BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Formal Languages and Compilers / Algorithms

Implementation of an imperative language IFJ15 interpreter

Documentation

13th December 2015

Tomáš Ženčák, xzenca00	25 %	Team 036, variant a/2/II
Ondřej Valeš, xvales03	25 %	Extensions: SIMPLE, FUNEXP, WHILE
Nikola Valešová, xvales02	25 %	
Radek Vít, xvitra00	25 %	

Contents:

1. Introduction

This documentation describes the development and implementation of an imperative language IFJ15 interpreter. The whole documentation is divided into chapters, which define design and development of the sub-sections of the program. It also includes attachments, such as the LL-grammar, the precedence table and the finite state machine diagram, which we used to create the whole interpreter.

2. Task

The IFJ15 language is a subset of the C++ language. Both languages are case-sensitive (they distinguish between uppercase and lowercase letters) and they are both statically typed (they offer the basic deriving of data types and the data types of variables are destined in advance).

We divided the work on this project into subsections. All of them are very important and we would not be able to compile the whole interpreter without any of them. The basic tasks are:

- lexical analysis
- syntactic analysis
- semantic analysis
- inner code generator
- built-in functions
- testing
- interpret

2.1 Variant of the task

Our chosen variant of the task was a/2/II:

- Knuth-Morris-Pratt algorithm
- Heap sort
- Hash table

3. Implementation

The whole interpreter consists of the following parts:

3.1 Extensions

We decided to implement the following extensions:

- SIMPLE – supports using conditional statement if without the else branch and allowing to use one statement instead of a compound statement with conditional statements and cycles,
- FUNEXP – calling a function can be a part of an expression and expressions can be used as parameters of functions when calling it,
- WHILE – the interpreter supports also while and do-while cycles.

3.2 Lexical analysis

The input of a scanner is a source program containing a source code. The scanner reads the source code character by character and transforms lexemes into tokens. It works as a finite state machine. Whitespace characters and comments are read but they are not sent to parser, scanner continues in reading lexemes. After a lexeme is read successfully, scanner sends parser the corresponding token. In order to get next token, parser calls scanner by function *get_token()*. When an error occurs in the source code, the scanner is able to recover and return a specified return value.

3.3 Syntactic analysis

The main parser was written as an LL-table-driven predictive parser. Expression parsing is done using a precedence parser, which is called by the main parser whenever an expression is encountered. Given that there is no context required for determining what action should be taken for any given nonterminal string and input terminal, and therefore little to no interaction was required between the two parsers, implementing both of them was fairly straightforward.

3.4 Semantic analysis

The semantic analyser was written separately from the parser in order to minimize dependencies between the two. This enabled us to efficiently test both analysers, as well as quickly pinpoint which analyser a bug was in. The semantic analyser analyses the sequence of rules generated by the parser statement by statement, with only the symbol table and scope information being preserved between statements.

The problem of variable name shadowing is solved by simply allowing duplication in the symbol table, with variables with the same name stored with LIFO semantics, therefore simulating a stack of names, when the relevant information is in fact kept in a hash table.

A slight problem with implementing the SIMPLE extension was that when the brace less notation is taken into account, the end of the scope can come after any statement, as opposed to only occurring at the end of a statement list. This was solved by adding a “brace-enclosed” flag to all scopes and dividing statements into scope creating and scope destroying. Scope creating statements created their own scope without the brace enclosed flag, while scope destroying statements ended all enclosing non-brace-enclosed scopes. The composite statement would set the “brace-enclosed” flag of the innermost scope, unless it was already set, in which case it created its own “brace-enclosed” scope. End of a composite statement always ended the innermost scope and then any number of non-brace-enclosed scopes. This allowed us to retain statement independence.

Another extension we have decided to implement, FUNEXP, also posed a problem. For optimization purposes we have decided to defer pushing constants and named variables to the top of the stack until necessary and our interpreter’s calling conventions dictate that all parameters must be pushed to the execution stack in order. When coupled by the LR derivation created by the expression parser, this proved to be somewhat problematic as the first parameter of a function call can only be recognized as such after the second parameter is already completely processed. If the first parameter wasn’t pushed and the second had to be computed at runtime and therefore the instructions to push it, as well as the appropriate arithmetic instructions, were already generated, wrong parameter order would result. Fortunately, thanks to a good intermediate instruction representation, it was possible

to simply insert a push instruction before the calculation of the second parameter without disrupting any references in the calculation of the second parameter.

Implementing the WHILE extension was fairly simple, as the existing code architecture made adding new types of cycles very simple.

3.5 Instructions and interpreter

Our interpreter simulates a stack machine with random access to all stack elements. Instructions of each user defined functions form directed graphs, where each instruction (node) has one or two successors (left and right successors; all instructions but the last of each function have a left successor). This allows for both conditional and unconditional jumping. Each instruction determines which of its two successors will assume the position of the next instruction at runtime. Function calls are realized with a separate stack of return positions and immediate execution of the called function's instructions. Upon returning the last instruction of the caller's last performed instruction is popped from the stack and the left-hand following instruction is executed.

Our instruction set is moderately simple, as all instructions perform elementary tasks. Some instructions can perform actions that do not directly simulate a stack machine when called with the appropriate parameters. This allows optimization when generating code, as immediate values can be compared or otherwise operated without needing to explicitly push them to stack.

Built-in functions are realized as operations behaving the same way as user defined functions, although they are executed within a single instruction. This makes the built-in functions much faster than user defined functions.

We optimised our interpreter by storing strings' lengths (thus limiting the need to calculate their lengths greatly) and minimising excess copying of strings when copying them. These two optimizations alone caused a heavy (up to two times) increase in performance in string-heavy interpreted programs.

3.6 Hash table

The hash table is implemented as an array of singly linked lists which allows duplication and stores newly inserted synonyms at the start of the list. This means that items with the same key have LIFO semantics, which was extremely helpful when the hash table was used as a symbol table during semantic analysis.

The hash function is based on a vague recollection of the hash function we used last semester in C Language course.

3.7 Knuth-Morris-Pratt algorithm

In the function *find*() is implemented the Knuth-Morris-Pratt algorithm that searches for the first occurrence of a string in a text. First it creates fail table containing indices of string where search can continue after failed match. Then text is compared letter-by-letter with string, if a mismatch occurs index of string is changed according to fail table and search continues. When end of string is reached, the search ends successfully, when end of text is reached, the search fails. The return value is either

the index where the match started or -1 if no match was found. When an error occurs, internal error is invoked.

3.8 Heap sort

Our implementation of heap sort is pretty much just rewriting of the algorithm, as presented during IAL lectures, into the C language. The only slight deviation from standard heap sort is in the fact that our sift-down procedure first decrements the pointer to the heap in order to get 1-based indexing. This eliminates some adjustments necessary in 0-based indexing.

4. Testing

Our aim was to test every function and every part of the program thoroughly. Every week we split the tasks to program and test, so that every member or our team got to try both kinds of work and every part of the code was not tested by its developer. When the whole interpreter was completed, all of us started testing the functionality of our program and when anyone found an error, we posted the corresponding IFJ15 program to our group and tried to fix it.

5. Working in a team

We set up weekly meetings, where we discussed the ways of solving current problems and split the tasks for the following week. We also agreed on the interface among parts of our interpreter in order to avoid problems with compatibility. We used instant messaging via social networks to communicate after school and a GIT repository and Google Drive to share files.

5.1 Division of work

We started by implementing the built-in functions and the Knuth-Morris-Pratt search algorithm, which were implemented by Ondřej Valeš, the heap sort and hash table implementation realized by Tomáš Ženčák, the main file and error functions and warnings created by Nikola Valešová and lexical analysis done by Radek Vít and Nikola Valešová. With these functions and modules finished, we started testing. After that we went on with creating the syntactic and semantic analysis, which were mostly implemented by Tomáš Ženčák. We decided to include the SIMPLE and FUNEXP extensions and to implement them when creating the parser for the first time. Afterwards we started writing instructions that represent the basic tasks of an interpreter and the core of our interpreter, which executes the source program statement by statement. In the end, we decided to implement the WHILE extension as well and to add it to our LL-grammar.

When any part of the interpreter was done, those of us who weren't involved in implementing this particular part started testing in order to find mistakes and weak points in the source code.

6. Conclusion

Implementation of the IFJ15 interpreter has been the biggest challenge during our studies so far. We managed to create a functional program with some extensions as well. Every one of us gained experience with teamwork and also programming.

7. Code metrics

Lines of code: 1228

Comment lines: 383

Number of files: 32 + Makefile

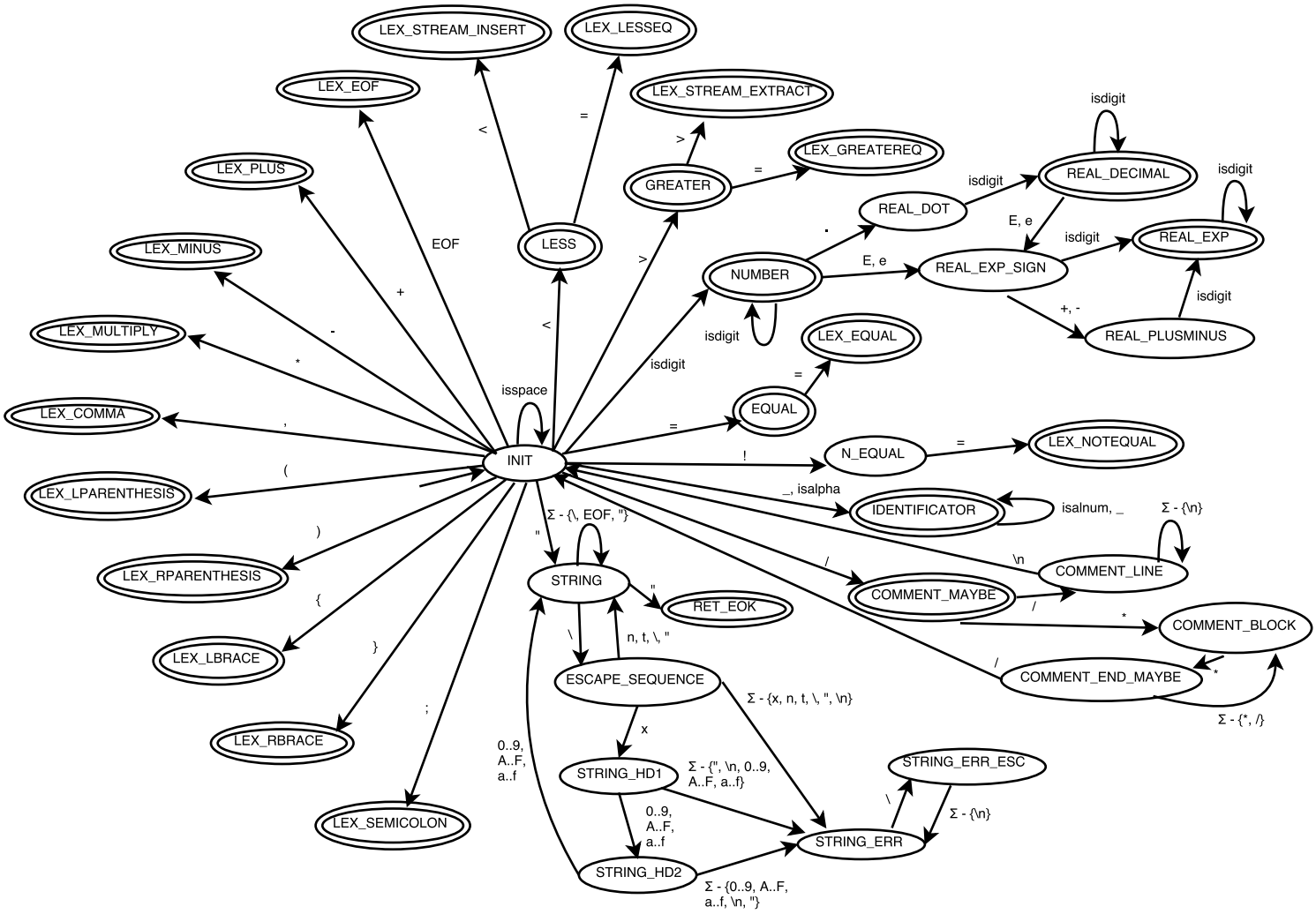
Size of executable file: 109,316 B (Linux system, 64bit architecture)

8. References

[1] HONZÍK, Prof. Ing. Jan M., CSc. *Algoritmy: Studijní opora*. 14-Q. 2014.

9. Attachments

9.1 Finite state machine diagram



9.2 LL-grammar

[func_decl_list] ⇒ [func_decl][func_decl_list]
[func_decl_list] ⇒ LEX_EOF
[func_decl] ⇒ [type]LEX_ID([func_decl_arg_list])[opt_func_body]
[func_decl_arg_list] ⇒ [type]LEX_ID[func_decl_arg_list_cont]
[func_decl_arg_list] ⇒ ε
[func_decl_arg_list_cont] ⇒ ,[type]LEX_ID[func_decl_arg_list_cont]
[func_decl_arg_list_cont] ⇒ ε
[opt_func_body] ⇒ ;
[opt_func_body] ⇒ [composite_statement]
[statement] ⇒ [composite_statement]
[statement] ⇒ [if_statement]
[statement] ⇒ [assignment];
[statement] ⇒ [for_statement]
[statement] ⇒ [return_statement];
[statement] ⇒ [input];
[statement] ⇒ [output];
[statement] ⇒ [var_def];
[statement] ⇒ [while_statement]
[statement] ⇒ [do_while];
[composite_statement] ⇒ {[statement_list]}
[if_statement] ⇒ if([expr][expr_end])[statement][else_clause]
[else_clause] ⇒ else[statement]
[else_clause] ⇒ ε
[statement_list] ⇒ [statement][statement_list]
[statement_list] ⇒ ε
[assignment] ⇒ LEX_ID=[expr][expr_end]
[for_statement] ⇒ for([var_def];[expr][expr_end];[assignment])[statement]
[return_statement] ⇒return [expr][expr_end]
[input] ⇒ cin>>[term][input_chain]
[input_chain] ⇒ >>[term][input_chain]
[input_chain] ⇒ ε
[output] ⇒ cout<<[term][output_chain]
[output_chain] ⇒ <<[term][output_chain]
[output_chain] ⇒ ε
[term] ⇒ LEX_ID
[term] ⇒ LEX_NUMBER
[term] ⇒ LEX_REAL_NUMBER
[term] ⇒ LEX_STRING_LITERAL
[var_def] ⇒ [type]LEX_ID[opt_var_init]
[var_def] ⇒ auto LEX_ID[opt_var_init]
[opt_var_init] ⇒ [var_init]
[opt_var_init] ⇒ ε
[var_init] ⇒ =[expr][expr_end]
[while_statement] ⇒ [while_clause][statement]
[do_while] ⇒ do[statement][while_clause]

[while_clause] ⇒ while([expr][expr_end])
[expr_end] ⇒ ε
[type] ⇒ int
[type] ⇒ double
[type] ⇒ string
[expr] ⇒ ([expr])
[expr] ⇒ LEX_ID
[expr] ⇒ LEX_NUMBER
[expr] ⇒ LEX_REAL_NUMBER
[expr] ⇒ LEX_STRING_LITERAL
[expr] ⇒ [expr]+[expr]
[expr] ⇒ [expr]-[expr]
[expr] ⇒ [expr]*[expr]
[expr] ⇒ [expr]/[expr]
[expr] ⇒ [expr]>[expr]
[expr] ⇒ [expr]<[expr]
[expr] ⇒ [expr]>=[expr]
[expr] ⇒ [expr]<=[expr]
[expr] ⇒ [expr]==[expr]
[expr] ⇒ [expr]!=[expr]
[expr] ⇒ LEX_ID()
[expr] ⇒ LEX_ID([expr])
[expr] ⇒ LEX_ID([func_call_arg_list])
[func_call_arg_list] ⇒ [expr], [expr]
[func_call_arg_list] ⇒ [func_call_arg_list],[expr]

9.3 Precedence table

relat = {<, >, >=, <=, !=, ==}
val = {NUMBER, REAL_NUMBER, STRING_LITERAL}

	+-	*/	relat	ID	val	(	)	,	$
+-	>	<	>	<	<	<	>	>	>
*/	>	>	>	<	<	<	>	>	>
relat	<	<	>	<	<	<	>	>	>
ID	>	>	>			=	>	>	>
val	>	>	>				>	>	>
(	<	<	<	<	<	<	=	<	
)	>	>	>				>		
,	<	<	<	<	<	<	>	>	
$	<	<	<	<	<	<			