KOI: Exercise 4: Top-down parsers, grammar transformations

You can do this exercise in any size group you wish. Don't worry if you find some of the questions difficult to answer, since they may be intended to initiate a discussion rather than have a single correct answer.

Preparations

Before this exercise you should have:

- Watched lecture 4 (divided into two Youtube videos, see this list), or read the corresponding course materials

Question 1

Here is a grammar. There are three terminals: **a**, **b** and **c**. There are three non-terminals: *S*, *T* and *U*. The start symbol is *S*.

S -> **a b** *T* | **b a** *T*
T -> **a b** *U* | **a a** *U*
U -> **a b c**

Below is the program **badparser-1**. It is an attempt to write a parser that implements the grammar above.

a) What is the language that this grammar defines?

b) Unfortunately, there is a problem with the parser below. Explain what the problem is.

c) We *could* use various programming tricks to fix the parser, but instead we will use a more general method, using a *grammar transformation*. What is the transformation we should use here?

d) Perform the grammar transformation, and show the new, transformed grammar.

e) Modify the program so the parser implements the new grammar. Verify that it can now correctly parse the language.

(Some helpful hints: The program only understands the tokens, or terminals, **a**, **b** and **c**. Finish the input with **EOF**, which in Linux is entered using **CTRL-D** on a separate line.a All other input, such as **d**, is ignored. You can compile and run the program if you want, but you don't have to.)

Download

```
// A simple (and wrong) parser: badparser-1.c

#include <stdio.h>
#include <ctype.h>
```

```
#include <stdlib.h>

int lookahead;

void error(void) {
    printf("Syntax error: unexpected token '%c'\n", lookahead);
    exit(EXIT_FAILURE);
}

// A simple scanner.
// Returns one of the tokens 'a', 'b' or 'c', or EOF. Other input is ignored.
int scan(void) {
    int input;
    while ((input = getchar()) != 'a' && input != 'b' && input != 'c' && input != EOF)
        ;
    return input;
}

void match(int expected) {
    if (lookahead == expected)
        lookahead = scan();
    else
        error();
}

// These are the non-terminals, calling each other recursively
void S(void), T(void), U(void);

void S(void) {
    if (lookahead == 'a') {
        match('a'); match('b'); T();
    }
    else if (lookahead == 'b') {
        match('b'); match('a'); T();
    }
    else {
        error();
    }
}

void T(void) {
    if (lookahead == 'a') {
        match('a'); match('b'); U();
    }
    else if (lookahead == 'a') {
        match('a'); match('a'); U();
    }
    else {
        error();
    }
}

void U(void) {
    match('a'); match('b'); match('c');
}

int main(void) {
    printf("Badparser 1. Enter input. End with EOF (CTRL-D on Linux).\n");
    lookahead = scan(); // To get started, so we have something to compare to
    S();
    if (lookahead != EOF)
        error();
    printf("Done!\n");
    return EXIT_SUCCESS;
}
```

Question 2

Here is a grammar. There are three terminals: **a**, **b** and **c**. There are two non-terminals: *S* and *T*. The start symbol is *S*.

S -> **a b** *T* **c**
T -> *T* **a** | **b**

Below is the program **badparser-2**. It is an attempt to write a parser that implements the grammar above.

a) What is the language that this grammar defines?

b) Unfortunately, there is a problem with the parser below. Explain what the problem is.

c) We *could* use various programming tricks to fix the parser, but instead we will use a more general method, using a *grammar transformation*. What is the transformation we should use here?

d) Perform the grammar transformation, and show the new, transformed grammar.

e) Modify the program so the parser implements the new grammar. Verify that it can now correctly parse the language.

Download

```
// A simple (and wrong) parser: badparser-2.c

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error(void) {
    printf("Syntax error: unexpected token '%c'\n", lookahead);
    exit(EXIT_FAILURE);
}

// A simple scanner.
// Returns one of the tokens 'a', 'b' or 'c', or EOF. Other input is ignored.
int scan(void) {
    int input;
    while ((input = getchar()) != 'a' && input != 'b' && input != 'c' && input != EOF)
        ;
    return input;
}

void match(int expected) {
    if (lookahead == expected)
        lookahead = scan();
    else
        error();
}

// These are the non-terminals, calling each other recursively
void S(void), T(void);

void S(void) {
    match('a'); match('b'); T(); match('c');
}

void T(void) {
    if (lookahead == 'b') {
        T(); match('a');
```

```
    }
    else if (lookahead == 'b') {
        match('b');
    }
    else {
        error();
    }
}

int main(void) {
    printf("Badparser 2. Enter input. End with EOF (CTRL-D on Linux).\n");
    lookahead = scan(); // To get started, so we have something to compare to
    S();
    if (lookahead != EOF)
        error();
    printf("Done!\n");
    return EXIT_SUCCESS;
}
```

Question 3

Here is a grammar. There are three terminals: **a**, **b** and **c**. There are two non-terminals: *S* and *T*. The start symbol is *S*.

S -> **a b** *T* **c**
T -> **a b c** | *U*
U -> **a b c**

Below is the program **badparser-3**. It is an attempt to write a parser that implements the grammar above.

a) What is the language that this grammar defines?

b) There is a problem with this *grammar*. Explain what the problem is.

c) Does the parser below correctly parse the language?

d) Try to fix the problem in the grammar, and show the new grammar.

e) Modify the program so the parser implements the new grammar. Verify that it can correctly parse the language.

Download

```
// A simple (but is it wrong?) parser: badparser-3.c

#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int lookahead;

void error(void) {
    printf("Syntax error: unexpected token '%c'\n", lookahead);
    exit(EXIT_FAILURE);
}

// A simple scanner.
// Returns one of the tokens 'a', 'b' or 'c', or EOF. Other input is ignored.
int scan(void) {
    int input;
    while ((input = getchar()) != 'a' && input != 'b' && input != 'c' && input != EOF)
        ;
        return input;
    }

    void match(int expected) {
        if (lookahead == expected)
            lookahead = scan();
        else
            error();
    }

    // These are the non-terminals, calling each other recursively
    void S(void), T(void), U(void);

    void S(void) {
        match('a'); match('b'); T(); match('c');
    }

    void T(void) {
        if (lookahead == 'a') {
            match('a'); match('b'); match('c');
        }
        else if (lookahead == 'a') {
            U();
        }
        else {
            error();
        }
    }

    void U(void) {
        match('a'); match('b'); match('c');
    }

    int main(void) {
        printf("Badparser 3. Enter input. End with EOF (CTRL-D on Linux).\n");
        lookahead = scan(); // To get started, so we have something to compare to
        S();
        if (lookahead != EOF)
            error();
        printf("Done!\n");
        return EXIT_SUCCESS;
    }
```

Question 4

In the question above, the problem was that the given grammar was ambiguous. Assuming that you managed to write a new, unambiguous grammar, it still describes exactly the same language. If you modified the given parser so it implements your new grammar, it still understands exactly the same language as the old parser. If it's all the same language, why does it matter if the grammar is ambiguous?

There are some solutions to some of the questions, but try to solve them yourself first.

Thomas Padron-McCarthy (thomas.padron-mccarthy@oru.se) September 14, 2023