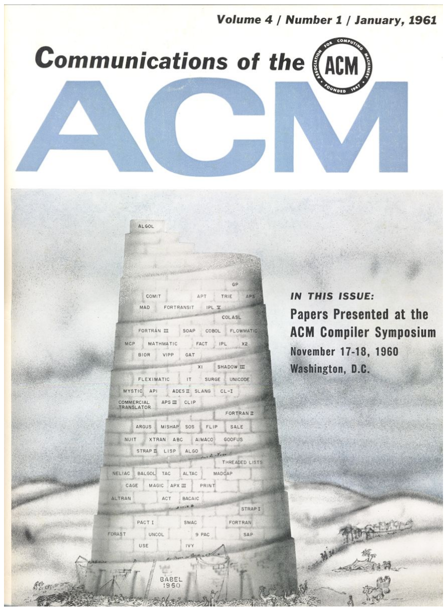
Programming Languages

CSE 3302, Summer 2019
Module 04
Semantic Analysis


Volume 4 / Number 1 / January, 1961
Communications of the ACM
IN THIS ISSUE:
Papers Presented at the ACM Compiler Symposium
November 17-18, 1960
Washington, D.C.
BABEL 1960

M04

Semantic Analysis

Semantic Analysis

- *Syntactic* analysis covers the *form* of the program.
- *Semantic* analysis covers the *meaning* of the program.

- Semantic analysis is required for two major reasons.
  - Enforcement of the rules for a *valid* program that go beyond the *form* of the program.
  - Determining *information* required for the *generation* of the output program.

Semantic Analysis

- In general, semantic analysis covers those items ① that we cannot capture in the language's *context-free grammar*.
  - Or those items ② that aren't *convenient* to capture in the CFG.
- For example,
  - ① Comparing argument lists in function calls to that given in the function's declaration.
    - CFGs can't match counted lists (that aren't properly nested).
  - ② Ensuring that a function has at least one return statement.
    - Requires complicating the CFG incredibly (and not usefully).

Semantic Analysis

- Generally, any rule requiring comparing items separated in the program or counting items will end up being a *semantic* check rather than a *syntactic* check.

C

```c
int f( int a, double b, char c );

float p() {
  // Seems easy, just count and compare
  // the types.  However, not possible
  // with a pure Context Free Grammar.
  int i = f( 1, 1.1, 'a' );

  // Need a return statement.  This case
  // (the last statement) is easy to put in
  // the CFG, but the general case (statement
  // is anywhere in the function body) forces
  // a very complicated CFG.  It's easier just
  // to do the check in the Semantic Analysis
  // phase instead.
  return i / 2.0;
}
```

Semantic Analysis

- Semantic rules may be either *Static* or *Dynamic*.
- *Static Semantic Rules* are those that can be enforced at *Compile Time*.
  - The compiler itself performs the check and reports any errors.
  - For example, misspelled keywords, use of undeclared variable, …
- *Dynamic Semantic Rules* cannot be enforced until *Run Time*.
  - Code is inserted in the program or a call to a library routine is made to perform the check and the error is reported during run time.
  - For example, divide by zero, array index out of bounds, …

Semantic Analysis

- Sometimes what seems to be a *Dynamic Check* can actually be done at *compile time*.
- One cannot do this perfectly, however, so almost no compiler even tries.
    - Researchers in proofs about programs are interested in this sort of thing.
- Basic Computability Theory tells us that we can't find these sorts of problems in general statically.

C

```c
float p( int i ) {
  // Return the reciprocal of the input.
  // Dynamic Check required here because
  // we don't know what i is until the
  // program runs.
  return 1.0 / i;
}

void q() {
  int n;
  float a;

  // Get an int value from the user.
  scanf( "%d", &n );

  // Print its reciprocal.
  printf( "%f", p( n ) );

  // Print the reciprocal of zero.
  // A clever compiler could catch this
  // "dynamic" error at compile time.
  printf( "%f", p( 0 ) );
}
```

Role of the Semantic Analyzer

- Languages differ greatly in the sorts of semantic rules they enforce. Some examples,
    - Python implements 'infinite precision' integer operations.
        - Overflow doesn't happen until you run out of memory.
        - Python just keeps moving the value to larger and larger objects.
    - On the other hand, Ada assigns a *specific type* to every numeric variable and requires the user to make *explicit conversions*.
    - Lisp preserves precision through integer *division*.
        - `(/ 4 2)` ⟶ 2, an *integer*, but `(/ 4 3)` ⟶ 4/3, a *rational* number.

Role of the Semantic Analyzer

- C is an example of a YAFIYGI language.
  - It enforces *no* dynamic semantic checks.
  - Whatever the hardware does, C does.
    - If the underlying hardware checks for divide by zero, one will get an exception on a divide by zero, otherwise *not*.
    - The same goes for out-of-bounds memory references. If it causes a hardware error, you'll hear about it, otherwise *not*.
- "You Asked For It, You Got It!" — Whatever you write, that's what you get. :)

Role of the Semantic Analyzer

- Java on the other hand goes to other extreme.
- Java takes great pains to check as many rules as possible, both *statically* and *dynamically*.
- This tends to catch not only *inadvertent errors* (*bugs*) but also *malicious code* that attempts to damage or compromise memory or files.

Role of the Semantic Analyzer

- Some compilers provide a way to disable dynamic semantic checks to improve execution speed.
- Are semantic checks always a good thing?
- What penalties are paid for semantic checks?
- Some teams use semantic checks while a system is being developed or debugged, but remove them for production use.
    - Usually the reasoning is that they want increased performance. Is that a valid concern?
    - Suppose there's a divide by zero? Exhaustion of the heap?
- *No simple answers here!*

[*On Disabling Checks*]

It is absurd to make elaborate security checks on debugging runs, when no trust is put in the results, and then remove them in production runs, when an erroneous result could be expensive or disastrous. What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?

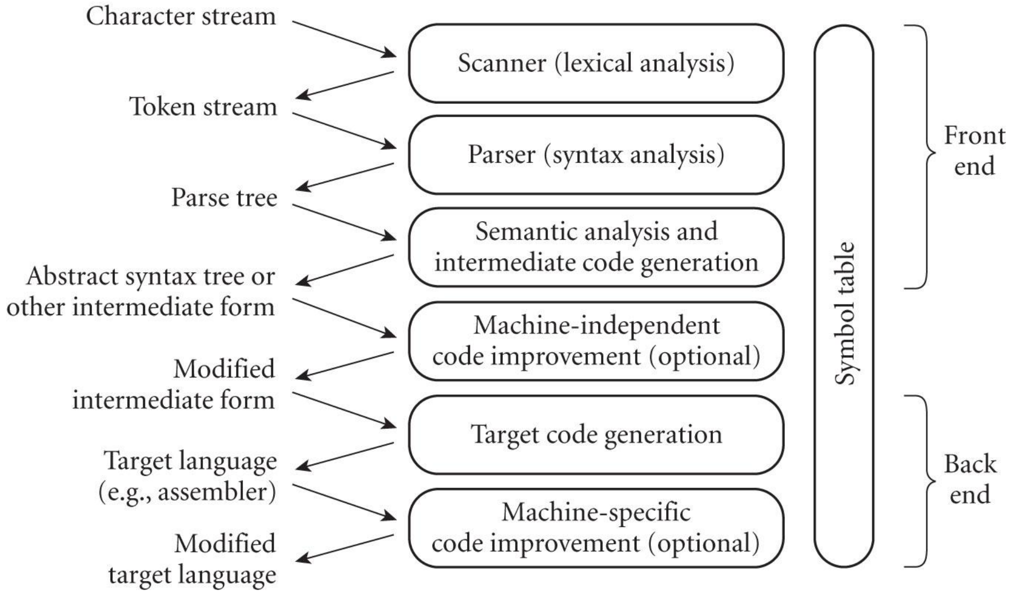
— *C. A. R. (Tony) Hoare* (1973)

Inventor of QuickSort, Hoare logic, the language CSP, …
Lead the team that created the first Algol 60 compiler
Extensive work in *proving* the correctness of programs
Received the Turing Award (1980)

Role of the Semantic Analyzer

- Ultimately, the role of the Semantic Analyzer is to
  - *Enforce* all *Static Semantic Rules*.
  - *Annotate* the parse tree with any information required by the *Intermediate Code Generator*. This includes …
    - *Clarifications*: For example, *resolution* of overloaded operations, *identification* of which binding is meant for a name reference.
    - *Dynamic Semantic Checks*: Satisfying whatever requirements there are for checks that must be performed at run time.
- Usually, the Semantic Analyzer produces the *Annotated Syntax Tree* used by the Code Generation phase of compilation.

Phases of Compilation


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis and intermediate code generation
Abstract syntax tree or other intermediate form
Machine-independent code improvement (optional)
Modified intermediate form
Target code generation
Target language (e.g., assembler)
Machine-specific code improvement (optional)
Modified target language
Symbol table
Front end
Back end

Role of the Semantic Analyzer

- Aspects of *Parsing*, *Syntax Tree Construction*, *Semantic Analysis*, and *Intermediate Code Generation* can be somewhat interleaved.
  - There's lots of variety among compilers in this.
  - One common method is to interleave the construction of the syntax tree with parsing and then follow with separate phases for semantic analysis and code generation.
- Both the *Semantic Analysis* and *Intermediate Code Generation* phases annotate the parse tree with attributes.

Assertions

- *Assertions* are logical formulae regarding the values of program data.
  - Expected to be *true* when execution reaches them.
  - Written by the *programmer*, not created by the compiler.
  - Used to reason about the correctness of the program.
- For example, in Java there's an `assert` statement, e.g.

```
assert denominator != 0;
```

  will throw the `AssertionError` exception if `denominator == 0`.

Assertions

- In C and C++, `assert` is a *macro*, e.g.

  ```
  assert( denominator != 0 );
  ```

  will generate a message similar to this before aborting the run,

  ```
  Assertion failed: denominator != 0, file filename, line line number
  ```

  if `denominator == 0`.
- Because it's a *macro*, it has access to the program's filename and the line number the assertion appears on.
- Asserts in C and C++ are usually *disabled* after the debugging phase of system development. :(

'Structured' Assertions

- Beyond simple assertions, we have *Invariants*, *Preconditions*, and *Postconditions*.
  - *Invariants* are expected to be *true* at all '*clean points*' of a body of code, usually the beginning and end of subroutines. Loop invariants are checked before and after each iteration.
  - *Preconditions* are expected to be *true* at the beginning of a subroutine and *Postconditions* at the end of a subroutine or at each `return` statement.
- Euclid, Eiffel, and Ada provide explicit support for these.
  - Euclid and Eiffel allow *disabling* these checks for run-time efficiency.

‘Structured’ Assertions

- Invariants, Preconditions, and Postconditions are statements about the *correctness* of the program.
- A *precondition* failure indicates a problem with a *caller*.
- A *postcondition* failure indicates a problem with a *callee*.
- An *invariant* failure indicates a problem in the *encompassed code*: the body of a subroutine, a loop, etc.
- These failures should *never* occur, no matter what the input.
  - Any failure indicates a *programming* problem.
- They thus differ from *exceptions* which can occur during normal program execution.

[*Exceptions*]

- Software exceptions were developed originally in Lisp.
- From the earliest time, the mechanism was used to handle *Conditions* as well as *Errors* .
  - The distinction being one can (usually) recover from a *Condition* but not (necessarily) from an *Error*.
  - *Big gray area here*! There is no clean break.
  - For example, is divide by zero a *Condition* or an *Error*? It’s a matter of interpretation.

[*Exceptions*]

- Languages differ in the extent of support they give exceptions and exception handling.
  - Python regards them as a normal part of processing.
  - Go omits them entirely.
    - … but does have a `panic` / `recover` mechanism.
  - Java has *checked* and *unchecked* exception categories.
    - Checked exceptions *have* to be handled and the presence of a handler is enforced at compile time.
- Most languages have a version of the `try` / `catch` / `finally` construct, though the exact syntax and semantics vary.

'Design by Contract'

- *Design by Contract* is a methodology that prescribes *formal*, *precise*, and ***verifiable*** interface specifications.
- The specifications are referred to as *contracts* between a *client* and a *supplier*.
- *Preconditions* and *Postconditions* can be used to implement these specifications.
  - *Preconditions* verify what the client is supposed to provide.
  - *Postconditions* verify what the supplier is supposed to return.
  - Eiffel has these constructs explicitly. Extensions provide them for Java, C++.

Static Analysis

- Compile-time algorithms that predict run-time behavior are known as *static analyzers*.
- A static analyzer is said to be *precise* if it allows the compiler to determine whether a program will *always* follow a rule.
- *Type checking* is *static* and *precise* in languages such as Ada and ML.
  - The compiler *knows* that a variable will always be used in a type-safe way.
- In Lisp, Smalltalk, Python, Ruby, … type checks are *dynamic*, but still completely safe.
  - There is however a *run time* overhead for the checks.

Static Analysis

- A static analyzer doesn't have to be *precise* to still be *useful*.
- A compiler will do as much analysis as possible at compile time and then generate the *minimal* dynamic check required.
  - For example, Java does static type checking, but *type casting* and *dynamically-loaded classes* may require run-time checks.
  - Many compilers do extensive analysis to minimize the need for dynamic checks on array subscripts, variant record tags, dangling pointers, …
  - This last point is one kind of *performance optimization*.

Static Analysis

- Aside from pure execution performance enhancement, code may also be improved by …
  - *Alias Analysis*: Determining when values can safely be *cached in registers*, to avoid the cost of reading and writing memory, or to be computed *out of order* or to be accessed by *concurrent threads*.
  - *Escape Analysis*: Determining if all accesses to an object are in a given context, possibly allowing stack instead of heap allocation or for access to it not to require *locks*.
  - *Subtype Analysis*: Determining when an object is of a given subtype so that dynamic method dispatch can be avoided.

Static Analysis

- An optimization is said to be …
  - *Unsafe* if it *might* lead to incorrect code in a program.
  - *Speculative* if it *usually* improves performance, but *may degrade* in some cases.
- A compiler is *conservative* if it applies only optimizations that are both *safe* and *effective*.
- A compiler is *optimistic* if it applies *speculative* or *unsafe* optimizations.
  - Dynamic checks are inserted to *undo* the effects of *unsafe* code when it goes wrong.
  - This kind of compiler is fairly unusual. Why?

Static Analysis

- Language designers can eliminate the need for some *dynamic checks* by strengthening *static semantic rules*.
  - Programs which fail *conservative* analysis are simply banned.
- ML avoids the *dynamic type checking* that Lisp must perform by forbidding certain programming idioms that Lisp supports.
  - This disallows some legal, valid, and useful Lisp programming techniques.
- The *definite assignment rule* of C# ensures that no variable can be referenced before it is assigned a value.
  - This disallows some legal, valid, and useful C programming techniques.

Review Questions (1)

1. What determines whether a language rule is a matter of *syntax* or of *static semantics*?
2. Why is it impossible to detect certain program errors at compile time even though they may be detected at run time?
3. What are *assertions*? What is their purpose?
4. Which languages support assertions?
5. What is *precise static analysis*?