

CS 355 Programming Language Design  
Homework 1: Turtle Interpreter  
Due 02/04/14

## 1. Introduction

For this programming project, you will create a simple interpreter (*turtle*) in C (using a *recursive descent parsing* technique) for the Turtle Programming Language described in Section 2. Your interpreter will build an *abstract syntax tree* (AST) for each top-level statement and execute this tree on the fly. The program will read the input source code from standard input and write a Portable Gray Map (PGM) image to standard output. This image contains the line drawing specified by the input program. You can view the image by standard image viewers.

Reviewing Section 6.6 in the textbook can be helpful in doing this assignment.

## 2. Turtle Programming Language

The grammar for our high-level turtle language is given in Figure 1. Curly braces {} and square brackets [] are used as meta-symbols as described in the caption to make the grammar more compact (they also hint at iteration and branching respectively in the corresponding recursive descent subroutines).

<i>program</i>	→	<i>stmt_seq</i> \$	(1)
<i>stmt_seq</i>	→	<i>stmt</i> { <i>stmt</i> }	(2)
<i>stmt</i>	→	<i>assign</i>   <i>while_stmt</i>   <i>if_stmt</i>   <i>action</i>	(3)
<i>assign</i>	→	IDENT ASSIGN <i>expr</i>	(4)
<i>block</i>	→	<i>stmt</i> { <i>stmt</i> }	(5)
<i>while_stmt</i>	→	WHILE <i>bool</i> DO <i>block</i> OD	(6)
<i>if_stmt</i>	→	IF <i>bool</i> THEN <i>block</i> {ELSIF <i>bool</i> THEN <i>block</i> } [ELSE <i>block</i> ] FI	(7)
<i>action</i>	→	HOME   PENUP   PENDOWN   FORWARD <i>expr</i>	(8)
	→	LEFT <i>expr</i>   RIGHT <i>expr</i>   PUSHSTATE   POPSTATE	(9)
<i>expr</i>	→	<i>term</i> {+ <i>term</i>   - <i>term</i> }	(10)
<i>term</i>	→	<i>factor</i> {* <i>factor</i>   / <i>factor</i> }	(11)
<i>factor</i>	→	- <i>factor</i>   + <i>factor</i>   ( <i>expr</i> )   IDENT   REAL	(12)
<i>bool</i>	→	<i>bool_term</i> {OR <i>bool_term</i> }	(13)
<i>bool_term</i>	→	<i>bool_factor</i> {AND <i>bool_factor</i> }	(14)
<i>bool_factor</i>	→	NOT <i>bool_factor</i>   ( <i>bool</i> )   <i>cmp</i>	(15)
<i>cmp</i>	→	<i>expr</i> <i>cmp_op</i> <i>expr</i>	(16)
<i>cmp_op</i>	→	=   NE   <   LE   >   GE	(17)

Figure 1: Grammar for Turtle language. {*a*} specifies that *a* can occur zero or more times and [*a*] denotes that *a* is optional. The \$ in the first production indicates that there should be no more tokens following *stmt\_seq*. Production 2 represents a sequence of top-level statements, whereas Production 5 denotes a list of statements nested inside another statement construct.

The terminals (i.e., tokens) of the languages are +, -, \*, /, (, ), =, <, >, and the multi-character tokens are listed in Table 1. The reserved words are given in Table 2. The remainder of a line is ignored following a # symbol which allows the programmer to insert comments into the source code.

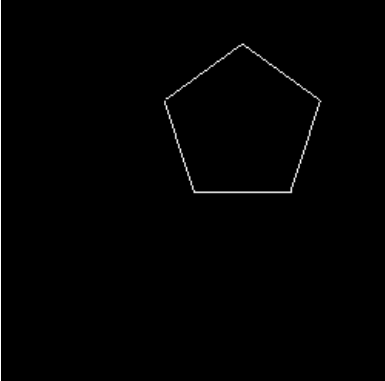
<i>token</i>	<i>regular expression</i>
IDENT	<code>[a-zA-Z\_][a-zA-Z0-9\_]*</code>
REAL	<code>[0-9]+([\.] [0-9]*)?</code>
ASSIGN	<code>:=</code>
NE	<code>&lt;&gt;</code>
LE	<code>&lt;=</code>
GE	<code>&gt;=</code>

Table 1: Multi-character tokens.

OR	AND	NOT	WHILE
DO	OD	IF	THEN
ELSIF	ELSE	FI	HOME
PENDOWN	PENUP	FORWARD	RIGHT
LEFT	PUSHSTATE	POPSTATE	

Table 2: Reserved words.

The code below is a simple program for drawing an N side polygon. Note that our language is case sensitive and reserved words are always in upper case. Variables (i.e., user defined identifiers) are used to hold floating point numbers and are never declared, but “pop” into existence the first time they are referenced and are initialized with a value of zero.

Program	Output
<pre> N := 5           # number of sides L := 10          # length of a side THETA := 360 / N # internal angle PENDOWN I := 1 WHILE I &lt;= N DO     FORWARD L     LEFT THETA     I := I + 1 OD PENU </pre>	

### 3. Lexical Analysis

You will be provided with the C source code (in `/cs_Share/class/cs355/hw1`) for a scanner that returns the next token in the input stream read from stdin.

```

enum { /* non-single char tokens */
    IDENT_ = 256, ASSIGN_, REAL_, NE_, LE_, GE_, OR_, AND_, NOT_,
    WHILE_, DO_, OD_, IF_, THEN_, ELSIF_, ELSE_, FI_,
    HOME_, PENUP_, PENDOWN_, FORWARD_, RIGHT_, LEFT_, PUSHSTATE_, POPSTATE_
};

typedef union { /* lexeme associated with certain tokens */
    float f; /* REAL_ */
    char *s; /* IDENT_ */
} LVAL;

extern int lineno; /* current source code line number */

```

```

/*
 * Returns the next token/lexeme read from stdin.
 * Returns 0 when there are no more tokens.
 */

```

```

int nextToken(LVAL *lval);

```

Tokens are encoded as `int` via the `enum` shown above (single character tokens are encoded with their ASCII value) and `lval` represents the lexeme associated with the token.

## 4. Symbol Table

Since all variables are global and representing floating-point numbers, we can simply represent a symbol table as a linked list where each node holds the identifier string (i.e., the symbol) and its associated value. Variables are not declared and they “pop into existence” when they are first referenced (they initially hold the value 0). Variables are updated via assignment statements and can be referenced in expressions.

### 4.1. Recursive Decent Parsing and On-the-fly execution

There is one subroutine for each non-terminal in the grammar. The right hand side (RHS) of the corresponding productions dictates the body of each subroutine:

- Non-terminals on the RHS map to (possibly recursive) subroutine calls and
- Terminal symbols have to be “matched.”

When there are multiple rewrite rules for a non-terminal, deciding which procedure to call is determined using via a single “look-ahead” token. Since *program* is the start symbol, we begin parsing by fetching the first token and calling *program()* (shown in Figure 3) which starts the parsing process. Using the first production as a guide, *program()* invokes *stmt\_seq()* which parses and executes each top-level statement.

```

int lookahead; /* next token */
LVAL lval;     /* value associated with token */
...
void program(void) {
    stmt_seq();
    if (lookahead != 0) syntax_error("Extraneous input!");
}
...
int main(void) {
    ...
    lookahead = nextToken(&lval);
    program();
    ...
    return 0;
}

```

Figure 3: Parsing is initiated in *main()* by reading the first look-ahead token and calling the subroutine *program()* associated with the start symbol *program*.

### 4.2. Abstract Syntax Trees

The interpreter proceeds by building an abstract syntax tree (AST) for each top-level statement and then executing it as shown by the code in Figure 4.

```

void stmt_seq(void) {
    do {
        Stmt *s = stmt(); /* build syntax for next statement */
        executeStmt(s);   /* execute syntax tree */
    }
}

```

```

        destroyStmt(s); /* delete syntax since it is no longer needed */
    } while (stmtPrefix()); /* keep looping if there are more statements */
}

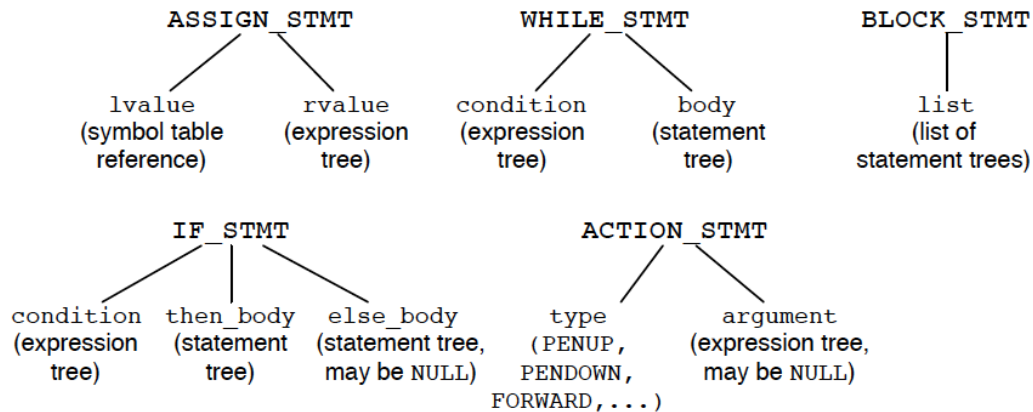
```

Figure 4: Using Production 2 from the grammar in Figure 1 as our guide, this routine fetches and executes each top-level syntax tree. The `stmtPrefix()` function returns true when the lookahead token is a prefix for another statement (i.e., is `IDENT`, `WHILE`, `IF`, etc. . . ).

In this case, there are two flavors of syntax trees: *statement trees* and *expression trees*. A statement tree represents a program statement that can be executed and an expression tree denotes an arithmetic or boolean expression that can be evaluated. The structure of each tree depends on the specific statement it represents.

#### 4.2.1. Statement Trees

Figure 5 shows five different kinds of statement trees. What each statement tree has in common with the others is that it can be executed. Therefore we would like to represent all statement trees using a single data type named *Stmt*.



We use the “old-school” technique for encoding multiple data structures into one using C's union mechanism as shown below.

```

typedef struct Stmt {
    int type; /* ASSIGN_STMT, WHILE_STMT, BLOCK_STMT, etc... */
    union {
        struct {SymTab *lval; Expr *rval;} assign_;
        struct {Expr *cond; struct Stmt *body;} while_;
        struct {struct Stmt *list;} block_;
        ...
    } s;
    struct Stmt *next; /* link-list field used by block statements */
} Stmt;

void executeStmt(Stmt *stmt) { /* executes a statement tree */
    switch(stmt->type) {
        case ASSIGN_STMT:
            stmt->s.assign_.lval->val = evalExpr(stmt->s.assign_.rval);
            break;
        case WHILE_STMT:
            while (evalExpr(stmt->s.while_.cond) != 0)
                executeStmt(stmt->s.while_.body);
            break;
    }
}

```

```

        ...
    }
}

```

Figure 6: Stmt data type that uses a C union to encode multiple statement tree structures as one data type. The executeStmt() function “executes” any given statement tree.

#### 4.2.2. Expression Trees

An expression tree represents an expression that can be evaluated - in our case all expressions evaluate to a floating-point number (Boolean expressions evaluate to zero (false) or non-zero (true)). Figure 8 shows how we might encode all the various types of expression trees using the single type named Expr. Figure 9 mimics Production 10 from the grammar in Figure 1 to parse arithmetic expressions.

```

typedef struct Expr {
    int type; /* NUM_EXPR, VAR_EXPR, ADD_EXPR, ... */
    union {
        float num; /* NUM_EXPR */
        SymTab *sym; /* VAR_EXPR */

        /* unary operation (NEG_EXPR, ...) */
        struct Expr *unary;

        /* binary operation (ADD_EXPR,...) */
        struct {struct Expr *left, *right;} binary;
    } op;
} Expr;

float evalExpr(Expr *expr) { /* evaluate and expression tree */
    switch(expr->type) {
        case NUM_EXPR: return expr->op.num;
        case VAR_EXPR: return expr->op.sym->val;
        case ADD_EXPR: return evalExpr(expr->op.binary.left) +
            evalExpr(expr->op.binary.right);
        ...
    }
}

```

Figure 8: Expression tree data type and function for evaluating them.

```

Expr *expr(void) {
    Expr *e = term();
    while(1) {
        if (lookahead == '+') {
            match('+');
            e = createBinaryExpr(ADD_EXPR, e, term());
        } else if (lookahead == '-') {
            match('-');
            e = createBinaryExpr(SUB_EXPR, e, term());
        } else {
            break;
        }
    }
    return e;
}

```

}

*Figure 9: Recursive descent function for parsing arithmetic expressions.*

## 5. What to submit

You will be provided with C source code for the scanner and turtle graphics functions along with some test input programs. Please test your code thoroughly before submission. Implement all grammar constructs. Useful error message concerning lexical or syntax errors (along with a graceful exit with a non-zero status code) are required. Please create an archive file that contains the following:

- A text file named README that contains the following:
  - Author and contact information (email address);
  - Brief overview of the project including what works and what doesn't work yet.
  - List of instructions necessary for compiling, linking, and executing your program;
  - List of all files in the archive.
- All source code.
- A Makefile for building program.
- Any pertinent test files.

This homework will be submitted electronically on Angel. Please double check the files (e.g., download and open them) after you submit. It is due at midnight on the due date.