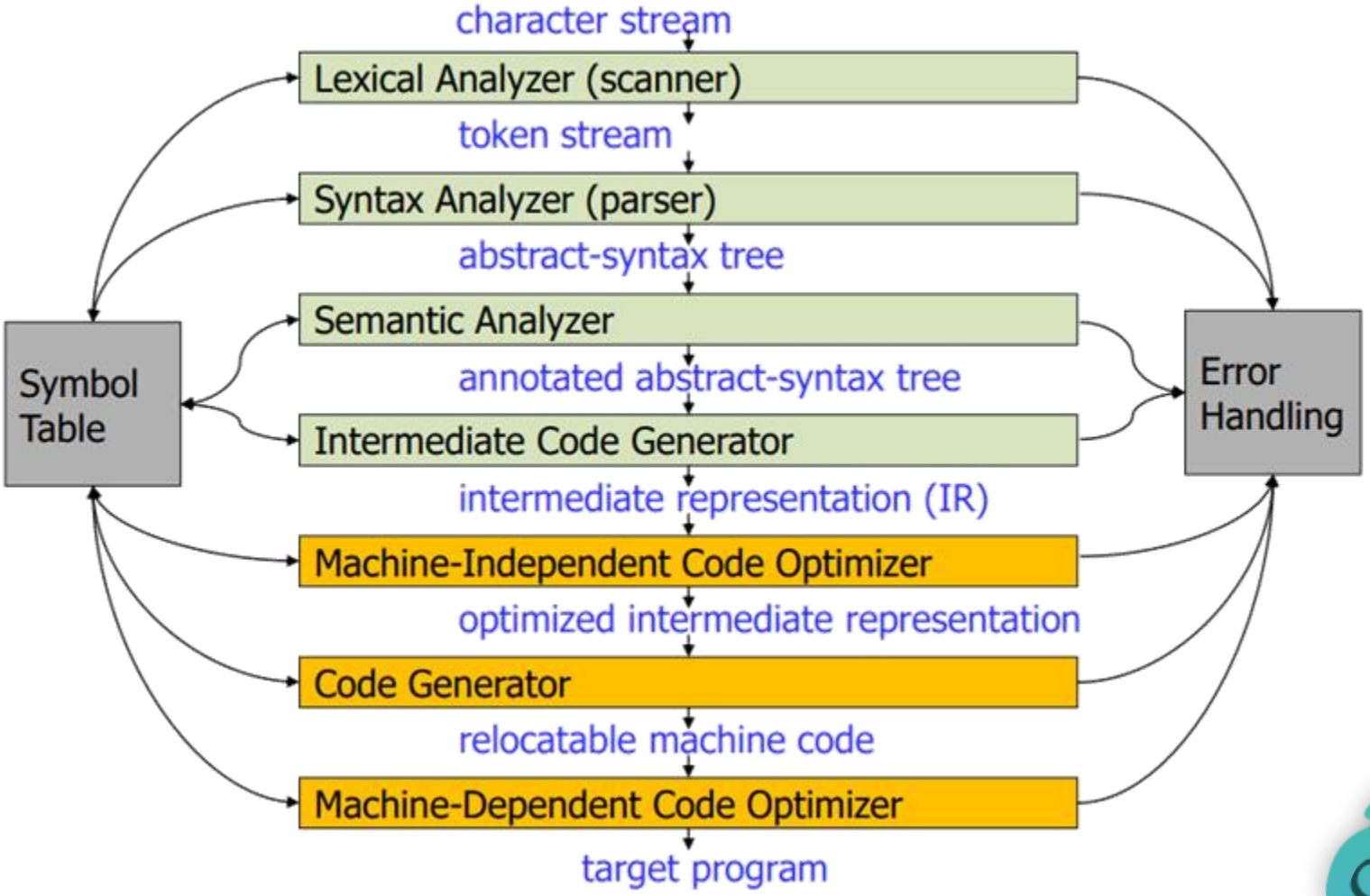
國立中山大學 National Sun Yat-sen University
資訊工程學系 Department of Computer Science and Engineering

Compiler

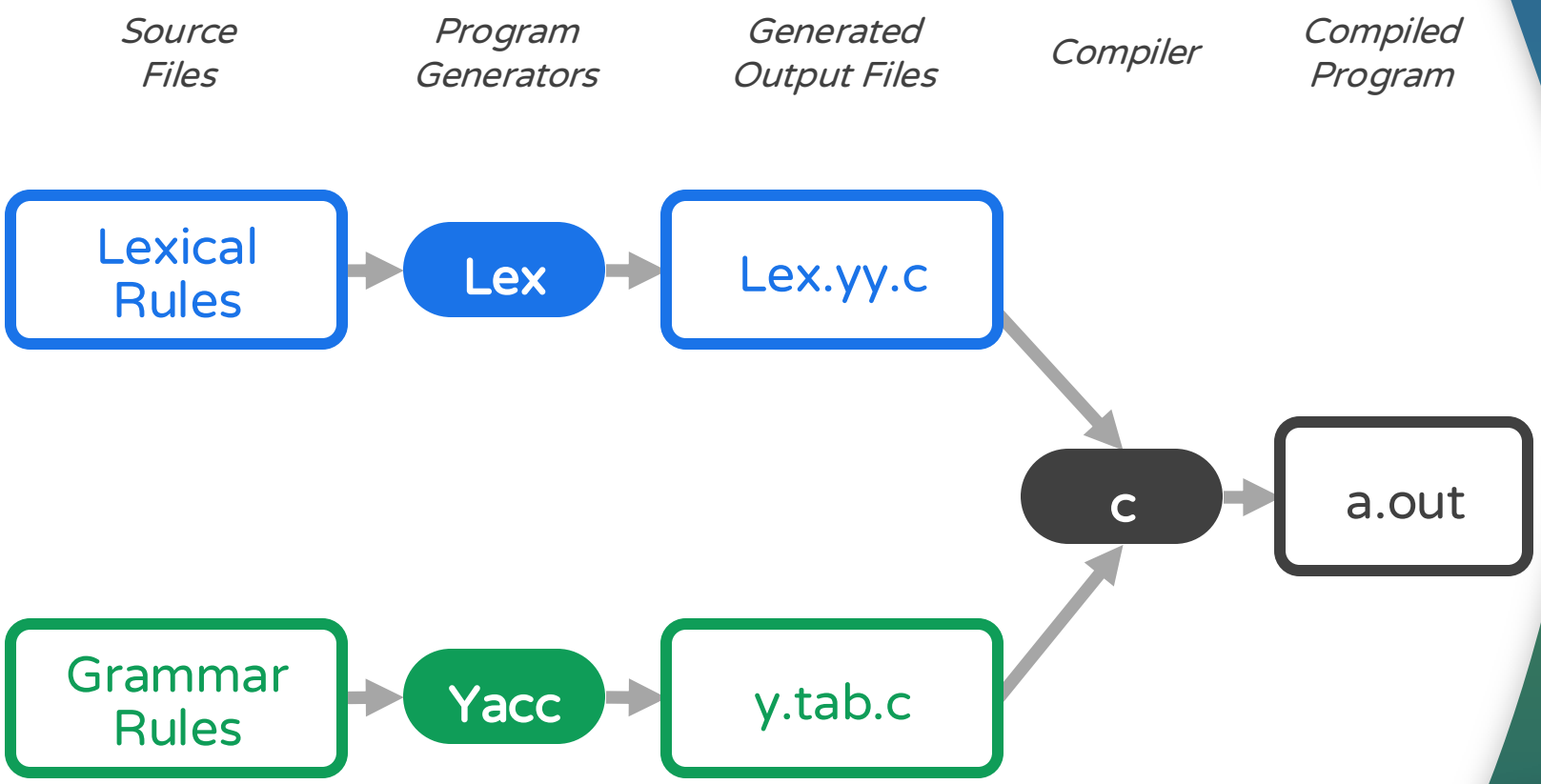
Yacc Parser

TA| Xiang, Long, Ding

The Structure of a Compiler



Lex & Yacc


Source Files
Program Generators
Generated Output Files
Compiler
Compiled Program
Lexical Rules
Lex
Lex.yy.c
Grammar Rules
Yacc
y.tab.c
c
a.out

Lex & Yacc

Lex	Yacc
Lexical Analyzer	Yet Another Compiler Compiler
Compare the strings described in the standard language.	Determine if a sentence is grammatically correct.
Cut the input data into smaller units: Tokens	Organize the tokens logically.

Yacc work

Purpose

- The purpose of Yacc is to "check if the syntax is valid".

Operation

- Yacc will treat the Input as a Sequence of Tokens
  - A sequence of more than one consecutive tokens will form a Grammar.
- Lex is just a routine of Yacc.
  - Responsible for sending the token back to Yacc

Yacc grammar

- Suppose we want to design a simple computer parser grammar, where NUMBER is the token captured by Lex.

```
expression → NUMBER
expression → expression + NUMBER
expression → expression - NUMBER
```

- The same LHS can be combined together, and each RHS is separated by |. In Yacc, this syntax will be represented as:

```
expression : NUMBER
           | expression + NUMBER
           | expression - NUMBER
```

Situations that Yacc cannot handle

- Yacc cannot handle the syntax for Ambiguous.
- Yacc cannot handle syntax that requires referencing more than one token.

- Please rewrite the syntax, or ensure the precedence order (%left).

```
Phrase → cart_animal AND CART
       | work_animal AND PLOW

cart_animal → HORSE | GOAT
work_animal → HORSE | OX
```

Yacc program

Yacc Format

Divided into three parts, each separated by %%.

```
Definition
%%
Grammars
%%
User Code
```

Definition

calc.y

```
%{
#include <stdio.h>
int yylex();
double ans = 0;
void yyerror(const char* message) {
    printf("Invaild format\n");
};
%}

%union {
    float   floatVal;
    int     intVal;
}
%type <floatVal> NUMBER
%type <floatVal> expression term factor group
%token PLUS MINUS MUL DIV
%token LP RP
%token NUMBER NEWLINE

%%
```

Definition

calc.y

```
%{
#include <stdio.h>
int yylex();
double ans = 0;
void yyerror(const char* message) {
    printf("Invaild format\n");
};
%}
```

Definition

calc.y

```
%union {
    float   floatVal;
    int     intVal;
}

%type <floatVal>   NUMBER
%type <floatVal>   expression term
                   factor group

%token PLUS MINUS MUL DIV
%token LP RP
%token NUMBER NEWLINE

%%
```

%type usually declares Non-Terminals

%token usually declares Terminals

Definition

Although NUMBER is a terminal, we still need to know its actual value!

```
%type <floatVal>   NUMBER
%type <floatVal>   expression term
                   factor group
```

Purpose of %type

Besides passing the token's "type" to Yacc,
we may need to know its "actual value" (usually non-terminals).

Therefore, we need to define the token's value type here so that Lex can pass the actual value to Yacc by storing yylval.

Grammars

calc.y

```
%%
lines :/* empty */
    | lines expression NEWLINE {printf("%lf\n", $2);}
    ;
expression : term { $$ = $1; }
    | expression PLUS term { $$ = $1 + $3; }
    | expression MINUS term { $$ = $1 - $3; }
    ;
term : factor {  $$ = $1; }
    | term MUL factor { $$ = $1 * $3; }
    | term DIV factor { $$ = $1 / $3; }
    ;
factor : NUMBER { $$ = $1;}
    | group {$$ = $1; }
    ;
group : LP expression RP { $$ = $2; }
    ;
%%
```

Grammars

calc.y

```
expression : term { $$ = $1; }
```

$$ $1

```
    | expression PLUS term { $$ = $1 + $3; }
```

$1 $2 $3

```
    | expression MINUS term { $$ = $1 - $3; }
```

$1 $2 $3

```
    ;
```

User Code

calc.y

```
%%

int main() {
    yyparse();
    return 0;
}
```

Modify Lex program

Definition

calc.l

```
%{
#include "y.tab.h"
#include <stdio.h>
%}

Digit [0-9]+

%%
```

Grammars

calc.l

```
%%

{Digit}       { sscanf(yytext, "%f",
                &yylval.floatVal); return NUMBER;}
\+            {return PLUS;}
\-            {return MINUS;}
\*            {return MUL;}
\/            {return DIV;}
\(            {return LP;}
\)            {return RP;}
\n            {return NEWLINE;}
.             {return yytext[0];}

%%
```

Using Yacc

How to use Lex File

- First, you must install the flex program to compile your lex file.
  - `sudo apt-get install bison` (using Ubuntu as an example)
- Compile cau.y (generating y.tab.c and y.tab.h)
  - `bison –y –d cau.y`
- Compile cau.lex (generating lex.yy.c)
  - `flex cau.l`
- Generate the executable file by using gcc (generating the calc executable)
  - `gcc lex.yy.c y.tab.c -ly –lfl -o calc`
- Execution method
  - `./calc < testfile`

Compilation process

- The example includes a pre-written makefile for your reference.

```
all:   clean y.tab.c lex.yy.c
       gcc lex.yy.c y.tab.c -ly -lfl -o calc

y.tab.c:
     bison -y -d cau.y

lex.yy.c:
     flex cau.l

clean:
     rm -f calc lex.yy.c y.tab.c y.tab.h
```

- Running "make all" will compile and generate "calc".

Error Handling

test1.java

INPUT

```
3++9
5/2
9*3
3+5
4**6
5+***6+*6
4+3*9-10*8
```

OUTPUT

```
** Syntax Error at Line 1 **
Line 2: 5 / 2
Line 3: 9 * 3
Line 4: 3 + 5
** Syntax Error at Line 5 **
** Syntax Error at Line 6 **
Line 7: 4 + 3 * 9 - 10 * 8
```

Result– Test1

test1.java

INPUT

```
/* Test file: Perfect test file
 * Compute sum = 1 + 2 + ... + n
 */
class sigma {
  // "final" should have
const_expr
  final int n = 10;
  int sum, index;

  main()
  {
    index = 0;
    sum = 0;
    while (index ⩽ n)
    {
      sum = sum + index;
      index = index + 1;
    }
    print(sum);
  }
}
```

OUTPUT

```
line  1: /* Test file: Perfect test file
line  2:  * Compute sum = 1 + 2 + ... + n
line  3:  */
line  4: class sigma {
line  5: // "final" should have const_expr
line  6: final int n = 10 ;
line  7: int sum , index ;
line  8:
line  9: main ( )
line 10: {
line 11: index = 0 ;
line 12: sum = 0 ;
line 13: while ( index ⩽ n )
line 14: {
line 15: sum = sum + index ;
line 16: index = index + 1 ;
line 17: }
line 18: print ( sum ) ;
line 19: }
line 20: }
```

Result – Test2

test2.java

INPUT

```
/* Test file: ... */
class Point
{
    static int counter ;
    int x, y ;
    /*Duplicate declare x*/
    int x ;
    void clear()
    {
        x = 0 ;
        y = 0 ;
    }
}
```

OUTPUT

```
line  1: /* Test file:  ... */
line  2: class Point
line  3: {
line  4: static int counter ;
line  5: int x , y ;
line  6: /*Duplicate declare x*/
line  7: int x ;
> 'x' is a duplicate identifier.
line  8: void clear ( )
line  9: {
line 10: x = 0 ;
line 11: y = 0 ;
line 12: }
line 13: }
```

Result – Test3

test3.java

INPUT

```
/* Test file of ... */
class Point {
    int z;
    int x y ;
    /*Need  ',' before y*/
    float w;
}
class Test {
    int d;
    Point p = new Point()
    /*Need ';' at EOL*/
    int w,q;
}
```

OUTPUT

```
line  1: /* Test file of ... */
line  2: class Point {
line  3: int z ;
Line  4, char: 12, a syntax error at "y"
line  4: int x y ;
line  5: /*Need  ',' before y*/
line  6: float w ;
line  7: }
line  8: class Test {
line  9: int d ;
line 10: Point p = new Point ( )
Line 10, char: 17, statement without semicolon
line 11: /*Need ';' at EOL*/
line 12: int w , q ;
line 13: }
```

About Homework II

Recommended env

- Install Ubuntu on a virtual machine
  - Ubuntu 22

Homework Submission

- **DUE DATE: x/xx 23:59**
- **The design of Yacc is much more complex than that of Lex, so please start writing it immediately.**
- **The program demo environment is Ubuntu 22.04.2 LTS.**
- **Please refer to the test files on the course webpage to verify your program.**
- **Please submit your assignment on time; late submissions will receive a 30% reduction in grade.**
- **Please compress your assignment into a single compressed file and upload it to the online university, naming the file "student id_hw2.zip".**
- **A demo session will be scheduled after the submission deadline. Please arrive at the EC5023 Database Systems Lab on time to find the teaching assistant for the demo.**

Issues

- Your parser should be able to generate proper error messages, when it encounters an error.
  - For example: the line number where the error occurred, the position of the character, and an explanation of the reason for the error.
- When the parser encounters an error, it should process as much input as possible.
  - In other words, the parser should perform recovery, when it encounters an error.

Scoring method

20% of the test data	Three of the six publicly available test data sets will be randomly selected. These three must contain the same error message as the question (the error message can be represented in different ways).
10% of the test data	Two hidden test data points, randomly combined from publicly available test data points.
5%	Note: Explains how to process each Statement.
5%	Readme.pdf (Please refer to the first page of the assignment instructions for the content).
5% + 5%	Oral Q&A *2
~%	Bonus

Contact Information

Feel free to ask the teaching assistant questions.

丁襄龍
clovedragon12@gmail.com
EC5023 DBSL