Motivation: Why study about compilers?

Because Compilers ...

- Are an essential part of applied computer science
- Are very relevant to computational linguistics
- Are implemented using classical programming techniques
- Employ important software engineering principles
- Train you in developing software for transforming one structure to another (programs, files, transactions, ...)
- Train you to think in terms of “description languages”.
- Parsing files of some complex syntax is very common in many applications.
Compiler architecture (front end)

- Compiler architecture (front end)
  - Jack Compiler
  - Syntax Analyzer
  - Code Generation

Syntax analysis: understanding the structure of the source code

- Tokenizing: creating a stream of "atoms"
- Parsing: matching the atom stream with the language grammar

XML output = one way to demonstrate that the syntax analyzer works

Code generation: reconstructing the semantics using the syntax of the target code.

Tokenizing / Lexical analysis / scanning

- Remove white space
- Construct a token list (language atoms)
- Things to worry about:
  - Language specific rules: e.g. how to treat "++"
  - Language-specific classifications: keyword, symbol, identifier, integer constant, string constant...

While we are at it, we can have the tokenizer record not only the token, but also its lexical classification (as defined by the source language grammar).

C function to split a string into tokens

- char* strtok (char* str, const char* delimiters);
- str: string to be broken into tokens
- delimiters: string containing the delimiter characters

```
/* strtok example */
#include <stdio.h>
#include <string.h>

int main ()
{
    char str[] = "This, a sample string.");
    char *pch;
    printf ("Splitting string "%s" into tokens:\n", str);
    while (pch = strtok (strrchr (str, " ",&str)));
    while (pch != NULL)
    {
        printf ("%s", pch);
        pch = strtok (NULL, " ",&str)));
    }
    return 0;
}
```

Output:

- splitting string "This, a sample string." into tokens: This
- a sample
- string

Jack Tokenizer

```
if (x < 153) {let city = "Paris";}
```

Tokenizer's output

```
<tokens>
<keyword> if </keyword>
<symbol> ( </symbol>
<identifier> x </identifier>
<symbol> < </symbol>
<integerConstant> 153 </integerConstant>
<symbol> ) </symbol>
<keyword> let </keyword>
<identifier> city </identifier>
<symbol> = </symbol>
<stringConstant> Paris </stringConstant>
<symbol> ; </symbol>
</tokens>
```
Parsing

- The tokenizer discussed thus far is part of a larger program called parser.
- Each language is characterized by a grammar. The parser is implemented to recognize this grammar in given texts.
- The parsing process:
  - A text is given and tokenized
  - The parser determines whether or not the text can be generated from the grammar.
  - In the process, the parser performs a complete structural analysis of the text.
- The text can be in an expression in a:
  - Natural language (English, ...)
  - Programming language (Jack, ...).

Parsing examples

- **English**
  - He ate an apple on the desk.

- **Jack**
  - (5+3)*2 - sqrt(9*4)

Regular expressions

- **a|b*:**
  - {ε, "a", "b", "bb", "bbb", ...}

- **(a|b)*:**
  - {ε, "a", "b", "aa", "ab", "ba", "bb", "aaa", ...}

- **ab*(c|ε):**
  - {ε, "ac", "ab", "abc", "abb", "abbc", ...}

Context-free grammar

- **S → ():**
- **S → (S):**
- **S → SS:**
- **S → a|aS|bS:**
  - strings ending with 'a'
- **S → x:**
- **S → y:**
- **S → S+S:**
- **S → S*S:**
- **S → S/S:**
- **S → (S):**
- **(x+y)*x-x*y/(x+x):**

Simple (terminal) forms / complex (non-terminal) forms

Grammar = set of rules on how to construct complex forms from simpler forms

Highly recursive.
Recursive descent parser

\[ A = bBcC \]
\[ A = (bB)^* \]

\[
A() \\
\{ \\
\hspace{1em} \text{if (next()=="b")} \{ \text{eat('b'); } B(); \} \\
\hspace{1em} \text{else if (next()=="c")} \{ \text{eat('c'); } C(); \} \\
\}
\]

A typical grammar of a typical C-like language

```
while (expression) {
  if (expression) 
    statement;
  while (expression) {
    statement; 
  } 
}
```

```
whileStatement: 'while' '(' expression ')' statement
ifStatement: simpleIf 
  | ifElse
simpleIf: 'if' '(' expression ')' statement
ifElse: 'if' '(' expression ')' statement 
  'else' statement
statementSequence: 'null' // null, i.e. the empty sequence 
  | statement '; ' statementSequence
expression: // definition of an expression comes here 
  // more definitions follow
```

Parse tree

```
while (count<=100) {
  ** demonstration */
  count++; // ...
}
```

Tokenized:

```
while(count<=
100) {
count
++;
... 
```

Recursive descent parsing

```
... 
statement: whileStatement 
  | ifStatement 
  | ... // other statement possibilities follow
  | '{' statementSequence '}'
whileStatement: 'while' '(' expression ')' statement
ifStatement: ... // if definition comes here
statementSequence: 'null' // null, i.e. the empty sequence 
  | statement '; ' statementSequence 
expression: // definition of an expression comes here 
  // more definitions follow
```

- Highly recursive
- LL(0) grammars: the first token determines in which rule we are
- In other grammars you have to look ahead 1 or more tokens
- Jack is almost LL(0).
The Jack grammar

```
x: x appears verbatim
x: x is a language construct
x?: x appears 0 or 1 times
x*: x appears 0 or more times
x|y: either x or y appears
(x,y): x appears, then y
```

Program structure: A Jack program is a collection of classes, each appearing in a separate file. The compilation unit is a class. A class is a sequence of tokens structured according to the following context-free syntax:

- `class`: `class` className `stuff` `{` `stuff` `}`
- `function`: `function` `stuff` `{` `stuff` `}`
- `variable decl`: `var` `stuff` `;`
- `expression`: `stuff` `;`
- `identifier`: `stuff`
- `symbol`: `stuff`
- `integer constant`: `stuff`
- `string constant`: `stuff`
- `program`: `stuff`

The syntax analyzer's algorithm shown in this slide:

- If `xxx` is non-terminal, output: `<xxx`
- Recursive code for the body of `xxx`
- `)`
- If `xxx` is terminal (keyword, symbol, constant, or identifier), output: `<xxx`
- `xxx`
- `)`
- If `xxx` is variable, output: `<xxx`
- `value`

JackTokenizer: a tokenizer for the Jack language (proposed implementation)
JackTokenizer (cont.)

| symbol       | --  | Char | Returns the character which is the current token. Should be called only when 
|              |     |      | tokenType() is Symbol. |
| identifier   | --  | String | Returns the identifier which is the current token. Should be called only when 
|              |     |      | tokenType() is IDENTIFIER |
| instVal      | Int |      | Returns the integer value of the current token. Should be called only when 
|              |     |      | tokenType() is INT_CONST |
| stringVal    | String |      | Returns the string value of the current token, without the double quotes. Should 
|              |     |      | be called only when tokenType() is STRING_CONST |

CompilationEngine: a recursive top-down parser for Jack

The CompilationEngine effects the actual compilation output.

It gets its input from a JackTokenizer and emits its parsed structure into an 
output file/stream.

The output is generated by a series of compilexxx() routines, one for every 
syntactic element xxx of the Jack grammar.

The contract between these routines is that each compilexxx() routine should 
read the syntactic construct xxx from the input, advance() the tokenizer 
exactly beyond xxx, and output the parsing of xxx. 
Thus, compilexxx() may only be called if indeed xxx is the next syntactic 
element of the input.

In the first version of the compiler, which we now build, this module emits a 
structured printout of the code, wrapped in XML tags (defined in the specs of 
project 10). In the final version of the compiler, this module generates 
executable VM code (defined in the specs of project 11).

In both cases, the parsing logic and module API are exactly the same.

CompilationEngine (cont.)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
</table>
| Constructor       | Input stream file, Output stream file | --      | Creates a new compilation engine with the given input and output. The next routine 
called must be compileClass(). |
| CompileClass      | --        | --      | Compiles a complete class. |
| CompileClassVarDec | --        | --      | Compiles a static declaration or a field 
declaration. |
| CompileSubroutine | --        | --      | Compiles a complete method, function, or 
constructor. |
| compileParameterList | --        | --      | Compiles a (possibly empty) parameter list, not 
including the enclosing "()". |
| compileVarDec     | --        | --      | Compiles a var declaration. |

CompilationEngine (cont.)

<table>
<thead>
<tr>
<th>Routine</th>
<th>Arguments</th>
<th>Returns</th>
<th>Function</th>
</tr>
</thead>
</table>
| compileStatements | --        | --      | Compiles a sequence of statements, not 
including the enclosing "[]". |
| compileDo        | --        | --      | Compiles a do statement. |
| compileLet       | --        | --      | Compiles a let statement. |
| compileWhile     | --        | --      | Compiles a while statement. |
| compileReturn    | --        | --      | Compiles a return statement. |
| compileIf        | --        | --      | Compiles an if statement, 
possibly with a trailing else clause. |
CompilationEngine (cont.)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Syntax Analysis</th>
<th>Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CompileExpression</td>
<td>Compiles an expression.</td>
<td></td>
</tr>
<tr>
<td>CompileTerm</td>
<td>Compiles a term. This routine is faced with a slight difficulty when trying to decide between some of the alternative parsing rules. Specifically, if the current token is an identifier, the routine must distinguish between a variable, an array entry, and a subroutine call. A single look-ahead token, which may be one of &quot;[&quot;, &quot;,&quot;, or &quot;]&quot;, suffices to distinguish between the three possibilities. Any other token is not part of this term and should not be advanced over.</td>
<td></td>
</tr>
<tr>
<td>CompileExpressionList</td>
<td>Compiles a (possibly empty) comma-separated list of expressions.</td>
<td></td>
</tr>
</tbody>
</table>

Summary and next step

- Syntax analysis: understanding syntax
- Code generation: constructing semantics

The code generation challenge:
- Extend the syntax analyzer into a full-blown compiler that, instead of generating passive XML code, generates executable VM code
- Two challenges: (a) handling data, and (b) handling commands.

Perspective

- The parse tree can be constructed on the fly
- Syntax analyzers can be built using:
  - Lex tool for tokenizing (flex)
  - Yacc tool for parsing (bison)
  - Do everything from scratch (our approach ...)
- The Jack language is intentionally simple:
  - Statement prefixes: let, do, ...
  - No operator priority
  - No error checking
  - Basic data types, etc.
- Richer languages require more powerful compilers
- The Jack compiler: designed to illustrate the key ideas that underlie modern compilers, leaving advanced features to more advanced courses
- Industrial-strength compilers: (LLVM)
  - Have good error diagnostics
  - Generate tight and efficient code
  - Support parallel (multi-core) processors.