4 Semantic Analysis

Chapter Contents

1. Type Systems
   - Purpose of a Type System
   - Components of a Type System
2. The Attribute-Grammar Framework
   - Evaluation Methods
   - Circularity

Semantic Analysis

Semantic analysis or **context-sensitive analysis** derives facts from the source code using contextual knowledge.

Example Consider an identifier x used in the program being compiled. Before it can emit executable code for computations involving x, the compiler must answer many questions.

- What kind of value is stored in x?
- How much space does x need?
- If x is a procedure, what arguments does it take?
  What kind of value, if any, does it return?
- How long must x's value be preserved?
- Who is responsible for allocating space for x (and initializing it)?

Many, if not all, of these questions reach **beyond** the context-free syntax of the source language.

Semantic Analysis

Most of these questions can be answered if the **type** of x is known. This knowledge is not encoded in the grammar of the language, but specified in the **language specification**.

Example The specification of Algol-like languages, such as C or Oberon, defines a "declare before use" rule. We could try to enforce this rule with the following production.

$$1 \mid \textit{ProcedureBody} \rightarrow \textit{Declarations StatementSequence}$$

While this syntactic constraint enforces a certain code structure, it does **nothing** to check the language rule.

Context-free grammars deal with syntactic categories, not with specific words. In terms of their syntactic structure, the following two statements are the same.

```
int i = 1 + 2;        double d = 100 / 3;
```

Semantic Analysis

Enforcing the “declare before use” rule requires a deeper level of knowledge than **cannot** be encoded in the context-free grammar.

To solve this particular problem, the compiler typically uses a **symbol table**

- on declaration, it inserts a new symbol
- at each reference, it looks up the symbol
- a lookup failure indicates a missing declaration

While this ad-hoc solution bolts onto the parser, it uses mechanisms well outside the scope of context-free languages.

Semantic Analysis

```
1  #include <memory>
2  #include <string>
3  #include <unordered_map>
4
5  class SymbolTable
6  {
7     private:
8        std::unordered_map<std::string, const Node*> table_;
9
10    public:
11       explicit SymbolTable();
12       ~SymbolTable();
13
14       void beginBlock();
15       void insert(const std::string &name, const Node* node);
16       const Node* lookup(const std::string &name) const;
17       void endBlock();
18 };
```

Introduction to Type Systems

A **type** specifies a set of properties held in common by all values of that type.

Types can be specified by **membership**

- a value of type `integer` might be any whole number `i` in the range $-2^{31} \leq i < 2^{31}$
- type `rgb_color` could be defined as the set `{red, green, blue}`

Types can be specified by **rules**

- the declaration of a `RECORD` or an `ARRAY` in Oberon-0 defines a type
- the declaration of a `class` in C++ or Java also defines a type

Some types are predefined by the programming language, others are constructed by the programmer.

The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a **type system**.

Purpose of Type Systems

The type system creates a **second vocabulary** for describing both the form and behavior of valid programs.

Analyzing a program from the perspective of its type system yields information that cannot be obtained using the techniques of scanning and parsing.

In a compiler, this information is typically used for the following purposes

- ensuring runtime safety
- improving expressiveness
- generating better code

Ensuring Runtime Safety

A well-designed type system helps the compiler **detect** and **avoid runtime errors**. In order to accomplish this, the compiler must infer a type for each expression.

Implicit Conversion

Many languages specify rules that allow an operator to combine values of different type and require that the compiler insert conversions as needed.

The alternative is to require the programmer to write an explicit conversion or cast.

A language in which every expression can be assigned an unambiguous type is called a **strongly typed** language, otherwise its **untyped** (*e.g.*, Assembler) or **weakly typed**.

If every expression can be typed at compile time, the language is **statically typed**. The language is **dynamically typed**, if some expression can only be typed at runtime.

Ensuring Runtime Safety

Example FORTRAN 77 has a very simple type system with just a handful of types. For each operator, the result type of an expression is defined by a table based on its operands.

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	illegal
complex	complex	complex	illegal	complex

Example APL lacks declarations, allows a variable's type to change at any assignment, and lets the user enter arbitrary code at input prompts.

$\longrightarrow$ For such a language, a compiler cannot infer types for all expressions.

Improving Expressiveness

A well-constructed type system allows the language designer to **specify behavior more precisely** than is possible with context-free rules.

Operator overloading is a good example how context-dependent semantics that cannot be specified by a context-free grammar can make a language more expressive.

- most languages overload arithmetic operators, *e.g.*, `+`, `-`, `*`, `/`, `%`, *etc*.
- type information determines the effect of autoincrementing a pointer in C
- object-oriented languages use type information to call overloaded methods

Untyped and Weakly-Typed Languages

In BCPL, the only type is a `word`, which can contain any bit-pattern. The interpretation of that bit pattern is determined by the operator applied to the cell. Thus, BCPL uses `+` for integer addition and `#+` for floating-point addition.

Generating Better Code

A well-designed type system provides the compiler with detailed information that can often be used to **produce more efficient translations**.

Example Consider implementing addition in FORTRAN 77. The compiler can completely determine the types of all expressions and use this information to emit case-specific code.

Type of			Code
a	b	a + b	
integer	integer	integer	iADD r_a, r_b ⇒ r_{a+b}
integer	real	real	i2f r_a ⇒ r_{a_f} fADD r_{a_f}, r_b ⇒ r_{a_f+b}
integer	double	double	i2d r_a ⇒ r_{a_d} dADD r_{a_d}, r_b ⇒ r_{a_d+b}

Generating Better Code

For types that cannot be determined at compile time, this checking is **deferred until runtime**. To type safety, the compiler must emit additional code.

This approach adds **significant overhead** to each operation

- nested `if-then-else` structure requires tests and jumps
- each variable now needs two registers: one for `val` and one for `tag`

Performing type inference and checking at compile time eliminates of overhead.

From a performance perspective, the static approach is **always** preferable.

```
1  if tag(a) = integer then
2  |  if tag(b) = integer then
3  |  |  val(c) ← val(a) + val(b)
4  |  |  tag(c) ← integer
5  |  else
6  |  |  ...
7  else if tag(a) = real then
8  |  ...
9  else
10 |  ...
```

Type Checking

Taken together, these activities are often called **type checking**, a misnomer that lumps together the separate activities of **type inference** and **identifying type-related errors**.

An untyped language might be implemented in a way that catches certain errors, while a strongly typed, statically checkable language might be implemented with runtime checking.

Examples

- ML, Haskell and Modula-3 are strongly typed languages that can be statically checked
- Common Lisp has a strong type system that must be checked dynamically
- ANSI C is a typed language, but some implementations do not identify all type errors

The theory underlying **type systems** encompasses a large and complex body of knowledge that is beyond the scope of this course.

Components of Type Systems

A type system for a typical modern language has **four** major components

- a set of base or built-in types
- rules for constructing new types from the existing types
- a method for determining if two types are equivalent or compatible
- rules for inferring the type of each source-language expression

Note Additionally, many languages include rules for the implicit conversion of values from one type to another based on context.

Base of Built-in Types

Most programming languages include base types for some, if not all, of the following kinds of data: **numbers**, **characters**, and **booleans**.

The precise definitions for base types, and the operators defined for them, vary across languages.

Example

Lisp includes both a rational and complex number type. Rational numbers are, essentially, pairs of integers interpreted as ratios, *e.g.*, `(/ 2 3)`. Complex numbers are realized in a similar way: `#c(1 2)`.

Numbers

Programming languages typically support both **limited-range integers** and **approximate real numbers**, often called floating-point numbers.

Which aspects, if any, of the underlying hardware architecture are exposed by a language is an important part of the type system.

Examples

Both FORTRAN and C specify the size of numbers in relative terms, *e.g.*, `double` is twice as large as `float`, but they give the compiler control over the length of the smallest category of number.

In contrast, Java's `float` type specifies a 32-bit IEEE floating-point number, while its `double` type specifies a 64-bit IEEE floating-point number.

Characters

Many languages include a type to represent **single characters**.

- initially encoded as single-byte (8-bit) value, usually mapped into ASCII character set
- more recently expressed in the Unicode standard format, which requires 16 bits

Most languages assume that the character set is **ordered** so that standard comparison operators, such as <, =, and >, work intuitively, enforcing lexicographic ordering.

Conversion between a character and an integer appears in some languages.

Booleans

Most programming languages include a **boolean** type with two values: `true` and `false`.

Standard operations provided for booleans

- **logical and**, typically written as `and` or `&&`
- **logical or**, typically written as `or` or `||`
- **exclusive or**, typically written as `xor` or `^`
- **logical not**, typically written as `not` or `!`

Boolean values, or boolean-valued expressions, are often used for **control flow**.

Example

The C standard (C11, §7.18) considers boolean values as a subrange of the unsigned integers, restricted to the values zero (`false`) and one (`true`).

Compound and Constructed Types

Base types adequately abstract from the kinds of data handled directly by the hardware, but they are often inadequate to represent the information domains needed by programs.

Programs routinely deal with more complex data structures

- records, objects
- arrays, tables
- lists, stacks, queues
- trees, graphs
- …

Tying these organizations to the type system improves the compiler's ability to detect ill-formed programs and lets the language express higher-level operations.

Arrays

The essential property of an array is that the program can compute the address of an element by using indexes as subscripts.

Example For the following Java array `int[][] a = new int[10][10];` the compiler computes the address of element `a[7][2]` as `@a` $+ 7 \times (10 \times 4) + 2 \times 4$.

Support for operations on arrays varies widely

- assignment of whole or partial arrays
- element-by-element arithmetic operations

Some languages include the array's dimensions in its type, *i.e.*, a 10×10 array of integers has a different type than a 12×12 array of integers

Most languages allow arrays of any base type, while some languages also allow arrays of constructed types.

Strings

Some programming languages treat strings as a constructed type.

- most languages limit this built-in support to the **character string**
- PL/I supports more than one string type, *e.g.*, bit string and character string

Some languages support character strings by handling them as **arrays of characters**.

A true string type differs from an array type in several important ways

- operations that make sense on strings may not have analogs for arrays
- string comparison should work from **lexicographic order**
- actual length of a string may differ from its allocated array size

Enumerated Types

An enumerated type, introduced in Pascal, lets the programmer use self-documenting names for small sets of constants.

Pascal

```
type Month = (Jan, Feb, Mar Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
```

ANSI C

```
enum WeekDay {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

The compiler maps each element of an enumerated type to a **distinct** value.

- elements of an enumerated type are **ordered**
- comparisons between elements of the same **type make** sense
- operations that compare **different** enumerated types make no sense
- in some languages values of enumerated types can be used to **index into arrays**

Structures

Structures, or records, group together multiple objects of arbitrary type. The elements, or members, of the structure are typically given explicit names.

Pascal

```
1 type Node = record
2                left:  ^Node;
3                right: ^Node;
4                value: integer;
5             end;
```

ANSI C

```
1 struct Node {
2    struct Node* left;
3    struct Node* right;
4    int          value;
5 }
```

The type of a structure is the **ordered product** of the types of the individual elements that it contains, *e.g.*, $(\texttt{Node*}) \times (\texttt{Node*}) \times \texttt{int}$.

These new types should have the **same** essential properties that a base type has.

Variants

Many programming languages allow the creation of a type that is the union of other types.

Pascal

```
1 type Mixed =
2    record case integer of
3       0: (number: integer);
4       1: (truth:  boolean);
5       2: (month:  Month);
6    end;
```

ANSI C

```
1 union Mixed {
2    int     number;
3    bool    truth;
4    WeekDay day;
5 }
```

The type of a union is the **alternation** of the types of the individual elements that it contains, *e.g.*, `int` ∪ `bool` ∪ `WeekDay`.

Unions can also include structures of distinct types, even when the individual structure types have **different** lengths.

Pointers

Pointers are abstract memory addresses that let the programmer manipulate arbitrary data structures.

Pointers are created when objects are created

- `new` in Java creates and returns an explicitly typed object
- other languages use a polymorphic routine that takes the return type as a parameter
- `malloc` in ANSI C returns a pointer to `void`, forcing the programmer to cast the value
- some languages provide an operator that returns the address of an object, *e.g.*, `&` in C

Some languages allow **direct manipulation** of pointers. Arithmetic operations on pointers, *e.g.*, autoincrement and autodecrement, allow the program to construct new pointers.

The ability to construct new pointers **seriously reduces** the ability of both the compiler and its runtime system to reason about pointer-based computations and to optimize such code.

Type Equivalence

Any programming language with a nontrivial type system must include an unambiguous rule to decide whether or not two different type declarations are **equivalent**.

Historically, two general approaches have been tried

- **name equivalence**: two types are equivalent iff they have the same name
- **structural equivalence**: two types are equivalent iff they have the same structure

Both policies have strengths and weaknesses

- name equivalence assumes that identical names occur as a deliberate act
- structural equivalence assumes that identically structured objects can be safely used in each other's place

The type system's definition of type equivalence determines how types need to be represented in the compiler.

Inference Rules

In general, type inference rules specify, for each operator, the **mapping** between the operand types and the result type.

- table-based representation, *e.g.*, FORTRAN example on Slide 198
- rule-based specification, *e.g.*, numeric promotion in Java

Inference rules enable compiler to handle **mixed-type expressions** correctly

- if illegal, compiler needs to report a type error
- if legal, compiler might need to emit code to perform implicit conversion

Inferring the type of **constants**

- based on literal, *e.g.*, in Java, `2` is an `int`, but `2.0` is a `double`
- based on use, *e.g.*, in `sin(2)`, `2` is an `double`, but in `int x = 2`, it is a `int`

Inferring Types for Expressions

With declared types for variables, implied types for constants, and a complete set of type-inference rules, we can assign types to any expression over variables and constants.

Conceptually, the compiler can assign a type to each value in the expression during a simple **postorder tree** walk on its parse tree.

This should let the compiler detect every violation of an inference rule, and report it at compile time.

- if the language lacks one or more of the features that make this simple style of inference possible, the compiler will need to use **more sophisticated techniques**
- if compile time type inference becomes too difficult, the compiler writer may need to **move some of the analysis and checking to runtime**

Interprocedural Aspects of Type Inference

Even in the simplest type systems, expressions contain **function calls**. To check those calls, the compiler needs a **type siganature** for each function.

Examples

In C, the programmer can provide a **function prototype** as a specification of the types of the formal parameters and return value(s) of a function.

```
unsigned int strlen(const char *s);
```

In Haskell, functions can be annotated with a type signature.

```
map :: (a -> b)-> [a] -> [b]
```

The function `map` exhibits **parametric polymorphism**: its result type is a function of its argument types.

Interprocedural Aspects of Type Inference

To perform accurate type inference, the compiler needs a type signature for every function.

That information can be obtained in several ways

- require that the entire program be presented for compilation as a unit
- require a type signature for each function, *e.g.*, mandatory function prototypes
- defer type checking until either link time or runtime, when this information is available
- embed the compiler in a program-development system that gathers the requisite information and makes it available to the compiler on demand

Note All of these approaches have been used in real systems.

The Attribute-Grammar Framework

One formalism that has been proposed for performing context-sensitive analysis is the **attribute grammar**, or **attributed context-free grammar**.

An attribute grammar consists of a context-free grammar augmented by a set of rules

- each rule defines one value, or attribute, in terms of the values of other attributes
- rules associate the attribute with a specific grammar symbol
- each instance of the grammar symbol in a parse tree has an instance of the attribute
- rules are functional, *i.e.*, they imply no specific evaluation order

Example Consider a context-free grammar $SBN = (T, NT, S, P)$ that generates all **signed** binary numbers, such as `-101`, `+11`, `-01`, and `+11111001100`, but excludes **unsigned** binary numbers, such as `10`.

The Attribute-Grammar Framework

$$
\mathrm{P} = \left\{
\begin{array}{lcl}
\textit{Number} & \rightarrow & \textit{Sign List} \\
\textit{Sign} & \rightarrow & \texttt{+} \\
 & | & \texttt{-} \\
\textit{List} & \rightarrow & \textit{List Bit} \\
 & | & \textit{Bit} \\
\textit{Bit} & \rightarrow & \texttt{0} \\
 & | & \texttt{1}
\end{array}
\right\}
$$

$$\mathrm{T} = \{\texttt{+}, \texttt{-}, \texttt{0}, \texttt{1}\}$$

$$\mathrm{NT} = \{\textit{Number}, \textit{Sign}, \textit{List}, \textit{Bit}\}$$

$$S = \{\textit{Number}\}$$

We can build an attribute grammar that annotates *Number* with the value of the signed binary number that it represents.

The following attributes are needed.

Symbol	Attributes
Number	`value`
Sign	`negative`
List	`position, value`
Bit	`position, value`

The Attribute-Grammar Framework

	Production	Attribution Rules
1	*Number* → *Sign List*	*List*.position ← 0 if *Sign*.negative then *Number*.value ← −*List*.value else *Number*.value ← *List*.value
2	*Sign* → +	*Sign*.negative ← false
3	*Sign* → -	*Sign*.negative ← true
4	*List* → *List Bit*	*Bit*.position ← *List*.position *List*.value ← *Bit*.value
5	*List* → *Bit*	$List_1$.position ← $List_0$.position + 1 *Bit*.position ← $List_0$.position $List_0$.value ← $List_1$.value + *Bit*.value
6	*Bit* → 0	*Bit*.value ← 0
7	*Bit* → 1	*Bit*.value ← $2^{Bit.position}$

Note Subscripts are added to a symbol if it appears multiple times in a single production.

The Attribute-Grammar Framework

The rules add attributes to the parse tree nodes by their names. An attribute mentioned in a rule must be instantiated for **every occurrence** of that kind of node.

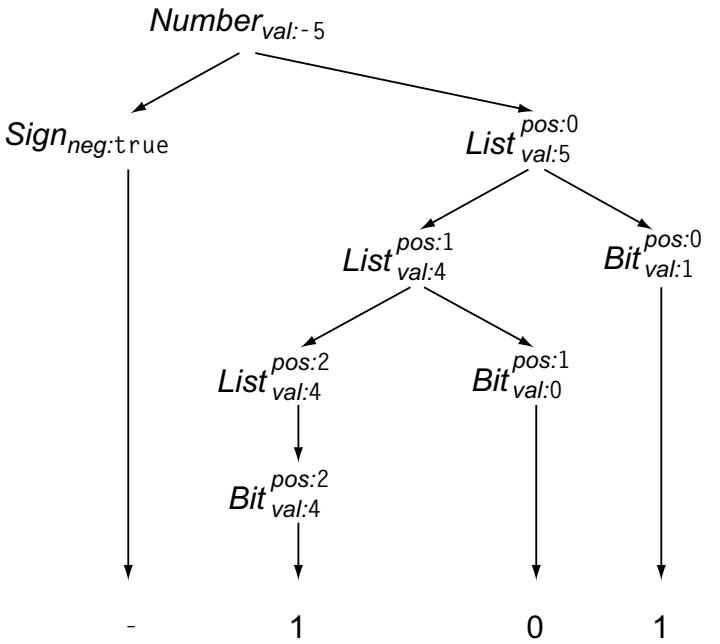
Each rule specifies the value of one attribute in terms of **literal constants** and the **attributes of other symbols** in the production.

- rules can pass information from the production's left-hand side to its right-hand side
- rules can also pass information in the other direction

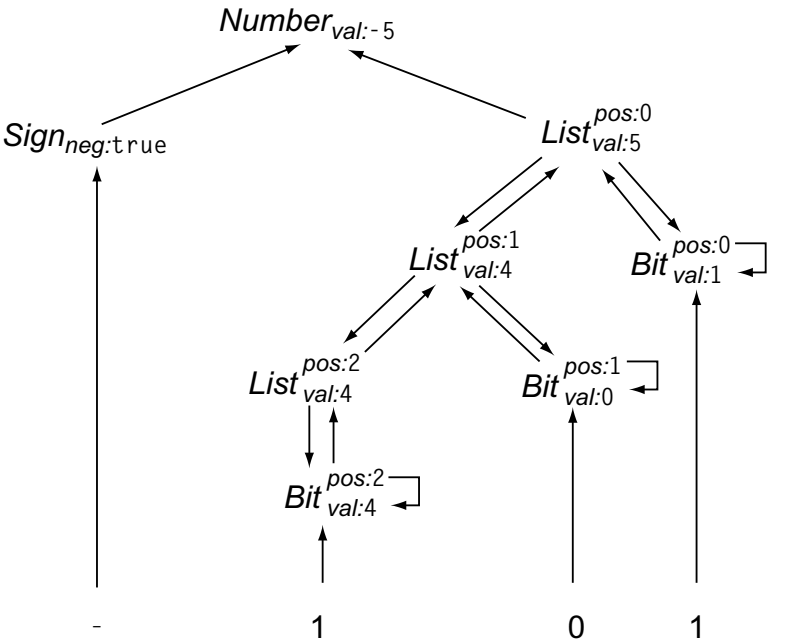
Each rule implicitly defines a **set of dependencies**, *i.e.*, the attribute being defined depends on each argument to the rule. Taken over the entire parse tree, these dependencies form an **attribute-dependence graph**.

The Attribute-Grammar Framework

Parse Tree for `-101`



Dependence Graph for `-101`



The Attribute-Grammar Framework

The bidirectional flow of values that we noted earlier shows up in the dependence graph. We distinguish between attribute attributes based on the direction of value flow.

Synthesized Attributes

- defined by bottom-up information flow
- draws values from the node itself, its descendants, and constants

Inherited Attributes

- defined by top-down and lateral information flow
- draws values from the node itself, its parent and siblings, and constants

Note The syntax of the attribution rules allows a rule to reference its own result, either directly or indirectly. Such **circular** attribute grammars are ill-formed.

The Attribute-Grammar Framework

Based on attribute grammars, tools can take a **high-level**, **nonprocedural** specification and automatically produce an implementation.

One critical insight behind the attribute-grammar formalism is the notion that the attribution rules can be associated with productions in the contextfree grammar.

Since the rules are functional, the values produced are independent of evaluation order, for any order that respects the relationships encoded in the attribute-dependence graph.

In practice, any order that evaluates a rule only after all of its inputs have been defined respects the dependences

Evaluation Methods

The attribute-grammar model has practical use only if we can build evaluators that interpret the rules to evaluate an instance of the problem automatically.

Such attribute evaluation techniques fall into **three** major categories

- **Dynamic Methods** use the structure of a particular attributed parse tree
- **Rule-Based Methods** rely on a static analysis of the attribute grammar
- **Oblivious Methods** consider neither the attribute grammar nor the particular attributed parse tree to determine the order of evaluation

Circularity

Circular attribute grammars can give rise to cyclic attribute-dependence graphs. Our models for evaluation fail when the dependence graph contains a cycle.

A compiler that uses attribute grammars can use **two** approaches to handle circularity.

Avoidance

- restrict to a class of attribute grammars that cannot give rise to circular dependences
- for example, restricting the grammar to use only synthesized and constant attributes
- more general classes exist, *e.g.*, strongly noncircular attribute grammars,

Evaluation

- assign a value to every attribute, even those involved in cycles
- evaluator might iterate over the cycle and assign appropriate or default values
- this approach avoids the problems associated with a failure to fully attribute the tree

In practice, most attribute-grammar systems are restrict to **noncircular** grammars.

More Applications

Attribute grammar are a **general** framework that has many applications in a compiler.

- **evaluation rules** as used in the example with signed binary numbers
- **type inference rules** to identify type-related errors
- rules to **estimate execution time** in cycles
- rules for **ad-hoc syntax-directed translation**

Oberon-0 Compiler Project

The Oberon-0 compiler gives rise to the **first two** applications. Evaluation rules are used to calculate the value of arithmetic expressions over constants. Type inference rules are used to enfore the language rules for typing in Oberon-0.

The **visitor** pattern can be used to annotate the abstract syntax tree with attributes.