Semantic Analysis – Attribute Grammars

Martin Sulzmann

Syntax versus Semantics

Syntax Analysis

- When is a program syntactically valid?
- Formalism: Regular expressions + context-free grammars

Semantic Analysis

- What is the meaning of a program?
- Mostly context-*sensitive* conditions:
  - Proper variable declarations
  - Type correctness
  - Dynamic method dispatch
  - ...

Attribute Grammars (AGs)

Objective

A tool for simplifying semantic analysis (Knuth 1968).

Approach

Decorate CFG rules with actions (*semantic rules*).

Example

Rule	Action
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow E_1 - T$	$E.val = E_1.val - T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow (E)$	$T.val = E.val$
$T \rightarrow num$	$T.val = num.vale$

For us mainly construction of AST, something we yet have to discuss

Attributes + Semantic Rules

AGs = Attributes + Semantic Rules

$$\underbrace{E \;\rightarrow\; E_1 - T}_{CFG\;\; Rule} \qquad \underbrace{E.val = E_1.val - T.val}_{Semantic\;\; Rule}$$

- Decorated CFG.
- Nonterminals may have attributes, e.g. $E.val$.
- Define meaning of attributes via semantic rules.

Semantic Rules

Examples

- Semantic rules involve computations:

$$E \rightarrow E_1 - T \quad E.val = E_1.val - T.val$$

- Semantic rules involve copy operations:

$$E \rightarrow T \quad E.val = T.val$$

Two Types of Attributes

Semantic Rule

For each CFG rule $A \rightarrow \alpha_1 ... \alpha_n$. some semantic rule $b = f(c_1, ..., c_k)$.

Synthesized Attributes

- Belongs to left-hand side nonterminal (Context-insensitive)
- For example, b is a synthesized attribute of A and $c_1, ..., c_k$ are attributes of $\alpha_1, ..., \alpha_n$.
- Terminal symbols usually have synthesized attributes only (e.g. consider *id*, *num*).

Inherited Attributes

- Belongs to right-hand side nonterminal (Context-sensitive).
- For example, b is an inherited attribute of some α_i and $c_1, ..., c_k$ are attributes of $A, \alpha_1, ..., \alpha_n$. Hence, b depends on $c_1, ..., c_k$.

Synthesized Attributes

Example

- Attribute *val* belongs to left-hand side.

$$
\begin{array}{ll}
E \to E_1 + T & E.val = E_1.val + T.val \\
E \to T & E.val = T.val \\
T \to T_1 * F & T.val = T_1.val * F.val \\
T \to F & T.val = F.val \\
F \to num & F.val = num.value
\end{array}
$$

- Evaluate "3*5+4"
- A definition that only uses syntesized attributes is an *S-attributed* definition.

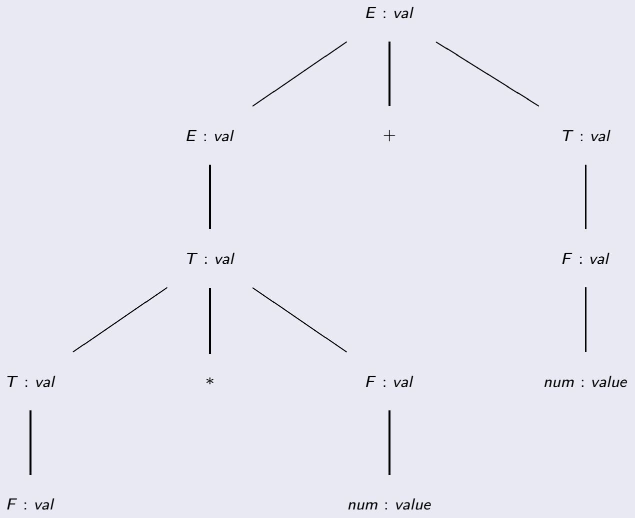
Evaluation of Synthesized Attributes

Observations

- Evaluation of semantic rules during parsing.
- S-attributed definition can be evaluated as soon as right-hand side ("Handle") is recognized.
- Corresponds to LR parsing strategy!

Evaluation Tree

Example



AST Construction with S-Attributes

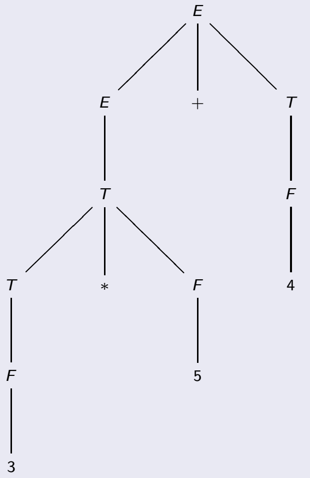
Example

$$
\begin{array}{ll}
E \rightarrow E_1 + T & E.node = mknode('+', E_1.node, T.node) \\
E \rightarrow T & E.node = T.node \\
T \rightarrow T_1 * F & T.node = mknode('*', T_1.node, F.node) \\
T \rightarrow F & T.node = F.node \\
F \rightarrow num & F.node = mkleaf(num, num.entry)
\end{array}
$$

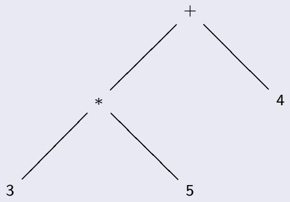
From Parse Tree to AST

Example

Parse Tree



AST



yacc Example

yacc Grammar with Semantic Rules

```
lines : lines line
| line
;
line : expr CR  {printf("= %d\n",$1); }
;
expr : expr PLUS term  {$$ = $1 + $3; }
| term
;
term : term TIMES factor {$$ = $1 * $3; }
| factor
;
factor : LPAREN expr RPAREN {$$ = $2;}
| INT
;
```

S-Attributes: Short Summary

Observations

- Evaluation of S-attributed definition corresponds to LR parsing strategy.
- yacc supports S-attributed definitions.
- Main application: AST construction
- Aside: Dynamic versus static semantics.
  - Static semantics: First generate AST for later (static) processing.
  - Dynamic semantics: See above yacc specification for a simple interpreter.

Inherited Attributes

Example

- C style type declarations such as "int id1,id2".
- CFG + semantic rules:

$$
\begin{array}{llll}
D & \rightarrow & TL & L.in = T.type \\
T & \rightarrow & int & T.type = int \\
T & \rightarrow & real & T.type = real \\
L & \rightarrow & L_1 id & L_1.in = L.in;\ addtype(id.entry, L.in) \\
L & \rightarrow & id & addtype(id.entry, L.in)
\end{array}
$$

- *in* is a *inherited* attribute as the corresponding nonterminal appears on the right-hand side.
- Evaluation based on LL parsing strategy

L-Attributed Definitions

Definition

- CFG rule $A \to X_1 ... X_n$
- The following holds
  - Inherited attributes of X_i depend only on:
    1. inherited attributes of A
    2. attributes of $X_1, ..., X_{i-1}$
  - Synthesized attributes of A depend only on its inherited attributes and arbitrary rhs attributes.
  - Syntesized attributes of an action depends only on its inherited attributes.
- Evaluation order:
  $Inh(A), Inh(X_1), Syn(X_1), ..., Inh(X_n), Syn(X_n), Syn(A)$.
- This is precisely the order of evaluation for an LL parser!

Left-Recursion and Semantic Rules

Objective

- Eliminate left recursion (so we can apply LL parser).
- Transform semantic rules accordingly.

Example (incomplete)

$E \to E_1 + T$	$E.val = E_1.val + T.val$
$E \to T$	$E.val = T.val$
$T \to (E)$	$T.val = E.val$
$T \to num$	$T.val = num.value$
$E \to TR$	???
$R \to +TR_1$	???
$R \to \epsilon$	???
$T \to (E)$	
$T \to num$	

Left-Recursion and Semantic Rules (2)

The Trick

$$
\begin{array}{ll}
E \rightarrow TR & R.in = T.val;\; E.val = R.val \\
R \rightarrow +TR_1 & R_1.in = R.in + T.val;\; R.val = R_1.val \\
R \rightarrow \epsilon & R.val = R.in \\
T \rightarrow (E) & \\
T \rightarrow num &
\end{array}
$$

- Via *in* propagate type info "down" the parse tree.

Summary

Attributed Grammars

- Domain-specific (language) formalismus for semantic analysis.
- S-attributed definitions $\Rightarrow$ LR parser.
- L-attributed definitions $\Rightarrow$ LL parser.

Application ("old school")

Formalize compilation (or interpretation) process via semantic rules (type checking, intermediate code, optimizations, ...).

Today

- Use semantic rules to generate AST.
- Semantic analysis phases operate on AST written in general-purpose languages such as OCaml.