CS415 Compilers

Context-Sensitive Analysis part 2

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
Announcements

• Homework #5 due this Saturday, April 6.
• Project will be posted by tomorrow.
• Homework #6 will be posted by tomorrow.
Review - Beyond Syntax

Telling the story

- The attribute grammar formalism is important
  - Succinctly makes many points clear
  - Sets the stage for actual, *ad-hoc* practice (e.g.: yacc/bison)
- The problems with attribute grammars motivate practice
  - Non-local computation
  - Need for centralized information

We will cover attribute grammars, then move on to *ad-hoc* ideas (syntax-directed translation schemes)
Attribute Grammars (AGs)

What is an attribute grammar?
• Each symbol in the derivation (instance of a token or non-terminal) may have a value, or *attribute*;
• A context-free grammar augmented with a set of rules
• The rules specify how to compute a value for each attribute

*Example grammar*

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
<td>→</td>
</tr>
<tr>
<td><strong>Sign</strong></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This grammar describes signed binary numbers

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

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

<table>
<thead>
<tr>
<th>Productions</th>
<th>Attribution Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Number → Sign List</code></td>
<td><code>List.pos ← 0</code>&lt;br&gt;<code>If Sign.neg</code>&lt;br&gt;<code>then Number.val ← − List.val</code>&lt;br&gt;<code>else Number.val ← List.val</code></td>
</tr>
<tr>
<td><code>Sign → ±</code></td>
<td><code>Sign.neg ← false</code></td>
</tr>
<tr>
<td>`</td>
<td>`</td>
</tr>
<tr>
<td><code>List₀ → List₁ Bit</code></td>
<td><code>List₁.pos ← List₀.pos + 1</code>&lt;br&gt;<code>Bit.pos ← List₀.pos</code>&lt;br&gt;<code>List₀.val ← List₁.val + Bit.val</code></td>
</tr>
<tr>
<td>`</td>
<td>Bit`</td>
</tr>
<tr>
<td><code>Bit → 0</code></td>
<td><code>Bit.val ← 0</code></td>
</tr>
<tr>
<td>`</td>
<td>1`</td>
</tr>
</tbody>
</table>

Symbol | Attributes
---|---
`Number` | val
`Sign` | neg
`List` | pos, val
`Bit` | pos, val
### Attribute Grammars

**Productions**

\[
\text{List}_0 \rightarrow \text{List}_1 \text{ Bit}
\]

**Attribution Rules**

\[
\begin{align*}
\text{List}_1.\text{pos} & \leftarrow \text{List}_0.\text{pos} + 1 \\
\text{Bit}.\text{pos} & \leftarrow \text{List}_0.\text{pos} \\
\text{List}_0.\text{val} & \leftarrow \text{List}_1.\text{val} + \text{Bit}.\text{val}
\end{align*}
\]

- semantic rules define partial dependency graph
- value flow top down or across: **inherited attributes**
- value flow bottom-up: **synthesized attributes**
**Attribute Grammars**

- **Semantic rules associated with production** $A \rightarrow \alpha$ **have to specify the values for all**
  - synthesized attributes for $A$ (root)
  - inherited attributes for grammar symbols in $\alpha$ (children)

  $\Rightarrow$ **rules must specify local value flow!**

- **Terminals can be associated with values returned by the scanner.** These input values are associated with a synthesized attribute.

- **Starting symbol cannot have inherited attributes.**
Example revisited

compute the decimal value of a signed binary number

For “–101”
Example revisited

compute the decimal value of a signed binary number

All that is left is the attribute dependence graph.
This succinctly represents the flow of values in the problem instance.

The dependence graph **must** be acyclic
compute the decimal value of a signed binary number

All that is left is the attribute dependence graph.

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

The **dynamic methods** topologically sort this graph, then evaluates edges/nodes in that order.

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.
Using Attribute Grammars

Attribute grammars can specify context-sensitive actions
- Take values from syntax
- Perform computations with values
- Insert tests, logic, ...

<table>
<thead>
<tr>
<th>Synthesized Attributes</th>
<th>Inherited Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Use values from children &amp; from constants</td>
<td>• Use values from parent, constants, &amp; siblings</td>
</tr>
<tr>
<td>• S-attributed grammars: synthesized attributes only</td>
<td>• <strong>L-attributed</strong> grammars:</td>
</tr>
<tr>
<td>• Evaluate in a single bottom-up pass</td>
<td>A [\rightarrow X_1 X_2 \ldots X_n] and each inherited attribute of (X_i) depends on</td>
</tr>
<tr>
<td>Good match to LR parsing</td>
<td>- attributes of (X_1 X_2 \ldots X_{i-1}), and</td>
</tr>
<tr>
<td></td>
<td>- inherited attributes of (A)</td>
</tr>
</tbody>
</table>

\(S\text{-attributed} \subset L\text{-attributed}\)

• Evaluate in a single top-down pass (left to right)

Good match for LL parsing
An Extended Example

Grammar for a basic block

\[
\begin{align*}
Block_0 & \rightarrow Block_1 \text{ Assign} \\
& \quad \text{Assign} \\
Assign & \rightarrow \text{Ident} = \text{Expr} ; \\
Expr_0 & \rightarrow Expr_1 + \text{Term} \\
& \quad \text{Expr}_1 - \text{Term} \\
& \quad \text{Term} \\
\text{Term}_0 & \rightarrow \text{Term}_1 \times \text{Factor} \\
& \quad \text{Term}_1 / \text{Factor} \\
& \quad \text{Factor} \\
\text{Factor} & \rightarrow ( \text{Expr} ) \\
& \quad \text{Number} \\
& \quad \text{Identifier}
\end{align*}
\]

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)

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Value Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block&lt;sub&gt;0&lt;/sub&gt; → Block&lt;sub&gt;1&lt;/sub&gt; Assign</td>
<td>Block&lt;sub&gt;0&lt;/sub&gt;.cost ← Block&lt;sub&gt;1&lt;/sub&gt;.cost + Assign.cost</td>
</tr>
<tr>
<td>Assign</td>
<td>Block&lt;sub&gt;0&lt;/sub&gt;.cost ← Assign.cost</td>
</tr>
<tr>
<td>Expr&lt;sub&gt;0&lt;/sub&gt; → Expr&lt;sub&gt;1&lt;/sub&gt; + Term</td>
<td>Assign.cost ← COST(store) + Expr&lt;sub&gt;1&lt;/sub&gt;.cost</td>
</tr>
<tr>
<td>Expr&lt;sub&gt;1&lt;/sub&gt; − Term</td>
<td>Expr&lt;sub&gt;0&lt;/sub&gt;.cost ← Expr&lt;sub&gt;1&lt;/sub&gt;.cost + COST(add) + Term.cost</td>
</tr>
<tr>
<td>Term</td>
<td>Expr&lt;sub&gt;0&lt;/sub&gt;.cost ← Term.cost</td>
</tr>
<tr>
<td>Term&lt;sub&gt;0&lt;/sub&gt; → Term&lt;sub&gt;1&lt;/sub&gt; * Factor</td>
<td>Term&lt;sub&gt;0&lt;/sub&gt;.cost ← Term&lt;sub&gt;1&lt;/sub&gt;.cost + COST(mult) + Factor.cost</td>
</tr>
<tr>
<td>Term&lt;sub&gt;1&lt;/sub&gt; / Factor</td>
<td>Term&lt;sub&gt;0&lt;/sub&gt;.cost ← Term&lt;sub&gt;1&lt;/sub&gt;.cost + COST(div) + Factor.cost</td>
</tr>
<tr>
<td>Factor</td>
<td>Term&lt;sub&gt;0&lt;/sub&gt;.cost ← Factor.cost</td>
</tr>
<tr>
<td>Factor → ( Expr )</td>
<td>Factor.cost ← Expr.cost</td>
</tr>
<tr>
<td>Number</td>
<td>Factor.cost ← COST(loadl)</td>
</tr>
<tr>
<td>Identifier</td>
<td>Factor.cost ← COST(load)</td>
</tr>
</tbody>
</table>

These are all synthesized attributes!
Values flow from rhs to lhs in prod'ns
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

What about an improvement?
- Values are loaded only once per basic block (not at each use)
- Need to track which values have been already loaded

Things will get more complicated.
Adding load tracking

- Need sets Before and After for each production
- Question: synthesized or inherited?
- Must be initialized, updated, and passed around the tree

<table>
<thead>
<tr>
<th>Factor $\rightarrow$ ( Expr )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
</tr>
<tr>
<td>Identifier</td>
</tr>
<tr>
<td>Factor.cost $\leftarrow$ Expr.cost;</td>
</tr>
<tr>
<td>Expr.Before $\leftarrow$ Factor.Before;</td>
</tr>
<tr>
<td>Factor.After $\leftarrow$ Expr.After</td>
</tr>
<tr>
<td>Factor.cost $\leftarrow$ COST(loadi);</td>
</tr>
<tr>
<td>Factor.After $\leftarrow$ Factor.Before</td>
</tr>
<tr>
<td>If (Identifier.name $\notin$ Factor.Before) then</td>
</tr>
<tr>
<td>Factor.cost $\leftarrow$ COST(load);</td>
</tr>
<tr>
<td>Factor.After $\leftarrow$ Factor.Before $\cup$ Identifier.name</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>Factor.cost $\leftarrow$ 0</td>
</tr>
<tr>
<td>Factor.After $\leftarrow$ Factor.Before</td>
</tr>
</tbody>
</table>

This looks more complex!
Load tracking adds complexity
But, most of it is in the “copy rules”
Every production needs rules to copy Before & After

A sample production

\[
\begin{array}{ll}
\text{Expr}_0 & \rightarrow \text{Expr}_1 + \text{Term} \\
\text{Expr}_0.\text{cost} & \leftarrow \text{Expr}_1.\text{cost} + \text{COST}(\text{add}) + \text{Term.}\text{cost} \\
\text{Expr}_1.\text{Before} & \leftarrow \text{Expr}_0.\text{Before} \\
\text{Term.}\text{Before} & \leftarrow \text{Expr}_1.\text{After} \\
\text{Expr}_0.\text{After} & \leftarrow \text{Term.}\text{After}
\end{array}
\]

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

- Non-local computation needed lots of supporting rules
- “Complex” local computation is relatively easy

The Problems
- Copy rules increase cognitive overhead
- Copy rules increase space requirements
  → Need copies of attributes
- Result is an attributed tree
  → Must build the parse tree
  → Either search tree for answers or copy them to the root
Addressing the Problem

What would a good programmer do?

- 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.4)
  - When its done, information is in the table!
  - Cures most of the problems
- Unfortunately, this design violates the functional, AG paradigm
  - Do we care?
Ad-hoc syntax-directed translation

- Associate pieces of code with each production
- At each reduction, the corresponding code is executed
- 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, ...)
  - 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)

This looks cleaner & simpler than the AG sol’n! "cost" and Table[ ] are global variables.

#### action on reduction (code)

<table>
<thead>
<tr>
<th>Grammar Symbol</th>
<th>Production</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block₀ → Block₁ Assign</td>
<td>{ }</td>
<td>{ }</td>
</tr>
<tr>
<td>Assign</td>
<td>Ident = Expr ; Assign</td>
<td></td>
</tr>
<tr>
<td>Expr₀ → Expr₁ + Term</td>
<td>{cost ← cost + COST(add);}</td>
<td></td>
</tr>
<tr>
<td>Expr₁ - Term</td>
<td>{cost ← cost + COST(sub);}</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>Term₀ → Term₁ * Factor</td>
<td>{cost ← cost + COST(mult);}</td>
<td></td>
</tr>
<tr>
<td>Term₁ / Factor</td>
<td>{cost ← cost + COST(div);}</td>
<td></td>
</tr>
<tr>
<td>Factor</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>( Expr )</td>
<td>{ }</td>
<td></td>
</tr>
<tr>
<td>Number</td>
<td>{cost ← cost + COST(loadi);}</td>
<td></td>
</tr>
<tr>
<td>Identifier</td>
<td>{i ← hash(Identifier);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>if (Table[i].loaded = false)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>then {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>cost ← cost + COST(load);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Table[i].loaded ← true;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

One missing detail: initializing "cost"; (we ignore "Table[ ]" for now)
Reworking the Example (with load tracking)

- 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 (marker production)!

```
| Start   | → | Init Block | { } |
| Init    | → | ε          | { cost ← 0; } |
| Block₀  | → | Block₁ Assign | { } |
|         |   | Assign     | { } |
| Assign  | → | Ident = Expr ; | { cost ← cost + COST(store); } |
```
Reworking the Example
(with load tracking)

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

However, Table[ ] still needs to be initialized
Example — Building an Abstract Syntax Tree

- Assume constructors for each node
- Stack holds pointers to nodes

<table>
<thead>
<tr>
<th>Goal</th>
<th>Expr</th>
<th>action on reduction (code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr</td>
<td>Expr + Term</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td></td>
<td>Expr - Term</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td></td>
<td>Term</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td>Term</td>
<td>Term * Factor</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td></td>
<td>Term / Factor</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr )</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>{ $$ = $$1;}</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>{ $$ = $$1;}</td>
</tr>
</tbody>
</table>
Making Ad-hoc SDT Work

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

- Need a place to store the attribute and their values
  - Stash them in the stack, along with state and symbol
  - Push three items each time, pop $3 \times |\beta|$ symbols
- Need a naming scheme to access them
  - $n$ translates into stack location: $\text{top} - 3 \times (|\beta| - n)$ with $A \rightarrow \beta$
- Need to sequence rule applications
  - On every reduce action, perform the action rule

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
    and apply the rule on reduction to the appropriate part
  - Introduce marker productions $M \rightarrow \varepsilon$ with appropriate action
### YACC: parse.y

#### parse.y:

```
{%
#include <stdio.h>
#include "attr.h"
int yylex();
void yyerror(char * s);
#include "symtab.h"
%

union {tokentype token; }

%token PROG PERIOD PROC VAR ARRAY RANGE OF
%token INT REAL DOUBLE WRITELN THEN ELSE IF
%token BEG END ASG NOT
%token EQ NEQ LT LEQ GEQ GT OR EXOR AND DIV NOT
%token <token> ID CCONST ICONST RCONST

%start program

%%
program : PROG ID ';' block PERIOD { } 
;
block : BEG ID ASG ICONST END { } 
;

%
void yyerror(char* s) {
    fprintf(stderr,"%s
",s);
}
int main() {
    printf("1\t");
yyparse();
    return 1;
}

Will be included verbatim in parse.tab.c

List and assign attributes
```

#### Rules with semantic actions

```
Main program and “helper” functions; may contain initialization code of global structures. Will be included verbatim in parse.tab.c
```
You do not have to change the scanner (scan.l)

How to specify and use attributes in YACC?

- Define attributes as types in attr.h
  
  typedef struct info_node {int a; int b} infonode;

- Include type attribute name in %union in parse.y
  
  %union {tokentype token; infonode myinfo; ... }

- Assign attributes in parse.y to
  
  - Terminals: %token <token> ID ICONST
  
  - Non-terminals: %type <myinfo> block variables procdecls cmpdstmt

- Accessing attribute values in parse.y
  
  - use $$, $1, $2 ... etc. notation:

    block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt

        { $$.a = $1.a + $2.a + $4.b; }
Example on ilab: ~uli/cs415/examples/LexYacc

Relationship between practice and attribute grammars

Similarities
• Both rules & actions associated with productions
• Application order determined by tools
• (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 (purely) functional
• AG rules are higher level than ad-hoc actions
Applications of SDT (Semantic Analysis)

• 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 → 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
    ▪ Check actual vs. formal parameter list
    ▪ Positional or keyword associations

ассumes table is <i>global</i>
Things to do and next class

Start working on the project!

Type systems and type checking

Intermediate representations
Read EaC: Chapter 5.1 – 5.3

Code generation
Read EaC: Chapter 7.1 – 7.5