CS 1720 Programming Languages

Fall 2023, Final Project - A Tiny Programming Language

Synopsis

Your final project is to implement a minimal structured imperative programming language similar to C.

```
func factorial_recursion(n)
[
    if n < 2 [
        return 1;
    ] else [
        return n * factorial_recursion(n-1);
    ]
]

func factorial_loop(n)
[
    let p;
    p = n;
    while n > 0 [
        n = n - 1;
        p = p * n;
    ]
    return p;
]

func main()
[
    let n;
    n = 5;
    print factorial_loop(n);
    print factorial_recursion(n);
]
```

1 A Structured Imperative Language

Your minimalistic language should support the following: functions and recursion; booleans and ints, basic logical (`! & |`), relational (`== < >`), and arithmetic operations (`+ - * /`); a `let` statement; an assignment statement; `if-then-else`, `while`, and `return` control flow statements; and, a `print` statement.

2 Parsing

In the parsing phase, you should apply three different methods: finite state machines (FSMs), recursive descent, and Pratt expression parsing.

Grammar

Write down a grammar for your language in some EBNF form in a file `Grammar.pdf`. Use comments to structure your rule set. You may use color coding to differentiate between different types of symbols (terminals, non-terminals, meta operators). The grammar will most likely change and evolve while you program.

Lexical Analysis

Implement the lexer code as a FSM by hand. Optionally, attach file position information to each token, which can be used in later phases for error localization.

Syntactical Analysis

Implement the main parser using recursive descent. Use a Pratt parser for expression parsing. Optionally, add error localization and descriptions.

3 Semantical Analysis

Implement a semantical analysis phase that checks if variables have been declared with `let` before being used. Optionally, you can implement function arity checking, some basic type checking, or a constant folding optimization.

4 Execution

Implement an execution engine that takes an analyzed ADT of your language and executes it. You find some example code in `PL_F23_Executing_231116.zip`. The execution engine should be able to execute small example programs such as the one given earlier.

5 Command Line Interface

Write a tiny command line tool (use your languages name) that takes a file in your language and executes it. Optionally, add flags to run only certain phases of the translation process (tokenize, parse, analyze).

```
> tpl -te hello.tpl
TPL (Tiny Programming Language) Version 0.4.2
command(s):
  tokenize (-t --tokenize)
  execute (-e --execute)

FUNC ID("main") PAREN_L PAREN_R BRACKET_L
PRINT LIT_STR("Hello, TPL!") SEMICOLON BRACKET_R

Hello, TPL!

> _
```

6 Submission Format

Submit your code in a single `Final_<first_last>.zip` file where first and last are your group speaker's name (e.g., `Final_Stephan_Ohl.zip`).

The zip file should contain (at least):

- Group.txt — group members' names
- Grammar.pdf — the grammar of your language and any additional descriptions
- Cargo Project — the code of your project; builds command line tool
- Presentation.pdf — the slides of your final project presentation in PDF format

7 Optional Features

Make your language and runtime more powerful by implementing some of the following features.

- Nested Blocks — Introduce nested blocks and implement the corresponding scope semantics.
- First-Order Functions — Introduce a type for variables that store a function references; implement the corresponding call syntax and, possibly, anonymous function definition syntax.
- More Types — Add types such as Char, Strings, Lists, and Maps with corresponding literals and operations.
- References — Introduce reference types that can be used to implement call-by-value.
- Static Type System — Add type annotations and static type checking to the analyzer.
- Optimization: Constant Folding — Evaluate literal expressions before runtime.
- Optimization: Algebra — Apply simple laws such as $p \vee T \equiv T$ and $x \cdot 0 = 0$.
- Warnings: Unused Variables — Warn about variables that are never used.
- Warnings: Dead Code — Warn about functions never called and branches that can never be executed.