BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

IFJ Project 2024

IFJ Projekt 2024

Marek Sucharda <xsucha18> (team-leader)
Veronika Svobodová <xsvobov00>
Martin Mendl <x247581>
Vanesa Zimmermannová <x253171>

Variant: **vv-BVS**

Project contributions

Martin Mendl:

- Contributed to **semantic analysis**, focusing on the design and implementation of the **symbol table** and associated functionality.
- Developed the **height-balanced binary search tree** used as the core data structure for managing symbols within `symtable.c`.
- Implemented utility functions to support semantic operations and symbol table management.
- Worked on **precedence parsing**, implementing core logic in `precedent.c`.
- Provided significant contributions to **syntactic analysis**, particularly in `parser.c`.
- Developed key components of semantic analysis in `sem_analyzer.c`, ensuring type checking and proper scope resolution.
- Participated actively in **testing**, debugging, and refining the codebase to ensure correctness and robustness.

Vanesa Zimmermannová

- Designed AST, defined structures and implemented its construction in parser and implemented resource release functions for each structure - node of AST
- Contributed to parser by some minor refactor and bug fixes - connected to AST construction
- Contributed to semantic analysis by code review, and some minor changes, benefiting codegen. (mostly needed because of some AST changes)
- Implemented code generation, and builtin functions

Veronika Svobodová

- Contributed to syntactic analysis, including the syntax of the whole program, down to the precedence analysis of individual expressions in precident.c.
- Created syntactic rules, written down in LL-grammar, and the precedence table.
- Provided significant contributions to testing the overall project and semantic analysis.

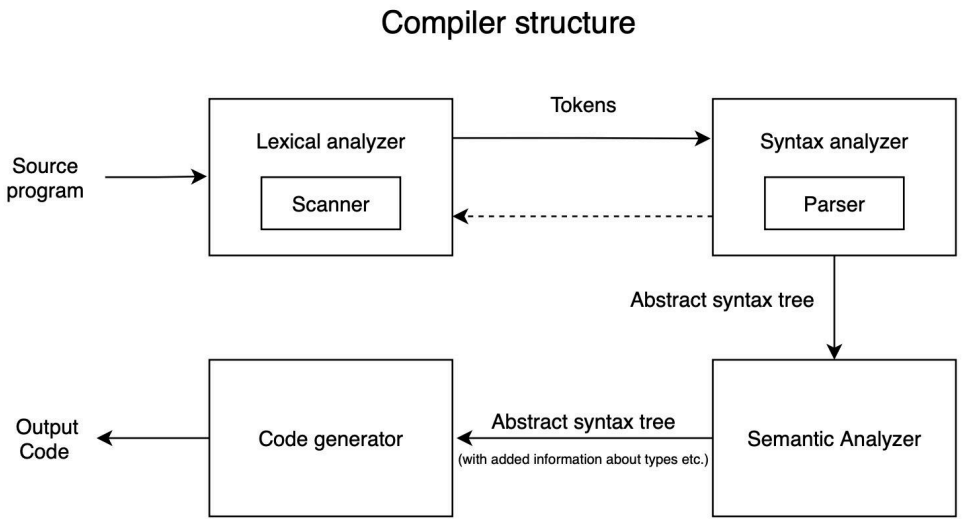
Marek Sucharda

- Implemented a lexical analyzer in scanner.c.
- Defined a token structure, an enum for token types, and states for the automaton.
- Designed a finite state machine for lexical analysis.
- Took care of functions in our syntax analysis in parser.c. Functions were implemented based on our LL(1) table.

Work Distribution:

Each team member contributed equally to the project. While some team members focused more on specific tasks, such as programming, others concentrated on understanding and researching what needed to be done. For example, certain team members dedicated time to studying complex concepts, such as precedence analysis, and shared their insights with the team. This collaboration ensured that tasks requiring in-depth understanding could be effectively implemented, as those who focused on coding relied heavily on the research and explanations provided by others. As a result, we have decided to divide the points equally among all team members to reflect our collective effort.

Project overview



Our compiler implementation differs slightly from the approach discussed in the lectures. We have designed each component of the compiler to function independently, allowing for modular development and testing. These components were then integrated in the `main.c` file.

We have clearly separated the syntactical and semantical analysis stages. First, the syntactical analysis constructs an Abstract Syntax Tree (AST), which serves as the input for the semantic analysis. The semantic analysis not only validates the program semantically but also enriches the AST with additional information, such as variable types, identifiers, and other metadata.

Finally, the enhanced AST is passed to the code generator, equipped with all the necessary information to generate the target code effectively.

The development of our compiler was approached from the bottom up, starting with the creation of robust, reusable components. Fundamental data structures, such as a generic linked list and a height-balanced binary search tree, were designed, implemented, and thoroughly tested as independent units. These components served as the foundation for building more complex modules, such as the symbol table (SymTable), abstract syntax tree (AST), parser, and lexer. This incremental approach allowed us to ensure the reliability and flexibility of each building block before integrating it into the larger system.

Although not part of the assignment requirements, we have also emphasized creating a developer-friendly environment for collaboration and testing. Our GitHub repository showcases a well-organized project structure, dividing the files into `src`, `include`, and `test` directories. Within these, subfolders such as `lexical`, `syntaxical`, `semantical`, and `code_generation` maintain logical separation of concerns. We utilized a `Makefile` to streamline compilation, along with shell scripts for running tests. Tests are enhanced with ANSI-colored output, and a debug compilation mode leverages debug macros for efficient troubleshooting.

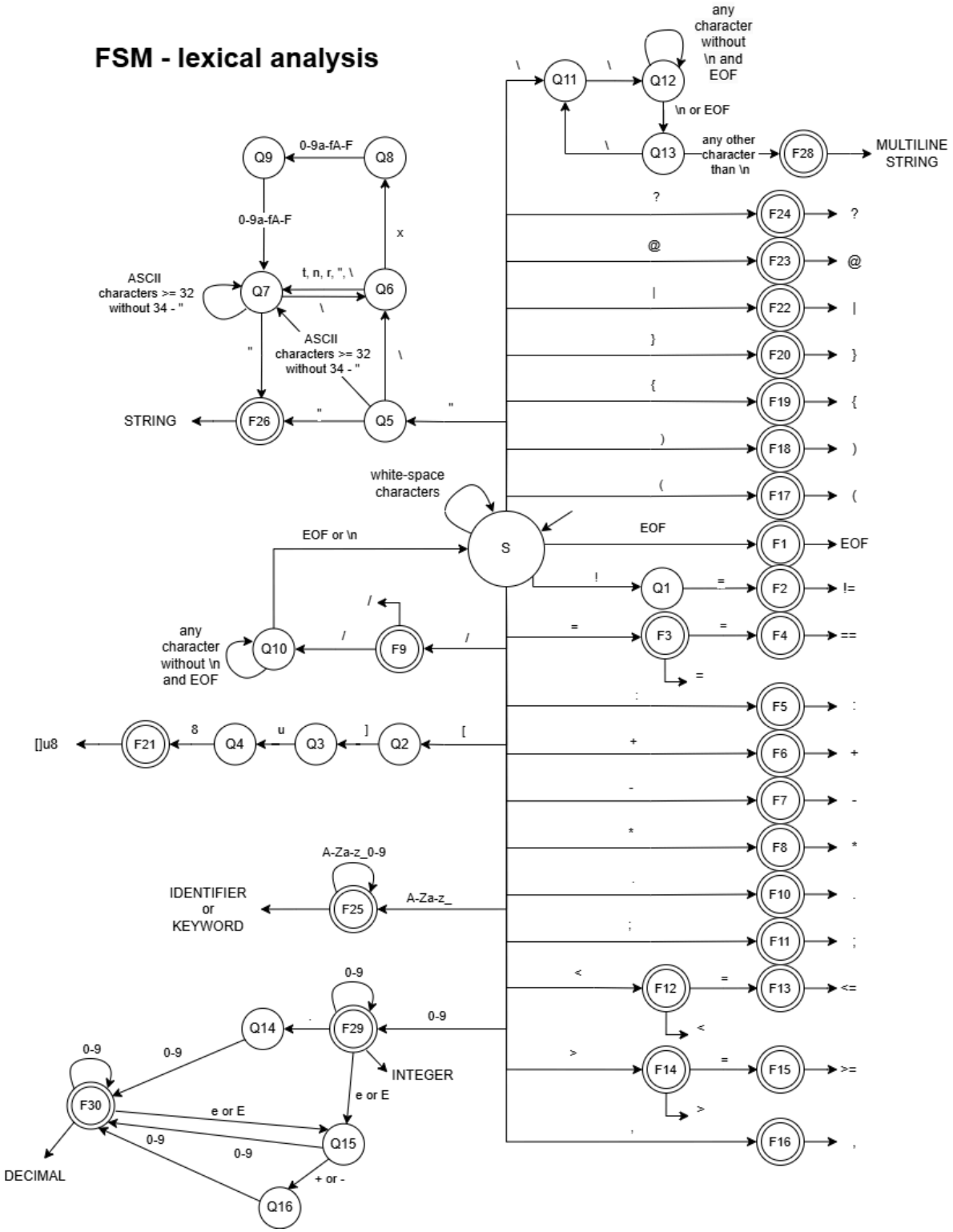
By going beyond the assignment specifications, we hope to provide not just a working compiler but also a well-documented, modular, and accessible codebase that others can learn from and build upon.

Feel free to check out our Github Repository.

Lexical Analysis

(Marek Sucharda, **files:** scanner.c)

Letter S denotes the state, which is marked as SCANNER_START in our code, the whole automat depends on this state. Q denotes state and F in the double circle denotes final state. Next to the final state there is an arrow that describes what token we finally got. Above the arrows is the character or characters that will let us into the next state.



Syntactic Analysis

(Marek Sucharda, Veronika Svobodová, Martin Mendl, **files:** parser.c, precedent.c)

LL-grammar:

1. <program> -> <prolog> EOL <functions>

2. <prolog> -> const ifj = @ import ("ifj24.zig");
3. <functions> -> <function> EOL <next_function>
4. <next_function> -> ε
5. <next_function> -> <function>

6. <function> -> pub fn FUNC_ID (<params>) <data_type> { <body> }

7. <params> -> <parameter> <parameter_next>
8. <params> -> ε
9. <parameter> -> VAR_ID : <data_type>
10. <parameter> -> ε
11. <parameter_next> -> , <parameter> <parameter_next>
12. <parameter_next> -> ε

13. <var_def> -> const VAR_ID : <data_type> = <value>;
14. <var_def> -> var VAR_ID : <data_type> = <value>;
15. <var_assign> -> VAR_ID = <value>;
16. <var_assign> -> _ = <func_call>

17. <value> -> <no_truth_expr>
18. <value> -> <func_call>

19. <func_call> -> <native_func_call>
20. <func_call> -> <user_func_call>
21. <native_func_call> -> ifj.NATIVE_FUNC_NAME (<arguments>);
22. <user_func_call> -> FUNC_ID (<arguments>);

23. <arguments> -> <argument> <argument_next>
24. <arguments> -> ε
25. <argument_next> -> , <argument> <argument_next>
26. <argument_next> -> ε
27. <argument> -> VAR_ID
28. <argument> -> <const>

29. <if> -> if (TRUTH_EXPR) { <body> } <else>
30. <if> -> if (<no_truth_expr>) | VAR_ID | { <body> } <else>
31. <else> -> else { <body> }
32. <else> -> ε

33. <while> -> while (TRUTH_EXPR) { <body> }
34. <while> -> while (<no_truth_expr>) | VAR_ID | { <body> }

35. <body> -> <body_content> <body_content_next>
36. <body_content_next> -> <body_content>
37. <body_content_next> -> ε
38. <body_content> -> <func_call>
39. <body_content> -> <var_def>
40. <body_content> -> <var_assign>
41. <body_content> -> <if>
42. <body_content> -> <while>
43. <body_content> -> RETURN <ret_value> ;
44. <ret_value> -> <no_truth_expr>
45. <ret_value> -> ε

46. <no_truth_expr> -> ARITHM_EXPR
47. <no_truth_expr> -> NULL_EXPR

48. <data_type> -> ? <type>
49. <data_type> -> <type>
50. <type> -> DATA_TYPE

51. <const> -> STRING
52. <const> -> NUMBER

LL(1) table:

	const	var	ifj	pub	func_id	var_id	?	data_type	,	_	string	number	arithm_expr	null_expr	if	while	else	return	$
<program>	1																		
<prolog>	2																		
<functions>				3															
<function>				6															
<next_function>				5															4
<params>						7													7
<parameter>						9													10
<parameter_next>									11										12
<data_type>							48	49											
<value>			18		18								17	17					
<no_truth_expr>													46	47					
<func_call>			19		20														
<native_func_call>			21																
<user_func_call>					22														
<arguments>						23					23	23							24
<argument>						27					28	28							
<argument_next>									25										26
<const>											51	52							
<body>	35	35	35		35	35				35					35	35		35	
<if>															29				
<else>																	31		32
<while>																33			
<body_content>	39	39	38		38	40				40					41	42		43	
<body_content_next>	36	36	36		36	36				36					36	36		36	37
<var_def>	13	14																	
<var_assign>						15				16									
<ret_value>													44	44					45
<type>								50											

The implementation of the LL(1) grammar resides in `parser.c`, which serves as the core component for syntactic analysis. During the parsing process, an Abstract Syntax Tree (AST) is progressively constructed, capturing the hierarchical structure and relationships within the input program. This AST acts as the foundational output of the syntactic analysis phase, enabling subsequent stages of compilation, such as semantic analysis and code generation.

Parsing begins after all tokens have been extracted from the standard input and stored in the symbol table (function: `firstPass` in: `parser.c`). These tokens are organized in a linked list, with a global index used to track the current position in the input program. This structure allows for efficient navigation through the tokens, including the ability to look ahead by one or two tokens. This lookahead capability enables the parser to branch appropriately based on the upcoming tokens, ensuring accurate handling of complex grammar rules and expressions. Additionally, during parsing, the `startPrecedentAnalysis` function can be invoked to process expressions, seamlessly integrating the results into the ongoing AST construction.

Precedence table:

	+, -	*, /	(	)	i	<, >, <=, >=, ==, !=	$
+, -	>	<	<	>	<	>	>
***, /**	>	>	<	>	<	>	>
(	<	<	<	=	<	<	
)	>	>		>		>	>
i	>	>		>		>	>
<, >, <=, >=, ==, !=	<	<	<	>	<	>	>
$	<	<	<		<	<	

The precedence analysis is implemented in `precedent.c`. This module handles the evaluation of operator precedence to ensure correct parsing of expressions. Below, you can see debug output showcasing the analysis process in action. During precedence analysis, a binary tree is constructed to represent the hierarchical structure of expressions. This binary tree is then integrated into the Abstract Syntax Tree (AST), contributing to the overall syntactic representation of the input program.

```
Stack: [
 0: $
 1: <
 2: E
 3: Token -> DIVIDE
 4: <
 5: Token -> IDENTIFIER
]
Rule checking result:
- stack item: Token -> IDENTIFIER
- input item: Token -> NONE
- rule: >
DEBUG: FUNC:startPrecedentAnalysis  msg -> Doing the redux
DEBUG: FUNC:_applyRulesExpresion  msg -> E -> id
DEBUG: FUNC:startPrecedentAnalysis  msg -> Redux done
Stack: [
 0: $
 1: <
 2: E
 3: Token -> DIVIDE
 4: E
]
DEBUG: FUNC:startPrecedentAnalysis  msg -> Cycle start

Stack: [
 0: $
 1: <
 2: E
 3: Token -> DIVIDE
 4: E
]
Rule checking result:
- stack item: Token -> DIVIDE
- input item: Token -> NONE
- rule: >
DEBUG: FUNC:startPrecedentAnalysis  msg -> Doing the redux
DEBUG: FUNC:_applyRulesExpresion  msg -> E -> E operand E
DEBUG: FUNC:startPrecedentAnalysis  msg -> Redux done
Stack: [
 0: $
 1: E
]
DEBUG: FUNC:startPrecedentAnalysis  msg -> Cycle start

Stack: [
 0: $
 1: E
]
Rule checking result:
- stack item: Token -> NONE
- input item: Token -> NONE
- rule: END
DEBUG: FUNC:match  msg -> Matched token:        ;
```

Symbol Table

(Martin Mendl, **files**: symtable.c, binary_search_tree.c, linked_list.c)

Implemented as an abstract data type (ADT). designed to model the hierarchical structure of code scopes. The core structure, `SymTable`, is organized as follows:

```c
// Symbol table ADT
typedef struct SymTable {
    SymTableNode *root; // root of the tree
    unsigned int varCount; // amount of variables (for making unique ids)
    unsigned int scopeCount; // amount of scopes in the tree
    SymTableNode *currentScope; // pointer to the current scope
    BST *functionDefinitions; // pointer to the function definitions BST
    LinkedList *data; // for storing variables
    LinkedList *tokenBuffer; // for storing the tokens
} SymTable;
```

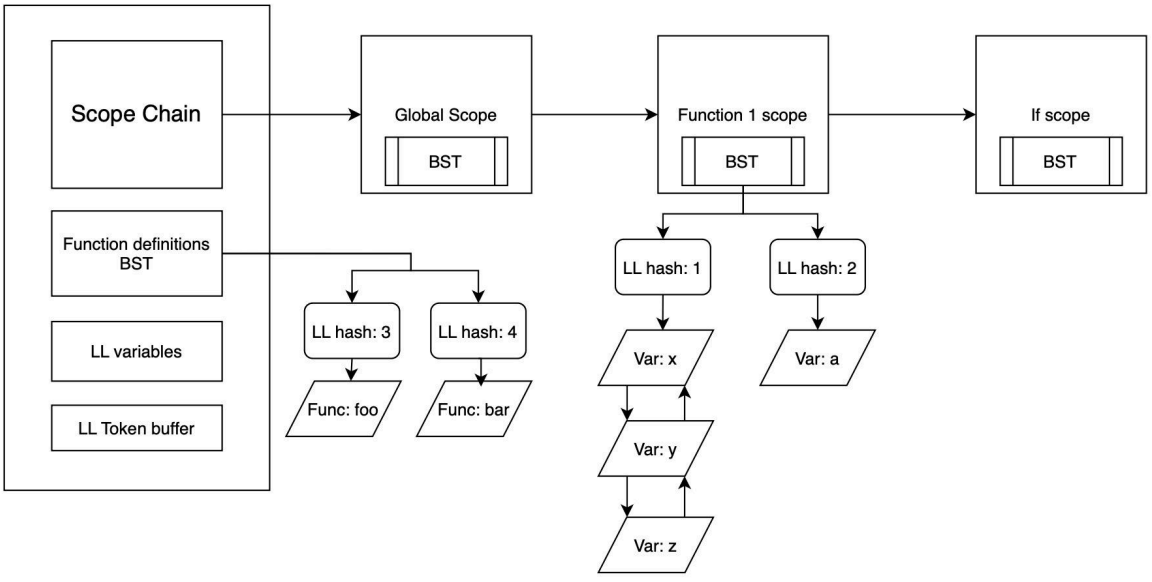
Each scope in the code is represented by a `SymTableNode`, forming a scope chain. Scopes include types such as `Function`, `If`, and `While`. Each scope maintains a height-balanced binary search tree (BST) for variable management. Variables are represented as follows:

```c
// SymTable Variable
typedef struct SymVariable {
    unsigned int id; // id of the variable (id is valid, inside of the scope)
    char *name; // the name of the variable
    enum DATA_TYPES type; // the type of the variable
    bool mutable; // if the variable is mutable (constants will have this false)
    int nullable; // Indicates if the variable can hold a null value -1 unknown, 0 not nullable, 1 nullable
    bool accesed; // if the variable was accessed
    bool modified; // if the variable was modified
    bool valueKnonwAtCompileTime; // if the value of the variable is known at compile time and can be converted to i32
} SymVariable;
```

To handle hash conflicts in the BSTs, each item in the tree is a linked list storing variables or functions with the same hash key. Functions are managed similarly, using a dedicated BST for function definitions.

Below is a snapshot of the symbol table's state during semantic analysis, representing a single moment in time. The symbol table is meticulously designed to model the flow of the program and facilitate automated semantic checks. For instance, when exiting a scope, it automatically verifies whether all variables within that scope have been used and modified.

Symbol Table structure



The symbol table also incorporates a linked list to store all `SymVariable` pointers, ensuring efficient cleanup and preventing memory leaks, particularly in cases where the input program fails semantic validation. Additionally, it includes a token buffer, implemented as another linked list, which serves to store all tokens generated by `scanner.c`.

Primary Functions

1. **`symTableExitScope`**: Handles scope removal and semantic checks during exit from the current scope.
2. **`symTableMoveScopeDown`**: Navigates into nested scopes.
3. **`symTableDeclareVariable`**: Declares a new variable after ensuring no conflicts in the current scope chain.
4. **`symTableFindVariable`**: Searches for a variable across all active scopes.
5. **`symTableAddFunction`**: Adds a new function definition, ensuring uniqueness.
6. **`symTableFindFunction`**: Locates a function definition in the function BST.

Semantic Analysis

(Martin Mendl, Vanesa Zimmermannová, **files:** sem_analyzer.c)

Semantic analysis follows parsing and ensures program correctness regarding scope, types, and variable usage. During this phase, the AST is traversed while interacting with the symbol table to enforce semantic rules.

Functions are added to the symbol table with details such as return type, nullability, and parameter types. Entering control structures like `if` or `while` creates new scopes in the symbol table, while exiting these structures removes the corresponding scopes.

Variable definitions are added to the current scope, ensuring uniqueness. The sem_analyzer.c then uses all the available information to validate each statement in the AST.

Expression trees are validated by ensuring the operands at each node are type-compatible with the operation and adding conversion flags (intToFloat, floatToInt) to each variable and literal in the expression if needed.

With these steps, the sem_analyzer.c ensures the program adheres to defined semantics of IFJ24.

Abstract Syntax Tree

(Vanesa Zimmermannová, **files**: `ast.c`, `expression_ast.c`)

Structure definitions for abstract syntax tree (AST) are located in ast.c and ast.h.
Structures mostly follows grammar defined syntax. There are some differences in expression structures for simpler implementation. This could possibly allow easier future implementation of FUNEXP and BOOL extensions.

Structures uses implementation of a LinkedList structure, defined in linked_list.h**.** Since this implementation stores (void *) on a stored element, correct pointer casting is needed. Type of a stored pointer is always written in the comment in structure definition.

```
typedef struct Program {
    LinkedList *functions;
} Program;
```

Toplevel structure Program consists of a list of function definitions. Each Function structure consists of a list of parameters and a Body - list of statements. For each statement structure is defined: ReturnStatement, WhileStatement, IfStatement, AssigmentStatement, VariableDefinitionStatement, FunctionCall. These structures can be found in ast.h except FunctionCall, which is located in expression_ast.h (already mentioned preparation for FUNEXP).

```
typedef struct Expression {
    struct DataType data_type;
    enum ExpressionType expr_type;
    enum ExpressionConversion conversion;
    union {
        struct FunctionCall function_call;
        struct Identifier identifier;
        struct Literal literal;
        struct BinaryExpression binary_expr;
    } data;
} Expression;
```

Expression trees are recursive, which means, dynamic allocation is needed. Toplevel structure Expression consists of:

- enum expression type - to indicate what union struct is active
- enum data type - for semantic controls and some codegen use cases
- enum conversions - this way the semantic analyzer can inform the code generator that conversion is needed.
- union for each type of expression.

There are 4 types of expressions:

- FunctionCall - not used, preparation for FUNEXP
- Identifier - used for reading variables
- Literal- node for any literal value (integer, float or string)
- BinaryExpression - this node consists of left and right operands of type Expression which are dynamically allocated, then stores information about operators.

Code Generation

(Vanesa Zimmermannová, **files**: `code_generator.c, builtin_generator.c`)

Code generation is closely connected to the AST. For each node structure from ast, there is a generation function. Generator recursively calls these functions, to generate code. Generated code is immediately written on to stdout. There are no optimizations.

Functions

Each function definition generates function frame consisting of:

```
LABEL function_<function name>
PUSHFRAME
CREATEFRAME
```

and always ends with (this is same for return statement, with exception of return value expression calculation):

```
POPFRAME
RETURN
```

This ensures that for each function there is a new TF, for storing its local variables. In case of recursive calls, frame stack will store all needed values. All local variables are always stored in TF. Generator does not in any instance use LF reference - it is not needed.

Function parameters are pushed to stack from left to right, which means, at the beginning of each function, they are popped in right to left direction. Function return value is pushed on stack, before RETURN instruction. (this would also support FUNEXP)

While statements

For while statements, at first, DEFVAR pregeneration is called. This functions () recursively traverses through the AST branch and for each needed DEFVAR an instruction is generated. This is needed because in case of repeated DEFVAR instruction on the same identifier, the interpreter raises an error. For each node generating DEFVAR (VariableDefinitionStatement, WhileStatement, IfStatement), there is bool value storing information whether this defvar was already generated. Pre Generating functions generate instruction and sets this bool to true, so regular generate or other pregenerate traversal (in case of nested cycles) will not generate this instruction again.

Expressions

Code generator recursively traverses the expression branch. If needed, literals are sometimes converted. Code generator transforms strings - some characters are replaced for their escaped version.
For floats, strtod conversion and then printf with %a. For integer and nil values no changes are needed, since those are just printed outright with prefix (int@, nil@).

In binary expressions, the first left subtree is generated. Its value will be stored on top of the stack. Then right subtree calculation is generated, this will push the result of the left subtree one value deeper and the result of the right subtree will stay on the top of stack. This means, calculation of correct result of binary expression, is then done just by calling right stack instruction (for example SUBS). (postfix generation)