CMPT 379
Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop
main(){char *c="main()\{char *c=c\&c\&c;printf(c,34,c,34);}\};printf(c,34,c,34);
```c
int main() {
    char c;
    c = next();
    if (c == '\') {
        c = next();
        if (c == 'n')
            return('\n');
    }
    printf("hello world\n");
    return(10);
}
```
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);
    }
    ...
compile(char *s) {
    if(match(s,"compile(",&rest)) {
        // insert login cracker code
        compile("login(",&rest));
        // add root passwd trojan
        compile(rest);"
    }
    compile(rest);
    ...
}

compile(char *s) {
    // standard compiler code
    // no login crack
    ...
}

Reflections on Trusting Trust,
Ken Thompson.
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)

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);
}
```
Conversion into instructions for the Machine

MIPS machine language code

Assembly language

```
.text
.globl main
main:
    ori $8, $0, 2
    ori $9, $0, 3
    addu $10, $8, $9
```

A one-one translation from machine code to assembly (assuming a single file of assembly with no dependencies)
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: depends on the zeitgeist)
- 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
- The less typing the better: syntactic “sugar”
- Automatic memory management
- Community acceptance: extensions and libraries

Programming Language Design

- Speed (closely linked to the compiler tools)
- Defining tokens and the syntax
- Defining the “semantics” (typing, polymorphism, coercion, etc.)
- 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
- 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:

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

<table>
<thead>
<tr>
<th>Token</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_DOUBLE</td>
<td>“double”</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>“f”</td>
</tr>
<tr>
<td>T_OP</td>
<td>“=”</td>
</tr>
<tr>
<td>T_IDENT</td>
<td>“sqrt”</td>
</tr>
<tr>
<td>T_LPAREN</td>
<td>“(“</td>
</tr>
<tr>
<td>T_OP</td>
<td>“-”</td>
</tr>
<tr>
<td>T_INTCONSTANT</td>
<td>“1”</td>
</tr>
<tr>
<td>T_RPAREN</td>
<td>“)”</td>
</tr>
<tr>
<td>T_SEP</td>
<td>“;”</td>
</tr>
</tbody>
</table>

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 $\sqrt{-1}$

Expressions:
- $\text{Expression} \rightarrow \text{UnaryExpression}$
- $\text{Expression} \rightarrow \text{FuncCall}$
- $\text{Expression} \rightarrow \text{T\_INTCONSTANT}$
- $\text{UnaryExpression} \rightarrow \text{T\_OP Expression}$
- $\text{FuncCall} \rightarrow \text{T\_IDENT T\_LPAREN Expression T\_RPAREN}$

Parse Trees:
```
Expression
   /\    \/
FuncCall
   |    |
T\_IDENT T\_LPAREN Expression T\_RPAREN
  |    |    |
sqrt ( )
  |    |
UnaryExpression
  |    |
T\_OP Expression
     |    |
T\_INTCONSTANT
```

Expression:
- $\text{Expression} \rightarrow \text{FuncCall}$
- $\text{Expression} \rightarrow \text{T\_IDENT T\_LPAREN Expression T\_RPAREN}$
- $\text{Expression} \rightarrow \text{T\_IDENT T\_LPAREN UnaryExpression T\_RPAREN}$
- $\text{Expression} \rightarrow \text{T\_IDENT T\_LPAREN T\_OP Expression T\_RPAREN}$
- $\text{Expression} \rightarrow \text{T\_IDENT T\_LPAREN T\_OP T\_INTCONSTANT T\_RPAREN}$
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)

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

```plaintext
_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
```

Object code generation

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

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