PROJECT PROCESSING OF IFJ AND IAL SUBJECTS

Zbyněk Křivka, Dominika Regéciová, Lukáš Zobal, Radim Kocman
email: {curve, iregeciova, izobal}@fit.vutbr.cz
September 24, 2018

1 General information

Project name: Implementation of IFJ18 imperative language translator.
Information: a discussion forum and a wiki page of the IFJ in IS FIT.
Trying to submit: Friday November 23, 2018, 23:59 (optional).
Date of Submission: Wednesday 5 December 2018, 23:59.
Delivery method: via IS FIT to the IFJ subject data warehouse.

Rating:

- Each student receives a maximum of 25 points in the IFJ (15 overall project functionality, 5 documentation, 5 defense).
- Each IAL receives a maximum of 15 points (5 overall project functionality, 5 defense, 5 documentation).
- Max. 30% of your individual assessment of basic functionality in the subject In addition, IFJ is a creative approach (various extensions, etc.).
- **Granting credit from IFJ and IAL is conditional upon obtaining min. 20 points in progress semester. In addition, in the IFJ of these 20 points, you must earn at least 4 points per program. Moving part of the project.**
- Documentation will be evaluated by a maximum of half the points of the pro- will also reflect the percentage distribution of points and will be rounded to the full points.
- Points written behind the program section including extensions will also be rounded off and in case of exceeding 15 points, the term "Project - Premium Points" in IFJ is entered.

Research teams:

- The project will be solved by three to four teams. Teams with a different number of members are desert.
- Team registration is done by logging in to the appropriate IS FIT entry. Registration is two-phase. In the first phase, individual project variants are subscribed **only** team leaders (capacity is limited to 1). In the second phase, other members (the capacity will be increased to 4). Team leaders will be full

authority over the composition of his team. Also, the mutual communication between the taught-
teams and teams will be best done through leaders (ideally in a copy of others
team members). As a result, each member will be the first registered member to
the leader of this team. All project dates can be found in IS FIT and others
information on the subject pages [1] .

- The input contains two variants that differ only in the way the table is implemented
symbols and are identified by the Roman numeral I or II. Each team has its own
the number to which the selected entry option relates. The choice of variants is done
joining the team in the IS FIT.

2 Entering

Create a C program that loads the source code written in the source language IFJ18
and translates it into the target language IFJcode18 (InterCode). If a translation is made without errors,
the return value 0 (zero) is returned. If an error has occurred, return time is returned.
note as follows:

- 1 - bug in the lexical analysis program (bad structure of the current lex).
- 2 - program error in syntax analysis (program syntactic error).
- 3-semantic error in the program - undefined function / variable, attempt to redefine
neither function / variable, etc.
- 4-semantic / runtime error of type compatibility in arithmetic, string and
relational expressions.
- 5-semantic error in the program - bad number of parameters in the function call.
- 6 - other semantic errors.
- 9 - runtime error by zero.
- 99 - internal error of the compiler, ie not affected by the input program (eg alo-
memory, etc.).

The translator will retrieve the IFJ18 control program from the standard input and gene-
output the resulting IFJcode18 intermediate code (see chapter 10) to standard output. Everything
error messages, warnings, and debug statements to standard error output; ie it will be
it is a console application (a so-called filter) without a graphical user interface. For in-
interpretation of the resulting program in the target language IFJcode18 will be on the subject pages
available interpreter.

Keywords are bold and some lexemas are used to increase legibility in apo-
strofech, and the apostroph symbol is not part of the language!

1 http://www.fit.vutbr.cz/study/courses/IFJ/public/project

3 Description of the programming language

IFJ18 is a simplified subset of the Ruby 2.0 language [2] , which is dynamically typed [3]
imperative (object-oriented) language with functional elements.

3.1 General features and data types

The IFJ18 programming language **depends** on the letter size of both identifiers and keywords (so-called *case-sensitive*).

- The *identifier* is defined as a non-empty sequence of digits, letters (small and large,) and an underscore character (' _ ') starting with **a** lowercase or underscore. In addition the function identifier can end with a question mark (' **?** ') or an exclamation mark (' **!** ').
- IFJ18 includes, in addition, the following *keywords* , which have a specific meaning, and therefore must not be identified as identifiers [4] :

def, do, else, end, if, not, nil, then, while.

- *Integer literal* (C-int range) consists of a non-empty sequence of digits and expresses the value of the entire non-negative number in the decimal system [5] .
- The *decimal literal* (C-double range) also expresses non-negative numbers in decimal the literal consists of the whole and the decimal part, or the whole part and the exponential the whole or the decimal part and the exponent. The whole and the decimal part is formed a non-empty sequence of digits. Exponent is integer, starting with ' **e** ' or ' **E** ' followed by the optional ' + ' (plus) or ' - ' (minus) and the last part a non-empty sequence of digits. There must be no other character between the parts, all and the decimal part separates the character ' **.** '(dot) [6] .
- The *string literal* is double-sided with double quotes (**"** , ASCII value 34). It consists of any number of characters written on a single line of the program. Possible there is also an empty string (**""**). Characters with an ASCII value greater than 31 (except **"** a \") write directly. Some other characters can be written using the escape sequence: ' \' ',' \ **n** ', ' \ **t** ', ' \ ', ' \\ '. Their meaning coincides with the corresponding character constants language Ruby [7] . The character in a string can also be specified using a general hexadecimal escape sequence ' \ **x** *hh* ', where *hh* is a one-digit or two-digit hexadecimal number from **0** to **FF** .
  Chain length is not limited (or only available memory). For example, string literal

"Hello \ nSve'te \ s \\\ x22"

2 https://ruby-doc.org/core-2.0.0/; server Merlin is available for students interpreter ruby version 2.0.0 and interactive irb console.
3 Individual variables have a data type determined by the value / object they contain.
4 Non-collision must also be ensured against the built-in function identifiers. The extensions can then be other keywords are used, but we will test them only if we implement the appropriate extensions.
5 Excessive digits 0 are not allowed because they represent the number entry in the octal system.
6 Excessive digits 0 in integer are error, exponent ignored.
7 http://ruby-doc.com/docs/ProgrammingRuby/html/tut_stdtypes.html

  represents a string
  Hi
  Svete \ **"** Do not consider strings that contain multibyte Unicode characters (e.g., UTF-8).
- A special case is the **nil** value (the so-called unknown value), which in terms of IFJ18 does not have Type [8] .
- The *data types* for each literal are designated **Integer** , **Float,** and **String** . Types are used only internally and are important for semantic controls.
- *Term* is any literal (integer, decimal or string), **nil** or variable identifier.
- IFJ18 supports *line* and *block comments* as well as Ruby. Line Co- the mentor starts with a grid character ('#', ASCII value 35), and is considered as a comment everything that follows to the end of the line. Block comment begins with string ' = **begin** ' located at the beginning of the line and ending with the string ' = **end** ' the beginning of the new line. Hierarchical nesting of block comments is not supported. The commented characters are also on rows starting with ' = **begin** ' or ' = **end** ', but these

must be separated from ' = **begin** ' / ' = **end** ' by at least one space or tab.

4 Language structure

IFJ18 is a structured programming language supporting variable definitions and user-
basic functions, assignment command, and function call, including recursive.
The control program input point is an unlabeled disjoint sequence of commands between defi-
any user functions, the so-called *main body of the* program.

4.1 Basic language structure

The program consists of a sequence of definitions of user functions and commands. Commands off
function definition is the main body of the program. In the body of the function definition in the main body of the program
there may be any (even zero) number of IFJ18 commands.

The individual constructions of the IFJ18 are (but exceptions) one-line and, as such,
are always terminated by the end of line (**EOL**). For multi-line definitions
constructions, the additional **EOL** characters are explicitly mentioned [9] . White characters without comments (i.e.
gaps or tabs) may occur in any number between any lexemes
(unless stated otherwise), at the beginning and end of the source text. Comments can be added
women between commands, one-line commands (even within command sequences) and multi-line commands
commands or function definitions where line breaks (**EOLs**) are set.

4.2 The main body of the program

The main body of the program is the unlabeled sequence of IFJ18 commands that blend in with the definitions
function (at the highest level of commands, the definition of the function can not, therefore, interfere with the integrity of the command, and

8 In Ruby, the value of the NilClass instance is nil.

9 The outer structures mentioned are the **EOL** sign as a *white sign* . It is, for example, possible for clarity
code to insert a blank line between two commands.

not even compound). The main body of the program can also be an empty sequence of commands when
only the return value of the program is returned (for possible values see chapter 2) . The whole
the end of the source file. The structure of the individual commands is described
in the following chapters.

4.2.1 Definition of variables

IFJ18 language variables are only local (even if defined in the main body of the program).
Local variables and function parameters have a range in the function where they were defined (from
the place of their definition after the end of the function). IFJ18 does not contain a specific de-
variable finiters, but the definition of the variable takes place within the first assignment of the value to
variables (see description of commands below), even if this assignment is not executed (ie,
part of the initialization can be recognized by the definition of the variable when generating the code). By definition
the variable is initialized by the default **nil** value . If it is a definition
the Runtime Assignment command is actually executed, the variable initializes to the value of the at-
of the ordered expression. The variable can not be used before its definition (error 3).

Can not define a variable of the same name as any of the functions already defined, and
on the other hand, the function of the same name can not be defined as some of the variables already defined in the chapter
the body of the program. Each variable used in the program must be defined, otherwise
This is a semantic error 3.

4.3 Definition of user functions

Each function must be defined before it is called a *function call* (call command
is defined below). When calling a function from the main body of the program, the function must be first
defined. However, the definition of a function may not be lexically placed before your call if
is called from another function that is called after defining both functions. They are therefore allowed
reciprocal recursive calls to two or more functions without declaring functions. The function definition is

of the following shape:

- *The function definition* is a multi-row construction (head and body) in the form:

  def *id* **(list of** *parameters* **) EOL**
  command sequence
  end

  - The definitions of each formal parameter are separated by a comma (' , '), the last of them does not show a comma. The list may also be empty. Parameters are always transmitted by value.
  - The function body is a sequence of subcommands (see Section 4 .4) . The body's functions are hers parameters are understood as predefined local variables. The result of the function is given the value of the last evaluated body function statement.

In terms of semantics, it is necessary to check whether the number of parameters in the header matches function definition and call function. Redefining functions and overloading features is not allowed.

4.4 Command syntax and semantics

All commands in IFJ18 also give the final value (but they are not unlike Ruby, mrs syntaktically as expressions). The result of the command may be **nil** (eg. In the evaluation function without any internal commands or only containing a cycle command). Sub-command means:

- *Assignment command* :
  id = expression
  The command semantics is as follows: The command executes an expression expression *expression* (see kapitola 5) and eventually assigning its value to left *id* . *id* . The left operand must be a variable (so-called l-value) and after assigning it will be *id of the* same type as the type of value expression *expression* . The result of the command is the value of the expression *expression* . Part ' *id* = ' can be omitted, so the evaluated *term* is not assigned anywhere but can serve as a return value. feature note.
- *Conditional* :
  if *expression* **then EOL**
  command sequence 1
  else EOL
  sekvence_příkazů 2
  end
  The command semantics is as follows: First, the given *expression* is evaluated (typically used a relational operator). If the evaluated expression is true, it is done *sekvence_příkazů* 1, otherwise executes *sekvence_příkazů* second If the resulting value ration is not truthful (ie true or false - in the basic assignment only as the relation of the relational operator application in Section 5.1) is considered **nil** untrue and other truths. *Command Sequence* 1 and *Command Sequence* 2 (mo- also be empty) are again the sequence of subcommands defined in this section (recursive definitions). The result of the command is the result of the last executed command from the executed sequence of commands. If the sequence is empty, the result is **nil** .
- *Cycle command* :
  while *expression* **to EOL**
  command sequence
  end
  The cycle command semantics is as follows: Repeats the execution of the sequence of commands *sekvence_příkazů* (see instructions in this section) so long as the value of the expression pravdivá. The rules for determining the veracity of the expression are the same as for conditional expression command. The result of this command is always **nil** .
- *Call a built-in or a user-defined function* :

id = *function_name* (*Input_parameter list*)

Input_parameter list is a list of terms (see section 3.1) of separated comma-
me [10] . The list may also be empty. Gaps around the input parameters list
may be omitted. The semantics of built-in functions will be described in Chapter 6 .

10 The function call parameter is not an expression. This is part of the optional point-to-point extension of the project FUNEXP.

The semantics of the function call is as follows: The command assures the passing of parameters by the value a transmission of control to the body of the function. If the function call command contains a different number parameters that the function expects (that is, what is stated in its header, function functions), this is an error of 5. After returning from the body of the function, it is possible saving the result of the function (ie the value of the last evaluated function body command) to *id* and continue running the program immediately after the call command is done, functions. The result of the command is the resulting value of the called function. Analogous with an assignment command, you can omit part ' *id* = '.

5 Expressions

The terms are formed by terms, brackets and binary arithmetic, string and relational operators.

If necessary, implicit conversions of operands, function parameters, of expressions or features. Possible implicit conversions of data types are: (a) **Integer** on **Float** , (b) **Float** to **Integer** (cropped).

For erroneous combinations of data types (after possible default conversions) that you are able to validate when you translate [11] , return error 4. Other Type Controls generate the interdiction and run the interpreter for a type of incompatibility during a runtime error with return code 4.

5.1 Arithmetic, chain and relational operators

Standard binary operators + , - , * denote addition, subtraction [12] and multiplication. If both operandy type **Integer** , is also the **integer** type. If there is one or [13] or both type operands **Float** , the result is **Float** . In addition, the + operator performs two types of operands **String** their concatenation.

Operator / denotes division of two numeric operands. If at least one operand **Float,** is the result of the **Float** division , otherwise the operator is executed as an integer division and result is **Integer** . When dividing by zero, runs are running [14] error 9.

For relational operators < , > , <= , > = , == ,! = , The result of the comparison is true and that they have the same semantics as in the Ruby language. These operators work with operandy of the same type, namely **Integer** , **Float** or **String** . If one operand is **Integer** a the second **Float** is an **Integer** operand being converted to **Float** . For strings to compare performs lexicographically. Operators == a **!** = Allow to compare operands of different types (including **nil**), if the implicit conversion between **Integer** and **Float** is not the result untruth. Without the BOOLOP extension, it is not possible to work with the result of the comparison and can use it only for **if** and **while** .

11 Static error type detection in constant expressions can be variously sophisticated, so evaluating tests will recognize error 4 in these situations as a run-time (return from an interpreter).

12 Numerical literals are non-negative, but the result of the expression assigned to the variable is no longer negative.

13 Of course, the implicit conversion of the second operand will also occur on **Float** .

14 When you recognize the zero division as a constant expression, it is possible to report error 9 already during translation.

5.2 Priority of operators

Operator priority can be explicitly modified by subdirectory bracketing. The following table shows Operator priorities (top top):

Priority	Operators	Associativity
1	* /	left
2	+ -	left
3	< <= > >=	none
4	== !=	none

6 Built-in functions

The translator will provide some basic built-in features that can be used in IFJ18 programs. You can take advantage of the embedded code generation code specialized instructions of IFJcode18.

Built-in functions for loading literals and listing terms :

- **inputs ()**
  inputi ()
  inputf ()
  Built-in functions from the standard input will load one row of finite-him. **inputs** returns this string without further editing, including the end of row [15] symbol (the loaded string does not support escape sequences). In case of **input** and **inputf** are the initial white characters are ignored. The characters behind the first are also ignored inappropriate character (including). **inputi** reads and returns an integer, **inputf** decimal number. In the case of a missing input value or its bad format, the function **nil** is zero, 0, or 0.0.
- *Dump command* :
  print (*term* 1 **,** *term* 2 **, ... ,** *term* **)**
  The built-in command has any non-zero number of parameters created by the terms separated by comma. The command semantics is as follows: Gradually from left to right, it passes (described in more detail in section 3.1) and lists their values according to the type free output, without any delimiters and in proper format. Value of an expression type **Integer** will be printed with '% d' [16] , the **float** expression value then power '% and' [17] . **nil** is listed as an empty string. **Print** always returns returns **nil** value .

Embedded Chain Functions : The following built-in features also work any implicit parameter conversion and error 4 if the parameter type is wrong.

15 Looping an empty string actually requires that you load the subsequent line end, which will also be part of the return value of the **inputs** function .
16 Formatting String of the standard C **printf** function (standard C99 and later).
17 C language **printf** string for precise hexadecimal representation of the decimal number.

- **length (** *s* **)** - Returns the length (number of characters) of a string specified by a single parameter . E.g. **length ("x \ nz")** returns 3.

- **substr (** *s* **,** *i* **,** *n* **)** - Returns the substring of the specified string . The second parameter is the start of the desired substring (counted from zero) and the third parameter determines substring length. If the index is out of bounds 0 to **length (** *s* **)** or <0, the function returns **nil** . If > **length (** *s* **)** - , as a string is returned from the -th character all the remaining characters of string .
- **ord** ***(s, i)*** - Returns the ordinal value (ASCII) character at the position in the chain. If it is position beyond the string (0 to **length (** *s* **)** - 1), returns **nil** .
- **chr (** *i* **)** - Returns a single-character string with a character whose ASCII code is entered by para- by subway . A case where is out of range [0; 255] leads to a runtime error with chain.

7 Implementing the symbol table

The symbol table will be implemented using an abstract data structure that is in your the assignment for that team marked with Roman numerals I-II, as follows:

I) Implement the symbol table using the binary search tree.

II) Implement the table of symbols using the scattered table.

The implementation of the symbol table will be stored in the symtable.c file (or symtable.h). See section 12 .2 for details .

8 Examples

This chapter presents three simple examples of IFJ18 controllers.

8.1 Factorial calculation (iterative)

```
# Program 1: Factorial Calculation (iterative)
print "Enter the number of the number to be calculated: ␣"
a = inputi
if a <0 then
    print ("\ nFaktorial␣nelze␣spocitat \ n")
else
    send = 1
    while a> 0
        send = send * a
        a = a - 1
    end
    print "\ nThe result is:" sent, "\ n"
end
```


8.2 Factorial calculation (recursively)

```
# Program 2: Calculation of the factorial (recursive)
def factorial (n)
    if n <2 then
        result = 1
    else
        decremented_n = n - 1
        temp_result = factorial decremented_n
        result = n * temp_result
    end # if
    result
end # function factor
```

```
# The main body of the program
print "Enter the number of the number to be calculated: ␣"
a = inputi
if a <0 then
   print "\ nFaktorial␣nelze␣spocitat \ n"
else
   send = factorial a
   print ("\ nResults:", sent, "\ n")
end
```

8.3 Working with strings and built-in functions

```
# Program 3: Works with strings and built-in functions
s1 = "This␣je␣nejaky␣text"
s2 = s1 + ", ␣ktery␣jeste␣trochu␣obohatime"
print s1, "\ n", s2, "\ n"
s1len = length (s1)
s1len = s1len - 4 + 1
s1 = substr (s2, s1len, 4)
print "4␣znaky␣od␣" s1len ",␣znaku␣v␣ \", "s2" \ "", s1, "\ n"
print "Enter" Sequence "and" Sequence ","
print "pricemz␣se␣pismena␣nesmeji␣v␣posloupnosti␣opak: ␣"
s1 = inputs
while s1! = "abcdefgh \ n"
      print "Spatne␣zadana␣posloupnost, ␣zkuste␣znovu: ␣"
      s1 = inputs
end
```

9 Testing recommendations

The IFJ18 programming language is deliberately designed to be almost compatible with the Ruby's tongue [18] . If the student is not sure what the destination code should exactly do for some IFJ18 source code, you can verify it as follows. IS FIT will download

18 Online Documentation from 2016 to Ruby 2.0.0: http://ruby-doc.org/core-2.0.0/

from *File* to Subject IFJ from *Project* folder ifj18.rb file containing code that adds IFJ18 compatibility with the ruby Ruby interpreter on the merlin server. Showbor ifj18.rb contains definitions of built-in functions that are part of IFJ18, but missing in Ruby's basic libraries or redefining them.

Your program in IFJ18, stored in the testPrg.ifj file, for example To do this, use the following command on the merlin server:

```
ruby -r ./ifj18.rb testPrg.ifj <test.in> test.out
```

This makes it easy to check what the specified source code should be doing, respectively. vygene- the target destination code. But you need to realize that Ruby's language is a superset of IFJ18, and hence can process structures that are not allowed in IFJ18 (eg richer syntax and the semantics of most commands, or even backward incompatibilities). List these differences will appear on the wiki site and can be discussed at the IFJ subject forum.

10 Target language IFJcode18

The target language IFJcode18 is an intermediate code that includes three-way instructions (typically three arguments) and stackers (typically without parameters and working with values on a data saber). Each instruction consists of an operating code (keyword with the instruction name), u which does not matter in case insensitive. The rest of the instructions are operands,

for which the case-size matters. The operands are separated by arbitrary
non-zero number of spaces or tabs. The trim is used to separate the
instructions so that there is a maximum of one instruction per line, and one is not allowed
instruction to write on multiple rows. Each operand is made up of a variable, a constant, or
labels. IFJcode18 supports one-line comments starting with the grid (#).
The code in IFJcode18 starts with the introductory line with a dot followed by the name of the language:

.IFJcode18

10.1 Evaluation interpreter ic18int

A command interpreter is available for evaluation and testing of the IFJcode18 semicolon
row (ic18int):

ic18int prg.code <prg.in> prg.out

Artist behavior can be modified using the command line switches / parameters. On-
you can get a story to them using the --help switch.
If the interpretation is error free, the return value 0 (zero) is returned. Error
cases, the following return values correspond:

- 50 - incorrectly entered input parameters at the command line when starting an interpreter.

- 51 - Analysis error (lexical, syntactic) of input code in IFJcode18.
- 52 - error in semantic checks of input code in IFJcode18.
- 53 - runtime error - bad types of operands.
- 54 - Runtime Interpretation Error - Access to a non-existent variable (frame exists).
- 55 - runtime error - the frame does not exist (eg reading from an empty stack
  frames).
- 56 - Runtime Interpretation Error - Missing Value (in a variable or on a data)
  stack).
- 57 - Runtime Error of Interpretation - Poor Operand Value (eg Zero, Poor
  return value of the EXIT instruction).
- 58 - Runtime Interpretation Error - Faulty string work.
- 60 - internal interpreter error, ie unaffected by the input program (eg error alo-
  memory, an error when opening a file with a control program, etc.).

10.2 Memory Model

Values are most often stored in the named variables during interpretation
are grouped into so-called frames, which are essentially dictionaries of variables with their values.
IFJcode18 offers three types of frames:

- global, we designate GF (Global Frame), which is automatically at the beginning of the interpretation
  initialized as empty; serves to store global variables;
- local, we denote LF (Local Frame), which is not defined at the beginning and refers to
  top / current frame frame frame; serve to store local pro-
  (The frame tray can be used advantageously when scratched or recursive
  call functions.);
- temporary, we mark TF (Temporary Frame), which is used for preparing a new or
  cleaning the old frame (for example, when calling or completing a function) that can be moved
  nut on the frame tray and become the current local framework. At the beginning of the interpretation
  the temporary frame is undefined.

Overlapping (previously embedded) local frames in the frame stack can not be accessed

before removing later added frames.

Another option for storing unnamed values is the data bin used
stock instructions.

10.3 Data types

Interpreter IFJcode18 works dynamically with operand types, so the type of variable (or palocation) is given by the contained value. Unless otherwise stated, implicit conversions are involved forbidden. The interpreter supports a special value / type nil and four basic data types (int, bool, float and string) whose ranges and precisions are compatible with the IFJ18 language.

The entry of each constant in IFJcode18 consists of two separate parts
(character @, no white characters), constant type designation (int, bool, float, string, nil) and the

constants (number, literal, nil). E.g. float@0x1.2666666666666p +0, bool @ true, nil @ nil or int @ -5.

The int type represents a 32-bit integer (C-int range). The bool type represents the rightdivine value (true or false). The float type describes the decimal number (C-double range) and in the case of writing constants, use the format string '% a' in the C language for the function **printf** . The string string literal is written as a printable sequence in the case of a constant ASCII characters (excluding white characters, grid (#) and backslash (\)) and escape sequences, so it is not enclosed by quotation marks. Escape sequence that is required for ASCII characters kódem 000-032, 035 a 092, je tvaru \ , kde je dekadické číslo v rozmezí 000-255 složené právě ze tří číslic; např. konstanta

string@retezec\032s\032lomitkem\032\092\032a\010novym\035radkem

reprezentuje řetězec

retezec s lomitkem \ a
novym#radkem

Pokus o práci s neexistující proměnnou (čtení nebo zápis) vede na chybu 54. Pokus o čtení hodnoty neinicializované proměnné vede na chybu 56. Pokus o interpretaci instrukce s operandy nevhodných typů dle popisu dané instrukce vede na chybu 53.

10.4 Instrukční sada

U popisu instrukcí sázíme operační kód tučně a operandy zapisujeme pomocí neterminálních symbolů (případně číslovaných) v úhlových závorkách. Neterminál ⟨ *var* ⟩ značí proměnnou, ⟨ *symb* ⟩ konstantu nebo proměnnou, ⟨ *label* ⟩ značí návěští. Identifikátor proměnné se skládá ze dvou částí oddělených zavináčem (znak @; bez bílých znaků), označení rámce LF, TF nebo GF a samotného jména proměnné (sekvence libovolných alfanumerický a speciálních znaků bez bílých znaků začínající písmenem nebo speciálním znakem, kde speciální znaky jsou: _, -, $, &, %, *, !, ?). E.g. GF@_x značí proměnnou _x uloženou v globálním rámci.

Na zápis návěští se vztahují stejná pravidla jako na jméno proměnné (tj. část identifikátoru za zavináčem).

Instrukční sada nabízí instrukce pro práci s proměnnými v rámcích, různé skoky, operace s datovým zásobníkem, aritmetické, logické a relační operace, dále také konverzní, vstupně/výstupní a ladicí instrukce.

10.4.1 Práce s rámci, volání funkcí

MOVE ⟨ *var* ⟩ ⟨ *symb* ⟩ Přiřazení hodnoty do proměnné
Zkopíruje hodnotu ⟨ *symb* ⟩ do ⟨ *var* ⟩. E.g. MOVE LF @ par GF @ var performs copyvání hodnoty proměnné var v globálním rámci do proměnné par v lokálním rámci.
CREATEFRAME Vytvoř nový dočasný rámec
Vytvoří nový dočasný rámec a zahodí případný obsah původního dočasného rámce.

PUSHFRAME Přesun dočasného rámce na zásobník rámců
PřesuňTFnazásobníkrámců.RámecbudekdispozicipřesLFapřekryjepůvodnírámce na zásobníku rámců. TF bude po provedení instrukce nedefinován a je třeba jej před dalším použitím vytvořit pomocí CREATEFRAME. Pokus o přístup k nedefinovanému rámci vede na chybu 55.
POPFRAME Přesun aktuálního rámce do dočasného
Přesuň vrcholový rámec LF ze zásobníku rámců do TF. Pokud žádný rámec v LF není k dispozici, dojde k chybě 55.
DEFVAR ⟨ *var* ⟩ Definuj novou proměnnou v rámci
Definuje proměnnou v určeném rámci dle ⟨ *var* ⟩. Tato proměnná je zatím neinicializovaná a bez určení typu, který bude určen až přiřazení nějaké hodnoty.
CALL ⟨ *label* ⟩ Skok na návěští s podporou návratu
Uloží inkrementovanou aktuální pozici z interního čítače instrukcí do zásobníku volání a provede skok na zadané návěští (případnou přípravu rámce musí zajistit další instrukce).
RETURN Návrat na pozici uloženou instrukcí CALL
Vyjme pozici ze zásobníku volání a skočí na tuto pozici nastavením interního čítače instrukcí (úklid lokálních rámců musí zajistit další instrukce).

10.4.2 Práce s datovým zásobníkem

Operační kód zásobníkových instrukcí je zakončen písmenem „S“. Zásobníkové instrukce načítají chybějící operandy z datového zásobníku a výslednou hodnotu operace ukládají zpět na datový zásobník.

PUSHS ⟨ *symb* ⟩ Vlož hodnotu na vrchol datového zásobníku
Uloží hodnotu ⟨ *symb* ⟩ na datový zásobník.
POPS ⟨ *var* ⟩ Vyjmy hodnotu z vrcholu datového zásobníku
Není-li zásobník prázdný, vyjme z něj hodnotu a uloží ji do proměnné ⟨ *var* ⟩, jinak dojde k chybě 56.
CLEARS Vymazání obsahu celého datového zásobníku
Pomocná instrukce, která smaže celý obsah datového zásobníku, aby neobsahoval zapomenuté hodnoty z předchozích výpočtů.

10.4.3 Aritmetické, relační, booleovské a konverzní instrukce

V této sekci jsou popsány tříadresné i zásobníkové verze instrukcí pro klasické operace pro výpočet výrazu. Zásobníkové verze instrukcí z datového zásobníku vybírají operandy se vstupními hodnotami dle popisu tříadresné instrukce od konce (tj. typicky nejprve ⟨ *symb* 2 ⟩ a poté ⟨ *symb* 1 ⟩).

ADD ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Součet dvou číselných hodnot
Sečte ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨ *var* ⟩.
SUB ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Odečítání dvou číselných hodnot
Odečte ⟨ *symb* 2 ⟩ od ⟨ *symb* 1 ⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨ *var* ⟩.

MUL ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Násobení dvou číselných hodnot
Vynásobí ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ (musí být stejného číselného typu int nebo float) a výslednou hodnotu téhož typu uloží do proměnné ⟨ *var* ⟩.
DIV ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Dělení dvou desetinných hodnot
Podělí hodnotu ze ⟨ *symb* 1 ⟩ druhou hodnotou ze ⟨ *symb* 2 ⟩ (oba musí být typu float) a výsledek přiřadí do proměnné ⟨ *var* ⟩ (též typu float). Dělení nulou způsobí chybu 57.
IDIV ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Dělení dvou celočíselných hodnot
Celočíselně podělí hodnotu ze ⟨ *symb* 1 ⟩ druhou hodnotou ze ⟨ *symb* 2 ⟩ (musí být oba typu int) a výsledek přiřadí do proměnné ⟨ *var* ⟩ typu int. Dělení nulou způsobí chybu 57.
ADDS/SUBS/MULS/DIVS/IDIVS Zásobníkové verze instrukcí ADD, SUB, MUL, DIV a IDIV
LT/GT/EQ ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Relační operátory menší, větší, rovno
Instrukce vyhodnotí relační operátor mezi ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ (stejného typu; int, bool, float nebo string) a do booleovské proměnné ⟨ *var* ⟩ zapíše false při neplatnost nebo true v případě platnosti odpovídající relace. Řetězce jsou porovnávány lexikograficky a false je menší než true. Pro výpočet neostrých nerovností lze použít AND/OR/NOT. S operandem typu nil lze porovnávat pouze instrukcí EQ, jinak chyba 53.
LTS/GTS/EQS Zásobníková verze instrukcí LT/GT/EQ

AND/OR/NOT ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Základní booleovské operátory
Aplikuje konjunkci (logické A)/disjunkci (logické NEBO) na ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ nebo negaci na ⟨ *symb* 1 ⟩ (NOT má pouze 2 operandy) a výsledek zapíše do ⟨ *var* ⟩ (všechny operandy jsou typu bool).
ANDS/ORS/NOTS Zásobníková verze instrukcí AND, OR a NOT

INT2FLOAT ⟨ *var* ⟩ ⟨ *symb* ⟩ Převod celočíselné hodnoty na desetinnou
Převede celočíselnou hodnotu ⟨ *symb* ⟩ na desetinné číslo a uloží je do ⟨ *var* ⟩.
FLOAT2INT ⟨ *var* ⟩ ⟨ *symb* ⟩ Převod desetinné hodnoty na celočíselnou (oseknutí)
Převede desetinnou hodnotu ⟨ *symb* ⟩ na celočíselnou oseknutím desetinné části a uloží ji do ⟨ *var* ⟩.
INT2CHAR ⟨ *var* ⟩ ⟨ *symb* ⟩ Převod celého čísla na znak
Číselná hodnota ⟨ *symb* ⟩ je dle ASCII převedena na znak, který tvoří jednoznakový řetězec přiřazený do ⟨ *var* ⟩. Je-li ⟨ *symb* ⟩ mimo interval [0;255], dojde k chybě 58.
STRI2INT ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Ordinální hodnota znaku
Do ⟨ *var* ⟩ uloží ordinální hodnotu znaku (dle ASCII) v řetězci ⟨ *symb* 1 ⟩ na pozici ⟨ *symb* 2 ⟩ (indexováno od nuly). Indexace mimo daný řetězec vede na chybu 58.
INT2FLOATS/FLOAT2INTS/ INT2CHARS/STRI2INTS Zásobníkové verze konverzních instrukcí

10.4.4 Vstupně-výstupní instrukce

READ ⟨ *var* ⟩ ⟨ *type* ⟩ Načtení hodnoty ze standardního vstupu
Načte jednu hodnotu dle zadaného typu ⟨ *type* ⟩∈ {int, float, string, bool} (včetně případné konverze vstupní hodnoty float při zadaném typu int) a uloží tuto hodnotu do proměnné ⟨ *var* ⟩. Formát hodnot je kompatibilní s chováním příkazů **inputs** , **inputi** a **inputf** jazyka IFJ18.
WRITE ⟨ *symb* ⟩ Výpis hodnoty na standardní výstup

Vypíše hodnotu ⟨ *symb* ⟩ na standardní výstup. Formát výpisu je kompatibilní s vestavěným příkazem **print** jazyka IFJ18 včetně výpisu desetinných čísel pomocí formátovacího řetězce ”%a”.

10.4.5 Práce s řetězci

CONCAT ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Konkatenace dvou řetězců
Do proměnné ⟨ *var* ⟩ uloží řetězec vzniklý konkatenací dvou řetězcových operandů ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ (jiné typy nejsou povoleny).

STRLEN ⟨ *var* ⟩ ⟨ *symb* ⟩ Zjisti délku řetězce
Zjistí délku řetězce v ⟨ *symb* ⟩ a délka je uložena jako celé číslo do ⟨ *var* ⟩.

GETCHAR ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Vrať znak řetězce
Do ⟨ *var* ⟩ uloží řetězec z jednoho znaku v řetězci ⟨ *symb* 1 ⟩ na pozici ⟨ *symb* 2 ⟩ (indexováno celým číslem od nuly). Indexace mimo daný řetězec vede na chybu 58.

SETCHAR ⟨ *var* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Změň znak řetězce
Zmodifikuje znak řetězce uloženého v proměnné ⟨ *var* ⟩ na pozici ⟨ *symb* 1 ⟩ (indexováno celočíselně od nuly) na znak v řetězci ⟨ *symb* 2 ⟩ (první znak, pokud obsahuje ⟨ *symb* 2 ⟩ více znaků). Výsledný řetězec je opět uložen do ⟨ *var* ⟩. Při indexaci mimo řetězec ⟨ *var* ⟩ nebo v případě prázdného řetězec v ⟨ *symb* 2 ⟩ dojde k chybě 58.

10.4.6 Práce s typy

TYPE ⟨ *var* ⟩ ⟨ *symb* ⟩ Zjisti typ daného symbolu
Dynamicky zjistí typ symbolu a do ⟨ *var* ⟩ zapíše řetězec značící tento typ (int, bool, float, string nebo nil). Je-li ⟨ *symb* ⟩ neinicializovaná proměnná, označí její typ prázdným řetězcem.

10.4.7 Instrukce pro řízení toku programu

Neterminál ⟨ *label* ⟩ označuje návěští, které slouží pro označení pozice v kódu IFJcode18. V případě skoku na neexistující návěští dojde k chybě 52.

LABEL ⟨ *label* ⟩ Definice návěští
Speciální instrukce označující pomocí návěští ⟨ *label* ⟩ důležitou pozici v kódu jako potenciální cíl libovolné skokové instrukce. Pokus o redefinici existujícího návěští je chybou 52.

JUMP ⟨ *label* ⟩ Nepodmíněný skok na návěští
Provede nepodmíněný skok na zadané návěští ⟨ *label* ⟩.

JUMPIFEQ ⟨ *label* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Podmíněný skok na návěští při rovnosti
Pokud jsou ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ stejného typu (jinak chyba 53) a zároveň se jejich hodnoty rovnají, tak provede skok na návěští ⟨ *label* ⟩.

JUMPIFNEQ ⟨ *label* ⟩ ⟨ *symb* 1 ⟩ ⟨ *symb* 2 ⟩ Podmíněný skok na návěští při nerovnosti
Jsou-li ⟨ *symb* 1 ⟩ a ⟨ *symb* 2 ⟩ stejného typu (jinak chyba 53), ale různé hodnoty, tak provede skok na návěští ⟨ *label* ⟩.

JUMPIFEQS/JUMPIFNEQS ⟨ *label* ⟩ Zásobníková verze JUMPIFEQ, JUMPIFNEQ
Zásobníkové skokové instrukce mají i jeden operand mimo datový zásobník, a to návěští ⟨ *label* ⟩, na které se případně provede skok.

EXIT ⟨ *symb* ⟩ Ukončení interpretace s návratovým kódem
Ukončí vykonávání programu a ukončí interpret s návratovou chybou ⟨ *symb* ⟩, kde ⟨ *symb* ⟩ je celé číslo v intervalu 0 až 49 (včetně). Nevalidní celočíselná hodnota ⟨ *symb* ⟩ vede na chybu 57.

10.4.8 Ladící instrukce

BREAK Výpis stavu interpretu na stderr
Na standardní chybový výstup (stderr) vypíše stav interpretu v danou chvíli (tj. během vykonávání této instrukce). Stav se mimo jiné skládá z pozice v kódu, výpisu globálního, aktuálního lokálního a dočasného rámce a počtu již vykonaných instrukcí.

DPRINT ⟨ *symb* ⟩ Výpis hodnoty na stderr

Vypíše zadanou hodnotu (*cmd*) na standardní chybový výstup (stderr). Výpisy touto instrukcí bude možné vypnout pomocí volby interpretu (viz nápověda interpretu).

11 Pokyny ke způsobu vypracování a odevzdání

Tyto důležité informace nepodceňujte, neboť projekty bude částečně opravovat automat a nedodržení těchto pokynů povede k tomu, že automat daný projekt nebude schopen přeložit, zpracovat a ohodnotit, což může vést až ke ztrátě všech bodů z projektu!

11.1 Obecné informace

Za celý tým odevzdá projekt jediný student. Všechny odevzdané soubory budou zkomprimovány programem ZIP, TAR+GZIP, nebo TAR+BZIP do jediného archivu, který se bude jmenovat xlogin00.zip, xlogin00.tgz, nebo xlogin00.tbz, kde místo zástupného řetězce xlogin00 použijte školní přihlašovací jméno **vedoucího** týmu. Archiv nesmí obsahovat adresářovou strukturu ani speciální či spustitelné soubory. Názvy všech souborů budou obsahovat pouze malá písmena, číslice, tečku a podtržítko (ne velká písmena ani mezery – krom souboru Makefile!).

Celý projekt je třeba odevzdat v daném termínu (viz výše). Pokud tomu tak nebude, je projekt považován za neodevzdaný. Stejně tak, pokud se bude jednat o plagiátorství jakéhokoliv druhu, je projekt hodnocený nula body, navíc v IFJ ani v IAL nebude udělen zápočet a bude zváženo zahájení disciplinárního řízení.

Vždy platí, že je třeba při řešení problémů aktivně a konstruktivně komunikovat nejen uvnitř týmu, ale občas i se cvičícím. Při komunikaci uvádějte login vedoucího a číslo týmu.

11.2 Dělení bodů

Odevzdaný archív bude povinně obsahovat soubor **rozdeleni** , ve kterém zohledníte dělení bodů mezi jednotlivé členy týmu (i při požadavku na rovnoměrné dělení). Na každém

řádku je uveden login jednoho člena týmu, bez mezery je následován dvojtečkou a po ní je bez mezery uveden požadovaný celočíselný počet procent bodů bez uvedení znaku %. Každý řádek (i poslední) je poté ihned ukončen jedním znakem ⟨LF⟩ (ASCII hodnota 10, tj. unixové ukončení řádku, ne windowsovské!). Obsah souboru bude vypadat například takto ($_{\text{LF}}$ zastupuje unixové odřádkování):

xnovak01:30 $_{\text{LF}}$
xnovak02:40 $_{\text{LF}}$
xnovak03:30 $_{\text{LF}}$
xnovak04:00 $_{\text{LF}}$

Součet všech procent musí být roven 100. V případě chybného celkového součtu všech procent bude použito rovnoměrné rozdělení. Formát odevzdaného souboru musí být správný a obsahovat všechny registrované členy týmu (i ty hodnocené 0 %).

Vedoucí týmu je před odevzdáním projektu povinen celý tým informovat o rozdělení bodů. Každý člen týmu je navíc povinen rozdělení bodů zkontrolovat po odevzdání do IS FIT a případně rozdělení bodů reklamovat u cvičícího ještě před obhajobou projektu.

12 Požadavky na řešení

Kromě požadavků na implementaci a dokumentaci obsahuje tato kapitola i několik rad pro zdárné řešení tohoto projektu a výčet rozšíření za prémiové body.

12.1 Závazné metody pro implementaci překladače

Projekt bude hodnocen pouze jako funkční celek, a nikoli jako soubor separátních, společně nekooperujících modulů. Při tvorbě lexikální analýzy využijete znalosti konečných automatů. Při konstrukci syntaktické analýzy založené na LL-gramatice (vše kromě výrazů) **povinně** využijte buď **metodu rekurzivního sestupu** (doporučeno), nebo prediktivní analýzu řízenou LL-tabulkou. Výrazy zpracujte pouze pomocí **precedenční syntak-**

tické analýzy . Vše bude probíráno na přednáškách v rámci předmětu IFJ. Implementation bude provedena v **jazyce C** , čímž úmyslně omezujeme možnosti použití objektově orientovaného návrhu a implementace. Návrh implementace překladače je zcela v režii řešitelských týmů. Není dovoleno spouštět další procesy a vytvářet nové či modifikovat existující soubory (ani v adresáři /tmp). Nedodržení těchto metod bude penalizováno značnou ztrátou bodů!

12.2 Implementace tabulky symbolů v souboru symtable.c

Implementaci tabulky symbolů (dle varianty zadání) proveďte dle přístupů probíraných v předmětu IAL a umístěte ji do souboru symtable.c. Pokud se rozhodnete o odlišný způsob implementace, vysvětlete v dokumentaci důvody, které vás k tomu vedly, a uveďte zdroje, ze kterých jste čerpali.

12.3 Textová část řešení

Součástí řešení bude dokumentace vypracovaná ve formátu PDF a uložená v jediném souboru **dokumentace.pdf** . Jakýkoliv jiný než předepsaný formát dokumentace bude ig-

norován, což povede ke ztrátě bodů za dokumentaci. Dokumentace bude vypracována v českém, slovenském nebo anglickém jazyce v rozsahu cca. 3-5 stran A4.

V dokumentaci popisujte návrh (části překladače a předávání informací mezi nimi), implementaci (použité datové struktury, tabulku symbolů, generování kódu), vývojový cyklus, způsob práce v týmu, speciální použité techniky a algoritmy a různé odchylky od přednášené látky či tradičních přístupů. Nezapomínejte také citovat literaturu a uvádět reference na čerpané zdroje včetně správné citace převzatých částí (obrázky, magické konstanty, vzorce). Nepopisujte záležitosti obecně známé či přednášené na naší fakultě.

Dokumentace musí povinně obsahovat (povinné tabulky a diagramy se nezapočítávají do doporučeného rozsahu):

- 1. strana: jména, příjmení a přihlašovací jména řešitelů (označení vedoucího) + údaje o rozdělení bodů, identifikaci vaší varianty zadání ve tvaru “Tým *číslo* , varianta ” a výčet identifikátorů implementovaných rozšíření.
- Rozdělení práce mezi členy týmu (uveďte kdo a jak se podílel na jednotlivých částech projektu; povinně zdůvodněte odchylky od rovnoměrného rozdělení bodů).
- Diagram konečného automatu, který specifikuje lexikální analyzátor.
- LL-gramatiku, LL-tabulku a precedenční tabulku, podle kterých jste implementovali váš syntaktický analyzátor.

Dokumentace nesmí :

- obsahovat kopii zadání či text, obrázky [19] nebo diagramy, které nejsou vaše původní (kopie z přednášek, sítě, WWW, …).
- být založena pouze na výčtu a obecném popisu jednotlivých použitých metod (jde o váš vlastní přístup k řešení; a proto dokumentujte postup, kterým jste se při řešení ubírali; překážkách, se kterými jste se při řešení setkali; problémech, které jste řešili a jak jste je řešili; atd.)

V rámci dokumentace bude rovněž vzat v úvahu stav kódu jako jeho čitelnost, srozumitelnost a dostatečné, ale nikoli přehnané komentáře.

12.4 Programová část řešení

Programová část řešení bude vypracována v jazyce C bez použití generátorů lex/flex, yacc/-bison či jiných podobného ražení a musí být přeložitelná překladačem gcc. Při hodnocení budou projekty překládány na školním serveru merlin. Počítejte tedy s touto skutečností (především, pokud budete projekt psát pod jiným OS). Pokud projekt nepůjde přeložit či nebude správně pracovat kvůli použití funkce nebo nějaké nestandardní implementační tech-

niky závislé na OS, ztrácíte právo na reklamaci výsledků. Ve sporných případech bude vždy za platný považován výsledek překladu na serveru merlin bez použití jakýchkoliv doplňujících tečných nastavení (proměnné prostředí, …).

Součástí řešení bude soubor Makefile sloužící pro překlad projektu pomocí příkazu make. Pokud soubor pro sestavení cílového programu nebude obsažen nebo se na jeho

19 Vyjma obyčejného loga fakulty na úvodní straně.

základě nepodaří sestavit cílový program, nebude projekt hodnocený! Jméno cílového programu není rozhodující, bude přejmenován automaticky.

Binární soubor (přeložený překladač) v žádném případě do archívu nepřikládejte!

Úvod **všech** zdrojových textů musí obsahovat zakomentovaný název projektu, přihlašovací jména a jména studentů, kteří se na něm skutečně autorsky podíleli.

Veškerá chybová hlášení vzniklá v průběhu činnosti překladače budou vždy vypisována na standardní chybový výstup. Veškeré texty tištěné řídicím programem budou vypisovány na standardní výstup, pokud není explicitně řečeno jinak. Kromě chybových/ladicích hlášení vypisovaných na standardní chybový výstup nebude generovaný mezikód přikazovat výpis žádných znaků či dokonce celých textů, které nejsou přímo předepsány řídicím programem. Základní testování bude probíhat pomocí automatu, který bude postupně vaším překladačem kompilovat sadu testovacích příkladů, kompilát interpretovat naším interpretem jazyka IFJcode18 a porovnávat produkované výstupy na standardní výstup s výstupy očekávanými. Pro porovnání výstupů bude použit program diff (viz info diff). Proto jediný neočekávaný znak, který bude při hodnotící interpretaci vámi vygenerovaného kódu svévolně vytisknut, povede k nevyhovujícímu hodnocení aktuálního výstupu, a tím snížení bodového hodnocení celého projektu.

12.5 Jak postupovat při řešení projektu

Při řešení je pochopitelně možné využít vlastní výpočetní techniku. Instalace překladače gcc není nezbytně nutná, pokud máte jiný překladač jazyka C již instalován a nehodláte využívat vlastností, které gcc nepodporuje. Před použitím nějaké vyspělé konstrukce je dobré si ověřit, že jí disponuje i překladač gcc na serveru merlin. Po vypracování je též vhodné vše ověřit na serveru Merlin, aby při překladu a hodnocení projektu vše proběhlo bez problémů. V *Souborech* předmětu v IS FIT je k dispozici skript is_it_ok.sh na kontrolu většiny formálních požadavků odevzdávaného archivu, který doporučujeme využít.

Teoretické znalosti, potřebné pro vytvoření projektu, získáte během semestru na přednáškách, wiki stránkách a diskuzním fóru IFJ. Postupuje-li Vaše realizace projektu rychleji než probírání témat na přednášce, doporučujeme využít samostudium (viz zveřejněné záznamy z minulých let a detailnější pokyny na wiki stránkách IFJ). Je nezbytné, aby na řešení projektu spolupracoval celý tým. Návrh překladače, základních rozhraní a rozdělení práce lze vytvořit již v první čtvrtině semestru. Je dobré, když se celý tým domluví na pravidelných schůzkách a komunikačních kanálech, které bude během řešení projektu využívat (instant messaging, konference, verzovací systém, štábní kulturu atd.).

Situaci, kdy je projekt ignorován částí týmu, lze řešit prostřednictvím souboru rozdeleni a extrémní případy řešte přímo se cvičícími. Je ale nutné, abyste si vzájemně (nespoléhejte pouze na vedoucího), nejlépe na pravidelných schůzkách týmu, ověřovali skutečný pokrok v práci na projektu a případně včas přerozdělili práci.

Maximální počet bodů získatelný na jednu osobu za programovou implementaci je **20** včetně bonusových bodů za rozšíření projektu.

Nenechávejte řešení projektu až na poslední týden. Projekt je tvořen z několika částí (např. lexikální analýza, syntaktická analýza, sémantická analýza, tabulka symbolů, generování mezikódu, dokumentace, testování!) a dimenzován tak, aby jednotlivé části bylo možno navrhnout a implementovat již v průběhu semestru na základě

znalostí získaných na přednáškách předmětů IFJ a IAL a samostudiem na wiki stránkách a diskuzním fóru předmětu IFJ.

12.6 Pokusné odevzdání

Pro zvýšení motivace studentů pro včasné vypracování projektu nabízíme koncept nepovinného pokusného odevzdání. Výměnou za pokusné odevzdání do uvedeného termínu (několik týdnů před finálním termínem) dostanete zpětnou vazbu v podobě procentuálního hodnocení aktuální kvality vašeho projektu.

Pokusné odevzdání bude relativně rychle vyhodnoceno automatickými testy a studentům zaslána informace o procentuální správnosti stěžejních částí pokusně odevzdaného projektu z hlediska části automatických testů (tj. nebude se jednat o finální hodnocení; proto nebudou sdělovány ani body). Výsledky nejsou nijak bodovány, a proto nebudou individuálně sdělovány žádné detaily k chybám v zaslaných projektech, jako je tomu u finálního termínu. Využití pokusného termínu není povinné, ale jeho nevyužití může být vzato v úvahu jako přitěžující okolnost v případě různých reklamací.

Formální požadavky na pokusné odevzdání jsou totožné s požadavky na finální termín a odevzdání se bude provádět do speciálního termínu „Projekt - Pokusné odevzdání“ předmětu IFJ. Není nutné zahrnout dokumentaci, která spolu s rozšířeními pokusně vyhodnocena nebude. Pokusně odevzdává nejvýše jeden člen týmu (nejlépe vedoucí), který následně obdrží jeho vyhodnocení a informuje zbytek týmu.

12.7 Registrovaná rozšíření

V případě implementace některých registrovaných rozšíření bude odevzdaný archív obsahovat soubor **rozsireni** , ve kterém uvedete na každém řádku identifikátor jednoho implementovaného rozšíření (řádky jsou opět ukončeny znakem ⟨LF⟩).

V průběhu řešení (do stanoveného termínu) bude postupně (případně i na váš popud) aktualizován ceník rozšíření a identifikátory rozšíření projektu (viz wiki stránky a diskuzní fórum k předmětu IFJ). V něm budou uvedena hodnocená rozšíření projektu, za která lze získat prémiové body. Cvičícím můžete během semestru zasílat návrhy na dosud neuvedená rozšíření, která byste chtěli navíc implementovat. Cvičící rozhodnou o přijetí/nepřijetí rozšíření a hodnocení rozšíření dle jeho náročnosti včetně přiřazení unikátního identifikátoru. Body za implementovaná rozšíření se počítají do bodů za programovou implementaci, takže stále platí získatelné maximum 20 bodů.

12.7.1 Bodové hodnocení některých rozšíření jazyka IFJ18

Popis rozšíření vždy začíná jeho identifikátorem. Většina těchto rozšíření je založena na dalších vlastnostech jazyka Ruby 2.0. Podrobnější informace lze získat ze specifikace jazyka [2] Ruby 2.0. Do dokumentace je potřeba (kromě zkratky na úvodní stranu) také uvést, jak jsou implementovaná rozšíření řešena.

- BOOLOP: Podpora typu **Boolean** , booleovských hodnot **true** a **false** , booleovských výrazů včetně kulatých závorek a základních booleovských operátorů (**not** , **!** , **and** , **&&** , **or** , ||), jejichž priorita a asociativita odpovídá jazyku Ruby. Pravdivostní

hodnoty lze porovnávat jen operátory == a **!=** . Dále podporujte výpisy hodnot typu **Boolean** (+1,0 bodu).

- BASE: Celočíselné konstanty je možné zadávat i ve dvojkové (číslo začíná znaky ' **0b** '), osmičkové (číslo začíná znakem ' **0** ') a v šestnáctkové (číslo začíná znaky ' **\x** ') soustavě (+0,5 bodu).
- CYCLES: Překladač bude podporovat i cykly tvořené trojicemi klíčových slov **until** - **do** - **end** , **begin** - **end** - **until** a **begin** - **end** - **while** . Dále bude podporovat klíčová slova **break** , **next** a **redo** bez případných parametrů (+1,0 bodu).
- DEFINED: Rozšíření přidává novou speciální vestavěnou funkci **defined?** s jedním parametrem, kterým je identifikátor proměnné nebo funkce. Výsledkem volání funkce **defined?** je **nil** , pokud není v době vykonání příkazu definována žádná proměnná, či funkce daného jména. Je-li proměnná daného jména definována, vrací řetězec **"local-variable"** . A nakonec, je-li definována funkce (i vestavěná) daného jména, vrací se řetězec **"method"** (+0,5 bodu).
- FUNEXP: Volání funkce může být součástí výrazu, zároveň mohou být výrazy v parametrech volání funkce. Všimněte si, že v případě vynechání kulatých závorek u volání funkce je třeba přesně sledovat počet uvedených skutečných parametrů. If závorky při volání funkce uvádíme, tak je nezbytné, aby mezi identifikátorem funkce a otevírací závorkou nebyl bílý znak. Seznam parametrů bude vždy zapsán na jednom řádku, což není třeba kontrolovat (+1,5 bodu).
- IFTHEN: Podporujte zjednodušený podmíněný příkaz **if** - **then** bez části **else** , rozšířený podmíněný příkaz s volitelným vícenásobným výskytem **elsif** - **then** části a ve výrazech podporujte ternární operátor **?:** (+1,0 bodu).

13 Opravy zadání

- 19. 9. 2018 – Soubor ifj18.rb (viz Soubory ve WIS) doplněn o redefinici print a kvůli rozsahu vypuštěn z textu zadání. Drobné změny jsou popsány na fóru.
- 24. 9. 2018 – Doplnění instrukce **EXIT** a upřesnění způsobu reakce na chybu typové kontroly ve výrazech a vestavěných funkcích. Špatný počet parametrů při volání funkce je nyní chyba 5 a dělení nulou 9. Oprava sazby diakritiky.