dummy_ast.rs

```rust
use std::{io::stderr, ops::Deref};
use swc::{Compiler, config::SourceMapsConfig};
use swc_ecma_parser::{lexer::Lexer, Syntax, EsConfig, StringInput, Parser};
use swc_common::{sync::Lrc, errors::Handler, SourceMap, FileName, SourceFile, DUMMY_SP};
use swc_ecma_ast::{EsVersion, Module, ModuleItem, Stmt, Decl, FnDecl, Ident, Function};
use swc_ecma_ast::{ReturnStmt, BlockStmt, Expr, BinExpr, BinaryOp, Lit, Number};

/// init ast generate function

type MyLexer<'a> = Lexer<'a, StringInput<'a>>;

type MyParser<'a> = Parser<MyLexer<'a>>;

fn init_compiler() -> (Lrc<SourceMap>, Compiler) {
    let cm: Lrc<SourceMap> = Default::default();
    let compiler = Compiler::new(cm.clone());
    return (cm.clone(), compiler);
}

fn init_file(cm: Lrc<SourceMap>, src: String) -> Lrc<SourceFile> {
    let filename = FileName::Custom("dummy_file.js".into());
    let fm = cm.new_source_file(filename, src.into());
    return fm;
}

fn init_lexer(fm: &SourceFile) -> MyLexer {
    let lexer = Lexer::new(
        Syntax::Es(EsConfig {
            jsx: false,
            fn_bind: false,
            decorators: true,
            decorators_before_export: false,
            export_default_from: false,
            import_assertions: false,
            static_blocks: false,
            private_in_object: false,
            allow_super_outside_method: false,
        }),
        EsVersion::Es2016,
        StringInput::from(fm),
        None,
    );
    return lexer;
}

fn init_parser<'a>(lexer: MyLexer<'a>, compiler: &'a Compiler) -> MyParser<'a> {
    let mut parser = Parser::new_from(lexer);

    let _handler = Handler::with_emitter_writer(Box::new(stderr()), Some(compiler.cm.clone()));
    let list_error = parser.take_errors();
    if list_error.iter().len() > 0 {
        let mut err_msg = "".to_owned();
        for err in list_error {
            let msg = err.into_kind().msg().to_string();
            err_msg.push_str(msg.as_str());
        }
    }

    return parser;
}


/// do parse and transform action

fn parser_parse(mut parser: MyParser, show_ast: bool) -> Module {
    let module = parser.parse_module().unwrap();
    println!(">>>>> parse success\n");

    let ast = serde_json::to_string_pretty(&module).expect("failed to serialize");

    if show_ast {
        println!(">>>>> ast json is \n{}\n", ast);
    }

    return module;
}

fn ast_transform(module: Module, f: fn(Module) -> Module) -> Module {
    return f(module);
}

fn print_result(compiler: Compiler, module: &Module) {
    let result = compiler.print(
        module,
        None,
        None,
        false,
        EsVersion::Es2016,
        SourceMapsConfig::Bool(false),
        &Default::default(),
        None,
        false,
        None,
    ).unwrap();

    println!(">>>>> generate success\n");

    println!(">>>>> generate code \n\n{}\n", result.code)
}

/// define your own ast code
```

```rust

type SpecifierInfo = Vec<(String, String, f64)>;

fn calc_rtn_literal_number(module: &Module) -> SpecifierInfo {
    let mut specifiers = vec![];
    for item in &module.body {
        if let ModuleItem::Stmt(Stmt::Decl(Decl::Fn(fn_struct))) = item {
            let fn_struct_str = format!("{:?}", &fn_struct);
            let func_name = &fn_struct.ident.sym.to_string();
            let func_body = &fn_struct.function.body;

            if let Some(block_stmt) = func_body {
                if block_stmt.stmts.iter().len() == 1 {
                    let this_block_stmt = block_stmt.stmts.iter().next();
                    if let Some(rtn_stmt) = this_block_stmt {
                        if let Stmt::Return(ReturnStmt { arg, .. }) = rtn_stmt {
                            if let Some(expr) = arg {
                                let this_expr = expr.deref();
                                if let Expr::Bin(BinExpr { op, left, right, .. }) = this_expr {
                                    if &BinaryOp::Add == op {
                                        let l_expr = left.deref();
                                        if let Expr::Lit(Lit::Num(Number { value: lv, .. })) = l_expr {
                                            let r_expr = right.deref();
                                            if let Expr::Lit(Lit::Num(Number { value: rv, .. })) = r_expr {
                                                let (add_left, add_right) = (*lv, *rv);
                                                specifiers.push((func_name.clone(), fn_struct_str, add_left + add_right));
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    for item in &specifiers {
        let (func_name, _fn_struct_str, rtn_number) = item;
        println!("func_name: {}, rtn_number: {}", func_name, rtn_number);
    }

    if specifiers.iter().len() > 0 {
        println!()
    }

    return specifiers;
}

fn replace_literal_number(module: &Module, specifiers: SpecifierInfo) -> Module {
    let mut build_module = Module {
        span: DUMMY_SP,
        body: vec![],
        shebang: module.shebang.clone(),
    };

    let stat_temp = |stmt| { ModuleItem::Stmt(stmt) };
    let decl_temp = |decl| { stat_temp(Stmt::Decl(decl)) };
    let fn_temp = |fn_struct| { decl_temp(Decl::Fn(fn_struct)) };

    for item in &module.body {
        let mut same_name_struct = false;
        let mut rtn_number = 0 as f64;

        match item {
            ModuleItem::Stmt(stmt) => {
                match stmt {
                    Stmt::Decl(decl) => {
                        match decl {
                            Decl::Fn(fn_struct) => {
                                let this_fn_struct_str = format!("{:?}", &fn_struct);
                                let this_func_name = &fn_struct.ident.sym.to_string();

                                for spec in specifiers.iter() {
                                    let (fn_name, fn_struct_str, rtn_value) = spec;
                                    if (fn_name == this_func_name) && (&this_fn_struct_str == fn_struct_str) {
                                        same_name_struct = true;
                                        rtn_number = *rtn_value;
                                    }
                                }

                                if same_name_struct {
                                    let literal_number = Expr::Lit(Lit::Num(Number { span: DUMMY_SP, value: rtn_number }));
                                    let literal_node = Stmt::Return(ReturnStmt { span: DUMMY_SP, arg: Some(Box::from(literal_number))
                                    let new_fn_struct = FnDecl {
                                        ident: Ident {
                                            span: DUMMY_SP,
                                            sym: fn_struct.ident.sym.clone(),
                                            optional: fn_struct.ident.optional.clone(),
                                        },
                                        declare: fn_struct.declare.clone(),
                                        function: Function {
                                            params: fn_struct.function.params.clone(),
                                            decorators: fn_struct.function.decorators.clone(),
                                            span: DUMMY_SP,
                                            body: Some(BlockStmt { span: DUMMY_SP, stmts: vec![literal_node] }),
                                            is_generator: fn_struct.function.is_generator.clone(),
                                            is_async: fn_struct.function.is_async.clone(),
                                            type_params: fn_struct.function.type_params.clone(),
                                            return_type: fn_struct.function.return_type.clone(),
                                        },
                                    };

                                    build_module.body.push(fn_temp(new_fn_struct))
                                } else {
                                    build_module.body.push(fn_temp((*fn_struct).clone()))
                                }
                            }
                            _ => { build_module.body.push(decl_temp((*decl).clone())) }
                        }
                    }
                    _ => { build_module.body.push(stat_temp((*stmt).clone())) }
                }
            }
            _ => { build_module.body.push((*item).clone()) }
        }
    }

    return build_module;
}


fn my_ast_func(module: Module) -> Module {
    let specifiers = calc_rtn_literal_number(&module);
    let build_module = replace_literal_number(&module, specifiers);
    return build_module;
}


fn main() {
    let dummy_js = "function calc(){return 1+2;}".into();

    let (cm, compiler) = init_compiler();
    let fm = init_file(cm, dummy_js);
    let lexer = init_lexer(&fm.deref());
    let parser = init_parser(lexer, &compiler);

    let mut module = parser_parse(parser, false);
    module = ast_transform(module, my_ast_func);
    print_result(compiler, &module);

    let _stdout_str = "\
>>>>> parse success

func_name: calc, rtn_number: 3

>>>>> generate success

>>>>> generate code

function calc() {
    return 3;
}

";
}
```