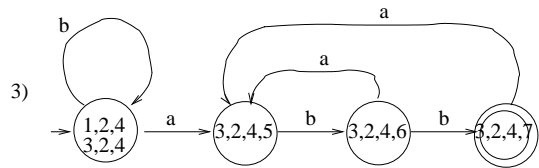
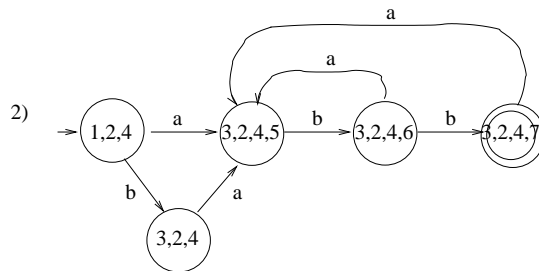
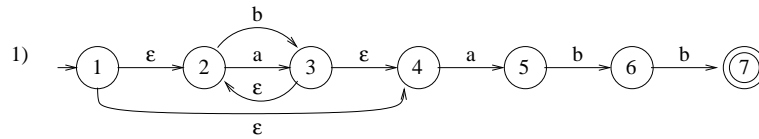


IMPLEMENTATION LECTURE 2 Homework questions

- 1) Give a corresponding NFA to $(a|b)^*abb$.
- 2) Convert this to a DFA.
- 3) Now minimize the DFA.



Use flex to generate a program which counts the number of lines, characters and words in standard input.

```
int num_lines = 0, num_chars = 0, num_words = 0;

%%
\n                {++num_lines; ++num_chars;}
[a-zA-z]* {++num_words; num_chars += yyleng;}
.                {++num_chars;}

%%
main() {
    yylex();
    printf("# of lines= %d, # of words= %d, # of characters= %d\n",
        num_lines, num_words, num_chars);
}
```

IMPLEMENTATION LECTURE 3 Homework questions

Use the left-recursion algorithm to remove recursion from the grammar

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Sd \end{aligned}$$

Assume the ordering is S, A , i.e. S is A_1 and A is A_2 . The first time through the loop $i = 1$ so we simply eliminate direct left recursion from S which does nothing. Next time with $i = 2$ and $j = 1$ we obtain

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Ac \mid Aad \mid bd \end{aligned}$$

Now we eliminate direct recursion from A and obtain

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bdA' \\ A' &\rightarrow cA' \mid adA' \mid \epsilon \end{aligned}$$

IMPLEMENTATION LECTURE 4 Homework questions

Give the predictive parsing table for the grammar

- (P1) $S \rightarrow B C D$
- (P2) $B \rightarrow \epsilon$
- (P3) $B \rightarrow b b B$
- (P4) $C \rightarrow \epsilon$
- (P5) $C \rightarrow c C$
- (P6) $D \rightarrow \epsilon$
- (P7) $D \rightarrow d D$

We have that

- $FIRST(b) = b$
- $FIRST(c) = c$
- $FIRST(d) = d$
- $FIRST(d D) = d$
- $FIRST(D) = d, \epsilon$
- $FIRST(c C) = c$
- $FIRST(C) = c, \epsilon$
- $FIRST(b b B) = b$
- $FIRST(B) = b, \epsilon$
- $FIRST(B C D) = b, c, d, \epsilon$
- $FIRST(S) = b, c, d, \epsilon$

and

- $FOLLOW(B) = c, d, \$$
- $FOLLOW(C) = d, \$$
- $FOLLOW(D) = \$$
- $FOLLOW(S) = \$$

The parsing table for this grammar is

	b	c	d	\$
<i>S</i>	P1	P1	P1	P1
<i>B</i>	P3	P2	P2	P2
<i>C</i>		P5	P4	P4
<i>D</i>			P7	P6

and use your table (with table-driven predictive parsing) to determine if $b c$ and $d c$ are in the language of the grammar.

Left as an exercise!

IMPLEMENTATION LECTURE 5 Homework questions

Left as exercises.

IMPLEMENTATION LECTURE 6 Homework questions

Construct the SLR parsing table for the grammar

- (P1) $S \rightarrow B C D$
- (P2) $B \rightarrow \epsilon$
- (P3) $B \rightarrow b b B$
- (P4) $C \rightarrow \epsilon$
- (P5) $C \rightarrow c C$
- (P6) $D \rightarrow d$
- (P7) $D \rightarrow d D$

First we augment the grammar with the production

- (P0) $S' \rightarrow S$

Now we compute the LR(0) collection is

$$\begin{array}{l}
 I_0 : \quad S' \rightarrow \cdot S \quad S \rightarrow \cdot B C D \\
 \quad \quad B \rightarrow \cdot \quad B \rightarrow \cdot b b B \\
 \text{goto}(I_0, b) = I_1 : \quad B \rightarrow b \cdot b B \\
 \text{goto}(I_0, B) = I_2 : \quad S \rightarrow B \cdot C D \quad C \rightarrow \cdot \\
 \quad \quad C \rightarrow \cdot c C \\
 \text{goto}(I_0, S) = I_3 : \quad S' \rightarrow S \cdot \\
 \text{goto}(I_1, b) = I_4 : \quad B \rightarrow b b \cdot B \\
 \quad \quad B \rightarrow \cdot \quad B \rightarrow \cdot b b B \\
 \text{goto}(I_2, C) = I_5 : \quad S \rightarrow B C \cdot D \quad D \rightarrow \cdot d \\
 \quad \quad D \rightarrow \cdot d D \\
 \text{goto}(I_2, c) = I_6 : \quad C \rightarrow c \cdot C \\
 \quad \quad C \rightarrow \cdot \\
 \quad \quad C \rightarrow \cdot c C \\
 \text{goto}(I_4, B) = I_7 : \quad B \rightarrow b b B \cdot \\
 \text{goto}(I_4, b) = I_1 \\
 \text{goto}(I_5, d) = I_8 : \quad D \rightarrow d \cdot \quad D \rightarrow d \cdot D \\
 \quad \quad D \rightarrow \cdot d \quad D \rightarrow \cdot d D \\
 \text{goto}(I_5, D) = I_{11} : \quad S \rightarrow B C \cdot D \\
 \text{goto}(I_6, C) = I_9 : \quad C \rightarrow c C \cdot \\
 \text{goto}(I_6, c) = I_6 \\
 \text{goto}(I_8, d) = I_8 \\
 \text{goto}(I_8, D) = I_{10} : \quad D \rightarrow d D \cdot
 \end{array}$$

We have that

$$\begin{array}{l}
 FOLLOW(B) = c, d \\
 FOLLOW(C) = d \\
 FOLLOW(D) = \$ \\
 FOLLOW(S) = \$
 \end{array}$$

The LR parsing table for this grammar is

<i>STATE</i>	<i>ACTION</i>				<i>GOTO</i>			
	<i>b</i>	<i>c</i>	<i>d</i>	<i>\$</i>	<i>S</i>	<i>B</i>	<i>C</i>	<i>D</i>
0	<i>s1</i>	<i>r2</i>	<i>r2</i>		3	2		
1	<i>s4</i>							
2		<i>s6</i>	<i>r4</i>				5	
3				<i>acc</i>				
4	<i>s1</i>	<i>r2</i>	<i>r2</i>			7		
5			<i>s8</i>					11
6		<i>s6</i>					9	
7		<i>r3</i>	<i>r3</i>					
8			<i>s8</i>	<i>r6</i>				10
9			<i>r5</i>					
10				<i>r7</i>				
11				<i>r1</i>				

where

- si* is shift and stack state *i*
- rj* is reduce using production *j*
- acc* is accept.

IMPLEMENTATION LECTURE 7 Homework questions

Give an algorithm to compute the non-terminals in a grammar which generate ϵ . [Hint: look at the algorithm for computing the FIRST set]

One technique is just to check if $\epsilon \in FIRST(X)$ for each non-terminal X .

Consider the grammar

$$exp \rightarrow exp + exp \mid \mathbf{int}$$

Give a Chomsky Normal Form grammar for it.

We don't need to use ϵ -elimination.

$$\begin{aligned} exp &\rightarrow expY \mid \mathbf{int} \\ Y &\rightarrow Xexp \\ X &\rightarrow + \end{aligned}$$

Now use the CKY Algorithm with the Chomsky Normal Form grammar to parse the string: **int + int + int**.

	1	2	3	4	5
1	<i>exp</i>	\emptyset	<i>exp</i>	\emptyset	<i>exp</i>
2		<i>X</i>	<i>Y</i>	\emptyset	<i>Y</i>
3			<i>exp</i>	\emptyset	<i>exp</i>
4				<i>X</i>	<i>Y</i>
5					<i>exp</i>

IMPLEMENTATION LECTURE 8 Homework questions

The reason for using attribute grammars is that they have a higher expressive power than normal context-free grammars. We know that $L = a^n b^m c^n d^m$ is not a context-free language.

Write an attribute grammar for L .

Production	Semantic Rules
$s \rightarrow as\ bs\ cs\ ds$	if $c.n = a.n \wedge d.n := b.n$
$as \rightarrow \epsilon$	$as.n := 0$
$as_1 \rightarrow Aas_2$	$as_1.n := as_2.n + 1;$
$bs \rightarrow \epsilon$	$bs.n := 0$
$bs_1 \rightarrow Bbs_2$	$bs_1.n := bs_2.n + 1;$
$cs \rightarrow \epsilon$	$cs.n := 0$
$cs_1 \rightarrow Ccs_2$	$cs_1.n := cs_2.n + 1;$
$ds \rightarrow \epsilon$	$ds.n := 0$
$ds_1 \rightarrow Dds_2$	$ds_1.n := ds_2.n + 1;$

Give the missing attribute rules and conditions for the other productions in the grammar for the desk top calculator language.

Grammar rule: $program \rightarrow declarations\ statements$

Attribution Rules:

```
declarations.envin := [ ] (* the empty array*)
statements.env := declarations.env
```

Grammar rule: $declarations1 \rightarrow declaration ; declarations2$

Attribution Rules:

```
declaration.envin := declarations1.envin
declarations2.envin := declaration.env
declarations1.env := declarations2.env
```

Grammar rule: $declarations \rightarrow \epsilon$

Attribution Rules:

```
declarations.env := declarations.envin
```

Grammar rule: $declaration \rightarrow \text{int } IDENT$

Attribution Rules:

```
declaration.env :=
update(declaration.envin, IDENT.id, int)
```

Condition:

```
lookup(declaration.envin, IDENT.id) = null
```

Grammar rule: $declaration \rightarrow \text{real } IDENT$

Attribution Rules:

```
declaration.env :=
update(declaration.envin,IDENT.id,real)
```

Condition:

```
lookup(declaration.envin,IDENT.id) = null
```

None.

Grammar rule: $statements \rightarrow \epsilon$

Attribution Rules:
Grammar rule: $statements1 \rightarrow \text{assgn};statements2$

Attribution Rules:

```
assgn.env := statements1.env
statements2.env := statements1.env
```

Grammar rule: $assgn \rightarrow IDENT := exp$

Attribution Rules:

```
exp.env := assgn.env
exp.reqtype := lookup(assgn.env,IDENT.id)
assgn.opn :=
  if exp.reqtype = int then int_asg else real_asg
```

Condition:

```
coercible(exp.type, exp.reqtype)
```

where $coercible(T_1, T_2)$ holds if $T_1 = T_2$, or $T_1 = int$ and $T_2 = real$.

Grammar rules: $exp1 \rightarrow exp2 + exp3$

$exp1 \rightarrow exp2 - exp3$

$exp1 \rightarrow exp2 / exp3$

$exp1 \rightarrow exp2 * exp3$

$exp1 \rightarrow exp2 * exp3$

Attribution Rules:

```
exp2.env := exp1.env
exp3.env := exp1.env
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.opn :=
  if exp1.type = int then int_add else real_add
exp2.reqtype := exp1.type
exp3.reqtype := exp1.type
```

Grammar rule: $exp1 \rightarrow (exp2)$

Attribution Rules:

```
exp2.env := exp1.env
exp1.type := exp2.type
exp1.opn := nullop
exp2.reqtype := exp2.type
```

Grammar rule: $exp \rightarrow IDENT$

Attribution Rules:

```
exp.type := lookup(exp.env, IDENT.id)
exp.opn := nullop
```

Grammar rule: $exp \rightarrow REALCONST$

Attribution Rules:

```
exp.type := real
exp.opn := nullop
```

Grammar rule: $exp \rightarrow INTCONST$

Attribution Rules:

```
exp.type := int
exp.opn := nullop
```

Now use this to type check the program

```
real x;
x := 100 + 8.9;
```

Left as an exercise!

Give a visit sequence for the production

$assgn \rightarrow IDENT := exp$ The attribution rules are

```
exp.env := assgn.env
exp.reqtype := lookup(assgn.env, IDENT.id)
assgn.opn :=
  if exp.reqtype = int then int_asg else real_asg
with condition
```

```
coercible(exp.type, exp.reqtype)
```

One possible visit sequence is

```
evaluate exp.reqtype
evaluate assgn.opn
evaluate exp.env
visit exp
check coercible(exp.type, exp.reqtype)
```

Extend the large example and give attribute rules and conditions for the production

$exp \rightarrow floor(exp)$,

where *floor* takes a real and returns an integer.

Grammar rule: $exp1 \rightarrow floor(exp2)$

Attribution Rules:

```
exp2.env := exp1.env
exp2.reqtype := real
exp1.type := int
exp.1opn := floor
```

Condition:

```
coercible(exp2.type, real)
```

Or you might require that $exp2.type = int$.

IMPLEMENTATION LECTURE 9 Homework questions

Work out the type of

```
fun fst (x,y) = x;
```

It is $(a \times b) \rightarrow a$. and

```
fun fst_fst z = fst (fst z);
```

It is $((a \times b) \times c) \rightarrow a$.

Work out the type of

```
fun twice f x = f(f(x));
```

It is $(a \rightarrow a) \rightarrow a \rightarrow a$.

Can tail recursion optimisation be used with the linear time version of reverse? If so give the equivalent ML code after the optimisation has been applied.

```
(* linear time reverse *)
fun rev1 ([], ys) = ys
  | rev1 ((x::xs), ys) = rev1 (xs, (x::ys));

fun reverse xs = rev1 (xs, []);
```

Yes tail recursion optimisation works for `rev1` as the call to `rev1` in the body is a tail call. The optimised code is equivalent to

```
fun rev1(xs,ys) =
  while not null(!xs)do
    (xs:= tail(!xs) ; ys := head(!xs)::!ys);
  !ys;
```

IMPLEMENTATION LECTURE 10 Homework questions

How do you compute the value of the static link in the procedure being called?

In a Pascal like language a procedure can only call a procedure that is within the same block or in a surrounding block or in the next level down.

Imagine that procedure p is at level K and calls procedure q at level L . If q is local to p . i.e. $L = K + 1$ then q 's static link is to the stack frame for p . Otherwise we follow $K - L$ static links up the stack frame of p and set q to the static link of that stack frame. For instance if q is at the same level as p then q 's static link is set to that of p .

Give attribute rules to generate code for a *while* loop and a procedure call (first give the grammar!). Left as an exercise.