Why Study Compilers?

- Compilers enable programming at a high level language instead of machine instructions.
  - Malleability, Portability, Modularity, Simplicity, Programmer Productivity
  - Also Efficiency and Performance

Compilers Construction touches many topics in Computer Science

- Theory
  - Finite State Automata, Grammars and Parsing, data-flow
- Algorithms
  - Graph manipulation, dynamic programming
- Data structures
  - Symbol tables, abstract syntax trees
- Systems
  - Allocation and naming, multi-pass systems, compiler construction
- Computer Architecture
  - Memory hierarchy, instruction selection, interlocks and latencies, parallelism
- Security
  - Detection of and Protection against vulnerabilities
- Software Engineering
  - Software development environments, debugging
- Artificial Intelligence
  - Heuristic based search for best optimizations

Power of a Language

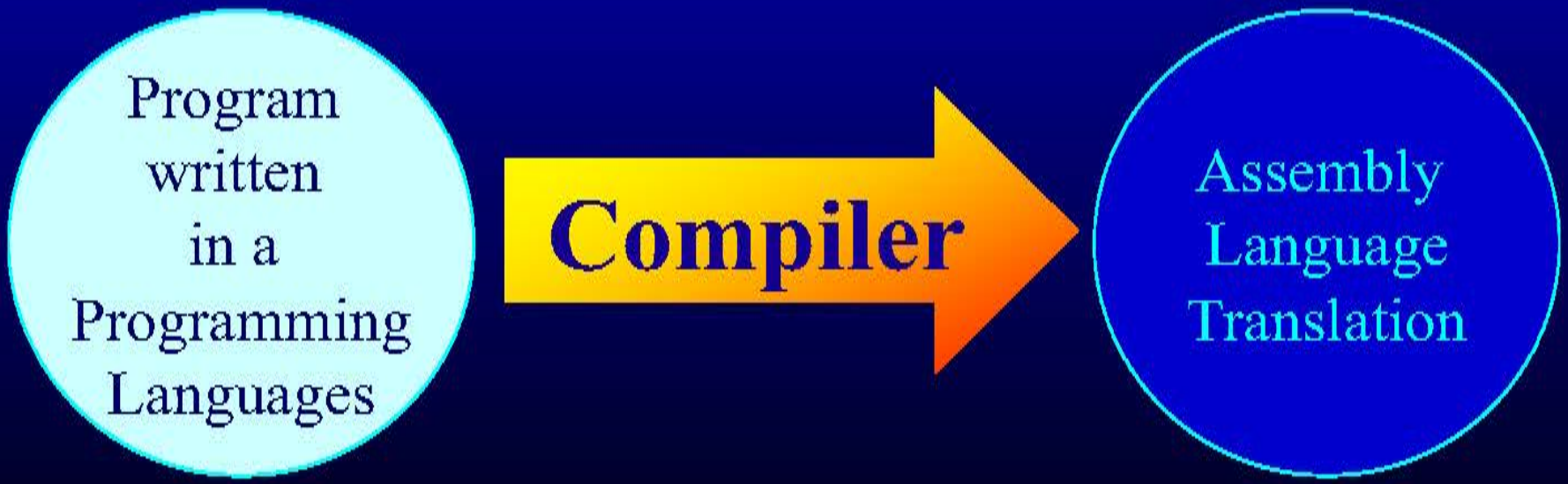
- Can use to describe any action
  - Not tied to a "context"
- Many ways to describe the same action
  - Flexible

How to instruct a computer

- How about natural languages?
  - English??
  - "Open the pod bay doors, Hal."
  - "I am sorry Dave, I am afraid I cannot do that"
  - We are not there yet!!
- Natural Languages:
  - Powerful, but…
  - Ambiguous
    - Same expression describes many possible actions

1. How to instruct the computer

- Write a program using a programming language
  - High-level Abstract Description
- Microprocessors talk in assembly language
  - Low-level Implementation Details



1. How to instruct the computer

- Input: High-level programming language
- Output: Low-level assembly instructions

- Compiler does the translation:
  - Read and understand the program
  - Precisely determine what actions it require
  - Figure-out how to faithfully carry-out those actions
  - Instruct the computer to carry out those actions

Input to the Compiler

- Standard imperative language (Java, C, C++)
  - State
    - Variables,
    - Structures,
    - Arrays
  - Computation
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (conditionals, loops)
    - Procedures

Output of the Compiler

- State
  - Registers
  - Memory with Flat Address Space
- Machine code – load/store architecture
  - Load, store instructions
  - Arithmetic, logical operations on registers
  - Branch instructions

Example (input program)

```
int sumcalc(int a, int b, int N)
{
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
}
```

Example (Output assembly code)

```
sumcalc:
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    %edx, -12(%rbp)
        movl    $0, -20(%rbp)
        movl    $0, -24(%rbp)
        movl    $0, -16(%rbp)
.L2:    movl    -16(%rbp), %eax
        cmpl    -12(%rbp), %eax
        jg      .L3
        movl    -4(%rbp), %eax
        leal    0(,%rax,4), %edx
        leaq    -8(%rbp), %rax
        movq    %rax, -40(%rbp)
        movl    %edx, %eax
        movq    -40(%rbp), %rcx
        cltd
        idivl   (%rcx)
        movl    %eax, -28(%rbp)
        movl    -28(%rbp), %edx
        imull   -16(%rbp), %edx
        movl    -16(%rbp), %eax
        incl    %eax
        imull   %eax, %eax
        addl    %eax, %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        movl    -8(%rbp), %eax
        movl    %eax, %edx
        imull   -24(%rbp), %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        leaq    -16(%rbp), %rax
        incl    (%rax)
        jmp     .L2
.L3:    movl    -20(%rbp), %eax
        leave
        ret
```

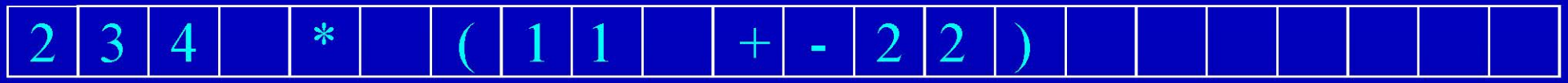
```
        .size   sumcalc, .-sumcalc
        .section
.Lframe1:
        .long   .LECIE1-.LSCIE1
.LSCIE1:.long   0x0
        .byte   0x1
        .string ""
        .uleb128 0x1
        .sleb128 -8
        .byte   0x10
        .byte   0xc
        .uleb128 0x7
        .uleb128 0x8
        .byte   0x90
        .uleb128 0x1
        .align  8
.LECIE1:.long   .LEFDE1-.LASFDE1
        .long   .LASFDE1-.Lframe1
        .quad   .LFB2
        .quad   .LFE2-.LFB2
        .byte   0x4
        .long   .LCFI0-.LFB2
        .byte   0xe
        .uleb128 0x10
        .byte   0x86
        .uleb128 0x2
        .byte   0x4
        .long   .LCFI1-.LCFI0
        .byte   0xd
        .uleb128 0x6
        .align  8
```

Mapping Time Continuum Compilation to Interpretation

- Compile time
  - Ex: C compiler
- Link time
  - Ex: Binary layout optimizer
- Load time
  - Ex: JIT compiler
- Run time
  - Ex: Java Interpreter

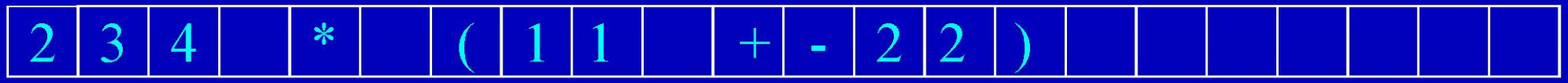
Anatomy of a Computer

Lexical Analyzer (Scanner)



Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op

Lexical Analyzer (Scanner)



Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op

18..23 + val#ue

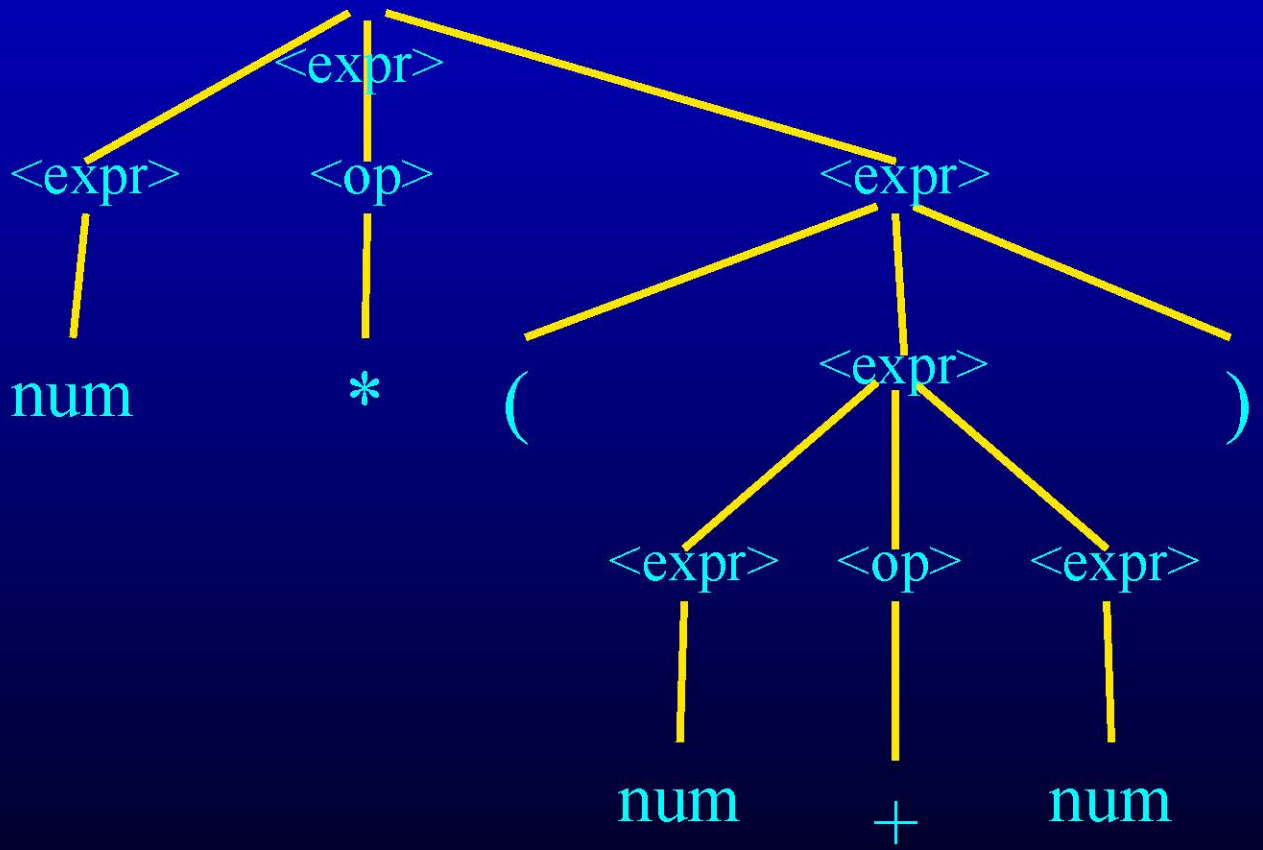
Variable names cannot have '#' character

Not a number

Anatomy of a Computer

Syntax Analyzer (Parser)

num ‘*’ ‘(‘ num ‘+’ num ‘)’



Syntax Analyzer (Parser)

```
int * foo(i, j, k))
    int i;
    int j;
{
    for(i=0; i j) {
    fi(i>j)
        return j;
}
```

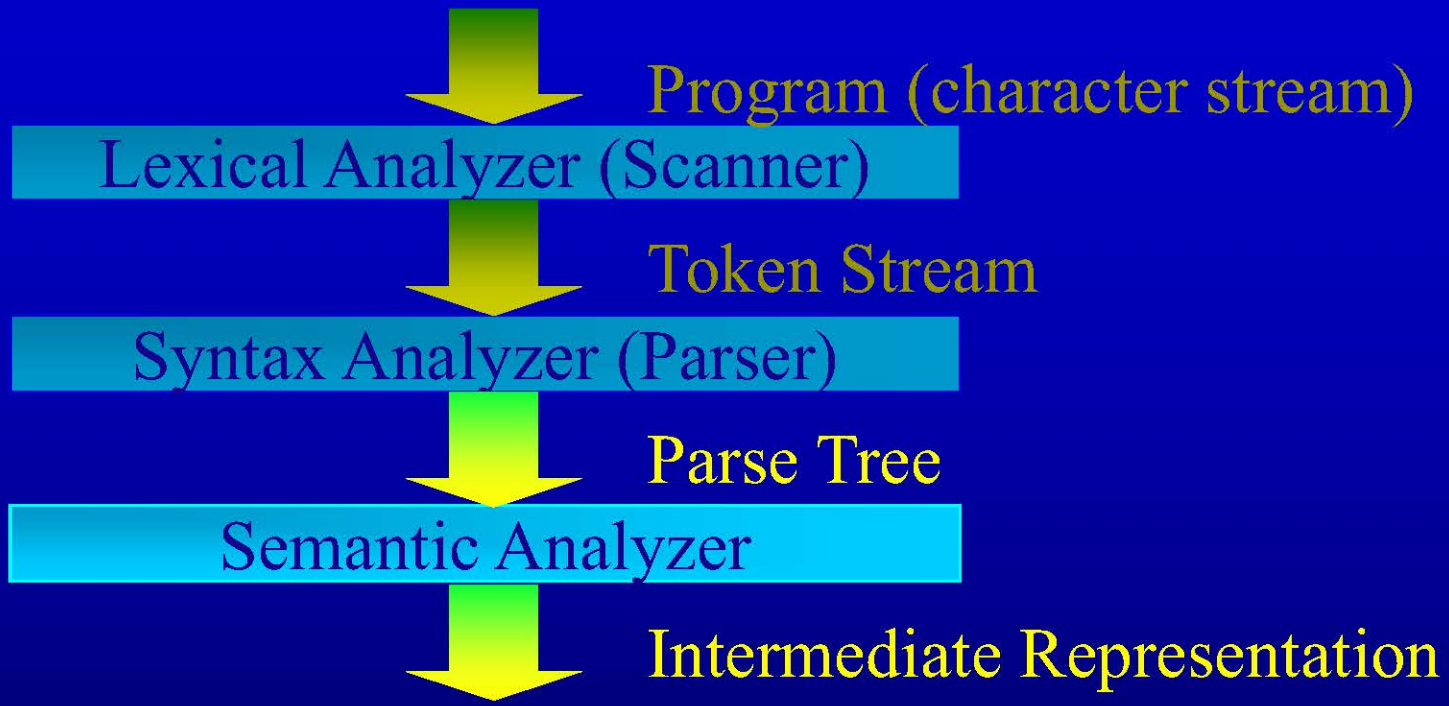
Extra parentheses

Missing increment

Not an expression

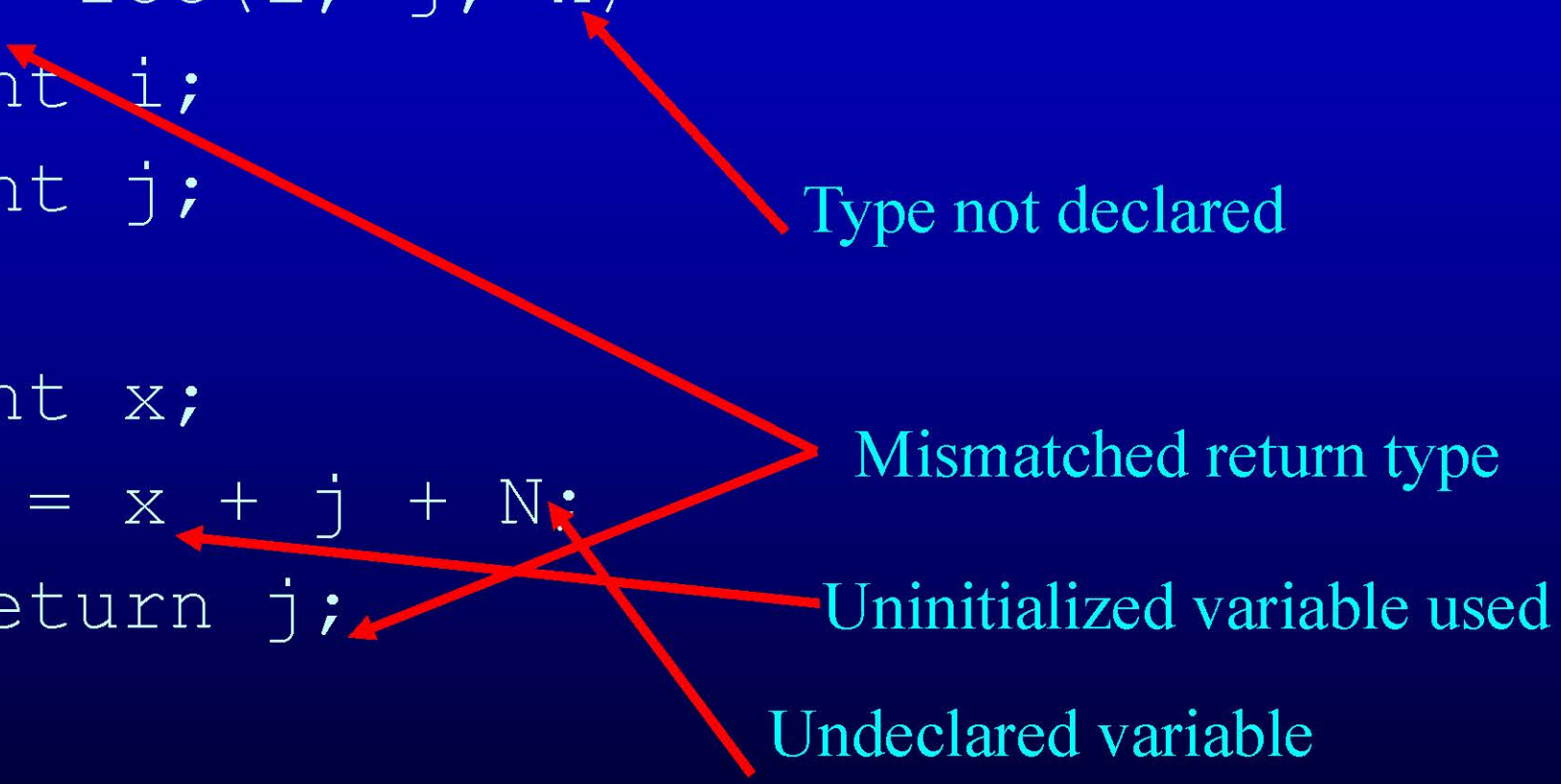
Not a keyword

Anatomy of a Computer

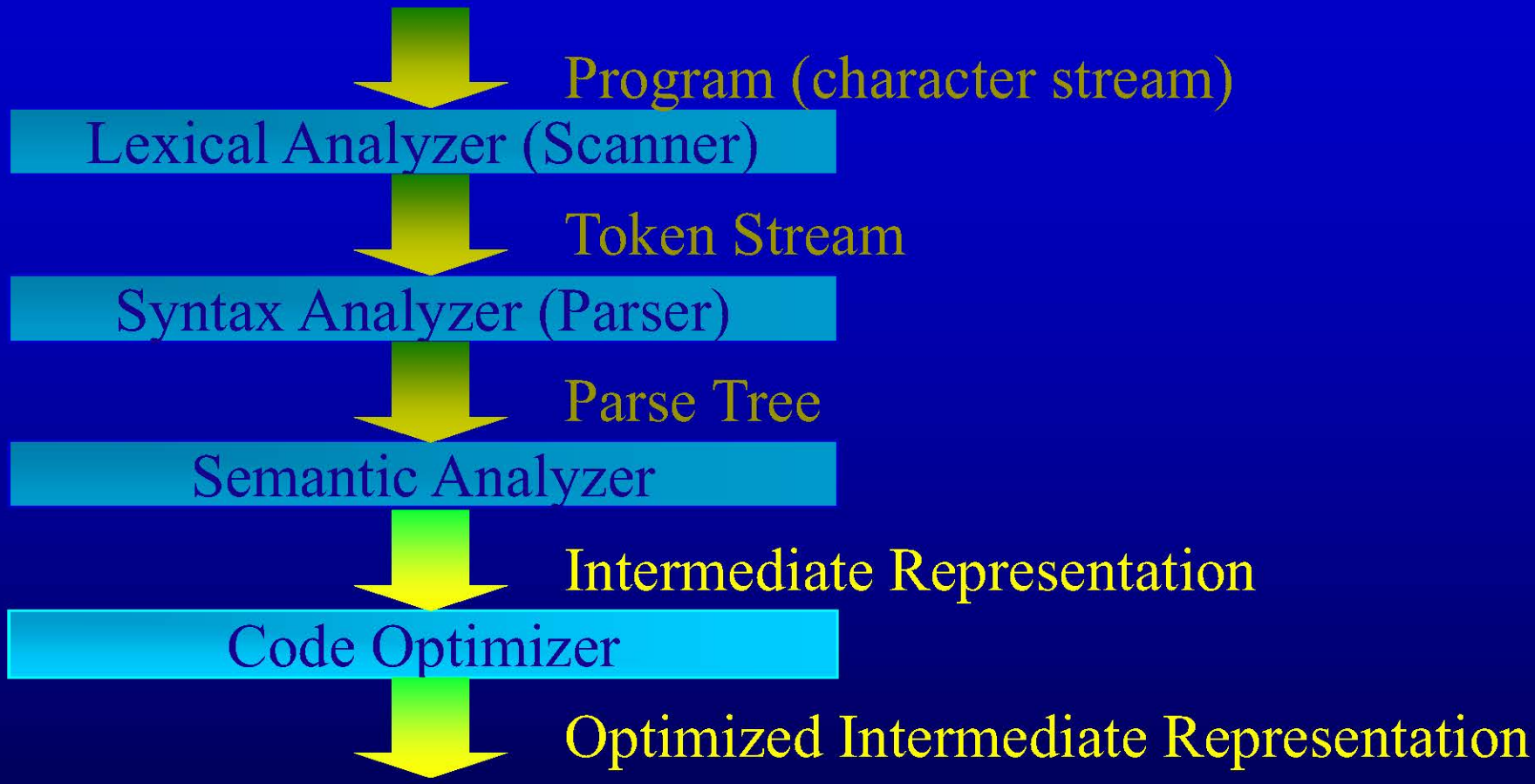


Semantic Analyzer

```
int * foo(i, j, k)
    int i;
    int j;
{
    int x;
    x = x + j + N;
    return j;
}
```

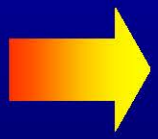


Anatomy of a Computer



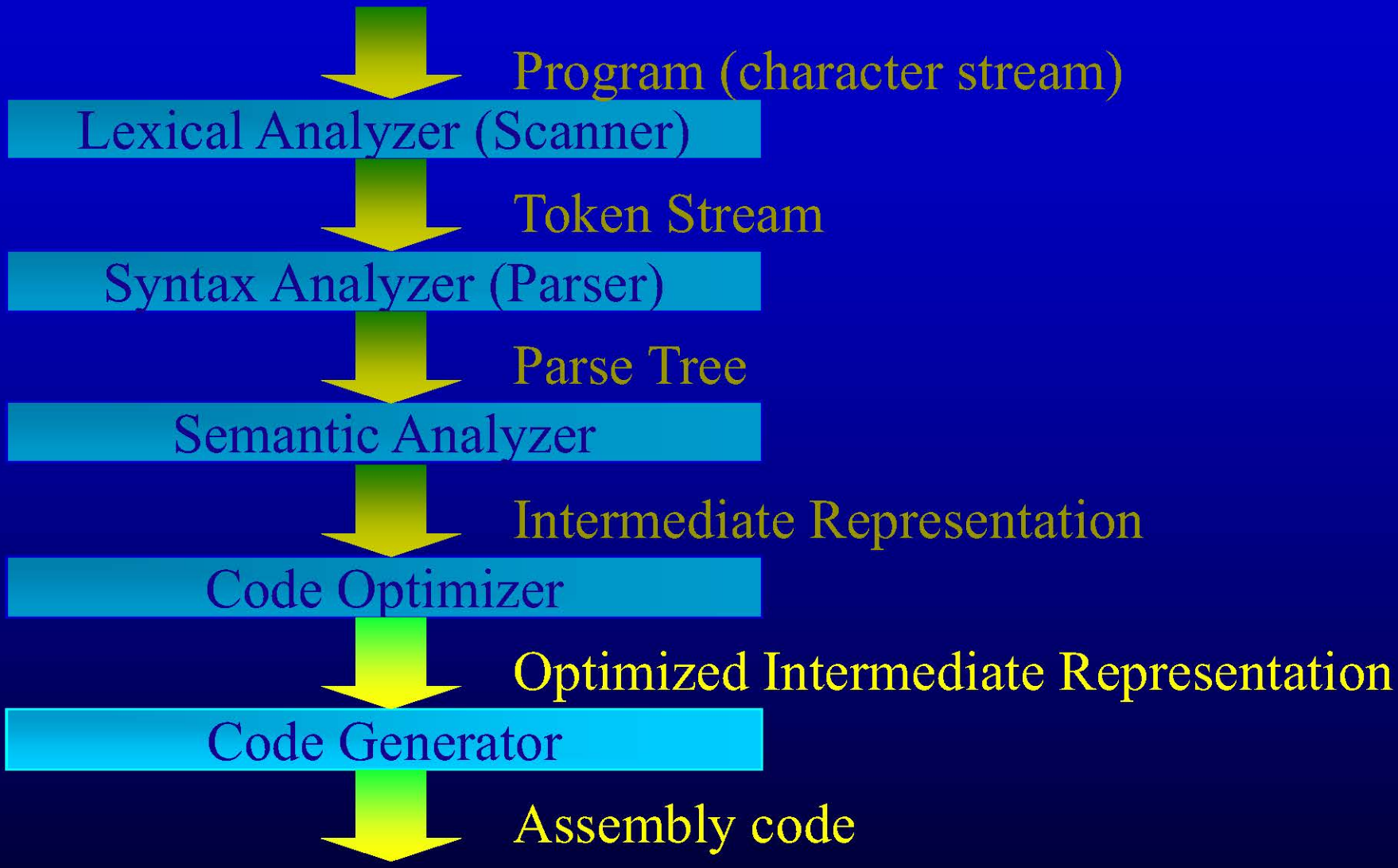
Optimizer

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x+4*a/b*i+(i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
```



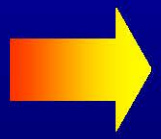
```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}
```

Anatomy of a Computer



Code Generator

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
     u = ((a<<2)/b);
     v = 0;
    for(i = 0; i <= N; i++) {
          t = i+1;
           x = x + v + t*t;
          v = v + u;
     }
    return x;
}
```



```
sumcalc:
        xorl    %r8d, %r8d
        xorl    %ecx, %ecx
        movl    %edx, %r9d
        cmpl    %edx, %r8d
        jg      .L7
        sall    $2, %edi
.L5:    movl    %edi, %eax
        cltd
        idivl   %esi
        leal    1(%rcx), %edx
        movl    %eax, %r10d
        imull   %ecx, %r10d
        movl    %edx, %ecx
        imull   %edx, %ecx
        leal    (%r10,%rcx), %eax
        movl    %edx, %ecx
        addl    %eax, %r8d
        cmpl    %r9d, %edx
        jle     .L5
.L7:    movl    %r8d, %eax
        ret
```

Program Translation

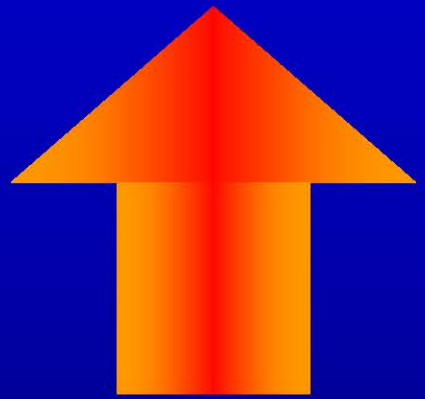
- Correct
  - The actions requested by the program has to be faithfully executed
- Efficient
  - Intelligently and efficiently use the available resources to carry out the requests
  - (the word optimization is used loosely in the compiler community – Optimizing compilers are never optimal)

Efficient Execution

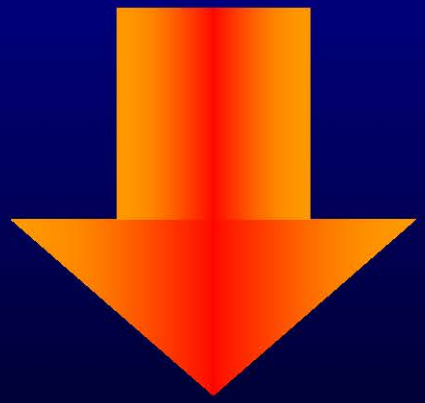
- Mapping from High to Low
  - Simple mapping of a program to assembly language produces inefficient execution
  - Higher the level of abstraction $\Rightarrow$ more inefficiency
- If not efficient
  - High-level abstractions are useless
- Need to:
  - provide a high level abstraction
  - with performance of giving low-level instructions

Efficient Execution help increase the level of abstraction

- Programming languages
  - From C to OO-languages with garbage collection
  - Even more abstract definitions

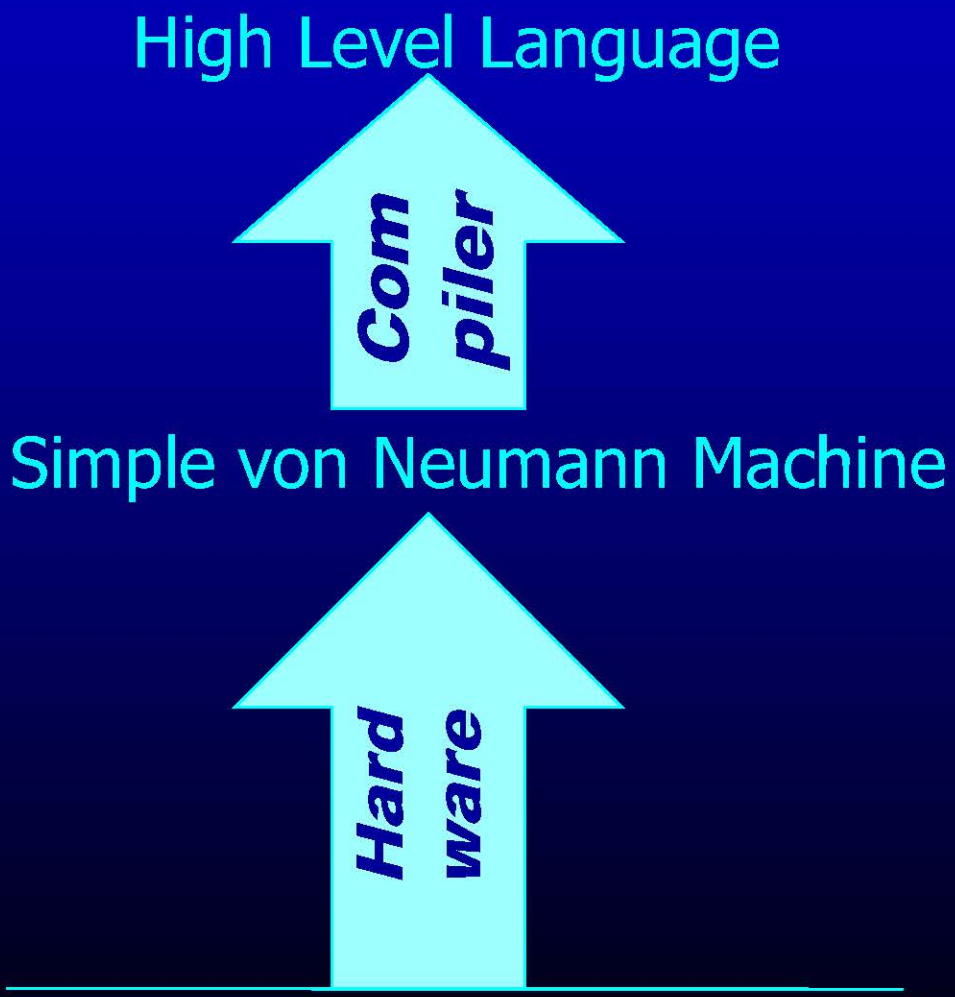


- Microprocessor
  - From simple CISC to RISC to VLIW to ….

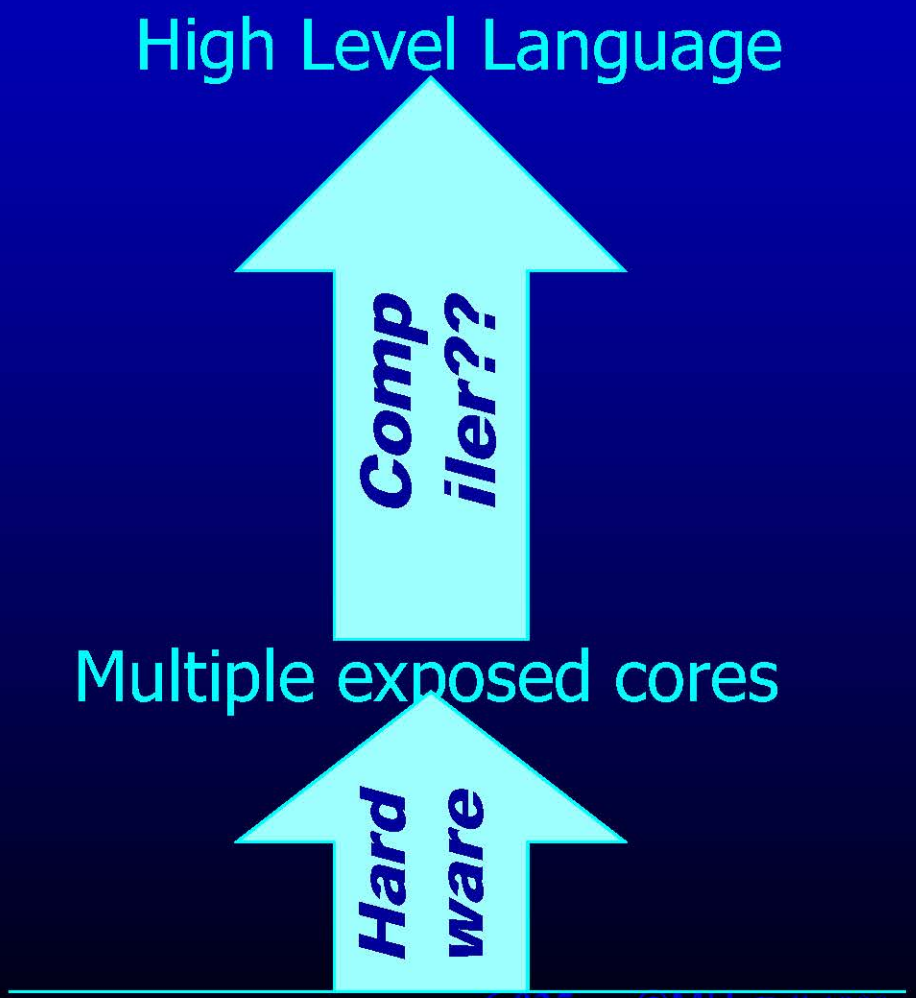


The Multicore Dilemma

- Superscalars

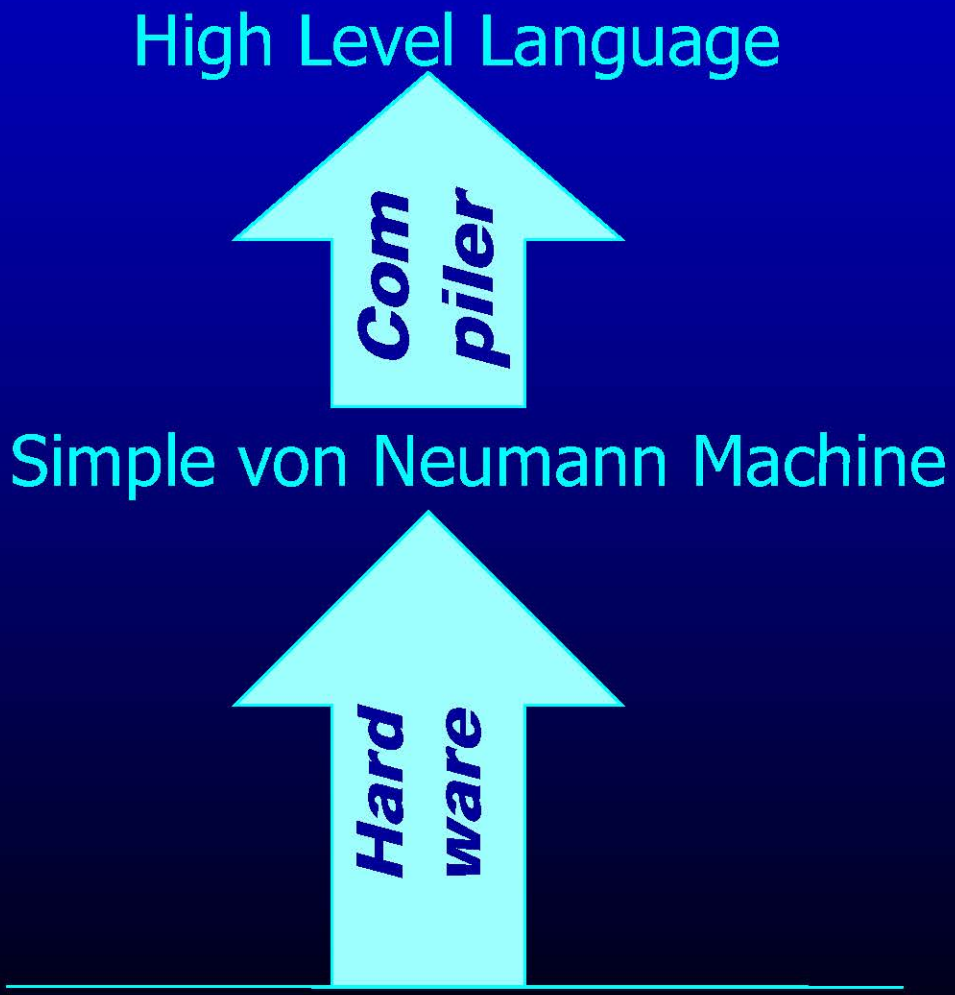


- Multicores

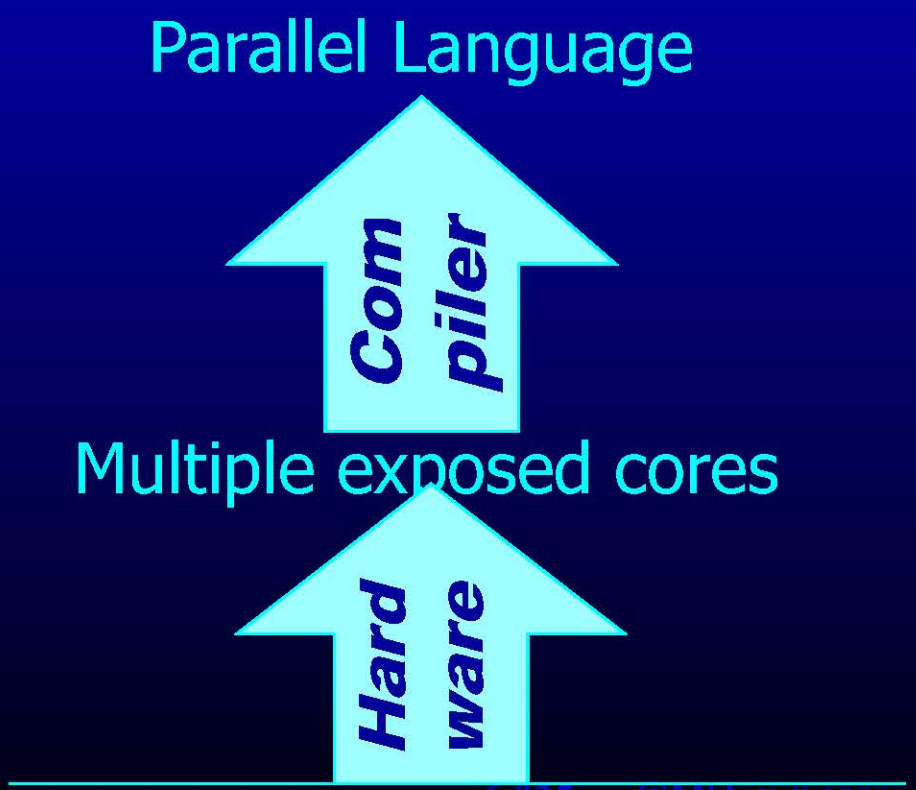


The Multicore Dilemma

- Superscalars



- Multicores



Optimization Example

```
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}
```

```
        pushq   %rbp
        movq    %rsp, %rbp
        movl    %edi, -4(%rbp)
        movl    %esi, -8(%rbp)
        movl    %edx, -12(%rbp)
        movl    $0, -20(%rbp)
        movl    $0, -24(%rbp)
        movl    $0, -16(%rbp)
.L2:    movl    -16(%rbp), %eax
        cmpl     12(%rbp), %eax
        jg      .L3
        movl    -4(%rbp), %eax
        leal    0(,%rax,4), %edx
        leaq    -8(%rbp), %rax
        movq    %rax, -40(%rbp)
        movl    %edx, %eax
        movq    -40(%rbp), %rcx
        cltd
        idivl   (%rcx)
        movl    %eax, -28(%rbp)
        movl    -28(%rbp), %edx
        imull   -16(%rbp), %edx
        movl    -16(%rbp), %eax
        incl    %eax
        imull   %eax, %eax
        addl    %eax, %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        movl    -8(%rbp), %eax
        movl    %eax, %edx
        imull    24(%rbp), %edx
        leaq    -20(%rbp), %rax
        addl    %edx, (%rax)
        leaq    -16(%rbp), %rax
        incl    (%rax)
        jmp      L2
.L3:    movl    -20(%rbp), %eax
        leave
        ret
```

Lets Optimize...

```
int sumcalc(int a, int b, int N)
{
    int i, x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
       x = x + (4*a/b)*i + (i+1)*(i+1);
       x = x + b*y;
    }
    return x;
}
```

Constant Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    x = x + (4*a/b)*i + (i+1)*(i+1);
    x = x + b*y;
}
return x;
```

Constant Propagation

```
int i, x, y;
 x = 0;
 y = 0;
 for(i = 0; i <= N; i++) {
   x = x + (4*a/b)*i + (i+1)*(i+1);
   x = x + b*y;
 }
 return x;
```

Constant Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x + b*0;
}
return x;
```

Algebraic Simplification

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x + b*0;
}
return x;
```

Algebraic Simplification

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x + b*0;
}
return x;
```

Algebraic Simplification

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x;
}
return x;
```

Copy Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x;
}
return x;
```

Copy Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);
  x = x;
}
return x;
```

Copy Propagation

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
  x = x + (4*a/b)*i + (i+1)*(i+1);

}
return x;
```

Common Subexpression Elimination

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {

    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```

Common Subexpression Elimination

```
int i, x, y;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {

    x = x + (4*a/b)*i + (i+1)*(i+1);
}
return x;
```

Common Subexpression Elimination

```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Dead Code Elimination

```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Dead Code Elimination

```
int i, x, y, t;
x = 0;
y = 0;
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Dead Code Elimination

```
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Loop Invariant Removal

```
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Loop Invariant Removal

```
int i, x, t;
x = 0;

for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + (4*a/b)*i + t*t;
}
return x;
```

Loop Invariant Removal

```
int i, x, t, u;
x = 0;
u = (4*a/b);
for(i = 0; i <= N; i++) {
    t = i+1;
    x = x + u*i + t*t;
}
return x;
```

Compilers Optimize Programs for...

- Performance/Speed
- Code Size
- Power Consumption
- Fast/Efficient Compilation
- Security/Reliability
- Debugging

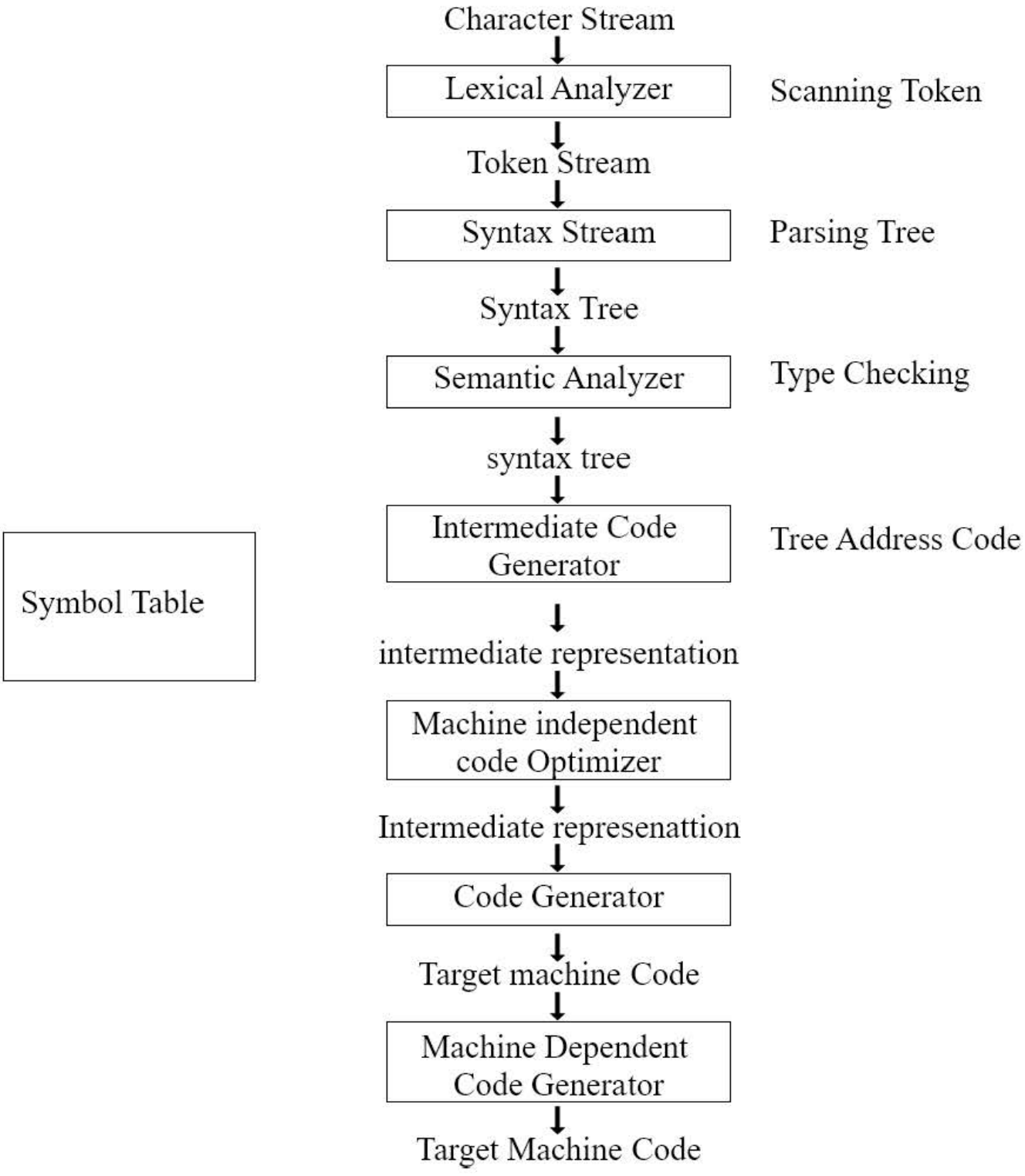
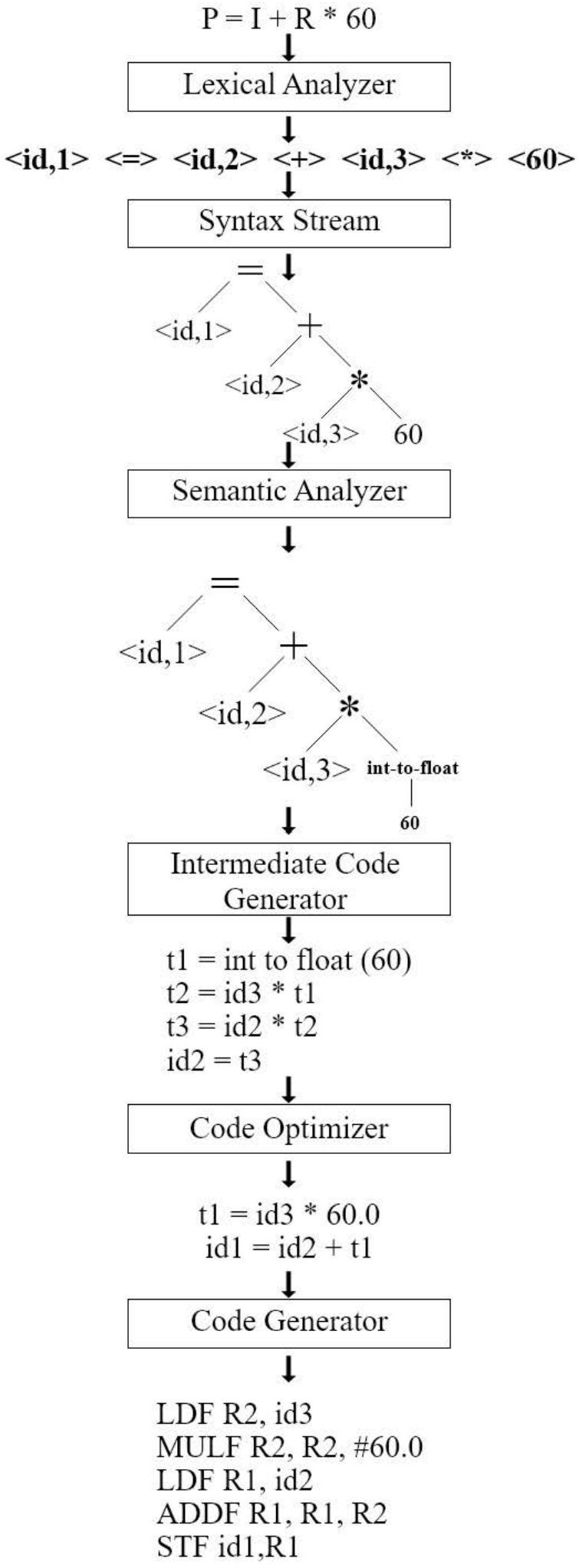


Figure: Phases of Compiler


P = I + R * 60
Lexical Analyzer
<id,1> <=> <id,2> <+> <id,3> <*> <60>
Syntax Stream
=
<id,1>
+
<id,2>
*
<id,3>
60
Semantic Analyzer
=
<id,1>
+
<id,2>
*
<id,3>
int-to-float
60
Intermediate Code Generator
t1 = int to float (60)
t2 = id3 * t1
t3 = id2 * t2
id2 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1,R1

1. Parser Generator –
It produces syntax analyzers (parsers) based on a grammatical description of programming language or on a context-free grammar. Example: PIC, EQM

2. Scanner Generator –
It generates lexical analyzers from regular expression description based on tokens of a language.
Example: Lex

3. Syntax directed translation engines –
It generates intermediate code with three address format from the input that consists of a parse tree.

4. Automatic code generators –
It generates the machine language for a target machine.

5. Data-flow analysis engines –
It is used in code optimization. Data flow analysis is a key part of the code optimization.

6. Compiler construction toolkits –
It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.