Course Overview

Mooly Sagiv
msagiv@tau.ac.il
TA: Omer Tripp
omertripp@gmail.com

Textbook: Modern Compiler Design
Grune, Bal, Jacobs, Langendoen
CS0368-3133-01@listserv.tau.ac.il
Outline

• Course Requirements
• High Level Programming Languages
• Interpreters vs. Compilers
• Why study compilers (1.1)
• A simple traditional modern compiler/interpreter (1.2)
• Subjects Covered
• Summary
Course Requirements

• Compiler Project 50%
  – Translate Java Subset into X86
• Final exam 50% (must pass)
Lecture Goals

• Understand the basic structure of a compiler
• Compiler vs. Interpreter
• Techniques used in compilers
High Level Programming Languages

• Imperative
  – Algol, PL1, Fortran, Pascal, Ada, Modula, and C
  – Closely related to “von Neumann” Computers

• Object-oriented
  – Simula, Smalltalk, Modula3, C++, Java, C#, Python
  – Data abstraction and ‘evolutionary’ form of program development
    • Class An implementation of an abstract data type (data+code)
    • Objects Instances of a class
    • Fields Data (structure fields)
    • Methods Code (procedures/functions with overloading)
    • Inheritance Refining the functionality of a class with different fields and methods

• Functional
  – Lisp, Scheme, ML, Miranda, Hope, Haskel, OCaml, F#

• Functional/Imperative
  – Rubby

• Logic Programming
  – Prolog
Other Languages

• Hardware description languages
  – VHDL
  – The program describes Hardware components
  – The compiler generates hardware layouts

• Scripting languages
  – Shell, C-shell, REXX, Perl
  – Include primitives constructs from the current software environment

• Web/Internet
  – HTML, Telescript, JAVA, Javascript

• Graphics and Text processing
  TeX, LaTeX, postscript
  – The compiler generates page layouts

• Intermediate-languages
  – P-Code, Java bytecode, IDL, CLR
Interpreter

• A program which interprets instructions

• **Input**
  – A program
  – An input for the program

• **Output**
  – The required output
Example

```c
int x;
scanf("%d", &x);
x = x + 1;
printf("%d", x);
```

```
C interpreter
```
Compiler

- A program which compiles instructions

- **Input**
  - A program

- **Output**
  - An object program that reads the input and writes the output
Example

```c
int x;
scanf("%d", &x);
x = x + 1;
printf("%d", x);
```

Sparc-cc-compiler

```
add %fp,-8, %l1
mov %l1, %o1
call scanf
ld [%fp-8],%l0
add %l0,1,%l0
st %l0,[%fp-8]
ld [%fp-8], %l1
mov %l1, %o1
call printf
```

assembler/linker

object-program
Remarks

• Both compilers and interpreters are programs written in high level languages
• Requires additional step to compile the compiler/interpreter
• Compilers and interpreters share functionality
Bootstrapping a compiler

L1 Compiler

L2 Compiler source

L2 Compiler

Program source

Program

Input

Executable compiler

Executable program

Output

= Y

= X
Conceptual structure of a compiler

Source text → Frontend (analysis) → Semantic Representation → Backend (synthesis) → Executable code

Compiler
Conceptual structure of an interpreter

Source text

Frontend (analysis)

Semantic Representation

interpretation

Output

Input

X

Y
Interpreter vs. Compiler

- Conceptually simpler (the definition of the programming language)
- Easier to port
- Can provide more specific error report
- Normally faster
- [More secure]

- Can report errors before input is given
- More efficient
  - Compilation is done once for all the inputs --- many computations can be performed at compile-time
  - Sometimes even \[\text{compile-time} + \text{execution-time} < \text{interpretation-time}\]
Interpreters provide specific error report

- **Input-program**
  
  ```c
  scanf("%d", &y);
  if (y < 0)
      x = 5;
  ...
  if (y <= 0)
      z = x + 1;
  ```

- **Input data** y=0
Compilers can provide errors before actual input is given

- **Input-program**
  ```c
  scanf(“%”, &y);
  if (y < 0)
      x = 5;
  ...
  if (y <= 0)
  /* line 88 */
  z = x + 1;
  ```

- **Compiler-Output**
  “line 88: x may be used before set"
Compilers can provide errors before actual input is given

- **Input-program**

```c
int a[100], x, y;
scanf("%d", &y);
if (y < 0)
    /* line 4*/
    y = a;
```

- **Compiler-Output**

  “line 4: improper pointer/integer combination: op ="
Compilers are usually more efficient

```c
scanf("%d", &x);
y = 5;
z = 7;
x = x + y*z;
printf("%d", x);
```

Sparc-cc-compiler

```
add    %fp,-8, %l1
mov    %l1, %o1
call   scanf
mov    5, %l0
st     %l0,[%fp-12]
mov    7, %l0
st     %l0,[%fp-16]
ld     [%fp-8], %l0
ld     [%fp-8],%l0
add    %l0, 35 ,%l0
st     %l0,[%fp-8]
ld     [%fp-8], %l1
mov    %l1, %o1
call   printf
```
Why Study Compilers?

• Become a compiler writer
  – New programming languages
  – New machines
  – New compilation modes: “just-in-time”
• Using some of the techniques in other contexts
• Design a very big software program using a reasonable effort
• Learn applications of many CS results (formal languages, decidability, graph algorithms, dynamic programming, ...)
• Better understating of programming languages and machine architectures
• Become a better programmer
Why study compilers?

• Compiler construction is successful
  – Proper structure of the problem
  – Judicious use of formalisms

• Wider application
  – Many conversions can be viewed as compilation

• Useful algorithms
Proper Problem Structure

- Simplify the compilation phase
- Portability of the compiler frontend
- Reusability of the compiler backend
- Professional compilers are integrated
Judicious use of formalisms

- Regular expressions (lexical analysis)
- Context-free grammars (syntactic analysis)
- Attribute grammars (context analysis)
- Code generator generators (dynamic programming)
- But some nitty-gritty programming
Use of program-generating tools

• Parts of the compiler are automatically generated from specification

![Diagram showing the use of Jlex for generating regular expressions from an input program and generating tokens as output.](image)
Use of program-generating tools

- Parts of the compiler are automatically generated from specification

- Context free grammar
- Jcup
- Tokens
- parser
- Syntax tree
Use of program-generating tools

- Simpler compiler construction
- Less error prone
- More flexible
- Use of pre-canned tailored code
- Use of dirty program tricks
- Reuse of specification
Wide applicability

• Structured data can be expressed using context free grammars
  – HTML files
  – Postscript
  – Tex/dvi files
  – …
Generally useful algorithms

- Parser generators
- Garbage collection
- Dynamic programming
- Graph coloring
A simple traditional modular compiler/interpreter (1.2)

• Trivial programming language
• Stack machine
• Compiler/interpreter written in C
• Demonstrate the basic steps
The abstract syntax tree (AST)

- Intermediate program representation
- Defines a tree - Preserves program hierarchy
- Generated by the parser
- Keywords and punctuation symbols are not stored (Not relevant once the tree exists)
Syntax tree

```
expression
  \ /
 number  *
   \   /
      '5'
expression
  \ /
 identifier  \\
  \   \     \\
   '+'  identifier
     \     \\   \    \\   \  \  \\
      'a'     b'
```
Abstract Syntax tree

```
      "*"
     /    /
  "5"   "+
   |    /  /
  "a"  "b"
```
Annotated Abstract Syntax tree

```
5 * a + b
```

- `5`: Type: integer, Location: reg
- `*`: Type: real, Location: reg1
- `+`: Type: real, Location: reg2
- `a`: Type: real, Location: sp+8
- `b`: Type: real, Location: sp+24
Structure of a demo compiler/interpreter

Lexical analysis

Syntax analysis

Context analysis

Intermediate code (AST)

Code generation

Interpretation
Input language

• Fully parameterized expressions
• Arguments can be a single digit

expression → digit | ‘(‘ expression operator expression ‘)’
operator → ‘+’ | ‘*’
digit → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
Driver for the demo compiler

```c
#include "parser.h" /* for type AST_node */
#include "backend.h" /* for Process() */
#include "error.h" /* for Error() */

int main(void) {
    AST_node *icode;

    if (!Parse_program(&icode)) Error("No top-level expression");
    Process(icode);

    return 0;
}
```
Lexical Analysis

• Partitions the inputs into tokens
  – DIGIT
  – EOF
  – ‘*’
  – ‘+’
  – ‘(‘
  – ‘)’
• Each token has its representation
• Ignores whitespaces
Header file lex.h for lexical analysis

/* Define class constants */

/* Values 0-255 are reserved for ASCII characters */

#define EoF 256
#define DIGIT 257

typedef struct {int class; char repr;} Token_type;
extern Token_type Token;
extern void get_next_token(void);
#include "lex.h"
static int Layout_char(int ch) {
    switch (ch) {
    case ' ': case '\t': case '\n': return 1;
    default:                        return 0;
    }
}
token_type Token;
void get_next_token(void) {
    int ch;
    do {
        ch = getchar();
        if (ch < 0) {
            Token.class = EoF; Token.repr = '#';
            return;
        }
    } while (Layout_char(ch));
    if ('0' <= ch && ch <= '9') {Token.class = DIGIT;}
    else {Token.class = ch;}
    Token.repr = ch;
}
Parser

- Invokes lexical analyzer
- Reports syntax errors
- Constructs AST
Parser Environment

```c
#include "lex.h"
#include "error.h"
#include "parser.h"
static Expression *new_expression(void) {
    return (Expression *)malloc(sizeof (Expression));
}
static void free_expression(Expression *expr) {free((void *)expr);}
static int Parse_operator(Operator *oper_p);
static int Parse_expression(Expression **expr_p);
int Parse_program(AST_node **icode_p) {
    Expression *expr;
    get_next_token();    /* start the lexical analyzer */
    if (Parse_expression(&expr)) {
        if (Token.class != EoF) {
            Error("Garbage after end of program");
        }
        *icode_p = expr;
        return 1;
    }
    return 0;
}
```
typedef int Operator;

typedef struct _expression {
    char type;          /* 'D' or 'P' */
    int value;          /* for 'D' */
    struct _expression *left, *right; /* for 'P' */
    Operator oper;      /* for 'P' */
} Expression;

typedef Expression AST_node; /* the top node is an Expression */

extern int Parse_program(AST_node **);
AST for \((2 \times ((3 \times 4) + 9))\)
Top-Down Parsing

- Optimistically build the tree from the root to leaves
- Try every alternative production
  - For $P \rightarrow A_1 A_2 \ldots A_n \mid B_1 B_2 \ldots B_m$
  - If $A_1$ succeeds
    - If $A_2$ succeeds
      - if $A_3$ succeeds
        - ... 
      - Otherwise fail
  - Otherwise fail
  - If $B_1$ succeeds
    - If $B_2$ succeeds
      - ... 
    - No backtracking
- Recursive descent parsing
- Can be applied for certain grammars
static int Parse_operator(Operator *oper) {
    if (Token.class == '+') {
        *oper = '+'; get_next_token(); return 1;
    }
    if (Token.class == '*') {
        *oper = '*'; get_next_token(); return 1;
    }
    return 0;
}
static int Parse_expression(Expression **expr_p) {
    Expression *expr = *expr_p = new_expression();
    if (Token.class == DIGIT) {
        expr->type = 'D'; expr->value = Token.repr - '0';
        get_next_token(); return 1;
    }
    if (Token.class == '(') {
        expr->type = 'P'; get_next_token();
        if (!Parse_expression(&expr->left)) { Error("Missing expression"); }
        if (!Parse_operator(&expr->oper)) { Error("Missing operator"); }
        if (!Parse_expression(&expr->right)) { Error("Missing expression"); }
        if (Token.class != ')') { Error("Missing )"); }
        get_next_token(); return 1;
    }
    /* failed on both attempts */
    free_expression(expr); return 0;
}
AST for \((2 \times ((3 \times 4) + 9)))\)
Context handling

• Trivial in our case
• No identifiers
• A single type for all expressions
Code generation

• Stack based machine
• Four instructions
  – PUSH n
  – ADD
  – MULT
  – PRINT
Code generation

#include "parser.h"
#include "backend.h"

static void Code_gen_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        printf("PUSH %d\n", expr->value);
        break;
    case 'P':
        Code_gen_expression(expr->left);
        Code_gen_expression(expr->right);
        switch (expr->oper) {
        case '+': printf("ADD\n"); break;
        case '*': printf("MULT\n"); break;
        }
        break;
    }
}

void Process(AST_node *icode) {
    Code_gen_expression(icode); printf("PRINT\n");
}
Compiling \((2*(((3*4)+9))\)
Generated Code Execution

Stack

PUSH 2
PUSH 3
PUSH 4
MULT
PUSH 9
ADD
MULT
PRINT

Stack

2
Generated Code Execution

PUSH 2
PUSH 3
PUSH 4
MULT
PUSH 9
ADD
MULT
PRINT

Stack | Stack
---|---
2 | 3
3 | 2
54 |
Generated Code Execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>PUSH 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

55
### Generated Code Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Generated Code Execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>12</td>
<td>9</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>PUSH 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generated Code Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 9</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Generated Code Execution

<table>
<thead>
<tr>
<th>Operation</th>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>21</td>
<td>42</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PUSH 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Generated Code Execution

<table>
<thead>
<tr>
<th>Stack</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td></td>
</tr>
<tr>
<td>PUSH 4</td>
<td>42</td>
</tr>
<tr>
<td>MULT</td>
<td></td>
</tr>
<tr>
<td>PUSH 9</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td></td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
</tr>
</tbody>
</table>
Interpretation

• Bottom-up evaluation of expressions
• The same interface of the compiler
#include "parser.h"
#include "backend.h"

static int Interpret_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        return expr->value;
        break;
    case 'P': {
        int e_left = Interpret_expression(expr->left);
        int e_right = Interpret_expression(expr->right);
        switch (expr->oper) {
        case '+': return e_left + e_right;
        case '*': return e_left * e_right;
        }
        break;
    }
}

void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
Interpreting \( (2*(((3*4)+9))) \)
A More Realistic Compiler

Program text
- input
- characters
- tokens
- Lexical Analysis
- Syntax Analysis
- AST
- Context Handling
- Annotated AST
- Intermediate code
- generation
- IC

Intermediate code
- IC optimization
- IC
- Code generation
- symbolic instructions
- Target code
- optimization
- Machine code
generation
- bit patterns
- Executable code
generation

file

file
Required systems

- Responsible for language dependent dynamic resource allocation
- Memory allocation
  - Stack frames
  - Heap
- Garbage collection
- I/O
- Interacts with operating system/architecture
- Important part of the compiler
Shortcuts

• Avoid generating machine code
• Use local assembler
• Generate C code
Tentative Syllabus

• Overview (1)
• Lexical Analysis (1)
  – Regular expressions to Finite State Automaton
• Parsing (3 lectures)
  – Grammars, Ambiguity, Efficient Parsers: Top-Down and Bottom-UP
• Semantic analysis (1)
  – Type checking
• Operational Semantics
• Code generation (4)
• Assembler/Linker Loader (1)
• Object Oriented (1)
• Garbage Collection (1)
Summary

• Phases drastically simplifies the problem of writing a good compiler
• The frontend is shared between compiler/interpreter