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.
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.
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 “++”.
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 “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.
Translation Steps: Abstract Syntax Tree (AST)

- Represents the syntactic structure of the program, hiding a few details that are irrelevant to later phases of compilation.

- For instance, consider a statement of the form:

\[
\text{if } (m == 0) \text{ S1 else S2}
\]

where S1 and S2 stand for some block of statements. A possible AST for this statement is:

```
If-then-else

==

m

0

AST for S1

AST for S2
```
Decoration of the AST with semantic information is necessary for later phases of translation. For instance, the AST:

If-then-else

becomes:

If-then-else

where the types are indicated explicitly.
Translation Steps: 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.
Translation Steps: Intermediate Code Generation Example

If-then-else

\[ \text{AST for S1} \quad \text{AST for S2} \]

becomes

\[ \text{R1} \leftarrow \text{mem}(m) \]
\[ \text{cmp R1, 0} \]
\[ \text{jz .L1} \]
\[ \text{jmp .L2} \]
\[ \text{.L1:} \]
\[ \text{.... code for S1} \]
\[ \text{jmp .L3} \]
\[ \text{.L2:} \]
\[ \text{.... code for S2} \]
\[ \text{jmp .L3} \]
\[ \text{.L3:} \]

\( == : \text{boolean} \)

\( m : \text{integer} \quad 0 : \text{integer} \)

\( \text{AST for S1} \quad \text{AST for S2} \)
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.

- **Intraprocedural optimizations**: reorder, remove or add instructions to change the structure of generated code within each function. Code transformations guided by static analysis.

- **Interprocedural optimizations**: Guided by interprocedural static analysis.
Translation Steps: 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.
Translation Steps: Final Code Generation Example

```
R1 ← mem(m) ⇒ movl 8(%ebp), %esi
    cmp R1, 0
    jz .L1
    jmp .L2
.L1:
    .... code for S1
    jmp .L3
.L2:
    .... code for S2
    jmp .L3
.L3:
```

**Source Program**

Lexical Analysis

Parsing

Semantic Analysis (e.g., type checking)

Intermediate code Generation

Code Optimization(s)

Final code generation

**Target Program**
Broader Applications of Languages

- **Command Interpreters**: bash, ksh, Powershell, ...
- **Programming**: Java, Python, C++, Rust, Go, Haskell, Scala, OCaml, ...
- **Document Structuring**: \LaTeX, HTML, RTF, troff, ...
- **Page Definition**: PDF, PostScript, ...
- **Databases**: SQL, ...
- **Hardware Design**: VHDL, VeriLog, ...
- **Domain-Specific Languages (DSL)**