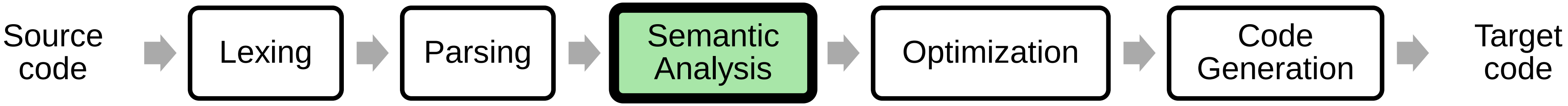
Semantic Analysis

Semantic Analysis


Source code → Lexing → Parsing → Semantic Analysis → Optimization → Code Generation → Target code

- Lexer and Parser have verified that the program is *lexically* and *syntactically* well-formed
- **Semantic Analysis (SA)**: No strict definition.
- Basically, SA is about checking everything that...
  - ...we cannot check in the parser (because not syntax-related, e.g., wrong types)
  - ...we don’t want to check in the parser (because we want to keep the AST and parser code nice and clean)
  - ...can be checked statically (without running the program)
  - ...can be checked in reasonable time

Things to check in Semantic Analysis

- Goal: *reject the largest number of incorrect programs, accept the largest number of correct programs*
- An (incomplete) list of things to check:
  - All used constants, variables, functions,... have been declared
  - Correct types in
    - Arithmetic and boolean operations (wrong: `“Hello”/3.2`)
    - Assignments and initializations (wrong: `int x = “Hello”`)
    - Function calls
    - Return statements
  - Java: thrown exceptions are declared or catched
  - Java, C++: access specifiers (private/public/protected) are respected
  - ...

Limitations of CFGs

- What we said two slides ago:
  - *Semantic Analysis: No strict definition, basically check everything that...we can’t check in the parser (because not syntax-related, e.g., wrong types)*

- Is this really true? Couldn’t we write a CFG to prevent duplicate variable names, wrong types, etc.?
  - No, we can't (except for some simple languages)
  - Do you remember the Pumping Lemma to show that a language is not a regular language? There is also a Pumping Lemma that shows that a CFG cannot do the above things

Implementing the Semantic Analysis

- Two ways:
  1. The more “formal” way. Some parser generator tools like bison and ANTLR allow to add semantic information to the language definition
     - (Very simple) Example:

       $Expr \rightarrow Term + Term$

       $$\boldsymbol{type(Expr) := type(Term_1)\ \ if\ type(Term_1) == type(Term_2)}$$

       $Term \rightarrow number$

       $$\boldsymbol{type(Term) := integer}$$

     - This is called an *Attribute Grammar*

  1. As one or multiple traversals of the AST
     - Quite intuitive, but implemented by hand
     - **That’s what we will do in this course**

Example: Type Checking as an AST Traversal

- Imagine the following AST implemented as classes (if your compiler is written in an object-oriented language)

```
class Program {
     List<Function> functions;          // Very simple example!
}                                       // no new types, no global vars,
class Function {                        // no return types in functions,...
     List<Parameter> parameters;
     List<Statement> body;
}

abstract class Statement { }

class AssignmentStatement extends Statement {
     Identifier leftSide;
     Expression rightSide;
}
class IfStatement extends Statement {
     Expression condition;
     List<Statement> thenStatements;
     List<Statement> elseStatements;
}
```

... and so on

Example: Type Checking as an AST Traversal (2)

- Checking that the expressions and statements are using the types correctly: We start at the root of the AST and then traverse it

```
void checkTypes(Program prog) {
    for(var func : prog.functions) {
        checkTypes(func);
    }
}

void checkTypes(Function func) {
    for(var stmt : func.body) {
        checkTypes(stmt);
    }
}
```

Example: Type Checking as an AST Traversal (3)

- Checking statements

```
void checkTypes(Statement stmt) {
    if(stmt instanceof AssignmentStatement s) {
        var leftType = getTypeOfExpression(s.leftSide);
        var rightType = getTypeOfExpression(s.rightSide);
        if(!leftType.equals(rightType))
            throw new TypeErrorException();
    }
    else if(stmt instanceof IfStatement) {
        // do type checking of if statement
        ...
    }
    else if...
        // and so on
    }
}
```

- Works, but it's ugly! `instanceof` is often a sign of poor OO. Check the Visitor Design pattern for a more elegant implementation https://www.youtube.com/watch?v=KLRun3MFZXg&list=PLBMhFQpVgBPlQGLicbIRrd45_x7jscum1&index=4 https://en.wikipedia.org/wiki/Visitor_pattern