

Where are we?

Aho, Sethi, and Ullman:

Chapters 1, & 2

good background material

Chapter 3

lexical analysis (*scanning*)

Chapter 4

syntax analysis (*parsing*)

1. sec 4.1, 4.2, 4.3, — background
2. sec 4.4 — top-down parsing
3. sec 4.6 — operator-precedence
4. sec 4.5, 4.7 — LR parsing
5. sec 4.8 — ambiguous grammars
6. sec 4.9 — parser generators

Chapters 5 & 6

context-sensitive analysis

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. Does $*p$ reference the result of a `malloc()`?
9. Is x defined before it is used?
10. Is an array reference *in bounds*?
11. 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 complete
 - (b) haven't found useful subclass
(ndcfgs are $O(n^{2.81})$)
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 `variable`
 2. checks that `lhs.type` and `rhs.type` are consistent or conformable

See chapter 6, Aho, Sethi, and Ullman

Attribute grammars

To formalize such systems, 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 \leftarrow TYPE.TYPE
TYPE := int	TYPE.TYPE \leftarrow integer
TYPE := char	TYPE.TYPE \leftarrow char
LIST ₀ := LIST ₁ , id	addTYPE(id.entry, LIST ₀ .in) LIST ₁ .in \leftarrow LIST ₀ .in
LIST := id	addTYPE(id.entry, LIST.in)

Example attribute grammar

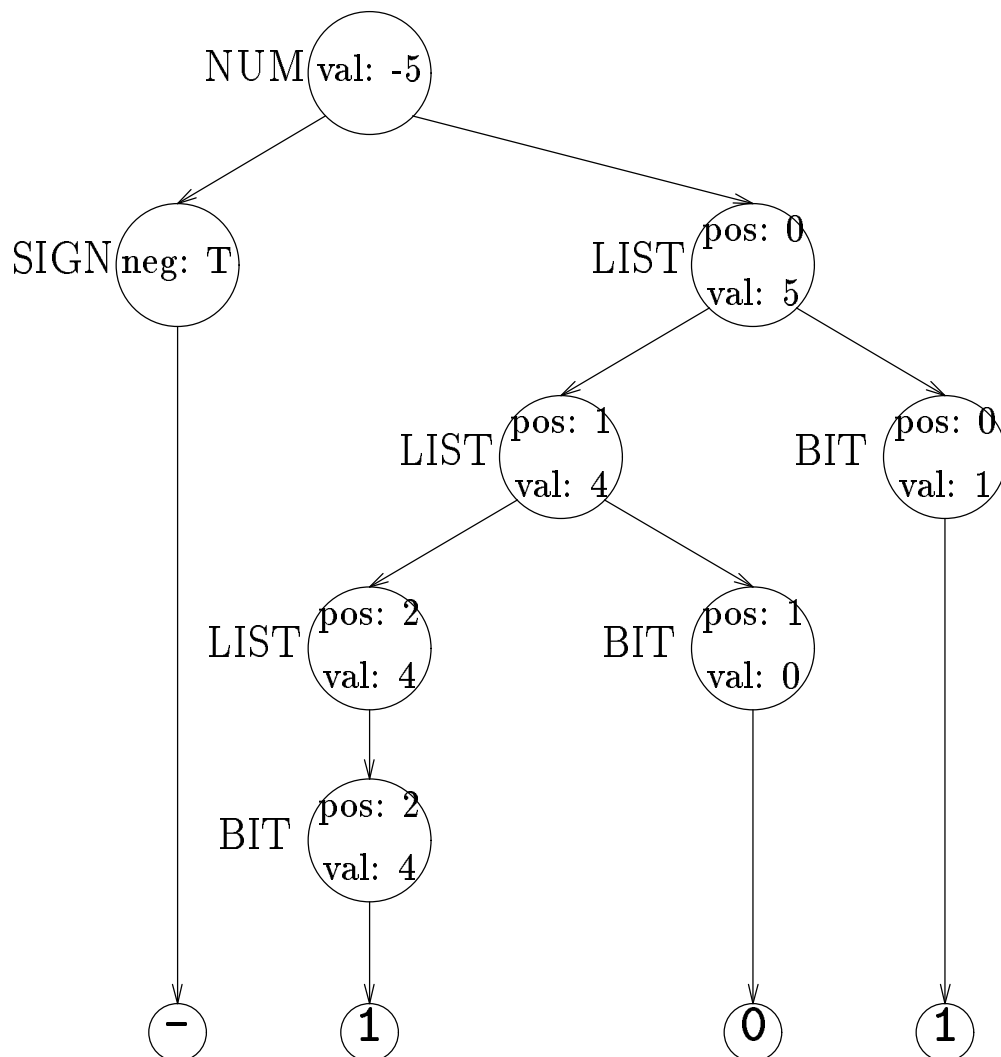
A grammar to evaluate signed binary numbers

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 ^{B.pos}

Attribute grammars

Example



- `val` and `neg` are *synthesized* attributes
- `pos` is an *inherited* attribute

Attribute grammars

Aho, Sethi, & Ullman describe *syntax-directed definitions*. These are just *attribute grammars* by another name.

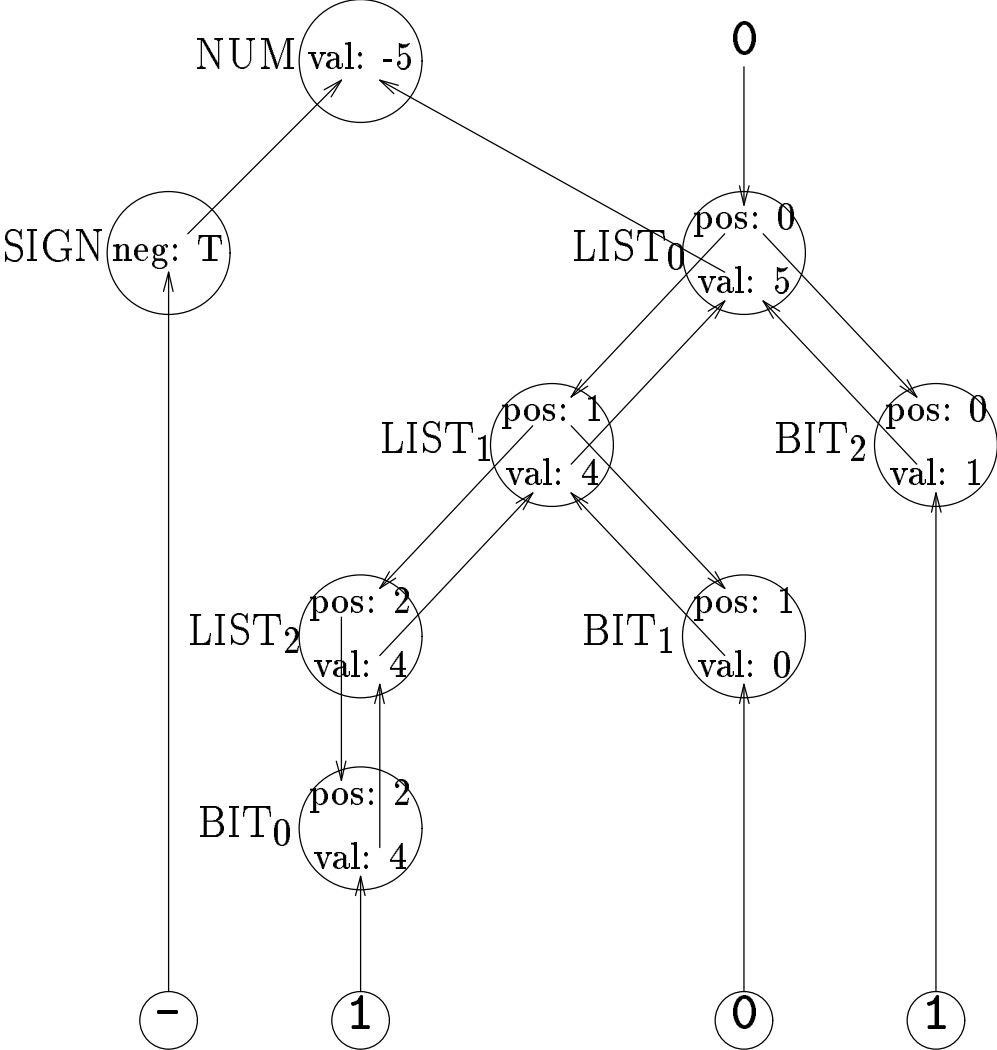
Attribute grammar

- 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 attribute* – value computed from children
- *inherited attribute* – value computed from siblings & parent
- *key notion*: induced dependency graph

Example grammar evaluation



Attribute types

Synthesized attributes

derives its value from constants and children

- only synthesized attributes \Rightarrow S-attributed grammar
- S-attributed grammars can be evaluated in one bottom-up pass
- useful in many contexts (*calculator, AS&U 281*)

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 attribute rules are “more natural”

Mechanical translation of inherited attributes is more problematic

We want to use both kinds of attribute

Attribute grammars

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

(*AS&U's taxonomy*)

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

Attribute grammars

A topological order for the example

1. SIGN.neg
2. LIST₀.pos
3. LIST₁.pos
4. LIST₂.pos
5. BIT₀.pos
6. BIT₁.pos
7. BIT₂.pos
8. BIT₀.val
9. LIST₂.val
10. BIT₁.val
11. LIST₁.val
12. BIT₂.val
13. LIST₀.val
14. NUM.val

Evaluate in this order

Yields NUM.val: -5

Build a recursive treewalk evaluator for each node type

Attribute grammars

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

- \exists grammar $g \ni$ testing g takes exponential time
- SNC grammars can be tested in polynomial time
- failing the SNC test isn't conclusive

Parse-tree evaluators discover circularity dynamically.

See §5.10 of Aho, Sethi, & Ullman

Attribute grammars

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

- Intel's 80286 Pascal system
- Cornell Program Synthesizer (& Generator)
- someone did a VHDL compiler
- Structure editors for code, theorems, ...

And, of course, Ph.D. theses and papers

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

Intel built a Pascal compiler that used an attribute grammar evaluator to perform context-sensitive analysis.

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