The Thue Language

COMP 261, Spring 2013

Team Six: Aaron, Jenna, Kenny, Nolan

For our team research project, we decided to examine the Thue programming language, an extremely simple progamming language with a strong relationship to the Turing machine. Due to the simplicity of Thue, we were able to write an interpreter for the language using Python. In demonstrating our team's research, we will first cover the background and syntax of the Thue language, followed by the complete code of our Thue interpreter.

1 Semi-Thue Systems:

The basis of the Thue Language is the concept of the Semi-Thue System, named after the Norwegian mathematician Axel Thue. It is a string rewriting system that is isomorphic to a universal grammar, and thus is Turing-complete. In this way, a simple way of looking at Semi-Thue Systems is to see them as similar to a CFG, the difference being that Semi-Thue Systems can describe Turing Machines, rather than just Pushdown Automata. One can build a Turing Machine that can simulate the derivation of a string with a Semi-Thue System. This Turing Machine would be nondeterministic to account for the possibility of the nondeterministic rules in the Semi-Thue System.

What is interesting about Semi-Thue systems is that they build upon the concept of a Context Free Grammar; a CFG can be seen as simply a Semi-Thue System with restrictions applied:

1. There must be a clear distinction between terminals and variable symbols
2. All rules must have one single, nonterminal variable as the LHS

3. All complete dervivations must begin with one single start variable, and end with a string consisting of only terminal symbols

2 The Thue Programming Language:

The programming language, Thue, implements the Semi-Thue System into a minimalistic language; the language, itself, belongs to a family of programming languages known as Turing Tarpits: those that exhibit Turing-completeness while having as few linguistic elements as possible. The syntax of Thue is thus remarkably simple, and quite similar to the creation of a CFG. Setting up the transition rules of a Thue program follows this pattern:

$$LHS ::= RHS$$

To specify characters or strings to be output, a tilde (˜) is added before the RHS, otherwise the argument will be interpreted as a transition:

$LHS ::=˜Hello$

Finally, an additional ::= is added at the end of the rules section; the following line will be interpreted as the input string. Let us look at this in the context of the classic Hello World! program:

a::=˜Hello World!

::=

a

This program outputs Hello World!, since it has symbol a as its input, and a Hello World! This is simple enough, so let us examine how Thue can express complex transitions that are not possible

on CFGs:

b::=~0

b::=~1

ac::=abc

::=

abc

This shows two major features of Thue, nondeterminism and long LHS support. When run, this code will randomly and recursively add zeroes and ones to the output string in an infinite loop. This is possible through the transition rule ac::=abc, which demonstrates Thue ability to nondeterministically choose zero or one and support a long LHS.

3 A Simple Thue Interpreter

As part of our project, we wrote a simple Thue interpreter in Python. While any number of languages could have been chosen, and a self-hosting Thue would have been more impressive, Python was chosen primarily for its short development cycle, ease of string manipulation, and familiarity to the author. The program was written from scratch following the language specification at *esolangs.org/wiki/Thue*, and as such may have a few unexpected, and undocumented, "features", but to the current knowledge of the authors has no bugs.

The code itself is fairly short and can be seen below:

```
#!/usr/bin/python
```

```
# thue.py - Aaron Laursen
#
# written for python version 3.3.1
#
# run as:
# python thue.py  rules.txt
#
# also accepts rules on stdin

import fileinput
import random

def main():
    start, rules = readRules()
    print(start)
    print(rules)
    print(runReps(start, rules))
    return

def readRules():
    # reads in a rules file of the form:
    #
    # lhs1::=rhs1
    # lhs2::=rhs2
    #
    # ::=start_state
    #
    # note that whitespace within lines is relevant,
    # empty lines are ignored,
    # literal "\n" in file if replaced with newline
    rules={}
    start=""
    for line in fileinput.input():
        if "::=" not in line:
            continue
        line=line[:-1].replace("\\n","\n")
        p=line.split("::=")
        if p[0]=="":
            start=p[1]
            continue
        if p[0] not in rules:
```

```
            rules[p[0]]=[]
        rules[p[0]].append(p[1])
    return start, rules

def parseRep(s):
    # takes care of ":::" and "~" in rpelacement
    while ":::" in s:
        s.replace(":::",input())
    if "~" in s:
        p=s.split("~",1)
        s=p[0]
        print(p[1])
    return s

def findAll(l,s):
    #finds all occurances of l in s
    if l=="" and s!="":
        return[]
    if l=="" and s=="":
        return [0,]
    p=[]
    i=s.find(l)
    while i !=-1:
        p.append(i)
        i=s.find(l,i+1)
    return p

def runReps(s, rules):
    # repeatedly applies rules to state until it cannot apply more
    while True:
        print("State:"+s)
        cand=[] #generate all posible rule applications
        for l in rules:
            if l not in s:
                continue
            for p in findAll(l,s):
                for r in rules[l]:
                    cand.append( (l,r,p) )
        if len(cand)==0:
            return s
        i=random.randint(0,len(cand)-1) #pick a rule randomly
```

```
        s=applyRule( s,cand[i] )

def applyRule(s, can):
    # takes a state and a candidate tuple and returns new state
    rhs = parseRep(can[1])
    s= s[:can[2]] + rhs + s[can[2]+len(can[0]):]
    return s

main()
```

A set of rules for this interpreter has the form:

```
lhs_1::=rhs_1
lhs_2::=rhs_2
lhs_3::=rhs_3
...
lhs_n::=rhs_n
::=$start_state$
```

For example, the rules to add one to the input _1111111111_ might be represented as:

```
1_::=1++
0_::=1

01++::=10
11++::=1++0

_0::=_
_1++::=10

::=_1111111111_
```

The program will then output the start state, rules, and state trace (series of states after each replacement), followed by the final state. For the above example, this should be:

```
_1111111111_
{'_0': ['_'], '1_': ['1++'], '0_': ['1'], '01++': ['10'],
'_1++': ['10'], '11++': ['1++0']}
State:_1111111111_
State:_1111111111++
State:_111111111++0
State:_11111111++00
State:_1111111++000
State:_111111++0000
State:_11111++00000
State:_1111++000000
State:_111++0000000
State:_11++00000000
State:_1++000000000
State:10000000000
10000000000
```

In many ways, the code speaks for itself, and the reader it encouraged to run it at their leisure (a copy in a more convenient form should have been included with this document). Please note, however, that it is written for Python 3, and will fail on earlier versions.

4 Conclusion

Throughout this paper, we have introduced the Thue language, and the systems underlying it. We also present our own program capable of interpreting the Language. This new formulation shows yet another Turing-equivalent, and new way of viewing the power of computations.