# From Lexical and Syntax Analysis to Semantic Analysis

Some of the following material is taken from Cooper's class.

# Find 6 problems with this code. These issues go beyond syntax.

```
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 kinds of questions does the semantic analysis/code generator need to answer?

# Semantic Analysis = Values/Meaning
# Context-Sensitive Analysis

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 parsing, formalism won; here, 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*

| | | | |
|---|---|---|---|
| 1 | *Number* | → | *Sign List* |
| 2 | *Sign* | → | + |
| 3 | | \| | - |
| 4 | *List* | → | *List Bit* |
| 5 | | \| | *Bit* |
| 6 | *Bit* | → | 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

# Example Trees for this Grammar

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

**Consider strings:**

1011

-10

# Adding Attributes/semantic rules:

| Productions | | | Attribution Rules |
|---|---|---|---|
| *Number* | → | *Sign List* | |
| *Sign* | → | + | |
| | \| | - | |
| $List_0$ | → | $List_1$ *Bit* | |
| | \| | *Bit* | |
| *Bit* | → | 0 | Bit.val = 0 |
| | \| | 1 | Bit.val = 2 ^ Bit.pos |

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

# Answers – the semantic rules

$List.pos \leftarrow 0$

*if Sign.neg*

*then* $Number.val \leftarrow - List.val$

*else* $Number.val \leftarrow List.val$

$Sign.neg \leftarrow false$

$Sign.neg \leftarrow true$

$List_1.pos \leftarrow List_0.pos + 1$

$Bit.pos \leftarrow List_0.pos$

$List_0.val \leftarrow List_1.val + Bit.val$

$Bit.pos \leftarrow List.pos$

$List.val \leftarrow Bit.val$

$Bit.val \leftarrow 0$

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

# Try Evaluating the values for examples

- **Consider strings:**

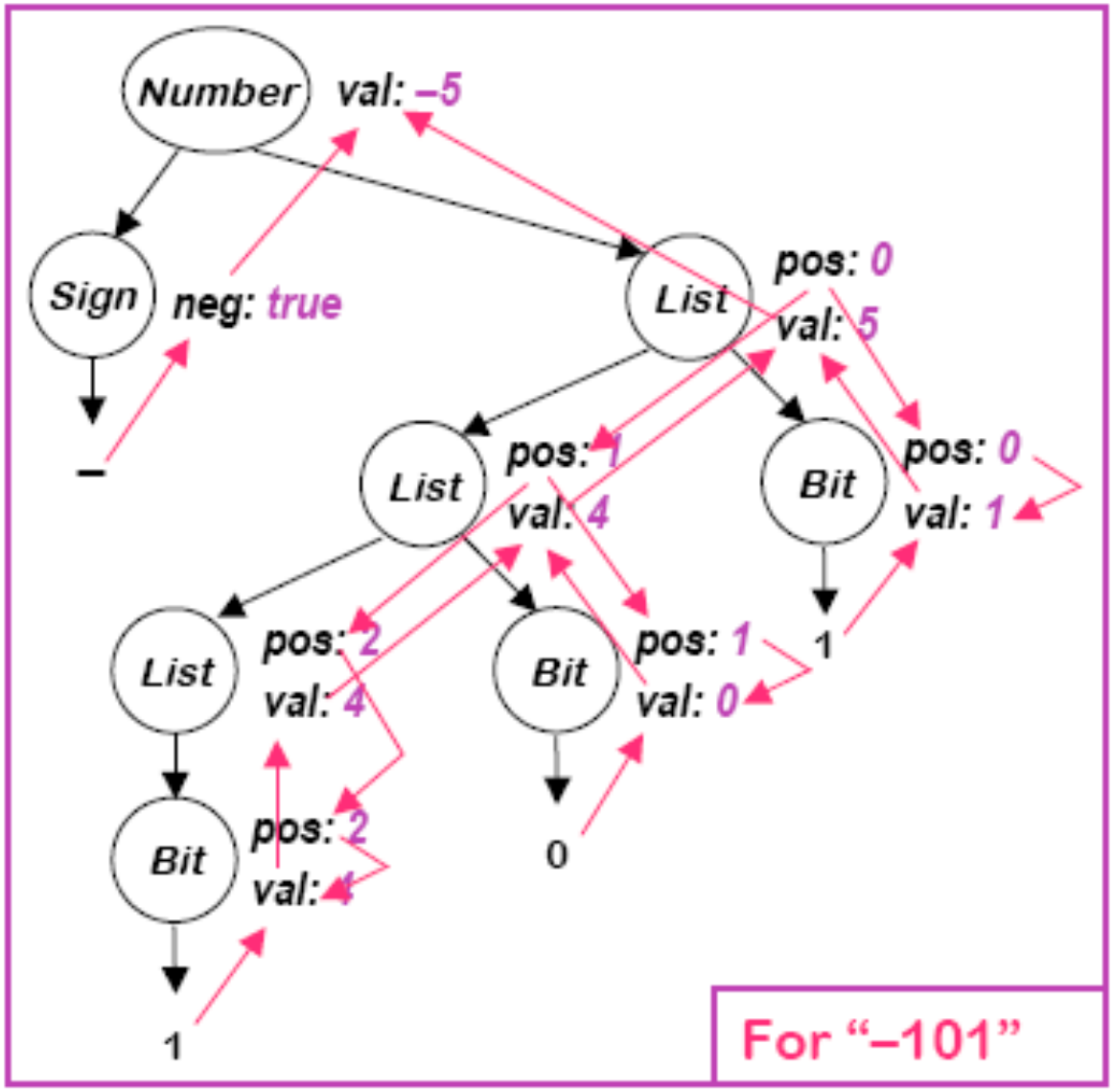
- 1011

- -10

# Some Terminology/Observations

- Attributes
- Attribute rules
- Decorating the tree
- Dependences among attributes -> evaluation order
- Attribute dependence graph
- Inherited attributes versus synthesized attributes

# Which attributes are inherited? Synthesized?



# Try another rule set for same problem

| Productions | | | Attribution Rules |
|---|---|---|---|
| *Number* | → | *Sign List* | |
| *Sign* | → | + | |
| | \| | - | |
| $List_0$ | → | $List_1$ *Bit* | |
| | \| | *Bit* | |
| *Bit* | → | 0 | Bit.val = 0 |
| | \| | 1 | Bit.val = 1 |

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

# 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

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

# 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

# Evaluation Methods

## Dynamic, dependence-based methods

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

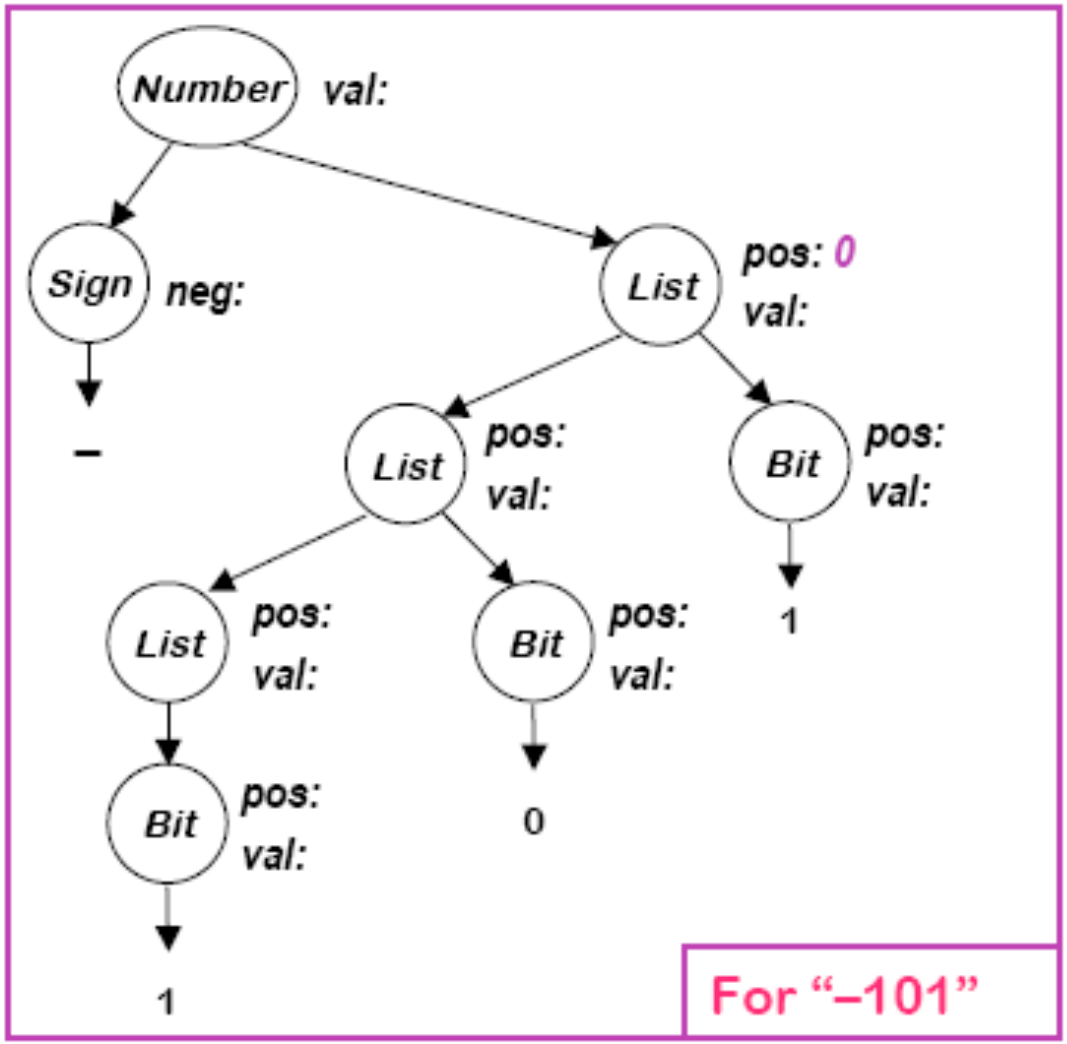
## Rule-based methods *(treewalk)*

- 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

# Take another look at example




Inherited attributes?

Synthesized attributes?

Attributed dependency graph?

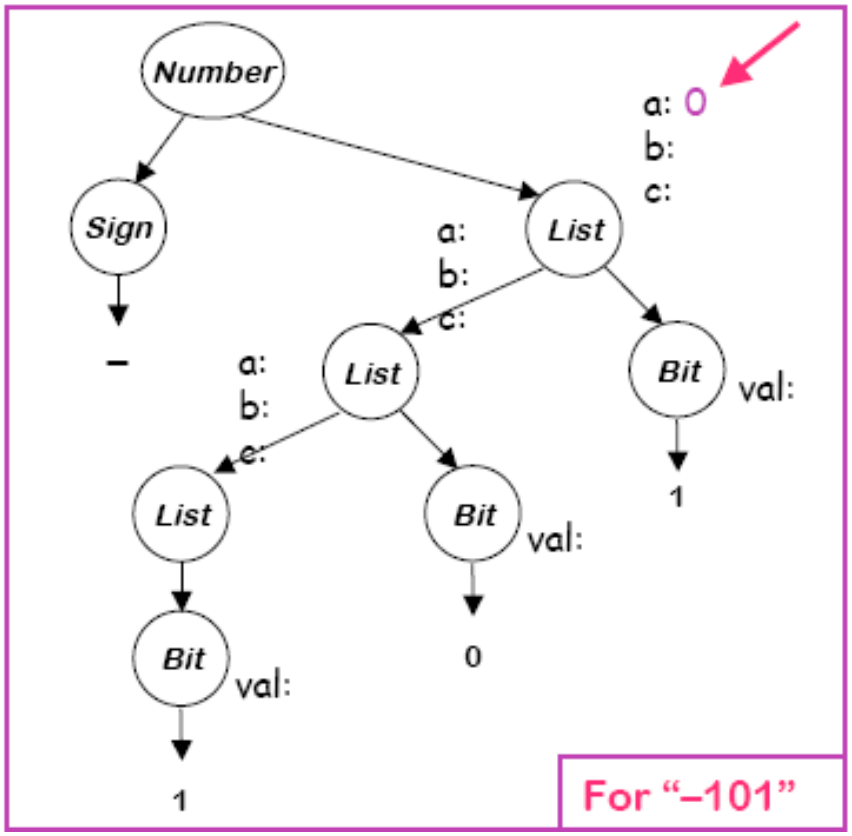
Note: can only evaluate oncircular dependency graphs. General circularity problem is exponential.

# Where is the circularity?

## A Circular Attribute Grammar

| Productions | | | Attribution Rules |
|---|---|---|---|
| *Number* | → | *List* | $List.a \leftarrow 0$ |
| $List_0$ | → | $List_1$ *Bit* | $List_1.a \leftarrow List_0.a + 1$<br>$List_0.b \leftarrow List_1.b$<br>$List_1.c \leftarrow List_1.b + Bit.val$ |
| | \| | *Bit* | $List_0.b \leftarrow List_0.a + List_0.c + Bit.val$ |
| *Bit* | → | *0* | $Bit.val \leftarrow 0$ |
| | \| | *1* | $Bit.val \leftarrow 1$ |

# Circular Grammar Example




| Productions | | | Attribution Rules |
|---|---|---|---|
| *Number* | → | *List* | $List.a \leftarrow 0$ |
| $List_0$ | → | $List_1$ *Bit* | $List_1.a \leftarrow List_0.a + 1$<br>$List_0.b \leftarrow List_1.b$<br>$List_1.c \leftarrow List_1.b +$ *Bit.val* |
| | \| | *Bit* | $List_0.b \leftarrow List_0.a +$<br>$List_0.c +$ *Bit.val* |
| *Bit* | → | *0* | $Bit.val \leftarrow 0$ |
| | \| | *1* | $Bit.val \leftarrow 1$ |

# An Extended Attribute Grammar Example

Grammar for a basic block *(§ 4.3.3)*

| | | | |
|---|---|---|---|
| 1 | $Block_0$ | → | $Block_1$ *Assign* |
| 2 | | \| | *Assign* |
| 3 | $Assign_0$ | → | *Ident* = *Expr* ; |
| 4 | $Expr_0$ | → | $Expr_1$ + *Term* |
| 5 | | \| | $Expr_1$ - *Term* |
| 6 | | \| | *Term* |
| 7 | $Term_0$ | → | $Term_1$ * *Factor* |
| 8 | | \| | $Term_1$ / *Factor* |
| 9 | | \| | *Factor* |
| 10 | *Factor* | → | ( *Expr* ) |
| 11 | | \| | *Number* |
| 12 | | \| | *Ident* |

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*

Hey, this looks useful !

# An Extended Example (*continued*)

| | | | | |
|---|---|---|---|---|
| 1 | $Block_0$ | → | $Block_1$ Assign | $Block_0.cost \leftarrow Block_1.cost +$ Assign.cost |
| 2 | | \| | Assign | $Block_0.cost \leftarrow$ Assign.cost |
| 3 | $Assign_0$ | → | Ident = Expr ; | Assign.cost ← COST(store) + Expr.cost |
| 4 | $Expr_0$ | → | $Expr_1$ + Term | $Expr_0.cost \leftarrow Expr_1.cost +$ COST(add) + Term.cost |
| 5 | | \| | $Expr_1$ - Term | $Expr_0.cost \leftarrow Expr_1.cost +$ COST(sub) + Term.cost |
| 6 | | \| | Term | $Expr_0.cost \leftarrow$ Term.cost |
| 7 | $Term_0$ | → | $Term_1$ * Factor | $Term_0.cost \leftarrow Term_1.cost +$ COST(mult) + Factor.cost |
| 8 | | \| | $Term_1$ / Factor | $Term_0.cost \leftarrow Term_1.cost +$ COST(div) + Factor.cost |
| 9 | | \| | Factor | $Term_0.cost \leftarrow$ Factor.cost |
| 10 | Factor | → | ( Expr ) | Factor.cost ← Expr.cost |
| 11 | | \| | Number | Factor.cost ← COST(loadI) |
| 12 | | \| | Ident | Factor.cost ← COST(load) |

# An Extended Example (*continued*)

Properties of the example grammar

- All attributes are synthesized ⇒ 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