`minipy` language specification

Rishav Kundu 2019121007

September 5, 2020

1 Overview

`minipy` is a simple language with a Python-like syntax, making use of indentation for the purpose of demarcating blocks. A simple program demonstrating minipy is shown below:

```
def add(int8 a, int8 b) -> int8:
    return a + b
print(add(2, 3))
```

Some salient features of `minipy` are: (1 and 2D) arrays, conditionals, loops, polymorphic functions, recursion and IO routines.

2 Semantics

2.1 Statements

Programs in minipy consist of zero or more ”simple statement”s. Each simple statement is either a declaration, an assignment or a function call.

These simple statements can be interspersed by blocks. We have a number of blocks available: `for`, `while`, `if-elif-else` and `def`. These blocks can contain more simple statements, or nested blocks. There is no semantic limit on the nesting depth.

When the word statement is used, it implies that both simple statements and blocks are permitted.

2.2 Data Types

`minipy` offers the following ”basic” data types: `int8`, `uint8`, `int32`, `uint32`, `int32`, `uint64`, `int64`, `bool`.

Array data types can be constructed by using the following syntax:

`basic_type[expr][expr]`

where expr is an expression that resolves to an integer-like value at compile-time or run-time. Array types are homogenous. To access a value from an array type, use the usual notation: `a[i]` where `i` is a valid index.

Note: Only 1D and 2D arrays are supported at the moment.

2.3 Identifiers and literals

Identifiers are used for naming variables and functions. They must start with a letter, optionally followed by zero or more letters, digits, or underscores. Some valid identifiers are: `foo, foo1, f_o_o_1, f1oo`.

The following keywords are reserved and may not be used as identifiers: `True`, `False`, `None`, `int8`, `uint8`, `int32` ,`uint32`, `int64`, `uint64`, `if`, `elif`, `else`, `for`, `while`, `def`, `return`.

Literals are of three types:

- Boolean literals are either `True` or `False`.
- Numeric literals are of the format:

  `(+ | -)? (0x | 0b | 0o)? [0-9A-Fa-f]+`.

  `0x, 0b, 0o` specify an optional *base* which indicates that the number is to be considered as hexadecimal, binary or octal respectively. Note that in case a base prefix is used, the alphabet for the number is constrained respectively. `50, 0x50, 0b10, 0o77` are all valid numeric literals.
- Array literals are defined like so:
  - 1D arrays: `{1, 2, 3}`
  - 2D arrays: `{{1, 2, 3}, {4, 5, 6}}`

  Note that array literals are not constrained to containing other literals, they may contain expressions (see later section on expressions) that resolve to compatible datatypes.

Also note that array literals must be homogenous, that is, they must contain values that are of the same type. In the case of 2D arrays, this means that subarrays must all be of same type and size.

2.4 Operators

`minipy` offers the usual arithmetic operators `+`, `-`, `*`, `/`, `**` (exponentiation) and `%`. We also have the comparison operators `<`, `>`, `==`, `<=` and `>=` with can be used with the same (numeric) types. For the binary logical operators, we have `and`, `or` and `not` which can be used with `bool` values.

We also have support for ternary conditions. They are of the form:

`expr if expr else epr`

Where the 2nd and 3rd expressions should resolve to bool types.

Operator precendence is as usual (can be viewed in the grammar).

2.5 Expressions

Expressions in `minipy` are constructs that produce values. They correspond loosely to arithmetic or boolean expressions. More precisely, they are of the following form:

```
expr :
    unary expr
    | expr binop expr
    | fn_call
    | literal
    | IDENT
    | IDENT '[' expr ']'
    | IDENT '[' expr ']' '[' expr ']'
    | expr 'if' expr 'else' expr
    | '(' expr ')'
     ;
```

where `unary` and `binop` are unary and binary operators. `fn_call` refers to a function call and `IDENT` refers to an identifier. The following are all valid examples of expressions: `2 + (3 + 5) * 8`, `not False`, `a + b`, `succ(1) + succ(2)`.

Note that we can associate a data type with an expression.

2.6 Assignment and Declarations

Declaration and assignment take the following form:

```
(datatype)? ident = expr
```

The datatype is optional, in which case it is considered to be an assignment operation. The RHS can be omitted in the case of a declaration, in which case the default value for the datatype is used (0 initialized, False).

2.7 Conditionals

`minipy` has the usual:

```
if expr:
    statements...
elif expr:
    statements...
else:
    statements...
```

where `expr` are expressions that resolve to booleans. Both `elif` and `else` are optional, with usual semantics. More than one `elif` is permitted.

Note that if blocks **do not** create new scopes. Variables declared inside are visible outside.

2.8 Loops

`minipy` has two loops available:

```
while expr:
    statements...


for (simple_stmt; expr; simple_stmt):
    statements...
```

`expr` should resolve to a boolean value. As with if blocks, loops **do not** create new scopes. Any of the components of a for declaration is allowed to be empty.

2.9 Functions

Functions can be defined and called as so.

```
def add(int8 a, int8 b) -> int8:
    return a + b

c = add(2, 3)
```

The datatype following the `->` denotes the return type. A function may take 0 or more arguments (potentially heterogenous), but return exactly one data type (optionally the special vaue `None` to indicate the lack of a return type.

Note that funtion calls may be used in expressions. The return type of the function **must be** compatible with the other operations used in the expression. For example, `int8 a = foo()` where foo returns a `None` type is **invalid**.

Functions are allowed to call themselves. There is no semantic limit on the recursion depth. Each invocation creates a fresh scope.

Functions are call-by-value. Any data passed into a function is copied and modifications inside the function are not reflected outside.

Functions are *polymorphic*. That is, multiple functions can have the same name as long as they differ in their argument list. Note that polymorphism **does not** consider the function returrn type.

The `return` keyword may be used to exit out of the function. A value to be returned must be specified if the function has a non-`None` return type, and those data types must match. A function with a non-`None` return type must have *at least* one `return` statement on every branch. `return` statements are not allowed on the top-level scope (that is, outside any functions).

2.9.1 Scope and Visibility

Functions create a new scope, that is, any identifiers declared inside of them are not visible outside the function. No other blocks create scopes.

3 Semantic Checks

1. Variables must not be declared twice. However, declarations inside functions are allowed to shadow outer declarations.
2. The RHS of an assignment must be compatible with the datatype.
3. Variables in expressions must be declared beforehand and in-scope.

4. The expressions used for conditionals and loop conditions must be of bool type.
5. Operators must be used on appropriate types.
6. Expressions must typecheck wherever they are used, for example in function invocations.
7. Dimensions for arrays must be strictly positive.
8. If base prefixes are used with numeric literals, they must use the appropriate alphabet.
9. `return` may only be used inside a function.

4 IO Routines

`minipy` has the following functions for IO. All of these read from standard input.

1. `readInt8()`
2. and so on for the other data types.

These return a single value of the given data-type. A runtime error is thrown if the appropriate value cannot be read.

For output, `minipy` provides the `print()` function. It is overloaded and takes exactly one argument of any **basic** type.

5 Grammar

```
prog : stmt+ EOF ;

datatype_base : 'uint32' | 'uint64' | 'int32' | 'int64' | 'int8' | 'uint8' | 'bool';
datatype : datatype_base
         | datatype_base '[' expr']'
         | datatype_base '[' expr ']' '[' expr ']'
         ;


unary: 'not';
binop_exp: '**' ;
binop1: '*' | '/' | '%' ;
binop2: '+' | '-';
```

```
binop3: '<' | '>' | '<=' | '!=' | '>=' | '==' ;
binop4: 'and' | 'or';
assignment_op: '=' ;

array_literal_1d: '{' expr (',' expr)* '}';
array_literal_2d: '{' array_literal_1d (',' array_literal_1d)* '}';
array_literal: array_literal_1d | array_literal_2d;

bool_literal: 'True' | 'False' ;
char_literal: CHAR;

literal: INT_LITERAL | bool_literal | array_literal | char_literal ;

fn_call: IDENT '(' (expr (',' expr)*)? ')' ;


expr :
    unary expr
    |<assoc=right> expr binop_exp expr
    | expr binop1 expr
    | expr binop2 expr
    | expr binop3 expr
    | expr binop4 expr
    | fn_call
    | literal
    | IDENT
    | IDENT '[' expr ']'
    | IDENT '[' expr ']' '[' expr ']'
    | '(' expr ')'
     ;

block: NEWLINE INDENT stmt+ DEDENT;

if_stmt: 'if' expr ':' block ('elif' expr ':' block)* ('else' ':' block )? ;
for_stmt: 'for' '(' simple_stmt ';' expr ';' simple_stmt ')' ':' block ;
while_stmt: 'while' expr ':' block ;

function_decl: 'def' IDENT '(' datatype IDENT (',' datatype IDENT)* ')' '->' (datatype | 'N

return_stmt: 'return' expr ;

decl : datatype IDENT (assignment_op expr)? |
       datatype '[' expr ']' IDENT (assignment_op expr)? |
       datatype '[' expr ']' '[' expr ']' IDENT (assignment_op expr)? ;

assignment : (IDENT |
             IDENT '[' expr ']' |
             IDENT '[' expr ']'  '[' expr ']' ) assignment_op expr;

simple_stmt: decl | assignment | fn_call | return_stmt;

stmt : simple_stmt NEWLINE
    | if_stmt
    | for_stmt
    | while_stmt
    | function_decl
    ;


/*Tokens*/

CHAR: '\''[A-Za-z0-9 ]'\'';
IDENT: [a-zA-Z][a-zA-Z0-9_]*;
INT_LITERAL: [0-9A-Fa-f]+;
WS: ' '->skip;

fragment SPACES
 : [ \t]+
 ;

NEWLINE [...omitted (see note)]
```

Note: The lexer must produce `INDENT` and `DEDENT` tokens when the indentation level changes. This cannot be done using a context-free grammar. I have used some sample code from ANTLR docs to implement these tokens. This requires the use of the ANTLR API to track some internal state in the lexer.

https://github.com/antlr/grammars-v4/blob/master/python/python3/Python3.g4