CS415 Compilers

Context-Sensitive Analysis

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

- Fifth homework deadline extension: Monday, April 5
- Second project has been posted
Context-Sensitive Analysis

EaC Chapter 4
ALSU Chapter 5
Beyond Syntax

There is a level of correctness that is deeper than grammar.

```c
fie(a,b,c,d)
    int a, b, c, d;
{ ... }
fee() {
    int f[3],g[1], h, i, j, k;
    char *p;
    fie(h,i,"ab",j, k);
    k = f * i + j;
    h = g[17];
    printf("<%s,%s>.\n", p, q);
    p = 10;
}
```

What is wrong with this program?
(let me count the ways …)

- declared g[1], used g[17]
- wrong number of args to fie()
- “ab” is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

All of these are
“deeper than syntax”

To generate code, we need to understand its meaning!
Beyond Syntax

These questions are part of context-sensitive analysis

- Answers depend on “values”, i.e., something that needs computation; not parts of speech
- Questions & answers involve non-local information

How can we answer these questions?

- Use formal methods
  - Context-sensitive grammars
  - Attribute grammars
  (attributed grammars)
- Use ad-hoc techniques
  - Symbol tables
  - Ad-hoc code
  (action routines)

In scanning & parsing, formalism won; somewhat different story here.
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)
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>Number</td>
<td>Sign List</td>
</tr>
<tr>
<td>Sign</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>-</td>
</tr>
<tr>
<td>List</td>
<td>List Bit</td>
</tr>
<tr>
<td>Bit</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1</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
compute the decimal value of a signed binary number

For “–101”
Example parse tree

compute the decimal value of a signed binary number

For “–101”
Example parse tree

compute the decimal value of a signed binary number

For “–101”
Example parse tree

compute the decimal value of a signed binary number

For “–101”
Example parse tree

compute the decimal value of a signed binary number

For “−101”
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><strong>Number</strong> → <strong>Sign List</strong></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><strong>Sign</strong> → <code>±</code></td>
<td><code>Sign.neg ← false</code></td>
</tr>
<tr>
<td>`</td>
<td><code> </code>=`</td>
</tr>
<tr>
<td><strong>List₀</strong> → <strong>List₁ Bit</strong></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>` <strong>Bit</strong></td>
</tr>
<tr>
<td><strong>Bit</strong> → <code>0</code></td>
<td><code>Bit.val ← 0</code></td>
</tr>
<tr>
<td>`</td>
<td><code> </code>1`</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Number</strong></td>
<td>val</td>
</tr>
<tr>
<td><strong>Sign</strong></td>
<td>neg</td>
</tr>
<tr>
<td><strong>List</strong></td>
<td>pos, val</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td>pos, val</td>
</tr>
</tbody>
</table>
Attribute Grammars

<table>
<thead>
<tr>
<th>Productions</th>
<th>Attribution Rules</th>
</tr>
</thead>
</table>
| $List_0 \rightarrow List_1 Bit$ | $List_1.pos \leftarrow List_0.pos + 1$
| | $Bit.pos \leftarrow List_0.pos$
| | $List_0.val \leftarrow List_1.val + Bit.val$

- semantic rules define partial dependency graph
- value flow top down or across: *inherited attributes*
- value flow bottom-up: *synthesized attributes*
Note:  
- semantic rules associated with production $A \rightarrow \alpha$ have to specify the values for all segments:
  - synthesized attributes for $A$ (root)
  - inherited attributes for grammar symbols in $\alpha$ (children)
  ⇒ 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.
compute the decimal value of a signed binary number

For “–101”

If we show the computation ...

& then peel away the parse tree ...
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.

For “–101”

The dependence graph must be acyclic
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 **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
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: synthesized attributes only
- Evaluate in a single bottom-up pass

Good match to LR parsing

**Inherited Attributes**
- Use values from parent, constants, & siblings
- **L-attributed** grammars:
  \[ A \rightarrow X_1 X_2 \ldots X_n \]
  and each inherited attribute of \( X_i \) depends on
  - attributes of \( X_1 X_2 \ldots X_{i-1} \), and
  - inherited attributes of \( A \)
- Evaluate in a single top-down pass (left to right)

Good match for LL parsing

\[ S\text{-attributed} \subseteq L\text{-attributed} \]
Attribute Grammars

- 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
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
YACC: parse.y

parse.y:

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?

1. Define attributes as types in attr.h

```c
typedef struct info_node {int a; int b} infonode;
```

2. Include type attribute name in %union in parse.y

```c
%union {tokentype token; infonode myinfo; ... }
```

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

4. 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;}
```
More syntax-directed translation

Type checking

Symbol tables

Intermediate representations

Read EaC: Chapters 5.1 - 5.3