Compiler construction in4303 – answers

Koen Langendoen

Delft University of Technology
The Netherlands

Compiler construction
in4303 – lecture 1

Introduction

Lexical analysis by hand

Chapter 1 – 2.15

AST exercise (5 min.)

- expression grammar

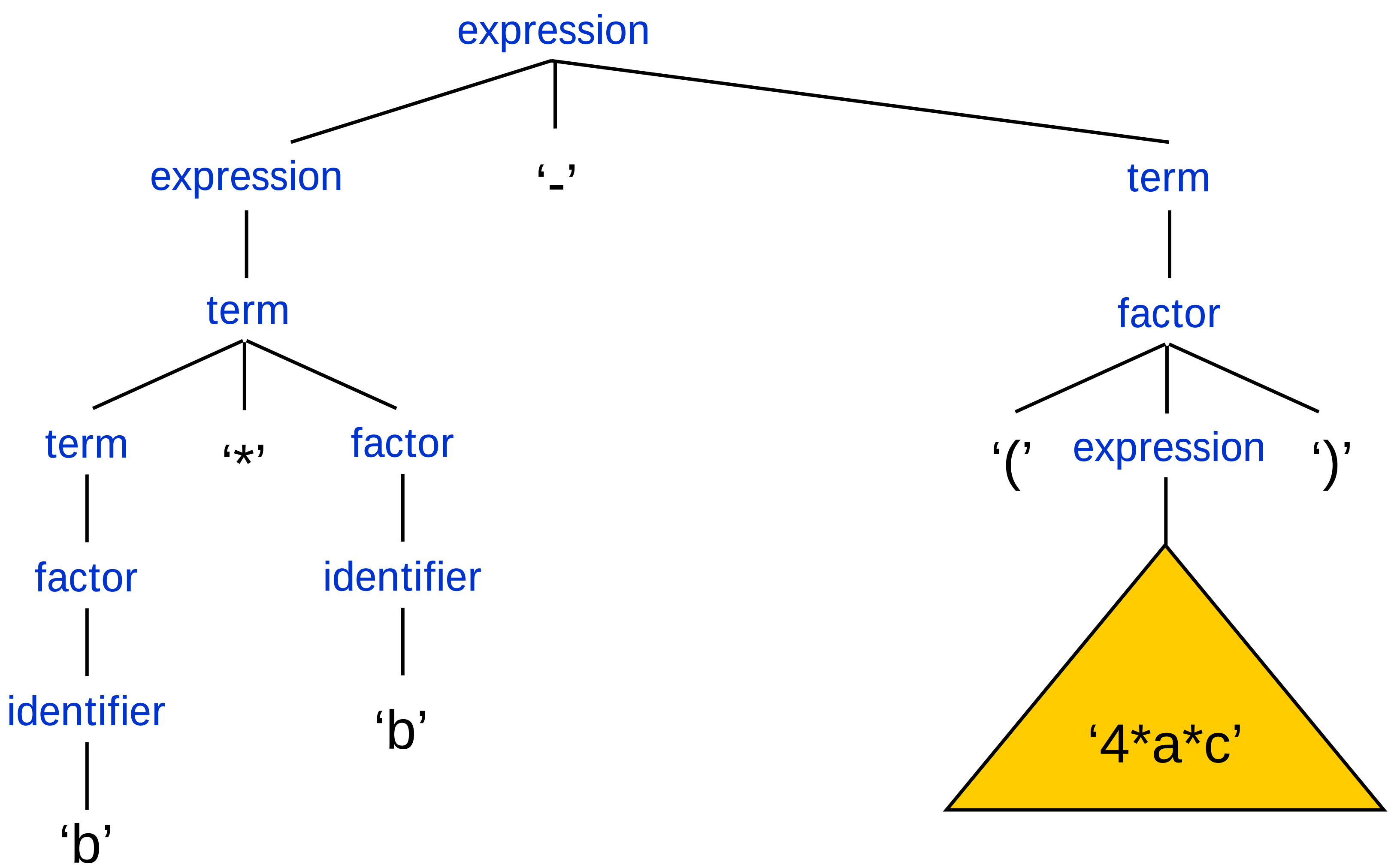
  expression → expression '+' term | expression '-' term | term
  term → term '*' factor | term '/' factor | factor
  factor → identifier | constant | '(' expression ')'

- example expression

  b*b – (4*a*c)

- draw parse tree and AST

answer
parse tree: b*b – (4*a*c)



Exercise (5 min.)

- write down regular descriptions for the following descriptions:
  - an integral number is a non-zero sequence of digits optionally followed by a letter denoting the base class (b for binary and o for octal).
  - a fixed-point number is an (optional) sequence of digits followed by a dot ('.') followed by a sequence of digits.
  - an identifier is a sequence of letters and digits; the first character must be a letter. The underscore _ counts as a letter, but may not be used as the first or last character.

Answers

base → [bo]

integral_number → *digit+ base?*

dot → [.]

fixed_point_number → *digit* dot digit+*

letter → [a-zA-Z]

digit → [0-9]

underscore → [_]

letter_or_digit → *letter* | *digit*

letter_or_digit_or_und → *letter_or_digit* | *underscore*

identifier → *letter (letter_or_digit_or_und* letter_or_digit)?*

Compiler construction
in4303 – lecture 2

Automatic lexical analysis

Chapter 2.1.6 - 2.1.13

FSA exercise (6 min.)

- draw an FSA to recognize integers

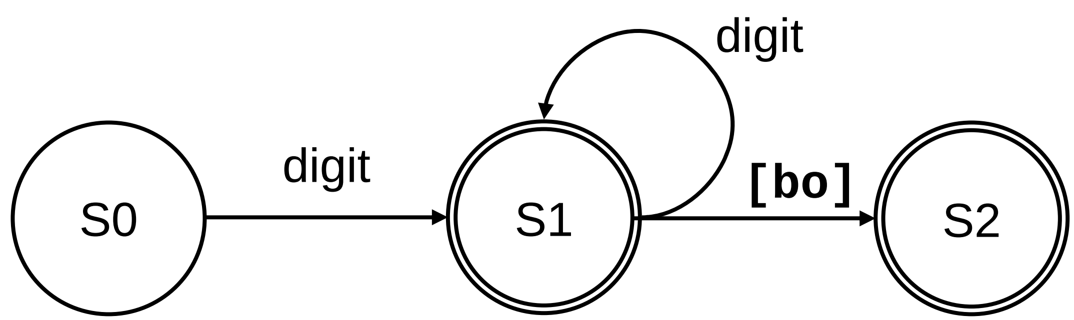
  $$base \rightarrow \text{[bo]}$$

  $$integer \rightarrow digit+\ base?$$

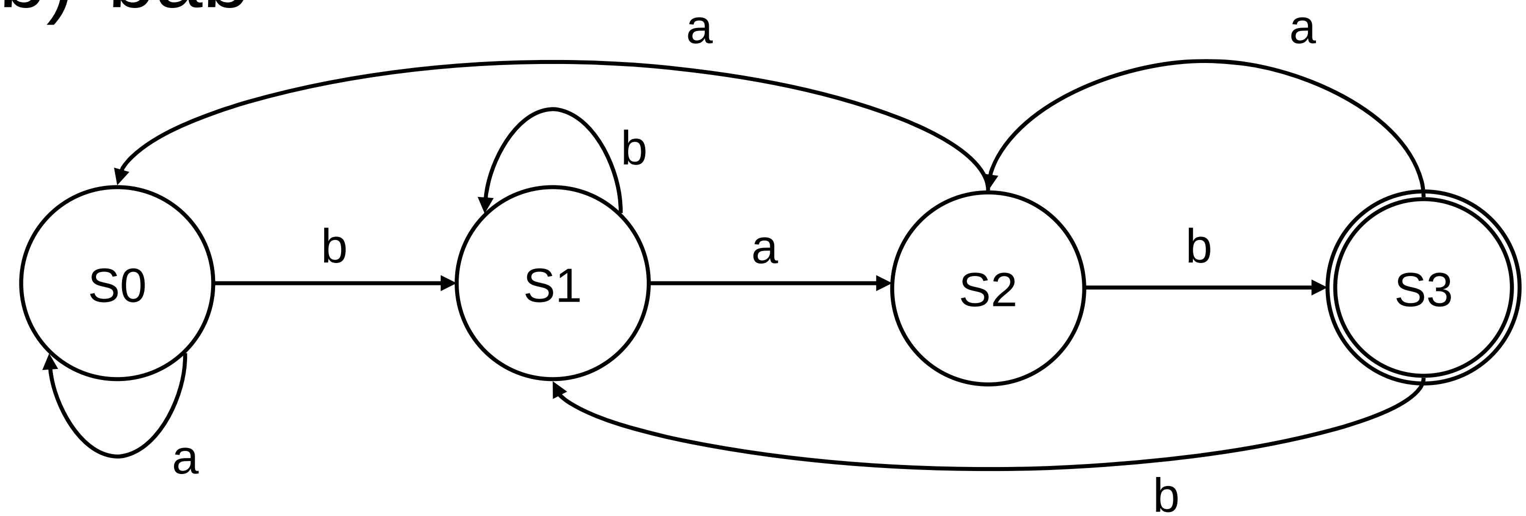
- draw an FSA to recognize the regular expression (a|b)*bab

Answers

- integer



- (a|b)*bab



Exercise
FSA construction (7 min.)

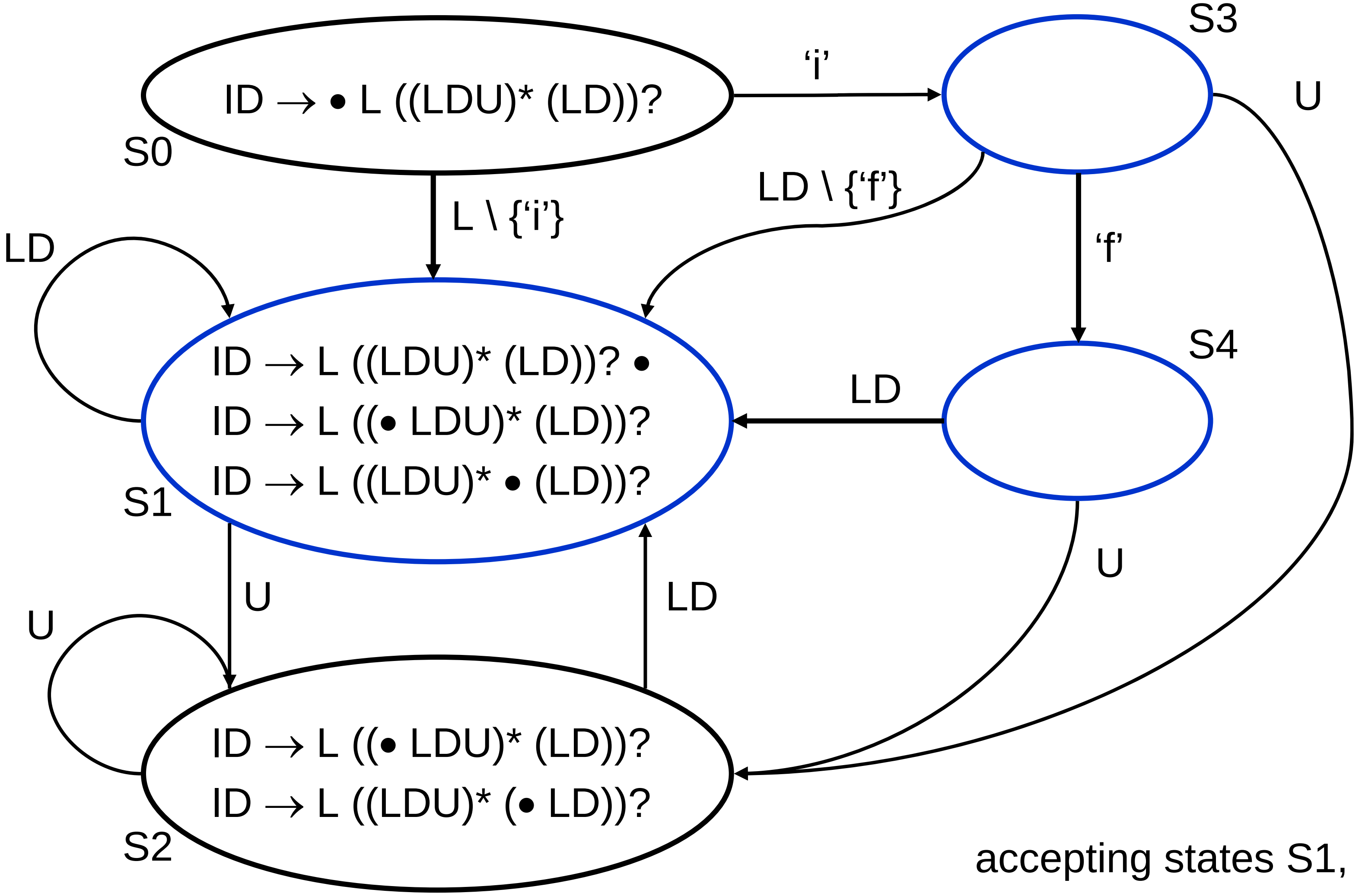
- draw the FSA (with item sets) for recognizing an identifier:

identifier → *letter* (*letter_or_digit_or_und** *letter_or_digit*)?

- extend the above FSA to recognize the keyword ‘if’ as well.

if → *‘i’ ‘f’*

Answers



accepting states S1, S3, and S4

Compiler construction
in4303 – lecture 3

Top-down parsing

Chapter 2.2 - 2.2.4

Exercise (4 min.)

- determine the FIRST set of the non-terminals in our expression grammar

input → expression EOF
expression → term rest_expression
term → IDENTIFIER | '(' expression ')'
rest_expression → '+' expression | ε

- and, for this grammar

S → A B
A → A a | ε
B → B b | b

Answers

- FIRST(input) = { IDENT, ‘(‘ }
  FIRST(expression) = { IDENT, ‘(‘ }
  FIRST(term) = { IDENT, ‘(‘ }
  FIRST(rest_expression) = { ‘+’, ε }

- FIRST(S) = { ‘a’, ‘b’ }
  FIRST(A) = { ‘a’, ε }
  FIRST(B) = { ‘b’ }

Exercise (4 min.)

- determine the FIRST set of the non-terminals in our expression grammar

$$
\begin{aligned}
&\text{input} \rightarrow \text{expression EOF} \\
&\text{expression} \rightarrow \text{term rest\_expression} \\
&\text{term} \rightarrow \text{IDENTIFIER} \mid \text{‘(’ expression ‘)’} \\
&\text{rest\_expression} \rightarrow \text{‘+’ expression} \mid \varepsilon
\end{aligned}
$$

- and, for this grammar

$$
\begin{aligned}
&S \rightarrow A\ B \\
&A \rightarrow A\ \text{‘a’} \mid \varepsilon \\
&B \rightarrow B\ \text{‘b’} \mid \text{‘b’}
\end{aligned}
$$

Answers

- FIRST(input) = { IDENT, ‘(‘ }
  FIRST(expression) = { IDENT, ‘(‘ }
  FIRST(term) = { IDENT, ‘(‘ }
  FIRST(rest_expression) = { ‘+’, ε }

- FIRST(S) = { ‘a’, ‘b’ }
  FIRST(A) = { ‘a’, ε }
  FIRST(B) = { ‘b’ }

Exercise (7 min.)

- make the following grammar LL(1)

  expression → expression '+' term | expression '-' term | term
  term → term '*' factor | term '/' factor | factor
  factor → '(' expression ')' | func-call | identifier | constant
  func-call → identifier '(' expr-list? ')'
  expr-list → expression (',' expression)*

- and what about

  S → if E then S (else S)?

Answers

- substitution

  F → ‘(‘ E ‘)’ | ID ‘(‘ expr-list? ‘)’ | ID | constant

- left factoring

  E → E (‘+’ | ‘-’) T | T

  T → T (‘*’ | ‘/’) F | F

  F → ‘(‘ E ‘)’ | ID (‘(‘ expr-list? ‘)’)? | constant

- left recursion removal

  E → T ((‘+’ | ‘-’) T)*

  T → F ((‘*’ | ‘/’) F)*

- if-then-else grammar is ambiguous

Compiler construction
in4303 – lecture 4

Bottom-up parsing

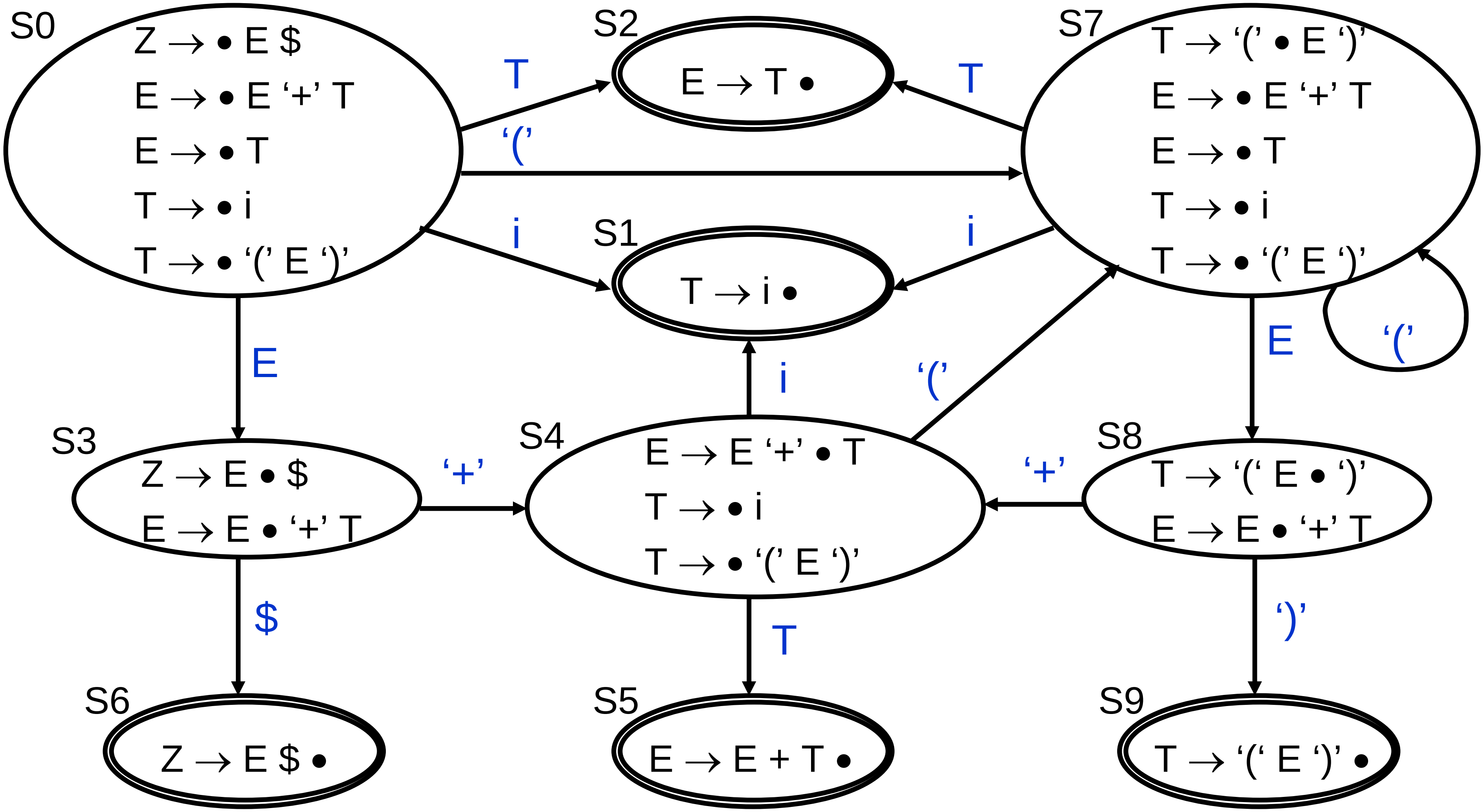
Chapter 2.2.5

Exercise (8 min.)

- complete the transition diagram for the LR(0) automaton

- can you think of a single input expression that causes all states to be used? If yes, give an example. If no, explain.

Answers (fig 2.89)



Answers

The following expressions exercise all states

(i + i) $

(i) + I $

i + (i) $

...

Exercise (6 min.)

- derive the SLR(1) ACTION/GOTO table (with shift-reduce conflict) for the grammar:

$$S \rightarrow A \mid x\ b$$
$$A \rightarrow a\ A\ b \mid x$$

Answers

state	stack symbol / look-ahead token				
	a	b	x	$	A
0	s4		s1		s3
1		s2/r4		r4	
2				r2	
3				r1	
4	s4		s5		s6
5		r4		r4	
6		s7			
7		r3		r3	

1: $S \rightarrow A$

2: $S \rightarrow x\ b$

3: $A \rightarrow a\ A\ b$

4: $A \rightarrow x$

FOLLOW(S) = {$}

FOLLOW(A) = {$,b}

Compiler construction
in4303 – lecture 5

Semantic analysis

Assignment #1

Chapter 6.1

Exercise (3 min.)

complete the table

expression construct	result kind (lvalue/rvalue)
constant	rvalue
variable	
identifier (non-variable)	
&lvalue	
*rvalue	
V[rvalue]	
rvalue + rvalue	
lvalue = rvalue	

V stands for lvalue or rvalue

Answers

complete the table

expression construct	result kind (lvalue/rvalue)
constant	rvalue
variable	lvalue
identifier (non-variable)	rvalue
&lvalue	rvalue
*rvalue	lvalue
V[rvalue]	V
rvalue + rvalue	rvalue
lvalue = rvalue	rvalue

V stands for lvalue or rvalue

Exercise (5 min.)

```
%token DIGIT
%token REG
%right '='
%left '+'
%left '*'
%%
expr : REG '=' expr        {        ??         }
     | expr '+' expr       { $$ = $1 + $3;}
     | expr '*' expr       { $$ = $1 * $3;}
     | '(' expr ')'        { $$ = $2;}
     | REG                 {        ??         }
     | DIGIT
     ;
%%
```

Extend the interpreter to a desk calculator with registers named a – z. Example input: v=3*(w+4)

Answers

```
%{
int reg[26];
%}
%token DIGIT
%token REG
%right '='
%left '+'
%left '*'
%%
expr : REG '=' expr      { $$ = reg[$1] = $3;}
     | expr '+' expr     { $$ = $1 + $3;}
     | expr '*' expr     { $$ = $1 * $3;}
     | '(' expr ')'      { $$ = $2;}
     | REG               { $$ = reg[$1];}
     | DIGIT
     ;
%%
```

Answers

```
%%

yylex()
{   int c = getchar();

    if (isdigit(c)) {
        yylval = c - '0';
        return DIGIT;
    } else if ('a' <= c && c <= 'z') {
        yylval = c - 'a';
        return REG;
    }
    return c;
}
```

Exercise (5 min.)

```
eval( Expr *e)
{

        ???

}
```

- write the code to evaluate an expression

Answers

```
eval( Expr *e)
{       static int reg[26];

        switch (e->type) {
          case EXPR_REG:
                return reg[e->reg];

          case EXPR_DIG:
                return e->dig;

          case EXPR_BIN:
                switch (e->op) {
                  case '=': return reg[e->reg] = eval(e->right);
                  case '+': return eval(e->left) + eval(e->right);
                  case '*': return eval(e->left) * eval(e->right);
                }
       }
}
```

Compiler construction in4303 – lecture 6

AST processing

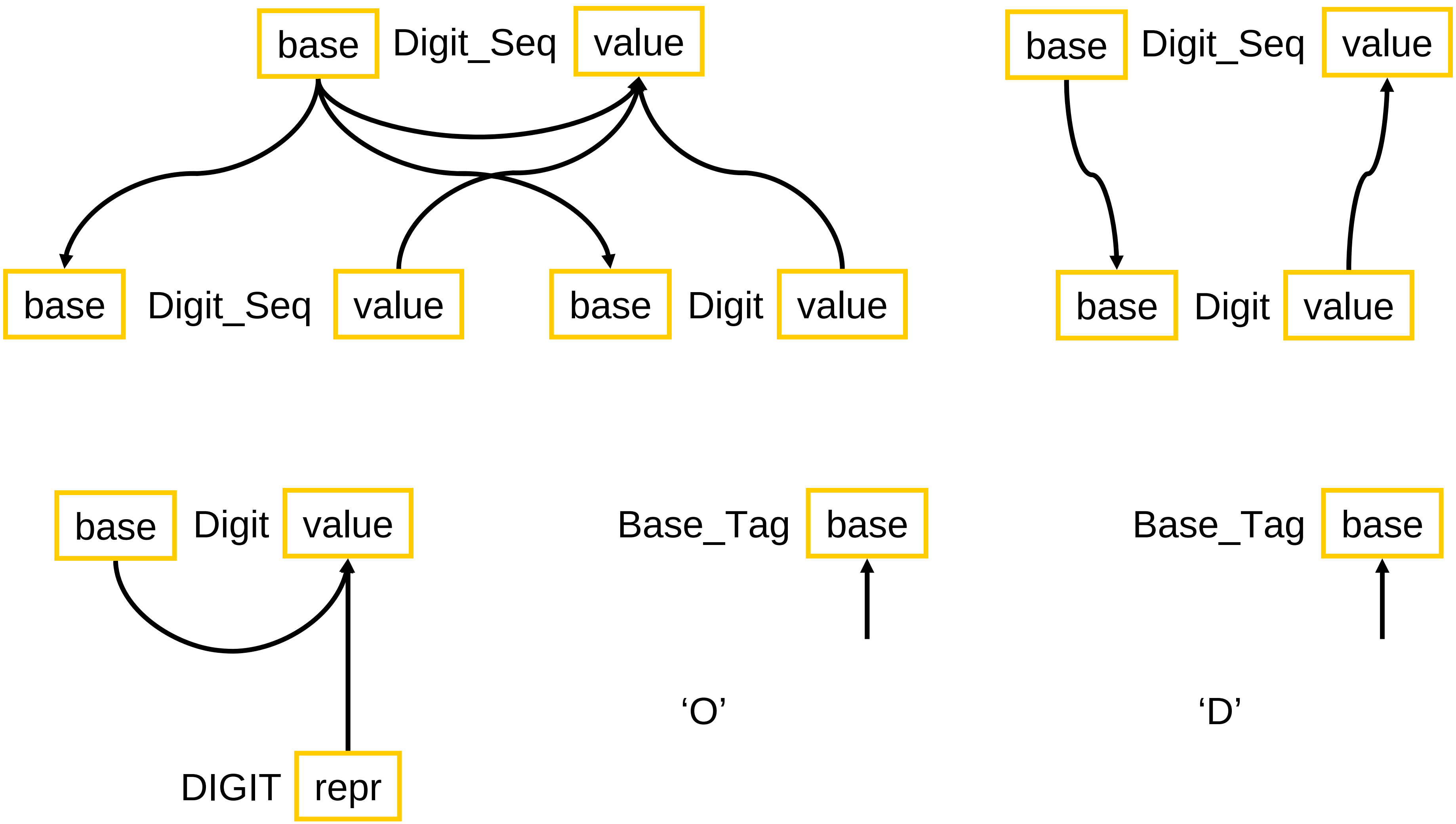
attribute grammars

Chapter 3.1

Exercise (5 min.)

- draw the other dependency graphs for the integral-number attribute grammar

Answers


base
Digit_Seq
value
base
Digit_Seq
value
base
Digit_Seq
value
base
Digit
value
base
Digit
value
base
Digit
value
DIGIT
repr
Base_Tag
base
'O'
Base_Tag
base
'D'

Exercise (4 min.)

```
WHILE Number.value is not set:
        walk number(Number);
```

- how many tree walks are necessary to evaluate the attributes of the AST representing the octal number '13O' ?
- how many for '1234D' ?

Answers

- any integral number can be evaluated with two walks

Compiler construction
in4303 – lecture 7

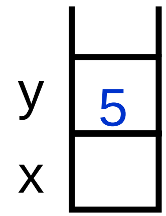
AST processing
manual methods

Chapter 3.2

Exercise (5 min.)

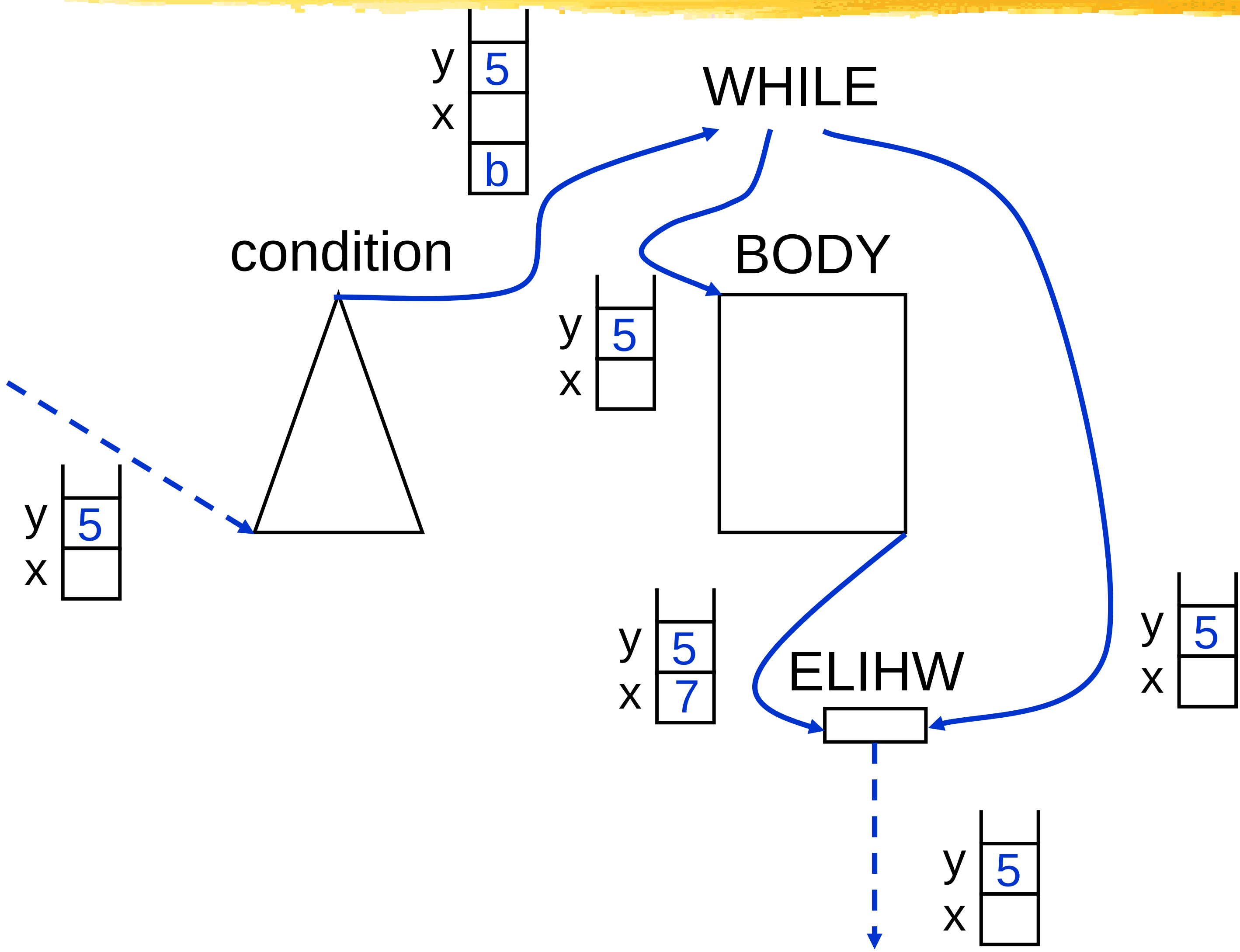
- draw the control flow graph for

  while C **do** S **od**



- propagate initial stack
  when C represents y>x
  and S stands for x=7

Answers



Exercise (5 min.) + Answers

- determine the KILL and GEN sets for the property "V is live here" for the following statements

statement S	KILL	GEN
$v = a \oplus b$	{v}	{a,b}
v = M[i]	{v}	{i}
M[i] = v	{}	{i,v}
$f(a_1, \ldots ,a_n)$	{}	$\{a_1, \ldots ,a_n\}$
$v = f(a_1, \ldots ,a_n)$	{v}	$\{a_1, \ldots ,a_n\}$
if a>b **then** goto L_1 **else** goto L_2	{}	{a,b}
L:	{}	{}
goto L	{}	{}

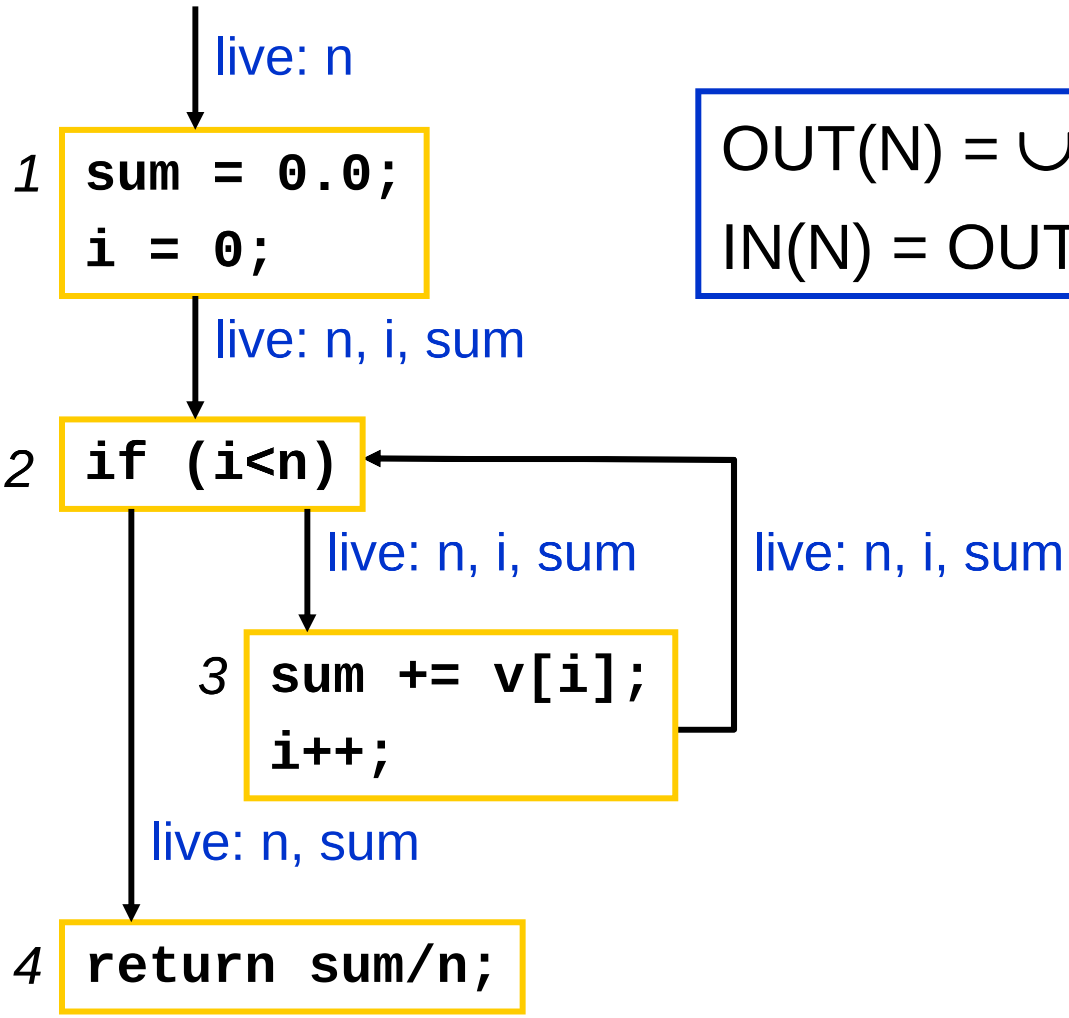
Exercise (7 min.)

- draw the control-flow graph for the following code fragment

```
double average(int n, double v[])
{  int i;
   double sum = 0.0;

   for (i=0; i<n; i++) {
      sum += v[i];
   }
   return sum/n;
}
```

- perform backwards live analysis using the KILL and GEN sets from the previous exercise

Answers



$$OUT(N) = \bigcup_{M \in successor[N]} IN(M)$$

$$IN(N) = OUT(N) \setminus KILL(N) \cup GEN(N)$$

statement	KILL	GEN
1	sum, i	
2		i, n
3	sum, i	sum, i
4		sum, n

Answers

process nodes top to bottom (from 0 to 4)

statement			iteration 1		iteration 2		iteration 3	
	KILL	GEN	OUT	IN	OUT	IN	OUT	IN
0							n	
1	sum, i				i, n	n	i, n, sum	n
2		i, n		i, n	i, n, sum	i, n, sum	i, n, sum	i, n, sum
3	sum, i	sum, i	i, n	i, n, sum	i, n, sum	i, n, sum	i, n, sum	i, n, sum
4		sum, n		n, sum		n, sum		n, sum

processing nodes in reverse (from 4 to 0) requires only two iterations

Compiler construction
in4303 – lecture 8

Interpretation

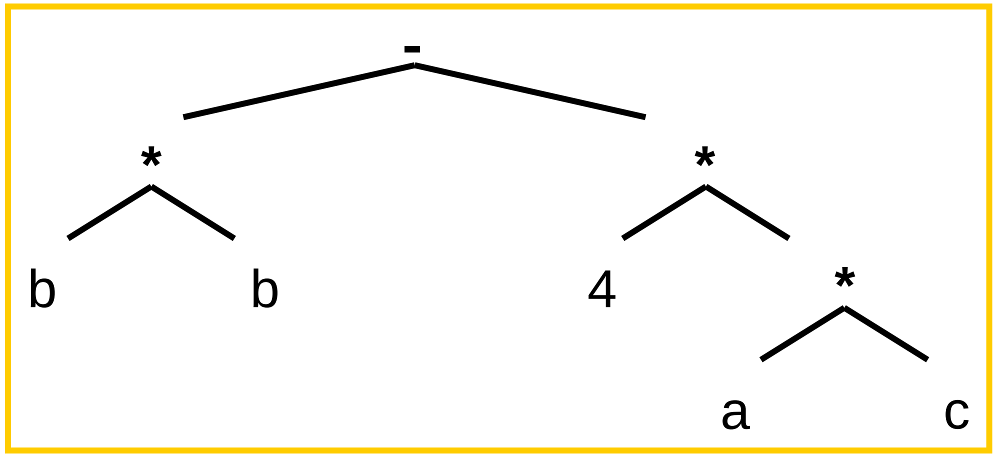
Simple code generation

Chapter 4 – 4.2.4

Exercise (5 min.)

- generate code for the expression

  b*b – 4*a*c



  on a register machine with 32 registers numbered R1 .. R32

Answers

Load_Mem b, R1

Load_Mem b, R2

Mul_Reg R2, R1

Load_Const 4, R2

Load_Mem a, R3

Load_Mem c, R4

Mul_Reg R4, R3

Mul_Reg R3, R2

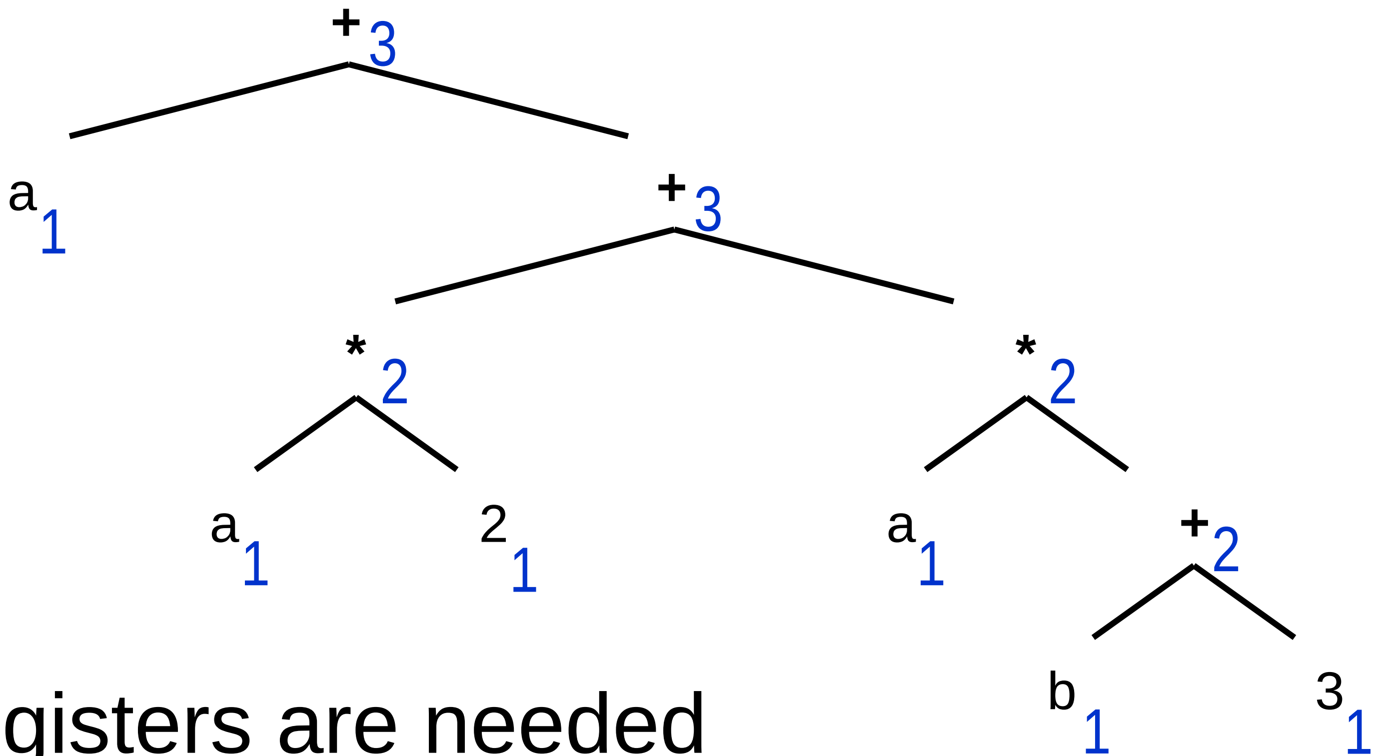
Sub_Reg R2, R1

Exercise (4 min.)

- determine the number of registers needed to evaluate the expression

$$\boxed{a + a*2 + a*(b+3)}$$

Answers



- three registers are needed
- but only two if you apply left-factoring!

$$a + a*2 + a*(b+3) = a*(b+6)$$

Compiler construction
in4303 – lecture 9

Code generation

Chapter 4.2.5, 4.2.7,
4.2.11 – 4.3

Exercise (5 min.)

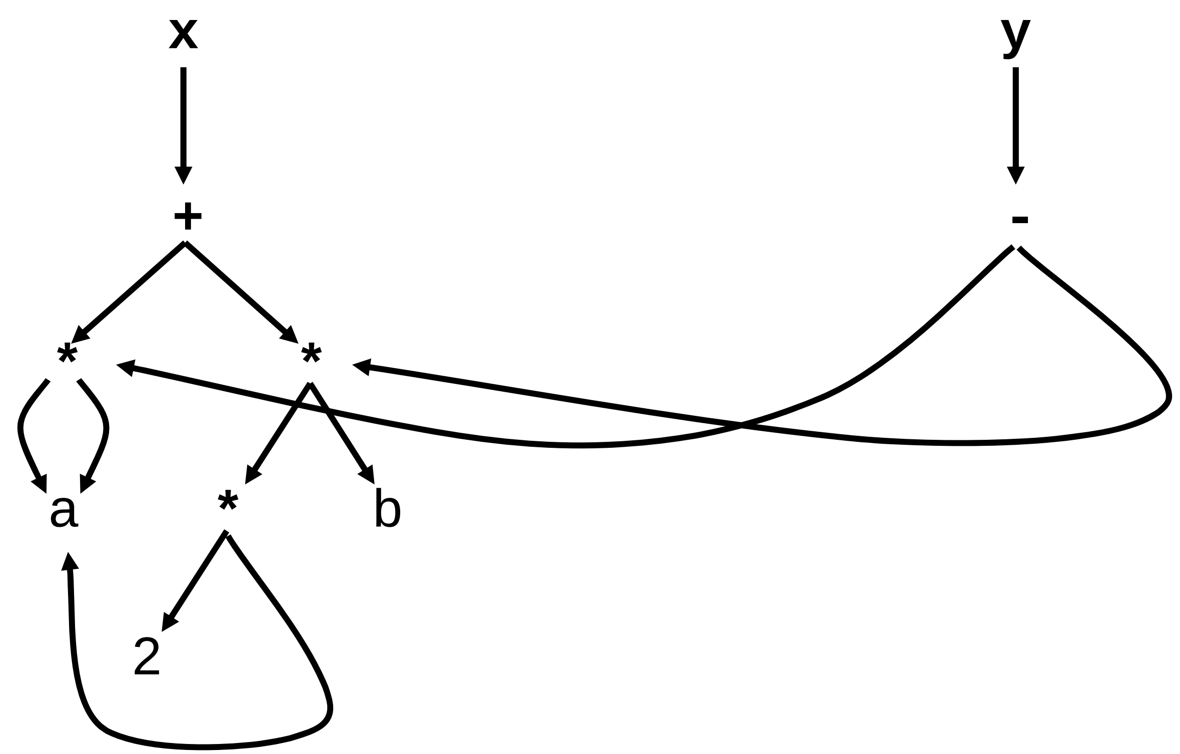
- given the code fragment

```
x := a*a + 2*a*b + b*b;
y := a*a – 2*a*b + b*b;
```

draw the dependency graph before and after common subexpression elimination.

Answers

dependency graph after CSE



Exercise (7 min.)

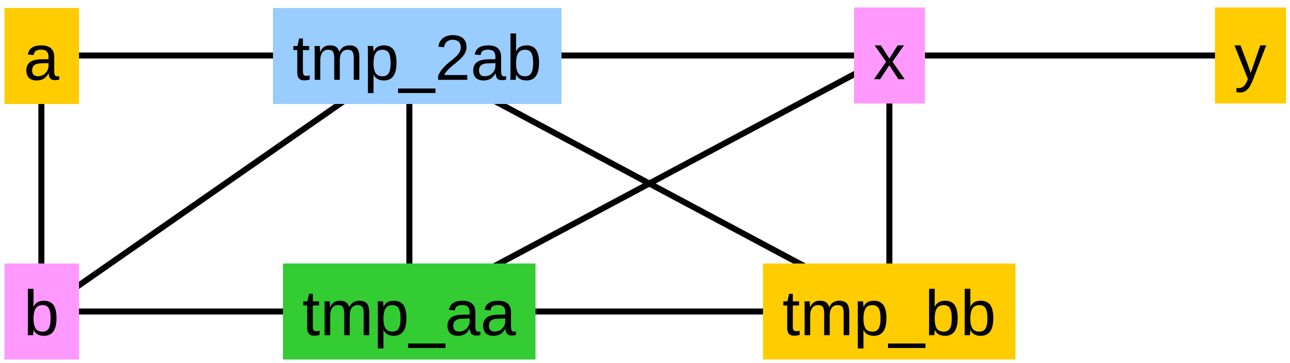
```
{  int tmp_2ab = 2*a*b;
   int tmp_aa = a*a;
   int tmp_bb = b*b;

   x := tmp_aa + tmp_2ab + tmp_bb;
   y := tmp_aa - tmp_2ab + tmp_bb;
}
```

given that a and b are live on entry and dead on exit, and that x and y are live on exit:

(a) construct the register interference graph

(b) color the graph; how many register are needed?

Answers



4 registers are needed

Compiler construction
in4303 – lecture 10

Imperative & OO languages:
context handling

Chapter 6.2

Exercise (4 min.)

Give the method tables for **`Rectangle`** and **`Square`**

```
abstract class Shape {
   boolean IsShape() {return true;}
   boolean IsRectangle() {return false;}
   boolean IsSquare() {return false;}
   abstract double SurfaceArea();
}
class Rectangle extends Shape {
   double SurfaceArea { ... }
   boolean IsRectangle() {return true;}
}
class Square extends Rectangle {
   boolean IsSquare() {return true;}
}
```

Answers

Method table for **`Rectangle`**	Method table for **`Square`**
`IsShape_Shape_Shape`	`IsShape_Shape_Shape`
`IsRectangle_Shape_Rectangle`	`IsRectangle_Shape_Rectangle`
`IsSquare_Shape_Shape`	`IsSquare_Shape_Square`
`SurfaceArea_Shape_Rectangle`	`SurfaceArea_Shape_Rectangle`

Exercise (4 min.)

- given an object **e** of class **E**, give the compiled code for the calls

```
e.m1()
e.m3()
e.m4()
```

Answers

e.m1()	(*(e->dispatch_table[0]))((Class_C *) e)
e.m3()	(*(e->dispatch_table[2]))((class_D *)((char *)e + sizeof(Class_C)))
e.m4()	(*(e->dispatch_table[3]))((class_D *)((char *)e + sizeof(Class_C)))

Compiler construction
in4303 – lecture 11

Imperative & OO languages:
routines & control flow

Chapter 6.3 - 6.4

Exercise (5 min.)

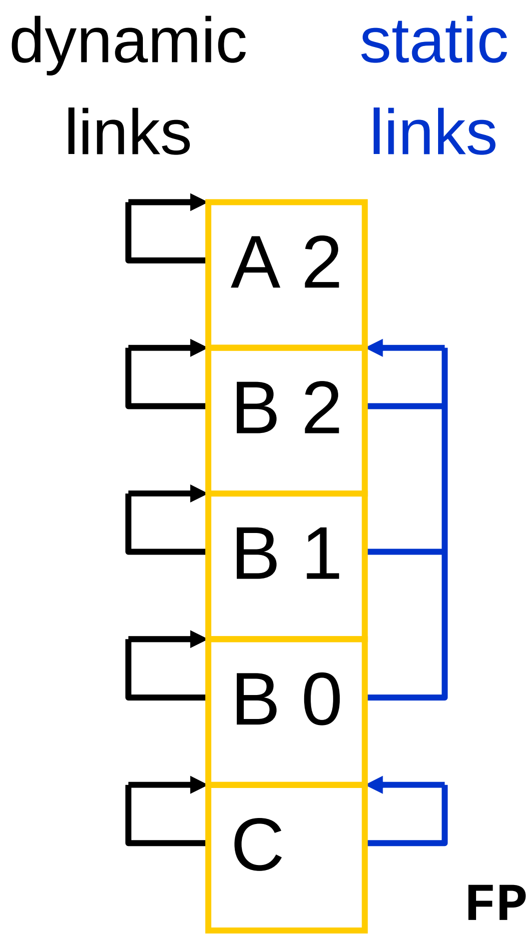
- Given the GNU C routine:

```
void A(int a) {
   void B(int b) {
      void C(void) {
         printf("C called, a = %d\n", a);
      }
      if (b == 0) C() else B(b-1);
   }
   B(a);
}
```

a) draw the stack that results from the call A(2)

b) how does C access the parameter (a) from A?

Answers



`FP->static_link->static_link->param[#a]`

Exercise (3 min.)

while -statement

`WHILE` *condition* **`DO`** *statement* **`END WHILE`**

- give a translation schema for while statements

Answers

while -statement

`WHILE` *condition* **`DO`** *statement* **`END WHILE`**

```
  GOTO test_label;
loop_label:
```
Code statement(*statement*)
```
test_label:
```
Boolean control code(*condition*, loop_label , No_label)

Compiler construction
in4303 – lecture 12

Memory Management

Chapter 5

Exercise (5 min.)

- give the pseudo code for free() when using free lists indexed by size.

Answers

```
PROCEDURE Free (Block pointer):
  SET Chunk pointer TO Block pointer – Admin size;
  SET Index TO ²log(Chunk pointer .size);

  IF Index <= 10:
    SET Chunk pointer .next TO Free list[Index];
    SET Free list[Index] TO Chunk pointer;
  ELSE
    SET Chunk pointer .free TO True;
    // Coalesce subsequent free chunks
```

Compiler construction
in4303 – lecture 13

Functional programming

Chapter 7

Exercise (5 min.)

- infer the polymorphic type of the following higher-order function:

```
filter f []     = []
filter f (x:xs) = if not(f x) then  filter f xs
                  else x : (filter f xs)
```

Answers

```
filter f []     = []
filter f (x:xs) = if not(f x) then  filter f xs
                  else x : (filter f xs)
```

filter :: t_1 -> t_2 -> t_3

```
filter f []     = []
```

filter :: t_1 -> [a] -> [b]

```
filter f (x:xs) = if not(f x) then  filter f xs
                  else x : (filter f xs)
```

```
     x :: a
     f :: a -> Bool
filter :: (a -> Bool) -> [a] -> [a]
```

Exercise (6 min.)

- infer the strict arguments of the following recursive function:

```
g x y 0 = x
g x y z = g y x (z-1)
```

- how many iterations are needed?

Answers

```
g x y 0 = x
g x y z = g y x (z-1)
```

```
g x y z = if z == 0 then x
          else g y x (z-1)
```

step	assumption	result
1	{x,y,z}	{x,z}
2	{x,z}	{z}
3	{z}	{z}
4		

Compiler construction in4303 – lecture 14

Logic programs

Chapter 8

Exercise (5 min.)

Specify Prolog rules for the binary relations

- **`offspring`** (nakomeling)
- **`spouse`** (echtgenoot)

using the **`parent`** relation.

The bait hides the hook

```
parent(dick,koen).
parent(lia,koen).

parent(koen,allard).
parent(koen,linde).
parent(koen,merel).
```

Answers

```
offspring(X,Y) :- parent(X,Y).
offspring(X,Y) :- parent(X,Z), offspring(Z,Y).
```

```
spouse(X,Y) :- parent(X,Z), parent(Y,Z),
               X \= Y, !.
```

`X \= Y` to rule out **`spouse(koen,koen)`**

`!` to rule out duplicates (multiple children)