Context-sensitive analysis

What context-sensitive questions might the compiler ask?

1. Is x a scalar, an array, or a function?
2. Is x declared before it is used?
3. Are any names declared but not used?
4. Which declaration of x does this reference?
5. Is an expression type-consistent?
6. Does the dimension of a reference match the declaration?
7. Where can x be stored? (heap, stack,…)
8.
9. Does *p reference the result of a malloc()?
10. Is x defined before it is used?
11. Is an array reference in bounds?
12. Does function foo produce a constant value?

These cannot be answered with a context-free grammar

Context-sensitive analysis

Why is context-sensitive analysis hard?

- need non-local information
- answers depend on values, not on syntax
- answers may involve computation

How can we answer these questions?

1. write context-sensitive grammars and parse
   A. general problem is P-space compete
   B. haven't found useful subclass
2. use ad hoc techniques
   A. symbol tables and code
   B. yacc "action routines"
3. formal methods
   A. syntax-directed translation (attr. grammars)
   B. type systems and checking algorithms

Attribute grammars

Idea: attribute the tree

- can add attributes(fields) to each node
- specify equations to define values(unique)
- both inherited and synthesized attributes

Example

To ensure that constants are immutable:

- add type and class attributes to expression nodes
- rules for production on “:=” that
  1. checks that lhs.class is varible
  2. checks that lhs.type and rhs.type are consistent or conformable

Attribute grammars

To formalize such systems, D. Knuth introduced attribute grammars.

- grammar-based specification of tree attributes
- value assignments associated with productions
- each attribute uniquely defined
- label identical terms uniquely

Attribute grammars can be used to specify context-sensitive actions

Example

Production	Evaluation Rules
DECL := TYPE LIST	LIST.in ← TYPE.type
TYPE := int	TYPE.type ← integer
TYPE := char	TYPE.type ← char
$LIST_0$:= $LIST_1$, id	id.type ← $LIST_0$.in
	$LIST_1$.in ← $LIST_0$.in
LIST := id	id.type ← LIST.in

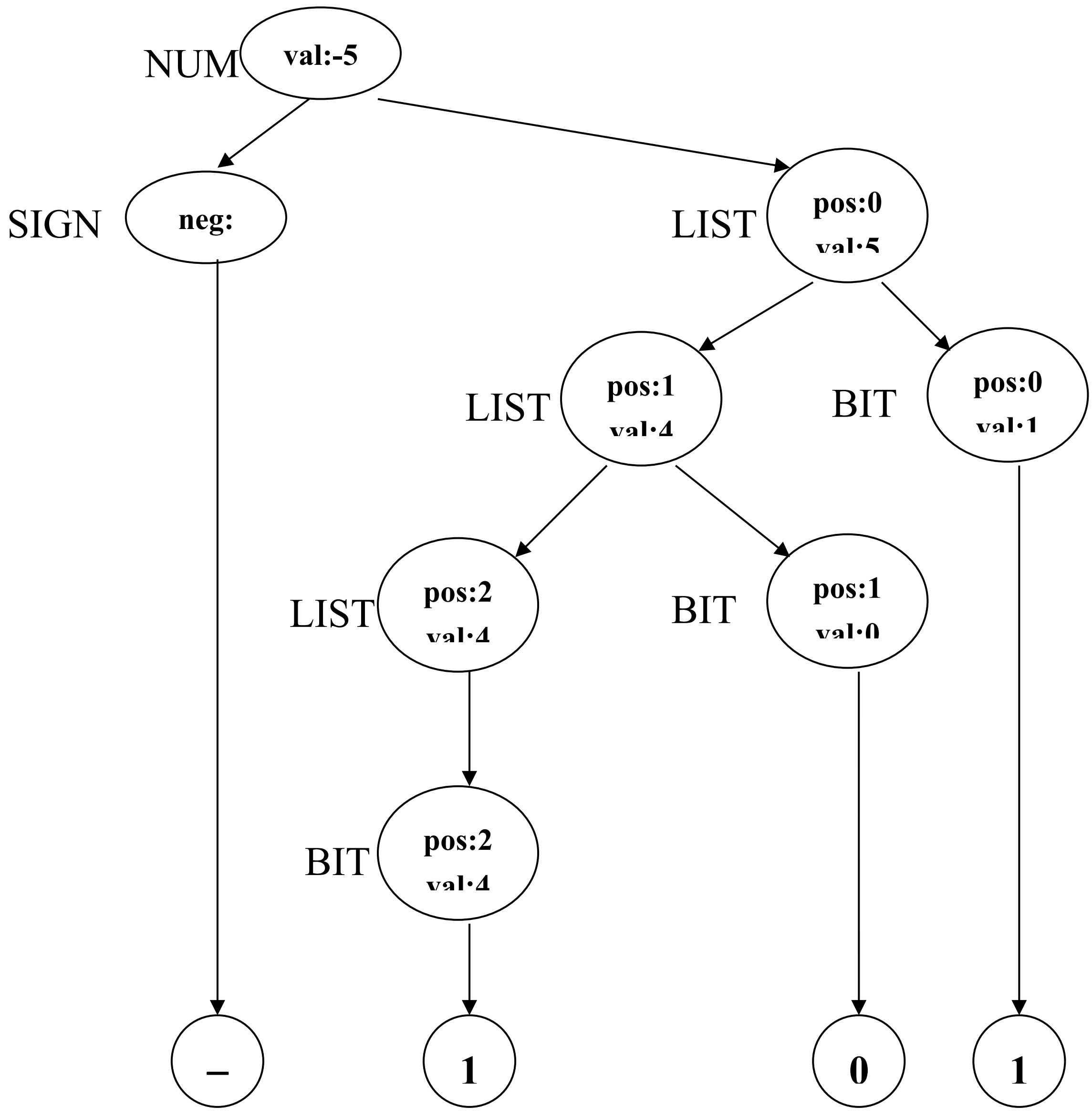
Example attribute grammar

A grammar to evaluate signed binary numbers
due to Scott K. Warren, Rice Ph. D.

Production	Evaluation Rules
1 NUM ::= SIGN LIST	LIST.pos $\leftarrow$ 0
	NUM.val $\leftarrow$ if SIGN.neg
	then –LIST.val
	else LIST.val
2 SIGN ::= +	SIGN.neg $\leftarrow$ false
3 SIGN ::= -	SIGN.neg $\leftarrow$ true
4 LIST ::= BIT	BIT.pos $\leftarrow$ LIST.pos
	LIST.val $\leftarrow$ BIT.val
5 $LIST_0$ ::= $LIST_1$ BIT	$LIST_1$.pos $\leftarrow$ $LIST_0$.pos + 1
	BIT.pos $\leftarrow$ $LIST_0$.pos
	$LIST_0$.val $\leftarrow$ $LIST_1$.val + BIT.val
6 BIT ::= 0	BIT.val $\leftarrow$ 0
7 BIT ::= 1	BIT.val $\leftarrow$ $2^{BIT.pos}$

Attribute grammars

Example



- val and neg are synthesized attributes
- pos is an inherited attributes

Attribute grammars

Aho, Sethi, & Ullman describe
Syntax-directed definitions. These are just
Attribute grammars by another name.

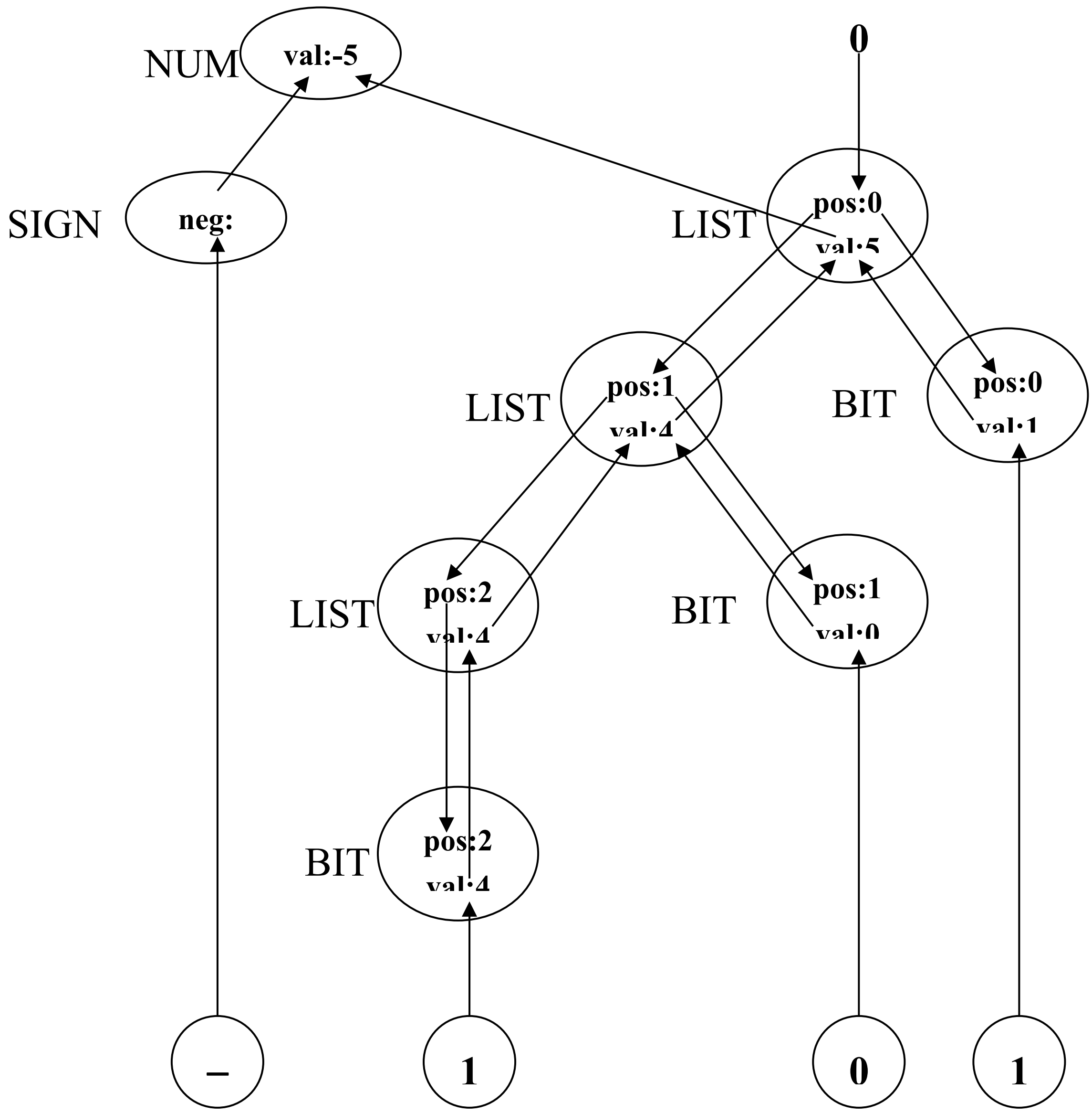
Attribute grammar

- generalization of context-free grammar
- each grammar symbol has an associated set of attributes
- augment grammar with rules that define values
- high-level specification, independent of evaluation scheme

Dependences between attributes

- values are computed from constants & other attributes
- synthesized attributes – value computed from children
- inherited attribute – value computed from siblings & parent
- key notion: induced dependency graph

Example


NUM
val:-5
0
SIGN
neg:
LIST
pos:0
val:5
LIST
pos:1
val:4
BIT
pos:0
val:1
LIST
pos:2
val:4
BIT
pos:1
val:0
BIT
pos:2
val:4
–
1
0
1

Attribute types

Synthesized attributes

derives its value from constants and children

- only synthesized attributes => S-attributed grammar
- S-attributed grammars can be evaluated in one bottom-up pass
- useful in many contexts(calculator,..)

S-attributed grammar is good match to LR parsing.

Inherited attributes

derives its value from constants, siblings, and parent

- used to express context(context-sensitive)
- can always rewrite to avoid inherited attributes
- inherited attributes rules are “more natural”

Mechanical translation of inherited attributes is more problematic.

We want to use both kinds of attribute.

Attribute types

The attribute dependency graph

- nodes represent attributes
- edges represent the flow of values
- graph is specific to parse tree
- size is related to parse tree's size
- can be built alongside parse tree

The dependency graph must be acyclic

Evaluation order

- topological sort the dependency graph to order attributes
- using this order, evaluate the rules

This order depends on both the grammar and the input string.

Evaluation methods

Parse-tree methods(dynamic)

1. build the parse tree
2. build the dependency graph
3. topological sort the graph
4. evaluate it(cyclic graph fails)

Rule-based methods(treewalk)

1. analyze rules at compiler-generation time
2. determine a static ordering at that time
3. evaluate nodes in that order at compile time

Oblivious methods(passes, dataflow)

1. ignore the parse tree and grammar
2. choose a convenient order and use it
   (forward-backward passes, alternating passes)

Problems

- circularity
- best evaluation strategy is grammar dependent

Strongly Non-Circular grammars

Idea: can evaluate each instance of a node in the same order

Implementation: use order to build recursive evaluator

Circularity testing

- SNC grammars can be tested in polynomial time

Parse-tree evaluators discover circularity dynamically.

Problems

- Space – both magnitude & management
- Copy rules – time & space
- Parse-tree evaluators – need dependency graph
- Rule-based evaluators – good compromise
- Oblivious evaluators – need graph, inefficient

These systems have seen limited practical use

Applications

- Cornell Program Synthesizer(& Generator)
- Someone did a VHDL compiler
- Structure editors for code, theorems,…

Attribute grammars

Advantages

- Clean formalism
- Automatic generation of evaluator
- High-level specification

Disadvantages

- evaluation strategy determines efficiency
- increased space requirements
- results distributed over tree
- circularity testing

Historically, attribute grammar evaluators have been deemed too large for commercial-quality compilers.