CSE341
Programming Languages
(Fall 2023)
Homework #3 Documentation

Muhammed Yasir Güneş

210104004079

TURKISH

Syntax analysis yapmak için parsing technique olarak “Recursive Descent Parser” metodunu kullandım. Bu method non-terminaller sembolleri recursive function olarak tanımlar.

NON-TERMINAL SYMBOLS:

- START
- EXPR
- FUNCT

TERMINAL SYMBOLS:

- OP_OP
- OP_CP
- KW_EXIT
- KW_IF
- VALUEF
- IDENTIFIER
- Etc.

Program START non-terminal symbol’ü ile başlar. Burada girilen inputun hangi tokene uyduğuna göre devam edilir.

Start şu grammar kurallarına sahiptir:

START:

EXPR

| FUNCT

| OP_OP KW_EXIT OP_CP (this is for exitting the program (exit))

START inputun hangi sembol ile başladığını bulmak için isExpressionStart ve isFunctionStart fonksiyonlarını kullanır. Bu fonksiyonların mantığını anlayabilmek için önce şunu anlayalım:

gpp_interpreter fonksiyonunun içinde DFA çalışır ve inputu tokenlere ayırır. Bu tokenlere ayırırken de bu tokenleri *tokens_as_symbols* global listeye ekler. Aynı zamanda tokens_copy adında bir listeye de tokenlerin sade input halini kaydeder, bunun amacı örneğin sembolü VALUEF olan bir değişkenin 4b1 değerine ulaşabilmektir.

Bu *tokens_as_symbols* ve tokens_copy listelerini işlemek ve dolaşmak için de *lookahead* global değişkenini, getNextToken ve match fonksiyonunu kullanıyorum.

getNextToken tokens_copy ve *tokens_as_symbols* listelerini bir kez pop eder. Ve *tokens_as_symbols* listesinden pop ettiği değeri döndürür. Bu dönen değeri de match fonksiyonu ile *lookahead* değişkenine atıyoruz bu sayede sonraki sembollere erişiyoruz.

Şimdi isExpressionStart ve isFunctionStart fonksiyonlarına geri dönelim. Bunlar *tokens_as_symbols* ve tokens_copy listelerini iterate ederek hangi non-terminal symbole gitmeleri gerektiğini anlar anladıktan sonra *tokens_as_symbols* ve tokens_copy listelerini Start fonksiyonun başındaki önceki haline geri döndürür ki en baştan işlenmeye başlanabilsin. Daha sonra da eğer isFunctionStart is Funct recursive fonksiyonunu çağırır veya eğer isExpressionStart ise EXPR recursive fonksiyonunu çağırır değilse OP_OP KW_EXIT OP_CP olup olmadığını kontrol eder o da değilse syntax error döndürür.

EXPR fonksiyonunu açalım.
EXPR fonksiyonu şu grammar rules'lara sahiptir.

EXPR:

OP_OP OP_PLUS EXPR EXPR OP_CP

| OP_OP OP_MINUS EXPR EXPR OP_CP

| OP_OP OP_MULTIPLY EXPR EXPR OP_CP

| OP_OP OP_DIVIDE EXPR EXPR OP_CP

| OP_OP KW_IF EXPR EXPR EXPR OP_CP

| OP_OP IDENTIFIER OP_CP

| OP_OP IDENTIFIER EXPR OP_CP

| OP_OP IDENTIFIER EXPR EXPR OP_CP

| VALUEF

| IDENTIFIER

EXPR fonksiyonunda hesaplama kısımları için add_valuef, divide_valuef, subtract_valuef, multiply_valuef fonksiyonu kullanıyorum. Bu fonksiyonlar örneğin 4b1 değerinin 4 ve 1 değerlerini ayırıp gerekli işlemleri yapıp bir sonuç elde eder, sonucu sadeleştirir ve sonucu döndürür.

If için eğer ilk EXPR 0b1'den farklıysa ikinci EXPR'i döndürür. 0b1 ise üçüncü EXPR'i döndürür.

OP_OP IDENTIFIER.. kısımları fonksiyonları çağırmak içindir. Fonksiyonlar için sistem şu şekilde: Fonksiyonlar için bir class tanımladım *defined_function* adında. Her tanımlanan fonksiyon için bir instance oluşturuyorum ve defined_functions listesine bu instance'i ekliyorum. *defined_function* function_name, function_arguments, function_body değişkenlerine sahiptir. Fonksiyonları çağırırken fonksiyonu defined_functions listesinde arıyorum eğer bulabildiysem fonksiyonun body kısmını *tokens_as_symbols* kısmına ekliyorum ve o şekilde fonksiyonu işleme sokuyorum.

FUNCT fonksiyonunu açalım.
FUNCT fonksiyonu şu grammar rules'lara sahiptir.

FUNCT:

OP_OP KW_DEF IDENTIFIER EXPR OP_CP

| OP_OP KW_DEF IDENTIFIER IDENTIFIER EXPR OP_CP

| OP_OP KW_DEF IDENTIFIER IDENTIFIER IDENTIFIER EXPR OP_CP

Bu kısım yeni fonksiyon tanımlamak içindir. Yukarıda anlattığım şekilde fonksiyonu tanımlar.

gpp_interpret fonksiyonu driver fonksiyondur. Opsiyonel olarak bir file alabilir. Eğer file varsa file'daki her line için işlem yapar ve sonucu ekrana bastırır. Eğer file verilmemişse kullanıcıdan satır satır kullanıcı (exit) girene kadar input alır ve sonucu ekrana bastırır.

ENGLISH

I used the "Recursive Descent Parser" method as a parsing technique to perform syntax analysis. This method defines non-terminals symbols as a recursive function.

NON-TERMINAL SYMBOLS:

- START
- EXPR
- FUNCT

TERMINAL SYMBOLS

- OP_OP
- OP_CP
- KW_EXIT
- KW_IF
- VALUEF
- IDENTIFIER
- Etc.

The program starts with the START non-terminal symbol. The program continues according to which token matches the input entered here.

Start has the following grammar rules:

START:

EXPR

| FUNCT

| OP_OP KW_EXIT OP_CP (this is for exitting the program (exit))

START uses the isExpressionStart and isFunctionStart functions to find out which symbol the input starts with. To understand the logic of these functions, let's first understand this:

DFA runs inside the gpp_interpreter function and splits the input into tokens. While separating it into tokens, it adds these tokens to the *tokens_as_symbols* global list. At the same time it saves the plain input version of the tokens in a list called tokens_copy, for example to get the value 4b1 of a variable whose symbol is VALUEF.

To manipulate and traverse these *tokens_as_symbols* and tokens_copy lists, I use the *lookahead* global variable, getNextToken and the match function.

getNextToken pops the tokens_copy and *tokens_as_symbols* lists once. And returns the popped value from the *tokens_as_symbols* list. We assign this return value to the *lookahead* variable with the match function so that we can access the next symbols.

Now let's go back to the isExpressionStart and isFunctionStart functions. They iterate through the *tokens_as_symbols* and tokens_copy lists to see which non-terminal symbol they should go to, and then return the *tokens_as_symbols* and tokens_copy lists to their previous state at the start of the Start function so that they can be processed from the beginning. Then it calls the Funct recursive function if isFunctionStart is Funct recursive, or the EXPR recursive function if isExpressionStart is EXPR recursive, and if not, it checks for OP_OP KW_EXIT OP_CP and returns a syntax error if not.

Let's open the EXPR function.

The EXPR function has the following grammar rules.

EXPR

OP_OP OP_PLUS EXPR EXPR OP_CP

| OP_OP OP_MINUS EXPR EXPR EXPR OP_CP

| OP_OP OP_MULTIPLY EXPR EXPR OP_CP

| OP_OP OP_DIVIDE EXPR EXPR OP_CP

| OP_OP KW_IF EXPR EXPR EXPR EXPR OP_CP

| OP_OP IDENTIFIER OP_CP

| OP_OP IDENTIFIER EXPR OP_CP

| OP_OP IDENTIFIER EXPR EXPR OP_CP

| VALUEF

| IDENTIFIER

I use add_valuef, divide_valuef, divide_valuef, subtract_valuef, multiply_valuef functions for the calculation parts of the EXPR function. These functions, for example, separate the 4 and 1 values of 4b1, perform the necessary operations, obtain a result, simplify the result and return the result.

For If, if the first EXPR is different from 0b1, it returns the second EXPR. If 0b1, it returns the third EXPR.

OP_OP IDENTIFIER... is for calling functions. The system for functions is as follows: I defined a class for functions called *defined_function*. I create an instance for each defined function and add this instance to the defined_functions list. *defined_function* has variables function_name, function_arguments, function_body. When calling functions, I look for the function in the defined_functions list and if I find it, I add the body part of the function to *tokens_as_symbols* and process the function that way.

Let's open the FUNCT function.

The FUNCT function has the following grammar rules.

FUNCT

OP_OP KW_DEF IDENTIFIER EXPR OP_CP

| OP_OP KW_DEF IDENTIFIER IDENTIFIER EXPR OP_CP

| OP_OP KW_DEF IDENTIFIER IDENTIFIER IDENTIFIER EXPR OP_CP

This section is for defining a new function. It defines the function as I described above.

The gpp_interpret function is the driver function. It can optionally take a file. If there is a file, it performs operations for each line in the file and prints the result on the screen. If no file is given, it takes input from the user line by line until the user enters (exit) and prints the result on the screen.