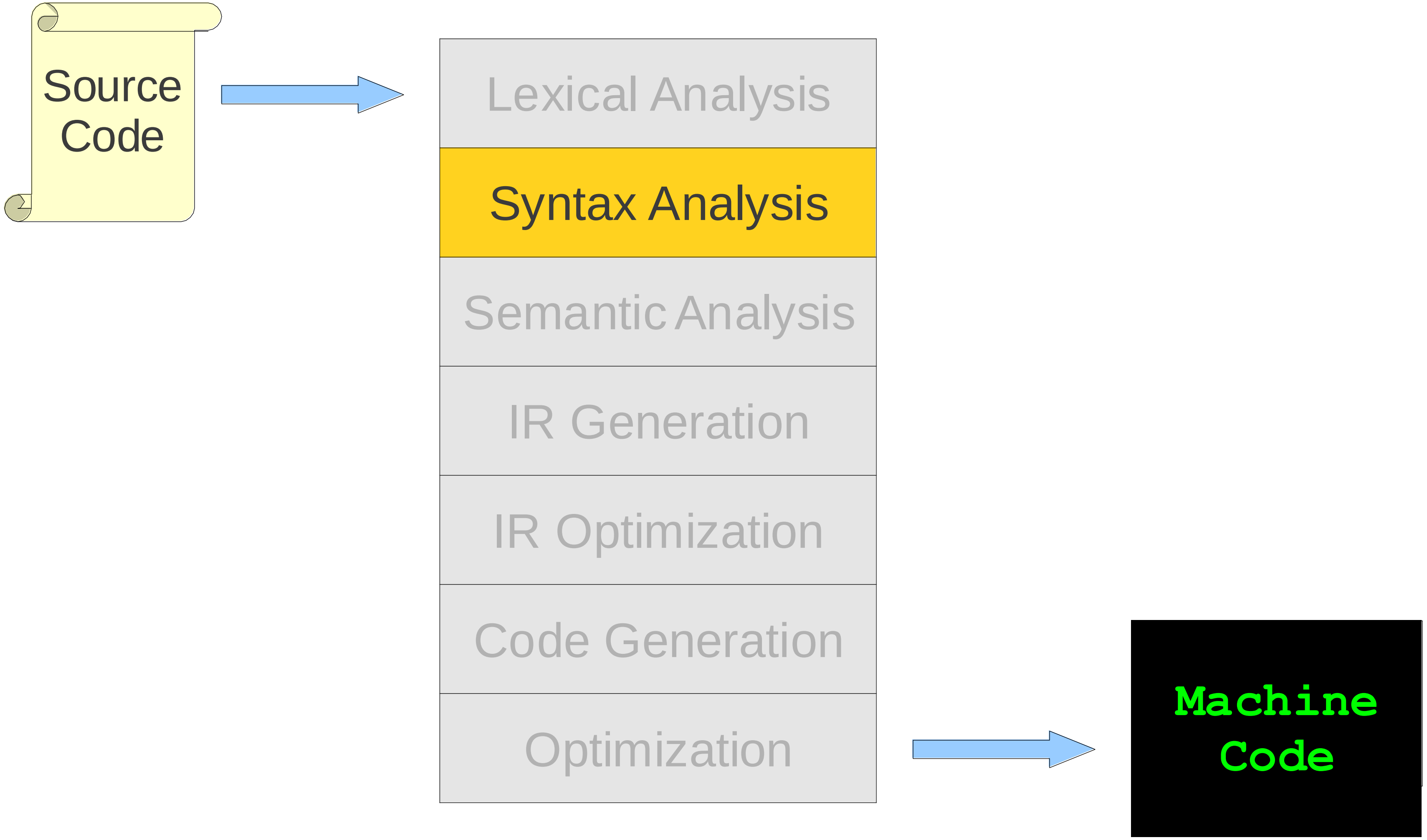
Compiler
– 3-7. Syntactic Error Handler & Parser Generator –

JIEUNG KIM

jieungkim@yonsei.ac.kr

YONSEI UNIVERSITY

Where are we?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Outlines

- Role of the syntax analysis (parser)
- Context free grammar
- Push down automata
- Top-down parsing
- Bottom-up parsing
- Simple LR
- More powerful LR parsers and other issues in parsers
- Syntactic error handler
- Parser generator

Syntactic error handler

Syntactic error handler

- Kinds of errors
  - Errors in structure
  - Missing operator
  - Misspelled keywords
  - Unbalanced parenthesis

Syntactic error handler

- Kinds of errors
  - Example: Missing closing braces

```
void printHelloNinja( String s ) {
  // function - body
```

  - Example: Missing semicolon

```
x = a + b * c  //missing semicolon
```

  - Example: Errors in expression

```
a = (b + c * (c + d);      //missing closing parentheses
i = j * + c ;       // missing argument between "*" and "+"
```

Syntactic error handler

- Panic mode recovery
  - Successive characters from the input are removed one at a time until a designated set of synchronizing tokens is found.
  - Synchronizing tokens are deli-meters such as; or }
  - It's easy to implement and guarantees not to go into an infinite loop.
  - A considerable amount of input is skipped without checking it for additional errors.

```
int main() {
    int a = 5
    printf("Hello World!");
}
```

- Missing semicolon after `int a = 5`.
- The parser skips input until it finds the next ; or a closing brace } after encountering the error.

YONSEI UNIVERSITY

Syntactic error handler

- Statement mode recovery
  - It performs the necessary correction on the remaining input.
  - The rest of the input statement allows the parser to parse ahead.
  - The correction can be deletion of extra semicolons, replacing the comma with semicolons, or inserting a missing semicolon.
  - While performing correction, utmost care should be taken for not going in an infinite loop.
  - A disadvantage is that it finds it difficult to handle situations where the actual error occurred before pointing of detection.

Syntactic error handler

- Statement mode recovery

```
int main() {
    int a = 5
    printf("Hello World!");
}
```

- Missing semicolon after `int a = 5`.
- The parser automatically inserts the missing semicolon, tra nsforming the code into:

```
int main() {
    int a = 5;
    printf("Hello World!");
}
```

- Parsing continues as if the error was corrected in place.

Syntactic error handler

- Statement mode recovery

```
int main() {
    int a = ;
}
```

  - Instead of inserting a valid value or skipping the =, it keeps inserting a semicolon

```
int a = ;;
```

  - The parser again detects an error after this "correction" because ;; is invalid.
  - The recovery strategy tries to "fix" the issue again by adding another token:

```
int a = ;;;

      ⋮

int a = ;;;;
```

YONSEI UNIVERSITY

Syntactic error handler

- Error production
  - If a user has knowledge of common errors that can be encountered then, these errors can be incorporated by augmenting the grammar with error productions that generate erroneous constructs.
  - If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
  - The disadvantage is that it’s difficult to maintain.

Syntactic error handler

- Error production

```
int main() {
    int a = 5 5;
}
```

- Extra 5 after the assignment.
- The parser includes a rule in the grammar to recognize an extra token in the assignment.

```
assignment → identifier '=' number number ';'
```

- The parser detects the error and can give a specific message like "Unexpected token 5 in assignment", allowing further recovery.

Syntactic error handler

- Global correction
  - The parser examines the whole program and tries to find out the closest match for it which is error-free.
  - The closest match program has less number of insertions, deletions, and changes of tokens to recover from erroneous input.
  - Due to high time and space complexity, this method is not implemented practically.

YONSEI UNIVERSITY

Syntactic error handler

- Global correction

```
int main() {
    int a == 5;
    print("Hello World!");
}
```

- == used instead of = in the assignment.
- print should be printf.

- The parser identifies both the incorrect assignment operator and the incorrect function name.
- Suggests the minimal set of changes to correct them.

```
int main() {
    int a = 5;
    printf("Hello World!");
}
```

YONSEI UNIVERSITY

Parser generator

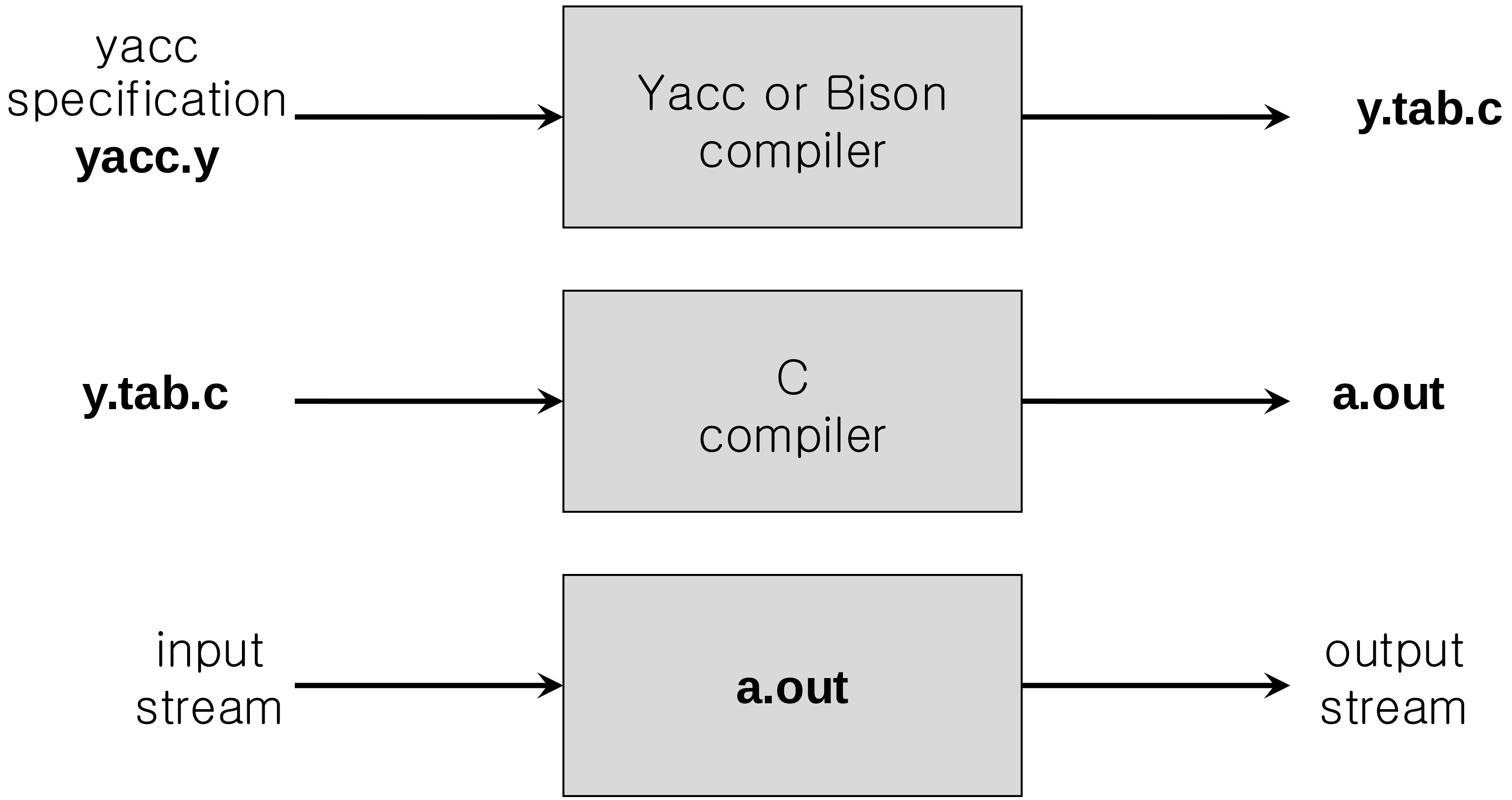
Parser generator

- ANTLR, Yacc, and Bison
  - *ANTLR* tool
    - Generates LL(*k*) parsers
  - *Yacc* (Yet Another Compiler Compiler)
    - Generates LALR(1) parsers
  - *Bison*
    - Improved version of Yacc

YONSEI UNIVERSITY

Parser generator

- Parser generator



Parser generator

- Yacc specification
  - A *yacc specification* consists of three parts:

```
yacc declarations, and C declarations within %{ %}
%%
translation rules
%%
user-defined auxiliary procedures
```

  - The *translation rules* are productions with actions:

$production_1$ { $semantic\ action_1$ }
$production_2$ { $semantic\ action_2$ }
…
$production_n$ { $semantic\ action_n$ }

Parser generator

- Writing a grammar with Yacc
  - Productions in Yacc are of the form

```
Nonterminal    : tokens/nonterminals { action }
               | tokens/nonterminals { action }
               …
               ;
```

  - Tokens that are single characters can be used directly within productions, e.g., **'+'**
  - Named tokens must be declared first in the declaration part using

```
%token TokenName
```

Parser generator

- Yacc specification
  - Declaration
    - The declaration parts are optional
    - Include & token
  - Translation rules: pair of translation rules

```
<head> : <body>        { <semantic action> }
       | <body>        { <semantic action> }
       | <body>        { <semantic action> }
       ;
```

For `E ::= E+T | T`

```
expr : expr '+' term { $$ = $1 + $3;}
     | term
     ;
```

`$$`: non-terminal of head
`$1`: the first grammar symbol (expr)
`$3`: the third grammar symbol (term)
– What is the second?

Parser generator

- Yacc specification
  - Supporting C routine
    - Must have `yylex( )`
    - Receive token by global variable `yyval`

YONSEI UNIVERSITY

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line     : expr '\n'                 { printf("%d\n", $1); }
         ;
expr     : expr'+'term               { $$ = $1+$3; }
         | term
         ;
term     : term'*'factor             { $$ = $1*$3; }
         | factor
         ;
factor : '(' expr ')'                { $$ = $2; }
         | DIGIT
         ;
%%
yylex() {
  int c;
  c = getchar();
  if (isdigit(c)) {
    yylval = c-'0';
    return DIGIT;
  }
  return c;
}
```

To combine it with Lex

Parser generator

- Ambiguous grammar
  - Yacc is able to handle ambiguous grammar.
  - Default rules:
    - Reduce/reduce conflict resolved by choosing the production rule listed first.
    - A shift/reduce conflict is resolved in favor of shift.
  - User can declare the conflict resolving rules.

Parser generator

- Error recovery in Yacc
  - Use error production
    - `A ::= error α`
  - Yacc encounters an error
    - Pops the symbol from stack until error is legal (which means there is error production)

      `A ::= error α`

      and compare input with α

YONSEI UNIVERSITY

```
E ::= E+E | E-E | E*E | E/E | -E | number
%left '+' '-'
%nonassoc '<'
%prec <terminal>
```

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double        /* double type for Yacc stack */
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines    : lines expr `\n'              {  printf("%g\n", $2); }
         | lines `\n'
         | /* empty */
         | error '\n'                   { yyerror("reenter previous line:");
                                          yyerrok; }
         ;
expr     : expr '+' expr                { $$ = $1 + $3; }
         | expr '-' expr                { $$ = $1 - $3; }
         | expr '*' expr                { $$ = $1 * $3; }
         | expr '/' expr                { $$ = $1 / $3; }
         | '(' expr ')'                 { $$ = $2; }
         | '-' expr %prec UMINUS        { $$ = -$2; }
         | NUMBER
         ;
%%
yylex() {
  int c;
  while ((c = getchar()) == ' ');
  if ((c == '.') || (isdigit(c))) {
    ungetc(c, stdin);
    scanf("%lf, &yylval);
    return NUMBER;
  }
  return c;
}
```

YONSEI UNIVERSITY

Parser generator

- `yacc.py` in PLY
  - Quite similar to Yacc
  - The `ply.yacc` module implements the parsing component of PLY
  - Syntax is usually specified in terms of a BNF grammar

```
expression : expression + term
          | expression - term
          | term term : term * factor
          | term / factor
          | factor
factor : NUMBER | ( expression )
```

```python
import ply.yacc as yacc

# Get the token map from the lexer. This is required.
from calclex import tokens

def p_expression_plus(p):
  'expression : expression PLUS term’
  p[0] = p[1] + p[3]

def p_expression_minus(p):
  'expression : expression MINUS term’
  p[0] = p[1] - p[3]

def p_expression_term(p):
  'expression : term’
  p[0] = p[1]

def p_term_times(p):
  'term : term TIMES factor’
  p[0] = p[1] * p[3]

def p_term_div(p):
  'term : term DIVIDE factor’
  p[0] = p[1] / p[3]
```

```python
def p_term_factor(p):
  'term : factor’
  p[0] = p[1]

def p_factor_num(p):
  'factor : NUMBER’
  p[0] = p[1]

def p_factor_expr(p):
  'factor : LPAREN expression RPAREN’
  p[0] = p[2]

# Error rule for syntax errors
def p_error(p):
  print("Syntax error in input!")

# Build the parser
parser = yacc.yacc()

while True:
  try:
    s = input('calc > ‘)
  except EOFError:
    break
  if not s: continue
  result = parser.parse(s)
  print(result)
```

```
expression : expression PLUS expression          # level = 1, left
           | expression MINUS expression         # level = 1, left
           | expression TIMES expression         # level = 2, left
           | expression DIVIDE expression        # level = 2, left
           | LPAREN expression RPAREN            # level = None (not specified)
           | NUMBER                              # level = None (not specified)
```

```
precedence = (
  ('left', 'PLUS', 'MINUS’),
  ('left', 'TIMES', 'DIVIDE’),
)
```

Questions?