© Manuel Cargaleiro

Anatomy of a Compiler

Masters in Informatics and Computing Engineering
(MIEIC), 3rd Year

João M. P. Cardoso
Dep. de Engenharia Informática, Faculdade de Engenharia (FEUP),
Universidade do Porto, Porto, Portugal
Email: jmpc@fe.up.pt

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Sources

- Some of the PowerPoint slides are based on various sources:
  - From the course "*6.035 Computer Language Engineering*," by Martin Rinard, MIT, USA
  - From the course "CMPUT 680: Compiler Design and Optimization,"by José Nelson Amaral, University of Alberta, Canada
  - Books, internet sources, etc.
- The instructors acknowledge the authors of the slides and other info in this course

Importance of the Compiler

- Important role in the development of complex applications in short/feasible time
- Many of the knowledge and techniques are used by other tools:
    - Translation of languages and information
    - Interpretation of descriptions (e.g., html, *postscript*, *latex*, *word files*)
    - Processing words and sentences in Internet Browsers

Compiler Construction

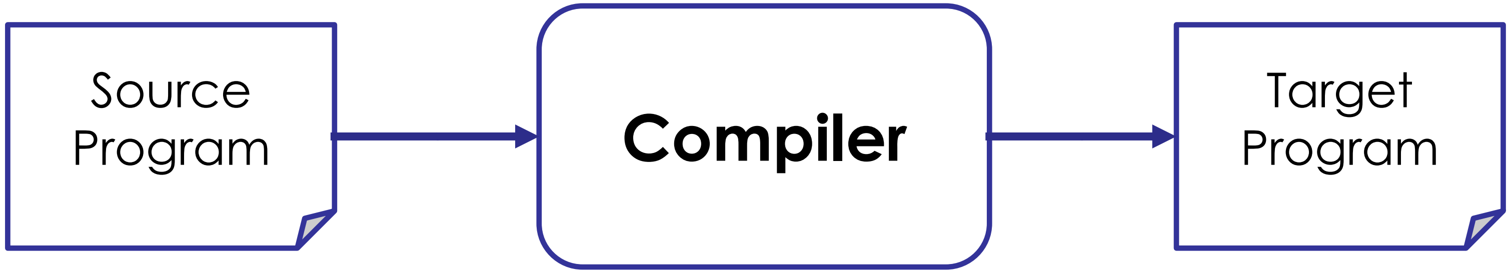
- "Compiler construction is an exercise in engineering design. The compiler writer must choose a path through a decision space that is filled with diverse alternatives, each with distinct costs, advantages, and complexity. Each decision has an impact on the resulting compiler. The quality of the end product depends on informed decisions at each step of way."
- "Thus, there is no *right* answer for these problems. Even within "well understood" and "solved" problems, nuances in design and implementation have an impact on both the behavior of the compiler and the quality of the code that it produces."
- In Cooper, Keith D., and Torczon, Linda, *Engineering a Compiler*, Morgan Kaufmann, 2nd edition, February 21, 2011.

Programming

- *Assembly* is tedious, error prone, with low productivity
  - But it can enable more optimized (for energy or performance) implementations
- High-level programming languages increase the abstraction layer
- How to implement a language?
  - Interpreter
  - Compiler

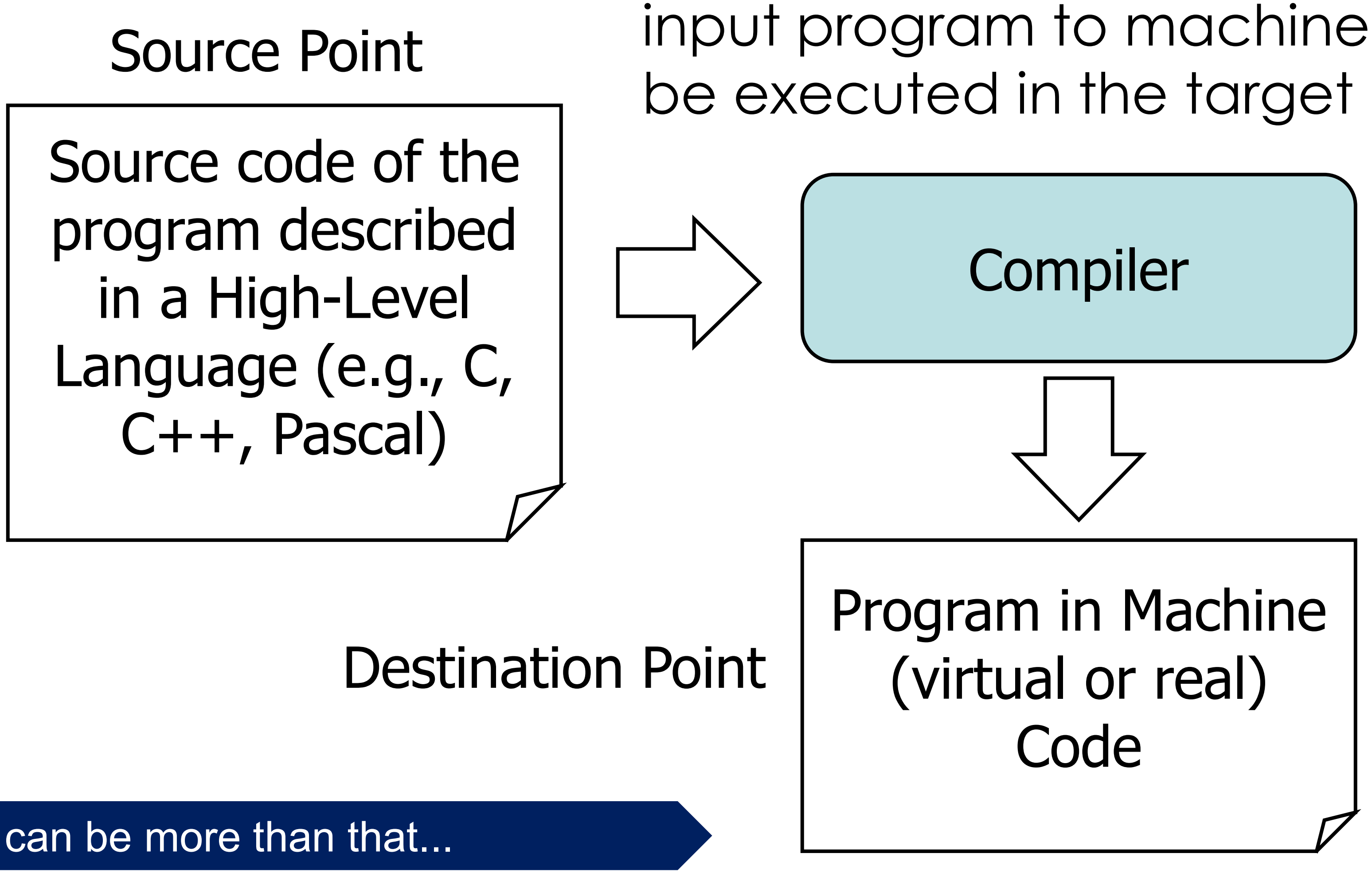
What is a Compiler?

- It is a software program that translates the **text** (and?) that represents a program to **machine code** able to be executed in the target computer
- **However, a compiler can be more than that...**



Compiler

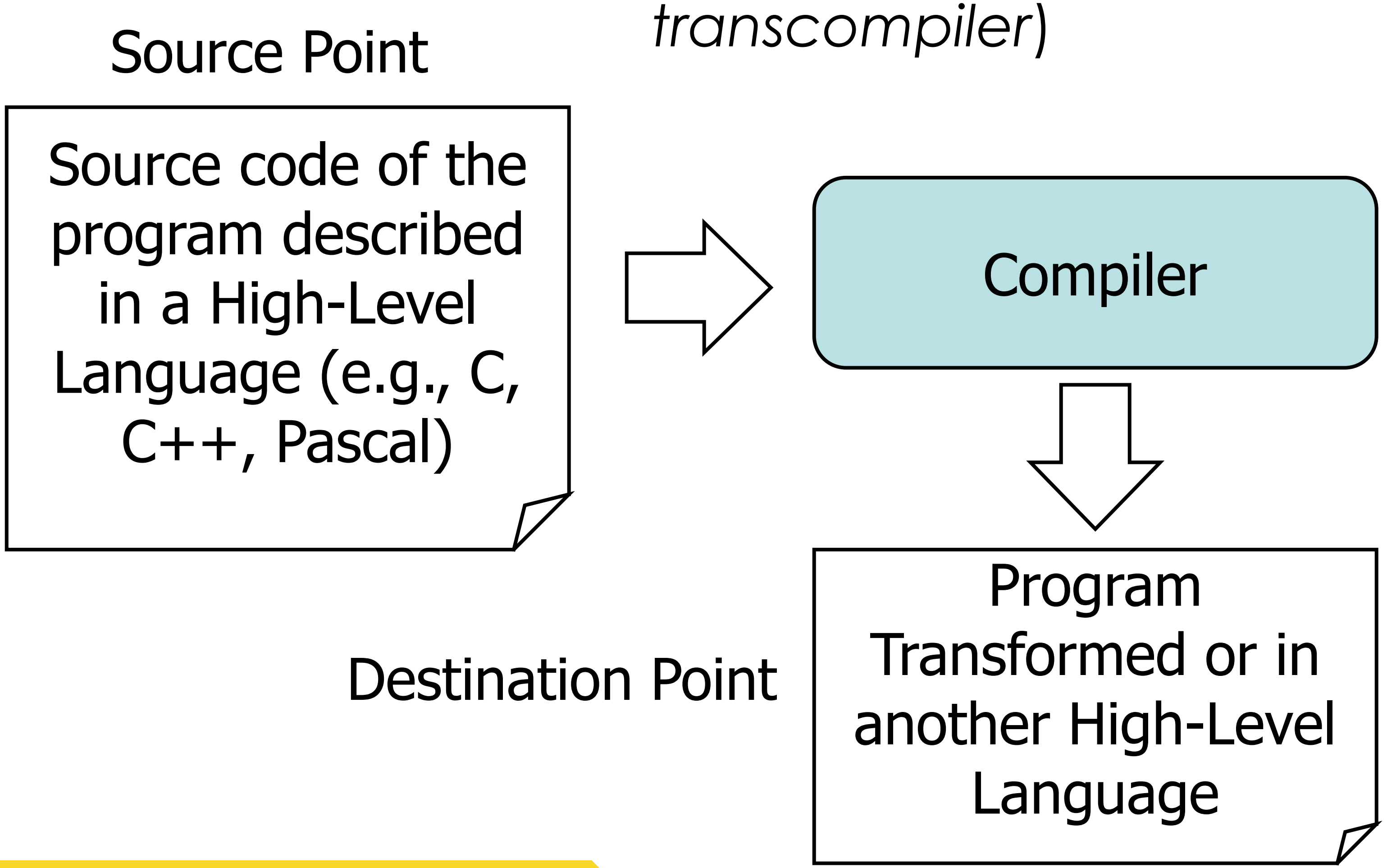
- A software program that translates the input program to machine code able to be executed in the target computer


Source Point
Source code of the program described in a High-Level Language (e.g., C, C++, Pascal)
Compiler
Program in Machine (virtual or real) Code
Destination Point

However, a compiler can be more than that...

Compiler

- Source to source compiler (*transpiler* or *transcompiler*)



Source Point

Source code of the program described in a High-Level Language (e.g., C, C++, Pascal)

Compiler

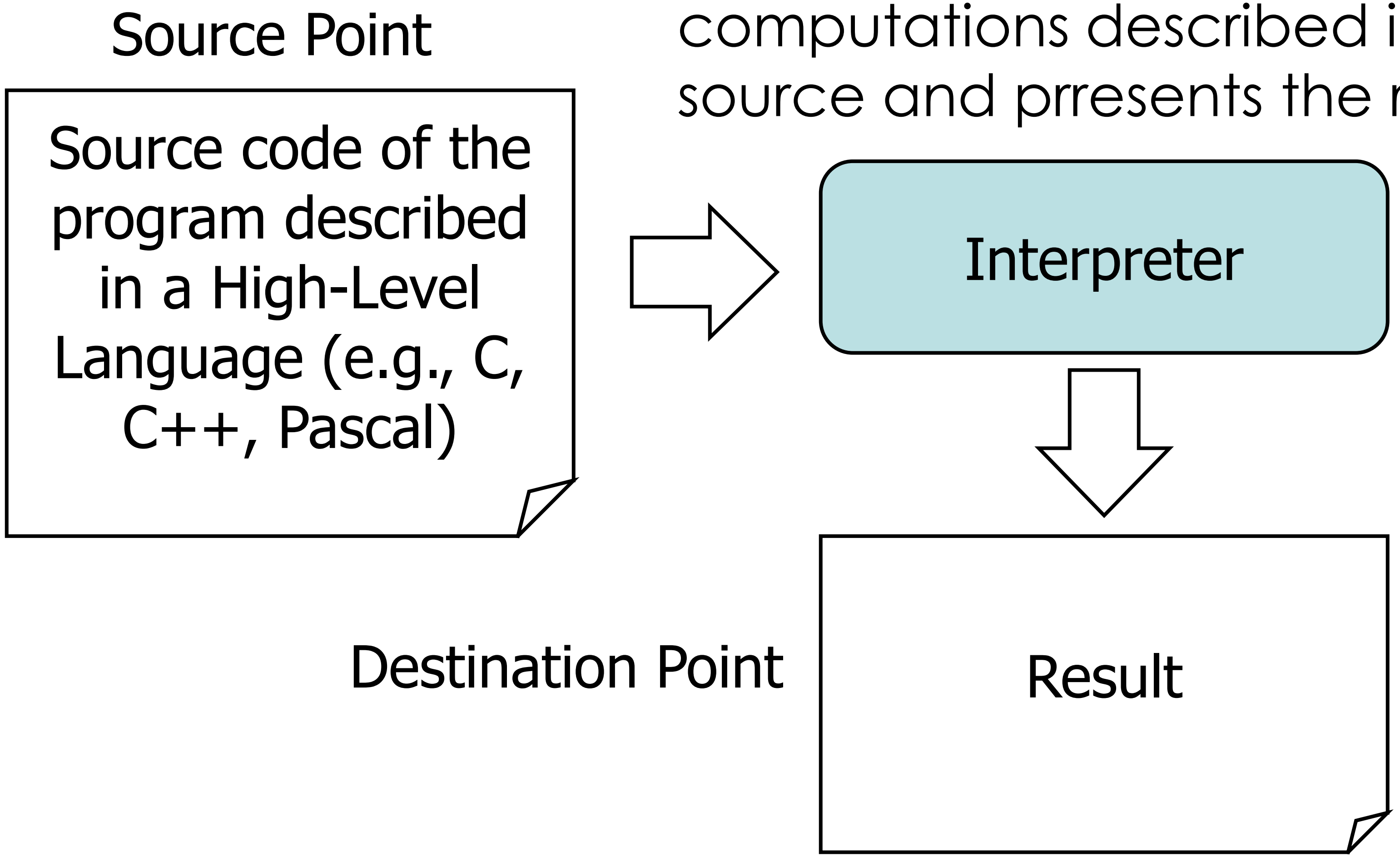
Destination Point

Program Transformed or in another High-Level Language

https://en.wikipedia-on-ipfs.org/wiki/Transpiler.html

Interpreter

- A software program that performs the computations described in the input source and prresents the results



Source Point

Source code of the program described in a High-Level Language (e.g., C, C++, Pascal)

Interpreter

Destination Point

Result

Other important components

- Pre-processor
  - Is it a source to source compiler?
- Linker
  - What is the main function of a linker?

Source Point

```
int sum(int A[], int N) {
   int i, sum = 0;
   for(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

- Imperative language (e.g., C/C++, Java, Pascal)
  - State
    - Scalar variables
    - Array variables
    - Structs
    - Classes, Objects, Fields (OO languages)
  - Computations
    - Expressions (arithmetic, logical, etc.)
    - Assignment statements
    - Control flow (if, switch, etc.)
    - Procedures/functions (methods in OO languages)

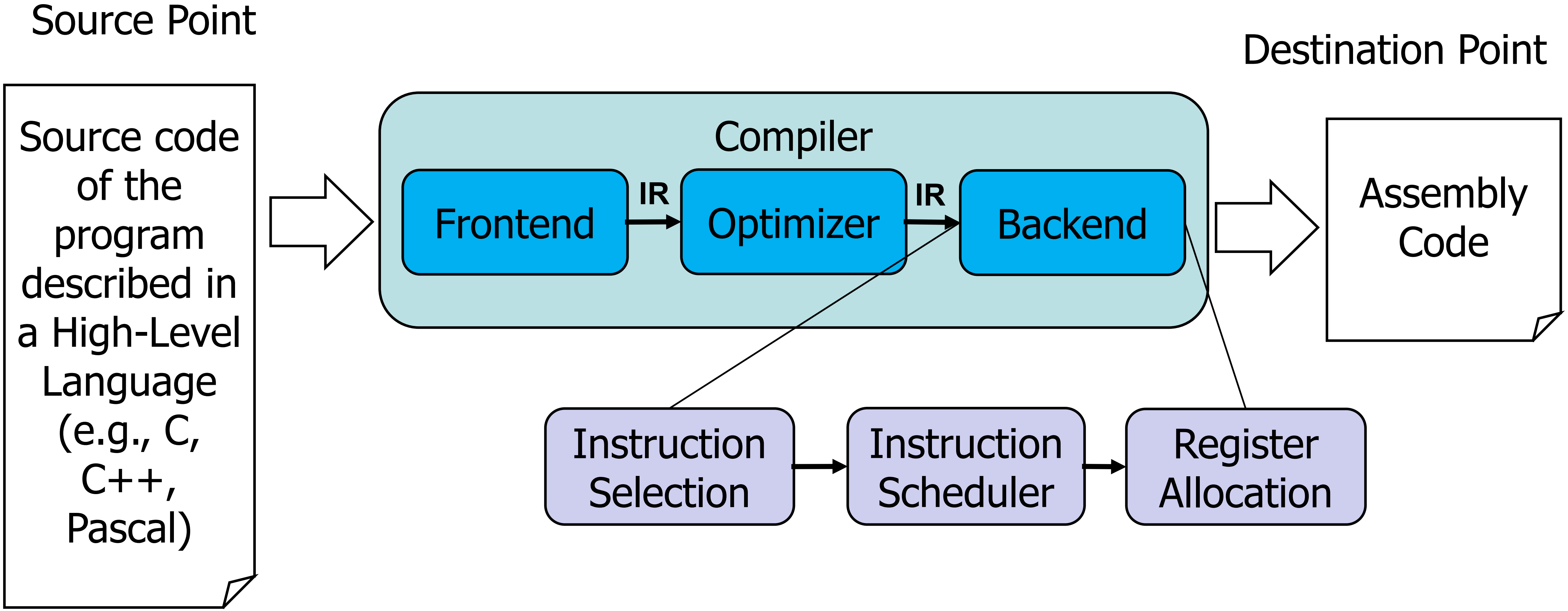
Destination Point

```
int sum(int A[], int N) {
   int i, sum = 0;
   For(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
Sum:  Addi  $t0, $0, 0
      Addi  $v0, $0, 0
Loop: beq   $t0, $a1, End
      Add   $t1, $t0, $t0
      Add   $t1, $t1, $t1
      Add   $t1, $t1, $a0
      Lw    $t2, 0($t1)
      Add   $v0, $v0, $t2
      Addi  $t0, $t0, 1
      J     Loop
End:  jr    $ra
```

- Machine code that describes the program using the ISA (*Instruction-Set Architecture*) of the target microprocessor
  - State
    - Memory
    - Registers (state, general purpose)
  - Computations
    - Instructions of the ISA (MIPS):
      - Lw $3, 100($2)
      - Add $3, $2, $1
      - Bne $2, $3, label
      - …

Compiler Stages



Topics

- Study of the compiler stages
- Construction of a simplified compiler
- Lectures are suitable to discuss doubts related to the implementation of a compiler
- Lectures focus on the fundamental techniques and knowledge to construct a compiler
  - Which techniques? How to apply them? What are the challenges?

Compilers

FROM HIGH-LEVEL TO *ASSEMBLY*

An Adventure

- How to implement the computing structures used in programming languages in *assembly*?
  - Review of the concepts related to *assembly* programming using the MIPS R3000
- In computer architecture you have done the role of the compiler. Now you are going to learn how to construct a compiler!

From High-Level to *Assembly*

- Target: MIPS R3000

```
Int sum(int A[], int N) {
   Int i, sum = 0;
   For(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
# $a0 stores the address of A[0]
# $a1 stores the value of N
Sum:    Addi    $t0, $0, 0        # i = 0
        Addi    $v0, $0, 0        # sum = 0
Loop:   beq     $t0, $a1, End     # if(i == N) goto End;
        Add     $t1, $t0, $t0     # 2*i
        Add     $t1, $t1, $t1     # 2*(2*i) = 4*i
        Add     $t1, $t1, $a0     # 4*i + base(A)
        Lw      $t2, 0($t1)       # load A[i]
        Add     $v0, $v0, $t2     # sum = sum + A[i]
        Addi    $t0, $t0, 1       # i++
        J       Loop              # goto Loop;
End:    jr      $ra               # return
```

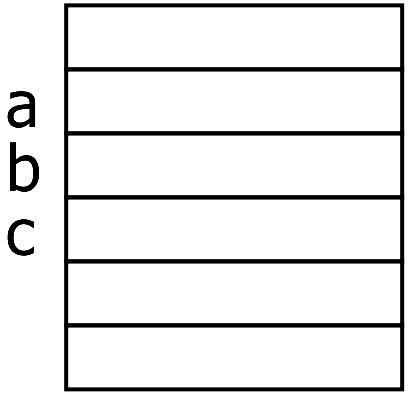
From High-Level to *Assembly*

- Global variables:
  - Stored in memory
  - For each use of a global variable the compiler needs to generate a *load/store*

```
int a, b, c;

... fun(...) {
   ...
}
```

Memory



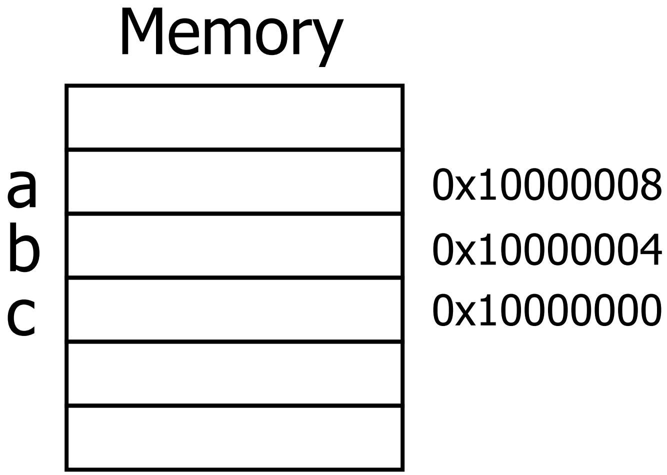
Global Variables

```
Int a, b, c;

void fun() {
   c = a + b;
}
```

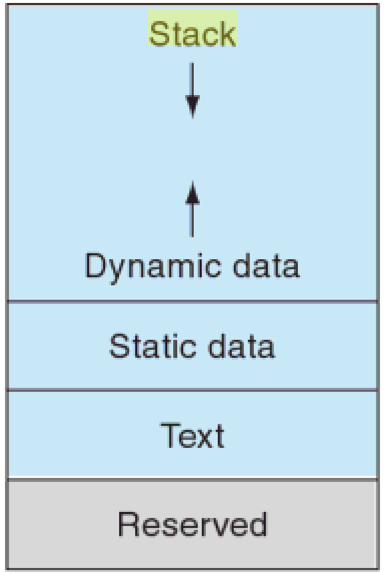
```
.data      0x10000000
a:         .space    4
b:         .space    4
c:         .space    4
           .text
fun:       …
           la        $t1, a
           lw        $t1, 0($t1)
           la        $t2, b
           lw        $t2, 0($t2)
           add       $t3, $t2, $t1
           la        $t4, c
           sw        $t3, 0($t4)
           …
```

Memory Allocation

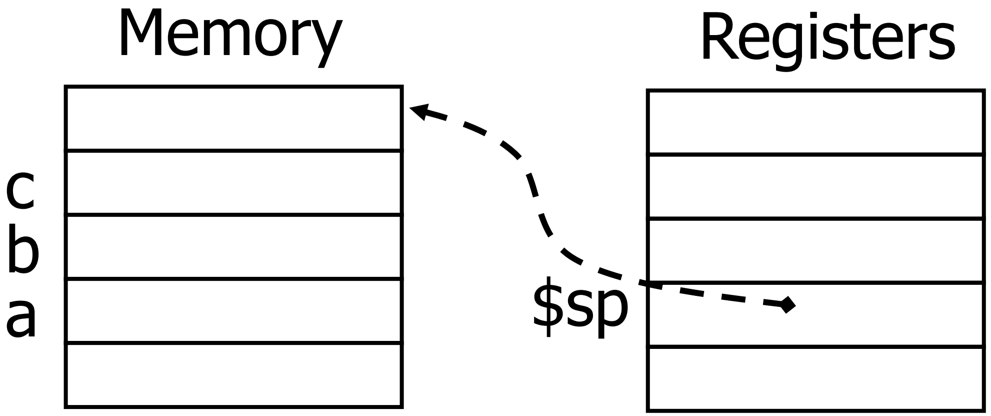


From High-Level to *Assembly*

- Procedure invocations
  - Each procedure has states
    - Local variables
    - Returning address
  - State is stored in the memory region known by call stack (there is a register with the address of the current stack position)
- Call stack
  - It is on the top of the memory
  - The stack grows from the top to the bottom



```
void fun() {
    int a, b, c;
    ...
    c = a + b;
    ...
}
```

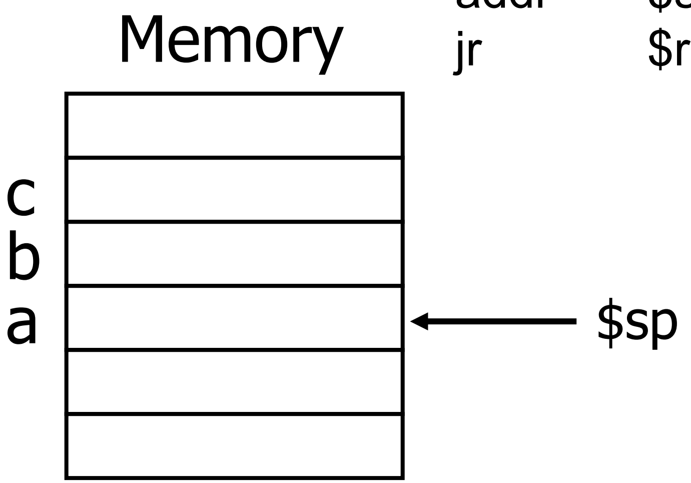


Local Variables

- Example:

```
fun:    addi    $sp, $sp, -12       Reserve stack space
        …
        lw      $t1, 0($sp)         Load a
        lw      $t2, 4($sp)         Load b
        add     $t3, $t2, $t1       a + b
        sw      $t3, 8($sp)         store c
        …
        addi    $sp, $sp, 12        Free stack space
        jr      $ra
```

```
void fun() {
    int a, b, c;
    …
    c = a + b;
    …
}
```



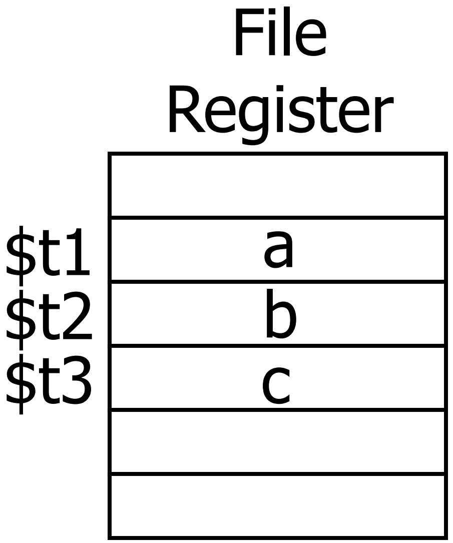
Local Variables

- Access to the internal microprocessor registers is much faster
  - But they are in limited number!
  - Thus, not all local variables can be stored in internal registers!
- In the past the assignment of variables to internal registers was done/guided by the programmer:
  - The C language includes a keyword to assign variables to internal registers: **register** (*e.g.*, register int c;)
- Today compilers are much more efficient
  - The assignment of variables to internal registers is fully delegated to them

Local Variables

- Using registers from the File Register of the microprocessor

```
void fun() {
    int a, b, c;
    ...
    c = a + b;
    ...
}
```



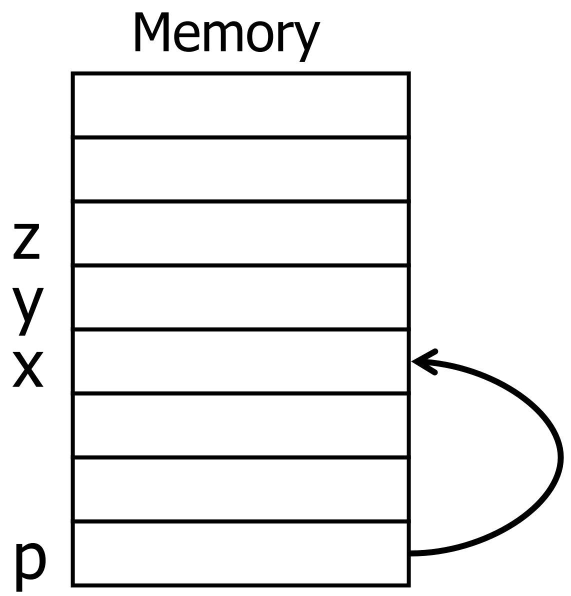
```
fun:      …
          add      $t3, $t2, $t1
          …
          jr       $ra
```

From High-Level to *Assembly*

- Implementing structs
  - Structs consist of one or more fields
- Each struct is stored in contiguous memory locations

```
typedef struct {
  int x, y, z;
} foo;

foo *p;
```

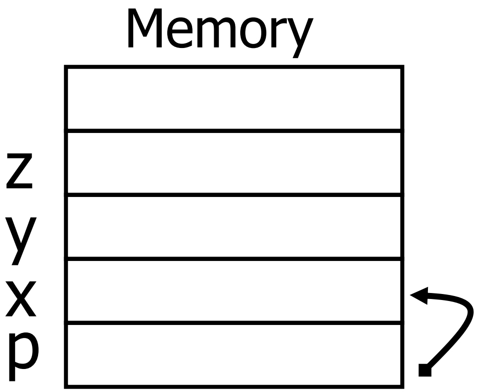


From High-Level to *Assembly*

- Example with a local struct:

```
typedef struct {
  int x, y, z;
} foo;

fun() {
  foo *p;
  p->x = p->y + p->z;
}
```



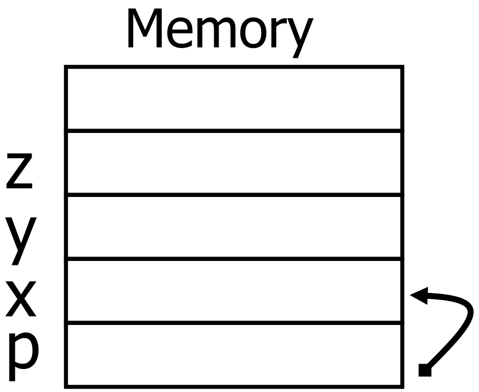
```
fun:    addi    $sp, $sp, -16       Reserve stack space
                …
        lw      $t1, 0($sp)         Address of p
        addi    $t1, $t1, 8         address p->y
        lw      $t2, 0($t1)         Load p->y
        lw      $t1, 0($sp)         Address of p
        addi    $t1, $t1, 12        address p->z
        lw      $t3, 0($t1)         Load p->z
        add     $t3, $t2, $t3       p->y + p->z
        lw      $t1, 0($sp)         Address of p
        addi    $t1, $t1, 4         address p->x
        sw      $t3, 0($t1)         store em p->x
        …
        addi    $sp, $sp, 16        Free stack space
        jr      $ra
```

From High-Level to *Assembly*

- Example with a local struct (optimized)

```
typedef struct {
  int x, y, z;
} foo;

fun() {
  foo *p;
  p->x = p->y + p->z;
}
```



```
fun:    addi    $sp, $sp, -16       Reserve stack space
        …
        lw      $t2, 8($sp)         Load p->y
        lw      $t3, 12($sp)        Load p->z
        add     $t3, $t2, $t3       p->y + p->z
        sw      $t3, 4($sp)         store em p->x
        …
        addi    $sp, $sp, 16        Free stack space
        …
```

Alignment and Packing

- Alignment requirements:
  - Integer types **int** (4 bytes) start in addresses with 2 LSBs == "00"
  - Integer types **short** (2 bytes) start in addresses with LSB == '0'
- Alignment requires:
  - Filling between fields to ensure alignment
  - Packing of fields to ensure memory savings

Alignment

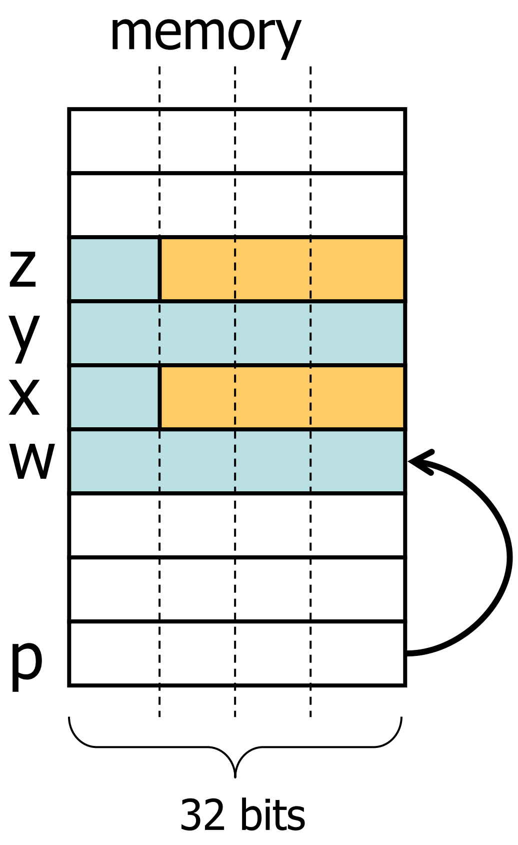
```
typedef struct {
  int w;
  char x;
  int y;
  char z;
} foo;

foo *p;
```

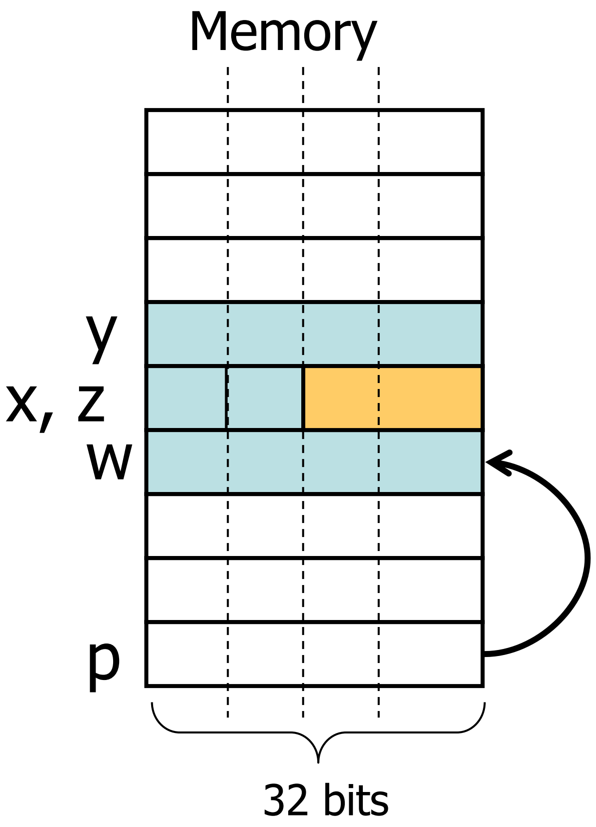
🟧 free

🟦 occupied

Naive organization

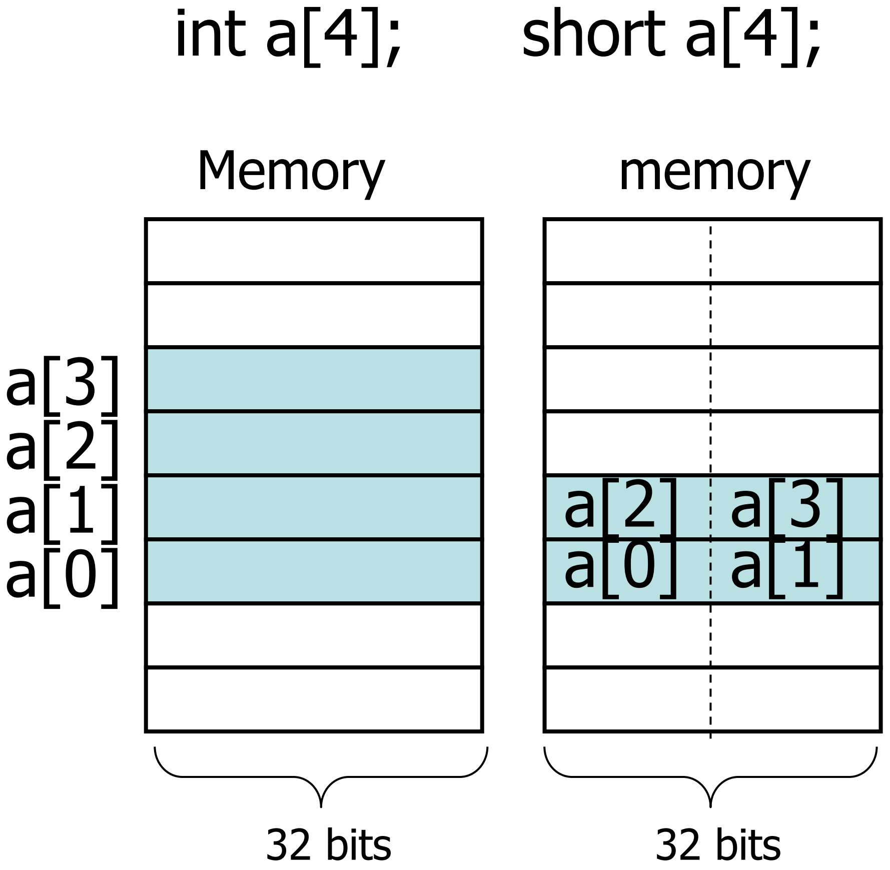


Packing (optimized)
(saves 4 bytes)



Arrays

- Assignment of memory positions for the array elements
- Elements are stored contiguously



Arrays

- Using processor registers to store the variables i and j:

```
int a[4];
proc() {
  int i, j;
  …
  i = a[j];
  …
}
```

```
        .data
A:      .space 16
        .text
Proc:

        …
        la      $t0, A
        addi    $t2, $0, 4
        mult    $t1, $t2
        mflo    $t2
        add     $t3, $t2, $t0
        lw      $t4, 0($t3)
```

Address of a[j] = address of a[0] + (4 × j) = a + (4 × j)

Expressions

- a = b * c + d – e;
- a assigned to $t4; b to $t0; c to $t1; d to $t2; e to $t3

```
...
mult    $t0, $t1
mflo    $t4
sub     $t5, $t2, $t3
add     $t4, $t4, $t5
...
```

```
...
mult    $t0, $t1
mflo    $t4
add     $t4, $t4, $t2
sub     $t4, $t4, $t3
...
```

Conditional Structures/Constructs

If(a == 1) b = 2;

- a em $t0; b em $t1

```
         ...
         addi    $t2, $0, 1
         bne     $t2, $t0, skip_if
         addi    $t1, $0, 2
Skip_if: ...
```

If(a == 1) b = 2;

else b = 1;

- a em $t0; b em $t1

```
         ...
         addi    $t2, $0, 1
         bne     $t2, $t0, else
         addi    $t1, $0, 2
         j       skip_if
Else:    addi    $t1, $0, 1
Skip_if: ...
```

Conditional Structures/Constructs

- *Branch-delay*
  - The processor always executes the instruction (or a number of instructions) following a conditional branch instruction (being the jump executed or not)
  - When it is not possible to move an instruction to just after the branch instruction a **nop** needs to be inserted

If(a == 1) b = 2;

c = a+1;

- a to $t0; b to $t1

```
         ...
         addi    $t2, $0, 1
         bne     $t2, $t0, skip_if
         addi    $t3, $t0, 1
         addi    $t1, $0, 2
Skip_if: ...
```

Loops

- The control flow (while, for, do while, etc.) is transformed in jumps:

```
int sum(int A[], int N) {
   int i, sum = 0;
   for(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
# $a0 store address A[0]
# $a1 store the value of N
# $t0 stores the value of i
# $v0 stores the value of sum
Sum:    Addi    $t0, $0, 0       # i = 0
        Addi    $v0, $0, 0       # sum = 0
Loop:   beq     $t0, $a1, End    # if(i == N) goto End;
        Add     $t1, $t0, $t0    # 2*i
        Add     $t1, $t1, $t1    # 2*(2*i) = 4*i
        Add     $t1, $t1, $a0    # 4*i + base(A)
        Lw      $t2, 0($t1)      # load A[i]
        Add     $v0, $v0, $t2    # sum = sum + A[i]
        Addi    $t0, $t0, 1      # i++
        J       Loop             # goto Loop;
End:    jr      $ra              # return
```

Loops

- Optimizations
  - Keep i and the address of a[i] in registers
  - Determine the address of a[0] before the loop body, and increment by 4 (in the case of accesses to 32-bit words) in the loop body
  - If the loop executes at least one iteration (N > 0) move branch of the loop to the end of the loop body

Loops

- Assembly code after optimizations:

```
Int sum(int A[], int N) {
   Int i, sum = 0;
   For(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
# $a0 store the address of A[0]
# $a1 store the value of N
# $t0 stores the value of i
# $v0 stores the value of sum
Sum:     addi     $t0, $0, 0        # i = 0
         addi     $v0, $0, 0        # sum = 0
         slt      $t0, $a1, 1       # check  if iter > 0
          bne     $t0, $0, End      # if so skip loop
Loop:    Lw       $t2, 0($a0)       # load A[i]
         Add      $v0, $v0, $t2     # sum = sum + A[i]
         addi     $a0, $a0, 4       # add 4 to address
         Addi     $t0, $t0, 1       # i++
         bne      $t0, $a1, Loop    # if(i != N) goto Loop;
End:     jr       $ra               # return
```

Loops

- Assembly code after optimizations (considering that N>0):

```
Int sum(int A[], int N) {
  Int i, sum = 0;
  For(i=0; i<N; i++) {
    sum = sum + A[i];
  }
  return sum;
}
```

```
# $a0 store the address of A[0]
# $a1 store the value of N
# $t0 stores the value of i
# $v0 stores the value of sum
Sum:    Addi    $t0, $0, 0         # i = 0
        Addi    $v0, $0, 0         # sum = 0
Loop:   Lw      $t2, 0($a0)        # load A[i]
        Add     $v0, $v0, $t2      # sum = sum + A[i]
        addi    $a0, $a0, 4        # add 4 to address
        Addi    $t0, $t0, 1        # i++
        bne     $t0, $a1, Loop     # if(i != N) goto Loop;
End:    jr      $ra                # return
```

Loops

- Most processors have now SIMD (Single Instruction Multiple Data) units
  - Compilers include loop vectorization to take advantage of the SIMD units

Topic introduced in the lecture and to be revisited during the course...

Let’s see what a real compiler does

- gcc 5.4, –O3, target = MIPS

```c
int sum(int A[], int N) {
   int i, sum = 0;
   for(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
sum(int*, int):
        blez    $5,$L4
        sll     $5,$5,2

        addu    $5,$4,$5
        move    $2,$0
$L3:
        lw      $3,0($4)
        addiu   $4,$4,4
        bne     $5,$4,$L3
        addu    $2,$2,$3
        j       $31
        nop
$L4:
        j       $31
        move    $2,$0
```

w/ gcc –O0

```
sum(int*, int):
        addiu   $sp,$sp,-24
        sw      $fp,20($sp)
        move    $fp,$sp
        sw      $4,24($fp)
        sw      $5,28($fp)
        sw      $0,12($fp)
        sw      $0,8($fp)
$L3:
        lw      $3,8($fp)
        lw      $2,28($fp)
        nop
        slt     $2,$3,$2
        beq     $2,$0,$L2
        nop
        lw      $2,8($fp)
        nop
        sll     $2,$2,2
        lw      $3,24($fp)
        nop
        addu    $2,$3,$2
        lw      $2,0($2)
        lw      $3,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,12($fp)
        lw      $2,8($fp)
        nop
        addiu   $2,$2,1
        sw      $2,8($fp)
        b       $L3
        nop

$L2:
        lw      $2,12($fp)
        move    $sp,$fp
        lw      $fp,20($sp)
        addiu   $sp,$sp,24
        j       $31
        nop
```

Compiler Explorer: https://gcc.godbolt.org/

Let’s see what a real compiler does (cont.)

- gcc 5.4, –O3, target = MIPS

```
#define N 1000
int sum(int A[]) {
   int i, sum = 0;
   for(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

```
sum(int*):
        addiu   $5,$4,4000
        move    $2,$0
$L2:
        lw      $3,0($4)
        addiu   $4,$4,4
        bne     $5,$4,$L2
        addu    $2,$2,$3

        j       $31
        nop
```

w/ gcc –O0

```
sum(int*):
        addiu   $sp,$sp,-24
        sw      $fp,20($sp)
        move    $fp,$sp
        sw      $4,24($fp)
        sw      $0,12($fp)
        sw      $0,8($fp)
$L3:
        lw      $2,8($fp)
        nop
        slt     $2,$2,1000
        beq     $2,$0,$L2
        nop

        lw      $2,8($fp)
        nop
        sll     $2,$2,2
        lw      $3,24($fp)
        nop
        addu    $2,$3,$2
        lw      $2,0($2)
        lw      $3,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,12($fp)
        lw      $2,8($fp)
        nop
        addiu   $2,$2,1
        sw      $2,8($fp)
        b       $L3
        nop

$L2:
        lw      $2,12($fp)
        move    $sp,$fp
        lw      $fp,20($sp)
        addiu   $sp,$sp,24
        j       $31
        nop
```

Compiler Explorer: https://gcc.godbolt.org/

Procedures

- Protocol between the callee procedures and the called procedures
  - Depends on the processor
  - For MIPS:
    - Procedure expects argeuments in the registers $a0-$a3
    - Stores values to return in the registers $v0-$v1
    - Other forms of parameter passing use the call stack (e.g., whenever the number of arguments surpass the number of registers used for arguments)

Summary

- Whichs are the main responsabilities of a compiler?
  - **Hide** from programmers the low-level machine concepts
  - Produce **efficient code** as fast as possible
  - Assign variables to internal registers or to memory
  - **Calculate** expressions with constants
  - **Maintain** the **original functionality**
  - **Generate instructions** and support the procedure call conventions used by the target machine
  - **Optimizations**

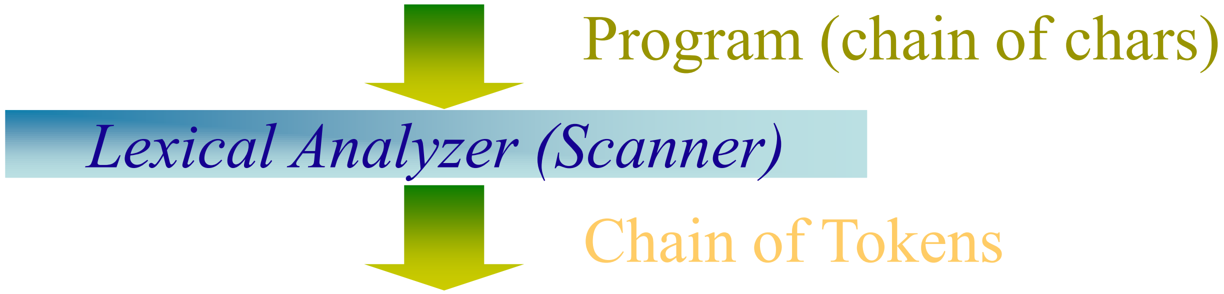
Compilers

ANATOMY OF A COMPILER

Travelling

- From the text that represents the computaions to the machine code
- Two phases:
    - Analysis
        - Recognizing the statements of the source code and storage in internal structures
    - Synthesis
        - Generation of code (e.g., assembly) from the internal structures

Lexical Analysis



Program (chain of chars)

Lexical Analyzer (Scanner)

Chain of Tokens

Lexical Analysis

```
/* a simple expression*/
y = b*x +c; // assign to y
```

Lexical Analyzer (Scanner)

```
ID(y)  EQ  ID(b)  TIMES  ID(x)
PLUS  ID(c)  SEMICOLON
EOF
```

Lexical Analysis

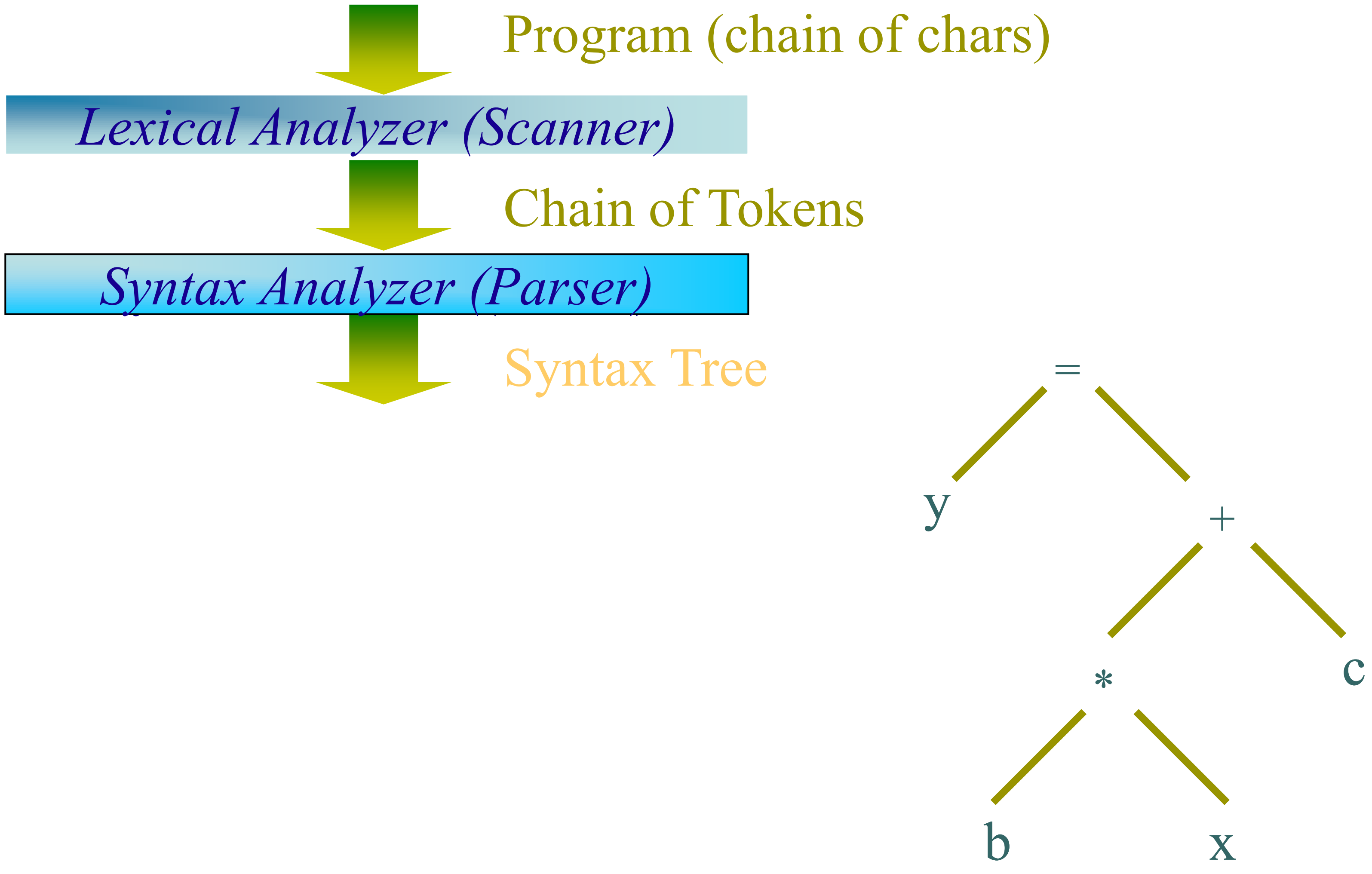
- Code recovering

```
/* exemplo
Int sum(int A[], int N) {
   Int i, 5sum = 0;
   For(i=0; i<N; i++) {
      sum = sum + A[i];
   }
   return sum;
}
```

It misses a */

It is neither a reserved word nor an identifier

Syntactic Analysis



Program (chain of chars)

Lexical Analyzer (Scanner)

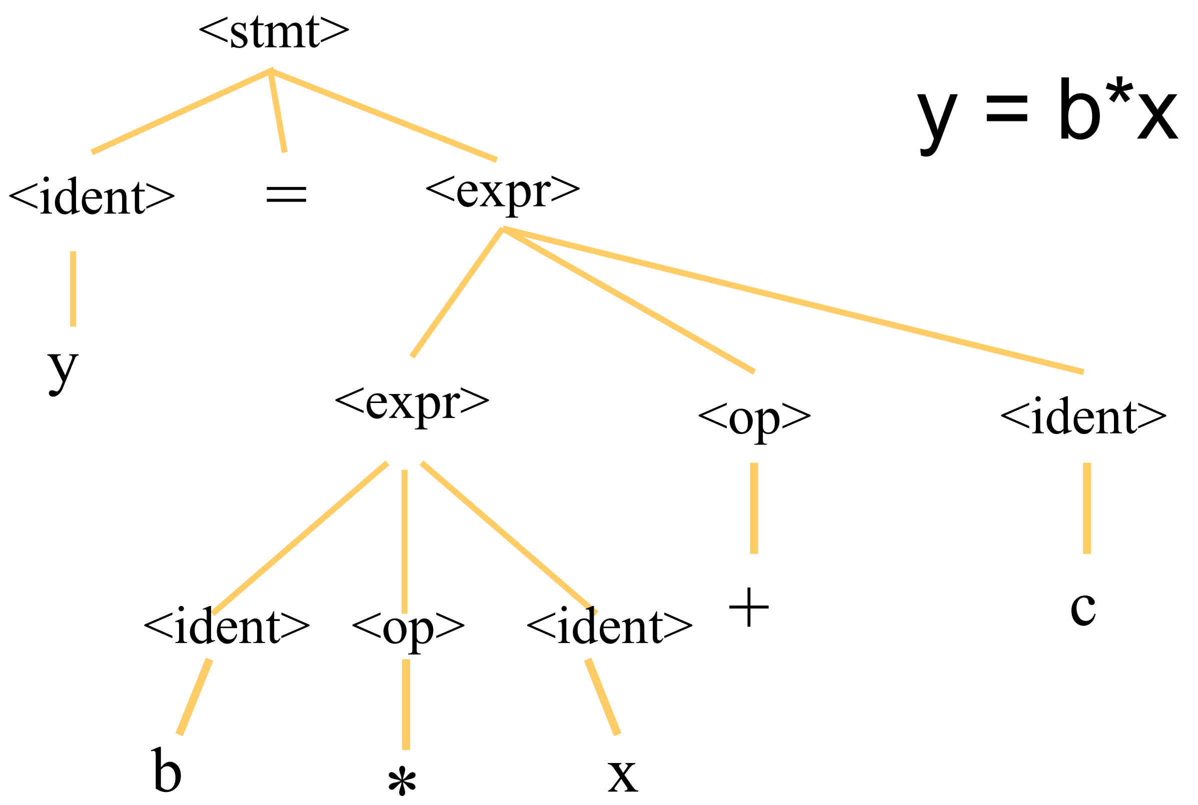
Chain of Tokens

Syntax Analyzer (Parser)

Syntax Tree

Syntactic Analysis

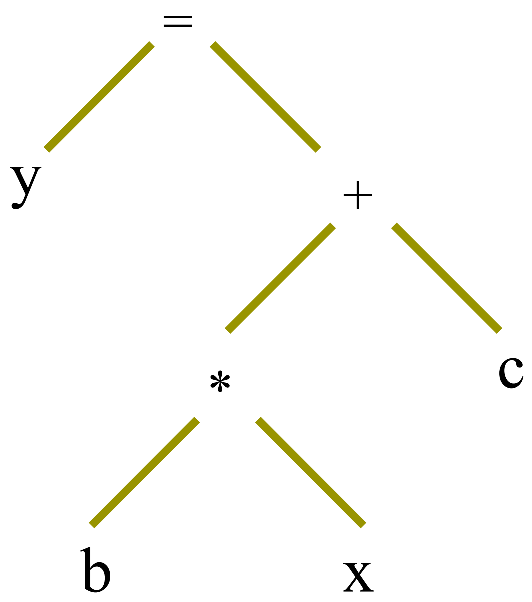
- Syntax Tree (concrete)



y = b*x + c

Syntactic Analysis

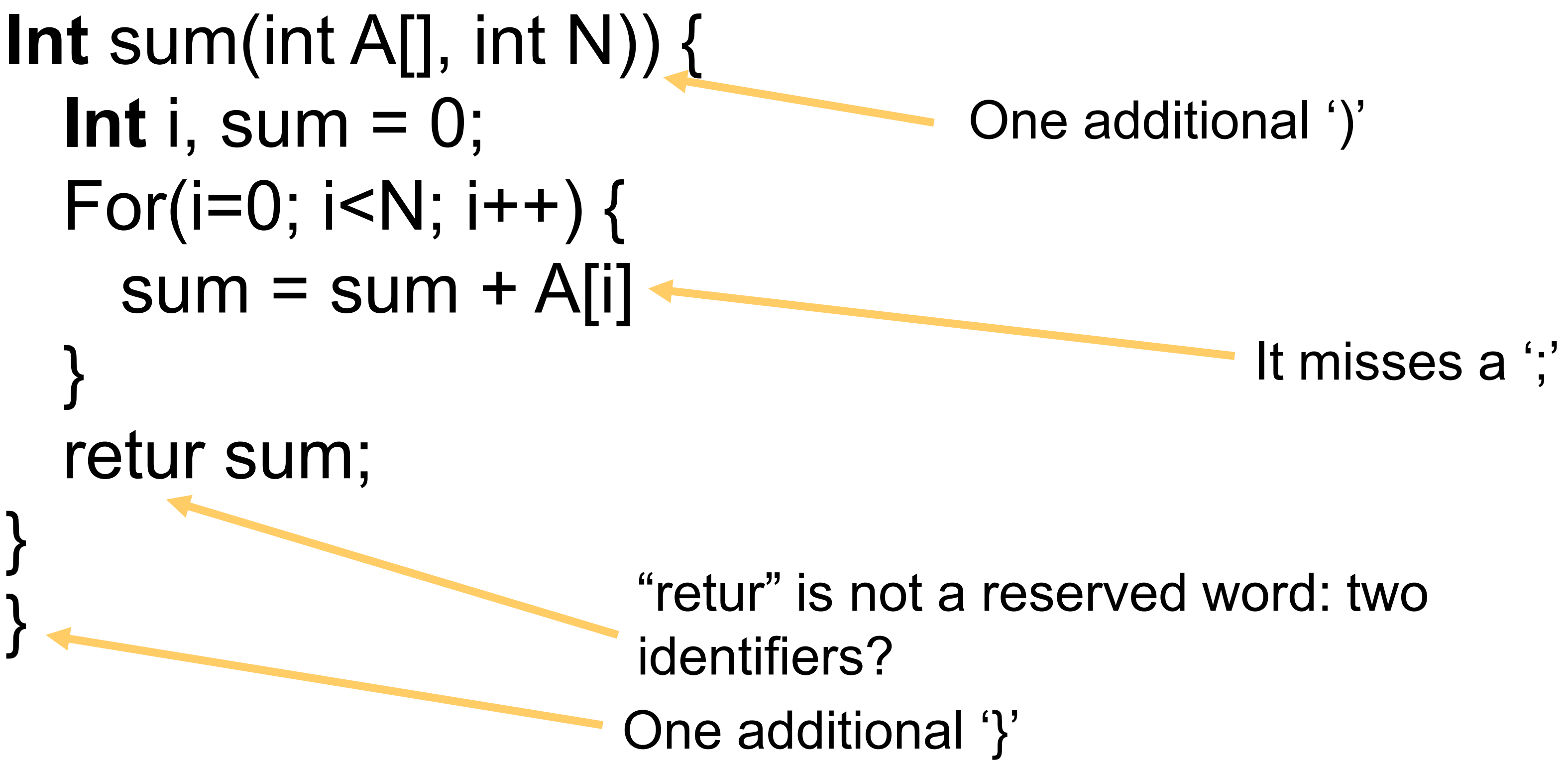
- Syntax Tree (abstract): AST



y = b*x + c

Syntactic Analysis

- Error recovering



```
Int sum(int A[], int N)) {
  Int i, sum = 0;
  For(i=0; i<N; i++) {
    sum = sum + A[i]
  }
  retur sum;
}
}
```

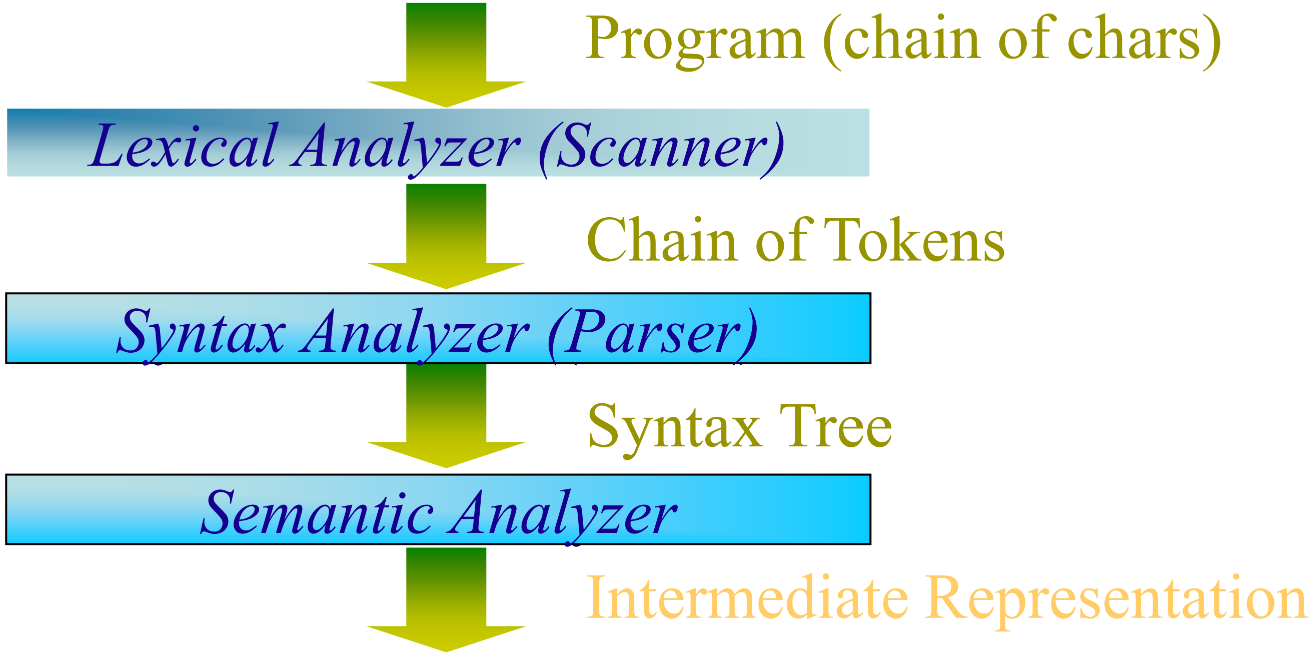
One additional ‘)’

It misses a ‘;’

“retur” is not a reserved word: two identifiers?

One additional ‘}’

Semantic Analysis



Program (chain of chars)

Lexical Analyzer (Scanner)

Chain of Tokens

Syntax Analyzer (Parser)

Syntax Tree

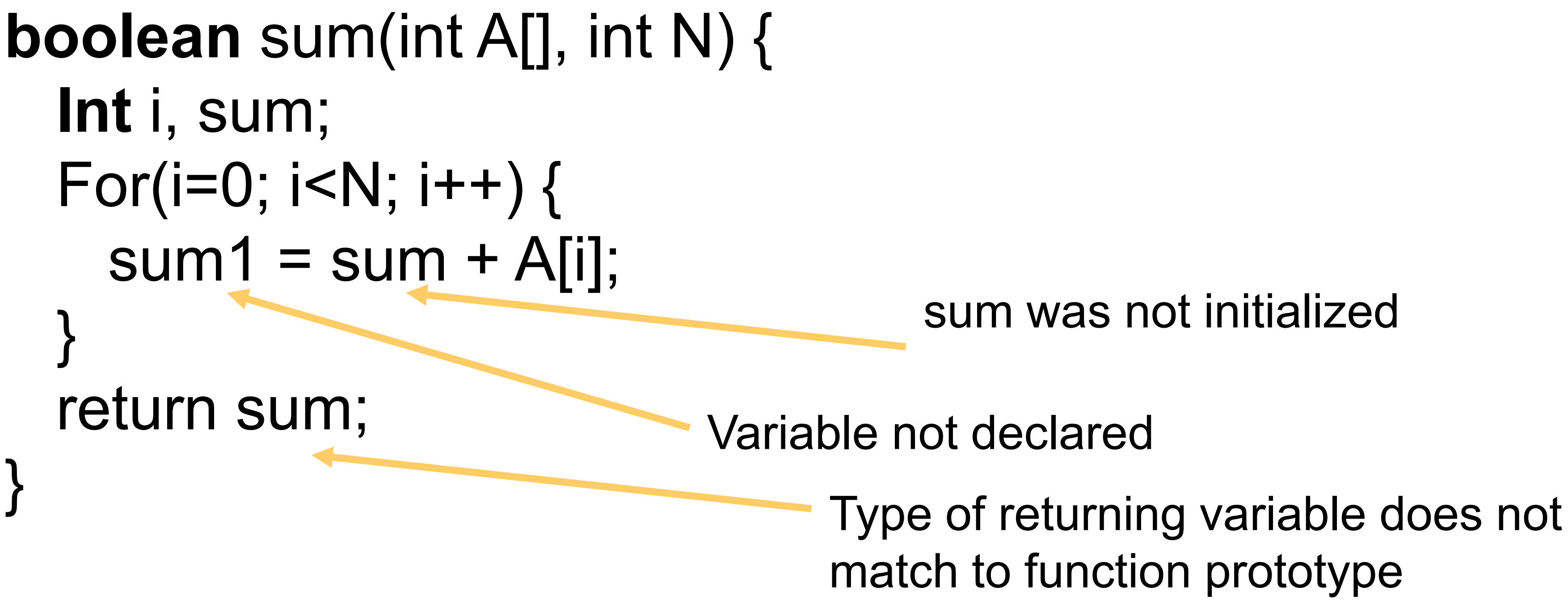
Semantic Analyzer

Intermediate Representation

```
tmp1 = b*x;
tmp2 = tmp1 + c;
```

Semantic Analysis

- Error identification



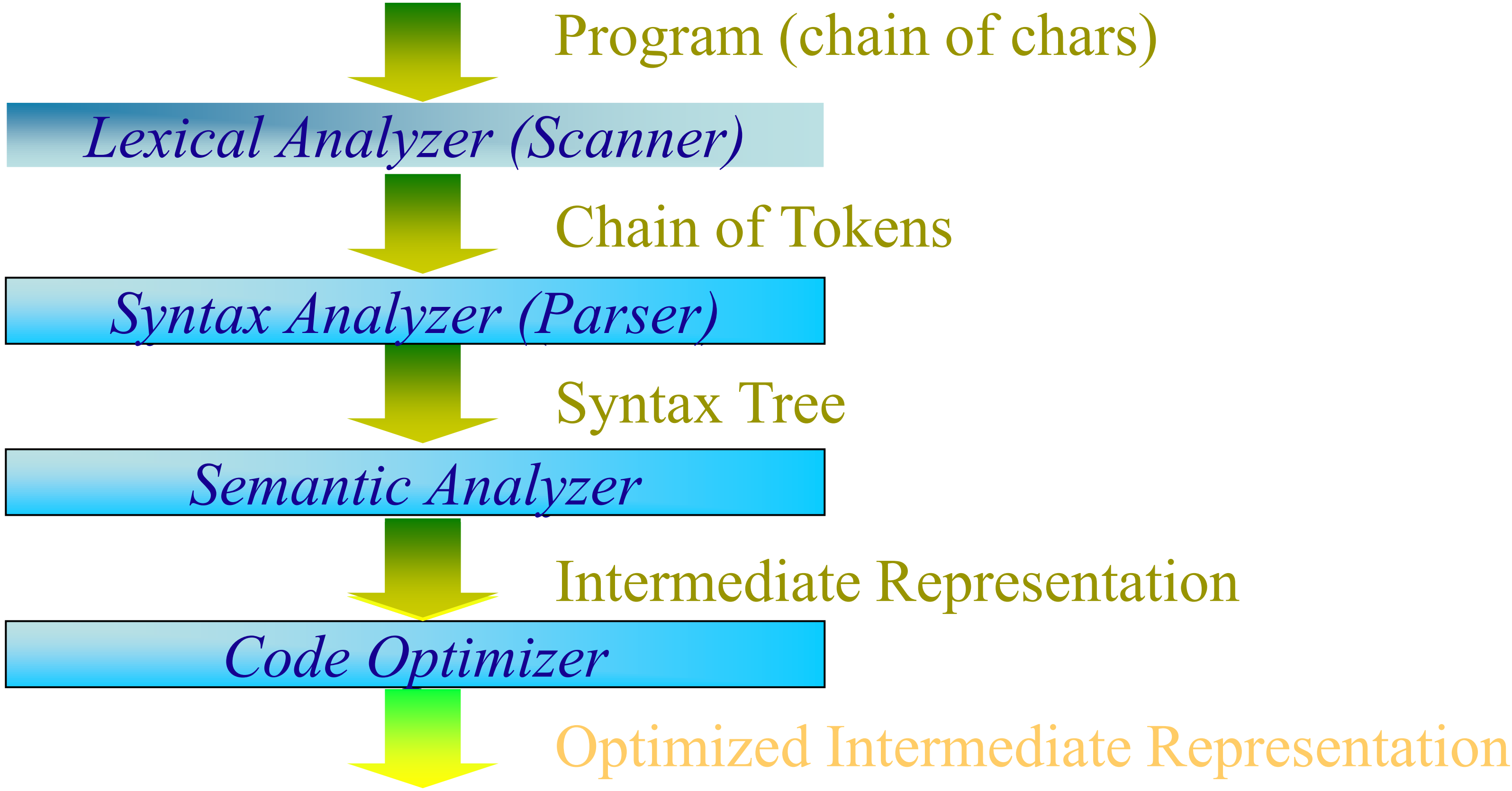
```
boolean sum(int A[], int N) {
   Int i, sum;
   For(i=0; i<N; i++) {
      sum1 = sum + A[i];
   }
   return sum;
}
```

sum was not initialized

Variable not declared

Type of returning variable does not match to function prototype

Code Optimization



Program (chain of chars)

Lexical Analyzer (Scanner)

Chain of Tokens

Syntax Analyzer (Parser)

Syntax Tree

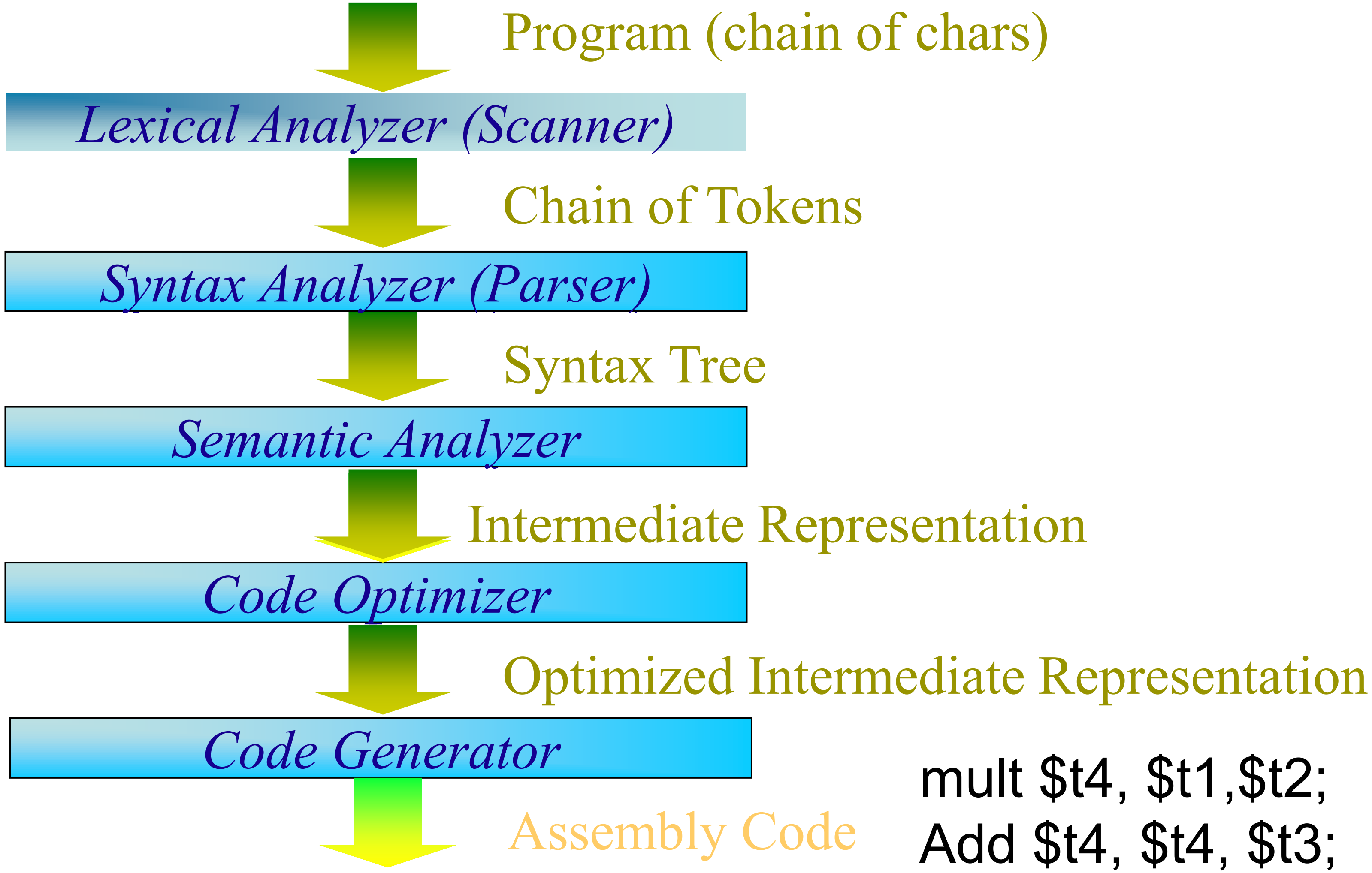
Semantic Analyzer

Intermediate Representation

Code Optimizer

Optimized Intermediate Representation

Code generation (e.g., assembly)



```
mult $t4, $t1,$t2;
Add $t4, $t4, $t3;
```

Next Steps

- Learn the techniques for each of the compiler stages
- Understand their use, the trade-offs, and how to implement them