Compilers Course
Lecture 7: Attribute Grammars

Attribute Grammars are a way of expressing computations on parse trees and abstract syntax trees.

An Attribute Grammar (AG) consists of:

- A context-free grammar (CFG).
- Declarations stating that certain symbols in the grammar have attributes (named values) of certain types.
- For each production, a set of equations relating the attributes of the symbols in that production.

Example: expression evaluator

ICON.val:int
E.val:int

E → ICON
    E.val = ICON.val
E → E1+E2
    E.val = E1.val + E2.val
E → E1*E2
    E.val = E1.val + E2.val
E → (E1)
    E.val = E1.val

Given a parse tree, an attribute grammar results in a set of equations:

- Each node is given a unique name
- Each node is given attribute variables named after the node
- Each internal node in the parse tree corresponds to some production: the AG equations for that production are instantiated for that particular node (attributes are named), and the equations are added to the global set of equations
- The attributes of terminals are typically predetermined.
- The global set of equations is solved (if possible)

For example, 2*(3+4) will result in

E0.val = E1.val * E2.val
E1.val = ICON(2).val
E2.val = E3.val
E3.val = E4.val + E5.val
E4.val = ICON(3).val
E5.val = ICON(4).val

Solving this gives E0.val = 2*7 = 14.

It is often simpler to just draw the parse tree, add attributes to each node in the tree, and then solve the equations.

In this example the attribute values "flow" from the leaves up to internal nodes, where new values are computed which flow further up in the tree. These simple kinds of attributes are called "Synthesized Attributes".

Inherited Attributes

It is possible to have attribute values flow from the top of the tree down towards the leaves. These types of attributes are called "Inherited Attributes".

Example:

```
ICON.val:int
ID.name:string
E.val:int
E.env:string → int     // environment mapping identifiers to values

E → let ID = E1 in E2
    E1.env = E.env
    E2.env = E.env[ID.name → E1.val] // add binding to environment
    E.val = E2.val
E → ID
    E.val = E.env[ID.name]          // look up binding in environment
E → ICON
    E.val = ICON.val
```

Now, to evaluate an expression like "let X = 5 in X", the environment attribute flows *down* the tree, and is extended at "let" expressions. It is then used to compute the values of the leaves, which then flow *up* towards the root.

In this AG, "env" is an inherited attribute and "val" is a synthesized attribute.

L-attributed grammars

Mixing inherited and synthesized attributes can cause cycles in the equation system, making it unsolvable.

If the attributes can be evaluated using a single top-down left-to-right traversal of the tree (visit node, visit children left-to-right, visit node again) then the AG is called "L-attributed" (L=left).

L-attributed AGs are significant because they can be evaluated during parsing.

Syntax-directed translation

An AG can be used to compute almost anything.

Typically, AGs are used for type checking, construction of abstract syntax trees, and even simple code generation.

Using AGs to express these processes is called Syntax-Directed Translation.

Constructing ASTs

In modern compilers we want to decouple type checking and code generation from parsing, so we only want the parser to construct an AST. Using an AG makes this easy:

E.ast: the type of expr ast nodes

E → ICON
    E.ast = mk_icon(ICON.value)
E → E1 + E2
    E.ast = mk_add(E1.ast, E2.ast)
E → (E1)
    E.ast = E1.ast

So when parsing, all we need is a way to associate a synthesized attribute with each symbol, and a way to execute a "construct and assign" statement in each derivation step.

Note that we can still use AGs to formalize type-checking and code generation: we just apply the AGs to the AST instead of the parse tree.