Syntactic Directed Translation

Attribute Grammars
&
Syntax-Directed Definitions

What is Syntax-Directed Translation?

- Translation Process guided by Context-Free Grammars
  - Attach *Attributes* to *Grammar Symbols*
  - *Values* and *Semantic Rules* Associated with *Productions*
  - *Attributive Grammar*
- Two Flavors:
  - Syntax-Directed Definitions (order is implicit; more abstract)
  - Translation Schemes (explicit order; more concrete)
- Why Use This ? Very Powerful Mechanism
  - Used to Build Parse Tree
  - Perform Semantic Analysis such as Type Checking
  - Generate Code

Attribute Grammars

What is an Attribute Grammar?

- A Context-Free Grammar Augmented with a Set of Rules
- Each Symbol in the derivation has a set of values, or *Attributes*
- The Rules specify How to Compute a value for each Attribute

Example Grammar

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

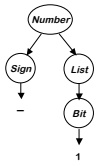
This grammar describes signed binary numbers

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

Examples

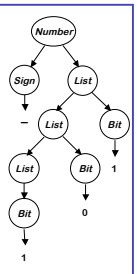
For "–1"

Number → *Sign List*
→ *– List*
→ *– Bit*
→ *– 1*



For "–101"

Number → *Sign List*
→ *Sign List Bit*
→ *Sign List 1*
→ *Sign List Bit 1*
→ *Sign List 1 1*
→ *Sign Bit 0 1*
→ *Sign 1 0 1*
→ *– 101*



Attribute Grammars

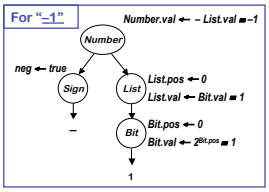
Add rules to compute the decimal value of a signed binary number

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

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

Back to the Examples

For "–1"



One possible evaluation order:

1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

Other orders are possible

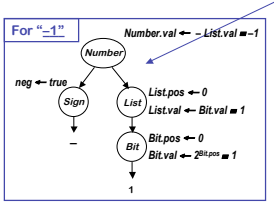
Knuth suggested a data-flow model for Evaluation

- Independent Attributes First
- Others in Order as Input Values become Available

Back to the Examples

Rules + parse tree imply an attribute dependence graph

For "–1"



One possible evaluation order:

1 List.pos
2 Sign.neg
3 Bit.pos
4 Bit.val
5 List.val
6 Number.val

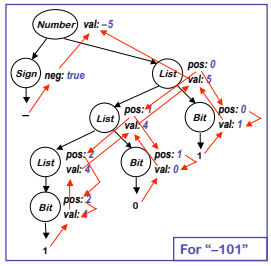
Other orders are possible

Evaluation order must be consistent with the attribute dependence graph

Knuth suggested a data-flow model for Evaluation

- Independent Attributes First
- Others in Order as Input Values become Available

Back to the Examples



This is the complete attribute dependence graph for "–101".

It shows the flow of *all* attribute values in the example.

Some flow downward
→ ***Inherited Attributes***

Some flow upward
→ ***Synthesized Attributes***

A rule may use attributes in the parent, children, or siblings of a node

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

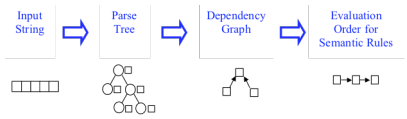
Good match to LR parsing

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

Attributes, Rules and Evaluation Order

- Attributes:
  - Synthesized: Depend on Values from Children or Itself
  - Inherited: Depend on Values from Parent, Siblings or Itself
- How to Determine the Evaluation Order of Rules?
  - Construct the Parse Tree with Attributes
  - Build an Attribute Dependence Graph
  - Topological Sort It and Evaluate the Rules



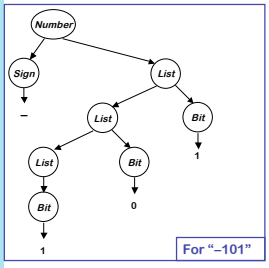
Evaluation Methods

Dynamic, dependence-based methods
- Build the Parse Tree
- Build the Dependence Graph
- Topological Sort the Dependence Graph
- Evaluate Attributes in Topological Order

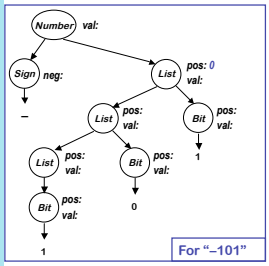
Rule-based methods *(treewalk)*
- Analyze Rules at Compiler-Generation Time
- Determine a Fixed (static) Ordering
- Evaluate Nodes in that Order

Oblivious Methods *(passes, dataflow)*
- Ignore Rules & Parse Tree
- Pick a Convenient Order (at design time) & Use It

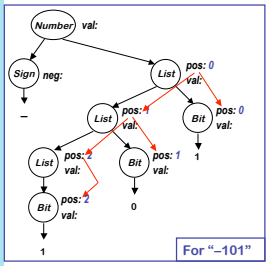
Example



Example

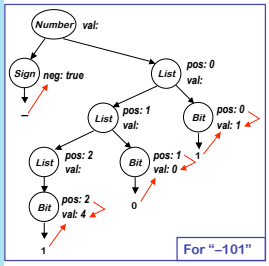


Example



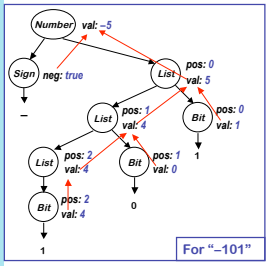
Inherited Attributes

Back to the Example



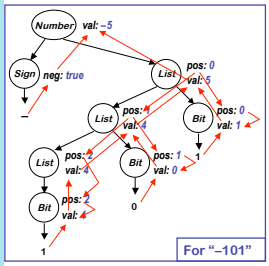
Synthesized attributes

Example



Synthesized attributes

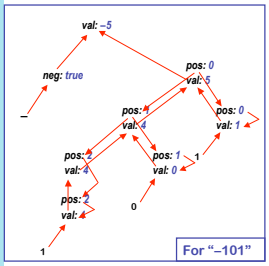
Example



If we show the computation ...

& then peel away the parse tree ...

Example



All that is left is the attribute dependence graph.

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

The dynamic methods sort this graph to find independent values, then work along graph edges.

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

Circularity

We can only evaluate Acyclic Instances

- We can prove that some grammars can only generate instances with acyclic dependence graphs
- Largest such class is "strongly non-circular" grammars (*SNC*)
- *SNC* grammars can be tested in polynomial time
- Failing the *SNC* test is not conclusive

Many evaluation methods discover circularity dynamically

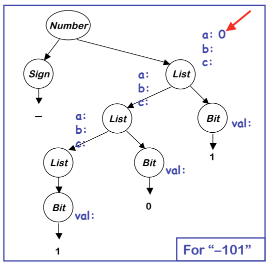
⇒ Bad property for a compiler to have

SNC grammars were first defined by Kennedy & Warren

A Circular Attribute Grammar

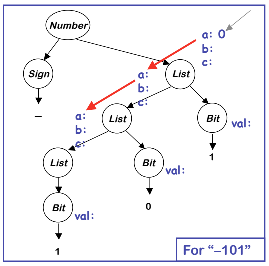
Productions			Attribution Rules
Number	→	*List*	$List.a \leftarrow 0$
$List_0$	→	*List*, *Bit*	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
	|	*Bit*	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
Bit	→	0	$Bit.val \leftarrow 0$
	|	1	$Bit.val \leftarrow 2^{Bit.pos}$

Circular Grammar



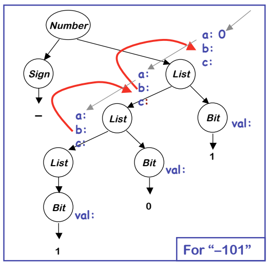
$Number \rightarrow List$	$List.a \leftarrow 0$
$List_0 \rightarrow List_1\ Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
$\mid Bit$	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$\mid 1$	$Bit.val \leftarrow 2^{Bit.pos}$

Circular Grammar



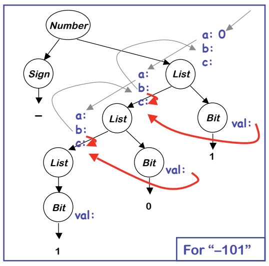
$Number \rightarrow List$	$List.a \leftarrow 0$
$List_0 \rightarrow List_1\ Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
$\mid Bit$	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$\mid 1$	$Bit.val \leftarrow 2^{Bit.pos}$

Circular Grammar



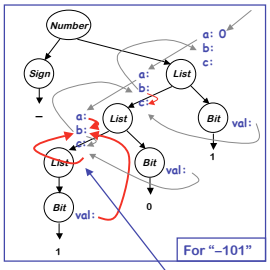
$Number \rightarrow List$	$List.a \leftarrow 0$
$List_0 \rightarrow List_1\ Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
$\mid Bit$	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$\mid 1$	$Bit.val \leftarrow 2^{Bit.pos}$

Circular Grammar



$Number \rightarrow List$	$List.a \leftarrow 0$
$List_0 \rightarrow List_1\ Bit$	$List_1.a \leftarrow List_0.a + 1$ $List_0.b \leftarrow List_1.b$ $List_1.c \leftarrow List_1.b + Bit.val$
$\mid Bit$	$List_0.b \leftarrow List_0.a + List_0.c + Bit.val$
$Bit \rightarrow 0$	$Bit.val \leftarrow 0$
$\mid 1$	$Bit.val \leftarrow 2^{Bit.pos}$

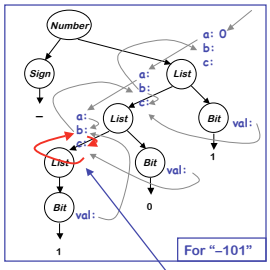
Circular Grammar



Number → List	*List.a ← 0*
List$_0$ → List$_1$ Bit	*List$_1$.a ← List$_0$.a + 1* *List$_0$.b ← List$_1$.b* *List$_1$.c ← List$_1$.b + Bit.val*
| Bit	*List$_0$.b ← List$_0$.a + List$_0$.c + Bit.val*
Bit → 0	*Bit.val ← 0*
| *1*	*Bit.val ← 2$^{Bit.pos}$*

Here is the circularity …

Circular Grammar



Number → List	*List.a ← 0*
List$_0$ → List$_1$ Bit	*List$_1$.a ← List$_0$.a + 1* *List$_0$.b ← List$_1$.b* *List$_1$.c ← List$_1$.b + Bit.val*
| Bit	*List$_0$.b ← List$_0$.a + List$_0$.c + Bit.val*
Bit → 0	*Bit.val ← 0*
| *1*	*Bit.val ← 2$^{Bit.pos}$*

Here is the circularity …

An Example: Estimating Cycle Counts

Grammar for a Basic Block

Block$_0$	→	*Block$_1$ Assign*
	|	*Assign*
Assign	→	*Ident = Expr ;*
Expr$_0$	→	*Expr$_1$ + Term*
	|	*Expr$_1$ – Term*
	|	*Term*
Term$_0$	→	*Term$_1$ * Factor*
	|	*Term$_1$ / Factor*
	|	*Factor*
Factor	→	*(Expr)*
	|	*Number*
	|	*Identifier*

Assumptions:

- Each operation has a COST
- Add them, bottom up
- Assume a load per value
- Assume no reuse via registers

Simple problem for an *AG*

An Example (*continued*)

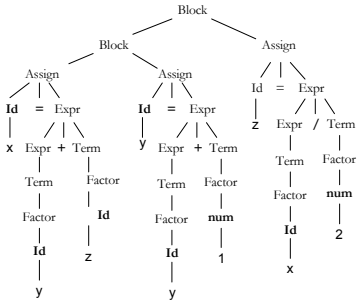
Block$_0$	→ *Block$_1$ Assign*	Block$_0$.cost ← Block$_1$.cost + Assign.cost
	| *Assign*	Block$_0$.cost ← Assign.cost
Assign	→ *Ident = Expr ;*	Assign.cost ← COST(store) + Expr.cost
Expr$_0$	→ *Expr$_1$ + Term*	Expr$_0$.cost ← Expr$_1$.cost + COST(add) + Term.cost
	| *Expr$_1$ - Term*	Expr$_0$.cost ← Expr$_1$.cost + COST(add) + Term.cost
	| *Term*	Expr$_0$.cost ← Term.cost
Term$_0$	→ *Term$_1$ * Factor*	Term$_0$.cost ← Term$_1$.cost + COST(mult) + Factor.cost
	| *Term$_1$ / Factor*	Term$_0$.cost ← Term$_1$.cost + COST(div) + Factor.cost
	| *Factor*	Term$_0$.cost ← Factor.cost
Factor	→ *(Expr)*	Factor.cost ← Expr.cost
	| *Number*	Factor.cost ← COST(loadI)
	| *Identifier*	Factor.cost ← COST(load)

These are all synthesized attributes !

Values flow from *RHS* to *LHS* in prod'ns

An Example (*continued*)

x = y + z
y = y + 1
z = x / 2



cost(loadI) = 1
cost(load) = 2
cost(store) = 2
cost(add) = 3
cost(div) = 9

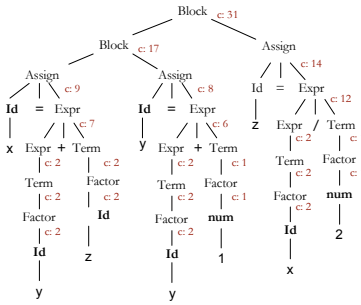
An Example (*continued*)

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

An Example (*continued*)

x = y + z
y = y + 1
z = x / 2



cost(loadI) = 1
cost(load) = 2
cost(store) = 2
cost(add) = 3
cost(div) = 9

An Interesting Example - Tracking Loads

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded
- Assumes an Infinite Register Set to Hold Variables

A Better Execution Model

Adding load tracking

- Need Sets *Before* and *After* for each production
- Must be initialized, updated, and passed around the tree

Factor → (Expr)	Factor.cost ← Expr.cost ; Expr.Before ← Factor.Before ; Factor.After ← Expr.After
| Number	Factor.cost ← COST(loadi) ; Factor.After ← Factor.Before

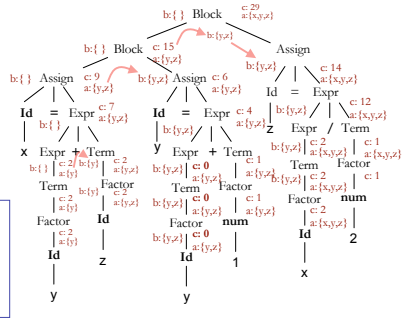
| Identifier	If (Identifier.name ∉ Factor.Before)
	then
	Factor.cost ← COST(load);
	Factor.After ← Factor.Before
	∪ Identifier.name
	else
	Factor.cost ← 0
	Factor.After ← Factor.Before

This looks more complex!

An Example (*continued*)

x = y + z
y = y + 1
z = x / 2

cost(loadI) = 1
cost(load) = 2
cost(store) = 2
cost(add) = 3
cost(div) = 9



A Better Execution Model

- Load tracking adds complexity
- But, most of it is in the "copy rules"
- Every production needs rules to copy *Before* & *After*

A sample production

$Expr_0$ → $Expr_1$ + Term	$Expr_0$.cost ← $Expr_1$.cost + COST(add) + Term.cost ; $Expr_1$.Before ← $Expr_0$.Before ; Term.Before ← $Expr_1$.After; $Expr_0$.After ← Term.After

These copy rules multiply rapidly
Each creates an instance of the set
Lots of work, lots of space, lots of rules to write

An Even Better Model

What about accounting for Finite Register Sets?

- *Before* & *After* must be of Limited Size
- Adds complexity to *Factor→Identifier*
- Requires more Complex Initialization

Jump from Tracking Loads to Tracking Registers is small

- Copy rules are already in place
- Some local code to perform the Allocation

The Moral of the Story

- *Non-local* computation needed lots of supporting rules
- Complex *local* computation was relatively easy

The Problems

- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  - Need copies of attributes
  - Can use pointers, for even more cognitive overhead
- Result is an attributed tree *(somewhat subtle points)*
  - Must build the parse tree
  - Either search tree for answers or copy them to the root

Addressing the Problem

If you gave this problem to a real programmer

- Introduce a central repository for facts
- Table of names
  - Field in table for loaded/not loaded state
- 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.3)*
  - When its done, information is in the table !
  - Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - Do we care?

Reworking the Example (*with load tracking*)

$Block_0$	→	$Block_1$ Assign	
	|	Assign	cost ← 0;
Assign	→	Ident = Expr ;	cost← cost + COST(store);
$Expr_0$	→	$Expr_1$ + Term	cost← cost + COST(add);
	|	$Expr_1$ – Term	cost← cost + COST(sub);
	|	Term	
$Term_0$	→	$Term_1$ * Factor	cost← cost + COST(mult);
	|	$Term_1$ / Factor	cost← cost + COST(div);
	|	Factor	
Factor	→	(Expr)	
	|	Number	cost← cost + COST(loadi);
	|	Identifier	{ i← hash(Identifier);
			if (Table[i].loaded = false)
			then {
			cost ← cost + COST(load);
			Table[i].loaded ← true;
			}
			}

This looks cleaner & simpler !

Summary

- Attribute Grammar
  - Augment CFG with Attributes and Rules
  - Inherited and Synthesized Attributes
- Syntax-Directed Definitions
  - Find Dependence Graph and Evaluation Order
  - Useful for Semantic Analysis