Compilation A
Exam

11.07.03

Don't Panic

- Read through all instructions and all questions before starting on the exam.
- You can write the solutions on the exam directly or on extra paper. If you use extra paper put your name and the number of the exercise on the paper.
- Write legibly, we will only give credits for answers that we can read.
- If you want a translation for a technical term, we will give it.
- There should be enough time, but do not stay very long on questions you cannot solve; you might miss an easy one at the end. There are 8 questions and 100 points, which mean a bit more than 10 min per question or 1 minute per point.
- As a first thing, put your name on the exam and check whether your exam is complete that is, whether you have all 8 questions.

Name: Maximilian Correct

Question	Max	Score
1.	15	15
2.	6	6
3.	15	15
4.	12	12
5.	10	10
6.	15	15
7.	17	17
8.	10	10
Subtotal	**100**	100
Project		
Part I	10	
Part II	20	
Part III	10	
Subtotal	**40**	
Total	**140**	

Cheat Sheet

Here is some information that might be useful when solving the exam questions.

BNF

In BNF a production (rule) consists of a left-hand-side (lhs) and a right-hand-side (rhs). The rhs consists of a (possibly empty) sequence of terminals and non-terminals. We use | for alternatives. We use juxtaposition for concatenation. Concatenation binds stronger than |.

Names introduced on the left-hand-side can be used on the right-hand-side. A BNF grammar can be recursive, that is, the right-hand-side of a production can refer to the symbol that is being defined in that production (or to another production which in turn refers to the symbol being defined).

EBNF

In addition to the BNF notation EBNF allows us to use:

- ( ) for grouping.
- ε for the empty word.
- [E] for one or zero occurrences of E.
- { E } for zero or more occurrences of E.

Regular expressions

A regular expression can be described by a number of EBNF rules without recursion. There are alternative ways of writing regular expressions. The most common is:

- E* for {E}.
- E? for [E].
- E+ for E{E}.
- [ac] for (“a” |”c”).
- [a-d] for (“a” |”b” |”c” |”d”).
- [^0-9] for anything but a digit.

JLex

In JLex we can use regular expressions as described above. In addition, names can be defined as:

```
Name = E
```

`We can refer to the name by writing:`

```
{Name}
```

Also, we can use the period (.) as shorthand for [^\n]

Name: Maximilian Correct

Exam (100p)

1) Regexps (15p)

Given the following specification adapted from Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000, *Section 2.3 Common Syntactic Constructs*

This section defines some symbols used widely in the grammar.

1. **White space** consists of one or more space characters (“ “), carriage returns (\r), line feeds (\n), or tabs (\t).
2. Characters are classified for convenience as letters, digits, or other characters. [...] [Definition: A **Name** is a token beginning with a letter, an underscore or a colon, and continuing with letters, digits, dots, hyphens, underscores, or colons, together known as **name characters**.] An Nmtoken (**name token**) is any mixture of name characters.

And this specification adapted from section 2.4 Character Data and Markup:

> The ampersand character (“&”) and the left angle bracket (“<”) may appear in their literal form only when used as markup delimiters. [Definition: All text that is not markup delimiters (“&”,”<”) constitutes the **character data** of the document.]

Assume you are given the definitions of `Letter` and `Digit`:

1. Write down a (JLex) regular expression for white space: (2p)

   `WhiteSpace` = (" " | "\r" | "\t" | "\n")+ *alternative solution:* [\r\t\n]+

2. Write down (JLex) regular expressions for tokens Name, Nmtoken. (4p)

   `NameChar` = `{Letter}|{Digit}|“.”|“-“|“_”|“:"`
   `Name` = ({Letter} | "_" | ":") {NameChar}*
   `Nmtoken` = {NameChar}+

3. Write down a (JLex) regular expression for one or more Names and one or more Nmtokens, separated by white space as defined in 1. (4p)

   `Names` = {Name} ({WhiteSpace} {Name})*
   `Nmtokens` = {Nmtokens} ({WhiteSpace} {Nmtokens})*

alternatively:

   Names = ({Name} {WhiteSpace})* {Name}
   Nmtokens = ({Nmtokens} {WhiteSpace})* {Nmtoken}

Note: Leading and trailing white space is not covered by the specification and leads to subtraction of points.

4. Write down a (JLex) regular expression for character data. (5p)

   `CharData` = [^&<]+

*Note: In JLex, braces {foo} denote **abbreviations** for given regular expressions. Nevertheless, we allow omitting them. Confused people who used the EBNF notation {X} instead of X* got only a little penalty, if the error is made consistently.*

Name: Maximilian Correct

2) Tokens (6p)

Split the following Java-code into a sequence of tokens. For each token note if it has any associated value, and in that case what the value is.

```
int max(int x, int y) { /* Return the max of two values */
   return (x>y) ? x : y;
}
```

INT ID(max) LPAR INT ID(x) COMMA INT ID(y) RPAR LBRACE
RETURN LPAR ID(X) GT ID(y) RPAR
QMARK ID(x) COLON ID(y) SEMI
RBRACE

Note that we don't want comments, line breaks or other white spaces as tokens, it makes parsing too difficult.

3) Let E be an eff (15p)

Write a *non-recursive* EBNF grammar for the language which consists of the characters 0, 1, and -, where each hyphen (-) is always followed by a digit, and where each sentence ends with a digit. Examples: 0, 01-001-1, -0-1 (6p)

G ::= E { E }
E ::= 0 | 1 | -0 | -1

Just G ::= { E } is not sufficient, since the empty string, not ending in a digit, would be accepted.

Alternative solutions:

G ::= {Digit | Hyph} | Digit
Hyph ::= - Digit
Digit ::= 0 | 1

G ::= { [-] (0|1) } | 0 | 1

Write a *recursive* EBNF grammar for the language which consists of a sequence of the character a followed by a sequence of the character b, where there is exactly two more a's than b's. Examples: aa, aaaabb, aaaaabbb. (9p)

G ::= aa AB
AB ::= a AB b | ε

Alternative solution:

G ::= a G b | aa

Note: The first symbol on the left hand side in a grammar is the start symbol. If you want to start with some other symbol you have to explicitly note that start symbol. No points have been deduced for this error.

Name: Maximilian Correct

4) XML documents (12p)

Here is a simplified grammar for XML documents in EBNF.

```
Document    ::= [Xmldecl] Element
Xmldecl     ::= '< ?xml version="1.0"? >'
Element     ::= STag Content ETag
STag        ::= '<' Name { Attribute } '>'
ETag        ::= '</' Name '>'
Attribute   ::= Name '="' CharData '"'
Content     ::= { CharData | Element }
Name        ::= ...
CharData    ::= ...
```

Rewrite this grammar in BNF, that is, without the use of "{ }" and "[ ]".

```
Document    ::= Element
              | Xmldecl Element
Element     ::= STag ETag
              | STag Content ETag
STag        ::= '<' Name '>
              | '<' Name AttrList '>'
AttrList    ::= Attribute
              | Attribute AttrList
Content     ::= CharData
              | Element
              | CharData Content
              | Element Content
```

Other rules are unchanged.

Note: This is a 100% epsilon-free BNF grammar that needs only one extra nonterminal (AttrList).

In general, an epsilon-free grammar exists for every grammar whose start-symbol does not derive the empty word. Solutions with epsilon are nevertheless accepted.

Name: Maximilian Correct

5) While in this scope (10p)

Given the following Java code for parts of a semantic analyzer which does not handle nested scope:

```
...
    // A symbol table implemented as a list of symbols.
    Symbol symtab = null;

    // Find the first occurrence of a name in a list of symbols.
    Symbol lookup(String name) {
      for(Symbol s = symtab; s != null; s = s.next) {
        if (name.equals(s.name)) return s;
      }
      return null;
    }

    // Add a new symbol to the head of the list of symbols.
    Symbol addSymbol(String name, Type type) {
      return symtab = new Symbol(name, type, symtab);
    }
...
    /** Analyze and attribute the tree node. */
    public Type analyze(Tree tree) {
      tree.apply(this);
      return tree.type;
    }
...
    public void caseWhile(While tree) {

      expectType(tree.cond.pos, Type.Bool, analyze(tree.cond));
      Symbol oldSymtab = symtab; // Save a pointer to the symbol tabel
      analyze(tree.body);
      symtab = oldSymtab; // Restore the pointer, thus forgetting any new symbols.
      tree.type = Type.NoType;

    }
```

How would you rewrite the handling of the while statement so that a new scope is entered in the body of the while loop? That is, any new symbols found in the body of the while should be forgotten after the analysis of the body.

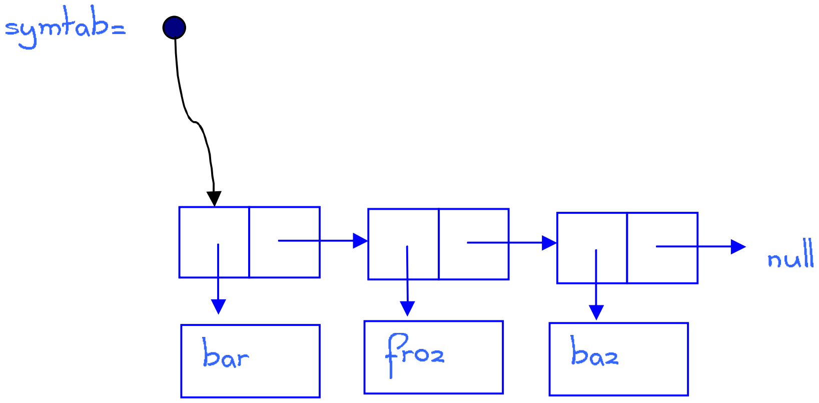
1. Add the necessary code to the caseWhile code given above.
2. Explain how the added code works. [Use the next page if you need.] See next page...

Name: Maximilian Correct

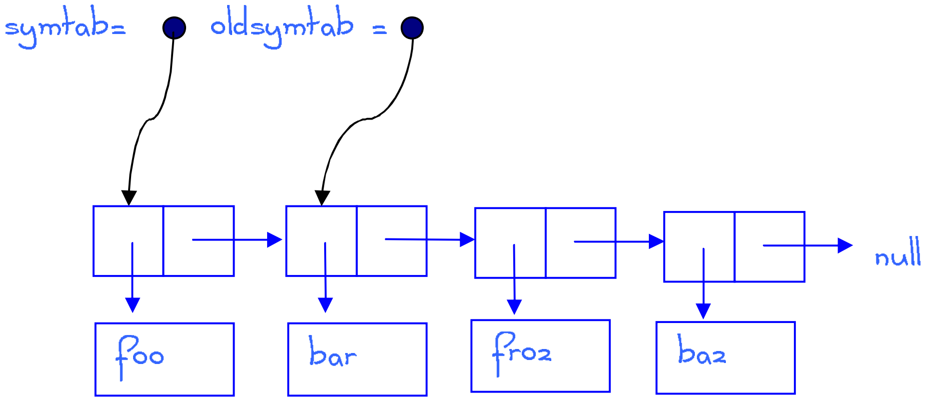
Since the symbol table is implemented by a linked list and symbols that we add to the list are inserted at the head of the list all we need to do is to remember where the head of the list was before analyzing the body. After the body is analyzed (which might add new local symbols to the head of the list) we can restore symtab by making it point to the old head. Example: Imagine we have some code like:

```
baz := ...
froz := ...
bar := ...
While (...) { foo := ...}
```

Before analayze(tree.body):



after analayze(tree.body):



By setting symtab to oldSymtab after tree.body has been analyzed we effectively "forget" the new symbol foo.

Name: Maximilian Correct

6) Take it from the top (15p)

Write a top down parser in Java for the following Grammar. The parser only needs to recognize the grammar; you do not need to construct an AST. You may assume that there are:

- A `nextToken()` function, which stores the next token in the variable `int token`.
- A function `error()` which signals an error in the input.
- That the integer constants `IDENT`, `LBRACKET`, `RBRACKET`, and `NUMLIT` are defined.

Grammar:

```
E ::= IDENT { LBRACKET E RBRACKET }
    |  NUMLIT
```

Code:

```
// Straight-forward implementation
class Parser1 implements Tokens {
   int token;
   void error(String errMsg) { /* prints the error message */ }
   void nextToken() { /* stores the next token in 'token' */ }

   void E() {
      nextToken();
      if (token == IDENT) {
          nextToken();
          while (token = LBRACKET) {
              E();
              nextToken();
              if (token == RBRACKET)
                  // problem: we loose the next
                  // token if it's not LBRACKET.
                  // we need to have a means of
                  // pushing back the token
                  // into the token stream
                  nextToken();
              else
                  error("'[' expected!");
          }
      } else if (token != NUMLIT)
          error("Identigier or number expected!");
      }
   }
```

```
//Alternative implementation:
// One token readahead;
// avoids the problem from the previous implementation
class Parser2 implements Tokens {
    // retrieve the first token
    int token = nextToken();
    void error(String errMsg) {}
    void nextToken() {}
    void E() {
        if (token == IDENT) {
            nextToken();
            while (token == LBRACKET) {
                nextToken(); E(); nextToken();
                if (token == RBRACKET) nextToken();
                else error("'[' expected!")
            }
        } else if (token == NUMLIT) nextToken();
        else error("Identifier or number expected!");
    }
}
```

Name: Maximilian Correct

7) Abstract Syntax Tree (17p)

Consider the following EBNF grammar:

```
E ::= IDENT { LBRACKET E RBRACKET }
   |   LPAR E RPAR
   |   NUMLIT
```

1. Design an abstract syntax for the grammar. (You may write this as a grammar with node-names (as in the lectures) or as Java or pseudo-Java code.) (14p)

Solution in Java:

```
class E {}
class IDENT extends E {
    String name;
    E[] args;
    public IDENT(String name, E[] args) {
        this.name = name;
        this.args = args;
    }
}
class NUMLIT extends E {
    int value;
    public NUMLIT(int value) {
        this.value = value;
    }
}
```

2. Given your abstract syntax draw the abstract syntax tree for the expression "`(x [y [4]] [2])`". (3p)

```
E expr = new IDENT("x",
                   new E[] {new IDENT("y",
                                      new E[] {new NUMLIT(4)}),
                            new NUMLIT(2)});
```

Name: Maximilian Correct

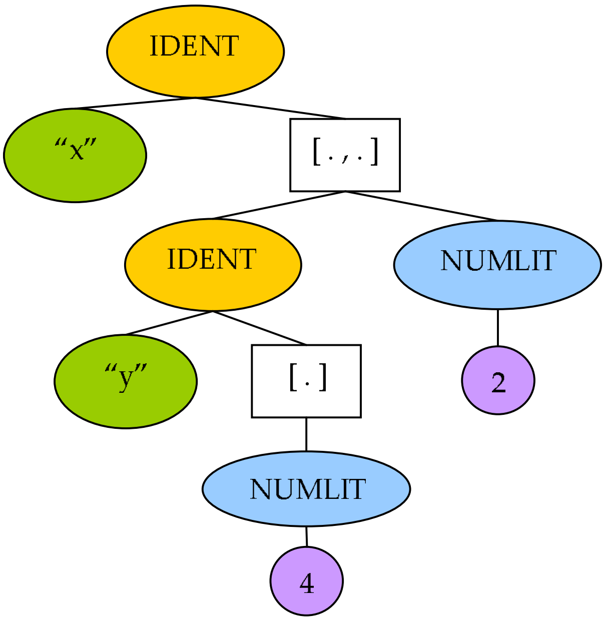
Alternative solution

1:

[E] denotes a list/array of Es

E ::= IDENT String [E]
    | NUMLIT Int

2:


IDENT
“x”
[. , .]
IDENT
NUMLIT
“y”
[.]
2
NUMLIT
4

Name: Maximilian Correct

8) Ad us proprium (10p)

If you feel that you have studied completely different subjects of the course than the ones we have asked you about, now is your chance!

1. Design an exam question. The question should be on an appropriate level of difficulty. (6p)
2. Answer your question. (4p)