COMP 412
FALL 2010

Context-sensitive Analysis, Part II From AGs to ad-hoc methods

Comp 412

The Moral of the Story

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

A good rule of thumb is that the compiler touches all the space it allocates, usually multiple times

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 the problem of estimating cycle counts to a competent junior or senior CS major, …

- 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 of an AG.

Do we care?

The Realist's Alternative

Ad-hoc syntax-directed translation

- Build on bottom-up, shift-reduce parser
- 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

Similar ideas work for top-down parsers…

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)*

1	*Block*$_0$	$\rightarrow$	*Block*$_1$ *Assign*	
2		|	*Assign*	
3	*Assign*$_0$	$\rightarrow$	*Ident = Expr ;*	cost $\leftarrow$ cost + COST(store)
4	*Expr*$_0$	$\rightarrow$	*Expr*$_1$ + *Term*	cost $\leftarrow$ cost + COST(add)
5		|	*Expr*$_1$ - *Term*	cost $\leftarrow$ cost + COST(sub)
6		|	*Term*	
7	*Term*$_0$	$\rightarrow$	*Term*$_1$ * *Factor*	cost $\leftarrow$ cost + COST(mult)
8		|	*Term*$_1$ / *Factor*	cost $\leftarrow$ cost + COST(div)
9		|	*Factor*	
10	*Factor*	$\rightarrow$	(*Expr*)	
11		|	*Number*	cost $\leftarrow$ cost + COST(loadI)

12		|	*Ident*	i $\leftarrow$ hash(Ident);
				if (Table[i].loaded = false)
				then {
				cost $\leftarrow$ cost + COST(load)
				Table[i].loaded $\leftarrow$ true
				}

This looks cleaner & simpler than the AG sol'n !

One missing detail: initializing cost

Reworking the Example (*with load tracking*)

0	*Start*		*Init Block*	
.5	*Init*		ε	cost $\leftarrow$ 0
1	$Block_0$	$\rightarrow$	$Block_1$ *Assign*	
2		|	*Assign*	
3	$Assign_0$	$\rightarrow$	*Ident = Expr ;*	cost $\leftarrow$ cost + COST(store)

and so on as shown on previous slide…

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

We split the production to create a reduction in the middle — for the sole purpose of hanging an action there. This trick has many uses.

Reworking the Example *(with load tracking)*

1	$Block_0$	→	$Block_1$ *Assign*	$$ ← $1 + $2
2		|	*Assign*	$$ ← $1
3	$Assign_0$	→	*Ident = Expr ;*	$$ ← COST(store) + $3
4	$Expr_0$	→	$Expr_1$ *+ Term*	$$ ← $1 + COST(add) + $3
5		|	$Expr_1$ *- Term*	$$ ← $1 + COST(sub) + $3
6		|	*Term*	$$ ← $1
7	$Term_0$	→	$Term_1$ ** Factor*	$$ ← $1 + COST(mult) + $3
8		|	$Term_1$ */ Factor*	$$ ← $1 + COST(div) + $3
9		|	*Factor*	$$ ← $1
10	*Factor*	→	*(Expr)*	$$ ← $2
11		|	*Number*	$$ ← COST(loadI)

12		|	*Ident*	i ← hash(Ident);
				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 to initialize "cost"

Example — Assigning Types in Expression Nodes

$F_\times$	Int 16	Int 32	Float	Double
Int 16	Int 16	Int 32	Float	Double
Int 32	Int 32	Int 32	Float	Double
Float	Float	Float	Float	Double
Double	Double	Double	Double	Double

- Assume typing functions or tables F_+, F_-, $F_\times$, and $F_\div$

1	*Goal*	$\rightarrow$	*Expr*	$$ = $1;
2	*Expr*	$\rightarrow$	*Expr* + *Term*	$$ = F_+($1,$3);
3		|	*Expr* - *Term*	$$ = F_-($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	$\rightarrow$	*Term* * *Factor*	$$ = $F_\times$($1,$3);
6		|	*Term* / *Factor*	$$ = $F_\div$($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	$\rightarrow$	(*Expr*)	$$ = $2;
9		|	number	$$ = type of num;
10		|	ident	$$ = type of ident;

Example — Building an Abstract Syntax Tree

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

1	*Goal*	→	*Expr*	$$ = $1;
2	*Expr*	→	*Expr* + *Term*	$$ = MakeAddNode($1,$3);
3		|	*Expr* - *Term*	$$ = MakeSubNode($1,$3);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = MakeMulNode($1,$3);
6		|	*Term* / *Factor*	$$ = MakeDivNode($1,$3);
7		|	*Factor*	$$ = $1;
8	*Factor*	→	(*Expr*)	$$ = $2;
9		|	number	$$ = MakeNumNode(token);
10		|	ident	$$ = MakeIdNode(token);

Example — Emitting ILOC

- Assume *NextRegister()* returns a virtual register name
- Assume *Emit()* can format assembly code

1	*Goal*	→	*Expr*	
2	*Expr*	→	*Expr* + *Term*	$$ = *NextRegister*(); *Emit*(add, $1,$3, $$);
3		|	*Expr* - *Term*	$$ = *NextRegister*(); *Emit*(sub, $1, $3, $$);
4		|	*Term*	$$ = $1;
5	*Term*	→	*Term* * *Factor*	$$ = *NextRegister*(); *Emit*(mult,$1, $3, $$)
6		|	*Term* / *Factor*	$$ = *NextRegister*()' *Emit*(div, $1, $3, $$);
7		|	*Factor*	$$ = $1;

Example — Emitting ILOC

- Assume *NextRegister()* returns a virtual register name
- Assume *Emit()* can format assembly code
- Assume *EmitLoad()* handles addressability & gets a value into a register

8	*Factor*	→	(*Expr*)	\$\$ = \$2;
9		|	number	\$\$ = *NextRegister()*; *Emit*(loadi,Value(lexeme),\$\$);
10		|	ident	\$\$ = *NextRegister()*; *EmitLoad*(ident,\$\$);

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

1	*Number*	$\rightarrow$	*Sign List*	$$ = $1 x $2;
2	*Sign*	$\rightarrow$	+	$$ = 1;
3		|	-	$$ = -1;
4	*List*$_0$	$\rightarrow$	*List*$_1$ *Bit*	$$ = 2 * $1 + $2;
5		|	*Bit*	$$ = $1;
6	*Bit*	$\rightarrow$	0	$$ = 0;
7		|	1	$$ = 1

The key step

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

Limitations

Of course, the same scheme works in an attribute grammar

1	*Number*	$\rightarrow$	*Sign List*	Number.val $\leftarrow$ Sign.neg x List.val;
2	*Sign*	$\rightarrow$	+	Sign.neg $\leftarrow$ 1;
3		|	-	Sign.neg $\leftarrow$ -1;
4	$List_0$	$\rightarrow$	$List_1$ *Bit*	$List_0$.val $\leftarrow$ 2 * $List_1$.val + Bit.val;
5		|	*Bit*	List.val $\leftarrow$ Bit.val;
6	*Bit*	$\rightarrow$	0	Bit.val $\leftarrow$ 0;
7		|	1	Bit.val $\leftarrow$ 1

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!

Making Ad-hoc SDT Work

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

```
stack.push(INVALID);
stack.push(s0);                       // initial state
token = scanner.next_token();
loop forever {
      s = stack.top();
      if ( ACTION[s,token] == "reduce A→β" ) then {
         stack.popnum(2*|β|);      // pop 2*|β| symbols
         s = stack.top();
         stack.push(A);               // push A
         stack.push(GOTO[s,A]);  // push next state
      }
      else if ( ACTION[s,token] == "shift si" ) then {
            stack.push(token); stack.push(si);
            token ← scanner.next_token();
      }
      else if ( ACTION[s,token] == "accept"
                              & token == EOF )
            then break;
      else throw a syntax error;
}
report success;
```

From an earlier lecture

Augmented LR(1) Skeleton Parser

```
stack.push(INVALID);
stack.push(NULL);
stack.push(s0);                      // initial state
token = scanner.next_token();
loop forever {
    s = stack.top();
    if ( ACTION[s,token] == "reduce A→β" ) then {

        /* insert case statement here */

        stack.popnum(3*|β|);         // pop 3*|β| symbols
        s = stack.top();
        stack.push(A);               // push A
        stack.push(GOTO[s,A]);       // push next state
    }
    else if ( ACTION[s,token] == "shift si" ) then {
        stack.push(token); stack.push(si);
        token ← scanner.next_token();
    }
    else if ( ACTION[s,token] == "accept"
                    & token == EOF )
        then break;
    else throw a syntax error;
}
report success;
```

To add yacc-like actions

- Stack 3 items per symbol rather than 2 (3rd is $$)
- Add case statement to the reduction processing section
  - → Case switches on production number
  - → Each case clause holds the code snippet for that production
  - → Substitute appropriate names for $$, $1, $2, …
- Slight increase in parse time
- 50% increase in stack space

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

Use macros

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* → *<u>fum</u>*
    - → Make it: *fee* → *fie* → *<u>fum</u>*

      and tie the action to this new reduction

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

STOP

Alternative Strategy

What if you <u>need</u> to perform actions that do not fit well into the Ad-hoc Syntax-Directed Translation framework?

- Build the abstract syntax tree using SDT
- Perform the actions during one or more treewalks
  - In an OOL, think of this problem as a classic application of the visitor pattern
  - Perform arbitrary computation in treewalk order
  - Make multiple passes if necessary

Again, a competent junior or senior CS major would derive this solution after a couple of minutes of thought.

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