Assignment A: Abstract Syntax Trees and Type Checking

Vladyslav Shevchuk
Matrikelnummer: k12220092
Email: vladadshev@gmail.com

Formal Semantics of Programming Languages (SS 2025)

23 April 2025

1 Explanation of the Program Implementation

This assignment implements an abstract syntax and type checking system for a simple imperative language with procedures and a unified expression domain. We use Python's data classes to define AST nodes, and we recursively type check expressions and commands using a context-based lookup. Expressions can return either `int` or `bool`, and the type checker raises exceptions if any rule is violated.

2 How to Use

To run the pre-existing tests simply run the *main.py* file.

2.1 To create your own test:

1. **Define a Program:**
   Create your program using the provided AST classes. For example:

```
prog = Seq(
    Assign("x", IntLiteral(5)),
    Assign("y", BinOp("+", Var("x"), IntLiteral(2)))
)
```

2. **Define the Context:**
   Use a Python dictionary to store variable type bindings:

```
context = {}
```

3. **Run the Type Checker:**
   Pass the program and context to the type checker:

```
type_check_cmd(prog, context)
print("Program is well-typed:", context)
```

4. **Handle Errors:**
   Use a try/except block to catch and handle type errors:

```
try:
    type_check_cmd(prog, context)
except TypeError as e:
    print("Type error:", e)
```

3 Grammar and Type Rules

Modified Grammar

```
Expr ::= IntLiteral(n)
       | BoolLiteral(b)
       | Var(name)
       | BinOp(op, Expr, Expr)
       | UnOp(op, Expr)

Cmd  ::= Assign(var, Expr)
       | Seq(Cmd, Cmd)
       | If(Expr, Cmd, Cmd)
       | While(Expr, Cmd)
```

Typing Rules

- `IntLiteral` → `int`
- `BoolLiteral` → `bool`
- `Var(name)` → context lookup
- `UnOp(’-’)` on `int` → `int`
- `UnOp(’not’)` on `bool` → `bool`
- `BinOp(’+|-|*|/’)` → both sides `int` → `int`
- `BinOp(’=’, ’<=’)` → both sides `int` → `bool`
- `BinOp(’and’, ’or’)` → both sides `bool` → `bool`
- `If`, `While` condition must be `bool`

4 Test Case Explanations

- **Test 1:** Valid assignment and arithmetic (`int`). Passes.
- **Test 2:** Boolean `and` in `If`. Passes.
- **Test 3:** Invalid `int and bool`. Raises `TypeError`.
- **Test 4:** Reassigning different type. Allowed unless strict mode enabled.
- **Test 5:** Valid `While` with `<=`. Passes.

5 Source Code

```
from dataclasses import dataclass
from typing import Union

# Expressions
@dataclass
class IntLiteral:
    value: int
```

```python
@dataclass
class BoolLiteral:
    value: bool

@dataclass
class Var:
    name: str

@dataclass
class BinOp:
    op: str
    left: 'Expr'
    right: 'Expr'

@dataclass
class UnOp:
    op: str
    expr: 'Expr'

Expr = Union[IntLiteral, BoolLiteral, Var, BinOp, UnOp]

# Commands
@dataclass
class Assign:
    var: str
    expr: Expr

@dataclass
class Seq:
    first: 'Cmd'
    second: 'Cmd'

@dataclass
class If:
    cond: Expr
    then_branch: 'Cmd'
    else_branch: 'Cmd'

@dataclass
class While:
    cond: Expr
    body: 'Cmd'

Cmd = Union[Assign, Seq, If, While]

# Type checking functions
def type_check_expr(expr: Expr, context: dict) -> str:
    if isinstance(expr, IntLiteral):
        return 'int'
    elif isinstance(expr, BoolLiteral):
        return 'bool'
    elif isinstance(expr, Var):
        return context.get(expr.name, 'undefined')
    elif isinstance(expr, UnOp):
        et = type_check_expr(expr.expr, context)
        if expr.op == 'not':
            if et != 'bool':
                raise TypeError("Expected bool in 'not'")
            return 'bool'
        elif expr.op == '-':
            if et != 'int':
                raise TypeError("Expected int in unary '-'")
            return 'int'
    elif isinstance(expr, BinOp):
        lt = type_check_expr(expr.left, context)
        rt = type_check_expr(expr.right, context)
        if expr.op in {'+', '-', '*', '/'}:
            if lt == rt == 'int':
                return 'int'
            raise TypeError("Arithmetic operations require int")
        elif expr.op in {'=', '<='}:
            if lt == rt == 'int':
                return 'bool'
            raise TypeError("Comparison requires int")
        elif expr.op in {'and', 'or'}:
            if lt == rt == 'bool':
                return 'bool'
            raise TypeError("Logical operations require bool")
    raise NotImplementedError(f"Unknown expr: {expr}")

def type_check_cmd(cmd: Cmd, context: dict):
    if isinstance(cmd, Assign):
        etype = type_check_expr(cmd.expr, context)
        context[cmd.var] = etype
    elif isinstance(cmd, Seq):
        type_check_cmd(cmd.first, context)
        type_check_cmd(cmd.second, context)
    elif isinstance(cmd, If):
        ctype = type_check_expr(cmd.cond, context)
        if ctype != 'bool':
            raise TypeError("Condition in If must be bool")
        type_check_cmd(cmd.then_branch, context)
        type_check_cmd(cmd.else_branch, context)
    elif isinstance(cmd, While):
        ctype = type_check_expr(cmd.cond, context)
        if ctype != 'bool':
            raise TypeError("Condition in While must be bool")
        type_check_cmd(cmd.body, context)
    else:
        raise TypeError(f"Unknown command type: {type(cmd)}")

# TESTING
def run_tests():
    print("Running Type Checker Tests...\n")

    # Test 1: Valid integer assignment and arithmetic
    try:
        prog1 = Seq(
            Assign("a", IntLiteral(5)),
            Assign("b", BinOp("+", Var("a"), IntLiteral(10)))
        )
        context1 = {}
        type_check_cmd(prog1, context1)
        print("Test 1 Passed:", context1)
    except Exception as e:
```

```
    print("Test 1 Failed:", e)

# Test 2: Valid boolean logic and if statement
try:
    prog2 = Seq(
        Assign("flag", BoolLiteral(True)),
        If(
            BinOp("and", Var("flag"), BoolLiteral(False)),
            Assign("result", IntLiteral(1)),
            Assign("result", IntLiteral(0))
        )
    )
    context2 = {}
    type_check_cmd(prog2, context2)
    print("Test 2 Passed:", context2)
except Exception as e:
    print("Test 2 Failed:", e)

# Test 3: Invalid use of integer in boolean operation
try:
    prog3 = Assign("c", BinOp("and", IntLiteral(1), BoolLiteral(
       True)))
    context3 = {}
    type_check_cmd(prog3, context3)
    print("Test 3 Failed: Expected TypeError")
except TypeError as e:
    print("Test 3 Passed:", e)
except Exception as e:
    print("Test 3 Failed with unexpected error:", e)

# Test 4: Incompatible variable reassignment (if enforcing strict
   typing)
try:
    prog4 = Seq(
        Assign("v", IntLiteral(3)),
        Assign("v", BoolLiteral(True))  # Optional stricter
           enforcement
    )
    context4 = {}
    type_check_cmd(prog4, context4)
    print("Test 4 Passed (allowed reassignment):", context4)
except TypeError as e:
    print("Test 4 Passed (caught type error):", e)
except Exception as e:
    print("Test 4 Failed with unexpected error:", e)

# Test 5: Valid while loop with boolean condition
try:
    prog5 = While(
        BinOp("<=", Var("x"), IntLiteral(10)),
        Assign("x", BinOp("+", Var("x"), IntLiteral(1)))
    )
    context5 = {"x": "int"}
    type_check_cmd(prog5, context5)
    print("Test 5 Passed:", context5)
except Exception as e:
    print("Test 5 Failed:", e)
```

```

if __name__ == "__main__":
    run_tests()
```

Listing 1: AST Definitions and Type Checking