Template

v0.0.1

https://github.com/SillyFreak/typst-my-package

Clemens Koza

Abstract

A template for typst packages

Contents

I Introduction

This is a template for typst packages. It provides the `parse()` and `eval()` functions:

lib.typ

```typ
#let parse(expr) = {
  // this is the "interesting" part: calling into the Rust parser
  cbor.decode(_p.parse(cbor.encode(expr)))
}
/// -> int
#let eval(expr, ..vars) = {
  assert(vars.pos().len() == 0)
  let vars = vars.named()

  let inner-eval(expr) = {
    if expr.type == "number" { expr.value }
    else if expr.type == "variable" { vars.at(expr.name) }
    else if expr.type == "binary" {
      let (operator, left, right) = expr
      (left, right) = (inner-eval(left), inner-eval(right))
      if operator == "add" { left + right }
      else if operator == "sub" { left - right }
      else if operator == "mul" { left * right }
      else if operator == "div" { left / right }
      else { panic("unexpected binary operator: " + operator) }
    }
    else { panic("unexpected expression type: " + expr.type) }
  }

  inner-eval(parse(expr))
}
```

Here they are in action:

```typ
$2 * (2 + x) arrow.double.long$ #my-package.parse("2 * (2 + x)")
```

$2 * (2 + x) \Longrightarrow$ (
  type: "binary",
  operator: "mul",
  left: (type: "number", value: 2),
  right: (
    type: "binary",
    operator: "add",
    left: (type: "number", value: 2),
    right: (type: "variable", name: "x"),
  ),
)

```typ
$2 * (2 + x) arrow.double.long^(x=3)$ #my-package.eval("2 * (2 + x)", x: 3)
```

$2 * (2 + x) \overset{x=3}{\Longrightarrow} 10$

II Plugin implementation

This plugin uses LALRPOP to implement a parser for a domain specific language (DSL) in Rust; in particular, the parser is for mathematical expressions based on the LALRPOP tutorial.

Here are some relevant excerpts from the plugin's code:

ast.rs

```rust
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize, Clone, PartialEq)]
#[serde(tag = "type")]
#[serde(rename_all = "kebab-case", rename_all_fields = "kebab-case")]
pub enum Expr<'input> {
    Number {
        value: i32,
    },
    Variable {
        name: &'input str,
    },
    Binary {
        operator: Operator,
        left: Box<Expr<'input>>,
        right: Box<Expr<'input>>,
    },
}

#[derive(Serialize, Deserialize, Clone, Copy, PartialEq)]
#[serde(rename_all = "kebab-case", rename_all_fields = "kebab-case")]
pub enum Operator {
    Mul,
    Div,
    Add,
    Sub,
}
```

parser/mod.rs

```rust
use crate::ast;

lalrpop_util::lalrpop_mod!(
    #[allow(clippy::all)] grammar,
    "/parser/grammar.rs"
);

pub type Error<'a> = ParseError<usize, Token<'a>, &'static str>;
pub type Result<'a, T> = std::result::Result<T, Error<'a>>;

pub fn parse(source: &str) -> Result<ast::Expr<'_>> {
    let parser = grammar::ExprParser::new();
    parser.parse(source)
}
```

grammar.lalrpop

```lalrpop
use crate::ast::{Expr, Operator};

Tier<Op, NextTier>: Expr<'input> = {
    <left: Tier<Op, NextTier>> <operator: Op> <right: NextTier> => {
        let left = Box::new(left);
        let right = Box::new(right);
        Expr::Binary { operator, left, right }
    },
    NextTier
};

pub Expr = Tier<ExprOp, Factor>;

ExprOp: Operator = {
    "+" => Operator::Add,
    "-" => Operator::Sub,
};

Num: i32 = {
    r"[0-9]+" =>? i32::from_str(<>)
        .map_err(|_| ParseError::User {
            error: "number is too big"
        })
};

Var: &'input str = {
    r"[_\p{ID_Start}][_\p{ID_Continue}-]*"
};
```

III Module reference

III.a `my-package`

`parse(expr: str) -> dict`

Parses an expression via a WASM plugin.

```typ
#my-package.parse("1") \
#my-package.parse("1 + 1") \
#my-package.parse("foo") \
#my-package.parse("foo + 1")
```

```
(type: "number", value: 1)
(
  type: "binary",
  operator: "add",
  left: (type: "number", value: 1),
  right: (type: "number", value: 1),
)
(type: "variable", name: "foo")
(
  type: "binary",
  operator: "add",
  left: (type: "variable", name: "foo"),
  right: (type: "number", value: 1),
)
```

Parameters:

`expr` (`str`) – the expression to parse

`eval(expr: str, ..vars: arguments) -> int`

Evaluates an expression in Typst, by traversing the abstract syntax tree (AST) created in Rust.

```typ
#my-package.eval("1") \
#my-package.eval("1 + 1") \
#my-package.eval("foo", foo: 1) \
#my-package.eval("foo + 1", foo: 1)
```

```
1
2
1
2
```

Parameters:

`expr` (`str`) – the expression to evaluate

`..vars` (`arguments`) – the variable assignments in the expression