Course Overview
Mooly Sagiv
msagiv@tau.ac.il
TA: Roman Manevich
rumster@tau.ac.il
http://www.cs.tau.ac.il/~msagiv/courses/wcc06.html
TA: http://www.cs.tau.ac.il/~rumster/wcc06/
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 45% (must pass)
• Theoretical Exercise 5%
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#
  – Data abstraction and “evolutionary” form of program development
  • Class: An implementation of an abstract data type (data+code)
  • Object: Instance of a class
  • Field: Data (structure fields)
  • Method: Code (procedures/functions with overloading)
  • Inheritance: Refining the functionality of a class with different fields and methods
• Functional
  – Lisp, Scheme, ML, Miranda, Hope, Haskel
• Logic Programming
  – Prolog

Other Languages

• Hardware description languages
  – VHDL
  – The program describes Hardware components
  – The compiler generates hardware layouts
• Shell-languages Shell, C-shell, REXX
  – Include primitives constructs from the current software environment
• Graphics and Text processing
  – TeX, LaTeX, postscript
  – The compiler generates page layouts
• Web/Internet
  – HTML, MAWL, Telescript, JAVA
• Intermediate-languages
  – P-Code, Java bytecode, IDL, CLR
Interpreter

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

• **Output**
  - The required output

```
source-program

program’s input --> interpreter --> program’s output
```

Example

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

```
5 --> C interpreter --> 6
```

Compiler

• **Input**
  - A program

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

```
source-program

program’s input --> compiler --> object-program --> program’s output
```
Example

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

Sparc-cc-compiler

add %fp-8, %fp-8, %l1
mov %fp-8, %fp-8
call scanf
li [%fp-8], %fp-8
add %fp-8, %fp-8
st %fp-8, [%fp-8]
li [%fp-8], %fp-8
mov %fp-8, %fp-8
call printf

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
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

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 compilation + execution-time < interpretation-time
Interpreters provide specific error report

• Input-program
  
  ```
  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
  
  ```
  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
  
  ```
  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

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

Compiler vs. Interpreter

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>preprocessing</td>
<td>processing</td>
<td>preprocessing</td>
<td>processing</td>
</tr>
</tbody>
</table>

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](image)

- Jlex
- regular expressions
- input program
- scanner
- tokens

- 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
  - ...

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
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

Abstract Syntax tree

Annotated Abstract Syntax tree
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

```
#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

```c
/* 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 '	': case '
': 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;
        }
        switch (ch) {
        case ' ': case '	': case '
': return 1;
        case '0' : case '9': return 2;
        default: return 3;
        }
        switch (ch) {
        case '0' : case '9': return 2;
        default: return 3;
        }
        if (ch < '0' || ch > '9') {
            Token.class = DIGIT;
            Token.repr = ch;
            return;
        }
    } while (Layout_char(ch));
    if (ch < '0' || ch > '9') {
        Token.class = DIGIT;
        Token.repr = ch;
        return;
    }
```
Parser

• Invokes lexical analyzer
• Reports syntax errors
• Constructs AST

Parser Environment

```
#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;
}
```

Parser Header File

```
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 \((2 \times ((3 \times 4) + 9))\)

Parse_Operator

```c
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;
}
```

Parsing Expressions

- Try every alternative production
  - For \(F \rightarrow A_1 A_2 \ldots A_n | B_1 B_2 \ldots B_m\)
    - If \(A_1\) succeeds
      - If \(A_2\) succeeds
        - If \(A_3\) succeeds
        - ... , No backtracking
    - If \(B_1\) succeeds
      - If \(B_2\) succeeds
- Recursive descent parsing
- Can be applied for certain grammars
- Generalization: LL1 parsing
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 * ((3*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

```c
#include    "parser.h"
#include    "backend.h"
static void Code_gen_expression(Expression *expr) {
    switch (expr->type) {
    case 'D':
        printf("PUSH %d
", expr->value);
        break;
    case 'P':
        Code_gen_expression(expr->left);
        Code_gen_expression(expr->right);
        switch (expr->oper) {
        case '+': printf("ADD
"); break;
        case '*': printf("MULT
"); break;
        }
        break;
    }
}

void Process(AST_node *icode) {
    Code_gen_expression(icode); printf("PRINT\n");
}
```

Compiling \(2((3*4)+9)\)

```
PUSH 2
PUSH 3
PUSH 4
MULT
PUSH 9
ADD
MULT
PRINT
```
### Generated Code Execution

<table>
<thead>
<tr>
<th>Command</th>
<th>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Stack 1</th>
<th>Stack 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>PUSH 3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>ADD</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>PRINT</td>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
### Generated Code Execution

<table>
<thead>
<tr>
<th>Command</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>2</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>2</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>Command</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>2</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>2</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>Command</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>2</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>MULT</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 9</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>2</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>2</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 3</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>PUSH 4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>MULT</td>
<td>42</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>

### Interpretation
- Bottom-up evaluation of expressions
- The same interface of the compiler
```c
#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;
        case '=': return e_right;
        }
        break;
    }
    default: ProcessAST_node(*expr); {
        printf("%d\n", Interpret_expression(icode));
    }
}

void Process(AST_node *icode) {
    printf("%d\n", Interpret_expression(icode));
}
```

---

**Interpreting \((2* ((3*4)+9))\)**

![Diagram of the expression \((2* ((3*4)+9))\)](image)

**A More Realistic Compiler**

![Diagram of the compiler process](image)
Runtime 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 (2)
- Parsing (2 lectures)
- Semantic analysis (1)
- Code generation (4-5)
- 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