Quiz

Compiler Construction, Fall 2013

Wednesday, October 23rd, 2013

Last Name : ______________________________

First Name : ______________________________

Exercise	**Points**	**Achieved Points**
1	15	
2	30	
3	15	
Total	60	

General notes about this quiz

- You are allowed to use any printed material that you brought yourself to the exam. You are not allowed to use any notes that were not typed-up. Also, you are not allowed to exchange the notes with other students taking the quiz.
- You have in total **3 hours 45 minutes**.
- It is advisable to do the questions you know best first.
- Please write the answer of each question on a **separate sheet**.
- Please use a permanent pen.
- Make sure you write your name on each sheet of paper.
- Have your CAMIPRO card ready on the desk.

Problem 1: Lexer for Morse code (15 points)

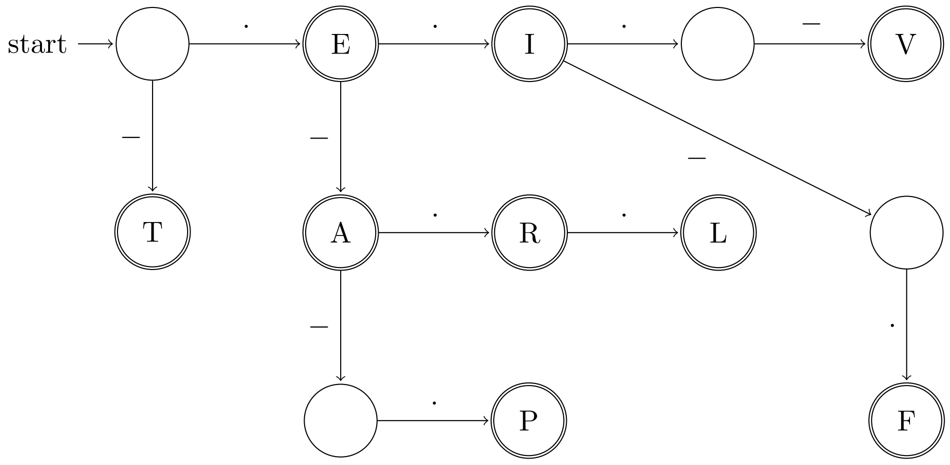
Morse code is a way to encode letters using dots (`.`) and dashes (`-`). Here is the encoding of a subset of letters that we consider in this problem.

Token name	Characters
A	`.-`
E	`.`
I	`..`
T	`-`
P	`.--.`
L	`.-..`
R	`.-.`
V	`...-`
F	`..-.`

In this question, we consider input alphabet $\Sigma_I = \{\texttt{.}, \texttt{-}\}$ consisting of dots (`.`), dashes (`-`). The output alphabet consists of the letters above, $\Sigma_O = \{A, E, I, T, P, L, R, V, F\}$.

a) [**10 pts**] Create a lexer that recognizes the above subset of Morse code tokens. Represent the lexer as a deterministic automaton with edges labeled by characters from Σ_I. The lexer should use the longest match (maximal munch) rule and recognize only one token at a time. Label the accepting states of the lexer with the appropriate letters from Σ_O.

Solution



b) [**5 pts**] Give the sequence of tokens produced by running your lexer on this input:

```
......-.-.......-
```

Note that there are no spaces in the sequence; there are 6 initial dots, and there are 7 dots before the final dash. The result should be a sequence of tokens from the set Σ_O.

Solution

```
I  I  F    T I  I  V
..|..|..-.|-|..|..|...-
```

Problem 2: LL(1) Parsing (30 points)

Consider a grammar for expressions where the multiplication sign is optional.

```
ex ::= ex + ex | ex * ex | ex ex | ID | INTLIT
```

a) [**5 pts**] Compute nullable, first, and follow sets for this grammar alternatives. Use this information to check whether this grammar is LL(1).

Solution

nullable(ex) = $\emptyset$
first(ex) = {ID, INTLIT}
first(ex+ex) = {ID, INTLIT}
first(ex*ex) = {ID, INTLIT}
first(ex ex) = {ID, INTLIT}
first(ID) = {ID}
first(INTLIT) = {INTLIT}
follow(ex) = {ID, INTLIT, +, *}
follow(ex+ex) = {ID, INTLIT, +, *}
follow(ex*ex) = {ID, INTLIT, +, *}
follow(ex ex) = {ID, INTLIT, +, *}
follow(ID) = {ID, INTLIT, +, *}
follow(INTLIT) = {ID, INTLIT, +, *}

Because first(ex + ex) $\cap$ first(ex $*$ ex) = {ID, INTLIT} $\neq \emptyset$, this grammar is not LL(1).

b) [**10 pts**] Find a grammar recognizing a subset of the above expressions, where "3 x" should be accepted, but not strange expressions "x 3" or "3 6 x" nor "x 3 y". In other words, a constant can appear without explicit multiplication only if it is at the beginning of the sub-expression connected through omitted multiplications. If the grammar you design is not LL(1), then transform it so that it is LL(1). The final result should in any case be an LL(1) grammar that does not allow strange expressions and in which the priority of $*$ and the omitted multiplication are higher than the priority of +, so that e.g. $x + 3 * z$ is parsed into a tree of the same shape as $x + (3 * z)$. Do not worry about the associativity of operators, only their priorities.

Solution

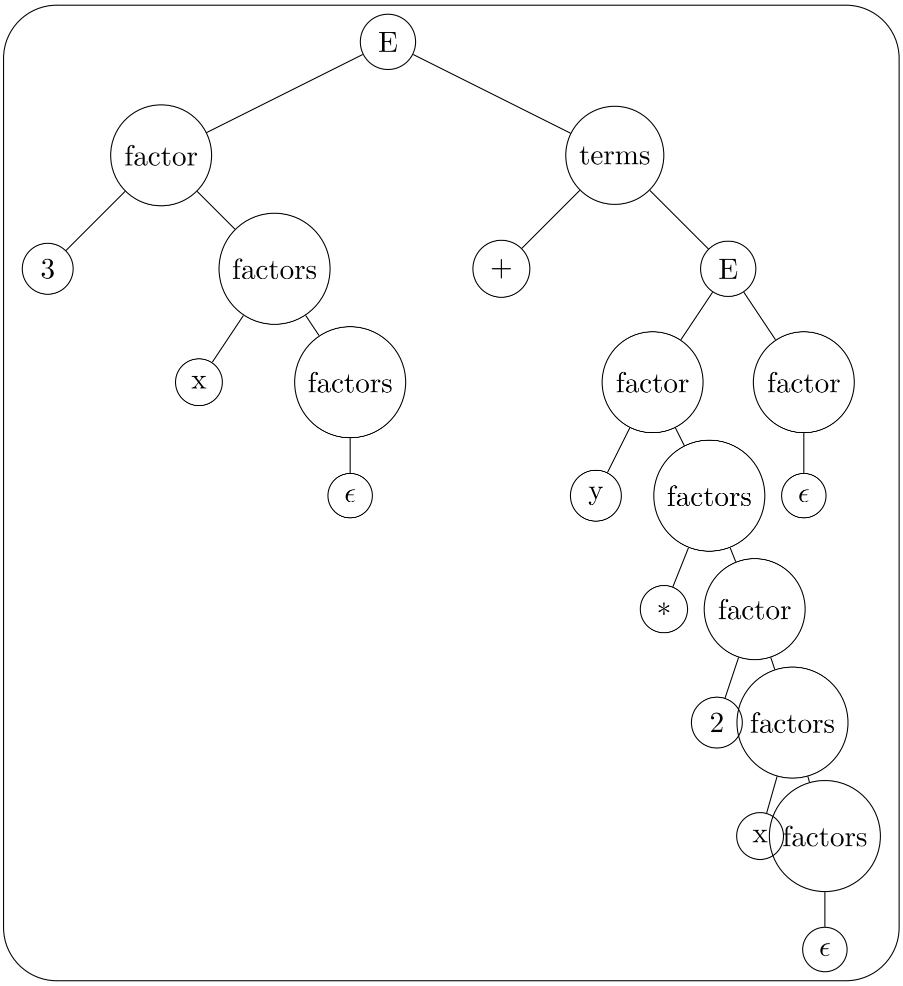
E ::= factor terms
terms := ϵ | + E
factor := INT factors | ID factors
factors ::= ϵ | $*$ factor | ID factors

This is one possibility. Many student came up with clever ideas which work as well.

c) [**5 pts**] Using your grammar, draw the parse tree for the following expression:

```
3 x + y * 2 x
```

Solution



d) **[10 pts]** Create the LL(1) parsing table of your grammar. Run an LL(1) parser on the above example input and mark all table entries that the parser will examine.

Solution

	+	*	INT	ID	EOF
E			factor terms ■	factor terms ■	
factor			INT factors ■	ID factors ■	
factors	ϵ ■	* factor ■		ID factors ■	ϵ ■
terms	+ E ■				ϵ ■

Problem 3: CYK Parsing (15 points)

Consider this grammar for a simple language (inspired somewhat by Objective CAML). The starting non-terminal is S and the only other non-terminals are: LET, LETDEF, FUNCTION, IF, IDCALL, LEXP, EXPR, STMT. The terminals are:

- identifier: ID
- integer constant literal: INTLIT
- keywords: in, let, function, if, then, else

- punctuation and operators: =, −>, +, (,), ;

S := EXPR | LET in S
LET := let LETDEF
LETDEF := IDCALL = EXPR
FUNCTION := function ID −> S
IF := if EXPR then EXPR else EXPR
IDCALL := ID IDCALL | ID
LEXP := ϵ | EXPR LEXP
EXPR := ID LEXP| INTLIT | EXPR + EXPR | ID EXPR LEXP | (S) | STMT ; EXPR
STMT := S ; STMT | STMT ; S

a) **[10 pts]** Convert this grammar to Chomsky Normal Form by ensuring the following properties.

1. terminals t occur alone on the right-hand side: X:=t
2. no unproductive non-terminals symbols
3. no productions of arity more than two
4. no nullable symbols except for the start symbol
5. no single non-terminal productions X::=Y
6. no non-terminals unreachable from the starting one

It is sufficient if you only write the final grammar that you obtain. Note that your grammar must derive the same sequence of tokens from the initial non-terminal S as the original grammar.

Solution

6.no non-terminals unreachable from the starting one (remove IF and FUNCTION) 2.no unproductive non-terminals symbols (remove STMT)
1.terminals t occur alone on the right-hand side: X:=t

S := EXPR | LET IN S
LET := K_LET LETDEF
LETDEF := IDCALL EQ EXPR
IDCALL := I IDCALL | ID
LEXP := ϵ | EXPR LEXP
EXPR := I LEXP | INTLIT | EXPR P EXPR | I EXPR LEXP | LP S RP
I := ID
EQ := '='
IN := in
K_LET := let
LP := (
RP :=)
P := +

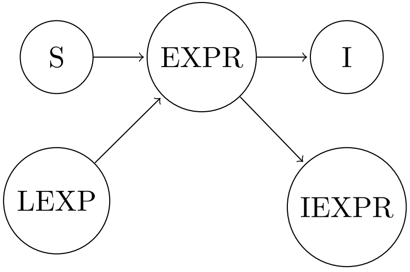
3.no productions of arity more than two

S := EXPR | LETIN S
LETIN := LET IN
LET := K_LET LETDEF
LETDEF := IDCALLEQ EXPR
IDCALLEQ := IDCALL EQ
IDCALL := I IDCALL | ID
LEXP := ϵ | EXPR LEXP
EXPR := I LEXP | INTLIT | EXPRP EXPR | IEXPR LEXP | LPS RP
EXPRP := EXPR P
IEXPR := I EXPR
LPS := LP S
I := ID
EQ := '='
IN := in
K_LET := let
LP := (
RP :=)
P := +

4.no nullable symbols except for the start symbol. Only LEXP is nullable.

S := EXPR | LETIN S
LETIN := LET IN
LET := K_LET LETDEF
LETDEF := IDCALLEQ EXPR
IDCALLEQ := IDCALL EQ
IDCALL := I IDCALL | ID
LEXP := EXPR | EXPR LEXP
EXPR := I | I LEXP | INTLIT | EXPRP EXPR | IEXPR | IEXPR LEXP | LPS RP
EXPRP := EXPR P
IEXPR := I EXPR
LPS := LP S
I := ID
EQ := '='
IN := in
K_LET := let
LP := (
RP :=)
P := +

5.no single non-terminal productions X::=Y
We draw the graph of single rewriting rules.



S := ID | I LEXP | INTLIT | EXPRP EXPR | IEXPR LEXP | I EXPR | LPS RP | LETIN S

LETIN := LET IN
LET := K_LET LETDEF
LETDEF := IDCALLEQ EXPR
IDCALLEQ := IDCALL EQ
IDCALL := I IDCALL | ID
LEXP := ID | INTLIT | | I LEXP | EXPRP EXPR | IEXPR LEXP | I EXPR | LPS RP | EXPR LEXP
EXPR := ID | INTLIT | I LEXP | EXPRP EXPR | I EXPR | IEXPR LEXP | LPS RP
EXPRP := EXPR P
IEXPR := I EXPR
LPS := LP S
I := ID
EQ := '='
IN := in
K_LET := let
LP := (
RP :=)
P := +

All tokens are reachable from the starting one, so this is the final solution.

b) [**5 pts**] Use the CYK algorithm and your normal form to parse the following sequence of four identifiers (ID tokens):

```
my cool compiler works
```

Show all parse trees that you obtain. The trees should have S in the root and cover the entire sequence of four tokens.

Solution

The CYK algorithm is a bottom-up parsing algorithm. First we parse one token at a time, then we regroup two consecutive derived non-terminals. ID can be parsed as:

```
S → ID
LEXP → ID
EXPR → ID
I → ID
```

ID ID can be parsed as:

```
S → I LEXP → ID ID
LEXP → I LEXP → ID ID
LEXP → EXPR LEXP → ID ID
LEXP → I EXPR → ID ID
EXPR → I LEXP → ID ID
EXPR → I EXPR → ID ID
IEXPR → I EXPR → ID ID
```

ID ID ID can be parsed as (ID ID) ID:

```
LEXP → EXPR LEXP → I LEXP LEXP → ID ID ID
        EXPR LEXP → I EXPR LEXP → ID ID ID
        EXPR LEXP → EXPR LEXP LEXP → ID ID ID
EXPR → IEXPR LEXP → I EXPR LEXP → ID ID ID
S → IEXPR LEXP → I EXPR LEXP → ID ID ID
```

and ID (ID ID):

```
S →I LEXP
     LEXP → I LEXP → ID ID
     LEXP → EXPR LEXP → ID ID
     LEXP → I EXPR → ID ID
S → I EXPR
      EXPR → I LEXP → ID ID
      EXPR → I EXPR → ID ID
      EXPR → EXPR LEXP → ID ID
LEXP → I LEXP →
          LEXP → I LEXP → ID ID
          LEXP → EXPR LEXP → ID ID
          LEXP → I EXPR → ID ID
LEXP → I EXPR
         EXPR → I LEXP → ID ID
         EXPR → I EXPR → ID ID
LEXP → EXPR LEXP
            LEXP → I LEXP → ID ID
            LEXP → EXPR LEXP → ID ID
            LEXP → I EXPR → ID ID
EXPR → I LEXP →
          LEXP → I LEXP → ID ID
          LEXP → EXPR LEXP → ID ID
          LEXP → I EXPR → ID ID
EXPR → I EXPR →
          EXPR → I LEXP → ID ID
          EXPR → I EXPR → ID ID
IEXPR → I EXPR
          EXPR → I LEXP → ID ID
          EXPR → I EXPR → ID ID
```

Finally, ID ID ID ID, starting from S, can be parsed as:

ID (ID ID ID)
S →I LEXP : 11 parse trees
S →I EXPR : 6 parse trees
(ID ID) (ID ID)
S → IEXPR LEXP : 3 parse trees
(ID ID ID) ID
S → IEXPR LEXP : 2 parse trees

Total: 22 parse trees