Compiler I: Syntax Analysis

Course map

Abstract design
Chapters 9, 12

Human Thought

Machine Language
Chapters 4 - 5

Computer Architecture
Chapters 4 - 5

Hardware Platform
Chapters 1 - 3

Gate Logic
Chapters 7 - 8

VM Translator

Compiler

Virtual Machine

Software hierarchy

Assembly Language

Electrical Engineering

Physics

Hardware
hierarchy

Assembler
Chapter 6

Operating Sys.

H.L. Language

abstract interface

Chapters 10 - 11

Compiler I: Syntax Analysis
The big picture

Modern compilers:
- **Front-end:** from high-level language to intermediate language
- **Back-end:** from intermediate language to binary code.

Compiler anatomy (front end)

**Front-end:**
- **Syntax analysis:** understanding the code
- **Code generation:** constructing semantics

**Syntax analyzer:**
- **Tokenizing:** creating a list of "atoms"
- **Parsing:** matching the atom list with the language grammar
- **XML output:** proof that the syntax analyzer is parsing correctly.
Tokenizing / Lexical analysis

Remove white space
Construct a token list (language atoms)

Things to worry about:

- Language specific rules:
  e.g. how to treat "++"
- Language specific token types:
  keyword, identifier, operator, constant, ...

Jack Tokenizer

Source code

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

Tokenizer's output

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

- Each language has a grammar
- A text is given:
  - The parser, using the grammar, can either accept or reject the text
  - In the process, the parser performs a complete analysis of the text
- The language can be:
  - Context-dependent (English, ...)
  - Context-free (Jack, ...).

Examples

<table>
<thead>
<tr>
<th>Context Free</th>
<th>Context Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>(5+3)<em>2 - sqrt(9</em>4)</td>
<td>she discussed sex with her priest</td>
</tr>
</tbody>
</table>

Parse 1:
- discussed
- sex
- her priest

Parse 2:
- discussed
- sex
- her priest
A typical grammar (C/Java-like)

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple (terminal) forms / complex (non-terminal) forms</td>
<td>Grammar = set of rules on how to construct complex forms from simpler forms</td>
</tr>
<tr>
<td>Highly recursive</td>
<td></td>
</tr>
</tbody>
</table>

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

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

Parse tree

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

```
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, 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)

parseStatement()  
parseWhileStatement()  
parseIfStatement()  
parseStatementSequence()  
parseExpression().

A linguist view on parsing

Parsing:

One of the mental processes involved in sentence comprehension, in which the listener determines the syntactic categories of the words, joins them up in a tree, and identifies the subject, object, and predicate, a prerequisite to determining who did what to whom from the information in the sentence.

(Steven Pinker,  
The Language Instinct)
Another perspective on parsing

The Jack grammar

**Lexical elements:**

The Jack language includes five types of terminal elements (tokens):

- **keyword**
- **symbol**
- **integerConstant**
- **stringConstant**
- **identifier**

**Program structure:**

A Jack program is a collection of class, 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:: classDecl
classDecl:: {className (class?superClass*) classVarDec* type* subroutineDec*}

classVarDec:: {static|field} type varDecList

type:: int|char|boolean|className

subroutineDec:: {constructor|function|method} (void|type) subroutineName

parameterList:: (type varDecList | voidDec)*

subroutineBody:: {"{" statements "}"

voidDec:: varDecList

className:: identifier

varDecList:: varDec ("," varDec)*
```

- `*` appears verbatim
- `x` is a language construct
- `x?` appears 0 or 1 times
- `x*` appears 0 or more times
- `x|y` either `x` or `y` appears
- `x,y` appears, then `y`
The Jack grammar (cont.)

Statements:
- `statement`:
  - `statement`:
  - `letStatement | doStatement | whileStatement | returnStatement`
  - `letStatement`: `let varName ('expression')? 'expression'`
  - `doStatement`: `do subroutineCall`?
  - `whileStatement`: `while ('expression') ('statement')`
  - `returnStatement`: `return 'expression'`?

Expressions:
- `expression`: `term (op term)*`
- `term`: `integerConstant | stringConstant | keywordConstant | varName | varName '(' expression ')' | subroutineCall '(' expression ')' | unaryOp term`
- `subroutineCall`: `subroutineName '(' expressionList ')' | subroutineName '(' expressionList ')'`
- `expressionList`: `(expression (',' expression)*?)`
- `op`: `+ | - | * | / | % | & | | ^ | ? | =`
- `unaryOp`: `~ | !`
- `keywordConstant`: `true | false | null | this`

Class Bar {
    method Fraction foo(int y) {
        var int temp; // a variable
        let temp = (xxx+12)*-63;
        ...
        ...
    }
}

Syntax analyzer in action

- Using the language grammar, a programmer can write a syntax analyzer program.
- The syntax analyzer takes a source text file and attempts to match it on the language grammar.
- If successful, it generates a parse tree in some structured format, e.g., XML.

This analyzer’s algorithm:
- If `xxx` is a non-terminal, output:
  - `<xxx>`
  - Recursive code for the body of `xxx`:
    - `</xxx>`
- If `xxx` is terminal (keyword, symbol, constant, or identifier), output:
  - `<xxx>`
  - `xxx value`
  - `</xxx>`
Summary and next step

- Syntax analysis: understanding the code
- Code generation: constructing semantics

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

The parse tree can be constructed “on-line”
- Bottom up compilation is harder and more powerful
- Syntax analyzers are typically built using tools like:
  - `Lex` for tokenizing
  - `Yacc` for parsing
- Jack simplifications:
  - Statement prefixes: `let, do, ...`
  - No operator priority
  - No error checking
  - Many more simplifications.