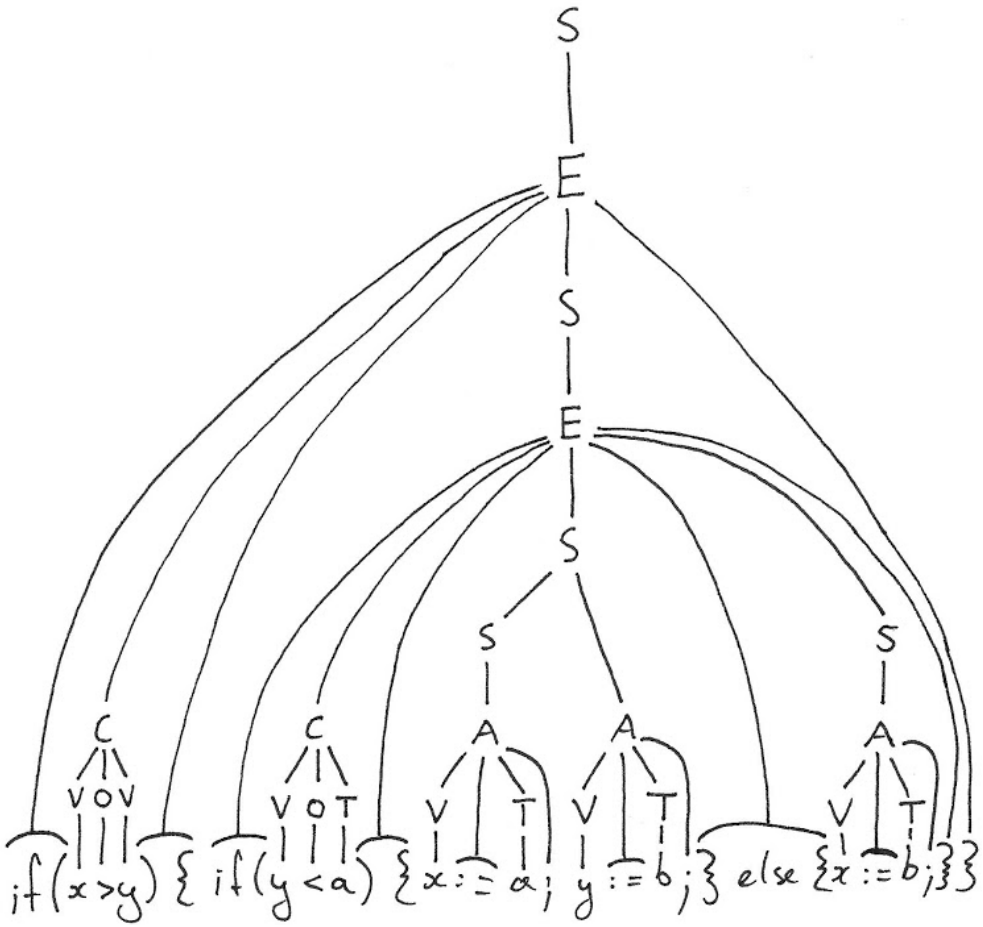
COMP2022 Assignment 2 - Anton Jurisevic - 311180051 - ajur4521

1)

a) **Symbols of the grammar G:**
Variables: {**S, E, A, T, V, C, O**}
Terminals: {**i, f, (,), {, }, e, l, s, :, =, ;, a, b, x, y, <, >**}

b) **The parse tree for "if (x > y) { if (y < a) {x := a; y :=b;} else {x := b;}}":**



2) **G is not LL(1) for the following reasons:**
i. The production rule **S → SA** is left-recursive. This may be remedied by modifying the rule for **S**, and adding a new rule, **R**:

S → AR | ER
R → AR | ε

ii. The rule **E → if (C) {S} | if (C) {S} else {S}** can be left factored:
E → if (C) {S} F
F → ε | else {S}

iii. `C -> VOT | TOV | VOV | TOT` contains non-disjoint **`FIRST`** sets (it can be left factored). It may be simplified in the following manner:

```
C → DOD
D → V | T
```

With these modifications, we obtain an equivalent grammar, **`G'`**.

```
S → AR | ER
R → AR | ε
E → if (C) {S} F
F → else {S} | ε
A → V := T;
T → a | b
V → x | y
C → D O D
D → V | T
O → < | >
```

3) **FIRST and FOLLOW sets of G'**

```
FIRST(S) = {x, y, i}        FOLLOW(S) = {$, }}
FIRST(R) = {x, y, ε}        FOLLOW(R) = {$, }}
FIRST(E) = {i}              FOLLOW(E) = {x, y, $, }}
FIRST(F) = {e, ε}           FOLLOW(F) = {x, y, $, }}
FIRST(A) = {x, y}
FIRST(T) = {a, b}
FIRST(V) = {x, y}
FIRST(C) = {a, b, x, y}
FIRST(D) = {a, b, x, y}
FIRST(O) = {<, >}
```

Although the **`FOLLOW`** sets of **`E`** and **`S`** are not necessary to build the parse table for **`G'`**, they are required for determining the **`FOLLOW`** sets of **`R`** and **`F`**, which *are* necessary, so I have included them here. To wit:

```
FOLLOW(R) = FOLLOW(S) = {$} U FIRST("}") = {$, }}
FOLLOW(F) = FOLLOW(E) = FIRST(R)\{ε} U FOLLOW(S) = {x, y, $, }}
```

We can prove that **`G'`** is LL(1) by examining the rules and showing, first, that the **`FIRST`** sets of the rules of each variable are disjoint and, second, that a variable's **`FIRST`** and **`FOLLOW`** sets are disjoint.

However, it is sufficient that constructing the parse table for a grammar yields entries containing at most one production rule. In this event, given any variable, no ambiguity exists in choosing a rule to produce a certain terminal. This determinism means looking ahead in the string is not required. Since we can resolve each symbol as it comes, and with no extra memory, the grammar is LL(1).

The two approaches are equivalent, but a parse table is produced for **G'** in part 4, so we can save ourselves some labour if we don't examine its production rules directly. Since the resulting table defines a function, it follows that **G'** is LL(1).

4) **Constructing the parse table for G'**
First we must produce an empty table **T** with a column for each terminal **a** in the grammar (plus the end-of-input marker), and a row for each variable **V**. This table is then populated by making sure it satisfies two rules for that grammar:

1. **if (∃ (V → α): ε ≠ a ∈ FIRST(α)) then α ∈ T[V,a]**
2. **if (∃ (V → α): ε ∈ FIRST(α), a ∈ FOLLOW(V)) then α ∈ T[V,a]**

The table satisfying these rules for the grammar **G'** follows, noting that I have left out the empty columns for the terminals **f, (,), {, l, s, :, =, ;**. These symbols are in the FIRST or the FOLLOW set of no variable, so they have no associated production rules.

	i	}	e	a	b	x	y	<	>	$
S	ER	-	-	-	-	AR	AR	-	-	-
R	-	ε	-	-	-	AR	AR	-	-	ε
E	if(C){S}F	-	-	-	-	-	-	-	-	-
F	-	ε	else{S}	-	-	ε	ε	-	-	ε
A	-	-	-	-	-	V:=T;	V:=T;	-	-	-
T	-	-	-	a	b	-	-	-	-	-
V	-	-	-	-	-	x	y	-	-	-
C	-	-	-	DOD	DOD	DOD	DOD	-	-	-
D	-	-	-	T	T	V	V	-	-	-
O	-	-	-	-	-	-	-	<	>	-

5) **LL(1) parser implementation**

See the accompanying code for details, but I include a sample invocation which should cover parts i - iii.

```
ajur4521@ucpu1% python CFLparse.py tests/error_insertion.txt
if(x<a){x:=y;;}$ S$
if(x<a){x:=y;;}$ ER$
if(x<a){x:=y;;}$ if(C){S}FR$
f(x<a){x:=y;;}$ f(C){S}FR$
(x<a){x:=y;;}$ (C){S}FR$
x<a){x:=y;;}$ C){S}FR$
x<a){x:=y;;}$ DOD){S}FR$
x<a){x:=y;;}$ VOD){S}FR$
x<a){x:=y;;}$ xOD){S}FR$
<a){x:=y;;}$ OD){S}FR$
<a){x:=y;;}$ <D){S}FR$
a){x:=y;;}$ D){S}FR$
a){x:=y;;}$ T){S}FR$
a){x:=y;;}$ a){S}FR$
){x:=y;;}$ ){S}FR$
{x:=y;;}$ {S}FR$
x:=y;;}$ S}FR$
x:=y;;}$ AR}FR$
x:=y;;}$ V:=T;R}FR$
x:=y;;}$ x:=T;R}FR$
:=y;;}$ :=T;R}FR$
=y;;}$ =T;R}FR$
y;;}$ T;R}FR$

ERROR: y obtained; but variable T expects one of ['a', 'b']

REJECTED
```

6) **Modifying the parser's grammar**

The program reads its grammar from a text file, where each line contains a production rule in the usual format. The start variable is taken to be the left hand side of the first production rule.The associated parse table is then constructed and used to check the input string.However, care should be taken that the supplied grammar is LL(1): it will be rejected otherwise. Some grammars can cause the program to loop infinitely or otherwise malfunction, for example those with cyclic production dependencies. Nevertheless, if this condition is satisfied, the parser is fully generic, and modifying the grammar is a simple matter of supplying a different file.

7) **Error recovery**

As a string is being parsed, if an error is encountered, the assumption is made that the substring that has already been successfully parsed is correct, and so the program tries to modify the remaining unconsumed symbols in order to produce a correct string. Determining how they should be modified might seem to depend upon the nature of the error:

i. the stack and remaining string both have valid terminals at the front, but they do not match

ii. the stack has a variable at the front, the remaining string has a valid terminal at the front, but no production rule exists for obtaining that terminal given this variable

iii. the remaining string has a symbol at the front which is not a terminal of our grammar

On a string of symbols there are only two basic operations: insertion and deletion. Any other operation, such as a point mutation of a single character, can be performed by the correct sequence of insertions and deletions. That is, if we arrive at an incorrect symbol, either it was itself inserted incorrectly, or something correct which should have preceded it was incorrectly removed.

We might then expect that we could simply try all possible sequences of removals and insertions of valid symbols until we arrive at a correct string. Indeed, this is basically what the program does, but we can save a little work by using the foregoing error categories.

First, we can observe that if the incorrect symbol is not a terminal of our grammar, then it must have gotten there by an insertion, so we can immediately remove it without checking strings that insert symbols before it.

Second, if the stack has a variable at the front, this circumscribes the set of possible terminals we can correctly prepend to the remaining string to be exactly those terminals whose entries in the relevant row of the parse table are defined.

Hence, the following algorithm was devised to find the nearest correct substring to that which was rejected, given that the part of the string which was parsed successfully is retained. In broad strokes, it checks if a string is valid; if not, at the first invalid character, all strings produced by reasonable insertions or deletions at that character are added to a queue, then we grab the next item off the queue and repeat the process.

If **parse()** is the usual LL(1) table-driven parsing algorithm, then let **reject_string** be the final remaining string of an invocation of **parse(s)**, where **s** was rejected, and take **table** to be the parse table used inside **parse().**

```
Add reject_string to a queue Q
Repeat:
   string := Q.dequeue()

   if parse(string) accepts:
      return string and halt
   else:
      s := the top of the final stack from parse(string)
      r := the first invalid symbol of string

      // reverse potential insertion
      d := string with r removed
      if we haven't tried parse(d):
         Q.enqueue(d)

      // reverse potential deletions
      if r is a valid terminal:
         if s is a terminal:
            t := string with stack.top inserted before r
            if we haven't tried parse(t):
               Q.enqueue(t)
         else (s must be a variable):
            for each terminal p where table[s, p] is defined:
               v := string with p inserted before r
               if we haven't tried parse(v):
                  Q.enqueue(v)
```

Because candidate strings are modified once, then added to queue, the number of modifications of **string** monotonically increases as the algorithm iterates. Thus, the first valid string produced is guaranteed to have been arrived at by the minimal number of modifications. Further, we can be assured that it will halt because deletions are performed before insertions, and because once the stack is exhausted (by repeated calls to **parse()**), if any symbols remain in our string, they will eventually all be deleted in some item added to the queue, thus yielding a valid string. But beware, while it may halt eventually, its worst case complexity is exponential, depending heavily on input string and grammar

Thus, for the sake of an example, if the incorrect string (on **G'**)
"if(x<a){x:=y;;}" is offered, this method will produce the substitute
"if(x<a){x:=a;}".
Slightly less unimpressive, given a garbled version of the string from part 1.b),
"**if (x >) { if(y) {x := =; y := b;} ese ay:{x := b;}}**",
by luck, it correctly recovers the original, albeit rather slowly.
If more than one corrected string is desired, it is a trivial matter to save valid strings and halt after there are no more possible, or some threshold number have been generated.
I have not, however, examined the question of whether other substitutes might be available if we were to modify parts of the string that come before, but not immediately before, the first invalid symbol. It seems *plausible* that such modifications could produce better overall results, although I provide no proof one way or the other.
Nor was serious consideration given to any other approaches at all.

8) **Testing files and a test summary**
Grammars and parse table construction were tested by supplying the program with various grammars, both LL(1) and not, of which are included in the submission both **G** and **G'**, along with a small grammar, bc.grm, accepting **a|b*c***, and another which produces arithmetic expressions. All grammars are treated as expected. The ones which are LL(1) can be run, the ones which are not are rejected.

Parsing was tested for each of these grammars with various inputs, some examples of which are included in the submission. **G'** itself was tested most strenuously. Some sample input files for it are located in the root of the tests/ folder. Several files exist containing valid strings. In particular, the example data, the expression from part 1.b), and a relatively long string were included.
Many more files are devoted to testing every circumstance in which a string should be rejected.

These results are summarised briefly in the table on the next page. For reasons of space, I will not include the expected or actual traces, only the result itself. These were examined as well, of course; but they, along with various other tests and files and sample grammars are omitted for the same reason. Let what follows be a representative sample.

The lion's share of the work on this assignment was, of course, not in the parser itself, but in the error-recovery feature. So each of these, even the ones ostensibly for testing out other grammars, do double-duty by also putting that feature through its paces. Some, in particular error_nonsense.txt and error_garbled.txt were designed explicitly for determining how well major errors could be recovered. By the same token, a file such as accept.txt which is accepted by **G'**, is rejected instantly by bc.grm, but a valid surrogate for that grammar can nonetheless be constructed to replace it - although

error_empty.txt is actually accepted by bc.grm.
Indeed, the error correction was successful in all cases, except for error_long.txt, in which instance I got tired of waiting for it to complete. Where the languages have some overlap in the terminal set, the closest strings can be very mildly amusing. For example, accept.txt contains the following: "`if(x<a){if(y>b){x:=a;y:=b;}}`" And apparently its closest string in arith.grm is then, "`(x)+(y)=y=1`", which I suppose implies that x = 0.

input	**expected result**	**actual result**	**description**
accept.txt	accepted	accepted	provided test
example.txt	accepted	accepted	expression from 1.b)
long.txt	accepted	accepted	a relatively long one, for utilising all production rules
error_missing_semicolon.txt	rejected	rejected	missing symbol
error_empty.txt	rejected	rejected	empty file
error_garbled.txt	rejected	rejected	example.txt, but muddled
error_insertion.txt	rejected	rejected	an accepted string with one terminal added
error_invalid_terminal.txt	rejected	rejected	input with terminals not in the grammar
error_long.txt	rejected	rejected	long.txt, but with mistakes
error_nonsense.txt	rejected	rejected	input with no similarity at all to valid strings
error_wrong_symbol.txt	rejected	rejected	a valid string, but with cunning substitutions, rather than just insertions or deletions
arith/truth.txt	rejected by G', accepted by arith.grm	rejected by G' accepted by arith.grm	a basic arithmetic expression valid as such, but not valid under G'
arith/falsehood.txt	rejected by G' rejected by arith.grm	rejected by G' rejected by arith.grm	a basic arithmetic expression, but malformed
arith/complicated.txt	rejected by G' rejected by arith.grm	rejected by G' rejected by arith.grm	a slightly longer expression
bc/accept.txt	rejected by G' accepted by bc.grm	rejected by G' accepted by bc.grm	very basic first test of another grammar
bc.reject.txt	rejected by G' rejected by bc.grm	rejected by G' rejected by bc.grm	contains the wrong pattern, and some symbols not in the grammar