Language And Translators - TP Correction

Francesco Nieri

June 2024

Contents

1 TP1 **2**
1.1 Question 1 2
1.1.1 Question 1.1 2
1.1.2 Question 1.2 3
1.2 Question 2 4
1.2.1 Question 2.1 4
1.2.2 Question 2.2 4
1.3 Question 3 4
1.3.1 Question 3.1 4
1.3.2 Question 3.2 5
1.3.3 Question 3.3 5
1.3.4 Question 3.4 6
1.3.5 Question 3.5 7
1.3.6 Question 3.6 8

2 TP2 **9**
2.1 Question 1 9
2.1.1 Question 1.1 9
2.1.2 Question 1.2 9
2.1.3 Question 1.3 10
2.1.4 Question 1.4 11
2.2 Question 2 11
2.2.1 Question 2.1 11
2.2.2 Question 2.2 11
2.2.3 Question2.3 12
2.3 Question 3 13
2.4 Question 4 13
2.4.1 Question 4.1 13
2.4.2 Question 4.2 14
2.4.3 Question 4.3 14

3 TP3 **15**
3.1 Question 1 . . . 15
3.2 Question 2 . . . 15
3.2.1 Question 2.1 . . . 15
3.2.2 Question 2.2 . . . 16
3.2.3 Question 2.3 . . . 16
3.3 Question 3 . . . 17
3.3.1 Question 3.1 . . . 17
3.3.2 Question 3.2 . . . 17
3.3.3 Question 3.3 . . . 18

4 TP4 **21**
4.1 Question 1 . . . 21
4.2 Question 2 . . . 21
4.2.1 Question 2.1 . . . 21
4.2.2 Question 2.2 . . . 23
4.2.3 Question 2.3 . . . 23
4.2.4 Question 2.4 . . . 23

5 TP5 **25**
5.1 Question 1 . . . 25
5.2 Question 2 . . . 25
5.3 Question 3 . . . 26

6 TP6 **27**
6.1 Question 1 . . . 27
6.2 Question 2 . . . 27
6.3 Question 3 . . . 31

7 TP7 **32**
7.1 Question 1 . . . 32
7.2 Question 2 . . . 33
7.3 Question 3 . . . 33
7.4 Question 4 . . . 34

1 TP1

1.1 Question 1

Consider this piece of code from a fictional C-like programming language:

```
int x2 = 25;
while(x2>0) {
    // increment x2 by one
    x2++;
    int y = fread(file) % 10;
    if(y<=4) {
        x2 = x2 - 1;
    }
    printf("Value: %d", y);
}
```

1.1.1 Question 1.1

Give the symbol classes that a lexer would need to lex this code (in the course, we have seen some basic classes, like Identifier, or Number). Also make symbol classes for whitespaces (space, newline), and comments.

We will combine question 1 and 3 together, the item on the left is the symbol class, on the right we have the regular expression.

- Type: `int`
- Space: `\w`
- Identifier: $[a-zA-Z][azA-Z0-9]^*$
- Assignment op: $=$
- Number: $[0-9]^+$
- String: ” .* ”
- Special char: $[; (\,)\{\}]$
- Keyword: (if|while)
- Comparison operator: $[> \mid \leq]$
- Increment operator: $++$
- Comment : // .*\n
- Math operator: $(-|\%)$
- EOL character: ;
- Param separator: ,

1.1.2 Question 1.2

Write the sequence of symbols (<Token,Attribute>) that a lexer would generate for this code.

```
<Type, int >
<Identifier, x2 >
<Assignment operator, = >
<Number, 25 >
<EOL character, \backslashn >
<Keyword, while>
<Special character, ( >
<Identifier, x2>
<Comparison operator, > >
<Number, 0 >
<Special character, ) >
<Special character, \{ >
<Comment, //...>
<Identifier, x2>
<Increment operator, ++>
<EOL character, ;>
<Type, int>
<Identifier, y>
<Assignment operator, =>
<Identifier, fread>
<Special character, (>
<Math operator, \%>
<EOL character, ;>
<Keyword, if >
<Special character, (>
<Identifier, y>
<Comparison operator, \le>
<Number, 4>
<Special character, )>
<Special character, \{>
<Identifier, x2>
<Assignment operator, =>
<Identifier, x2>
<Math operator, ->
<Number, 1>
<EOL character, ;>
<Special character, \}>
<Identifier, printf>
<Special character, (>
<String, "Value: \%d">
<Param separator, ,>
<Identifier, y>
<Special character, )>
<EOL character, ;>
```

```
<Special character, \}>
```

1.2 Question 2

1.2.1 Question 2.1

The language of a RE can be described in English words. For example, one could say that the RE $(0|1)^*1$ stands for "all binary numbers ending with a 1". Describe the RE

$$(0|1)^*0(0|1)^*0(0|1)^*$$

in English words.

Any binary sequence that contains at least two zeros

1.2.2 Question 2.2

Construct a RE for "all binary numbers starting with a 1 and containing exactly one pair of consecutive 0".

$$1(01|1)^*00(1|10)^*$$

1.3 Question 3

1.3.1 Question 3.1

Draw the NFA for RE $a(bc)^*(d|ef)$ Indicate which state is the initial state and which states are the final states. See the NFA in question 3.5 on how to indicate the initial state and final states in an NFA diagram.

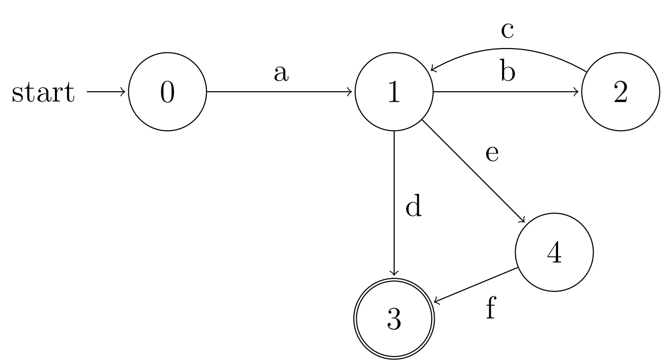


Figure 1: NFA for $a(bc)^*(d|ef)$

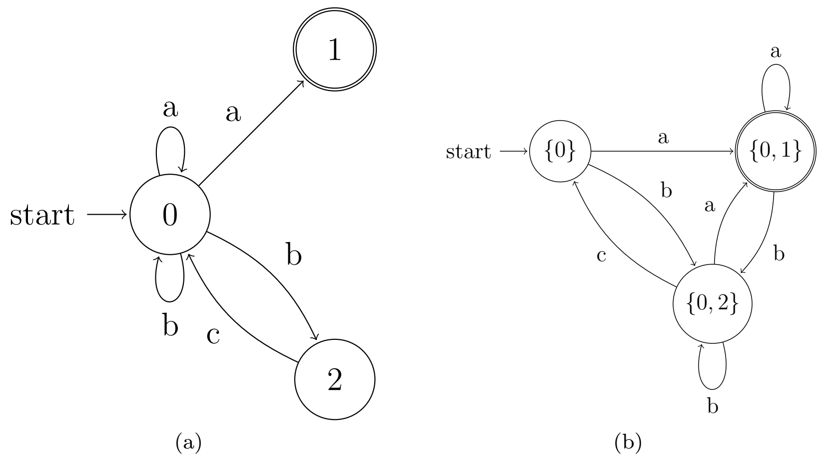


Figure 2: NFA and DFA for $(a|b|bc)^*a$

1.3.2 Question 3.2

Draw the NFA for $(a|b|bc)^*a$ and construct the DFA. The + sign can be used to express that a pattern must appear at least one time. What would be the NFA for the RE $(a|b|bc)^+a$?

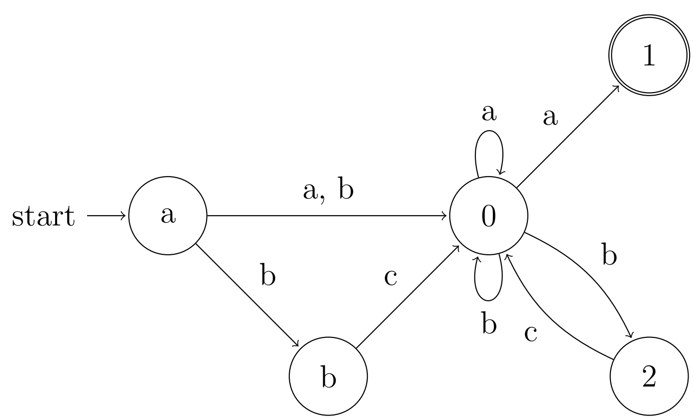
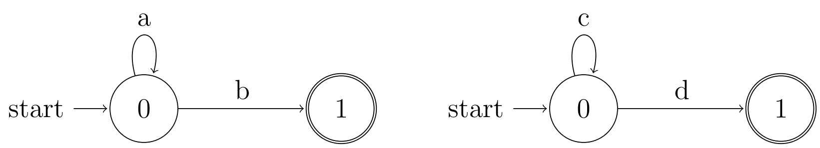


Figure 3: NFA for $(a|b|bc)^+a$

1.3.3 Question 3.3

ε-transitions are very convenient if you want to combine two NFAs. Draw first the NFAs for a^*b and c^*d and then draw an NFA for (a^*b) | (c^* d) using ε-transitions.



(a) (b)

Figure 4: NFA and DFA for $(a|b|bc)^*a$

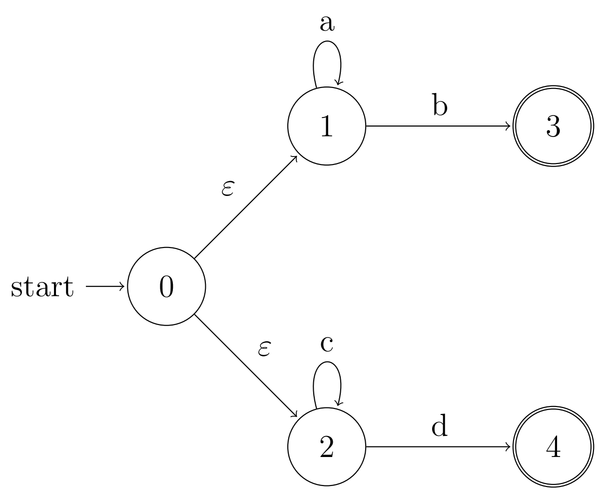


Figure 5: NFA for (a^*b) | (c^* d)

1.3.4 Question 3.4

Transform the NFA for (a*b) | (c^*d) from question 3.3 into an NFA without ε-transitions.

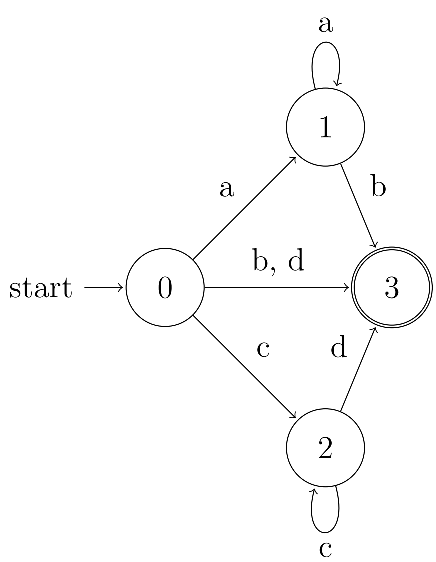


Figure 6: NFA for (a*b) | (c^*d) without ε-transitions

1.3.5 Question 3.5

When dealing with NFAs with ε-transitions, it is useful to think about the ε-closure ε(s) of a state s, which is defined as the set of states that can be reached from that state s by doing one or several (!) ε-transition steps. For the below NFA over the alphabet Ω = 0,1, give the ε-closures ε(A), ε(B), ε(C) of the states A, B, C.

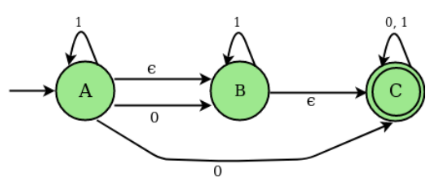


Figure 7

ε-closures:

$$\varepsilon\{A\} = \{A, B, C\}$$
$$\varepsilon\{B\} = \{B, C\}$$
$$\varepsilon\{C\} = \{C\}$$

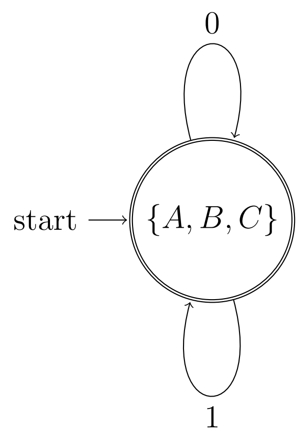


Figure 8: DFA for the above graph

1.3.6 Question 3.6

Here is a modification of the powerset construction method seen in the course that also works with NFAs containing ε-transitions:
• The initial state of the powerset automaton is ε(s0).
• The set of transitions T' of the powerset automaton is defined as:
$\forall Q \subseteq S, c \in \Omega : (Q, c, R) \in T' for R = \varepsilon(\{t | s \in Q, (s, c, t) \in T)$ **Compare this modified method with the method that we have seen in the course. What is the difference? Transform the NFAs from question 3.3 (the combined NFA) and from question 3.5 into DFAs using this modified powerset construction method.**

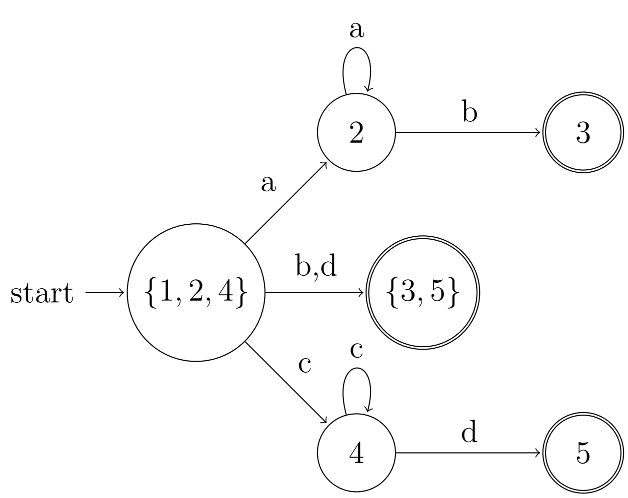


Figure 9: Combined DFA from NFAs

2 TP2

2.1 Question 1

Consider the CFG G with the rules:

$$S \rightarrow a \mid (S) \mid S \cdot S \mid S + S \mid -S$$

2.1.1 Question 1.1

Give the set of terminal symbols used in the rules of G

$$\Sigma = \{a, (,), -, \cdot, +\}$$

2.1.2 Question 1.2

Give a leftmost analysis of the input $a \cdot (-a + (a))$ and draw the syntax tree.

$(a \cdot (-a + (a)), S, \varepsilon)$
$(a \cdot (-a + (a)), S \cdot S, 3)$ Rule 3 to expand S
$(a \cdot (-a + (a)), a \cdot S, 31)$ Rule 1 to expand S
$(\cdot(-a + (a)), \cdot S, 31)$ Match a
$((-a + (a)), S, 31)$ Match $\cdot$
$((-a + (a)), (S), 321)$ Rule 2 to expand S
$(-a + (a)), S), 312)$ Match (
$((-a + (a)), S + S), 3124)$ Rule 4 to expand S
$(-a + (a)), -S + S), 31245)$ Rule 5 to expand S
$(a + (a)), S + S), 31245)$ Match -
$(a + (a)), a + S), 312451)$ Rule 1 to expand S
$(+(a)), +S), 312451)$ Match a
$((a)), S), 312451)$ Match +
$((a)), (S)), 3124512)$ Rule 2 to expand S
$(a)), S)), 3124512)$ Match (
$(a)), a)), 31245121)$ Rule 1 to expand S
$()),)), 31245121)$ Match a
$(),), 31245121)$ Match)
$(\varepsilon, \varepsilon, 31245121)$ Match)

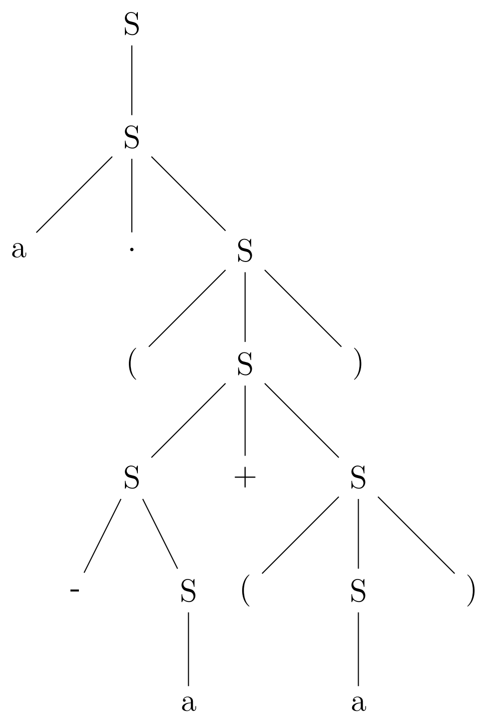


Figure 10: Syntax tree for $a \cdot (-a + (a))$

2.1.3 Question 1.3

Give a rightmost analysis of the input $a \cdot (-a + (a))$
$(a \cdot (-a + (a)), S, \varepsilon)$
$(a \cdot (-a + (a)), S \cdot S, 3)$ Rule 3 to expand S
$(a \cdot (-a + (a)), S \cdot (S), 32)$ Rule 2 to expand S
$(a \cdot (-a + (a)), S \cdot (S, 32)$ Match)
$(a \cdot (-a + (a), S \cdot (S + S, 324)$ Rule 4 to expand S
$(a \cdot (-a + (a), S \cdot (S + S, 324)$ Rule 4 to expand S
$(a \cdot (-a + (a), S \cdot (S + (S), 3242)$ Rule 2 to expand S
$(a \cdot (-a + (a, S \cdot (S + (S, 3242)$ Match)
$(a \cdot (-a + (a, S \cdot (S + (a, 32421)$ Rule 1 to expand S
$(a \cdot (-a + (, S \cdot (S + (, 32421)$ Match a
$(a \cdot (-a+, S \cdot (S+, 32421)$ Match (
$(a \cdot (-a, S \cdot (S, 32421)$ Match +
$(a \cdot (-a, S \cdot (-S, 324215)$ Rule 5 to expand S
$(a \cdot (-a, S \cdot (-a, 3242151)$ Rule 1 to expand S
$(a \cdot (-, S \cdot (-, 3242151)$ Match a
$(a \cdot (, S \cdot (, 3242151)$ Match -
$(a \cdot, S \cdot, 3242151)$ Match (

$(a, S, 3242151)$ Match $\cdot$
$(a, a, 32421511)$ Rule 1 to expand S
$(\varepsilon, \varepsilon, 32421511)$ Match a

2.1.4 Question 1.4

Show that G is ambiguous.
With a leftmost derivation we can do the following:

$$S \xrightarrow{5} -S \xrightarrow{4} -S + S \xrightarrow{1} -a + S \xrightarrow{1} -a + a = \texttt{5411}.$$

But also:

$$S \xrightarrow{4} S + S \xrightarrow{5} -S + S \xrightarrow{1} -a + S \xrightarrow{1} -a + a = \texttt{4511}$$

2.2 Question 2

Extend the CFG for arithmetic expressions from the slides

$$\begin{gathered} E \rightarrow E + T \mid T \\ T \rightarrow T \cdot F \mid F \\ F \rightarrow (E) \mid Number \mid Identifier \end{gathered}$$

by:

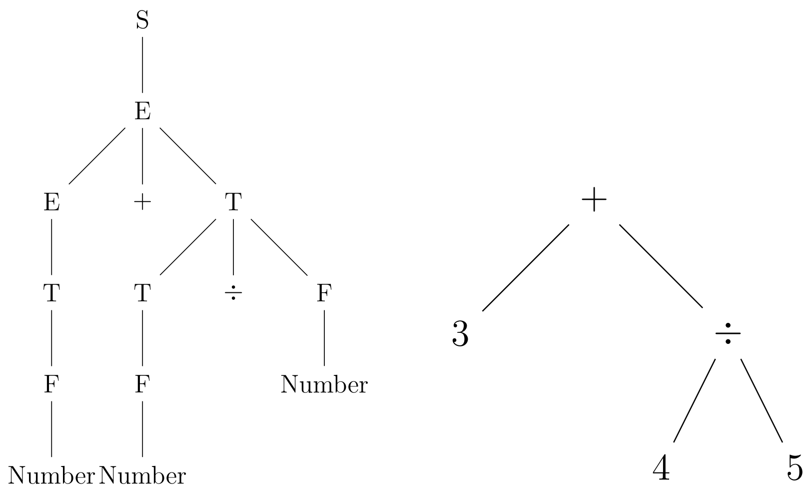
2.2.1 Question 2.1

adding the binary minus operator and the division operator

$$\begin{gathered} E \rightarrow E + T \mid E - T \mid T \\ T \rightarrow T \cdot F \mid T/F \mid F \\ F \rightarrow (E) \mid Number \mid Identifier \end{gathered}$$

2.2.2 Question 2.2

Write the syntax tree for 3 + 4/5. Then manually turn it into a more readable AST.



(a) (b)

Figure 11: AST for 3 + 4/5 and readable AST

2.2.3 Question2.3

You will notice that the AST in question 2 does not only reflect the syntactic structure of the input but also the precedence of the / operator over the + operator. Add a new operator < to the grammar that has lower precedence than the + operator and the / operator and test the new grammar on the input $4 + 7 < 3 + 4/5$.

$$
\begin{gathered}
D \to D < E \mid E \\
E \to E + T \mid E - T \mid T \\
T \to T \cdot F \mid T/F \mid F \\
F \to (E) \mid Number \mid Identifier
\end{gathered}
$$

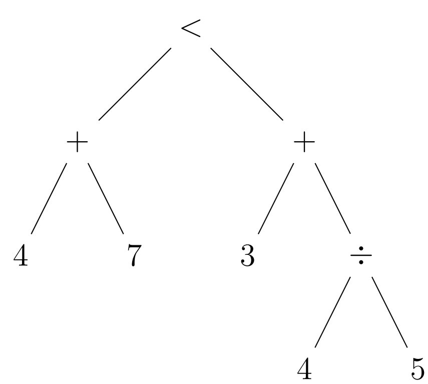


Figure 12

2.3 Question 3

Here is a CFG for regular expressions over the alphabet a, b. That means the CFG describes how regular expressions look like.

$$S \to R$$
$$R \to R + R \mid RR \mid R\cdot \ \ |T$$
$$T \to (R)|\ a \mid b$$

Note that we are using the plus-symbol "+" (e.g., "a+b") to represent the choice in regular expression $(a|b)$ **to avoid confusion with the | used by the CFG itself. Run the NTA of this grammar for the input** $a + (b\cdot)$

$(a + (b\cdot), S, \varepsilon)$
$(a + (b\cdot), R, 1)$ Rule 1 to expand S
$(a + (b\cdot), R + R, 12)$ Rule 2 to expand S
$(a + (b\cdot), T + R, 125)$ Rule 5 to expand R
$(a + (b\cdot), a + R, 1257)$ Rule 7 to expand T
$(+(b\cdot), +R, 1257)$ Match a
$((b\cdot), R, 1257)$ Match +
$((b\cdot), T, 1257)$ Rule 5 to expand R
$((b\cdot), (R), 125756)$ Rule 6 to expand T
$(b\cdot), R), 125756)$ Match (
$(b\cdot), R\cdot), 1257564)$ Rule 4 to expand R
$(b\cdot), T\cdot), 1257645)$ Rule 5 to expand R
$(b\cdot), b\cdot), 125767458)$ Rule 8 to expand R
$(\cdot), \cdot), 125767458)$ Match b
$(),), 125767458)$ Match $\cdot$
$(\varepsilon, \varepsilon, 125767458)$ Match)

2.4 Question 4

Consider the CFG G with the two rules

$$S \to \ ab \mid ac$$

over the terminal symbols $\{a, b, c\}$.

2.4.1 Question 4.1

Explain why G is not LL(1). Is it LL(2) ?
A grammar is LL(1) if and only if, for all rules

$$A \to \beta|\gamma$$
$$la(A \to \beta) \cap la(A \to \gamma) = \varnothing$$

with $(\beta \neq \gamma)$

It is not LL(1) as:

$$first_1(S \to ab) = first_1(S \to ac) = \{a\}$$

It is LL(2) as:

$$first_2(S \to ab) = \{ab\} \neq first_2(S \to ac) = \{ac\}$$

2.4.2 Question 4.2

Write a CFG that is LL(1) and that generates the same language as G. Hint: the new grammar has two non-terminal symbols.

$$\begin{gathered} S \to aR \\ R \to b \mid c \end{gathered}$$

2.4.3 Question 4.3

Give a CFG similar to G that is not LL(5)

$$S \to aaaaab \mid aaaaac$$

3 TP3

3.1 Question 1

Show that any regular language can be generated by an LL(1) language.

For a regular language, we can write a regular grammar. That grammar corresponds to an NFA that we can transform into a DFA. We will show that the DFA can be translated to a LL(1) grammar.

Since a LL(1) grammar generates (or recognizes) a LL(1) language, we have achieved our goal.

To translate a DFA into an LL(1) grammar, we do the following:

The states of the DFA become non-terminal symbols and the transitions become terminal symbols in the grammar.

Each transition in the DFA is translated to a rule in the grammar.

Example: the transition A $\xrightarrow{a}$ B of the DFA with two states A and B becomes the rule A $\rightarrow$ a B.

In the course, we have seen the theorem: A grammar is LL(1) if and only if for all rules $A \rightarrow \beta|\gamma(\text{with}\beta \neq \gamma)$ $la(A \rightarrow \beta) \cap la(A \rightarrow \gamma) = \varnothing$ Having $la(A \rightarrow \beta) \cap la(A \rightarrow \gamma) \neq \varnothing$ is not possible in our case because that would mean that the DFA had a state with two transitions with the same symbol, i.e., a non-deterministic transition.

3.2 Question 2

Consider the following grammar. This a CFG for regular expressions over the alphabet {a, b}

$$\begin{gathered} S \rightarrow R \\ R \rightarrow R + T \mid RT \mid R\cdot \mid T \\ T \rightarrow (R) \mid a \mid b \end{gathered}$$

3.2.1 Question 2.1

Transform the grammar into a grammar without left recursion and prove that the result is an LL(1) grammar.

$$\begin{gathered} S \rightarrow R \\ R \rightarrow TR' \\ R' \rightarrow +TR' \mid TR' \mid \cdot R' \mid \varepsilon \\ T \rightarrow (R) \mid a \mid b \end{gathered}$$

It is LL(1) as:

$$\begin{gathered} la(S \rightarrow R) = \{(, a, b\} \\ la(R \rightarrow TR') = \{(, a, b\} \end{gathered}$$

$$\cap = \varnothing \begin{cases} la(R' \to +TR') = \{+\$ \\ la(R' \to TR') = \{(, a, b\} \\ la(R' \to \cdot TR') = \{\cdot\} \\ la(R' \to \varepsilon) = \{\varepsilon,)\} \end{cases} \tag{1}$$

$$\cap = \varnothing \begin{cases} la(T \to (R)) = \{(\} \\ la(T \to a) = \{a\} \\ la(T \to b) = \{b\} \end{cases} \tag{2}$$

3.2.2 Question 2.2

Specify the Deterministic Top-Down Automaton for the transformed grammar. For the possible actions, either write the transitions or give the action table.

	S	R	T	R’	a	b	+	·	(	)	ε
a	(R, 1)	(TR’, 2)	(a, 8)	(TR’, 4)	pop						
b	(R, 1)	(TR’, 2)	(b, 8)	(TR’, 4)		pop					
+				(+TR’,3)			pop				
·				(· R’,5)				pop			
a	(R, 1)	(TR’, 2)	((R), 7)	(TR’, 4)					pop		
)				(ε, 6)						pop	
ε				(ε,6)							accept

3.2.3 Question 2.3

Run the deterministic automaton on the input $a + (b\cdot)$
$(a + (b\cdot), S, \varepsilon)$
$(a + (b\cdot), R, 1)$ Rule 1 to expand S
$(a + (b\cdot), TR', 12)$ Rule 2 to expand R
$(a + (b\cdot), aR', 128)$ Rule 8 to expand T
$(+(b\cdot), R', 128)$ Match a
$(+(b\cdot), +TR', 1283)$ Rule 3 to expand R’
$((b\cdot), TR', 1283)$ Match +
$((b\cdot), (R)R', 12837)$ Rule 7 to expand T
$(b\cdot), R)R', 12837)$ Match (
$(b\cdot), TR')R', 128372)$ Rule 2 to expand R
$(b\cdot), bR')R', 1283729)$ Rule 9 to expand T
$(\cdot), R')R', 1283729)$ Match b
$(\cdot), \cdot R')R', 12837295)$ Rule 5 to expand R’
$(), R')R', 12837295)$ Match $\cdot$
$(), \varepsilon)R', 128372956)$ Rule 6 to expand R’
$(),)R', 128372956)$ Match ε

$(\varepsilon, R', 128372956)$ Match)
$(\varepsilon, \varepsilon, 1283729566)$ Rule 6 to expand R'

3.3 Question 3

Here is a CFG for Boolean expressions:

Expression → Factor | Expression or Factor | Expression and Factor
Factor → not Factor | (Expression) | true | false

3.3.1 Question 3.1

Show the syntax tree for the expression

true and false or (false and true)

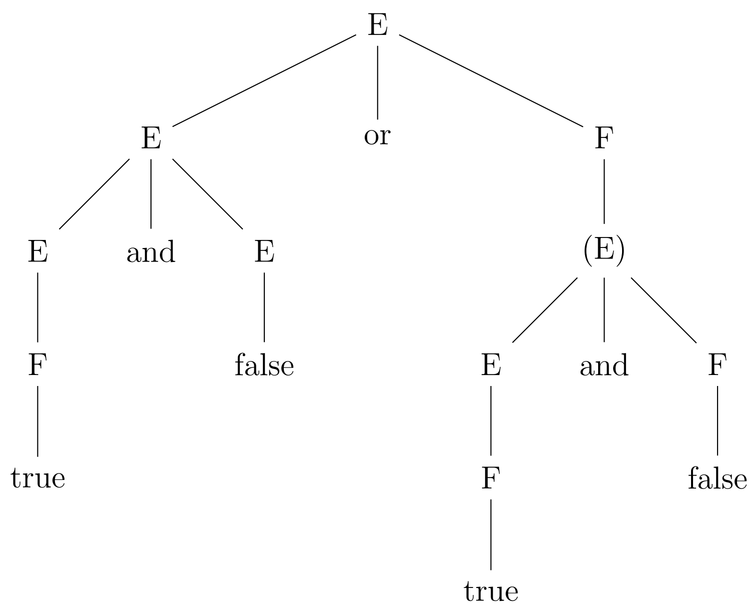


Figure 13: Syntax tree

3.3.2 Question 3.2

Transform the grammar into a grammar without left recursion.

$$E \to FE'$$
$$E' \to or\ FE' \mid and\ FE' \mid \varepsilon$$
$$F \to not\ F \mid (E) \mid true \mid false$$

3.3.3 Question 3.3

Write the outline of a recursive descent parser for the transformed grammar, similar to what we did in the video on LL(1) parsing. The main challenge here is to design good data structures for the abstract syntax tree.

```
enum Symbol {
    Or,
    And,
    Not,
    True,
    False,
    OpenParen,
    ClosingParen,
    END
}

class Expression {
    Expression leftHand;
    Symbol operation;
    Expression rightHand;

    public Expression(Expression left, Symbol op, Expression
        right) {
            leftHand = left;
            operation = op;
            rightHand = right;
    }

    public Symbol getReverseOperation() {
        switch (operation) {
            case Symbol.Or:
                return Symbol.And;
            case Symbol.And:
                return Symbol.Or;
            case Symbol.True:
                return Symbol.False;
            case Symbol.False:
                return Symbol.True;
            default:
                throw new ParserException(
"Unsupported expression operation"
);
            }
    }

    public boolean isFactor() {
            return operation == Symbol.True || operation ==
                Symbol.False;
```

```
    }

    public void setLeftHand(Expression expr) {
            leftHand = expr;
    }
}

class Lexer {
    List<Symbol> symbols;
    Symbol lookahead;
    Symbol nextSymbol(); // I can't be bothered to implement that, so pretend it exists


    public Lexer(List<Symbol> symbols) {
            this.symbols = symbols;
        this.lookahead = nextSymbol();
    }

    private Symbol match(Symbol symbol) {
            if (lookahead != symbol) {
                throw new ParserException("No match");
        }
            Symbol matchingSymbol = lookahead;
        lookahead = lexer.nextSymbol();
            return matchSymbol;
    }



    private Factor parseFactor() {
        switch (lookahead) {
            case Symbol.True:
                match(Symbol.True);
                return new Expression(null, Symbol.True, null);
            case Symbol.False:
                match(Symbol.False);
                return new Expression(null, Symbol.False, null);
            case Symbol.Not:
                match(Symbol.Not);
                Expression factorToNegate = parseFactor();
                Symbol reverseOp = factorToNegate.getReverseOperation();
                return new Expression(null, reverseOp, null);
            case Symbol.OpenParen:
                match(Symbol.OpenParen);
                Expression expr = parseExpression();
                match(Symbol.ClosingParen);
                return expr;
```

```
            default:
                throw new ParserException("No match");
        }
    }


    private Expression parseExpression() {
        Expression leftHand = parseFactor();
        assert leftHand.isFactor();
        Expression expr = parseExpressionPrime();
        if (expr != null) {
            expr.setLeftHand(leftHand);
            return expr;
        } else {
            return leftHand;
        }
    }

    private Symbol parseExpressionPrime() {
        switch (lookahead) {
            case Symbol.END:
                match(Symbol.END);
                return new Expression(null, Symbol.END, null);
            case Symbol.And:
                match(Symbol.And);
                Expression rightHand = parseExpression();
                return new Expression(null, Symbol.And,
                    rightHand);
            case Symbol.Or:
                match(Symbol.Or);
                Expression rightHand = parseExpression();
                return new Expression(null, Symbol.Or,
                    rightHand);
            default: // epsilon rule
                return null;
        }
    }

    // Lexer entry point for outside users
    public Expression parse() {
            return parseExpression();
    }
}
```

4 TP4

4.1 Question 1

Why is nobody interested in LL(0) parsers? How does an LL(0) grammar look like?

An LL(0) parser is a parser that doesn't do a lookahead to decide which rule to apply. It can only take the next input symbol and check whether it matches or not. Such a parser could only recognize one word.

4.2 Question 2

Consider the following grammar:

$$
\begin{gathered}
S' \to S \\
S \to A \mid Bd \\
A \to aAb \mid Be \\
B \to aBc \mid ac
\end{gathered}
$$

4.2.1 Question 2.1

Describe the deterministic LR(0) parsing automaton of this grammar and give the parsing table

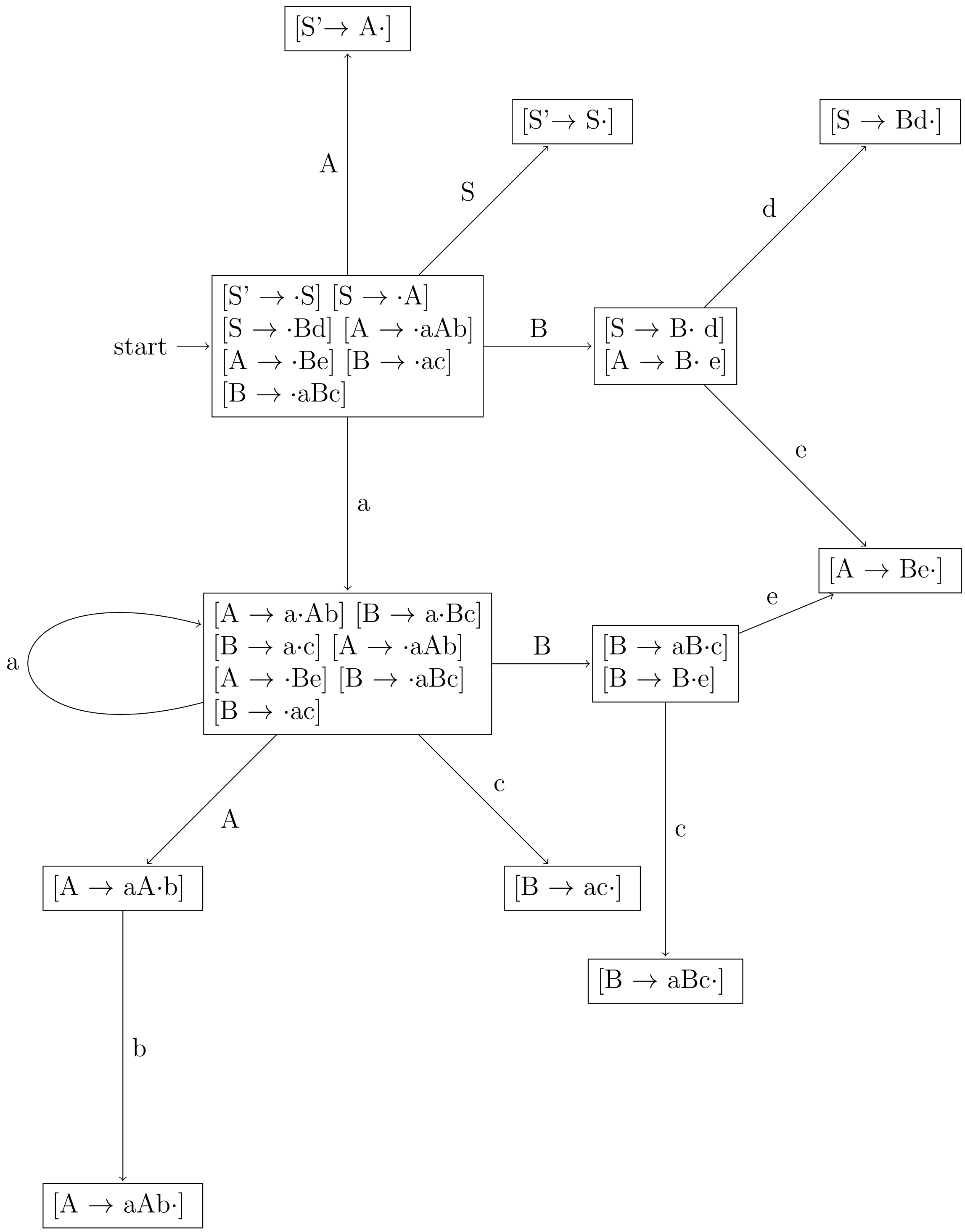


Figure 14: LR(0) goto automaton

	S	A	B	a	b	c	d	e	action
$I_0 = \varepsilon$	I_1	I_2	I_3	I_6					shift
$I_1 = S$									accept
$I_2 = A$									reduce 2
$I_3 = B$							I_4	I_5	shift
$I_4 = Bd$									reduce 3
$I_5 = Be$									reduce 5
$I_6 = a$		I_7	I_9	I_6		I_{11}			shift
$I_7 = aA$					I_8				shift
$I_8 = aAb$									reduce 4
$I_9 = aB$						I_{10}		I_5	shift
$I_{10} = aBc$									reduce 6
$I_{11} = ac$									reduce 7
$I_{12} = \emptyset$									error

Table 1: Parsing table for G

4.2.2 Question 2.2

Show that the grammar is not LL(1).

After reading a, it could be either rule 3 or rule 5/6, so the grammar isn't LL(1)

4.2.3 Question 2.3

Run the automaton on the input aaacebb

$(aaacebb, I_0, \varepsilon)$ Shift
$(aacebb, I_0I_6, \varepsilon)$ Shift
$(acebb, I_0I_6I_6, \varepsilon)$ Shift
$(cebb, I_0I_6I_6I_6, \varepsilon)$ Shift
$(ebb, I_0I_6I_6I_6I_{11}, \varepsilon)$ Shift
$(ebb, I_0I_6I_6I_9, 7)$ Reduce
$(bb, I_0I_6I_6I_9I_5, 7)$ Shift
$(bb, I_0I_6I_6I_7, 75)$ Reduce
$(b, I_0I_6I_6I_7I_8, 75)$ Shift
$(b, I_0I_6I_7, 754)$ Reduce
$\varepsilon, I_0I_6I_7I_8, 754)$ Shift
$(\varepsilon, I_0I_2, 7544)$ Reduce
$(\varepsilon, I_0I_1, 75442)$ Reduce
$(\varepsilon, \varepsilon, 754421)$ Reduce

4.2.4 Question 2.4

Modify the grammar such that it has a shift/reduce or reduce/reduce conflict.

We can add some rules to create shift/reduce or reduce/reduce conflicts
Shift/reduce = When ε: [A $\rightarrow$ $\cdot$] in I.
Reduce/reduce = When a: [A $\rightarrow$ a$\cdot$], [B $\rightarrow$ C $\cdot$] in item set

$$A \rightarrow aAb \mid Be \mid \varepsilon$$
$$\text{LR(0)(aA)} = \{[(\text{A} \rightarrow \text{aA}\cdot\text{b}, [\text{A} \rightarrow \text{aA}\cdot]]$$

5 TP5

5.1 Question 1

Write type inference rules for the double type in Java

Double constants are of type double:

$$\frac{}{\vdash \text{any double constant: double}}$$

Adding two doubles:

$$\frac{\vdash e_1 : \text{double}, \vdash e_2 : \text{double}}{\vdash e_1 + e_2 : \text{double}}$$

Adding a double and an integer gives a double:

$$\frac{\vdash e_1 : \text{double}, \vdash e_2 : \text{int}}{\vdash e_1 + e_2 : \text{double}}$$

$$\frac{\vdash e_1 : \text{int}, \vdash e_2 : \text{double}}{\vdash e_1 + e_2 : \text{double}}$$

As you can see, allowing type conversions from int to double means that we have to repeat rules.
To reduce the number of rules, we can define that

$$\text{int} \leq \text{double}$$

In this example, we then only need two rules for the add operator:

$$\frac{\vdash e_1 : \text{double}, \vdash e_2 : \text{T} \leq \text{double}}{\vdash e_1 + e_2 : \text{double}}$$

$$\frac{\vdash e_1 : \text{int}, \vdash e_2 : \text{double}}{\vdash e_1 + e_2 : \text{double}}$$

5.2 Question 2

Write a type inference rule for array expressions. The rule should be applicable to expressions like a[i+3]

Let's assume we have already specified the type rules for variables and arithmetic expressions like $i + 3$. Then we can write for array expressions:

$$\frac{\begin{array}{c}\text{a is an identifier}\\ \text{a has the type of an array of int in the symbol table}\\ \vdash e : \text{int}\end{array}}{\vdash a[e] : T}$$

Write a type inference rule for the ternary conditional operator of Java and C, for example: $a < 3?b : 5$
For the ternary conditional operator, we have:

$$\frac{\begin{array}{c}\vdash b : \text{bool} \\ \vdash e_1 : T \\ \vdash e_2 : T\end{array}}{\vdash b?e_1 : e_2 : T}$$

5.3 Question 3

How would you check whether a function uses return statements correctly? Think about situations where this can be sometimes difficult to check. For example, how would you do it for this function:

```
int min(inta, int b) {
    int m;
    if (a<b) {
        m = a;
        return m;
    }
    else
        m = b;
}
```

If you represent the start of the function and the end of the function as nodes in the control flow graph, any path from the start node to the end node must contain a return statement.

6 TP6

6.1 Question 1

Give the intermediate representation in TAC for the following expression:

Give the intermediate representation in TAC for the following expression:

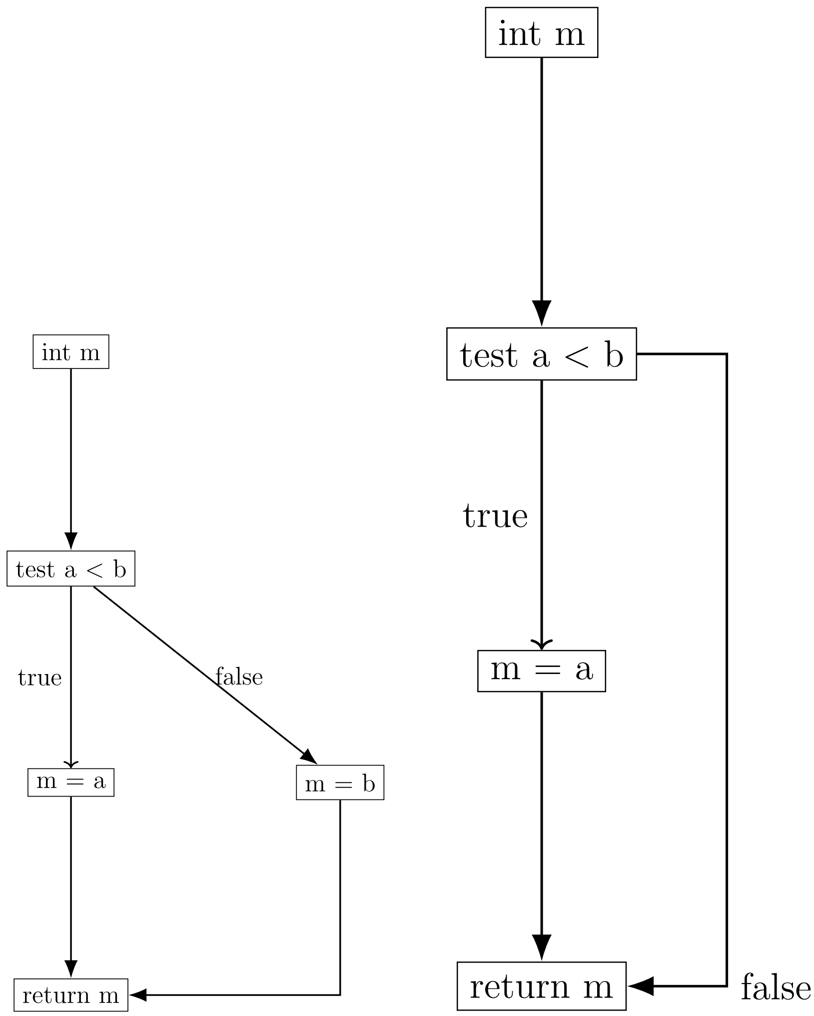
$$x = (-b + \text{sqrt}(b^2 - 4 \cdot a \cdot c)) / (2 \cdot a)$$

```
x0 = b * b
x1 = 4 * a
x2 = x1 * c
x3 = x0 - x2
x4 = sqrt(x3)
x5 = -b
x6 = x5 + x4
x7 = 2 * a
x8 = x6 / x7
x = x8
```

6.2 Question 2

For each of the following C functions, give the control flow graph (CFG), the minimized SSA form, and the non-SSA form without parameterized labels (the form that can be used to generate assembly code)

```
int min(int a, int b) {
    int m;
    if(a<b)
        m = a;
    else
        m = b;
    return m;
}
int minAlternative(int a, int b) {
    int m = b;
    if(a<b)
        m = a;
    return m;
}
int squareSum(int n) {
    int sum=0;
    for(int i=1;i<=n;i++){
        sum += i*i;
    }
    return sum;
}
```



(a) (b)

Figure 15: CFG for min and minAlternative $(a|b|bc)^*a$

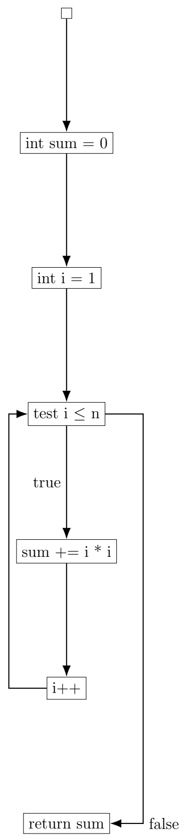


Figure 16: CFG for squareSum

Non minimized SSA

```
squareSum(n0):
    sum0 = 0
    i0 = 1
    goto loop(n0, sum0, i0)
loop(n1, sum1, i1):
    if i1<=n1 then goto body(n1,sum1,i1) else goto end(n1,sum1,
        i1)
body(n2, sum2, i2):
    t0 = i2*i2
    sum3 = sum2 + t0
    i3 = i2 + 1
    goto loop(n2, sum3, i3)
end(n3, sum4, i4):
    return sum4
```

Minimized SSA

```
squareSum(n0):
    sum0 = 0
    i0 = 1
    goto loop(sum0, i0)
loop(sum1, i1):
    if i1<=n0 then goto body else goto end
body:
    t0 = i1*i1
    sum3 = sum1 + t0
    i3 = i1+1
    goto loop(sum3, i3)
end:
    return sum1
```

Non minimized non-SSA

```
squareSum(n):
    sum = 0
    i = 1
    goto loop
loop:
    if i<=n then goto body else goto end
body:
    t = i*i
    sum = sum + t
    i = i +1
    goto loop
end:
    return sum
```

Minimized non-SSA

```
squareSum(n0):
    sum0 = 0
    i0 = 1
    sum1 = sum0
    i1 = i0
    goto loop
loop(sum1, i1):
    if i1<=n0 then goto body else goto end
body:
    t0 = i1*i1
    sum3 = sum1 + t0
    i3 = i1+1
    sum1 = sum3
    i1 = i3
    goto loop
end:
    return sum1
```

6.3 Question 3

Design a simple algorithm that verifies that a function has no missing return statement. The algorithm should also work for complex functions that contain many if-statements, loops, etc.

If you represent the start of the function and the end of the function as nodes in the control flow graph, any path from the start node to the end node must contain a return statement.

7 TP7

7.1 Question 1

Translate the following function to IR in SSA form and determine the liveness ranges of the variables. Draw the interference graph. Then, allocate registers for an ARM-like CPU and generate machine code, assuming that the parameter "c" of the function is passed in register r1 (without using the stack. That's more efficient!) and that the return value of the function should be in register r0.

	t1	c1	a1	t0	a0	b0	c0
a0 = c0 · 2							live
b0 = a0 + 1					live		live
c1 = c0 + b0					live	live	live
t0 = b0 · 2		live			live	live	
a1 = t0 + a0		live		live	live		
t1 = c1 + a1		live	live				
return t1	live						

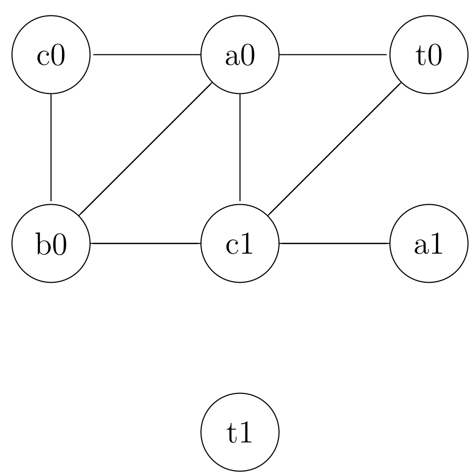


Figure 17: Interference graph

```
pop r0
c0 = r1 (required by the ABI)
a0 = r0
b0 = r2
c1 = r1
t0 = r2
a1 = r0
t1 = r0 (required by the ABI)
```

7.2 Question 2

In the course, we only saw code generation for functions with parameters and local variables. Now, let's look at an example with global variables:

```
int x;
int y[10];

void add(int v) {
    y[x] = v;
    x++;
}
```

Translate the above function "add" first to IR code and then to machine code. Assume that the global variables "x" and "y" start at address 0x10000 and 0x10004, respectively, in main memory.

```
// parameter is v0
t0 = 0x10004
t1 = *t0
 // read value of x
t2 = t1 * 4
 // each array element is four bytes
t3 = 0x10000
t4 = t0 + t3
 // address of y[x]
*t4 = v0
 // store v in y[x]
//-----------
t5 = 0x10004
t6 = *t5
t7 = t6 + 1
*t5 = t7
 // store x+1 in x
```

Of course, this code can be optimized. Instead of loading again the variable x in t6, we could just re-use t1.

7.3 Question 3

Function calls are expensive because they involve a lot of operations (pushing the arguments on the stack, making backups of registers, jumping to the function, etc.). Many compilers can perform an optimization called function inlining where the code of the called function is directly inserted at the call location, thus avoiding the call. Let's look at the following example:

```
int f(int v) {
    v = v + 2;
```

```
    return v*v;
}
int g(int a, int b) {
    int c = f(a);
    int d = 2*f(a+b);
    return c+d;
}
```

Take the role of the compiler and inline the function f at the two places where it is called. Do this inthe IR, not in the source code. Think about a strategy how to handle the variables. By the way, compilers only inline small functions. What could be the reason?

```
// parameters are a0 and b0
v0 = a0
 // calling function f with argument a
v1 = v0 + 2
t0 = v1 * v1
c0 = t0
 // c = f(a)
//-----------
v2 = a0 + b0
 // calling function f with argument a+b
v3 = v2 + 2
t1 = v3 * v3
d0 = 2 * t1
 // d = 2*f(a+b)
//-----------
t2 = c0 + d0
return t2
```

Why do compilers only inline small functions? Code size! Inlining increases the number of instructions in the code and therefore reduces the efficiency of the instruction cache.

7.4 Question 4

In most CPUs, integer multiplications are slower than additions or bit shifting. Think about ways to reduce the strength of multiplications, i.e., find ways to avoid the multiplication instruction for arithmetic expressions where one of the operands is a constant, for example

$$x \cdot 2$$
$$x \cdot 3$$
$$x \cdot 4$$
$$x \cdot 5$$

$$x \cdot 2 = \text{x+x or x<<1}$$
$$x \cdot 3 = \text{x+x+x or x<<1 + x}$$

$$x \cdot 4 = \text{x<<2}$$
$$x \cdot 5 = \text{x<<2 + x}$$
$$\text{x } \cdot (2^n + 1) = \text{(x<<n) + x}$$