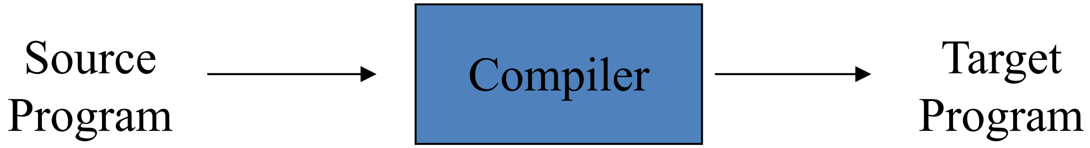
Chapter 1 introduction

What is a compiler?

A compiler is program that translates a source language into an equivalent target language



A compiler is a complex program

From 10,000 to 1,000,000 lines of codes

Compilers are used in many forms of computing

Command interpreters, interface programs

What is a compiler?

```
while (i > 3) {
  a[i] = b[i];
  i ++
}
```

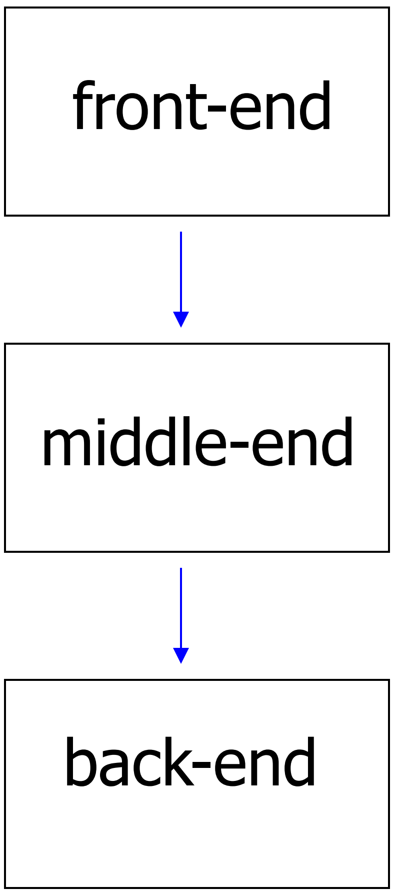
C program

compiler
does this

```
mov eax, ebx
add eax, 1
cmp eax, 3
jcc eax, edx
```

assembly
program

Compilers are complex



- text file to abstract syntax
  - lexing; parsing
- abstract syntax to intermediate form (IR)
  - type checking; analysis; optimizations;
- IR to machine code
  - code generation; register allocation; more optimization

What is the Main Content of this Course?

Describing

- Techniques
- Data structures
- Algorithm

for translating programming languages into executable code.

A Tiger language

- A small language
  - nested function
  - record values with implicit pointers
  - arrays, integer and string variables
  - a few simple structured control constructs

1.1 Modules and Interfaces

1.1 Modules and Interfaces

Two Important Concepts

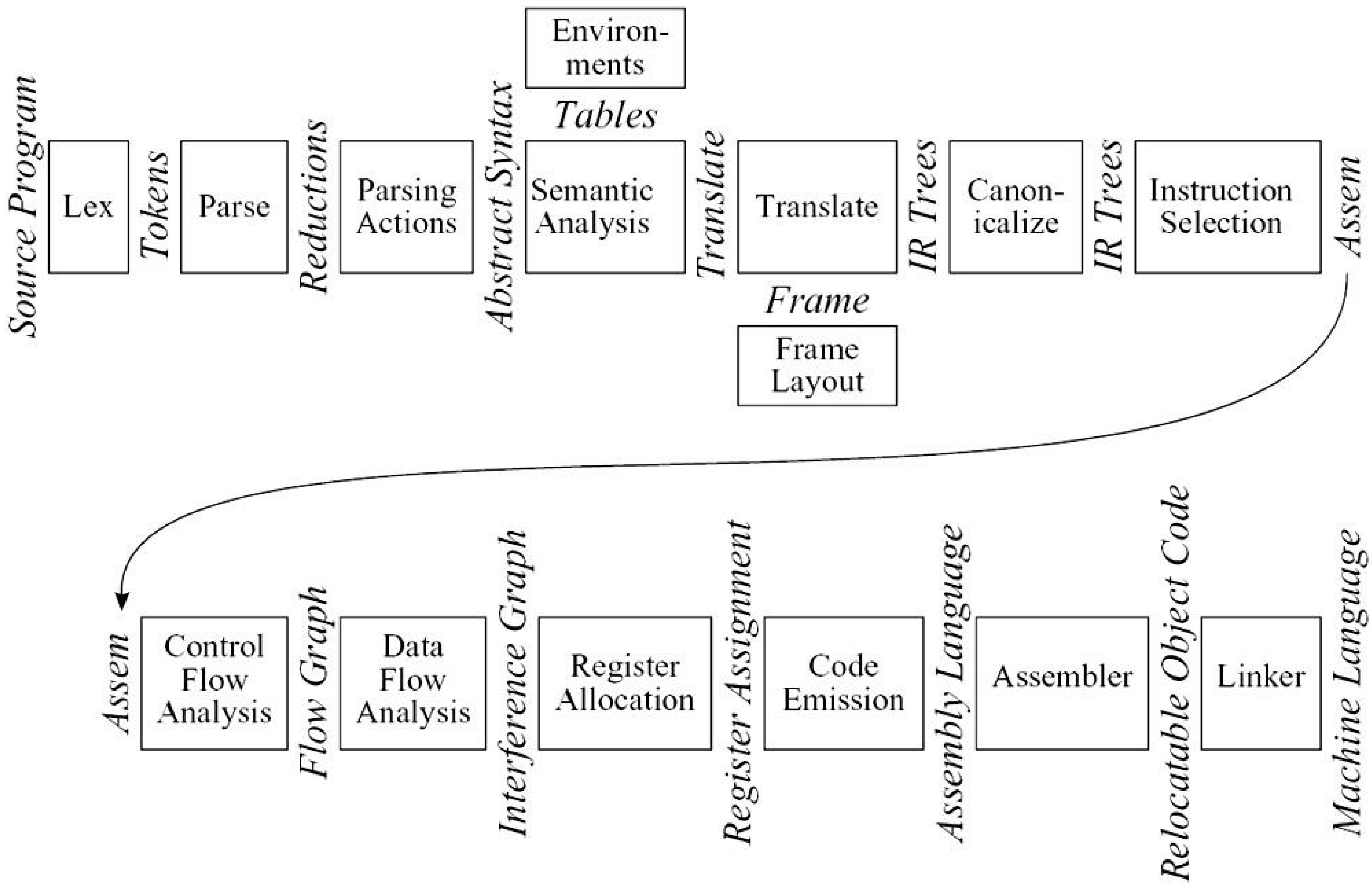
Phases: one or more modules

Operating on the different abstract "*languages*" during compiling process

Interfaces:

Describe the information exchanged between modules of the compiler

1.1 Modules and Interfaces



The phases, interfaces in a typical compiler

1.1 Modules and Interfaces

Each chapter of Part I

Table 1.2: Description of compiler phases.

Chapter	Phase	Description
2	Lex	Break the source file into individual words, or *tokens*.
3	Parse	Analyze the phrase structure of the program.
4	Semantic Actions	Build a piece of *abstract syntax tree* corresponding to each phrase.
5	Semantic Analysis	Determine what each phrase means, relate uses of variables to their definitions, check types of expressions, request translation of each phrase.
6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.

1.1 Modules and Interfaces

Each chapter of Part I

6	Frame Layout	Place variables, function-parameters, etc. into activation records (stack frames) in a machine-dependent way.
7	Translate	Produce *intermediate representation trees* (IR trees), a notation that is not tied to any particular source language or target-machine architecture.
8	Canonicalize	Hoist side effects out of expressions, and clean up conditional branches, for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a *control flow graph* that shows all the possible flows of control the program might follow when it executes.

1.1 Modules and Interfaces

Each chapter of Part I

10	Control Flow Analysis	Analyze the sequence of instructions into a *control flow graph* that shows all the possible flows of control the program might follow when it executes.
10	Dataflow Analysis	Gather information about the flow of information through variables of the program; for example, *liveness analysis* calculates the places where each program variable holds a still-needed value (is *live*).
11	Register Allocation	Choose a register to hold each of the variables and temporary values used by the program; variables not live at the same time can share the same register.
12	Code Emission	Replace the temporary names in each machine instruction with machine registers.

1.2 Tools and Software

1.2 Tools and Software

Two of the most useful abstractions:

(1) Context-free grammars for parsing

(2) Regular expressions for lexical analysis.

Two tools for compiling:

(1) Yacc converts a grammar into a parsing program

(2) Lex converts a declarative specification into a lexical analysis program

The programming project in the book can be compiled using any ANSI-standard C compiler, along with *Lex* and *Yacc.*

1.3 Data structures for tree languages

1.3 Data structures for tree languages

- The important data structures
  - trees, with several node types, each of which has different attributes.
  - Such trees can occur at many of the phase-interfaces.

1.3 Data structures for tree languages

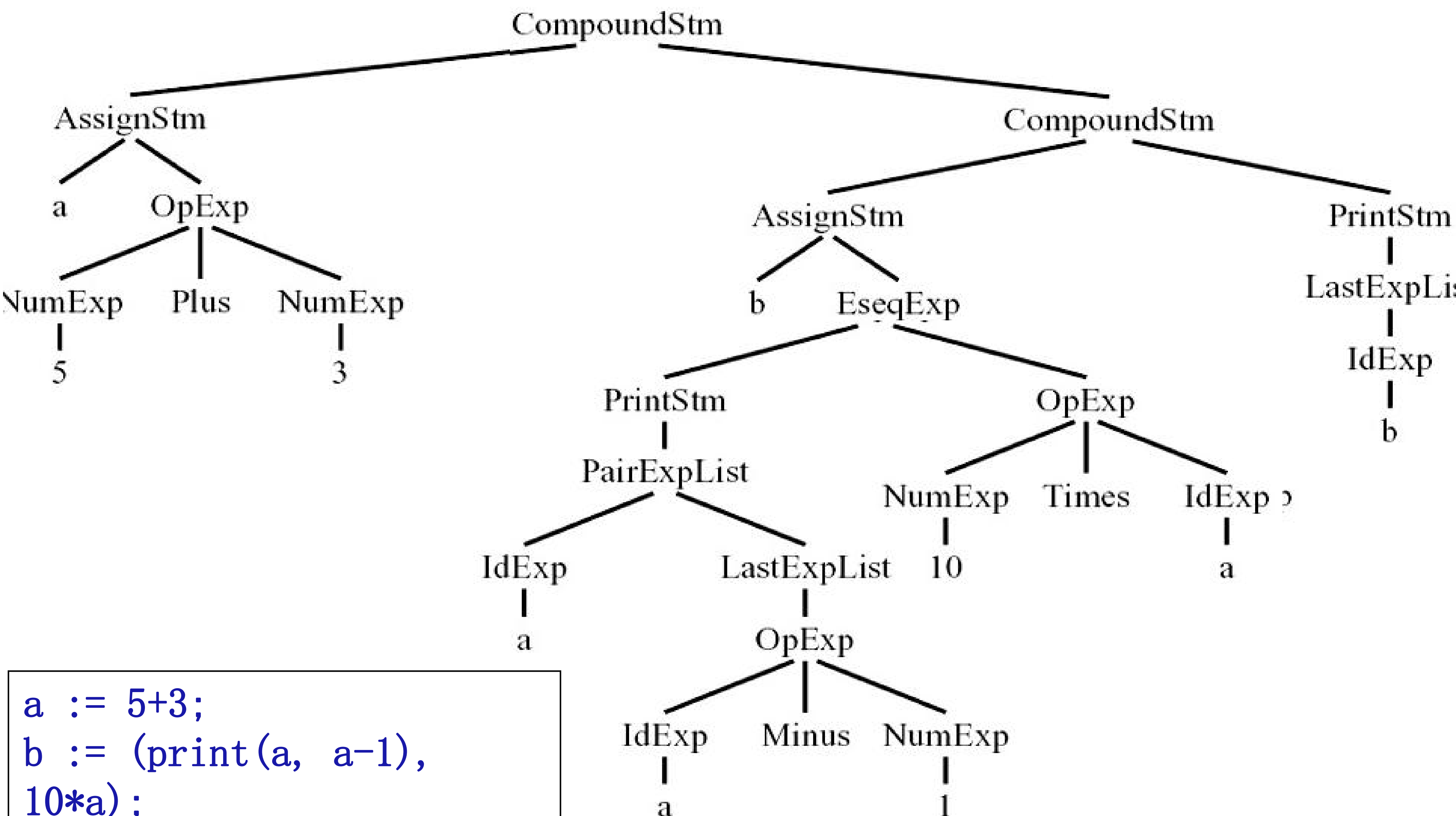
```
a := 5+3;
b := (print(a, a-1), 10
print(b)

8 7
80
```

GRAMMAR 1.3: A stra
language.

- *Stm* → *Stm* ; *Stm*
- *Stm* → id := *Exp*
- *Stm* → print (*ExpList*)
- *Exp* → id
- *Exp* → num
- *Exp* → *Exp Binop Exp*

- *Exp* → (*Stm*, *Exp*)
- *ExpList* → *Exp*, *ExpList*
- *ExpList* → *Exp*
- *Binop* →+
- *Binop* →−
- *Binop* →×
- *Binop* → /

1.3 Data structures for tree languages




```
a := 5+3;
b := (print(a, a-1),
10*a);
print(b)
```

1.3 Data structures for tree languages

Each grammar symbol can corresponds to a **typedef** in the data structures:

Grammar	Typedef
Stm	**A-stm**
Exp	**A-exp**
ExpList	**A-expList**
id	**string**
num	**int**

1.3 Data structures for tree languages

```
Typedef char *string;
Typedef struct A_stm_ *A_stm;
Typedef struct A_exp_ *A_exp;
Typedef struct A_expList_ *A_expList
Typedef enum {A_plus, A_Minus, A_times, A_div} A_binop

Struct A_stm_ { enum {A_compoundStm, A_assignStm, A_printStm} Kind；
                union { struct {A_stm stm1, stm2;} compound;
                        struct {string id; A_exp exp;} assign;
                        struct {A_expList exps;} print;
                      } u;
              }

A_stm A_CompoundStm(A_stm stm1, A_stm stm2);
A_stm A_AssignStm(string id, A_exp exp;}
A_stm A_PrintStm(A_expList exps);


…………..
```

1.3 Data structures for tree languages

One constructor :

- belongs to the union for its left-hand-side symbol

The constructor names

- indicated on the right-hand side of Grammar 1.3

Right-hand-side components

- represented in the data structures

Struct of each grammar symbol:

- A union to carry these values, and
- **A kind field** to indicate which variant of the union is valid

1.3 Data structures for tree languages

A constructor function:

- Malloc and initialize the data structure
- Such as **CompoundStm, AssignStm**, etc

```
A-stm A_CompoundStm(A_stm stm1, A_stm stm2){
        A_stm s = checked_malloc(sizeof(*s));
        s->kind = A_compoundStm;
        s->u.compound.stm1=stm1;
        s->u.compound.stm2=stm2;
        return s;
}
```

1.3 Data structures for tree languages

```
typedef struct A_stm_  *A_stm;
struct A_stm_
   { enum { A_compoundStm, A_assignStm, A_printStm} kind;
     union { struct {A_stm stm1,stm2;} compound;
              struct {string id; A_exp exp;} assign;
              struct {A_expList exps; } print;
               }  u;
};

A_stm A_CompoundStm (A_stm stm1,A_stm stm2);
```

1.3 Data structures for tree languages

Programming style

several conventions for representing tree data structures

1. Trees are described by a grammar.

2. A tree is described by one or more typedef, each corresponding to a symbol in the grammar.

3. Each typedef defines a pointer to a corresponding struct. The struct name (ends in an underscore) is never used anywhere except in the declaration of the typedef and the definition of the struct itself.

1.3 Data structures for tree languages

Programming style

4. Each struct contains a kind field(enum), and a u field(union).

   An enum showing different variants, one of each grammar rule; and a u field, which is a union.

5. There is **more than one nontrivial(value-carraying) symbol** in the right-hand side of a rule

   (example: the rule CompoundStm),

   The union has a **component that is itself a struct** comprising these values

   (example: the compound element of the A_stm union).

1.3 Data structures for tree languages

Programming style

6. There is only one nontrivial symbol in the right-hand side of a rule,
   The union will have a component that is the value (example: the num field of the A_exp union)

7. Every class will have a constructor function that initializes all the fields.
   The malloc function shall never be called directly, except in these constructor functions.

1.3 Data structures for tree languages

Programming style

8. Each module (head file) shall have a **prefix unique** to that module (example, **A_** in **Program 1.5**)

9. Typedef names(after the prefix) shall start with **lowercase letters**;
   constructor functions(after the prefix) with **uppercase**;
   enumeration atoms(after the prefix) with lowercase;
   and union variants(which have no prefix) with lowercase.

```
A_stm A_CompoundStm(A_stm stm1, A_stm
stm2){   A_stm s = checked_malloc(sizeof(*s));
         s->kind = A_compoundStm;
         s->u.compound.stm1=stm1;
         s->u.compound.stm2=stm2;
         return s;
}
```

1.3 Data structures for tree languages

Modularity principle for C programs

1. Each phase or module of the compiler belongs in its **own ".c"** file, with a corresponding **".h" file**.

2. Each module shall have **a prefix unique** to that module.
   - All global names exported by the module shall start with the prefix

3. All functions shall have prototype
   - The C compiler shall be told to warn about uses of functions without prototypes

1.3 Data structures for tree languages

Modularity principle for C programs

4. #include“util.h”

The inclusion of **assert.h** encourages the liberal use of assertion by the C programmer

5. The string type means a **heap-allocated string** that will not be modified after its initial creation.

6. C’s malloc function returns NULL if there is no memory left.
   checked_malloc (never return NULL)

7. We will **never call free**.

The end