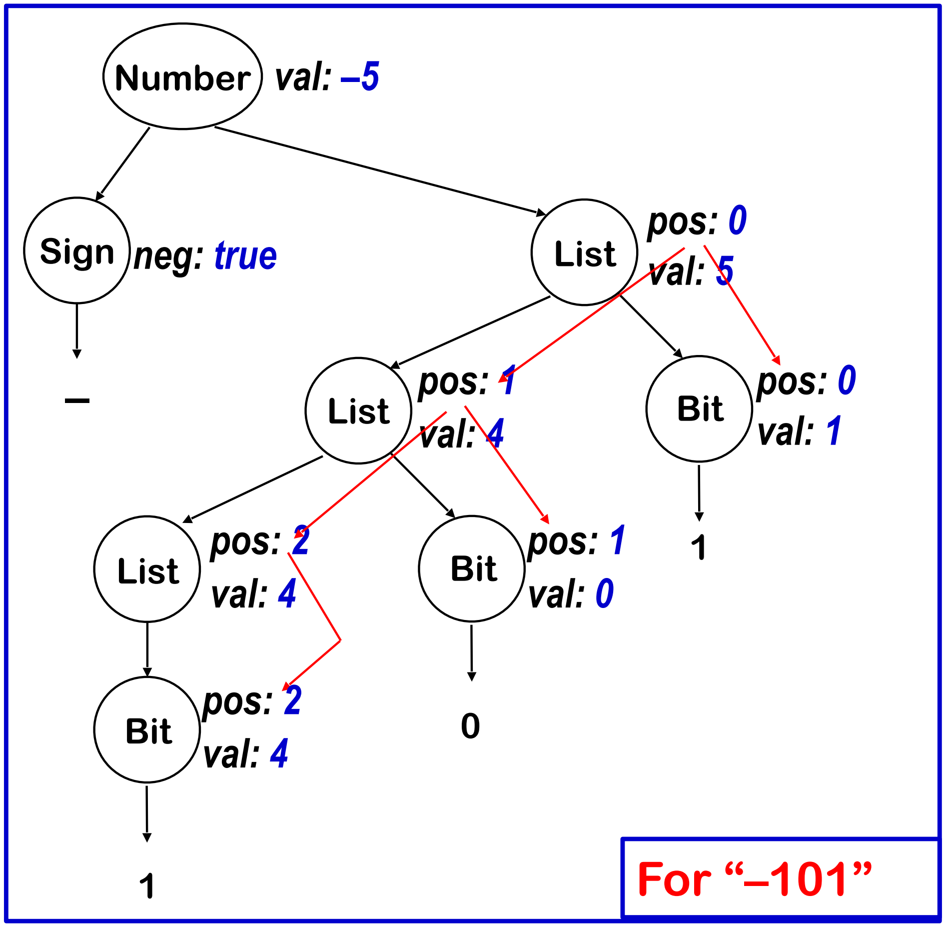
# Back to the Example



# Back to the Example

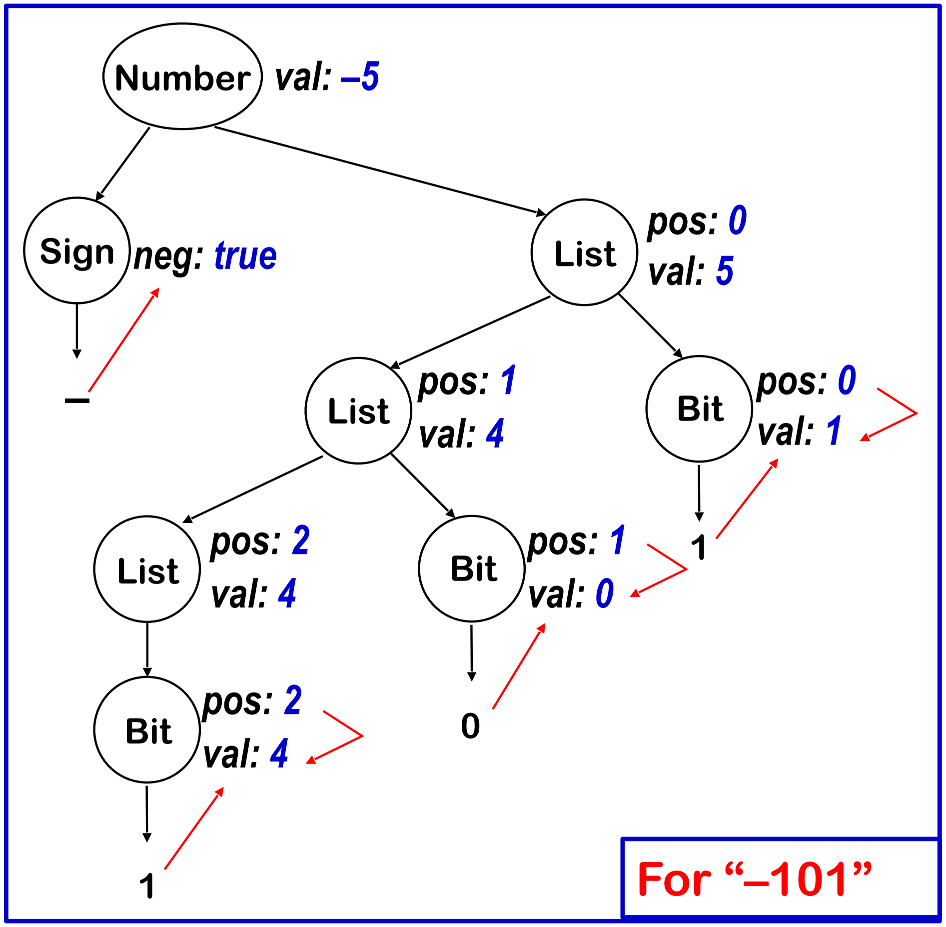


# Back to the Example



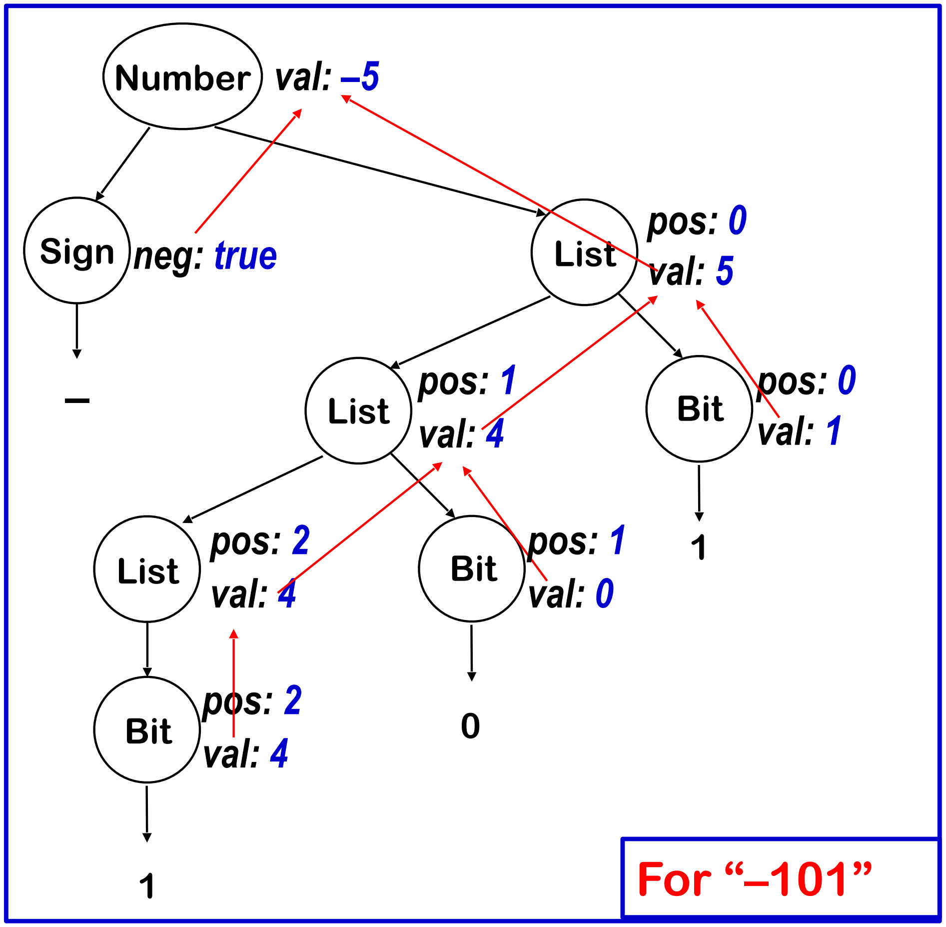

Inherited Attributes

# Back to the Example



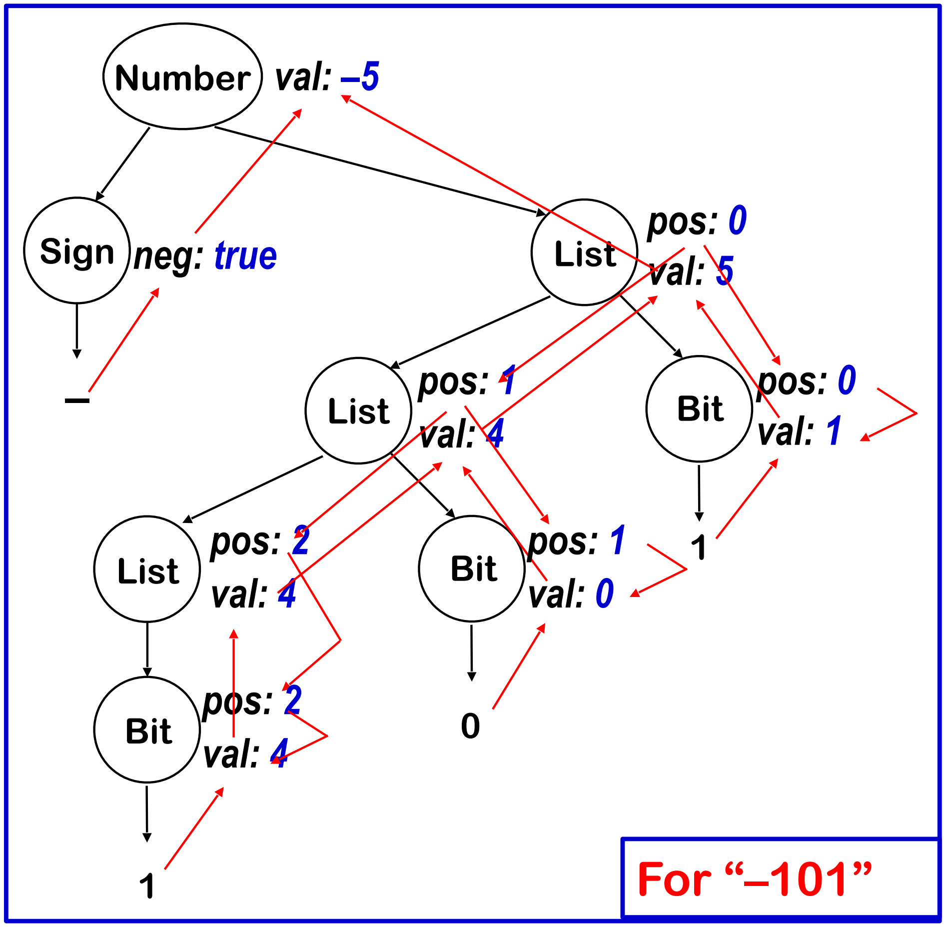

Synthesized attributes

# Back to the Example




Synthesized attributes

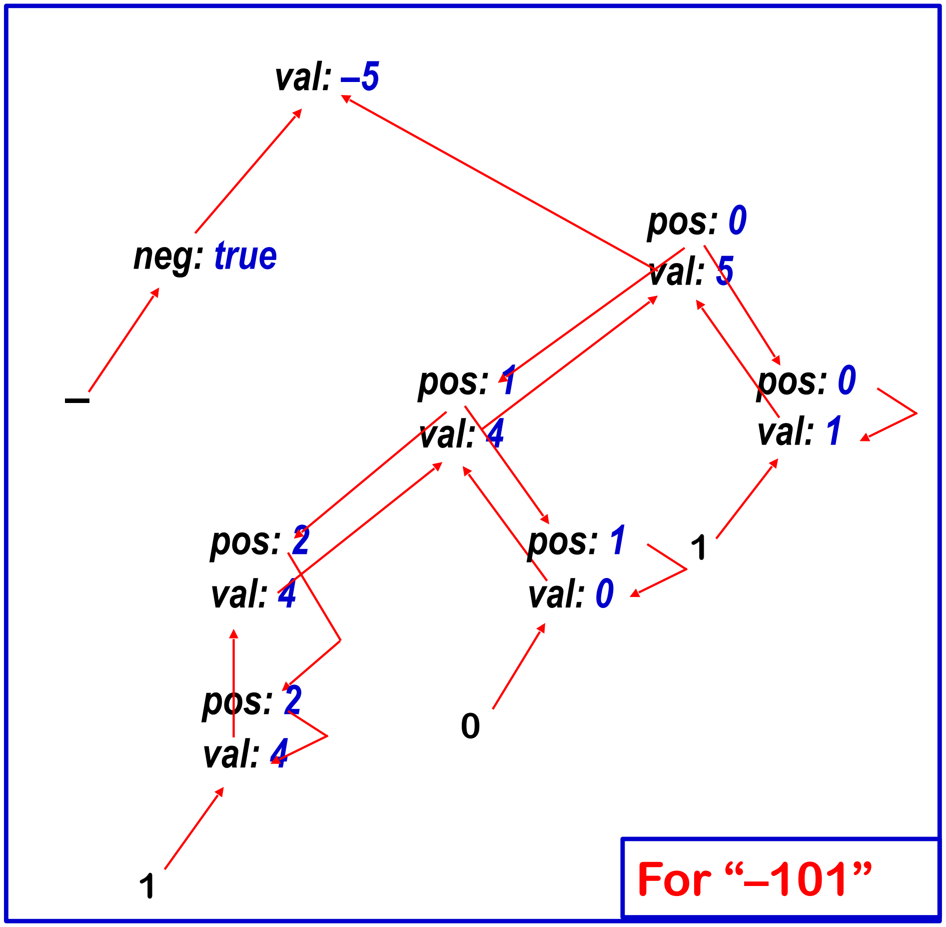
# Back to the Example




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

# Attribute grammar tradeoffs

- **Pros**: compiler writer thinks declaratively, lets a single engine do all the computation
  - Similar to parsers and parser-generators
- **Cons**: restrictive enough that they can be a bit awkward for expressing some practical analyses

# Ad-hoc analysis

- Declare program state per AST node (analogous to node features) and **globally**
- Define code that's run on completed AST to update its program state when tree is parsed (analogous to rules)
- **Advantage**: don't need to write out explicit copies of data through tree nodes
- **Disadvantage**: have to implicitly reason about order of dependencies

# Ad-hoc analysis

- Declare type of data computed for each AST non-terminal
- Declare global data
- For each parse rule **A -> B C**, define code that is run when rule is applied. Can refer to:
  - Constants, global data
  - Features of **A** node
  - Features of **B** node
  - Features of **C** node

# Type checking as ad-hoc analysis

- Out of all of the features declared, only ones relevant for the next compiler phases are:
  - **prog**: **isWellTyped**
  - **vardecls**: **idType**
  - **expression**: **typeOf**
- Other features (**idType**, **paramType**, **resType**, **curRes** at statement, expression), i.e. the **typing contexts**, are irrelevant
- Candidates to be represented in mutable global in an ad-hoc analysis

# Type checking as ad-hoc analysis

- **Basic idea**: given program **P**, check that **P** is well-typed…
  - **Q**: for each assignment **x := e**…?
  - **Q**: for each **if-then-else**
  - **Q**: for each operator **op** and expression **e0 op e1**…?
  - **Q**: for each function **f**?
  - **Q**: for each call?

# Type checking as ad-hoc analysis

- Dependency analysis implies:
  - Type context of parent is computed before
    type context of children
  - Typing context is computed before **typeOf**, **isWellTyped**
- Assume in addition that **typeof**, **isWellTyped** is computed before type contexts of siblings

# Summary

- Semantic analysis: determining properties of an AST
- Attribute grammars: description of equations to generate and solve
- Ad-hoc analysis: arbitrary code that maintains global state, conditions for when to run it
  - Simpler when information domain is complex, and has to be copied around a lot with attribute grammars