These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University
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
• 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.
Review: Cycle Count Example

Grammar for a basic block

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

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!
# Review: Cycle Count Example

<table>
<thead>
<tr>
<th>Block₀ → Block₁ Assign</th>
<th>Block₀.cost ← ( Block₁.cost + ) Assign.cost Assign.cost ← Assign.cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign → Ident = Expr ;</td>
<td>Assign.cost ← ( \text{COST}(\text{store}) + ) Expr.cost</td>
</tr>
<tr>
<td>Expr₀ → Expr₁ + Term</td>
<td>Expr₀.cost ← ( \text{Expr₁.cost} + ) ( \text{COST}(\text{add}) + ) Term.cost</td>
</tr>
<tr>
<td></td>
<td>Expr₀.cost ← ( \text{Expr₁.cost} + ) ( \text{COST}(\text{sub}) + ) Term.cost</td>
</tr>
<tr>
<td></td>
<td>Expr₀.cost ← Term.cost</td>
</tr>
<tr>
<td>Term₀ → Term₁ * Factor</td>
<td>Term₀.cost ← ( \text{Term₁.cost} + ) ( \text{COST}(\text{mult}) + ) Factor.cost</td>
</tr>
<tr>
<td></td>
<td>Term₀.cost ← ( \text{Term₁.cost} + ) ( \text{COST}(\text{div}) + ) Factor.cost</td>
</tr>
<tr>
<td></td>
<td>Term₀.cost ← Factor.cost</td>
</tr>
<tr>
<td>Factor → ( Expr )</td>
<td>Factor.cost ← ( \text{Expr.cost} )</td>
</tr>
<tr>
<td></td>
<td>Factor.cost ← ( \text{COST}(\text{loadI}) )</td>
</tr>
<tr>
<td></td>
<td>Factor.cost ← ( \text{COST}(\text{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 \( \Rightarrow \) 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**

<table>
<thead>
<tr>
<th>Production</th>
<th>Factor statements</th>
</tr>
</thead>
</table>
| `Factor → ( Expr )`                                                      | `Factor.cost ← Expr.cost ;`  
|                                                                          | `Expr.Before ← Factor.Before ;`  
|                                                                          | `Factor.After ← Expr.After`  
|                                                                          | `Number`  
|                                                                          | `Factor.cost ← COST(loadi) ;`  
|                                                                          | `Factor.After ← Factor.Before`  
|                                                                          | `Identifier`  
|                                                                          | `If (Identifier.name ∉ Factor.Before)`  
|                                                                          | `then`  
|                                                                          | `Factor.cost ← COST(load) ;`  
|                                                                          | `Factor.After ← Factor.Before`  
|                                                                          | `∪ Identifier.name`  
|                                                                          | `else`  
|                                                                          | `Factor.cost ← 0`  
|                                                                          | `Factor.After ← 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

| Expr₀ \(\rightarrow\) Expr₁ + Term | Expr₀.cost \(\leftarrow\) Expr₁.cost + COST(add) + Term.cost ;
|                                          | Expr₁.Before \(\leftarrow\) Expr₀.Before ;
|                                          | Term.Before \(\leftarrow\) Expr₁.After;
|                                          | Expr₀.After \(\leftarrow\) Term.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
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
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 (parser generator)
- 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

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

Start → Init Block
Init → ε
Block₀ → Block₁ Assign
Assign → Ident = Expr ;

... 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)

<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>
</tbody>
</table>

$$ \leftarrow \$1 + \$2 ; 
$$ \leftarrow \$1 ; 
$$ \leftarrow \text{COST(store)} + \$3 ; 
$$ \leftarrow \$1 + \text{COST(add)} + \$3 ; 
$$ \leftarrow \$1 + \text{COST(sub)} + \$3 ; 
$$ \leftarrow \$1 ; 
$$ \leftarrow \$1 + \text{COST(mult)} + \$3 ; 
$$ \leftarrow \$1 + \text{COST(div)} + \$3 ; 
$$ \leftarrow \$1 ; 
$$ \leftarrow \$2 ; 
$$ \leftarrow \text{COST(loadi)} ; 
\{ i \leftarrow \text{hash(Identifier)} ; 
\quad \text{if (Table}[i].\text{loaded} = \text{false)} ; 
\quad \text{then} \{ 
\quad \quad $$ \leftarrow \text{COST(load)} ; 
\quad \quad \text{Table}[i].\text{loaded} \leftarrow \text{true} ; 
\quad \quad \} 
\quad \text{else} $$ \leftarrow 0 
\quad \} 

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
- Assume LR(1) parser stack holds pointers to nodes
- Assume yacc syntax

<table>
<thead>
<tr>
<th>Expr</th>
<th>Term</th>
<th>Factor</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expr +</td>
<td>Expr -</td>
<td>( Expr )</td>
<td>Expr</td>
</tr>
<tr>
<td>Term</td>
<td>Term *</td>
<td>number</td>
<td>$1;</td>
</tr>
<tr>
<td></td>
<td>Factor</td>
<td>id</td>
<td>$1;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$1;</td>
</tr>
</tbody>
</table>

$$ = \text{MakeAddNode}(\$1,\$3);$$
$$ = \text{MakeSubNode}(\$1,\$3);$$
$$ = \text{MakeMulNode}(\$1,\$3);$$
$$ = \text{MakeDivNode}(\$1,\$3);$$
$$ = \text{MakeNumNode}(\text{token});$$
$$ = \text{MakeIdNode}(\text{token});$$
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 ($A \rightarrow \beta$)

• Need a naming scheme to access them
  → $n$ translates into stack location: top - $3 \times (|\beta| - n)$

• 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

Example:

\[
A \rightarrow a_1a_2a_3a_4 \quad --- \text{need to insert action rule after } a_2
\]

Converted to:

\[
A \rightarrow Ba_3a_4
\]
\[
B \rightarrow a_1a_2 \quad (\text{insert the action here})
\]
**Typical Uses (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

*table is global*
Is This Really “Ad-hoc”? 

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
Types and Type Systems

**Types**

*Type:* A set of values and meaningful operations on them

Types provide semantic “sanity checks” (consistency checks) and determine efficient implementations for data objects.

Types help identify:

→ errors, if an operator is applied to an incompatible operand
  ▪ dereferencing of a non-pointer
  ▪ adding a function to something
  ▪ incorrect number of parameters to a procedure
  ▪ …

→ which operation to use for overloaded names and operators, or what type coercion to use (e.g.: 3.0 + 1)

→ identification of polymorphic functions
**Type system**: Each language construct (operator, expression, statement, ...) is associated with a type expression. The type system is a collection of rules for assigning type expressions to these constructs.

**Type expressions** for

- basic types: integer, char, real, boolean, typeError
- constructed types, e.g., one-dimensional arrays:
  \[ \text{array}(lb, \ub, \text{elem\_type}) \], where elem\_type is a type expression

A type checker implements a type system. It computes or “constructs” type expressions for each language construct.
Types and Type Systems

Example type inference rule:

\[
\begin{array}{c}
E \vdash e_1 : \text{integer} \\
E \vdash e_2 : \text{integer}
\end{array} \quad \Rightarrow \quad E \vdash (e_1 + e_2) : \text{integer}
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

Questions: How to specify rules that allow type coercion (type widening) from integers to reals in arithmetic expressions?

\[ 3.0 + 1 \quad \text{or} \quad 1 + 3.0 \]
Example type inference rule pointer dereferencing:

\[
\frac{\text{E} \vdash \text{e} : ???}{\text{E} \vdash *\text{e} : ???}
\]

where $E$ is a type environment that maps constants and variables to their type expressions.
Example type inference rule pointer dereferencing:

\[
 E \vdash e : \text{pointer(integer)} \\
\]

\[
 E \vdash *e : \text{integer} \\
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

\text{pointer(…)} is part of the \textit{type expression language} such as \text{array(…)}. 
Example type inference rule pointer dereferencing:

\[ E \vdash e : \text{pointer}(\beta) \]
\[ \Rightarrow E \vdash *e : \beta \]

where \( E \) is a type environment that maps constants and variables to their type expressions.

Type expressions may also contain type variables such as \( \beta \). Type variables can denote any type expression.

Type variables are needed to express polymorphic types.
Example type inference rule address computation:

\[
E \vdash e : \text{integer} \\
\hline
E \vdash \&e : ??? 
\]

where \( E \) is a type environment that maps constants and variables to their type expressions.

What about a polymorphic version of this rule?
Formal proof that a program can be typed correctly.

```c
int a;
...
...*(&a) + 3...
```
Programmers may define their own types and give them names:

```plaintext
type my_int is int;
...
int a;
my_int b;
...
... a + b ...
```

*Type names can also be part of the type expression language.*

Note: *type names and type variables are different!*
Type Equivalence

**Structural** -- type equivalence: **type names** are expanded

**Name** -- type equivalence: **type names** are not expanded

Example:

```
type A is array(1..10) of integer;
type B is array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

**Answer:** structural equivalence:

name equivalence:
**Type Equivalence**

**Structural** -- type equivalence: type names are expanded

**Name** -- type equivalence: type names are not expanded

Example:

```plaintext
type A is array(1..10) of integer;
type B is array(1..10) of integer;
a : A;
b : B;
c, d: array(1..10) of integer;
e: array(1..10) of integer;
```

**Answer:** structural equivalence: \((a, b, c, d, e)\)

name equivalence: \((a); (b); (c, d, e)\);
Revisit our type inference rule for “+”.

```c
exp : exp ’+’ exp { if ($1 == integer && $3 == integer)
    $$ = integer;
    else {
        $$ = typeError;
        printf(“\n***Error: illegal operand types\n”);
    }
}
```

PROJECT HINT: The definition of type expression as C types (structs) should be done in `attr.h`. `attr.c` may contain helper functions. The assignment of type expression C types to terminals and nonterminals of the grammar is done in `parse.y`.
Intermediate Representation
Read EaC Chapter 5.1 – 5.5