ΕΠΛ323 - Θεωρία και Πρακτική Μεταγλωττιστών

Lecture 1
Introduction
Elias Athanasopoulos
eliasathan@cs.ucy.ac.cy
Contract - Logistics

• Office hours
  – Every Tuesday, 10:00 – 12:00, B105 (ΘΕΕ01)

• Credits: 7.5 ECTS

• Lecture Timetable
  – Monday, Thursday, 12:00 – 13:30, 006 (ΧΩΔ01)
  – Wednesday, 9:00-10:00, 110 (ΧΩΔ01), only when announced!
Contract - Logistics

• Labs (Dr. Petros Panayi)
  – Every Wednesday, B103 (ΘΕΕ01)
  – Group 1: 15:00 - 17:00
  – Group 2: 17:00 - 19:00
Contract - Logistics

• Score
  – Programming Assignments (Quizzes), 15%
  – Project (multiple steps), 15%
  – Midterm, 30%
  – Final, 40%

• Requirements
  – Average grade of written exams should be at least 4.5
  – Final grade should be at least 5
Contract - Logistics

• Required courses
  – CS132 (C programming),
  – CS211 (Complexity),
  – CS231 (Data Structures)

Students with conflicts should let me know
Contract - Logistics

• Reading material
  
Communication

• Labs
  – Blackboard

• Lectures
  – http://www.cs.ucy.ac.cy/courses/EPL323
INTRODUCTION

(Chapter 1 from Dragon Book)
Why this course is important?

• Many **core concepts** of CS in a real-life setting
  – CS132: Behind the scenes of C programming
  – CS211, CS231: DFAs, parsing algorithms
  – CS372, CS221: Assembly, Computer Architecture

• Compilers are everywhere
  – From exotic devices (IoT) to web browsers (JavaScript engines, DOM parsers, etc.)

• Tweaking compilers for Security
Compiler is a tough one

• Significant effort to build a compiler from scratch
  – First Fortran compiler took 18 staff-years

• Nowadays, many tools exist to help
  – Our understanding is much better
  – Better techniques, better programming languages
What is a compiler?

• A compiler is a program that
  – reads a program written in one language (source)
  – and translates it to an equivalent program in another language (target)
  – **important**: error reporting during translation
Examples

• GCC (Gnu Compiler Collection)
  – gcc, g++, javac, etc.

• LLVM (Low Level Virtual Machine)
  – clang, clang++

• “Compilers” are everywhere,
  – Pretty printers (i.e., color syntax in editors), static checkers, interpreters (i.e., scripting languages), etc.
Analysis-Synthesis Model

• There are two parts in compilation:
  – Analysis
  – Synthesis

• Analysis
  – Breaks up the source program to subparts and creates intermediate representation(s)

• Synthesis
  – Constructs the target program from intermediate representation(s)
Example 1 (LaTeX)

\begin{table}[tb]
  \centering
  \caption{We name gadgets based on their type (prefix), payload (body), and exit instruction (suffix). In total, we name 2\times3\times3=18 different gadget types.}
  \begin{tabular}{|c|c|c|}
    \hline
    \textbf{Gadget type} & \textbf{Payload instructions} & \textbf{Exit instruction} \\
    \hline
    \hline
    \textbf{Prefix} & \textbf{Body} & \textbf{Suffix} \\
    \hline
    \begin{tabular}{l}
      CS - Call site\\
      EP - Entry point\\
    \end{tabular} & ... & \\
    \hline
  \end{tabular}
\end{table}

TABLE II: We name gadgets based on their type (prefix), payload (body), and exit instruction (suffix). In total, we name 2\times3\times3=18 different gadget types.

<table>
<thead>
<tr>
<th>Gadget type</th>
<th>Payload instructions</th>
<th>Exit instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>Body</td>
<td>Suffix</td>
</tr>
<tr>
<td>CS - Call site</td>
<td>IC - Indirect call</td>
<td>R - Return</td>
</tr>
<tr>
<td>EP - Entry point</td>
<td>F - Fixed function call</td>
<td>IC - Indirect call</td>
</tr>
<tr>
<td>none</td>
<td>Other instructions</td>
<td>IJ - Indirect jump</td>
</tr>
</tbody>
</table>

$pdflatex$ main.tex
Example 2 (Database)

```
SELECT AVG(grade)
FROM student, class
WHERE class.name = "epl223" AND class.id = student.id;
```
Requirements

• Compiler
  – Reliability
  – Fast execution
  – Low memory overhead
  – Good error reporting
  – Error recovery
  – Portability
  – Maintainability

• Target program
  – Fast execution
  – Low memory overhead
Source code/program

• Easy to read/write by human

```c
int expr(int n) {
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```
Assembly and Machine Code

- Optimized for execution by a machine (CPU)
- Less descriptive
- Hard to be processed by a human

```
lda $30,-32($30)
stq $26,0($30)
stq $15,8($30)
bis $30,$30,$15
bis $16,$16,$1
stl $1,16($15)
lds $f1,16($15)
sts $f1,24($15)
ldl $5,24($15)
bis $5,$5,$2
s4addq $2,0,$3
ldl $4,16($15)
mull $4,$3,$2
ldl $3,16($15)
```
Optimizations

- Compilers have several layers of optimizations

```c
int expr(int n){
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

No optimizations

```c
$ gcc -00
```

```
.expr:
    stw 31,-4(1)               lwz 11,64(31)
    stwu 1,-40(1)              addi 9,11,1
    mr 31,1                    mullw 0,0,9
    stw 3,64(31)               stw 0,24(31)
    lwz 0,64(31)               lwz 0,24(31)
    mr 9,0                     mr 3,0
    slwi 0,9,2                 b L..2
    lwz 9,64(31)               L..2:
    mullw 0,0,9                lwz 1,0(1)
    lwz 11,64(31)              lwz 31,-4(1)
    addi 9,11,1                blr
    mullw 0,0,9
```

Optimizations

```c
$ gcc -O3
```

```
.expr:
    addi 9,3,1
    slwi 0,3,2
    mullw 3,3,0
    mullw 3,3,9
    mullw 3,3,9
    blr
```
Cross-compiler

• Compilers can generate code for different machines (targets)

```c
int expr(int n){
    int d;
    d = 4 * n * n * (n + 1) * (n + 1);
    return d;
}
```

For x86

```
$ gcc -O3 -b i586
```

```c
expr:
    pushl %ebp
    movl %esp, %ebp
    movl 8(%ebp), %eax
    leal 1(%eax), %edx
    imull %eax, %eax
    imull %edx, %eax
    imull %edx, %eax
    sall $2, %eax
    popl %ebp
    ret
```

For PowerPC

```
$ gcc -O3 -b powerpc
```

```c
.exp:
    addi 9,3,1
    slwi 0,3,2
    mullw 3,3,0
    mullw 3,3,9
    mullw 3,3,9
    blr
```
Compilation life cycle

• Phases
  – Source code is transformed to intermediate representations
  – Each intermediate representation is suitable for a particular processing (lexical, syntax, optimization, etc.)

• In each phase the program is translated to a form closer to the machine representation and less similar to the (human-oriented) source representation
Compiler Phases

1: Tokens (Διακριτικά)
2: Syntax tree (Συντατικό δένδρο)
3: Intermediate code (Ενδιάμεσος κώδικας)
Analysis of the source program

• Linear analysis
  – Source is treated as a stream of characters (left-to-right) and is grouped into tokens

• Hierarchical analysis
  – Tokens are further grouped in larger grammatical structures (e.g., nested parentheses and blocks)

• Semantic analysis
  – Certain checks are performed to ensure the validity of the identified grammatical structures
Lexical Analysis
Λεξιλογική Ανάλυση

• Linear scanning

• Consider the expression
  \[ \text{position} := \text{initial} + \text{rate} \times 60 \]

• Lexical analysis produces
  \[ \text{id}(1) \ \text{op}(:=) \ \text{id}(2) \ \text{op}(+) \ \text{id}(3) \ \text{op}(*) \ \text{cons}(60) \]
  \[ \text{id} \text{: identifier, \ op: operator, \ cons: constant} \]

• Symbol Table

<table>
<thead>
<tr>
<th></th>
<th>position</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>initial</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>rate</td>
<td>...</td>
</tr>
<tr>
<td>4</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Syntax Analysis

• Hierarchical
• Involves grouping the tokens into grammatical phases
• Constructs the structure with the token relationship

position := initial + rate * 60
Simple Grammar

• The hierarchical structure of the program is usually expressed by recursive rules

1. Any identifier is an expression
2. Any number is an expression
3. If $expression_1$ and $expression_2$ are expressions, then so are:
   
   $expression_1 + expression_2$
   $expression_1 * expression_2$
   $( expression_1 )$
Applying the grammar

(1) Any identifier is an expression
(2) Any number is an expression
(3) If expression\(_1\) and expression\(_2\) are expressions, then so are:

\[
\text{expression}_1 + \text{expression}_2 \\
\text{expression}_1 * \text{expression}_2 \\
( \text{expression}_1 )
\]
Semantic Analysis

- Checks the program for semantic errors
- Gathers type information
- Operands and operators
- Type-checking

```
position := initial + rate * 60
```

`int_2_real()` is an extra node for converting 60 to a real number. Remember: the machine representation of integers and real numbers is different!
Error detection and reporting

- All phases can issue errors
- A compiler that stops at the first error is not helpful
- Most of the errors are handled in the syntax/semantic analysis phases
  - Lexical analysis detects errors where a stream of characters does not form a valid token
  - Syntax analysis detects errors where the stream of valid tokens violate the structure rules (syntax)
  - Semantic analysis detects errors where the syntax is valid by the operation not (adding an array with a real number)
Intermediate Code and Optimization

• Each phase produces intermediate code
  
  \[
  \text{temp1} := \text{int}_2\_\text{real}(60) \\
  \text{temp2} := \text{id}(3) \times \text{temp1} \\
  \text{temp3} := \text{id}(2) + \text{temp2} \\
  \text{id}(1) := \text{temp3}
  \]

• Optimization
  
  \[
  \text{temp1} := \text{id}(3) \times 60.0 \\
  \text{id}(1) := \text{id}(2) + \text{temp1}
  \]

three-address code: a simple assembly-like language, which consists of instructions, each of which has at most three operands

Operator: τελεστής
Operand: τελεστέος
Code Generation

• The last phase of the compiler is the generation of the target code

• Register allocation
  – Each expression should use registers that are available

• Relocation information
  – Variables are stored in relocatable addresses

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```
Generic Picture
Compiler and Friends

Source code

Compiler

Assembly code

Assembler

Machine code

Linker

Executable

OS Loader
Front and Back ends

- Separation of common tasks
- Makes design and implementation easier
- K compilers for N machines
  - N back ends, K front ends
  - Instead of K*N compilers
Passes

• A *pass* is when the compiler reads the source code (or intermediate files)
• The number of passes depends on the source and target language and the running environment
• Different phases that cooperate can be grouped to a single pass (not always possible)
• When grouping is not possible
  – **Backpatching**: leave empty information that is going to be filled by a later phase/pass
Compiler-construction Tools

- Parser generators
- Scanner generators
- Syntax-directed translation engines
- Automatic code generators
- Data-flow engines