Compilers Course
Lecture 2: An Overview

Compilation Example :
Source code in C:

```
int sum(int a[], int n)
{
   int s = 0, i = 0;
   while (i < n) {
      s = s + a[i];
      i = i + 1;
   }
   return s;
}
```

sum() in hypothetical assembly code:

```
        .text
        .globl sum          // In: r0=a, r1=n. Out: r0=s
sum:
        mov 0, r2           // r2 := 0
        mov 0, r3           // r3 := 0
loop:
        cmp r3, r1          // flags := (r3 - r1)
        jge done            // jump if flags say >=
        shl r3, 2, r4       // r4 := r3 << 2
        load [r0+r4], r4    // r4 := MEM4[r0+r4]
        add r2, r4, r2      // r2 := r2 + r4
        add r3, 1, r3       // r3 := r3 + 1
        jmp loop
done:
        mov r2, r0          // r0 := r2
        ret
```

Q: How does a compiler translate sum from C to assembly code?

1- Lexical Analysis (Scanning)

- Input is a text file = sequence of characters = string .
- Define the set of words in this language: keywords, identifiers, constants, special symbols, comments (tokens) .
- Partition string into words = instances of tokens .
- Compute attributes for generic token classes (identifiers, constants) .

Specify tokens via regular expressions:

"int" → INT
[0-9]+ → ICON
"while" → WHILE
etc

Token stream ("word" sequence):

INT ID("sum") LPAREN INT ID("a") LBRACK RBRACK COMMA INT ID("n") RPAREN
LBRACE
INT ID("s") EQ ICON(0) COMMA ID("i") EQ ICON(0) SEMI
WHILE LPAREN ID("i") LT ID("n") RPAREN LBRACE
ID("s") EQ ID("s") PLUS ID("a") LBRACK ID("i") RBRACK SEMI
ID("i") EQ ID("i") PLUS ICON(1) SEMI
RBRACE
RETURN ID("s") SEMI
RBRACE

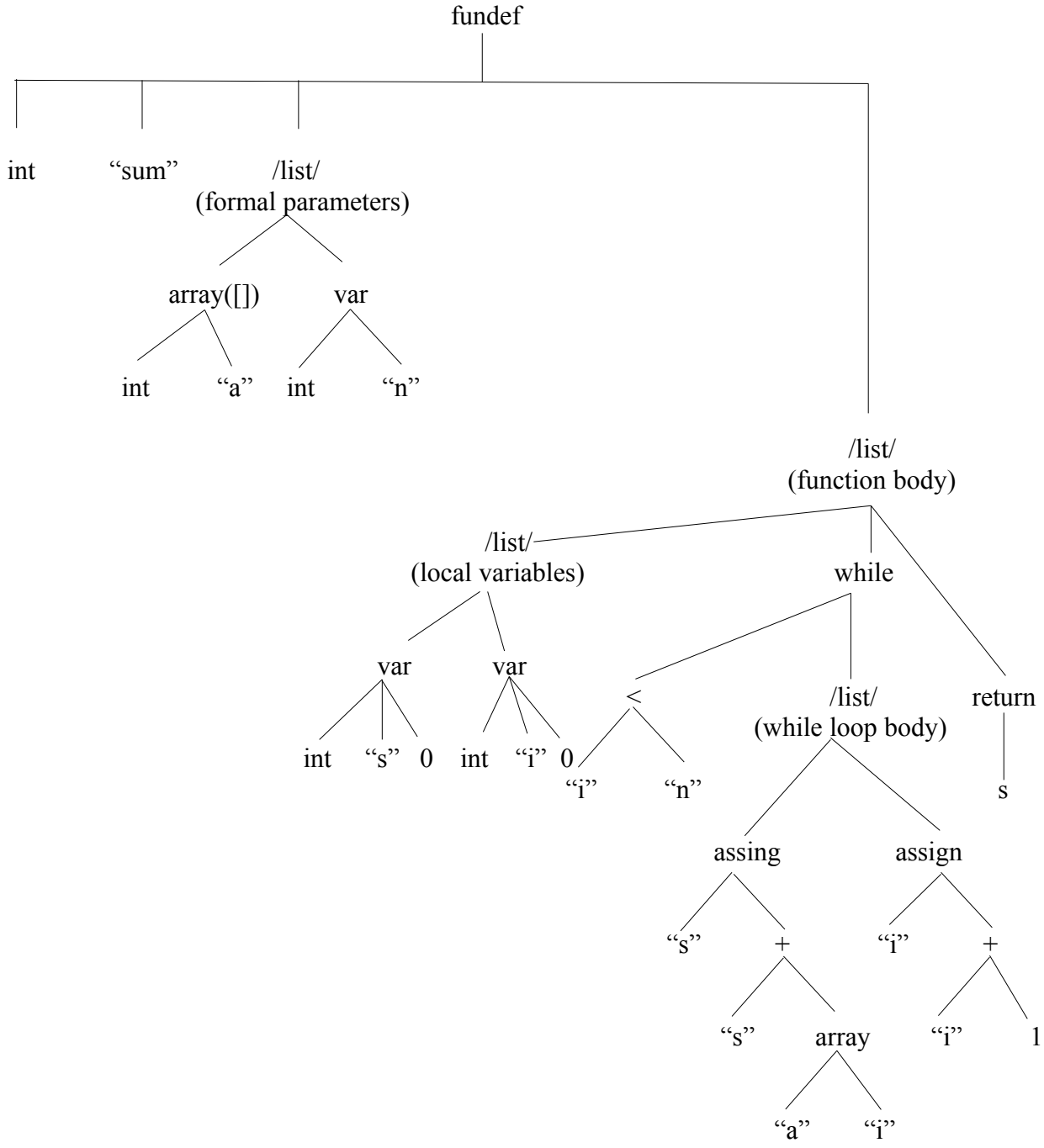
2- Syntax Analysis (Parsing)

- Input is a sequence of tokens (words) .
- Define the set of syntactic categories of the language: expressions, statements, declarations .
- Build a tree representing the syntactic categories of the input and their nesting .

Specify the syntax via Context-Free Grammar:

```
expr ::= ICON
       | ID
       | ID LBRACK expr RBRACK
       | expr op expr
op   ::= EQ | PLUS | LT
stmt ::= WHILE LPAREN expr RPAREN stmt
       | expr SEMI
       | RETURN expr SEMI
       | LBRACE stmt... RBRACE
```

Abstract Syntax Tree (AST):



3. Static Analysis (Type Checking)

- Check that each identifier is declared before being used.
- Annotate expressions with types.
- Check that subexpressions of some operator have the correct types for that operator.
- Usually via preorder traversal of the AST.
- Information about identifiers is recorded in a symbol table.

4. Generate intermediate code (IR: Intermediate Representation)

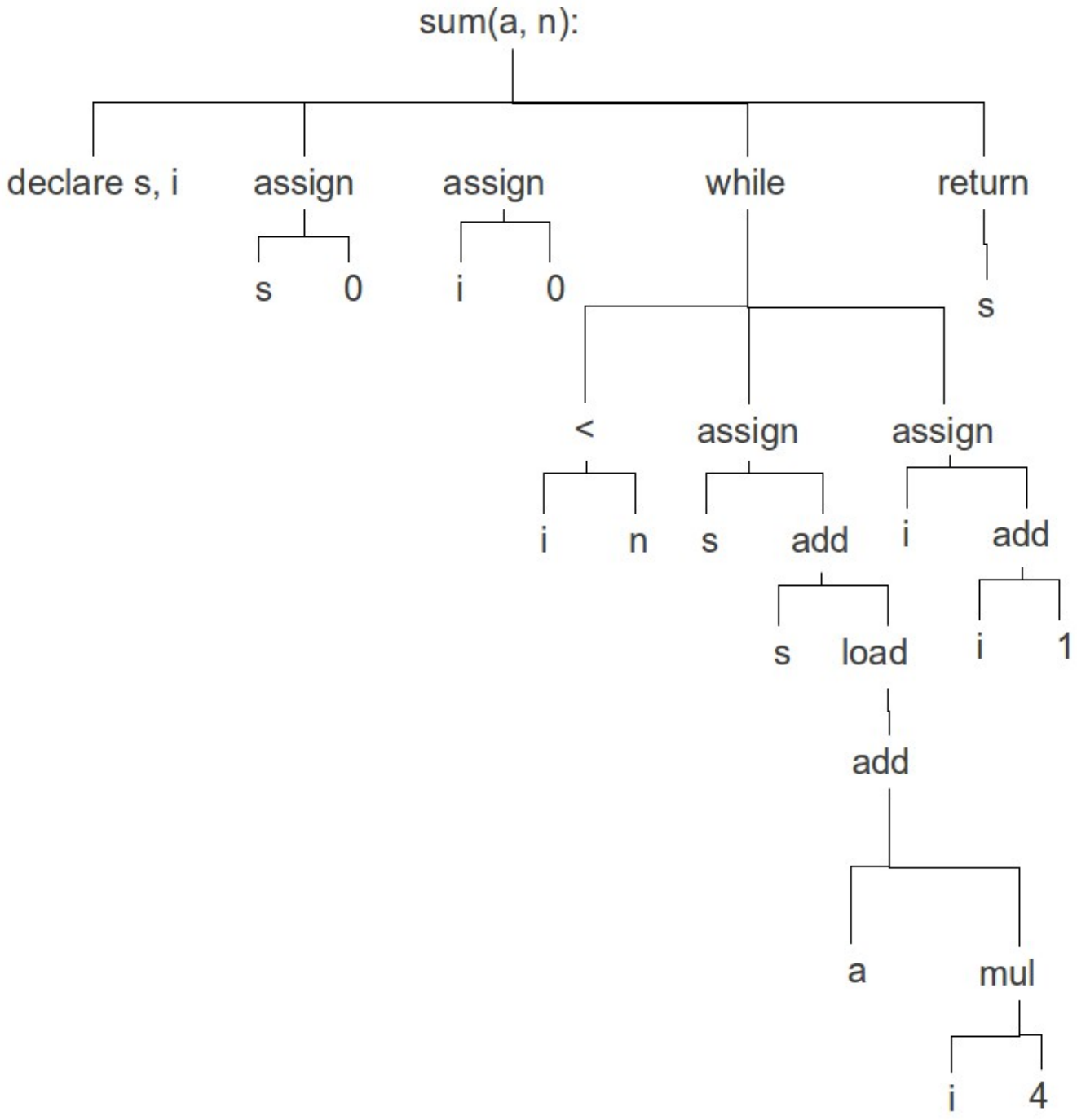
- Bridge the semantic gap between the input language and the target machine.
- Express complex operations in terms of simpler ones.
- Make implicit/hidden operations explicit.
- Flatten control structures to sequencing, jumps, and conditional jumps.
- Flatten expressions by first evaluating subexpressions and storing their results in temporary variables.
- Done via a traversal of the AST.

Sequential register-oriented code (RTL, quadruples):

```
sum(a, n):
   local s, i, t1, t2, t3, t4, t5
   s = 0
   i = 0
Lwhile:
   if i >= n goto Lnext
   t1 = i * 4
   t2 = a + t1
   t3 = load t2
   t4 = s + t3
   s = t4
   t5 = i + 1
   i = t5
   goto Lwhile
Lnext:
   return s
```

Other intermediate representations

Tree-like: more concrete/low-level version of the AST.



Reverse polish notation (RPN): virtual stack machine, popular in interpreters.

```
sum(a, n):
   local s, i
   push 0
   set s
   push 0
   set i
Lwhile:
   push i
   push n
   lt
   if-not-goto Lreturn
   push s
   push a
   push i
   push 4      } computes &a[i]
   mul
   add
   load
   add
   set s
   push i
   push 1
   add
   set i
   goto Lwhile
Lreturn:
   push s
   return
```

5. Optimisations (optional):

- Inline functions when possible and profitable.
- Eliminate unnecessary/repeated calculations.
- Simplify calculations.
- Combine calculations.

6. Instruction Selection:

- Translate the IR to target code.
- Temporaries are still symbolic.
- Do moves between symbolic temps and real registers at function entry and exit, and at recursive function calls.
- Often simple scan of the IR that expands each operation to equivalent target instructions:

```
        .text
        .globl sum
sum:
        local a, n, s, i, t1, t3, t4, t5
        mov r0, a
        mov r1, n
        mov 0, s
        mov 0, i
Lwhile:
        cmp i, n
        jge Lnext
        shl i, 2, t1
        load [a+t1], t3 // t2 = a+t1 folded into the load
        add s, t3, t4
        mov t4, s
        add i, 1, t5
        mov t5, i
        jmp Lwhile
Lnext:
        mov s, r0
        ret
```

7. Register Allocation:

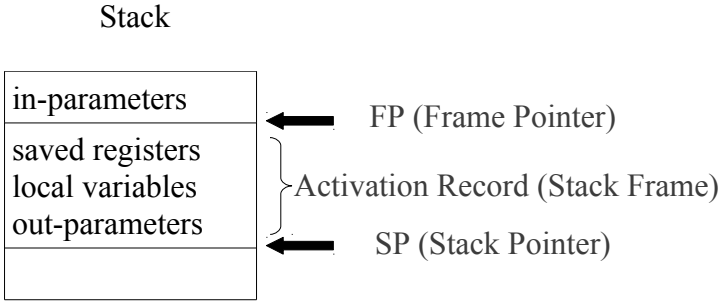
- Try to map temps onto the real registers.
- For good results requires complex analysis of the code (loop structures, liveness analysis, graph coloring).
- In the example: a = r0, n = r1, s = t4 = r2, i = t5 = r3, t1 = t3 = r4.
- Remove redundant moves (mov a reg to itself).

```
        .text
        .globl sum
sum:
        mov 0, r2
        mov 0, r3
Lwhile:
        cmp r3, r1
        jge Lnext
        shl r3, 2, r4
        load [r0+r4], r4
        add r2, r4, r2
        add r3, 1, r3
        jmp Lwhile
Lnext:
```

```
        mov r2, r0
        ret
```

8. Create Activation Records on the Stack:

- Registers are global, but a procedure's variables are local to that particular activation of that procedure.
- Stack is needed for saving registers during recursive calls.
- Also needed for storing temps that do not fit in the small set of real registers.




```
sum:
        sub SP, SUM_FRAME_SIZE, SP
        store FP, [SP+SUM_FRAME_SIZE-4]
        add SP, SUM_FRAME_SIZE, FP
        ...
Lreturn:
        load [SP+SUM_FRAME_SIZE-4], FP
        add SP, SUM_FRAME_SIZE, SP
        ret
```

Prologue

Epilogue

Rewrite references to remaining virtual registers as loads or stores to FP or SP + constant offset.

(Not needed in the example.)

Now we have final machine-specific code which can be converted to binary form and written to file or loaded into memory.

Summary

Analysis Phase:

- Partition character stream into "words" (lexical analysis, scanning).
- Recognize sentence structure (syntactic analysis, parsing) and build tree representation of it (AST: abstract syntax trees).
- Traverse AST and verify static correctness (semantic analysis, type checking).
- This is called the compiler's "front-end".

Synthesis Phase:

- Analyse the AST.
- Generate intermediate code (IR: intermediate representation):
  - Expose implicit operations (e.g. type conversions).
  - Expand complex operations.
  - Simplify/optimise the code (optional).
- Analyse IR(i), generate IR(i+1):
  - Done in 0 or more steps.
  - Typical for high-level languages (to bridge large semantic gap to machine code).
- Generate low-level code from final IR:
  - Symbolic assembly code.
  - Select machine specific instructions (e.g. MIPS, x86).
  - Assume infinite number of virtual registers (temporaries).
- Register allocation
  - Try to map the virtual registers onto a fixed finite number of physical registers.
  - Partial mapping: in general this will not succeed for all virtual registers.
- Introduce activation records:
  - The stack is a memory area where procedures create activation records at calls and remove them before returns.
  - Space for local variables and temporaries that were not assigned to physical registers.
  - Space to save registers over recursive calls.
  - Space for actual parameters in recursive calls.
- Write final machine code to file or convert it to binary form and load it into memory for execution.

Slogan 1: Stepwise simplification.
Slogan 2: Divide-and-conquer.

System view

Assembler:

- Input = assembly code (text file).
- Output = object code (binary file).

Object code:

Machine code (procedures)
Initialised data
Sybmol table
Relocation info

The symbol table describes where in the code and data segments the different procedures and variables are stored. The relocation information lists places in the code and data segments that reference external procedures or variables.

Linker:

- Input = a number of object files.
- Output = an executable file (code + data).

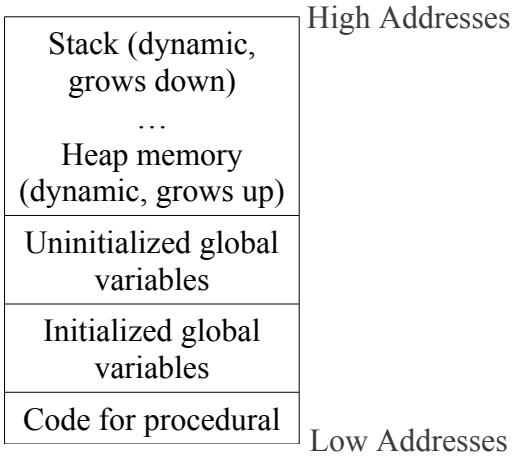
The linker combines the code segments, the data segments, and converts symbolic references to physical references.

Loader:

- Input = an executable file.
- Output = a new process.

Part of the operating system. Code and data is loaded into memory. A stack is allocated. The registers are initialized.

Typical memory layout for a process:



Registers:
SP = points to the initial stack top.
PC = points to the initial code (main).
R0 ... Rn = junk or zeros.