Introduction to language theory and compilation

First assignment: TYPY

Pham Quang Vinh & Balon-Perin Alexandre

Academic Year 2010-2011

Table of Contents

Introduction

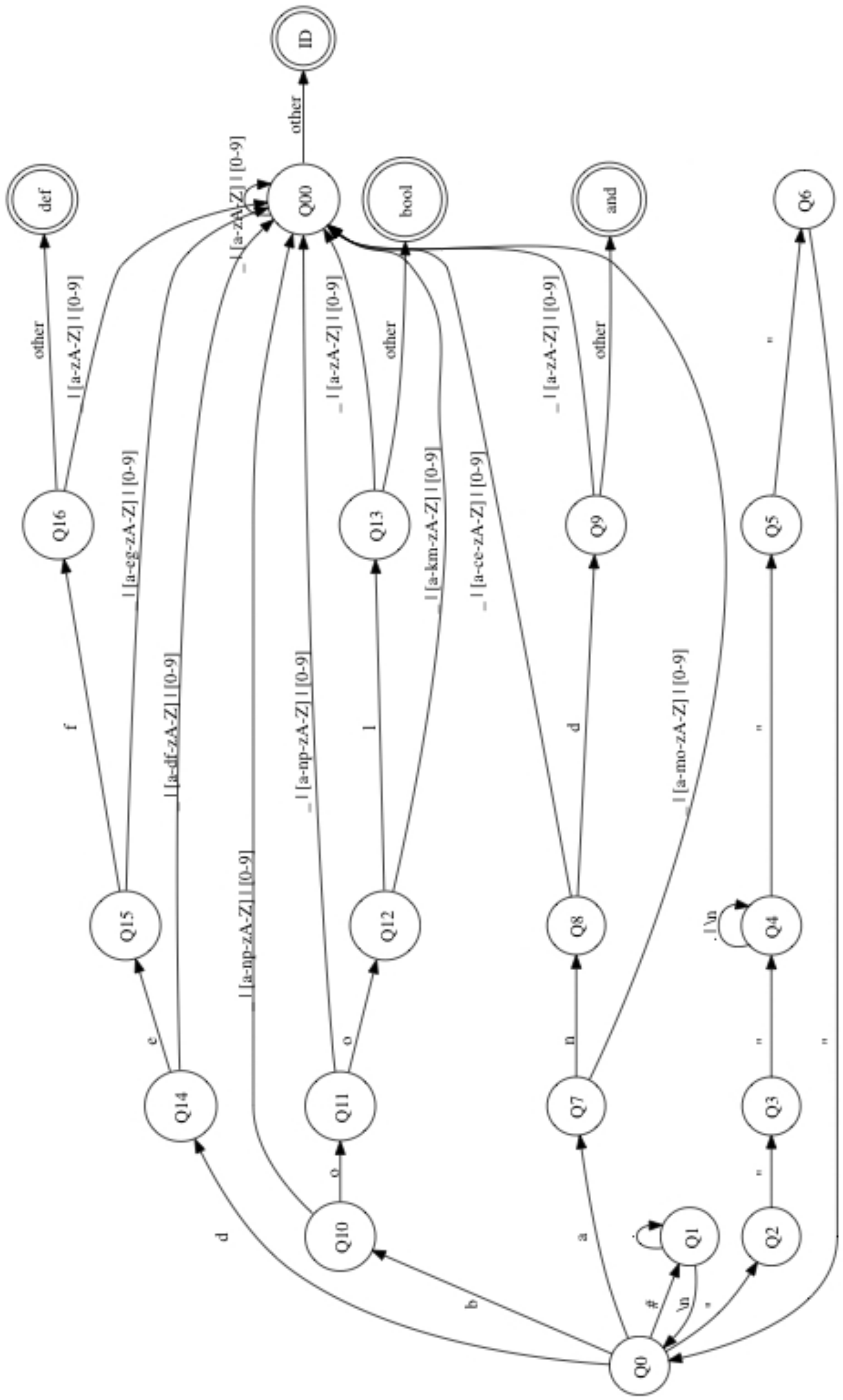
As part of the course “Introduction to language theory and Compilation”, we were asked to build an LL(1) compiler for the language TYPY. The goal is to control a 64x64 color screen that is only compatible with the low-level language of the GPMachine. As a result, the code generated by the compiler must be understandable by a GPMachine.

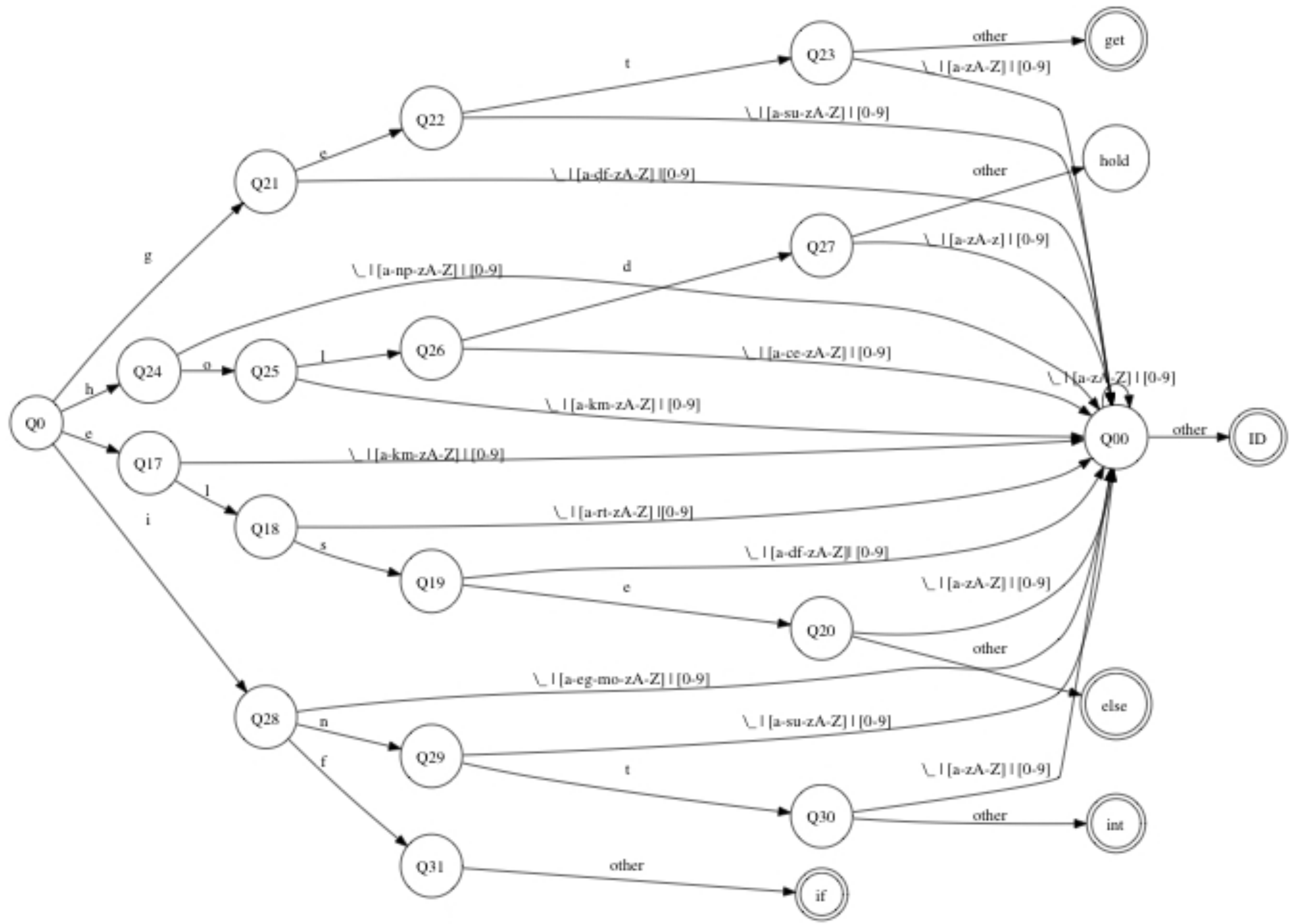
Identify the lexical units (or tokens) of TYPY and their corresponding regular expressions

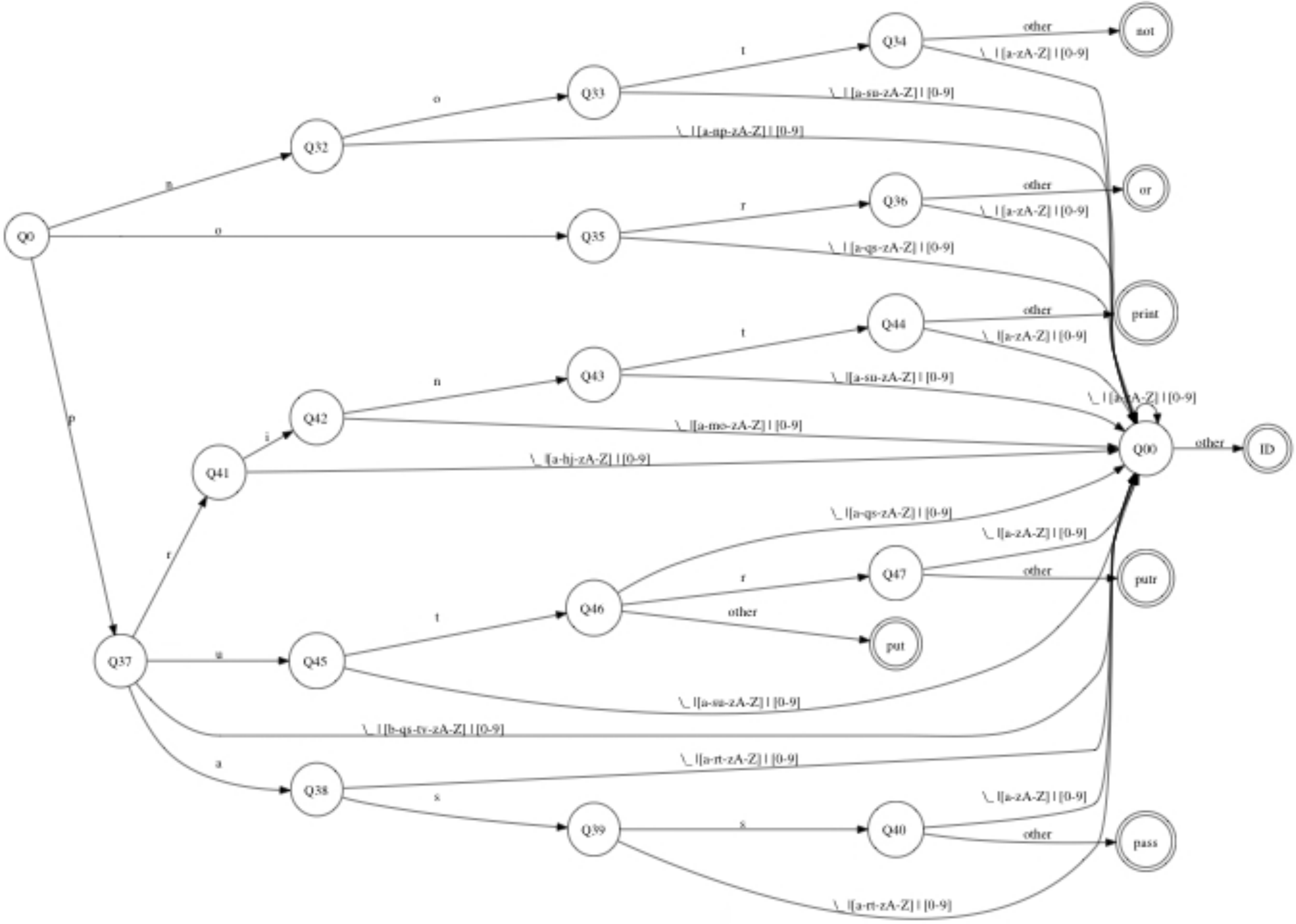
Token	Regular expression
ID	(_ | [a-zA-Z]) (_ | [a-zA-Z] | [0-9])*
INTEGER_LITERAL	[0-9]+
BOOLEAN_LITERAL	"False" | "True"
INDENT	Recognised by algorithm
DEDENT	Recognised by algorithm
and	"and"
bool	“bool”
def	^"def"
else	“else”
get	“get”
hold	“hold”
int	“int”
if	“if”
not	“not”
or	“or”
pass	“pass”
print	“print”
put	“put”
putr	“putr”
read	“read”
release	“release”
reset	“reset”
return	“return”
while	“while”
†	\n
,	“,”
(	“(“
)	“)”
+	“+”
- (binary)	“-“
*****	“*”
/	“/”
==	“==”
=	“=”
<	“<”
<=	“<=”
>	“>”
>=	“>=”
!=	“!=”
:	“:”
– (unary)	“–“

Give a deterministic finite automaton that recognises them correctly

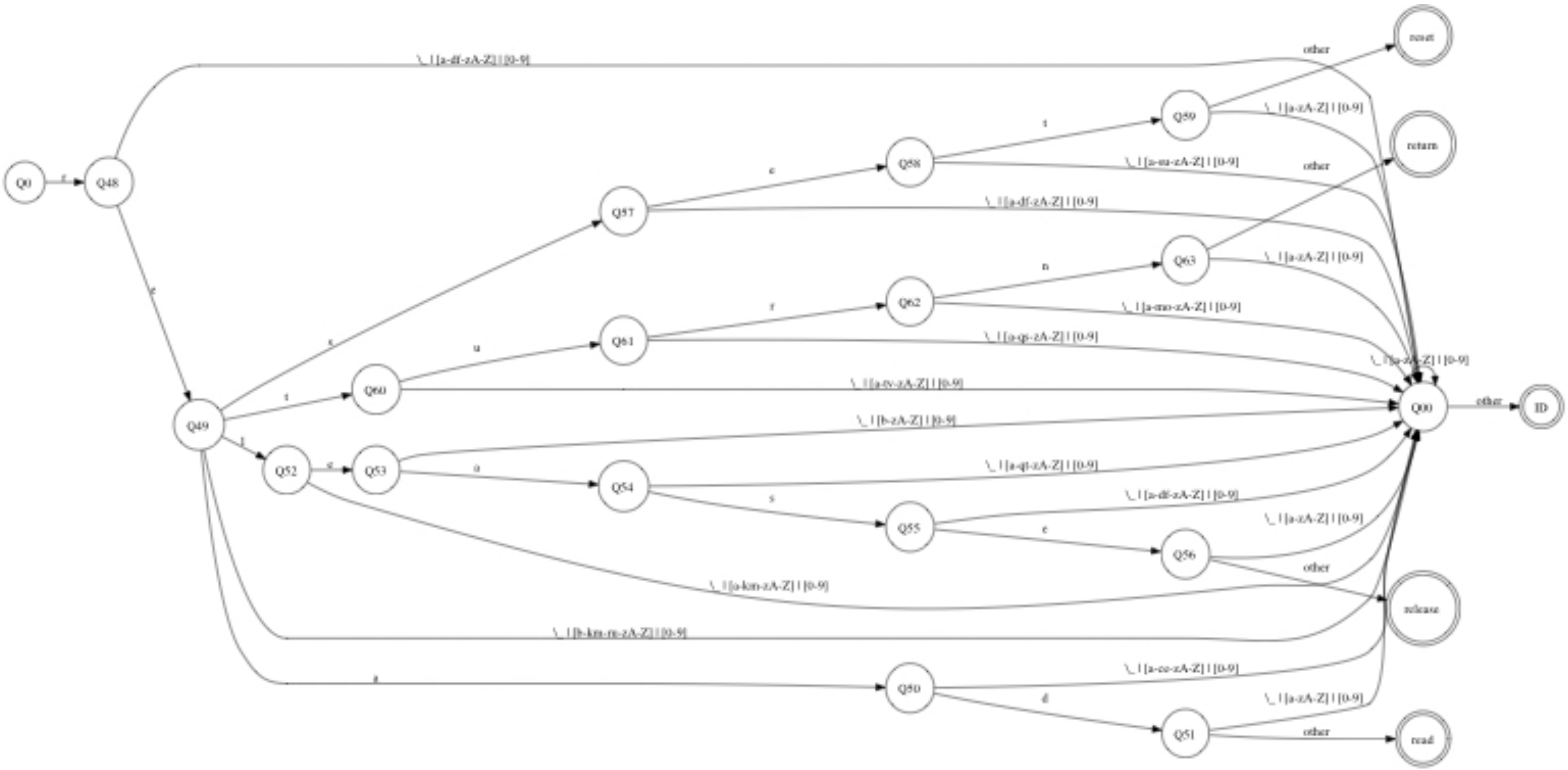
In order to draw understandable graphs we decided to duplicate the initial state and the state Q00 on each graph. The representation of the DFA should take place on one and only graph.



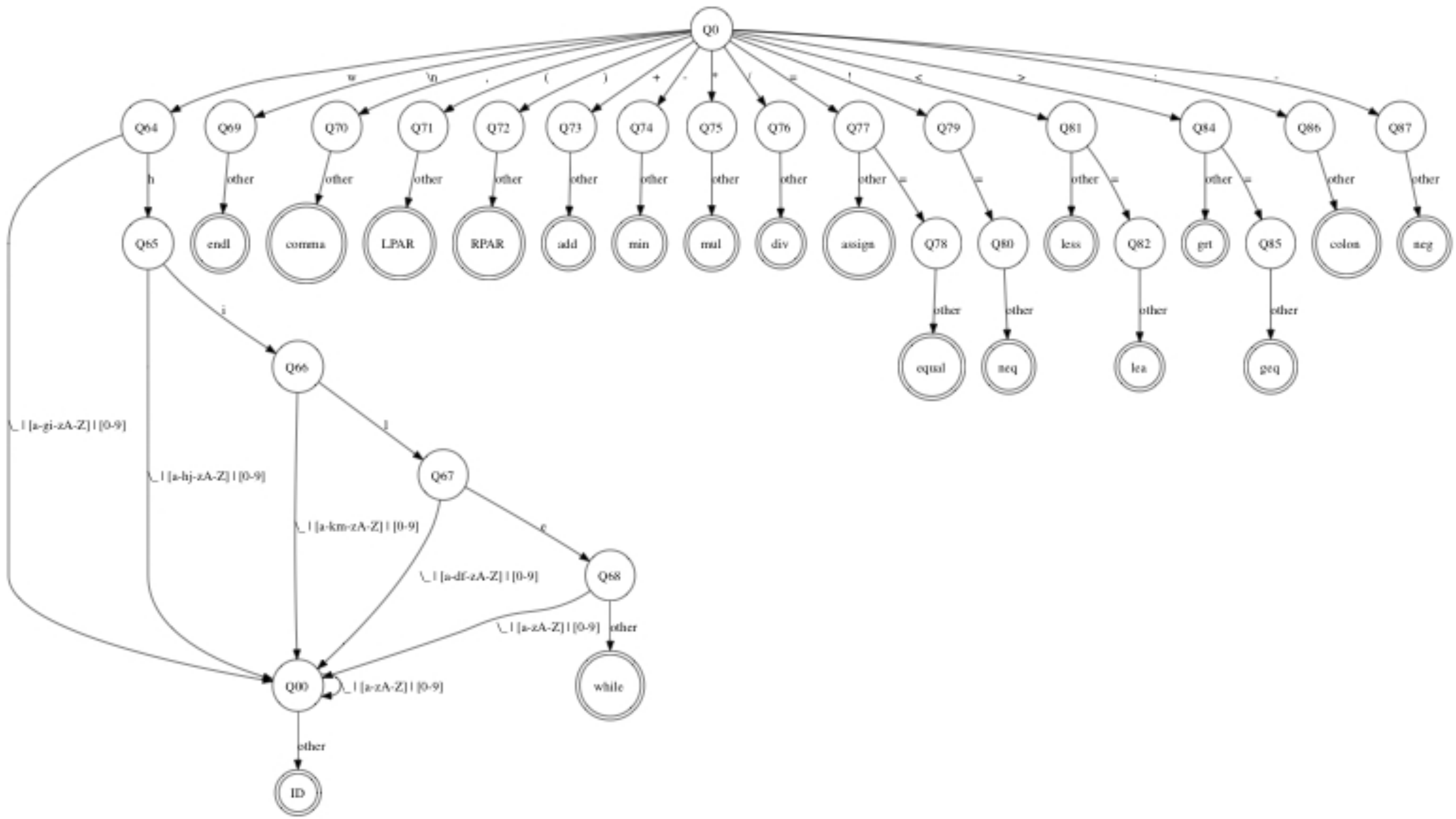

Q0
h
e
g
i
Q24
Q17
Q21
Q28
o
l
e
n
f
Q25
Q18
Q22
Q29
Q31
l
s
t
Q26
Q19
Q23
Q30
d
e
t
Q27
Q20
other
if
Q00
other
ID
get
hold
else
int
other
other
other
other
_ | [a-zA-Z] | [0-9]
_ | [a-su-zA-Z] | [0-9]
_ | [a-df-zA-Z] |[0-9]
_ | [a-zA-z] | [0-9]
_ | [a-np-zA-Z] | [0-9]
_ | [a-ce-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]
_ | [a-km-zA-Z] | [0-9]
_ | [a-km-zA-Z] | [0-9]
_ | [a-rt-zA-Z] |[0-9]
_ | [a-df-zA-Z]| [0-9]
_ | [a-zA-Z] | [0-9]
_ | [a-eg-mo-zA-Z] | [0-9]
_ | [a-su-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]



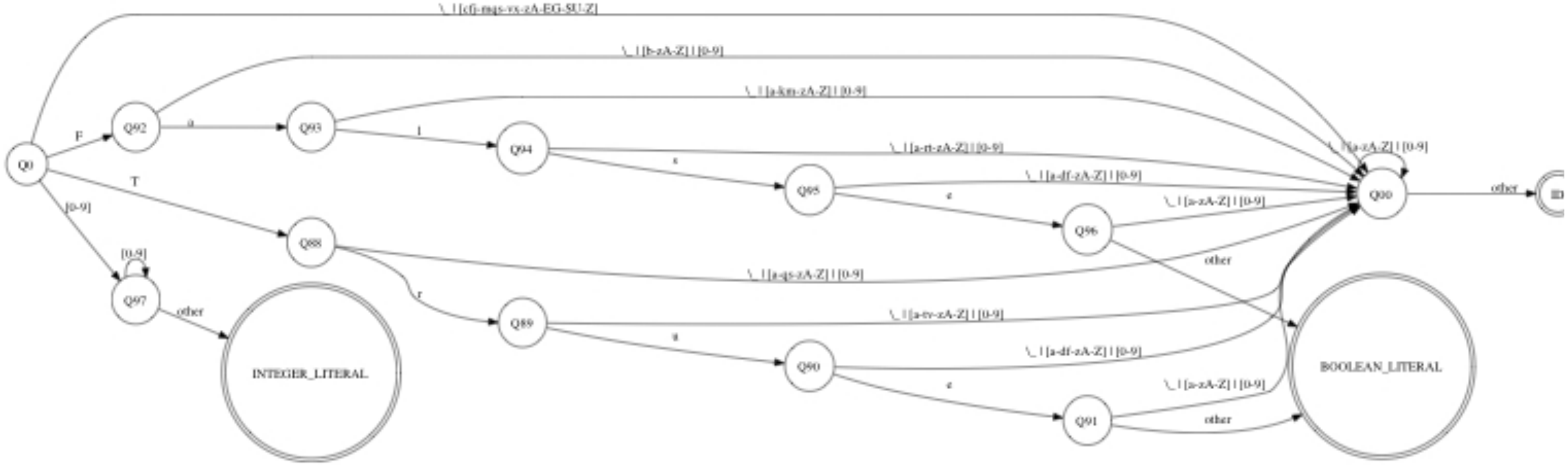
Q0
n
o
p
Q32
Q33
Q34
Q35
Q36
Q37
Q38
Q39
Q40
Q41
Q42
Q43
Q44
Q45
Q46
Q47
Q00
ID
not
or
print
put
putr
pass
other
t
r
i
u
a
s
_ | [a-zA-Z] | [0-9]
_ | [a-su-zA-Z] | [0-9]
_ | [a-np-zA-Z] | [0-9]
_ | [a-qs-zA-Z] | [0-9]
_ |[a-mo-zA-Z] | [0-9]
_ |[a-hj-zA-Z] | [0-9]
_ |[a-qs-zA-Z] | [0-9]
_ |[a-su-zA-Z] | [0-9]
_ | [b-qs-tv-zA-Z] | [0-9]
_ |[a-rt-zA-Z] | [0-9]



Q0
r
Q48
_ | [a-df-zA-Z] | [0-9]
e
Q49
s
Q57
e
Q58
t
Q59
other
reset
_ | [a-zA-Z] | [0-9]
_ | [a-su-zA-Z] | [0-9]
_ | [a-df-zA-Z] | [0-9]
t
Q60
u
Q61
r
Q62
n
Q63
other
return
_ | [a-zA-Z] | [0-9]
_ | [a-mo-zA-Z] | [0-9]
_ | [a-qs-zA-Z] | [0-9]
_ | [a-tv-zA-Z] | [0-9]
l
Q52
e
Q53
a
Q54
s
Q55
e
Q56
other
release
_ | [b-zA-Z] | [0-9]
_ | [a-qt-zA-Z] | [0-9]
_ | [a-df-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]
_ | [a-km-zA-Z] | [0-9]
_ | [b-km-rt-zA-Z] | [0-9]
a
Q50
d
Q51
other
read
_ | [a-ce-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]
Q00
_ | [a-zA-Z] | [0-9]
other
ID



Q0
w
\n
,
(
)
+
-
*
/
=
!
<
>
:
-
Q64
Q69
Q70
Q71
Q72
Q73
Q74
Q75
Q76
Q77
Q79
Q81
Q84
Q86
Q87
h
other
=
Q65
endl
comma
LPAR
RPAR
add
min
mul
div
assign
Q78
Q80
less
Q82
grt
Q85
colon
neg
equal
neq
lea
geq
i
Q66
l
Q67
e
Q68
_ | [a-gi-zA-Z] | [0-9]
_ | [a-hj-zA-Z] | [0-9]
_ | [a-km-zA-Z] | [0-9]
_ | [a-df-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]
Q00
while
ID



_ | [cfj-mqs-vx-zA-EG-SU-Z]
_ | [b-zA-Z] | [0-9]
_ | [a-km-zA-Z] | [0-9]
_ | [a-rt-zA-Z] | [0-9]
_ | [a-df-zA-Z] | [0-9]
_ | [a-zA-Z] | [0-9]
_ | [a-qs-zA-Z] | [0-9]
_ | [a-tv-zA-Z] | [0-9]
Q0
Q92
Q93
Q94
Q95
Q96
Q88
Q89
Q90
Q91
Q97
Q00
F
T
[0-9]
a
l
s
e
r
u
other
INTEGER_LITERAL
BOOLEAN_LITERAL

Program a scanner that recognises the tokens of TYPY.

We implemented a scanner in the class DFA.java by reading each character on input and follow the paths described in the DFA with a switch/case. Moreover, the scanner implements the algorithm for checking indentation and can response the next token and the corresponding value when called.
The symbol table will be implemented further in the parser phase.

Modify the grammar:

1. **In order for it to become LL(1);**
2. **And to account for operator precedence and associativity, which are given in Table 1.**

Remove unproductive symbols

Let's apply the following algorithm:

Grammar RemoveUnproductive(**Grammar** $G = \langle V, T, P, S \rangle$) **begin**
- $V_0 \leftarrow \emptyset$;
- $i \leftarrow 0$;
- **repeat**
  - $i \leftarrow i+1$;
  - $V_i \leftarrow \{A \mid A \rightarrow \alpha \in P \wedge \alpha \in (V_{i-1} \cup T)^*\} \cup V_{i-1}$;
- **until** $V_i = V_{i-1}$;
- $V' \leftarrow V_i$;
- $P' \leftarrow$ set of rules of P that do not contain variables in $V \setminus V'$;
- return($G' = \langle V', T, P', S \rangle$) ;

i	V_i
0	$\varnothing$
1	{V_0 , <FUNCTIONLIST>, <FTYPE>, <ARGS>, <TYPE>, <VARDECL>, <IDLIST>, <SIMPLESTMT>, <EXPRESSION>, <OP>, <SCREENCOMMAND>, <CALLARGS>}
2	{V_1 , <STATEMENT>, <ARGLIST>, <VARLINE>, <CALL>, <EXPRLIST>}
3	{V_2 , <STATEMENTLIST>}
4	{V_3 , <MAIN>, <BODY>}
5	{V_4 , <PROGRAM>, <FUNCTION>, <IF>, <WHILE>}
6	{V_5 , <COMPOUNDSTMT>}
7	{V_6}

As we could expect there are no unproductive symbol.

Remove inaccessible symbol

Let's apply the following algorithm:

Grammar `RemoveInaccessible`(**Grammar** $G = \langle V, T, P, S \rangle$) **begin**

- $V_0 \leftarrow \{S\}$; $i \leftarrow 0$;
- **repeat**
  - $i \leftarrow i+1$;
  - $V_i \leftarrow \{X \mid \exists A \to \alpha X \beta \text{ in } P \wedge A \in V_{i-1}\} \cup V_{i-1}$;
- **until** $V_i = V_{i-1}$;
- $V' \leftarrow V_i \cap V$; $T' \leftarrow V_i \cap T$;
- $P' \leftarrow$ set of rules of P that only contain variables from V_i ;
- `return`($G' = \langle V', T', P', S \rangle$) ;

Again all symbols are accessible.

Ambiguity removal

Operator precedence (top to bottom) and associativity

Operators	Associativity
()	
`not`, unary − (rule 35)	right
*, /	left
+, binary − (rule 42)	left
>, <, >=, <=, !=, ==	left
`and`	left
`or`	left

The result of the transformations on the grammar for precedence and associativity are shown in the final grammar. (see below)

Left recursion removal & Left factoring

The two following algorithms were eventually applied on the given grammar to produce the final LL(1) grammar .

By applying those, we can remove the variable <CALL> because the terminal ID is the prefix of two productions of the same variable(<SIMPLESTMT> and <EXPRESSION>).

`RemoveLeftRecursion`(**Grammar** $G = \langle V, T, P, S \rangle$) **begin**

  while *G contains a left recursive variable A* **do**

    Let $E = \{A \to A\alpha, A \to \beta, \ldots, A \to \zeta\}$ be the set of rules that have A as left-hand side ;

    Let $\mathcal{U}$ and $\mathcal{V}$ be two new variables ;

    $V = V \cup \{\mathcal{U}, \mathcal{V}\}$;

    $P = P \setminus E$;

    $P = P \cup \{A \to \mathcal{U}\mathcal{V}, \mathcal{U} \to \beta, \ldots, \mathcal{U} \to \zeta, \mathcal{V} \to \alpha\mathcal{V}, \mathcal{V} \to \varepsilon\}$;

`LeftFactor`(**Grammar** $G = \langle V, T, P, S \rangle$) **begin**

  while *G has at least two rules with the same left-hand side and a common prefix* **do**

    Let $E = \{A \to \alpha\beta, \ldots, A \to \alpha\zeta\}$ be such a set of rules ;

    Let $\mathcal{V}$ be a new variable;

    $V = V \cup \mathcal{V}$;

    $P = P \setminus E$;

    $P = P \cup \{A \to \alpha\mathcal{V}, \mathcal{V} \to \beta, \ldots, \mathcal{V} \to \zeta\}$;

Final Grammar

[1] <PROGRAM>	→ <VARDECL><FUNCTIONLIST><MAIN>$
[2] <FUNCTIONLIST>	→ <FUNCTION> † <FUNCTIONLIST>
[3]	→ ε
[4] <FUNCTION>	→ def <FTYPE> ID (<ARGS>): <BODY>
[5] <FTYPE>	→ <TYPE>
[6]	→ ε
[7] <ARGS>	→ <ARGLIST>
[8]	→ ε
[9] <ARGLIST>	→ <TYPE> ID <NEXTARG>
[10] <NEXTARG>	→ ,<ARGLIST>
[11]	→ ε
[12] <TYPE>	→ int
[13]	→ bool
[14] <BODY>	→ † INDENT <VARDECL> <STATEMENTLIST> DEDENT
[15] <VARDECL>	→ <VARLINE> <VARDECL>
[16]	→ ε
[17] <VARLINE>	→ <TYPE> <IDLIST> †
[18] <IDLIST>	→ ID <NEXTID>
[19] <NEXTID>	→ , <IDLIST>
[20]	→ ε
[21] <MAIN>	→ <VARDECL> <STATEMENTLIST>
[22] <STATEMENTLIST>	→ <STATEMENT> <NEXTSTATEMENT>
[23] <NEXTSTATEMENT>	→ <STATEMENTLIST>
[24]	→ ε
[25] <STATEMENT>	→ <SIMPLESTMT> †
[26]	→ <COMPOUNDSTMT>
[27] <SIMPLESTMT>	→ ID <SIMPLESTMT-TAIL>
[28]	→ read(ID)
[29]	→ print(<EXPRESSION>)
[30]	→ return <EXPRESSION>
[31]	→ <SCREENCOMMAND>
[32] <SIMPLESTMT-TAIL>	→ = <EXPRESSION>
[33]	→ (<CALLARGS>)
[34] <COMPOUNDSTMT>	→ <IF>
[35]	→ <WHILE>
[36] <IF>	→ if <EXPRESSION>: <BODY> <IF-TAIL>
[37] <IF-TAIL>	→ else: <BODY>
[38]	→ ε
[39] <WHILE>	→ while <EXPRESSION>: <BODY>
[40] <SCREENCOMMAND>	→ put(<EXPRESSION>, <EXPRESSION>, <EXPRESSION>)
[41]	→ putr(<EXPRESSION>, <EXPRESSION>)
[42]	→ get(<EXPRESSION>, <EXPRESSION>)
[43]	→ hold
[44]	→ release
[45]	→ reset
[46] <CALLARGS>	→ <EXPRLIST>
[47]	→ ε
[48] <EXPRLIST>	→ <EXPRESSION> <NEXTEXPR>

[49] <NEXTEXPR>	→ , <EXPRLIST>
[50]	→ ε
[51] <EXPRESSION>	→ <V><T>
[52] <T>	→ or <EXP1><T>
[53]	→ ε
[54]<EXP1>	→<EXP2><T1>
[55]<T1>	→ and <EXP2><T1>
[56]	→ ε
[57]<EXP2>	→ <EXP3><T2>
[58]<T2>	→ > <T2-TAIL>
[59]	→ < <T2-TAIL>
[60]	→ == <EXP3><T2>
[61]	→ != <EXP3><T2>
[62]	→ ε
[63]<T2-TAIL>	→<EXP3><T2>
[64]	→ = <EXP3><T2>
[65]<EXP3>	→ <EXP4><T3>
[66]<T3>	→ + <EXP4><T3>
[67]	→ - <EXP4><T3>
[68]	→ ε
[69]<EXP4>	→ <EXP5><T4>
[70]<T4>	→ *<EXP5><T4>
[71]	→ / <EXP5><T4>
[72]	→ ε
[73]<EXP5>	→ - <EXP5>
[74]	→ not <EXP5>
[75]	→ <EXP6>
[76]<EXP6>	→ (<EXPRESSION>)
[77]	→ ID<EXP6-TAIL>
[78]	→ INTEGER_LITERAL
[79]	→ BOOLEAN_LITERAL
[80]	→ ε
[81]<EXP6-TAIL>	→ (<CALLARGS>)
[82]	→ ε

Give the LL(1) parsing table for your new grammar as well as the needed First() and Follow() calculations to derive it.

	int	bool	def	(	)	:	,	INDENT	DEDENT	†	ID	print	read	return
PROGRAM	P1	P1	P1	x	x	x	x	x	x	x	P1	P1	P1	P1
FUNCTIONLIST	P3	P3	P2	x	x	x	x	x	x	x	P3	P3	P3	P3
FUNCTION	x	x	P4	x	x	x	x	x	x	x	x	x	x	x
FTYPE	P5	P5	x	x	x	x	x	x	x	x	P6	x	x	x
ARGS	P7	P7	x	x	P8	x	x	x	x	x	x	x	x	x
ARGLIST	P9	P9	x	x	x	x	x	x	x	x	x	x	x	x
NEXTARG	x	x	x	x	P11	x	P10	x	x	x	x	x	x	x
TYPE	P12	P13	x	x	x	x	x	x	x	x	x	x	x	x
BODY	x	x	x	x	x	x	x	x	x	P14	x	x	x	x
VARDECL	P15	P15	P16	x	x	x	x	x	x	x	P16	P16	P16	P16
VARLINE	P17	P17	x	x	x	x	x	x	x	x	x	x	x	x
IDLIST	x	x	x	x	x	x	x	x	x	x	P18	x	x	x
NEXTID	x	x	x	x	x	x	P19	x	x	P20	x	x	x	x
MAIN	P21	P21	P21	x	x	x	x	x	x	x	P21	P21	P21	P21
STATEMENTLIST	x	x	x	x	x	x	x	x	x	x	P22	P22	P22	P22
NEXTSTATEMENT	x	x	x	x	x	x	x	x	P24	x	P23	P23	P23	P23
STATEMENT	x	x	x	x	x	x	x	x	x	x	P25	P25	P25	P25
SIMPLESTMT	x	x	x	x	x	x	x	x	x	x	P27	P28	P29	P30
SIMPLESTMT-TAIL	x	x	x	P33	x	x	x	x	x	x	x	x	x	x
COMPOUNDSTMT	x	x	x	x	x	x	x	x	x	x	x	x	x	x
IF	x	x	x	x	x	x	x	x	x	x	x	x	x	x
IF-TAIL	x	x	x	x	x	x	x	x	P38	x	P38	P38	P38	P38
WHILE	x	x	x	x	x	x	x	x	x	x	x	x	x	x
SCREENCOMMAND	x	x	x	x	x	x	x	x	x	x	x	x	x	x
CALLARGS	x	x	x	P46	P47	x	x	x	x	x	P46	x	x	x
EXPRLIST	x	x	x	P48	x	x	x	x	x	x	P48	x	x	x
NEXTEXPR	x	x	x	x	P50	x	P49	x	x	x	x	x	x	x
EXPRESSION	x	x	x	P51	x	x	x	x	x	x	P51	x	x	x
T	x	x	x	x	P53	P53	P53	x	x	P53	x	x	x	x
EXP1	x	x	x	P54	x	x	x	x	x	x	P54	x	x	x
T1	x	x	x	x	P56	P56	P56	x	x	P56	x	x	x	x
EXP2	x	x	x	P57	x	x	x	x	x	x	P57	x	x	x
T2	x	x	x	x	P64	P64	P64	x	x	P64	x	x	x	x
EXP3	x	x	x	P65	x	x	x	x	x	x	P65	x	x	x
T3	x	x	x	x	P68	P68	P68	x	x	P68	x	x	x	x
EXP4	x	x	x	P69	x	x	x	x	x	x	P69	x	x	x
T4	x	x	x	x	P72	P72	P72	x	x	P72	x	x	x	x
EXP5	x	x	x	P75	P75	P75	P75	x	x	P75	P75	x	x	x
EXP6	x	x	x	P76	P80	P80	P80	x	x	P80	P77	x	x	x
EXP6-TAIL	x	x	x	P81	P82	P82	P82	x	x	P82	x	x	x	x

	if	else	while	put	putr	get	hold	release	reset	or	and	>	<	<=
PROGRAM	P1	x	P1	P1	P1	P1	P1	P1	P1	x	x	x	x	x
FUNCTIONLIST	P3	x	P3	P3	P3	P3	P3	P3	P3	x	x	x	x	x
FUNCTION	x	x	x	x	x	x	x	x	x	x	x	x	x	x
FTYPE	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ARGS	x	x	x	x	x	x	x	x	x	x	x	x	x	x
ARGLIST	x	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTARG	x	x	x	x	x	x	x	x	x	x	x	x	x	x
TYPE	x	x	x	x	x	x	x	x	x	x	x	x	x	x
BODY	x	x	x	x	x	x	x	x	x	x	x	x	x	x
VARDECL	P16	x	P16	P16	P16	P16	P16	P16	P16	x	x	x	x	x
VARLINE	x	x	x	x	x	x	x	x	x	x	x	x	x	x
IDLIST	x	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTID	x	x	x	x	x	x	x	x	x	x	x	x	x	x
MAIN	P21	x	P21	P21	P21	P21	P21	P21	P21	x	x	x	x	x
STATEMENTLIST	P22	x	P22	P22	P22	P22	P22	P22	P22	x	x	x	x	x
NEXTSTATEMENT	P23	x	P23	P23	P23	P23	P23	P23	P23	x	x	x	x	x
STATEMENT	P26	x	P26	P25	P25	P25	P25	P25	P25	x	x	x	x	x
SIMPLESTMT	x	x	x	P31	P31	P31	P31	P31	P31	x	x	x	x	x
SIMPLESTMT-TAIL	x	x	x	x	x	x	x	x	x	x	x	x	x	x
COMPOUNDSTMT	P34	x	P35	x	x	x	x	x	x	x	x	x	x	x
IF	P36	x	x	x	x	x	x	x	x	x	x	x	x	x
IF-TAIL	P38	P37	P38	P38	P38	P38	P38	P38	P38	x	x	x	x	x
WHILE	x	x	P39	x	x	x	x	x	x	x	x	x	x	x
SCREENCOMMAND	x	x	x	P40	P41	P42	P43	P44	P45	x	x	x	x	x
CALLARGS	x	x	x	x	x	x	x	x	x	x	x	x	x	x
EXPRLIST	x	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTEXPR	x	x	x	x	x	x	x	x	x	x	x	x	x	x
EXPRESSION	x	x	x	x	x	x	x	x	x	x	x	x	x	x
T	x	x	x	x	x	x	x	x	x	P52	x	x	x	x
EXP1	x	x	x	x	x	x	x	x	x	x	x	x	x	x
T1	x	x	x	x	x	x	x	x	x	P56	P55	x	x	x
EXP2	x	x	x	x	x	x	x	x	x	x	x	x	x	x
T2	x	x	x	x	x	x	x	x	x	P64	P64	P58	P59	P60
EXP3	x	x	x	x	x	x	x	x	x	x	x	x	x	x
T3	x	x	x	x	x	x	x	x	x	P68	P68	P68	P68	P68
EXP4	x	x	x	x	x	x	x	x	x	x	x	x	x	x
T4	x	x	x	x	x	x	x	x	x	P72	P72	P72	P72	P72
EXP5	x	x	x	x	x	x	x	x	x	P75	P75	P75	P75	P75
EXP6	x	x	x	x	x	x	x	x	x	P80	P80	P80	P80	P80
EXP6-TAIL	x	x	x	x	x	x	x	x	x	P82	P82	P82	P82	P82

	>=	==	!=	=	*	/	+	-(binary)	not	–(unary)	INTEGER_LITERAL	BOOLEAN_LITERAL	$
PROGRAM	x	x	x	x	x	x	x	x	x	x	x	x	x
FUNCTIONLIST	x	x	x	x	x	x	x	x	x	x	x	x	x
FUNCTION	x	x	x	x	x	x	x	x	x	x	x	x	x
FTYPE	x	x	x	x	x	x	x	x	x	x	x	x	x
ARGS	x	x	x	x	x	x	x	x	x	x	x	x	x
ARGLIST	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTARG	x	x	x	x	x	x	x	x	x	x	x	x	x
TYPE	x	x	x	x	x	x	x	x	x	x	x	x	x
BODY	x	x	x	x	x	x	x	x	x	x	x	x	x
VARDECL	x	x	x	x	x	x	x	x	x	x	x	x	x
VARLINE	x	x	x	x	x	x	x	x	x	x	x	x	x
IDLIST	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTID	x	x	x	x	x	x	x	x	x	x	x	x	x
MAIN	x	x	x	x	x	x	x	x	x	x	x	x	x
STATEMENTLIST	x	x	x	x	x	x	x	x	x	x	x	x	x
NEXTSTATEMENT	x	x	x	x	x	x	x	x	x	x	x	x	P24
STATEMENT	x	x	x	x	x	x	x	x	x	x	x	x	x
SIMPLESTMT	x	x	x	x	x	x	x	x	x	x	x	x	x
SIMPLESTMT-TAIL	x	x	x	P32	x	x	x	x	x	x	x	x	x
COMPOUNDSTMT	x	x	x	x	x	x	x	x	x	x	x	x	x
IF	x	x	x	x	x	x	x	x	x	x	x	x	x
IF-TAIL	x	x	x	x	x	x	x	x	x	x	x	x	P38
WHILE	x	x	x	x	x	x	x	x	x	x	x	x	x
SCREENCOMMAND	x	x	x	x	x	x	x	x	x	x	x	x	x
CALLARGS	x	x	x	x	x	x	x	x	P46	P46	P46	P46	x
EXPRLIST	x	x	x	x	x	x	x	x	P48	P48	P48	P48	x
NEXTEXPR	x	x	x	x	x	x	x	x	x	x	x	x	x
EXPRESSION	x	x	x	x	x	x	x	x	P51	P51	P51	P51	x
T	x	x	x	x	x	x	x	x	x	x	x	x	x
EXP1	x	x	x	x	x	x	x	x	P54	P54	P54	P54	x
T1	x	x	x	x	x	x	x	x	x	x	x	x	x
EXP2	x	x	x	x	x	x	x	x	P57	P57	P57	P57	x
T2	P61	P62	P63	x	x	x	x	x	x	x	x	x	x
EXP3	x	x	x	x	x	x	x	x	P65	P65	P65	P65	x
T3	P68	P68	P68	P68	x	x	P66	P67	x	x	x	x	x
EXP4	x	x	x	x	x	x	x	x	P69	P69	P69	P69	x
T4	P72	P72	P72	P72	P70	P71	P72	P72	x	x	x	x	x
EXP5	P75	P75	P75	P75	P75	P75	P75	P75	P74	P73	P75	P75	x
EXP6	P80	P80	P80	P80	P80	P80	P80	P80	P80	P80	P78	P79	x
EXP6-TAIL	P82	P82	P82	P82	P82	P82	P82	P82	x	x	x	x	x

The table was built using the following algorithm

Action table construction algorithm

begin
  $M \leftarrow \times$;
  foreach $A \rightarrow \alpha$ **do**
    foreach $a \in First^1(\alpha)$ **do**
      $M[A,a] \leftarrow M[A,a] \cup \text{Produce}(A \rightarrow \alpha)$;
    if $\varepsilon \in First^1(\alpha)$ **then**
      foreach $a \in Follow^1(A)$ **do**
        $M[A,a] \leftarrow M[A,a] \cup \text{Produce}(A \rightarrow \alpha)$;
  foreach $a \in T$ **do** $M[a,a] \leftarrow \text{Match}$;
  $M[\$, \varepsilon] \leftarrow \text{Accept}$;

And the needed First() and Follow() were found with those algorithms

***First^k* sets construction algorithm**

begin
  foreach $a \in T$ **do** $\text{First}^k(a) \leftarrow \{a\}$
  foreach $A \in V$ **do** $\text{First}^k(A) \leftarrow \emptyset$
  repeat
    foreach $A \in V$ **do**
      $\text{First}^k(A) \leftarrow \text{First}^k(A) \cup \{x \in T^* \mid A \rightarrow Y_1Y_2 \dots Y_n \wedge x \in \text{First}^k(Y_1) \oplus^k \text{First}^k(Y_2) \oplus^k \cdots \oplus^k \text{First}^k(Y_n)\}$
  until *stability*

***Follow^k* sets construction algorithm**

begin
  foreach $A \in V$ **do** $\text{Follow}^k(A) \leftarrow \emptyset$;
  repeat
    if $B \rightarrow \alpha A \beta \in P$ **then**
      $\text{Follow}^k(A) \leftarrow \text{Follow}^k(A) \cup \{\text{First}^k(\beta) \oplus^k \text{Follow}^k(B)\}$;
  until *stability*;

Variable	First()	Follow()
<PROGRAM>	int, bool, def, ID, print, read, return, if, while, put, putr, get, hold, release, reset	
<FUNCTIONLIST>	def	int, bool, ID, print, read, return, if, while, put, putr, get
<FUNCTION>	def	
<FTYPE>	int, bool	ID
<ARGS>	int, bool	)
<ARGLIST>	int, bool	
<NEXTARG>	,	)
<TYPE>	int, bool	
<BODY>	\n	
<VARDECL>	int, bool	ID, def, read, print, return, pass, put, putr, get, hold, release, reset, if, while
<VARLINE>	int, bool	
<IDLIST>	ID	
<NEXTID>	,	\n
<MAIN>	int, bool, ID, def, read, print, return, pass, put, putr, get, hold, release, reset, if, while	
<STATEMENTLIST>	ID, read, print, return, pass, put, putr, get, hold, release, reset, if, while	
<NEXTSTATEMENT>	ID, read, print, return, pass, put, putr, get, hold, release, reset, if, while	$, DEDENT
<STATEEMENT>	ID, read, print, return, pass, put, putr, get, hold, release, reset, if, while	
<SIMPLESTMT>	ID, read, print, return, pass, put, putr, get, hold, release, reset	

<SIMPLESTMT-TAIL>	=)	
<COMPOUNDSTMT>	if, while	
<IF>	if	
<IF-TAIL>	else	ID, read, print, return, pass, put, putr, get, hold, release, reset, if, while, DEDENT, $
<WHILE>	while	
<SCREENCOMMAND>	put, putr, get, hold, release, reset	
<CALLARGS>	–, not, ID, (, INTEGER_LITERAL, BOOLEAN_LITERAL	)
<EXPRLIST>	–, not, ID, (, INTEGER_LITERAL, BOOLEAN_LITERAL	
<NEXTEXPR>	,	)
<EXPRESSION>	–, not, ID, (, INTEGER_LITERAL, BOOLEAN_LITERAL	,) : \n DEDENT $
<T>	or	,) : \n or
<EXP1>	–, not, ID, (, INTEGER_LITERAL, BOOLEAN_LITERAL	,) : \n DEDENT $ or
<T1>	and	,) : \n or
<EXP2>	–, not, ID, (, INTEGER_LITERAL, BOOLEAN_LITERAL	,) : \n DEDENT $ and
<T2>	< > = !	,) : \n or and
<EXP3>	- , not ID (INTEGER_LITERAL BOOLEAN_LITERAL	,) : \n DEDENT $ + -
<T3>	+ -	,) : \n or and < > = !
<EXP4>	- , not ID (INTEGER_LITERAL BOOLEAN_LITERAL	,) : \n DEDENT $ * /
<T4>	* /	,) : \n or and < > = ! + -
<EXP5>	- , not ID (INTEGER_LITERAL BOOLEAN_LITERAL	,) : \n DEDENT $ – not

<EXP6>	(ID INTEGER_LITERAL BOOLEAN_LITERAL	,) : \n DEDENT $
<EXP6-TAIL>	(	,) : \n DEDENT $

Decorate the grammar to produce adequate low level code for GPMachine/CSS

[1] <PROGRAM>→ <VARDECL>[ujp @main]<FUNCTIONLIST>[define @main]<MAIN>$ [stp]
[2] <FUNCTIONLIST> → <FUNCTION> † <FUNCTIONLIST>
[3] → ε
[4] <FUNCTION>→ def <FTYPE> ID [define @id.value] (<ARGS>): <BODY> [{retf, retp}]
[5] <FTYPE> → <TYPE>
[6] → ε
[7] <ARGS> → <ARGLIST>
[8] → ε
[9] <ARGLIST> → <TYPE> ID <NEXTARG>
[10] <NEXTARG> → ,<ARGLIST>
[11] → ε
[12] <TYPE> → int
[13] → bool
[14] <BODY> → † INDENT <VARDECL> <STATEMENTLIST> DEDENT
[15] <VARDECL> → <VARLINE> <VARDECL>
[16] → ε
[17] <VARLINE> → <TYPE> <IDLIST> †
[18] <IDLIST> → ID [ldc {i,b} 0] <NEXTID>
[19] <NEXTID> → , <IDLIST>
[20] → ε
[21] <MAIN> → <VARDECL><STATEMENTLIST>
[22] <STATEMENTLIST> → <STATEMENT> <NEXTSTATEMENT>
[23] <NEXTSTATEMENT> → <STATEMENTLIST>
[24] → ε
[25] <STATEMENT> → <SIMPLESTMT> †
[26] → <COMPOUNDSTMT>
[27] <SIMPLESTMT> →ID{L=symbols.getLocationOf(id,name)}<SIMPLESTMT-TAIL>
[28] → read(ID{L=symbols.getLocationOf(id,name)}) [read] [str {i,b} 0 L]
[29] → print(<EXPRESSION>) [prin]
[30] → return <EXPRESSION> [str {i,b} 0 L]
[31] → <SCREENCOMMAND>
[32] <SIMPLESTMT-TAIL> → = <EXPRESSION>[str {i,b} 0 L]
[33] → (<CALLARGS>) [cup N id.value {mst1, }] ;mst1 si appel d'ailleurs que «main»
[34] <COMPOUNDSTMT> → <IF>
[35] → <WHILE>
[36] <IF> → if <EXPRESSION>[{fjp @else, }]: <BODY>[ujp @after_if]<IF-TAIL>[define @after_if] ;fjp si else
[37] <IF-TAIL> → else: [define @else]<BODY>
[38] → ε
[39] <WHILE> → while [define @while]<EXPRESSION>[fjp @after_while]: <BODY>[ujp @while] [define @after_while]
[40] <SCREENCOMMAND> → put(<EXPRESSION>, <EXPRESSION>, <EXPRESSION>) [put]
[41] → putr(<EXPRESSION>, <EXPRESSION>) [putr]
[42] → hold [hold]
[43] → release [rls]
[44] → reset [rst]

[45] <CALLARGS>	→ [{mst 0,mst 1}]<EXPRLIST>
[46]	→ ε
[47] <EXPRLIST>	→ <EXPRESSION> <NEXTEXPR>
[48] <NEXTEXPR>	→ , <EXPRLIST>
[49]	→ ε
[50] <EXPRESSION>	→ <EXP1><T>
[51] <T>	→ or <EXP1>[or b] <T>
[52]	→ ε
[53]<EXP1>	→<EXP2><T1>
[54]<T1>	→ and <EXP2>[and b]<T1>
[55]	→ ε
[56]<EXP2>	→ <EXP3><T2>
[57]<T2>	→ > <EXP3>[grt i]<T2>
[58]	→ < <EXP3>[les i]<T2>
[59]	→ <= <EXP3>[leq i]<T2>
[60]	→ >= <EXP3>[geq i]<T2>
[61]	→ == <EXP3>[equ {i,b}]<T2>
[62]	→ != <EXP3>[neq {i,b}]<T2>
[63]	→ ε
[64]<EXP3>	→ <EXP4><T3>
[65]<T3>	→ + <EXP4>[add i]<T3>
[66]	→ - <EXP4>[sub i]<T3>
[67]	→ ε
[68]<EXP4>	→ <EXP5><T4>
[69]<T4>	→ *<EXP5>[mul i]<T4>
[70]	→ / <EXP5>[div i]<T4>
[71]	→ ε
[72]<EXP5>	→ - <EXP5>[neg i]
[73]	→ not <EXP5>[not b]
[74]	→ <EXP6>
[75]<EXP6>	→ (<EXPRESSION>)
[76]	→ ID{L=symbols.getLocationOf(id,name)}<EXP6-TAIL>
[77]	→ INTEGER_LITERAL [ldc i INTEGER_LITERAL.val]
[78]	→ BOOLEAN_LITERAL [ldc b BOOLEAN_LITERAL.val]
[79]	→ get(<EXPRESSION>, <EXPRESSION>) [get]
[80]	→ ε
[81]<EXP6-TAIL>	→ (<CALLARGS>)[cup N @id.value]
[82]	→ ε[{lod,ldo} {i,b} 0 address]

Program a recursive decent LL(1) parser that outputs code for GPMachine/CSS

The recursive descent LL(1) parser was implemented in RecursiveDescentParser.java.
It actually follows the decorated grammar. A function was declared for each variable of the grammar. Each of these functions are asked to match the tokens following the grammar thanks to the scanner and generating output code contained in the decoration.

Code generation is located in Generator.java that write GPMachine instructions in a text file according to what the parser tells it to write. It uses an instance of the class OutputCode to delegate the writing in the file itself.
The whole program is exception sensible, i.e., as soon as something goes wrong (the token was not the one expected, the called variable has not been declared, ...) it throws an exception and stops the parsing and tries to tell you what went wrong.

A symbol table is also created during the parsing. Each declaration of variable, function or argument triggers the storing of their type and their name in a symbol table. To be even sharper, we must say that the functions and their arguments are stored in instances of the class Function.java and the three elements (functions, variables, arguments) are stored in an instance of the class SymbolTable.java.

In order to create effective output code, we also implemented the class AddressTable.java whose each instance represents a stack frame. It simulates the behavior of the stack, meaning it is going to store names and address on the stack of each variable or argument. That class helps us to manage relative addressing when creating output code.

The compiler must support systematic type checking

TyPy being a typed language (bool and int) we had to check whether the types were respected at each moment. The type checking happens during parsing phase too. In order to do that, we implemented a lot of tests on types when having an operation (type checking of the two expressions concerned), having a return statement (type checking of the returned expression and the function type). The check uses the symbol table created almost simultaneously.

The compiler does not need to support variable shadowing

In order to do that, the parser checks when declaring variables that the name is not corresponding to another global variable. It uses the symbol table that knows whether a variable has been declared as global or not.

How to use the compiler

- Open a terminal
- Move to project folder
- Launch the Makefile with the instruction: make
- Run the program with the instruction: java Main
- When asked, enter the location of your file containing the TYPY code
- A file named "output.txt" is generated in the folder of the project