Effort Level

The effort required by this course is **High**

... but so are the rewards:

- Hands on experience in large-scale programming ($>$ 3000 lines of Java code).
- Use of high-level tools.
- Exposure to inner workings of Object Oriented Programming.
- In-depth knowledge of *how* programs written in high-level languages are translated and executed.

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.

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

Prerequisites

Courses:

- **CSE 213/214:** Foundations, Data structures
- **CSE 303:** Automata Theory

Programming Experience:

- **Java:** classes, object, `new` etc.
- **UNIX:** Debuggers (*e.g.*, `jdb, jbuilder`), `make`, etc.

Hardware/Software Environment:

- Sun/Solaris (Graduate class), Free BSD (undergraduate lab).

Organization

Concepts and Basic Ideas in the lectures

Concrete Implementation in a large programming project:

Build your own compiler in 6 (easy?) steps.

40% of final grade

Other units of evaluation:

Mid-term Exam (30% of final grade)

Final Exam (30% of final grade)

What is a Compiler?

Programming problems are easier to solve in *high-level languages*

Languages closer to the level of the problem domain, *e.g.*,

- FORTRAN: numerical problems
- Tcl/Tk: graphical user interfaces

Solutions are usually more efficient (faster, smaller) when written in machine language

Language that reflects to the cycle-by-cycle working of a processor

Compilers are the bridges:

Tools to *translate* 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
...
```


The Rules of the Game

Project work:

- Individual projects. **No Collaboration.**
- Projects due on stated due date. Penalty of −20% per day for each additional day. **No Extensions, No Excuses.**
- Best 5 scores (out of possible 6) will be taken for programming projects.

  Essentially, you get 1 "free" project, for use in case of emergencies. Use this freedom wisely.
- Limit discussion of projects to *problems*, not *solutions*.
- Illegal collabortion and plagiarism will be treated with maximum seriousness.

Additional Information

Web page for CSE 304:

`http://www.ug.cs.sunysb.edu/~cse304`

All project announcements will be posted in the course home pages.

Office hours: Tue,Thu, 10-11am.

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 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.

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**

Phases of Translation

2. Parsing: (Syntax Analysis Phase)

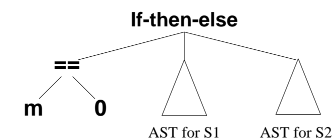
- Uncover the *structure* of a sentence in the program from a stream of *tokens*.
- For instance, the phrase "`x = +y`", which is recognized as four tokens, representing "x", "=" and "+" and "y", has the structure **=(x, +(y))**, i.e., an assignment expression, that operates on "x" and the expression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

Typically, compilers build an *abstract syntax tree* directly, skipping the construction of parse trees.

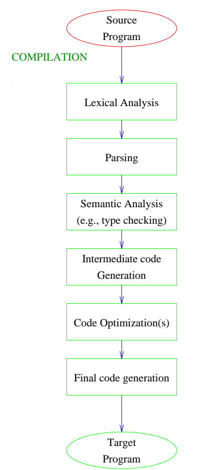
Abstract Syntax Tree (AST)

- 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:



Translation Process

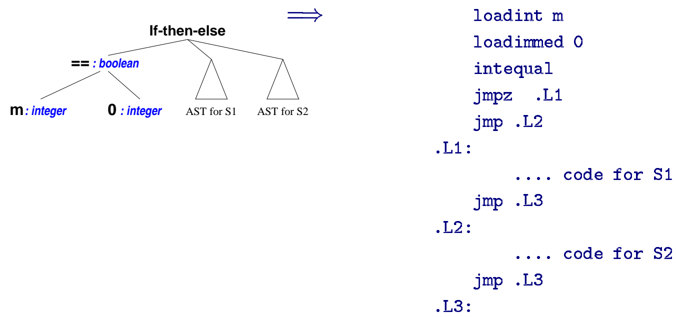


Steps of Translation

1. Lexical Analysis: (Syntax Analysis Phase)

- Convert the *stream of characters representing input program* into a sequence of *tokens*.
- Tokens are the "words" of the programming language.
- For instance, 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 "++".

Intermediate Code Generation, an Example




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


Phases of Translation

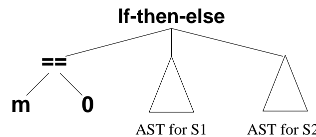
5. Code Optimization

- Apply a series of transformations to improve the time and space efficiency of the generated code.
- *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.

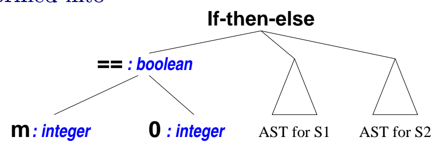
Phases of Translation

3. Type Checking: (Semantic Analysis)

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



is transformed into



Phases of Translation

4. Intermediate Code Generation:

- 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: portability.

Final Code Generation, an Example

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


Code Optimization, an Example

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


Phases of Translation

6. Final Code Generation

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