

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.

THE PART

An Extended Attribute Grammar Example

Grammar for a basic block

(§ 4.3.3)

Block ₀	_ →	Block1 Assign
		Assign
Assign	\rightarrow	Ident = Expr ;
Expr ₀	→	Expr1 + Term
		Expr1 - Term
		Term
Term₀	\rightarrow	Term ₁ * Factor
		Term₁ / Factor
		Factor
Factor	\rightarrow	(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	\rightarrow	Block ₁ Assign	Block₀.cost ← Block₁.cost +
	_	_	Assign.cost
		Assign	Block₀.cost ← Assign.cost
Assign	\rightarrow	Ident = Expr ;	Assign.cost ← COST(store) +
			Expr.cost
Expr ₀	\rightarrow	Expr1 + Term	$Expr_{0}.cost \leftarrow Expr_{1}.cost +$
			<pre>COST(add) + Term.cost</pre>
		Expr1 - Term	$Expr_{0}.cost \leftarrow Expr_{1}.cost +$
			<pre>COST(add) + Term.cost</pre>
		Term	Expr₀.cost ← Term.cost
Term ₀	\rightarrow	Term ₁ * Factor	$Term_{0}.cost \leftarrow Term_{1}.cost +$
			<pre>COST(mult) + Factor.cost</pre>
		Term1 / Factor	$Term_{0}.cost \leftarrow Term_{1}.cost +$
			COST (div) + Factor.cost
		Factor	Term₀.cost ← Factor.cost
Factor	\rightarrow	(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



(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



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 ← 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
			\cup { Identifier.name }
			else
			Factor.cost ← 0
			Factor.After ← 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 _o →	Expr₁	+ Term	Expr _o .cost ← Expr₁.cost +
			COST(add) + Term.cost ;
			Expr₁.Before ← Expr₀.Before ;
			Term.Before ← Expr₁.After;
			Expr₀.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





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* \rightarrow 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
 - Must build the parse tree
 - Either search tree for answers or copy them to the root

Comp 412 Fall 2005

(somewhat subtle points)

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

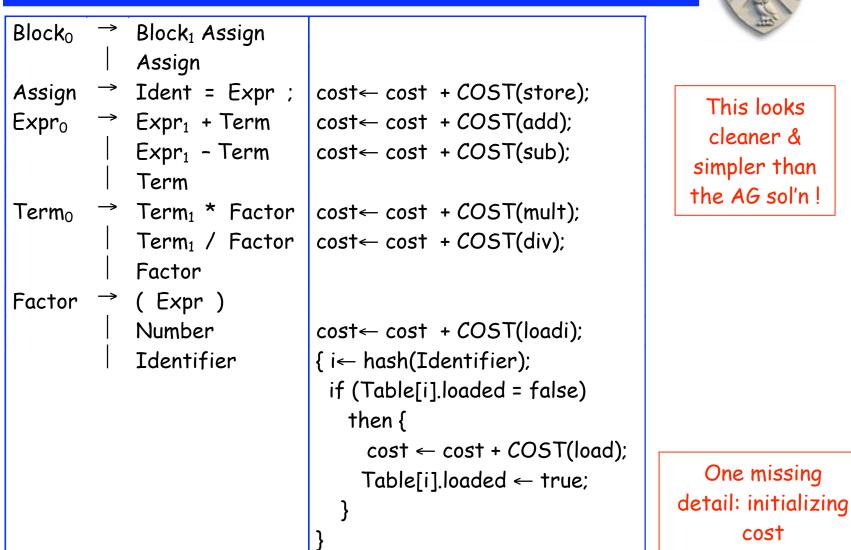
Ad-hoc syntax-directed translation

- 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

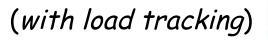






Reworking the Example

Comp 412 Fall 2005





Start	\rightarrow	Init Block	
Init	\rightarrow	ε	cost ← 0;
		Block ₁ Assign Assign	
Assign	\rightarrow	Ident = Expr ;	$cost \leftarrow cost + COST(store);$

... 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!



		3	
Block ₀	→	Block ₁ Assign	\$\$ ← \$1 + \$2 ;
		Assign	\$\$ ← \$1 ;
Assign		Ident = Expr ;	\$\$← COST(store) + \$3;
Expr ₀		Expr1 + Term	\$\$← \$1 + COST(add) + \$3;
		Expr1 - Term Term	\$\$← \$1 + COST(sub) + \$3; \$\$ ← \$1;
Term ₀	→	Term ₁ * Factor	\$\$ ← \$1 + COST(mult) + \$3;
		Term ₁ / Factor	\$\$ ← \$1 + COST(div) + \$3;
		Factor	\$\$ ← \$1;
Factor	→ 	(Expr) Number Identifier	<pre>\$\$ ← \$2; \$\$ ← COST(loadi); { i← hash(Identifier); if (Table[i].loaded = false) then { \$\$ ← COST(load); Table[i].loaded ← true; } else \$\$ ← 0 }</pre>

Reworking the Example

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

Comp 412 Fall 2005

Example — Building an Abstract Syntax Tree



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

Goal	\rightarrow	Expr	\$\$ = \$1;
Expr	\rightarrow	Expr + Term	\$\$ = MakeAddNode(\$1,\$3);
	Ι	Expr - Term	\$\$ = MakeSubNode(\$1,\$3);
	Ι	Term	\$\$ = \$1 ;
Term	\rightarrow	Term * Factor	\$\$ = MakeMulNode(\$1,\$3);
	Ι	Term / Factor	\$\$ = MakeDivNode(\$1,\$3);
	Ι	Factor	\$\$ = \$1 ;
Factor	\rightarrow	<u>(</u> Expr <u>)</u>	\$\$ = \$2;
	Ι	number	\$\$ = MakeNumNode(token);
	Ι	<u>id</u>	\$\$ = MakeIdNode(token);

Reality



Most parsers are based on this *ad-hoc* style of contextsensitive 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
- Simple error checking/type checking
 - Define before use \rightarrow lookup on reference
 - Dimension, type, ... \rightarrow check as encountered
 - Type conformability of expression \rightarrow 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