|  |  |  |
| --- | --- | --- |
| Теория алгоритмов и формальных языков  Лабораторная работа №3  Язык программирования: Кумир | Студент | Окороджи Дюк  Макросян Карен |
| Группа | ПРИН — 367 |
| Дата |  |
| Преподаватель | Ерофеев А.А. |
| Оценка |  |

Перечень возможностей языка

|  |  |  |  |
| --- | --- | --- | --- |
| Возможность | Вес | Особенности в языке | Кто реализует |
| Локальные переменные встроенных типов данных :   * целые числа (один тип), * символы * строки   Выражения с использованием локальных переменных, арифметических операций (4 вида), операций сравнения и присваивания | 4 min | Возведение в степень – а\*\*б | Дюк |
| Одномерные массивы (или контейнеры),  операция доступа к элементу массива | 3 min |  | Карен |
| Числовые (целые числа), символьные и строковые константы (литералы) с поддержкой всех видов констант и служебных последовательностей символов | 2 min |  | Карен |
| Управляющие структуры: развилки | 3 min |  | Дюк |
| Управляющие структуры: циклы | 3 min | Цикл нцкц | Карен |
| Функции (процедуры) | 3 min |  | Дюк |
| Классы/функции для работы с консолью (ввод/вывод базовых типов данных) | 2 min |  | Карен |

Грамматика реализуемого языка в форме Бэкуса-Науэра.

program:

ALG ';' func\_pref\_list NACH ';' KON ';'

| ALG ';' func\_pref\_list NACH ';' stmt\_list KON ';'

| program ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';'

| program TYPE ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';'

;

stmt\_list:

stmt

| stmt\_list stmt

;

stmt:

expr ';'

| if\_stmt ';'

| loop\_stmt ';'

| switch\_stmt ';'

| TYPE id\_list ';'

| VVOD id\_list ';'

| VIVOD expr\_list ';'

;

if\_stmt:

IF expr THEN stmt\_list VSE

| IF expr THEN stmt\_list ELSE stmt\_list VSE

;

switch\_stmt:

VIBOR switch\_lines VSE

| VIBOR switch\_lines ELSE stmt\_list VSE

;

switch\_lines:

PRI expr ':' stmt\_list

| switch\_lines PRI expr ':' stmt\_list

;

loop\_stmt:

NZ DLYA ID OT CELVAL DO CELVAL stmt\_list KZ

| NZ DLYA ID OT CELVAL DO CELVAL SHAG CELVAL stmt\_list KZ

| NZ stmt\_list KZ

| NZ POKA expr ';' stmt\_list KZ

| NZ CELVAL RAZ stmt\_list KZ

| NZ stmt\_list KZ\_PRI expr

;

expr:

expr '+' expr

| expr '-' expr

| expr '\*' expr

| expr '/' expr

| '-' expr %prec UMINUS

| expr POW expr

| '(' expr ')'

| expr NOTEQ expr

| expr OR expr

| expr AND expr

| DA

| NET

| NS

| expr '>' expr

| expr '<' expr

| expr LESSEQ expr

| expr MOREEQ expr

| expr '=' expr

| ID ASSIGN expr

| ID '[' CELVAL ']' ASSIGN expr

| CELVAL

| VESHVAL

| ID '[' CELVAL ']'

| ID

;

expr\_list:

expr

| expr\_list ',' expr

;

id\_list:

ID

| ID '[' CELVAL ':' CELVAL ']'

| id\_list ',' ID

| id\_list ',' ID '[' CELVAL ':' CELVAL ']'

;

arg\_list:

/\* empty \*/

| arg\_listE

;

arg\_listE:

ARGTYPE TYPE ID

| ARGTYPE TYPE ID '[' CELVAL ']'

| TYPE ID

| TYPE ID '[' CELVAL ']'

| arg\_listE ',' ARGTYPE TYPE ID

| arg\_listE ',' ARGTYPE TYPE ID '[' CELVAL ']'

| arg\_listE ',' TYPE ID

| arg\_listE ',' TYPE ID '[' CELVAL ']'

;

func\_pref\_list:

/\* empty \*/

| func\_pref\_listE

;

func\_pref\_listE:

FUNCPREF ';'

| FUNCPREF expr ';'

| func\_pref\_listE FUNCPREF

;

%%

void yyerror(char const \*s)

{

printf("%s",s);

}

Перечень структур, использующихся для хранения в памяти абстрактного синтаксического дерева

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct ProgramInfo  {  struct StmtListInfo \* prefix;  struct PrefListInfo \* funcpreflist;  struct StmtListInfo \* stmtlist;  struct FuncListInfo \* functions;  }; | Корневая структура, содержащая в себе все остальные структуры программы. Описывает символ грамматики **program.** | |
| prefix  funcpreflist  stmtlist  functions | Префекс  Cписок Функциий префекс  Список Statements  Cписок Функции |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct StmtListInfo  {  struct StmtInfo \* first;  struct StmtInfo \* last;  }; | Список объявлений. Структура описывает символ грамматики StmtListInfo**.** | |
| first | Указатель на первое объявление |
| last | Указатель на последнее объявление |
|

|  |  |  |
| --- | --- | --- |
| Перечисление | Описание перечисления | |
| enum LoopType  {  \_\_DLYA,  \_\_DLYASHAG,  \_\_PROST,  \_\_PRI,  \_\_POKA,  \_\_RAZ  }; | Перечисление описывает тип объявления для структуры LoopType**.** | |
| DLYA | Тип цикул ДЛЯ |
| DLYASHAG | Тип Цикул От До |
| PROST  PRI  POKA  RAZ | Тип Цикул просто  Тип Цикул ПРИ  Тип цикул ПОКА  Тип цикул РАЗ |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Структура | | Описание структуры | | |
| struct StmtInfo  {  enum StmtType type;  struct ExprInfo \* expr;  struct IfStmtInfo \* ifstmt;  struct LoopStmtInfo \* loopstmt;  struct SwitchStmtInfo \* switchstmt;  int vartype;  struct IdListInfo \* idlist;  char \* funcname;  struct ExprListInfo \* exprlist;  struct StmtInfo \* next;  }; | | Спецификация переменной. Структура описывает символ грамматики **var\_specification.** | | |
| type | Список идентификаторов с типом | |
| expr | Список выражений | |
| ifstmt | Список цикул для | |
| loopstmt | Список цикул Выбор | |
|  | switchstmt | | | Список цикул Нц Кц | |
| idlist | | | Список ID | |
| funcname | | | Указателья на имя функций | |
| next | | | Указателья на следущ statement | |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct StmtListInfo  {  struct StmtInfo \* first;  struct StmtInfo \* last;  }; | Структура описывает символ грамматики StmtListInfo**.** | |
| first | Указателья на первой |
| last | Указателья на последний |
|
|

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct PrefListEInfo  {  int type;  struct ExprInfo \* expr;  struct PrefListEInfo \* next;  }; | Список спецификаций префекс. Структура описывает символ грамматики PrefListEInfo**.** | |
| Expr | Указатель на выражений |
| Next | Указатель на cледущ Префекс |
| Type | Размер списка |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct PrefListInfo  {  struct PrefListEInfo \* preflist;  }; | Список спецификаций префекс. Структура описывает символ грамматики PrefListInfo**.** | |
|
| preflist | Список префекс |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| enum ArgListType  {  \_\_ARGTYPEID,  \_\_ARGTYPEIDARR,  \_\_TYPEID,  \_\_TYPEIDARR  }; |  | |
| ARGTYPEID | Идентификатор функции |
| ARGTYPEIDARR | Идентификатор  Тип ID функции |
| TYPEID  TYPEIDARR |  |

|  |  |  |
| --- | --- | --- |
| Перечисление | Описание перечисления | |
| enum IdListType  {  \_\_ID,  \_\_ARR  }; | Перечисление описывает тип структуры IdListType**.** | |
| ID | Тип задан идентификатором |
| ARR | Вещественное число |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| enum StmtType  {  \_\_EXPR,  \_\_IF,  \_\_SWITCH,  \_\_LOOP,  \_\_TYPE,  \_\_FUNCCALL,  \_\_VVOD,  \_\_VIVOD  }; | Перечисление описывает тип структуры StmtType**.** | |
| EXPR | Тип выражения |
| IF  SWITCH  LOOP  TYPE  FUNCCALL | Условие Если  Условие При  Условие Цикул  Тип задан идентификатором  Вызов функций |

|  |  |  |
| --- | --- | --- |
| Перечисление | Описание перечисления | |
| enum ExprType  {  \_ADD,  \_SUB,  \_MUL,  \_DIV,  \_UNARSUB,  \_POW,  \_NOTEQ,  \_OR,  \_AND,  \_DA,  \_NET,  \_NS,  \_GREAT,  \_LESS,  \_GREATEQ,  \_LESSEQ,  \_EQ,  \_IDASSIGN,  \_ARRASSIGN,  \_CELVAL,  \_VESHVAL,  \_ARRID,  \_ID  }; | Перечисление описывает тип выражения (тип структуры **Expression** и **PrimaryExpression**). | |
| ADD | Оператор сложение |
| SUB | Оператор Вычисление |
| MUL | Оператор умножение |
| DIV | Оператор деление |
| POW | Оператор в степени |
| OR  AND  DA  NET  NS  GREAT  LESS  GREATEQ  LESSEQ  EQ | Логические операторы и другие |
| CELVAL | Целое Число |
| VESHVAL | Выществыние число |
| ARRID | Массив ID |
| ID | ID |
|
|
|
|
|
|
|
|
|
|
|
|
|
|
|

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct StmtInfo  {  enum StmtType type;  struct ExprInfo \* expr;  struct IfStmtInfo \* ifstmt;  struct LoopStmtInfo \* loopstmt;  struct SwitchStmtInfo \* switchstmt;  int vartype;  struct IdListInfo \* idlist;  char \* funcname;  struct ExprListInfo \* exprlist;  struct StmtInfo \* next;  }; | Оператор. Структура описывает символ грамматики **statement.** | |
| type | Тип оператора |
| expr | выражений |
| ifstmt | Оператор если |
| loopstmt | Оператор цикул |
| switchstmt | Оператор выбор |
| next | Указателья на следущ оператор |
| exprlist | Список выражений |
| funcname | Указатель на имя функций |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct StmtListInfo  {  struct StmtInfo \* first;  struct StmtInfo \* last;  }; | Список операторов. Структура описывает символ грамматики StmtListInfo**.** | |
| first | Указатель на первый оператор |
| last | Указатель на последний оператор |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct FuncListInfo  {  int type;  char \* FuncListInfo;  struct ArgListInfo \* arglist;  struct PrefListInfo \* funcpreflist;  struct StmtListInfo \* stmtlist;  struct FuncListInfo \* next;  }; | Простой оператор. Структура описывает символ грамматики FuncListInfo**.** | |
| FuncListInfo | Указателя на функций |
| arglist | Указателя на спсок функций |
| funcpreflist | Список выражений слева |
| Stmtlist  next | Список statement    Указателья на следущ функций |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct ExprInfo  {  struct ExprInfo \* left;  struct ExprInfo \* right;  struct ExprInfo \* next;  enum ExprType type;  char \* id;  int celval;  float veshval;  }; | Структура описывает символ грамматики ExprInfo**.** | |
| left | Указателья на левая часть выражений |
| right | Указателья на правая часть выражений |
| Next  Type  Celval  veshval | Указателья на следущ выраж  Тип выражений  Целое число  Float число |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct ExprListInfo  {  struct ExprInfo \* first;  struct ExprInfo \* last;  }; | Структура описывает символ грамматики ExprListInfo**.** | |
| first | Указателья на первое |
| last | Указателья на последнее |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct IfStmtInfo  {  struct ExprInfo \* expr;  struct StmtListInfo \* stmtlist;  struct StmtListInfo \* elsestmtlist;  }; | Оператор ДЛЯ. Структура описывает символ грамматики IfStmtInfo**.** | |
| expr | Условие оператора ДЛЯ |
| elsestmtlist | Тело оператора Иначе |
| stmtlist | Тело оператора ДЛЯ |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct SwitchStmtInfo  {  struct SwitchLinesInfo \* lines;  struct StmtListInfo \* stmtlist;  }; |  | |
| lines | Тип условия для оператора switch. |
| stmtlist | Условие оператора switch |
|

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct LoopStmtInfo  {  enum LoopType type;  char \* var;  int from;  int to;  int shag;  struct ExprInfo \* expr;  struct StmtListInfo \* stmtlist;  }; | Список оператора LoopStmtInfo. Структура описывает символ грамматики LoopStmtInfo | |
| type | Тип Цикул |
| expr | Услове оператор |
| stmtlist | Спиоск statement |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct ArgListEInfo  {  enum ArgListType type;  int argtype;  int vartype;  char \* id;  int arrind;  struct ArgListEInfo \* next;  }; | Case функций. Структура описывает символ грамматикиArgListEInfo**.** | |
| type | Тип Функций |
| id | Указатель на ID функций |
| next | Указатель на следующий функций |

|  |  |  |
| --- | --- | --- |
| Структура | Описание структуры | |
| struct ArgListInfo  {  struct ArgListEInfo \* arglist;  }; |  | |
| arglist | Список Функций |

**Тестовые примеры**

алг;

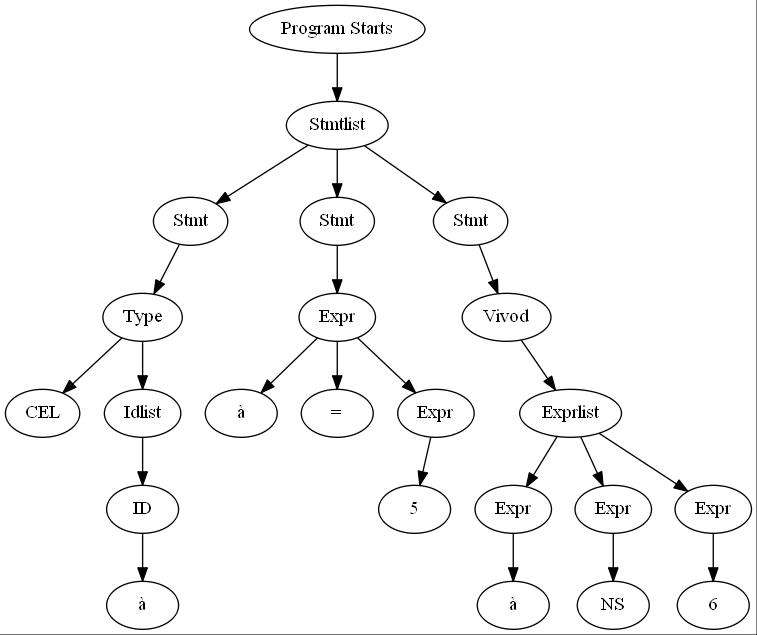
нач;

цел а;

а := 5;

вывод а, нс, 6;

кон;



**Исходный код**

Текст сканера (входного файла для flex):

%{

#include "simple\_tab.h"

#include <stdlib.h>

#include <string.h>

#include <conio.h>

#include <locale.h>

extern int yyparse(void);

char \* formText(char \* str);

%}

%option noyywrap

%option never-interactive

digit [0-9]

number [1-9]{digit}\*|0

hexnum "$"[0-9a-fA-F]+

float {number}"."{digit}\*

expfloat ({float}|{number})[еЕeE]("+"|"-")?{number}

identifier [a-zA-Zа-яА-ЯёЁ@\_][a-zA-Zа-яА-ЯёЁ0-9@\_]\*

identifier2 [a-zA-Zа-яА-ЯёЁ0-9@\_]+

quote1 "\'"[^']\*"\'"

quote2 "\""[^"]\*"\""

%%

алг { return ALG; }

дано { yylval.int\_const = 1; return FUNCPREF; }

надо { yylval.int\_const = 2; return FUNCPREF; }

нач { return NACH; }

кон { return KON; }

арг { yylval.int\_const = 1; return ARGTYPE; }

рез { yylval.int\_const = 2; return ARGTYPE; }

аргрез { yylval.int\_const = 3; return ARGTYPE; }

";" return ';';

цел { yylval.int\_const = 1; return TYPE; }

вещ { yylval.int\_const = 2; return TYPE; }

лог { yylval.int\_const = 3; return TYPE; }

сим { yylval.int\_const = 4; return TYPE; }

лит { yylval.int\_const = 5; return TYPE; }

таб { yylval.int\_const = 6; return TYPE; }

целтаб { yylval.int\_const = 7; return TYPE; }

вещтаб { yylval.int\_const = 8; return TYPE; }

логтаб { yylval.int\_const = 9; return TYPE; }

симтаб { yylval.int\_const = 10; return TYPE; }

литтаб { yylval.int\_const = 11; return TYPE; }

и return AND;

или return OR;

не return NE;

да return DA;

нет return NET;

утв return UTV;

выход return VIHOD;

ввод return VVOD;

вывод return VIVOD;

нс return NS;

если return IF;

то return THEN;

иначе return ELSE;

вс(е|ё) return VSE;

выбор return VIBOR;

при return PRI;

нц return NZ;

кц return KZ;

кц\_при return KZ\_PRI;

раз return RAZ;

пока return POKA;

для return DLYA;

от return OT;

до return DO;

шаг return SHAG;

{number} {

yylval.int\_const = atoi(yytext);

return CELVAL;

}

{hexnum} {

int d;

sscanf(yytext+1, "%x", &d);

yylval.int\_const = d; return CELVAL;

}

{float} {

yylval.float\_const = atof(yytext);

return VESHVAL;

}

{expfloat} {

yylval.float\_const = atof(yytext);

return VESHVAL;

}

{identifier} {

yylval.id = formText(yytext);

return ID;

}

{identifier2} {

yylval.id = formText(yytext);

return ID;

}

{quote1} {

yylval.id = formText(yytext);

return QUOTE1;

}

{quote2} {

yylval.id = formText(yytext);

return QUOTE2;

}

"," return ',';

"(" return '(';

")" return ')';

"[" return '[';

"]" return ']';

":" return ':';

":=" return ASSIGN;

"+" return '+';

"-" return '-';

"\*" return '\*';

"/" return '/';

"\*\*" return POW;

"=" return '=';

"<>" return NOTEQ;

"<" return '<';

">" return '>';

"<=" return LESSEQ;

">=" return MOREEQ;

" "|"\t" ;

"|".\* ;

"#".\* ;

" "{1,3}|"\n"|"\n\r"|"\r\n"|"\r"; /\*skip\*/

. ;

%%

char \* formText(char \* str)

{

char \* res = (char \*)malloc(sizeof(char)\*1024);

strcpy(res, str);

return res;

}

Текст парсера (входного файла для bison):

%{

#include "functions.h"

#include "tree\_print.h"

void yyerror(char const \*s);

extern int yylex(void);

struct ProgramInfo \* CreateProgram (struct StmtListInfo \* prefix, struct PrefListInfo \* funcpreflist, struct StmtListInfo \* stmtlist, struct FuncListInfo \* functions);

struct FuncListInfo \* CreateFunctionList (int type, char \* funcname, struct ArgListInfo \* arglist, struct PrefListInfo \* funcpreflist, struct StmtListInfo \* stmtlist, struct FuncListInfo \* prev);

struct StmtListInfo \* CreateStatementList (struct StmtInfo \* stmt, struct StmtListInfo \* stmtlist);

struct StmtInfo \* CreateFromExprStatement (enum StmtType type, struct ExprInfo \* expr);

struct StmtInfo \* CreateFromIfStatement (enum StmtType type, struct IfStmtInfo \* ifstmt);

struct StmtInfo \* CreateFromLoopStatement (enum StmtType type, struct LoopStmtInfo \* loopstmt);

struct StmtInfo \* CreateFromSwitchStatement (enum StmtType type, struct SwitchStmtInfo \* switchstmt);

struct StmtInfo \* CreateFromTypeListStatement (enum StmtType type, int vartype, struct IdListInfo \* idlist);

struct StmtInfo \* CreateFromFuncCallStatement (enum StmtType type, char \* id, struct ExprListInfo \* exprlist);

struct StmtInfo \* CreateFromInOutStatement (enum StmtType type, struct ExprListInfo \* exprlist, struct IdListInfo \* idlist);

struct IfStmtInfo \* CreateIfStatement (struct ExprInfo \* expr, struct StmtListInfo \* stmtlist, struct StmtListInfo \* elsestmtlist);

struct SwitchStmtInfo \* CreateSwitchStatement (struct SwitchLinesInfo \* lines, struct StmtListInfo \* stmtlist);

struct SwitchLinesInfo \* CreateSwitchLines (struct ExprInfo \* expr, struct StmtListInfo \* stmtlist, struct SwitchLinesInfo \* prev);

struct LoopStmtInfo \* CreateLoopStatement (enum LoopType type, char \* var, int from, int to, int shag, struct ExprInfo \* expr, struct StmtListInfo \* stmtlist);

struct ExprInfo \* CreateCompExpression (enum ExprType type, struct ExprInfo \* left, struct ExprInfo \* right);

struct ExprInfo \* CreateSimpleExpression (enum ExprType type, int a, float b, char \* c, struct ExprInfo \* expr, struct ExprListInfo \* exprlist);

struct ExprListInfo \* CreateExpressionList (struct ExprInfo \* expr, struct ExprListInfo \* exprlist);

struct IdListInfo \* CreateIdList (int val, char \* id, struct IdListInfo \* prev);

struct ArgListEInfo \* CreateArgListE (int argtype, int vartype, char \* id, int arrind, struct ArgListEInfo \* prev);

struct ArgListInfo \* CreateArgList(struct ArgListEInfo \* ale);

struct PrefListEInfo \* CreatePrefListE (int type, struct ExprInfo \* expr, struct PrefListEInfo \* prev);

struct PrefListInfo \* CreatePrefList (struct PrefListEInfo \* ple);

%}

%union {

int int\_const;

float float\_const;

char\* str;

char\* id;

struct ProgramInfo \* prog;

struct FuncListInfo \* funclist;

struct ExprInfo \* exp;

struct ExprListInfo \* explst;

struct IdListInfo \* idlst;

struct StmtInfo \* stm;

struct StmtListInfo \* stmlst;

struct PrefListEInfo \* prflste;

struct PrefListInfo \* prflst;

struct IfStmtInfo \* ifstm;

struct SwitchStmtInfo \* swstm;

struct SwitchLinesInfo \* swln;

struct LoopStmtInfo \* lpstm;

struct ArgListEInfo \* arglste;

struct ArgListInfo \* arglst;

}

%type <prog> program;

%type <funclist> func\_list;

%type <exp> expr;

%type <explst> expr\_list;

%type <idlst> id\_list;

%type <stm> stmt;

%type <stmlst> stmt\_list;

%type <prflste> func\_pref\_listE;

%type <prflst> func\_pref\_list;

%type <ifstm> if\_stmt;

%type <swstm> switch\_stmt;

%type <swln> switch\_lines;

%type <lpstm> loop\_stmt;

%type <arglste> arg\_listE;

%type <arglst> arg\_list;

%token <int\_const> CELVAL

%token <float\_const> VESHVAL

%token <id> ID

%token NOTEQ

%token ASSIGN

%token POW

%token NEWLINE

%token VIBOR

%token PRI

%token NZ

%token KZ

%token KZ\_PRI

%token IF

%token ELSE

%token THEN

%token OT

%token DO

%token VSE

%token DLYA

%token SHAG

%token POKA

%token RAZ

%token DA

%token NET

%token OR

%token AND

%token ALG

%token NACH

%token KON

%token <int\_const> TYPE

%token <int\_const> FUNCPREF

%token <int\_const> ARGTYPE

%token NS

%token VVOD

%token VIVOD

%token ZNACH

%token NE

%token UTV

%token VIHOD

%token <str> QUOTE1

%token <str> QUOTE2

%token TRASH

%right ASSIGN

%left OR

%left AND

%left '<' '>' '=' LESSEQ MOREEQ NOTEQ

%left '-' '+'

%left '\*' '/'

%left POW

%left UMINUS

%nonassoc '[' ']'

%nonassoc '(' ')'

%%

program:

stmt\_list ALG ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateProgram($1, $4, $7, NULL); print($$); }

| ALG ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateProgram(NULL, $3, $6, NULL); print($$); }

| stmt\_list ALG ';' func\_pref\_list NACH ';' stmt\_list KON ';' func\_list { $$ = CreateProgram($1, $4, $7, $10); print($$); }

| ALG ';' func\_pref\_list NACH ';' stmt\_list KON ';' func\_list { $$ = CreateProgram(NULL, $3, $6, $9); print($$); }

;

func\_list:

ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateFunctionList(-1, $2, $4, $7, $10, NULL); }

| TYPE ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateFunctionList($1, $3, $5, $8, $11, NULL); }

| func\_list ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateFunctionList(-1, $3, $5, $8, $11, $1); }

| func\_list TYPE ALG ID '(' arg\_list ')' ';' func\_pref\_list NACH ';' stmt\_list KON ';' { $$ = CreateFunctionList($2, $4, $6, $9, $12, $1); }

;

stmt\_list:

stmt { $$ = CreateStatementList($1, NULL); }

| stmt\_list stmt { $$ = CreateStatementList($2, $1); }

;

stmt:

expr ';' { $$ = CreateFromExprStatement(\_\_EXPR, $1); }

| if\_stmt ';' { $$ = CreateFromIfStatement(\_\_IF, $1); }

| loop\_stmt ';' { $$ = CreateFromLoopStatement(\_\_LOOP, $1); }

| switch\_stmt ';' { $$ = CreateFromSwitchStatement(\_\_SWITCH, $1); }

| TYPE id\_list ';' { $$ = CreateFromTypeListStatement(\_\_TYPE, $1, $2); }

| VVOD id\_list ';' { $$ = CreateFromInOutStatement(\_\_VVOD, NULL, $2); }

| VIVOD expr\_list ';' { $$ = CreateFromInOutStatement(\_\_VIVOD, $2, NULL); }

;

if\_stmt:

IF expr THEN stmt\_list VSE { $$ = CreateIfStatement($2, $4, NULL); }

| IF expr THEN stmt\_list ELSE stmt\_list VSE { $$ = CreateIfStatement($2, $4, $6); }

;

switch\_stmt:

VIBOR switch\_lines VSE { $$ = CreateSwitchStatement($2, NULL); }

| VIBOR switch\_lines ELSE stmt\_list VSE { $$ = CreateSwitchStatement($2, $4); }

;

switch\_lines:

PRI expr ':' stmt\_list { $$ = CreateSwitchLines($2, $4, NULL); }

| switch\_lines PRI expr ':' stmt\_list { $$ = CreateSwitchLines($3, $5, $1); }

;

loop\_stmt:

NZ DLYA ID OT CELVAL DO CELVAL stmt\_list KZ { $$ = CreateLoopStatement(\_\_DLYA, $3, $5, $7, 1, NULL, $8); }

| NZ DLYA ID OT CELVAL DO CELVAL SHAG CELVAL stmt\_list KZ { $$ = CreateLoopStatement(\_\_DLYASHAG, $3, $5, $7, $9, NULL, $10); }

| NZ stmt\_list KZ { $$ = CreateLoopStatement(\_\_PROST, NULL, -1, -1, -1, NULL, $2); }

| NZ stmt\_list KZ\_PRI expr { $$ = CreateLoopStatement(\_\_PRI, NULL, -1, -1, -1, $4, $2); }

| NZ POKA expr ';' stmt\_list KZ { $$ = CreateLoopStatement(\_\_POKA, NULL, -1, -1, -1, $3, $5); }

| NZ CELVAL RAZ stmt\_list KZ { $$ = CreateLoopStatement(\_\_RAZ, NULL, 0, $2, -1, NULL, $4); }

;

expr:

expr '+' expr { $$ = CreateCompExpression(\_ADD, $1, $3); }

| expr '-' expr { $$ = CreateCompExpression(\_SUB, $1, $3); }

| expr '\*' expr { $$ = CreateCompExpression(\_MUL, $1, $3); }

| expr '/' expr { $$ = CreateCompExpression(\_DIV, $1, $3); }

| '-' expr %prec UMINUS { $$ = CreateCompExpression(\_UNARSUB, $2, NULL); }

| expr POW expr { $$ = CreateCompExpression(\_POW, $1, $3); }

| '(' expr ')' { $$ = $2; }

| expr NOTEQ expr { $$ = CreateCompExpression(\_NOTEQ, $1, $3); }

| expr OR expr { $$ = CreateCompExpression(\_OR, $1, $3); }

| expr AND expr { $$ = CreateCompExpression(\_AND, $1, $3); }

| DA { $$ = CreateSimpleExpression(\_DA, -1, -1, NULL, NULL, NULL); }

| NET { $$ = CreateSimpleExpression(\_NET, -1, -1, NULL, NULL, NULL); }

| NS { $$ = CreateSimpleExpression(\_NS, -1, -1, NULL, NULL, NULL); }

| expr '>' expr { $$ = CreateCompExpression(\_GREAT, $1, $3); }

| expr '<' expr { $$ = CreateCompExpression(\_LESS, $1, $3); }

| expr LESSEQ expr { $$ = CreateCompExpression(\_LESSEQ, $1, $3); }

| expr MOREEQ expr { $$ = CreateCompExpression(\_GREATEQ, $1, $3); }

| expr '=' expr { $$ = CreateCompExpression(\_EQ, $1, $3); }

| ID ASSIGN expr { $$ = CreateSimpleExpression(\_IDASSIGN, -1, -1, $1, $3, NULL); }

| ID '[' CELVAL ']' ASSIGN expr { $$ = CreateSimpleExpression(\_ARRASSIGN, $3, -1, $1, $6, NULL); }

| CELVAL { $$ = CreateSimpleExpression(\_CELVAL, $1, -1, NULL, NULL, NULL); }

| VESHVAL { $$ = CreateSimpleExpression(\_VESHVAL, -1, $1, NULL, NULL, NULL); }

| ID '[' CELVAL ']' { $$ = CreateSimpleExpression(\_ARRID, $3, -1, $1, NULL, NULL); }

| ID { $$ = CreateSimpleExpression(\_ID, -1, -1, $1, NULL, NULL); }

| ID '(' expr\_list ')' { $$ = CreateSimpleExpression(\_FUNCCALL, -1, -1, $1, NULL, $3); }

| ID '(' ')' { $$ = CreateSimpleExpression(\_FUNCCALL, -1, -1, $1, NULL, NULL); }

;

expr\_list:

expr { $$ = CreateExpressionList($1, NULL); }

| expr\_list ',' expr { $$ = CreateExpressionList($3, $1); }

;

id\_list:

ID { $$ = CreateIdList(-1, $1, NULL); }

| ID '[' CELVAL ']' { $$ = CreateIdList($3, $1, NULL); }

| id\_list ',' ID { $$ = CreateIdList(-1, $3, $1); }

| id\_list ',' ID '[' CELVAL ']' { $$ = CreateIdList($5, $3, $1); }

;

arg\_list:

/\* empty \*/ { $$ = 0; }

| arg\_listE { $$ = CreateArgList($1); }

;

arg\_listE:

ARGTYPE TYPE ID { $$ = CreateArgListE($1, $2, $3, -1, NULL); }

| ARGTYPE TYPE ID '[' CELVAL ']' { $$ = CreateArgListE($1, $2, $3, $5, NULL); }

| TYPE ID { $$ = CreateArgListE(-1, $1, $2, -1, NULL); }

| TYPE ID '[' CELVAL ']' { $$ = CreateArgListE(-1, $1, $2, $4, NULL); }

| arg\_listE ',' ARGTYPE TYPE ID { $$ = CreateArgListE($3, $4, $5, -1, $1); }

| arg\_listE ',' ARGTYPE TYPE ID '[' CELVAL ']' { $$ = CreateArgListE($3, $4, $5, $7, $1); }

| arg\_listE ',' TYPE ID { $$ = CreateArgListE(-1, $3, $4, -1, $1); }

| arg\_listE ',' TYPE ID '[' CELVAL ']' { $$ = CreateArgListE(-1, $3, $4, $6, $1); }

;

func\_pref\_list:

/\* empty \*/ { $$ = 0; }

| func\_pref\_listE { $$ = CreatePrefList($1); }

;

func\_pref\_listE:

FUNCPREF ';' { $$ = CreatePrefListE($1, NULL, NULL); }

| FUNCPREF expr ';' { $$ = CreatePrefListE($1, $2, NULL); }

| func\_pref\_listE FUNCPREF { $$ = CreatePrefListE($2, NULL, $1); }

| func\_pref\_listE FUNCPREF expr ';' { $$ = CreatePrefListE($2, $3, $1); }

;

%%

void yyerror(char const \*s)

{

printf("%s",s);

}

Функции заполнения AST:

#include <stdio.h>

#include <malloc.h>

#include <string.h>

#include "tree\_structs.h"

struct ProgramInfo \* CreateProgram (struct StmtListInfo \* prefix, struct PrefListInfo \* funcpreflist, struct StmtListInfo \* stmtlist, struct FuncListInfo \* functions)

{

struct ProgramInfo \* prog = (struct ProgramInfo \*)malloc(sizeof(struct ProgramInfo));

prog->prefix = prefix;

prog->funcpreflist = funcpreflist;

prog->stmtlist = stmtlist;

prog->functions = functions;

return prog;

}

struct FuncListInfo \* CreateFunctionList (int type, char \* funcname, struct ArgListInfo \* arglist, struct PrefListInfo \* funcpreflist, struct StmtListInfo \* stmtlist, struct FuncListInfo \* prev)

{

struct FuncListInfo \* functionlist = (struct FuncListInfo \*)malloc(sizeof(struct FuncListInfo));

functionlist->type = type;

functionlist->funcname = funcname;

functionlist->arglist = arglist;

functionlist->funcpreflist = funcpreflist;

functionlist->stmtlist = stmtlist;

functionlist->next = NULL;

if (prev != NULL)

{

prev->next = functionlist;

functionlist->first = prev->first;

}

else

functionlist->first = functionlist;

return functionlist;

}

struct StmtListInfo \* CreateStatementList (struct StmtInfo \* stmt, struct StmtListInfo \* stmtlist)

{

stmt->next = NULL;

if (stmtlist == NULL)

{

struct StmtListInfo \* sl = (struct StmtListInfo \*)malloc(sizeof(struct StmtListInfo));

sl->first = sl->last = stmt;

return sl;

}

stmtlist->last = stmtlist->last->next = stmt;

return stmtlist;

}

struct StmtInfo \* CreateFromExprStatement (enum StmtType type, struct ExprInfo \* expr)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->expr = expr;

stmt->next = NULL;

return stmt;

}

struct StmtInfo \* CreateFromIfStatement (enum StmtType type, struct IfStmtInfo \* ifstmt)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->ifstmt = ifstmt;

stmt->next = NULL;

return stmt;

}

struct StmtInfo \* CreateFromLoopStatement (enum StmtType type, struct LoopStmtInfo \* loopstmt)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->loopstmt = loopstmt;

stmt->next = NULL;

return stmt;

}

struct StmtInfo \* CreateFromSwitchStatement (enum StmtType type, struct SwitchStmtInfo \* switchstmt)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->switchstmt = switchstmt;

stmt->next = NULL;

return stmt;

}

struct StmtInfo \* CreateFromTypeListStatement (enum StmtType type, int vartype, struct IdListInfo \* idlist)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->vartype = vartype;

stmt->idlist = idlist;

stmt->next = NULL;

return stmt;

}

struct StmtInfo \* CreateFromInOutStatement (enum StmtType type, struct ExprListInfo \* exprlist, struct IdListInfo \* idlist)

{

struct StmtInfo \* stmt = (struct StmtInfo \*)malloc(sizeof(struct StmtInfo));

stmt->type = type;

stmt->exprlist = exprlist;

stmt->idlist = idlist;

stmt->next = NULL;

return stmt;

}

struct IfStmtInfo \* CreateIfStatement (struct ExprInfo \* expr, struct StmtListInfo \* stmtlist, struct StmtListInfo \* elsestmtlist)

{

struct IfStmtInfo \* ifstmt = (struct IfStmtInfo \*)malloc(sizeof(struct IfStmtInfo));

ifstmt->expr = expr;

ifstmt->stmtlist = stmtlist;

ifstmt->elsestmtlist = elsestmtlist;

return ifstmt;

}

struct SwitchStmtInfo \* CreateSwitchStatement (struct SwitchLinesInfo \* lines, struct StmtListInfo \* stmtlist)

{

struct SwitchStmtInfo \* switchstmt = (struct SwitchStmtInfo \*)malloc(sizeof(struct SwitchStmtInfo));

switchstmt->lines = lines->first;

switchstmt->stmtlist = stmtlist;

return switchstmt;

}

struct SwitchLinesInfo \* CreateSwitchLines (struct ExprInfo \* expr, struct StmtListInfo \* stmtlist, struct SwitchLinesInfo \* prev)

{

struct SwitchLinesInfo \* switchlines = (struct SwitchLinesInfo \*)malloc(sizeof(struct SwitchLinesInfo));

switchlines->expr = expr;

switchlines->stmtlist = stmtlist;

switchlines->next = NULL;

if (prev != NULL)

{

prev->next = switchlines;

switchlines->first = prev->first;

}

else

switchlines->first = switchlines;

return switchlines;

}

struct LoopStmtInfo \* CreateLoopStatement (enum LoopType type, char \* var, int from, int to, int shag, struct ExprInfo \* expr, struct StmtListInfo \* stmtlist)

{

struct LoopStmtInfo \* loopstmt = (struct LoopStmtInfo \*)malloc(sizeof(struct LoopStmtInfo));

loopstmt->type = type;

loopstmt->var = var;

loopstmt->from = from;

loopstmt->to = to;

loopstmt->shag = shag;

loopstmt->expr = expr;

loopstmt->stmtlist = stmtlist;

return loopstmt;

}

struct ExprInfo \* CreateCompExpression (enum ExprType type, struct ExprInfo \* left, struct ExprInfo \* right)

{

struct ExprInfo \* exp = (struct ExprInfo \*)malloc(sizeof(struct ExprInfo));

exp->type = type;

exp->left = left;

exp->right = right;

return exp;

}

struct ExprInfo \* CreateSimpleExpression (enum ExprType type, int a, float b, char \* c, struct ExprInfo \* expr, struct ExprListInfo \* exprlist)

{

struct ExprInfo \* exp = (struct ExprInfo \*)malloc(sizeof(struct ExprInfo));

exp->type = type;

exp->celval = a;

exp->veshval = b;

exp->id = c;

exp->left = expr;

exp->arglist = exprlist;

return exp;

}

struct ExprListInfo \* CreateExpressionList (struct ExprInfo \* expr, struct ExprListInfo \* exprlist)

{

expr->next = NULL;

if (exprlist == NULL)

{

struct ExprListInfo \* el = (struct ExprListInfo \*)malloc(sizeof(struct ExprListInfo));

el->first = el->last = expr;

return el;

}

exprlist->last = exprlist->last->next = expr;

return exprlist;

}

struct IdListInfo \* CreateIdList (int val, char \* id, struct IdListInfo \* prev)

{

struct IdListInfo \* idlist = (struct IdListInfo \*)malloc(sizeof(struct IdListInfo));

idlist->celval = val;

idlist->id = id;

idlist->next = NULL;

if (prev != NULL)

{

prev->next = idlist;

idlist->first = prev->first;

}

else

idlist->first = idlist;

return idlist;

}

struct ArgListEInfo \* CreateArgListE (int argtype, int vartype, char \* id, int arrind, struct ArgListEInfo \* prev)

{

struct ArgListEInfo \* arglist = (struct ArgListEInfo \*)malloc(sizeof(struct ArgListEInfo));

arglist->argtype = argtype;

arglist->vartype = vartype;

arglist->id = id;

arglist->arrind = arrind;

arglist->next = NULL;

if (prev != NULL)

{

prev->next = arglist;

arglist->first = prev->first;

}

else

arglist->first = arglist;

return arglist;

}

struct ArgListInfo \* CreateArgList(struct ArgListEInfo \* ale)

{

struct ArgListInfo \* al = (struct ArgListInfo \*)malloc(sizeof(struct ArgListInfo));

al->arglist = ale->first;

return al;

}

struct PrefListEInfo \* CreatePrefListE (int type, struct ExprInfo \* expr, struct PrefListEInfo \* prev)

{

struct PrefListEInfo \* preflist = (struct PrefListEInfo \*)malloc(sizeof(struct PrefListEInfo));

preflist->type = type;

preflist->expr = expr;

preflist->next = NULL;

if (prev != NULL)

{

prev->next = preflist;

preflist->first = prev->first;

}

else

preflist->first = preflist;

return preflist;

}

struct PrefListInfo \* CreatePrefList (struct PrefListEInfo \* ple)

{

struct PrefListInfo \* pl = (struct PrefListInfo \*)malloc(sizeof(struct PrefListInfo));

pl->preflist = ple->first;

return pl;

}

Функции печати AST:

#include <stdio.h>

#include <string.h>

#include "tree\_structs.h"

char dotfile[10] = "tree.dot";

FILE \* file;

int nodeid = 0;

void printFuncList(struct FuncListInfo \* funclist, int idfrom);

void printFunc(struct FuncListInfo \* func, int idfrom);

void printPrefList(struct PrefListInfo \* preflist, int idfrom);

void printPrefListE(struct PrefListEInfo \* preflist, int idfrom);

void printArgList(struct ArgListInfo \* arglist, int idfrom);

void printArgListE(struct ArgListEInfo \* arglist, int idfrom, int argnum);

void printStmtList(struct StmtListInfo \* stmtlist, int idfrom);

void printStmt(struct StmtInfo \* stmt, int idfrom);

void printIfStmt(struct IfStmtInfo \* ifstmt, int idfrom);

void printSwitchStmt(struct SwitchStmtInfo \* switchstmt, int idfrom);

void printSwitchLine(struct SwitchLinesInfo \* switchline, int idfrom);

void printLoopStmt(struct LoopStmtInfo \* loopstmt, int idfrom);

void printExprList(struct ExprListInfo \* exprlist, int idfrom);

void printExpr(struct ExprInfo \* expr, int idfrom);

void printIdList(struct IdListInfo \* idlist, int idfrom);

void printId(struct IdListInfo \* idlist, int idfrom);

void printByStr(int parent, int child, char \* name);

void printByNum(int parent, int child, int name);

void printByFloat(int parent, int child, float name);

void printNode(int id, char \* name);

void printRelation(int parent, int child);

char \* getTypeName(int type\_id);

char \* getArgTypeName(int argtype\_id);

char \* transliterateId(char \* id);

char \* transliterateChar(char ch);

void print(struct ProgramInfo \* tree)

{

int id = nodeid++;

if (tree == NULL)

return;

file = fopen(dotfile, "w");

fputs("digraph G {\n", file);

printNode(id, "Program Starts");

if (tree->prefix != NULL)

printStmtList(tree->prefix, id);

if (tree->funcpreflist != NULL)

printPrefList(tree->funcpreflist, id);

if (tree->stmtlist != NULL)

printStmtList(tree->stmtlist, id);

if (tree->functions != NULL)

printFuncList(tree->functions, id);

fputs("}Program Ends", file);

fclose(file);

}

void printFuncList(struct FuncListInfo \* funclist, int idfrom)

{

int id = nodeid++;

struct FuncListInfo \* fl = funclist->first;

printByStr(idfrom, id, "FuncList");

do {

printFunc(fl, id);

fl = fl->next;

} while (fl != NULL);

}

void printFunc(struct FuncListInfo \* func, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, func->funcname);

if (func->type)

printByStr(id, nodeid++, getTypeName(func->type));

if (func->arglist != NULL)

printArgList(func->arglist, id);

if (func->funcpreflist != NULL)

printPrefList(func->funcpreflist, id);

if (func->stmtlist != NULL)

printStmtList(func->stmtlist, id);

}

void printPrefList(struct PrefListInfo \* preflist, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, "PrefList");

if (preflist->preflist != NULL)

printPrefListE(preflist->preflist, id);

}

void printPrefListE(struct PrefListEInfo \* preflist, int idfrom)

{

int id = nodeid++;

if (preflist->type == 1)

printByStr(idfrom, id, "DANO");

else

printByStr(idfrom, id, "NADO");

if (preflist->expr != NULL)

printExpr(preflist->expr, id);

if (preflist->next != NULL)

printPrefListE(preflist->next, idfrom);

}

void printArgList(struct ArgListInfo \* arglist, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, "ArgList");

if (arglist->arglist != NULL)

printArgListE(arglist->arglist, id, 1);

}

void printArgListE(struct ArgListEInfo \* arglist, int idfrom, int argnum)

{

int id = nodeid++;

if (argnum == 1 && arglist->argtype == 0)

arglist->argtype = 1;

printByNum(idfrom, id, argnum);

if (arglist->argtype != -1)

printByStr(id, nodeid++, getArgTypeName(arglist->argtype));

printByStr(id, nodeid++, getTypeName(arglist->vartype));

printByStr(id, nodeid++, "ID");

printByStr(nodeid-2, nodeid++, arglist->id);

if (arglist->arrind != -1)

{

printByStr(idfrom, nodeid++, "ArrInd");

printByNum(nodeid-2, nodeid++, arglist->arrind);

}

if (arglist->next != NULL)

printArgListE(arglist->next, idfrom, ++argnum);

}

void printStmtList(struct StmtListInfo \* stmtlist, int idfrom)

{

int id = nodeid++;

struct StmtInfo \* si = stmtlist->first;

printByStr(idfrom, id, "Stmtlist");

while (si != NULL)

{

printStmt(si, id);

si = si->next;

}

}

void printStmt(struct StmtInfo \* stmt, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, "Stmt");

switch (stmt->type)

{

case \_\_EXPR:

printExpr(stmt->expr, id);

break;

case \_\_IF:

printIfStmt(stmt->ifstmt, id);

break;

case \_\_SWITCH:

printSwitchStmt(stmt->switchstmt, id);

break;

case \_\_LOOP:

printLoopStmt(stmt->loopstmt, id);

break;

case \_\_TYPE:

printByStr(id, nodeid++, "Type");

printByStr(nodeid-2, nodeid++, getTypeName(stmt->vartype));

printIdList(stmt->idlist, nodeid-2);

break;

case \_\_VVOD:

printByStr(id, nodeid++, "Vvod");

printIdList(stmt->idlist, nodeid-1);

break;

case \_\_VIVOD:

printByStr(id, nodeid++, "Vivod");

printExprList(stmt->exprlist, nodeid-1);

break;

}

}

void printIfStmt(struct IfStmtInfo \* ifstmt, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, "IfStmt");

printExpr(ifstmt->expr, id);

printStmtList(ifstmt->stmtlist, id);

if (ifstmt->elsestmtlist != NULL)

printStmtList(ifstmt->elsestmtlist, id);

}

void printSwitchStmt(struct SwitchStmtInfo \* switchstmt, int idfrom)

{

int id = nodeid++;

struct SwitchLinesInfo \* sl = switchstmt->lines;

printByStr(idfrom, id, "SwitchStmt");

do {

printSwitchLine(sl, id);

sl = sl->next;

} while (sl != NULL);

if (switchstmt->stmtlist != NULL)

{

printByStr(id, nodeid, "Default");

printStmtList(switchstmt->stmtlist, nodeid++);

}

}

void printSwitchLine(struct SwitchLinesInfo \* switchline, int idfrom)

{

int id = nodeid++;

printByStr(idfrom, id, "SwitchLine");

printExpr(switchline->expr, id);

printStmtList(switchline->stmtlist, id);

}

void printLoopStmt(struct LoopStmtInfo \* loopstmt, int idfrom)

{

int id = nodeid++;

switch (loopstmt->type)

{

case \_\_DLYA:

printByStr(idfrom, id, "Dlya Loop");

printByStr(id, nodeid++, loopstmt->var);

printByNum(id, nodeid++, loopstmt->from);

printByNum(id, nodeid++, loopstmt->to);

printStmtList(loopstmt->stmtlist, id);

break;

case \_\_DLYASHAG:

printByStr(idfrom, id, "DlyaShag Loop");

printByStr(id, nodeid++, loopstmt->var);

printByNum(id, nodeid++, loopstmt->from);

printByNum(id, nodeid++, loopstmt->to);

printByNum(id, nodeid++, loopstmt->shag);

printStmtList(loopstmt->stmtlist, id);

break;

case \_\_PROST:

printByStr(idfrom, id, "Prost Loop");

printStmtList(loopstmt->stmtlist, id);

break;

case \_\_PRI:

printByStr(idfrom, id, "Pri Loop");

printExpr(loopstmt->expr, id);

printStmtList(loopstmt->stmtlist, id);

break;

case \_\_POKA:

printByStr(idfrom, id, "Poka Loop");

printExpr(loopstmt->expr, id);

printStmtList(loopstmt->stmtlist, id);

break;

case \_\_RAZ:

printByStr(idfrom, id, "Raz Loop");

printByNum(id, nodeid++, loopstmt->to);

printStmtList(loopstmt->stmtlist, id);

break;

}

}

void printExprList(struct ExprListInfo \* exprlist, int idfrom)

{

int id = nodeid++;

struct ExprInfo \* ei = exprlist->first;

printByStr(idfrom, id, "Exprlist");

while (ei != NULL)

{

printExpr(ei, id);

ei = ei->next;

}

}

void printExpr(struct ExprInfo \* expr, int idfrom)

{

int id = nodeid++;

printNode(id, "Expr");

printRelation(idfrom, id);

switch (expr->type)

{

case \_ADD:

printExpr(expr->left, id);

printByStr(id, nodeid++, "+");

printExpr(expr->right, id);

break;

case \_SUB:

printExpr(expr->left, id);

printByStr(id, nodeid++, "-");

printExpr(expr->right, id);

break;

case \_MUL:

printExpr(expr->left, id);

printByStr(id, nodeid++, "\*");

printExpr(expr->right, id);

break;

case \_DIV:

printExpr(expr->left, id);

printByStr(id, nodeid++, "/");

printExpr(expr->right, id);

break;

case \_UNARSUB:

printByStr(id, nodeid++, "-");

printExpr(expr->left, id);

break;

case \_POW:

printExpr(expr->left, id);

printByStr(id, nodeid++, "POW");

printExpr(expr->right, id);

break;

case \_NOTEQ:

printExpr(expr->left, id);

printByStr(id, nodeid++, "!=");

printExpr(expr->right, id);

break;

case \_OR:

printExpr(expr->left, id);

printByStr(id, nodeid++, "||");

printExpr(expr->right, id);

break;

case \_AND:

printExpr(expr->left, id);

printByStr(id, nodeid++, "&&");

printExpr(expr->right, id);

break;

case \_DA:

printByStr(id, nodeid++, "TRUE");

break;

case \_NET:

printByStr(id, nodeid++, "FALSE");

break;

case \_NS:

printByStr(id, nodeid++, "NS");

break;

case \_GREAT:

printExpr(expr->left, id);

printByStr(id, nodeid++, ">");

printExpr(expr->right, id);

break;

case \_LESS:

printExpr(expr->left, id);

printByStr(id, nodeid++, "<");

printExpr(expr->right, id);

break;

case \_GREATEQ:

printExpr(expr->left, id);

printByStr(id, nodeid++, ">=");

printExpr(expr->right, id);

break;

case \_LESSEQ:

printExpr(expr->left, id);

printByStr(id, nodeid++, "<=");

printExpr(expr->right, id);

break;

case \_EQ:

printExpr(expr->left, id);

printByStr(id, nodeid++, "==");

printExpr(expr->right, id);

break;

case \_IDASSIGN:

printByStr(id, nodeid++, expr->id);

printByStr(id, nodeid++, "=");

printExpr(expr->left, id);

break;

case \_ARRASSIGN:

printByStr(id, nodeid++, expr->id);

printByNum(id, nodeid++, expr->celval);

printByStr(id, nodeid++, "=");

printExpr(expr->left, id);

break;

case \_CELVAL:

printByNum(id, nodeid++, expr->celval);

break;

case \_VESHVAL:

printByFloat(id, nodeid++, expr->veshval);

break;

case \_ARRID:

printByStr(id, nodeid++, expr->id);

printByNum(id, nodeid++, expr->celval);

break;

case \_ID:

printByStr(id, nodeid++, expr->id);

break;

case \_FUNCCALL:

printByStr(id, nodeid++, "FuncCall");

printByStr(nodeid-2, nodeid++, expr->id);

if (expr->arglist != NULL)

printExprList(expr->arglist, nodeid-3);

break;

}

}

void printIdList(struct IdListInfo \* idlist, int idfrom)

{

int id = nodeid++;

struct IdListInfo \* il = idlist->first;

printByStr(idfrom, id, "Idlist");

do {

printId(il, id);

il = il->next;

} while (il != NULL);

}

void printId(struct IdListInfo \* idlist, int idfrom)

{

int id = nodeid++;

if (idlist->celval == -1)

{

printByStr(idfrom, id, "ID");

printByStr(id, nodeid++, idlist->id);

}

else

{

printByStr(idfrom, id, "ARRID");

printByStr(id, nodeid++, idlist->id);

printByNum(id, nodeid++, idlist->celval);

}

}

void printByStr(int parent, int child, char \* name)

{

printNode(child, name);

printRelation(parent, child);

}

void printByNum(int parent, int child, int name)

{

char num[1000];

\_itoa(name, num, 10);

printNode(child, num);

printRelation(parent, child);

}

void printByFloat(int parent, int child, float name)

{

char num[1000];

sprintf(num, "%f", name);

printNode(child, num);

printRelation(parent, child);

}

void printNode(int id, char \* name)

{

char num[1000];

\_itoa(id, num, 10);

fputs("a", file);

fputs(num, file);

fputs(" [label=\"", file);

fputs(name, file);

fputs("\"];\n", file);

}

void printRelation(int parent, int child)

{

char num[1000];

\_itoa(parent, num, 10);

fputs("a", file);

fputs(num, file);

fputs("->a", file);

\_itoa(child, num, 10);

fputs(num, file);

fputs(";\n", file);

}

char \* getTypeName(int type\_id)

{

switch (type\_id)

{

case 1: return "CEL";

case 2: return "VESH";

case 3: return "LOG";

case 4: return "SIM";

case 5: return "LIT";

case 6: return "TAB";

case 7: return "CELTAB";

case 8: return "VESHTAB";

case 9: return "LOGTAB";

case 10: return "SIMTAB";

case 11: return "LITTAB";

}

}

char \* getArgTypeName(int argtype\_id)

{

switch (argtype\_id)

{

case 1: return "ARG";

case 2: return "RES";

case 3: return "ARGRES";

}

}

char \* transliterateId(char \* id)

{

char res[256];

char \* tmp;

int i = 0;

memset(res, 0, sizeof(res));

for (i = 0; i < strlen(id); ++i)

{

tmp = transliterateChar(id[i]);

if (i == 0)

strcpy(res, tmp);

else

strcat(res, tmp);

}

res[strlen(res)] = '\0';

return res;

}

char \* transliterateChar(char ch)

{

char res[2];

switch (ch)

{

case 'а':

return "a";

case 'б':

return "b";

case 'в':

return "v";

case 'г':

return "g";

case 'д':

return "d";

case 'е':

return "e";

case 'ё':

return "e";

case 'ж':

return "j";

case 'з':

return "z";

case 'и':

return "i";

case 'й':

return "y";

case 'к':

return "k";

case 'л':

return "l";

case 'м':

return "m";

case 'н':

return "n";

case 'о':

return "o";

case 'п':

return "p";

case 'р':

return "r";

case 'с':

return "s";

case 'т':

return "t";

case 'у':

return "u";

case 'ф':

return "f";

case 'х':

return "h";

case 'ц':

return "c";

case 'ч':

return "ch";

case 'ш':

return "sh";

case 'щ':

return "sh";

case 'ъ':

return "";

case 'ы':

return "i";

case 'ь':

return "\'";

case 'э':

return "e";

case 'ю':

return "u";

case 'я':

return "ya";

default:

res[0] = ch;

res[1] = '\0';

return res;

}

}