Compilers (L.EIC026)

Spring 2022

Faculdade de Engenharia da Universidade do Porto
Departamento de Engenharia Informática

Midterm Exam with Answers

April 20, 2022 from 17:30 to 19:30

Please label all pages you turn in with your name and student number.

Name: _____ Number: _____

Grade:

Problem 1 [10 points]:
Problem 2 [40 points]:
Problem 3 [25 points]:
Problem 4 [25 points]:

Total:

INSTRUCTIONS:

1. This is an open-book exam but you may bring one A4 sheet with notes either typed or handwritten for your own personal use during the exam.

2. Please identify all the pages where you have answers that you wish to be graded. Also make sure to label each of the additional pages with the problem you are answering.

3. Use black or blue ink. No pencil answers allowed.

4. Staple or bind additional answer sheets together with this document to avoid being misplaced or worse, lost. Make sure this cover page is stapled at the front.

5. Please avoid laconic answers so that we can understand you understood the concepts being asked.

Problem 1 [10 points] Consider the regular expression below where symbol `id` stands for an identifier token. The various punctuation characters, like '`)`' are indicated between quotes which are not part of the matched words, but only part of the specification. The concatenation operator is explicitly indicated with the "." character.

`'(' . ( (id) . (',' . (id))* )? . ')'`

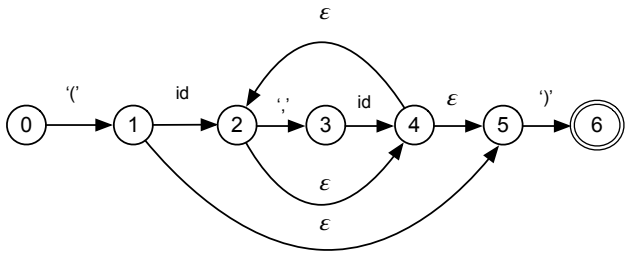
As an example, language generated by this regular expression includes "`()`" and "`(id,id,id)`".

Considering this regular expression, answer the following:

a) Convert the RE provided into an NFA using the Thompson construction (you are allowed some simplifications of the construction to reduce the number of states on ϵ-transitions).
b) Convert the NFA derived in section b) to a DFA using the subset construction. Show the mapping between the states in the NFA and the resulting DFA.
c) Argue that the DFA obtained in b) above is either minimal or not.

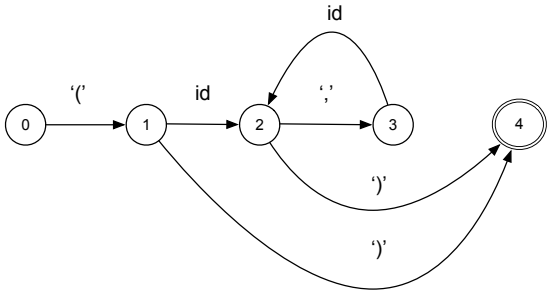
Solution:

a) The provided regular expression can be captured by the NFA below which already includes some state simplification related to epsilon transitions.



b) The DFA below results from the application of the Subsect construction on the NFA depicted in a) above. Here the mapping of states from DFA to subsets of state of the original NFA is shown on the left-hand-side.

DFA	NFA
0	{ 0 }
1	{ 1, 5 }
2	{ 2, 4, 5 }
3	{ 3 }
4	{ 6 }



c) The path from start 0 (the starting state) and state 4 (the unique final state) spells the simples case of '`()`', whereas the state 2 is required for the case of single id in parenthesis. The subsequent case of multiple pairs of ", `id`" are captured by the loop defined by states 2 and 3, so all states are clearly required. Note again, that missing transitions are implicit transitions to an error state.

Problem 2 [40 points] Consider the following CFG grammar G,

$S \rightarrow a\ S$
$S \rightarrow a\ b\ S$
$S \rightarrow c$

where 'a', 'b' and 'c' are terminals, and 'S' the single non-terminal and start symbol. For this grammar, determine the following:

a) This G does not have the LL(1) property, i.e., if it not parseable using the LL parsing algorithm described in class with a single lookahead token. Why?
b) Provide a remedy for G below so that it has the LL(1) property.
c) Considering the original G grammar. Can the common prefix issue be solved using additional lookahead tokens (analogous to the mechanism used by the JavaCC21 parser), say with 2 lookahead tokens?
d) Build the LL parsing table for the modified grammar you defined in b) as discussed in class by explicitly computing the FIRST, FOLLOW and NULLABLE sets for the various non-terminals and productions.

Solution:

a) No. The first two productions have a common terminal prefix, so the FIRST sets corresponding to those two productions have a non-empty intersection. In other words, given the parser in the position of expanding the non-terminal symbol S, and given the input toke as the 'a' terminal, the parsing algorithm will not be able to decide which of the two productions to use (to expand).

b) We can eliminate the common prefix in this grammar by the introduction of another non-terminals, namely T which is responsible for the sequences of 'a' characters. The revised grammar below is a possible solution (among many).

$S \rightarrow a\ T$
$S \rightarrow c$
$T \rightarrow b\ S$
$T \rightarrow S$

c) Yes, with a lookahead of 2 token you can choose at the beginning which of the two productions to use in case the input is simply "ab".
d) We begin by computing the FIRST, FOLLOW and which non-terminals are Nullable. Then build the LL parsing table and check that the resulting table has no multiply-defined entries.

FIRST(S) = {a, c}, FOLLOW(S) = {$}
FIRST(T) = {a, b, c}, FOLLOW(T) = {$} and T. S are non-nullable

The LL parsing table is as follows:

	a	b	c
S	S → a T		S → c
T	T → S	T → bS	

Problem 3 [25 points] Consider a language with **int** and **float** types and an attribute grammar with rules for carrying out the type check of arithmetic expressions as illustrated in the snippet example below. Here the "**+**" addition operator is overloaded for integers and numbers in floating point.

Grammar Production:

exp_1 -> exp_2 "**+**" exp_3

Semantic Rule:

if *(*exp_2*.type* == exp_3*.type)* **then** exp_1*.type* = exp_2*.type* **else** *type_error(*exp_1*)*

Questions:

a) As you can see, the rule will raise a type error whenever there is a type mismatch between the associated types for exp_2 and exp_3. Change the semantic rule to enable implicit coercion of **int** to **float** when needed.

b) Add a **bool** type for Booleans, and a production for boolean expressions of the form (exp_1 **or** exp_2). Assuming that integers zero and non-zero may be overloaded to represent either an integer value or the Boolean constants False and True, respectively, define new semantic rules to type check expressions consisting of a number, expressions of the form (exp_2 "+" exp_3) and expressions of the form (exp_1 **or** exp_2). Assume that in a Boolean expression, when you mix Booleans and integers the resulting type is Boolean. In the generated code, an explicit test will be inserted to check if the actual run-time value is either zero (false) or non-zero (true).

Solution:

a) The rule needs to check if, in the case of a type mismatch, one of the types is an integer and the other a float. If that is the case, the rule will "promote" the integer type to a floating point and assign the result's type as a floating point.

Grammar Production:

exp_1 -> exp_2 "**+**" exp_3

Semantic Rule:

if *(*exp_2*.type* == exp_3*.type)* **then**
  exp_1*.type* = exp_2*.type*;
else
  if (*(*exp_2*.type* == int*)* **and** *(*exp_2*.type* == float*)) **then***
    exp_1*.type* = *float*;
  else
    if (*(*exp_2*.type* == float*)* **and** *(*exp_2*.type* == int*)) **then***
      exp_1*.type* = *float*;
    else
      type_error(exp_1*)*;

b) In this case we need to check two things. First, if there are identical types, whatever they are, we still need to check if they are either integer or Boolean. Second, if there is a mismatch, if they can be coerced. Lastly, if they cannot be coerced, an error must be raised.

Grammar Production:

exp_1 -> exp_2 "**or**" *exp_3*

Semantic Rule:

if *(exp_2.type == exp_3.type)* **then** // identical types case
  if (*(exp_2.type* != *int)* ***and*** *(exp_2.type* != *bool))* **then**
    type_error(exp_1);
  else
    exp_1.type == exp_2.type // same type, whatever that is
else // different types cases
  if ((*exp_2.type == int)* ***and*** *(exp_3.type == bool*)) **or**
    ((*exp_2.type == bool)* ***and*** *(exp_3.type == int*)) **then**
      exp_1.type == bool;
  else
    type_error(exp_1);

Problem 4 [25 points] In the PASCAL language one can define functions and procedures that are local to other procedures, *i.e.,* they are only visible within the scope of that immediately enclosing procedure. This is analogous to the use of nested blocks in the C language. For example, in the code below procedure `f3` is only visible inside the code of procedure `f2`. But not inside the code of `f1` or `main`.

```
01:  procedure main
02:    integer a, b, c;
03:    procedure f1(a,b);
04:      integer a, b;
05:      call f2(b,a);
06:      end;
07:    procedure f2(y,z);
08:      integer y, z;
09:      procedure f3(m,n);
10:        integer m, n;
11:      end;
12:      procedure f4(m,n);
13:       integer m, n;
14:      end;
15:     call f3(c,z);
16:     call f4(c,z);
17:     end;
18:  ...
19:  call f1(a,b);
20:  end;
```

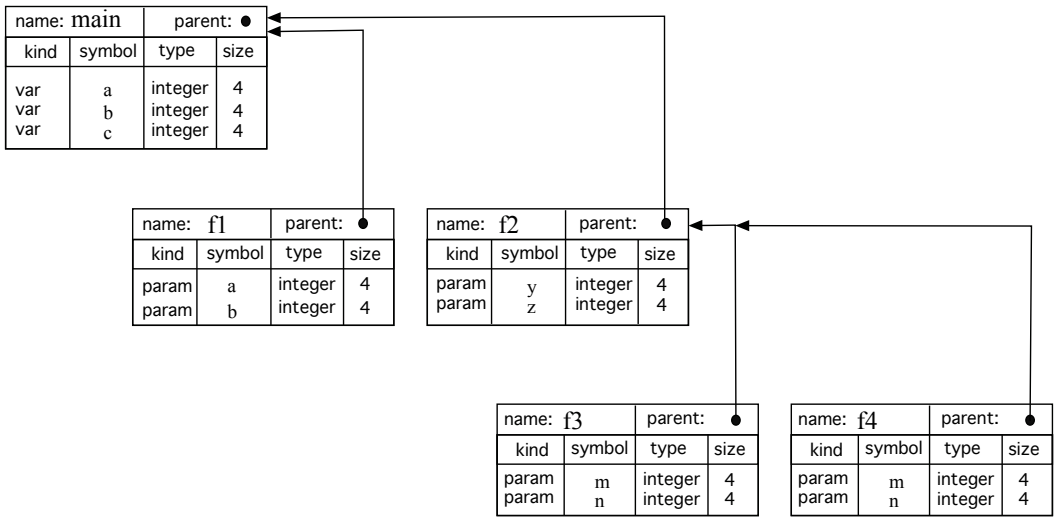
For this specific code below answer the following questions:

a) Draw the symbol tables for each of the procedures in this code (including `main`) and show their nesting relationship by linking them via a pointer reference in the structure (or record) used to implement them in memory. Include the entries or fields for the local variables, arguments and any other information you find relevant for the purposes of code generation, such as its type and location at run-time.

b) For the statement in line 15 what are the specific instance of the variables used in this statement the compiler needs to locate? Explain how the compiler obtains the data corresponding to each of these variables table.

Solution:

a) The figure below depicts the internal data and relative organization of the symbol tables related to the various functions and `main` procedure in this program.



b) For the statement in line 15, the symbol "`c`" refers to the scalar variable in the `main` procedure, whereas the symbol "`z`" refers to the scalar variable in the `f2` procedure. The compiler uncovers which procedure variable or parameter a given symbol corresponds to by traversing the tree of symbol tables up to the "root" in this case the symbol table of the `main` procedure.