LuCompiler

A very barebones approach at compiling.

How a compiler works

A compiler is a program which converts source code into machine code, which the Computer understands and can execute. It does this by following 4 major steps:

1. **Lexical analysis**
2. **Construction of the Abstract Syntax Tree**
3. **Semantics analysis**
4. **Code Generation (Assembly)**
5. **(Optional) Optimization**

For this Compiler, (For any compiler really) we'll use a process called bootstrapping in which a compiler is itself compiled using an existing compiler.

▼ 1 - Lexical analysis

This is the phase in which the compiler turns the source code into tokens, which are atomic units of information.

Here the file microC.l defines the syntax used in the MicroC language and tells the Lexer (Flex) how to interprete the code.

```
%{
#include "ast.h"
#include "microc.tab.h"
#include <stdlib.h>
#include <string.h>
extern void yyerror(const char *s);
```

```
int yywrap(void) {
    return 1;
}
%}
%%
"int"         { return INT; }
"void"        { return VOID; }
"if"          { return IF; }
"else"        { return ELSE; }
"while"       { return WHILE; }
"for"         { return FOR; }
"return"      { return RETURN; }
[a-zA-Z][a-zA-Z0-9]* { yylval.identifier = strdup(yytext); return IDENTIFIE
R; }
[0-9]+        { yylval.number = atoi(yytext); return NUMBER; }
"+"           { return PLUS; }
"-"           { return MINUS; }
"*"           { return MULT; }
"/"           { return DIVIDE; }
"="           { return ASG_OP; }
"=="          { return EQ_OP; }
"!="          { return NOT_EQ_OP; }
"<"           { return LESS_THAN_OP; }
">"           { return GREATER_THAN_OP; }
"<="          { return LESS_EQ_OP; }
">="          { return GREATER_EQ_OP; }
"&&"          { return AND_OP; }
"||"          { return OR_OP; }
"!"           { return NOT_OP; }
"("           { return LPAR; }
")"           { return RPAR; }
"{"           { return LBRACE; }
"}"           { return RBRACE; }
";"           { return SCOLON; }
","           { return COMMA; }
[ \t\n]+      ;
```

```
.           { fprintf(stderr, "Carattere non riconosciuto: %s\n", yytext); }
%%
```

Here the first section marked by `%{}%` is the declaration part of the YACC file, in which we include the libraries, in this case we also include `ast.h` which we'll see later. We also define `microc.tab.h` che viene generato dal Parser Bison, after that we find `yyerror` which is the error handling function Used by Bison, Extern indicates it is defined in another module. At last we find yywrap which is a Callback function used by Flex when it has reached EOF (End of File).

> 💡 **All the functions and variables which start with `yy` are the ones used by Flex**

We then find the Pattern Matching section marked by `%%` to the left we ffind the symbol we want to recognize and to the right in curly brackets the return type which we define, this basically says if you see "token" then return token_name. Here the rules follow a sort of right of way approach which is in descending order.

Special mention to the string token here: for which the regex is of trivial understanding but for the behavior associated with it i want to pay more attention. Here `yylval` is a Union defined in Bison to pass values to the parser, it sweeps the token then gives them to the parser. the Union has a `Identifier` attribute for which we dinamically allocate space in the heap based on the `yytext` variable which is a pointer to the current string in the code.

Then there's the rest of the tokens for which i wont go into detail as they're self explainatory.

> 💡 **A Union is an Abstract Data Type in C which can contains multiple types of values but in which only one of those values can be active at a time. In other words only a member at a time can hold information.**

▼ 1.5 - ast.h

Here we define the header file for the Abstract Syntax Tree.

An AST is a data structure that represents the structure of a program in the form of a tree. It's called abstract 'cause it does not represent the minute details of the real syntax, just the structure.

```
#ifndef AST_H
#define AST_H

#include <stdio.h>
#include <stdlib.h>

typedef struct Node Node;
typedef struct List List;

// Dichiarazione della variabile globale ast_root
extern Node* ast_root;

// Definizione dei tipi di nodo
typedef enum {
    NODE_PROGRAM,
    NODE_FUNCTION,
    NODE_DECLARATION,
    NODE_NUMBER,
    NODE_IDENTIFIER,
    NODE_PLUS,
    NODE_MINUS,
    NODE_MULT,
```

```
    NODE_DIVIDE,
    NODE_ASSIGN_OP,
    NODE_EQUAL_OP,
    NODE_NOT_EQUAL_OP,
    NODE_LESS_THAN_OP,
    NODE_GREATER_THAN_OP,
    NODE_IF_STMT,
    NODE_IF_ELSE_STMT,
    NODE_WHILE_STMT,
    NODE_RETURN_STMT,
    NODE_EXPR_STMT
} NodeType;

// Struttura di un nodo dell'albero sintattico
struct Node {
    NodeType type;
    union {
        // NODES BINARI
        struct {
            Node* left;
            Node* right;
        } binary_op;

        // NODE DI ASSEGNAZIONE
        struct {
            char* identifier;
            Node* expression;
        } assign_op;

        // NODES DI ISTITUZIONE
        struct {
            Node* expression;
        } expr_stmt;

        struct {
            Node* expression;
```

```
        } return_stmt;

        // NODES DI DICHIARAZIONE
        struct {
            char* identifier;
        } declaration_stmt;

        // NODE IF / IF-ELSE
        struct {
            Node* condition;
            List* if_body;
            List* else_body;
        } if_stmt;

        // NODE WHILE
        struct {
            Node* condition;
            List* while_body;
        } while_stmt;

        // NODES FOGLIA
        int number_val;
        char* identifier_name;

        // NODES DI PROGRAMMA E FUNZIONE
        struct {
            Node* function;
        } program_node;

        struct {
            char* name;
            List* declarations;
            List* statements;
        } function_def;
    };
};
```

```
// Struttura di una lista per le istruzioni e le dichiarazioni
struct List {
    Node* node;
    List* next;
};

// Funzioni per la creazione dei nodi
Node* new_node(NodeType type, ...);
List* new_list(Node* node, List* next);

// Funzioni helper per la creazione dei nodi specifici
Node* create_function_node(char* name, List* declarations, List* stateme
nts);
Node* create_declaration_node(char* identifier);
Node* create_return_node(Node* expression);
Node* create_expr_stmt_node(Node* expression);
Node* create_assign_node(char* identifier, Node* expression);
Node* create_binary_op_node(NodeType type, Node* left, Node* right);
Node* create_number_node(int value);
Node* create_identifier_node(char* name);
Node* create_if_node(Node* condition, List* if_body, List* else_body);
Node* create_while_node(Node* condition, List* while_body);
List* create_list_node(Node* node, List* next);

// Funzioni per la stampa
void print_ast(Node* node, int indent);
void print_list(List* list, int indent);

// Funzione per la pulizia della memoria
void free_ast(Node* node);
void free_list(List* list);

#endif
```

We first define the `Node` struct and the `List` struct which will be essential to the AST. We then declare the enum `NodeType` which will be used to identify all the possible node types, each one describes a behavior present in the source code.

We can classify the nodes in:

- Structural - NODE_PROGRAM, NODE_FUNCTION;
- Declarative - NODE_DECLARATION;
- Expression - NODE_NUMBER, NODE_IDENTIFIER, arithmetic/logic operators;
- Flow Control - NODE_IF_STMT, NODE_WHILE_STMT. NODE_RETURN_STMT.

Then we define the Node structure, it is made of a `NodeType type` field, then a union structure which contains all the possible structures depending on which node we are dealing with. For example the if node:

```
struct {
        Node* condition;  → the condition itself is a node.
        List* if_body;
        List* else_body;
    } if_stmt;
```

and

```
struct {
        Node* left;
        Node* right;
    } binary_op;
```

Which is used in all mathematical or logical operations involving two operands. E.g 2 > 4 becomes

```
struct binary_op→ >
Node* left → 2
Node* right → 4
```

💡 **In MicroC the only datatype used in mathematical operations is Integer**

Descending further we find the definition for the List struct, which is used to maintain longer and connected code portions, like large if statements, function parameters etc... they are also part of the ast and act just like any other node, just containing more sons than others. A List never exists by itself as it is always inside a node which is then added to the tree. After that we then find various variadic and normal functions for the creations of Nodes and Lists as well as the ones for the creation of all the nodes. Then some utility functions...

▼ 2 - ast.c

```
#include "ast.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>

void print_list(List *list, int indent);
List* reverse_list(List* list);

Node* new_node(NodeType type, ...) {
    Node* node = (Node*)malloc(sizeof(Node));
    if (!node) {
        perror("Errore di allocazione della memoria");
        exit(1);
    }
    node->type = type;

    va_list args;
    va_start(args, type);
```

```
    switch (type) {
        case NODE_PROGRAM:
            node->program_node.function = va_arg(args, Node*);
            break;
        case NODE_FUNCTION:
            node->function_def.name = va_arg(args, char*);
            node->function_def.declarations = va_arg(args, List*);
            node->function_def.statements = va_arg(args, List*);
            break;
        case NODE_DECLARATION:
            node->declaration_stmt.identifier = va_arg(args, char*);
            break;
        case NODE_NUMBER:
            node->number_val = va_arg(args, int);
            break;
        case NODE_IDENTIFIER:
            node->identifier_name = va_arg(args, char*);
            break;
        case NODE_PLUS:
        case NODE_MINUS:
        case NODE_MULT:
        case NODE_DIVIDE:
        case NODE_EQUAL_OP:
        case NODE_NOT_EQUAL_OP:
        case NODE_LESS_THAN_OP:
        case NODE_GREATER_THAN_OP:
            node->binary_op.left = va_arg(args, Node*);
            node->binary_op.right = va_arg(args, Node*);
            break;
        case NODE_ASSIGN_OP:
            node->assign_op.identifier = va_arg(args, char*);
            node->assign_op.expression = va_arg(args, Node*);
            break;
        case NODE_IF_STMT:
        case NODE_IF_ELSE_STMT:
            node->if_stmt.condition = va_arg(args, Node*);
```

```
            node->if_stmt.if_body = va_arg(args, List*);
            node->if_stmt.else_body = va_arg(args, List*);
            break;
        case NODE_WHILE_STMT:
            node->while_stmt.condition = va_arg(args, Node*);
            node->while_stmt.while_body = va_arg(args, List*);
            break;
        case NODE_RETURN_STMT:
            node->return_stmt.expression = va_arg(args, Node*);
            break;
        case NODE_EXPR_STMT:
            node->expr_stmt.expression = va_arg(args, Node*);
            break;
    }
    va_end(args);
    return node;
}

List* reverse_list(List* list) {
    List* prev = NULL;
    List* current = list;
    List* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    return prev;
}

List* new_list(Node *node, List *next) {
    List* list = (List*)malloc(sizeof(List));
    if (!list) {
```

```
        perror("Errore di allocazione della memoria");
        exit(1);
    }
    list->node = node;
    list->next = next;
    return list;
}


Node* create_function_node(char* name, List* declarations, List* stateme
nts) {
    // Inverti le liste per ottenere l'ordine corretto
    List* reversed_declarations = reverse_list(declarations);
    List* reversed_statements = reverse_list(statements);
    return new_node(NODE_FUNCTION, name, reversed_declarations, rever
sed_statements);
}

Node* create_declaration_node(char* identifier) {
    return new_node(NODE_DECLARATION, identifier);
}

Node* create_return_node(Node* expression) {
    return new_node(NODE_RETURN_STMT, expression);
}

Node* create_expr_stmt_node(Node* expression) {
    return new_node(NODE_EXPR_STMT, expression);
}

Node* create_assign_node(char* identifier, Node* expression) {
    return new_node(NODE_ASSIGN_OP, identifier, expression);
}

Node* create_binary_op_node(NodeType type, Node* left, Node* right) {
    return new_node(type, left, right);
```

```
}

Node* create_number_node(int value) {
    return new_node(NODE_NUMBER, value);
}

Node* create_identifier_node(char* name) {
    return new_node(NODE_IDENTIFIER, name);
}

Node* create_if_node(Node* condition, List* if_body, List* else_body) {
    List* reversed_if_body = reverse_list(if_body);
    List* reversed_else_body = else_body ? reverse_list(else_body) : NULL;

    if (else_body) {
        return new_node(NODE_IF_ELSE_STMT, condition, reversed_if_body, r
eversed_else_body);
    } else {
        return new_node(NODE_IF_STMT, condition, reversed_if_body, revers
ed_else_body);
    }
}

Node* create_while_node(Node* condition, List* while_body) {
    List* reversed_while_body = reverse_list(while_body);
    return new_node(NODE_WHILE_STMT, condition, reversed_while_body);
}

List* create_list_node(Node* node, List* next) {
    return new_list(node, next);
}

void print_ast(Node *node, int indent) {
    if (!node) return;

    for (int i = 0; i < indent; i++) printf("  ");
```

```
switch (node→type) {
    case NODE_PROGRAM:
        printf("Program\n");
        print_ast(node→program_node.function, indent + 1);
        break;
    case NODE_FUNCTION:
        printf("Function: %s\n", node→function_def.name);
        for (int i = 0; i < indent + 1; i++) printf("  ");
        printf("Declarations:\n");
        print_list(node→function_def.declarations, indent + 2);
        for (int i = 0; i < indent + 1; i++) printf("  ");
        printf("Statements:\n");
        print_list(node→function_def.statements, indent + 2);
        break;
    case NODE_DECLARATION:
        printf("Declaration: %s\n", node→declaration_stmt.identifier);
        break;
    case NODE_NUMBER:
        printf("Number: %d\n", node→number_val);
        break;
    case NODE_IDENTIFIER:
        printf("Identifier: %s\n", node→identifier_name);
        break;
    case NODE_PLUS:
        printf("+\n");
        print_ast(node→binary_op.left, indent + 1);
        print_ast(node→binary_op.right, indent + 1);
        break;
    case NODE_MINUS:
        printf("-\n");
        print_ast(node→binary_op.left, indent + 1);
        print_ast(node→binary_op.right, indent + 1);
        break;
    case NODE_MULT:
        printf("*\n");
```

```
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_DIVIDE:
            printf("/\n");
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_ASSIGN_OP:
            printf("=\n");
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("Identifier: %s\n", node->assign_op.identifier);
            print_ast(node->assign_op.expression, indent + 1);
            break;
        case NODE_EQUAL_OP:
            printf("==\n");
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_NOT_EQUAL_OP:
            printf("!=\n");
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_LESS_THAN_OP:
            printf("<\n");
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_GREATER_THAN_OP:
            printf(">\n");
            print_ast(node->binary_op.left, indent + 1);
            print_ast(node->binary_op.right, indent + 1);
            break;
        case NODE_IF_STMT:
            printf("If Statement\n");
```

```
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("Condition:\n");
            print_ast(node->if_stmt.condition, indent + 2);
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("If Body:\n");
            print_list(node->if_stmt.if_body, indent + 2);
            if (node->if_stmt.else_body) {
                for (int i = 0; i < indent + 1; i++) printf("  ");
                printf("Else Body:\n");
                print_list(node->if_stmt.else_body, indent + 2);
            }
            break;
        case NODE_IF_ELSE_STMT:
            printf("If-Else Statement\n");
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("Condition:\n");
            print_ast(node->if_stmt.condition, indent + 2);
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("If Body:\n");
            print_list(node->if_stmt.if_body, indent + 2);
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("Else Body:\n");
            print_list(node->if_stmt.else_body, indent + 2);
            break;
        case NODE_WHILE_STMT:
            printf("While Statement\n");
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("Condition:\n");
            print_ast(node->while_stmt.condition, indent + 2);
            for (int i = 0; i < indent + 1; i++) printf("  ");
            printf("While Body:\n");
            print_list(node->while_stmt.while_body, indent + 2);
            break;
        case NODE_RETURN_STMT:
            printf("Return Statement\n");
            print_ast(node->return_stmt.expression, indent + 1);
```

```
            break;
        case NODE_EXPR_STMT:
            printf("Expression Statement\n");
            print_ast(node->expr_stmt.expression, indent + 1);
            break;
    }
}

void print_list(List *list, int indent) {
    List *current = list;
    while (current != NULL) {
        print_ast(current->node, indent);
        current = current->next;
    }
}

void free_ast(Node* node) {
    if (!node) return;

    switch (node->type) {
        case NODE_PROGRAM:
            free_ast(node->program_node.function);
            break;
        case NODE_FUNCTION:
            free(node->function_def.name);
            free_list(node->function_def.declarations);
            free_list(node->function_def.statements);
            break;
        case NODE_DECLARATION:
            free(node->declaration_stmt.identifier);
            break;
        case NODE_IDENTIFIER:
            free(node->identifier_name);
            break;
        case NODE_PLUS:
        case NODE_MINUS:
```

```
        case NODE_MULT:
        case NODE_DIVIDE:
        case NODE_EQUAL_OP:
        case NODE_NOT_EQUAL_OP:
        case NODE_LESS_THAN_OP:
        case NODE_GREATER_THAN_OP:
            free_ast(node->binary_op.left);
            free_ast(node->binary_op.right);
            break;
        case NODE_ASSIGN_OP:
            free(node->assign_op.identifier);
            free_ast(node->assign_op.expression);
            break;
        case NODE_IF_STMT:
        case NODE_IF_ELSE_STMT:
            free_ast(node->if_stmt.condition);
            free_list(node->if_stmt.if_body);
            free_list(node->if_stmt.else_body);
            break;
        case NODE_WHILE_STMT:
            free_ast(node->while_stmt.condition);
            free_list(node->while_stmt.while_body);
            break;
        case NODE_RETURN_STMT:
            free_ast(node->return_stmt.expression);
            break;
        case NODE_EXPR_STMT:
            free_ast(node->expr_stmt.expression);
            break;
        default:
            break;
    }
    free(node);
}

void free_list(List* list) {
```

```
    List* current = list;
    while (current != NULL) {
        List* next = current->next;
        free_ast(current->node);
        free(current);
        current = next;
    }
}
```

Here we also include `<stdarg.h>` which is used to manage va_list, va_start, etc..

We then have the code portion which creates a new Node, we first use malloc to allocate memory, then check for failures and exit if one occours, then have `va_list args` for variable parameters, then a call to the `va_start(args,type)` function which initializes args making it point to the first parameter after type. This mechanism makes it possible to access parameters of an unknown type at compile-time. it sets args to point at the first parameter after type, it then continues until there are no parameters left to read in memory. The parameters are then assigned to the attributes of each node type. We then find the reverse_list function which is used beacuse during parsing adding elements to the head of the list insead of the tail is more efficient. We use the three pointer iterative method, here we first save the current->next pointer in the next variable then we set the pointer to prev, which in the first iteration is NULL, then we move forward by setting prev to current and current to next, rinse and repeat..

Then there's the List creation function along with the print functions and the freeing functions...

2.5 - microC.y

This file is used by Bison a.k.a the parser. This file defines the grammar rules and builds the AST.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include "ast.h"
#include "microc.tab.h"

Node* ast;
void yyerror(const char* s);
extern int yylex();
%}

%union {
   int number;
   char* identifier;
   Node* node;
   List* list;
}

%token INT
%token VOID
%token RETURN
%token IF
%token ELSE
%token WHILE
%token FOR
%token SCOLON
%token LBRACE
%token RBRACE
%token LPAR
%token RPAR
%token COMMA
%token ASG_OP
%token PLUS
%token MINUS
%token MULT
```

```
%token DIVIDE
%token EQ_OP
%token NOT_EQ_OP
%token LESS_THAN_OP
%token GREATER_THAN_OP
%token LESS_EQ_OP
%token GREATER_EQ_OP
%token AND_OP
%token OR_OP
%token NOT_OP

%token<number> NUMBER
%token<identifier> IDENTIFIER

%type<node> program
%type<node> function_declaration
%type<node> declaration_statement
%type<node> statement
%type<node> expression_statement
%type<node> expression
%type<node> assignment_expression
%type<node> relational_expression
%type<node> additive_expression
%type<node> multiplicative_expression
%type<node> primary_expression
%type<node> if_statement
%type<node> while_statement
%type<node> return_statement
%type<list> declarations
%type<list> statements

%% //regole grammaticali

program: function_declaration { ast = $1; };

function_declaration: INT IDENTIFIER LPAR RPAR LBRACE declarations stat
```

```
ements RBRACE {
   $$ = create_function_node($2, $6, $7);
};

declarations: declarations declaration_statement { $$ = create_list_node
($2, $1); }
          | /* empty */ { $$ = NULL; };

statements: statements statement { $$ = create_list_node($2, $1); }
         | /* empty */ { $$ = NULL; };

statement: expression_statement { $$ = $1; }
        | if_statement { $$ = $1; }
        | while_statement { $$ = $1; }
        | return_statement { $$ = $1; };

declaration_statement: INT IDENTIFIER SCOLON { $$ = create_declaration_
node($2); };

return_statement: RETURN expression SCOLON { $$ = create_return_node
($2); };

expression_statement: expression SCOLON { $$ = create_expr_stmt_node
($1); };

expression: assignment_expression { $$ = $1; };

assignment_expression: IDENTIFIER ASG_OP expression { $$ = create_assi
gn_node($1, $3); }
                    | relational_expression { $$ = $1; };

relational_expression: additive_expression EQ_OP additive_expression { $$
= create_binary_op_node(NODE_EQUAL_OP, $1, $3); }
                    | additive_expression NOT_EQ_OP additive_expression { $$ =
create_binary_op_node(NODE_NOT_EQUAL_OP, $1, $3); }
                    | additive_expression LESS_THAN_OP additive_expression {
```

```
$$ = create_binary_op_node(NODE_LESS_THAN_OP, $1, $3); }
                | additive_expression GREATER_THAN_OP additive_expression
{ $$ = create_binary_op_node(NODE_GREATER_THAN_OP, $1, $3); }
                | additive_expression { $$ = $1; };

additive_expression: additive_expression PLUS multiplicative_expression {
$$ = create_binary_op_node(NODE_PLUS, $1, $3); }
                | additive_expression MINUS multiplicative_expression { $$ = cr
eate_binary_op_node(NODE_MINUS, $1, $3); }
                | multiplicative_expression { $$ = $1; };

multiplicative_expression: multiplicative_expression MULT primary_express
ion { $$ = create_binary_op_node(NODE_MULT, $1, $3); }
                    | multiplicative_expression DIVIDE primary_expression { $$
= create_binary_op_node(NODE_DIVIDE, $1, $3); }
                    | primary_expression { $$ = $1; };

primary_expression: NUMBER { $$ = create_number_node($1); }
                | IDENTIFIER { $$ = create_identifier_node($1); }
                | LPAR expression RPAR { $$ = $2; };

if_statement: IF LPAR expression RPAR LBRACE statements RBRACE { $$ =
create_if_node($3, $6, NULL); }
            | IF LPAR expression RPAR LBRACE statements RBRACE ELSE LBRA
CE statements RBRACE { $$ = create_if_node($3, $6, $10); };

while_statement: WHILE LPAR expression RPAR LBRACE statements RBRA
CE { $$ = create_while_node($3, $6); };

%%

void yyerror(const char *s) {
   fprintf(stderr, "Errore di parsing: %s\n", s);
}
```

The first section defined by `%{}%` is the prologue, like microC.l, in which we include the various libraries. We also decleare the lex `int yylex` generated by Flex. We then find the Union structure which returns the possible datatypes associated with the tokens. yylval will be of this Union type, and it is used by Flex to pass data to Bison. We then have the various token declarations e.g `%token` delcares a token without a semantic value associated with it, `%type<*>` declares a non terminal of a specific type which are crucial to have structure in our program, they are a "higher level of token". Most of them are Node* while declaratios and statements are List* which is used for type-checking while building the AST. Think of them as a big clump of tokens which are structure in a specifc way, and we refer to them as oppsose to every little piece forming it.

The last section, the one in which we define grammar rules is then indicated by `%%` . Program: is the starting grammar symbol, and $1 means the value of the first symbol to the right.

```
program: function_declaration { ast = $1; };
```

Here the first LHS (Left-Hand side) symbol represents the entire program. and we say, the entire program is a function, were function_declaration is the RHS symbol, we then say, the resulting node of the function which is $1 is to be stored inside a globagl pointer called ast. We then do the same thing but with the other constructs such as if's, multiplications, so on and so forth...

$$ represents the value of the LHS non terminal while $n represent the LHS's

3 - codegen.h

Here we have the header file for the process of assembly code generation:

```
#ifndef CODEGEN_H
#define CODEGEN_H

#include "ast.h" // Per accedere alla struttura dei nodi dell'AST

// Struttura per un singolo elemento della tabella dei simboli
typedef struct Symbol {
    char* name;
    int offset;
    struct Symbol* next;
} Symbol;

// Funzioni per la tabella dei simboli
void add_symbol(char* name, int offset);
int get_symbol_offset(char* name);
void free_symbol_table();

// Prototipo della funzione principale di generazione del codice
void generate_code(Node* ast_root);
void generate_assembly(Node* ast, const char* filename);
#endif // CODEGEN_H
```

Here we define the Symbol table, a fundamental structure used to keep track of every identifer used in the program. Here it is defined by its name, offset in memory and a pointer to the next identifier. Then we have some functions prototypes used for code generation, memory deallocation and symbol creation...

3.5 - codegen.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "ast.h"
#include "codegen.h"
//ao
// Tabella dei simboli per tenere traccia delle variabili locali.
// Questa è una lista concatenata semplice che associa il nome della variabile all'offset dello stack.
// Sta cosa serve al compilatore per evitare anche di ridefinire variabili, control lare la semantica del codice e viene gestita dal stm symbol table manager che lavora con l'error handler circa.
typedef struct Symbol {
    char* name;
    int offset;
    struct Symbol* next;
} Symbol;

static Symbol* symbol_table = NULL;
static int offset_counter = -4;
static int label_count = 0;

// Aggiunge un nuovo simbolo alla tabella dei simboli.
void add_symbol(char* name, int offset) {
    Symbol* new_symbol = (Symbol*)malloc(sizeof(Symbol));
    if (!new_symbol) {
        perror("Errore di allocazione del simbolo");
        exit(EXIT_FAILURE);
    }
    new_symbol->name = strdup(name);
    new_symbol->offset = offset;
    new_symbol->next = symbol_table;
    symbol_table = new_symbol;
}
```

```
// Restituisce l'offset di una variabile dalla tabella dei simboli.
int get_symbol_offset(char* name) {
    Symbol* current = symbol_table;
    while (current) {
        if (strcmp(current→name, name) == 0) {
            return current→offset;
        }
        current = current→next;
    }
    fprintf(stderr, "Errore: variabile '%s' non dichiarata.\n", name);
    return 0;
}

// Libera la memoria allocata per la tabella dei simboli.
void free_symbol_table() {
    Symbol* current = symbol_table;
    while (current) {
        Symbol* temp = current;
        current = current→next;
        free(temp→name);
        free(temp);
    }
    symbol_table = NULL;
}

// Prototipi delle funzioni di generazione del codice.
static void generate_statement(Node* node, FILE* output_file);
static void generate_statements(List* list, FILE* output_file);
static void generate_expression(Node* node, FILE* output_file);
static char* generate_label();

// Funzione principale per la generazione del codice assembly.
void generate_assembly(Node* ast, const char* filename) {
    FILE* output_file = fopen(filename, "w");
    if (!output_file) {
        perror("Impossibile aprire il file di output");
```

```
        return;
    }

    // Genera il codice a partire dalla radice dell'AST.
    generate_statement(ast, output_file);
    free_symbol_table();
    fclose(output_file);
}

// Genera il codice per una singola espressione.
static void generate_expression(Node* node, FILE* output_file) {
    if (!node) return;

    switch (node->type) {
        case NODE_NUMBER:
            // Sposta il valore numerico nel registro EAX.
            fprintf(output_file, "  movl $%d, %%eax\n", node->number_val);
            break;
        case NODE_IDENTIFIER:
            // Sposta il valore della variabile dal suo offset nello stack a EAX.
            fprintf(output_file, "  movl %d(%%ebp), %%eax\n", get_symbol_offset
(node->identifier_name));
            break;
        case NODE_PLUS:
        case NODE_MINUS:
        case NODE_MULT:
        case NODE_DIVIDE:
            // Le espressioni binare vengono risolte valutando il lato sinistro,
            // mettendo il risultato sullo stack, valutando il lato destro,
            // e poi eseguendo l'operazione.
            generate_expression(node->binary_op.left, output_file);  // CORREZION
E: prima sinistra
            fprintf(output_file, "  pushl %%eax\n");
            generate_expression(node->binary_op.right, output_file); // poi destra
            fprintf(output_file, "  popl %%ebx\n");
```

```
            if (node->type == NODE_PLUS) {
                fprintf(output_file, "  addl %%ebx, %%eax\n");
            } else if (node->type == NODE_MINUS) {
                fprintf(output_file, "  subl %%eax, %%ebx\n");
                fprintf(output_file, "  movl %%ebx, %%eax\n");
            } else if (node->type == NODE_MULT) {
                fprintf(output_file, "  imull %%ebx, %%eax\n");
            } else if (node->type == NODE_DIVIDE) {
                fprintf(output_file, "  movl %%ebx, %%eax\n");  // CORREZIONE: me
tti il dividendo in EAX
                fprintf(output_file, "  cdq\n");
                fprintf(output_file, "  popl %%ebx\n");  // divisore in EBX
                fprintf(output_file, "  idivl %%ebx\n");
            }
            break;
        case NODE_ASSIGN_OP:
            // Valuta l'espressione a destra e assegna il risultato alla variabile.
            generate_expression(node->assign_op.expression, output_file);
            fprintf(output_file, "  movl %%eax, %d(%%ebp)\n", get_symbol_offset
(node->assign_op.identifier));
            break;
        case NODE_EQUAL_OP:
        case NODE_NOT_EQUAL_OP:
        case NODE_LESS_THAN_OP:
        case NODE_GREATER_THAN_OP:
            // Genera codice per le operazioni di confronto.
            generate_expression(node->binary_op.left, output_file);   // CORREZIO
NE: prima sinistra
            fprintf(output_file, "  pushl %%eax\n");
            generate_expression(node->binary_op.right, output_file);  // poi destra
            fprintf(output_file, "  popl %%ebx\n");
            fprintf(output_file, "  cmpl %%eax, %%ebx\n");  // CORREZIONE: confr
onta ebx con eax

            if (node->type == NODE_EQUAL_OP) {
                fprintf(output_file, "  sete %%al\n");
```

```
        } else if (node->type == NODE_NOT_EQUAL_OP) {
            fprintf(output_file, "  setne %%al\n");
        } else if (node->type == NODE_LESS_THAN_OP) {
            fprintf(output_file, "  setl %%al\n");
        } else if (node->type == NODE_GREATER_THAN_OP) {
            fprintf(output_file, "  setg %%al\n");
        }
        // Mette il risultato (0 o 1) nel registro EAX.
        fprintf(output_file, "  movzbl %%al, %%eax\n");
        break;
    }
}

// Genera il codice per una singola istruzione.
static void generate_statement(Node* node, FILE* output_file) {
    if (!node) return;

    switch (node->type) {
        case NODE_PROGRAM:
            generate_statement(node->program_node.function, output_file);
            break;
        case NODE_FUNCTION:
            fprintf(output_file, ".globl main\n");
            fprintf(output_file, "main:\n");
            fprintf(output_file, "  pushl %%ebp\n");
            fprintf(output_file, "  movl %%esp, %%ebp\n");

            // Conta le variabili dichiarate per allocare spazio sullo stack
            List* decl_list = node->function_def.declarations;
            int var_space = 0;
            while(decl_list) {
                var_space += 4;
                decl_list = decl_list->next;
            }
            if (var_space > 0) {
                fprintf(output_file, "  subl $%d, %%esp\n", var_space);
```

```
        }

        // Aggiungi le dichiarazioni alla tabella dei simboli
        decl_list = node->function_def.declarations;
        offset_counter = -4;
        while(decl_list) {
            add_symbol(decl_list->node->declaration_stmt.identifier, offset_coun
ter);
            offset_counter -= 4;
            decl_list = decl_list->next;
        }

        generate_statements(node->function_def.statements, output_file);

        // Epilogo della funzione (solo se non c'è già un return esplicito)
        fprintf(output_file, "  movl $0, %%eax\n");
        fprintf(output_file, "  movl %%ebp, %%esp\n");
        fprintf(output_file, "  popl %%ebp\n");
        fprintf(output_file, "  ret\n");
        break;
    case NODE_RETURN_STMT:
        generate_expression(node->return_stmt.expression, output_file);
        fprintf(output_file, "  movl %%ebp, %%esp\n");
        fprintf(output_file, "  popl %%ebp\n");
        fprintf(output_file, "  ret\n");
        break;
    case NODE_EXPR_STMT:
        generate_expression(node->expr_stmt.expression, output_file);
        break;
    case NODE_IF_STMT: {
        char* end_if_label = generate_label();
        generate_expression(node->if_stmt.condition, output_file);
        fprintf(output_file, "  cmpl $0, %%eax\n");
        fprintf(output_file, "  je %s\n", end_if_label);
        generate_statements(node->if_stmt.if_body, output_file);
        fprintf(output_file, "%s:\n", end_if_label);
```

```
            free(end_if_label);
            break;
        }
        case NODE_IF_ELSE_STMT: {
            char* else_label = generate_label();
            char* end_if_else_label = generate_label();
            generate_expression(node->if_stmt.condition, output_file);
            fprintf(output_file, "  cmpl $0, %%eax\n");
            fprintf(output_file, "  je %s\n", else_label);
            generate_statements(node->if_stmt.if_body, output_file);
            fprintf(output_file, "  jmp %s\n", end_if_else_label);
            fprintf(output_file, "%s:\n", else_label);
            generate_statements(node->if_stmt.else_body, output_file);
            fprintf(output_file, "%s:\n", end_if_else_label);
            free(else_label);
            free(end_if_else_label);
            break;
        }
        case NODE_WHILE_STMT: {
            char* start_while_label = generate_label();
            char* end_while_label = generate_label();
            fprintf(output_file, "%s:\n", start_while_label);
            generate_expression(node->while_stmt.condition, output_file);
            fprintf(output_file, "  cmpl $0, %%eax\n");
            fprintf(output_file, "  je %s\n", end_while_label);
            generate_statements(node->while_stmt.while_body, output_file);
            fprintf(output_file, "  jmp %s\n", start_while_label);
            fprintf(output_file, "%s:\n", end_while_label);
            free(start_while_label);
            free(end_while_label);
            break;
        }
    }
}

// Genera il codice per una lista di istruzioni.
```

```
static void generate_statements(List* list, FILE* output_file) {
    if (!list) return;
    List* current = list;
    while(current) {
        generate_statement(current->node, output_file);
        current = current->next;
    }
}

// Genera un'etichetta unica.
static char* generate_label() {
    char* label_name = (char*)malloc(16);
    if (!label_name) {
        perror("Errore di allocazione");
        exit(EXIT_FAILURE);
    }
    sprintf(label_name, "L%d", label_count++);
    return label_name;
}
```

Here we first have the definition of the `symbol_table`, `offset_counter` and `label_count` all being static to achieve internal linakge. The offset counter is set to -4 as the integer size is 4bytes and we substract this from the *rbp*.

We then have the body for the afore mentioned functions in `codegen.h`. We then find the major code generation function. `generate_assembly` in which we first declare a file pointer and then call the `generate_Statement` function recursively on the ast, then we deallocate memory and close the file pointer.

After that we find the `generate_expression` function which generates the assembly code for a single expression, for example let's look at the first case of the switch were a number is to be generated, here we perform a Move Long (32bit) operation, the value of which is $%d (where $ is a prefix for constants) to the EAX register.

We then find some more assembly isntructions the meaning of which I am to lazy to go back and remember :D

And at last we have the generate statements function body and then under that the generate_label function which allcoates some space for a string then writes on the opened file the label which here are L1 L2 L3....

4 - main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "ast.h"
#include "codegen.h"

// Dichiarazione delle funzioni esterne del parser
extern int yyparse();
extern FILE *yyin;
extern Node* ast; // Questa è la variabile definita nel parser

// Definizione della variabile globale per la radice dell'AST
Node* ast_root = NULL;

int main(int argc, char **argv) {
    if (argc < 2) {
        fprintf(stderr, "Uso: %s <file_di_input.mc>\n", argv[0]);
        return 1;
    }

    // Apri il file di input
    yyin = fopen(argv[1], "r");
    if (!yyin) {
        perror("Errore nell'apertura del file di input");
        return 1;
    }

    // Analisi sintattica e costruzione dell'AST
```

```
    printf("Parsing in corso...\n");
    int result = yyparse();

    // Chiudi il file di input
    fclose(yyin);

    // Se l'analisi sintattica ha avuto successo, genera l'output
    if (result == 0 && ast) {
        ast_root = ast; // Copia il riferimento
        printf("AST generato con successo. Stampa dell'AST:\n");
        print_ast(ast_root, 0);

        printf("\nGenerazione del codice assembly...\n");
        generate_assembly(ast_root, "output.s");
        printf("Codice assembly salvato in 'output.s'.\n");

        free_ast(ast_root);
    } else {
        fprintf(stderr, "Errore di parsing. Impossibile generare l'AST.\n");
        return 1;
    }

    return 0;
}
```

Here we define all the variables the parser and Lexer are going to use to access the ast and then the file in which it will write the assembly code. Here most of the code is commented. This is the most abstracted view of the process.

Here’s the pipeline of the seconf half of the code:

1. **Verify the success of parsing and existence of AST;**
2. **Debug print the AST;**
3. **Assembly generation**
4. **Memory cleanup**

5. **Returning 0 if successful**

5 - test.mc

This is a test code that can be used to test the compiler.

```
int main() {
  int a;
  while(a < 10){
    a = a + 1;
  }
  return a + 2;
}
```

Here is how to build and execute

```
flex microc.l
bison -d microc.y
gcc -o luCompiler main.c ast.c codegen.c microc.tab.c lex.yy.c -m32

./luCompiler test.mc

gcc -o test_program output.s -m32

./test_program

echo %status
```