CSE216 Programming Abstractions
Parser

YoungMin Kwon

Parsing

- Context Free Grammar (CFG)
  - Generator for a CF language
- Parser
  - Language recognizer
  - Assembles tokens into a syntax tree
  - Calls scanner to get the tokens of the input program

Context Free Grammar

- CFG consists of
  - Terminals
    - Tokens from Scanner
    - E.g.) `ID`, `NUM`, `λ`, `.`
  - Nonterminals
    - Syntactic variables that denote sets of tokens
    - They help define the language generated by the grammar
    - E.g.) `expr`
      - `expr ≡ { NUM, ID, λID.NUM, λID.λID.ID, λID.ID NUM, …}`

```
expr → ID
     | NUM
     | λ ID . expr
     | expr expr
```

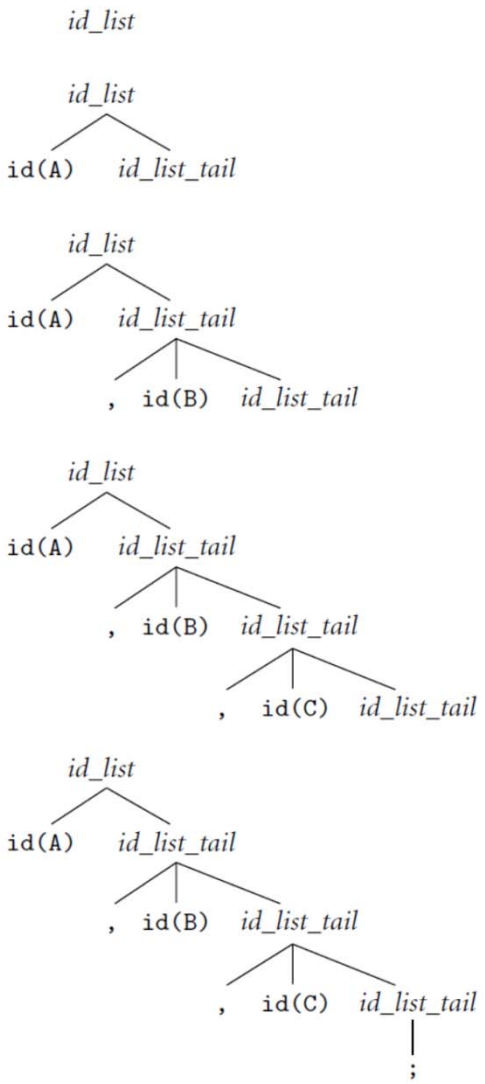
Context Free Grammar

```
expr → ID
     | NUM
     | λ ID . expr
     | expr expr
```

- CFG consists of (cont'd)
  - Production rules
    - How the terminals and non-terminals are combined to form strings
  - Start symbol
    - A nonterminal that denotes the language defined by the grammar
    - E.g.) `expr`

Parsing

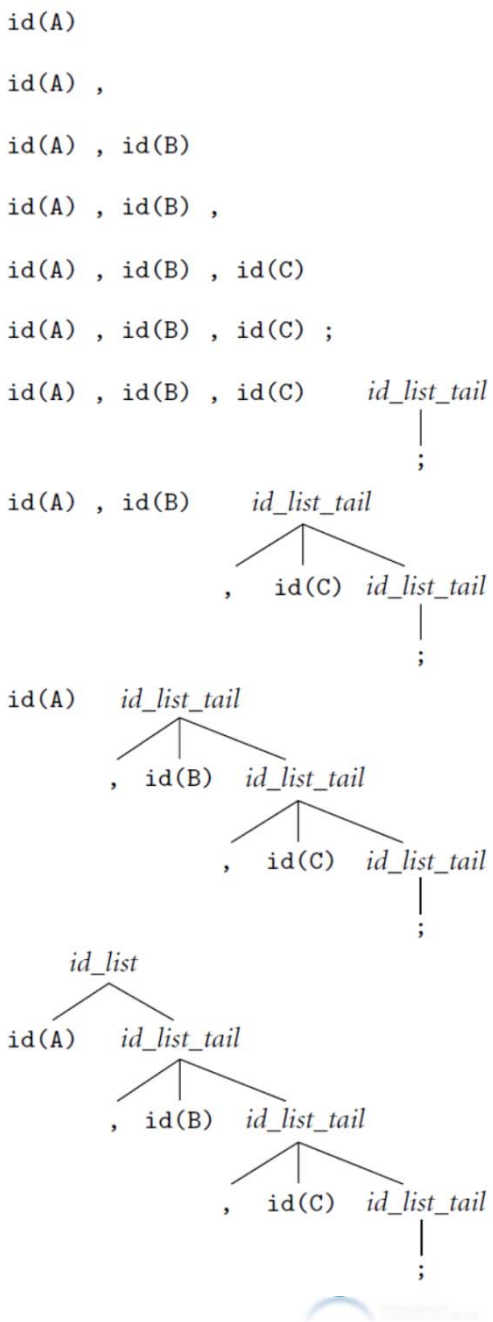
- Top-down parsing
  - Parse tree is built from top to bottom
  - Predictive parsing algorithm
  - LL Parsers,
    Recursive Descent Parser



id_list ⟶ `id` *id_list_tail*
id_list_tail ⟶ `,` `id` *id_list_tail*
id_list_tail ⟶ `;`

Parsing

- Bottom-up parsing
  - Parse tree is built from bottom to top
  - Shift-reduce parsing algorithm
  - LR parsers



Recursive Descent Parsing

- Recursive descent parsing
  - Top-down (predictive) parsing
  - Suitable for a simple grammar
  - Parsers can be written by hand

Recursive Descent Parsing

- How to write a recursive descent parser
  - Write a subroutine for each non-terminal
  - Input-token is the token available from the scanner
  - Match to consume and update the input-token

Example: Simple Calculator

- An example program

```
//program square test
//(a + b)^2 = a*a + 2*a*b + b*b
{
    read a
    read b
    c := a + b
    write c * c
    write a*a + 2*a*b + b*b
    write a*a + 2*a*b + b*b - c*c
}
```

Simple Calculator

- Syntax in CFG

```
program
     -> stmt EOF

stmt -> stmt_assignment
     |  stmt_read
     |  stmt_write
     |  stmt_compound

stmt_assignment
     -> ID := expr

stmt_read
     -> READ ID

stmt_write
     -> WRITE expr

stmt_compound
     -> { stmt_list }
```

```
stmt_list
     -> stmt
     |  stmt_list stmt

expr -> expr_add

expr_add
     -> expr_mul
     |  expr_add + expr_mul
     |  expr_add - expr_mul

expr_mul
     -> expr_factor
     |  expr_mul * expr_factor
     |  expr_mul / expr_factor

expr_factor
     -> NUM
     |  ID
     |  ( expr )
```

Simple Calculator

- Recursive Descent Parser
  - Functions for non-terminals

```
static void parse_program();
static void parse_stmt();
static void parse_stmt_assignment();
static void parse_stmt_read();
static void parse_stmt_write();
static void parse_stmt_compound();
static void parse_stmt_list();

static void parse_expr();
static void parse_expr_add();
static void parse_expr_mul();
static void parse_expr_factor();
```

```
//scanner state
static scan_t sc;

static void match(int expected) {
    ON_FALSE_EXIT( sc.token == expected,
              strmsg("syntax error: in %d, expected %d, but has %d (%s)",
                     sc.line, expected, sc.token, sc.text));
    read_token(&sc);
}

static void parse_program() {
    parse_stmt();
    match(TOK_EOF);
}
```

```
program
     -> stmt EOF
```

```
static void parse_stmt() {
    if(sc.token == TOK_ID)
        parse_stmt_assignment();
    else if(sc.token == TOK_READ)
        parse_stmt_read();
    else if(sc.token == TOK_WRITE)
        parse_stmt_write();
    else if(sc.token == '{')
        parse_stmt_compound();
    else
        ON_FALSE_EXIT(0,
                strmsg("syntax error: in %d near (%s), not a statement",
                       sc.line, sc.text));
}
```

```
stmt -> stmt_assignment
      |  stmt_read
      |  stmt_write
      |  stmt_compound
```

```c
static void parse_stmt_assignment() {
    match(TOK_ID);
    match(TOK_ASSIGN);
    parse_expr();
}

static void parse_stmt_read() {
    match(TOK_READ);
    match(TOK_ID);
}

static void parse_stmt_write() {
    match(TOK_WRITE);
    parse_expr();
}

static void parse_stmt_compound() {
    match('{');
    parse_stmt_list();
    match('}');
}

static void parse_stmt_list() {
    while(sc.token != '}')
        parse_stmt();
}
```

```
stmt_assignment
     -> ID := expr

stmt_read
     -> READ ID

stmt_write
     -> WRITE expr

stmt_compound
     -> { stmt_list }

stmt_list
     -> stmt
     |  stmt_list stmt
```

```
static void parse_expr() {
    return parse_expr_add();
}

static void parse_expr_add() {
    parse_expr_mul();
    while(sc.token == '+' ||
          sc.token == '-') {
        match(sc.token);
        parse_expr_mul();
    }
}

static void  parse_expr_mul() {
    parse_expr_factor();
    while(sc.token == '*' ||
          sc.token == '/') {
        match(sc.token);
        parse_expr_factor();
    }
}
```

```
expr -> expr_add

expr_add
     -> expr_mul
     |  expr_add + expr_mul
     |  expr_add - expr_mul

expr_mul
     -> expr_factor
     |  expr_mul * expr_factor
     |  expr_mul / expr_factor
```

```
static void parse_expr_factor() {
    if(sc.token == TOK_NUM)
        match(TOK_NUM);
    else if(sc.token == TOK_ID)
        match(TOK_ID);
    else if(sc.token == '(') {
        match('(');
        parse_expr();
        match(')');
    }
}

void parse(char *fname) {
    open_scan(&sc, fname);
    read_token(&sc);
    parse_program();
    close_scan(&sc);
}
```

```
expr_factor
     -> NUM
     |  ID
     |  ( expr )
```

Simple Calculator

- Scanner

```
enum tokens {
    TOK_ALL = 256,  //match for the whole expr
    TOK_WS,         //white space
    TOK_COMMENT,    //comment
    TOK_READ,       //read
    TOK_WRITE,      //write
    TOK_ASSIGN,     //assignment
    TOK_NUM,        //number
    TOK_ID,         //identifier
    TOK_EOF,        //end of file
    TOK_COUNT = (TOK_EOF - TOK_ALL + 1) //token count
};

#define TOK_REGEXP  "^([ \t\n\r]+)|"              /*whitespace*/\
                    "^(//[^\n]*\n)|"              /*comment*/\
                    "^(read)|"                    /*read*/\
                    "^(write)|"                   /*write*/\
                    "^(:=)|"                      /*assignment*/\
                    "^([0-9]+)|"                  /*number*/\
                    "^([_a-zA-Z][_a-zA-Z0-9]*)"   /*identifier*/
```

```
typedef struct scan {
    regex_t regexp;     //regular expression
    int token;          //current token
    char buff[1024];    //current line
    char *pos;          //current position in the buffer
    char text[256];     //text for the current token
    int line;
    FILE *fp;           //file pointer
} scan_t;

//open a scanner
extern void open_scan(scan_t *sc, char *fname);

//close a scanner
extern void close_scan(scan_t *sc);

//read token and return 0 if it is EOF
extern int read_token(scan_t *sc);
```

```
void open_scan(scan_t *sc, char *fname) {
    sc->line = 0;
    sc->token = 0;
    sc->buff[0] = 0;
    sc->pos = sc->buff;
    sc->text[0] = 0;
    sc->fp = stdin;

    //initialize regexp
    int res = regcomp(&sc->regexp, TOK_REGEXP, REG_EXTENDED);
    if(res != 0) {
        char msg[256];
        regerror(res, &sc->regexp, msg, sizeof(msg));
        ON_FALSE_EXIT(0, strmsg("error in regcomp: %d, %s", res, msg));
    }

    //initialize fp
    if(fname && fname[0]) {
        sc->fp = fopen(fname, "rt");
        ON_FALSE_EXIT(sc->fp != NULL, strmsg("cannot open %s", fname));
    }
}
```

```
void close_scan(scan_t *sc) {
    //free regexp
    regfree(&sc->regexp);

    //close fp
    if(sc->fp == NULL || sc->fp != stdin) {
        fclose(sc->fp);
        sc->fp = NULL;
    }
}
```

```
//read token and return 0 if it is EOF
int read_token(scan_t *sc) {
    while(1) {
        //read a line from the file
        if(*sc->pos == '\0') {
            char *res = fgets(sc->buff, sizeof(sc->buff), sc->fp);
            if(res == NULL) { //EOF
                sc->token = TOK_EOF;
                strcpy(sc->text, "EOF");
                return 0;
            }
             sc->pos = sc->buff; //reset pos to the beginning of buff
             sc->line++;         //increate the line count
        }

        //read a token
        sc->pos += read_token_from_str(&sc->regexp, sc->pos,
                                       &sc->token, sc->text);

         //skip a white spaces or a comment
         if(sc->token == TOK_WS ||
            sc->token == TOK_COMMENT)
             continue;

        //return with the current token
        return 1;
    }
}
```

```
//return the length of tok_text
static int read_token_from_str(regex_t *regexp, char *str,
                               int *token, char *tok_text) {
    regmatch_t matches[TOK_COUNT];

    int res = regexec(regexp, str, TOK_COUNT, matches, 0);
    if(res == 0) {  //if there is a match
        int i = 0;
        for(i = 1; i < TOK_COUNT; i++) {
            if(matches[i].rm_so != 0)
                continue;

            *token = i + TOK_ALL;
            return token_text(&matches[i], str, tok_text);
        }
    }
    else {  //no match -> punctuators
        *token = str[0];
        tok_text[0] = str[0];
        tok_text[1] = 0;
        return 1;
    }
}
```

```
//copy the token text to tok_text
static int token_text(regmatch_t *match, char *str, char *tok_text) {
    int i = 0, j = match->rm_so;
    while(j < match->rm_eo)
        tok_text[i++] = str[j++];
    tok_text[i] = 0;

    return match->rm_eo - match->rm_so;
}

//
//app.c
//
int main(int argc, char **argv) {
…

    //parse the program
    parse(fname);

    printf("Success\n");
    return 0;
}
```

Test

```
> type program1.txt
//program square test
//(a + b)^2 = a*a + 2*a*b + b*b
{
    read a
    read b
    c := a + b
    write c * c
    write a*a + 2*a*b + b*b
    write a*a + 2*a*b + b*b - c*c
}
> a.exe program1.txt
Success
```

```
> type program2.txt
//program gcd
{
    read a
    read b
    while (a != b) {
        if (a > b)
            a := a - b
        else
            b := b - a
    }
    write a
}

> a.exe program2.txt
syntax error: in 4, expected 260,
but has 40 (()
in file: parser.c, function: match,
line: 32
```

Exercise

- Extend the simple calculator with
  - If-statement
  - While-statement
  - Comparison expressions (==, !=, <=, <, >=, >)

Exercise

- Syntax in CFG

```
program
     -> stmt EOF

stmt -> stmt_assignment
     |  stmt_read
     |  stmt_write
     |  stmt_compound
     |  stmt_if
     |  stmt_while
…

stmt_if
     -> IF ( expr ) stmt ELSE stmt

stmt_while
     -> WHILE ( expr ) stmt
…
```

```
expr -> expr_comp

expr_comp
     -> expr_add
     |  expr_add EQ expr_add
     |  expr_add NE expr_add
     |  expr_add LE expr_add
     |  expr_add <  expr_add
     |  expr_add GE expr_add
     |  expr_add >  expr_add

…
```

```
/*TODO: add below to tokens
    TOK_IF,          //if
    TOK_ELSE,        //else
    TOK_WHILE,       //while

    TOK_EQ,          //equal
    TOK_NE,          //not equal
    TOK_LE,          //less  than or equal to
    TOK_GE,          //greater than or equal to
*/
enum tokens {
    TOK_ALL = 256,  //match for the whole expr
…
};

/*TODO: add regular expressions for
    if, else, while
    equal, not-equal, less-than-equal, greater-than-equal
*/
#define TOK_REGEXP  "^([ \t\n\r]+)|"                /*whitespace*/\
…
```

```
//TODO: implement
static void parse_stmt_if();
static void parse_stmt_while();
static void parse_expr_comp();

static void parse_expr() {
    //TODO: call parse_expr_comp()
    return parse_expr_comp();
}
```

Test

```
> type program1.txt
//program square test
//(a + b)^2 = a*a + 2*a*b + b*b
{
    read a
    read b
    c := a + b
    write c * c
    write a*a + 2*a*b + b*b
    write a*a + 2*a*b + b*b - c*c
}
> a.exe program1.txt
Success
```

```
> type program2.txt
//program gcd
{
    read a
    read b
    while (a != b) {
        if (a > b)
            a := a - b
        else
            b := b - a
    }
    write a
}

> a.exe program2.txt
Success
```

A Solution

```
enum tokens {
    TOK_ALL = 256,  //match for the whole expr
    TOK_WS,         //white space
    TOK_COMMENT,    //comment
    TOK_READ,       //read
    TOK_WRITE,      //write
    TOK_IF,         //if
    TOK_ELSE,       //else
    TOK_WHILE,      //while
    TOK_ASSIGN,     //assignment
    TOK_EQ,         //equal
    TOK_NE,         //not equal
    TOK_LE,         //less  than or equal to
    TOK_GE,         //greater than or equal to
    TOK_NUM,        //number
    TOK_ID,         //identifier
    TOK_EOF,        //end of file
    TOK_COUNT = (TOK_EOF - TOK_ALL + 1) //token count
};
```

```
#define TOK_REGEXP  "^([ \t\n\r]+)|"                 /*whitespace*/\
                    "^(//[^\n]*\n)|"                 /*comment*/\
                    "^(read)|"                       /*read*/\
                    "^(write)|"                      /*write*/\
                    "^(if)|"                         /*if*/\
                    "^(else)|"                       /*else*/\
                    "^(while)|"                      /*while*/\
                    "^(:=)|"                         /*assignment*/\
                    "^(==)|"                         /*EQ*/\
                    "^(!=)|"                         /*NE*/\
                    "^(<=)|"                         /*LE*/\
                    "^(>=)|"                         /*GE*/\
                    "^([0-9]+)|"                     /*number*/\
                    "^([_a-zA-Z][_a-zA-Z0-9]*)"      /*identifier*/
```

```
static void parse_stmt() {
    if(sc.token == TOK_ID)
        parse_stmt_assignment();
    else if(sc.token == TOK_READ)
        parse_stmt_read();
    else if(sc.token == TOK_WRITE)
        parse_stmt_write();
    else if(sc.token == '{')
        parse_stmt_compound();
    //TODO: add if staement
    else if(sc.token == TOK_IF)
        parse_stmt_if();
    //TODO: add while statement
    else if(sc.token == TOK_WHILE)
        parse_stmt_while();
    else
…
}
```

```
stmt -> stmt_assignment
      |  stmt_read
      |  stmt_write
      |  stmt_compound
      |  stmt_if
      |  stmt_while
```

```
//TODO: implement static void parse_stmt_if()
static void parse_stmt_if() {
    match(TOK_IF);
    match('(');
    parse_expr();
    match(')');
    parse_stmt();
    match(TOK_ELSE);
    parse_stmt();
}

//TODO: implement static void parse_stmt_while()
static void parse_stmt_while() {
    match(TOK_WHILE);
    match('(');
    parse_expr();
    match(')');
    parse_stmt();
}
```

```
stmt_if
     -> IF ( expr ) stmt ELSE stmt

stmt_while
     -> WHILE ( expr ) stmt
```

```
static void parse_expr() {
    //TODO: call parse_expr_comp()
    return parse_expr_comp();
}

//TODO: implement static void parse_expr_comp()
static void parse_expr_comp() {
    parse_expr_add();
    if( sc.token == TOK_EQ || sc.token == TOK_NE ||
        sc.token == TOK_LE || sc.token == '<' ||
        sc.token == TOK_GE || sc.token == '>' ) {
        match(sc.token);
        parse_expr_add();
    }
}
```

```
expr -> expr_comp

expr_comp
     -> expr_add
     |  expr_add EQ expr_add
     |  expr_add NE expr_add
     |  expr_add LE expr_add
     |  expr_add <  expr_add
     |  expr_add GE expr_add
     |  expr_add >  expr_add
```