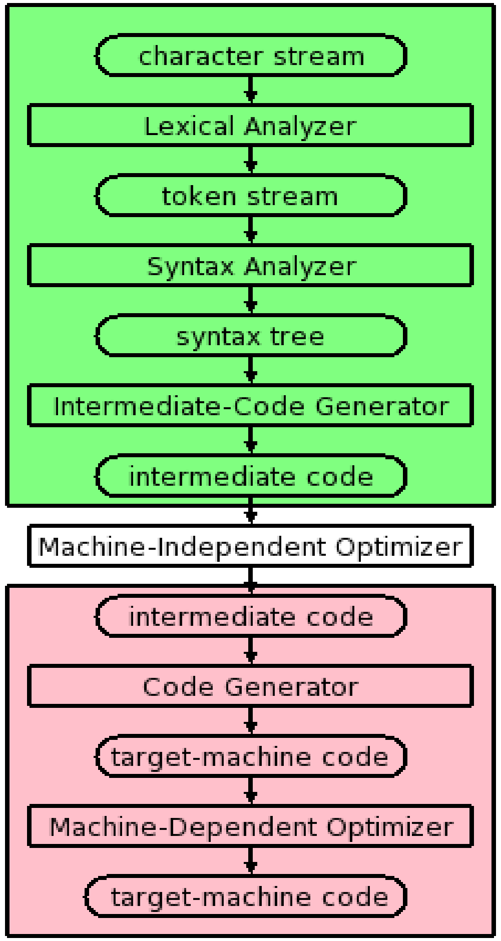
CS 420 - Compilers

Dr. Chen-Yeou (Charles) Yu

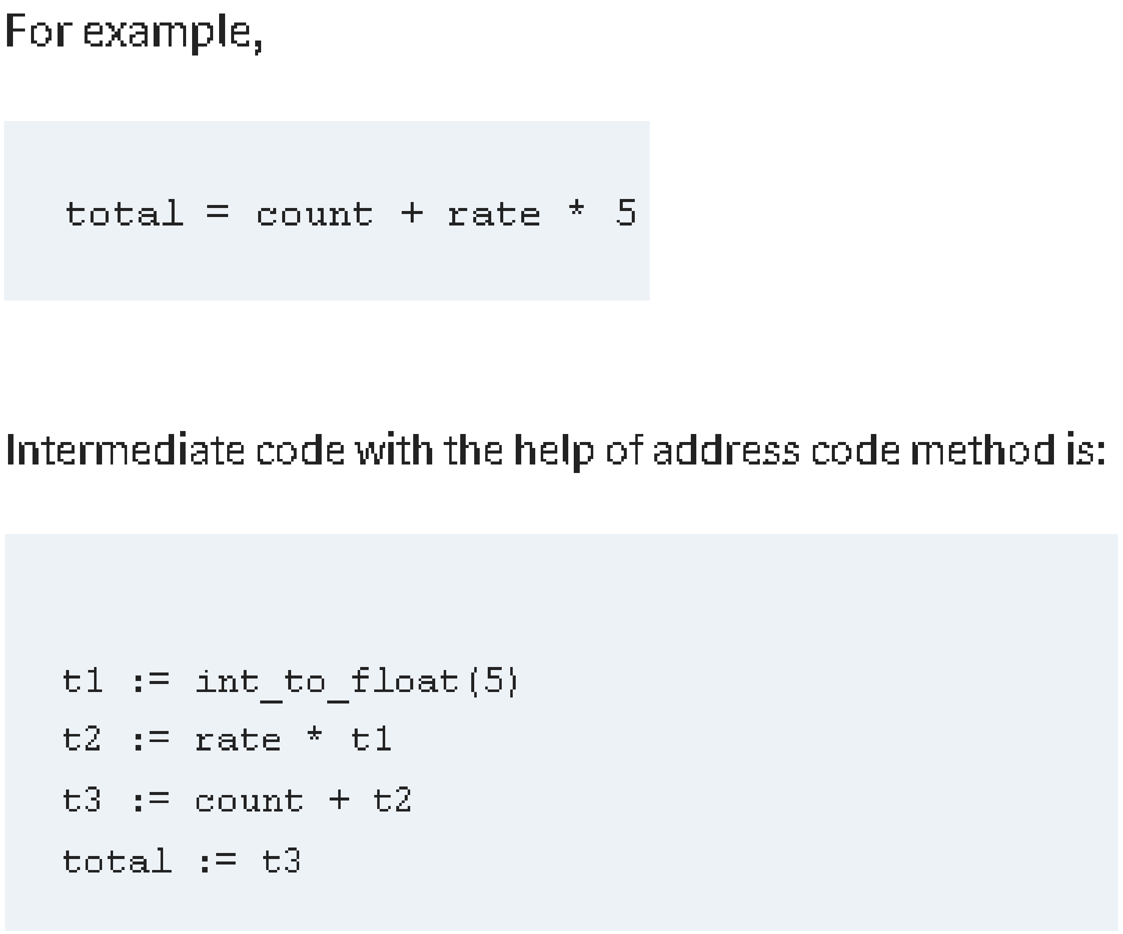
- The Structure of a Compiler (Some of them are in Part1)
    - Lexical Analysis (or Scanning) (in Part1)
    - Syntax analysis (parsing) (in Part1)
    - Semantic Analysis (in Part1)
    - Intermediate code generation (in this Part)
    - Code optimization (in this Part)
    - Code generation (in this Part)
    - Symbol-Table Management (in this Part)
    - Error Handling Routing (not specifically pointed out in the book)
- Marked in RED color are the 6 phases in compiler
- Summary



Intermediate code generation

- Intermediate code (IC) is between the high-level and machine level language
  - Sort of. It is on the bottom of the “front-end”
- Jobs has to be done in this stage (or phase)
  - IC should be generated from the semantic representation of the source program
  - Allows you to maintain precedence ordering of the source language
    - x = y + 5 * z
    - You don’t even need to put “(“ and “)”, in some advanced languages, they JUST know the execution order
  - It holds the correct number of operands of the instruction

Intermediate code generation



For example,

```
total = count + rate * 5
```

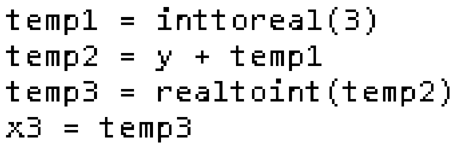
Intermediate code with the help of address code method is:

```
t1 := int_to_float(5)
t2 := rate * t1
t3 := count + t2
total := t3
```

Don’t you see there is there is a lot of recursion?

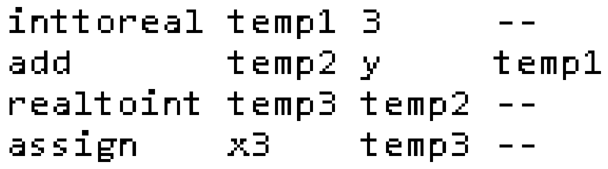
Intermediate code generation

- Another style is to translate the input source into “three-address code”.
- That means machine operations take (up to) three operands: two source and one target.
- In the following example, it is two source and one target for operands and one for operator
- For this source,

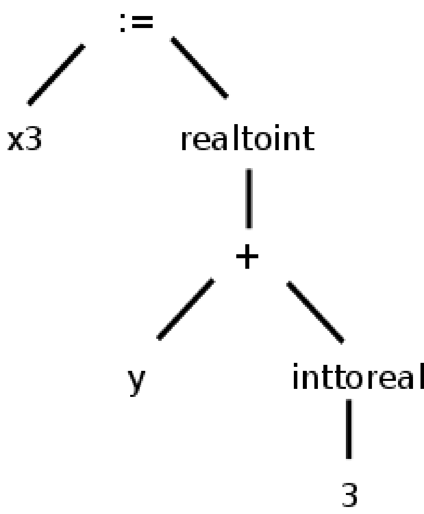


```
temp1 = inttoreal(3)
temp2 = y + temp1
temp3 = realtoint(temp2)
x3 = temp3
```

- Can be expressed in 3-address code as well
- Three-address code can include instructions with **fewer** than 3 operands..

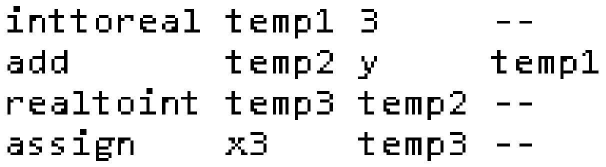


```
inttoreal temp1 3     --
add       temp2 y     temp1
realtoint temp3 temp2 --
assign    x3    temp3 --
```



Intermediate code generation

- Sometimes three-address code is called quadruples because one can view the previous code sequence as



```
inttoreal temp1 3     --
add       temp2 y     temp1
realtoint temp3 temp2 --
assign    x3    temp3 --
```

- Each quad has the form

operation target source1 source2

Code optimization

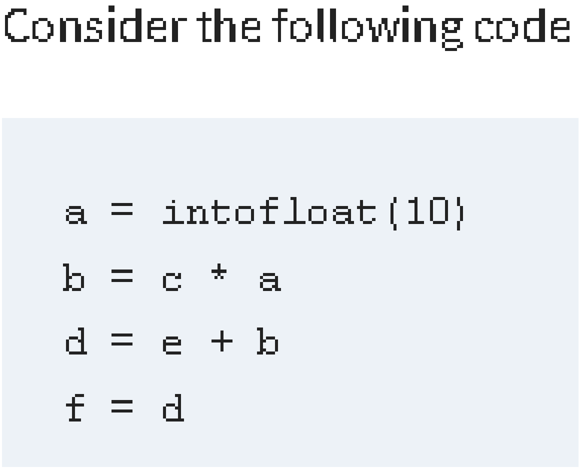
- Code optimization for Intermediate code (generated from previous stage).
- This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources.
- The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less space.

Code optimization

- Jobs has to be done in this stage (or phase)
  - It helps you to establish a trade-off between execution and compilation speed
  - Improves the running time of the target program
  - Generates streamlined code still in intermediate representation
  - Removing unreachable code and getting rid of unused variables
  - Removing statements which are not altered from the loop

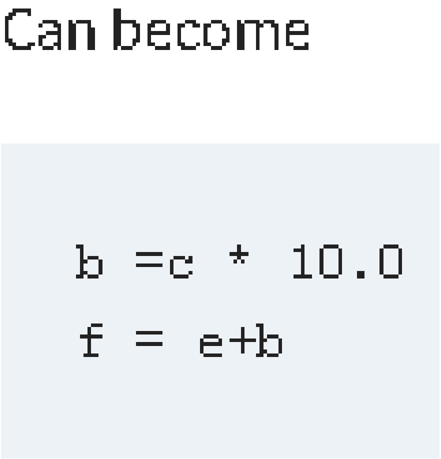
Code optimization

- An example:



Consider the following code

```
a = intofloat(10)
b = c * a
d = e + b
f = d
```



Can become

```
b =c * 10.0
f = e+b
```

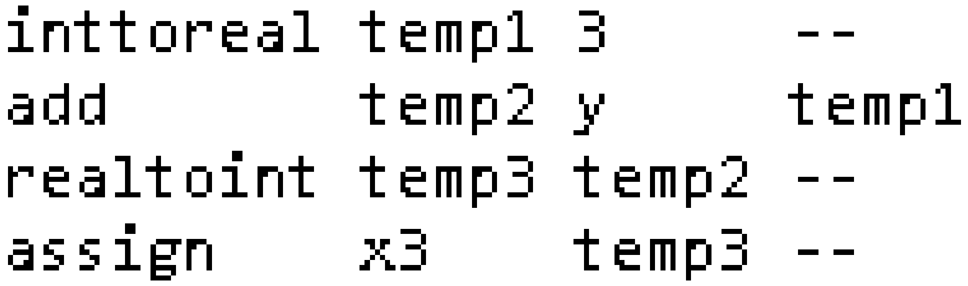
Code optimization

- Another example:
  - The first 2 lines can be combined as:

  add   temp2   y   3.0

  - The last 2 lines can be combined into

  realtoint   x3   temp2



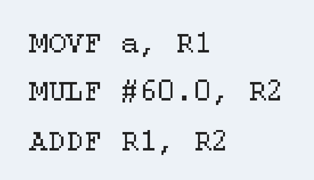
```
inttoreal temp1 3      --
add       temp2 y      temp1
realtoint temp3 temp2 --
assign    x3    temp3 --
```

Code generation

- Code generation is the last and final phase of a compiler.
- It gets inputs from code optimization phases
- The objective of this phase is to allocate storage and generate relocatable machine code.
- It also allocates memory locations for variables.
- The instructions in the intermediate code are converted into machine instruction.
- This phase converts the optimize or intermediate code into the target language.
- All the memory locations and registers are also selected and allocated during this phase.

Code generation

- An **Example:**
- a = b + 60.0
  Would be possibly translated to registers.
- This doesn’t look like intel x86 assembly



```
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2
```

Code generation

- Some processors (e.g., the MIPS architecture) use three-address instructions.
- Other processors permit only two addresses; the result overwrites one of the sources

Symbol-Table Management

- A symbol table contains a record for each identifier with fields for the attributes of the identifier.
- The symbol table stores information about program variables that will be used across phases.
  - This includes type information and storage locations.
- This component makes it easier for the compiler to search the identifier record and retrieve it quickly.
- The symbol table and error handler interact with all the phases and symbol table update correspondingly.
- The symbol table also helps you for the scope management

Symbol-Table Management

- A possible point of confusion:
  - The storage location does **not** give the location where the compiler has stored the variable.
  - Instead, it gives the location where the compiled program will store the variable.

Error Handling Routing

- In the compiler design process error may occur in all the below-given phases (for example):
  - **Lexical analyzer**: Wrongly spelled tokens
  - **Syntax analyzer**: Missing parenthesis
  - **Intermediate code generator**: Mismatched operands for an operator
  - **Code Optimizer**: When the statement is not **reachable**
  - **Code Generator**: When the memory is full or **proper registers** are not **allocated**
  - **Symbol tables**: Error of multiple declared identifiers

Error Handling Routing

- The error may be encountered in any of the above phases.
- Most common errors are invalid character sequence in scanning, invalid token sequences in type, scope error.
- After finding errors, the phase needs to deal with the errors to continue with the compilation process.

Summary

- Lexical Analysis is the first phase when compiler scans the source code
- Syntax analysis is all about discovering structure in text
- Semantic analysis checks the semantic consistency of the code (types)
- Once the semantic analysis phase is over the compiler, generate intermediate code (IR) for the target machine
- Code optimization phase removes unnecessary code line and arranges the sequence of statements
- Code generation phase gets inputs from code optimization phase and produces the page code or object code as a result