COP 3402 Systems Software

Lecture 4: Compilers & Interpreters
Outline

1. Compiler and interpreters
2. Compilers
3. Interpreters
4. PL/0 lexical tokens
Compilers / Interpreters

• Programming languages are notations for describing computations (to programmers and computers).

• There are three general ways for performing these computations:
  1. Compilation
  2. Interpretation
  3. Hybrid Implementation
A compiler is a program that takes a high level language (such as C) as input, and translates it to a low-level representation (machine language).
Compilers

The process of compilation takes place in several phases:

**Front end**
- Lexical Analyzer/Scanner
- Syntactic Analyzer/Parser
- Semantic Analyzer

**Back end**
- Code generator
Compilers

Lexical Analyzer/Scanner

Syntax Analyzer/Parser

Intermediate Code generator (Semantic Analyzer)

Code Generator

Code Optimizer (optional)

Machine Language

Computer

Symbol Table

Lexical units (Tokens)

Parse Trees

Intermediate Code
EXAMPLE:
(source: not PL/0, target: not PM/0)

\[ \text{Fahrenheit} := 32 + \text{celsius} \times 1.8 \]

Lexical Analyzer (Scanner)
converts a stream of character into a stream of tokens

```
[id,1][:=][int,32][+][id,2][*][real,1.8][;]
```

Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td>fahrenheit</td>
<td>real</td>
</tr>
<tr>
<td>celsius</td>
<td>real</td>
</tr>
</tbody>
</table>

Syntax Analyzer (Parser)
constructs syntactic structure of the program

```
:=

id_1

int_{32}

+ 

id_2

* 

real_{1.8}
```
Symbol Table

<table>
<thead>
<tr>
<th></th>
<th>fahrenheit</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>celsious</td>
<td>real</td>
</tr>
</tbody>
</table>

Lecture 4: Compilers & Interpreters

Context Analyzer

Determines the type of the identifier

\[ \text{id}_1 + \text{int}_{32} \times \text{id}_2 \text{ real 1.8} \]

\[ \text{id}_1 \rightarrow_r \text{inttoreal} \times_r \text{id}_2 \text{ real 1.8} \]
Symbol Table

<table>
<thead>
<tr>
<th></th>
<th>fahrenheit</th>
<th>real</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>celsius</td>
<td>real</td>
</tr>
</tbody>
</table>

Intermediate Code Generator

```
Temp1 := inttoreal(32)
Temp2 := id2
Temp2 := Temp2 * 1.8
Temp1 := Temp1 + Temp2
Id1 := Temp1
```
Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td></td>
</tr>
<tr>
<td>fahrenheit</td>
<td>real</td>
</tr>
<tr>
<td>2.</td>
<td></td>
</tr>
<tr>
<td>celsius</td>
<td>real</td>
</tr>
</tbody>
</table>

Lecture 4: Compilers & Interpreters

Intermediate code

Temp1 := inttoreal(32)
Temp2 := id2
Temp2 := Temp2 * 1.8
Temp1 := Temp1 + Temp2
Id1 := Temp1

Code Optimizer

Temp1 := id2
Temp1 := Temp1 * 1.8
Temp1 := Temp1 + 32.0
Id1 := Temp1

optimized code
Symbol Table

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fahrenheit  real</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>celsius  real</td>
</tr>
</tbody>
</table>

Code Generator

```
Temp1 := id2
Temp1 := Temp1 * 1.8
Temp1 := Temp1 + 32.0
Id1 := Temp1
```

Optimized code:
```
movf id2, r1
mulf #1.8, r1
addf #32.0, r1
movf r1, id1
```

Assembly instructions
Lexical Analyzer: 
transforms a stream of characters of the source program and produces *lexical tokens*; it discards white space and comments between the tokens

Lexical tokens of a program are:
- Identifiers
- Numbers
- Reserved words
- Arithmetic and logical operators
- ...

Syntax Analyzer:
gets tokens from the lexical analyzer and uses them to construct a hierarchical structure called *parse tree*.

Parse trees represent the syntactic structure of the program.
Compilers

Intermediate Code Generator:
produces a program in a different language representation:
  Assembly language
  Language similar to assembly language
  Language higher than assembly language

Note: Semantic Analysis is an integral part of the intermediate code generator

Optimization:
makes programs smaller or faster or both.

most optimization is done at the level of intermediate code.
(for example, tree reduction, vectorization).

See The LLVM Compiler Infrastructure [http://llvm.org/]
Compilers

**Code Generator:**
translates the optimized intermediate code into machine language.

**Symbol Table:**
serves as a database for the compilation process.

contains type and attribute information of each user-defined name in the source program.

<table>
<thead>
<tr>
<th>Index</th>
<th>name</th>
<th>type</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fahrenheit</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>celsius</td>
<td>real</td>
<td></td>
</tr>
</tbody>
</table>
Compilers

**Machine Language**
A program in machine language (assembly language) needs, in general, to be translated to object code for execution

**Assembler** is a program that translates machine language into object code
Compilers

Machine Language
To run a program in object code, in general,
• some other code (libraries) and
• some routines from the operating system (i.e. input/output routines) are needed
Interpreters

Programs are interpreted (executed) by another program called the interpreter.

Advantages: Easy implementation of many source-level debugging operations because all run-time errors refer to source-level units.

Disadvantages: 10 to 100 times slower because statements are interpreted each time the statement is executed.

Background:
Early sixties → APL, SNOBOL, Lisp.
By the 80s → rarely used.
Recent years → Significant comeback (some Web scripting languages such as JavaScript and PHP)
Interpreters

Source program → Interpreter → Result

Input data
Hybrid Implementation Systems

Java program

They translate high-level language programs to an intermediate language designed to facilitate easy interpretation

Translator

Byte code

Byte code interpreter

Machine A

Intermediate code

Byte code interpreter

Machine B
Interpreters

Just-In-Time (JIT) implementation

Programs are translated to an intermediate language.

During execution, the intermediate language methods are compiled into machine code when they are called.

The machine code version is kept for subsequent calls.

.NET and Java programs are implemented as JIT systems.