Documentation for IFJ and IAL project

Implementation of IFJ24 imperative language translator

team xjordan00, variation vv-BVS

Team captain:	**Nikola Jordanov**	xjordan00	**25%**
Other members:	Lucie Pojslová	xpojsll00	25%
	Alexander Žikla	xziklaa00	25%
	Jakub Turek	xturekj00	25%

4.12.2024

Contents

Teamwork

Division of work between members

Nikola Jordanov(xjordan00)

Semantic analyzer

LL table

Precedence table

Symbol table

Abstract syntax tree implementation

Alexander Žikla (xziklaa00)

Lexical analyzer

Code generation

Lucie Pojslová (xpojsll00)

Syntactic analyzer

Jakub Turek (xturekj00)

Code generation

Documentation

Deviations from an even distribution of points

In our project, we aimed for an equal distribution of tasks among all members, and overall, we succeeded in maintaining a fair balance. However, there was a situation where one member faced challenges with code generation, and another member stepped in to provide additional support.

Development cycle

Our development began with designing a finite state machine (FSM) for the lexical analyzer, followed by implementing a dynamic string structure to store names and values effectively. We then focused on constructing an LL parser using recursive descent, translating each grammar rule into corresponding functions. A key part of our semantic analysis was building an efficient symbol table to manage variable and function scopes, which helped with proper symbol resolution. During the code generation phase, we implemented stack-based logic for handling values and introduced unique label counters to avoid conflicts, particularly with nested IF statements. Abstract Syntax Tree (AST) traversal was central to our semantic analysis, allowing us to ensure that all function and variable references were correct. Throughout the process, we iteratively improved each component, addressing challenges like ambiguous character handling, scope management, and nested operations.

Implementation of IFJ24 language translator

Lexical analyzer

We started by designing a finite state machine (FSM) to guide the development of the lexical scanner. Initially, we created custom data structures for tokens and their attributes, ensuring we could handle various token types effectively. We also implemented a dynamic string structure and its associated functions, used for storing function names, variable names (VAR), and string values. During the implementation, we discovered that our initial FSM design did not account for all possible characters, which necessitated several modifications throughout the process.

One specific problem involved handling the names of functions and variables correctly. Built-in functions that begin with "ifj." caused a conflict, allowing other functions and variables to have a dot in their names, leading to incorrect parsing. To solve this, we introduced additional states (State_IFJ_<1-4>), which allowed us to handle spaces before and after the dot, as well as properly manage function and variable names that start with "ifj"

Syntax analyzer

The parsing process began with defining LL rules that would guide the construction of the parser. These rules were crucial for enabling a recursive descent parsing approach. LL parsing rules form a basis for understanding the syntactical structure of the language, ensuring that all grammatical constructs are correctly recognized.

The recursive descent parser was implemented in a step-by-step manner. Each grammar rule was translated into a function, with recursion handling nested expressions and complex structures. The use of recursive descent allows a straightforward, top-down approach, making it easy to track and debug each step of the parsing process. This form of parsing is advantageous for ensuring that each token and structure is processed in an orderly, hierarchical fashion.

Semantic analyzer

To streamline the management of symbols and improve lookup efficiency, we implemented a semantic analyzer with a search list for symbol table checks. This choice made it easier to handle the scoping rules of the language and ensure that any variable or function was correctly associated with its defining context. While implementing this part, we faced some interesting challenges, such as handling nested scopes and differentiating between local and global symbols, but we found that using a search list

gave us the flexibility we needed to efficiently manage these complexities.

Generation

We began working on the code generator during the creation of the parser and semantic analysis phases. At this stage, we could declare functions and define those responsible for generating code in the IFJcode24 language (e.g., GenBuiltInFuncs()). Once we defined the structure of the abstract syntax tree (AST) and its various types (e.g., Binary operations), we proceeded to implement the necessary functions.

A major challenge was related to using stack-based instructions. It was difficult to determine the correct logic for pushing, popping, and transferring values to ensure the generated code was valid and usable.

Another challenge involved handling nested IF statements. Overlapping labels caused errors due to name conflicts. We resolved this issue by introducing a counter system: a global static variable (GlobalIfCounter) and a local counter (LocalIfCounter). At the beginning of each IF statement, the value of GlobalIfCounter was assigned to LocalIfCounter, and GlobalIfCounter was then incremented.This approach ensured that each label used within IF statements had a unique name.

Data structures

Abstract Syntax Tree Traversal

The Abstract Syntax Tree (AST) traversal formed the backbone of our semantic analysis. This process involved verifying the presence of specific functions and variables in the symbol table, as well as checking the correctness of

expressions and function calls. Essentially, our approach combined different tasks into a cohesive analysis phase.

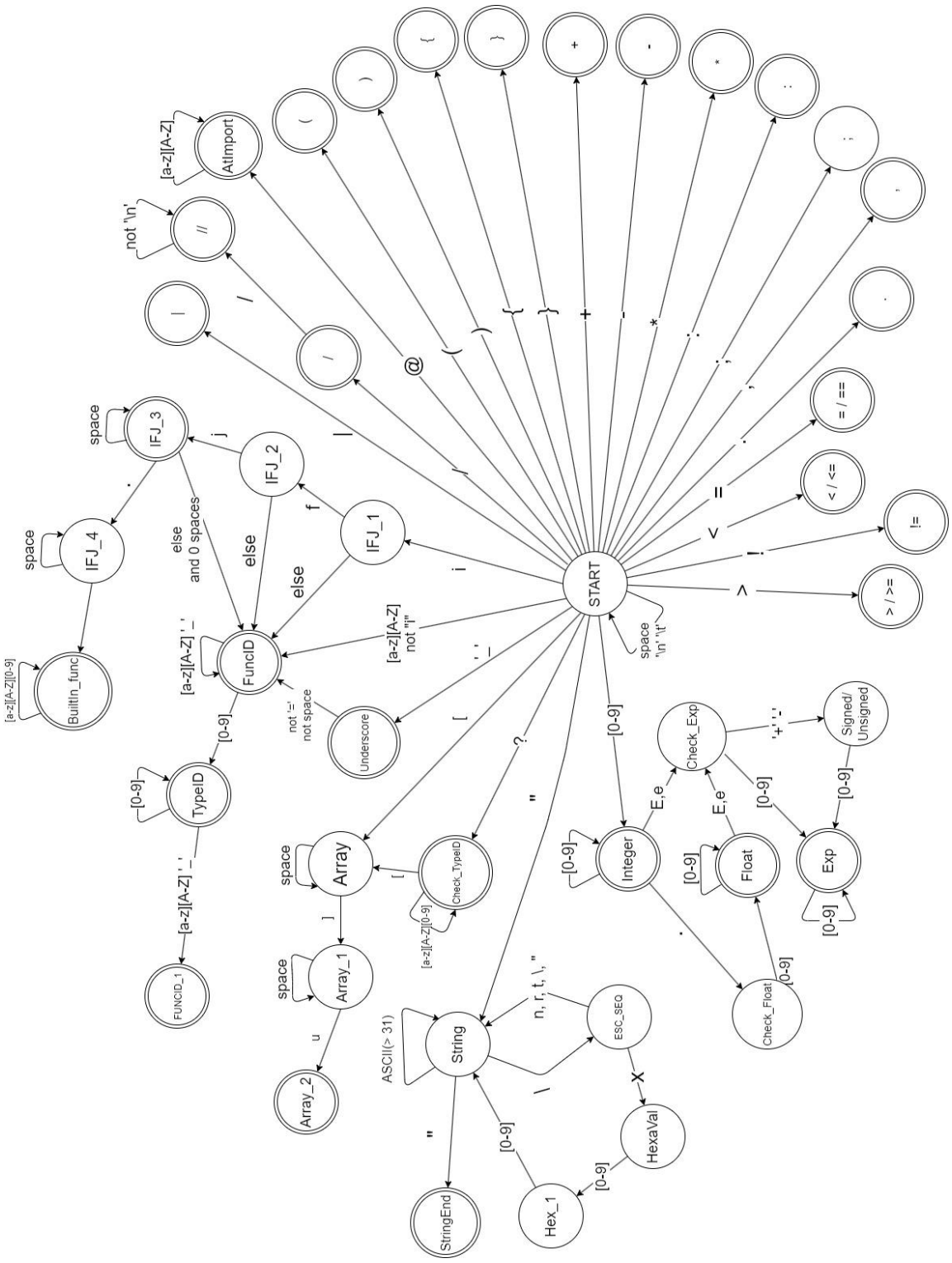
As we traversed the AST, we ensured that every reference to a variable or function matched a valid entry in the symbol table. This allowed us to catch undeclared variables, type mismatches, and other semantic errors early in the process. Alongside these checks, we also generated search trees for each function or sub-expression, which added another layer of structure to our analysis.

Stack

We used a stack during the AST traversal to manage function calls and handle nested scopes effectively. The stack allowed us to maintain context as we moved through different layers of the AST. This approach made it easier to keep track of active scopes and ensured that symbol resolution was both correct and efficient.

Attachments

Finite State Machine diagram



Precedence table

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

LL grammar

PROG → PROLOG STATEMENT NEXT_STATEMENT
PROLOG → token_const token_ifj token_= token_@import token_(token_"ifj24.zig" token_) token_;
STATEMENT → FCE
NEXT_STATEMENT → STATEMENT NEXT_STATEMENT | ε
FCE → token_pub token_fn token_id token_(ARGS_DEF token_) TYP token_{ FCE_BODY token_}
FCE_ID → token_ifj token_. IMPORTED_ID | token_id
IMPORTED_ID → token_readstr | token_readi32 | token_readf64 | token_write | token_i2f | token_f2i | token_string | token_length | token_concat | token_substring | token_strcmp | token_ord | token_chr

CONST_INIT → CONST_DECL CONST_DEF token_;
VAR_INIT → VAR_DECL VAR_DEF token_;
CONST_DEF → token_= EXPR | ε
VAR_DEF → token_= EXPR | ε
TYPE_DECL → token_: TYP | ε
NULL_TYP → token_?i32 | token_?f64 | token_?[]u8
NN_TYP → token_i32 | token_f64 | token_[]u8 | token_void
TYP → NULL_TYP | NN_TYP
CONST_DECL → token_const token_id TYPE_DECL
VAR_DECL → token_var token_id TYPE_DECL
ARGS_DEF → ARG_DEF NEXT_ARG_DEF | ε
ARG_DEF → token_id token_: TYP
NEXT_ARG_DEF → token_, ARGS_DEF | ε
FCE_BODY → BODY
RETURN → token_return RET_EXPR token_;
RET_EXPR → EXPR | ε
BODY → BODY_STATEMENT NEXT_BODY_STATEMENT
NEXT_BODY_STATEMENT → BODY | ε

BODY_STATEMENT → IF_BLOCK | WHILE_LOOP | VAR_INIT | CONST_INIT | ASSIGN_VAL | FCE_CALL token_; | RETURN | DISPOSE
DISPOSE → token_underscore token_= EXPR
ASSIGN_VAL → token_id token_= EXPR token_;
IF_BLOCK → IF ELSE
IF → token_if token_(EXPR token_) NON_NULL_ID token_{ BODY token_}
ELSE → token_else token_{ BODY token_} | ε
NON_NULL_ID → token_| token_id token_| | ε
WHILE_LOOP → token_while token_(EXPR token_) NON_NULL_ID token_{ BODY token_}
FCE_CALL → FCE_ID token_(ARGS token_)
ARGS → ARG NEXT_ARG | ε
ARG → token_id | EXPR
NEXT_ARG → token_, ARGS | ε
EXPR → OPERAND EXPR_NEXT
OPERAND → FCE_CALL | token_id | token_null | VAL | token_(EXPR token_)
VAL → token_int | token_float | token_string
EXPR_NEXT → OPERATOR OPERAND EXPR_NEXT | ε
OPERATOR → token_* | token_/ | token_+ | token_- | token_== | token_!= | token_< | token_> | token_<= | token_>=