BRNO UNIVERSITY OF TECHNOLOGIES
FACULTY OF INFORMATION TECHNOLOGIES

Implementation of compiler of imperative language IFJ22

Documentation

Team xzarsk04, Variation - TRP
Extension Identifiers

December 7, 2022

25 % Jiří Soukup (xsouku17)
25 % Samuel Šimún (xsimun04)
25 % Martin Tomašovič (xtomas36)
25 % **Daniel Žárský (xzarsk04)**

Contents

1 Introduction

The goal of the project assigned as part of the IFJ subject at FIT BUT Brno was to create a program in C language that will translate source code from standard input in IFJ22 language to IFJcode22 language to standard output. In case of an error, it returns the corresponding error output. The solution took place in teams of four.

2 Proposal & Implementation

2.1 Scanner

The lexical analyzer loads the text from the program input character by character. It is implemented as a Mealy finite state machine that is managed by both the current state and the loaded input.

It identifies the loaded lexems based on the characteristics of the input language and stores their type and optional value in the Token structure. If the automaton loads a lexically impermissible combination of characters, it returns a lexical error to the parser, which is further propagated to the standard error output.

It converts special characters, written using escape sequences, into the decimal system and saves them as a character. Numbers are stored as character strings and are formatted only during target code generation.

2.2 Syntactic analysis

The controlling part of the entire program is syntactic analysis. Parsing consists of two parts: **top-down** (uses LL grammar) and **bottom-up** (precedence parsing)

2.2.1 Parser

Parser is main part of the program which controls flow of the program. It is called at the beginning of compiling. Initialize all data structures and controls validating of beginning of program. It is implemented in **syntactic.c**.

It checks all parts of program and theirs return types and handling unexpected events. Parser takes first token of command and decides with which operation will be served. After that it calls functions from **gramatic_rules.c**.

When the end of program is detected and all tokens are stored in Token buffer, buffer is send to semantic control. If semantic control doesn't raise any error, the buffer will be generated by **generator.c**.

If all parts are successfully done, the program will deallocate all sources and terminate himself without error code.

2.2.2 Top to bottom

In addition to the processing of mathematical and logical expressions, the syntactic analysis is governed by the LL-grammar, which is implemented in the file **gramatic_rules.c** (the LL-grammar is attached at the end of the documentation).

Special functions have been created for all rules, individual functions are called recursively (this is a recursive descent method). a new token is requested from the lexer using the **Get_token(data)** function, its value then decides which function to call next. The data parameter gives a pointer to the structure in which the TODOs are located.

If it is determined by the rules to switch to solving expressions, the **check_expression** function is called and it switches to bottom up solving.

The initial prolog, as well as the first token on input, is processed by **syntactic.c**, which then calls the necessary function in **grammatical_rules.c**. Checks then take place in these functions until they recursively return to **syntactic.c**.

2.2.3 Bottom up

Syntactic analysis bottom up is implemented based on precedence analysis which depends on precedence table. Expression is divided into tokens which are pushed or replaced by rules based on precedence table.

Precedence analysis is done when on the top of the stack is $. If rule doesn't exist or precedence table sets combination as invalid, the error is occurred.

2.3 Semantic analysis

Semantics's check is performed according to the rules of **IFJ22 language**. It checks wether right values are assigned to variables and passed as arguments to functions. It also evaluates expression data types.

Checks function declarations. When semantics's error is detected, program ends and returns corresponding error value.

2.4 Generator

We have executed blind generation based on dynamic array of tokens which where loaded from include. The tokens remain unchanged during syntactic and semantic analysis. We have implemented functions that convert strings and floats from IFJ22 to the format of IJFcode22. It is implemented in **generator.c**

3 Algorithms and Structures

3.1 Token Buffer

The Token_buffer structure is implemented as a dynamic array of tokens, which is used to temporarily store them. Memory is dynamically allocated and *deallocated* by the *realloc* function. The field is indexed from zero and is stored in a syntax data structure.

3.2 Syntactic Stack

Syntactic_stack (implemented in **syntactic_stack.c**) is the auxiliary stack (LIFO) of syntactic analysis. It is implemented as a singly linked list to keep the structure dynamic.

3.3 Buffer

Buffer is a dynamic array (implemented in **buffer.c**) of characters. It is used to store the tokens value or id. It's size is only limited by the size of the heap.

3.4 Symbol Table

There are two types of abstract data types. More classic hash table used as storage for variables and phashtable used for storing information about function's parameters. There are adjustments in these ADT against classic ones. In the insert functions, there is mechanism returning error if the function already exists, so it helps find redeclaration errors. It is implemented in **symtable.c**

Hash function has slight adjustment it adds a constant to the ascii value of the first character of given input string. This is used to distinguish strings with characters that are in different order for example ”abc” and ”cba”.

4 Team work

4.1 General

We solved the project in a team of four. At the beginning, thanks to studying the lectures, we created a complete plan ahead of time. We started with the solution at the end of September, we submitted a trial submission. There was a slight delay compared to the plan

4.2 Communication

We met in person every week to inform ourselves about the current progress and to be able to solve any questions. In the end, before handing over, the number of meetings intensified and we also added online meetings.

4.3 Software

We used the GitHub versioning system to manage individual versions, thanks to which we could work simultaneously with the current version of all parts of the program.

We used the CLion program for development.

4.4 Split of work

The work on the project was divided equally depending on the capabilities of the given team members and regardless of the consequences on their mental health

NAME	*PERCENTS*	*WORK*
Daniel Žárský	25 %	fill out what did you do
Martin Tomašovič	25 %	fill out what did you do
Samuel Šimún	25 %	parser, expression, Token_buffer, Git master
Jiří Soukup	25 %	grammatical rules, precedence table, documentation

5 Conclusion

From the beginning of the semester, the project seemed manageable, but as time went on, we gained more information and the project became almost unsolvable.

Fortunately, however, we started work early and everyone worked together, so we managed to solve the problems gradually. The personal meetings that took place every week helped us the most. GitHub was also a big help for us, which enabled us to work concurrently.

We obtained information mainly from the lectures of the IAL and IFJ subjects, then we used open forums on the Internet and book sources.

During the solution we faced the ambiguity of the assignment, but after a personal consultation everything became clear.

This project was the biggest challenge of our studies so far. He brought us a lot of new knowledge, which, however, was redeemed by a large amount of time.

Doxygen documentation (**documentation**).

A LL – grammar

- <prog> -> TYPE_PROLOG_START <main_statements> <main-return> <prolog_end> EOF
- <main_statements> -> E
- <main_statements> -> <main_statement> <main_statements>
- <main_statement> -> <function-definition>
- <main_statement> -> <while>
- <main_statement> -> <asignment>
- <main_statement> -> <expression> TYPE_SEMICOLON
- <main_statement> -> <call-function>
- <main_statement> -> <condition>
- <prolog_end> -> E
- <prolog_end> -> TYPE_PROLOG_END
- <function-definition> -> KEYWORD_FUNCTION TYPE_FUNCTION_ID LEFT_BRACKET <f_params> RIGHT_BRACKET TYPE_COLON <type_function>
- <type-function> -> <data_type> LEFT_COMPOUND <f-statements> KEYWORD_RETURN <return-rest> RIGHT_COMPOUND
- <type-function> -> KEYWORD_VOID LEFT_COMPOUND <f-statements> <return> RIGHT_COMPOUND
- <data_type> -> KEYWORD_STRING_Q
- <data_type> -> KEYWORD_INTEGER_Q
- <data_type> -> KEYWORD_FLOAT_Q
- <data_type> -> KEYWORD_FLOAT
- <data_type> -> KEYWORD_INTEGER
- <data_type> -> KEYWORD_STRING
- <f-params> -> E
- <f-params> -> <data-type> TYPE_VARIABLE_ID <f-rest-params>
- <f-rest-params> -> E
- <f-rest-params> -> COMMA <data-type> TYPE_VARIABLE_ID <f-rest-params>
- <main-return> -> E
- <main-return> -> KEYWORD_RETURN <return-rest>
- <return> -> KEYWORD_RETURN TYPE_SEMICOLON

- <return> -> E
- <f-call-params> -> E
- <f-call-params> -> TYPE_VARIABLE_ID <f-call-rest-params>
- <f-call-rest-params> -> E
- <f-call-rest-params> -> TYPE_COMMA TYPE_VARIABLE_ID <f-call-rest-params>
- <f-statements> -> E
- <f-statements> -> <f-statement><f-statements>
- <assignment> -> TYPE_VARIABLE_ID TYPE_EQUAL <after-equal> TYPE_SEMICOLON
- <after-equal> -> <expression>
- <after-equal> -> <call-function>
- <f-statement> -> <call-function>
- <f-statement> -> <assignment>
- <f-statement> -> <while>
- <f-statement> -> <condition>
- <f_statement> -> <expression> TYPE_SEMICOLON
- <return-rest> -> <expression> TYPE_SEMICOLON
- <return-rest> -> <call-function>
- <while> -> WHILE LEFT_BRACKET <expression> RIGHT_BRACKET LEFT_COMPOUND <f-statements> RIGHT_COMPOUND
- <condition> -> IF LEFT_BRACKET <expression > RIGHT_BRACKET LEFT_COMPOUND <f-statements> RIGHT_COMPOUND ELSE LEFT_COMPOUND <f-statements> RIGHT_COMPOUND
- <call-function> -> TYPE_FUNCTION_ID LEFT_BRACKET <f-call-params> RIGHT_BRACKET TYPE_SEMICOLON

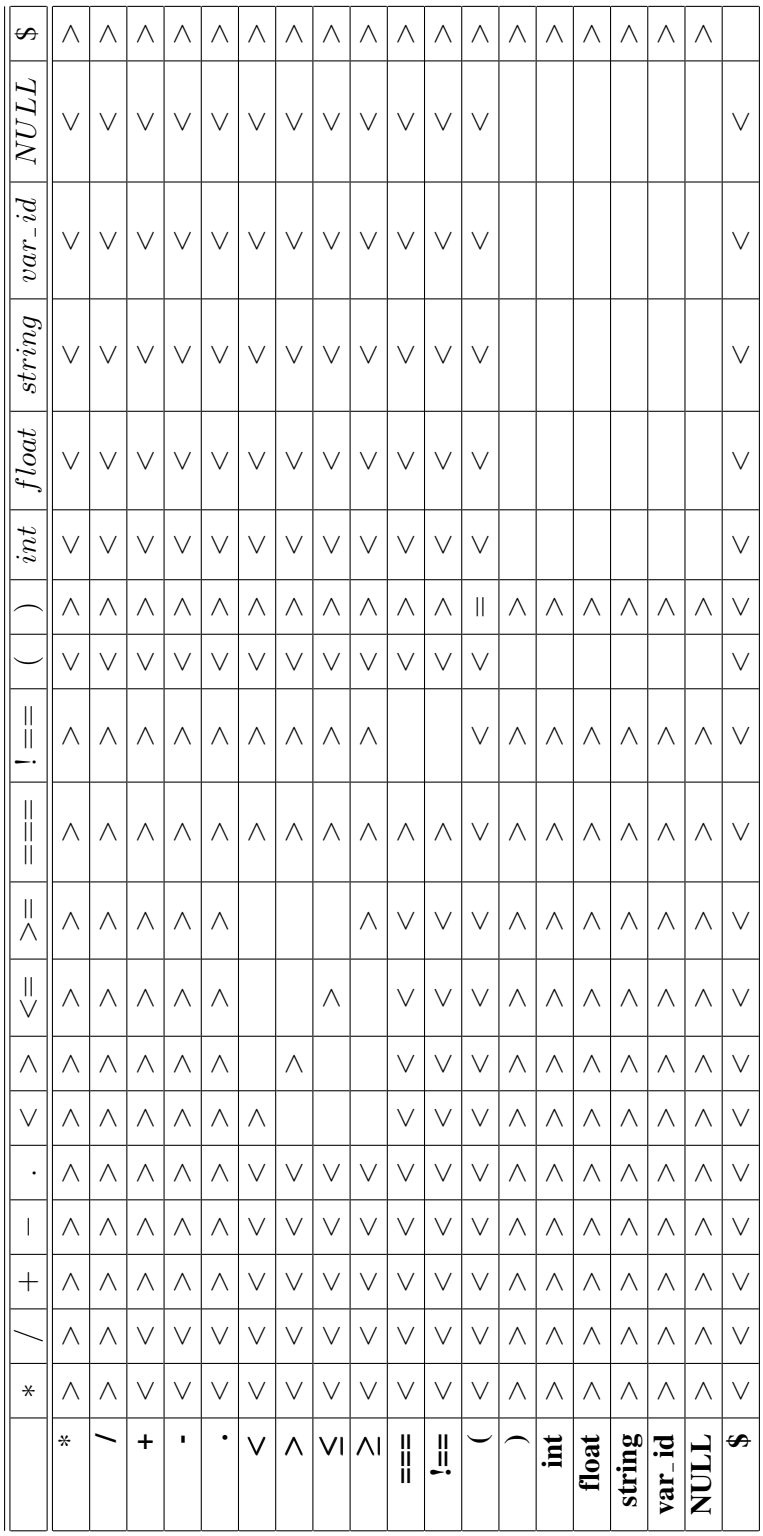
Table 1: LL – grammar for syntax analysis

B LL – table

	TYPE_PROLOG_START	E	EOF	TYPE_SEMICOLON	TYPE_PROLOG_END	KEYWORD_FUNCTION	TYPE_FUNCTION_ID	LEFT_BRACKET	RIGHT_BRACKET	TYPE_COLON	KEYWORD_RETURN	TYPE_COMPOUND	KEYWORD_VOID	KEYWORD_INT	KEYWORD_FLOAT	KEYWORD_STRING	TYPE_INTEGER	TYPE_FLOAT	TYPE_STRING	TYPE_VARIABLE_ID	TYPE_COMMA	TYPE_EQUAL	KEYWORD_WHILE	KEYWORD_IF	KEYWORD_ELSE	$
prog	1																								
m_statements		2	11	14		3	13	12						10	9	8	7	6	5				4	15		
main_return																										
m_statement				28		27	26	25						23	22	21	20	19	17			18	16	29		
f_definition						30																			31	
while																							32		33	
assignment														34	35	36	37	38	39						40	
expression								41												42					43	
call_function							44																		45	
condition																								46	47	
f_params		48												49	50	51										
type_function													52	53	54	55										
data_type														56	57	58										
f_statements		59		60			61	62					63	64	65	66	67	68	69				70	71		
f_statement				72			73	74					75	76	77	78	79	80	81			82	83	84		
return_rest							85	86									90	89	88	87						
return		92									91															
f_rest_params		93																			94					
f_call_params		99															98	97	96	95						
after_equal							99										103	101	102	103						
prolog_end		104			105																					

Table 2: Grammatical table

C Precedent Analysis



	*	/	+	-	.	<	>	<=	>=	===	!==	(	)	int	float	string	var_id	NULL	$
*	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	∨	∧	∧	∧	∧	∧	∧	∨
/	∧	∧	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
+	∧	∧	∧	∧	∧	∨	∨	∨	∨	∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
-	∧	∧	∧	∧	∧	∨	∨	∨	∨	∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
.	∧	∧	∧	∧	∧	∨	∨	∨	∨	∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
<	∧	∧	∧	∧	∧			∧		∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
>	∧	∧	∧	∧	∧				∧	∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
<=	∧	∧	∧	∧	∧					∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
>=	∧	∧	∧	∧	∧		∧			∨	∨	∨	∧	∧	∧	∧	∧	∧	∨
===	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	∨	∧	∧	∧	∧	∧	∧	∨
!=	∧	∧	∧	∧	∧	∧	∧	∧	∧			∨	∧	∧	∧	∧	∧	∧	∨
==	∧	∧	∧	∧	∧	∧	∧	∧	∧			∨	∧	∧	∧	∧	∧	∧	∨
(	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
)	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	∧	=	∧	∧	∧	∧	∧	∧	∨
int	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
float	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
string	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
var_id	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
NULL	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨							∨
$	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	∨	

Table 3: Precedence table

D Lexical Finite Machine

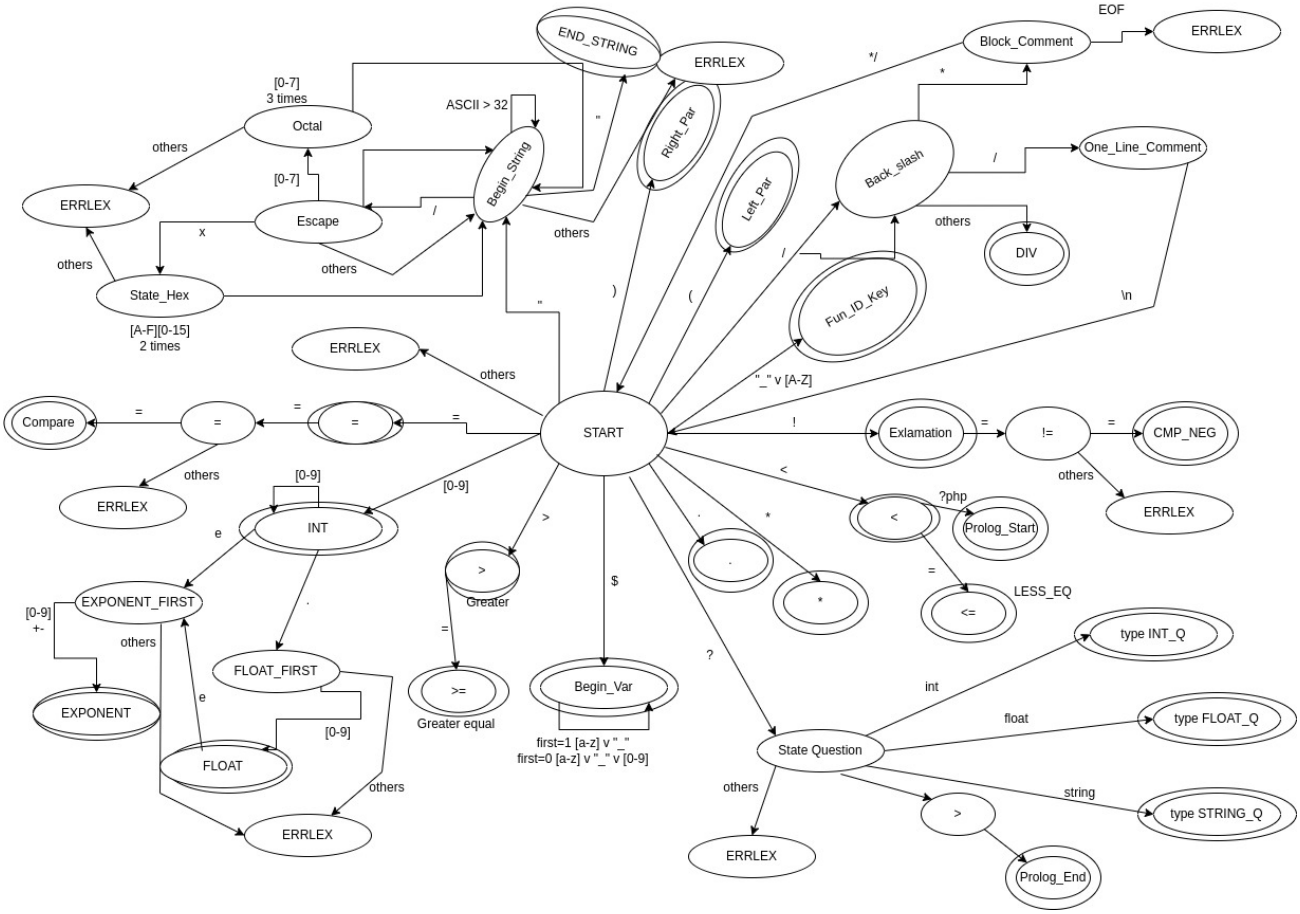


Figure 1: Diagram of Finite State Machine