Ministerstwo Nauki i Szkolnictwa Wyższego

Organizacja IX Konferencji Modelowanie Matematyczne w Fizyce i Technice (MMFT 2017)
- zadanie finansowane w ramach umowy 829/P-DUN/2017 ze środków Ministra Nauki i Szkolnictwa Wyższego przeznaczonych na działalność upowszechniającą naukę.

Learning tools in course on Semantics of Programming Languages

William Steingartner, Valerie Novitzká

Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia

September 18-21, 2017

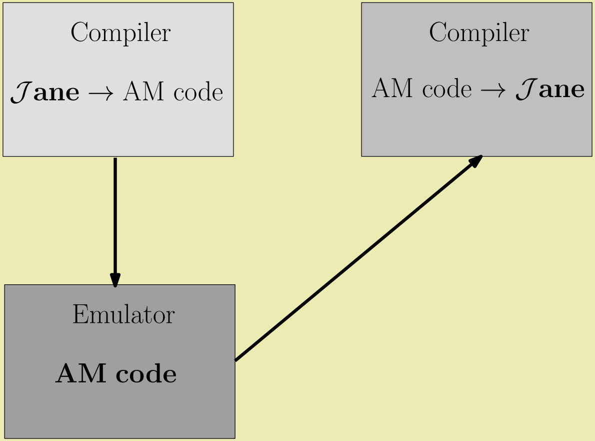
Introduction

Motivation

- on the present the computer science increases making use of formal models to help the understanding of complex software systems and to reason about their behaviour;
- all of the tools and techniques based on formal methods in software development are well grounded in formal models of system execution which are rooted in the formal semantics of the underlying programming languages;
- semantics of programming languages is important for software engineers and IT experts to understanding the meaning of programs and/or behaviour of them;
- we present a package of learning tools which plays an important *rôle* in our course on Semantics of programming languages;
- this package is dedicated for teachers and also for students.

Our software package

How it works


Compiler
$\mathcal{J}$**ane** → AM code
Compiler
AM code → $\mathcal{J}$**ane**
Emulator
AM code

Language $\mathcal{J}ane$

- consists of traditional syntactic constructions of imperative languages;
- for defining formal syntax of $\mathcal{J}ane$ the following syntactic domains are introduced:

$n \in \mathbf{Num}$	- for digit strings;
$x \in \mathbf{Var}$	- for variable names;
$e \in \mathbf{Expr}$	- for arithmetic expressions;
$b \in \mathbf{Bexpr}$	- for Boolean expressions;
$S \in \mathbf{Statm}$	- for statements.

Language $\mathcal{J}ane$ - Syntax

The elements $n \in \mathbf{Num}$ and $x \in \mathbf{Var}$ have no internal structure from semantic point of view.

The syntactic domain $\mathbf{Expr}$ consists of all well-formed arithmetic expressions created by the following production rule

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e \mid (e).$$

Boolean expression from $\mathbf{Bexpr}$ can be of the following structure:

$$b ::= \texttt{false} \mid \texttt{true} \mid e = e \mid e \leq e \mid \neg b \mid b \wedge b \mid (b).$$

As the statements $S \in \mathbf{Statm}$ we consider five elementary Dijkstra's statements:

$$S ::= x := e \mid \texttt{skip} \mid S; S \mid \texttt{if}\ b\ \texttt{then}\ S\ \texttt{else}\ S \mid \texttt{while}\ b\ \texttt{do}\ S.$$

Structural operational semantics of $\mathcal{J}ane$

- the meaning of program is expressed as a change of state s, where

$$s \in \mathbf{State} = \mathbf{Var} \to \mathbb{Z};$$

- state plays passive role when a value of arithmetic or Boolean expression is evaluated:

$$\mathscr{E} : \mathbf{Expr} \to (\mathbf{State} \to \mathbb{Z}) \qquad \mathscr{B} : \mathbf{Bexpr} \to (\mathbf{State} \to \mathbb{B})\,;$$

- the semantics of statements is defined by rules which describe how a state is changed by an execution of a given statement;
- the change of state in structural operational semantics is expressed as transition relation.

Transition relations

Transition relation describes individual step of the statement execution:

$$\langle S, s\rangle \Rightarrow \alpha$$

Possible outcomes are:

- if the computation of S from s has terminated and the final state is s':

$$\langle S, s\rangle \Rightarrow s';$$

- if the computation of S from a state s is not completed and the execution continues with the substatement S' in a state s':

$$\langle S, s\rangle \Rightarrow \left\langle S', s'\right\rangle;$$

- if no rule is applicable, then the execution is stopped and this configuration is a stuck.

Operational semantics as transition system

- a model of structural operational semantics is a transition system which models a program behavior on a state space;
- the change of state is defined for particular statements by inference rules;
- a transition $\langle S, s\rangle \Rightarrow s'$ is a relation between input state s and output state s';

$$\langle x := e, s\rangle \Rightarrow s[x \mapsto \mathscr{E}[\![e]\!]s] \qquad (1_{os})$$

$$\langle \texttt{skip}, s\rangle \Rightarrow s \qquad (2_{os})$$

$$\frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}\ (3^1_{os}) \qquad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}\ (3^2_{os})$$

$$\frac{\mathscr{B}[\![b]\!]s = \mathbf{tt}}{\langle \texttt{if}\ \ b\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2, s\rangle \Rightarrow \langle S_1, s\rangle}\ (4^{\mathbf{tt}}_{os})$$

$$\frac{\mathscr{B}[\![b]\!]s = \mathbf{ff}}{\langle \texttt{if}\ \ b\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2, s\rangle \Rightarrow \langle S_2, s\rangle}\ (4^{\mathbf{ff}}_{os})$$

$$\langle \texttt{while}\ \ b\ \ \texttt{do}\ \ S, s\rangle \Rightarrow \langle \texttt{if}\ \ b\ \ \texttt{then}\ \ (S; \texttt{while}\ \ b\ \ \texttt{do}\ \ S)\ \ \texttt{else}\ \ \texttt{skip}, s\rangle \qquad (5_{os})$$

Example 1

Consider the statement

$$\texttt{if}\ (x <= y)\ \texttt{then}\ max := y\ \texttt{else}\ max := x$$

and the initial state $s = [x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$. The derivation sequence is as follows:

$$\langle \texttt{if}\ (x <= y)\ \texttt{then}\ max := y\ \texttt{else}\ max := x, [x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]\rangle \Rightarrow$$

$$\langle max := x, [x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]\rangle \Rightarrow$$

$$[max \mapsto \mathbf{9}, x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]\,.$$

It follows from the resulting state, that the maximum of the given input values is $\mathbf{9}$.

Abstract machine

- is a part of structural operational semantics and it serves for abstarct implementation of programs;
- consists of
  - instructions;
  - a sequence of instructions forms a code of AM;
  - a set of translation functions translate program in $\mathcal{J}ane$ to an AM code, e.g.:

$$\mathscr{T}\mathscr{E}[\![\texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2]\!] = \mathscr{T}\mathscr{B}[\![b]\!] : \texttt{BRANCH}\,(\mathscr{T}\mathscr{S}[\![S_1]\!], \mathscr{T}\mathscr{S}[\![S_2]\!])\,;$$

  - the semantics of instructions;
- an execution of a code is expressed as a sequence of configurations of the form

$$\langle c, st, s\rangle\,.$$

Compiler from $\mathcal{J}ane$ to AM code

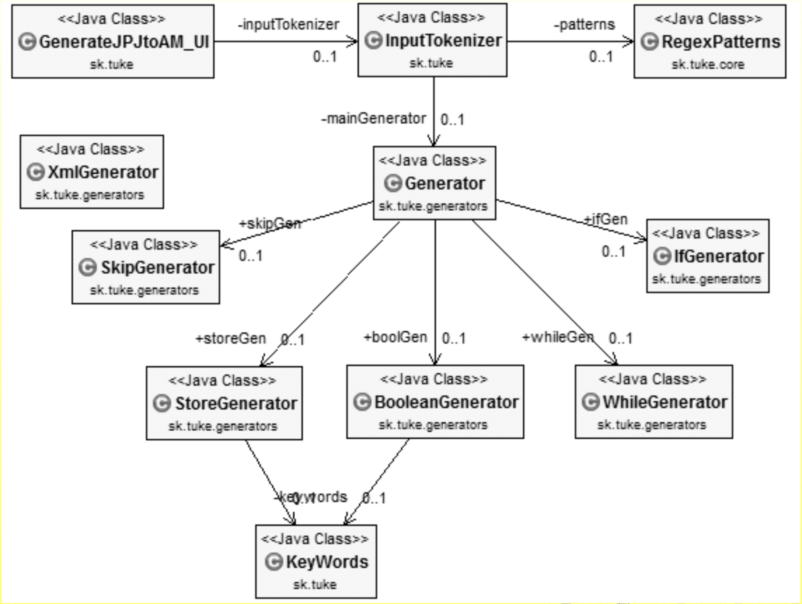
Module specification

- program is designed as an application providing the translation of a program written in $\mathcal{J}ane$ language into code of AM;
- input is a source program in $\mathcal{J}ane$, output is a final code: either sequence of AM instructions or XML form;
- program contains obvious compiler phases: lexical analysis (*tokenization*), syntax analysis (*top-down parsing with error recovery*), semantic analysis (*type mismatch control*);

Classes

- the main class *GenerateJPJtoAM_UI* provides communication and interaction with user;
- class *InputTokenizer* represents a lexical analysis;
- class *RegexPatterns* is used in syntax analysis and it provides the regular expressions for matching the keywords of the language;
- class *Generator* is used as generator of instructions;
- classes *StoreGenerator*, *SkipGenerator*, *IfGenerator*, *WhileGenerator*, *BooleanGenerator*, *XmlGenerator* - for particular statements, expressions and XML output.

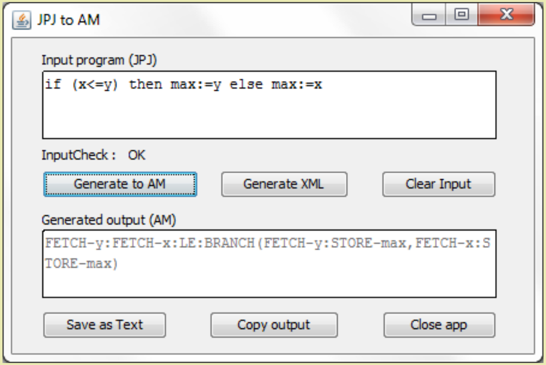
Inside the compiler



Example 2: Example of using the compiler

Consider the program from Example 1. Our module translates it to the AM code:

$$\mathtt{FETCH} - y : \mathtt{FETCH} - x : \mathtt{LE} : \mathtt{BRANCH}\,(\mathtt{FETCH} - y : \mathtt{STORE} - max, \mathtt{FETCH} - x : \mathtt{STORE} - max)$$



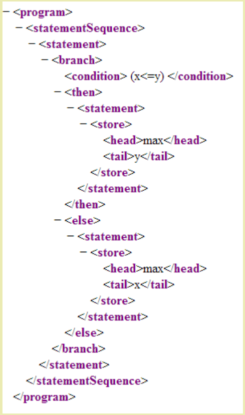
Input program (JPJ)

```
if (x<=y) then max:=y else max:=x
```

InputCheck : OK

Generated output (AM)

```
FETCH-y:FETCH-x:LE:BRANCH(FETCH-y:STORE-max,FETCH-x:S
TORE-max)
```



```
- <program>
  - <statementSequence>
    - <statement>
      - <branch>
          <condition> (x<=y) </condition>
        - <then>
          - <statement>
            - <store>
                <head>max</head>
                <tail>y</tail>
              </store>
            </statement>
          </then>
        - <else>
          - <statement>
            - <store>
                <head>max</head>
                <tail>x</tail>
              </store>
            </statement>
          </else>
        </branch>
      </statement>
    </statementSequence>
  </program>
```

Reverse compiler from AM code to $\mathcal{J}ane$

Module specification

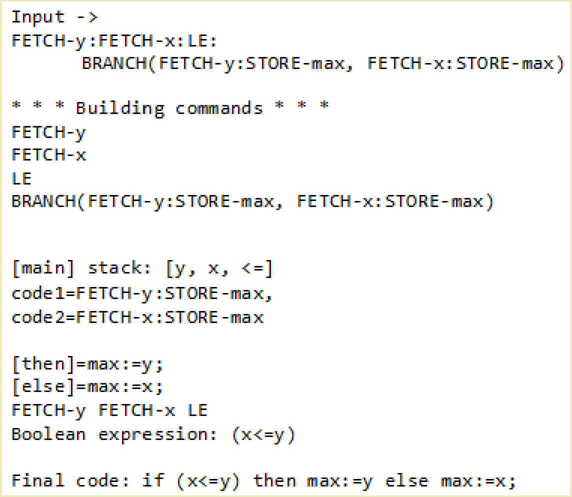
- program is designed as an application providing reverse translation of an AM code into program written in $\mathcal{J}ane$ language;
- program reads an input and starts with splitting the input sequence into particular instructions storing them in a list;
- the next step is recognizing the instructions in the list by matching with paterns and building up the program code;
- a stack is used for building up the arithmetic and Boolean instructions.

The main rules for instructions

- `STORE` provides assignment with an expression on the right-hand side constructed from stack;
- `EMPTYOP` provides an empty statement;
- `BRANCH` runs two other recursive code buildings: both for the code, then the `if` – `then` statement is constructed with the expression constructed from the stack;
- `LOOP` runs two other recursive code buildings: for Boolean expression and for the code, then the `while` – `do` statement is constructed.

Example 3: Example of using the reverse compiler

Consider the instruction sequence from Example 2. Our module translates it to the program code:



```
Input ->
FETCH-y:FETCH-x:LE:
       BRANCH(FETCH-y:STORE-max, FETCH-x:STORE-max)

* * * Building commands * * *
FETCH-y
FETCH-x
LE
BRANCH(FETCH-y:STORE-max, FETCH-x:STORE-max)


[main] stack: [y, x, <=]
code1=FETCH-y:STORE-max,
code2=FETCH-x:STORE-max

[then]=max:=y;
[else]=max:=x;
FETCH-y FETCH-x LE
Boolean expression: (x<=y)

Final code: if (x<=y) then max:=y else max:=x;
```

Emulator of AM code

Module specification

- program is designed as an application providing full processing of code respecting the semantics of AM instructions;
- an input code is analyzed by matching the patterns of instructions, and the sequence is split into particular instructions;
- during the analysis all variables in an input source are found and put into the table of variables;
- during the stepwise execution the new state on a stack and in memory state are computed;
- the program displays an actual AM configuration before and after the execution of an actual instruction;
- although the input program is syntactically correct, it can contain logical error and during the program execution an infinite cycle can occur. Emulator identifies the number of loops and if this number is greater or equal to the upper limit of using the virtual machine stack, then the cycle is marked as infinite.

Example 4: Example of using the code emulator

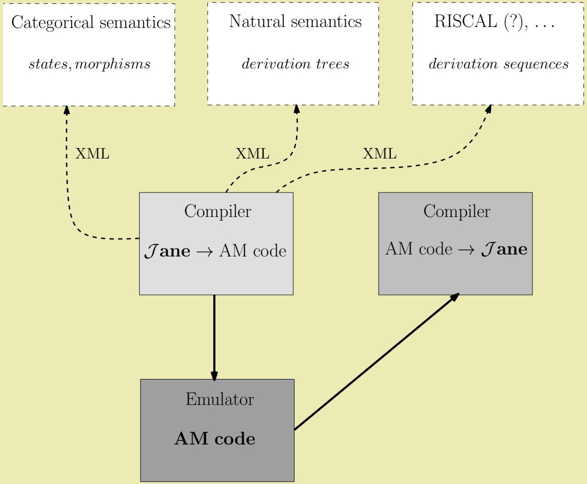
Consider the code in Example 2. This module provides stepwise execution of AM code as it is illustrated in the following table.

Instruction	Stack	State
$\texttt{FETCH} - y$	ε	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
$\texttt{FETCH} - x$	$\mathbf{7}$	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
$\texttt{LE}$	$\mathbf{9} : \mathbf{7}$	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
$\texttt{BRANCH}(\ldots)$	$\mathbf{ff}$	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
$\texttt{FETCH} - x$	ε	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
$\texttt{STORE} - max$	$\mathbf{9}$	$[x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$
ε	ε	$[max \mapsto \mathbf{9}, x \mapsto \mathbf{9}, y \mapsto \mathbf{7}]$

Every row in this table describes one execution step. The first column contains an instruction to be executed, the second column contains stack and the third contains the state before execution. The last row provides the result state with the empty stack.

The future

How it could work



Thank you for your attention