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.

Translation Process

![Translation Process Diagram]

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 “++”.

Phases of Translation
• 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 irrelevant 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:

![If-then-else AST](image)

**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

![If-then-else AST](image)

is transformed into

![If-then-else AST](image)

**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.

**Intermediate Code Generation, an Example**
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.

**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:
```

**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.

**Final Code Generation, an Example**

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

---

**Phases of Translation**
Broader Applications of Languages

- Command Interpreters: `csh`, `perl`, ...
- Programming: `FORTRAN`, `SmallTalk`, ...
- Document Structuring: `troff`, `LaTeX`, `HTML`, ...
- Page Definition: `PostScript`, `PCL`, ...
- Databases: `SQL`, ...
- Hardware Design: `VHDL`, `VeriLog`, ...
- ... and many many more

Language Processing

Flexible control: programmable combination of primitive operations.

- Express input to the system in a well defined *language*.
- Translate the input into the sequence of primitive operations.
  - Direct execution
  - Byte code emulation
  - Object code compilation

Language processing techniques have evolved over the last 30 years.
In almost every domain, at least three steps can be identified: *lexical analysis, parsing, and syntax-directed translation.*