Lehrstuhl Softwaretechnik & Programmiersprachen
Fakultät Wirtschaftsinformatik & Angewandte Informatik

SWT-PCC-M: Principles of Compiler Construction

Prof. Dr. Gerald Lüttgen and Dr. Eugene Yip Winter Semester 2020

Written Assignment / Hausarbeit

Strict deadline: 22 December 2020 at 12:00 CET

Completing the written assignment:

- Please read the assignment questions carefully before answering. Always briefly justify your answers and be as concise as possible.
- You shall answer all questions. Your answer for each question shall be written down on a new sheet of white A4 paper—not thin, transparent, or recycled paper, and not teared from a notebook—with your name and student number clearly marked on each page. You will lose marks if your answers are unreadable.
- The assignment has to be completed by each student individually. Any plagiarism or collusion will imminently lead to the loss of all marks of the assignment.

Submitting the assignment:

- The answers to the questions—**with a filled and signed copy of the *Ehrenwörtliche Erklärung* (see page 13) as the cover page**—have to be submitted via the module's VC course as a single PDF file.
- Your Lex and Yacc code for Question 6 shall also be submitted via the module's VC course, and shall be readable and executable with `lex` and `yacc`. This is in addition to submitting the Lex and Yacc code as an appendix to your PDF file.

Questions & answers:

- Questions concerning the understanding of this assignment will be answered, usually within three working days, via email to `eugene.yip@uni-bamberg.de` and copied (cc'ed) to `sekretariat.swt@uni-bamberg.de`. If a question is of general interest, it will be posted in the module's VC forum along with its answer. No questions concerning the assignment will be answered during the module's lectures and practicals.

Question 1: Scanning & Language Theory [10m]

Question 1.1 [2m]

Write down the DFA and the regular expression, over the alphabet $\Sigma = \{\texttt{a}, \texttt{b}\}$, that each recognise the strings `aa`, `aabb`, `aabbaa`, `aabbaabb`, `aabbaabbaa`, ...

Question 1.2 [2m]

Use *Thompson's construction* to produce an NFA that recognises strings described by the regular expression $\texttt{c}|\texttt{a}|\texttt{b}^*$.

Question 1.3 [3m]

Use the *subset construction* to transform the NFA you gave for Question 1.2 into an equivalent DFA. Be sure to show your working when calculating the new combined states.

Question 1.4 [3m]

Apply the DFA state minimisation algorithm to the DFA you gave for Question 1.3. Make sure you explain the reason why a set of states is partitioned for each step of the algorithm. Draw the DFA resulting from the minimisation algorithm.

Question 2: Top-Down Parsing [20m]

Consider the following grammar for statement sequences:

$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \texttt{\{}\ \textit{stmt-seq}\ \texttt{\}} \mid \texttt{id = } \textit{exp} \mid \texttt{other} \\ \textit{stmt-seq} & \rightarrow & \textit{stmt}\ \texttt{;}\ \textit{stmt-seq} \mid \textit{stmt} \\ \textit{exp} & \rightarrow & \textit{exp}\ \texttt{*}\ \textit{exp} \mid \texttt{num} \end{array}$$

Question 2.1 [2m]

The given grammar is unsuitable for LL(1) parsing. State why and revise the grammar to make it suitable for LL(1) parsing.

Question 2.2 [6m]

Construct the LL(1) parse table for the revised grammar of Question 2.1.

Question 2.3 [4m]

Use the parse table from Question 2.2 to perform an LL(1) parse of the following string:

```
{id = num * num; other; {id = num}}
```

The white spaces shall be ignored; they have only been added for better readability. Show the parse stack, the current input string and the action taken for each step of the parse.

Question 2.4 [2m]

Use the error handling heuristics from Exercise Sheet 9 of the SWT-PCC-M practicals to annotate the parse table from Question 2.2 with error actions.

Question 2.5 [6m]

Use the parse table from Question 2.4 to perform an LL(1) parse, with error handling, of the following string:

```
{id = other; ;
```

Show the parse stack, the current input string and the action taken for each step of the parse.

Question 3: Bottom-Up Parsing [20m]

Consider the following grammar that recognises addition with parentheses:

$$\begin{array}{lcl} A' & \rightarrow & A \\ A & \rightarrow & A \texttt{ + } B \mid \texttt{n} \\ B & \rightarrow & \texttt{(} A \texttt{)} \end{array}$$

Examples of strings recognized by this grammar are `n + (n) + (n)` and `n + (n + (n))`.

Question 3.1 [7m]

Construct the DFA of LR(1) items that could be used to parse strings of the above grammar. You may construct the DFA directly, or produce the NFA of LR(1) items first and then apply the subset construction.

Question 3.2 [3m]

Use the DFA from Question 3.1 to construct the LR(1) parse table.

Question 3.3 [3m]

Use the parse table from Question 3.2 to perform an LR(1) parse of `n + (n + (n))`. Show the parse stack, the current input string and the action taken for each step of the parse.

Question 3.4 [7m]

Use the error handling heuristics from Exercise Sheet 9 of the SWT-PCC-M practicals and the parse table from Question 3.2 to perform an LR(1) parse, with error handling, of the string `n + n + ( +`. Show the parse stack, the current input string and the action taken for each step of the parse. Justify each error handling action.

Question 4: Semantic Analysis [10m]

Consider the following grammar for statements and expressions. Each `id` token has an attribute *type* of value `Bool` or `Int` associated with it by default.

$$
\begin{aligned}
\textit{stmt-seq} &\rightarrow \textit{stmt} \mid \textit{stmt}\ \texttt{;}\ \textit{stmt-seq} \\
\textit{stmt} &\rightarrow \texttt{if (} \textit{exp} \texttt{) then } \textit{stmt} \mid \texttt{id = } \textit{exp} \\
\textit{exp} &\rightarrow \texttt{id} \mid \textit{exp}\ \texttt{*}\ \textit{exp} \mid \textit{exp}\ \texttt{>}\ \textit{exp} \mid \texttt{num} \mid \texttt{true} \mid \texttt{false}
\end{aligned}
$$

Question 4.1 [2m]

Draw the parse tree for the following input:

```
z = false; if (x > y) then x = 2 * x
```

Question 4.2 [5m]

Construct attribute equations for this grammar to ensure that an input type-checks correctly. Assignment is only permitted between two identical types, multiplication and comparison is only permitted between expressions of type `Int`, and any expression used to determine control flow must be of type `Bool`.

Associate an attribute *ok* with the *stmt-seq* symbol, which is *true* if all statements in the sequence type check correctly, and *false* otherwise. You may add any additional attributes you wish to your attribute grammar.

Question 4.3 [3m]

Annotate your parse tree from Question 4.1 with attribute information using your attribute grammar. The types of the `id`s in the input are as follows: `x`.*type* = `Int`, `y`.*type* = `Int`, and `z`.*type* = `Bool`. Draw arrows on your parse tree to indicate dependencies. Give an ordering of the nodes that respects the attribute dependencies.

Question 5: Code Generation [20m]

Consider the C program below, which reads in values until ten have been greater than k, and writes them out. Assume that the data types have the following sizes: 8 bytes for addresses and temporaries, and 4 bytes for `int`.

```
 1 int a[10];
 2
 3 void print(void) {
 4     int i;
 5     for (i = 0; i < 10; i++) {
 6         write(a[i]);
 7         /* HERE 2 */
 8     }
 9 }
10
11 void f(int k) {
12     int i = 0;
13     while (i < 10) {
14         int u = read();
15         if (u < k) {
16             a[i] = u;
17             i++;
18             /* HERE 1 */
19         }
20     }
21     print();
22 }
23
24 void main(void) {
25     f(5);
26 }
```

Question 5.1 [2m]

Draw the content of the symbol table at code line 17, marked `HERE 1`, as would be seen during the semantic analysis phase. Clearly indicate where new scopes begin.

Question 5.2 [6m]

Using the notation from the lectures for a *stack-based runtime environment with no local procedures*, draw a diagram for the state of the runtime environment when execution reaches code line 7, marked `HERE 2`. Include the global variable area and the code area in your diagram (but not the generated code). Give the values for all variables (use '`?`' in the case of unknown values), and indicate any pointer by drawing an arrow in your diagram. Moreover, each function uses one local temporary. Include these temporaries in your diagram.

Question 5.3 [2m]

Describe how array element `a[2]` may be accessed from within function `f`.

Question 5.4 [5m]

Give the three-address code for function `f`.

Question 5.5 [5m]

Give the P-code for function `print`.

Question 6: Lex & Yacc [20m]

Question 6.1 [10m]

Write a Lex-based scanner and a Yacc-based parser that decompiles a sequence of P-code back into a single high-level assignment statement, which may consist of variable names, numbers, arithmetic operators, and parentheses. Variable names are strings of alphabetical characters, and numbers are sequences of digits. Your decompiler shall only consider the following P-code instructions:

```
lda <variable>   // Load the address of <variable>
lod <variable>   // Load the value of <variable>
ldc <number>     // Load the constant <number>

// The following instructions are defined with operand A at
// the top of the stack, and operand B second from the top.
sto   // B = A, store the value A at address B
mpi   // A * B, integer multiplication
dvi   // A / B, integer division
sbi   // A - B, integer subtraction
adi   // A + B, integer addition
```

For example, the high-level assignment statement “`x = (y + 2) * 3`” shall be decompiled from the following P-code:

```
lda x
ldc 3
ldc 2
lod y
adi
mpi
sto
```

Use the code skeletons provided on pages 10 and 11 (also available electronically via the Virtual Campus) to implement the Lex and Yacc parts. The code skeletons provide some glue code for reading from a file instead of `stdin`, and is configured to work with a Lex-generated scanner. You may find the following sequence of commands useful when compiling your Lex and Yacc parts into a single executable, e.g., named `Decompile`:

```
yacc -d yacc_template.y
lex lex_template.lex
gcc -o Decompile y.tab.c lex.yy.c

./Decompile example.txt
```

Question 6.2 **[10m]**

Augment your scanner and parser of Question 6.1 such that it handles lexical, syntactic, and semantic errors, and prints error messages with the following details: the error type (lexical, syntactic, or semantic), position of the error in the input file, the offending token or input string, and the expected tokens if applicable. Ensure that you adhere to the *"Try to parse as much as possible"* principle, while suppressing as many spurious error messages as possible. Additionally, the decompiled assignment statement shall only be printed if no errors were encountered.

For example, consider the following P-code:

```
ldc x
lod y5
adi
sto
```

The possible error messages could be the following:

```
Error 1 (line 1, characters 5-5):
syntax error, unexpected variable, expecting number.

Error 2 (line 2, characters 5-6):
lexical error, unrecognised input string `y5`.

Error 3 (line 4, characters 1-3):
semantic error, no operand available.
```

Information on Yacc error handling and recovery is available in the following sections of the online Bison manual (`https://www.gnu.org/software/bison/manual/html_node/`):

- Section 3.7.2: Token Type Names
- Section 4.7: The Error Reporting Function `yyerror`
- Section 6: Error Recovery

Lex Code Skeleton for Question 6

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>
    #include "y.tab.h"

    // This function is defined in your Yacc code
    extern int yyerror(const char *s);

    // Insert possible includes and C declarations here
%}

// Insert scanning options and definitions

%option noyywrap


%%

// Insert lexing rules


%%
```

Yacc Code Skeleton for Question 6

```
%{
    #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

    extern FILE *yyin;
    int yylex(void);
    int yyerror(const char *s);

    // Insert possible includes and C declarations here
%}

// Insert possible tokens, type information, etc, here
%locations

%%

// Insert grammar with corresponding actions here

%%

// Main function to parse from a file, specified as a parameter
int main (int argc, char *argv[]) {
    // Open a file handle to the user's specified file
    FILE *myfile = fopen(argv[1], "r");

    // Check that the file can be opened
    if (!myfile) {
        printf("I cannot open %s!\n", argv[1]);
        return -1;
    }

    // Set Lex to read from the file, instead of STDIN
    yyin = myfile;

    printf("\n");
    int parseResult = yyparse();
    printf("\n");

    return parseResult;
}

// Error handling function
int yyerror(const char *s) {
    // Define your own error handling messages here
    fprintf(stderr, "%s\n", s);

    return 0;
}
```

SWT-PCC-M: Principles of Compiler Construction

Prof. Dr. Gerald Lüttgen and Dr. Eugene Yip — Winter Semester 2020

Written Assignment / Hausarbeit

Ehrenwörtliche Erklärung

Ich habe die vorliegende Hausarbeit einschließlich der digitalen Abgabe selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt.

Matrikelnummer	Name
Ort/Datum	Unterschrift

Q1	**Q2**	**Q3**	**Q4**	**Q5**	**Q6**	**Σ**
/ 10	/ 20	/ 20	/ 10	/ 20	/ 20	/ 100

Gesamtnote	Datum
Prüfer	Unterschrift