CS415 Compilers

Attribute Grammars, Syntax-Directed Translation

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
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
  \[ A \rightarrow \text{error} \alpha \quad // \text{basic case} \]
- \( \alpha \) 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”, not parts of speech
• Questions & answers involve non-local information
• Answers may involve computation

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)

• 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
Review - Attribute Grammars

What is an attribute grammar?

- A context-free grammar augmented with a set of rules
- Each symbol in the derivation has a set of values, or attributes
- The rules specify how to compute a value for each attribute

**Example grammar**

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Production</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></td>
<td></td>
</tr>
<tr>
<td>List</td>
<td>→ List Bit</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Bit</td>
<td>→ 0</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.
Example

compute the decimal value of a signed binary number

For “−101”
Example

compute the decimal value of a signed binary number

For “–101”
Example

compute the decimal value of a signed binary number

For “–101”
Example

compute the decimal value of a signed binary number

For “–101”
Example

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>
</table>
| Number → Sign List | List.pos ← 0  
If Sign.neg then Number.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.pos ← List.pos  
List.val ← Bit.val |
| Bit → 0 | Bit.val ← 0 |
| | 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

<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**
Attribute Grammars

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)
  $\Rightarrow$ 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

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.

For “–101”

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

For “−101”

All that is left is the attribute dependence graph.

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

The **dynamic methods** sort this graph to find independent values, then work along graph edges.

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, ...

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 → X₁ X₂ ... Xₙ and each inherited attribute of Xᵢ depends on
  - attributes of X₁ X₂ ... Xᵢ₋₁ , and
  - inherited attributes of A
• Evaluate in a single top-down pass (left to right)
Good match for LL parsing
### Grammar for a basic block

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Contextual Description</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>Let’s estimate cycle counts</td>
</tr>
<tr>
<td>Assign         → Ident = Expr ;</td>
<td>• Each operation has a COST</td>
</tr>
<tr>
<td>Expr&lt;sub&gt;0&lt;/sub&gt; → Expr&lt;sub&gt;1&lt;/sub&gt; + Term</td>
<td>• Add them, bottom up</td>
</tr>
<tr>
<td>→ Expr&lt;sub&gt;1&lt;/sub&gt; – Term</td>
<td>• Assume a load per value</td>
</tr>
<tr>
<td>→ Term</td>
<td>• Assume no reuse</td>
</tr>
<tr>
<td>Term&lt;sub&gt;0&lt;/sub&gt; → Term1 * Factor</td>
<td>Simple problem for an AG</td>
</tr>
<tr>
<td>→ Term1 / Factor</td>
<td></td>
</tr>
<tr>
<td>→ Factor</td>
<td></td>
</tr>
<tr>
<td>Factor → ( Expr )</td>
<td></td>
</tr>
<tr>
<td>→ Number</td>
<td></td>
</tr>
<tr>
<td>→ Identifier</td>
<td></td>
</tr>
</tbody>
</table>

Hey, this looks useful!
An Extended Example (continued)

These are all synthesized attributes!

Values flow from rhs to lhs in prod’ns
Properties of the example grammar

• All attributes are synthesized $\Rightarrow$ S-attributed grammar

• Rules can be evaluated bottom-up in a single pass
  $\rightarrow$ 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

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

| Factor $\rightarrow$ ( Expr ) | Factor.cost $\leftarrow$ Expr.cost ;
Expr.Before $\leftarrow$ Factor.Before ;
Factor.After $\leftarrow$ Expr.After |
|-----------------------------|----------------------------------|
| Number                      | Factor.cost $\leftarrow$ COST(loadi) ;
Factor.After $\leftarrow$ Factor.Before |
| Identifier                  | If (Identifier.name \notin Factor.Before) then |
|                             | Factor.cost $\leftarrow$ COST(load) ;
Factor.After $\leftarrow$ Factor.Before |
|                             | \(\cup\) Identifier.name |
|                             | else |
|                             | Factor.cost $\leftarrow$ 0 |
|                             | Factor.After $\leftarrow$ Factor.Before |

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

\[
\text{Expr}_0 \rightarrow \text{Expr}_1 + \text{Term}
\]
\[
\text{Expr}_0.\text{cost} \leftarrow \text{Expr}_1.\text{cost} + \\
\quad \text{COST}\text{(add)} + \text{Term.}\text{cost} ; \\
\text{Expr}_1.\text{Before} \leftarrow \text{Expr}_0.\text{Before} ; \\
\text{Term.}\text{Before} \leftarrow \text{Expr}_1.\text{After} ; \\
\text{Expr}_0.\text{After} \leftarrow \text{Term.}\text{After}
\]

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 first
  - Either search tree for answers or copy them to the root
What would a good programmer do (with the shift-reduce parser)?

- 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
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 \textit{lhs} & \textit{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)

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

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

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

<table>
<thead>
<tr>
<th>Grammar Rule</th>
<th>Description</th>
<th>Action Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start → Init Block</td>
<td></td>
<td>cost ← 0;</td>
</tr>
<tr>
<td>Init → ε</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Block₀ → Block₁ Assign</td>
<td></td>
<td>cost ← cost + COST(store);</td>
</tr>
<tr>
<td>Assign → Ident = Expr ;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... 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 ([marker production](#)!)
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

| Block₀ → Block₁ Assign | $$ ← \$1 + \$2 ;$$ $\leftarrow \$1 ;$
| Assign                   | $$ ← \$1 ;$$ $\leftarrow \$1 ;$
| Assign                   | $$ ← COST(store) + \$3 ;$$ $\leftarrow COST(store) + \$3 ;$
| Expr₀ → Expr₁ + Term    | $$ ← \$1 + COST(add) + \$3 ;$$ $\leftarrow \$1 + COST(add) + \$3 ;$
| Expr₁ - Term            | $$ ← \$1 + COST(sub) + \$3 ;$$ $\leftarrow \$1 + COST(sub) + \$3 ;$
| Term                     | $$ ← \$1 ;$$ $\leftarrow \$1 ;$
| Term₀ → Term₁ * Factor   | $$ ← \$1 + COST(mult) + \$3 ;$$ $\leftarrow \$1 + COST(mult) + \$3 ;$
| Term₁ / Factor           | $$ ← \$1 + COST(div) + \$3 ;$$ $\leftarrow \$1 + COST(div) + \$3 ;$
| Factor                   | $$ ← \$1 ;$$ $\leftarrow \$1 ;$
| Factor → ( Expr )        | $$ ← \$2 ;$$ $\leftarrow \$2 ;$
| Number                   | $$ ← COST(loadi);$$ $\leftarrow COST(loadi);$  
| Identifier               | { i ← hash(Identifier); 
|                          | if (Table[i].loaded = false) 
|                          | then { 
|                          | $$ ← COST(load);$$ $\leftarrow COST(load);$  
|                          | Table[i].loaded ← true; 
|                          | } 
|                          | else $$ ← 0$$ $\leftarrow 0$

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

However, Table[ ] still needs to be initialized
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\n",s);
}
int main() {
    printf("1\t");
    yyparse();
    return 1;
}
Project #2: Things to do

- Learn/Review the C programming language
- Add error productions (syntax errors)
- Define and assign attributes to non-terminals
- Implement single-level symbol table
- Perform type checking and produce required error messages; note: actions may occur at “any” location on right-hand side (implicit use of marker productions)
You do have to (slightly) 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 + \$3.b;}
    ```
Typing, Symbol Table, intermediate representations

Read EaC: Chapters 5.1 - 5.3, 5.5