Program Transformation and Generation

Lecture 4
Attribute Grammars in Eli

February 26, 2008

Anthony M. Sloane
Programming Languages Research Group
Department of Computing, Macquarie University
Sydney, Australia
http://www.comp.mq.edu.au/plrg/

1

Attribute Grammars

A formal specification mechanism based on context-free grammars, tree structures and semantic equations.

Designed to replace hand-coded tree processing code with automatically-generated evaluators.

Eli

A comprehensive language processor generation system that uses attribute grammars as its main specification formalism.

2

Finding out more about the Eli System

Slides from ETAPS 2007 tutorial (on course website).

Open source downloads of the system and detailed documentation can be obtained from:
http://sourceforge.net/projects/eli-project

The main attribute grammar component of Eli is implemented by the LIGA tool from Uwe Kastens at the University of Paderborn.

3

Overview

Introduction to attribute grammars in Eli via a simple example.

A more complex example that inspires specification notation short-hands.

Demo of Eli (if time).

Demo of Eli (if time).

Attribute evaluation methods, particularly tree-walkers defined by a statically computed schedule.

4

Attribute Grammars

Originally developed by Knuth as a formal semantics definition method for programming languages, extending the context-free grammars commonly used to specify syntax.

Basic idea

Represent the information to be analysed as a tree whose structure conforms to a context-free grammar.

The semantics of a construct is given by the values of attributes associated with the node representing the construct.

Describe how to compute the attribute values by giving semantic equations that define them in terms of other attribute values.

5

Context

Program Transformation and Generation, Lecture 4, Anthony M Sloane

Program Transformation and Generation, Lecture 4, Anthony M Sloane

Program Transformation and Generation, Lecture 4, Anthony M Sloane

Program Transformation and Generation, Lecture 4, Anthony M Sloane
A Simple Attribute Grammar

```
ATTR value : int;
RULE: Number ::= Digit_Seq COMPUTE
    Number.value = Digit_Seq.value;
END;
RULE: Digit_Seq ::= Digit COMPUTE
    Digit_Seq.value = Digit;
END;
RULE: Digit_Seq ::= Digit_Seq Digit COMPUTE
    Digit_Seq[1].value = ADD (MUL (Digit_Seq[2].value, 10), Digit);
END;
```

Adding Bases

```
ATTR value, base : int;
RULE: Number ::= Digit_Seq Base_Tag COMPUTE
    Number.value = Digit_Seq.value;
    Digit_Seq.base = Base_Tag.value;
END;
RULE: Digit_Seq ::= Digit COMPUTE
    Digit_Seq.value = Digit;
END;
RULE: Digit_Seq ::= Digit_Seq Digit COMPUTE
    Digit_Seq[1].value = ADD (MUL (Digit_Seq[2].value, Digit_Seq[1].base), Digit);
    Digit_Seq[2].base = Digit_Seq[1].base;
END;
RULE: Base_Tag ::= 'O' COMPUTE
    Base_Tag.value = 8;
END;
RULE: Base_Tag ::= 'D' COMPUTE
    Base_Tag.value = 10;
END;
```

Synthesized and Inherited Attributes

Synthesized attributes

Defined at the node in terms of attributes of the node or its children (flow upward).

```
RULE: Digit_Seq ::= Digit COMPUTE
    Digit_Seq.value = Digit;
END;
```

Inherited attributes

Defined at the node’s parent in terms of attributes of the node’s parent or siblings (flow downward).

```
RULE: Number ::= Digit_Seq Base_Tag COMPUTE
    Digit_Seq.base = Base_Tag.value;
END;
```

Context Conditions

Boolean conditions that must be satisfied for a tree to conform to the semantics.

Older systems supported them by a special construct, as in

```
RULE: Digit_Seq ::= Digit COMPUTE
    CONDITION GE (Digit, Digit_Seq.base);
END;
```

but Eli just uses an expression that is not assigned to an attribute.

```
RULE: Digit_Seq ::= Digit COMPUTE
    IF (GE (Digit, Digit_Seq.base),
        message (ERROR, "illegal digit", 0, COORDREF));
END;
```

Side-Effects

As originally defined, attribute grammars are declarative and the expressions defining attributes must be referentially transparent (i.e., with no side-effects).

While formally clean, this decision makes it harder to efficiently implement aggregate attribute values (e.g., symbol tables) and to integrate I/O. (The same trade-offs occur when dealing with aggregates and I/O in functional languages.)

For this reason, Eli allows side-effects in computations.

The downside is that more care must be taken to ensure that computations happen in the desired order when value dependencies do not constrain the order of evaluation.

VOID attributes

Eli’s VOID attributes do not carry a value and hence do not incur a run-time cost. They exist solely to express non-value dependencies.

```
RULE: Number ::= Digit_Seq Base_Tag COMPUTE
    printf("Number.value = %d in base %d\n", Number.value, Base_Tag.value);
END;
RULE: Digit_Seq ::= Digit COMPUTE
    printf("Digit = %d\n", Digit);
END;
RULE: Digit_Seq ::= Digit_Seq Digit COMPUTE
    printf("Digit = %d\nDigit_Seq[1].value = %d\n", Digit, Digit_Seq[1].value);
END;
```
VOID attributes

Eli’s VOID attributes do not carry a value and hence do not incur a run-time cost. They exist solely to express non-value dependencies.

```plaintext
ATTR print : VOID;
```

```plaintext
RULE: Number ::= Digit_Seq Base_Tag COMPUTE
printf("Number.value = %d in base %d
\n", Number.value, Base_Tag.value)
DEPENDS ON Digit_Seq.print;
END;
```

```plaintext
RULE: Digit_Seq ::= Digit COMPUTE
Digit_Seq.print =
printf("Digit = %d
\n", Digit);
END;
```

```plaintext
RULE: Digit_Seq ::= Digit_Seq Digit COMPUTE
Digit_Seq[1].print =
printf("Digit = %d
Digit_Seq[1].value = %d
\n", Digit, Digit_Seq[1].value)
DEPENDS ON Digit_Seq[2].print;
END;
```

**A More Complex Problem**

Compute table layout information for HTML tables to render them in a browser.

We can’t output a table rendition as we see the table input because row heights and column widths depend on the entire contents of the table.

Represent the table as a tree and compute attributes to hold the layout information.

Example from Chapter Four of "Generating Software from Specifications" by Kastens, Sloane and Waite.

**Table Formatting Example**

```html
<TABLE>
  <TR><TD>Name</TD>   <TD>Lectures</TD>   <TD>Office Hours</TD></TR>
  <TR><TD>Kastens</TD> <TD>
    <TABLE>
      <TR><TD>175110</TD><TD>Software Development I</TD></TR>
      <TR><TD>175528</TD><TD>Programming Languages</TD></TR>
      <TR><TD>175775</TD><TD>Project Program Analysis</TD></TR>
    </TABLE>
  </TD>   <TD><TABLE>
    <TR><TD>Mon</TD><TD>11:30-12:15</TD></TR>
    <TR><TD>Thu</TD><TD>14:00-15:30</TD></TR>
  </TABLE></TD></TR>
  <TR><TD>Waite</TD> <TD>
    <TABLE>
      <TR><TD>ECEN 5523</TD><TD>Compiler Construction Tools</TD></TR>
      <TR><TD>ECEN 4553</TD><TD>Compiler Construction</TD></TR>
    </TABLE>
  </TD>   <TD><TABLE>
    <TR><TD>MWF</TD><TD>10-11</TD></TR>
    <TR><TD>M</TD><TD>3-4</TD></TR>
    <TR><TD>Th</TD><TD>2-3:30</TD></TR>
    <TR><TD>F</TD><TD>1-2</TD></TR>
  </TABLE></TD></TR>
</TABLE>
```

**HTML Table Structure**

```plaintext
RULE: Document ::= Table END;
RULE: Table ::= Rows END;
RULE: Rows ::= LISTOF Row END;
RULE: Row ::= Cells END;
RULE: Cells ::= LISTOF Cell END;
RULE: DataCell ::= Cell ::= Data END;
RULE: TableCell ::= Cell ::= Table END;
```

**Attribution (1)**

Computing Cell heights

**CONSTITUENTS** is short-hand for a "fold" in a sub-tree.

```plaintext
ATTR height : int;
RULE: Cell ::= Data COMPUTE Cell.height = 1;  END;
RULE: Cell ::= Table COMPUTE Cell.height = Table.height; END;
SYMBOL Row COMPUTE
SYNT.height = CONSTITUENTS Cell.height SHIELD Table
WITH (int, Maximum, IDENTICAL, ZERO);
END;
SYMBOL Table COMPUTE
SYNT.height = CONSTITUENTS Row.height SHIELD Table
WITH (int, AddPlus1, IDENTICAL, ZERO);
END;
```

**Attribution (2)**

Calculating vertical positions

A **CHAIN** is short-hand for a depth-first left-to-right threaded attribute.

**CHAIN down** is:

```plaintext
RULE: Document ::= Table COMPUTE
CHAINSTART Table.down = 0;
END;
RULE: Table ::= Rows COMPUTE
CHAINSTART HEAD.down = THIS.down;
END;
RULE: Row COMPUTE
THIS.down = ADD (THIS.down, THIS.height);
END;
```

```plaintext
RULE: DataCell COMPUTE
CHAINEND Tаблиц = THIS.down + THIS.height;
END;
RULE: TableCell COMPUTE
CHAINSTART HEAD.down = THIS.down;
CHAINSTART HEAD.height = THIS.height;
END;
```

```plaintext
SYMBOL Row COMPUTE
SYNT.height = CONSTITUENTS Cell.height SHIELD Table
WITH (int, Maximum, IDENTICAL, ZERO);
END;
```
Attribute Evaluation Strategies

Notes:
- static schedule vs dynamic
- S-attributed, L-attributed, multi-visit
- tree walking
- use examples from before

Evaluation

A hand-coded solution of the HTML table problem using the Visitor pattern yields around 280 lines of C++ to do the traversal and 184 lines to implement the tree structure.

In contrast, the Eli-based solution requires just 125 lines of specification for the attribution and 7 lines for the tree structure. Moreover, it’s faster and more memory efficient.

We don’t want the programmer to have to work out
- a precise traversal strategy that is compatible with the dependencies between computations, and
- a strategy for efficient storage of attribute values.

Static Scheduling

At generation time, analyse the dependencies between attributes and devise a traversal strategy that walks the tree in a compatible order.

The aim is to have a traversal strategy that works for any tree (that conforms to the context-free grammar).

Pros
- Low runtime cost, since schedule is pre-determined.

Cons
- Restriction on class of attribute grammar that can be handled in this way.

Attribute Dependence Cycles

An attribute that depends on itself is a problem and usually indicates an error in the specification.

Systems based on static analysis of attribute dependencies will usually reject any grammar that contains a cycle.

In some cases, cycles make some sense. E.g., when specifying data flow analysis in an iterative fashion where a value of an attribute at one step of the iteration depends on its value in an earlier iteration.

Some attribute grammar systems provide a mechanism for repeated evaluation of attributes that can handle this situation.

S-attributed Grammars

If an attribute grammar only has synthesized attributes (and no cycles), we can use a simplified traversal.

The dependencies only flow within a single node or up the tree, so a simple bottom-up traversal will suffice.

In fact, if a bottom-up parsing method is used, attributes can be evaluated entirely during parsing and no tree needs to be built.
L-attributed Grammars

Another useful class of grammars are the L-attributed where each synthesized attribute of a left-hand side symbol depends only on that symbol's inherited attributes or on any attribute of right-hand side symbols, and each inherited attribute of a right-hand side symbol depends only on inherited attributes of the left-hand side symbol or on any attribute of symbols to its left in the right-hand side.

L-attributed grammars are suited to evaluation during a top-down, recursive descent parse of the input, so no tree needs to be built.

Ordered Attribute Grammars

Most real grammars are not S- or L-attributed.

Ordered Attribute Grammars are the most well-known class of attribute grammar for which a static schedule can be efficiently computed, and which contains a sufficiently large proportion of the attribute grammars that arise in practice (at least for static analysis of programming languages).

Originally developed in the 1980s by Uwe Kastens for the GAG system, OAGs are now embodied in the LIGA attribute grammar system used by Eli.

Direct Dependencies

Direct dependencies are due to the semantic equations associated with a single context-free grammar rule.

RULE Number ::= Digit_Seq Base_Tag COMPUTE
Digit_Seq_base = Base_Tag.value;
END;

INDIRECT DEPENDENCY FOR RULE rule_1 #22 AT ROW 5; COL 5
ATNO  CLASS  PART              SYMB.ATTR                DEPENDS ON
   0    SYNT      0             Number.value                  [  2 ]
   1    INH       0          Digit_Seq.base                   [  4 ]
   2    SYNT      0          Digit_Seq.value                  [ ]
   3    SYNT      0          Digit_Seq.print                  [ ]
   4    SYNT      0           Base_Tag.value                  [ ]
END RULE

Induced Dependencies

Induced dependencies are due to the transitive closure of the direct dependencies across all context-free grammar rules and symbols.

RULE Digit_Seq ::= Digit COMPUTE
Digit_Seq_value = Digit;
END;
RULE Digit_Seq ::= Digit_Seq Digit COMPUTE
Digit_Seq[1].value = ADD (MUL (Digit_Seq[2].value, Digit_Seq[1].base), Digit);
Digit_Seq[2].base = Digit_Seq[1].base;
END;

INDUCED DEPENDENCY FOR SYMBOL Digit_Seq #19 AT ROW 5; COL 19
ATNO  CLASS  PART              SYMB.ATTR                DEPENDS ON
   0    INH     0          Digit_Seq.base                   [ ]
   1    SYNT    0          Digit_Seq.value                  [  0 ]
   2    SYNT    0          Digit_Seq.print                  [ ]
   3    SYNT    0           Base_Tag.value                  [ ]
END

In an OAG, the dependencies must be consistent to define a single order of evaluation for the attributes of a single symbol.

Visit Sequences

Visit sequences describe how to visit a node that is defined by a particular context-free grammar rule. A sequence comprises:

- evaluation of synthesized attributes of the current node,
- evaluation of inherited attributes of children nodes,
- visits to children nodes, and
- visits to the parent node.

RULE Number ::= Digit_Seq Base_Tag COMPUTE
Number.value = Digit_Seq.value;
Digit_Seq_base = Base_Tag.value;
END;

VISIT-SEQUENCE FOR RULE rule_1 #22 AT ROW 5; COL 5
NO KIND  SYMBNO VISITNO  SYMBOL         ATTRIBUTE or FCT
   1 VISIT      2       1  Base_Tag
   2 EVAL       1          Digit_Seq      base
   3 VISIT      1       1  Digit_Seq
   4 EVAL       0          Number         value
   5 LEAVE      0       1  TO ANCESTOR

Tree Walkers

Visit sequences can easily be turned into tree walkers, with one procedure per visit sequence. Calls to children are dispatched on the rule number, so as to obtain the correct visit procedure. Inherited attributes can be passed as parameters.

void _VS1rule_1(_TPPrule_1 _currn)
{
    int _AS1base;
    (VS1MAP[_currn->_desc2->_prod])(_currn->_desc2);
    _AS1base=_AVBase_Tag_value;
    (VS1MAP[_currn->_desc1->_prod])(_currn->_desc1,(&( _AS1base)));
    _AVNumber_value=_AVDigit_Seq_value;
}

In an OAG, the dependencies must be consistent to define a single order of evaluation for the attributes of a single symbol.
Attribute Storage

The other job of the attribute grammar system is to determine the best locations for attribute storage.

There are three main options:

- In the tree nodes: simple and works for all attributes, but wastes storage since all attribute values are maintained at the same time.
- In global variables: suitable for attributes whose multiple instances have lifetimes that do not overlap (e.g., DigitSeq.value).
- In stacks: suitable for attributes whose multiple instances have lifetimes that nest. Often implemented by local variables in the visit procedures (e.g., base attribute).

Dynamic Scheduling

An alternative to static scheduling is to wait until run-time and dynamically decide the order of attribute evaluation.

Pros

- A larger class of attribute grammar can be accommodated.
- Can be simpler to implement, particularly in lazy functional languages.

Cons

- Run-time overhead in time and space to decide what to do next and to keep track of what has been done.

Pros and Cons of Attribute Grammars

Pros

- Allow developer to concentrate on specifying their semantics rather than on details of traversal and storage and therefore resilient to evolution.
- Ideally suited for analysis of a static tree structure.
- Ok for relatively large scale transformation and generation problems.

Cons

- Awkward for problems consisting of many small transformations.

Overview of Next Week

Attribute grammars for transformation and generation: computing programs as attributes, example for WebDSL.
Static semantic analysis for WebDSL using attribute grammars.
Reusable attribution modules.