Context-sensitive Analysis II:
From Attribute grammars to ad-hoc syntax-directed translation

COMP 412
Fall 2005

Copyright 2005, Keith D. Cooper, Ken Kennedy & Linda Torczon, all rights reserved.
Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

An Extended Attribute Grammar Example

Grammar for a basic block *(§ 4.3.3)*

$Block_0$	→	*$Block_1$ 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*

Let's estimate cycle counts

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

Simple problem for an *AG*

Hey, this looks useful !

An Extended 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 Extended Example *(continued)*

Properties of the example grammar

- All attributes are synthesized $\Rightarrow$ 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

What about an improvement?

- Values are loaded only once per block (not at each use)
- Need to track which values have been already loaded

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	$\rightarrow$	**(Expr)**	**Factor.cost $\leftarrow$ Expr.cost ;** **Expr.Before $\leftarrow$ Factor.Before ;** **Factor.After $\leftarrow$ Expr.After**
	|	**Number**	**Factor.cost $\leftarrow$ COST(loadi) ;** **Factor.After $\leftarrow$ Factor.Before**
	|	**Identifier**	**If (Identifier.name $\notin$ Factor.Before)** **then** **Factor.cost $\leftarrow$ COST(load);** **Factor.After $\leftarrow$ Factor.Before $\cup$ { Identifier.name }** **else** **Factor.cost $\leftarrow$ 0** **Factor.After $\leftarrow$ Factor.Before**

This looks more complex!

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

$Expr_0 \rightarrow Expr_1 + Term$	$Expr_0$.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

And Its Extensions

Tracking loads

- Introduced *Before* and *After* sets to record loads
- Added ≥ 2 copy rules per production
  - Serialized evaluation into execution order
- Made the whole attribute grammar large & cumbersome

Finite register set

- Complicated one production (*Factor* → Identifier)
- Needed a little fancier initialization
- Changes were quite limited

Why is one change hard and the other easy?

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 chief programmer in COMP 314

- 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 it is done, information is in the table !
  - Cures most of the problems
- Unfortunately, this design violates the functional paradigm
  - Do we care?

The Realist's Alternative

Ad-hoc syntax-directed translation

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

Reworking the Example *(with load tracking)*

$Block_0$	→	$Block_1$ Assign	
	|	Assign	
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 than the AG sol'n !

One missing detail: initializing cost

Reworking the Example (*with load tracking*)

Start	→	Init Block	
Init	→	ε	cost ← 0;
$Block_0$	→	$Block_1$ Assign	
	|	Assign	
Assign	→	Ident = Expr ;	cost← cost + COST(store);

… and so on as in the previous version of the example …

- Before parser can reach Block, it must reduce Init
- Reduction by Init sets cost to zero

This is an example of splitting a production to create a reduction in the middle — for the sole purpose of hanging an action routine there!

Reworking the Example *(with load tracking)*

$Block_0$	→	$Block_1$ Assign	$$ ← $1 + $2 ;
	|	Assign	$$ ← $1 ;
Assign	→	Ident = Expr ;	$$← COST(store) + $3;
$Expr_0$	→	$Expr_1$ + Term	$$← $1 + COST(add) + $3;
	|	$Expr_1$ - Term	$$← $1 + COST(sub) + $3;
	|	Term	$$ ← $1;
$Term_0$	→	$Term_1$ * Factor	$$ ← $1 + COST(mult) + $3;
	|	$Term_1$ / Factor	$$ ← $1 + COST(div) + $3;
	|	Factor	$$ ← $1;
Factor	→	(Expr)	$$ ← $2;
	|	Number	$$ ← COST(loadi);

	|	Identifier	{ i← hash(Identifier);
			if (Table[i].loaded = false)
			then {
			$$ ← COST(load);
			Table[i].loaded ← true;
			}
			else $$ ← 0
			}

This version passes the values through attributes. It avoids the need for initializing "cost"

Example — Building an Abstract Syntax Tree

- Assume constructors for each node
- Assume stack holds pointers to nodes
- Assume yacc syntax

Goal	→	*Expr*	$$ = $1;
Expr	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
	|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
	|	*Term*	$$ = $1;
Term	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
	|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
	|	*Factor*	$$ = $1;
Factor	→	(*Expr*)	$$ = $2;
	|	number	$$ = MakeNumNode(token);
	|	id	$$ = MakeIdNode(token);

Reality

Most parsers are based on this *ad-hoc* style of context-sensitive analysis

Advantages

- Addresses the shortcomings of the AG paradigm
- Efficient, flexible

Disadvantages

- Must write the code with little assistance
- Programmer deals directly with the details

Most parser generators support a yacc-like notation

Typical Uses

- Building a symbol table
  - Enter declaration information as processed
  - At end of declaration syntax, do some post processing
  - Use table to check errors as parsing progresses

assumes table is global

- Simple error checking/type checking
  - Define before use → lookup on reference
  - Dimension, type, ... → check as encountered
  - Type conformability of expression → bottom-up walk
  - Procedure interfaces are harder
    - Build a representation for parameter list & types
    - Create list of sites to check
    - Check offline, or handle the cases for arbitrary orderings

Is This Really "Ad-hoc" ?

Relationship between practice and attribute grammars

Similarities

- Both rules & actions associated with productions
- Application order determined by tools, not author
- (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 functional
- AG rules are higher level than *ad-hoc* actions

Limitations

- Forced to evaluate in a given order: *postorder*
  - Left to right only
  - Bottom up only

- Implications
  - Declarations before uses
  - Context information cannot be passed down
    - How do you know what rule you are called from within?
    - Example: cannot pass bit position from right down
  - Could you use globals?
    - Requires initialization & some re-thinking of the solution
  - Can we rewrite it in a form that is better for the ad-hoc sol'n

Limitations

Can often rewrite the problem to fit S-attributed model

Number → *Sign List*	$\$\$ \leftarrow \$1 \times \$2$
Sign → <u>+</u>	$\$\$ \leftarrow 1$
| <u>-</u>	$\$\$ \leftarrow -1$
$List_0 \rightarrow List_1\ Bit$	$\$\$ \leftarrow 2 \times \$1 + \$2$
| Bit	$\$\$ \leftarrow \$1$
Bit → *0*	$\$\$ \leftarrow 0$
| *1*	$\$\$ \leftarrow 1$

The key step

We picked the original attribution rules to highlight features of attribute grammars, rather than to show you the most efficient way to compute the answer!

Of course, you can rewrite the AG in this same S-attributed style

Making Ad-hoc SDT Work

How do we fit this into an LR(1) parser?

- Need a place to store the attributes
  - Stash them in the stack, along with state and symbol
  - Push three items each time, pop 3 x $|\beta|$ symbols
- Need a naming scheme to access them
  - $n translates into stack location (top - 3n)
- Need to sequence rule applications
  - On every reduce action, perform the action rule
  - Add a giant case statement to the parser

Adds a rule evaluation to each reduction

  - Usually the code snippets are relatively cheap

Making Ad-hoc SDT Work

What about a rule that must work in mid-production?

- Can transform the grammar
  - Split it into two parts at the point where rule must go
  - Apply the rule on reduction to the appropriate part
- Can also handle reductions on shift actions
  - Add a production to create a reduction
    - Was: *fee* → *fum*
    - Make it: *fee* → *fie* → *fum*

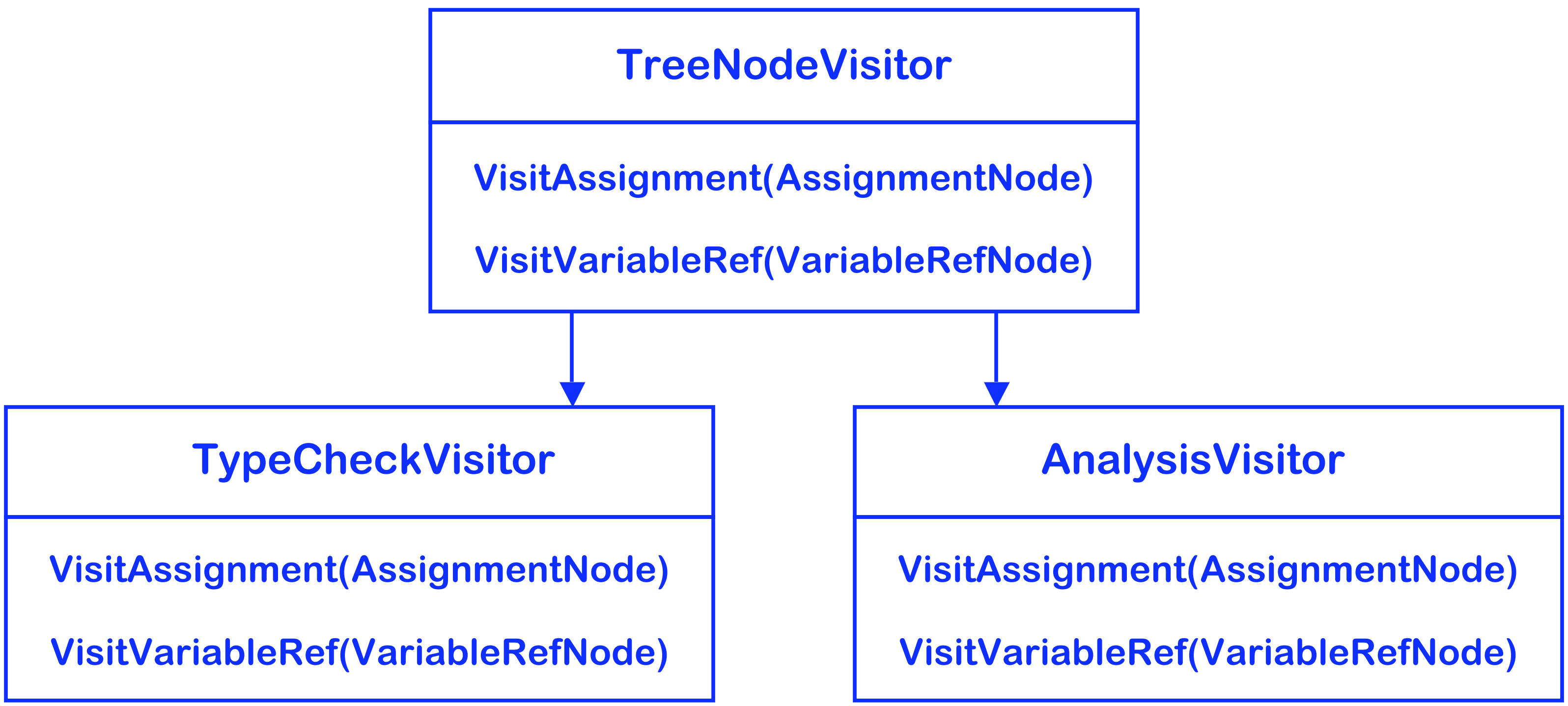
    and tie the action to this new reduction

Together, these let us apply rule at any point in the parse

Alternative Strategy

Use SDT to build an abstract syntax tree & do complex work in a tree walk

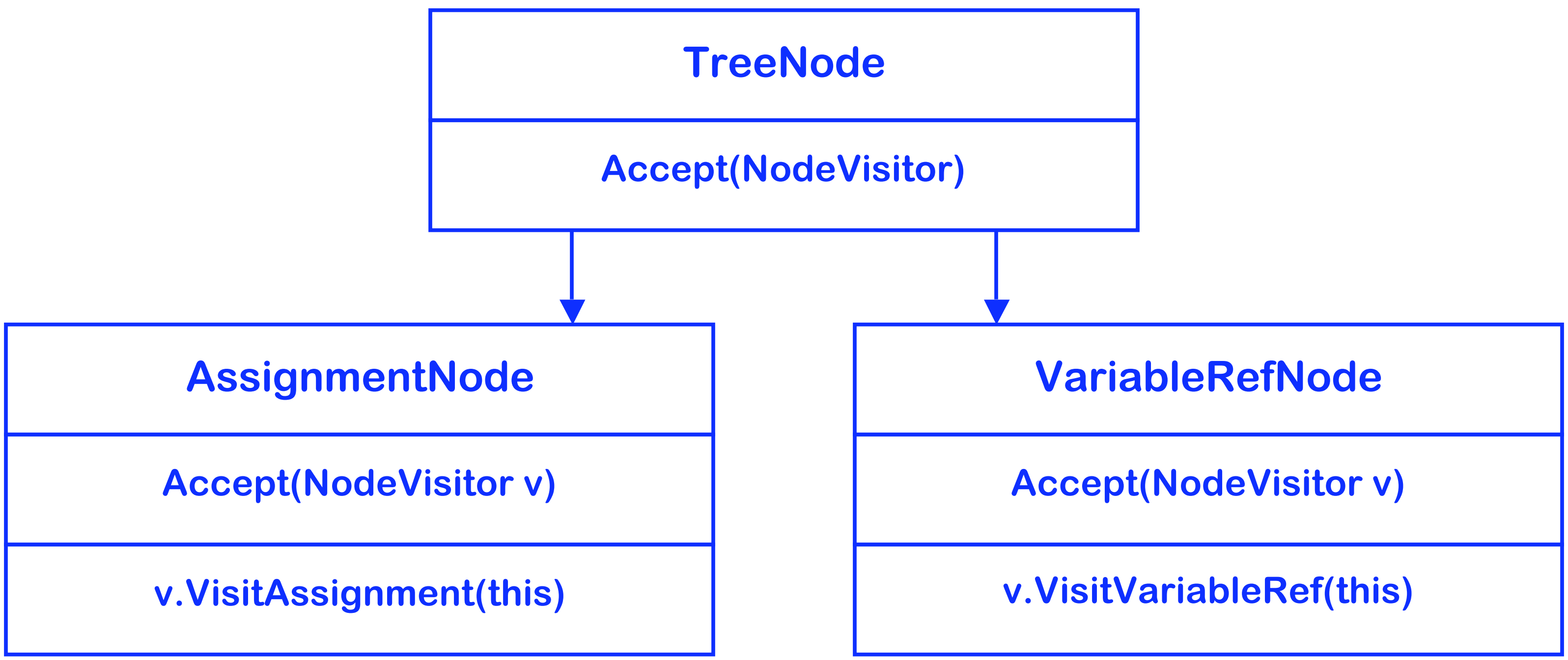
- Use tree walk routines
- Use “visitor” design pattern to add functionality



Visitor Treewalk I

Code parallels the tree's structure:

- Separates treewalk code from node handling code
- Facilitates change in processing without change to tree structure



Visitor Treewalk II

```
VisitAssignment(aNodePtr)
        // preprocess assignment
        (aNodePtr->rhs)->Accept(this);
        // postprocess rhs info;
        (aNodePtr->lhs)->Accept(this);
        // postprocess assignment;
```

Refers to current visitor!

To start the process:

```
AnalysisVisitor a; treeRoot->Accept(a);
```

Summary: Strategies for C-S Analysis

- Attribute Grammars
  - Pros: Formal, powerful, can deal with propagation strategies
  - Cons: Too many copy rules, no global tables, works on parse tree
- Postorder Code Execution
  - Pros: Simple and functional, can be specified in grammar (Yacc) but does not require parse tree
  - Cons: Rigid evaluation order, no context inheritance
- Generalized Tree Walk
  - Pros: Full power and generality, operates on abstract syntax tree (using Visitor pattern)
  - Cons: Requires specific code for each tree node type, more complicated