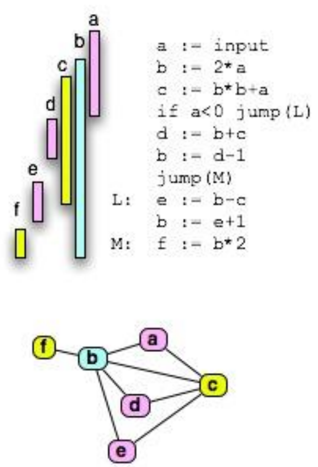
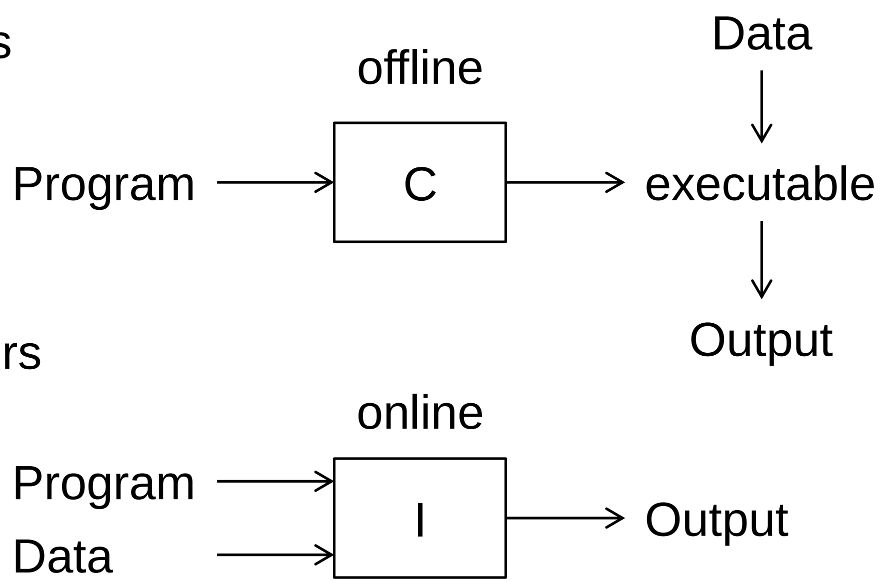
Introduction to Compilers



Introduction to Compilers

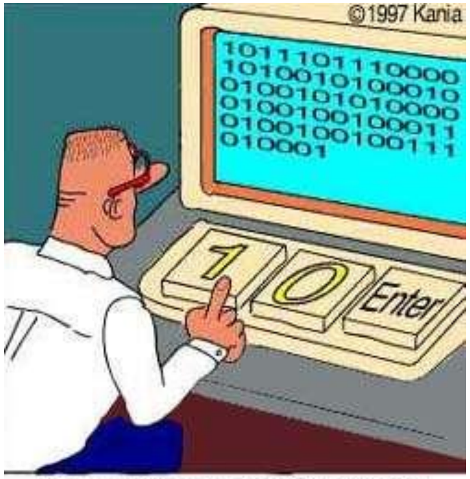
- **Compilers are language translators**
  - input: program in one language
  - output: equivalent program in another language

- **Two types**
  - Compilers
  - Interpreters



History of Compilers

- **From Machine Code…**



Real programmers code in binary.

- **…to Assembly Code…**
  - even for the earliest machines (UNIVAC, IBM 701/704), the cost of software exceeded the hardware cost

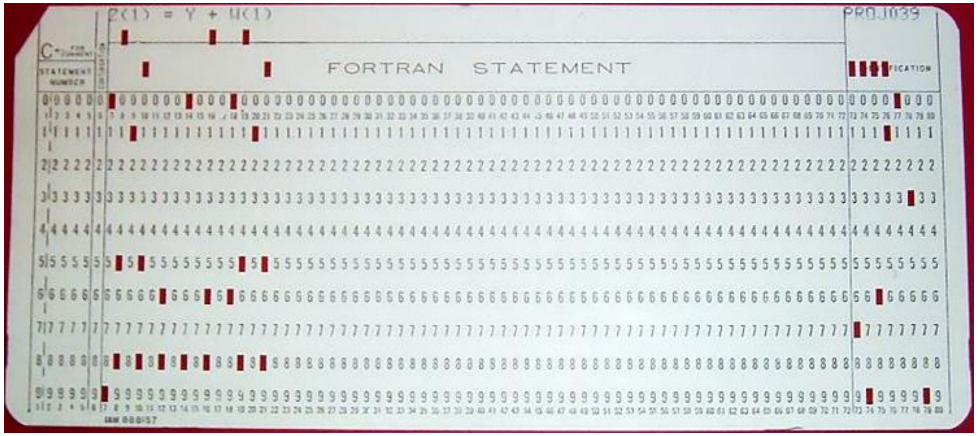
History of Compilers

- **…to High-Level Programming Languages**
  - Grace Hopper (1906 – 1992)
    - coined the term “compiler”
    - 1952: wrote the first ‘compiler’ A for the A-0 language
    - B-0, ARITH-MATIC, MATH-MATHIC, FLOW-MATIC, COBOL

  - John Backus (1924 – 2007)
    - 1953: Speedcoding
      interpreted: code ran 10-20x slower
      interpreter used up 310 memory words
    - 1955: FORTRAN I
      FORmula TRANslator
      translated high-level code to assembly
    - [Backus-Naur Form (BNF)]

FORTRAN I

- 1954-57: FORTRAN I project
  - compiler released in 1957
  - first successful high-level programming language

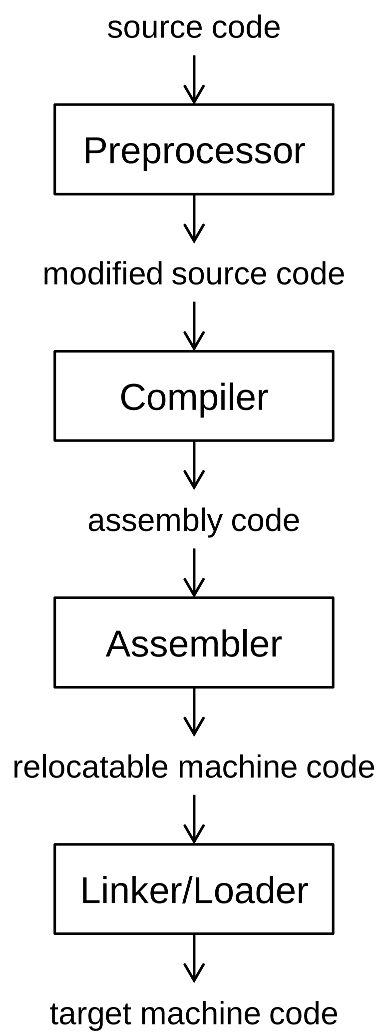


- 1958
  - more than 50% of all software is written in FORTRAN

- huge impact on computer science
  - led to an enormous body of theoretical work
  - even modern compilers preserve the outlines of the FORTRAN I compiler

The Structure of a Compiler

The Bigger Picture




```
#include <stdio.h>
#define N 1000

int main() {
  int A[N];

  a[i] = a[i] + 1;
  printf("%d", a[i]);
```

```
int printf(…);
…

int main() {
  int A[1000];

  a[i] = a[i] + 1;
  printf("%d", a[i]);
```

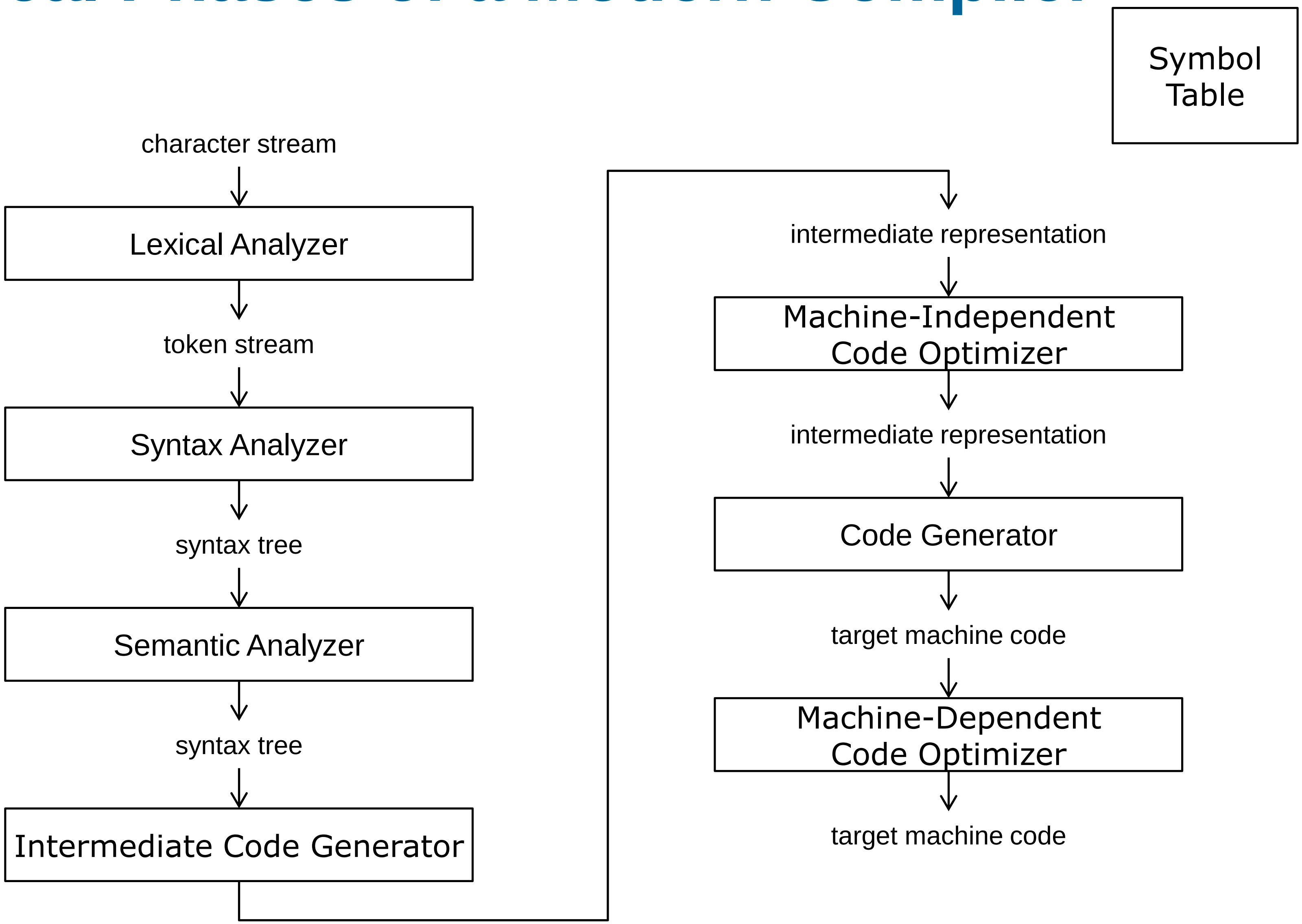
```
  movl $a, %eax
  addl (%eax, %ebx, 4), %ecx
  …
  call printf
```

```
 6: e8 fc ff ff ff   call 7<main+0x14>
                     7: R_386_PC32 printf
```

```
80483ba: e8 09 00 00 00   call 80483c8
```


CSE 컴퓨터공학부 Department of Computer Science & Engineering

Typical Phases of a Modern Compiler



Basic Structure of a Compiler

- Basic Structure of a Compiler
  1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Optimization
  5. Code Generation

acknowledgements: contains adapted material from Alex Aiken

Lexical Analysis

- **Lexical Analysis** or **Scanning**

  recognize the words in the input

This is an example sentence.

“This”, “is”, “an”, “example”, “sentence”, “.”

Lexical Analysis

- **Lexical Analysis** or **Scanning**

  recognize the words in the input

  Not as trivial as it looks. Consider:

  thisi sane xam plesent ence.

Lexical Analysis

- More formally:
  - input: character stream of source program
  - process:
    - split stream into *lexemes* according to the *grammar* of the language
    - build attributed *tokens*
  - output: token stream
    - a token contains the token name (id) and the token attributes

      < token name, attribute values >

Lexical Analysis

- Example:

position = initial + rate * 60

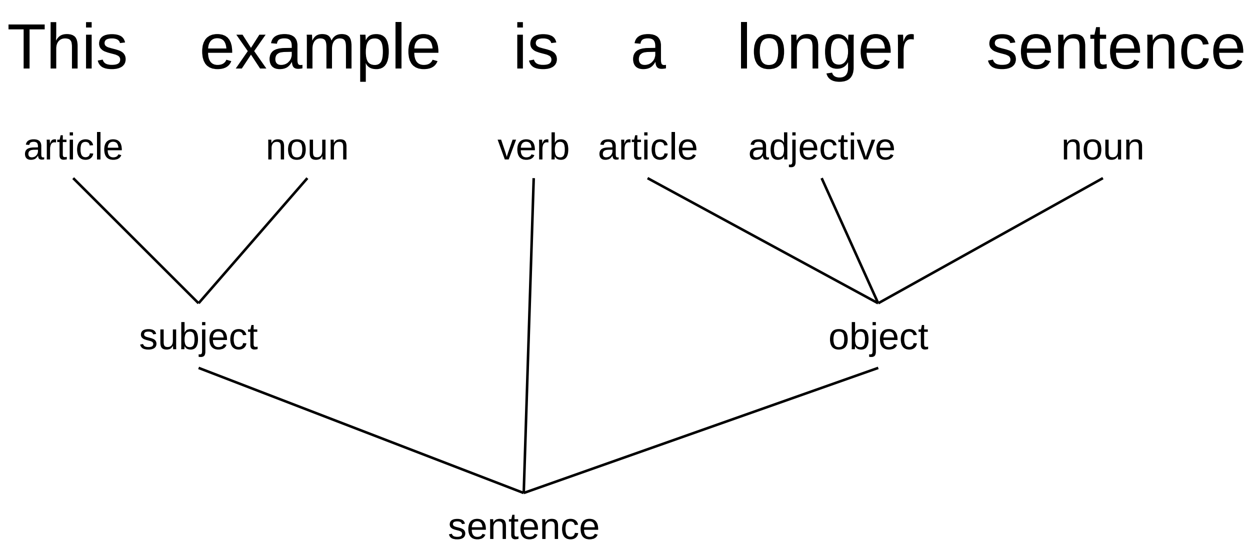
1. lexeme “position” → token <id, 1>
2. lexeme “=” → token <=>
3. lexeme “initial” → token <id, 2>
4. lexeme “+” → token <+>
5. lexeme “rate” → token <id, 3>
6. lexeme “*” → token <*>
7. lexeme “60” → token <number, 60>

<id,1> <=> <id,2> <+> <id,3> <*> <number,60>

Syntax Analysis

- **Syntax Analysis** or **Parsing**

  understand the structure of the input

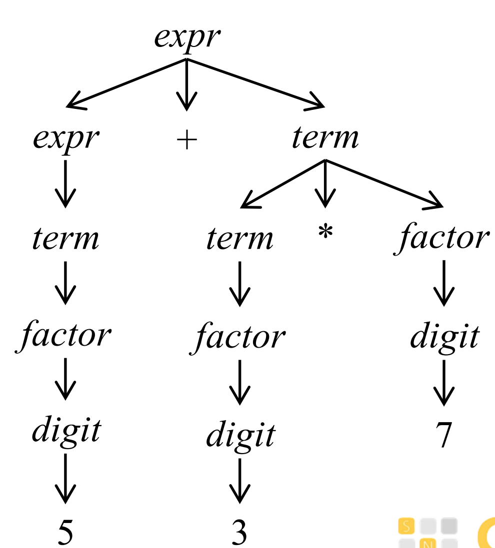


Syntax Analysis

- More formally:
  - input: token stream
  - process: transform token stream into a syntax (parse) tree grammar of the language by starting at the start symbol and repeatedly applying productions
  - output: syntax tree, symbol table

$expr \rightarrow expr + term \mid expr - term \mid term$
$term \rightarrow term * factor \mid term \,/\, factor \mid factor$
$factor \rightarrow \mathbf{digit} \mid (\ expr\ )$

```
< digit, 5 >, < + >,
< digit, 3 >, < * >,
< digit, 7 >
```

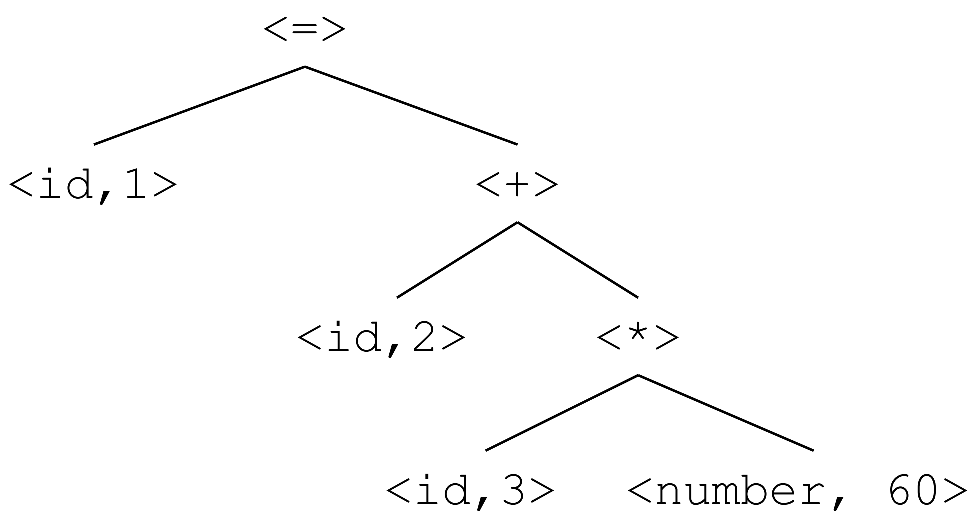


Syntax Analysis

- Example

```
position = initial + rate * 60
```

```
<id,1> <=> <id,2> <+> <id,3> <*> <number,60>
```

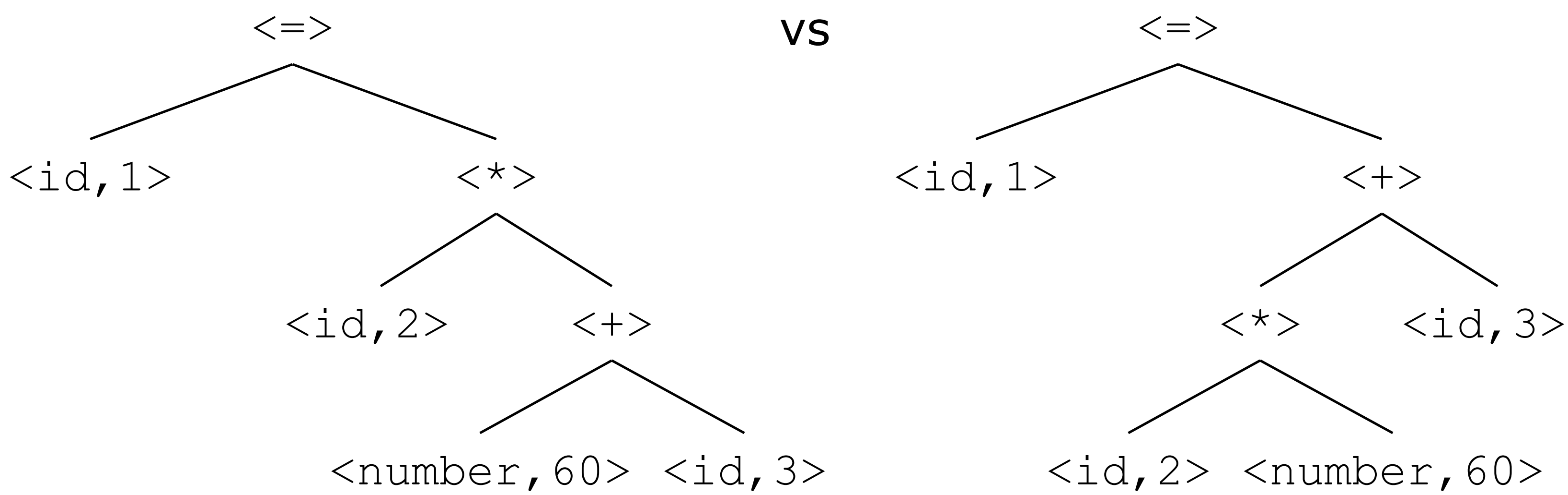


Syntax Analysis

- Another Example

```
position = rate * 60 + initial
```

```
<id,1> <=> <id,2> <*> <number,60> <+> <id, 3>
```



which syntax tree is correct?

Semantic Analysis

- **Semantic Analysis**

  understand the meaning of the input

  - this is too hard for compilers → semantic analysis is restricted to detecting inconsistencies

Tom said Jerry left his phone at home.

whose phone?

Tom said Tom left his phone at home.

how many people are we talking about?

Semantic Analysis

- **Semantic Analysis**

  compilers catch inconsistencies

Tom left her phone at home.

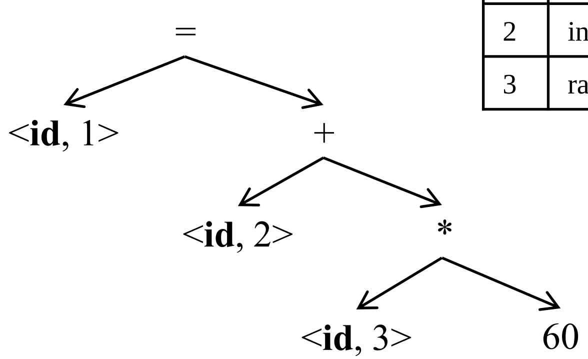
type mismatch

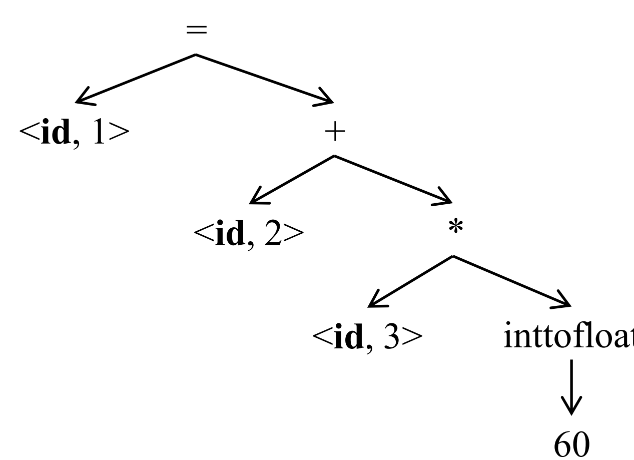
Semantic Analysis

- More formally:
  - input: syntax tree, symbol table
  - process: semantic consistency checking, type checking
  - output: (semantically sound) syntax tree, possibly added *coercions*

```
position = initial + rate * 60
```

1	position	float	…
2	initial	float	…
3	rate	float	…





Semantic Analysis

- Examples
  - type checking

```
public static void fun(int a, java.lang.String[] s)
{
  int sum = a + s;
  …
}
```

  - variable bindings

```
{
  int i = 4;
  {
    int i = 6;
    printf("%d\n", i);
  }
}
```

Optimization

- **Optimization**

  no strong counterpart in English; similar to editing.

Optimization is a little bit like editing.

⇩

Optimization is akin to editing.

Optimization here: reduce number of words.

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Optimization

- Optimization

  automatically modify programs so that they use less of some resource.

  - run faster
  - use less memory
  - use less (disk) space
  - consume less energy
  - reduce number of network accesses
  - …

Optimization

- Common (High-Level) Optimizations
  - common subexpression elimination
  - dead code elimination
  - code motion
  - constant propagation
  - partial-redundancy elimination
  - loop optimizations

Optimization

- Example

  `x = y * 0` can be optimized to `x = 0`

  unfortunately, this rule is not correct.

  Optimizations should to be **conservative**, i.e., be correct under *all* circumstances.

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Code Generation

- **Code Generation** or **CodeGen**

  translation into another language, e.g., English into Korean

I miss you

보고 싶다

Code Generation

- More formally
  - input: intermediate code
  - process:
    - map to machine code
    - register allocation
    - memory locations

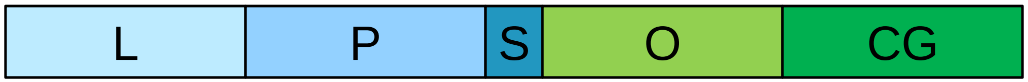
```
t1 = id3 * 60.0
id1 = id2 + t1
```

⇨

```
ldf   r2, id3
mulf  r2, r2, #60.0
ldf   r3, id2
addf  r3, r3, r2
stf   id1, r3
```

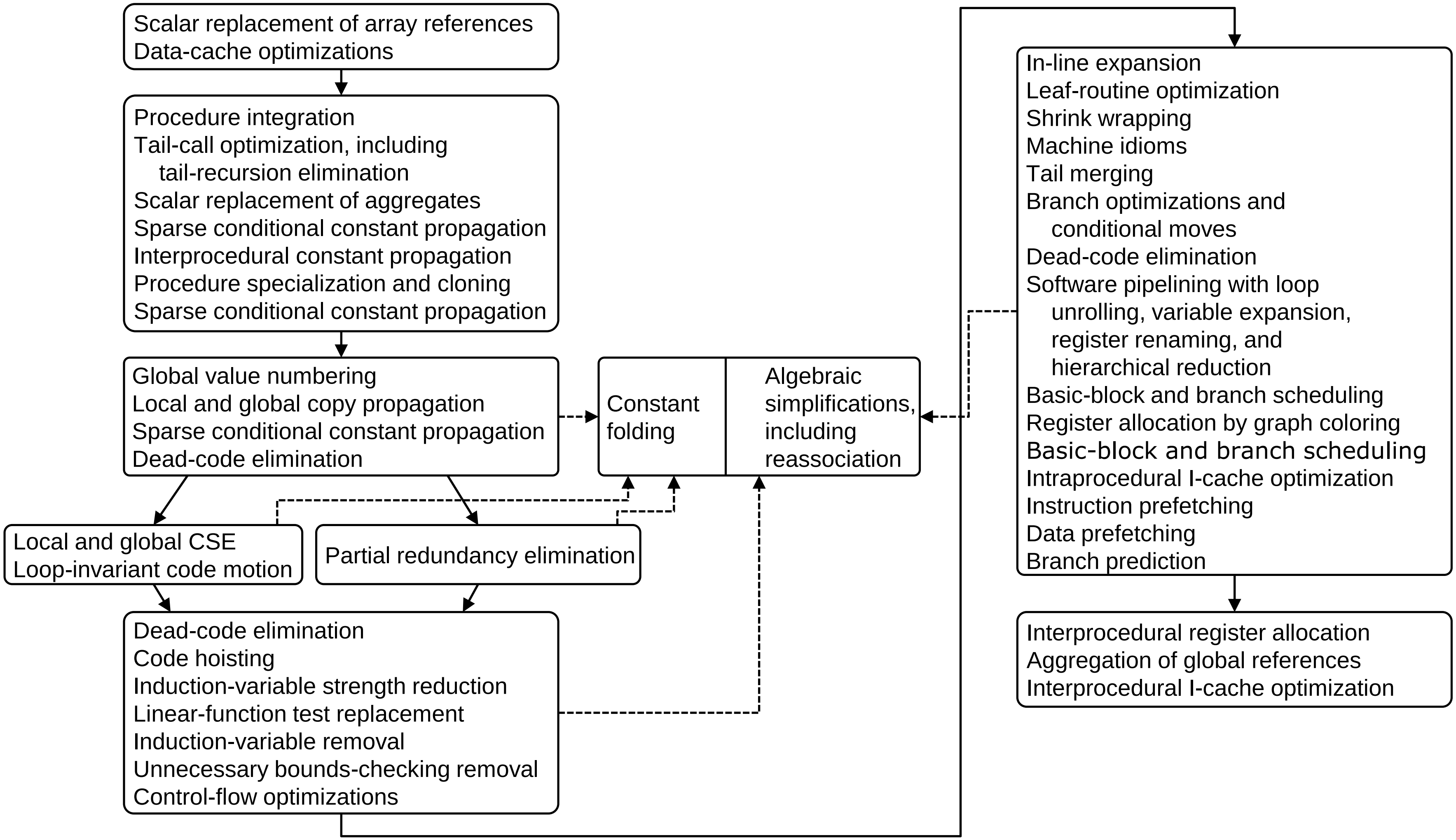
Proportions of the Compiler Phases

- FORTRAN



- modern compilers

Optimizations in a Modern Compiler



source: Steven Muchnick, Advanced Compiler Design & Implementation

Summary

- Compilers are fundamental to Computer Science

- Basic Structure of a Compiler

1.	Lexical Analysis	from a character stream to tokens
2.	Syntax Analysis	from tokens to an AST
3.	Semantic Analysis	type checking
4.	Optimization	varying degree of complexity
5.	Code Generation	from IR to machine code

- This class covers all phases except optimizations
- We will build a simple but fully functional compiler for a Pascal-based language