Department of Computer Science, Australian National University

COMP3610 – Principles of Programming Languages

Semester 2, 2009

Assignment 2: *Turtle* Compiler

Due: 5 pm Friday 23rd October, 2009

Outline

Your task in this assignment is to develop a compiler for a simple graphics-oriented programming language, *Turtle*, using Bison and Flex. The target is an artificial and basic line plotting machine, the *PDPlot-2*, described in more detail below. A simulator for this machine is available for you to test run the programs that your compiler produces.

1 Source language: *Turtle*

This section give a fairly detailed description of *Turtle* but it is not entirely rigorous. Some of the design details are left to your discretion and of course you are permitted to extend the language with extra features if you wish. However, you must not place extra syntactic or semantic *restrictions* on the language described here. Discuss it with me if you are unsure. Several example programs are given later in this handout and on the COMP3610 web pages.

1.1 Programs

A *Turtle* program begins with the key word `turtle` followed by an identifier naming the program. This is followed by a sequence of zero or more variable declarations, then zero or more function declarations, then a compound statement which is the body of the program.

1.2 Data types

There is only one data type: *integer*. Note that the target machine is 16-bit with 2's complement representation of integers.

1.3 Variable declarations

Variable declarations are in one the forms:

> `var` *ident* = *expression*
>
> `var` *ident*

The first declares a new variable with name *ident* and initialises it to the value of the *expression*. Note that the expression is evaluated in the scope at that point, so any variables or functions used in the expression must already be declared. See the discussion of scope below and look at some of the sample programs.

The second declares a new variable with name *ident* and initialises it to 0.

1.4 Function declarations

Function declarations are of the form:

```
fun ident (parameters)
    var declarations
    compound statement
```

The parameter list is a sequence of zero or more identifiers separated by commas. The variables declared in the function are local to the function.

The parameters are passed by *value*. That is, when a function is called a new store location is allocated for each parameter and it is initialised to the value of the corresponding argument expression (see the description of function calls below).

Functions may be recursive. It is up to you to decide how to manage the declaration of mutually recursive functions. One option is to use *prototype* declarations, as in C for example. Another approach is to allow a function call to precede its declaration and for your compiler to check later that it has eventually been declared.

1.5 Statements

There are several statements that are specific to the line graphics domain of application.

Up
The `up` command instructs the plotter pen to be lifted from the drawing surface. There is no effect if it is already raised.

Down
The `down` command instructs the plotter pen to be lowered to the drawing surface. There is no effect if it is already down.

Moveto
The statement:

```
moveto (expression, expression)
```

instructs the plotter to move the pen to the specified (*x*, *y*) coordinates.

The other statements are more general procedural statements, familiar from typical imperative programming languages.

Read
The statement:

```
read (identifier)
```

reads the next integer from the input stream and stores it in the variable named in the statement. For simplicity, each line of the input stream must consist of exactly one integer.

Assignment
The statement:

```
identifier = expression
```

evaluates the expression and stores it in the variable named on the left hand side of the assignment operator.

Conditional
The statement:

```
if (comparison) { statements }
```

executes the statements in the body if and only if the comparison condition is true. The statement:

```
if (comparison) { statements }
else { statements }
```

executes the first sequence of statements if the comparison condition is true and the second sequence if the condition is false.

Comparisons consist of two expressions with a relational operator, such as:

```
(expression == expression)
```

You should admit at least two relation operators: == and <.

For simplicity you may require the component statements to be compound statements. That is, they will always be delimited by braces even if there is only a single statement.

While loop
The statement:

```
while (comparison) { statements }
```

is a standard value loop. The comparison is evaluated and if it is true, the statements in the body are executed. This process is repeated until the comparison becomes false.

Return
The statement:

```
return expression
```

when it appears in a function, causes the expression to be evaluated and returned as the value of that function call. It should also cause the function call to terminate at that point.

A `return` statement in the body of the program (as distinct from a function) is illegal and your compiler should report it as an error.

Function call
Function calls are of the form:

```
identifier (arguments)
```

They may appear in expressions (as described below) or as statements, in which case the returned result is ignored.

When a function is called, the arguments (which are general expressions) are evaluated and stored in new locations corresponding to the parameters of the function definition. If a function is declared with *n* parameters, any calls of that function must have exactly *n* arguments.

Compound statement
A compound statement is constructed by surrounding a sequence of statements in braces:

```
{  statement
   statement
   ...
   statement  }
```

The statements are executed in order.

1.6 Expressions

Turtle expressions are simple arithmetic expressions built from:

- unsigned integer literals;
- variable identifiers; and
- function calls.

The arithmetic operators include addition, subtraction and multiplication (`+`,`-` and `*` respectively) with the usual precedence and associativity. There is also a unary negation operator (`-`) of higher precedence and parentheses to override the usual operator precedence.

Function calls in expressions
If a function call appears in an expression then its result is the value returned by a `return` statement in the function body. If no such `return` is executed, its value is `0`.

1.7 Scope of identifiers

The variables declared at the top level of the program are considered to be *global* and are in scope for the entire program, including the functions, unless a variable of the same name is declared as local to a function. As usual, variables declared within a function only exist within that function. You should ensure that your compiler gives an appropriate error message for incorrect variable references.

A function's formal parameters and its local variables occur in the same scope so they must have distinct names. For example, a declaration like:

```
fun beer (X,X,X,X)
   { ... }
```

is illegal, as is:

```
fun Hahn (Ice, Premium)
  var Ice
  ...
```

and so is:

```
var slab
var slab
```

Note that to allow a function to be directly recursive, the function name must be in scope within its own body.

A more subtle question is whether to treat function and variable names as sharing the same scope. Should any of the following declarations be illegal?

```
turtle namespace
  ...
  var a
  fun a () ...
  ...
```

```
fun b (b) ...
...
fun c ()
  var c
  ...
```

Provided that empty argument lists are indicated by '`()`' a compiler can always tell the difference between the use of a variable and a function call, in which case none of the uses of `a`, `b` or `c` above cause a problem. You might think it's bad programming practice to write code like this but language designers and their compiler writers should strive for flexibility rather than enforcing their own prejudices.

1.8 Numerals

Numeric literals are sequences of 1 or more digits. They are unsigned but there is a unary negation operator.

1.9 Identifiers

Identifiers begin with a letter and are sequences of letters, digits, underscores (_) and apostrophes (').

1.10 Reserved words

Turtle has the following reserved words:

```
turtle   var   fun   up   down   moveto
read   if   else   while   return
```

The following symbols are also lexical tokens of *Turtle*:

```
+   -   *   =   (   )   {   }   ,   <   ==
```

1.11 Comments

Comments are from `//` to the end of the line. There are no multi-line comments in *Turtle*. Of course you may include them if you wish.

2 Target machine: *PDPlot-2*

The *PDPlot-2* is a contrived line plotting machine with 16-bit words addressed from 0 to 2^{16}–1. It is not byte-addressable.

The machine has four registers:

- a program counter `PC` which is implicit and initialised to `0` at load time;
- a top of stack pointer `SP` that is also implicit;
- a global data pointer `GP` which is set to point to the lowest address of the data area (i.e. the next address past the code segment) at load time and remains fixed throughout execution;

- an activation frame pointer FP which is initially set to the same as GP but which is advanced to the bottom of the current activation record by the subroutine call instruction, and is reset to the previous activation record by the subroutine return instruction.

There are two condition codes, Z (zero) and N (negative). The condition codes are only changed by explicit use of the Test instruction.

Numbers, indexes and offsets are in 2's-complement representation.

2.1 Addressing modes

The only explicit addressing mode is *index* on GP and FP. So *x*(FP) selects the word whose address is the sum of the contents of FP and the 8-bit 2's-complement number *x*. Similarly for *x*(GP).

The stack operations are implicit auto-increment and auto-decrement on SP but there is also a programmer-controlled pop instruction. There is immediate addressing in the two-word instructions which is actually an implicit auto-increment on the PC, as usual.

2.2 Instructions

The machine has 19 instructions whose behaviours are described below. In this section we use an assembly language notation, but there is no assembler provided. (Building an assembler with Flex and Bison would be a straightforward exercise but is not necessary for this assignment.) The binary instruction formats are described in the next section.

Stack operations

Load *x*(GP), Load *x*(FP)
Pushes the value in the location given as an index offset of *x* from GP or FP onto the top of the stack. The index offset is a 2's-complement byte-length integer, –128 to +127. This is a one-word instruction.

Loadi #*n*
Pushes the value *n* onto the top of the stack. This is a two-word instruction, with *n* occupying the second word.

Store *x*(GP), Store *x*(FP)
Pops the stack and stores the value in the location given as an index offset of *x* from GP or FP. The index offset is a 2's-complement byte-length integer, –128 to +127. This is a one-word instruction.

Pop #*n*
Pops the top *n* words from the stack. Put another way, subtract *n* from the current value of the SP. This is a two-word instruction, with *n* occupying the second word. For our language, Pop is mainly useful for removing arguments following return from a subroutine call.

Add
The top two words of the stack are replaced by a single word containing their sum.

Sub
The top word of the stack is subtracted from the second-top word. The two words are replaced by a single word containing their difference.

Neg
The top word of the stack is negated (2's-complement).

Mul
The top two words of the stack are multiplied and replaced by a single word containing their product.

`Test`
The top of stack value is inspected and condition code `Z` is set if the value is zero, cleared if it is non-zero. Condition code `N` is set if the value is negative (i.e. the top bit of the word is set), cleared otherwise. The top of stack word is not affected, nor is the `SP`.

I/O and plotter operations

`Up`
Lifts the pen off the drawing surface. There is no effect if the pen is already up.

`Down`
Lowers the pen to the drawing surface. There is no effect if the pen is already down.

`Move`
Instructs the pen to move to the `(`*x*`,`*y*`)` coordinates specified in the top two words on the stack. The *y* coordinate is on top of stack, the *x* coordinate is the second-top word. following the action, both words are popped from the stack.

`Read` *x*`(GP), Read` *x*`(FP)`
An optionally signed numeric literal is read from standard input and the value is stored in the location given as an index offset of *x* from `GP` or `FP`. The index offset is a 2's-complement byte-length integer, –128 to +127. This is a one-word instruction.

Control operations

`Jump #`*a*
An unconditional jump to the instruction at address *a*. That is, the `PC` is set to the value *a*. This is a two-word instruction, with *a* occupying the second word.

`Jeq #`*a*
If condition code `Z` is set then jump to the instruction at address *a*. Recall that `Z` is set if and only if the most recently `test`ed value is zero. This is a two-word instruction, with *a* occupying the second word.

`Jlt #`*a*
If condition code `N` is set then jump to the instruction at address *a*. Recall that `N` is set if and only if the most recently `test`ed value is negative. This is a two-word instruction, with *a* occupying the second word.

`Halt`
Stop execution of the program.

Subroutine operations

`Jsr #`*a*
Jump to the subroutine with entry address *a*. The return information is stored on the stack. More precisely, the `Jsr #`*a* instruction takes the following sequence of steps:

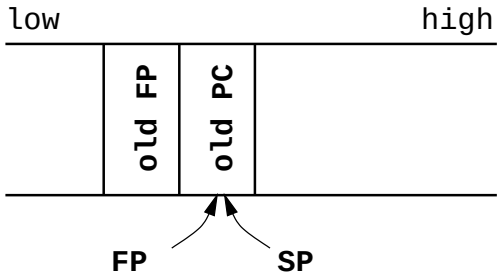
a. Push the address in the `FP` onto the stack;

b. Push the address in the `PC` onto the stack;

c. Set `FP` to point to the top of stack (i.e. `FP` = `SP`);

d. Jump to *a* (i.e. `PC` = *a*).

Immediately following a `Jsr` instruction the stack will look like this:



`Rts`
Return from subroutine. This instruction restores the values of the `FP` and `PC` saved by the most recent preceding `Jsr` instruction. In more detail:

a. The `SP` is set to point to the stack location two words below the `FP` (i.e. `SP` = `FP` – 2);

b. The `PC` is set to the value in the word currently pointed to by `FP` (see the diagram above);

c. The `FP` is set to the value in the word below that currently being pointed to by `FP` (again, see the diagram above).

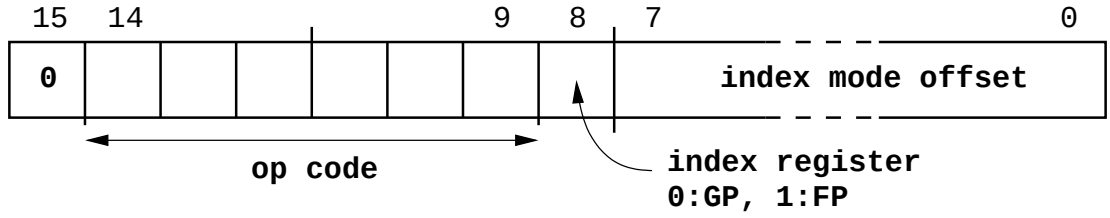
2.3 Instruction format

Instructions occupy a single word but those with immediate addressing are followed by a second word containing their operand.

The bottom byte is reserved for a 2's-complement offset of –128 to +127 for index mode instructions. The bottom bit of the top byte (bit 8) indicates the register to be indexed: 0 for `GP` and 1 for `FP`.

The top bit (15) is unused, and the op-code is bits 9 to 14 inclusive. Bit 14 distinguishes two-word instructions (bit 14 set) from one-word instructions (bit 14 cleared).

Here is a diagram:



The hexadecimal instruction formats are summarised in the following table:

Single word instructions

```
Halt  :  00 00
Up    :  0A 00
Down  :  0C 00
Move  :  0E 00
Add   :  10 00
Sub   :  12 00
Neg   :  22 00
Mul   :  14 00
Test  :  16 00
Rts   :  28 00
```

Single word instructions with offsets

```
Load  :  06 XX  or  07 XX (index on GP or FP respectively)
Store :  04 XX  or  05 XX
Read  :  02 XX  or  03 XX
```

Two word instructions

```
Jsr   :  68 00
Jump  :  70 00
Jeq   :  72 00
Jlt   :  74 00
Loadi :  56 00
Pop   :  5E 00
```

3 Building your compiler

This section give some suggestions and guidelines for the design and construction of your compiler. You are not obliged to follow them.

3.1 Symbol tables

For the *Turtle* language it is most convenient to maintain two separate symbol tables, one for function names and the other for parameter and variable names.

For functions you will need to keep a variety of information, including its entry address (for use in `Jsr` instructions) and the number of parameters.

Global variables will be addressed as an offset from `GP` while local variables are addressed as an offset from `FP` so your symbol table should indicate this distinction. The offset is simply the ordinal position of the declaration in the list. Parameters are also addressed relative to the `FP`, albeit with a *negative* index.

Note that parameters and local variables only exist in the scope of the function where they are defined so they must be removed from the symbol table when the compiler has finished translating the function. The simplest way to handle this requirement is to organise your symbol table as a stack.

3.2 Static semantic checks

Your compiler should ensure that it only accepts well-formed source programs. Therefore it should check and report static semantic errors, such as: undeclared identifiers (functions, variables or parameters); identifiers declared twice in the same scope; mismatch between the number of parameters in the definition of a function and the number of arguments in a call; a `return` statement in the main program, and so on.

3.3 Code generation

You should use an array (of short) to store your *PDPlot-2* code as it is generated. It is not possible to incrementally print your code to a file, because you may need to backpatch some jump instruction target addresses. Throughout this section I use an assembly language notation for ease of understanding.

Basics

At some points there is a close correspondence between the source and target languages, so the code generation is straightforward. For example, the `up` and `down` *Turtle* statements translate directly to *PDPlot-2* `Up` and `Down` instructions respectively.

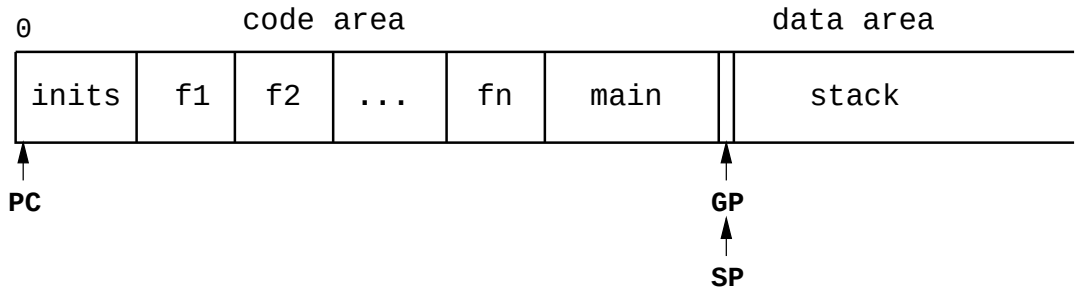
For the *Turtle* command `moveto(`*exp1*`,` *exp2*`)` your compiler should generate code to leave the values of *exp1* and *exp2* on top of the stack (in that order), followed by the `Move` instruction. A production and semantic action like:

```
stmt : MOVETO '(' exp ',' exp ')' { emit0 (move); }
```

should be sufficient because the semantic actions associated with parsing the two expressions will generate code to leave their values on top of the stack.

Machine representation

The following diagram indicates the suggested basic machine organisation for *Turtle* with the program ready to commence execution:



The *inits* section is the code to evaluate and load the global variables on top of stack (i.e. where the SP is pointing) so that they are in the expected position relative to the GP. For example, global declarations like:

```
turtle eg1
var x
var y = x + 42
```

should be translated to:

```
Loadi    #0          -- initialise x location to 0
Load     1(GP)       -- load x
Loadi    #42         -- load 42
Add                  -- initialise y location to x+42
```

The code for each function follows and then the code for the main body of the program appears. You will need to generate a jump from the end of the *inits* over the functions to the beginning of the start of *main*.

Function calls

For the call of a function like:

```
f (exp1, exp2, exp3)
```

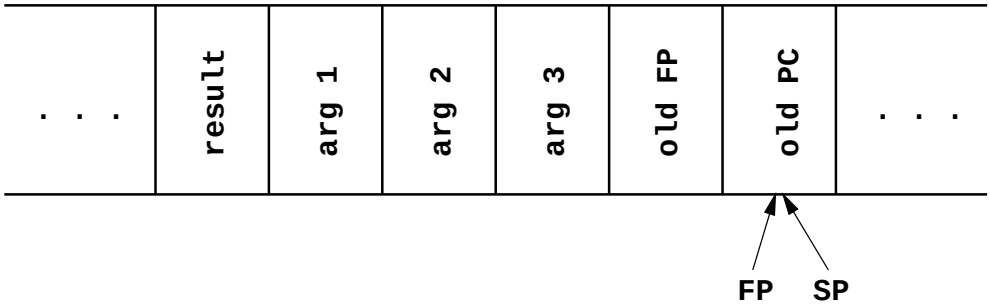
your compiler should generate code to do the following steps:

a. load 0 on the stack to reserve a location for the result of the function;

b. evaluate *exp1*, *exp2* and *exp3* in turn, thus leaving their values on the stack;

c. jump to the subroutine.

The resulting stack configuration will be:



where *arg1* is the value of *exp1* and so on. Your translation of function definitions should reflect this organisation. In particular:

- A `return` *exp* statement should evaluate *exp* and store its value in the reserved location. If the function has *n* parameters, this will be a `store` instruction, relative to the `FP` with offset (–*n*–2).
- The formal parameters are also referenced relative to the `FP`. Notice that the *first* parameter is (–*n*–1) away from `FP` and the *last* is at `-2(FP)`. Such is the nature of stacks. For example, assuming a definition like

  ```
  fun f (a, b, c, d)
    { ...
      a = c
      ...     }
  ```

  the assignment statement will be translated to:

  ```
  Load     -3(FP)          -- load c
  Store    -5(FP)          -- store in a
  ```

Backpatching jump addresses

In several situations your compiler will need to generate a jump instruction before it knows the target address. This happens in conditional statements, while loops and in the top level jump from the global initialisations over the function declarations, as mentioned above. For example, the statement:

```
if (exp1 == exp2) { body }
```

will translate to a template similar to the following. The addresses in the left hand column are only to make the example concrete.)

```
      < code to evaluate exp1 >
      < code to evaluate exp1 >
134   Sub
135   Test
136   Pop   #1
138   Jeq   #142
140   Jump  #???
142   < code for body >
???   ...
```

The target of the `Jeq` instruction is just a fixed offset to hop over the escape `Jump`, but your compiler cannot know the *target* of the escape `Jump` until after the code for the body has been generated.

A neat way to handle this is to store the address where the backpatch is to take place (`141` in the example above) as an *attribute* of an appropriate grammar symbol *before* parsing and translating the *`body`*. Then, *after* parsing the *`body`* you can backpatch that location with the current address.

For `if` statements, something like the following does the trick:

```
ifpart : IF comp        { ....
                          emit2 (jump, 0);
                          $$ = address-1; }

stmt   : ifpart block   { backpatch ($1, address); }
```

The `address` variable is an index into the code array indicating where the next instruction is to go.

In the `ifpart` production, the semantic action emits a `Jump` instruction with a dummy target address (`0`) and sets the attribute of the `ifpart` grammar symbol to index the element of the code array that contains the dummy value. Notice that this happens before the `block` is parsed.

Then, in the production for the full `if`-statement, the `ifpart` grammar symbol is available and hence so is its semantic attribute. After `block` has been parsed, and therefore translated, `address` will contain the required target of the `Jump` instruction, which we can now backpatch into the code location marked earlier as the attribute of `ifpart`. The `backpatch` function is simply an assignment to the specified element of the code array.

4 *PDPlot-2* simulator

Simulators for the *PDPlot-2* are available from the COMP3610 website. Note that the simulators expect code files to be *text*, not binary. This choice was originally made for compatability with http/cgi.

At present, the *PDPlot-2* simulators only have an address space of 4K which should be enough for your test programs. If you need more, let me know or just change the magic number in `PDPlot.hs` yourself.

The simulators may still have some defects. Students can be good stress testers so let me know if you believe you have found a defect in the simulator.

The output from *PDPlot-2* programs is rendered a simulator as a postscript image on an A4 page. The units of the co-ordinate system are pixels, so images should be limited to about 590 horizontal and 840 vertical.

A *dis*assembler is also provided at the COMP3610 web site. It may be a useful debugging tool. There are also a few sample *Turtle* programs and the corresponding *PDPlot-2* code that my compiler produces.

5 Notes

You may work in groups of up to *two* people for this assignment. Submissions from pairs will be assessed against exactly the same criteria as submissions from individuals.

Your submission should include Flex and Bison listings, test demonstrations and any other related files.

It is recommended that you work to develop a Flex/Bison specification of the grammar for *Turtle* without semantic actions by week 10 at the latest. You should show your grammar to me for feedback before proceeding. Weaknesses and errors in the syntactic definitions will almost certainly cause difficulties with the later stages of the assignment.

When building your code generation actions into your grammar, try to avoid being overwhelmed by details. Start with simple things like the evaluation of expressions, plotter directives and assignment statements. Take a bottom-up approach. In some productions, you may wish to include semantic actions between grammar symbols, rather than only at the end. Read up on this in the Bison manual first, and consult me before proceeding.

Instead of Flex, Bison, and the C programming language, you may use an equivalent lexer/parser generator pair such as Alex and Happy for the Haskell programming language. Other generators and languages may be acceptable, mail me if in doubt. Note that using tools besides Flex/Bison or Alex/Happy limits my ability to help you if you get stuck!

6 Marking Scheme

Approximately equal weight will be given to:

- grammar, token definitions and semanitc attributes
- symbol table and static semantic checks
- code generation

This assignment will count as 20% of the assessment in this course.

7 Submission

Submission is by e-mail to `Ben.Lippmeier@anu.edu.au`. Send a gzipped tarball of your Bison and Flex specifications (source files only) and any other associated files before the deadline. Include instructions for running your compiler. You should also provide several examples runs to demonstrate your program. Those examples should include demonstrations of your compiler handling a variety of syntactic errors and static semantic errors.

Make sure to clean out stray .o, .hi and executable files before submission.

8 Sample programs

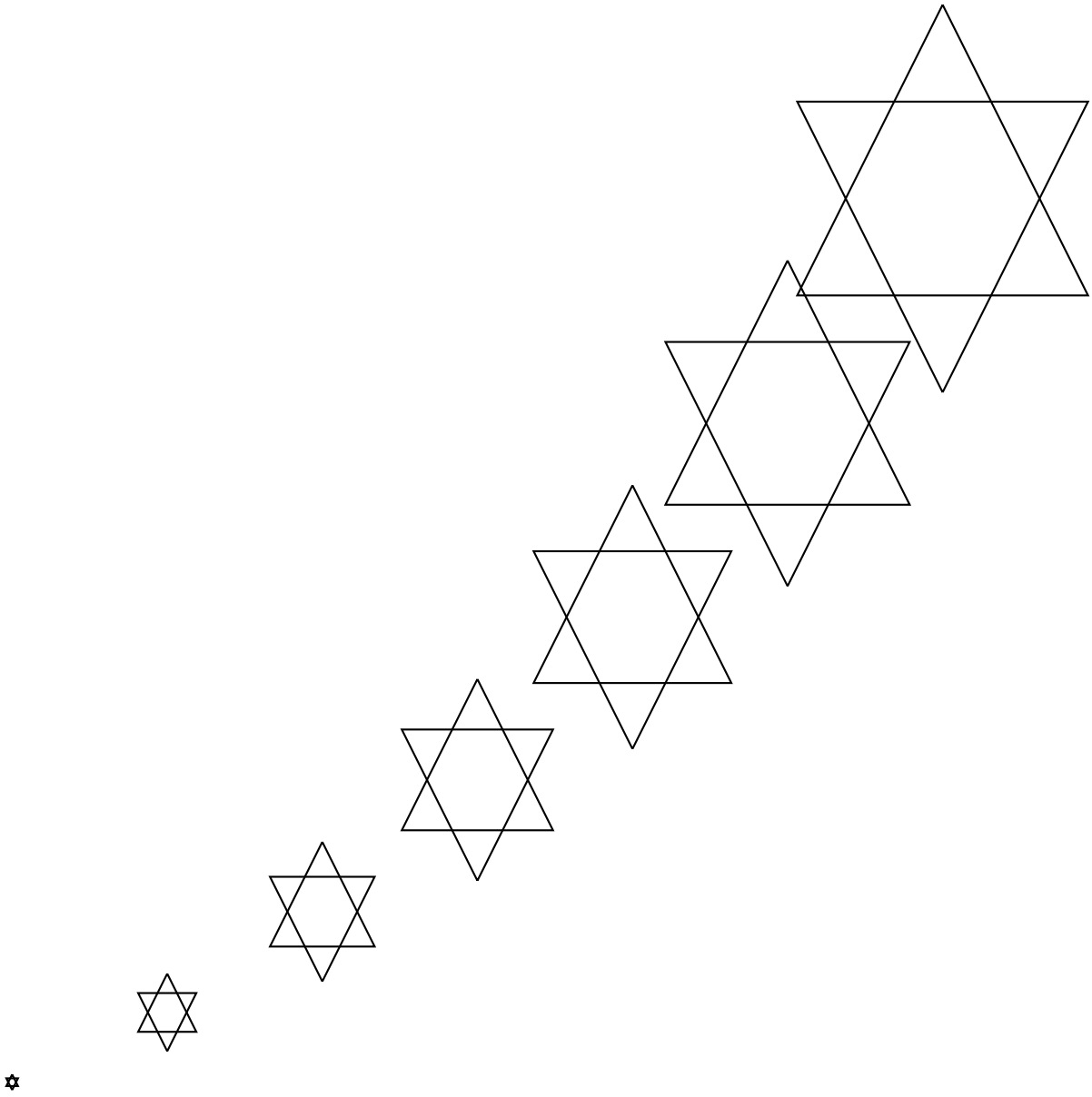
Here are a couple of simple *Turtle* programs. Their outputs follow.

The first one draws some stars:

```
turtle Twinkle
// Draw a few stars

var posX
var posY
var size

// Draw a single star
fun star (x, y, scale)
  var topY = y +8*scale
{
  up
  moveto (x, y)
  down
  moveto (x+3*scale, y+6*scale)
  moveto (x-3*scale, y+6*scale)
  moveto (x, y)
  up
  moveto (x, topY)
  down
  moveto (x+3*scale, y+2*scale)
  moveto (x-3*scale, y+2*scale)
  moveto (x, topY)
  up
}
// Main body of program
{
  posX = 500
  posY = 500
  size = 25
  while (0 < size) {
    star (posX, posY, size)
    posX = posX - 80
    posY = posY - 4*size
    size = size - 4
  }
}
```



This second program is a simple demonstration of a recursive function:

```
turtle perspective

// A simple recursive test

var width
var rate
var x = 200
var y = 300

// A rounding division function
fun div (dividend, divisor)
  var quotient = 0
{
  while (divisor < dividend) {
    quotient = quotient + 1
    dividend = dividend - divisor
  }
  if (dividend + dividend < divisor) {
    return quotient }
  else {
    return quotient + 1 }
}

// The recursive function
fun disappear (x, y, width, scale)
  var increment = div (width, scale * 2)
{
  if (0 < width) {
    up
    moveto (x, y)
    down
    moveto (x + width, y)
    disappear (x + increment,
               y + 2*increment,
               width - scale,
               scale) }
}

{
read (width)                              // about 500
read (rate)                               // about 20
disappear (x, y, width, rate)
}
```

Of course, you should begin testing your compiler with much simpler examples than these. Make sure you also test your compiler with a wide range of *illegal* programs, too.