Justin Stanley
COMS 440 Spring 2019
This project can be cloned through the following:
`git://git.linux.iastate.edu:jtst/jtst-coms440`
`https://git.linux.iastate.edu/jtst/jtst-coms440`

Documentation

Note: For Part 4 of the project I decided to rewrite all of the previous parts as well. So, the documentation for lexing, parsing, and type checking have been updated.

1 Lexer

1.1 Design

The updated lexer uses a simpler, basic flex implementation to keep things clean for the parser. The compiler uses a basic flex-based lexer. The source for the lexer is in `scanner.ll`. The lexer is written with many of the built in integrations with Bison.

2 Parser

2.1 design

The compiler uses a standard Bison-based parser, except it has been generated for a C++-based template. This slightly changes how the AST is built and makes things simpler in my opinion.

2.2 data structures

All AST nodes inherit from the `AST::Node` class in `ast/node.hh`. Each node tracks its location in the source code. The parser constructs different `AST` subclasses while building the parse tree. The root node type is `AST::Program`. The `AST::Scope` class maintains a name-safe list of variables and functions, and is used in multiple contexts (`AST::Program`, `AST::Function`).

3 Type checker

3.1 design

The type checker is implemented through the `AST`. Polymorphism is used to recursively type check different types of statements and expressions. Return statements are checked against the function return type, and all expressions are recursively checked based on their definition as well.

4 Code generation

4.1 design

Code generation is also implemented through the `AST`. Polymorphism is used in a very similar fashion to type checking. The `AST::Program` has a `generate_ir` function which recursively generates all of the code for the program. As our target architecture is stack-based, there was no need for register allocation. Each code generation function instead has a flag which indicates if a result from the code group should be left on the stack afterwards. This allows for a great deal of basic optimization – for instance, if `-(1+2);` appears as a statement no code will be generated (as the result is simply discarded). However, this optimization is still safe with evaluation; `-(main());` will still generate code to call the `main()` function, although the return value will be discarded and no unary operation is executed. `AST::LValue` also required a special type of code generation, as some operations needed to retrieve and store a value seperately – the generation functions were named `gen_store_code` and `gen_retrieve_code`.

4.1.1 Code generation: part 2

The compiler now supports branching in code generation. There were no major changes to code structure, but many `gen_code` methods were implemented for the `AST::Statement` subclasses. There was also the introduction of `AST::Statement::backpatch`, which is just a string substitution helper. It is used to implement code generation for `break` and `continue` statements.

5 Sources

Any `.hh` files in this list have their implementation in their respective `.cc` file.

filename	purpose
scanner.ll	lexer source
parser.yy	parser source
ast.hh	all AST types
driver.hh	compiler unit/state
util.hh	utility functions
main.cc	entry point
ast/expression.hh	AST expression types
ast/program.hh	AST program type
ast/function.hh	AST function type
ast/node.hh	AST base node type
ast/scope.hh	AST scope type
ast/statement.hh	AST statement types
ast/variable.hh	AST variable types