**Turtle Project Documentation**

**Starting Grammar of Our Project**

<hatch> ::= hatch <stmtList> soup

<stmtList> ::= <instincts> <declarations> <commandList>

<instincts> ::= instinct <commandList> endinstinct

<declarations> ::= turtle <identifier>

<commandList> ::= <commands> <moreCommands>

<moreCommands> ::= ; <commands> | empty

<commands> ::= <identifier> trail

| <identifier> notrail

| <identifier> <direction>

| <identifier> color <color>

<direction> ::= forward <expr>

| right <expr>

| left <expr>

<expr> ::= {use our grammar here for operation precedence}

<number> ::= [0-9]+ | -[0-9]+

<identifier> ::= [a-z]+

<color> ::= red | blue | black | white | yellow | orange | purple | pink

**Language Design Considerations and Decisions**

1. ***Handling Numbers***

We realized early on that we might have a problem handling numbers in this language, because **turn** can use both positive and negative numbers, while **forward** and **do** can only use positive numbers. To prevent allowing translation of **forward** and **do** with negative numbers, we designed in our Lex file two separate tokens dealing with numbers, **NUM** and **NEGNUM**. Negative numbers could only be the latter while positive numbers could only be the former. This meant that in our Yacc file, a negative number could not get associated with a **forward** or **do** command because we did not create rules that would allow this to happen. This isn’t the only way to deal with positive and negative numbers in this kind of situation. Programmers could instead do an error check in the Yacc file as part of the action statements and then print an error message. We chose our method very early on and it worked; so we stuck with it.

|  |  |  |
| --- | --- | --- |
| **Lex Snippet** | **Yacc Snippet** | **Output** |
| …..  DIGIT [0-9]  NEG [-]  %%  {DIGIT}+ {  strcpy( yylval.str, yytext);  yylval.ival = atoi(yytext);  return NUM; }  {NEG}{DIGIT}+ {  strcpy( yylval.str, yytext);  yylval.ival = atoi(yytext);  return NEGNUM; }  ….. | | ID TURN NUM ';' {  *//action statements*  }  | ID TURN NEGNUM ';' {  *//same action statements*  } | bill.lt(90)  bill.lt(-30) |

1. ***Turning***

Our **turn** command is left-based. That is to say, if an instruction says to “turn 30”, the turtle will go 30 degrees **left**. If one wants to go 30 degrees right, one would need to say “turn 330” or “turn -30.” We could have just as easily chosen a rightward orientation for **turn**, but it made the most sense to just choose one and we chose left.

Our **left** and **right** commands will go 90 degrees only. So to go 180 degrees using **right** commands, the program would have to use the **right** command twice in a row.

**Turn Specification:**

| ID TURN NUM ';' {

strcpy($$.str, " " );

strcat($$.str, $1.str );

strcat($$.str, ".lt(");

strcat($$.str,$3.str);

strcat($$.str, ")");

strcat($$.str, "\n");

}

**Output Examples:**

bill.lt(90)

bill.lt(-30)

1. ***Shell Concerns***

For the **shell** command we needed a way to have the Yacc file “know” which turtle we were talking about so that it could translate to python statements that had the right turtle’s name. We did this by declaring a global variable called “currentTurtle” which would store the name of the current turtle when a **shell** statement was found in the program. Then we would call the grammar rule “SHLCL” which is basically just like our main command list, except the specifications in SHLCL did not have “ID” at the beginning of them. So for a statement such as “forward 10”, instead of beginning the string with $1.str (where the turtle’s ID would ordinarily be), we would instead begin with strcpy($$.str, currentTurtle). This proved to be a good solution for our problem.

1. ***Specification in main command list:***

| ID { strcpy(currentTurtle,$1.str); } INSTINCT ID ';'

1. ***Global variable declaration:***

char currentTurtle[20] = "currentTurtle";

1. ***Specification example in shell command list:***

| COLOR CS ';' {

strcpy($$.str, currentTurtle );

strcat($$.str, $2.str);

}

1. ***Input***

alex shell

color red;

forward 560;

endshell;

1. ***Output***

alex.color('red')

currentTurtle.forward(12)

1. ***ID Considerations***

Our Lex file was written in such a way that any group of letters that wasn’t another token would be interpreted as a TURTLEID token. Unfortunately when we needed to add instincts to the yacc file, it became apparent that this would be a problem…how to differentiate Turtle IDs from Instinct IDs? In the end, we decided not to. We changed TURTLEID to just ID and used that in our specifications for both turtle names and instinct names. The structure of the specifications did not demand differentiation for the yacc to still produce correct python programs.

***\*Although there are two IDs in this grammar rule, the ordering in the input program puts them where they need to be in the output program. The { strcpy(currentTurtle,$1.str); is probably not necessary here because in the actual definition for the function, currentTurtle is a formal parameter.***

***Specification:***

| ID { strcpy(currentTurtle,$1.str); } INSTINCT ID ';' {

strcpy($$.str,$4.str);

strcat($$.str, "( ");

strcat($$.str, $1.str);

strcat($$.str, " );");

strcat($$.str, "\n");

}

***Output (based on turtle and instinct IDs):***

square(alex);

1. ***Nesting and Tabbing***

Python cares about white space at the beginning of lines. To make sure that various statements would remain a part of the block statements that they were associated with, we needed a way to make sure there would be tabs where we needed them. So in addition to grammar rules like “CL”, “SHLCL”, “INCL”, and “DOCL”, we also created some more grammar rules called “SHLCLTABS” and “DOCLTABS.” In the main command list (CL), when a **do** statement appeared, we would call “DOCLTABS” for example, and the outputs from grammar rule would produce strings with tabs where we needed to be. We did the same thing with SHLCLTABS. What this means is that our program can accept a **shell**, **instinct**, or **do** within a **do** block, or a **shell** or **do** statement within an **instinct** block, or an **instinct** or **do** statement within a **shell** block, and all the tabs are as they should be to make it apparent which statements are part of which blocks. So we can nest blocks within other blocks, and in some cases even nest within the inner blocks. This is most likely a really clunky approach to this problem, but it produced correct python programs so we are satisfied.

|  |  |
| --- | --- |
| **“Outer” Do in Yacc** | **“Inner” Do in Yacc, Called Within Shell** |
| DLCMD  : ID TRAIL ';' {  strcpy($$.str, " " );  strcat($$.str,$1.str);  strcat($$.str, ".pendown()");  strcat($$.str, "\n");  }  .... | DLCMDTABS  : TRAIL ';' {  strcpy($$.str, " " );  strcat($$.str, currentTurtle );  strcat($$.str, ".pendown()");  strcat($$.str, "\n");  }  …. |

***\*Notice the extra tab beginning the string in the right column.***

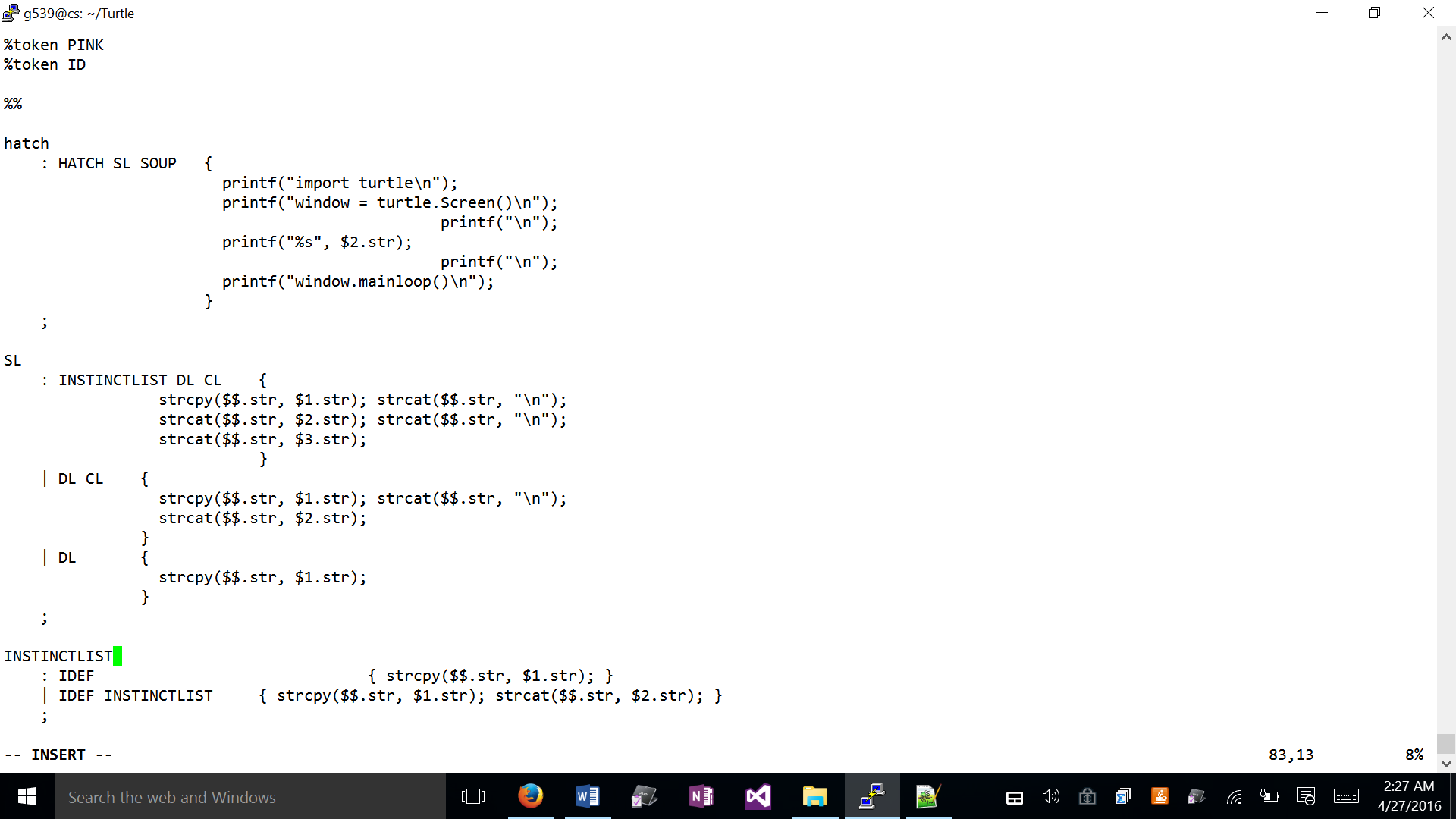
|  |  |
| --- | --- |
| ***Example Input:*** | ***Example Output:*** |
| do 2  alex instinct lineoftriangles;  bill shell  forward 10;  left;  turn -30;  endshell;  do 2  bill right;  bill forward 30;  enddo;  bill instinct lineoftriangles;  enddo; | for x in range(0, 2):  lineoftriangles( alex );  bill.forward(10)  bill.lt(90)  bill.lt(-30)  for x in range(0, 2):  bill.rt(90)  bill.forward(10)  lineoftriangles( bill ); |

1. ***Blank Lines***

We tried to have blank lines appear between major chunks of the program, such as between the instinct declarations, the variable declarations, and the statement list. This worked. In the process we also got some empty lines between statements in the statement list. Luckily Python doesn’t care and it would not affect the correctness of our programs, so we did not worry about a few extra blank lines.

1. ***Simple and Complex Programs***

After we added instincts to the program by including INSTINCTLIST in the specifications for the statement list (SL), we realized we could no longer run simplistic programs that didn’t contain instincts. To make sure our parser could still support simple programs, we added specifications in SL (along with appropriate action rules) that would work for programs that consisted of just declarations and statements, or just declarations.



1. ***Instincts to Functions***

Figuring out how to make instincts work was initially very confusing. We thought we might need some global variable array to store the names of instincts and then an action statement after an instinct was called to check which instinct to call. We were still thinking as if we were writing a *program* rather than a *parser*. But it was so simple. All we needed to do was print out a function declaration at the top of the program with a formal parameter according to python syntax, and a function call later in the program. It goes like this, where “currentTurtle” is the formal parameter, “jo” is a turle, and “square” is the instinct name:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | **Defining the Function/Instinct** |  |  |
| Input |  | Tokens passed to Yacc |  | Printed Output |
| instinct square  *statements;*  endInstinct | 🡪 | INSTINCT ID INCL ENDINSTINCT | 🡪 | def square(currentTurtle):  *statements* |
|  |  | **Calling the Instinct** |  |  |
| Input |  | Tokens Passed to Yacc |  | Printed Output |
| jo instinct square | 🡪 | ID INSTINCT ID |  | square(jo); |

The fact that “currentTurtle” is a formal parameter here while also being a global variable for other statements had no effect on the correctness of our program. Turning instincts into function declarations and function calls in python was easier than we thought it would be initially.

|  |  |
| --- | --- |
| ***Example Input:*** | ***Example Output:*** |
| instinct lineoftriangles  shell  forward 15;  right;  endshell;  do 5  color red;  turn 120;  forward 30;  color blue;  turn 120;  forward 30;  color yellow;  turn 120;  forward 30;  color pink;  turn 180;  forward 15;  right;  forward 40;  right;  forward 15;  enddo;  endinstinct | def lineoftriangles(currentTurtle):  currentTurtle.forward(15)  currentTurtle.rt(90)  for x in range(0, 5):  currentTurtle.color('red')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('blue')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('yellow')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('pink')  currentTurtle.lt(180)  currentTurtle.forward(15)  currentTurtle.rt(90)  currentTurtle.forward(40)  currentTurtle.rt(90)  currentTurtle.forward(15) |

1. ***Error Checking***

While some of our other solutions may be in-elegant, they still serve their purpose and produce correct programs. In terms of error checking, our project leaves much to be desired. When a user’s program has an error, the terminal prints out “syntax error” and that is all. There is no information about where the error is—the user only knows that something is wrong with their source program, and they have to go in and find it. But on the other hand, our project does not allow the creation of mostly-correct-but-not-quite-correct python programs, so we do have that going for us. However if we were to do this project against we would likely provide better error-checking.

**Project Location**

g539@cs:~/Turtle

**Project Files**

* lex.yy.c
* makefile
* pyturtle.output
* pyturtle.tab.c
* pyturtle.tab.h

**Yacc:**

pyturtle.y

**Lex:**

turtle.l

**Executable:**

turtle

(./turtle)

**Sample Input Files**

smallTest

bigTest

**Sample Output Files**

smallTest.py

bigTest.py

|  |  |
| --- | --- |
| **smallTest** | |
| **Input Program** | **Output Program** |
| hatch  instinct pentagon  color red;  forward 50;  turn 72;  color green;  forward 50;  turn 72;  color blue;  forward 50;  turn 72;  color pink;  forward 50;  turn 72;  color orange;  forward 50;  right;  notrail;  forward 40;  trail;  endinstinct  turtle alen;  alen trail;  do 9  alen instinct pentagon;  enddo;  soup | import turtle  window = turtle.Screen()  def pentagon(currentTurtle):  currentTurtle.color('red')  currentTurtle.forward(50)  currentTurtle.lt(72)  currentTurtle.color('green')  currentTurtle.forward(50)  currentTurtle.lt(72)  currentTurtle.color('blue')  currentTurtle.forward(50)  currentTurtle.lt(72)  currentTurtle.color('pink')  currentTurtle.forward(50)  currentTurtle.lt(72)  currentTurtle.color('orange')  currentTurtle.forward(50)  currentTurtle.rt(90)  currentTurtle.penup()  currentTurtle.forward(40)  currentTurtle.pendown()  alen=turtle.Turtle()  alen.shape('turtle')  alen.color('green')  alen.pendown()  for x in range(0, 9):  pentagon( alen );  window.mainloop() |
| **Picture Output** | |
|  | |

|  |  |
| --- | --- |
| **bigTest** | |
| **Input Program** | **Output Program** |
| hatch  instinct lineoftriangles  shell  forward 15;  right;  endshell;  do 5  color red;  turn 120;  forward 30;  color blue;  turn 120;  forward 30;  color yellow;  turn 120;  forward 30;  color pink;  turn 180;  forward 15;  right;  notrail;  forward 26;  trail;  forward 14;  right;  forward 15;  enddo;  endinstinct  instinct square  forward 20;  forward 20;  right;  forward 20;  right;  forward 20;  right;  forward 20;  right;  endinstinct  turtle alex;  turtle bill;  alex trail;  bill trail;  bill instinct square;  do 2  alex instinct lineoftriangles;  bill shell  forward 10;  left;  turn -30;  endshell;  do 2  bill right;  bill forward 30;  enddo;  bill instinct lineoftriangles;  enddo;  alex shell  turn 180;  forward 50;  instinct lineoftriangles;  do 5  turn 20;  forward 10;  enddo;  endshell;  soup | import turtle  window = turtle.Screen()  def lineoftriangles(currentTurtle):  currentTurtle.forward(15)  currentTurtle.rt(90)  for x in range(0, 5):  currentTurtle.color('red')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('blue')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('yellow')  currentTurtle.lt(120)  currentTurtle.forward(30)  currentTurtle.color('pink')  currentTurtle.lt(180)  currentTurtle.forward(15)  currentTurtle.rt(90)  currentTurtle.penup()  currentTurtle.forward(26)  currentTurtle.pendown()  currentTurtle.forward(14)  currentTurtle.rt(90)  currentTurtle.forward(15)  def square(currentTurtle):  currentTurtle.forward(20)  currentTurtle.forward(20)  currentTurtle.rt(90)  currentTurtle.forward(20)  currentTurtle.rt(90)  currentTurtle.forward(20)  currentTurtle.rt(90)  currentTurtle.forward(20)  currentTurtle.rt(90)  alex=turtle.Turtle()  alex.shape('turtle')  alex.color('green')  bill=turtle.Turtle()  bill.shape('turtle')  bill.color('green')  alex.pendown()  bill.pendown()  square( bill );  for x in range(0, 2):  lineoftriangles( alex );  bill.forward(10)  bill.lt(90)  bill.lt(-30)  for x in range(0, 2):  bill.rt(90)  bill.forward(30)  lineoftriangles( bill );  alex.lt(180)  alex.forward(50)  lineoftriangles( alex );  for x in range(0, 5):  alex.lt(20)  alex.forward(10)  window.mainloop() |
| **Picture Output** | |
|  | |