# MiniTS 프로젝트 설명서

## 1. 언어 개요 및 설계 의도

- TypeScript의 가장 작은 정적 타입 코어(number/boolean/string/void, 함수, 제어문)만 남긴 언어입니다.
- 목적: 렉싱 → 파싱 → 타입체크 → JS 코드 생성의 전 단계를 단순하고 읽기 쉬운 코드로 보여주는 것.
- 모든 타입은 명시적이며, 암묵 변환을 허용하지 않아 오류를 빠르게 드러냅니다.
- 동기: TypeScript가 JS로 어떻게 변환되는지 궁금해 컴파일러 수업을 들었습니다. 학교에서 miniC 컴파일러를 학습하며 컴파일 프로세스를 학습했습니다. 해당 지식을 바탕으로 miniTS를 만들어, 컴파일 과정을 직접 경험해볼 수 있었습니다.

## 2. 문법(Grammar) 정의

### 2.1 어휘 규칙

- 식별자: `[A-Za-z_][A-Za-z0-9_]*`
- 키워드: `let`, `function`, `if`, `else`, `while`, `for`, `return`, `true`, `false`
- 타입 키워드: `number`, `boolean`, `string`, `void`
- 리터럴
  - 숫자: 정수·실수(`123`, `3.14`)
  - 문자열: 쌍따옴표로 감싼 문자열, `\n`, `\t`, `\"`, `\\` 이스케이프 지원
  - 불리언: `true`, `false`
- 연산자: `+ - * / ! && || == != === !== < <= > >= =`
- 구분자: `( ) { } : , ;`
- 주석: 한 줄(`// ...`), 여러 줄(`/* ... */`, 중첩 미지원)
- 공백/개행은 토큰 경계만 분리하며 무시됩니다.

### 2.2 타입

- 원시 타입만 지원: `number | boolean | string | void`
- 변수: 타입 어노테이션 생략 가능. 생략 시 초기값으로 타입을 추론하며, 초기값이 없으면 오류.
- 함수: 매개변수와 반환 타입은 필수로 명시해야 합니다.

### 2.3 문법(EBNF)

```
Program         ::= Declaration*
Declaration     ::= VarDecl | FunctionDecl | Statement

VarDecl         ::= "let" Identifier (":" Type)? ("=" Expression)? ";"
FunctionDecl    ::= "function" Identifier "(" ParamList? ")" ":" Type Block
ParamList       ::= Param ("," Param)*
Param           ::= Identifier ":" Type
Type            ::= "number" | "boolean" | "string" | "void"

Statement       ::= IfStmt | WhileStmt | ForStmt | ReturnStmt | Block |
ExprStmt
Block           ::= "{" Declaration* "}"
```

```
IfStmt          ::= "if" "(" Expression ")" Statement ("else" Statement)?
WhileStmt       ::= "while" "(" Expression ")" Statement
ForStmt         ::= "for" "(" ForInit? ";" ForCond? ";" ForStep? ")"
Statement
ForInit         ::= VarDecl | Expression
ForCond         ::= Expression
ForStep         ::= Expression
ReturnStmt      ::= "return" Expression? ";"
ExprStmt        ::= Expression ";"
```

### 2.4 표현식 우선순위

```
Expression      ::= Assignment
Assignment      ::= Identifier "=" Assignment | LogicalOr
LogicalOr       ::= LogicalAnd ("||" LogicalAnd)*
LogicalAnd      ::= Equality ("&&" Equality)*
Equality        ::= Comparison (("==" | "===" | "!=" | "!==") Comparison)*
Comparison      ::= Term (("<" | "<=" | ">" | ">=") Term)*
Term            ::= Factor (("+" | "-") Factor)*
Factor          ::= Unary (("*" | "/") Unary)*
Unary           ::= ("!" | "-") Unary | Call
Call            ::= Primary ("(" Arguments? ")")*
Arguments       ::= Expression ("," Expression)*
Primary         ::= NumberLiteral
                  | StringLiteral
                  | "true" | "false"
                  | Identifier
                  | "(" Expression ")"
```

- 함수 호출은 후위로 연속 적용 가능(예: `foo()(1, 2)`), 파서는 이를 중첩 `CallExpr`로 만듭니다.
- `for` 문은 파서 단계에서 `while` + 블록으로 디소거됩니다(조건이 없으면 `true`, step은 while 블록의 마지막에 추가).

### 2.5 타입 규칙 요약

- `+`: `(number, number) → number`, `(string, string) → string`, 그 외 오류.
- `-`, `*`, `/`: 피연산자 둘 다 `number` 이어야 합니다.
- 논리 연산 `&&`, `||`: `boolean`만 허용.
- 비교 `< <= > >=`: `number`만 허용, 결과는 `boolean`.
- 동등 `== !=`: `void`를 제외한 타입끼리만 비교 가능, 결과는 `boolean`.
- 엄격 동등 `=== !==`: 좌우 타입이 동일해야 하며, 결과는 `boolean`.
- 대입: 대상 변수의 선언 타입과 동일해야 하며, 미선언 변수 대입은 오류입니다.
- 변수/매개변수 이름은 어떤 스코프에서도 중복 선언 불가(섀도잉 금지).
- `return`은 함수 선언의 반환 타입과 일치해야 하며, `void` 함수는 값을 반환할 수 없습니다.
- 함수는 전역에서만 선언하며, 시그니처는 먼저 수집되어 순서와 관계없이 호출할 수 있습니다.

### 2.6 스코프 규칙

- 전역 스코프 1개에서 시작하며, 블록(`{ ... }`)과 함수 본문 진입 시 새 스코프를 생성합니다.
- 변수를 선언하면 **현재 스코프뿐 아니라 모든 상위 스코프와도 이름이 중복될 수 없습니다**(섀도잉 금지). 이미 존재하는 이름을 다시 선언하면 오류.
- 변수 참조 시 현재 스코프에서 찾고 없으면 바깥 스코프를 따라 올라갑니다. 끝까지 없으면 `Undeclared variable` 오류.
- 함수는 전역에만 선언 가능합니다. 모든 함수 시그니처를 먼저 수집하므로 정의 순서와 관계없이 호출할 수 있습니다.

### 2.7 오류 메시지 규칙(대표 예시)

- 모든 오류는 `Error` 예외로 throw 되며 CLI에서 그대로 출력됩니다.
- 렉서
  - 알 수 없는 문자: `Unexpected character '<ch>' at line X, column Y`
  - 닫히지 않은 주석/문자열: `Unterminated block comment ...`, `Unterminated string literal ...`
- 파서(`parser.ts`)
  - 공통 프리픽스: `[ParseError] ...`
  - 예: `Expected ';' after expression at line X, column Y. Got: ...`
  - 잘못된 대입 대상: `[ParseError] Invalid assignment target at line X, column Y`
- 타입체커(`typechecker.ts`)
  - 공통 프리픽스: `[TypeError] ...`
  - 예: `Variable 'x' already declared`(스코프 섀도잉 금지), `Undeclared variable 'x'`, `Function must return 'number', got 'void'`, `Operator '+' requires (number, number) or (string, string), got 'number' and 'string'`
- CLI(`src/bin/minitsc.ts`)는 예외 메시지를 출력하고 종료 코드 1로 종료합니다.

## 3. 전체 구조 및 처리 흐름

```
source.minits
  ↓ Lexer (src/lexer/lexer.ts) : 문자열 → 토큰 리스트
  ↓ Parser (src/parser/parser.ts) : 토큰 → AST(Program)
      - for 문을 while 로 디소거
  ↓ TypeChecker (src/semantic/typechecker.ts) : AST 정적 검증
      - VarSymbolTable로 스코프 관리, 함수 시그니처 선등록
  ↓ CodeGenerator (src/codegen/codegen.ts) : AST → JavaScript 문자열
      - 타입 정보는 제거, 제어문/함수/대입을 JS로 직렬화
  ↓ 실행은 Node.js 등 JS 런타임에서 수행
```

- 중간 IR을 따로 두지 않고 AST를 그대로 타입검사 후 코드 생성에 사용합니다.
- CLI(`src/bin/minitsc.ts`)는 위 파이프라인을 연결해 파일을 받아 JS로 저장합니다.

### 3.1 샘플 파이프라인 출력(함수 예제)

테스트 소스:

```
function add(a: number, b: number): number {
  let result: number = a + b;
  return result;
```

```
}
let x: number = add(1, 2);
```

- Lexer 출력(토큰, 실제 로그 기반):


```
[
  { "lexeme": "function", "typeName": "Function", "line": 1, "column": 1
},
  { "lexeme": "add", "typeName": "Identifier", "line": 1, "column": 10 },
  { "lexeme": "(", "typeName": "LParen", "line": 1, "column": 13 },
  { "lexeme": "a", "typeName": "Identifier", "line": 1, "column": 14 },
  { "lexeme": ":", "typeName": "Colon", "line": 1, "column": 15 },
  { "lexeme": "number", "typeName": "NumberType", "line": 1, "column": 17
},
  { "lexeme": ",", "typeName": "Comma", "line": 1, "column": 23 },
  { "lexeme": "b", "typeName": "Identifier", "line": 1, "column": 25 },
  { "lexeme": ":", "typeName": "Colon", "line": 1, "column": 26 },
  { "lexeme": "number", "typeName": "NumberType", "line": 1, "column": 28
},
  { "lexeme": ")", "typeName": "RParen", "line": 1, "column": 34 },
  { "lexeme": ":", "typeName": "Colon", "line": 1, "column": 35 },
  { "lexeme": "number", "typeName": "NumberType", "line": 1, "column": 37
},
  { "lexeme": "{", "typeName": "LBrace", "line": 1, "column": 44 },
  { "lexeme": "let", "typeName": "Let", "line": 2, "column": 3 },
  { "lexeme": "result", "typeName": "Identifier", "line": 2, "column": 7
},
  { "lexeme": ":", "typeName": "Colon", "line": 2, "column": 13 },
  { "lexeme": "number", "typeName": "NumberType", "line": 2, "column": 15
},
  { "lexeme": "=", "typeName": "Equal", "line": 2, "column": 22 },
  { "lexeme": "a", "typeName": "Identifier", "line": 2, "column": 24 },
  { "lexeme": "+", "typeName": "Plus", "line": 2, "column": 26 },
  { "lexeme": "b", "typeName": "Identifier", "line": 2, "column": 28 },
  { "lexeme": ";", "typeName": "Semicolon", "line": 2, "column": 29 },
  { "lexeme": "return", "typeName": "Return", "line": 3, "column": 3 },
  { "lexeme": "result", "typeName": "Identifier", "line": 3, "column": 10
},
  { "lexeme": ";", "typeName": "Semicolon", "line": 3, "column": 16 },
  { "lexeme": "}", "typeName": "RBrace", "line": 4, "column": 1 },
  { "lexeme": "let", "typeName": "Let", "line": 5, "column": 1 },
  { "lexeme": "x", "typeName": "Identifier", "line": 5, "column": 5 },
  { "lexeme": ":", "typeName": "Colon", "line": 5, "column": 6 },
  { "lexeme": "number", "typeName": "NumberType", "line": 5, "column": 8
},
  { "lexeme": "=", "typeName": "Equal", "line": 5, "column": 15 },
  { "lexeme": "add", "typeName": "Identifier", "line": 5, "column": 17 },
  { "lexeme": "(", "typeName": "LParen", "line": 5, "column": 20 },
  { "lexeme": "1", "typeName": "NumberLiteral", "line": 5, "column": 21 },
  { "lexeme": ",", "typeName": "Comma", "line": 5, "column": 22 },
  { "lexeme": "2", "typeName": "NumberLiteral", "line": 5, "column": 24 },
  { "lexeme": ")", "typeName": "RParen", "line": 5, "column": 25 },
```

```
  { "lexeme": ";", "typeName": "Semicolon", "line": 5, "column": 26 },
  { "lexeme": "", "typeName": "EOF", "line": 6, "column": 1 }
]
```

- Parser 출력(AST JSON):

```
{
  "kind": "Program",
  "body": [
    {
      "kind": "FunctionDecl",
      "name": "add",
      "params": [
        { "name": "a", "paramType": "number" },
        { "name": "b", "paramType": "number" }
      ],
      "returnType": "number",
      "body": {
        "kind": "BlockStmt",
        "body": [
          {
            "kind": "VarDecl",
            "name": "result",
            "varType": "number",
            "init": {
              "kind": "BinaryExpr",
              "operator": "+",
              "left": { "kind": "IdentifierExpr", "name": "a" },
              "right": { "kind": "IdentifierExpr", "name": "b" }
            }
          },
          {
            "kind": "ReturnStmt",
            "argument": { "kind": "IdentifierExpr", "name": "result" }
          }
        ]
      }
    },
    {
      "kind": "VarDecl",
      "name": "x",
      "varType": "number",
      "init": {
        "kind": "CallExpr",
        "callee": { "kind": "IdentifierExpr", "name": "add" },
        "args": [
          { "kind": "NumberLiteralExpr", "value": 1 },
          { "kind": "NumberLiteralExpr", "value": 2 }
        ]
      }
    }
```

```
  ]
}
```

- CodeGenerator 출력(JavaScript):

```
function add(a, b) {
  let result = (a + b);
  return result;
}
let x = add(1, 2);
```

### 3.2 오류 사례(단계별)

| 단계 | 실패 입력(발췌) | 실제 메시지 예시 |
|---|---|---|
| 렉서 | `let x = 1$;` | `Unexpected character '$' at line 1, column 10` |
| 파서 | `let x: number = 1` | `[ParseError] Expected ';' after variable declaration at line 1, column 18. Got: EOF ()` |
| 타입 체커 | `let x: number = "a";` | `[TypeError] Variable 'x' expected type 'number' but got 'string'` |

## 4. 구현된 기능

- **Lexing (토큰화)**
  - 토큰 종류: 키워드(`let`, `function`, `if`, `else`, `while`, `for`, `return`, `true`, `false`), 타입 키워드(`number`, `boolean`, `string`, `void`), 식별자, 숫자 리터럴(정수·실수), 문자열 리터럴, 연산자/구분자(`+ - * / ! && || == != === !== < <= > >= = ( ) { } : , ;`).
  - 동작: 공백/주석(`//`, `/* */`) 스킵, 문자열 이스케이프(`\n`, `\t`, `\"`, `\\`) 처리, 잘못된 문자에 명시적 오류.
- **Parsing·AST**
  - 선언/구문: `let` 변수 선언(타입 생략 시 초기값으로 추론, 초기값 없으면 오류), 전역 함수 선언(매개변수·반환 타입 필수), `if/else`, `while`, `for`(파서에서 while+블록으로 디소거), `return`, 중첩 블록.
  - 표현식: 단항 `!`, `-`; 이항 산술/비교/논리/동등; 함수 호출 체이닝; 괄호식; 대입 표현식.
- **Type checking**
  - 스코프/심볼: 함수 시그니처 선수집, VarSymbolTable로 블록 스코프 관리, 섀도잉 금지, 미선언 변수 감지.
  - 규칙 적용: 변수 타입 추론/검증, 연산자별 타입 제약, 조건식은 boolean 강제, 반환 타입 일치, 함수 호출 인자 수/타입 검사, 오류 시 `[TypeError] ...` 메시지.
- **Code generation**
  - 타입 정보 제거 후 JavaScript 직렬화: 함수/블록/제어문/대입/함수 호출 모두 JS로 변환.
  - for 디소거 결과를 while 기반으로 출력, 연산자에 괄호를 걸어 우선순위 보존, 문자열은 `JSON.stringify`로 안전 출력.
- **CLI·빌드/테스트**
  - `minitsc <input.minits> [-o output.js]`로 파일 단위 컴파일, 실패 시 메시지 출력 후 종료 코드 1.

    - `npm run build`로 dist 생성, `npm run examples`로 `examples/` 전부 컴파일, `npm test`로 Vitest 단위 테스트 실행.

## 5. 미구현/제한 사항

- 객체, 배열, 튜플, 클래스, 제네릭, 유니언 등 고급 타입 부재.
- 모듈/임포트, 네임스페이스, 인터페이스 미지원.
- 함수 표현식/익명 함수, 고차 함수 없음(전역 함수 선언만 허용).
- 제어문 키워드 `break`/`continue` 미지원.
- 변수 섀도잉 금지, `const`나 재할당 불가능 변수 제공 안 함.
- 문자열은 쌍따옴표만 지원하며 템플릿 리터럴 없음.
- 런타임 라이브러리 없음: JS 코드 실행·입출력은 생성된 JS와 호스트 환경에 의존합니다.