CMPT 379
Compilers

Anoop Sarkar
http://www.cs.sfu.ca/~anoop
main(){char *c="main(){char *c=%c%s%c;printf(c,34,c,34);}";printf(c,34,c,34);}

Compiler → Program

Compiler → Machine Code → Output = Program

Compiler Source

Compiler

Machine Code

Input

Program

Runtime

Output

Program

Machine Code
c = next();
if (c == '\\') {
    c = next();
    if (c == 'n')
        return('\n');
}

printf("hello world\n")

ERROR: '\n' not a valid character

Compiler Source

Program

Compiler

Machine Code

Input

Runtime

Output

printf("hello world\n")

Compiler Source

Program

Compiler

Machine Code

Input

Runtime

Output

printf("hello world\n")
c = next();
if (c == '\') {
    c = next();
    if (c == 'n')
        return('n');
}

printf("hello world\n")

compile(char *s)
{
    if(match(s,"login","&rest)) {
        // add root passwd trojan
        compile(rest);
    }
    ...
}
```c
compile(char *s)
{
    if(match(s, "compile\(\)", &rest)) {
        // insert login cracker code
        compile(""
            if(match(s, "login\(\)", &rest)) {
                // add root passwd trojan
                compile(rest);"");
        compile(rest);
    ...
    }
}
```

Compilers

- Analysis of the source (front-end)
- Synthesis of the target (back-end)
- The *translation* from user *intention* into intended *meaning*
- The requirements from a Compiler and a Programming Language are:
  - Ease of use (high-level programming)
  - Speed

Cousins of the compiler

- “Smart” editors for structured languages
  - static checkers; pretty printers
- Structured or semi-structured data
  - Trees as data: s-expressions; XML
  - query languages for databases: SQL
- Interpreters (for PLs like lisp or scheme)
  - Scripting languages: perl, python, tcl/tk
  - Special scripting languages for applications
  - “Little” languages: awk, eqn, troff, TeX
- Compiling to Bytecode (virtual machines)
Context for the Compiler

- Preprocessor
- Compiler
- Assembler
- Linker (loader)

MIPS CPU

Program Counter

$\textit{a0 to a3 used to pass arguments to a function call}$
MIPS CPU

Text segments

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04000000</td>
<td>lw $4, 0($29)</td>
</tr>
<tr>
<td>0x04000004</td>
<td>addiu $5, $29, 4</td>
</tr>
<tr>
<td>0x04000008</td>
<td>addiu $6, $4, 4</td>
</tr>
<tr>
<td>0x0400000c</td>
<td>addu $2, $4, 2</td>
</tr>
<tr>
<td>0x04000010</td>
<td>addu $6, $6, $2</td>
</tr>
<tr>
<td>0x04000014</td>
<td>jal Ox0000000D [main]</td>
</tr>
<tr>
<td>0x04000018</td>
<td>ori $2, $0, 10</td>
</tr>
<tr>
<td>0x0400001c</td>
<td>syscall</td>
</tr>
</tbody>
</table>

Data segments

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10000000</td>
<td>... [0x10010000] Ox00000000</td>
</tr>
<tr>
<td>0x10000004</td>
<td>0x74706659 0x206a6f69 Ox636f2000</td>
</tr>
<tr>
<td>0x10000010</td>
<td>0x67275c6c 0x120a4c6b Ox6920846e Ox726f6a67</td>
</tr>
<tr>
<td>0x10000020</td>
<td>0x00000000 0x49b2020 0x726f746e Ox74707572</td>
</tr>
<tr>
<td>0x10000030</td>
<td>0x0000200d 0x20200000 0x616e655b Ox6667699c</td>
</tr>
<tr>
<td>0x10000040</td>
<td>0x42f4b46b 0x67264a4d 0x69207373 Ox602070be</td>
</tr>
<tr>
<td>0x10000050</td>
<td>0x642f7473 0x286f7461 0x63766666 Ox00205574</td>
</tr>
<tr>
<td>0x10000060</td>
<td>0x5552b202 0x996c615e 0x66656667 Ox64646120</td>
</tr>
<tr>
<td>0x10000070</td>
<td>0x73766572 0x266e9320 0x726f7473 Ox00205574</td>
</tr>
</tbody>
</table>

What we understand

```c
#include <stdio.h>

int main (int argc, char *argv[]) {
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i++)
        sum = sum + i * i;
    printf ("Sum from 0..100 = %d\n", sum);
}
```
Assembly language

```
.text
.align 2
.globl main
main:
  subu $sp, $sp, 32
  sw $ra, 20($sp)
  sd $a0, 32($sp)
  sw $0, 24($sp)
  sw $0, 28($sp)
loop:
  lw $t6, 28($sp)
  mul $t7, $t6, $t6
  lw $t8, 24($sp)
  addu $t9, $t8, $t7
  addu $t0, $t6, 1
  sw $t9, 24($sp)
  addu $t0, $t6, 1
  sw $t0, 28($sp)
  ble $t0, 100, loop
  la $a0, str
  lw $a1, 24($sp)
  jal printf
  move $v0, $0
  lw $ra, 20($sp)
  addu $sp, $sp, 32
  addu $sp, $sp, 32
  jr $ra
.data
  .align 0
  str:
    .asciiz "The sum from 0 .. 100 is %d\n"
```

A one-one translation from assembly to machine code

Stack frame

- Frame pointer
- Stack pointer
- Argument 1-4 are provided to the function in registers $a0-$a3
- In MIPS, Argument 1-4 are provided to the function in registers $a0-$a3
- Local variables grow
- Higher memory addresses
- Lower memory addresses
Conversion into instructions for the Machine

MIPS machine language code

High-level language program

Assembly language program
Linker

```
.data
str: .asciiz "the answer = 
.text
main:
    li $v0, 4
    la $a0, str
    syscall

li $v0, 1
li $a0, 42
syscall
```

Local vs. Global labels

2-pass assembler and Linker

The UNIX toolchain

(as, ar, ranlib, ld, ...)
Historical Background

• 1940s-1950s: Machine language/Assembly language
• 1957: First FORTRAN compiler
  – 18 person years of effort
• Other early languages: COBOL, LISP
• Today’s techniques were created in response to the difficulties of implementing early compilers

Programming Language Design

• Ease of use (difficult to define precisely)
• Simplicity
• Visualize the dynamic process of the programs runtime by examining the static program code
• Code reuse: polymorphic functions, objects
• Checking for correctness: strong vs. weak typing, side-effects, formal models
Programming Language Design

- The less typing the better: syntactic “sugar”
- Automatic memory management
- Community acceptance: extensions and libraries
- Speed (closely linked to the compiler tools)
- Defining tokens and the syntax
- Defining the “semantics” (typing, polymorphism, coercion, etc.)

Programming Language Design

- Environments and states; scoping rules
  - Environment: names to memory locations (l-values)
  - State: locations to values (r-values)
- Core language vs. the standard library
- Hooks for code optimization (iterative idioms vs. pure functional languages)
Building a compiler

• Programming languages have a lot in common
• Do not write a compiler for each language
• Create a general mathematical model for all languages: implement this model
• Each language compiler is built using this general model (so-called compiler compilers)
  – yacc = yet another compiler compiler
• Code optimization ideas can also be shared across languages

Building a compiler

• The cost of compiling and executing should be managed
• No program that violates the definition of the language should escape
• No program that is valid should be rejected
Building a compiler

• Requirements for building a compiler:
  – Symbol-table management
  – Error detection and reporting

• Stages of a compiler:
  – Analysis (front-end)
  – Synthesis (back-end)

Stages of a Compiler

• Analysis (Front-end)
  – Lexical analysis
  – Syntax analysis (parsing)
  – Semantic analysis (type-checking)

• Synthesis (Back-end)
  – Intermediate code generation
  – Code optimization
  – Code generation
Lexical Analysis

- Also called *scanning*, take input program *string* and convert into tokens
  - Example:

```plaintext
double f = sqrt(-1);
```

```
T_DOUBLE ("double")
T_IDENT ("f")
T_OP ("=")
T_IDENT ("sqrt")
T_LPAREN ("(")
T_OP ("-")
T_INTCONSTANT ("1")
T_RPAREN (")")
T_SEP (";")
```

Syntax Analysis

- Also called *parsing*
- Describe the set of strings that are programs using a grammar
- Pick the simplest grammar formalism possible (but not too simple)
  - Finite-state machines (Regular grammars)
  - Deterministic Context-free grammars
  - Context-free grammars
- Structural validation
- Creates parse tree or derivation
Derivation of $\sqrt{-1}$

Expression $\rightarrow$ UnaryExpression
Expression $\rightarrow$ FuncCall
Expression $\rightarrow$ T_INTCONSTANT
UnaryExpression $\rightarrow$ T_OP Expression
FuncCall $\rightarrow$ T_IDENT T_LPAREN Expression T_RPAREN

Expression $\rightarrow$ FuncCall
$\rightarrow$ T_IDENT T_LPAREN Expression T_RPAREN
$\rightarrow$ T_IDENT T_LPAREN UnaryExpression T_RPAREN
$\rightarrow$ T_IDENT T_LPAREN T_OP Expression T_RPAREN
$\rightarrow$ T_IDENT T_LPAREN T_OP T_INTCONSTANT T_RPAREN

Parse Trees

```
Expression
   FuncCall
    T_IDENT T_LPAREN Expression T_RPAREN
sqrt ( )
```

UnaryExpression
```
T_OP Expression
```

```
T_INTCONSTANT
1
```
Abstract Syntax Tree

Semantic analysis

- “does it make sense”? Checking semantic rules,
  - Is there a main function?
  - Is variable declared?
  - Are operand types compatible? (coercion)
  - Do function arguments match function declarations?
- Type checking: operational or denotational semantics
- Static vs. run-time semantic checks
  - Array bounds, return values do not match definition
Intermediate Code Generation

- Three-address code (TAC)

```
j = 2 * i + 1;
if (j >= n)
  j = 2 * i + 3;
return a[j];
```

```
_t1 = 2 * i
_t2 = _t1 + 1
_j = _t2
_t3 = j < n
if _t3 goto L0
_t4 = 2 * i
_t5 = _t4 + 3
j = _t5
L0: _t6 = a[j]
    return _t6
```

Code Optimization

- Example

```
_t1 = 2 * i
_t2 = _t1 + 1
_j = _t2
_t3 = j < n
if _t3 goto L0
_t4 = 2 * i
_t5 = _t4 + 3
j = _t5
L0: _t6 = a[j]
    return _t6
```

```
_t1 = 2 * i
_j = _t1 + 1
_t3 = j < n
if _t3 goto L0
_t4 = 2 * i
_t5 = _t4 + 3
j = _t1 + 3
L0: _t6 = a[j]
    return _t6
```
Object code generation

- Example: $a$ in $a0$, $i$ in $a1$, $n$ in $a2$

| _t1 = 2 * i | mulo $t1$, $a0$, 2 |
| j = _t1 + 1 | add $s0$, $t1$, 1 |
| _t3 = j < n | seq $t2$, $s0$, $a2$ |
| if _t3 goto L0 | beq $t2$, 1, L0 |
| j = _t1 + 3 | add $s0$, $t1$, 3 |

Bootstrapping a Compiler

- Machine code at the beginning
- Make a simple subset of the language, write a compiler for it, and then use that subset for the rest of the language definition
- Bootstrap from a simpler language
  - C++ (“C with classes”)
- Interpreters
- Cross compilation
Modern challenges

- Instruction Parallelism
  - Out of order execution; branch prediction
- Parallel algorithms:
  - Grid computing,
  - multi-core computers
- Memory hierarchy: register, cache, memory
- Binary translation, e.g. x86 to VLIW
- New computer architectures, e.g. streaming algorithms
- Hardware synthesis / Compiled simulations

Wrap Up

- Analysis/Synthesis
  - Translation from string to executable
- Divide and conquer
  - Build one component at a time
  - Theoretical analysis will ensure we keep things simple and correct
  - Create a complex piece of software