/\* Scanner \*/

#include <stdio.h>

#include <stdlib.h>

#include "reader.h"

#include "charcode.h"

#include "token.h"

#include "error.h"

extern int lineNo;

extern int colNo;

extern int currentChar;

extern CharCode charCodes[];

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void skipBlank()

{

while (currentChar != -1 && charCodes[currentChar] == CHAR\_SPACE)

readChar();

}

void skipComment1()

{

while (1)

{

//Read next character

readChar();

if (currentChar == -1)

{ // End of file

error(ERR\_ENDOFCOMMENT, lineNo, colNo); // comment end with EOF

}

else if (charCodes[currentChar] == CHAR\_TIMES)

{ //If next is asteric character

readChar(); //get next character

}

// Note: (\* then continue to get next character until meet next if

if (currentChar == -1)

{ // End of file

error(ERR\_ENDOFCOMMENT, lineNo, colNo); // comment end with EOF

}

else if (charCodes[currentChar] == CHAR\_RPAR)

{ // Next is right parathesis

// End of comment

readChar();

return;

}

}

}

void skipComment2() {

while (1) {

if (currentChar == '\n') break; // meet '\n' then end cmt

readChar();

}

}

Token \*readIdentKeyword(void)

{

int count = 0;

int flag = 0;

Token \*token = makeToken(TK\_IDENT, lineNo, colNo);

token->string[0] = currentChar;

//while (charCodes[currentChar] == CHAR\_LETTER || charCodes[currentChar] == CHAR\_DIGIT || charCodes[currentChar] == CHAR\_UNDERSCORE) // read underscore

while (charCodes[currentChar] == CHAR\_LETTER || charCodes[currentChar] == CHAR\_DIGIT)

{

if (count < MAX\_IDENT\_LEN) {

token->string[count] = toupper(currentChar); //not distinguish upper and lower

count++;

} else {

if (!flag) {

flag = 1;

error(ERR\_IDENTTOOLONG, lineNo, colNo - count);

}

}

// Add current character to identifier

//token->string[count] = currentChar;

// Increase identifier length

//count++;

// Get next character

readChar();

}

// End string

token->string[count] = '\0';

// Limit identifier length

//if (count > MAX\_IDENT\_LEN) // max identifier length in token.h

//{

// Error

// error(ERR\_IDENTTOOLONG, lineNo, colNo - count);

//}

//else

//{

// If this identifier is a keyword

TokenType type = checkKeyword(token->string); // checkKeyword in token.c

// Otherwise

if (type != TK\_NONE)

{

token->tokenType = type;

}

//}

return token;

}

Token \*readNumber(void)

{

Token \*token = makeToken(TK\_NUMBER, lineNo, colNo);

    int count = 0;

    while ((currentChar != EOF) && (charCodes[currentChar] == CHAR\_DIGIT)) {

        token->string[count++] = (char)currentChar;

if (count > NUMBER\_LEN) {

      error(ERR\_NUMBERTOOLONG, token->lineNo, token->colNo);

      return token;

  }

readChar();

    }

    token->string[count] = '\0';

    token->value = atoi(token->string);

    return token;

}

Token \*readConstChar(void) // example 'A', 'B', 'C', one single quote - letter - one single quote

// already have single quote

{

Token \*token = makeToken(TK\_CHAR, lineNo, colNo); // make token character

// Read next character

readChar();

if (currentChar == -1)

{ // End of file

error(ERR\_INVALIDCHARCONSTANT, token->lineNo, token->colNo);

}

else

{

switch (charCodes[currentChar])

{

// Escape character for Single quote:

case CHAR\_SINGLEQUOTE: // second single quote

// Read next character

token->string[0] = currentChar;

readChar();

if (charCodes[currentChar] == CHAR\_SINGLEQUOTE)

{

token->string[1] = '\0';

readChar();

return token;

}

else

{

error(ERR\_INVALIDCHARCONSTANT, token->lineNo, token->colNo);

}

break;

default:

// Add the character to the token string

token->string[0] = currentChar;

// Read next character

readChar();

switch (charCodes[currentChar])

{

case CHAR\_SINGLEQUOTE:

// End token

token->string[1] = '\0';

readChar();

return token;

default:

error(ERR\_INVALIDCHARCONSTANT, token->lineNo, token->colNo);

break;

}

break;

}

}

return token;

}

// Doc 1 token tinh tu vi tri hien tai

Token \*getToken(void)

{

Token \*token;

// int ln, cn;

if (currentChar == EOF) // If EOF

return makeToken(TK\_EOF, lineNo, colNo); // return token EOF

switch (charCodes[currentChar])

{

case CHAR\_SPACE:

skipBlank(); // skip blank, read next token

return getToken();

case CHAR\_LETTER:

return readIdentKeyword(); // token is identifier

case CHAR\_DIGIT:

return readNumber(); // token is number

case CHAR\_PLUS:

// Token plus

token = makeToken(SB\_PLUS, lineNo, colNo); // return token '+'

readChar(); // read next character

return token;

case CHAR\_MINUS:

// Token minus

token = makeToken(SB\_MINUS, lineNo, colNo); // return token '-'

readChar(); // read next character

return token;

case CHAR\_TIMES:

// Token Times

token = makeToken(SB\_TIMES, lineNo, colNo); // return token '\*'

readChar(); // read next character

return token;

case CHAR\_SLASH:

// Token Slash

token = makeToken(SB\_SLASH, lineNo, colNo); // return token '/'

readChar(); // read next character

if (charCodes[currentChar] == CHAR\_SLASH) {

skipComment2();

return getToken();

}

return token;

// Skip comment style '//', uncomment when meet '\n'

//token = makeToken(TK\_NONE,lineNo,colNo);

//readChar();

//if (charCodes[currentChar] == CHAR\_SLASH) {

// skipComment();

// return getToken();

//}

case CHAR\_EQ:

// Token Equal

token = makeToken(SB\_EQ, lineNo, colNo);

readChar();

return token;

case CHAR\_COMMA:

// Token Comma

token = makeToken(SB\_COMMA, lineNo, colNo);

readChar();

return token;

case CHAR\_SEMICOLON:

// Token Semicolon

token = makeToken(SB\_SEMICOLON, lineNo, colNo);

readChar();

return token;

case CHAR\_RPAR:

// Token right parathesis

token = makeToken(SB\_RPAR, lineNo, colNo);

readChar();

return token;

case CHAR\_LPAR: // meet '('

// Empty token

token = makeToken(TK\_NONE, lineNo, colNo);

// Get next character first

readChar();

switch (charCodes[currentChar])

{

case CHAR\_PERIOD: // meet '.'

// This is token LSEL

token->tokenType = SB\_LSEL; // return '(.' , LSEL: mark index of array

readChar(); // read next character

return token;

case CHAR\_TIMES: // meet '(\*', this is cmt

// This is a comment so free the allocated token first then skip comments

free(token);

skipComment1();

return getToken();

case CHAR\_SPACE:

readChar();

return getToken();

default:

// Token Left Parenthesis

token->tokenType = SB\_LPAR;

// readChar();

return token;

}

case CHAR\_GT:

// Token Greater

token = makeToken(SB\_GT, lineNo, colNo);

// If next character is '='

readChar();

if (charCodes[currentChar] == CHAR\_EQ)

{

// Token is Greater Than

token->tokenType = SB\_GE;

readChar();

}

return token;

case CHAR\_LT: // meet '<'

// Empty token

token = makeToken(TK\_NONE, lineNo, colNo);

// Check next character

readChar();

switch (charCodes[currentChar])

{

case CHAR\_EQ: // if next character is '='

// Token Lest Than or Equal

token->tokenType = SB\_LE; // token is '<='

readChar();

return token;

default:

// Token Least Than

token->tokenType = SB\_LT; // token is '<'

return token;

}

case CHAR\_EXCLAIMATION: // '!'

// Make empty token

token = makeToken(TK\_NONE, lineNo, colNo);

// If next character is not '='

readChar();

if (charCodes[currentChar] != CHAR\_EQ)

{

// it is an invalid token

error(ERR\_INVALIDSYMBOL, token->lineNo, token->colNo);

}

else

{

// else, it's token Not Equal

token->tokenType = SB\_NEQ; // '!='

}

return token;

case CHAR\_PERIOD:

// Token Period

token = makeToken(SB\_PERIOD, lineNo, colNo);

// If next character is Right Parenthesis

readChar();

if (charCodes[currentChar] == CHAR\_RPAR)

{

// it is token Right Parenthesis

token->tokenType = SB\_RSEL;

readChar();

}

return token;

case CHAR\_COLON:

// Token Semicolon

token = makeToken(SB\_SEMICOLON, lineNo, colNo);

// If next character is Equal

readChar();

if (charCodes[currentChar] == CHAR\_EQ)

{

// it is token Assignment

token->tokenType = SB\_ASSIGN; // ':='

readChar();

}

return token;

case CHAR\_SINGLEQUOTE:

return readConstChar(); // when meet single quote, start to read const char

case CHAR\_BACKSLASH:

token = makeToken(SB\_BACKSLASH,lineNo,colNo);

readChar();

return token;

default:

token = makeToken(TK\_NONE, lineNo, colNo);

error(ERR\_INVALIDSYMBOL, lineNo, colNo);

readChar();

return token;

}

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

void printToken(Token \*token)

{

printf("%d-%d:", token->lineNo, token->colNo);

switch (token->tokenType)

{

case TK\_NONE:

printf("TK\_NONE\n");

break;

case TK\_IDENT:

printf("TK\_IDENT(%s)\n", token->string);

break;

case TK\_NUMBER:

printf("TK\_NUMBER(%s)\n", token->string);

break;

case TK\_CHAR:

printf("TK\_CHAR(\'%s\')\n", token->string);

break;

case TK\_EOF:

printf("TK\_EOF\n");

break;

case KW\_PROGRAM:

printf("KW\_PROGRAM\n");

break;

case KW\_CONST:

printf("KW\_CONST\n");

break;

case KW\_TYPE:

printf("KW\_TYPE\n");

break;

case KW\_VAR:

printf("KW\_VAR\n");

break;

case KW\_INTEGER:

printf("KW\_INTEGER\n");

break;

case KW\_CHAR:

printf("KW\_CHAR\n");

break;

case KW\_ARRAY:

printf("KW\_ARRAY\n");

break;

case KW\_OF:

printf("KW\_OF\n");

break;

case KW\_FUNCTION:

printf("KW\_FUNCTION\n");

break;

case KW\_PROCEDURE:

printf("KW\_PROCEDURE\n");

break;

case KW\_BEGIN:

printf("KW\_BEGIN\n");

break;

case KW\_END:

printf("KW\_END\n");

break;

case KW\_CALL:

printf("KW\_CALL\n");

break;

case KW\_IF:

printf("KW\_IF\n");

break;

case KW\_THEN:

printf("KW\_THEN\n");

break;

case KW\_ELSE:

printf("KW\_ELSE\n");

break;

case KW\_WHILE:

printf("KW\_WHILE\n");

break;

case KW\_DO:

printf("KW\_DO\n");

break;

case KW\_FOR:

printf("KW\_FOR\n");

break;

case KW\_TO:

printf("KW\_TO\n");

break;

case SB\_SEMICOLON:

printf("SB\_SEMICOLON\n");

break;

case SB\_COLON:

printf("SB\_COLON\n");

break;

case SB\_PERIOD:

printf("SB\_PERIOD\n");

break;

case SB\_COMMA:

printf("SB\_COMMA\n");

break;

case SB\_ASSIGN:

printf("SB\_ASSIGN\n");

break;

case SB\_EQ:

printf("SB\_EQ\n");

break;

case SB\_NEQ:

printf("SB\_NEQ\n");

break;

case SB\_LT:

printf("SB\_LT\n");

break;

case SB\_LE:

printf("SB\_LE\n");

break;

case SB\_GT:

printf("SB\_GT\n");

break;

case SB\_GE:

printf("SB\_GE\n");

break;

case SB\_PLUS:

printf("SB\_PLUS\n");

break;

case SB\_MINUS:

printf("SB\_MINUS\n");

break;

case SB\_TIMES:

printf("SB\_TIMES\n");

break;

case SB\_SLASH:

printf("SB\_SLASH\n");

break;

case SB\_LPAR:

printf("SB\_LPAR\n");

break;

case SB\_RPAR:

printf("SB\_RPAR\n");

break;

case SB\_LSEL:

printf("SB\_LSEL\n");

break;

case SB\_RSEL:

printf("SB\_RSEL\n");

break;

case SB\_BACKSLASH:

printf("SB\_BACKSLASH\n");

break;

}

}

int scan(char \*fileName)

{

Token \*token;

if (openInputStream(fileName) == IO\_ERROR)

return IO\_ERROR;

token = getToken();

while (token->tokenType != TK\_EOF)

{

printToken(token);

free(token);

token = getToken();

}

free(token);

closeInputStream();

return IO\_SUCCESS;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

int main(int argc, char \*argv[])

{

if (argc <= 1)

{

printf("scanner: no input file.\n");

return -1;

}

if (scan(argv[1]) == IO\_ERROR)

{

printf("Can\'t read input file!\n");

return -1;

}

return 0;

}

/\* Parser \*/

#include <stdlib.h>

#include "reader.h"

#include "scanner.h"

#include "parser.h"

#include "error.h"

Token \*currentToken;

Token \*lookAhead;

void scan(void) {

Token\* tmp = currentToken;

currentToken = lookAhead;

lookAhead = getValidToken();

free(tmp);

}

void eat(TokenType tokenType) {

if (lookAhead->tokenType == tokenType) {

printToken(lookAhead);

scan();

} else missingToken(tokenType, lookAhead->lineNo, lookAhead->colNo);

}

void compileProgram(void) {

assert("Parsing a Program ....");

eat(KW\_PROGRAM);

eat(TK\_IDENT);

eat(SB\_SEMICOLON);

compileBlock();

eat(SB\_PERIOD);

assert("Program parsed!");

}

void compileBlock(void) {

assert("Parsing a Block ....");

if (lookAhead->tokenType == KW\_CONST) {

eat(KW\_CONST);

compileConstDecl();

compileConstDecls();

compileBlock2();

}

else compileBlock2();

assert("Block parsed!");

}

void compileBlock2(void) {

if (lookAhead->tokenType == KW\_TYPE) {

eat(KW\_TYPE);

compileTypeDecl();

compileTypeDecls();

compileBlock3();

}

else compileBlock3();

}

void compileBlock3(void) {

if (lookAhead->tokenType == KW\_VAR) {

eat(KW\_VAR);

compileVarDecl();

compileVarDecls();

compileBlock4();

}

else compileBlock4();

}

void compileBlock4(void) {

compileSubDecls();

compileBlock5();

}

void compileBlock5(void) {

eat(KW\_BEGIN);

compileStatements();

eat(KW\_END);

}

void compileConstDecls(void) {

assert("Parsing subconstants...");

while (lookAhead->tokenType == TK\_IDENT)

compileConstDecl();

assert("Subconstants parsed!");

}

void compileConstDecl(void) {

assert("Parsing constant...");

eat(TK\_IDENT);

eat(SB\_EQ);

compileConstant();

eat(SB\_SEMICOLON);

assert("Constant parsed!");

}

void compileTypeDecls(void) {

assert("Parsing subtypes...");

while (lookAhead->tokenType == TK\_IDENT)

compileTypeDecl();

assert("Subtypes parsed!");

}

void compileTypeDecl(void) {

assert("Parsing a type...");

eat(TK\_IDENT);

eat(SB\_EQ);

compileType();

eat(SB\_SEMICOLON);

assert("Type parsed!");

}

void compileVarDecls(void) {

// assert("Parsing variables...");

while(lookAhead->tokenType == TK\_IDENT)

compileVarDecl();

// assert("Variables parsed");

}

void compileVarDecl(void) {

// assert("Parsing a variable...");

eat(TK\_IDENT);

eat(SB\_COLON);

compileType();

eat(SB\_SEMICOLON);

// assert("Variable parsed!");

}

void compileSubDecls(void) {

assert("Parsing subtoutines ....");

if (lookAhead->tokenType == KW\_FUNCTION) {

compileFuncDecl();

compileSubDecls();

} else if (lookAhead->tokenType == KW\_PROCEDURE) {

compileProcDecl();

compileSubDecls();

}

assert("Subtoutines parsed ....");

}

void compileFuncDecl(void) {

assert("Parsing a function ....");

eat(KW\_FUNCTION);

eat(TK\_IDENT);

compileParams();

eat(SB\_COLON);

compileBasicType();

eat(SB\_SEMICOLON);

compileBlock();

eat(SB\_SEMICOLON);

assert("Function parsed ....");

}

void compileProcDecl(void) {

assert("Parsing a procedure ....");

eat(KW\_PROCEDURE);

eat(TK\_IDENT);

compileParams();

eat(SB\_SEMICOLON);

compileBlock();

eat(SB\_SEMICOLON);

assert("Procedure parsed ....");

}

void compileUnsignedConstant(void) {

assert("Parsing a constant....");

switch (lookAhead->tokenType) {

case TK\_NUMBER:

eat(TK\_NUMBER);

break;

case TK\_IDENT:

eat(TK\_IDENT);

break;

case TK\_CHAR:

eat(TK\_CHAR);

break;

default:

error(ERR\_INVALIDCONSTANT, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileConstant(void) {

//assert("Parsing a constant...");

switch(lookAhead->tokenType) {

case SB\_PLUS:

eat(SB\_PLUS);

compileConstant2();

break;

case SB\_MINUS:

eat(SB\_MINUS);

compileConstant2();

break;

case TK\_CHAR:

eat(TK\_CHAR);

break;

default:

compileConstant2();

break;

}

assert("Constant parsed!");

}

void compileConstant2(void) {

switch (lookAhead->tokenType) {

case TK\_IDENT:

eat(TK\_IDENT);

break;

case TK\_NUMBER:

eat(TK\_NUMBER);

break;

default:

error(ERR\_INVALIDCONSTANT, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileType(void) {

switch (lookAhead->tokenType) {

case KW\_INTEGER:

eat(KW\_INTEGER);

break;

case KW\_CHAR:

eat(KW\_CHAR);

break;

case TK\_IDENT:

eat(TK\_IDENT);

break;

case KW\_ARRAY:

eat(KW\_ARRAY);

eat(SB\_LSEL);

eat(TK\_NUMBER);

eat(SB\_RSEL);

eat(KW\_OF);

compileType();

break;

default:

error(ERR\_INVALIDTYPE, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileBasicType(void) {

switch (lookAhead->tokenType) {

case KW\_INTEGER:

eat(KW\_INTEGER);

break;

case KW\_CHAR:

eat(KW\_CHAR);

break;

default:

error(ERR\_INVALIDBASICTYPE, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileParams(void) {

switch (lookAhead->tokenType) {

case SB\_LPAR:

eat(SB\_LPAR);

compileParam();

compileParams2();

eat(SB\_RPAR);

break;

case SB\_COLON:

case SB\_SEMICOLON:

break;

default:

error(ERR\_INVALIDPARAM, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileParams2(void) {

switch (lookAhead->tokenType) {

case SB\_SEMICOLON:

eat(SB\_SEMICOLON);

compileParam();

compileParams2();

break;

case SB\_RPAR:

break;

default:

error(ERR\_INVALIDPARAM, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileParam(void) {

switch (lookAhead->tokenType) {

case TK\_IDENT:

eat(TK\_IDENT);

eat(SB\_COLON);

compileBasicType();

break;

case KW\_VAR:

eat(KW\_VAR);

eat(TK\_IDENT);

eat(SB\_COLON);

compileBasicType();

break;

default:

error(ERR\_INVALIDPARAM, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileStatements(void) {

compileStatement();

compileStatements2();

}

void compileStatements2(void) {

switch (lookAhead->tokenType) {

case SB\_SEMICOLON:

eat(SB\_SEMICOLON);

compileStatement();

compileStatements2();

break;

// Follow

case KW\_END:

break;

// Error

default:

error(ERR\_INVALIDSTATEMENT, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileStatement(void) {

switch (lookAhead->tokenType) {

case TK\_IDENT:

compileAssignSt();

break;

case KW\_CALL:

compileCallSt();

break;

case KW\_BEGIN:

compileGroupSt();

break;

case KW\_IF:

compileIfSt();

break;

case KW\_WHILE:

compileWhileSt();

break;

case KW\_FOR:

compileForSt();

break;

// EmptySt needs to check FOLLOW tokens

case SB\_SEMICOLON:

case KW\_END:

case KW\_ELSE:

break;

// Error occurs

default:

error(ERR\_INVALIDSTATEMENT, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileAssignSt(void) {

assert("Parsing an assign statement ....");

eat(TK\_IDENT);

if (lookAhead->tokenType == SB\_LSEL) {

compileIndexes();

}

eat(SB\_ASSIGN);

compileExpression();

assert("Assign statement parsed ....");

}

void compileCallSt(void) {

assert("Parsing a call statement ....");

eat(KW\_CALL);

eat(TK\_IDENT);

compileArguments();

assert("Call statement parsed ....");

}

void compileGroupSt(void) {

assert("Parsing a group statement ....");

eat(KW\_BEGIN);

compileStatements();

eat(KW\_END);

assert("Group statement parsed ....");

}

void compileIfSt(void) {

assert("Parsing an if statement ....");

eat(KW\_IF);

compileCondition();

eat(KW\_THEN);

compileStatement();

if (lookAhead->tokenType == KW\_ELSE)

compileElseSt();

assert("If statement parsed ....");

}

void compileElseSt(void) {

eat(KW\_ELSE);

compileStatement();

}

void compileWhileSt(void) {

assert("Parsing a while statement ....");

eat(KW\_WHILE);

compileCondition();

eat(KW\_DO);

compileStatement();

assert("While statement pased ....");

}

void compileForSt(void) {

assert("Parsing a for statement ....");

eat(KW\_FOR);

eat(TK\_IDENT);

eat(SB\_ASSIGN);

compileExpression();

eat(KW\_TO);

compileExpression();

eat(KW\_DO);

compileStatement();

assert("For statement parsed ....");

}

void compileArguments(void) {

switch (lookAhead->tokenType) {

case SB\_LPAR:

eat(SB\_LPAR);

compileExpression();

compileArguments2();

eat(SB\_RPAR);

break;

// Follow - same as call statement as statement:

case SB\_SEMICOLON:

case KW\_END:

case KW\_ELSE:

// Follow - term2

case SB\_TIMES:

case SB\_SLASH:

// Follow - expression3

// Follow (For statement)

case KW\_TO:

case KW\_DO:

// Follow (arguments2)

case SB\_COMMA:

// Follow (condition2)

case SB\_EQ:

case SB\_NEQ:

case SB\_LE:

case SB\_LT:

case SB\_GE:

case SB\_GT:

// Follow (factor)

case SB\_RPAR:

// Follow (indexes)

case SB\_RSEL:

// Follow (if statement)

case KW\_THEN:

break;

// Error

default:

error(ERR\_INVALIDARGUMENTS, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileArguments2(void) {

switch (lookAhead->tokenType) {

case SB\_COMMA:

eat(SB\_COMMA);

compileExpression();

compileArguments2();

break;

// Follow

case SB\_RPAR:

break;

// Error:

default:

error(ERR\_INVALIDARGUMENTS, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileCondition(void) {

compileExpression();

compileCondition2();

}

void compileCondition2(void) {

switch (lookAhead->tokenType) {

case SB\_EQ:

eat(SB\_EQ);

compileExpression();

break;

case SB\_NEQ:

eat(SB\_NEQ);

compileExpression();

break;

case SB\_LE:

eat(SB\_LE);

compileExpression();

break;

case SB\_LT:

eat(SB\_LT);

compileExpression();

break;

case SB\_GE:

eat(SB\_GE);

compileExpression();

break;

case SB\_GT:

eat(SB\_GT);

compileExpression();

break;

default:

error(ERR\_INVALIDCOMPARATOR, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileExpression(void) {

assert("Parsing an expression");

switch (lookAhead->tokenType) {

case SB\_PLUS:

eat(SB\_PLUS);

compileExpression2();

break;

case SB\_MINUS:

eat(SB\_MINUS);

compileExpression2();

break;

default:

compileExpression2();

break;

}

assert("Expression parsed");

}

void compileExpression2(void) {

compileTerm();

compileExpression3();

}

void compileExpression3(void) {

switch(lookAhead->tokenType) {

case SB\_PLUS:

eat(SB\_PLUS);

compileTerm();

compileExpression3();

break;

case SB\_MINUS:

eat(SB\_MINUS);

compileTerm();

compileExpression3();

break;

// Follow (statement)

case SB\_SEMICOLON:

case KW\_END:

case KW\_ELSE:

// Follow (For statement)

case KW\_TO:

case KW\_DO:

// Follow (arguments2)

case SB\_COMMA:

// Follow (condition2)

case SB\_EQ:

case SB\_NEQ:

case SB\_LE:

case SB\_LT:

case SB\_GE:

case SB\_GT:

// Follow (factor)

case SB\_RPAR:

// Follow (indexes)

case SB\_RSEL:

// Follow (if statement)

case KW\_THEN:

break;

// Error

default:

error(ERR\_INVALIDEXPRESSION, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileTerm(void) {

compileFactor();

compileTerm2();

}

void compileTerm2(void) {

switch (lookAhead->tokenType) {

case SB\_TIMES:

eat(SB\_TIMES);

compileFactor();

compileTerm2();

break;

case SB\_SLASH:

eat(SB\_SLASH);

compileFactor();

compileTerm2();

break;

// Follow - same as expression3

case SB\_PLUS:

case SB\_MINUS:

// Follow (statement)

case SB\_SEMICOLON:

case KW\_END:

case KW\_ELSE:

// Follow (For statement)

case KW\_TO:

case KW\_DO:

// Follow (arguments2)

case SB\_COMMA:

// Follow (condition2)

case SB\_EQ:

case SB\_NEQ:

case SB\_LE:

case SB\_LT:

case SB\_GE:

case SB\_GT:

// Follow (factor)

case SB\_RPAR:

// Follow (indexes)

case SB\_RSEL:

// Follow (if statement)

case KW\_THEN:

break;

default:

error(ERR\_INVALIDTERM, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileFactor(void) {

switch (lookAhead->tokenType) {

case TK\_NUMBER:

case TK\_CHAR:

compileUnsignedConstant();

break;

case SB\_LPAR:

eat(SB\_LPAR);

compileExpression();

eat(SB\_RPAR);

break;

case TK\_IDENT:

eat(TK\_IDENT);

switch(lookAhead->tokenType) {

case SB\_LSEL:

compileIndexes();

break;

case SB\_LPAR:

compileArguments();

break;

default:

break;

}

break;

default:

error(ERR\_INVALIDFACTOR, lookAhead->lineNo, lookAhead->colNo);

break;

}

}

void compileIndexes(void) {

if (lookAhead->tokenType == SB\_LSEL) {

eat(SB\_LSEL);

compileExpression();

eat(SB\_RSEL);

compileIndexes();

}

}