5. Semantic Analysis

Mircea Lungu
Oscar Nierstrasz

Thanks to Jens Palsberg and Tony Hosking for their kind permission to reuse and adapt the CS132 and CS502 lecture notes.
http://www.cs.ucla.edu/~palsberg/
http://www.cs.purdue.edu/homes/hosking/
William Tell is a folk hero of Switzerland; she was an exceptional marksman.
Conference in Vienna in 1964 best summarized by T. B. Steel:

“I don’t fully know myself how to describe the semantics of a language. I daresay nobody does or we wouldn’t be here”

The Genesis of Atribute Grammars
Donald E. Knuth
Roadmap

> Context-sensitive analysis
> Strategies for semantic analysis
> Attribute grammars
> Symbol tables and type-checking

Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- Attribute grammars
- Symbol tables and type-checking
{ 
  ...) 
  x := y[1]; 
  z := x + y; 
  ...) 
}
Compilation is *driven by the syntactic structure of the program as discovered by the parser*.

**Semantic routines:**

— interpret meaning of the program based on its syntactic structure

— two purposes:
  - *finish analysis* by deriving context-sensitive information
  - *begin synthesis* by generating the IR or target code

— associated with individual productions of a context free grammar or sub-trees of a syntax tree
Context-sensitive analysis

*What context-sensitive questions might the compiler ask?*

1. Is \( x \) scalar, an array, or a function?
2. Is \( x \) declared before it is used?
3. Are any names declared but not used?
4. Which declaration of \( x \) is being referenced?
5. Is an expression type-consistent?
6. Does the dimension of a reference match the declaration?
7. Where can \( x \) be stored? (heap, stack, ...)
8. Does \( *p \) reference the result of a `malloc()`?
9. Is \( x \) defined before it is used?
10. Is an array reference in bounds?
11. Does function `foo` produce a constant value?
12. Can \( p \) be implemented as a memo-function?

*These questions cannot be answered with a context-free grammar*
Context-sensitive analysis

> What are the challenges?
   — questions and answers involve non-local information
   — answers depend on values, not syntax
   — answers may involve computation

> Several approaches:
   — *symbol tables*: central store for facts; express checking code
   — *attribute grammars*: specify non-local computations; automatic evaluators
   — *language design*: simplify language; avoid problems
Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- Attribute grammars
- Symbol tables and type-checking
Alternatives for semantic processing

- One-pass compiler and synthesis
- Two-pass
  - compiler + peephole
  - compiler & IR synthesis + code generation pass
- Multi-pass
  - analysis
  - synthesis
One-pass compilers

> Interleave scanning, parsing and translation
  — no explicit IR
  — generate target code directly
    - emit short sequences of instructions on each parser action
    - little or no optimization possible (minimal context)

> Can add peephole optimization pass
  — extra pass over generated code through small window (“peephole”) of instructions
  — smooths out “rough edges” between code emitted by subsequent calls to code generator
Two-pass: analysis & IR synthesis + code generation

> Generate explicit IR as interface to code generator
  — linear (e.g., tuples)
  — can emit multiple tuples at a time for better code context

> Advantages
  — easier retargeting (IR must be expressive enough for different machines!)
  — can add optimization pass later (multi-pass synthesis)
Multi-pass analysis

> Several passes, read/write intermediate files
1. scan source file, generate tokens
   - *place identifiers and constants in symbol table*
2. parse token file
   - *generate semantic actions or linearized parse tree*
3. process declarations to symbol table
4. semantic checking with IR synthesis

> Motivations:
   — Historical: constrained address spaces
   — Language: e.g., declaration after use
   — Multiple analyses over IR tree
Multi-pass synthesis

> Passes operate on linear or tree-structured IR
> Options:
  — code generation and peephole optimization
  — multi-pass IR transformation
    - machine-independent then dependent optimizations
  — high-level to low-level IR transformation before code generation
    - e.g., in gcc high-level trees drive generation of low-level Register Transfer Language for machine-independent optimization
  — language-independent front ends
  — retargetable back ends
Roadmap

> Context-sensitive analysis
> Strategies for semantic analysis
> **Attribute grammars**
> Symbol tables and type-checking
> Add attributes to the syntax tree or PEG:
   — can add attributes (fields) to each node
   — specify equations to define values
   — propagate values up (synthesis) or down (inheritance)

> Example: ensuring that constants are immutable
   — add *type* and *class* attributes to expression nodes
   — add rules to production for `:=`
     1. *check that LHS.class is variable (not constant)*
     2. *check that LHS.type and RHS.type are compatible*
Attribute grammar actions

<table>
<thead>
<tr>
<th>PRODUCTION</th>
<th>SEMANTIC RULES</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D \rightarrow TL$</td>
<td>$L.in := T.type$</td>
</tr>
<tr>
<td>$T \rightarrow \text{int}$</td>
<td>$T.type := \text{integer}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{real}$</td>
<td>$T.type := \text{real}$</td>
</tr>
<tr>
<td>$L \rightarrow L_1, \text{id}$</td>
<td>$L_1.in := L.in$</td>
</tr>
<tr>
<td>$L \rightarrow \text{id}$</td>
<td>$\text{addtype(id.entry, L.in)}$</td>
</tr>
<tr>
<td>$\text{id}$</td>
<td>$\text{addtype(id.entry, L.in)}$</td>
</tr>
</tbody>
</table>

- tree attributes specified by grammar
- productions associated with attribute assignments
- each attribute defined uniquely and locally
- identical terms are labeled uniquely
Example: evaluating signed binary numbers

Attributed parse tree for -101

- **val** and **neg** are *synthetic* attributes
- **pos** is an *inherited* attribute
Attribute dependency graph

- **nodes** represent **attributes**
- **edges** represent **flow of values**
- graph must be **acyclic**
- **topologically sort** to order attributes
  — use this order to evaluate rules
  — order depends on both grammar and input string!

1. SIGN.neg 8. BIT0.val
2. LIST0.pos 9. LIST2.val
3. LIST1.pos 10. BIT1.val
4. LIST2.pos 11. LIST1.val
5. BIT0.pos 12. BIT2.val
6. BIT1.pos 13. LIST0.val
7. BIT2.pos 14. NUM.val

Evaluating in this order yields NUM.val = -5
Evaluation strategies

- **Parse-tree methods**
  1. build the parse tree
  2. build the dependency graph
  3. topologically sort the graph
  4. evaluate it

- **Rule-based methods**
  1. analyse semantic rules at compiler-construction time
  2. determine static ordering for each production’s attributes
  3. evaluate its attributes in that order at compile time

- **Oblivious methods**
  1. ignore the parse tree and the grammar
  2. choose a convenient order (e.g., left-to-right traversal) and use it
  3. repeat traversal until no more attribute values can be generated
Attribute grammars in practice

> **Advantages**
  — clean formalism
  — automatic generation of evaluator
  — high-level specification

> **Disadvantages**
  — evaluation strategy determines efficiency
  — increase space requirements
  — parse tree evaluators need dependency graph
  — results distributed over tree
  — circularity testing

Historically, attribute grammars have been judged too large and expensive for industrial-strength compilers.
Roadmap

- Context-sensitive analysis
- Strategies for semantic analysis
- Attribute grammars
- Symbol tables and type-checking
For compile-time efficiency, compilers often use a *symbol table*:— associates lexical *names* (symbols) with their *attributes*

What items should be entered?
— variable names
— constants
— procedure and function names
— literal constants and strings
— compiler-generated temporaries (we’ll get there)

Separate table of structure layouts for types (field offsets and lengths)

*A symbol table is a compile-time structure*
Symbol table information

> What kind of information might the compiler need?
  — textual name
  — data type
  — dimension information (*for aggregates*)
  — declaring procedure
  — **lexical level** of declaration
  — storage class (*heap, stack, text …*)
  — offset in storage
  — if record, pointer to structure table
  — if parameter, by-reference or by-value?
  — can it be aliased? to what other names?
  — number and type of arguments to functions
Lexical Scoping

With *lexical scoping* the definition of a name is determined by its *static scope*. A stack suffices to track the current definitions.

```
class C {
    int x;
    void m(int y) {
        int z;
        if (y>x) {
            int w=z+y;
            return w;
        }
        return y;
    }
}
```
Nested scopes: block-structured symbol tables

> What information is needed?
— when we ask about a name, we want the most recent declaration
— the declaration may be from the current scope or some enclosing scope
— innermost scope overrides declarations from outer scopes

> Key point: new declarations (usually) occur only in current scope

> What operations do we need?
— void put(Symbol key, Object value) — bind key to value
— Object get(Symbol key) — return value bound to key
— void beginScope() — remember current state of table
— void endScope() — restore table to state at most recent scope that has not been ended

May need to preserve list of locals for the debugger
Checking variable declarations in a hierarchical symbol table

```c
int x=1;
{
    int y = x;
    x = x+y;
}
{
    y = x - y;
}
```
Efficient Implementation of Symbol Tables

Implementation options
1. functional
2. imperative

How to ensure efficiency, with thousands of distinct identifiers in a large program?
Hash tables support an imperative (destructive) implementation.

```plaintext
int foo, bar;
foo = ++bar;
if (bar > 10) then
{
  boolean baz;
  baz = true;
}

// and assume
hash(foo) = hash(bar)
hash(baz) = hash(quux)
```
(Balanced) binary trees support a functional (non-destructive) implementation. A persistence data structure.
Attribute information

> Attributes are internal representations of declarations

> Symbol table associates names with attributes

> Names may have different attributes depending on their meaning:
  — *variables*: type, procedure level, frame offset
  — *types*: type descriptor, data size/alignment
  — *constants*: type, value
  — *procedures*: formals (names/types), result type, block information (local decls.), frame size
Static and Dynamic Typing

A language is **statically typed** if it is always possible to determine the (static) type of an expression based on the program text alone.

A language is **dynamically typed** if only values have fixed type. Variables and parameters may take on different types at run-time, and must be checked immediately before they are used.

A language is “strongly typed” if it is impossible to perform an operation on the wrong kind of object.

Type consistency may be assured by

I. compile-time type-checking,
II. type inference, or
III. dynamic type-checking.

See: Programming Languages course
Type expressions

Type expressions are a textual representation for types:

1. basic types: `boolean`, `char`, `integer`, `real`, etc.
2. type names
3. constructed types (constructors applied to type expressions):
   a) `array(I,T)` denotes `array` of elements type `T`, index type `I`
      e.g., `array (1...10, integer)`
   b) `T_1 \times T_2` denotes `Cartesian product` of type expressions `T_1` and `T_2`
   c) `record(…)` denotes `record` with named fields
      e.g., `record((a \times integer), (b \times real))`
   d) `pointer(T)` denotes the type “`pointer` to object of type `T`”
   e) `D \rightarrow R` denotes type of `function` mapping domain `D` to range `R`
      e.g., `integer \times integer \rightarrow integer`
Type descriptors are compile-time structures representing type expressions.

e.g., char × char → pointer(integer)
Type compatibility

Type checking needs to determine type equivalence

Two approaches:

> **Name equivalence**: each type name is a distinct type

> **Structural equivalence**: two types are equivalent iff they have the same structure (after substituting type expressions for type names)

— $s \equiv t$ iff $s$ and $t$ are the same basic types

— $\text{array}(s_1, s_2) \equiv \text{array}(t_1, t_2)$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$

— $s_1 \times s_2 \equiv t_1 \times t_2$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$

— $\text{pointer}(s) \equiv \text{pointer}(t)$ iff $s \equiv t$

— $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$ iff $s_1 \equiv t_1$ and $s_2 \equiv t_2$
Type compatibility: example

Consider:

type link = ^cell
var next : link;
var last : link;
var p : ^cell;
var q, r : ^cell;

Under name equivalence:
— next and last have the same type
— p, q and r have the same type
— p and next have different type

Under structural equivalence all variables have the same type

Ada/Pascal/Modula-2 are somewhat confusing: they treat distinct type definitions as distinct types, so
— p has different type from q and r (!)
Type compatibility: Pascal-style name equivalence

Build compile-time structure called a **type graph**:  
- each constructor or basic type creates a node  
- each name creates a leaf (associated with the type’s descriptor)

---

*Type expressions are equivalent if they are represented by the same node in the graph*
Consider:

```pascal
type link = ^cell
var cell = record
    info : integer;
    next : link;
end
```

Expanding `link` in the type graph yields:
Type compatibility: recursive types

Allowing cycles in the type graph eliminates `cell`:
Type rules

Type-checking rules can be formalized to prove soundness and correctness.

\[ f : A \rightarrow B, \ x : A \]
\[ \frac{}{f(x) : B} \]

If \( f \) is a function from \( A \) to \( B \), and \( x \) is of type \( A \), then \( f(x) \) is a value of type \( B \).
### Example: Featherweight Java

**Syntax:**

```
CL ::= class C extends C {C T; K H}
K ::= C(T T) {super(T); this.T = T;}
M ::= C m(C T) {return e;}
e ::= x | e.f | e.m(µ) | new C(µ) | (C)e
```

**Subtyping:**

```
C < C

C < D  D < E

CT(C) = class C extends D {...}

C < D
```

**Computation:**

```
fields(C) = C T

(neu C(µ)).f₁ → e₁

mbody(m, C) = (X, e₀)

(new C(µ)).m(d)
→ [d/X, new C(µ)/this]e₀

C < D

(D)(new C(µ)) → new C(µ)
```

**Expression typing:**

```
Γ ⊢ x ∈ Γ(x) (T-VAR)

Γ ⊢ e₀ ∈ C₀ fields(C₀) = C T

Γ ⊢ e₀.f₁ ∈ C₁ (T-FIELD)

Γ ⊢ e₀ ∈ C₀ mtype(m, C₀) = D → C

Γ ⊢ a ∈ C C ⊂ D

Γ ⊢ e₀.m(µ) ∈ C (T-INVK)

Γ ⊢ fields(C) = D T

Γ ⊢ a ∈ C C ⊂ D

Γ ⊢ new C(µ) ∈ C (T-NEW)

Γ ⊢ e₀ ∈ D D ⊂ C

Γ ⊢ (C)e₀ ∈ C (T-UCAST)

Γ ⊢ e₀ ∈ D C ⊂ D C ≠ D

Γ ⊢ e₀ ∈ C (T-DCAST)

Γ ⊢ e₀ ∈ D C ≠ D D ⊂ C stupid warning

Γ ⊢ (C)e₀ ∈ C (T-SCAST)
```

**Method typing:**

```
X: C, this : C ⊢ e₀ ∈ E₀ E₀ ⊂ C₀

CT(C) = class C extends D {...}

override(m, D, C₀ → C₀)

C₀ m (C T) {return e₀;} OK IN C
```

**Class typing:**

```
K = C(B g, C T) {super(g); this.T = T;}

fields(D) = D g H OK IN C

class C extends D {C T; K H} OK
```

---

Used to prove that generics could be added to Java without breaking the type system.

---

Igarashi, Pierce and Wadler, “Featherweight Java: a minimal core calculus for Java and GJ”, OOPSLA ’99
doi.acm.org/10.1145/320384.320395
Can you answer these questions?

- Why can semantic analysis be performed by the parser?
- What are the pros and cons of introducing an IR?
- Why must an attribute dependency graph be acyclic?
- Why would be the use of a symbol table at run-time?
- Why does Java adopt nominal (name-based) rather than structural type rules?
What you should know!

- Why is semantic analysis mostly context-sensitive?
- What is “peephole optimization”?
- Why was multi-pass semantic analysis introduced?
- What is an attribute grammar? How can it be used to support semantic analysis?
- What kind of information is stored in a symbol table?
- How is type-checking performed?
Attribution-ShareAlike 4.0 International (CC BY-SA 4.0)

You are free to:

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material for any purpose, even commercially.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

http://creativecommons.org/licenses/by-sa/4.0/