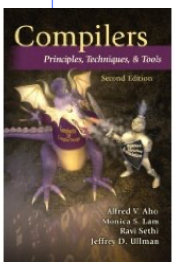
編譯器設計

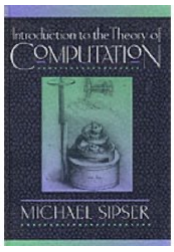
黃元欣
shin@csie.ntust.edu.tw

Syllabus

- Text Books



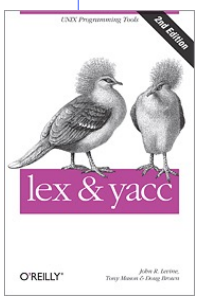
- *Compilers: Principles, Techniques, and Tools 2e*
  A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman
  Addison-Wesley 2007
  ISBN 0-321-48681-1



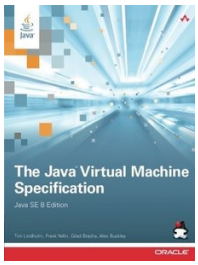
- *Introduction to the Theory of Computation*
  Michael Sipser
  PWS Publishing 1997
  ISBN 0-534-94728-X

Syllabus

- Reference Books



- *lex & yacc*
  J.R. Levine, T. Mason, and D. Brown
  O'Reilly 1995
  ISBN 1-56592-000-7



- *The Java™ Virtual Machine Specification, Java SE 8 Edition*
  Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley
  https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf

Syllabus

- Course Outline
  - Introduction
  - Lexical Analysis (Chap. 3)
  - Syntax Analysis (Chap. 4)
  - Syntax-Directed Translation (Chap. 5)
  - Run-time Organization (Chap.7)
  - Code Generation (Chap. 8)

Syllabus

- Grading
  - Programming Assignments 30%
  - Midterm 30%
  - Final 40%
- Office Hours
  - T8, W9 (T4-512, Tel: 6746)

Account Registration

You need to get an account at the class home page in order to submit programming assignments and download lecture slides

https://faculty.csie.ntust.edu.tw/~shin/compilers.html

Account Registration

Grades

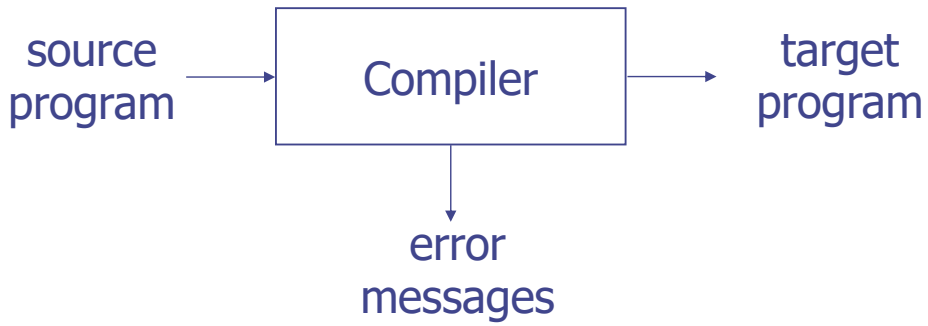
You can get the scores of all your examinations and programming assignments at the class home page:

https://faculty.csie.ntust.edu.tw/~shin/compilers.html

編譯器設計

Chapter 1: Introduction

Compilers



- A compiler is a program that
  - reads a program written in one language (the *source* language) and
  - translates it into an equivalent program in another language (the *target* language)

Compilers

- Examples
  - Compilers of C/C++, Fortran, Java, etc
  - Text formatters, e.g. TeX, LaTeX → $x^2 + y^2 = z^2$
  - Silicon compilers VHDL, Verilog ⇒ x/sup 2 + y/sup 2 = z/sup 2
  - Query interpreters, e.g SQL compilers
  - Preprocessors → #define max(x,y) (x>y)?x:y
  - Assemblers
  - Browsers HTML
  - Parallelizing compilers

Assembly Language to Machine Code

Tex

Hardware Description Language

Data base

a = max(b,c) → cpp (preprocessor) → gcc

replace

Programing Language ⇔ Equation / Circuit / DB Command / ⋮

Parallel Processing

Old Days: C / Fortran written by human and usually sequential → Compiler → Parallel C / Fortran

Nowadays: Using CUDA / OpenCL to build parallel processing program

Compilers and Assemblers

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

C compiler

Assembly language program (for MIPS)

```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

Assembler

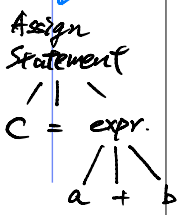
Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

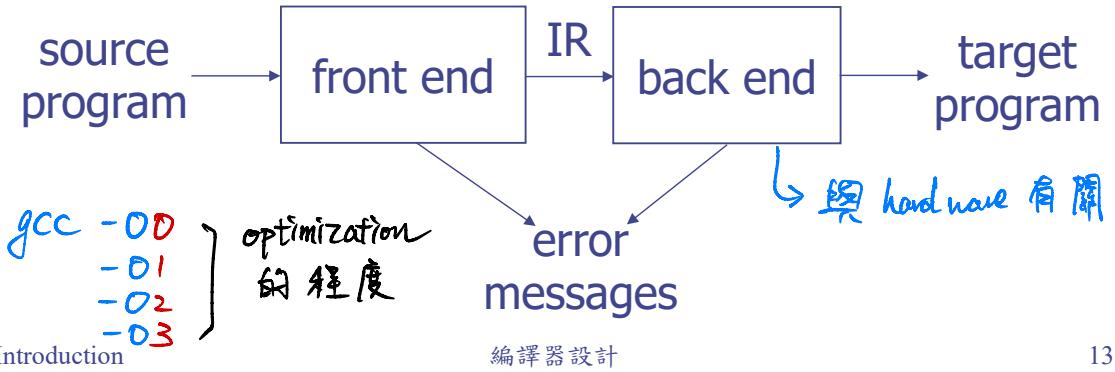
Analysis-Synthesis Model

Analysis first then Synthesis

We call it "Parse Tree"



- There are two parts to compilation [C = (a + b) expression]
  - Analysis (front end)
    - Breaks up the source program into constituent pieces 成份
    - Creates an intermediate representation (IR) 簡單的式子 or tree
  - Synthesis (back end)
    - Constructs the desired target program from the IR IR to Target Program
    - (Optionally) performs optimizations 優化



gcc -O0 -O1 -O2 -O3 } optimization 的程度

Analysis-Synthesis Model

- Some tools that perform analysis
  - Structure editors
    - Takes a sequence of commands as input to build a source program, e.g. with the user types while, the editor supplies the matching do
  - Pretty printers free-format fixed format → only used in old days
    - Analyzes a program and prints it in such a way that the structure of the program becomes clearly visible
  - Static checkers don't run program and try to catch the bugs
    - Reads and analyzes a program, and attempts to discover potential bugs without running the program
  - Interpreters
    - Instead of producing a target program as a translation, an interpreter performs the operations implied by the program

Why interpreter is slower than compiler?

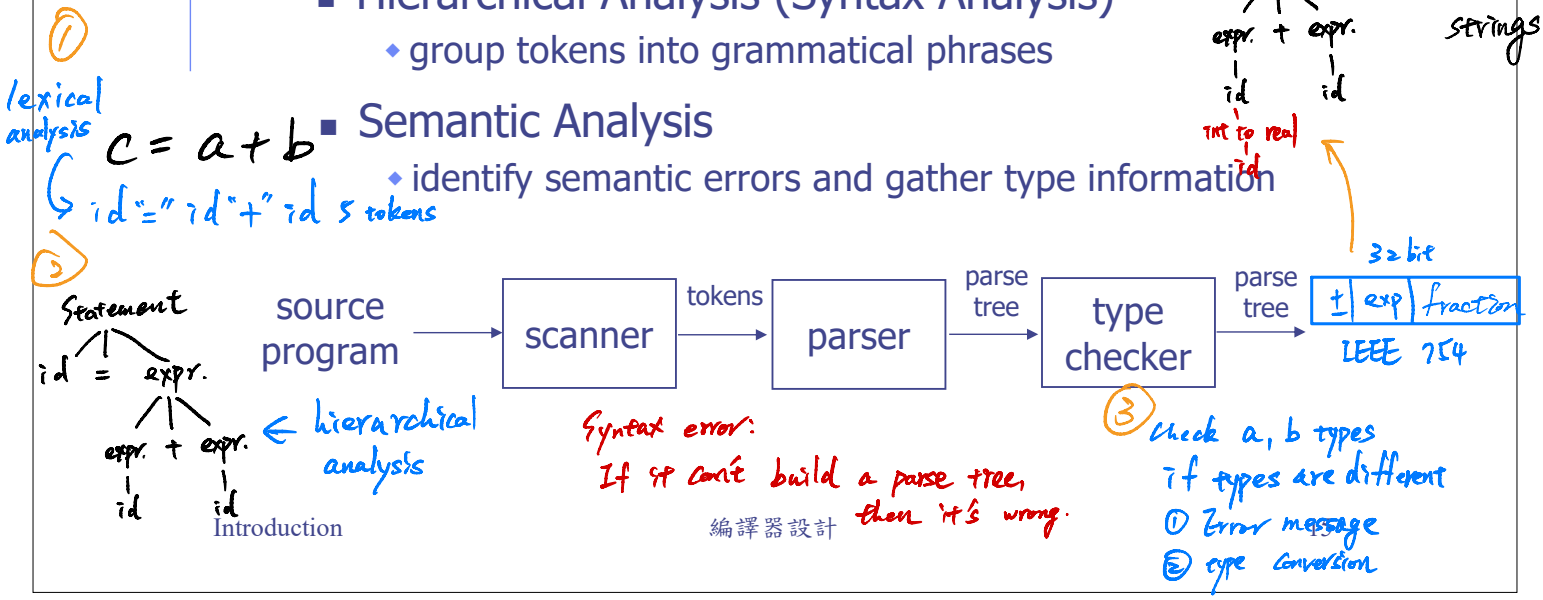
Compiler加法"+"轉成assembly language(add)，再轉成machine code的這個過程，只需要一個過程。但是interpreter將加法轉成machine code則需要很多machine code去模擬加法的過程，因此更花時間。

S. program → Compiler → T. program → PC → output

Java → S. program → Interpreter → output → slower

Analysis of the Source Program

- Analysis (front end) consists of 3 phases:
  - Linear Analysis (Lexical Analysis)
    - scan characters and group them into tokens
  - Hierarchical Analysis (Syntax Analysis)
    - group tokens into grammatical phrases
  - Semantic Analysis
    - identify semantic errors and gather type information



Lexical Analysis

- Mapping characters into tokens
  - Tokens: the basic unit of syntax
  - position = initial + rate * 60

    becomes

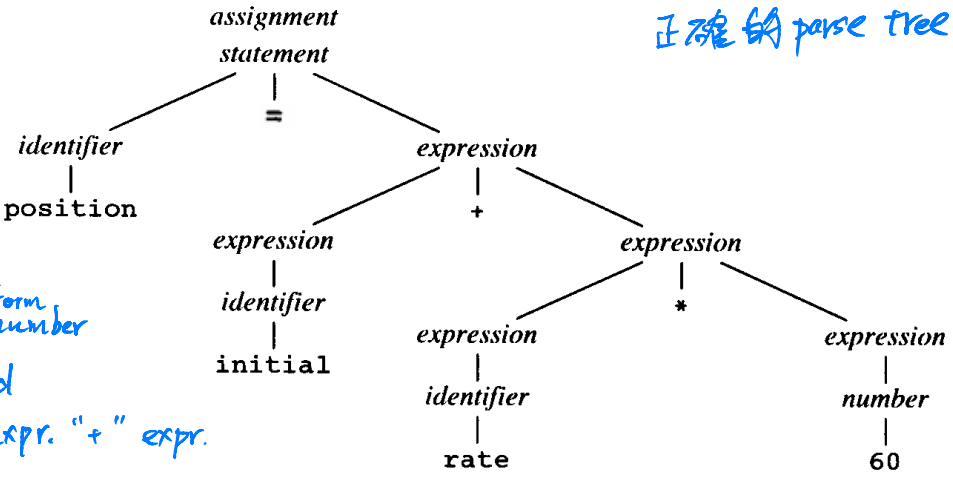
    ⟨id, position⟩ = ⟨id, initial⟩ + ⟨id, rate⟩ * ⟨num, 60⟩

    token attribute assign

Syntax Analysis

- Tokens are grouped into grammatical phrases that are used to synthesize output

看未端就知道是不是正確的 parse tree



It's called BNF

Format Form

expr. → number

expr. → id

expr. → expr. "+" expr.

Syntax Analysis

- The hierarchical structure of a program is usually expressed by recursive rules, e.g.
  - Any *identifier* is an expression
  - Any *number* is an expression
  - If $expression_1$ and $expression_2$ are expressions, so are

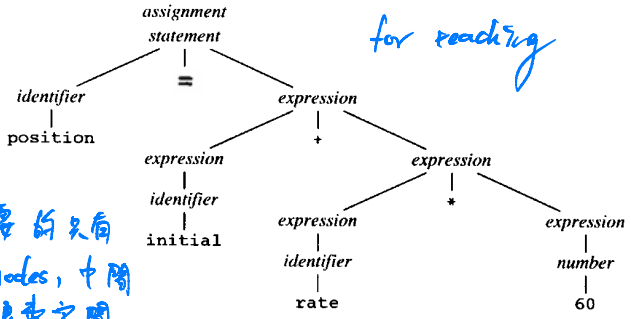
    $expression_1$ op $expression_2$

    ($expression_1$)

expr. → (expr.)

Parse Tree vs. Syntax Tree

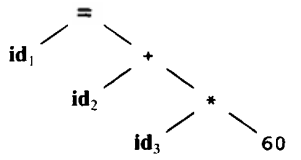
- Parse tree
  - describes the syntactic structure of the source program



for reaching

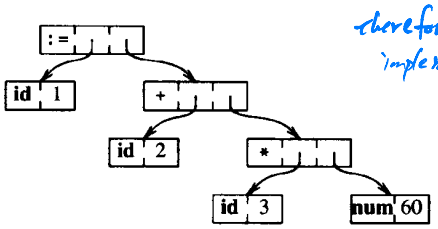
真正需要的只有末端的 nodes，中間的只會浪費空間

- Syntax tree
  - A more common internal representation of this syntactic structure
  - A compressed representation of the parse tree



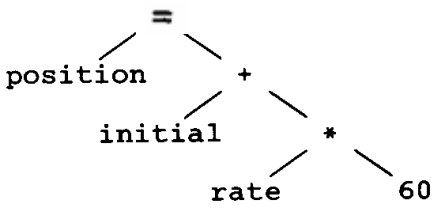
only use token to build the tree.

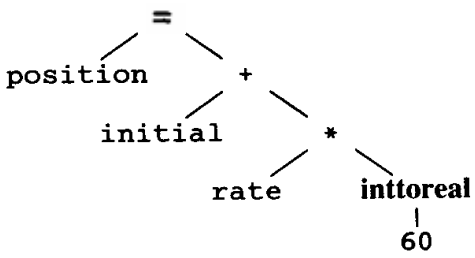
It saves memory space. therefore, it's usually implemented in real code



Semantic Analysis

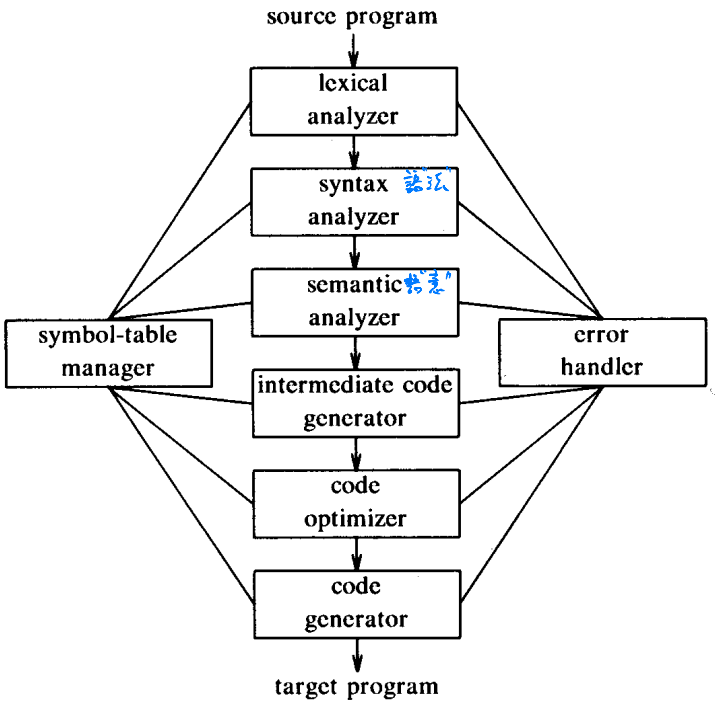
- Checks for semantic errors
- Gathers type information for the subsequent code-generation phase





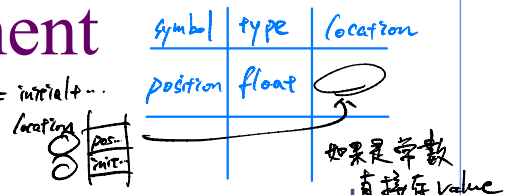
Phases of a Compiler

- A compiler operates in phases, each of which transforms the source program from one representation to another



Symbol-Table Management

在 lexical analysis 時放入



- Essential function of a compiler
  - To record the identifiers used in the source program and collect information about various attributes of each identifier 除了保留字(keywords, operators...) 放 compiler 不懂的字
    - e.g. allocated storage, type, scope, etc.
- Symbol table
  - A data structure containing a record for each identifier, with fields for the attributes
  - When an identifier is detected by the lexical analysis, it is entered into the symbol table
  - The attributes are determined during syntax analysis and semantic analysis
  - e.g. `float position, initial, rate;`

當 compiler 發現錯誤的時候，會試著 recover 並繼續往下走，才能再找到後面的 error

記錄 syntax error

Error Detection and Reporting

- Each phase can encounter errors
  - After detecting an error, a phase must deal with the error, so that compilation can proceed
    - allowing further errors to be detected
  - Lexical phase can detect errors where characters remaining in the input do not form any token
  - Syntax analysis phase detects errors where the token stream violates the syntax of the language
  - Semantic analysis phase tries to detect constructs that have the right syntactic structures but no meaning to the operation involved
    - e.g. `a = b + c;` where `b` is an array and `c` an integer

type conversion → show warning 語法對，語意不對

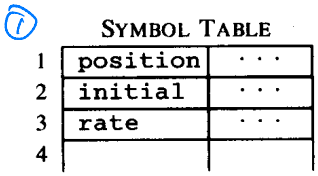
Lexical → forming token

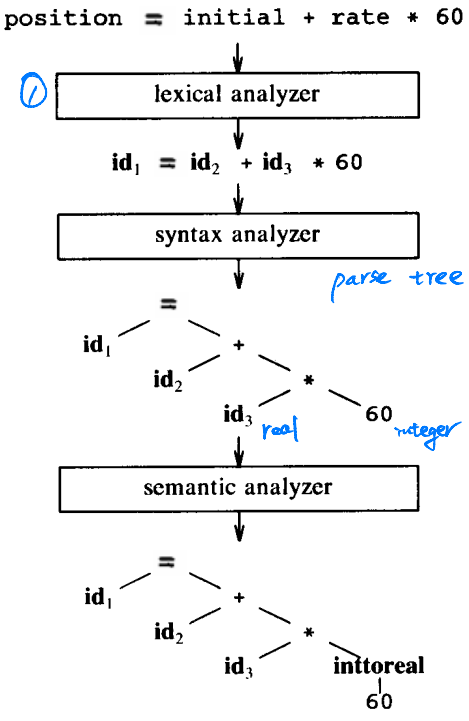
Syntax → token stream violates syntax

Semantic → no meaning to the operation

Analysis Phases

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis





parse tree → syntax tree (AST) Abstract Syntax Tree.

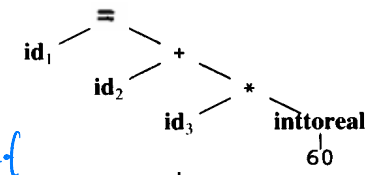
real integer

front End finish!

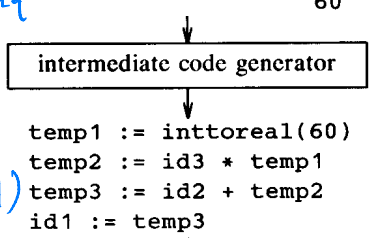
Intermediate Code Generation

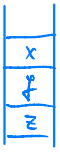
impact the efficiency of program hugely.

- Two properties
  - Easy to produce
  - Easy to translate into the target program
- Examples
  - Graph representations (GCC, LLVM)
  - Postfix notation
  - Three-address code



the back end
inter. code → tar. code





X = y op z
temp1 = inttoreal (60)
temp2 = id * temp1
temp3 = id + temp2
id = temp3

replace original node after simplifying
"temp" are restored in register,
they don't take up memory place.

也常用在也 Java (stack machine)
會很有效率
a = b+c×d
IR abcd×+=
push operand one by one

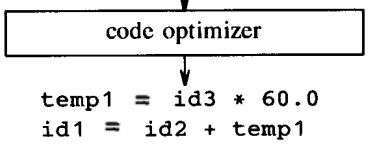
讀到 x → mul, pop c, d
+ → add, pop b
⋮

Code Optimization

- Attempts to improve the intermediate code
  - So that faster-running machine code will result

compiler 可自動轉換 不需再產生 code

temp1 = inttoreal(60) 60.0
temp2 = id3 * temp1 60.0
temp3 = id2 + temp2
id1 = temp3

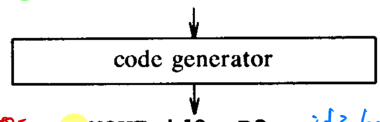


temp1 = id3 * 60.0
id1 = id2 + temp1

Code Generation

- Generates target code
  - Consisting of relocatable machine code or assembly code

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```




```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

(x86 相反 mov R1, id)

id3 load 到 CPU register

2-address (算完放 R2)

Assembly Language

2-address 做完放其中一個

3-address 做完放到第三個

Cousins of the Compiler

- Preprocessors
  - Preprocessors produce input to compilers
  - Macro processing cpp (preprocessor)
    - Allows users to define macros
  - File inclusion
    - Includes header files into the program text, e.g. `#include <stdio.h>`
  - "Rational" preprocessors
    - Augment older languages with more modern flow-of-control and data-structuring facilities
  - Language extensions
    - Add capabilities to languages by what amounts to built-in macro
      - e.g. HPF High-Performance Fortran 90's Parallel Computing

progma

HPF

當不需要的時候是 comment

需要的時候會被編入

Cousins of the Compiler

- Assemblers

```
mov a, R1          0001 01 00 00000000
add #2, R1         0011 01 10 00000010
mov R1, b          0010 01 00 00000100
```

- Loaders and Link-Editors

Compiler-Construction Tools

- Some general tools have been created for the automatic design of specific compiler components
  - Parser generators yacc
    - Producing syntax analyzers, normally from input that is based on a context-free grammar
  - Scanner generators 及 token
    - Automatically generating lexical analyzers, normally from a specification based on regular expressions
  - Automatic code generators
    - Taking a collection of rules that define the translation of each operation of the intermediate language into the machine language