Институт приборостроения, автоматизации и информационных технологий

Кафедра информационных систем и цифровых технологий

Дисциплина «Проектирование трансляторов»

Отчет к лабораторной работе № 2

«Построение лексического

и синтаксического анализаторов для языка LispKit»

Выполнили:

Василения Иван Валерьевич

Мельниченко Артём Олегович

Приняла:

Конюхова О. В.

Орёл, 2025г

Листинг файла flex.l

%{

#include <stdio.h>

#include <stdlib.h>

#include "y.tab.h"

void yyerror(const char \*s);

int scmp(char\* str) {

if (strcasecmp(yytext, str) == 0) return 1;

return 0;

}

%}

delim [ \t\n\r]+

ws {delim}+

letter [a-zA-Z]

digit [0-9]

integer [-]?{digit}+

id {letter}({letter}|{digit})\*

s\_atom {id}

d\_atom {integer}

%option noyywrap

%option case-insensitive

%%

{ws} { /\* Игнорируем пробелы \*/ }

{s\_atom} { if (scmp("QUOTE") == 1) return QUOTE;

if (scmp("CAR") == 1) return CAR;

if (scmp("CDR") == 1) return CDR;

if (scmp("CONS") == 1) return CONS;

if (scmp("ATOM") == 1) return ATOM\_PREDICATE;

if (scmp("EQUAL") == 1) return EQUAL;

if (scmp("ADD") == 1) return ADD;

if (scmp("SUB") == 1) return SUB;

if (scmp("MUL") == 1) return MUL;

if (scmp("DIVE") == 1) return DIVE;

if (scmp("REM") == 1) return REM;

if (scmp("LE") == 1) return LE;

if (scmp("COND") == 1) return COND;

if (scmp("LAMBDA") == 1) return LAMBDA;

if (scmp("LET") == 1) return LET;

if (scmp("LETREC") == 1) return LETREC;

yylval.str = strdup(yytext); return S\_ATOM; }

{d\_atom} { yylval.num = atoi(yytext); return D\_ATOM; }

"(" { return LP; }

")" { return RP; }

<<EOF>> { return 0;}

. { printf("Ошибка: неожиданный символ: %d %s\n", (int)yytext[0], yytext); exit(1); }

%%

Листинг файла grammar.y:

%{

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "attr\_checker.h"

%}

%union {

char\* str;

int num;

}

%token LP RP QUOTE CAR CDR CONS ATOM\_PREDICATE EQUAL ADD SUB MUL DIVE REM LE COND LAMBDA LET LETREC

%token <str> S\_ATOM

%token <num> D\_ATOM

%%

goal: S\_EXPR

{

printf("\033[32mВыражение принадлежит языку\033[0m\n");

}

;

S\_EXPR: ATOM

| LP QUOTE S\_EXPR RP

{

printf("QUOTE + S\_EXPR\n");

}

| LP CAR S\_EXPR RP

{

printf("CAR + S\_EXPR\n");

}

| LP CDR S\_EXPR RP

{

printf("CDR + S\_EXPR\n");

}

| LP CONS S\_EXPR S\_EXPR RP

{

printf("CONS + S\_EXPR\n");

}

| LP ATOM\_PREDICATE S\_EXPR RP

{

printf("ATOM + S\_EXPR\n");

}

| LP EQUAL S\_EXPR S\_EXPR RP

{

printf("EQUAL + S\_EXPR + S\_EXPR\n");

}

| LP ADD S\_EXPR S\_EXPR RP

{

printf("ADD + S\_EXPR + S\_EXPR\n");

}

| LP SUB S\_EXPR S\_EXPR RP

{

printf("SUB + S\_EXPR + S\_EXPR\n");

}

| LP MUL S\_EXPR S\_EXPR RP

{

printf("MUL + S\_EXPR + S\_EXPR\n");

}

| LP DIVE S\_EXPR S\_EXPR RP

{

printf("DIVE + S\_EXPR + S\_EXPR\n");

}

| LP REM S\_EXPR S\_EXPR RP

{

printf("REM + S\_EXPR + S\_EXPR\n");

}

| LP LE S\_EXPR S\_EXPR RP

{

printf("LE + S\_EXPR + S\_EXPR\n");

}

| LP COND S\_EXPR S\_EXPR S\_EXPR RP

{

printf("COND + S\_EXPR + S\_EXPR + S\_EXPR\n");

}

| LP LAMBDA LP PARAM RP S\_EXPR RP

{

printf("LAMBDA + PARAM + S\_EXPR\n");

}

| LP LET S\_EXPR PAIR\_LIST RP

{

printf("LET + S\_EXPR + PAIR\_LIST\n");

}

| LP LETREC S\_EXPR PAIR\_LIST RP

{

printf("LETREC + S\_EXPR + PAIR\_LIST\n");

}

| LP LIST\_S\_EXPR RP

;

PAIR:

LP VAR S\_EXPR RP

;

PAIR\_LIST:

PAIR

| PAIR\_LIST PAIR

;

PARAM:

PARAM\_REC

;

PARAM\_REC:

S\_ATOM

| PARAM\_REC S\_ATOM

;

VAR:

S\_ATOM

;

ATOM:

S\_ATOM

{

printf("S\_ATOM %s\n", $1);

}

| D\_ATOM

{

printf("D\_ATOM %d\n", $1);

}

;

LIST\_S\_EXPR:

/\* epsilon - пустой список \*/

| S\_EXPR\_LIST

;

S\_EXPR\_LIST:

S\_EXPR

| S\_EXPR\_LIST S\_EXPR

;

%%

int main (void) {

return yyparse();

}

void yyerror(const char\* s) {

fprintf(stderr, "Ошибка: %s\n", s);

}

Контрольные вопросы:

1. Понятие и правила построения S-выражений.

S-выражение (Symbolic Expression) — это основа синтаксиса языков семейства Lisp. Оно представляет собой способ записи данных и кода.

Правила построения:

1. Атом является S-выражением.
2. Последовательность S-выражений является S-выражением

2. Лексические единицы языка LispKit.

Лексические единицы (токены) языка LispKit включают:

* Ключевые слова и операторы: LAMBDA, LET, LETREC, IF, COND, CAR, CDR, CONS, ADD, SUB, MUL, DIV, REM, QUOTE.
* Скобки: (, ), [, ].
* Символьные атомы (идентификаторы): последовательности символов, не содержащие пробелов и специальных символов, используемых в других токенах (например, x, my-var, fact).
* Числовые атомы (целочисленные литералы): последовательности цифр, возможно, с отрицательным знаком (например, 42, -10).

3. Синтаксические конструкции языка LispKit.

Синтаксис LispKit строится на S-выражениях. Основные синтаксические конструкции:

1. Применение функции: (функция аргумент1 ... аргументN), где функция — выражение, вычисляющее функцию, а аргументы — другие выражения.
   * Пример: (ADD 5 7), (CAR my-list).
2. Специальные формы:
   * Лямбда-абстракция: (LAMBDA (параметр1 ... параметрN) тело).
   * Лексическое связывание:
     + (LET ((var1 val1) ... (varN valN)) тело)
     + (LETREC ((var1 val1) ... (varN valN)) тело) (для рекурсивных определений).
   * Условные ветвления: (COND (условие1 результат1) ... (else результатN)).
   * Цитирование: (QUOTE выражение)

4. Совместная работа генераторов лексического и синтаксического анализаторов TP LEX и TP YACC.

Совместная работа:

1. Лексер (от TP LEX) читает исходный код и, сопоставляя символы с регулярными выражениями, генерирует токены.
2. Для каждого распознанного токена лексер возвращает парсеру (от TP YACC) числовой код токена и его атрибут (например, значение числа или имя идентификатора).
3. Парсер, получая поток токенов, использует грамматические правила для построения дерева разбора (AST). Он вызывает лексер для получения следующего токена, когда ему нужно продвинуться по входному потоку.
4. Лексер и парсер взаимодействуют через общую функцию yylex(), которую вызывает парсер.

5. Понятие прохода. Взаимодействие блоков анализатора, построенного с помощью TP LEX и TP YACC.

Проход (pass) — это этап компиляции или интерпретации, во время которого программа обрабатывается от начала до конца.

В контексте анализатора, построенного на TP LEX и TP YACC, лексический и синтаксический анализ часто объединяются в один проход.

Взаимодействие блоков в этом проходе:

1. Парсер (YACC) является ведущим модулем. Он начинает выполнение, вызывая функцию yyparse().
2. Когда парсеру нужен следующий токен для принятия решения, он вызывает функцию yylex(), которая генерируется лексером (LEX).
3. Лексер читает символы из входного файла, пропускает пробелы и комментарии, и как только находит последовательность, соответствующую регулярному выражению, он:
   * Определяет тип токена.
   * Возвращает этот тип парсеру.
   * Может также поместить значение токена (атрибут) в глобальную переменную yylval.
4. Парсер получает код токена, использует его и значение yylval для построения дерева разбора и продолжает свою работу, пока не встретит конец файла или синтаксическую ошибку.

Таким образом, анализ происходит "на лету": лексер поставляет токены парсеру по мере необходимости в рамках одного последовательного прохода по исходному коду.

6. Напишите определения следующих функций на языке расширенного лямбда-исчисления:

6.1. Функция, возвращающая четные элементы из исходного списка.

(LETREC (MMUL (QUOTE (3 4 5 6)))

(MMUL (LAMBDA (X)

(COND (EQUAL X (QUOTE ()))

X

(cond

(EQUAL (QUOTE 0) (rem (car X) 2))

(CONS

(car X)

(MMUL (CDR X))

)

(MMUL (CDR X))

)

)

)

)

)

6.2. Функция, выбирающая элементы из списка, которые не являются атомами.

(LETREC (ISATOMS (QUOTE (3 4 5 6)))

(ISATOMS (LAMBDA (X)

(COND (EQUAL X (QUOTE ()))

X

(cond

(ATOM (CAR X) )

(ISATOMS (CDR X))

(CONS

(car X)

(ISATOMS (CDR X))

)

)

)

)

)

)

6.3. Функция реверсирования списка.

(LETREC (REV (QUOTE (1 2 3 4 5)) (QUOTE ()) )

(REV

(LAMBDA (X AC)

(COND

(EQUAL X (QUOTE ()))

AC

(REV (CDR X) (CONS (CAR X) AC))

)

)

)

)