Semantic Analysis

Attribute Grammars
Semantic Analysis
From Code Form To Program Meaning

Source Code

Compiler or Interpreter
Translation
Execution

Target Code

Interpretation
Phases of Compilation

- Character stream
- Token stream
- Parse tree
- Abstract syntax tree or other intermediate form
- Modified intermediate form
- Assembly or machine language, or other target language
- Modified target language
- Symbol table

- Scanner (lexical analysis)
- Parser (syntax analysis)
- Semantic analysis and intermediate code generation
- Machine-independent code improvement (optional)
- Target code generation
- Machine-specific code improvement (optional)
Specification of Programming Languages

• PLs require precise definitions (i.e. no ambiguity)
  - Language form (Syntax)
  - Language meaning (Semantics)

• Consequently, PLs are specified using formal notation:
  - Formal syntax
    • Tokens
    • Grammar
  - Formal semantics
    • Attribute Grammars (static semantics)
    • Dynamic Semantics
The Semantic Analyzer

- The principal job of the semantic analyzer is to enforce static semantic rules.
- In general, anything that requires the compiler to compare things that are separate by a long distance or to count things ends up being a matter of *semantics*.
- The semantic analyzer also commonly constructs a syntax tree (usually first), and much of the information it gathers is needed by the code generator.
Attribute Grammars

• Context-Free Grammars (CFGs) are used to specify the syntax of programming languages
  – *E.g.* arithmetic expressions

• How do we tie these rules to mathematical concepts?

• *Attribute grammars* are annotated CFGs in which *annotations* are used to establish meaning relationships among symbols
  – Annotations are also known as decorations

```
E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F
F → - F
F → ( E )
F → const
```
Attribute Grammars

Example

• Each grammar symbols has a set of attributes
  – E.g. the value of \( E_1 \) is the attribute \( E_1.val \)

• Each grammar rule has a set of rules over the symbol attributes
  – Copy rules
  – Semantic Function rules
    • E.g. sum, quotient

\[
\begin{align*}
1: \ E_1 & \rightarrow E_2 + T \\
\quad & \triangleright \ E_1.val := \text{sum} \ (E_2.val, T.val) \\
2: \ E_1 & \rightarrow E_2 - T \\
\quad & \triangleright \ E_1.val := \text{difference} \ (E_2.val, T.val) \\
3: \ E & \rightarrow T \\
\quad & \triangleright \ E.val := T.val \\
4: \ T_1 & \rightarrow T_2 \ * \ F \\
\quad & \triangleright \ T_1.val := \text{product} \ (T_2.val, F.val) \\
5: \ T_1 & \rightarrow T_2 / F \\
\quad & \triangleright \ T_1.val := \text{quotient} \ (T_2.val, F.val) \\
6: \ T & \rightarrow F \\
\quad & \triangleright \ T.val := F.val \\
7: \ F_1 & \rightarrow - F_2 \\
\quad & \triangleright \ F_1.val := \text{additive\_inverse} \ (F_2.val) \\
8: \ F & \rightarrow ( \ E ) \\
\quad & \triangleright \ F.val := E.val \\
9: \ F & \rightarrow \text{const} \\
\quad & \triangleright \ F.val := \text{const.val}
\end{align*}
\]
Attribute Flow

- Context–free grammars are not tied to an specific parsing order
  - *E.g.* Recursive descent, LR parsing
- Attribute grammars are not tied to an specific evaluation order
  - This evaluation is known as the *annotation* or *decoration* of the parse tree
Attribute Flow

Example

• The figure shows the result of annotating the parse tree for 
(1+3) * 2

• Each symbol has at most one attribute shown in the corresponding box
  – Numerical value in this example
  – Operator symbols have no value

• Arrows represent attribute flow
Attribute Flow Example

1: $E_1 \rightarrow E_2 + T$
   ▷ $E_1.val := \text{sum} \ (E_2.val, T.val)$

2: $E_1 \rightarrow E_2 - T$
   ▷ $E_1.val := \text{difference} \ (E_2.val, T.val)$

3: $E \rightarrow T$
   ▷ $E.val := T.val$

4: $T_1 \rightarrow T_2 * F$
   ▷ $T_1.val := \text{product} \ (T_2.val, F.val)$

5: $T_1 \rightarrow T_2 / F$
   ▷ $T_1.val := \text{quotient} \ (T_2.val, F.val)$

6: $T \rightarrow F$
   ▷ $T.val := F.val$

7: $F_1 \rightarrow - F_2$
   ▷ $F_1.val := \text{additive\_inverse} \ (F_2.val)$

8: $F \rightarrow ( E )$
   ▷ $F.val := E.val$

9: $F \rightarrow \text{const}$
   ▷ $F.val := \text{const.val}$
Attribute Flow

Synthetic and Inherited Attributes

• In the previous example, semantic information is pass up the parse tree
  – We call this type of attributes are called *synthetic attributes*
  – Attribute grammar with synthetic attributes only are said to be *S-attributed*

• Semantic information can also be passed down the parse tree
  – Using *inherited attributes*
  – Attribute grammar with inherited attributes only are said to be *non-S-attributed*
Attribute Flow
Inherited Attributes

- *L-attributed* grammars, such as the one on the next slide, can still be evaluated in a single left-to-right pass over the input.
- Each synthetic attribute of a LHS symbol (by definition of *synthetic*) depends only on attributes of its RHS symbols.
- Each inherited attribute of a RHS symbol (by definition of *L-attributed*) depends only on inherited attributes of the LHS symbol or on synthetic or inherited attributes of symbols to its left in the RHS.
- Top-down grammars generally require non-S-attributed flows
  - The previous annotated grammar was an S-attributed LR(1)
  - L-attributed grammars are the most general class of attribute grammars that can be evaluated during an LL parse.
LL Grammar

1: $E \rightarrow T \ TT$
   $\triangleright\ TT\text{.st} := T\text{.val}$  $\triangleright\ E\text{.val} := TT\text{.val}$

2: $TT_1 \rightarrow + \ T \ TT_2$
   $\triangleright\ TT_2\text{.st} := TT_1\text{.st} + T\text{.val}$  $\triangleright\ TT_1\text{.val} := TT_2\text{.val}$

3: $TT_1 \rightarrow - \ T \ TT_1$
   $\triangleright\ TT_2\text{.st} := TT_1\text{.st} - T\text{.val}$  $\triangleright\ TT_1\text{.val} := TT_2\text{.val}$

4: $TT \rightarrow \epsilon$
   $\triangleright\ TT\text{.val} := TT\text{.st}$

5: $T \rightarrow F \ FT$
   $\triangleright\ FT\text{.st} := F\text{.val}$  $\triangleright\ T\text{.val} := FT\text{.val}$

6: $FT_1 \rightarrow \ast \ F \ FT_2$
   $\triangleright\ FT_2\text{.st} := FT_1\text{.st} \times F\text{.val}$  $\triangleright\ FT_1\text{.val} := FT_2\text{.val}$

7: $FT_1 \rightarrow / \ F \ FT_2$
   $\triangleright\ FT_2\text{.st} := FT_1\text{.st} \div F\text{.val}$  $\triangleright\ FT_1\text{.val} := FT_2\text{.val}$

8: $FT \rightarrow \epsilon$
   $\triangleright\ FT\text{.val} := FT\text{.st}$

9: $F_1 \rightarrow - \ F_2$
   $\triangleright\ F_1\text{.val} := - F_2\text{.val}$

10: $F \rightarrow ( \ E \ )$
    $\triangleright\ F\text{.val} := E\text{.val}$

11: $F \rightarrow \text{const}$
    $\triangleright\ F\text{.val} := \text{const\text{.val}}$
Non-S-Attributed Grammars

Example

1. $E \rightarrow T \cdot TT$
   $\triangleright T, st := T, val$
   $\triangleright E, val := TT, val$

2. $TT_1 \rightarrow + T \cdot TT_2$
   $\triangleright TT_2, st := TT_1, st + T, val$
   $\triangleright TT_1, val := TT_2, val$

3. $TT_1 \rightarrow - T \cdot TT_1$
   $\triangleright TT_2, st := TT_1, st - T, val$
   $\triangleright TT_1, val := TT_2, val$

4. $TT \rightarrow \epsilon$
   $\triangleright TT, val := TT, st$

5. $T \rightarrow F \cdot FT$
   $\triangleright FT, st := F, val$
   $\triangleright T, val := FT, val$

6. $FT_1 \rightarrow \times F \cdot FT_2$
   $\triangleright FT_2, st := FT_1, st \times F, val$
   $\triangleright FT_1, val := FT_2, val$

7. $FT_1 \rightarrow / F \cdot FT_2$
   $\triangleright FT_2, st := FT_1, st \div F, val$
   $\triangleright FT_1, val := FT_2, val$

8. $FT \rightarrow \epsilon$
   $\triangleright FT, val := FT, st$

9. $F_1 \rightarrow - F_2$
   $\triangleright F_1, val := - F_2, val$

10. $F \rightarrow ( E )$
    $\triangleright F, val := E, val$

11. $F \rightarrow \text{const}$
    $\triangleright F, val := \text{const}, val$
Syntax Tree

- There is considerable variety in the extent to which parsing, semantic analysis, and intermediate code generation are interleaved.
- A *one-pass* compiler interleaves scanning, parsing, semantic analysis, and code generation in a single traversal of the input.
- A common approach interleaves construction of a syntax tree with parsing (eliminating the need to build an explicit parse tree), then follows with separate, sequential phases for semantic analysis and code generation.
Bottom-up Attribute Grammar to Construct a Syntax Tree

\[
\begin{align*}
E_1 & \rightarrow E_2 + T \\
& \quad \triangleright \text{E}_1.\text{ptr} := \text{make_bin_op ("+", E}_2.\text{ptr, T}.\text{ptr)} \\
E_1 & \rightarrow E_2 - T \\
& \quad \triangleright \text{E}_1.\text{ptr} := \text{make_bin_op ("-", E}_2.\text{ptr, T}.\text{ptr)} \\
E & \rightarrow T \\
& \quad \triangleright \text{E}.\text{ptr} := \text{T}.\text{ptr} \\
T_1 & \rightarrow T_2 \times F \\
& \quad \triangleright \text{T}_1.\text{ptr} := \text{make_bin_op ("\times", T}_2.\text{ptr, F}.\text{ptr)} \\
T_1 & \rightarrow T_2 / F \\
& \quad \triangleright \text{T}_1.\text{ptr} := \text{make_bin_op ("\div", T}_2.\text{ptr, F}.\text{ptr)} \\
T & \rightarrow F \\
& \quad \triangleright \text{T}.\text{ptr} := \text{F}.\text{ptr} \\
F_1 & \rightarrow - F_2 \\
& \quad \triangleright \text{F}_1.\text{ptr} := \text{make_unop ("+-", F}_2.\text{ptr)} \\
F & \rightarrow ( E ) \\
& \quad \triangleright \text{F}.\text{ptr} := \text{E}.\text{ptr} \\
F & \rightarrow \text{const} \\
& \quad \triangleright \text{F}.\text{ptr} := \text{make_leaf (const.val)}
\end{align*}
\]
Construction of the Syntax Tree

\[
\begin{align*}
E_1 & \rightarrow E_2 + T \\
& \quad \triangleright E_1.\text{ptr} := \text{make_bin_op} ("+", E_2.\text{ptr}, T.\text{ptr}) \\
E_1 & \rightarrow E_2 - T \\
& \quad \triangleright E_1.\text{ptr} := \text{make_bin_op} ("-", E_2.\text{ptr}, T.\text{ptr}) \\
E & \rightarrow T \\
& \quad \triangleright E.\text{ptr} := T.\text{ptr} \\
T_1 & \rightarrow T_2 * F \\
& \quad \triangleright T_1.\text{ptr} := \text{make_bin_op} ("\times", T_2.\text{ptr}, F.\text{ptr}) \\
T_1 & \rightarrow T_2 / F \\
& \quad \triangleright T_1.\text{ptr} := \text{make_bin_op} ("\div", T_2.\text{ptr}, F.\text{ptr}) \\
T & \rightarrow F \\
& \quad \triangleright T.\text{ptr} := F.\text{ptr} \\
F_1 & \rightarrow - F_2 \\
& \quad \triangleright F_1.\text{ptr} := \text{make_un_op} ("+/-", F_2.\text{ptr}) \\
F & \rightarrow ( E ) \\
& \quad \triangleright F.\text{ptr} := E.\text{ptr} \\
F & \rightarrow \text{const} \\
& \quad \triangleright F.\text{ptr} := \text{make_leaf} (\text{const.val})
\end{align*}
\]
Action Routines

- Automatic tools can construct a parser for a given context-free grammar
  - *E.g.* yacc
- Automatic tools can construct a semantic analyzer for an attribute grammar
  - An ad hoc techniques is to annotate the grammar with executable rules
  - These rules are known as *action routines*
Action Rules for the Previous LL(1) attribute grammar

\[
E \Rightarrow T \{ \text{TT} . st := T . v \} \text{TT} \{ E . v := \text{TT} . v \}
\]
\[
TT \Rightarrow + T \{ \text{TT}2 . st := \text{TT}1 . st + T . v \} \text{TT} \{ \text{TT}1 . v := \text{TT}2 . v \}
\]
\[
TT \Rightarrow - T \{ \text{TT}2 . st := \text{TT}1 . st - T . v \} \text{TT} \{ \text{TT}1 . v := \text{TT}2 . v \}
\]
\[
TT \Rightarrow \{ \text{TT} . v := \text{TT} . st \}
\]
\[
T \Rightarrow F \{ \text{FT} . st := F . v \} \text{FT} \{ T . v := \text{FT} . v \}
\]
\[
FT \Rightarrow * F \{ \text{FT}2 . st := \text{FT}1 . st * F . v \} \text{FT} \{ \text{FT}1 . v := \text{FT}2 . v \}
\]
\[
FT \Rightarrow / F \{ \text{FT}2 . st := \text{FT}1 . st / F . v \} \text{FT} \{ \text{FT}1 . v := \text{FT}2 . v \}
\]
\[
FT \Rightarrow \{ \text{FT} . v := \text{FT} . st \}
\]
\[
F \Rightarrow - F \{ F1 . v := - F2 . v \}
\]
\[
F \Rightarrow ( E ) \{ F . v := E . v \}
\]
\[
F \Rightarrow \text{const} \{ F . v := C . v \}
\]
Action Rules

• The ease with which rules were incorporated in the grammar is due to the fact that the attribute grammar is \textit{L–attributed}.

• The action rules for \textit{L–attributed} grammars, in which the attribute flow is depth–first left–to–right, can be evaluated in the order of the parse tree prediction for LL grammars.

• Action rules for \textit{S–attributed} grammars can be incorporated at the end of the right–hand sides of LR grammars. But, if action rules are responsible for a significant part of the semantic analysis, they will need more contextual information to do their job.
Static and Dynamic Semantics

- Attribute grammars add basic semantic rules to the specification of a language
  - They specify *static semantics*
- But they are limited to the semantic form that can be checked at compile time
- Other semantic properties cannot be checked at compile time
  - They are described using *dynamic semantics*
Dynamic Semantics

- Use to formally specify the behavior of a programming language
  - Semantic-based error detection
  - Correctness proofs
- There is not a universally accepted notation
  - Operational semantics
    - Executing statements that represent changes in the state of a real or simulated machine
  - Axiomatic semantics
    - Using predicate calculus (pre and post-conditions)
  - Denotational semantics
    - Using recursive function theory
Semantic Specification

• The most common way of *specifying* the semantics of a language is plain English
• There is a lack of formal rigor in the semantic specification of programming languages