TDDD55 Lesson 4
Exam preparation

John Tinnerholm

LINKÖPING UNIVERSITY

Agenda

- Hour I
  - Practice exercises for the exam in TDDD55
- Hour 2
  - Practice exercises for the exam in TDDD55
  - Questions and Lab demo
    - If time permits

Note for other readers of this PDF and ***not attending the Lesson***

➢ Some exercises have been simplified slightly

Exam Format 2023

- 30 Points
  - 15 Points to pass 50%
  - 10 points must be from the fundamentals section

- The rest of this presentation will be about examples from the 2015 and other exams

2023-01-14, 08:00 – 12:00

Hjälpmedel / Allowed material:

- Engelsk ordbok / Dictionary from/to English to/from your native language (book only; not electronic)

General instructions:

- Read the instructions and examination procedures for exams at LiU.
- Read all assignments carefully and completely before you begin.
- You may answer in Swedish or in English.
- Write clearly – unreadable text will be ignored. Be precise in your statements – imprecise formulations may lead to reduction of points. Motivate clearly all statements and reasoning. Explain calculations and solution procedures.
- The assignments are sorted into fundamentals (20p) and an advanced section (10p).
- The exam is designed for 30 points in 240 minutes. You may thus plan 8 minutes per point.
- Grading: U, 3, 4, 5.
- The preliminary threshold for passing (grade 3/C) is 15 points, of which 10 points should be from the fundamentals section.

Exercise 1: Formal Languages and Automata Theory

(TDDD55 only - 6p) **August 2015**

Consider the language **L** consisting of all strings **w** over the alphabet **{0, 1}** such that every string contains 00 once and 11 once. Example of strings in the language: 1001011, 1100.

Examples of strings **not** in the language: 0101, 10**11**, 010**11**0**11**, 0**00**. (The last example counts as two occurences of 00.)

(a) (2p) Construct a regular expression for L.

(b) (1.5p) Construct from the regular expression an NFA recognizing L.

(c) (2.5p) Construct a DFA recognizing L, either by deriving it from the NFA or by constructing it directly.

Solution proposal Exercise 1:

$\Sigma = \{0, 1\}$

$L_1 = \{w : w \text{ Contains } 00 \text{ once } \& \text{ } 11 \text{ once}\}$

a) Construct regular expression for L

$((1|\varepsilon)(01)^* 00 (10)^* 11 (01)^* (0|\varepsilon))$

$|\ ((0|\varepsilon)(10)^* 11 (10)^* 00 (10)^* (1|\varepsilon))$

Every string contains 00 once and 11 once

Solution proposal Exercise 1 continued

- Intuition behind the regular expression
- Essentially two paths
  - One path where we start with either a **1** followed by an infinite sequence of **01** until we reach **00**. Since we can not allow **01** after **00** we have an infinite sequence of **10** until we reach **11** then we repeat the structure
- Note the other regular expression
  - This regular expression is essentially the same sequence as above but with the pattern in reverse

Regular expression

$(1|\varepsilon)(01)^*00(10)^*11(01)^*(0|\varepsilon)$

$(0|\varepsilon)(10)^*11(01)^*00(10)^*(1|\varepsilon)$

Some tests:

0011 Matches regular expression #1

1100 Matches regular expression #2

Solution proposal Exercise 1 continued (1.5p) Construct from the regular expression an NFA recognizing L.

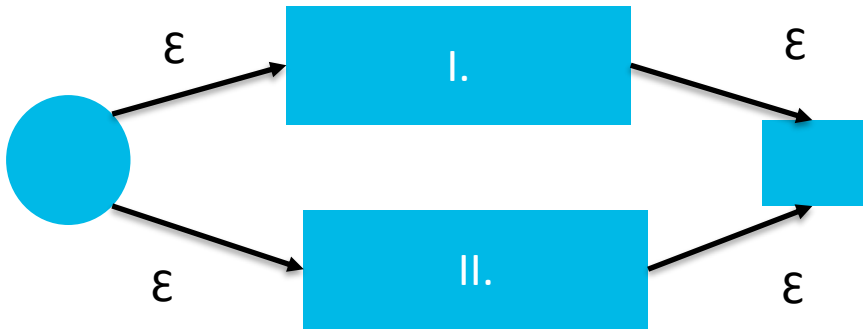
Our regular expressions:

I. (1|ε)(01)*00(10)*11(01)*(0|ε)

II. (0|ε)(10)*11(01)*00(10)*(1|ε)

The full expression can be expressed as

(**I** | **II**) ε



- For #1 we use I.
- For #2 we use II.

- Note here we use a template for how to construct an NFA for a "|" expression

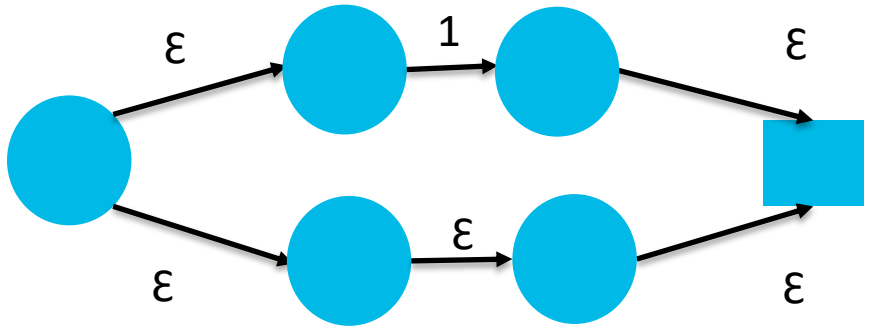
Solution proposal Exercise 1 continued

- Note for the first box we expand the contents of our first regular expression

$$((1|\varepsilon)(0\,1)^*\,00\,(1\,0)^*\,11\,(0\,1)^*\,(0|\varepsilon))$$



- Note we construct a subgraph using the Thompson construction for | for the first part of the **NFA**
  - (1| ε)

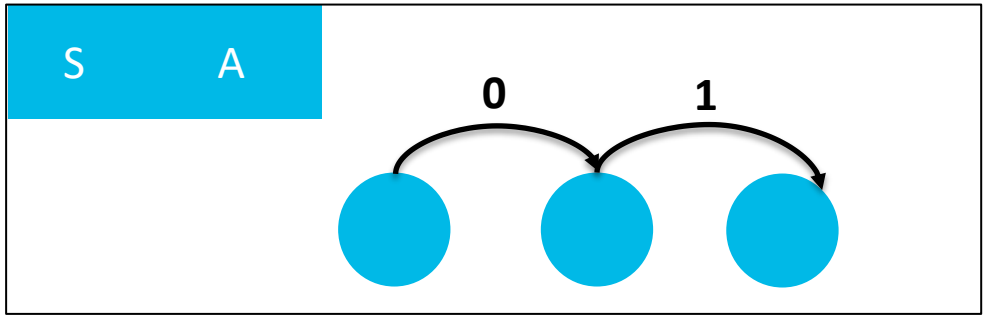


Solution proposal Exercise 1 continued

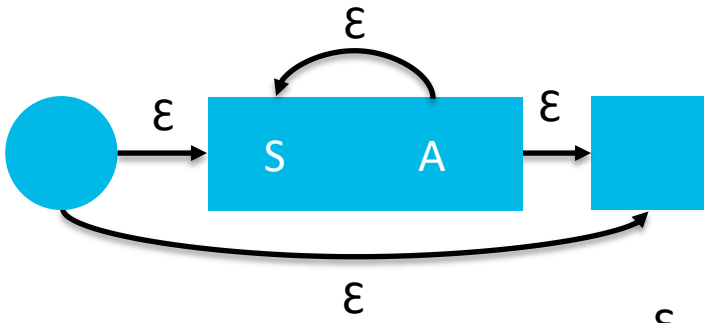
- Note for the first box we expand the contents of our first regular expression

$$((1|\varepsilon)\boxed{(01)^*}00(10)^*11(01)^*(01|\varepsilon))$$

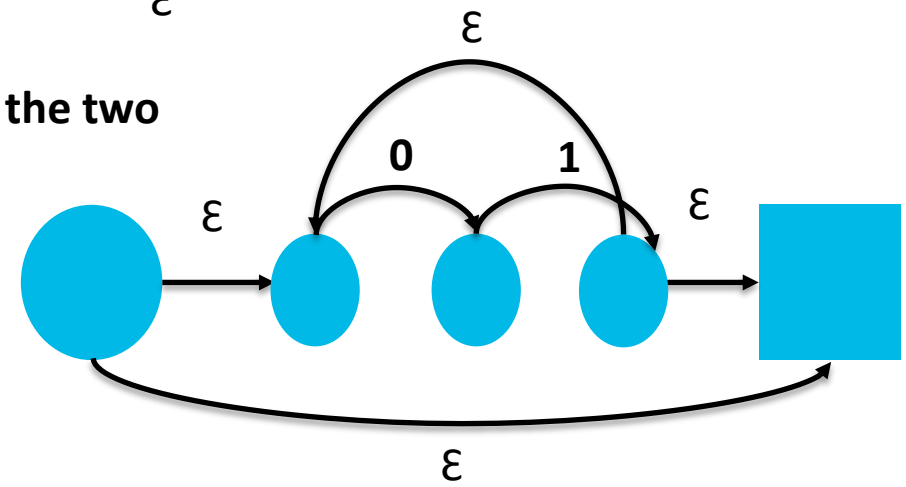
- **I. 0 Concatenated with 1**



- **II. The rule for the Kleene star $(01)^*$**

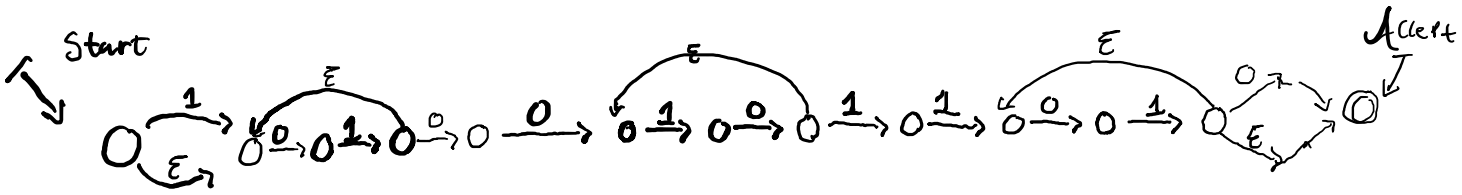


- **III. Combining the two**

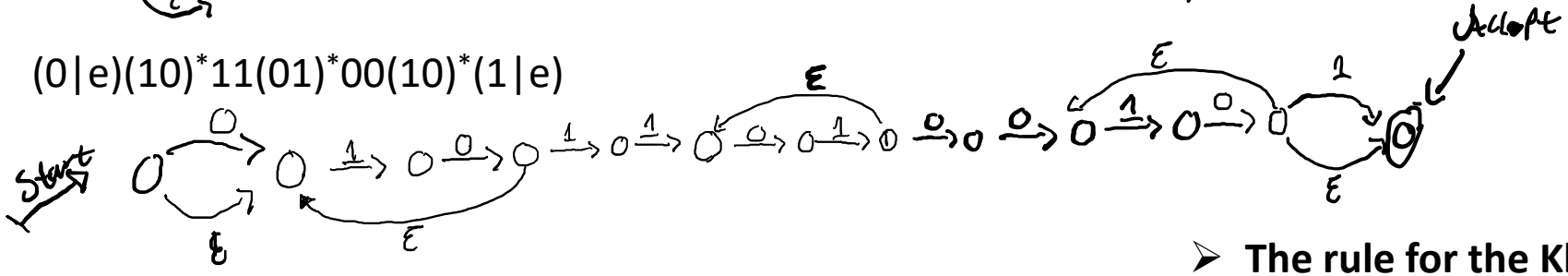


Solution proposal Exercise 1 continued (1.5p) Construct from the regular expression an NFA recognizing L.

I. (1|e)(01)*00(10)*11(01)*(0|e)



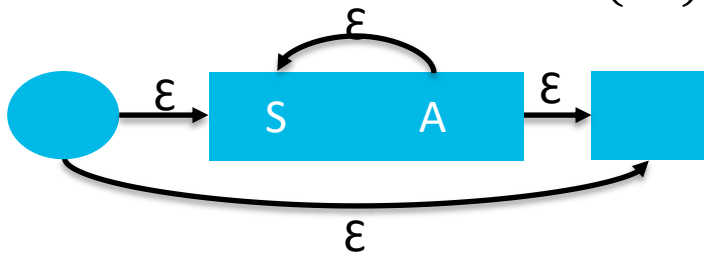
(0|e)(10)*11(01)*00(10)*(1|e)



This is slightly wrong, what is wrong?

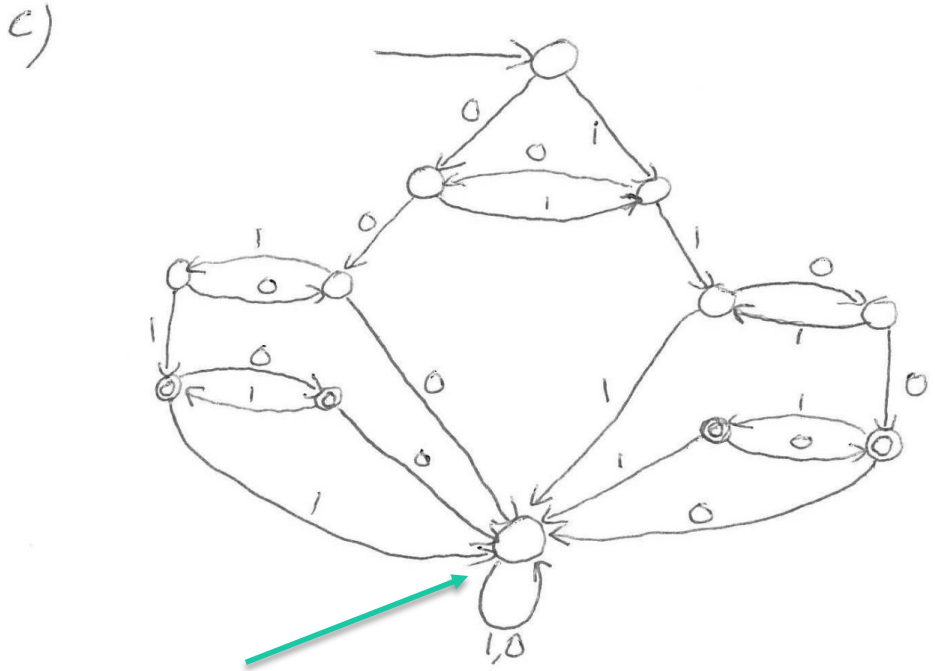
Hint: Think about the Kleene star

- **The rule for the Kleene star $(01)^*$**



Exercise 1) C

(c) (2.5p) Construct a DFA recognizing L, either by deriving it from the NFA or by constructing it directly.



- **A standard way of converting a NFA into a DFA can be done using Powerset construction:**

  Powerset construction - Wikipedia

- **(Note you need not use Thompsons algorithm to construct the NFA. However, it might be useful to learn some tricks from it)**

- ***Also note that the DFA is a special case of the NFA. That is, the NFA is a generalization. Hence, this DFA could also be the answer to question B***

Exercise 2

Exercise 2: Compiler Structure and Generators

(4p)

(a) (1p) What are the advantages and disadvantages of a multi-pass compiler (compared to an one-pass compiler)?

(b) (3p) Describe briefly what phases are found in a compiler. What is their purpose, how are they connected, what is their input and output?

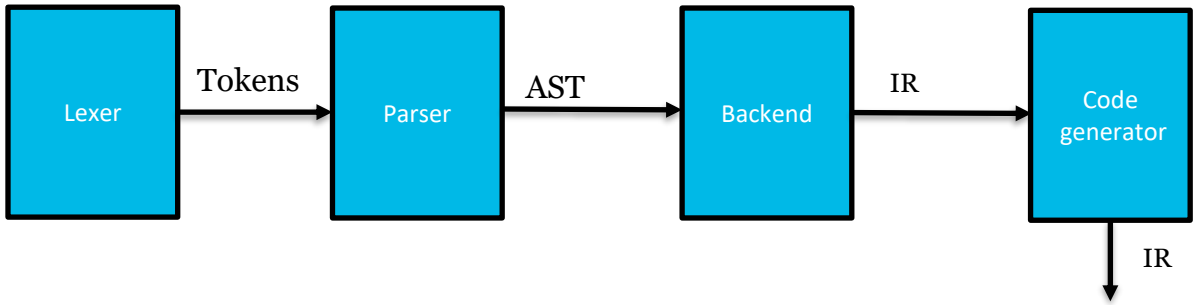
Solution

(a) (1p) What are the advantages and disadvantages of a multi-pass compiler (compared to an one-pass compiler)?

- Wider scope, better code generation better optimization
- Some languages require multi-pass compilers
  - For instance, Java **(Contrast how you work with Java with how you would work with a language such as C++/C)**
- The design is more modular

(b) (3p)
Describe briefly what phases are found in a compiler. What is their purpose, how are they connected, what is their input and output?

- The difference phases you have been doing in the lab. Explain with some details (Draw a figure)



Exercise 3 (Left out at Lesson 2024)

Exercise 3: Symbol Table Management

Describe what the compiler — using a symbol table implemented as a **hash table with chaining** and **block scoped control** — does in compiling a statically scoped, block-structured language when it handles:

(a) (0.5p) block entry

(b) (0.5p) block exit

(c) (0.5p) a variable declaration

(d) (0.5p) a variable use.

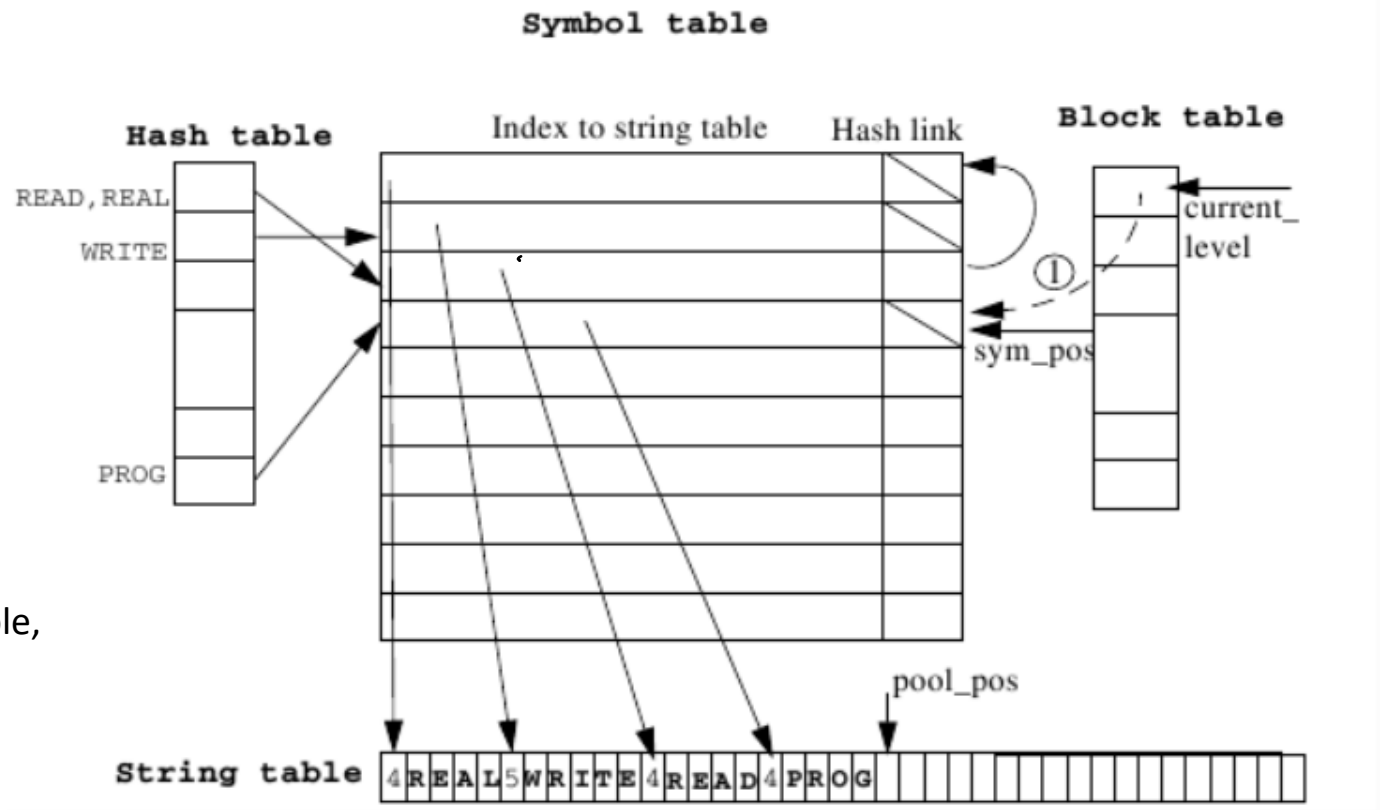
Exercise 3 Solution (A), Block Entry

Hashtable with chaining
See lecture IV

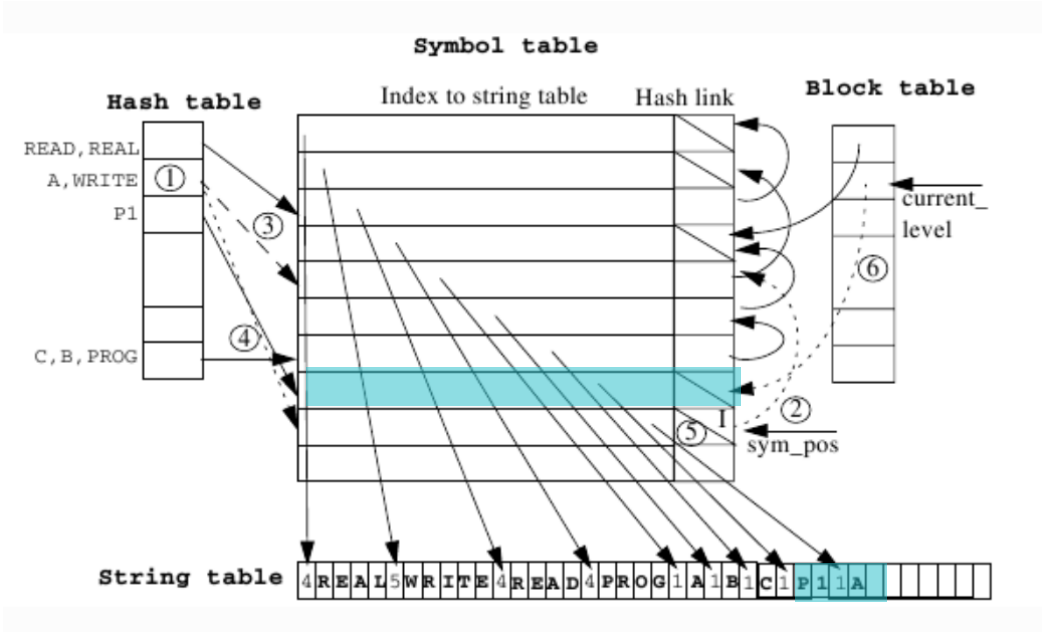
```
program prog;
  var a : integer; b : integer; c : integer;
  procedure p1;  // Opens a scope
    var a : integer;
  begin
    c := b + a;
  end;
begin
  c := b + a;
end.
```

Answer:

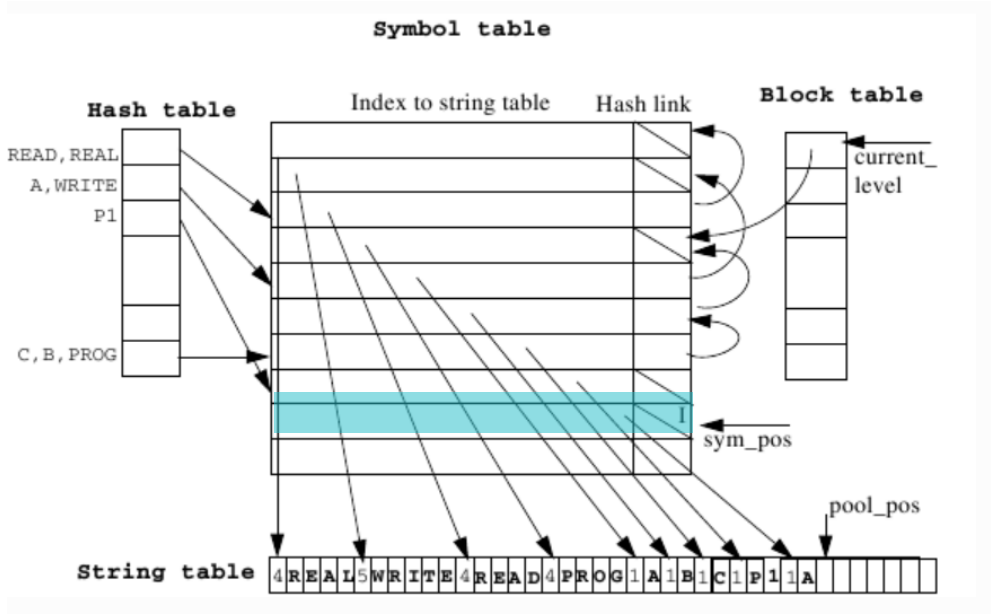
- Copy the current entry in the block table, one entry down. Increase the current block
- sym_table[block_table[current_level]]



Exercise 3 Solution (B), Block Exit



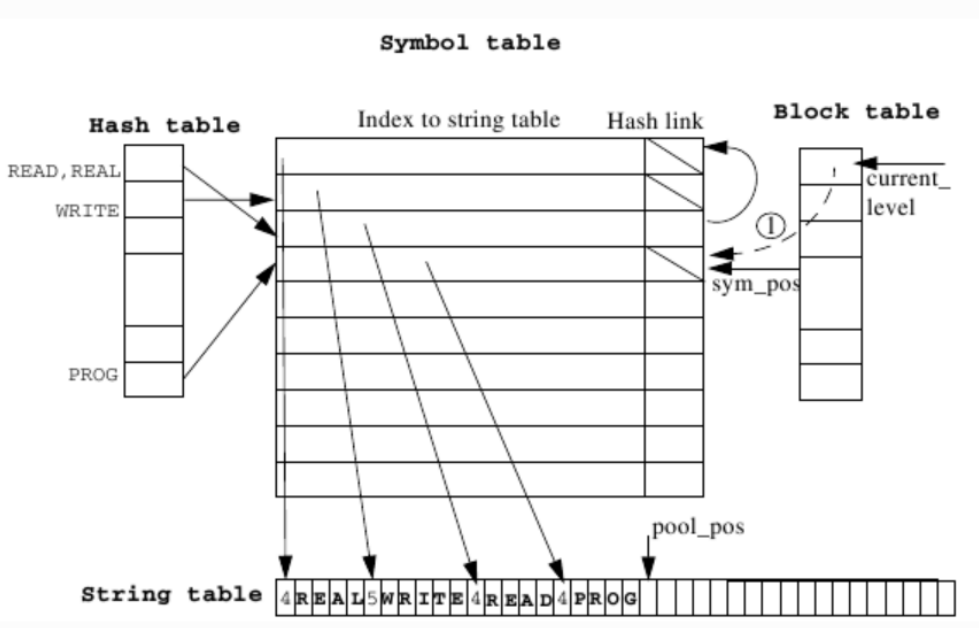
A dotted arrow means that the link is removed in this picture:	·····►
A dashed arrow means that the link is added in this picture:	- - ►
A normal (filled) arrow means that the link was added in any previous picture:	ture:——►

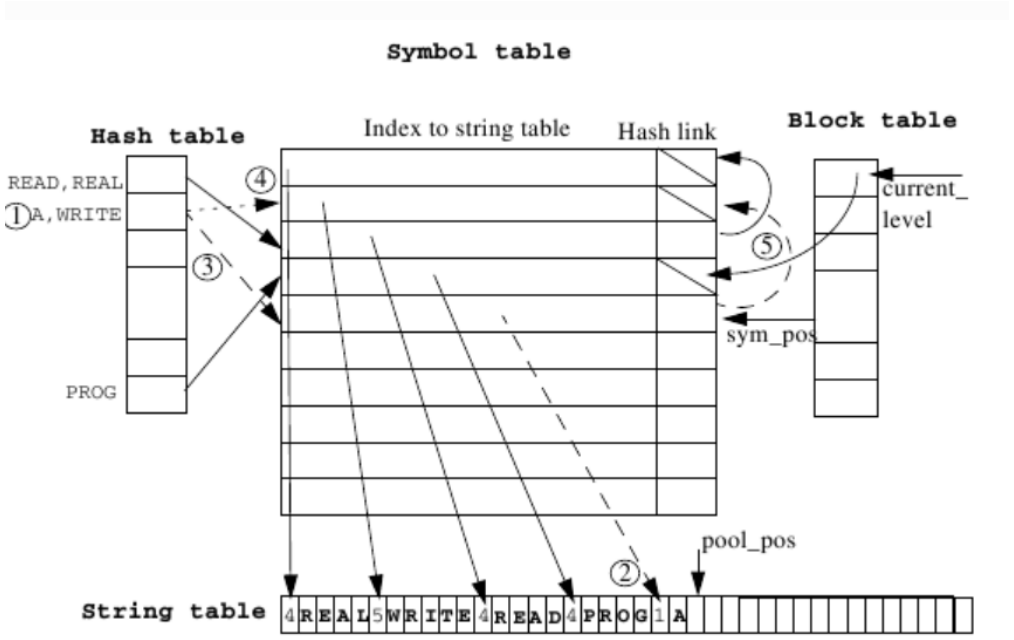


Answer:

- Decrease the current block. For all entries in the table between the old and the new current block, Point the entries in the hash table to the hash link in that entry

Exercise 3 Solution (C), Variable Declaration



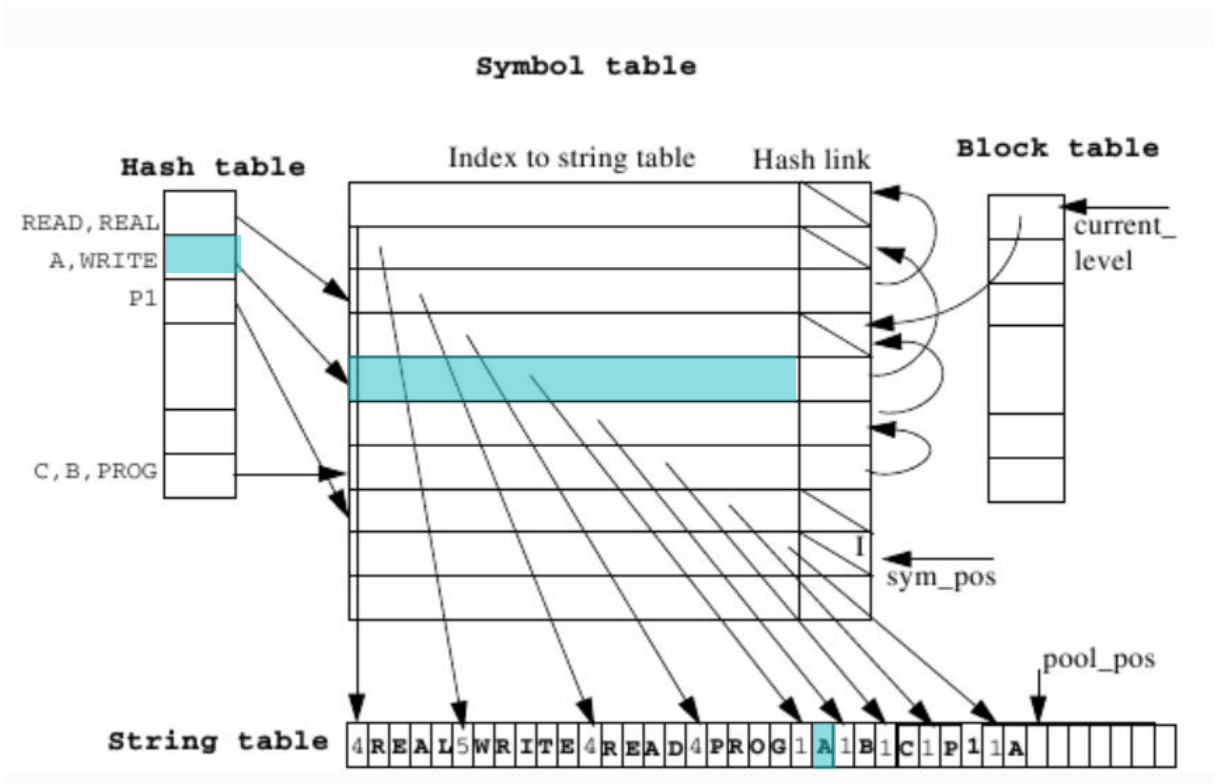


Answer:
Find the entry in the hash table (Hash it) **(1)**. If one exists see if it is in the current block.
If not add a new entry to the symbol table **(2)** and point its hash link to the previous one **(5)**
Increase the current entry in the block table **(sym_pos)**

A dotted arrow means that the link is removed in this picture:	·····►
A dashed arrow means that the link is added in this picture:	- - ►
A normal (filled) arrow means that the link was added in any previous picture:	ture:——►

Exercise 3 Solution (D), Variable Use

```
program prog;
  var a : integer; b : integer; c : integer;
  procedure p1;
    var a : integer;
  begin
    c := b + a;
  end;
begin
  c := b + a;
end.
```



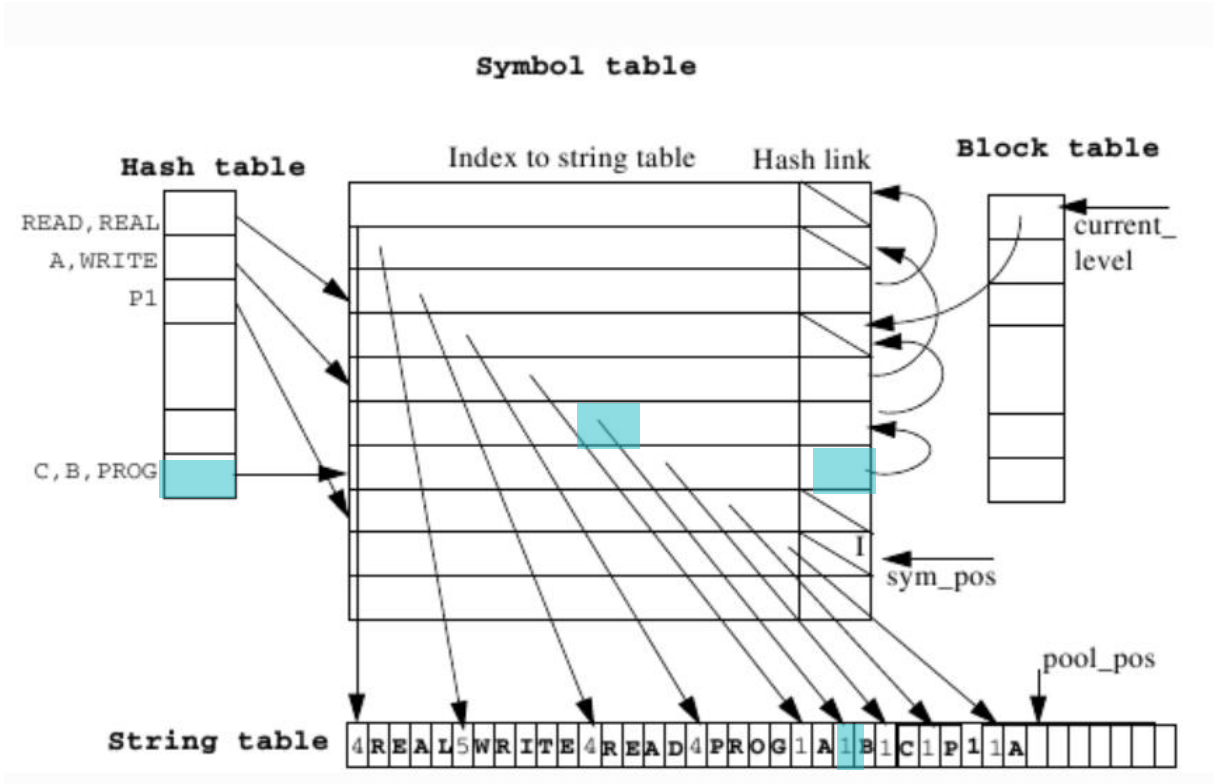
Answer:

- Look up the variable in the hash table follow the pointer to the symbol table
- **In this example we are looking up the variable a in the global scope**

A dotted arrow means that the link is removed in this picture:	·····►
A dashed arrow means that the link is added in this picture:	- - ►
A normal (filled) arrow means that the link was added in any previous picture:	ture:——►

Exercise 3 Solution (D), Variable Use

```
program prog;
  var a : integer; b : integer; c : integer;
  procedure p1;
    var a : integer;
  begin
    c := b + a;
  end;
begin
  c := b + a;
end.
```



Answer:

- Look up the variable in the hash table follow the pointer to the symbol table
- **In this example we are looking up the variable b in the global scope**

A dotted arrow means that the link is removed in this picture:	·····►
A dashed arrow means that the link is added in this picture:	- - ►
A normal (filled) arrow means that the link was added in any previous picture:	ture:——►

Exercise 4

Exercise 4: Top-Down Parsing

(a) (4.5p) Given a grammar with nonterminals <S>, <P>, <Q> and the following productions:

<S> ::= <S> u
| <P> <Q> v
| <P> <Q> w
<P> ::= <P> x
| y
<Q> ::= <Q> z
| e

where S is the start symbol, {u, v, w, x, y, z} are terminals. (e is the empty string!)

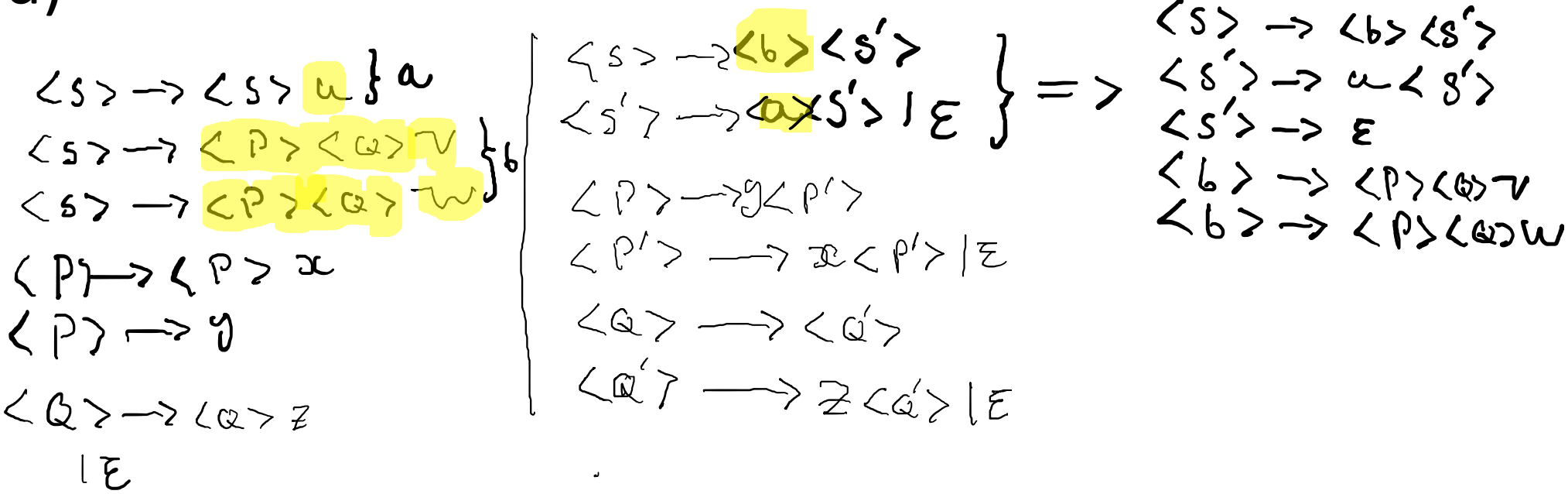
What is/are the problem(s) with this grammar if it is to be used for writing a recursive descent parser with a single token lookahead?

Resolve the problem(s) and write a recursive descent parser for the modified grammar. (Pseudocode/program code without declarations is fine. Use the function scan() to read the next input token, and the function error() to report errors if needed.)

(b) (0.5p) The theory for formal languages and automata says that a stack is required for being able to parse context-free languages. We have used such a stack, for instance, in the LL-item pushdown automaton in the lecture on top-down parsing. But where is the corresponding stack in a recursive descent parser?

Exercise 4 a)

S ::= S **u**
| P Q **v**
| P Q **w**
P ::= P **x**
| **y**
Q ::= Q **z**
| e



A → Aa | b
=>
A → bA’
A’ → aA’ | e

Exercise 4 a)

<S> -> <b><S'>
<S'> -> u<S’>
<S'> -> e

<P> -> y<P’>
<P'> -> x<P'>
<P'> -> e

<b> -> <P><Q>v
<b> -> <P><Q>w

<Q> - > <Q'>
<Q'> -> z<Q'>
<Q'> -> e

- Note that the derivation in the proposed solution differs slightly from my proposal
- e is epsilon

S rewritten:

$S \rightarrow Su \mid S_1$
$S_1 \rightarrow PQv \mid PQw$

Q rewritten:

$Q \rightarrow Q'$
$Q' \rightarrow zQ' \mid \varepsilon$

simplify

$Q \rightarrow zQ \mid \varepsilon$

$S \rightarrow S_1 S_2$
$S_2 \rightarrow u S_2 \mid \varepsilon$
$S_1 \rightarrow PQ S_3$
$S_3 \rightarrow v \mid w$

Exercise 4.a: Write a parser based on this grammar

- Live demo
- Pseudocode is okay!
- For exercise b where is the stack?
  - Hint what is needed for recursion?

Some valid strings:

yvu

yxvu

…

```
<S> ::= <S> u
    | <P> <Q> v
    | <P> <Q> w
<P> ::= <P> x
    | y
<Q> ::= <Q> z
    | e
```

Exercise 4:a Julia program implementing the proposed Parser

Live demo

Exercise 4:a Julia program implementing the proposed Parser

```
#Extra function can be ignored
global currentToken = ""

# <S> -> <b><S'>
# <S'> -> u<S’>
# <S'> -> ϵ
#<b> -> <P><Q>v
#<b> -> <P><Q>w

#<P> -> y<P’>
#<P'> -> x<P'>
#<P'> -> e

#<Q> - > <Q'>
#<Q'> -> z<Q'>
#<Q'> -> e
```

```
function scan()
  res = read(stdin, Char)
  if (res == "\n")
    println("Done!")
  end
  global currentToken = string(res)
end
```

```
function S()
  println("S->")
  b()
  SPrime()
end

function b()
  println("b->")
  P()
  Q()
  if currentToken == "v" || currentToken == "w"
    scan()
    return
  else
    throw("Expected v or w in SPrime token was $currentToken")
  end
end

function SPrime()
  println("SPrime->")
  if currentToken == "u"
    scan()
    SPrime()
  end
  #Epsilon
  return
end
```

```
function P()
  println("P->")
  if currentToken != "y"
    throw("Error expected y at P() token was $currentToken")
  end
  global currentToken = scan()
  PPrime()
end

function PPrime()
  println("PPrime->")
  if currentToken == "x"
    scan()
    PPrime()
  end
  #Epsilon
  return
end

function Q()
  println("Q->")
  QPrime()
end

function QPrime()
  println("QPrime->")
  if currentToken == "z"
    scan()
    QPrime()
  end
  #ϵ
end
```

```
function main()
  println("Enter expression  => ")
  scan()
  S()
end

main()
```

Exercise 5

Exercise 5: (TDDD55 only - 6p) LR parsing (Note 2020 Exam)

(a) (3p) Use the SLR(1) tables below to show how the string **aaaaadcbbbb** is parsed. You should show, step by step, how stack, input data etc. are changed during the parsing. Start state is 00, start symbol is S.

(b) (3p) Explain the concept of conflict in LR parsing — what it is and how it could be handled.

Exercise 5: Solution (a)

- To big to fit on one slide, I will share an external program ☺
- The following slides will keep the solution so you can look at it at home

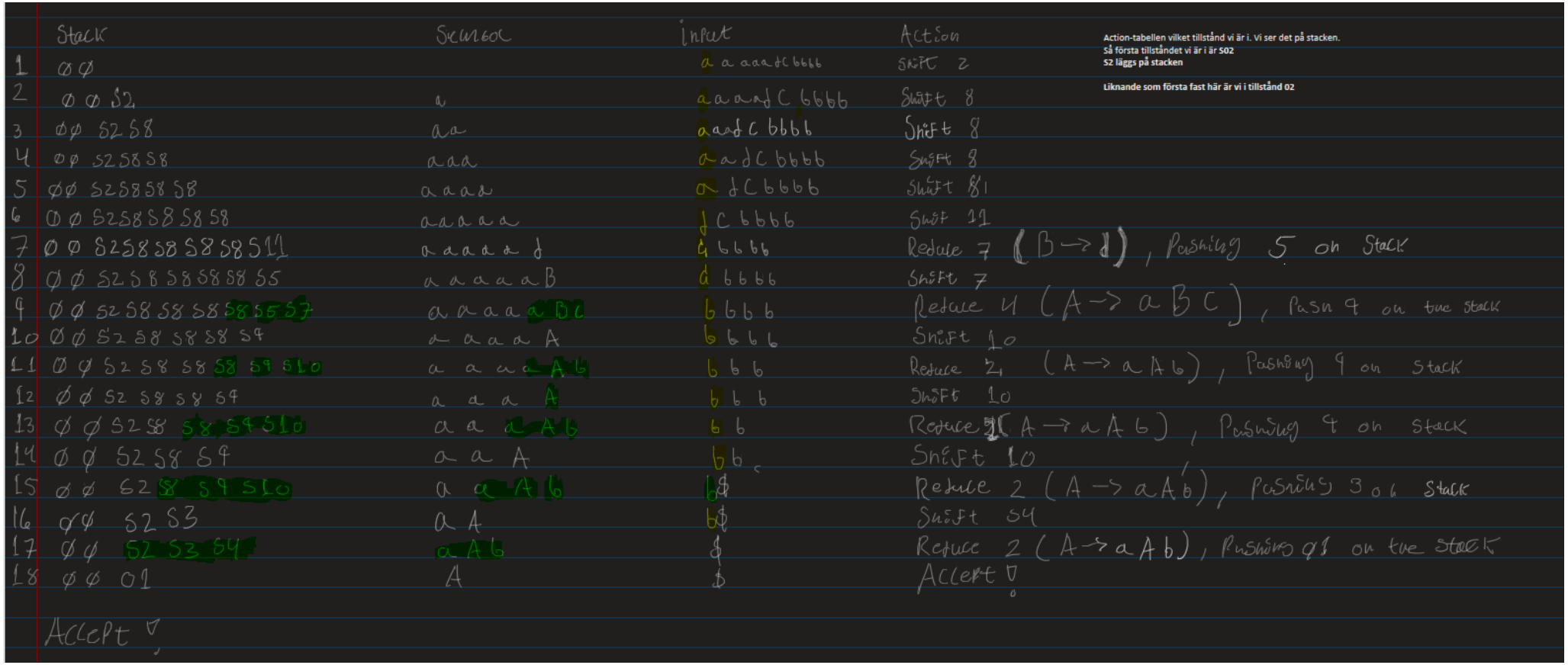
Exercise 5: **The grammar and the table**

1. S ::= A
2. A ::= a A b
3. | a B a
4. | a B c
5. B ::= a A b
6. | b A d
7. | d

Tables:

State	Action $	a	b	c	d	GOTO S	A	B
00	*	S02	*	*	*	*	01	*
01	A	*	*	*	*	*	*	*
02	*	S08	S12	*	S11	*	03	05
03	*	*	S04	*	*	*	*	*
04	R2	*	R2	*	R2	*	*	*
05	*	S06	*	S07	*	*	*	*
06	R3	*	R3	*	R3	*	*	*
07	R4	*	R4	*	R4	*	*	*
08	*	S08	S12	*	S11	*	09	05
09	*	*	S10	*	*	*	*	*
10	R2	R5	R2	R5	R2	*	*	*
11	*	R7	*	R7	*	*	*	*
12	*	S02	*	*	*	*	13	*
13	*	*	*	*	S14	*	*	*
14	*	R6	*	R6	*	*	*	*

Slide to look at at home



Solution

(b) (3p) Explain the concept of conflict in LR parsing — what it is and how it could be handled.

$E \rightarrow E + E$

$E \rightarrow E * E$

Input: E + E * E

- A conflict is when we want to write to two different shift/reduce in the same cell in a table. This occurs because of ambiguity. The problem can be solved by rewriting the grammar or using a more **Powerful LR parser/more lookahead, some examples:** LALR or GLR

Exercise 6 is for **TDDB44 only**

Exercise 7 Syntax Directed Translation

Exercise 5: (5p) Syntax-Directed Translation

A Pascal-like language is extended with a restartblock statement according to the following grammar:

```
<block>     ::= begin <stmt_list> end
<stmt_list> ::= <stmt_list><stmt> |
<stmt>      ::= <assignment> | ... | restartblock
```

(where "..." represents all other possible kinds of statements). restartblock means that execution restarts at the beginning of the immediately enclosing block.

Example:

```
begin
    x:=17;
L1: begin
        y:=y-42;
        if p=4711
L2:        then restartblock;
           else q:=q-1;
L3: end;
end;
```

where restartblock at L2 means a jump to L1 (i.e. the beginning of the enclosing block)

Exercise 7: (5p) Syntax-Directed Translation (a)

```
<block>     ::= begin <stmt_list> end
<stmt_list> ::= <stmt_list><stmt> |
<stmt>      ::= <assignment> | ... | restartblock
```

(a) (4p)
Write a syntax-directed translation scheme, with attributes and semantic rules, for translating s, and restartblocks inside them, to quadruples. The translation scheme should be used during bottom-up parsing. You are not allowed to define and use symbolic labels**, i.e. all jumps should have absolute quadruple addresses as their destinations.**
You may need to rewrite the grammar.
Explain all the attributes, functions, and instructions that you introduce**. State all your assumptions**.
(Since it is a syntax-directed translation scheme, not an attribute grammar, generation of a quadruple puts it in an array of quadruples and attribute values are ”small” values such as single quadruple addresses.)

Exercise 7: (5p) Syntax-Directed Translation (b)

(b) (1p)
What problem would occur in handling of the translation scheme if instead of restartblock there would be an exitblock statement that jumped to the end of the immediately enclosing block (instead of the begin), i.e. to L3 in this example?

Example:

```
begin
    x:=17;
L1: begin
        y:=y-42;
        if p=4711
L2:         then restartblock;
            else q:=q-1;
L3: end;
end;
```

Exercise 7: (5p) Syntax-Directed Translation (b)

- a)

<block> ::= <block_start> <stmt_list> ***end***
{
pop(block_starts)
}
<block_start> ::= ***begin***
{
push(block_starts, quads, count)
}
<stmt> ::= ***restartblock***
{
gen_quad(q_jmp, 0, 0, block_starts.top)
}

- The first assumption made here is that the other rules in the syntax directed translation scheme has been defined, **that is other rules for other statements in the language**
- The second assumption **is that we have a stack that contains the quadruple addresses of each block**
- The top element of the stack is assumed to be accessed by using **block_starts.top**

Here the translation scheme works in the following fashion.
A block is encountered, then the rules for <block_start> is evaluated
The result is that the existing quads are pushed to the block_starts stack. Then the <stmt_list> list is evaluated using existing rules assumed to exist in the program during the processing of <stmt_list> if a <stmt> contains the keyword restart block that we augmented the language with the use our augmented rule:
<stmt> ::= restartblock
In this rule we generate a new quad for a jmp we the absolute address being the current element at the top of the stack **block_starts**

Once this is done we finalize the rule for block by popping the stack! Since the solution is stack based we will always handle restart at the right block

Exercise 7: (5p) Syntax-Directed Translation (b)

- b)

<block> ::= <block_start> <stmt_list> ***end***

```
{
        pop(block_starts)
}
```

<block_start> ::= ***begin***

```
{
        push(block_starts, quads, count)
}
```

<stmt> ::= ***restartblock***

```
{
        gen_quad(q_jmp, 0, 0, block_starts.top)
}
```

Example:

```
begin
    x:=17;
L1: begin
        y:=y-42;
        if p=4711
L2:        then restartblock;
           else q:=q-1;
L3: end;
end;
```

b) We would need a structure containing all exit blocks in each block so we can update the offsets in them when <block> is reduced.

Exercise 8 (Included but will not be discussed 2024)

Exercise 8: (3p) Error Handling

Explain, define, and give examples of using the following concepts regarding error handling:

(a)(1p) Valid prefix property
(b) (1p) Phrase level recovery,
(c) (1p) Global correction.

- Detailed answered provided in the solution proposal
- **See also Lecture 13 (And the course book)**

Exercise 9 (Included, theoretical answer)

9. (3p) Memory management

(a) (1p) Non-local references: How does a static link work?
(b) (1p) Non-local references: How does a display work?
(c) (1p) Dynamic data: How is the actual size and contents of a dynamic array handled?

- Detailed answered provided in the Solution proposal
  - Lab 7: Memory Management and Intel Assembler — TDDB44 Compiler Construction documentation (liu.se) (For some more details)
    - **See also Lecture 9**

Exercise 10

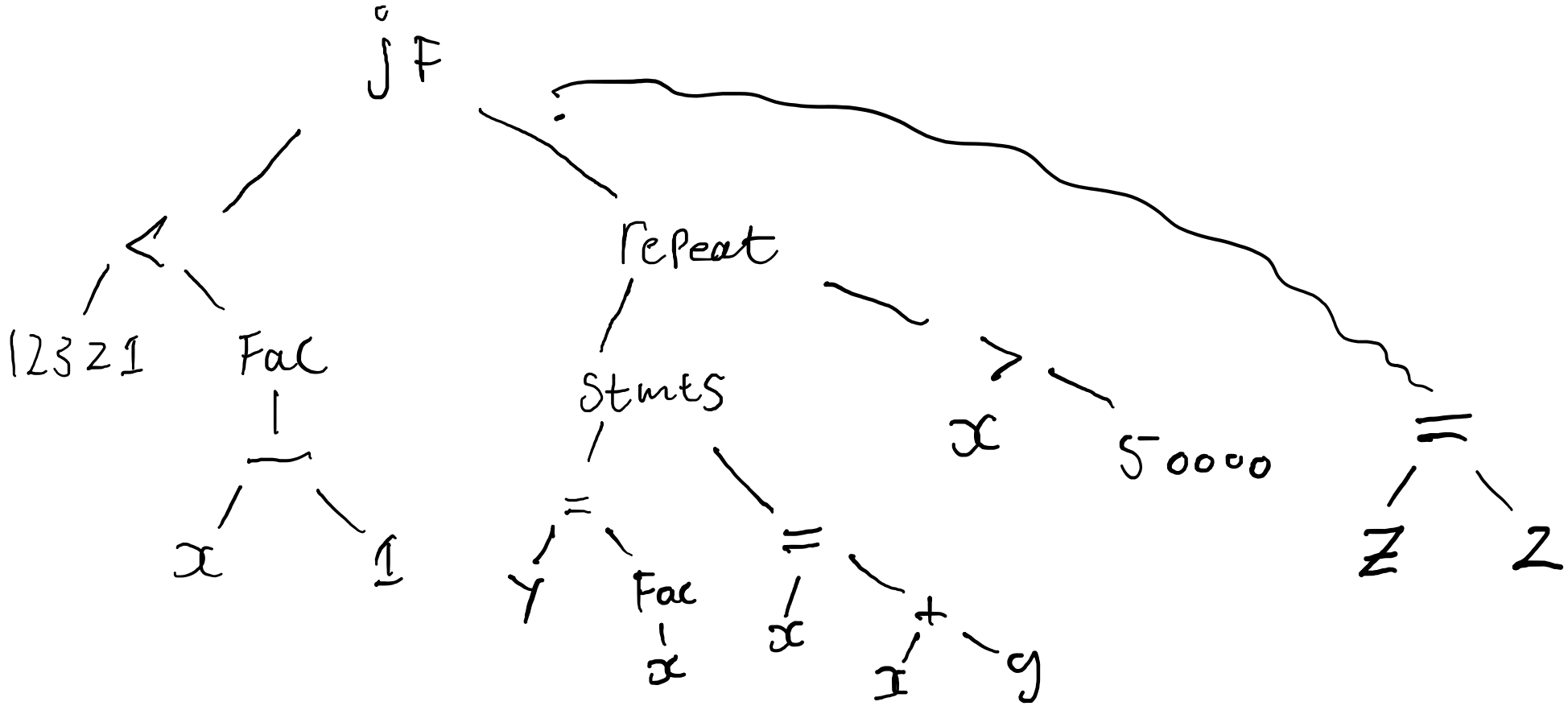
10. (6p) Intermediate Code Generation (A)

(a) (3p) Given the following code segment in a Pascal-like language:

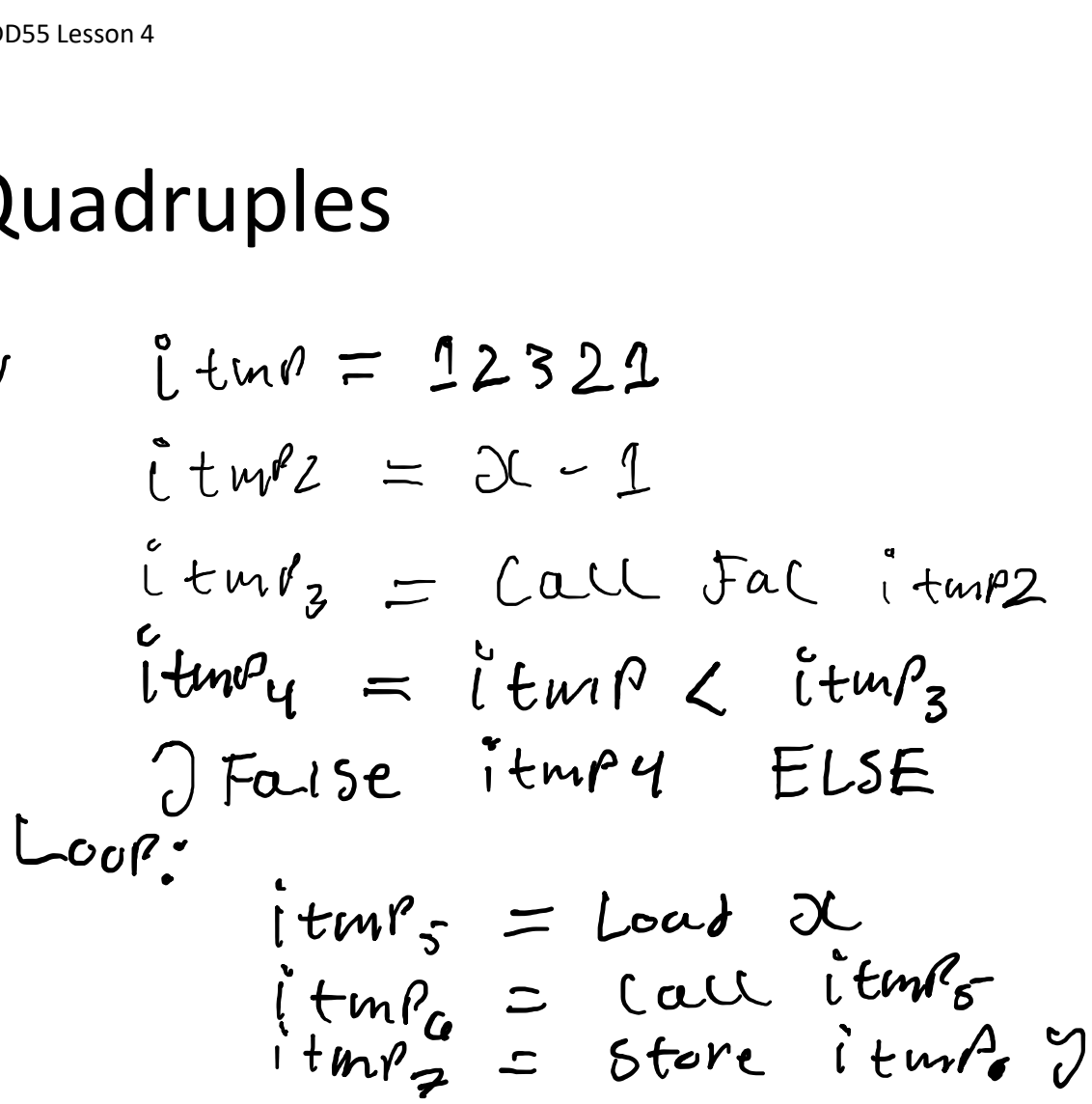
```
if 12321<fac(x-1)
   then repeat
          y=fac(x);
          x=x+y;
        until x>50000
   else z=2
```

Translate the code segment into an abstract syntax tree, quadruples, and postfix code.

Syntax tree



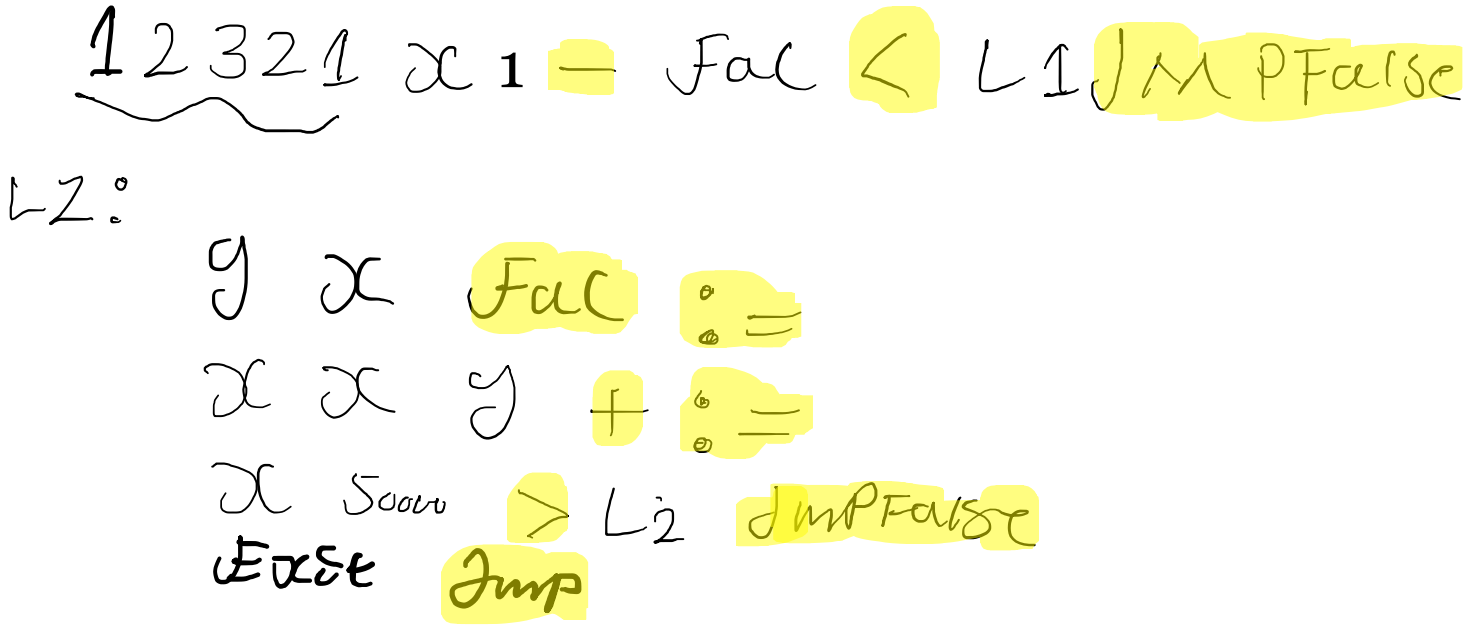
Quadruples



Conceptual, some loads and stores are omitted see complete solution on the course webpage

(Here I have used pseudo quads for speed and space reasons)

Postfix code



Think about this as a language with different operators and without if-else/loop constructs

Exercise 10 B

10. (6p) Intermediate Code Generation (B)

(b) (3p) Divide the following code into basic blocks, draw a control flow graph, and show as well as motivate the existing loop(s).

```
    goto L2
L1: x:=x+1
L2: x:=x+1
    x:=x+1
    if x=1 then goto L1
L3: if x=2 then goto L4
    goto L5
L4: x:=x+1
L5: x:=x+1
    if x=4 then goto L1
```

Basic Blocks

Solution

