Announcements

• Homework 6 due on Monday, April 9

• First project will be posted soon; please watch your 415 announcements
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 …)*
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>.
”,
            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.
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></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></td>
<td>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”
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

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></td>
<td>List.pos ← 0</td>
</tr>
<tr>
<td></td>
<td>If Sign.neg</td>
</tr>
<tr>
<td></td>
<td>then Number.val ← −List.val</td>
</tr>
<tr>
<td></td>
<td>else Number.val ← List.val</td>
</tr>
<tr>
<td><strong>Sign</strong></td>
<td>Sign.neg ← false</td>
</tr>
<tr>
<td></td>
<td>Sign.neg ← true</td>
</tr>
<tr>
<td><strong>List&lt;sub&gt;0&lt;/sub&gt;</strong></td>
<td>List&lt;sub&gt;1&lt;/sub&gt;.pos ← List&lt;sub&gt;0&lt;/sub&gt;.pos + 1</td>
</tr>
<tr>
<td></td>
<td>Bit.pos ← List&lt;sub&gt;0&lt;/sub&gt;.pos</td>
</tr>
<tr>
<td></td>
<td>List&lt;sub&gt;0&lt;/sub&gt;.val ← List&lt;sub&gt;1&lt;/sub&gt;.val + Bit.val</td>
</tr>
<tr>
<td><strong>List&lt;sub&gt;1&lt;/sub&gt;</strong></td>
<td>Bit.pos ← List&lt;sub&gt;1&lt;/sub&gt;.pos</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td>Bit.pos ← List.pos</td>
</tr>
<tr>
<td></td>
<td>List.val ← Bit.val</td>
</tr>
<tr>
<td><strong>Bit</strong></td>
<td>Bit.val ← 0</td>
</tr>
<tr>
<td></td>
<td>Bit.val ← 2&lt;sup&gt;Bit.pos&lt;/sup&gt;</td>
</tr>
</tbody>
</table>

<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>$List_0$</th>
<th>$List_1$ Bit</th>
</tr>
</thead>
</table>

#### Attribution Rules

- $List_1.pos \leftarrow List_0.pos + 1$
- $Bit.pos \leftarrow List_0.pos$
- $List_0.val \leftarrow List_1.val + Bit.val$

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

For “–101”
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

S-attributed ⊆ L-attributed

**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
An Extended Example

Grammar for a basic block

\[
\begin{align*}
    \text{Block}_0 & \rightarrow \text{Block}_1 \text{ Assign} \\
    \text{Assign} & \rightarrow \text{Ident} = \text{Expr} ; \\
    \text{Expr}_0 & \rightarrow \text{Expr}_1 + \text{Term} \\
    & \rightarrow \text{Expr}_1 - \text{Term} \\
    & \rightarrow \text{Term} \\
    \text{Term}_0 & \rightarrow \text{Term}_1 \ast \text{Factor} \\
    & \rightarrow \text{Term}_1 / \text{Factor} \\
    & \rightarrow \text{Factor} \\
    \text{Factor} & \rightarrow ( \text{Expr} ) \\
    & \rightarrow \text{Number} \\
    & \rightarrow \text{Identifier}
\end{align*}
\]

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)

These are all synthesized attributes!

Values flow from rhs to lhs in prod'ns

<table>
<thead>
<tr>
<th>Block_0</th>
<th>Block_1 Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assign</td>
</tr>
<tr>
<td>Assign</td>
<td>Ident = Expr ;</td>
</tr>
<tr>
<td>Expr_0</td>
<td>Expr_1 + Term</td>
</tr>
<tr>
<td></td>
<td>Expr_1 - Term</td>
</tr>
<tr>
<td></td>
<td>Term</td>
</tr>
<tr>
<td>Term_0</td>
<td>Term_1 * Factor</td>
</tr>
<tr>
<td></td>
<td>Term_1 / Factor</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr )</td>
</tr>
<tr>
<td></td>
<td>Number</td>
</tr>
<tr>
<td></td>
<td>Identifier</td>
</tr>
<tr>
<td>Block_0.cost</td>
<td>Block_1.cost + Assign.cost</td>
</tr>
<tr>
<td>Block_0.cost</td>
<td>Assign.cost</td>
</tr>
<tr>
<td>Assign.cost</td>
<td>COST(store) + Expr.cost</td>
</tr>
<tr>
<td>Expr_0.cost</td>
<td>Expr_1.cost + COST(add) + Term.cost</td>
</tr>
<tr>
<td>Expr_0.cost</td>
<td>Expr_1.cost + COST(add) + Term.cost</td>
</tr>
<tr>
<td>Expr_0.cost</td>
<td>Term.cost</td>
</tr>
<tr>
<td>Term_0.cost</td>
<td>Term_1.cost + COST(mult) + Factor.cost</td>
</tr>
<tr>
<td>Term_0.cost</td>
<td>Term_1.cost + COST(div) + Factor.cost</td>
</tr>
<tr>
<td>Term_0.cost</td>
<td>Factor.cost</td>
</tr>
<tr>
<td>Factor.cost</td>
<td>Expr.cost</td>
</tr>
<tr>
<td>Factor.cost</td>
<td>COST(loadl)</td>
</tr>
<tr>
<td>Factor.cost</td>
<td>COST(load)</td>
</tr>
</tbody>
</table>
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 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

| 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

<table>
<thead>
<tr>
<th>Expr₀</th>
<th>Expr₁ + Term</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Expr₀.cost ← Expr₁.cost + COST(add) + Term.cost ;</td>
</tr>
<tr>
<td></td>
<td>Expr₁.Before ← Expr₀.Before ;</td>
</tr>
<tr>
<td></td>
<td>Term.Before ← Expr₁.After ;</td>
</tr>
<tr>
<td></td>
<td>Expr₀.After ← Term.After</td>
</tr>
</tbody>
</table>

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 \((\text{hashing, § B.4})\)
  - 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 \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)

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

<table>
<thead>
<tr>
<th>Block₀</th>
<th>Block₁ Assign</th>
<th>Assign</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>{ }</td>
</tr>
<tr>
<td>Assign</td>
<td></td>
<td>{ }</td>
</tr>
<tr>
<td>Expr₀</td>
<td>Expr₁ + Term</td>
<td>{cost← cost + COST(add);}</td>
</tr>
<tr>
<td></td>
<td>Expr₁ - Term</td>
<td>{cost← cost + COST(sub);}</td>
</tr>
<tr>
<td></td>
<td>Term</td>
<td>{ }</td>
</tr>
<tr>
<td>Term₀</td>
<td>Term₁ * Factor</td>
<td>{cost← cost + COST(mult);}</td>
</tr>
<tr>
<td></td>
<td>Term₁ / Factor</td>
<td>{cost← cost + COST(div);}</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>{ }</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr )</td>
<td>{ }</td>
</tr>
<tr>
<td></td>
<td>Number</td>
<td>{cost← cost + COST(loadi);}</td>
</tr>
<tr>
<td></td>
<td>Identifier</td>
<td>i← hash(Identifier);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if (Table[i].loaded = false)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>then {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>cost ← cost + COST(load);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Table[i].loaded ← true;</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)*!
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  |   | Assign          |   | { $$ ← $1 + $2 ;} |
|          |   | Assign          |   |                |   | { $$ ← $1 ;}       |
| Assign   | → | Ident = Expr ; |   |                  |   | { $$ ← COST(store) + $3; } |
| Expr₀    | → | Expr₁ + Term   |   |                  |   | { $$ ← $1 + COST(add) + $3; } |
|          |   | Expr₁ - Term   |   |                  |   | { $$ ← $1 + COST(sub) + $3; } |
|          |   | Term            |   |                  |   | { $$ ← $1; }        |
| Term₀    | → | Term₁ * Factor |   |                  |   | { $$ ← $1 + COST(mult) + $3; } |
|          |   | Term₁ / Factor |   |                  |   | { $$ ← $1 + COST(div) + $3; } |
|          |   | Factor          |   |                  |   | { $$ ← $1; }        |
| Factor   | → | ( Expr )       |   |                  |   | { $$ ← $2; }        |
|          |   | Number          |   |                  |   | { $$ ← COST(loadi); } |
|          |   | Identifier      |   |                  |   | { i ← hash(Identifier); |
|          |   |                 |   |                  |   | if (Table[i].loaded = false) |
|          |   |                 |   |                  |   | then { |
|          |   |                 |   |                  |   | $$ ← COST(load); |
|          |   |                 |   |                  |   | Table[i].loaded ← true; |
|          |   |                 |   |                  |   | } |
|          |   |                 |   |                  |   | else $$ ← 0 |
|          |   |                 |   |                  |   | } |
- 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>{ $$ = MakeAddNode($1,$3); }</td>
</tr>
<tr>
<td></td>
<td>Expr - Term</td>
<td>{ $$ = MakeSubNode($1,$3); }</td>
</tr>
<tr>
<td></td>
<td>Term</td>
<td>{ $$ = $1; }</td>
</tr>
<tr>
<td>Term</td>
<td>Term * Factor</td>
<td>{ $$ = MakeMulNode($1,$3); }</td>
</tr>
<tr>
<td></td>
<td>Term / Factor</td>
<td>{ $$ = MakeDivNode($1,$3); }</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>{ $$ = $1; }</td>
</tr>
<tr>
<td>Factor</td>
<td>( Expr )</td>
<td>{ $$ = $2; }</td>
</tr>
<tr>
<td></td>
<td>number</td>
<td>{ $$ = MakeNumNode(token); }</td>
</tr>
<tr>
<td></td>
<td>id</td>
<td>{ $$ = MakeIdNode(token); }</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: 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
parse.y:

```c
%{
#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\n");
    yyparse();
    return 1;
}
```

Will be included verbatim in `parse.tab.c`

Rules with semantic actions

Main program and "helper" functions; may contain initialization code of global structures. Will be included verbatim in `parse.tab.c`

List and assign attributes
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`:

```
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/spring18/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

assumes table is global
Type checking

Symbol tables

Intermediate representations

Read EaC: Chapters 5.1 - 5.3