Assignment 2: Syntax Error Recovery

Chi Chun Chen, Shaojie Wang

October 1, 2017

1 Introduction

1.1 Labor division

Chi Chun Chen: mainly on exception-based syntax error recovery, AST building and output, and `if` and `do/check` extension.
Shaojie Wang: mainly on context specific follow set.

1.2 Features

(1) Translate the code to c++ (no error in g++).

(2) Write test cases with Makefile:

To run the tests, we can use the following command lines.

```
make test<n>
make tests
```

The first line is for single sample test ($n = 01, 02...$), and the second one is to test all samples together. The output of the command is:

```
$ make test04
./parse < test04.txt > output04.txt
diff --ignore-all-space correct04.txt output04.txt
```

The first line gets input from test04.txt and redirect output to output04.txt. The second line checks the difference between correct04.txt and output04.txt except all the blank spaces. If there are no extra outputs, then the AST output (parsing) is correct[1].

[1] The given example on the course website has some inconsistency on parenthesis of the `check` statements. While other relations have parenthesis outside, the relations coming after `check` have no parenthesis. TA told us to add parenthesis for `check`.

(3) Extend the language with `if` and `do/check` statement.

(4) Implement exception-based syntax error recovery as well as context specific error recovery, as described in Section 2.3.5 on the textbook's companion site. `std::set` is used as container for FIRST sets and FOLLOW sets.

(5) Output a syntax tree with the structure suggested.

(6) Build the syntax tree as an explicit data structure, and output the tree only when the parsing is correct.

(7) Implement a static semantic check to ensure that every check statement appears inside a do statement, and every do statement has at least one check statement that is inside it and not inside any nested do.

(8) Compile to C.

2 Code blocks and demos

2.1 AST build and output

While we are parsing the program, we build a global AST on the fly. The root node of the AST is defined as:

```
struct _st_list {
    st* l_child;
    _st_list* r_child;
};
```

Every time we expand a node, the left child is a statement, and the right child is a statement list, which will recursively expand until it produces an epsilon. For relation, we call the following functions:

```
void add_child_to_null_node(bin_op* root, bin_op *child) {
    if (!root)
        return;
    else {
        if (root->l_child == NULL) {
            root->l_child = child;
        }
        else if (root->r_child == NULL) {
            root->r_child = child;
        }
        else {
            add_child_to_null_node(root->r_child, child);
        }
    }
}

void add_or_create_swap_node(bin_op* binary_op, token tok) {
    if (binary_op->type == t_none) {
        binary_op->type = tok;
        strcpy(binary_op->name, print_names[tok]);
    } else {
        bin_op* new_node = (bin_op*) malloc(sizeof(bin_op));
        new_node->type = tok;
        new_node->r_child = NULL;
        strcpy(new_node->name, print_names[tok]);
        new_node->l_child = binary_op->r_child;
        binary_op->r_child = new_node;
    }
}
```

If left child is NULL, then put the operand in the left, or if right child is NULL, then put the operand in the right, otherwise, create a new node and then swap this new node with the right child using `add_or_create_swap_node`.

Take test04.txt (example on the course website) as an example.

```
$ ./parse < test04.txt
(program
[ (read "n")
(:= "cp" (num "2"))
(do
[(check  (> (id "n") (num "0")))
(:= "found" (num "0"))
(:= "cf1" (num "2"))
(:= "cf1s" (* (id "cf1") (id "cf1")))
(do
[(check  (<= (id "cf1s") (id "cp")))
(:= "cf2" (num "2"))
(:= "pr" (* (id "cf1") (id "cf2")))
(do
[(check  (<= (id "pr") (id "cp")))
(if
 (== (id "pr") (id "cp"))
[(:= "found" (num "1"))
]
)
(:= "cf2" (+ (id "cf2") (num "1")))
(:= "pr" (* (id "cf1") (id "cf2")))
]
)
(:= "cf1" (+ (id "cf1") (num "1")))
(:= "cf1s" (* (id "cf1") (id "cf1")))
]
)
(if
 (== (id "found") (num "0"))
[(write  (id "cp"))
(:= "n" (- (id "n") (num "1")))
]
)
(:= "cp" (+ (id "cp") (num "1")))
]
)
]
)
[static semantic check]: test do has check
do [1] has check in it
do [2] has check in it
do [3] has check in it
[static semantic check]: test check in do
check [1] is in do
check [2] is in do
check [3] is in do
Pass static semantic check, compile by typing ‘make compile‘!
```

The parsing did not detect any errors, so we output the AST using the `print_program_ast` function, which recursively calls `print_stmt_list`.

2.2 Exception-based error recovery

We create customized exception classes for *stmt_list*, *stmt*, *relation* and *expr*. Take *stmt* as an example.

```
struct StatementException : public exception {
    const char * what () const throw () {
        return "Statement Exception";
    }
};
```

Then we use the `try...catch...` syntax in C++ to implement exception-based error recovery. If the `switch` matches no tokens in `case` branches, it will go into the `default` branch, print error message and throw exceptions.

```
try {
    switch
        ...
        default:
            cerr << "Deleting token: " << token_image << endl;
            throw StatementException();
} catch (StatementException se) {
    has_error = true;
    while ((input_token = scan())) {
        // recover
        if (find(first_S.begin(), first_S.end(), input_token) != first_S.end()) {
            stmt();
            input_token = scan();
            return statement;
        } else if (find(follow_S.begin(), follow_S.end(),
                        input_token) != follow_S.end()) {
            input_token = scan();
            return statement;
        } else {
            input_token = scan();

            if (input_token == t_eof)
                return statement;
        }
    }
}
```

When an exception is caught, we will scan the next token. If the next token is in the FIRST set of the current processing non-terminal, the parsing continues. Else if the next token is in the FOLLOW set, return. Otherwise, if the next token is EOF, also return.

In test08.txt, we have a syntax error that has an extra `write` after the keyword `:=`.

```
$ cat test08.txt
read a read b Y := write a * b
```

After parsing this program, we have error message:

```
$ ./parse < test08.txt
Relation Exception , line number: 1, delete: write
```

When the parser meets with `write`, it throws an exception and deletes `write` so that the parser can continue its work.

2.3 Context specific error recovery

For epsilon productions, we call the `check_for_error` function for immediate error recovery.

```
void check_for_error(const char* symbol, set<int> follow_set) {
    set<int> first_set = FIRST(symbol);
    if (!(first_set.find(input_token) != first_set.end()
          || (EPS(symbol) && follow_set.find(input_token) != follow_set.end()))) {
        has_error = true;
        cerr << "\nError at " << symbol << " around line: " << lineno
            << ", using context specific follow to settle." << endl;
        do {
            cerr << "Delete token: " << names[input_token] << endl;
            input_token = scan();

        } while (!(first_set.find(input_token) != first_set.end()
                   || follow_set.find(input_token) != follow_set.end()
                   || starter.find(input_token) != starter.end()
                   || input_token == t_eof));
    }

}
```

Here, function names is passed in to parameter `const char* symbol`, indicating that we are at *exrp_tail*, *term_tail*, or *factor_tail*. The second parameter is context specific follow set. The big `if` statement captures unexpected tokens which $token \notin FIRST(symbol)$ and $token \notin follow\_set$. The EPS(symbol) condition is dismissed because we only call `check_for_error` function where EPS(symbol) is true. The `while` loop deletes upcoming tokens until the token satisfies $token \in FIRST(symbol)$ or $token \in follow\_set$ or $token \in starter$ or $token\ is\ EOF$. The "starter" here is a set contains tokens that can begin a pair, e.g. `(`, `if`, `do`.

In text09.txt, we have many duplicate "3"s after the valid write statement.

```
$ cat test09.txt
read a write ( a + 4 * 5 ) write 3 3 3 3 3 3
```

The error message is:

```
$ ./parse < test09.txt
Error at factor_tail around line: 1, using context specific follow to settle.
Delete token: 3
Delete token: 3
Delete token: 3
Delete token: 3
Delete token: 3
```

These "3"s do not satisfy the `if` conditions, so the parser keeps deleting until the EOF is scanned.

Another example is test06.txt.

```
$ cat test06.txt
```

```
read a read b read c write ( a * ( b + c
$ ./parse < test06.txt

Error at factor\_tail around line: 1, using context specific follow to settle.
Delete token: c

Error at term\_tail around line: 1, using context specific follow to settle.
Delete token: c

Error at expr\_tail around line: 1, using context specific follow to settle.
Delete token: c

match error around line: 1 , get c, insert: rparen

Error at factor\_tail around line: 1, using context specific follow to settle.
Delete token: c

Error at term\_tail around line: 1, using context specific follow to settle.
Delete token: c

Error at expr\_tail around line: 1, using context specific follow to settle.
Delete token: c
```

In this example, we find that "c" does not satisfy the `if` conditions in *factor_tail* and we output "Delete token: c" and propagate through *term_tail* and *expr_tail*.

Last example is the one on the textbook companion site (in test13.txt).

```
$ ./parse < test13.txt

Error at factor_tail around line: 1, using context specific follow to settle.
Delete token: X
```

The result is exactly what we expect.

2.4 Static semantic check

We implement a static semantic that makes sure that every `check R` is in the block of `do` statement. It also makes sure that every `do` statement contains at least one `check` statement. The messages are as follows.

```
[static semantic check]: test do has check
do [1] has check in it
do [2] has check in it
do [3] has check in it
[static semantic check]: test check in do
check [1] is in do
check [2] is in do
check [3] is in do
Pass static semantic check, compile by typing ‘make compile‘!
```

If there is no error in the semantic check, we translate the AST into C code.

2.5 Compile to C

With the already constructed AST, it is very simple to generate a C program. We just need to traverse the AST, use a big `switch...case...` statement to translate the current token into C style, and output. The generated C program is written into the file "test.c". We can test the test.c use "`make compile`". Use test04.txt as an example.

```
$ make compile
gcc test.c
./a.out
10
2
3
5
7
11
13
17
19
23
29
```

We can get the first 10 prime numbers by inputting "10".