MiniMoonBit 2024 程序设计语言规范、文法及说明

by MiniMoonBit Authors

目录

§0.1. 更新日志

- 2024-10-16 – 根据选手反馈，修正了一部分语法语义和实现的说明
- 2024-09-18 – 增加了对部分语法到 AST 映射的说明
- 2024-09-09 – 修复了一些语法定义中的错误，补充关于评测机制的描述
- 2024-09-06 – 修复语法定义中的错误，增加了对关键语义的提示
- 2024-09-01 – 初稿

§1. 语法定义

MiniMoonBit 的语法定义如下，以 ANTLR 语法编写。 你也可以在 我们的仓库里 找到同样的内容。

```
grammar MiniMoonBit;

prog: top_level* EOF;

// Top-level
//
// Top level declarations should start at the beginning of the line, i.e.
```

```
// token.column == 0. Since this is non-context-free, it is not included in this
// backend-agnostic ANTLR grammar.
top_level: top_let_decl | toplevel_fn_decl;
top_let_decl:
  'let' IDENTIFIER ':' type '=' expr ';';
toplevel_fn_decl: (main_fn_decl | top_fn_decl) ';';

// Function declarations
//
// `fn main` and `fn init` does not accept parameters and return type
main_fn_decl: 'fn' ('main' | 'init') fn_body;

top_fn_decl:
  'fn' IDENTIFIER '(' param_list? ')' '->' type fn_body;
param_list: param (',' param)*;
param: IDENTIFIER type_annotation;
fn_body: '{' stmt '}';

nontop_fn_decl:
  'fn' IDENTIFIER '(' nontop_param_list? ')' (
    '->' type
  )? fn_body;
nontop_param_list:
  nontop_param (',' nontop_param)*;
nontop_param: IDENTIFIER type_annotation?;

// Statements
stmt:
  let_tuple_stmt
  | let_stmt
  | fn_decl_stmt
  | assign_stmt
  | expr_stmt;

let_tuple_stmt:
  'let' '(' IDENTIFIER (',' IDENTIFIER)* ')' type_annotation? '=' expr ';'
    stmt;
let_stmt:
  'let' IDENTIFIER type_annotation? '=' expr ';' stmt;
type_annotation: COLON type;

fn_decl_stmt: nontop_fn_decl ';' stmt;

// x[y] = z;
assign_stmt: get_expr '=' expr ';' stmt;
get_expr: get_or_apply_level_expr '[' expr ']';

expr_stmt: expr;

// Expressions, in order of precedence.
expr: // not associative
  add_sub_level_expr '==' add_sub_level_expr
  | add_sub_level_expr '<=' add_sub_level_expr
  | add_sub_level_expr;

add_sub_level_expr: // left associative
  add_sub_level_expr '+' mul_div_level_expr
  | add_sub_level_expr '-' mul_div_level_expr
  | mul_div_level_expr;
```

```
mul_div_level_expr: // left associative
  mul_div_level_expr '*' if_level_expr
  | mul_div_level_expr '/' if_level_expr
  | if_level_expr;

if_level_expr: get_or_apply_level_expr | if_expr;
if_expr: 'if' expr block_expr ('else' block_expr)?;

get_or_apply_level_expr:
  value_expr # value_expr_
  // x[y]
  | get_or_apply_level_expr '[' expr ']' # get_expr_
  // f(x, y)
  | get_or_apply_level_expr '(' (expr (',' expr)*)? ')' # apply_expr;

// Value expressions
value_expr:
  unit_expr
  | group_expr
  | tuple_expr
  | bool_expr
  | identifier_expr
  | block_expr
  | neg_expr
  | floating_point_expr
  | int_expr
  | not_expr
  | array_make_expr;
unit_expr: '(' ')'; // ()
group_expr: '(' expr ')'; // (x)
tuple_expr:
  '(' expr (',' expr)+ ')'; // (x, y); 1-tuple is not allowed
block_expr: '{' stmt '}'; // { blah; blah; }
bool_expr: 'true' | 'false';
neg_expr: '-' value_expr;
floating_point_expr: NUMBER '.' NUMBER?; // 1.0 | 1.
int_expr: NUMBER; // 1
not_expr: 'not' '(' expr ')'; // not(x)
array_make_expr:
  'Array' ':' ':' 'make' '(' expr ',' expr ')'; // Array::make(x, y)
identifier_expr: IDENTIFIER;

// Types
type:
  'Unit'
  | 'Bool'
  | 'Int'
  | 'Double'
  | array_type
  | tuple_type
  | function_type;
array_type: 'Array' '[' type ']';
tuple_type: '(' type (',' type)* ')'; // (Int, Bool)
function_type:
  '(' type (',' type)* ')' '->' type; // (Int, Bool) -> Int

// Tokens
```

```
TRUE: 'true';
FALSE: 'false';
UNIT: 'Unit';
BOOL: 'Bool';
INT: 'Int';
DOUBLE: 'Double';
ARRAY: 'Array';
NOT: 'not';
IF: 'if';
ELSE: 'else';
FN: 'fn';
LET: 'let';
NUMBER: [0-9]+;
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*;
DOT: '.';
ADD: '+';
SUB: '-';
MUL: '*';
DIV: '/';
ASSIGN: '=';
EQ: '==';
LE: '<=';
LPAREN: '(';
RPAREN: ')';
LBRACKET: '[';
RBRACKET: ']';
LCURLYBRACKET: '{';
RCURLYBRACKET: '}';
ARROW: '->';
COLON: ':';
SEMICOLON: ';';
COMMA: ',';
WS: [ \t\r\n]+ -> skip;
COMMENT: '//' ~[\r\n]* -> skip;
```

§1.1. 映射到预定义的 AST 数据结构

ANTLR 语法定义的绝大部分都可以直接与模板中定义的语法树数据结构一一对应。 但是，有一些语法部分因为实现较为特殊，需要进行进一步的映射：

§1.1.1. 顶层数据结构

顶层定义 `prog: top_level* EOF;` 在 AST 上实际与非顶级的 `stmt` 相同，并且需要在结尾补上一个 Unit.

```
let foo: Int = 1;
fn bar() -> Int {
  1
}

fn main {
  // ...
}
```

对应的 AST 结构为：

```
Let(name: ("foo", Int), value: /* 1 */,
  LetRec(name: ("bar", Fn([], Int)), body: /* 1 */,
```

```
    LetRec(name: ("main", Fn([], Unit)), body: /* … */,
      Unit)))
```

§1.1.2. 类型的使用

函数声明中 `name: (String, Type)` 中的 `Type` 是函数整体的类型，而不是返回值类型。例如：

```
fn foo() → Int {
  1
}
```

对应的 AST 结构为：

```
LetRec(name: ("foo", Fn([], Int)), args: [], body: /* 1 */)
```

当某个变量或者函数没有标注类型时，其类型可以在 AST 中标为 `Var({val: None})`。

§1.1.3. 表达式

1-tuple 不合法，`(expr)` 是括号表达式。

§2. 预定义的函数

你的程序的主函数应当声明为 `minimbt_main`，遵循标准 C 调用约定，不接收任何参数，也不返回任何内容。你应当在汇编中将其声明为全局符号（`.global minimbt_main`）。

运行环境中需要预先定义以下辅助函数：

```
// 输入输出函数
/// 读取一个整数，如果失败返回 INT_MIN
fn read_int() → Int;
/// 打印一个整数，不带换行
fn print_int(i: Int) → Unit;
/// 读取一个字节，如果失败返回 -1
fn read_char() → Int;
/// 打印一个字节
fn print_char(c: Int) → Unit;
/// 打印一个换行
fn print_endline() → Unit;

// 数学函数
/// 整数和浮点数的互相转换
fn int_of_float(f: Double) → Int;
fn float_of_int(i: Int) → Double;
fn truncate(f: Double) → Int;  // 与 int_of_float 相同
/// 浮点数运算
fn floor(f: Double) → Double;
fn abs_float(f: Double) → Double;
fn sqrt(f: Double) → Double;
fn sin(f: Double) → Double;
fn cos(f: Double) → Double;
fn atan(f: Double) → Double;
```

我们会以标准 RISC-V C 调用约定在提供这些函数的实现，实现的名称为实际函数名称前加入 `minimbt_`，如 `minimbt_print_int`。 在实现时，你可以认为所有不在作用域中的函数名称都是外部函数，并在函数名前加入 `minimbt_` 转换为外部调用。

此外，为了实现闭包、数组、元组等特性，我们还提供了以下内存分配函数：

```
/// 内存分配函数
void* minimbt_malloc(int32_t size);
/// 分配对应大小的内存，并初始化所有元素为给定的值
int32_t* minimbt_create_array(int32_t n_elements, int32_t init);
double* minimbt_create_float_array(int32_t n_elements, double init);
void** minimbt_create_ptr_array(int32_t n_elements, void* init);
```

你将不需要释放内存。

§2.1. WASM 后端

在 WASM 后端中，你需要将主函数放置于 WASM 文件的 start section 中。

你将可以使用与 RISC-V 后端同样的辅助函数定义。这些函数定义将被声明在 `moonbit` 模块中。

```
(memory (import "moonbit" "memory") 10)
(func (import "moonbit" "minimbt_read_int") () (result i32))
(func (import "moonbit" "minimbt_print_int") (param i32))
(func (import "moonbit" "minimbt_read_char") () (result i32))
(func (import "moonbit" "minimbt_print_char") (param i32))
(func (import "moonbit" "minimbt_print_endline"))
```

```
(func (import "moonbit" "minimbt_int_of_float") (param f64) (result i32))
(func (import "moonbit" "minimbt_float_of_int") (param i32) (result f64))
(func (import "moonbit" "minimbt_truncate") (param f64) (result i32))
(func (import "moonbit" "minimbt_floor") (param f64) (result f64))
(func (import "moonbit" "minimbt_abs_float") (param f64) (result f64))
(func (import "moonbit" "minimbt_sqrt") (param f64) (result f64))
(func (import "moonbit" "minimbt_sin") (param f64) (result f64))
(func (import "moonbit" "minimbt_cos") (param f64) (result f64))
(func (import "moonbit" "minimbt_tan") (param f64) (result f64))

;; For Wasm 1 only
(func (import "moonbit" "minimbt_malloc" (param i32)))
(func (import "moonbit" "minimbt_create_array") (param i32) (param i32) (result i32))
(func (import "moonbit" "minimbt_create_float_array") (param i32) (param f64) (result
i32))
(func (import "moonbit" "minimbt_create_ptr_array") (param i32) (param i32) (result i32))
```

§2.2. JS 后端

在 JS 后端中，你需要提供一个 ES Module 模块，其中 `export default` 是你的程序的主函数。我们将通过 `globalThis` 将所有的辅助函数定义注入到你的程序的全局作用域中。

我们将不会提供 `minimbt_malloc` 函数，其他三个生成数组的内存分配函数将均返回给定大小的普通 JS 数组。

§3. 语义

§3.1. 语义

MiniMoonBit 遵循 MoonBit 的语义规则。由于编写完整的形式语义规则的复杂度可能较高，且不一定便于各位选手理解，我们并未计划在此处提供完整的形式语义规则。

在语义中值得注意的点如下：

§3.1.1. Expression

- MiniMoonBit 中没有隐式类型转换，算术表达式的两侧表达式的类型必须相同。
- MiniMoonBit 中没有可变变量。只有数组元素可以被修改。
- `Array::make(n, k)`（`array_make_expr`）会创建一个长度为 `n` 的数组，数组每个元素的值都是 `k`。你可以用上文提到的 `minimbt_create_{,float_,ptr_}array` 函数来实现。
- `if` 语句的两个条件分支都需要返回相同的类型。如果你没有写 else 分支，其类型为 `Unit`。

§3.1.2. Statement

- 函数需要先声明、再使用。声明的函数（除了 `main` 和 `init` 外）可以在函数体内调用自身。
- 所有定义的变量名、函数名都可以覆盖之前的定义。

  例如，`let a = 1; let a = 2; a` 中，最后的 `a` 的值是 `2`。
- 所有的 `stmt` 都会以 `expr_stmt` 结尾。该 `stmt` 的值就是 `expr_stmt` 的值。

  例如，`let a = 1; let b = 2; a + b` 的值是 `3`。
- 提示：在 MiniMoonBit 中实现形如其他语言中

  ```
  foo();
  bar();
  ```

  的语句序列的正确方式是

  ```
  let _ = foo();
  let _ = bar();
  ()
  ```

§3.2. Typing

MiniMoonBit 的类型规则如下：

§3.2.1. Value expressions

$$\frac{}{\vdash \texttt{()} : \texttt{Unit}} \quad \text{(T-Unit)}$$

$$\frac{}{\vdash \texttt{true} : \texttt{Bool}} \quad \text{(T-Bool-true)}$$

$$\frac{}{\vdash \texttt{false} : \texttt{Bool}} \quad \text{(T-Bool-false)}$$

$$\frac{}{\vdash \text{int\_expr}(\_) : \texttt{Int}} \quad \text{(T-Int)}$$

$$\frac{}{\vdash \text{float\_expr}(\_) : \texttt{Double}} \quad \text{(T-Double)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma \vdash -e_1 : \texttt{T}} \quad \text{(T-Neg)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Bool}}{\Gamma \vdash \texttt{not}(e_1) : \texttt{Bool}} \quad \text{(T-Not)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T}, n : \texttt{Int}}{\Gamma \vdash \texttt{Array::make}(n, e_1) : \texttt{Array}[\texttt{T}]} \quad \text{(T-Array-make)}$$

§3.2.2. Arithmetics

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma \vdash e_1 + e_2 : \texttt{T}} \quad \text{(T-Add)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma \vdash e_1 - e_2 : \texttt{T}} \quad \text{(T-Sub)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma \vdash e_1 * e_2 : \texttt{T}} \quad \text{(T-Mul)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}\}}{\Gamma \vdash e_1 / e_2 : \texttt{T}} \quad \text{(T-Div)}$$

§3.2.3. Comparisons

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}, \texttt{Bool}\}}{\Gamma \vdash e_1 \ \texttt{<=} \ e_2 : \texttt{Bool}} \quad \text{(T-Leq)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T}, e_2 : \texttt{T} \text{ where } \texttt{T} \in \{\texttt{Int}, \texttt{Double}, \texttt{Bool}\}}{\Gamma \vdash e_1 \ \texttt{==} \ e_2 : \texttt{Bool}} \quad \text{(T-Eq)}$$

§3.2.4. Get and apply

$$\frac{\Gamma \vdash e_1 : \texttt{Array}[\texttt{T}], e_2 : \texttt{Int}}{\Gamma \vdash e_1[e_2] : \texttt{T}} \quad \text{(T-Array-get)}$$

$$\frac{\Gamma \vdash f : (\texttt{T}_1, ...\texttt{T}_n) \to \texttt{T}, e_1 : \texttt{T}_1, ..., e_n : \texttt{T}_n}{\Gamma \vdash f(e_1, ..., e_n) : \texttt{T}} \quad \text{(T-Apply)}$$

§3.2.5. If

$$\frac{\Gamma \vdash e_1 : \texttt{Bool}, e_2 : \texttt{T}, e_3 : \texttt{T}}{\Gamma \vdash \texttt{if } e_1 \ \{e_2\} \ \texttt{else} \ \{e_3\} : \texttt{T}} \quad \text{(T-If)}$$

§3.2.6. Statements

$$\frac{\Gamma \vdash e_1 : \texttt{Array}[\texttt{T}], e_2 : \texttt{Int}, e_3 : \texttt{T}, s_1 : \texttt{T}_s}{\Gamma \vdash e_1[e_2] = e_3; s_1 : \texttt{T}_s} \quad \text{(T-Array-put)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{T} \qquad \Gamma, v : \texttt{T} \vdash s_1 : \texttt{T}_s}{\Gamma \vdash \texttt{let}\ v = e_1; s_1 : \texttt{T}_s} \quad \text{(T-Let-var-untyped)}$$

$$\frac{\Gamma \vdash e : \texttt{T} \qquad \Gamma, v : \texttt{T} \vdash s_1 : \texttt{T}_s}{\Gamma \vdash \texttt{let}\ v : \texttt{T} = e; s_1 : \texttt{T}_s} \quad \text{(T-Let-var-typed)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Tuple}[\texttt{T}_1, ..., \texttt{T}_n] \qquad \Gamma, v_1 : \texttt{T}_1, ..., v_n : \texttt{T}_n, \vdash s_1 : \texttt{T}_s}{\Gamma \vdash \texttt{let}(v_1, ..., v_n) = e_1; s_1 : \texttt{T}_s} \quad \text{(T-Let-tuple-untyped)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{Tuple}[\texttt{T}_1, ..., \texttt{T}_n] \qquad \Gamma, v_1 : \texttt{T}_1, ..., v_n : \texttt{T}_n, \vdash s_1 : \texttt{T}_s}{\Gamma \vdash \texttt{let}(v_1, ..., v_n) : (\texttt{T}_1, ..., \texttt{T}_n) = e_1; s_1 : \texttt{T}_s} \quad \text{(T-Let-tuple-typed)}$$

§3.2.7. Functions

Functions' parameters may not have types annotated, so type inference is needed. Fresh type variables are marked as `X`, and you can refer to Chapter 22 in TAPL on how to infer types.

$$\frac{\Gamma, ..., p_i : \texttt{T}_i, ... \vdash s_1 : \texttt{T}_r}{\Gamma \vdash \texttt{fn}\ f(..., p_i : \texttt{T}_i, ...)\{s_1\} : (..., \texttt{T}_i, ...) \to \texttt{T}_r} \quad \text{(T-Fn-param-typed)}$$

$$\frac{\Gamma, ..., p_i : \texttt{X}, ... \vdash s_1 : \texttt{T}_r}{\Gamma \vdash \texttt{fn}\ f(..., p_i, ...)\{s_1\} : (..., \texttt{X}, ...) \to \texttt{T}_r} \quad \text{(T-Fn-param-untyped)}$$

$$\frac{\Gamma \vdash s_1 : \texttt{Unit}}{\Gamma \vdash \texttt{fn main}\ \{s_1\} : () \to \texttt{Unit}} \quad \text{(T-Fn-main)}$$

$$\frac{\Gamma \vdash s_1 : \texttt{Unit}}{\Gamma \vdash \texttt{fn init}\ \{s_1\} : () \to \texttt{Unit}} \quad \text{(T-Fn-init)}$$