VYP Project Description

Luboš Lorenc
email: lorenc@fit.vutbr.cz
translation: Tomáš Kopeček
email: kopecek@fit.vutbr.cz

October 5, 2005

1 General Information

Project name: Implementation of Imperative Language *VYP05* Compiler.

Consultations:

- Ing. Luboš Lorenc (e-mail reservations are preferred)
  - Monday 13:15 – 14:50
  - in exceptional cases possibly at another time (on agreement)
- Ing. Rudolf Schönecker (e-mail reservations are preferred)
  - Wednesday 9:00 – 11:00
  - in exceptional cases possibly at another time (on agreement)

Date and way of commitment: Until 22nd of December 2005, 13:00, only by using of IS FIT.

Classification:

- 25 points at maximum (20 programming part, 5 documentation part).
- Documentation is allowed only with program. Only documentation is classified with maximum of 1 point.
- There is a possibility of achieving special points (beyond maximum) for creative approach (extensions ,etc.) with additional maximum of 25 points.

2 Assignment

Define programming language *VYP05* (described later) by appropriate grammar. Create syntax-driven compiler based on this grammar. Compiler will translate this language into the machine pseudo-code so it will be runnable in given interpreter.

All regular expressions used throughout following text are written if GNU like form used for extended regular expressions. Eventual spaces in regular expressions are typed almost everywhere (except cases of separation of two consecutive words) only for better readability and they are not the part of regular expression itself. Keywords are enclosed in apostrophes while character apostrophe is not the part of the language. Word `id` is always used as an substitution of any identifier. Closer description of regular expressions could be find for example in GNU/Linux systems after typing command `info grep`.

2.1 Programming Language Description

2.1.1 General Properties and Data Types

It is simple imperative language with two basic data types.

- Case of letters matters.
- Identifier **id** is defined by regular expression `[a-zA-Z][a-zA-Z0-9_]*`
- Whole number (integer) literal **int** (constant) is defined by expression `[0-9]+`
- Floating point number literal **float** (double precision) is defined by regular expression `[0-9]+\.[0-9]*(e[+-]?[0-9]+)?`
- Data types for these literals are marked with keywords
  - **int**
  - **float**
- There is a special data type **void** used only in conjunction with function result type. It marks void type — so it marks procedure instead of function.

2.1.2 Function Definitions

Language contains this definition of functions and procedures (only at global scope):

```
type id ( \(  parameters \) )? [{] body [}]
```

Declarations of parameters are separated by semicolon. Last declaration is not terminated by semicolon. Every declaration could contain keyword `'ref'`. This keyword marks that variable is called by reference (implicitly they are called by value). This parameter declaration is described by the following regular expression:

```
type ('ref')? id  (; type ('ref')? id)*
```

Function body contains algorithm itself written like this:

```
(stmt)*
```

Each statement `stmt` of language contain following language constructions:

- **Variable declarations:** `'var' dcl (, dcl)* ;`
  - Each `dcl` looks like this: `id (\[count\])? (, id (\[count\])?)* : type` where `type` is one of the two possible data types and `count` is number of elements in case of array declaration. Minimal array size is one element. Arrays are always indexed from zero (index of first element is 0). Part `\[count\]` is forbidden if it is not an array declaration.
  - Variables could be declared in arbitrary sequence. Redeclarations are forbidden. When there is a second declaration of declared variable there will be semantical error.
- **Conditional branching:**
  `'if' \( expr \) [{] (stmt)* [}] ('else' [{] (stmt)* [}])? 'fi'`
  - Condition is met when result of expression is non-zero..

  - Generate *short evaluation* code for connectives.

- **Cycle:** `'while' \( expr \) 'do' [{] (stmt)* [}] 'done'`
  - Expression `expr` holds same conditions as with conditional branching.
  - Semantics of this construction is same as semantics of construction *while* in C/C++ languages.

- **Function call:** `id \( (expr (, expr)*)? \) ;`
  - There must be the same number and types of real parameters as in definition of function (in case of function without parameters empty parenthesis will be used).
  - When function returns value then this value will be ignored.
  - Generate *short evaluation* code for connectives.
  - Functions can be called recursively without limitation on depth of recursion.
  - When parameters are given by reference then parameter is absolute address (compiler does not compute any conversions).

- **Assignment:** `id := expr ;`
  - Identifier must be declared before and expression must be of corresponding type.
  - Generate *short evaluation* code for connectives.

- **Function return value:** `'ret' expr ;`
  - Expression must be of adequate type.
  - Generate *short evaluation* code for connectives.

Every program have to contain special function `main` which is the main body of program. This function has no parameters and it does not return any value. Function `main` can be defined anywhere in source file. If you will not make some extension then all declarations and definitions after this function will be unreachable nevertheless. If there will be a case that there is some text after function `main` user should be informed with warning message.

Function return value could be ignored so every function could be used as a procedure without emitting error message (see function call example).

2.1.3 Global Variable Declarations

Language *VYP05* does not include any support for global variables implicitly. If you want then you can create some support as an extension with accepting of following policies.

Global variables could be declared in any point of program (except of function bodies), visibility is from the point of declaration. Global variables declaration looks like this:

```
id (\[count\])? (, id (\[count\])?)* : type ;
```

Meaning of all parts is same as in the local variable declaration. Variables can be also declared in arbitrary order. Redeclarations are not allowed. Redeclaration will cause a semantical error.

2.1.4 Expressions

Expressions can contain integers and floating point numbers, references to variables, function calls (also recursive calls) and following operators and function calls. Binary arithmetical operators and binary connectives are left-associative. Compiler must support implicit retyping of expressions of type **int** to type **float**. Unary operators @, $\#i$ a $\#f$ are right-associative.

+, −, ∗, /	T x T → T, T is int or float. Operation / is for whole number operands defined as whole number division.
%	int x int → int (modulo).
=, <>, >, <, >=, =<	T x T → int, T is int or float.
@	Unary prefix operator. T → int, T is of an arbitrary type. Gets absolute address of operand in memory (reference). In case of array the base address of array is returned. Possible extension could implement getting of address of any array element.
$\#i$	Unary prefix operator. int → int. Memory cell in given address is interpreted as a number of type **int**.
$\#f$	Unary prefix operator. int → float. Memory cell in given address is interpreted as a number of type **float**.
and, or	T x T → int, T is int or float.
not	T → int, T is int or float.
readint()	Without parameters → int. Reads number of type int from standard input.
readfloat()	Without parameters → float. Reads number of type float from standard input.
printint()	int → without return value. Writes parameter int to standard output.
printfloat()	float → without return value. Writes parameter float to standard output.

Expression can contain parenthesis and unary minus. Precedence of operators is as follows (biggest priority first):

()	parenthesis, function call
$@, \#i, \#f$	
−	unary minus
∗, /, %	
+, −	
>, >=, <, =<	
=, <>	
not	
and	
or	

Note on operators @, $\#i$ and $\#f$. These operators does not examine relevance of referenced data in any way. For variable `x` of type **int** will command `x:=#i123` result in writing memory

part from byte 123 to byte 126 byte-by-byte without looking if it is some meaningful integer number. If `x` and `y` are variables of type **int** then `x:=#i@y` is correct copying of value of `y` into variable `x`.

2.1.5 Notes on implementation

Design and realization of compiler could be done in any possible way. Because of size of compiler and current trends in area of compilers I suggest maximal use of automatized tools for compiler parts generation. Mainly tools **lex** or **flex** for scanner generation and **bison** or **yacc** for LALR parser generation.

Support of functions readint, readfloat, printint a printfloat is implemented in given interpreter with use of special instructions (see chapter 2.2.2). There is only need to make correct linkage on generated code level.

All error and warning messages should be written to *standard error output*. Of course that there will be nothing else then warnings and errors on this output. *Breaking of this rule is evaluated as non-delivered project!*

2.2 Target machine pseudo-code and target file format

Target file must correspond to following fixed format.

File is read by lines. Each word could be separated by arbitrary separator. Normal space is recommended. Word is block of text separated from both sided by one or more separators. Separators are standard separator characters like space, tabulator, etc. (exact list could be find in documentation of standard C macro *isspace*()). Before first and after last word in the line is no separator needed.

2.2.1 Target File Description

- Target file is composed as a sequence of instructions. Each instruction is on separate line.
- Every line beginning with % is supposed to be a remark and during interpretation will be skipped. After each instruction can be remark also without any introductory character (Interpreter reads exactly number of words needed to compute given instruction. The rest is skipped.)
- Interpreter can be switched to the interactive debugging environment. This mode will be started when line composed from the only character @ is read.

2.2.2 Target Processor and Instruction Set Description

Target processor consists of:

- 16 general registers numbered 0–15,
- program counter (PC register),
- base pointer register (BP register),
- byte-organized memory with size defined in the time of start of the interpreter,
- byte-organized stack with size defined in the time of start of the interpreter,
- debugging environment.

'X' means one of letters I or F in the following text (for operand type **int** or **float**).

- **Arithmetic operations**
  - $x + y \rightarrow z$; x, y, z are register numbers
    `ADDX x y z`
  - $x - y \rightarrow z$; x, y, z are register numbers
    `SUBX x y z`
  - $x * y \rightarrow z$; x, y, z are register numbers
    `MULX x y z`
  - $x/y \rightarrow z$; x, y, z are register numbers (it is whole number division for I)
    `DIVX x y z`
  - $x\%y \rightarrow z$; x, y, z are register numbers (modulo)
    `MOD x y z`
- **Comparison operators** — First two operands can be of any (but both the same) type and target operand (register) is a number of type **int**.
  - $x > y \rightarrow z$; x, y, z are register numbers
    `GTX x y z`
  - $x >= y \rightarrow z$; x, y, z are register numbers
    `GEX x y z`
  - $x < y \rightarrow z$; x, y, z are register numbers
    `LTX x y z`
  - $x <= y \rightarrow z$; x, y, z are register numbers
    `LEX x y z`
  - $x == y \rightarrow z$; x, y, z are register numbers
    `EQX x y z`
  - $x <> y \rightarrow z$; x, y, z are register numbers
    `NEX x y z`
- **Zero value test** — this operation can have first operand of any type. Result is of type **int**.
  - $x == 0 \rightarrow y$; x, y are register numbers
    `ISZX x y`
- **Memory operations**
  - $mem(x) \rightarrow y$ — contents of memory at address given by register x is written to register y. If instead of register number x is given any number preceded by character # then it is direct address of the memory cell.
    `LDDX x y`
  - $mem(x+BP) \rightarrow y$ — contents of memory at address given by register x incremented by value of register BP is written to register y. If instead of register number x is given any number preceded by character # then it is direct address of the memory cell.
    `LDBX x y`
  - $x \rightarrow mem(y)$ — contents of register x is written to the memory. Computation of address is the same as in instruction LDDX.
    `STDX x y`

- $x \rightarrow mem(y + BP)$ — contents of register x is written to the memory. Computation of address is the same as in instruction LDBX.
  `STBX x y`

These instructions always read or writes correct number of bytes belonging to the type of instruction. So for whole number operations it is always four bytes. Warning — addresses are not multiplied by size of operands!

- **Program Counter Operations**
  - $PC \rightarrow x$ — contents of program counter (this instruction) is written to the register x. After that is PC incremented. `LDPC x`
- **Operations for Constants**
  - $inum \rightarrow reg$; reg is a register number, $inum$ is a decimal constant.
    `LDCI inum reg`
  - $fnum \rightarrow reg$; reg is a register number, $fnum$ is a floating constant in format
    `-?[0-9]+(\.[0-9]*)?(e-?[0-9]+)?`
    `LDCF fnum reg`
- **Move Operations**
  - $reg1 \rightarrow reg2$; $regx$ is a register number
    `MOV reg1 reg2`
- **Type Conversions**
  - $x \rightarrow y$; x is a register number. x contains value of type **int**, y is a number of register to which will be written this value converted to the type **float**.
    `I2F x y`
- **Jump Instructions** —

  Program is interpreted as indivisible block of instructions numbered from 0. Jumps are always absolute (JMP 10 will jump to 11th instruction from the beginning of the program, JMP 0 to first one, ...).

  - Conditional jump to the instruction while non-zero register
    `JTX reg instr`
  - Conditional jump to the instruction while zero register
    `JFX reg instr`
  - Unconditional jump
    `JMP instr`
- **Stack Operations** — these instructions pushes (or pops) number of bytes corresponding to the type of operand (e. g. for **int** 4 bytes).
  - $reg \rightarrow stack$; reg is a register number
    `PUSHX reg`
  - $stack \rightarrow reg$; reg is a register number
    `POPX reg`
- **Subprogram calls**

  - Subprogram (function) calls
    `CALL addr`
    Onto the stack is written value PC+1 (for continuing with next instruction after returning from subprogram) and then is PC set to the value `addr`. Constant `addr` must be an instruction of called subprogram. Do net forget that interpreter will not make stack frame for you. Also is good or necessary to write correct new base address to the register BP.
  - return from procedure or function.
    `RET`
    Gets the topmost value from the stack and writes it to the PC. If there is some return value then subprogram must make correct work itself. It have to pop saved PC, push return value and push PC. Again you have to take care of stack frame.

- **Pseudo-instructions for input/output operations**
  - *reg* → *stdout*; *reg* is a register number. Writes contents of register to the standard output as an **int**.
    `PRINTINT reg`
  - *reg* → *stdout*; *reg* is a register number. Writes contents of register to the standard output as a **float**.
    `PRINTFLOAT reg`
  - *stdin* → *reg*; *reg* is a register number. Reads number of type **int** from the standard input into the register.
    `READINT reg`
  - *stdin* → *reg*; *reg* is a register number. Readint number of type **float** from the standard input into the register.
    `READFLOAT reg`

3 Design and Delivery Instructions

3.1 General Information

All files will be compressed by programs TAR+GZIP or ZIP into the one archive. File name will be login.tgz or login.zip. Archive can not contain directory structure and special files. All file names inside archive will be from language defined by this regular expression `[a-z_.0-9]+`. Solutions will be tested by automatized tools and so violation of these rules will result into zero points. Archive file will be delivered to the IS FIT.

Whole project must be delivered before given date (see above). Also here can be project evaluated for zero points. At least there will be some point penalty corresponding to the delay. Similarly there will be zero point result for any plagiary work.

3.2 Solution Requirements

3.2.1 Text Part of Solution

Documentation will be part of solution. Format of documentation will be PDF or PS — other format is disallowed. Documentation will be in **Czech or English language**. This documentation have to include:

- Name, surname and login of author,
- description of your way of solving — cca 2–4 pages (pictures does not count),

- notices about possible extensions against basic requirements — corresponding to the implementation.

Documentation is forbidden to conclude:

- Copy of this file,
- text, which is not of your authorship (copies from lectures, network, WWW, ...)

Documentation will be saved in the only file with name login.XXX where XXX is **pdf** or **ps**. Any other format of documentation than these two are ignored, so it leads to the point loss.

3.2.2 Programming Part of Solution

Programming part of solution will be in language (ANSI) C/C++. Program must compile with compiler gcc/g++. Projects will be compiled under OS GNU/Linux for testing. Mind this especially if you will be working in another OS. If program couldn't be compiled or if it will be not working because of use of some function or non-standard implementation technique depending on OS you will lost your right of reclamation of results. In polemic cases will be always crucial result of compilation on server merlin.

Part of solution will be file login**.mak** supplying file Makefile (projects will be compiled as `gmake -f login.mak`[1]).

Binary file (compiled compiler) is definitely not meant to be delivered as a part of archive (archive will be large a binary file will be deleted — furthermore there will be point loss!). If there will be not makefile or compilation based on this file will file then there will be zero points. Notice that everything have to compile in environment of FIT network **without** any additional settings and only in given types of compilers.

First three lines of source files and documentation have to include name of the project, login of student and his name. If this condition will not be met, then project will be evaluated by 0 points.

Program takes name of the input file from the command line. Output of program will be saved into the file with the same name but different suffix **.out** and into the same directory as an input file. In case that input file has no suffix then only suffix **.out** will be added. For example for compilation `compile Tests/test01.vyp` will be created file with name *test01.out* in directory *Tests*. If not working — again 0 points.

All error messages will be always written only to the **standard error output**. Except errors and warnings will compiler emit no other messages. Same situation with generated program — it will display only information coded in source program! Projects will tested by automatized tools with set of testing examples and tools like diff. Printing of only one unexpected symbol will probably lead to the wrong result of your response and at least loss of some points. *Generally this type of error can lead to zero points!!!*

3.3 Consultations

Consultations will be held in terms given above and in exceptionally cases other times. Agreement is recommended. Agreement can solve eventual concern of more students at the same moment. If you need consultation please be prepared. Project is nontrivial. So if you will ever have a sense that you are lost or something is not clear — do not hesitate and contact me by an e-mail. There will be no consideration for late laments.

[1] Makefile is a file used by program make. It is definitely not a shell script! More information you will find in `info make` or at http://www.gnu.org. (File Makefile is only renamed to the login**.mak**)

3.4 Some Recommended Extensions

- register assignment
- other optimizations
  - constant folding
  - SHORT-EVALUATION chaining
  - optimization of cycles
  - dead code elimination
  - ...
- new arithmetic or logical operators with right associativity (e. g. square)
- global variables
- new program constructions
  - cycle *do — while*
  - cycle *for*
  - multiple conditional branching *case – of*
- remarks (trivial extension)

3.5 Principles of Point Distribution

Main principles for evaluating of delivered project:

- Program must generate at least some output which corresponds to the simplest input and interpreter should be able to correctly interpret it.
- If there is only a documentation then there is a maximum of one point for it.
- If program generates no output or output is not correctly or it does not correspond to the source program then program part is at maximum evaluated by two points.
- Testing will be on a set of testing examples (source tests). Every compiled example will be run in interpret and its output will be compared to the correct output. For each correctly compiled testing example there will be some sum of points.

3.5.1 Project is evaluated by 0 points if

- Will not be delivered at time (each new day of delay at least 20% point loss),
- will not be in proper format,
- couldn't be correctly compiled as is written in this description (e. g. Makefile must be changed, manual setting of paths, etc.),
- it will be a plagiary work of any kind,
- output of program will be not corresponding with description.
- ...

3.6 Project Howtos

Naturally it is possible to use own computer for solution. Installation of compiler gcc/g++ is not necessarily needed. If you have another compiler C/C++ and you do not want to use abilities of this exact one there is no problem. While using only C language there should be no problem at all. With C++ language you should check if your *state-of-the-art* constructions work also in gcc/g++. After finishing project you always should check everything in target compiler (and at server merlin) if it is working without problem.

For solution itself is good to make progress as this: your product should generate functional and correct code for some simple program constructions firstly. E. g. it can compile addition of two numbers and print result. Reason is that this thing allows classification of whole project and to admit more than symbolical points.

So you will make lexical analyzer at first. In second step you design grammar and after that you will make syntax analysis based on it. This analyzer can cope with definition of functions and printing of expressions with constants. Above these components you can build simple code generator. So you will have some basic microcompiler which is at least something.

After these steps you can add other symbolical constructions and other pieces of code generator. Correspondingly you make semantic analysis.

I recommend to take some time for implementation of a few extensions. Many times is sufficient to make little mistake caused by incautious and compiler crash can happened on a big number of testing examples. In this case is point classification non-adequate to the effort often. So implementation of recommended extensions show your concerment in the problematics. If you come for the reclamation of large and sophisticated project with extensions will have better chance for success than discussion above simple code full of errors evidently written in a hurry and without sufficient knowledge.

3.7 Additional Conditions

Testing of projects will be made on the newest version of compiler gcc which will be available at the date of testing. Currently it it version gcc 3.4.4. In the time of testing it will be probably some product of line 4.0.x.

3.8 Attitude to academical dishonesty

If you make some misconduct against academical moral codex (e. g. plagiary) your fault will be transcended to guarantee of course who is simultaneously chair of Disciplinary committee of FIT BUT. This problem typically leads to no credits for course. We explicitly notes that it can also lead to excommunication of academical community and so of the whole BUT.

Contents