CS344 - Parsing

Announcements

- First HW will be uploaded after class

Essay
Why study programming languages?
- You will need to choose appropriate languages at some point.
- Makes it easier to learn new ones.
- Learn obscure features. - Interview prep
- Knowledge of actual implementation costs. 
  Ex: Housekeeping functions
  Passing by reference
Why? (cont.)

- Make good use of debuggers, assemblers, etc.

- Add features to older languages as needed.
Compilation versus Interpretation

2 models:

- **Source program** → **Compiler** → **Target program** → **Output**
  - Compiler example: C++
  - Target program example: Python

- **Source program** → **Interpreter** → **Output**
  - Input example: Input

Pros & Cons

Interpreter:
- greater flexibility
- better debugging
- better with data that is dependant on input

Compilation:
- much faster
Compilation vs. Interpretation

In reality, most languages are both.

This is the key.

Fuzzy: how much a translator does.
Compilers

The process by which programming languages are turned into assembly or machine code is important in programming languages.

We'll spend some time on these compilers, although it isn't a focus of this class.
Compilers

Compilers are essentially translators, so must semantically understand the code.

Output: either assembly, machine code, some other output

Java: byte code

C++ $\rightarrow$ Code
Compilers begin by preprocessing:
- remove white space and comments
- include macros or libraries
- group characters into tokens
  \[ \text{ex: } \begin{array}{l}
  \text{for } (\text{int } i=0; i<10; i++) \\
  \quad i = i + 2;
  \end{array} \]
- identify high-level syntactical structures
  \[ \text{ex: } \begin{array}{l}
  \text{loops} \\
  \text{functions}
  \end{array} \]
Overview of Compilation

Character stream

Token stream

Parse tree

Abstract syntax tree or other intermediate form

Modified intermediate form

Target language (e.g., assembler)

Modified target language

Scanner (lexical analysis)

Parser (syntax analysis)

Semantic analysis and intermediate code generation

Machine-independent code improvement (optional)

Target code generation

Machine-specific code improvement (optional)

Symbol table

Front end

Back end
Scanning (lexical analysis)
- Divide program into tokens, or smallest meaningful units
  Ex: keywords, ( ), { }, /n, ;

- Scanning + tokenizing makes parsing much simpler
- While parsers can work character by character, it is slow.

- Note: Scanning is recognizing a regular language, e.g., via DFA
- Parsing
- Recognizing a context-free language, e.g. via PDA
- Finds the structure of the program (or the syntax)

Ex.: iteration-statement →
    while (expression) statement
statement → compound-statement

Outputs a parse tree.
Semantic Analysis

This discovers the meaning of the commands.

Actually only does static semantic analysis, consisting of all that is known at compile time.

(Some things - e.g. array out of bounds - are unknown until run time.)
By: (semantic analysis)

- Variables can't be used before being declared.
- Type checking.
- Identifiers are used in proper context.
- Functions have correct inputs & returns.

etc... (very language dependent)
Intermediate Form

This is the output of the "front end"

- Often this is an abstract syntax tree - a simplified version of a parse tree
- May also be a type of assembly-like code

<table>
<thead>
<tr>
<th>Index</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>void</td>
<td>type</td>
</tr>
<tr>
<td>2</td>
<td>int</td>
<td>type</td>
</tr>
<tr>
<td>3</td>
<td>getint</td>
<td>func: (1) \rightarrow (2)</td>
</tr>
<tr>
<td>4</td>
<td>putint</td>
<td>func: (2) \rightarrow (1)</td>
</tr>
<tr>
<td>5</td>
<td>i</td>
<td>(2)</td>
</tr>
<tr>
<td>6</td>
<td>j</td>
<td>(2)</td>
</tr>
</tbody>
</table>
Code generation and improvement

Abstract syntax tree or other intermediate form
    \[ i = j + k \]
Modified intermediate form
Target language (e.g., assembler)
Modified target language

Intermediate code generation
    Machine-independent code improvement (optional)
Target code generation
    Machine-specific code improvement (optional)

Symbol table
Back end

Creating correct code is generally not difficult.
Optimization of that code is.
Next Time

Scanning and regular languages.