RUTGERS
THE STATE UNIVERSITY OF NEW JERSEY

CS415 Compilers

Syntax Analysis
Part 6
and
Context-Sensitive Analysis

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Announcements

Roadmap for the remainder of the course

- Fourth homework:
  Due Friday, April 1
- Project #2 – Bottom-up parser and compiler
  Has been posted; due date Friday April 15
- Project #3 – Peephole optimizer for ILOC
  Will be posted April 15, due May 2 (tentative)
- Second midterm on Wednesday, April 6 (60 minutes in class)
- Final exam on May 10 (60 minutes at assigned location)
- At least 3 more homeworks

Bottom-up Parsing (Syntax Analysis)

EAC Chapters 3.4

YACC : parse.y (preview project #2)

parse.y :

```
%{
#include <stdio.h>
#include "attr.h"
int yylex();
void yyerror(char * s);
#include "symtab.h"
%}
```

Will be included verbatim in parse.tab.c

List of tokens

```
%union {tokentype token; }

%token PROG PERIOD PROC VAR ARRAY RANGE OF
%token INT REAL DOUBLE WRITELN THEN ELSE IF
%token BEG END ASG NOT
%token EQ NEQ LT LEQ GEQ GT OR EXOR AND DIV NOT
%token ID CCONST ICONST RCONST

%start program

%%
program : PROG ID ';' block PERIOD
    ;
block       : BEG ID ASG ICONST END
    ;
```

CFG rules

```
%%

void yyerror(char* s) {
        fprintf(stderr,"%s\n",s);
        }
int
main() {
    printf("1\t");
    yyparse();
    return 1;
    }
```

Main program and "helper" functions; may contain initialization code of global structures. Will be included verbatim in parse.tab.c

Error Recovery in Shift-Reduce Parsers

The problem: parser encounters an invalid token

Goal: Want to parse the rest of the file

Basic idea (panic mode):

- → Assume something went wrong while trying to find handle for nonterminal A
- → Pretend handle for A has been found; pop "handle", skip over input to find terminal that can follow A

Restarting the parser (panic mode):

- → find a restartable state on the stack (has transition for nonterminal A)
- → move to a consistent place in the input (token that can follow A)
- → perform (error) reduction (for nonterminal A)
- → print an informative message

Error Recovery in YACC

Yacc's (bison's) error mechanism (note: version dependent!)

- designated token **error**
- used in error productions of the form

  $A \rightarrow$ **error** α // basic case
- α specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the **error** token
- resumes parsing to match α

  special cases:
  - → α = w, where w is string of terminals: skip input until w has been read
  - → $\alpha = \varepsilon$: skip input until state transition on input token is defined
- error productions can have actions

Error Recovery in YACC

```
cmpdstmt: BEG stmt_list END
stmt_list : stmt
          | stmt_list ';' stmt
          | error { yyerror("\n***Error: illegal statement\n");}
```

This should

- throw out the erroneous statement
- synchronize at ";" or "end" (implicit: $\alpha = \varepsilon$)
- writes message "***Error: illegal statement" to stderr

```
Example:  begin a & 5 | hello ; a := 3 end
                 ↑              ↑ resume parsing
           ***Error: illegal statement
```

RUTGERS

Context-Sensitive Analysis

EaC Chapter 4

ALSU Chapter 5

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[1],
        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[1], 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"

To generate code, we need to understand its meaning !

Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on "values", i.e., something that needs computation; not parts of speech
- Questions & answers involve non-local information

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; somewhat 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 (e.g.: yacc/bison)
- The problems with attribute grammars motivate practice
  - → Non-local computation
  - → Need for centralized information

We will cover attribute grammars, then move on to *ad-hoc* ideas (syntax-directed translation schemes)

Attribute Grammars (AGs)

What is an attribute grammar?

- Each symbol in the derivation (instance of a token or non-terminal) may have a value, or *attribute;*
- A context-free grammar augmented with a set of rules
- The rules specify how to compute a value for each attribute

Example grammar

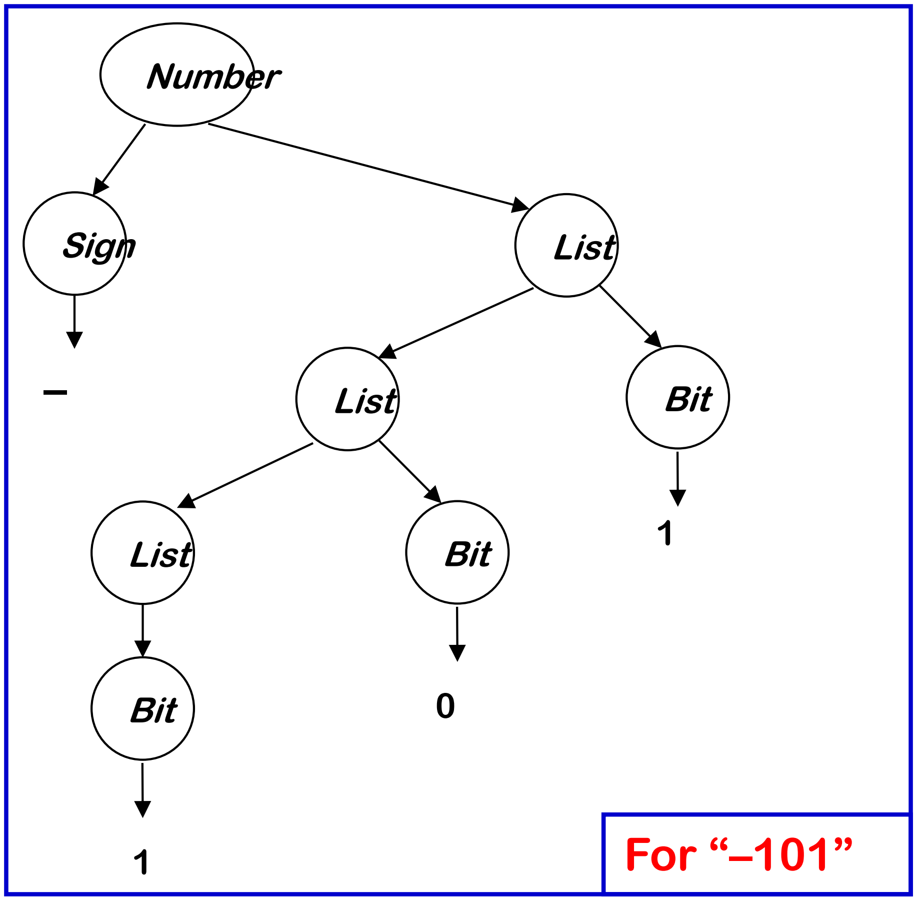
Number	→	Sign List
Sign	→	+
	|	–
List	→	List Bit
	|	Bit
Bit	→	0
	|	1

This grammar describes signed binary numbers

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

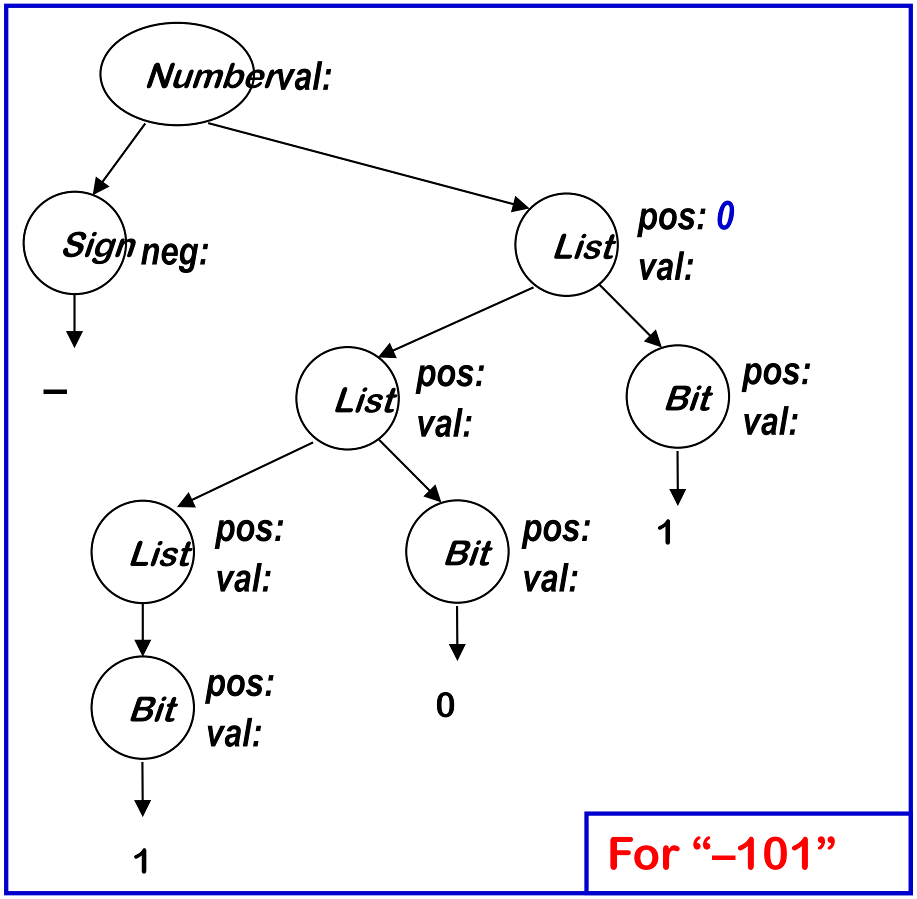
Example parse tree

compute the decimal value of a signed binary number



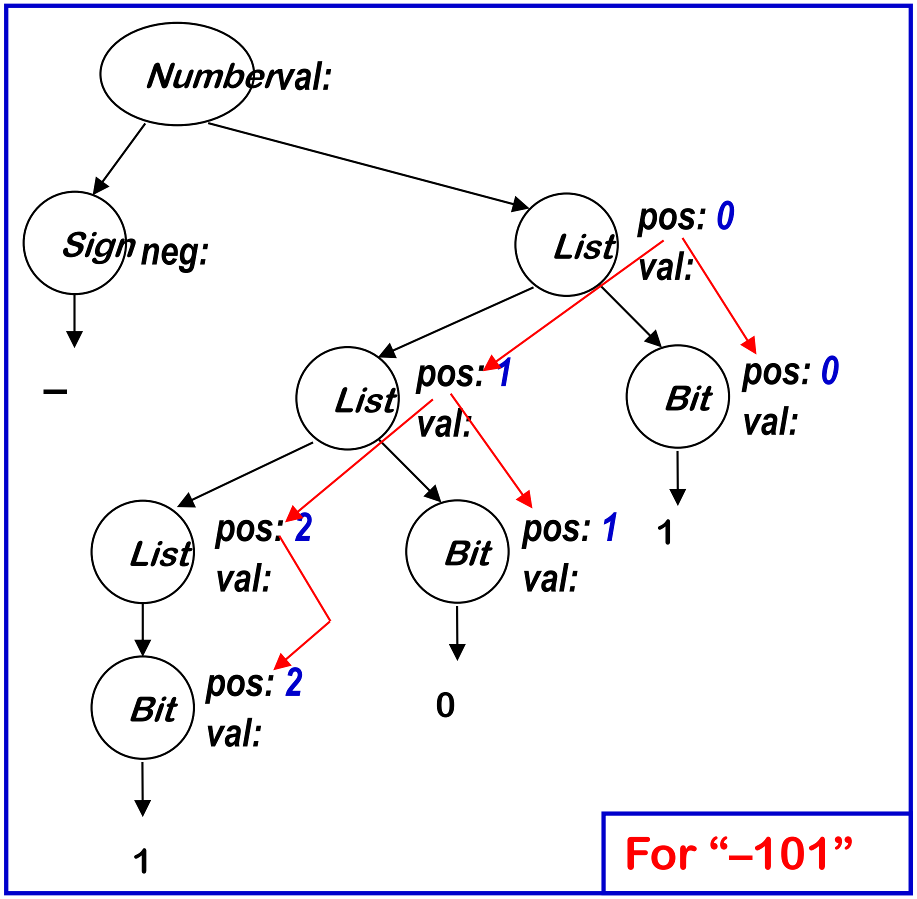
Example parse tree

compute the decimal value of a signed binary number



Example parse tree

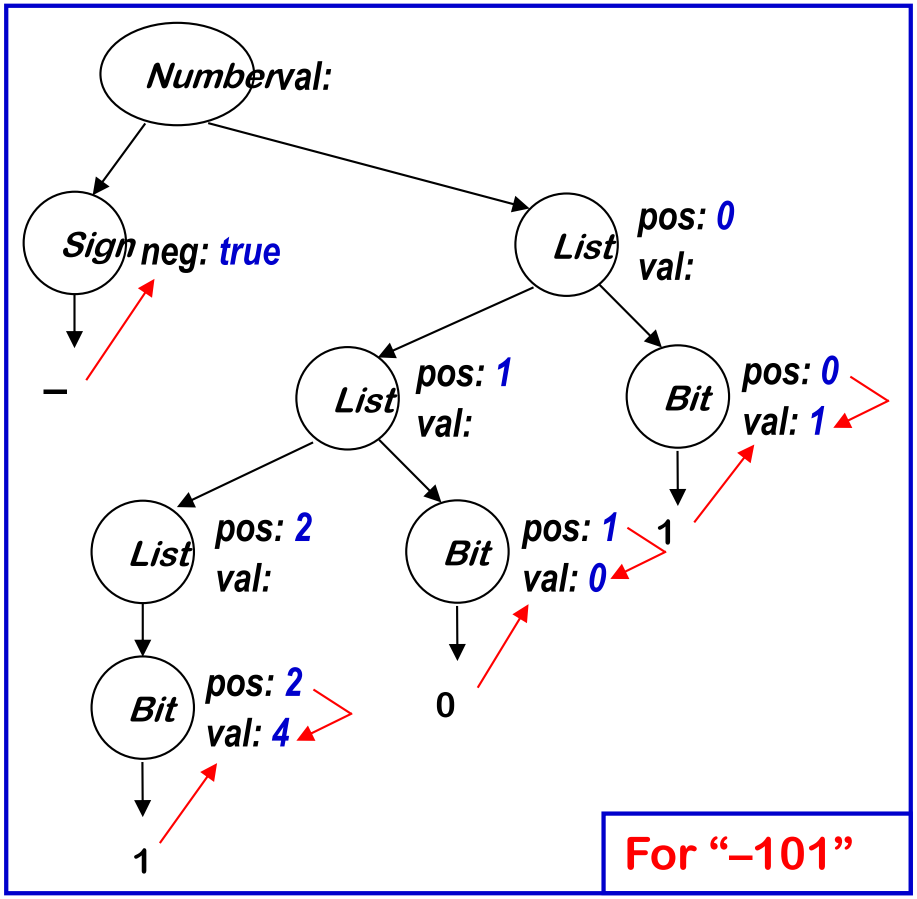
compute the decimal value of a signed binary number



Inherited Attributes

Example parse tree

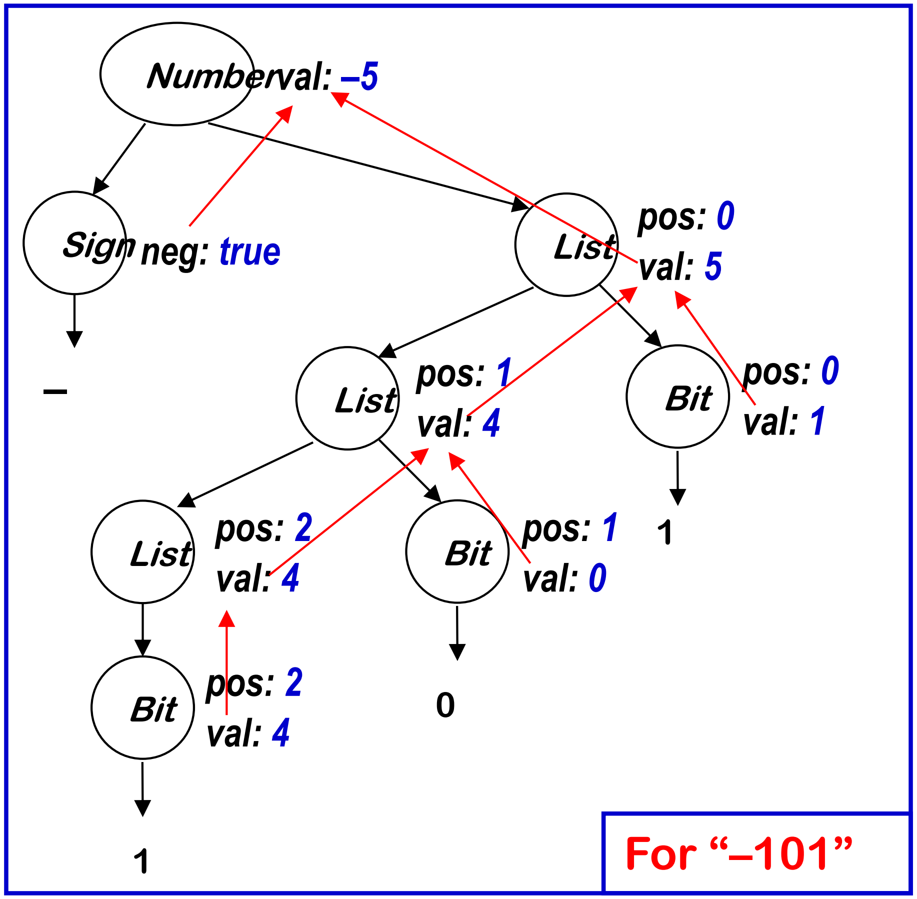
compute the decimal value of a signed binary number



Synthesized attributes

Example parse tree

compute the decimal value of a signed binary number



Synthesized attributes

Attribute Grammars

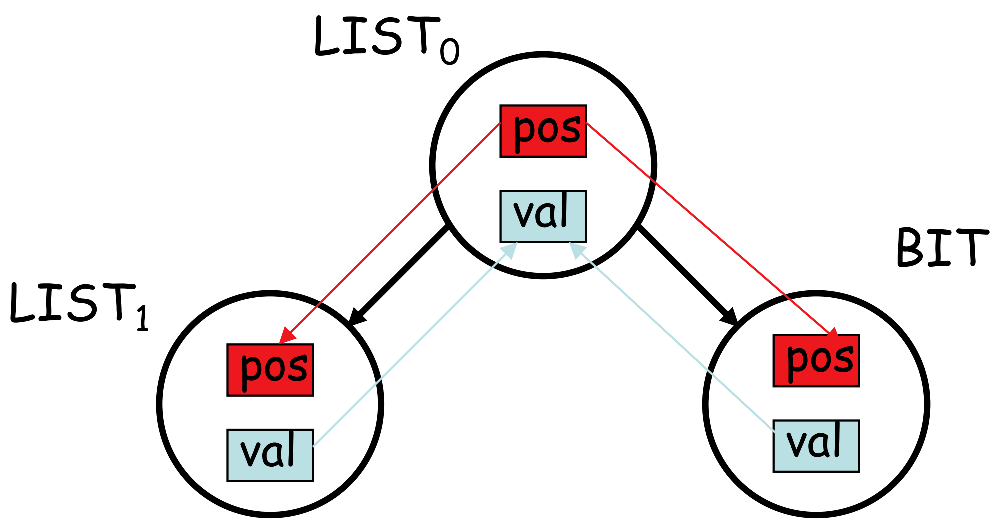
Add rules to compute the decimal value of a signed binary number

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*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos* $\leftarrow$ *List.pos* *List.val* $\leftarrow$ *Bit.val*
Bit	$\rightarrow$	0	*Bit.val* $\leftarrow$ *0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

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

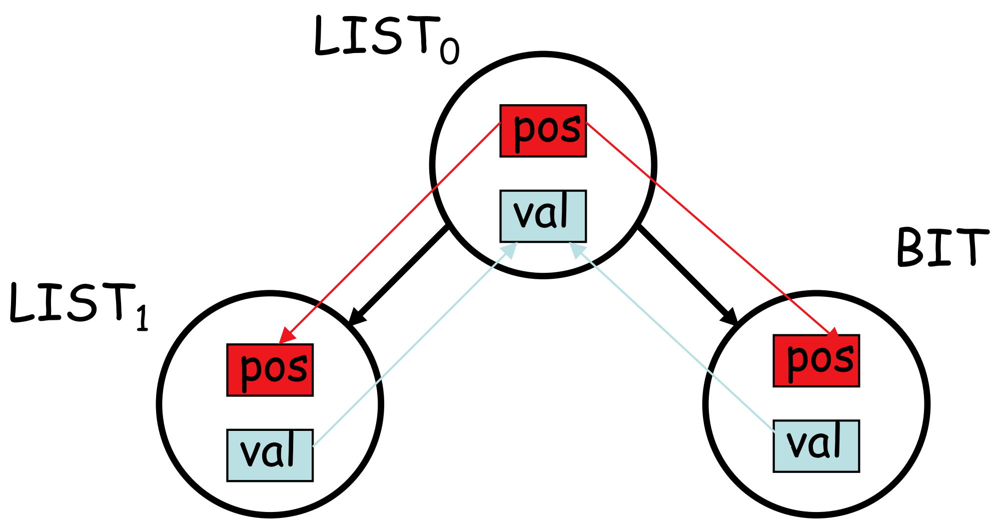
Attribute Grammars

Productions			**Attribution Rules**
$List_0$	$\rightarrow$	***$List_1$ Bit***	*$List_1$.pos $\leftarrow$ $List_0$.pos + 1* *Bit.pos $\leftarrow$ $List_0$.pos* *$List_0$.val $\leftarrow$ $List_1$.val + Bit.val*



- semantic rules define partial dependency graph
- value flow top down or across: inherited attributes
- value flow bottom-up: synthesized attributes

Attribute Grammars



Note:

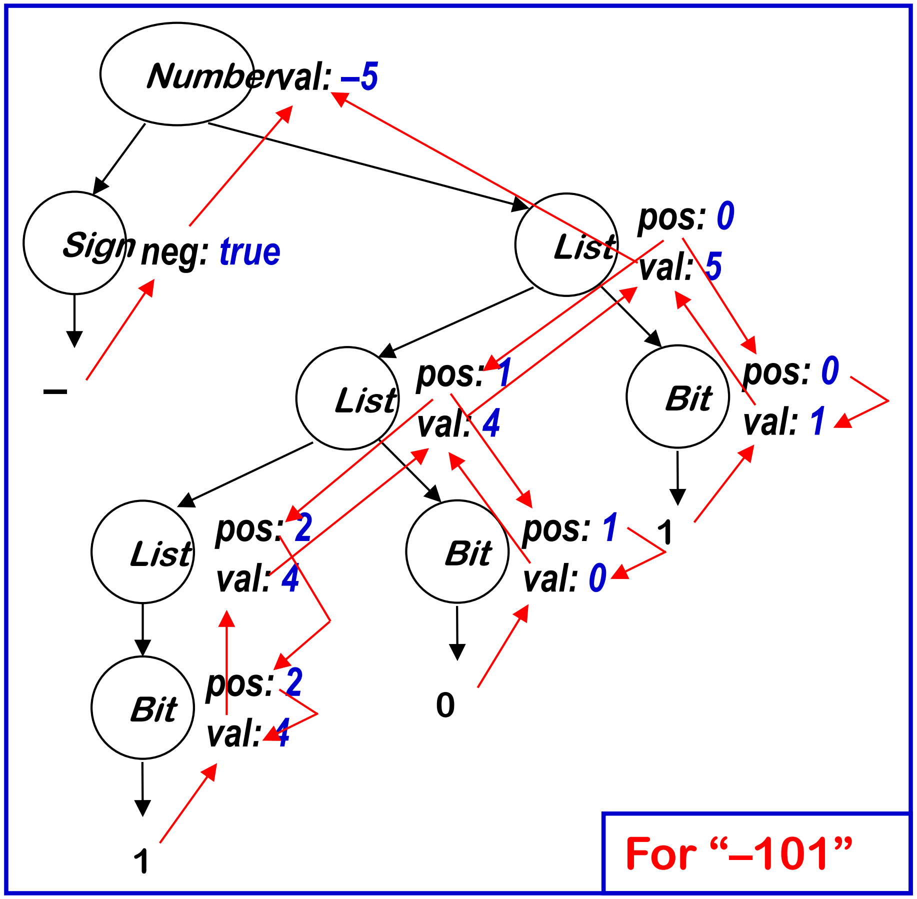
- semantic rules associated with production $A \rightarrow \alpha$ have to specify the values for all
  - synthesized attributes for A (root)
  - inherited attributes for grammar symbols in α (children)

  $\Rightarrow$ rules must specify **local value flow!**

- terminals can be associated with values returned by the scanner. These input values are associated with a synthesized attribute.
- Starting symbol cannot have inherited attributes.

Example revisited

compute the decimal value of a signed binary number

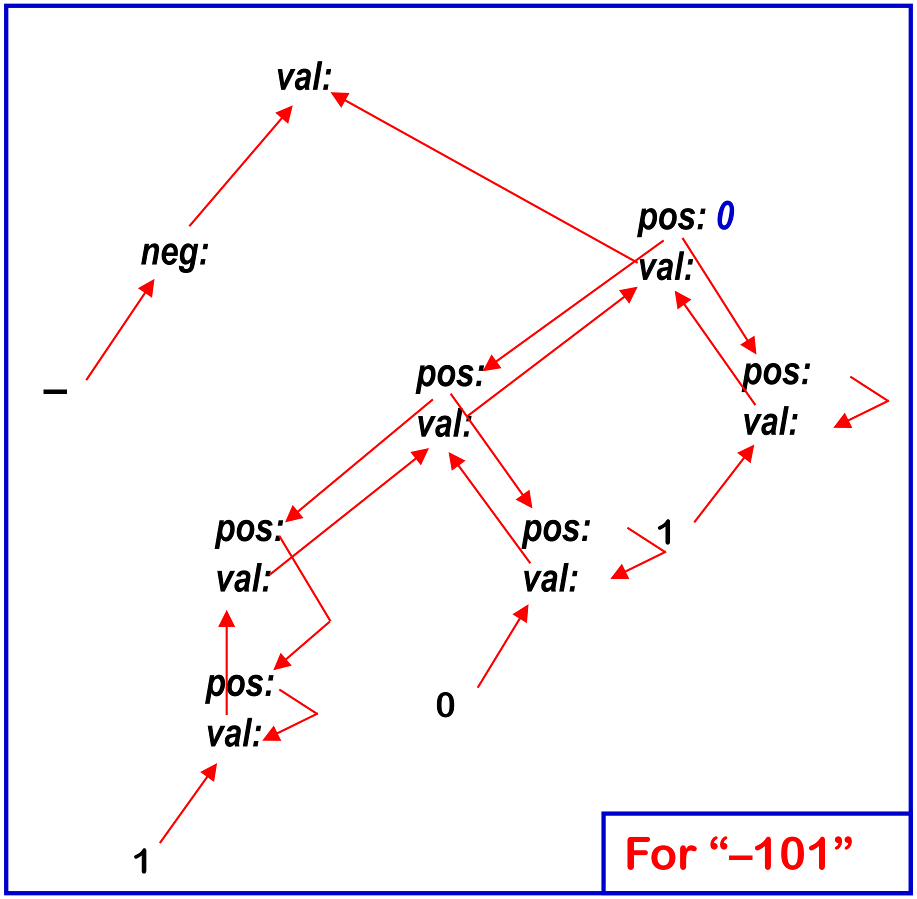


If we show the computation ...

& then peel away the parse tree ...

Example revisited

compute the decimal value of a signed binary number



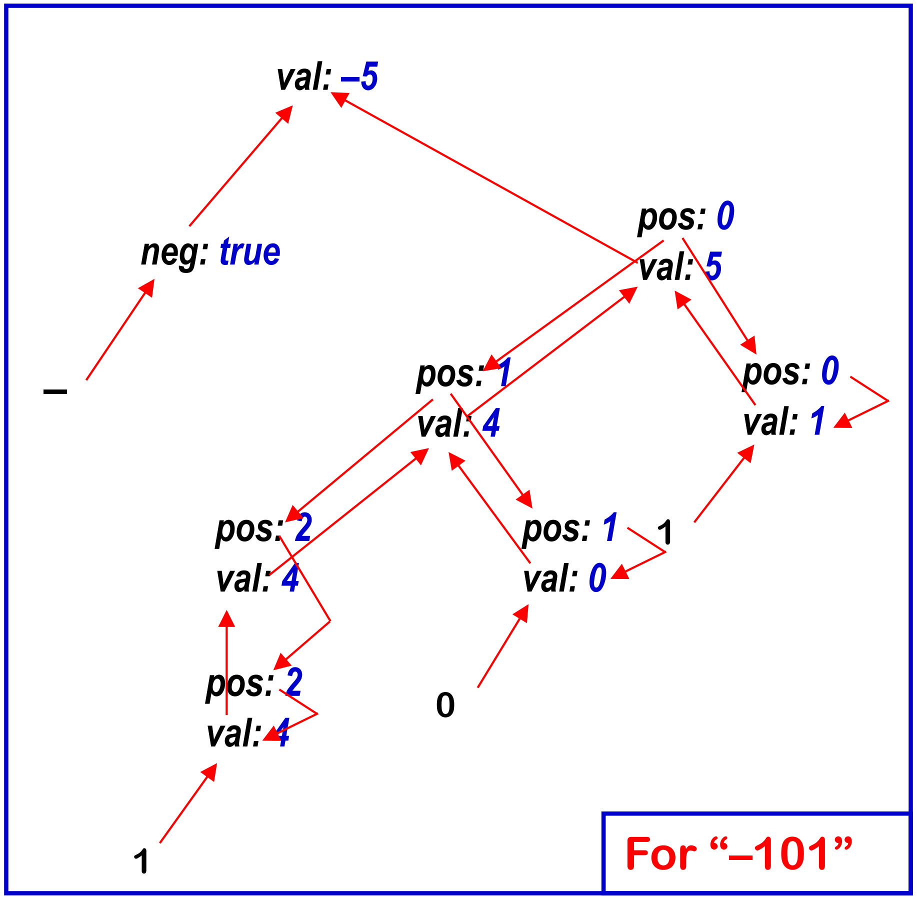
All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The dependence graph must be acyclic

Example revisited

compute the decimal value of a signed binary number



All that is left is the attribute dependence graph.

This succinctly represents the flow of values in the problem instance.

The dynamic methods topologically sort this graph, then evaluates edges/nodes in that order

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

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: synthesized attributes only
- Evaluate in a single bottom-up pass

Good match to LR parsing

$S\text{-attributed} \subset L\text{-attributed}$

Inherited Attributes

- Use values from parent, constants, & siblings
- L-attributed grammars:

$A \rightarrow X_1 X_2 \dots X_n$ and each inherited attribute of X_i depends on

- attributes of $X_1 X_2 \dots X_{i-1}$, and
- inherited attributes of A

- Evaluate in a single top-down pass (left to right)

Good match for LL parsing

Next class

More syntax-directed translation

Type checking

Symbol tables

Intermediate representations

Read EaC: Chapters 5.1 – 5.3