Recitation #1

Administrative

- **Labs** every week 8:30 – 9:20 a.m. and 9:30 – 10:20 a.m.
  - No enforcement of attendance, but if you never show up for the lecture or the lab we will have less sympathy for you if you run into problems later 😊
  - Doesn’t matter to which lab you go
  - Since Dr. Hughes doesn’t have lab assignments per se, the lab is driven by your questions and a few assignments of my choosing
- All assignments (besides the final one) are worth 20% of your grade
- Any questions on lecture content so far?

Use of IDE

- Choose the one you like the most:
  - Visual Studio 2010 Express ➔ you can download full version of Visual Studio 2010 from Dreamspark ([http://www.dreamspark.com](http://www.dreamspark.com))
    - Choose Visual C++ ➔ Empty Project
    - If you prefer you can change to C compiler (instead of C++)
    - Probably most advanced debugging tool
    - Best choice under Windows development
  - Eclipse CDT (Eclipse extension for C/C++) ➔ prefer Wascana ([http://www.eclipse.org/p/wascana](http://www.eclipse.org/p/wascana)), because it contains all Gnu development tools
    - Nice debugging capabilities
  - CodeBlocks ([http://www.codeblocks.org/](http://www.codeblocks.org/)) ➔ can use Gnu Gcc or Visual Studio compiler
    - Debugging is not quite as feature-rich
- All are freely available
- Eclipse and CodeBlocks can be used on Linux/Windows/Mac
- At the end submit vanilla C/C++ source code file ➔ No IDE or operating system-specific files please ➔ easier for me to grade if I can compile and run it automatically with Python script

Graphical Debugger

- Please NEVER use printouts for debugging your code
- A Graphical debugger is the way to go ➔ every IDE has a good debugger built-in
- Great way to see variable values and trace through code
- Give a little demo on Visual Studio
Most common sources of errors for students

1. **Make up your own test cases.** Many students just rely on the test cases on the assignment sheet or the additional test cases you give them, rather than making up their own and testing the code extensively.

2. **Modularity.** Students should learn to develop and test the code in stages/modules. One thing I noticed a lot in office hours is that students think of their code as this monolithic black box that should take your input file and print the desired output. If that doesn't work, they tinker in all kinds of different places without actually knowing what components of the code are actually working and which the culprits for the faulty output.

3. **Debugging.** This relates to (2), but I think any programmer should know how to use a graphical debugger. No matter what IDE you use to develop your code, it will include a graphical debugger. You can set breakpoints, watch variable values change, and step through the code line-by-line if necessary. Many students still rely on printouts in their code, which makes finding bugs a lot harder.

Structure of a Compiler

- Program compilation goes through several steps from source code to executable machine code
- Compiler should try to optimize for computer architecture (number of registers, instruction set)
- Different translation paradigms for program execution:
  - *Interpretation:* Translates source code into some efficient intermediate representation (code) and immediately executes this.
  - *Just-in-time (JIT) compilation:* Compile source code into machine-independent intermediate code (e.g. Java Bytecode, .NET Common Language Runtime), interpreter translates intermediate code into native machine code at runtime.
- **Compiler construction?**
  - Easiest solution: Write compiler in other high-level language
  - But then, how did the first high-level language come about → chicken-and-egg problem
  - In assembler? But then how was the compiler for assembler produced?
  - How does Bootstrapping work?
    - Define one language as a subset of another
    - First write compiler for subset 1 of assembler in machine code, this does very little, e.g. convert mnemonic opcodes into binary form.
    - Then write compiler for subset 2 of assembler in subset 1 of assembler, and so on.
    - The high-level language compiler bootstraps itself through low-level subsets.
  - Other solution: Have compiler for different machine, cross-compile to target machine.
Lexical Analyzer

- Splits up the source code into meaningful sequences called “lexemes” or “tokens” → \([\text{token} = \text{name}, \text{attribute} = \text{name}]\)
- No syntax or semantic analysis yet

Lexical units of a program are:
  - Identifiers (e.g. variables, function names)
  - special words (reserved words, e.g. “true”, “false”, “program”)
  - operators (e.g. “+”, “-“)
  - special symbols (e.g. “:=”, “;”, “(“)
  - Comments are ignored!

Pascal-S

- Subset (S) of Pascal
- The language is so small that it is very instructive to look at the compiler implementation
- Maybe look at example code on slide 80
- Arrays are 1-based
- Download DevPas from [http://www.bloodshed.net/devpascal.html](http://www.bloodshed.net/devpascal.html) → Take a look at example program Multiply

Assignment #1

- Part 1: Take Lexical Analyzer from PascalS and translate it to C or C++ – code → due Feb 2nd
- Part 2: Extend lexical analyzer to accept more features of the language → due Feb 9th
• Show output of program “PascalSLex.exe” on “multiply.pas”

![Program Output]

- Notice how source code is replaced by tokens, e.g. “program” → “programs” or “(“ → “lparent”

• Show source code of program “PascalSLex.pas”

- Notice list of defined tokens
- Maximum length of identifiers (e.g. variable names) is 10
- key: array [1..nkw] → set of keywords
- ksy: array [1..nkw] → tokens associated with keywords
- sps: array [char] → special symbols and their tokens
- Look in “init” function for assignment of those

Main program:

```
begin (* main program *)
  writeln;
  writeln('Pascal-S compiler/interpreter');
  write('Enter name of file to be compiled: '); readln(program);
  assign(srcfil,program);
  reset(srcfil);
  init;
  while true do insymbol;
  99;
  readln;
end. (* PascalSLex *)
```

- Show ASCII table
- Deal with Ordinals to convert easily from character to integer number in function “readscale”: ord(ch) - ord('0')
- Function “insymbol”: while ch <= ' ' ignores all special characters
  - If word, make characters lowercase by adding 32: 65 “A” → 97 “a” and so on
• Grammar for Pascal-S is given in EBNF form

**Extended Backus-Naur Form (EBNF)**

A [code](#), e.g. source code of a [computer program](#) consisting of [terminal symbols](#)—that is, visible characters, digits, punctuation marks, white space characters, etc.

The EBNF defines [production rules](#) where sequences of symbols are respectively assigned to a [nonterminal](#):

```
digit excluding zero = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
digit                = "0" | digit excluding zero;

This production rule defines the nonterminal digit which is on the left side of the assignment. The vertical bar represents an alternative and the terminal symbols are enclosed with quotation marks followed by a semicolon as terminating character. Hence aDigit is a 0 or a digit excluding zero that can be 1 or 2 or 3 and so forth until 9.

A production rule can also include a sequence of terminals or nonterminals, each separated by a comma:

```
twelve               = "1", "2";
two hundred one      = "2", "0", "1";
three hundred twelve = "3", twelve;
twelve thousand two hundred one = twelve , two hundred one;
```

Expressions that may be omitted or repeated can be represented through curly braces { ... }:

```
natural number = digit excluding zero , { digit };
```

In this case, the strings 1, 2, ..., 10, ..., 12345, ... are correct expressions. To represent this, everything that is set within the curly braces may be repeated arbitrarily often, including not at all.

An option can be represented through squared brackets [ ... ]. That is, everything that is set within the square brackets may be present just once, or not at all:

```
integer = "0" | [ "-" ] , natural number;
```

Therefore an integer is a zero (0) or a natural number that may be preceded by an optional minus sign.

EBNF also provides, among other things, the syntax to describe repetitions of a specified number of times, to exclude some part of a production, or to insert comments in an EBNF grammar.
Table of symbols

The following represents a proposed standard.

<table>
<thead>
<tr>
<th>Usage</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>definition</td>
<td>=</td>
</tr>
<tr>
<td>concatenation</td>
<td>,</td>
</tr>
<tr>
<td>termination</td>
<td>;</td>
</tr>
<tr>
<td>alternation</td>
<td></td>
</tr>
<tr>
<td>option</td>
<td>[ ... ]</td>
</tr>
<tr>
<td>repetition</td>
<td>{ ... }</td>
</tr>
<tr>
<td>grouping</td>
<td>( ... )</td>
</tr>
<tr>
<td>terminal string</td>
<td>&quot; ... &quot;</td>
</tr>
<tr>
<td>terminal string</td>
<td>' ... '</td>
</tr>
<tr>
<td>comment</td>
<td>(' ... ')</td>
</tr>
<tr>
<td>special sequence</td>
<td>? ... ?</td>
</tr>
<tr>
<td>exception</td>
<td>-</td>
</tr>
</tbody>
</table>