Brno University of Technology
Faculty of Information Technology

Project documentation

Compiler of an imperative programming language, IFJ22
Team xklajb00, TRP variant

Authors:
David Klajbl (xklajb00) 25%
Martin Horvát (xhorva17) 25%
Marián Tarageľ (xtarag01) 25%
Ondřej Chromý (xchrom19) 25%

December 7, 2022

Contents

1 Introduction

In this work we will briefly go over the process, architecture and techniques used in our implementation of a compiler for an imperative programming language within the `Formal Languages and Compilers` course at `FIT BUT`. The source language `IFJ22`, is a subset of the `PHP` programming language with some minor changes. The compiler was implemented entirely in the `C` programming language. The target language, which can be interpreted, is called `IFJcode22`.

2 Teamwork & Process

We've started to work on the project early into the semester and kept working in a consistent fashion throughout. As such, we've managed to finish the entire project on time and there are no rushed or unfinished parts.

We've had weekly in person meetings consistently throughout the semester, during which we've designed and agreed upon the most important aspects of the architecture, data structures and interfaces used to communicate between the individual modules.

Team member name	Work done
David Klajbl	Token array, AST, Precedence parser, Symbol table, Semantic analyzer, Built-in functions, Code generator
Martin Horvát	Dynamic string, Scanner, Label generator, Expression generator
Marián Tarageľ	Top-down parser, LL-grammar, LL-table
Ondřej Chromý	Precedence parser

3 Architecture & Implementation

3.1 Lexical analysis

The lexical analyser (LA) was the first implemented module, due to it being necessary for most further work. The basic design of the LA was taken from *Elements of Compiler Design* [1]. The LA is called by the Top-down parser with the `get_token()` function. `get_token()` generates and returns a `token_t` struct according to the processed lexeme. The `token_t` struct is used to pass tokens throughout the compiler, it contains their types along with their associated values, if required by the token type.

The `get_token()` function reads a character from stdin and according to the character value, calls further lexeme handler functions which handle and initialise one or several lexemes and token types.

The handlers utilise `while` loops (based on the lexeme regular expression) to read lexemes where possible and don't use looping `switch` statements. Even though this seemed like a good idea, an implementation using looping `switch` statements with a more concrete way to control the state of the LA would be a better solution, and would lead to better code readability and easier extensibility.

Keyword, optionally null type identifiers and the opening mark lexemes are implemented using lookup tables, which are just arrays of strings, containing reserved words, namely the values of given lexemes. The given handler tries to match any of these keywords when possible and when it isn't able to, the lexeme is either of some other type or a lexical error is set in the global `error` variable.

For the DFA specified by the LA, see Appendix B. The lexical analyzer is implemented in the `scanner.c/.h` files.

3.2 Syntactic analysis

3.2.1 Top-down parsing

The top-down syntactic analyzer was implemented as a recursive descent parser with backtracking. Backtracking is a technique that tries to apply each possible rule and if no rule gets matched, a syntax

error is returned. Due to the fact that we have eliminated epsilon rules from the grammar, the parser does not need to send back tokens from nested recursive calls. Functions in a top-down analyzer represent non-terminal symbols.

Besides checking syntax of a file, the parser generates an abstract syntax tree, whose root will be returned to the compilers main body. AST nodes are created and added to AST by semantic actions linked to a rule application. Created AST nodes inherit a tokens associated value.

Top-down analyzer closely cooperates with precedence parser, which parses expressions. Expressions are loaded to token array in the top-down parser and sent to the bottom-up parser. Switch from top-down to bottom-up happens when rules 39., 40., 41. and 42. (non-terminal symbol $\langle exp \rangle$) are applied. After the parsing finishes, the parser will the expression, in post-fix notation, to the AST node.

The parser is based on a LL(1) grammar, see Appendix C. For the LL table of the given grammar, see Appendix D. Grammar was inspired by the book *Elements of Compiler Design* [1].

Parser implementation is in `parser.c/.h` files.

3.2.2 Bottom-up parsing

The bottom-up precedence parser is used to verify the syntax of expressions and to convert expressions to post-fix format. It accepts a token array representing an expression as input, and if the expression is syntactically correct, the token array of the expression is converted to post-fix notation and returned.

The key components that the precedence analyzer utilizes are the precedence stack used to store terminals and non-terminals of the grammar rules, precedence table (implemented in `precedence_table.c/.h`), which determines operations of precedence parser and precedence reduction rules table (implemented in files `precedence_rules.c/.h`), which determines reduction actions based on the grammar rule handle on top of the precedence stack.

Precedence parser processes the input token array, token by token. The topmost terminal of the precedence stack and the current token on input specify the indexes of the precedence table. Value of the precedence table on given indexes determines the next precedence parser operation to execute. This is repeated until all input tokens have been processed and there is only one non-terminal at the top of the precedence stack or until a syntactic error occurs.

The reduction operation is particularly interesting. At first, all terminals and non-terminals are popped from the top of the precedence stack until a start of a handle is found. Every terminal and non-terminal is represented by a precedence rule element structure and by a token array. Next, precedence rule elements get matched to specific grammar rules encoded in the reduction rules table, which subsequently determines the corresponding reduction action. Based on the reduction action, token arrays of popped handle are merged into a single token array in a specific way to represent a sub-expression in post-fix format. A merged token array is then pushed onto precedence stack as a non-terminal, and handle is removed from topmost precedence stack terminal. If the syntactic analysis is successful and all precedence stack handles have been reduced, the non-terminal at the top of the precedence stack will contain the entire expression converted to post-fix format.

The bottom-up precedence parser is implemented in files `precedence_parser.c/.h`.

3.3 Semantic analysis

The semantic analyzer traverses the AST, which was created during the syntactic analysis, and performs several semantic checks. These include function redefinition and function parameter redefinition detection, missing or unexpected expression in return statement detection. Detection of the use of a undefined function is solved by recursively traversing all bodies of used functions inside called function and searching for use of undefined functions. Other semantic checks, such as type checks, cannot be detected statically and are carried out during run-time.

The semantic analyzer uses two separate symbol tables - global and local symbol tables. The global symbol table is dedicated to functions and global variables and the local symbol table is dedicated to local variables. Only one symbol table is active at a time, depending on the context in which the analyzer is currently located - either the main program body or a function body. After exiting a function context, the local symbol table is cleared of all its symbols. The global symbol table is freed only after the entire analysis has been performed. A special data structure (`semantic_context_t`) is used to store information about the current context of the semantic analyzer, which is passed as part of a recursive traversal of the AST.

Besides semantic checks, another function of the semantic analyzer is to enrich the AST with information necessary to generate the compiled target code. Used variables in compiled program are extracted and placed at the beginning of the main program body or at the beginning of a function body in special AST node. This solves the problem of variable redefinition inside loops in generated compiled program. Furtehermore, an AST node representing an implicit return statement or erroneous missing return are added to the end of function bodies. Information about the expected return type is added to the return statement AST nodes. Modified and enriched AST is then passed to the code generator, which no longer needs to work with the symbol table, as all necessary information has been added to the AST.

The semantic analyzer is implemented in the `semantic_analyzer.c/.h` files.

3.4 Code generation

After an abstract syntax tree (AST) is verified and enriched by the semantic analyzer, it's passed to the code generator, implemented in `generator.c/.h` and `expgen.c/.h`. The generator traverses the AST and generates code in the target language `IFJcode22`.

The code generated by the generator is structured as follows: at first, auxiliary global variables are defined (used for type checking, expression evaluation etc.), followed by the generation of the main program body, after this, user functions (including embedded functions) are generated, and lastly, all auxiliary functions (operations) are generated.

3.4.1 Checking for uninitialized variables

Due to specifics of the `IFJ22` language, a situation where an uninitialized variable is used in an expression can occur, this has to be checked by the generator beforehand. This is done by extracting all variables used in an expression and checking whether they do have a type, (`TYPE` instruction in `IFJcode22`), before the expression is evaluated. If they have no type at the time when the given expression should be executed, the program is exited returning a run-time semantic error.

3.4.2 Label uniqueness

In order to ensure program wide unique label identifiers, we've developed naming conventions which ensure uniqueness and no collision with user defined variable or function names. Uniqueness of repeating code construct labels (`if-else` statement, `while` loop, type checking etc.), is ensured by using functions with a `static` variable counter incremented after each newly generated label. Implementation of these functions is in the `generator_tools.c/.h` files.

3.4.3 Expression evaluation

When the generator needs to generate an expression it calls the `gen_expression()` function which takes as arguments a `token_array_t` token array and the context of the generator.

This function assumes that the token array is a syntactically correct expression in post-fix notation. It then does a single pass through the array generating a `PUSH` instruction for each operand and a

`CALL` instruction for each operator, depending on the operators type. The functions called when an operator is encountered have been generated as auxiliary functions by the generator. This way, generated expressions have remained quite concise.

There is one function for each arithmetic, string and relational operator. The functions all work in a similar fashion, that is, generate a `POP` operation for both operands, check their type compatibility with each other and in respect towards the operation, handle any type conversions if necessary. Then generate an instruction corresponding to the type of operation `ADD`, `GT`, `CONCAT` etc.

The expression generator uses global variables `GF@_lhs`, `GF@_rhs`, `GF@_tlhs` and `GF@_trhs`, which are to be understood as the left hand side and right hand side of a given operation in addition to variables holding the type currently stored in both.

If any type error occurs, an operand is incompatible with an operator or an operand is incompatible with another operand, the compiled program would exit during interpretation returning a run-time semantic error.

3.5 Data structures

3.5.1 Abstract syntax tree

An abstract syntax tree (AST) is used to create an internal representation of the structure of a compiled program. Thanks to this, compilation process can take place in three separate stages - syntax analysis (includes lexical analysis), semantic analysis and code generation. Separating semantic analysis and code generation from program parsing is particularly advantageous, as the parser itself is already very busy, and excessive complexity would be introduced to the code.

AST is generated during syntax analysis by semantic actions associated with grammar rules. The semantic analyzer traverses the AST, checks program semantics and enriches AST with the information required by the code generator to generate compiled target code.

The AST is implemented as a generic n-ary tree. Every tree node (`AST_node_t` structure) is identified by its type (`AST_node_type_t` enumeration) and represents key constructs of the compiled program. A dynamic array is used to store variable number of links to children nodes, which allows for easy and efficient tree traversal. Data stored inside a AST node can be either none, a string value (represents literal values, variable and function names), a token array (represents postfix expression), a data type of `IFJ22` language, or none.

Supported AST operations include the creation of a new AST node, insertion or appending of a new AST node to the children array of an existing parent node, and the destructor of a created AST. Function, which prints AST in text format to a specified output stream, was implemented to help visualize and verify the proper behaviour of the syntax analyzer.

Implementation can be found in files `abstract_syntax_tree.c/.h`.

3.5.2 Precedence parser stack

The precedence parser stack is a modified stack data structure used by the precedence parser during bottom-up expression syntax analysis. It is implemented as a singly linked list of precedence stack elements.

The precedence stack allows us to either push terminal (single token) or non-terminal (token array) to the top of the precedence stack. To distinguish terminal and non-terminal precedence stack elements, a special flag was added to the precedence stack element structure.

The pop operation removes the top element from the precedence stack and returns its token array and a precedence rule element structure, where information about the popped element is stored.

To indicate where the grammar rule handle begins, every precedence stack element has a handle start indicator flag. The handle start of the topmost terminal can be set or removed by a dedicated operation.

Finally, a function that returns the precedence table index of the topmost terminal was implemented.

The precedence parser stack is implemented in files `precedenc_stack.c/.h`.

3.5.3 Symbol table

The symbol table is implemented as a hash table with a re-sizable bucket size, which uses the `sdbm` hashing function [2]. Collisions are resolved by chaining synonymous keys into linked lists. Hash table keys are either variable or function identifiers. Because variable identifiers always start with the '$' symbol, which is an invalid character for a function identifier, collisions of identical names of a variable and a function are avoided. Thus the same symbol table can store both variable and function symbols simultaneously.

Bucket size is doubled automatically when the average length of symbol lists exceeds `AVG_LEN_MAX` constant after insertion of a symbol to the symbol table. Resizing is done by rehashing all symbols inside existing bucket to new resized hash table bucket.

Each symbol in the symbol table is stored in pairs with its symbol information (`symbol_info_t` structure type). Symbol information is used only for function symbols. It contains the return data type of the function, reference to the AST node of function body and flag indicating whether all functions, called from inside this function, were defined and don't call other undefined functions.

Supported operations on the symbol table structure type `symtable_t` include symbol table constructor, destructor, symbol insert and lookup operations and symbol table clear operation, which frees all symbols from symbol table, leaving it empty.

The symbol table is implemented in the `symtable.c/.h` files.

3.5.4 Token array

The token array is a dynamic array used for storing and manipulating with multiple tokens. It is used to represent an expression.

It is similar to a stack ADT in terms of supported operations. These include constructor, destructor, push and pop token operations, operation to append tokens from one token array to another and operation to reverse token array. To help visualize expressions dedicated function that prints token array was implemented.

Token array is implemented in `token_array.c/.h` files.

3.5.5 Dynamic string

`dynamic_string.c/.h` was used throughout some parts of the compiler namely, the lexical analyser and the label generators. This is essentially our implementation of a proper string data type as is used in several higher-level programming languages and yet is absent from `C`.

It features a `dynamic_string_t` struct with a defined interface upon it, including several standard string operations. Initializing, initializing from a string, deinitializing, concatenating, writing a character, writing an unsigned integer etc. All the resizing, index operations, memory operations are abstracted away from the user, which leads to better code readability and less errors.

3.6 Error handling

In order to reduce code complexity, we've opted to implement a global `error` variable which is of type `error_codes_t`, an `enum` which holds all possible input program and internal compiler errors. This greatly reduces the number of arguments being passed to functions within the code. Value of `error` is checked by the controlling modules. These modules check the value after every call of a function that could raise an error. When an error is raised, the program is exited returning the value of `error`.

`error` is implemented in the `error.c/.h` files.

4 Conclusion

We are generally satisfied with the work done on this project. We believe our compiler functions decently well and passes most standard tests (this we know). Improvements could be done by rewriting the lexical analyzer with implementation of looping `switch` statements alongside an inner state, followed by implementing a target code optimizer as the generated code is slow when interpreted. General improvements for managing a software project would be to write more tests and test during development. We believe following a test-driven development methodology would've allowed us to discover bugs earlier on, leading to less code patching.

References

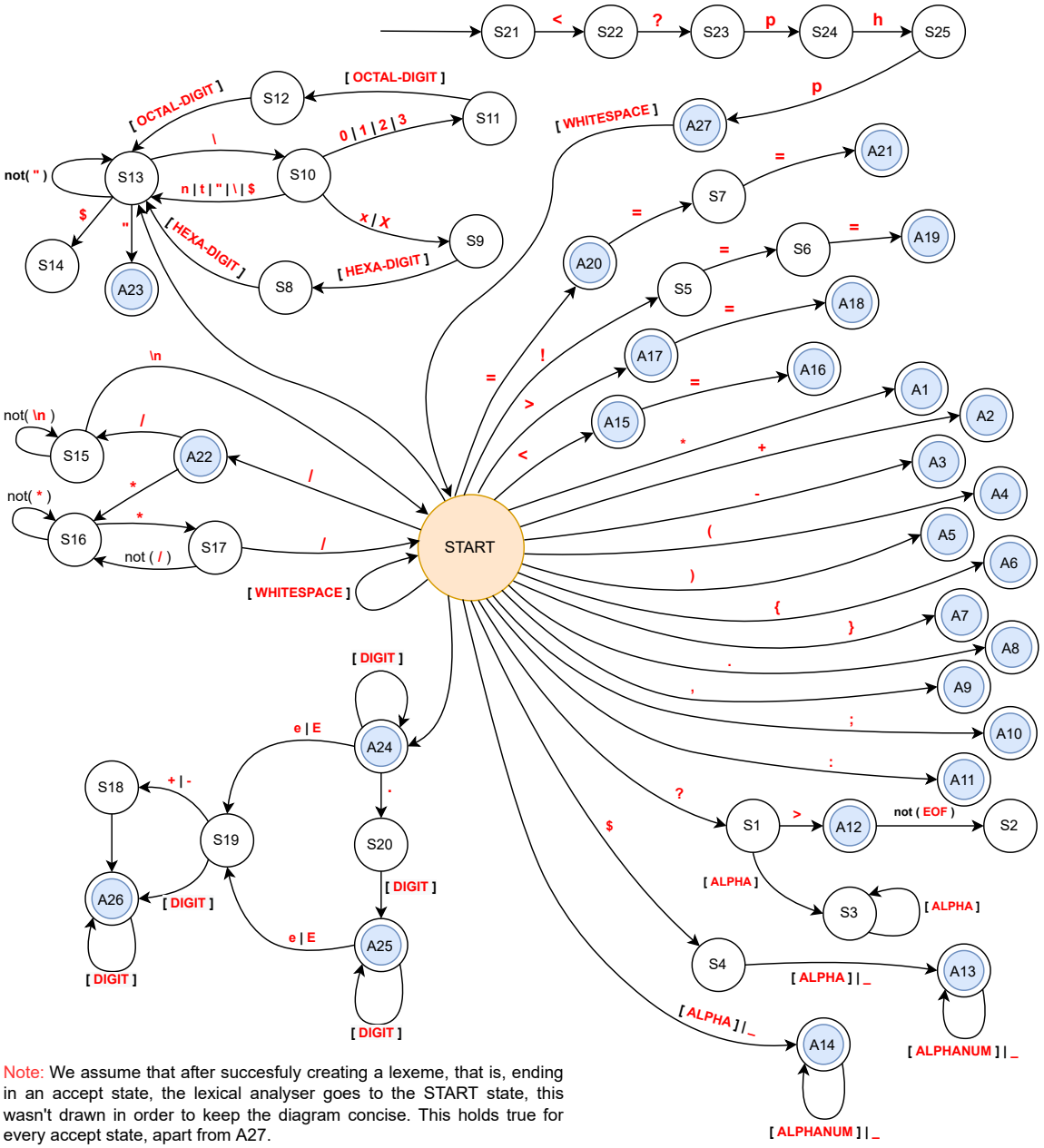
[1] Alexander Meduna. *Elements of compiler design*. Auerbach Publications, 2007.

[2] Ozan Yigit. A collection of non-cryptographic hash functions. `http://www.cse.yorku.ca/~oz/hash.html#sdbm`.

A Division of the implementation solution

File	Description
`main.c`	Main body of compiler program. Contains compiler entry point function `main`.
`token.c/.h`	Token ADT.
`token_array.c/.h`	Token array ADT.
`dynamic_string.c/.h`	Dynamic string ADT.
`scanner.c/.h`	Lexical analyzer.
`abstract_syntax_tree.c/.h`	Abstract syntax tree ADT.
`precedence_table.c/.h`	Precedence table and its interface.
`precedence_rules.c/.h`	Precedence analyzer reduction rule table and rule matching implementation.
`precedence_stack.c/.h`	Precedence parser stack ADT.
`precedence_parser.c/.h`	Bottom-up precedence parser implementation.
`parser.c/.h`	Top-down recursive descent parser implementation.
`symtable.c/.h`	Symbol table.
`semantic_analyzer.c/.h`	Semantic analyzer.
`generator_builtin.h`	Macro definitions containing IFJcode22 built-in function implementations.
`generator_tools.c/.h`	Helper tools used by code generator and expression generator to generate unique labels and print compiled source code to standard output.
`expgen.c/.h`	Expression generator.
`generator.c/.h`	Target source code generator.
`error.c/.h`	Error handling function implementation and global error variable definition.
`Makefile`	Automation of compiler program build process using Make.

B Diagram of DFA specified by the Lexical Analyzer



Note: We assume that after succesfuly creating a lexeme, that is, ending in an accept state, the lexical analyser goes to the START state, this wasn't drawn in order to keep the diagram concise. This holds true for every accept state, apart from A27.

Accept state -> lexeme:
A1 -> MUL
A2 -> ADD
A3 -> SUB
A4 -> LB
A5 -> RB
A6 -> LCB
A7 -> RCB
A8 -> CONCAT
A9 -> COMMA
A10 -> SCOLON
A11 -> COLON
A12 -> EPILOG
A13 -> VAR_ID
A14 -> FUNC_ID | KEYWORD
A15 -> LT
A16 -> LTE
A17 -> GT
A18 -> GTE
A19 -> NEQ
A20 -> ASSIGN
A21 -> EQ
A23 -> STR_LIT
A22 -> DIV
A24, A25, A26 -> FLT_LIT
A27 -> PROLOG

Edge specifiers:
ALPHANUM -> a-z, A-Z, 0-9
ALPHA -> a-z, A-Z
DIGIT -> 0-9
OCTAL-DIGIT -> 0-7
HEXA-DIGIT -> 0-9, a-f, A-F
WHITESPACE -> whitespace character
not(x) -> all characters apart from x

C LL-grammar

1. <program> -> <php_start> <program_body>
2. <php_start> -> <?php func_id (strict_types = int_lit) ;
3. <php_end> -> ?> EOF
4. <php_end> -> EOF
5. <program_body> -> <func_def> <program_body>
6. <program_body> -> <stmt> <program_body>
7. <program_body> -> <stmt_list_bracket_start> <program_body>
8. <program_body> -> <php_end>
9. <stmt_list_bracket_end> -> }
10. <stmt_list_bracket_end> -> <stmt> <stmt_list_bracket_end>
11. <stmt_list_bracket_end> -> <stmt_list_bracket_start> <stmt_list_bracket_end>
12. <stmt_list_bracket_start> -> { <stmt_list_bracket_end>
13. <func_def> -> function func_id (<param_list> : <return_type> {
<stmt_list_bracket_end>
14. <param_list> ->)
15. <param_list> -> non_void_type $var_id <param_next>
16. <param_next> ->)
17. <param_next> -> , non_void_type $var_id <param_next>
18. <return_type> -> void
19. <return_type> -> non_void_type
20. <stmt> -> if <if_stmt>
21. <stmt> -> while <while_stmt>
22. <stmt> -> return <exp> ;
23. <stmt> -> func_id <func_call> ;
24. <if_stmt> -> (<exp>) { <stmt_list_bracket_end> else { <stmt_list_bracket_end>
25. <while_stmt> -> (<exp>) { <stmt_list_bracket_end>
26. <func_call> -> (<arg_list>
27. <arg_list> ->)
28. <arg_list> -> <arg> <arg_next>
29. <arg_next> ->)
30. <arg_next> -> , <arg> <arg_next>
31. <arg> -> $var_id
32. <arg> -> lit
33. <stmt> -> <exp>
34. <stmt> -> var_id <exp/ass_exp/ass_func> ;
35. <exp/ass_exp/ass_func> -> = <ass_exp/ass_func>
36. <exp/ass_exp/ass_func> -> <exp>
37. <ass_exp/ass_func> -> func_id <func_call>
38. <ass_exp/ass_func> -> <exp>
39. <exp> -> (<exp_rest>
40. <exp> ->) <exp_rest>
41. <exp> -> operator <exp_rest>
42. <exp> -> lit <exp_rest>

lit = str_lit | int_lit | flt_lit | null
operator = *, /, +, -, ., <, >, <=, >=, ===, !==
non_void_type = int, ?int, float, ?float, string, ?string
others = "strict_types" | ":" | "else" | ";"

D LL-table

	<?php	?>	EOF	{	}	if	while	return	func_id	(	)	var_id	lit	operator	non_void_type	void	function	,	=	others
<program>	1																			
<php_start>	2																			
<php_end>		3	4																	
<program_body>		8	8	7		6	6	6	6	6	6	6	6	6			5			
<stmt_list_bracket_start>				12																
<stmt_list_bracket_end>					9	10	10	10	10	10	10	10	10	10						
<func_def>																	13			
<param_list>											14				15					
<param_next>											16							17		
<return_type>															19	18				
<stmt>						20	21	22	23	33	33	34	33	33						
<if_stmt>										24										
<while_stmt>										25										
<func_call>										26										
<arg_list>											27	28	28							
<arg_next>											29							30		
<arg>												31	32							
<exp/ass_exp/ass_func>									35	36	36		36	36					35	
<ass_exp/ass_func>									37	38	38		38	38						
<exp>										39	40		42	41						

Notation: others = strict_types, :, else, ;

E Precedence table

	= , - , .	* , /	=== , !==	<,>,<=,>=	(	)	i	$
= , - , .	R	S	R	R	S	R	S	R
* , /	R	R	R	R	S	R	S	R
=== , !==	S	S	R	S	S	R	S	R
<,>,<=,>=	S	S	R	R	S	R	S	R
(	S	S	S	S	S	E	S	X
)	R	R	R	R	X	R	X	R
i	R	R	R	R	X	R	X	R
$	S	S	S	S	S	X	S	X

Notation: S <=> '<' R <=> '>' E <=> '=' X <=> ' '

Header column contains topmost terminals of precedence stack.

Header row contains terminals on input.

Terminals with same precedence table values are sparated by commas in table headers.