RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Context-Sensitive Analysis Part 2

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 intro video (29 minutes) available on canvas: My Media
- Sample solution for Homework #3 has been posted
- Second midterm on Wednesday, April 6 (60 minutes in class)
- Final exam on May 10 (60 minutes at assigned location)

Midterm #2

Topics

- Regular expressions
- NFA and DFA
- Regular expressions to minimal DFA construction
- CFG
  - → Derivations
  - → Parse trees
  - → Ambiguity
- LL(1) parsing
  - → FIRST and FOLLOW sets
  - → Parse tables
  - → Recursive descent parsers
- LR(0) parsing
  - → LR(0) items
  - → LR(0) canonical collection and its construction
  - → ACTION and GOTO tables
  - → Shift/reduce and reduce/reduce conflicts

Attribute Grammars

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

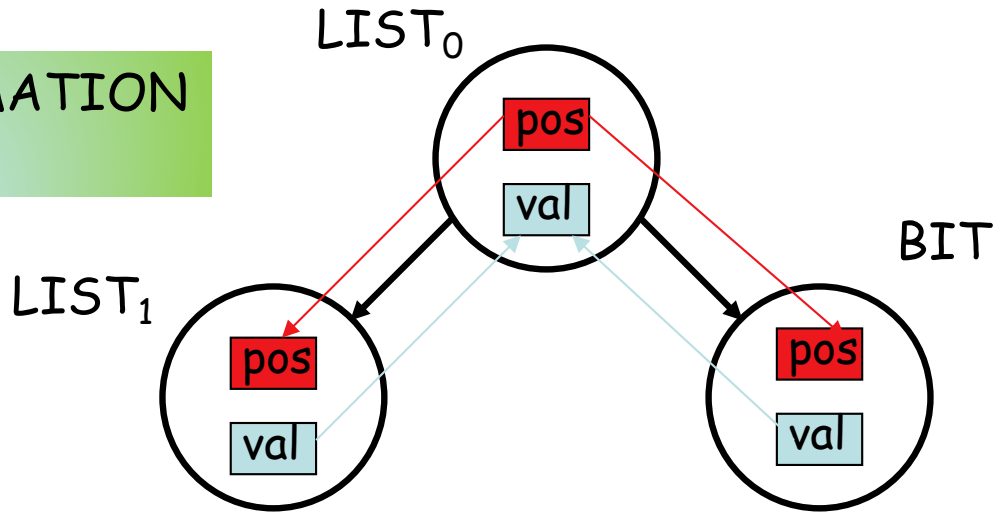
Productions			Attribution Rules
Number	→	*Sign List*	*List.pos ← 0* *If Sign.neg* *then Number.val ← – List.val* *else Number.val ← List.val*
Sign	→	+	*Sign.neg ← false*
	|	–	*Sign.neg ← true*
$List_0$	→	$List_1$ *Bit*	$List_1.pos \leftarrow List_0.pos + 1$ $Bit.pos \leftarrow List_0.pos$ $List_0.val \leftarrow List_1.val + Bit.val$
	|	*Bit*	*Bit.pos ← List.pos* *List.val ← Bit.val*
Bit	→	0	*Bit.val ← 0*
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

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

Attribute Grammars

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

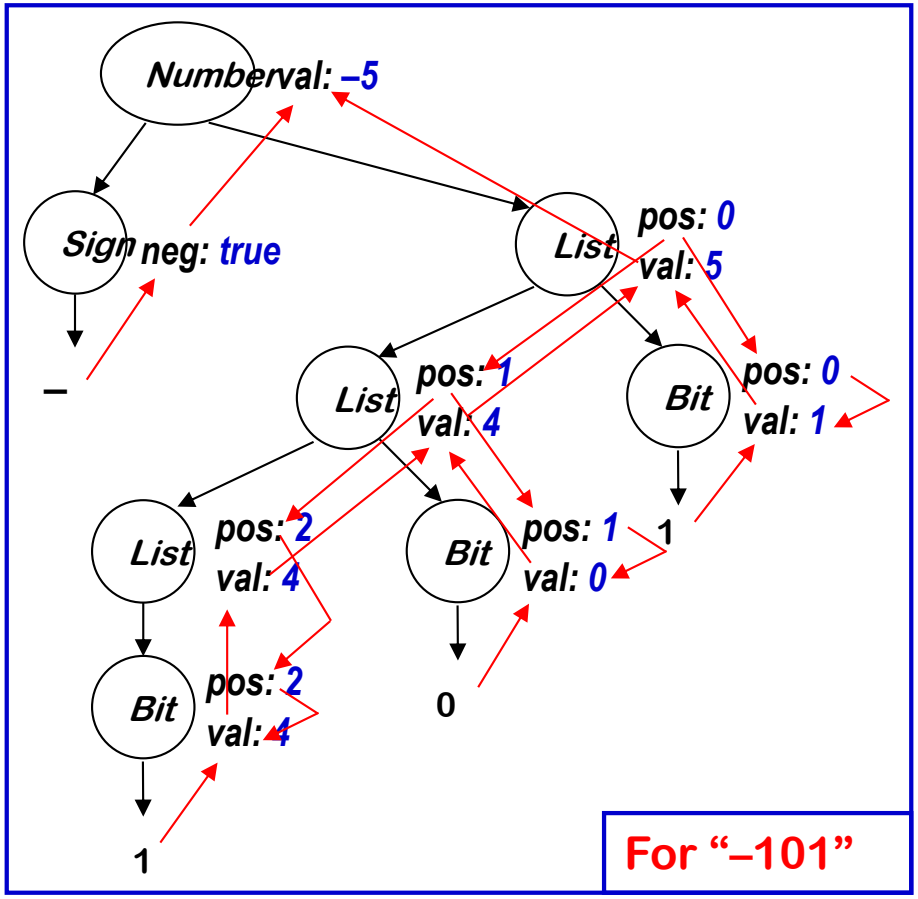
LOCAL INFORMATION FLOW



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

Example revisited

compute the decimal value of a signed binary number

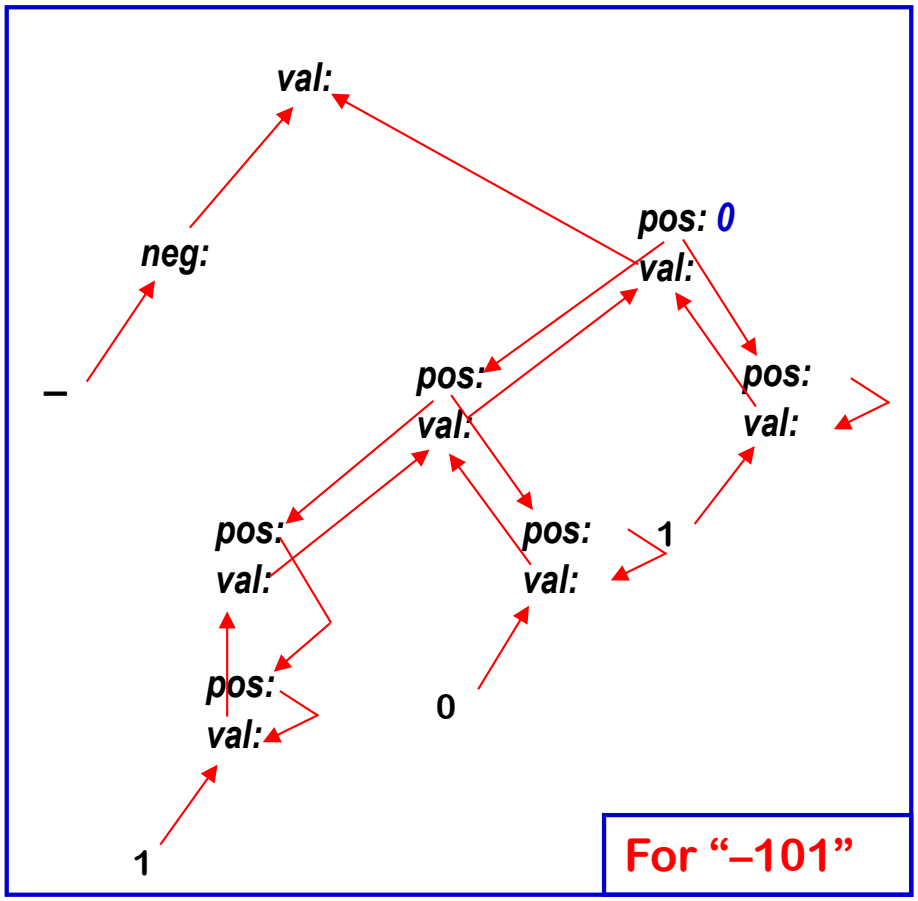


If we show the computation …

& then peel away the parse tree …

Example revisited

compute the decimal value of a signed binary number



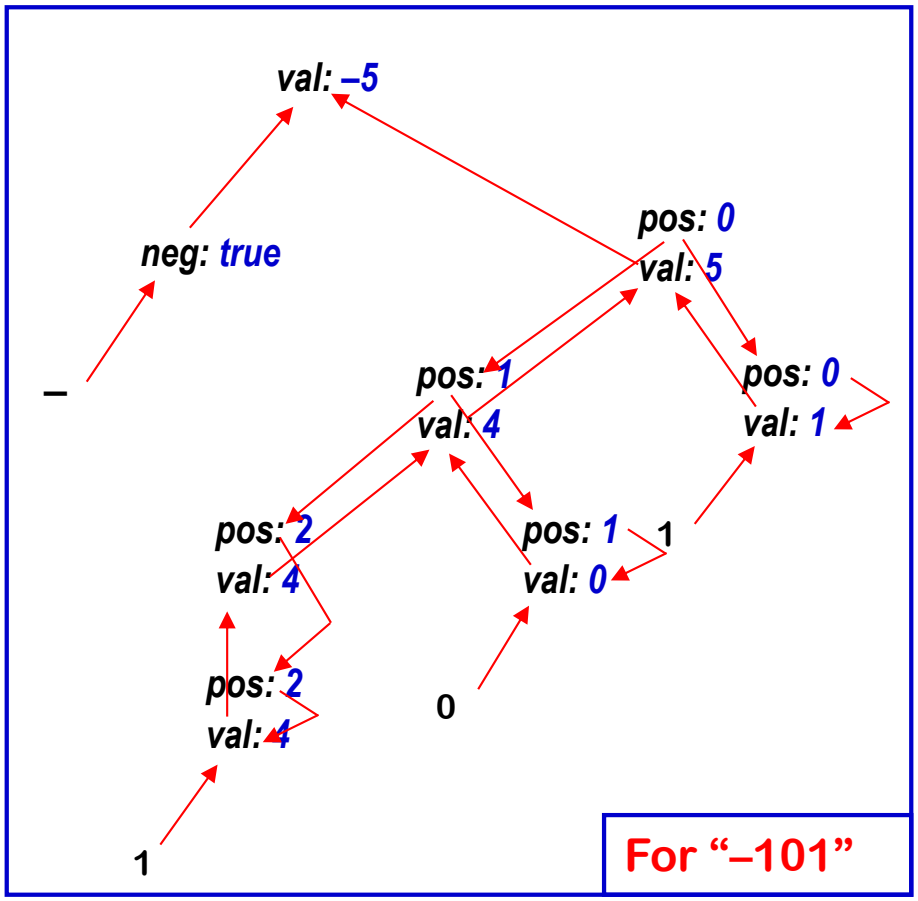
All that is left is the attribute dependence graph.

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-attributed ⊂ L-attributed

Inherited Attributes

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

$A \rightarrow X_1 X_2 \ldots X_n$ and each inherited attribute of X_i depends on
  - attributes of $X_1 X_2 \ldots X_{i-1}$, and
  - inherited attributes of A
- Evaluate in a single top-down pass (left to right)

Good match for LL parsing

Attribute Grammars

- Non-local computation needed lots of supporting rules
- "Complex" local computation is relatively easy

The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - → Need copies of attributes
- Result is an attributed tree
  - → Must build the parse tree
  - → Either search tree for answers or copy them to the root

Addressing the Problem

What would a good programmer do?

- Introduce a central repository for facts
- Table of names
  - → Fields in table keep information for names
- Avoids all the copy rules, allocation & storage headaches
- All inter-assignment attribute flow is through table
  - → Clean, efficient implementation
  - → Good techniques for implementing the table *(hashing, § B.4)*
  - → When its done, information is in the table !
  - → Cures most of the problems
- Unfortunately, this design violates the functional, AG paradigm
  - → Do we care?

The Realist's Alternative

Ad-hoc syntax-directed translation

- Associate pieces of code with each production
- At each reduction, the corresponding code is executed
- 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, …)
  - → Yacc introduced **$$**, **$1**, **$2**, … **$n**, left to right
- Need an evaluation scheme
  - → Fits nicely into LR(1) parsing algorithm

Project #2 (see "lex & yacc", Levine et al., O'Reilly)

→ You do not have to change the scanner (scan.l)

→ How to specify and use attributes in YACC?

- Define attributes as types in attr.h

  typedef struct info_node {int a; int b} infonode;

- Include type attribute name in %union in parse.y

  %union {tokentype token; infonode myinfo; ... }

- Assign attributes in parse.y to
  - Terminals: *%token <token> ID ICONST*
  - Non-terminals: *%type <myinfo> block variables procdecls cmpdstmt*

YACC : parse.y

parse.y :

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

%union {tokentype token;
        regInfo targetReg;
       }

%token PROG PERIOD VAR
%token INT BOOL PRT THEN IF DO FI ENDWHILE ENDFOR
%token ARRAY OF
%token BEG END ASG
%token EQ NEQ LT LEQ GT GEQ AND OR TRUE FALSE
%token WHILE FOR ELSE
%token <token> ID ICONST

%type <targetReg> exp
%type <targetReg> lhs

%start program

%nonassoc EQ NEQ LT LEQ GT GEQ
%left '+' '-' AND
%left '*' OR

%nonassoc THEN
%nonassoc ELSE
```

List and assign attributes

attr.h :

```
typedef union {int num; char *str;} tokentype;

typedef enum type_expression {TYPE_INT=0,
        TYPE_BOOL, TYPE_ERROR} Type_Expression;

typedef struct {
        Type_Expression type;
        int targetRegister;
        } regInfo;
```

Disambiguation rules

Project #2 (see "lex & yacc", Levine et al., O'Reilly)

At each reduction, the corresponding code is executed.

→ Accessing attribute values in parse.y

- use $$, $1, $2 … etc. notation:

block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt

{ $$.a = $1.a + $2.a + $4.b;}

- Implemented as

block : variables procdecls newsymbol cmpdstmt

{ $$.a = $1.a + $2.a + $4.b;}

newsymbol: ϵ {$2.b = $1.b + 1;}

YACC : parse.y

parse.y :

CFG rules with embedded actions

```
%%
program : {emitComment("Assign STATIC_AREA_ADDRESS to register \"r0\"");
           emit(NOLABEL, LOADI, STATIC_AREA_ADDRESS, 0, EMPTY);}
           PROG ID ';' block PERIOD { }
     ;

block: variables cmpdstmt { }
     ;

variables: /* empty */
     | VAR vardcls { }
     ;

vardcls: vardcls vardcl ';' { }
     | vardcl ';' { }
     | error ';' { yyerror("***Error: illegal variable declaration\n");}
     ;

. . .

exp  : exp '+' exp {
     int newReg = NextRegister();
     if (!(($1.type == TYPE_INT) && ($3.type == TYPE_INT))) {
         printf("*** ERROR ***: Operator types must be integer.\n");}
     $$.type = $1.type;
     $$.targetRegister = newReg;
         emit(NOLABEL, ADD, $1.targetRegister, $3.targetRegister, newReg);

. . .
```

Summary: Is This Really "Ad-hoc" ?

Example on ilab: ~uli/cs415/examples/LexYacc

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools
- (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 (purely) functional
- AG rules are higher level than *ad-hoc* actions

Types and Type Systems

Type: A set of values and meaningful operations on them

Types provide semantic "sanity checks" (consistency checks) and determine efficient implementations for data objects

Types help identify

- → errors, if an operator is applied to an incompatible operand
  - dereferencing of a non-pointer
  - adding a function to something
  - incorrect number of parameters to a procedure
  - ...
- → which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)
- → identification of polymorphic functions

Types and Type Systems

Type system: Each language construct (operator, expression, statement, …) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

Type expressions for

- → basic types: `integer, char, real, boolean, typeError`
- → constructed types, e.g., one-dimensional arrays:
  `array(lb, ub,` elem_type`)` , where elem_type is a type expression

A type checker implements a type system. It computes or "constructs" type expressions for each language construct.

Types and Type Systems

Example type inference rule:

$$\frac{E \vdash e_1 : \text{integer} \quad , \quad E \vdash e_2 : \text{integer}}{E \vdash e_1 + e_2 : \text{integer}}$$

where E is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

3.0 + 1 or 1 + 3.0

Things to do and next class

Work on the project!

Type systems

Code generation (EaC Chapter 7)

Optimization: CSE