CSE216 Programming Abstractions
Interpreter

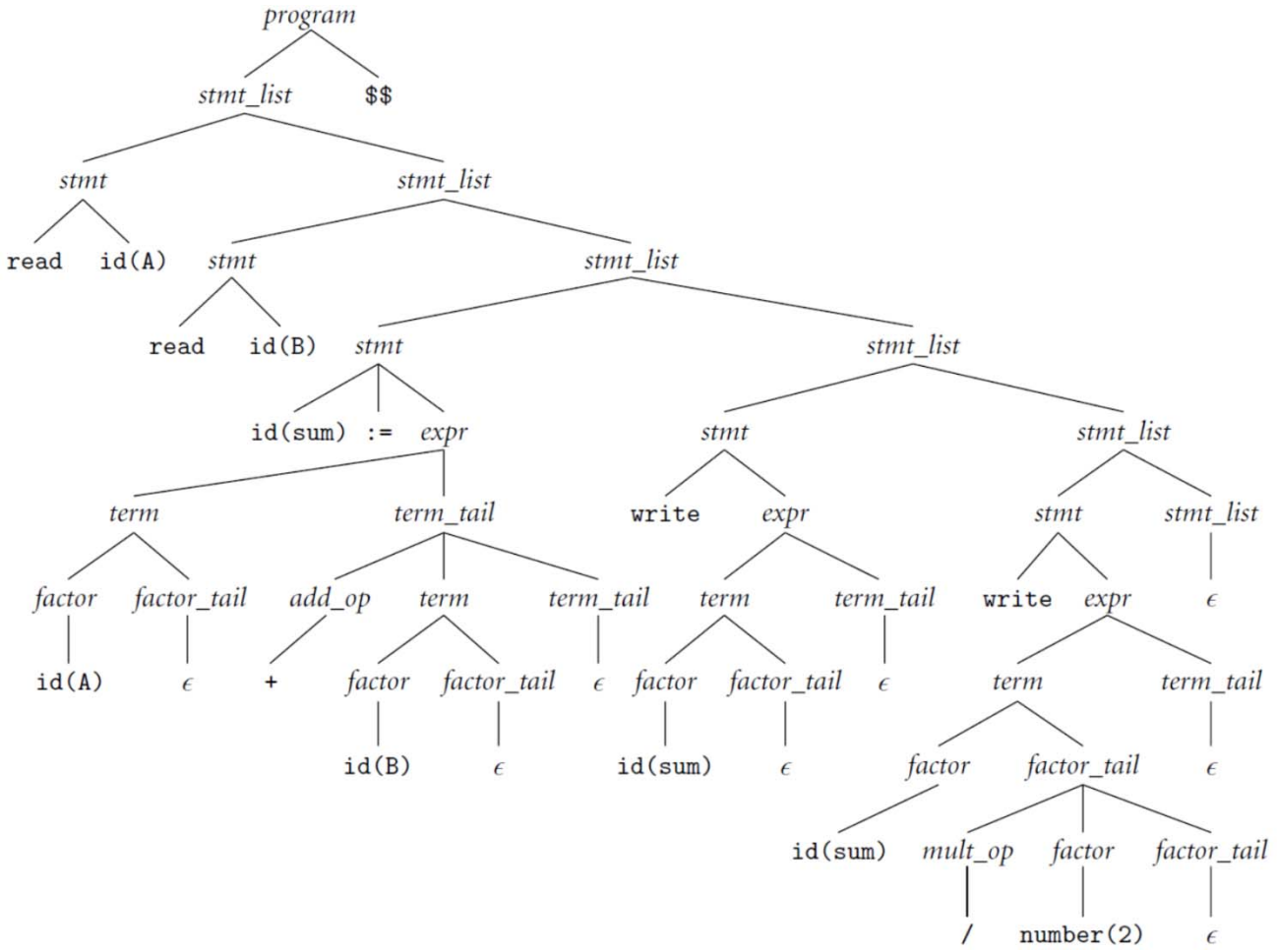
YoungMin Kwon

Building a Parse Tree

- Parse tree
  - A representation of high-level constructs in terms of their constituents
  - Constructs: nodes of a tree
    - E.g. Program, Statement, Expression
  - Constituents: children of a construct
  - Tokens: leaves of a tree

Parse Tree

- Example



Example: Simple Calculator

- Expression node

```
typedef struct expr {
    refobj_t ref;   //ref is at the beginning of rat
    int    ( *eval  )(struct expr *self);
    void   ( *print )(struct expr *self);
} expr_t;

//number
extern expr_t *expr_make_num(int n);

//variable
extern expr_t *expr_make_var(char *id);

//arithmetic operators
extern expr_t *expr_make_add(expr_t *a, expr_t *b);
extern expr_t *expr_make_sub(expr_t *a, expr_t *b);
extern expr_t *expr_make_mul(expr_t *a, expr_t *b);
extern expr_t *expr_make_div(expr_t *a, expr_t *b);
```

Simple Calculator

- Statement node

```
typedef struct stmt {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);
} stmt_t;

typedef struct stmt_list_entry {
    stmt_t *stmt;
    list_t lst;     //linked list
} stmt_list_entry_t;

extern stmt_t *stmt_make_assignment(char *id, expr_t *rhs);
extern stmt_t *stmt_make_read(char *id);
extern stmt_t *stmt_make_write(expr_t *expr);
extern stmt_t *stmt_make_compound(list_t *stmt_list_head);
```

Variable Store

- To look up and store variables

```
//
// variable store
//
typedef struct var_entry {
    char *name;
    int value;

    list_t lst; //linked list, embedded in struct
} var_entry_t;

extern void var_store_init();
extern void var_store_destroy();
extern int  var_store_get(char *name);
extern void var_store_set(char *name, int value);
```

Parser

- Functions for non-terminals
  - They return parse trees

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

static expr_t *parse_expr();
static expr_t *parse_expr_add();
static expr_t *parse_expr_mul();
static expr_t *parse_expr_factor();
```

Parser

```
static stmt_t *parse_program() {
    stmt_t *pgm = parse_stmt();
    match(TOK_EOF);
    return pgm;
}

static stmt_t *parse_stmt() {
    stmt_t *stmt = NULL;
    if(sc.token == TOK_ID)
        stmt = parse_stmt_assignment();
    else if(sc.token == TOK_READ)
        stmt = parse_stmt_read();
    else if(sc.token == TOK_WRITE)
        stmt = parse_stmt_write();
    else if(sc.token == '{')
        stmt = parse_stmt_compound();
    else
        …
    return stmt;
}
```

```
program
     -> stmt EOF

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

```
static stmt_t *parse_stmt_assignment() {
    char *id = strdup(sc.text);
    match(TOK_ID);
    match(TOK_ASSIGN);
    expr_t *expr = parse_expr();

    stmt_t *stmt = stmt_make_assignment(id, expr);

    expr->ref.release(&expr->ref);
    free(id);
    return stmt;
}
```

```
stmt_assignment
     -> ID := expr
```

```c
static stmt_t *parse_stmt_read() {
    match(TOK_READ);
    char *id = strdup(sc.text);
    match(TOK_ID);

    stmt_t *stmt = stmt_make_read(id);

    free(id);
    return stmt;
}

static stmt_t *parse_stmt_write() {
    match(TOK_WRITE);
    expr_t *expr = parse_expr();
    stmt_t *stmt = stmt_make_write(expr);

    expr->ref.release(&expr->ref);
    return stmt;
}
```

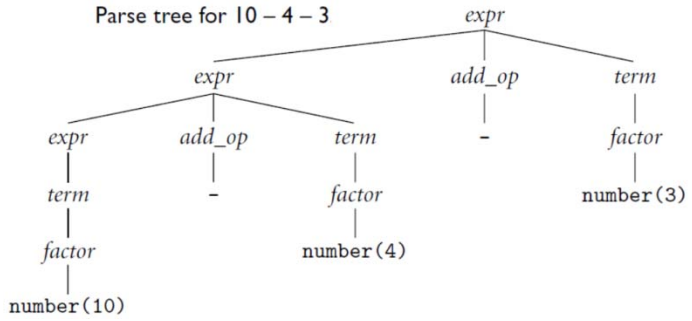
```
stmt_read
     -> READ ID

stmt_write
     -> WRITE expr
```

```
static stmt_t *parse_stmt_compound() {
    match('{');
    stmt_t *stmt = parse_stmt_list();
    match('}');

    return stmt;
}

static stmt_t *parse_stmt_list() {
    list_t *head = malloc(sizeof(list_t));
    list_init_head(head);

    while(sc.token != '}') {
        stmt_t *stmt = parse_stmt();
        stmt_list_entry_t *entry = malloc(sizeof(stmt_list_entry_t));
        entry->stmt = stmt;
        list_add_to_last(head, &entry->lst);
    }

    stmt_t *stmt = stmt_make_compound(head);
    //addref/release to the elements are skipped
    return stmt;
}
```

```
stmt_compound
     -> { stmt_list }

stmt_list
     -> stmt
     |  stmt_list stmt
```

```
static expr_t *parse_expr() {
    return parse_expr_add();
}

static expr_t *parse_expr_add() {
    expr_t *op1 = parse_expr_mul();
    expr_t *ret = op1;
    while(sc.token == '+' || sc.token == '-') {
        int tok = sc.token;
        match(sc.token);
        expr_t *op2 = parse_expr_mul();

        expr_t *tmp;
        if(tok == '+')
            tmp = expr_make_add(op1, op2);
        else
            tmp = expr_make_sub(op1, op2);

        op1->ref.release(&op1->ref);
        op2->ref.release(&op2->ref);

        ret = op1 = tmp; //left associativity: op1 is the resulting expr

    }
    return ret;
}
```

```
expr -> expr_add

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



```
static expr_t * parse_expr_mul() {
    expr_t *op1 = parse_expr_factor();
    expr_t *ret = op1;
    while(sc.token == '*' || sc.token == '/') {
        int tok = sc.token;
        match(sc.token);
        expr_t *op2 = parse_expr_factor();

        expr_t *tmp;
        if(tok == '*')
            tmp = expr_make_mul(op1, op2);
        else
            tmp = expr_make_div(op1, op2);

        op1->ref.release(&op1->ref);
        op2->ref.release(&op2->ref);

        ret = op1 = tmp; //left associativity: op1 is the resulting expr
    }
    return ret;
}
```

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

```
static expr_t *parse_expr_factor() {
    expr_t *ret = NULL;
    if(sc.token == TOK_NUM) {
        int n = atoi(sc.text);      //get int value from the scanner text
        match(TOK_NUM);
        ret = expr_make_num(n);
    }
    else if(sc.token == TOK_ID) {
        char *id = strdup(sc.text); //get var name from the scanner text
        match(TOK_ID);
        ret = expr_make_var(id);
        free(id);
    }
    else if(sc.token == '(') {
        match('(');
        ret = parse_expr();
        match(')');
    }
    return ret;
}

stmt_t *parse(char *fname) {
    open_scan(&sc, fname);
    read_token(&sc);  //load the first token for the parser
    stmt_t *stmt = parse_program();
    close_scan(&sc);
    return stmt;
}
```

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

Write Statement

```
typedef struct stmt_write {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    expr_t *expr;
} stmt_write_t;

static void exec_write(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_WRITE,
                  strmsg("tag (%d) is not OBJ_STMT_WRITE", self->ref.tag));

    stmt_write_t *stmt = (stmt_write_t*) self;
    int val = stmt->expr->eval(stmt->expr);
    printf("ans: %d\n", val);
}
```

```
static void print_write(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_WRITE,
                  strmsg("tag (%d) is not OBJ_STMT_WRITE", self->ref.tag));

    stmt_write_t *stmt = (stmt_write_t*) self;
    printf("write ");
    stmt->expr->print(stmt->expr);
    printf("\n");
}

static void release_write(refobj_t *ref) {
    ON_FALSE_EXIT(ref->tag == OBJ_STMT_WRITE,
                  strmsg("tag (%d) is not OBJ_STMT_WRITE", ref->tag));

    stmt_write_t *stmt = (stmt_write_t*) ref;
    refobj_decref(&stmt->ref);
    if(stmt->ref.cnt_ref == 0) {
        stmt->expr->ref.release(&stmt->expr->ref);
        refobj_free(&stmt->ref);
    }
}
```

```
stmt_t *stmt_make_write(expr_t *expr) {
    stmt_write_t* stmt = refobj_alloc(OBJ_STMT_WRITE,
                                      sizeof(stmt_write_t));
    stmt->ref.release  = release_write;
    stmt->exec         = exec_write;
    stmt->print        = print_write;

    stmt->expr         = expr;
    expr->ref.addref(&expr->ref);
    return (stmt_t*) stmt;
}
```

Read Statement

```
typedef struct stmt_read {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    char *id;
} stmt_read_t;

static void exec_read(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_READ,
                  strmsg("tag (%d) is not OBJ_STMT_READ", self->ref.tag));

    stmt_read_t *stmt = (stmt_read_t*) self;
    printf("enter %s : ", stmt->id);
    int val;
    ON_FALSE_EXIT(scanf("%d", &val) == 1,
                  strmsg("cannot read for %s", stmt->id));
    var_store_set(stmt->id, val); //save val at id
}
```

Assignment Statement

```
typedef struct stmt_assignment {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    char *id;
    expr_t *rhs;
} stmt_assignment_t;

static void exec_assignment(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_ASSIGN,
                  strmsg("tag (%d) is not OBJ_STMT_ASSIGN", self->ref.tag));

    stmt_assignment_t *stmt = (stmt_assignment_t*) self;
    int rhs = stmt->rhs->eval(stmt->rhs);
    var_store_set(stmt->id, rhs); //save the value of rhs at id
}
```

Compound Statement

```
typedef struct stmt_compound {
    refobj_t ref;   //ref is at the beginning of stmt
    void   ( *exec  )(struct stmt *self);
    void   ( *print )(struct stmt *self);

    list_t *stmt_list_head;
} stmt_compound_t;

static void exec_compound(stmt_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_STMT_COMPOUND,
                  strmsg("tag (%d) is not OBJ_STMT_COMPOUND", self->ref.tag));

    stmt_compound_t *stmt = (stmt_compound_t*) self;

    //execute each stmt in the list
    list_t *pos = stmt->stmt_list_head->next;
    for( ; pos != stmt->stmt_list_head; pos = pos->next) {
        stmt_list_entry_t *entry = containerof(pos, stmt_list_entry_t, lst);
        entry->stmt->exec(entry->stmt);
    }
}
```

Number Expression

```
typedef struct expr_num {
    refobj_t ref;   //first part is the same as expr_t
    int    ( *eval  )(struct expr *self);
    void   ( *print )(struct expr *self);

    int n;           //number
} expr_num_t;

static int eval_num(expr_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", self->ref.tag));
    expr_num_t *expr = (expr_num_t*) self;
    return expr->n;
}
```

```
static void print_num(expr_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", self->ref.tag));

    expr_num_t *expr = (expr_num_t*) self;
    printf("%d", expr->n);
}

static void release_num(refobj_t *ref) {
    ON_FALSE_EXIT(ref->tag == OBJ_EXPR_NUM,
                  strmsg("tag (%d) is not OBJ_EXPR_NUM", ref->tag));

    expr_num_t *expr = (expr_num_t*) ref;
    refobj_decref(&expr->ref);
    if(expr->ref.cnt_ref == 0) {
        refobj_free(&expr->ref);
    }
}
```

```
expr_t *expr_make_num(int n) {
    expr_num_t* expr  = refobj_alloc(OBJ_EXPR_NUM, sizeof(expr_num_t));
    expr->ref.release = release_num;
    expr->eval        = eval_num;
    expr->print       = print_num;

    expr->n = n;
    return (expr_t*)expr;
}
```

Variable Expression

```
typedef struct expr_var {
    refobj_t ref;   //first part is the same as expr_t
    int    ( *eval  )(struct expr *self);
    void   ( *print )(struct expr *self);

    char *id;       //variable
} expr_var_t;

static int eval_var(expr_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_VAR,
                  strmsg("tag (%d) is not OBJ_EXPR_VAR", self->ref.tag));

    expr_var_t *expr = (expr_var_t*) self;
    return var_store_get(expr->id); //look up id in the variable store
}
```

Expression for Common Operators

```
typedef struct expr_opr {
    refobj_t ref;   //first part is the same as expr_t
    int    ( *eval  )(struct expr *self);
    void   ( *print )(struct expr *self);

    int (*fp_opr)(int a, int b); //operator function
    char *str_opr;      //operator string
    struct expr *a;     //operand 1
    struct expr *b;     //operand 2
} expr_opr_t;

//common functions for binary operators
//
extern int  expr_eval_opr(expr_t *self);
extern void expr_print_opr(expr_t *self);
extern void expr_release_opr(refobj_t *ref);
extern expr_t *expr_make_opr(expr_t *a, expr_t *b, char *str_opr,
                             int (*fp_opr)(int a, int b));
```

Expression for Common Operators

```
int expr_eval_opr(expr_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_OPR,
                  strmsg("tag (%d) is not OBJ_EXPR_OPR", self->ref.tag));

    expr_opr_t *expr = (expr_opr_t*) self;
    int a = expr->a->eval(expr->a);
    int b = expr->b->eval(expr->b);
    int c = expr->fp_opr(a, b); //add, sub, mul, div, etc will be passed
    return c;
}

void expr_print_opr(expr_t *self) {
    ON_FALSE_EXIT(self->ref.tag == OBJ_EXPR_OPR,
                  strmsg("tag (%d) is not OBJ_EXPR_OPR", self->ref.tag));

    expr_opr_t *expr = (expr_opr_t*) self;
    expr->a->print(expr->a);
    printf(" %s ", expr->str_opr);
    expr->b->print(expr->b);
}
```

```
void expr_release_opr(refobj_t *ref) {
    ON_FALSE_EXIT(ref->tag == OBJ_EXPR_OPR,
                  strmsg("tag (%d) is not OBJ_EXPR_OPR", ref->tag));

    expr_opr_t *expr = (expr_opr_t*)ref;
    refobj_decref(&expr->ref);
    if(expr->ref.cnt_ref == 0) {
        expr->a->ref.release(&expr->a->ref);
        expr->b->ref.release(&expr->b->ref);
        free(expr->str_opr);
        refobj_free(&expr->ref);
    }
}
```

```
expr_t *expr_make_opr(expr_t *a, expr_t *b, char *str_opr,
                      int (*fp_opr)(int a, int b)) {
    expr_opr_t* expr  = refobj_alloc(OBJ_EXPR_OPR, sizeof(expr_opr_t));
    expr->ref.release = expr_release_opr;
    expr->eval        = expr_eval_opr;
    expr->print       = expr_print_opr;

    expr->a  = a;
    expr->b  = b;
    expr->a->ref.addref(&expr->a->ref);
    expr->b->ref.addref(&expr->b->ref);

    expr->fp_opr  = fp_opr;
    expr->str_opr = strdup(str_opr);
    return (expr_t*)expr;
}
```

Arithmetic Expression

```
//add
//
static int int_add(int a, int b) {
    return a + b;
}
expr_t *expr_make_add(expr_t *a, expr_t *b) {
    return expr_make_opr(a, b, "+", int_add);
}

//sub
//
static int int_sub(int a, int b) {
    return a - b;
}
expr_t *expr_make_sub(expr_t *a, expr_t *b) {
    return expr_make_opr(a, b, "-", int_sub);
}
…
```

Test

```
> spl.exe program1.txt
{
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
}
enter a : 2
enter b : 3
ans: 25
ans: 25
ans: 0
```

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

Assignment 10

- Extend the simple calculator interpreter with
  - If-statement
  - While-statement
  - Comparison expressions (==, !=, <=, <, >=, >)
- Implement all TODOs in
  - parser.c, stmt_if.c, stmt_while.c, expr_comp.c
- Due date: 6/14/2022

Assignment 10

```
# Makefile
TGT  = spl.exe
HSRC = common.h expr.h expr_opr.h list.h parser.h refobj.h scanner.h …
OBJS = app.o common.o list.o parser.o refobj.o scanner.o varstore.o
OBJS += expr_arith.o expr_comp.o expr_num.o expr_opr.o expr_var.o
OBJS += stmt_assignment.o stmt_compound.o stmt_if.o stmt_read.o …
LIBS = libregex.a # none for Linux and Mac; libregex.a for Windows

RM   = del  # rm for Linux and Mac, del for Windows
TRUE = cd . # true for Linux and Mac, cd . for Windows

.SUFFIXES:          # reset all suffixes
.SUFFIXES: .c .o    # suffixes to consider

# convert .c to .o
.c.o:; gcc -c $< -o $@

$(TGT): $(HSRC) $(OBJS)
    gcc -o $@ $(OBJS) $(LIBS)

clean:
    $(RM) *.o | $(TRUE)
```

Assignment 10

- Extended 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

…
```

Assignment 10

- Expected result

```
> a.exe program2.txt
{
read a
read b
while ( a != b )
{
if ( a > b )
a := a - b
else
b := b - a
}
write a
}
enter a : 24
enter b : 15
ans: 3
```

Thank you for your attention during the semester!

Any questions or comments?

Please provide your course evaluation and course outcome survey for ABET at
https://stonybrook.campuslabs.com/eval-home/
https://forms.gle/YAZag8ccNi1wgFyC6

SUNY Korea