Compiler Construction

Take-home test 1, 2018–2019

(updated 2019-05-20 10:00)

Submission deadline: Friday, 24 May 2019, 23:59 CEST

The take-home tests are to be done individually. You may discuss problems *and exchange* ideas *freely, but you must not share* solutions *(algorithms, code, text, drawings, etc.): do not* show *your solutions to anyone and do not* look *at other students' solutions.*

In packaging and submitting your solutions, please adhere to the following rules (which are meant to lighten the task of assessment). Failure to do so will result in the deduction of points.

- Submit your solution on CANVAS (also if you are late). You may submit multiple times, but *only the last submission will be assessed* (and its submission time will be used to deduct any points if you are late).
- Put all your files into a single .zip archive (use the .zip format, not .tar, .rar, or any other format). Loose files will not be assessed.
- Combine your textual answers into *one* text or PDF file named `answers.txt` or `answers.pdf`. Your name and student number must be included in the first line of the text of this answers file. Do not use any other formats: If you must use Word, convert to PDF. If you use LaTeX, submit only the generated PDF.
- You can create illustrations or drawings by (a) drawing them using a drawing program (such as Inkscape or GraphViz) or drawing package for LaTeX(such as TikZ), or (b) drawing them by hand on paper. You can include your drawings in your submission by (c) making them part of your PDF or (d) submitting them as separate (scanned or generated) PDF or JPEG files (but not in any other format) that you reference from your answers file. Any combination of (a) or (b) with (c) or (d) is allowed.
- Put all your classes, grammars and other programming resources into Java packages `pp.s1234567.q1_X`, replacing `1234567` by your own student number and `X` by the question number. You may reuse predefined classes from the lab sessions, provided they are completely unchanged; otherwise, copy them into your package. Reference the electronic resources from the text or PDF file with your answers.
- Any resources in separate files (inside the submitted .zip file) that are not mentioned in your answers file will not be assessed.

In case of late submission, the grading rules described on CANVAS apply.

Question 1 *(10 points)*

ANTLR is not the only tool that turns scanning rules and grammars in BNF-like notation into executable scanners and parsers. Find one other parser generator, which must not be LL(1), and compare it to ANTLR.

In particular,

- briefly describe syntactical differences in how scanner and parser rules are specified for the other tool compared to ANTLR,
- say what class of grammars the tool supports (and if that class is different from ANTLR's, give and explain a short example of a grammar that is supported by one but not the other tool),
- list the programming languages supported by the tool,
- say what type of scanners and parsers it generates (e.g. table-driven, recursive descent, ...),
- briefly present other features that the other tool has but ANTLR doesn't (if any).

Provide references (e.g. to specific pages of the tool's website or online documentation, or to research papers describing the tool) for all of your statements. □

Question 2 *(20 points)*

Ada is a statically typed, imperative, object-oriented programming language designed specifically for safety-critical and highly reliable systems. The *Ada language reference manual* is available at adaic.org.

1. Use the "Consolidated Ada 2012 Language Reference Manual (LRM)" to find out how number literals (i.e. the Ada equivalent of tokens like `0xFA56` in the statement `int i = 0xFA56;` in Java) are written in Ada. Where in the manual do you find this information (give a reference to the subsection and page number)? What format or formalism does it use? *(2 points)*
2. Give a DFA that recognises, or a regular expression that describes, *integer* number literals in Ada. For regular expressions, use the notation from the lecture slides. *(6 points)*

   Your DFA or regular expression does not *have to* enforce the last sentence of the "legality rules" for based literals given in the reference manual. If indeed it does *not*, briefly explain how you would need to change it to enforce this requirement. *(2 points)*
3. Write an ANTLR lexer grammar defining a `NUMBER` token type that implements the Ada integer number literal syntax. Use **`fragment`**s to keep your grammar readable (the grading will take the readability of your definition into account). *(6 points)*

   Write a (comprehensive) JUNIT test showing that your solution accepts and rejects input correctly. In particular, include all the (integer) examples listed in the reference manual. *(2 points)*

   Again, your lexer grammar does not *have to* enforce the last sentence of the "legality rules" for based literals. Again, if indeed it does *not*, briefly explain how you would enforce that rule if you were to use your lexer grammar in a larger parser for Ada(-like) programs. *(2 points)*

□

Question 3 *(35 points)*

Let us consider a simple language for list expressions. It has two kinds of values: digits and lists.

- A single digit is a valid list expression.
- A list is a valid list expression; it is written as a comma-separated sequence of list expressions enclosed in `[]`.

For example, `[1, 2, 3]` is the list containing the digit `1` as its first element, `2` as its second element, and `3` as its third element. `[]` is an empty list. `[[1], 2]` is the list containing the list containing digit `1` as its first element, and `2` as its second element.

Further list expressions can be constructed with the following operations:

`::` is a right-associative binary (infix) operator whose right operand must be a list. Its result is a new list whose first element is the left operand, with the following elements being the elements of the right operand. For example, `1::[2, 3]` results in the list `[1, 2, 3]`, and `[1]::[2]` results in the list `[[1], 2]`.

`+` is a left-associative binary (infix) operator whose operands must be lists. Its result is the concatenation of the two lists. It has lower precedence than `::`. For example, `[[1]] + [2]` results in the list `[[1], 2]`, and `[1] + 2::[3]` results in the list `[1, 2, 3]`.

`hd` is a right-associative unary (prefix) operator whose operand must be a non-empty list. It has higher precedence than `::`. Its result is the first element of its operand. For example, `hd [1, 2, 3]` results in the digit `1`, and `hd [[1], 2]` results in the list `[1]`.

`tl` is a right-associative unary (prefix) operator whose operand must be a non-empty list. It has the same precedence as `hd`. Its result is its operand with the first element removed. For example, `tl [[1], 2]`, `tl([1]) + [2]` and `tl [1] + [2]` all result in the list `[2]`; `tl [1]::[2]"` results in the list `[[], 2]`, and `tl [3]` results in `[]`.

As usual, parentheses `( )` can be used to delineate subexpressions, just like in the usual arithmetic expressions.

1. We defined `::` to have a higher precedence than `+`. Why can we not define both operators to have the same precedence? What would we need to change in our specification such that we could define them to have the same precedence? *(4 points)*
2. Give an unambiguous grammar in BNF (as used in the lecture) for the language of all list expressions. Use a start symbol `start` that does not occur on any right-hand side in your grammar. The grammar shall be such that the structure of its parse trees matches with the specified precedence and associativity rules. Use the token types `HD`, `TL`, `CONS` (for `::`), `APPEND` (for `+`), `LPAR` (for `(`), `RPAR` (for `)`), `LBRACKET` (for `[`), `RBRACKET` (for `]`), `COMMA` (for `,`), and `DIGIT` (for digits) as terminals. *(8 points)*

   Your grammar should not enforce typing requirements such as the need for certain operands to be lists or non-empty lists; e.g. `1 :: 2`, `tl []`, `hd 4` or `[5] + 6` should be valid list expressions according to your grammar (although they are not correctly typed).

   Note: You can take a look at the ANTLR grammar given for Question 4, but that grammar is not in BNF, and it uses ANTLR-specific features and rules (which are not part of BNF) to achieve the specified precedence and associativity. Your grammar must not rely on such features and rules.
3. Draw the parse trees for `hd hd [[1]] :: [2]` and `[1, 2] + [3] + [4]` according to your grammar. *(4 points)*

4. Is your grammar an LL(1) grammar? If yes, prove that it is. If not, say why not, transform it into an LL(1) grammar (keeping the start symbol `start` that does not occur on any right-hand side), and prove that the resulting grammar is LL(1). *(10 points)*

   You need to compute FIRST+ sets for your proof. Show the FIRST and FOLLOW sets that you use to derive the FIRST+ sets. Explain the conclusion that you draw from the FIRST+ sets that you compute.

5. Draw the parse tree for `[1] + [2] + [3]` according to your transformed grammar. Does it match with the specified associativity rules? *(3 points)*

6. Let us assume that we build a recursive descent parser for your transformed grammar where each function returns its corresponding parse(sub-)tree. Give pseudocode for the function(s) in such a parser that handle the `+` operator. The function(s) shall construct and return parse trees which match the specified associativity rules (i.e. they must reconstruct the correct parse tree in case your answer in subquestion 5 above was "no"). *(6 points)*

□

Question 4 *(35 points)*

Let us continue with the language for list expressions from the previous question, but excluding the `hd` and `tl` operators, using the following ANTLR grammar (also on CANVAS as file `ListExp.g4`):

```
grammar ListExp;

start : listExp EOF;
listExp : <assoc=right> listExp '::'  listExp # cons
        | listExp '+' listExp # concat
        | '[' ( listExp ( ',' listExp )* )?  ']'  # list
        | '(' listExp ')' # paren
        | DIGIT # digit
        ;

DIGIT : '0'..'9';
WS : [ \t\r\n]+ -> skip;
```

Your task is to implement type checking and inference for this language. Its types are $Digit$ and $List(n)$ with $n \in \mathbb{N}$. A digit has type $Digit$. Let $ElemTypes(\ell)$ be the set of all the types of the elements of list ℓ. Then the type of ℓ is $\max(\{0\} \cup \{n \mid List(n) \in ElemTypes(l)\}) + 1$, i.e. the n in $List(n)$ indicates the maximum nesting depth.

For example, the type of `[1, 2]` is $List(1)$, the type of `[1, [2], 3]` is $List(2)$, and the type of `[1, []::[]]` is $List(3)$, while list expressions `[] + 2` and `[3] :: 4` are incorrectly typed.

1. Use ANTLR tree listeners to implement type checking and inference for the list expression language, in file `TypeChecker.java`.

   Give a (comprehensive) JUNIT test in file `TypeCheckerTest.java` showing that your solution accepts well-typed and rejects incorrectly typed expressions correctly, and that it infers the correct type for well-typed expressions. *(10 points)*

   Bonus: Generate user-friendly error messages in case of incorrect types. A message shall indicate what the error is and where in the input it occurs. Provide a demo program showcasing your error messages for different operators. *(5 extra points)*

2. Describe what you would need to change in your type checker to support the `hd` and `tl` operations. *(5 points)*
3. Add support for variables, which are written as a single ASCII letter (`a..z`), to the grammar and to your type checker, as described below. Use a symbol table. *(15 points)*
   - Variable references (which are syntactically just the name of the variable) can occur wherever a digit can occur; for example, `[1, a::[3, 4]]` includes a reference to variable `a`.
   - Variables can be declared and defined in *program expressions*, which are similar to `let..in` expressions in HASKELL, but use a different syntax: They start with a curly bracket ({), contain zero or more variable declarations of the form $variable\text{-}name$ = $listExp$`;`, followed by a single list expression, and a closing curly bracket (}). The result of a program expression is the result of its (final) list expression, using the defined variable values. A program expression is a list expression.
     For example, { `[1, 2]` } is a program expression (with zero variable declarations) that results in the list `[1, 2]`, and { `a = 1, b = [a], a::b` } is a program expression with two variable declarations that results in the list `[1, 1]`.

   A variable declared in a program expression is visible to, and can be referenced in, all list expressions *inside* the same program expression that *follow* after the end of the variable's definition. In particular, it can be used in nested program expressions, but not outside of its defining program expression. A variable may be redeclared in nested list expressions; in such a case, a variable reference refers to the "most recent" declaration, i.e. the first one found in an enclosing program expression when moving up in the parse tree from the reference. Among the declarations of a single program expression, a variable may be declared at most one. That is, the program expression { `a = [4]; b = { a = 5; a }; b::a` } is a valid list expression (and results in the list `[4, 5]`), but { `a = 4; a = 5; a` } is incorrect due to the redeclaration of `a`.

   The following is another example of a valid list expression that includes program expressions:

   { `a = 2; b = [1] + [a]; b + 3 :: [{ a = 4; a }, 5]` }

   It results in the list `[1, 2, 3, 4, 5]`.
4. ANTLR is a so-called LL(*) parser. Do you think that it can parse your grammar from the previous subquestion with lookahead 1? Say why you think so or why not. *(2 points)*
5. What is the runtime of lookups in your symbol table in subquestion 3 above? Can you make it deterministic constant time? If yes, briefly describe how; if no, say why not. *(3 points)*

□