Compilers Project

Nicolas Hiillos 014801279

March 14, 2022

Instructions

cmake and C++ are required to compile the interpreter. After `./build.sh` is run, the interpreter can be started with `./build/mini-pl [filename]`. For testing purposes, the user can use `./build/mini-pl -s [filename]` to run merely the scanner, or `./build/mini-pl -p [filename]` to run the scanner+parser. Both commands print a readable result. Example programs are provided in `./test/`.

Implementation details

Scanner

The compiler uses a simple ad-hoc scanner and is based off the following regular expressions. The scanner consists of a long switch statement which applies these regular expressions one character at a time, looking ahead as necessary.

```
comment = "/*" ( non-* | "*" non-/ )* "*/"
        | "//" ( non-newline )* newline
slash = "/"
left_paren = "("
right_paren = ")"
minus = "-"
plus = "+"
asterisk = "*"
equal = "="
less = "<"
and = "&"
not = "!"
```

```
semicolon = ";"
assign = ":="
colon = ":"
range = ".."
keyword = "var" | "for" | "end" | "in" | "do" | "read" | "print" | "int" | "string" | "
boolean = "false" | "true"
integer = digit digit*
string = """ character character* """
identifier = letter ( digit | "_" | letter )*
```

The scanner handles errors by outputing tokens of type `ERROR` for unterminated strings and unexpected characters. These error tokens are then handled by the parser.

Parser

The interpreter uses the following, mostly unmodified, CFG.

```
<prog> ::= <stmts>
<stmts> ::= <stmt> “;” ( <stmt> “;” )*
<stmt> ::= “var” <var_ident> “:” <type> [ “:=” <expr> ]
           | <var_ident> “:=” <expr>
           | “for” <var_ident> “in” <expr> “..” <expr> “do”
               <stmts> “end” “for”
           | “read” <var_ident>
           | “print” <expr>
           | “assert” “(” <expr> “)”
<expr> ::= <opnd> <op> <expr>
           | [ <unary_opnd> ] <opnd>
<opnd> ::= <int>
           | <string>
           | <var_ident>
           | “(” <expr> “)”
<type> ::= “int” | “string” | “bool”
<var_ident> ::= <ident>
<reserved_keyword> ::=
“var” | “for” | “end” | “in” | “do” | “read” |
“print” | “int” | “string” | “bool” | “assert”
```

All infix operators are currently left-associative with equal precedence.

Each production of the CFG has a corresponding class, in `parser.cpp`. These classes can link to each other and form a syntax tree. This tree can then be traversed with the visitor pattern by classes inheriting the `TreeWalker` class. This parent class is used to implement a pretty printer of the syntax tree and the interpreter.

The parser output is ready to be interpreted. No changes are made to the syntax tree. The following program creates the following AST.

```
var nTimes : int := 0;
print "How many times? ";
read nTimes;
var x : int;
for x in 0..nTimes-1 do
print x;
print " : Hello, World!\n";
end for;
assert (x = nTimes);

// AST for the program above
(stmts
    (var ident:nTimes type:INT expr:(0))
    (print expr:("How many times? "))
    (read expr:nTimes)
        (var ident:x type:INT )
        (for ident:x from:(0) to:(nTimes - (1)) body:
        (print expr:(x))
        (print expr:(" : Hello, World!\n"))
    end for)
    (assert expr:(x = (nTimes)))
)
```

Interpreter

The interpreter contains a class which inherits `TreeWalker`, used for traversing the AST. It uses a type named `Variable` to store and operate on mini-pl variables. It stores named variables in a map. Variables can be looked up by their name from the map. The interpreter also contains a stack used for unnamed variables and intermediate results in expressions.

Semantic analysis is handled by the interpreter dynamically. Variables can not be referenced, if they have not been initialized. Variable types are also checked by the interpreter.

Future Improvements

- Add operator precedence
- Finish REPL
- Handle parse errors gracefully
- Improve runtime error handling
- Perform semantic analysis ahead of time

Work log

Date	Hours	Progress
[2022-02-08 Tue]	4	Initialize repo and begin planning
[2022-02-22 Tue]	2	Setup cmake
[2022-03-02 Wed]	5	Plan and document scanner, begin implementation
[2022-03-03 Thu]	8	Implement scanner
[2022-03-05 Sat]	7	Begin implementation of parser
[2022-03-08 Tue]	9	Parser
[2022-03-12 Sat]	2	Finish parser, start interpreter
[2022-03-13 Sun]	10	Finish interpreter, docs
TOTAL HOURS	47	