Lexical analyzer

xpojez00, xbudin05, xhribi00, xchupa03

October 31, 2021

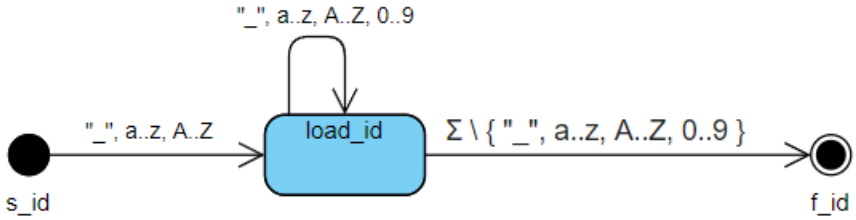
1 Introduction

Our approach was creating state machine (also referred as SM) for each category separately according to the project brief (in Czech). As seen in images beneath, we were not considering hexadecimal literals as valid values for integer. Our State Machines are affected by our approach of easier and cleaner implementations instead of more mathematical one. Nonetheless they should be considered equally correct. We have decided on distinguishing keywords in lexical analyzer and tokenizing them to off-load syntactic analyzer. Lexical analyzer should also find corresponding structure of the identifier in the symbol table[1] and send it as an attribute of the token[2].

2 SMs for each case separately

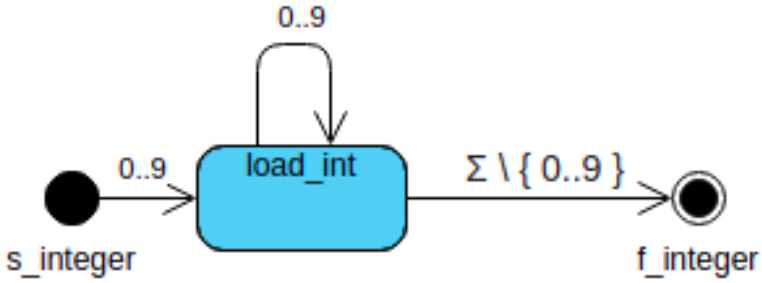
2.1 Identifiers or keywords

As stated above, we have decided for more programmatic approach and we consider identifiers and keywords the same type of lexeme. Distinguishing keywords is discussed later.



2.2 Integer

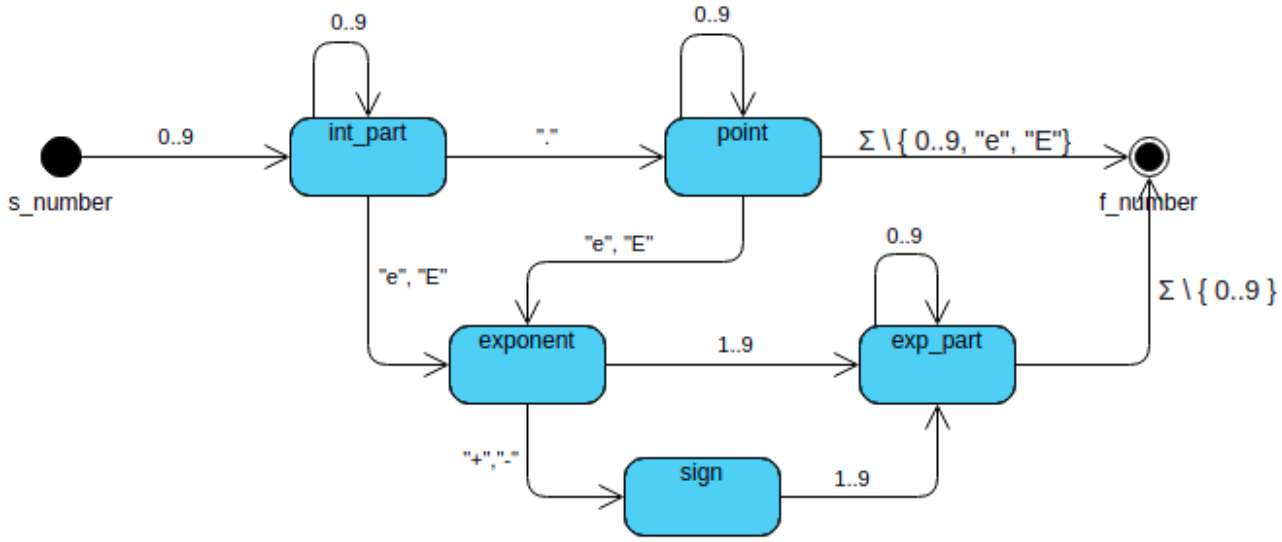
In our current state we do not consider hexadecimal literals to be possibly assigned to integers. It is not clear to us if it is or is not possible.



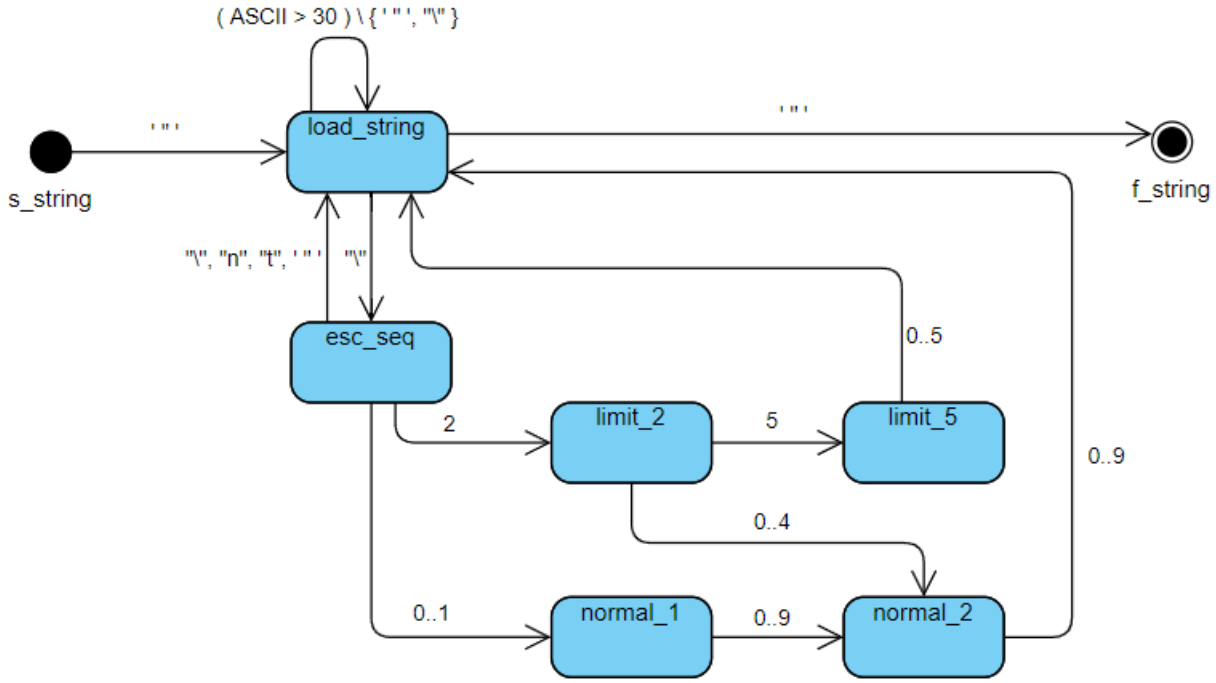
[1] Symbol table is implemented with hash table where every element is double linked list. Double linked list was selected for being able to add and remove elements in constant time while needing only the pointer to the current element for this operation

[2] Token is a struct containing enum value **type** to determinate what type it is and then **attribute**, which might be set depending on the value of **type**

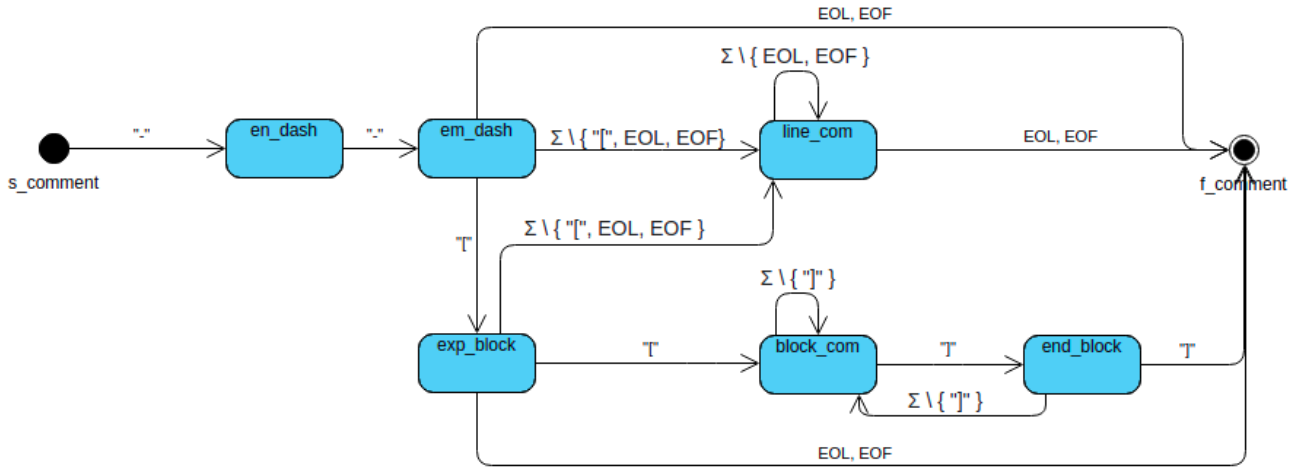
2.3 Floating point number



2.4 String

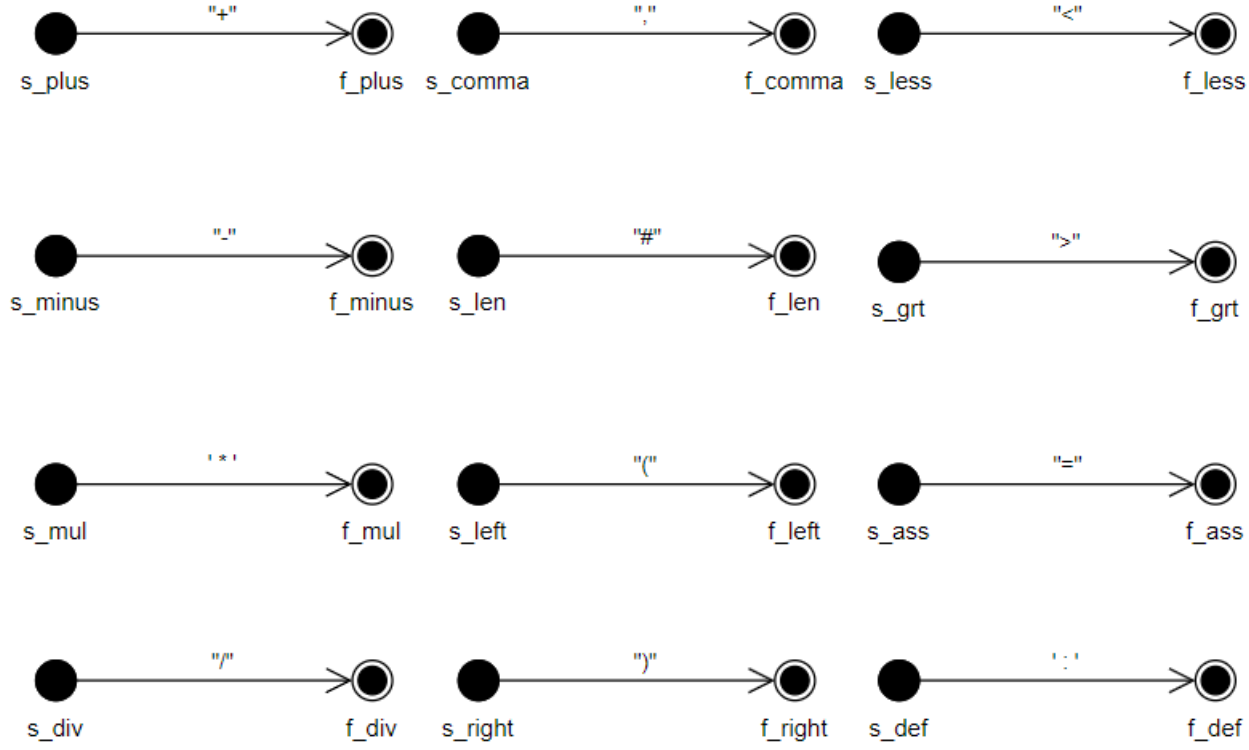


2.5 Line and block comment



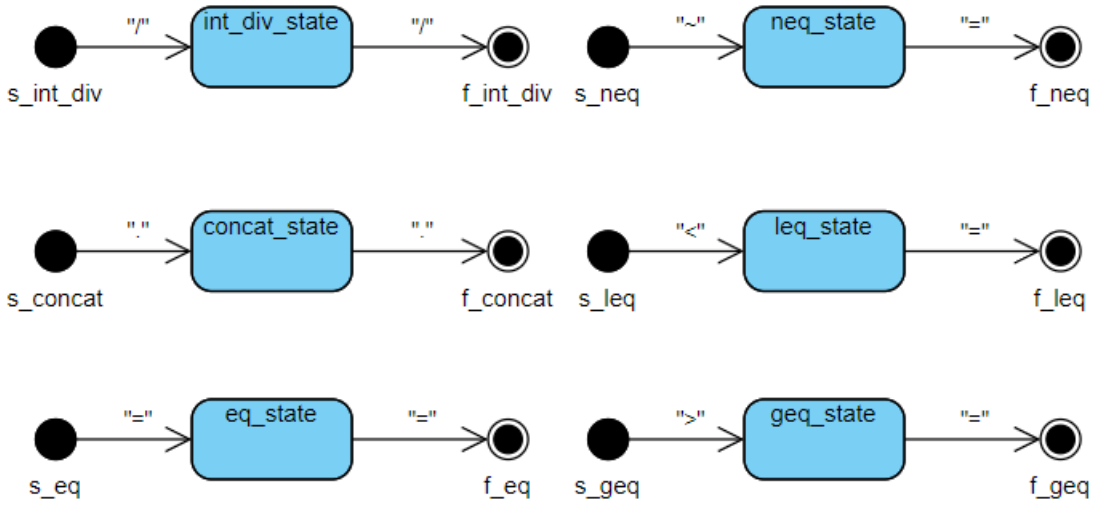
2.6 SMs with one transition

All SMs for one character lexemes are shown beneath



2.7 SMs with one state

All SMs with one state not considering starting and finishing state.



3 Creating single finite state machine

We have created one starting state and made transition to every starting state of separate SMs via ϵ. We have not made a single finishing state as we did not consider it important for our case.

3.1 Deleting ϵ transitions

We deleted every starting state to which we were connected via ϵ and preserved the character leading to another non-ϵ transition. We were left with SM without ϵ transitions.

3.2 Removing non-determinism

If we found a unique transition we preserved it with it's state unchanged. If there were any overlapping transitions we merged overlapping characters into an transition leading to new state, which was named after conjunction of states. For every disjunctive character we would create new transition to new state called after the original state with description it was derived from the original. All overlapping transitions were deleted with their corresponding states and their transitions were attached to the new state. To derived states were attached all leaving transitions. We continued this process until we checked every state in the SM.

3.3 Removing non-starting states

We would remove all non-starting or non-finishing states as non-starting do not affect the Finite state machine (also referred as FSM).

3.4 Removing non-finishing states and creating finite state machine

We have deleted all non-finishing states as their existence only makes this non finite and we created new finishing state to which we lead transition from each state, which could had gotten stuck. This was achieved by transition over Σ without characters leading to other states from the current state, to maintain determinism.

The final version is saved in **Lexical_analyzer_FSM.pdf**, you should open in it two-page mode.

3.5 Implementing details

We have implemented it with case conditions for state with variable **state** holding the enum value of the current state. Inside case we have if conditions for deciding the next state. We transition through the FSM as seen in the document. We store all read characters into another variable **lexeme**. After the FSM is iterated we know the final state and the lexeme.

4 Distinguishing keywords

Because of small amount of reserved keywords we have decided to create an array with all keywords. If the current identifier from FSM is in this array we know it is a keyword. This can be implemented with simple iteration through this array and comparing the lexeme with the values in the array.

5 Identifier

We find the element corresponding to the hashed lexeme, which we get from the Symbol table with a possible combination of arguments received from syntactic analyzer.

6 Final State Machine

$M = (Q, \Sigma, R, s, F)$
$Q = \{Start, load\_id, load\_int\_int\_part, point, exponent, sign, exp\_part, load\_string, esc\_seq, limit\_2, limit\_5, normal\_1, normal\_2, string\_finalize, plus\_finalize, mul\_finalize, div\_state\_int\_div\_state, int\_div\_finalize, def\_finalize, comma\_finalize, len\_finalize, left\_finalize, right\_finalize, less\_state\_leq\_state, leq\_finalize, grt\_state\_geq\_state, geq\_finalize, concat\_state, concat\_finalize, ass\_state\_eq\_state, eq\_finalize, neq\_state, neq\_finalize, minus\_en\_dash, em\_dash, line\_com, exp\_block, block\_com, end\_block, f\_id, f\_integer, f\_number, f\_string, f\_plus, f\_mul, f\_div, f\_int\_div, f\_def, f\_comma, f\_len, f\_left, f\_right, f\_less, f\_leq, f\_grt, f\_geq, f\_concat, f\_ass, f\_eq, f\_neq, f\_minus, error\_state, EOF\}$
$\Sigma = ASCII$
$R = see\ Lexical\_analyzer\_FSM.pdf$
$F = \{f\_id, f\_integer, f\_number, f\_string, f\_plus, f\_mul, f\_div, f\_int\_div, f\_def, f\_comma, f\_len, f\_left, f\_right, f\_less, f\_leq, f\_grt, f\_geq, f\_concat, f\_ass, f\_eq, f\_neq, f\_minus, error\_state, EOF\}$

LL grammar

xpojez00, xbudin05

November 7, 2021

1 Introduction

Our approach of creating LL table was to start from the easier **and** smaller parts **and** work our way to more complex non-terminals. At first, we filled the terminal set with all valid tokens. Our non-terminal **and** rule sets were empty. Starting from the variable declaration, value assignments **and** conditions. We are planning on implementing LL grammar using Predictive parsing.

2 Terminal set

$T = \{id, integer, number, string, "-", "+", "*", "/", "//", " : ", ",", ","\#", "(", ")", " < ", " <= ", " > ", " >= ", "..", " = ", " == ", " ~= ", \textbf{and}, \textbf{boolean}, \textbf{do}, \textbf{else}, \textbf{elseif}, \textbf{end}, \textbf{false}, \textbf{function}, \textbf{global}, \textbf{if}, \textbf{integer}, \textbf{local}, \textbf{nil}, \textbf{not}, \textbf{number}, \textbf{or}, \textbf{require}, \textbf{return}, \textbf{string}, \textbf{then}, \textbf{true}, \textbf{while}\}$

3 Non-Terminal set

$NT = \{< program >, < global\_scope >, < global\_statements >, < global\_statement >< function\_declare >, < function\_define >, < function\_call >, < parameters >, < parameter >, < parameter\_name >, < parameters\_defined >, < parameter\_defined >, < returning >, < scope >, < called\_parameters >, < scope\_statements >, < statements >, < statement >, < declare >, < id >, < if >, < while >, < scope\_return >, < return >, < declare\_assign >, < assign >, < condition >, < condition\_branch >, < lvalues >, < lvalue >, < rvalues >, < rvalue >, < expression >, < expression\_2 >, < expression\_3 >, < datatypes >, < datatype >, < unary\_operator >, < binary\_operator >\}$

4 Rule set

$< program >$	$\rightarrow$ **require**.$string$.$< global\_scope >$
$< global\_scope >$	$\rightarrow < global\_statements >$
$< global\_scope >$	$\rightarrow \epsilon$
$< global\_statements >$	$\rightarrow < global\_statements >$. $< global\_statement >$
$< global\_statements >$	$\rightarrow < global\_statement >$
$< global\_statement >$	$\rightarrow < function\_declare >$
$< global\_statement >$	$\rightarrow < function\_define >$
$< global\_statement >$	$\rightarrow < function\_call >$
$< function\_declare >$	$\rightarrow$ **global**.id." : ".**function**."(". $< parameters >$.")". $< returning >$
$< function\_define >$	$\rightarrow$ **function**.id."(". $< parameters\_defined >$.")". $< returning >$. $< scope >$ **.end**
$< function\_call >$	$\rightarrow$ id."(". $< called\_parameters >$.")"
$< parameters >$	$\rightarrow < parameters >$.",". $< parameter >$

$< parameters > \rightarrow < parameter >$
$< parameters > \rightarrow \epsilon$
$< parameter > \rightarrow < parameter\_name > . < datatype >$
$< parameter\_name > \rightarrow id." : "$
$< parameter\_name > \rightarrow \epsilon$
$< called\_parameters > \rightarrow < rvalues >$
$< called\_parameters > \rightarrow \epsilon$

$< parameters\_defined > \rightarrow < parameters\_defined > ."," . < parameter\_defined >$
$< parameters\_defined > \rightarrow < parameter\_defined >$
$< parameters\_defined > \rightarrow \epsilon$
$< parameter\_defined > \rightarrow id." : " . < datatype >$

$< returning > \rightarrow " : " . < datatypes >$
$< returning > \rightarrow \epsilon$

$< scope > \rightarrow < scope\_statements > . < scope\_return >$
$< scope\_statements > \rightarrow < statements >$
$< scope\_statements > \rightarrow \epsilon$
$< statements > \rightarrow < statements > . < statement >$
$< statements > \rightarrow < statement >$

$< statement > \rightarrow < declare >$
$< statement > \rightarrow < id >$
$< statement > \rightarrow < if >$
$< statement > \rightarrow < while >$

$< declare > \rightarrow$ **local**. $< lvalues > ." : " . < datatypes > . < declare\_assign >$
$< id > \rightarrow id."(" . < called\_parameters > .")"$
$< id > \rightarrow id. < assign >$
$< id > \rightarrow id."," . < lvalues > . < assign >$
$< if > \rightarrow$ **if**. $< condition >$.**end**
$< while > \rightarrow$ **while**. $< expression >$.**do**. $< scope >$.**end**
$< scope\_return > \rightarrow < return >$
$< scope\_return > \rightarrow \epsilon$

$< declare\_assign > \rightarrow < assign >$
$< declare\_assign > \rightarrow \epsilon$
$< assign > \rightarrow " = " . < rvalues >$
$< condition > \rightarrow < expression >$.**then**. $< scope > . < condition\_branch >$
$< return > \rightarrow$ **return**. $< rvalues >$
$< return > \rightarrow$ **return**

$< condition\_branch > \rightarrow$ **else**. $< scope >$
$< condition\_branch > \rightarrow$ **elseif**. $< condition >$
$< condition\_branch > \rightarrow \epsilon$

$< lvalues > \rightarrow < lvalues > ."," . < lvalue >$
$< lvalues > \rightarrow < lvalue >$
$< lvalue > \rightarrow id$
$< rvalues > \rightarrow < rvalues > ."," . < rvalue >$
$< rvalues > \rightarrow < rvalue >$
$< rvalue > \rightarrow < expression >$

$< expression > \rightarrow < expression > . < binary\_operator > . < expression\_2 >$
$< expression > \rightarrow < expression\_2 >$

$< expression\_2 >$ $\rightarrow < unary\_operator > . < expression\_3 >$
$< expression\_2 >$ $\rightarrow < expression\_3 >$
$< expression\_3 >$ $\rightarrow$ "(". $< expression >$.")"

$< expression\_3 >$ $\rightarrow$ *string*
$< expression\_3 >$ $\rightarrow$ *number*
$< expression\_3 >$ $\rightarrow$ *integer*
$< expression\_3 >$ $\rightarrow$ *id*
$< expression\_3 >$ $\rightarrow$ *id*."(". $< called\_parameters >$.")"
$< expression\_3 >$ $\rightarrow$ **true**
$< expression\_3 >$ $\rightarrow$ **false**
$< expression\_3 >$ $\rightarrow$ **nil**

$< datatypes >$ $\rightarrow < datatypes >$.",". $< datatype >$
$< datatypes >$ $\rightarrow < datatype >$

$< datatype >$ $\rightarrow$ **integer**
$< datatype >$ $\rightarrow$ **number**
$< datatype >$ $\rightarrow$ **string**
$< datatype >$ $\rightarrow$ **boolean**

$< unary\_operator >$ $\rightarrow$ "#"
$< unary\_operator >$ $\rightarrow$ **not**

$< binary\_operator >$ $\rightarrow$ " − "
$< binary\_operator >$ $\rightarrow$ " + "
$< binary\_operator >$ $\rightarrow$ " ∗ "
$< binary\_operator >$ $\rightarrow$ "/"
$< binary\_operator >$ $\rightarrow$ "//"
$< binary\_operator >$ $\rightarrow$ ".."
$< binary\_operator >$ $\rightarrow$ " < "
$< binary\_operator >$ $\rightarrow$ " <= "
$< binary\_operator >$ $\rightarrow$ " > "
$< binary\_operator >$ $\rightarrow$ " >= "
$< binary\_operator >$ $\rightarrow$ " == "
$< binary\_operator >$ $\rightarrow$ " = "
$< binary\_operator >$ $\rightarrow$ **and**
$< binary\_operator >$ $\rightarrow$ **or**
$<>$ $\rightarrow$

Precedence parsing table

xpojez00, xbudin05

December 3, 2021

1 Introduction

Operator precedence parser is used as a bottom-to-top parser in our Syntax analyzer for expressions only. We have opted for the implementation via table. Expressions are considered all possibilities, which are accepted by LL-grammar rules of $< expression >$[1]. We have introduced a new symbol (See Precedence parsing table). This symbol is used to notify the parser about function, which needs to be recursively parsed. We also need to allow this recursive call to write in the sequence of rules applied for later code generation. As mentioned above, we might use already used $< expression >$ rules to guarantee equality between the parsing of $< expression >$ between top-to-bottom and bottom-to-top parsers. As we have allowed the use of boolean variables and boolean expression our table might appear larger than expected.

2 Introduction to the problem

One problem with multiple returned values occurs. With allowance of function calls with expressions as arguments few problems emerge. These arguments are separated by commas. Small problem occurs with parsing commas, where we want them to be parsed only upon function calls. Another problem comes from the fact, that function can be called with zero arguments. To preserve determinism we can not allow use of ϵ. Problem with function having arguments as function calls makes the need for recursive parsing more apparent.

3 Our current solution

Function's arguments have to create in some way a tree[2], which will be represented with a root and as only non-terminal E[3], which is then added to the function call as a single argument. This E has to be able to consist of further arguments, where each argument is an expression as well. This is solved by allowance of rule as follows:

$$E \rightarrow E, E$$

To preserve simplicity we have opted to create one more rule, which will create an so-called *void* E, which is an expression. This E is then by default appended as first argument of every function call. As we can see the rule still allows for function calls with single or multiple arguments but also solves the problem of potential ϵ. Every new argument's tree is then appended with the afford-mentioned rule to create a single E tree containing arguments (See also Temporary forked sequence).

Only when parsing through a function call we are allowed to consume commas, otherwise this is not allowed, to preserve determinism with top-to-bottom parser and not consume it's commas. This is embedded in the Precedence parsing table

If there is an function call within current function call we can recursively call expression parsing upon this function call and append the whole E as a single E in our currently parsed argument.

[1] See Syntax_analyzer.pdf
[2] We are not creating tree just a sequence of rules. Referencing to tree just for easier explanation and picturing
[3] Semantic action upon this subtree is currently being discussed. Some ideas are already thought of

Note: Problem with multiple returned values is currently being discussed. Current idea of the early solution is to use the rule $E \rightarrow E, E$ to redeploy return values from the function. The first value is used always. The remaining values are used only in case, that the function is called as the last one and we use the amount of values needed to fill all arguments of the function call. Otherwise only the first value is used and the tree is cut. Please note this problem is not a problem in Precedence parsing table but rather of Semantic analyzer. Problem is mentioned here as Semantic analyzer is closely bounded with Precedence parsing table.

3.1 Temporary forked sequence

When a function's argument is another function and this function returns more than one number, we need to deterministically decide, whether to use only first, some or all returned values. This can be achieved with our solution of so called Temporary forked sequence. This means, that function is returning E, which can be then recursively parsed into all returned values as follows: $E \rightarrow E, E$. We split the initial E holding the final return values into to subtrees as E , E. We know, that the former E holds the first returned value and the later E contains subtree of all remaining return values. Now, when expecting new argument for the function call we need to check for remaining arguments. If any are present we abandoned this former E subtree and append there the new argument. Otherwise we use as many splits of the former E as the declaration of the return values allows. If there are any excessive we do not use them. In the picture below it is clear, why it is called a Temporary forked sequence. Please note, that when the function returns values into an expression only the first return value is used and others are ignored. This can be achieved by allowing Temporary forked sequence only in function calls. This solution will be implemented in Semantic analyzer rather than Syntax analyzer.

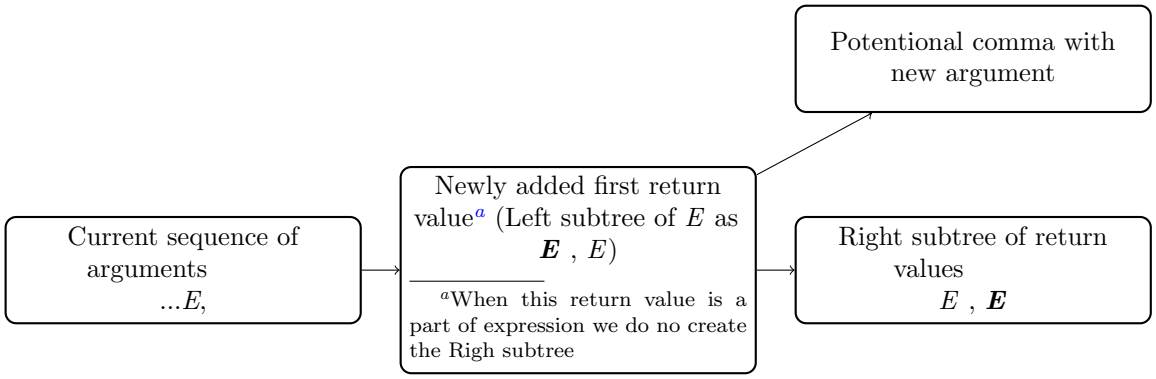


Figure 1: Function call as an argument of function call

4 Terminals

$\Sigma = \{"(","),"\#", \textbf{not}," * "," / "," // "," + "," - "," .. "," > "," < "," >= "," <= "," == "," \sim= ", \textbf{and}, \textbf{or}, i, \$, ","\}$[4]
Token $\in \Sigma$

$T = \Sigma \cup \{"\lceil"\}$
Top $\in T$

i is considered as set of terminals with equal precedence, where we will less likely make changes unlike to for example relation operators.
$i = \{string, number, integer, id, true, false, nil\}$

Furthermore we would like to point the fact that comma(",") is allowed to be parsed only under afford-mentioned conditions. **Void** is terminal which is not contained in the set of possible tokens (Literal evaluation of *id* is not considered as new token).

[4]Later referenced without quotes

5 Rules

For simplicity reasons we create additional two sets.

$binary\_operators = \{"*","/","//","+","-","..",">","<",">=","<=","==","\sim=", \textbf{and}, \textbf{or}\}$

$unary\_operators = \{"\#", \textbf{not}\}$

$E \to i$
$E \to bE : b \in unary\_operators$
$E \to EbE : b \in binary\_operators$
$E \to (E)$

$E \to \textbf{void}$
$E \to i[E)$
$E \to E, E$

6 Precedence parsing table

$<$ set top terminal as it can be parsed with attribute $<$ and push Token on stack
$>$ we consume stack until we find rule, where last terminal has attribute $<$
$=$ just push Token to Top and read next Token
α push [on Top, create so called *void E*, push it to the Top and read next Token
β $<$ to Top, push , on Top, call of Precedence parsing table expecting E as returned value & push this E on Top
γ set top terminal's attribute to $<$ but do not push Token on stack
X not a valid combination. Might be considered empty as well

Top \ Token	(	)	#	**not**	*	/	//	+	-	..	>	<	>=	<=	==	~=	**and**	**or**	i	,	$
(	<	=	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X	X
)	X	>	X	X	>	>	>	>	>	>	>	>	>	>	>	>	>	>	X	X	>
#	<	>	X	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	X	>
not	<	>	X	X	X	X	X	X	X	X	>	>	>	>	>	>	>	>	<	X	>
*	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	X	>
/	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	X	>
//	<	>	<	X	>	>	>	>	>	X	>	>	>	>	>	>	>	>	<	X	>
+	<	>	<	X	<	<	<	>	>	X	>	>	>	>	>	>	>	>	<	X	>
-	<	>	<	X	<	<	<	>	>	X	>	>	>	>	>	>	>	>	<	X	>
..	<	>	X	X	X	X	X	X	X	>	>	>	>	>	>	>	>	>	<	X	>
>	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
<	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
>=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
<=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
==	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
~=	<	>	<	<	<	<	<	<	<	<	>	>	>	>	>	>	>	>	<	X	>
and	<	>	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	>	<	X	>
or	<	>	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	>	<	X	>
i	α	>	X	X	>	>	>	>	>	>	>	>	>	>	>	>	>	>	X	X	>
,	X	>	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	>	X	
[	β	=	β	β	β	β	β	β	β	β	β	β	β	β	β	β	β	β	β	γ	X
$	<	X	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	<	X	X

Note:	β appears in positions, where it does not make sense as expression can not begin with for example an binary operator but recursive Precedence parsing should parse this as not an expression but consume it until right parenthesis or comma

7 Implementation details

As Precedence parsing table should parse only an expression, it should not consume token, which does not belong to the expression. This means, we should check, if we would be able to apply valid rule in the future or not. This way we may say, that the next token either does not belong to us or in case of stack not looking like

$$\$E$$

there has been an error between the current token and previously parsed expressions. We would have found this error after we would have found invalid combination or parsed whole expression as left side of each rule is always E.

In β we have to return read token and read after the recursive Precedence parsing call ends.

Semantic Analyzer

xpojez00, xbudin05

December 3, 2021

1 Introduction

Whenever a rule is applied we need to check if semantic attributes match accordingly. Semantic analyzer should be called only upon expressions and operations involving them, i.e. assignment, return values, function calls etc. With creation of semantic action upon a rule applied we are closely tied to generating the final code, which can be done from Abstract Syntax Tree. Tree is generated from expressions currently.

2 Structure

```
NodeType = {OPERATION, ID, FUNCTION, VALUE, VOID, NIL, POP, IF, ELSEIF, ELSE,
    NODE_WHILE, FUNCTION_DEFINITION, FUNCTION_POP, DECLARE, ASSIGN, LVALUES,
    RVALUES, RETURN}

SemanticType = {INTEGER, NUMBER, BOOLEAN, STRING, VOID, NIL}

struct Node {
    NodeType nodeType
    void *data
    vector *sons
    vector *returns
    SemanticType semanticType
    OperationType operation
    int attribute
}
```

Note: attribute can be used by Nodes, which need to remember for example a unique ID assigned to them or in case of function calls[1]

3 Abstract Syntax Tree

We have decided to create a *m-n-tree*. This tree has m sons as incoming parameters and in case of function we have also n sons as function returns. Whenever we want to call Node finalized i.e. ready to be generated into IFJcode21, we must first finalize all m sons before. In case of function we should generate n sons before finalization and before we check the semantic correctness. Meaning if Semantic types our parent expects are the same with our return values. This was special case for functions. In all other operators we simply just check if all our sons are of the same SemanticType and then the operation itself gains a SemanticType for itself. Most of the time it is logical sum i.e.

$$node.semanticType = \sum_{k=1}^{m} node.sons[k].semanticType$$

[1] Presume function call A has as the last argument function call B, we might need more than one return value from function call B as parameters of the function call A

In case of node being function, SemanticType defaultly equals to VOID unless it returns at least one value. Then the first returned value is the SemanticType of function.

Note: Semantic type in casual statements has no meaning this is why we include VOID as semantic type.

4 Creating boolean expression out of non-boolean expression

If we end with an expression of non-boolean type we create new expression, which is of OPERATION type and operator equals to $==$. We append as sons original expression and *nil node*. This is default behaviour defined by the Teal language.

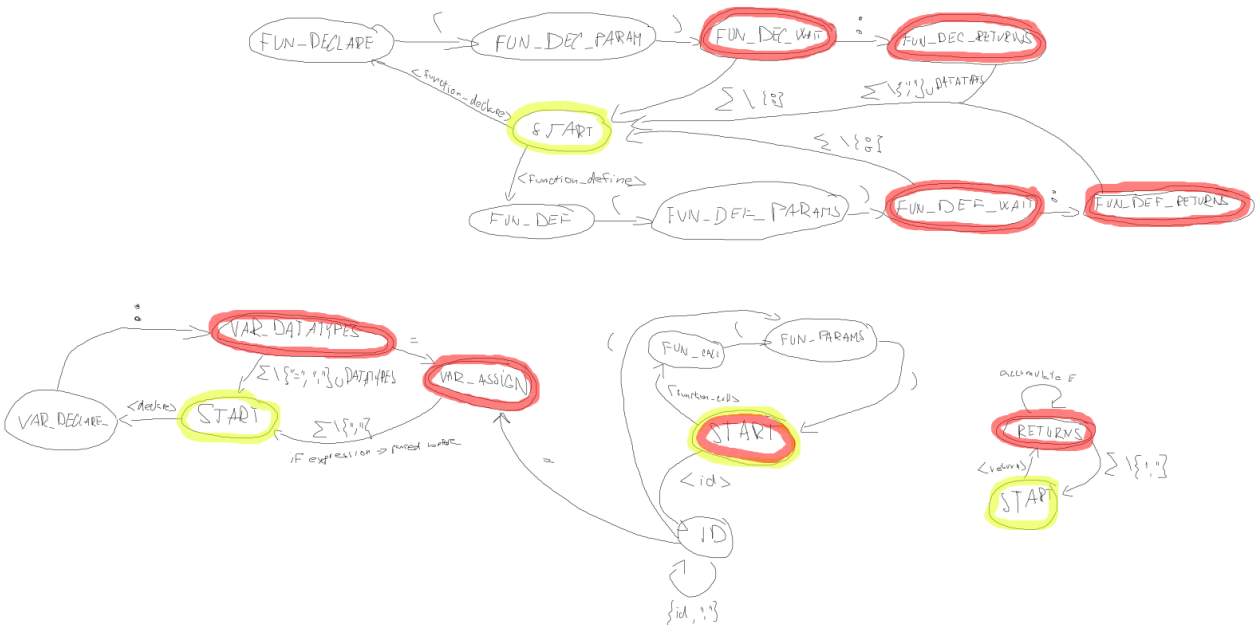
5 Possible Operators

Set of operators	Set of possible semantic values
$\{\#\}$	$\{STRING\}$
$\{\textbf{not}\}$	$\{BOOLEAN\}$
$\{*, /, +, -\}$	$\{INTEGER, NUMBER\}$
$\{//\}$	$\{INTEGER\}$
$\{..\}$	$\{STRING\}$
$\{>, <, >=, <=\}$	$\{INTEGER, NUMBER\}$
$\{==, \sim=\}$	$\{INTEGER, NUMBER, BOOLEAN, STRING\}$
$\{\textbf{and}, \textbf{or}\}$	$\{BOOLEAN\}$

6 Connection with LL-grammar

With Semantic actions in Top-to-Bottom parsing we also need to apply some Semantic rules as for example assignment of correct Semantic expression into variable. We have created new FSM, which will rely on detecting syntactic rules by LL-grammar. This FSM is therefore non-deterministic but non-terminals from FSM will notify us about special occasion as when we should evaluate our current FSM states. This in practice means whenever we go onto another statement we want to evaluate our FSM from its current state and values saved inside. After each new statement and follow up evaluation we go to default starting state of this FSM. We then set inside values to default and continue according to the FSM rules. We can see from START state we decide upon LL-grammar non-terminals and last state also usually gets us back to this starting state.

Note: After we evaluate statement, i.e. we move to START, we decide, whether the statement is semantically correct and if yes we then append this Node to Abstract Syntax Tree.



7 Code Generation

To generate code from AST we recursively descend from the root to leafs in post order. This way we have generated our children before we can use them. We can generate some parts of code before, between and after recursive descend to generate valid code. As header of file we generate *.IFJcode2021* and a JUMP $*function_calls*. As footer of the file we generate LABEL $*function_calls* and then all function calls in the source code. Between the JUMP and LABEL we generate all built-in-functions and defined functions from the source code.

7.1 DEFVAR in loops

To avoid generating this instruction inside loops we firstly generate all DEFVAR instructions before generating the statement Node with all instruction. Then we descend the Node second time but this time we generate all instructions except DEFVAR instructions.

Symbol Table

xbudin05, xpojez00

October 15, 2021

1 Introduction

We are going to discuss our idea of implementing the symbol table for project (in Czech). We have decided to go for implementation using hash map rather than binary tree. Binary tree would solve problem of hash map, where multiple identifiers could possibly receive the same hash but for the cost of extensive access time. We are also facing problem with multiple scopes in the code, where the deepest scope variable is covering all above. To save identifiers we have a lower part of the structure, which is Hash table and below and the upper part to structure the scope we are working in and all identifiers declared in it. Picture for clearance beneath.

1.1 Conventional solution

Usually, it is implemented as a stack of hash maps, where each element in stack is new hash map, to solve the problem of scope covering. One element of hash map might then be a list to solve problem of overlapping hashes. This solution is pretty easy to understand but we have decided to improve this solution and save memory by creating always only one hash map and having a few more other structures. Also as a benefit we believe, we have also saved time in searching the deepest declaration of a certain variable, where it should be O(1)[1] no matter the stack dept. In the conventional solution this time corresponds to O(n), where n is the depth of stack.

2 Implementation of hash table

2.1 Our initial solution

We have decided to address both problems, multiple hashes of different identifiers colliding and having multiple declarations of variable, with one ordered double linked list. Idea was to create hash map, where every element is ordered double linked list, where the same identifiers are ordered according to their respective scopes. This idea can be implemented in a more clear way but the idea is preserved.

2.2 Our current solution

We have opted for creating list, where each element is holding the name of the identifier and pointer to the stack[2] for this specific identifier. This list is not ordered. Here is also clear why we have decided to implement symbol table with hash map, where accessing mostly only lists with size one is instant and does not consume much memory. This list solves our first problem with multiple identifiers having the same hash value. For the second problem of finding the deepest variable, we have chosen stack, as it is one of the best structures considering we only need push, pop and top for accessing variable in the top scope, which is covering all above.

2.3 Element

Approximate definition of our Symbol element, where this structure points to string, which is saved only in the hash map's list element. This way we can avoid duplicating name for later free-ing of this

[1] We do not consider overlapping hashes
[2] Explained later why this abstract structure

string. Saving it only once and pointing to this string also helps with time saved upon not recreating and free-ing this string. Type lets us decide, what type of Element we have and how we should read data. More variables might be added in the final implementation.

```
struct Element {
    Type type;
    char* name;
    bool isDefined;
    void* data;
}
```

3 Implementation of Symbol table

3.1 Solving Scopes

To implement scopes themselves, we have decided to choose a stack for it's simplicity. Whenever we need to clear scope we just clear all content of the top element. All identifiers in the scope are saved in list, where each Element is a pointer to the Element saved in the Hash table. This way, when clearing the list, we can navigate from the Hash table down to the Element itself, as there is deterministically[3] only one Element matching.

3.2 Buffer

We have also implemented a buffer, which saves every scope, that we might want to preserve, but we want it to be removed from the hash map and stack, holding current state of scopes. We remove[4] Element from the hash map, pop stack holding current state of scopes and save this poped list[5] into the buffer.

4 Summary

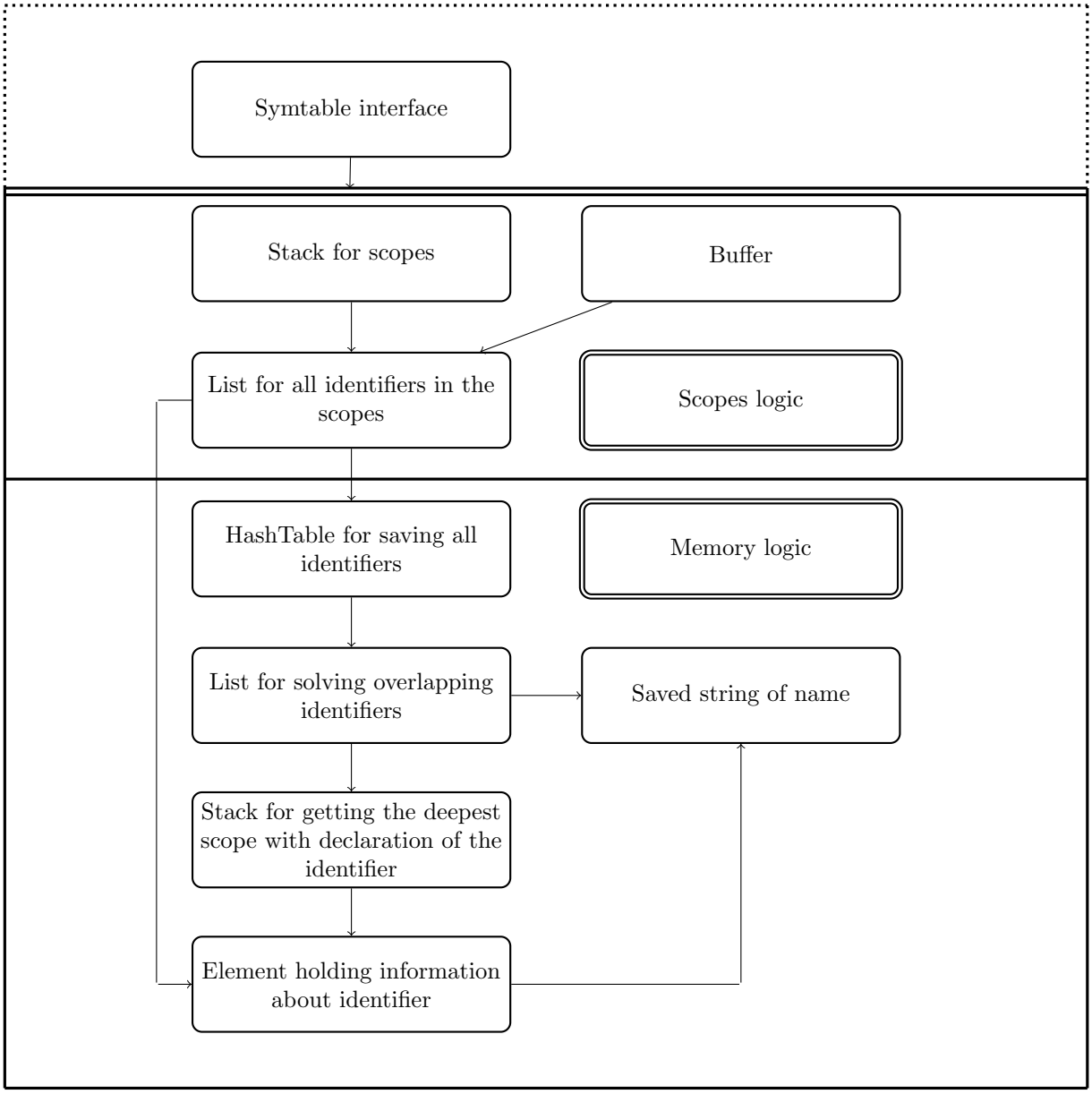
It is pretty apparent we have chosen a unique approach of harder implementation but for vast majority of cases instantaneous access and much better memory performance[6]. We have tried to explain idea as transparent and easy to understand as each problem is solved by separate structure layer.

[3] Every element can be destroyed only when clearing this list

[4] only the pointer

[5] This is the same list from stack, so Element can only be destroyed when clearing this list

[6] practical performance, in theory, if we would have number of identifiers equal to the size of hash map, with only single scope, we would use double the memory compared to the Conventional solution


Symtable interface
Stack for scopes
Buffer
List for all identifiers in the scopes
Scopes logic
HashTable for saving all identifiers
Memory logic
List for solving overlapping identifiers
Saved string of name
Stack for getting the deepest scope with declaration of the identifier
Element holding information about identifier