Compiler Principle

Prof. Dongming LU

Feb. 26th, 2024

Content

1. **INTRODUCTION**

2. LEXICAL ANALYSIS
3. PARSING
4. ABSTRACT SYNTAX
5. SEMANTIC ANALYSIS

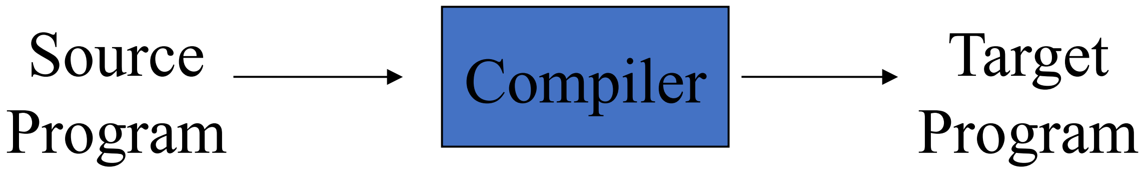
6. ACTIVATION RECORD
7. TRANSLATING INTO INTERMEDIATE CODE

8. OTHERS

1 Introduction

What is a compiler?

- A compiler is a program to translates one language to another



- 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

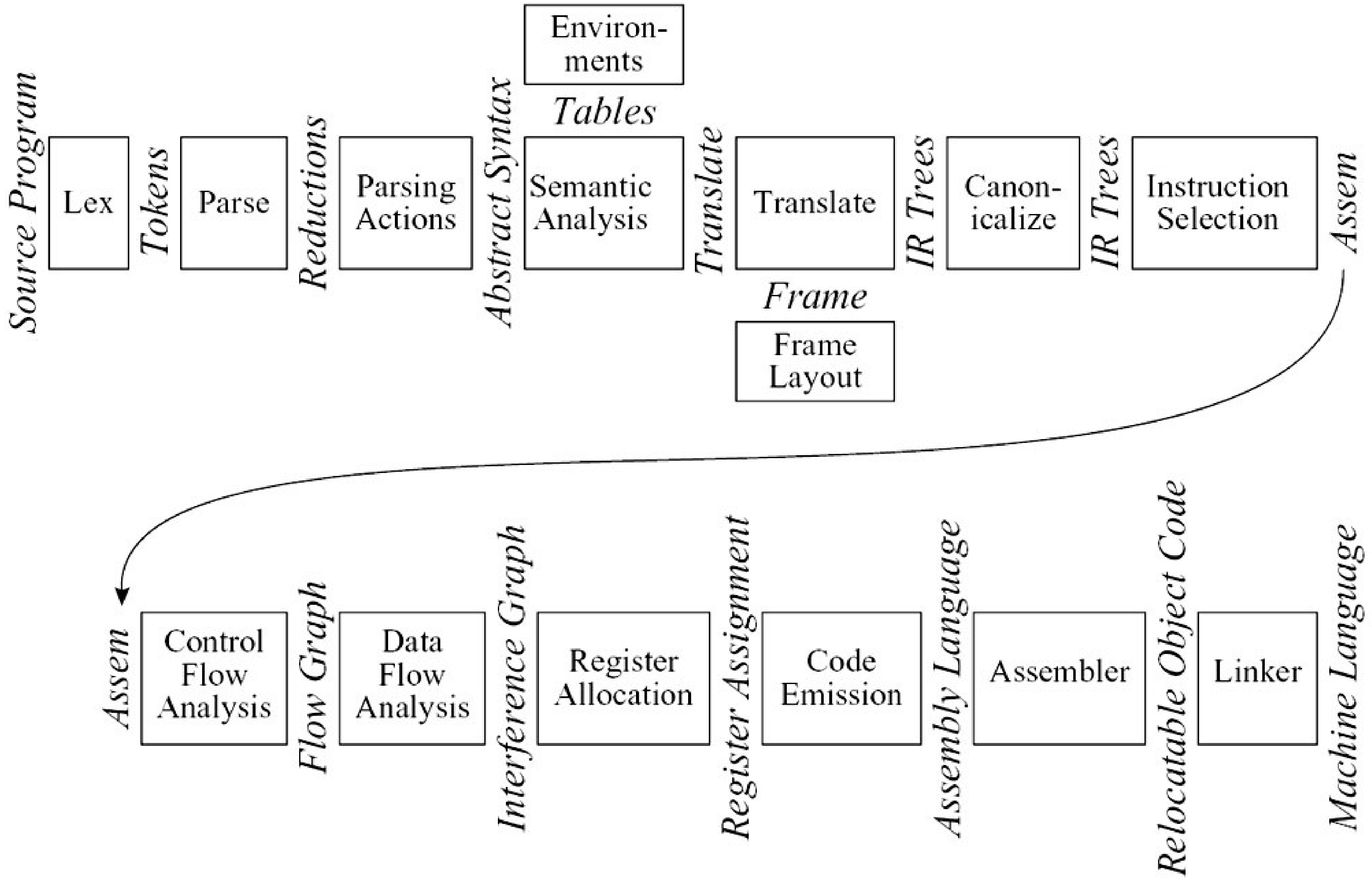
What will be discussed in this course?

Describing

- **Techniques**
- **Data structures**
- **Algorithms**

for translating programming languages into executable code.

A Real program language Tiger: Simple and Nontrivial



The phases, interfaces in a typical compiler

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

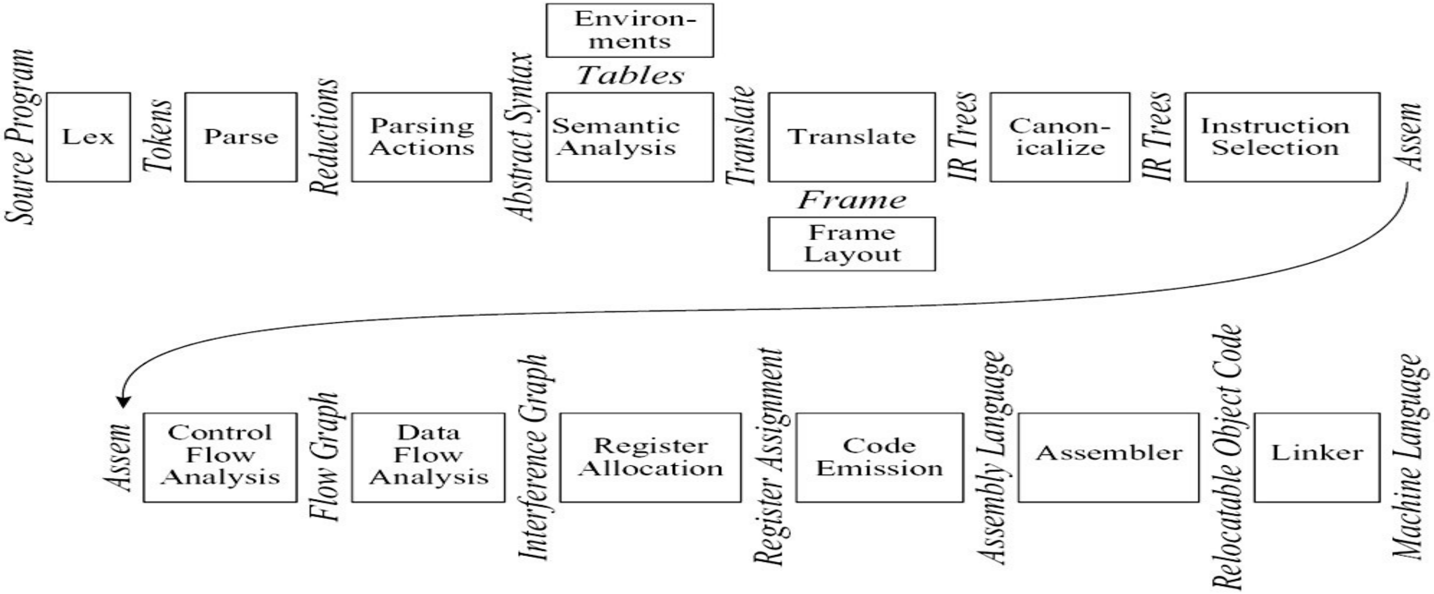
1.1 Modules and Interfaces

Modules

Role: implementing each phase

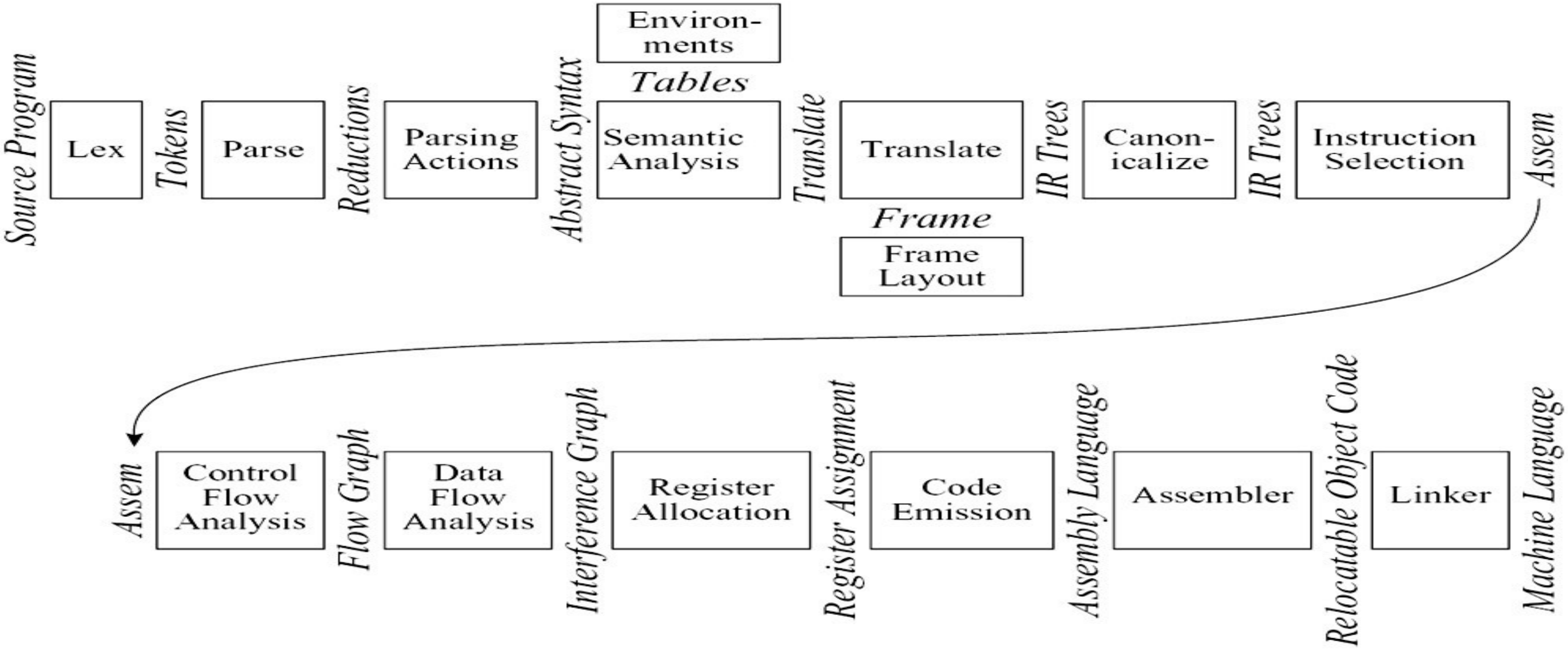
Advantage: allowing for reuse of the components

- **Changing the target-machine**
- **Changing the source language**



Interfaces

- The **data structures**: ***Abstract Syntax, IR Trees*** **and *Assem*.**
- A **set of functions**: **The *translate* interface**
  - ✓ A **function called by parser: The *token* interface**



Phases

- 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	Parsing Actions	Build a piece of *abstract syntax tree* corresponding to each phrase

Phases

- Description of compiler phases

Chapter	Phase	Description
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.
7	Translate	Produce *intermediate representation trees* (IR trees) Not tied to any particular source language or target-machine architecture.

Phases

- Description of compiler phases

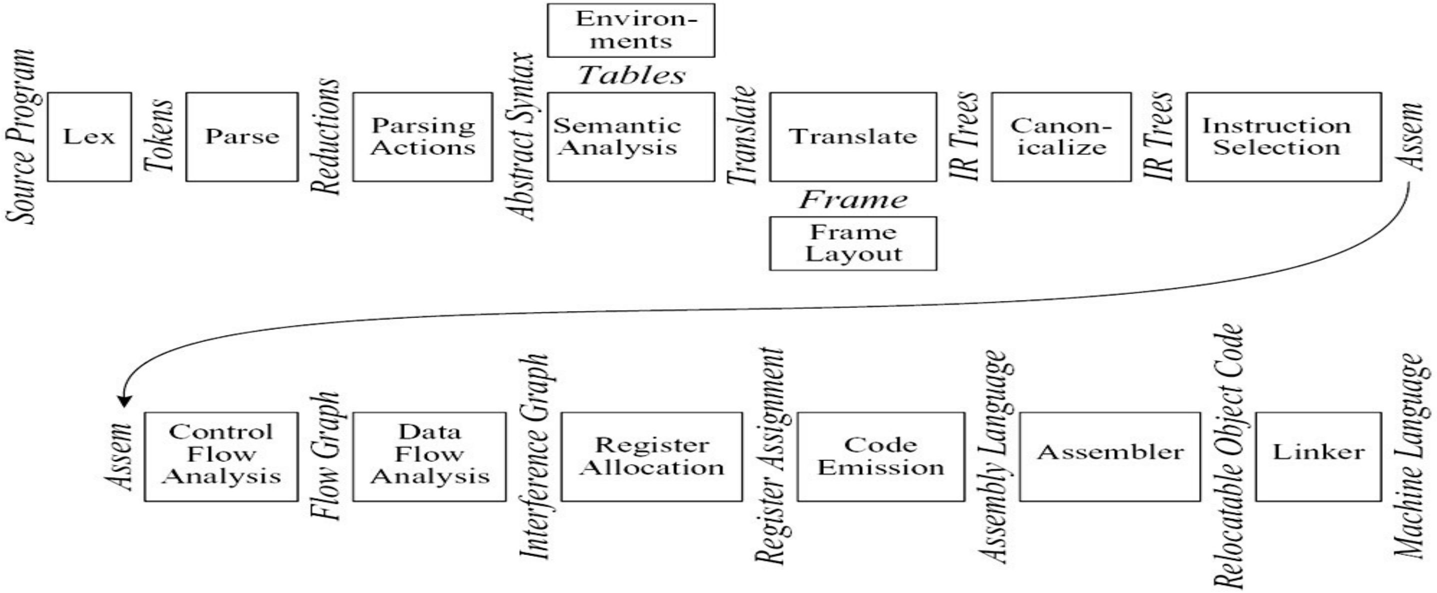
Chapter	Phase	Description
8	Canonicalize	Hoist side effects out of expressions Clean up conditional branches for the convenience of the next phases.
9	Instruction Selection	Group the IR-tree nodes into clumps that correspond to the actions of target-machine instructions.
10	Control Flow Analysis	Analyze the sequence of instructions into a *control flow graph* that shows all the possible flows of control the program might follow when it executes.

Phases

- **Description of compiler phases**

Chapter	Phase	Description
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**

Modularization



- Several modules maybe combined into one phase: Parse, Semantic Analysis, Translate, Canonicalize
- Instruction Selection maybe combined with Code Emission
- Simple compilers may omit the Control Flow Analysis, Data Flow Analysis, and Register Allocation phases

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

Intermediate Representations (IR)

The form of a compiling program

Trees Representation(TR)

- The main representation forms
- Several node types with different attributes

TR: described with grammars like programming languages

Introduce the concepts with a simple programming language

Syntax for a simple language

$Stm \rightarrow Stm;\ Stm$	(CompoundStm)
$Stm \rightarrow \mathbf{id} := Exp$	(AssignStm)
$Stm \rightarrow \mathbf{print}\ (ExpList)$	(PrintStm)
$Exp \rightarrow \mathbf{id}$	(IdExp)
$Exp \rightarrow \mathbf{num}$	(NumExp)
$Exp \rightarrow Exp\ Binop\ Exp$	(OpExp)
$Exp \rightarrow (Stm,\ Exp)$	(EseqExp)
$ExpList \rightarrow Exp,\ ExpList$	(PairExpList)
$ExpList \rightarrow Exp$	(LastExpList)
$Binop \rightarrow +$	(Plus)
$Binop \rightarrow -$	(Minus)
$Binop \rightarrow \times$	(Times)
$Binop \rightarrow /$	(Div)

Node types

GRAMMAR 1.3: A straight-line programming language.

Informal semantics of the language

Stm is a statement
Exp is an expression.

***s*1; *s*2** executes statement *s*1, then statement *s*2
i* :=*e evaluates the expression *e*, then "stores" the result in variable *i*.

print(*e*1, *e*2,…, *en*) displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

Id yields the current contents of the variable *i*
number evaluates to the named integer
***operator expression e*1 op *e*2** evaluates *e*1, then *e*2, then applies the given binary operator
***expression sequence* (*s, e*)** like the C-language "comma" operator, evaluating the statement *s* for side effects before evaluating the expression *e*.

An example of a program

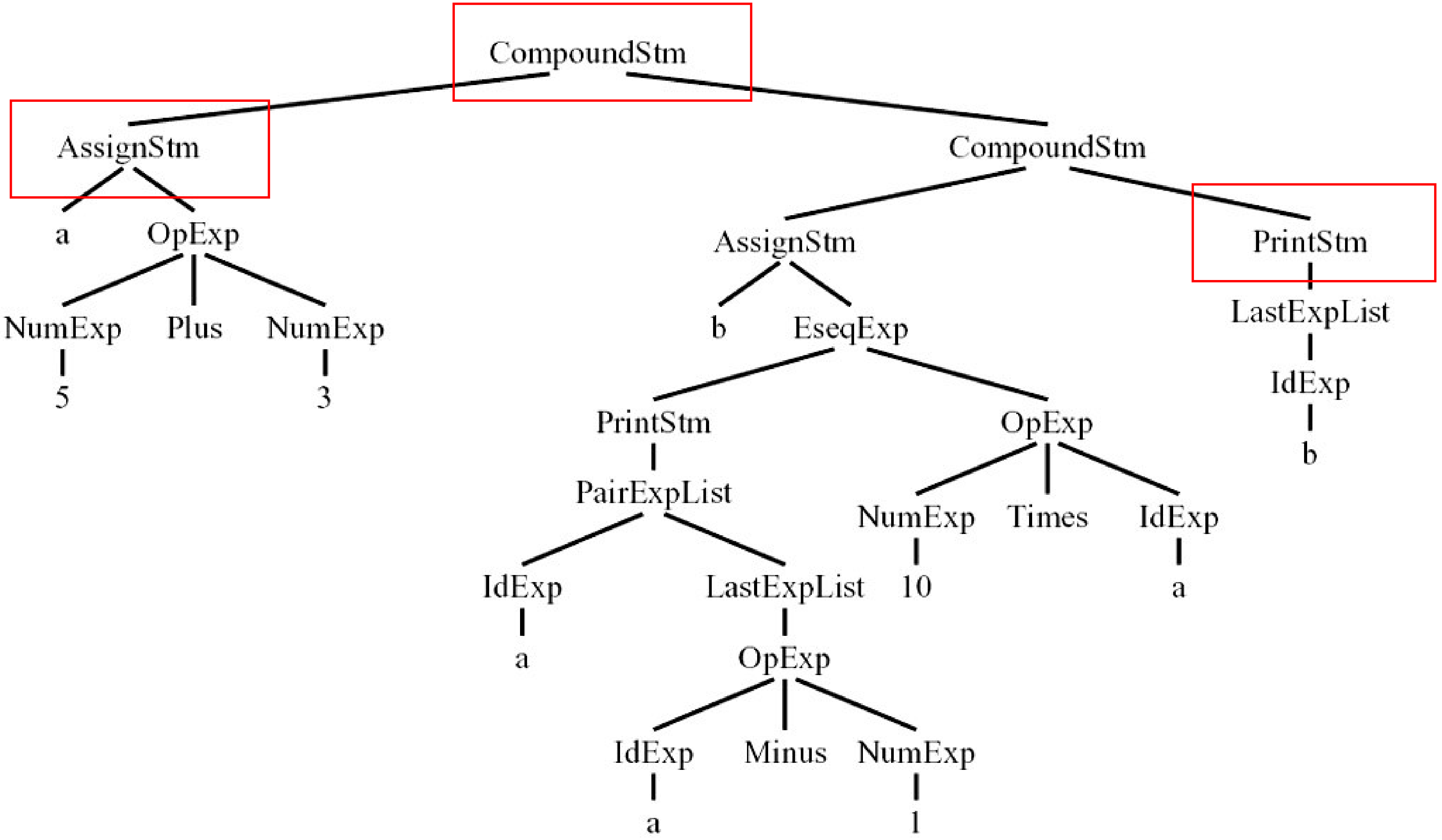
Executing the following program

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

prints

```
8 7
80
```

Tree representation of the previous program



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

Tree data structure definiton

- **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**

Data structure definition for this simple language

```
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);
```

Syntax for a simple language

Stm → *Stm*; *Stm*	(CompoundStm)
Stm → id := *Exp*	(AssignStm)
Stm → print (*ExpList*)	(PrintStm)
Exp → id	(IdExp)
Exp → num	(NumExp)
Exp → *Exp Binop Exp*	(OpExp)
Exp → (*Stm*, *Exp*)	(EseqExp)
ExpList → *Exp*, *ExpList*	(PairExpList)
ExpList → *Exp*	(LastExpList)
Binop →+	(Plus)
Binop →−	(Minus)
Binop →×	(Times)
Binop → /	(Div)

constructor names

GRAMMAR 1.3: A straight-line programming language.

One constructor for each grammar rule

The constructor names: indicated on the right-hand side of G

{

- The CompoundStm has two Stm's on the right- hand side;
  Stm → ***Stm*****;** ***Stm***

- The AssignStm has an identifier and an expression;
  Stm → **id :=** ***Exp***

}

Right-hand-side components: represented in the data structures

Struct of each grammar symbol

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

One constructor for each grammar rule

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;
}
```

One constructor for each grammar rule

Binop will be simpler.

Binop → **+**
Binop → **−**
Binop → **×**
Binop → **/**

Making a Binop struct - with union variants for Plus, Minus, Times, Div – will be overkill

- None of the variants would carry any data.

Instead, making an **enum type** A_binop.

Programming style

Several conventions for representing tree data structures in C

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, which ends in an underscore, is never used anywhere except in the declaration of the typedef and the definition of the struct itself.

Programming style

4. Each struct contains a kind fields
   - 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).

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.

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;
}
```

Modularity principle for C programs

Careful attention to modules and interfaces prevents chaos in a compiler program

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

Modularity principle for C programs

4. 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.

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

The end of Chapter 1