CS415 Compilers

Error Recovery

Context-Sensitive Analysis

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.

• Midterm sample solutions are available on sakai/Resources

  You have until **Thursday, April 11**, to challenge your grade. Please check with sample solution first if you have any questions.
The problem: parser encounters an invalid token
Goal: Want to parse the rest of the file

Basic idea (panic mode):
→ Assume something went wrong while trying to find handle for nonterminal A
→ Pretend handle for A has been found; pop “handle”, skip over input to find terminal that can follow A

Restarting the parser (panic mode):
→ find a restartable state on the stack (has transition for nonterminal A)
→ move to a consistent place in the input (token that can follow A)
→ perform (error) reduction (for nonterminal A)
→ print an informative message
Yacc’s (bison’s) error mechanism (note: version dependent!)

- designated token `error`
- used in error productions of the form
  \[ A \rightarrow \text{error } \alpha \] // basic case
- \( \alpha \) specifies synchronization points

When error is discovered

- pops stack until it finds state where it can shift the `error` token
- resumes parsing to match \( \alpha \)

  special cases:
  - \( \alpha = w \), where \( w \) is string of terminals: skip input until \( w \) has been read
  - \( \alpha = \varepsilon \): skip input until state transition on input token is defined
- error productions can have actions
Error Recovery in YACC

cmpdstmt: BEG stmt_list END
stmt_list : stmt
        | stmt_list ';' stmt
        | error { yyerror("\n***Error: illegal statement\n");}

This should
• throw out the erroneous statement
• synchronize at “;” or “end” (implicit: $\alpha = \varepsilon$)
• writes message “***Error: illegal statement” to stderr

Example: begin a & 5 | hello ; a := 3 end
         ↑             ↑ resume parsing
         ***Error: illegal statement
There is a level of correctness that is deeper than grammar

```
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 …)
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!
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

• Use ad-hoc techniques
  → Symbol tables
  → Ad-hoc code

*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)
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**

<table>
<thead>
<tr>
<th>Number</th>
<th>→</th>
<th>Sign List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sign</td>
<td>→</td>
<td>±</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>List</td>
<td>→</td>
<td>List Bit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bit</td>
</tr>
<tr>
<td>Bit</td>
<td>→</td>
<td>0</td>
</tr>
<tr>
<td></td>
<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
Example parse tree

compute the decimal value of a signed binary number

For “–101”
compute the decimal value of a signed binary number

For “–101”
compute the decimal value of a signed binary number

Inherited Attributes

For “–101”
Example parse tree

compute the decimal value of a signed binary number

For “–101”
compute the decimal value of a signed binary number

For “–101”

Example parse tree

Synthesized attributes
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>
</table>
| Number → Sign List | List.pos ← 0  
If Sign.neg  
thennumber.val ← − List.val  
else number.val ← List.val |
| Sign → + | Sign.neg ← false |
| | | |
| | Sign.neg ← true |
| List₀ → List₁ Bit | List₁.pos ← List₀.pos + 1  
Bit.pos ← List₀.pos  
List₀.val ← List₁.val + Bit.val |
| Bit | Bit.pos ← List.pos  
List.val ← Bit.val |
| Bit → 0 | Bit.val ← 0 |
| | |
| Bit → 1 | Bit.val ← 2^{Bit.pos} |

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

### Productions

<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
  - 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

If we show the computation ...

& then peel away the parse tree ...

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

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 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

\( S\)-attributed \( \subseteq \) \( L\)-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?
The Realist’s Alternative

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
parse.y:

List and assign attributes

Rules with semantic actions

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

- Include type attribute name in %union in parse.y
  
  ```c
  %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:

    ```c
    block : variables procdecls {$2.b = $1.b + 1;} cmpdstmt
           {$$.a = $1.a + $2.a + $4.b;}
    ```
Summary: Is This Really “Ad-hoc”?

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
More syntax-directed translation

Type checking

Symbol tables

Intermediate representations

Read EaC: Chapters 5.1 - 5.3