Course Objectives

To learn the process of translating a modern high-level language to executable code.

- Learn the fundamental techniques from lectures, text book and exercises from the book.
- Apply these techniques in practice to construct *a fully working compiler* for a non-trivial subset of Java called "`Decaf`".

In the end, you should be able to compile small Java-like programs with your compiler, and see it actually work!

What is a Compiler?

- Programming problems are easier to solve in *high-level languages*
  - High-level languages are closer to the problem domain
  - E.g. Java, Python, SQL, Tcl/Tk, ...
- Solutions have to be executed by a machine
  - Instructions to a machine are specified in a language that reflects to the cycle-by-cycle working of a processor
- **Compilers** are the bridges:
  - Software that *translates* programs written in high-level languages to efficient executable code.

An Example

```
int gcd(int m, int n)
{
  if (m == 0)
      return n;
  else if (m > n)
    return gcd(n, m);
  else
    return gcd(n%m, m);
}
```

```
gcd:
        pushl %ebp
        movl %esp,%ebp
        cmpl $0,8(%ebp)
        jne .L2
        movl 12(%ebp),%eax
        jmp .L1
        .align 16
        jmp .L3
        .align 16
.L2:
        movl 8(%ebp),%eax
        cmpl %eax,12(%ebp)
        jge .L4
        movl 8(%ebp),%eax
        pushl %eax
...
```


Example (contd.)

```
gcd:
        pushl %ebp
        movl %esp,%ebp
        pushl %esi
        pushl %ebx
        movl 8(%ebp),%esi
        movl 12(%ebp),%ebx
.L11:
        testl %esi,%esi
        jne .L8
        movl %ebx,%eax
        jmp .L13
        .align 16
.L8:
        cmpl %ebx,%esi
        jg .L14
        movl %ebx,%eax
        cltd
        idivl %esi
        movl %edx,%ebx
.L14:
        movl %esi,%ecx
        movl %ebx,%esi
        movl %ecx,%ebx
        jmp .L11
        .align 16
.L13:
        leal -8(%ebp),%esp
        popl %ebx
        popl %esi
        movl %ebp,%esp
        popl %ebp
        ret
```

Requirements

- In order to translate statements in a language, one needs to understand both
  - the *structure* of the language: the way "sentences" are constructed in the language, and
  - the *meaning* of the language: what each "sentence" stands for.
- Terminology:
  - Structure ≡ **Syntax**
  - Meaning ≡ **Semantics**

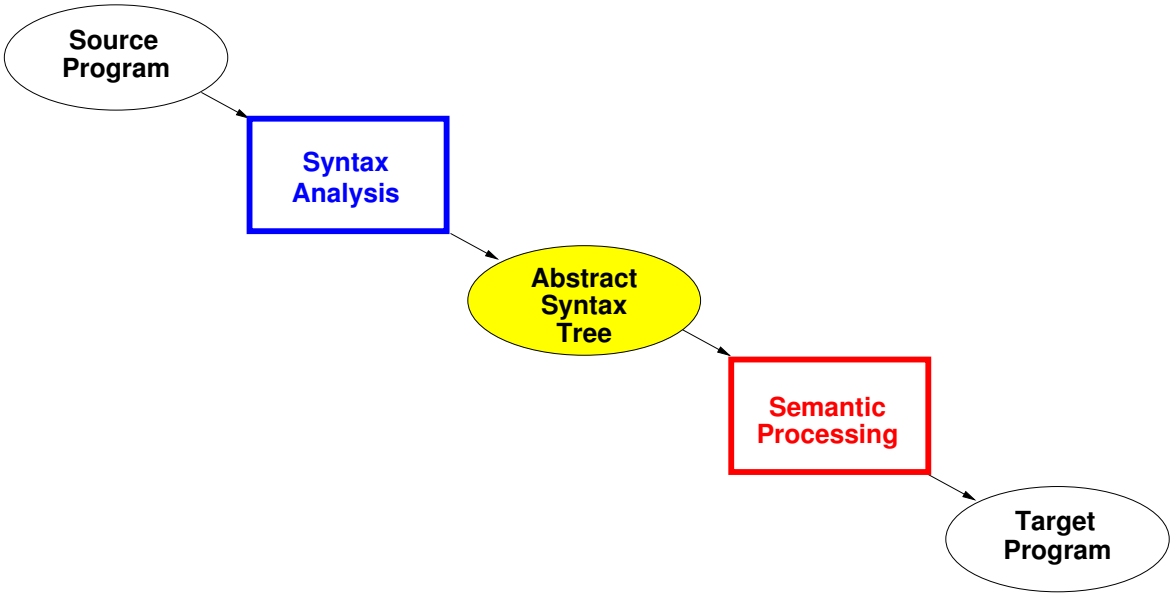
Translation Strategy

Classic Software Engineering Problem

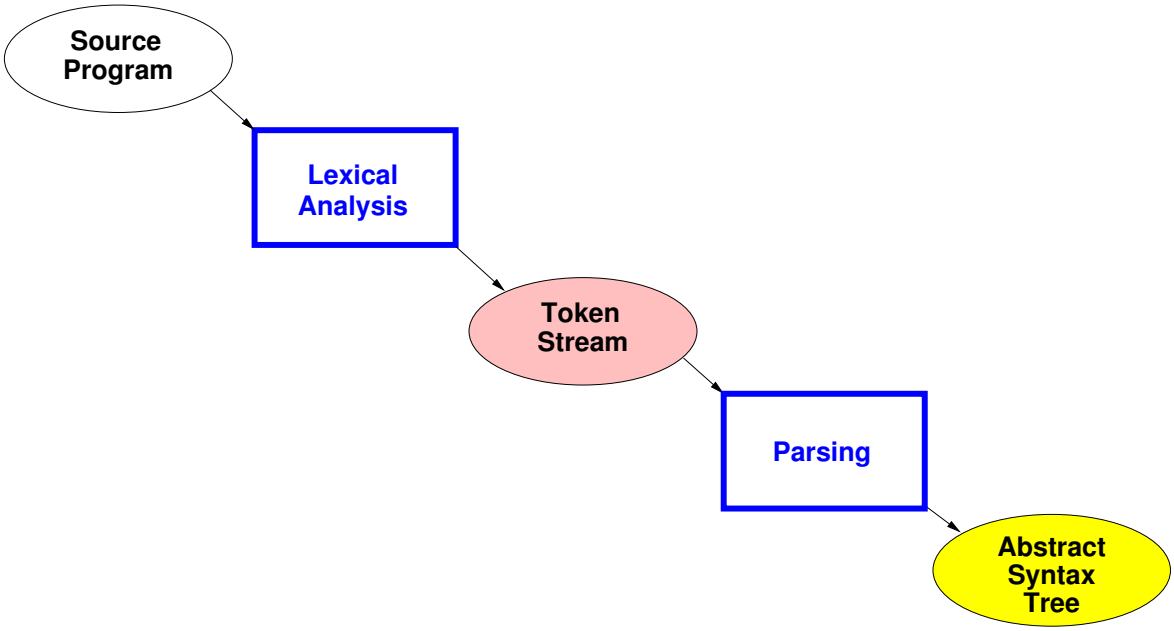
- **Objective:** Translate a program in a high level language into *efficient* executable code.
- **Strategy:** Divide translation process into a series of phases
  Each phase manages some particular aspect of translation.

Interfaces between phases governed by specific intermediate forms.

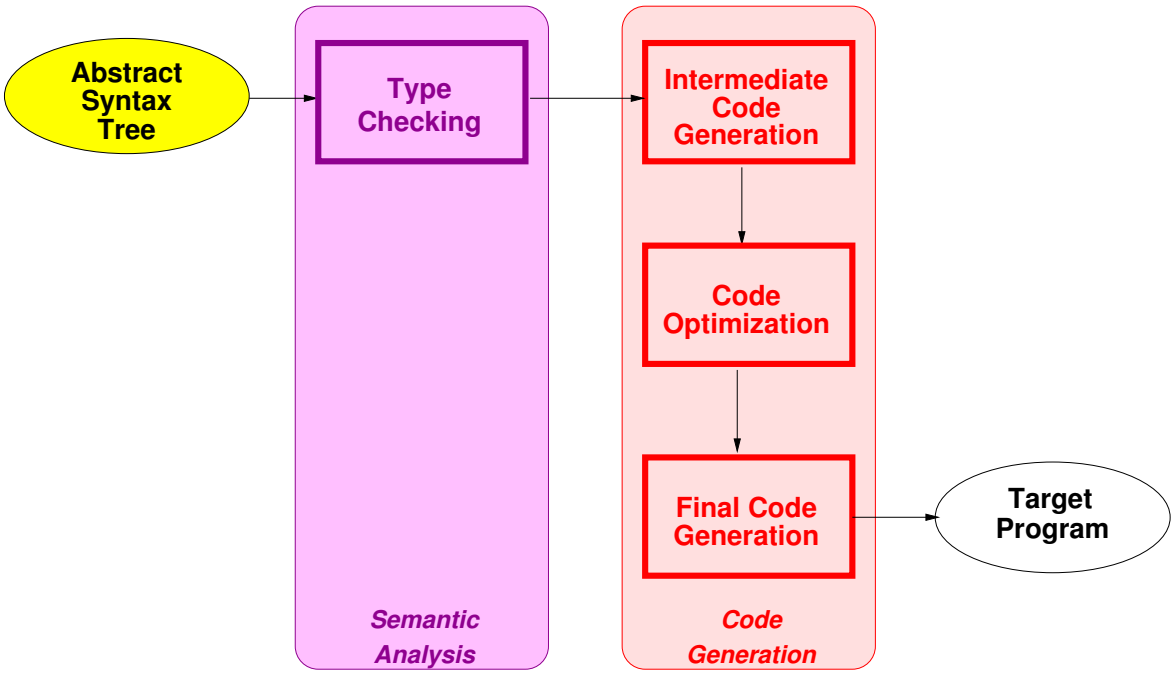
Translation Process



Syntax Analysis



Semantic Processing



Translation Steps

- **Syntax Analysis Phase:** Recognizes "sentences" in the program using the *syntax* of the language
- **Semantic Analysis Phase:** Infers information about the program using the *semantics* of the language
- **Intermediate Code Generation Phase:** Generates "abstract" code based on the syntactic structure of the program and the semantic information from Phase 2.
- **Optimization Phase:** Refines the generated code using a series of *optimizing* transformations.
- **Final Code Generation Phase:** Translates the abstract intermediate code into specific machine instructions.

Lexical Analysis

First step of syntax analysis

- **Objective:** Convert the *stream of characters representing input program* into a sequence of *tokens*.
- Tokens are the "words" of the programming language.
- Examples:
  - The sequence of characters "`static int`" is recognized as two tokens, representing the two words "`static`" and "`int`".
  - The sequence of characters "`*x++`" is recognized as three tokens, representing "`*`", "`x`" and "`++`".

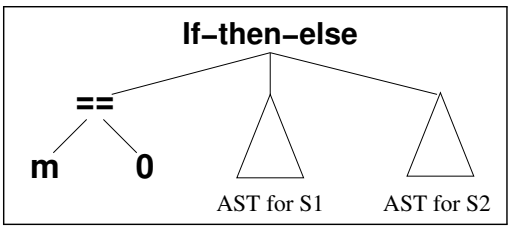
Parsing

Second step of syntax analysis

- **Objective:** Uncover the *structure* of a sentence in the program from a stream of *tokens*.
- For instance, the phrase "`x = +y`", which is recognized as four tokens, representing "x", "=" and "+" and "y", has the structure **=(x, +(y))**, i.e., an assignment expression, that operates on "x" and the expression "+(y)".
- **Output:** A *tree* called *abstract syntax tree* that reflects the structure of the input sentence.

Abstract Syntax Tree (AST)

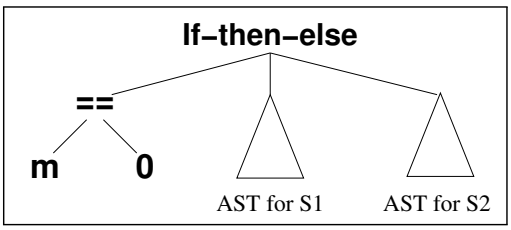
- Represents the syntactic structure of the program, hiding a few details that are irrelevent to later phases of compilation.
- For instance, consider a statement of the form: "`if (m == 0) S1 else S2`" where S1 and S2 stand for some block of statements. A possible AST for this statement is:



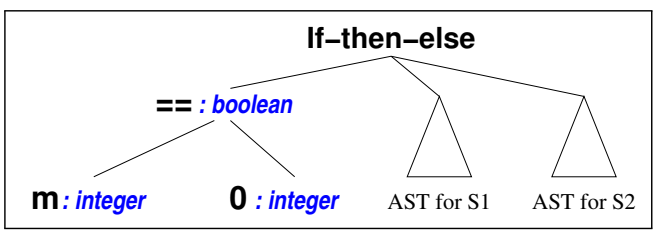
Type Checking

A instance of "Semantic Analysis"

- **Objective:** Decorate the AST with semantic information that is necessary in later phases of translation.
- For instance, the AST



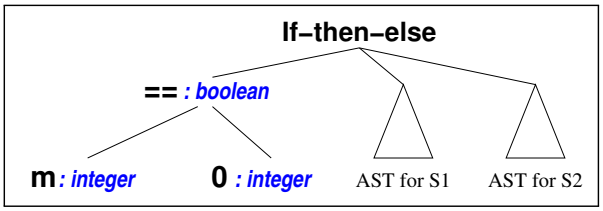
is transformed into



Intermediate Code Generation

- **Objective:** Translate each sub-tree of the decorated AST into *intermediate code*.
- Intermediate code hides many machine-level details, but has instruction-level mapping to many assembly languages.
- Main motivation for using an intermediate code is *portability*.

Intermediate Code Generation, an Example



$\Longrightarrow$

```
    loadint m
    loadimmed 0
    intequal
    jmpz  .L1
    jmp .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
    jmp .L3
.L3:
```

Code Optimization

- **Objective:** Improve the time and space efficiency of the generated code.
- Usual strategy is to perform a series of transformations to the intermediate code, with each step representing some efficiency improvement.
- *Peephole optimizations*: generate new instructions by combining/expanding on a small number of consecutive instructions.
- *Global optimizations*: reorder, remove or add instructions to change the structure of generated code.

Code Optimization, an Example

```
    loadint m
    loadimmed 0
    intequal
    jmpz  .L1
    jmp .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
    jmp .L3
.L3:
```

$\Longrightarrow$

```
    loadint m
    jmpnz  .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
.L3:
```

Final Code Generation

- **Objective:** Map instructions in the intermediate code to specific machine instructions.
- Supports standard object file formats.
- Generates sufficient information to enable symbolic debugging.

Final Code Generation, an Example

```
    loadint m
    jmpnz  .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
.L3:
```

$\Longrightarrow$

```
    movl  8(%ebp), %esi
    testl %esi, %esi
    jne  .L2
.L1:
        .... code for S1
    jmp .L3
.L2:
        .... code for S2
.L3:
```