UNIVERSITY
OF TURKU

Advanced Programming Techniques
Programming exercise

Programming exercise
Instructions
March 11, 2020
13 p., 1 app. p.

University of Turku
Department of Future Technologies
Software engineering
Advanced Programming Techniques

Contents

1 Introduction

This exercise complements the mainly theoretical aspects of the Advanced Programming Techniques (DTEK8101) course. The assignment is suitable for either a single person or for a group of 2–4. Since the course offers a diverse set of topics, you can pick one that seems most interesting and rewarding for you.

There are three categories of exercises to choose from. The first category (A) covers language compilation, the second (B) language interpretation and runtime issues, and the third one (C) involves building a small application in one of the "non-mainstream" languages & paradigms covered on the course. The last category contains multiple alternatives (games) to choose from. The first two are not recommended for a single person unless you are willing to use more time on the project than originally scheduled for the exercise. Only pick one assignment from one of the categories as your exercise project.

1.1 Assessment

The exercise descriptions list the goals and methods for the task and the additional requirements that depend on the size of the group. The implementations are graded using a scale from 0 to 6, based on how well each of the functional goals have been achieved. The remaining 3 points are rewarded based on the depth, accuracy, and insights of the documentation. 90% of the requested functionality and documentation results in full 9 points, otherwise a linear scale is used for determining the points. For groups of 2–4 people, by default all members in the group receive an equal number of points.

1.2 Schedule

The hard deadline for the projects is 30th June 2020 (2020/06/30).

1.3 Submitting the results

Either return the whole project & documentation as a zip package via Moodle or return only the documentation and provide a link to the project's git repository (if using utu's gitlab and using a private repository, make sure you add the teacher as a developer to the project, otherwise the code may not be visible).

1.4 Implementation languages

For exercises A and B, any implementation language can be picked, but the suggested language ecosystem (Java, Scala or any other JVM language) may require less effort due to the amount of existing, supporting code.

1.5 Custom topics

You may also suggest a custom topic for the exercise. The main requirement is that the project should somehow reflect the theory presented on the course.

1.6 Further questions

Feel free to ask `jmjmak@utu.fi`.

2 Category A: compilation

A simple Turing complete imperative language MiniImp was introduced on the second lecture. The language's preliminary constructs are explained in the lecture slides and the implementation is available from two git repositories.

This exercise assignment defines few further language constructs, which extends the language into **MiniImp+**. The goal is to compile programs written in MiniImp+ into source code of some existing language X (source to source compilation). The quality of the implementation is assessed by compiling a MiniImp+ demo application to X, then by successfully compiling the resulting code into an executable which behaves as intended.

In short, the main tasks of this exercise are:

1) extend the **MiniImp** parser into **MiniImp+**
2) implement code generation **MiniImp+** $\rightarrow$ **X**
3) implement a test program **T** in **MiniImp+**
4) assure that **T** compiles via **X** to an executable and behaves as intended
5) document the implementation

2.1 Existing code base

One can implement the exercise from scratch (even using a different language and paradigm), but a set of sample code is offered to guide and speed up the development.

The first implementation is a generic simple ANTLR project that just generates the lexer/parser classes form the grammar description:

- gitlab.utu.fi/tech/education/apt/miniimp

The second repository uses a more complex build system that automatically fetches the dependencies, offers a graphical (JavaFX) user interface, and shows how to build a parser & AST visitor (the fi.utu.apt.parsers.antlr classes). It also includes another parser written using a PEG parser framework (the fi.utu.apt.parsers.parboiled classes):

- gitlab.utu.fi/tech/education/apt/parser-experiments

2.2 Required language constructs

The existing implementations cover the parsing of:

- literals: numbers, truth values
- arithmetic operators: +, -, *, /
- boolean operators: not
- equivalence test: is
- control flow constructs: while, if
- I/O: write
- variable (re)definitions and references

The result of this exercise should also accept the following language constructs:

- boolean operators: and, or
- literals: strings
- type casts: from int to string, from string to int
- I/O: read (results in a string value)

2.3 Semantics and code generation

In this exercise we can skip formal definitions and proofs of the semantics and just assume that the implementation is the definition. It is sufficient if the demo program runs and otherwise the language seems to compile correctly.

It is probably easiest to assume that the literals and operations, the control flow structures, scoping rules etc. operate like in the target language. Once more backends are added (groups of 2–4), the backend semantics may differ and a better approach might be to restrict to functionality that behaves in similar ways in the different backends, or to implement separate transformations for each. Wrappers are possibly needed for I/O.

The process of code generation is rather straightforward with these assumptions. The abstract syntax tree needs to be traversed in order (e.g. using a recursive search routine or the Visitor pattern), generating code for each node in the tree.

2.4 Demo application

The demo application implements the rock / paper / scissors game:

1) Ask for the name of player 1 and store it
2) Ask for the name of player 2 and store it
3) Print that the game started with two players 1 and 2 (names)
4) Ask player 1 for the gesture (rock, paper, scissors) and store the input
5) Ask player 2 for the gesture (rock, paper, scissors) and store the input
6) Calculate the result of the game (player 1/2 wins or draw)
7) Print “X wins” or “draw”, where X is the name of the winning player
8) Ask whether the player wants to replay and store the input
9) If the input was ‘yes’, jump to the beginning of the program

2.5 Groups of 2–4

For each additional member, the compiler must generate code for a new backend language and paradigm. The parser must also accept one additional construct per additional member. If the first backend implementation is e.g. a imperative/procedural language, the next three could be: functional, stack based, object-oriented.

2.6 Documentation

The documentation should briefly (n+1 – n+2 pages), where n is the number of people working on the project, describe the main routines in the implementation, the overview of the program operation, and instructions for running the program.

3 Category B: interpretation

A simple Turing complete imperative language MiniImp was introduced on the second lecture. The language's preliminary constructs are explained in the lecture slides and the implementation is available from its git repository.

This exercise assignment defines few further language constructs, which extends the language into **MiniImp#**, a graphical drawing language. The goal is to interpret programs written in MiniImp#, providing feedback via the JavaFX user interface. The quality of the implementation is assessed by executing a MiniImp# demo application, then by evaluating the resulting graphics and console input / output.

In short, the main tasks of this exercise are:

1) extend the **MiniImp** parser into **MiniImp#**
2) implement code interpretation of **MiniImp#**
3) implement a test program(s) **T** in **MiniImp#**
4) assure that the **T** can be executed and behaves as intended
5) document the implementation

3.1 Existing code base

One can implement the exercise from scratch (even using a different language and paradigm), but a set of sample code is offered to guide and speed up the development:

- gitlab.utu.fi/tech/education/apt/parser-experiments

The code in the repository uses a build system that automatically fetches the dependencies. This is a particularly useful feature for this exercise project, since the JavaFX and other dependencies otherwise require some configuration.

The code base offers a graphical (JavaFX) user interface. The existing code demonstrates the use of the graphics canvas by drawing few primitive shapes on it.

The code base also shows how to build a parser & AST visitor (the fi.utu.apt.parsers.antlr classes) and includes another parser written using a PEG parser framework (the fi.utu.apt.parsers.parboiled classes). The PEG parser demonstrates the interpretation of numeric values. Instead of scalar numbers, the interpreter uses a concept of value range (min/max). The src/test folder provides examples of evaluating expressions. The existing evaluation engine can be reused for this work, but requires more code for bindings with the existing ANTLR based MiniImp parser. You can implement the final parser either using ANTLR or Parboiled.

3.2 Required language constructs

The existing implementations cover the parsing of:

- literals: numbers, truth values
- arithmetic operators: +, -, *, /
- boolean operators: not
- equivalence test: is
- control flow constructs: while, if
- I/O: write
- variable (re)definitions and references

The result of this exercise should also accept the following language constructs:

- Graphics primitives:
  - line (x,y,x2,y2,color)
  - filled rectangle (x,y,x2,y2,color)
  - filled circle (x,y,radius,color)

The listed graphics primitives should delegate their work to the respective graphics routines in the provided application platform (demonstrated by the existing code base). The color can be e.g. a special keyword, a string, or a number. The commands must accept any numeric expressions as arguments, e.g. 1+2*3.

3.3 Demo application

There should be a demo application written in MiniImp# that demonstrates all the listed language constructs. It should draw a happy smiley using the graphics primitives.

The groups of 2–4 students should also provide another demo application that demonstrates the additional language constructs in the form of an interactive application that reads input and draws different shapes depending on the input.

3.4 Groups of 2–4

For groups of 2–4, the result of this exercise should also accept the following language constructs:

- boolean operators: and, or
- literals: strings
- type casts: from int to string, from string to int
- I/O: read (results in a string value)
- Graphics primitives:

  - line (x,y,x2,y2)
  - rectangle (x,y,x2,y2)
  - circle (x,y,radius)
  - filled rectangle (x,y,x2,y2)
  - filled circle (x,y,radius)
  - draw text (x,y,string,fontsize)
  - settings of the pen color

The pen color command is now a separate command. It may require some state tracking in the interpreter. You may also want to consider consulting the documentation of the OOMCanvas interface in the OOMKit (which is used to provide the user interface):

- gitlab.utu.fi/tech/education/oom/oomkit

3.5 Documentation

The documentation should briefly (n+1 – n+2 pages), where n is the number of people working on the project, describe the main routines in the implementation, the overview of the program operation, and instructions for running the program.

4 Category C: games

Implement a simple game using one of the non-mainstream languages and paradigms mentioned during the course.

In short, the main tasks of this exercise are:

1) pick an implementation language
2) implement the game using the language
3) test the game and assure that it behaves as intended
4) document how you used the language and paradigm

4.1 Accepted languages

The list of accepted languages is:

- Eager functional languages
  - statically type checked: F#, OCaml, Standard ML, Elm
  - dynamically type checked: Clojure, Erlang, Elixir, Scheme
- Lazy functional languages: Haskell, Frege, Eta
- Pure object oriented languages: Self, Io
- Logic languages: Prolog, Mercury, miniKanren
- Stack-oriented languages: Factor

- Multi-paradigm: Scala (must use "advanced" functional/object oriented features)

If you choose Haskell as the implementation language, there is a sample project that demonstrates the use of user input and random numbers located here:

- gitlab.utu.fi/tech/education/apt/diceroll

Of course, the code might turn out to be useful for other functional languages as well.

You may also suggest some languages not mentioned here. The idea is to avoid imperative/procedural languages that are already familiar from the previous programming courses.

4.2 Topics for a game

The list of accepted games for a single person is:

- Card games
  - Twenty-One (card game)
  - Poker
- Minesweeper
  - minimum: using a grid of 10x10 and 10 bombs.
  - parameters may also be configured
- Battleship
  - does not need to provide visualization of the game state
- Sokoban
  - the game could display the state after each move as a string / list of strings
  - see the appendix A for a sample game start/end state & symbol notations

- Beast
- Flappy bird
  - requires more or less real-time game graphics

You may also suggest other games not mentioned here.

When working in a group of 2–4 persons, the suggested games may turn out to be too simple to implement. Thus, you will need to suggest a game or some other project with a similar level of effort. Do not start the project without a written permission.

4.3 Documentation

The documentation should briefly (2–3 pages) describe how the features of the language and paradigm were used for implementing the game. Focus on topics of the course such as the execution model, state, types, abstractions, the data and the control flow. Try to analyze whether the language supported the development process or actually hindered the work.

Appendix A Sokoban example

```
# = wall   . = floor  @ = player
= = box    $ = goal


START
##########
#..#.##.$#
#.=#....$#
#.....#..#
#####.#.##
#....=..##
#...##.@##
##########

END
##########
#..#.##.=#
#..#...@=#
#.....#..#
#####.#.##
#.......##
#...##..##
##########
```

Version 1.0-16-02-2020