JQER

Python, but not real Python

Jiaxuan Pan
Qianjun Chen
Eurey Noguchi
Roger Lu

April 26, 2021

1. Introduction & Proposal

JQER is a Python-like language with static typing. Today, Python is one of the top picks for beginner programmers for how simple the syntax is and its readability. However, we believe that due to the dynamic type system, it also introduces many problems since the programmer will not know if their program will run without errors until they execute it. Therefore, the main goal for JQER is to simplify writing of programs with Python-like syntax and introduce a static typing system.

2. Language Tutorial

2.1 Environment Setup

JQER was developed with OCaml. To use the compiler, OCaml and some dependencies must be installed. The installation can be done using our Docker image. The project can be downloaded from GitHub.

```
$ git clone https://github.com/eureyuri/JQER.git
$ cd JQER
$ docker build -t jqer .
$ docker run -d jqer
$ docker run --rm -it -v `pwd`:/home/jqer -w=/home/jqer jqer
```

2.2 Running and Testing

To run and test all test cases use our Makefile with

```
$ make
```

To run individual tests, run

```
$ bash testall.sh -k tests/test-file.jqer
```

2.3 Sample Program

Here is a sample program that will add two numbers and print. To run this file, we need to create a file with a .jqer extension such as add.jqer.
Some things to keep in mind are that a main method is mandatory in JQER and all variables must be declared at the beginning of the method before the value is assigned.

```
# this is the method for adding 2 numbers
int def add(int a, int b):
  return a + b

int def main():
  int a
  str hello
```

```
a = add(7, 2)
print(a)
hello = “Hello, world!”
print(hello)
```

Output

```
9
Hello, world!
```

3. Language Manual

3.1 Lexical Conventions

3.1.1 Line Structure

A JQER program is divided by a number of logical lines.

3.1.1.1 Logical Lines

The end of a logical line is denoted by the NEWLINE token. Logical lines cannot span multiple lines.

3.1.1.2 Blank lines

A logical line that only contains spaces, tabs, and comments are ignored.

3.1.1.3 Grouping

Whitespaces from tabs at the beginning of a logical line is used to determine the indentation level of the line. The indentation level is then used to resolve the grouping of statements as statement blocks. Here, the control flow keywords such as if, else, for, and while must be followed by a colon and the lines inside the control flow must have the same indentation level, unless there are further control flow keywords.

```
int def square_even(int x):
    if x % 2 == 0:
        return x * x
```

3.1.1.4 Whitespaces between tokens

Whitespaces except at the beginning of a logical line or in str literals are used to separate tokens. It is needed if the concatenation of two tokens could be interpreted as a different token.

3.1.2 Comments

JQER has single-line comments. A single line comment begins with a # (single pound symbol).

```
# This is a single line comment.
```

3.1.3 Identifiers (Names)

An identifier starts with an ASCII letter and can include other ASCII letters, decimal digits, as well as underscores. It is case sensitive and cannot be one of the keywords of JQER.

3.1.4 Keywords

The following identifiers are reserved for the use as keywords of the language and cannot be used as normal identifiers.

```
int str bool None
if else elif for while def return and or not continue break in
True False
```

3.1.5 Literals

Literals represent constant values of built-in types.

3.1.5.1 Int Literal

An int literal consists of any sequence of integers between 0 and 9.

3.1.5.2 Bool Literal

A bool literal can either be True or False.

3.1.5.3 Str Literal

A str literal is a sequence of characters surrounded by double (“) quotes.

3.2 Data Types

3.2.1 Primitives

Primitives are series of bytes of some fixed length, and there are three primitive types in JQER:

int (4 bytes, 2’s complement)
bool (1 byte, 00000001 for True, 00000000 for False).

3.2.2 Mutability

The primitive data types in JQER are immutable, including int, and bool. Str and tuple are not primitives, but are also immutable. All operations which modify these objects actually return new objects.

3.2.3 Standard Library

JQER provides a number of built-in types, including Str. These are implemented in JQER as built-in structures, and have a variety of operations.

3.2.4 None

None is a keyword that returns a reference to a predetermined object of a special type. There is only ever one object of this type.

3.3 Expression

3.3.1 Primary Expression

Primary expression involving identifier, constant. They are grouped left-to-right.

3.3.1.1 Identifier

An identifier is a primary expression, its type is specified by its declaration. An identifier which is declared ‘‘function returning . . .”, when used except in the function-name position of a call, is converted to ‘‘pointer to function returning . . .”.

3.3.1.2 Constant

An int, bool, str is a primary expression.

3.3.2 Tuple

A tuple is an expression but not a primary expression. Its type is specified by “(primary-expression, primary-expression)”. The primary expressions in the parentheses are two integers or identifiers that are variables containing integers.

3.3.3 Multiplicative operators

3.3.3.1 Expression * Expression

The binary * operator indicates multiplication. It only supports two int as operands.

3.3.3.2 Expression / Expression

The binary / operator indicates division. It only supports two int as operands.

3.3.3.3 Expression + Expression

The result is the sum or concatenation of the expressions. It supports two int as operands which is calculating the sum. It also supports str as operands, which concatenate two strs and return a new one.

3.3.3.4 Expression - Expression

The result is the difference of the operands. It supports two int as operands.

3.3.4 Relational operators

The relational operators group left-to-right.
The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield False if the specified relation is False and True if it is true. This operator is applicable only to int.

```
      expression < expression
      expression > expression
      expression <= expression
      expression >= expression
```

3.3.5 Equality operators

The == (equal to) and the != (not equal to). The comparison happens between two values.

```
      expression == expression
      expression != expression
```

3.3.6 Unary operators

Expressions with unary operators group right-to-left.

3.3.6.1 ! Expression

The result of the logical negation operator "!" is True if the value of the expression is False, False if the value of the expression is True. The type of the result is bool. This operator is applicable only to bool.

3.3.6.2 - Expression

The result is the negative of the expression, and has the same type. The type of the expression must be int.

3.3.7 Expression and Expression

The "and" operator returns True if both its operands are True, False otherwise. It guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is False. The operands need to be of type bool.

3.3.8 Expression or Expression

The "or" operator returns True if either of its operands is True, and False otherwise. It guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is True. The operands need to be of type bool.

3.3.9 identifier = expression

The assignment operator is grouped right-to-left. It requires an identifier as their left operand. The value is the value stored in the left operand after the assignment has taken place. The value of the expression replaces that of the object referred to by the identifier.

3.4 Declarations

Declarations are to specify the interpretation which JQER gives to each identifier. They do not necessarily reserve storage associated with the identifier. Declarations have the form

```
declaration:
          type-specifier declarator
```

3.4.1 Type specifiers

The type specifiers are

```
type-specifier:
     int
     str
     bool
     tuple
```

3.4.2 Declarators

The syntax of declarators:

```
declarator:
     identifier
     def declarator( )
```

3.4.3 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier. It is this identifier that is declared.
If a declarator is

```
def D( )
```

then the contained identifier has the type ‘‘function returning ...”, where “ ... ” is the type which the identifier would have had if the declarator had been simply D.

See more in the function definition section below.

3.5 Statements

Excepted as indicated, statements are executed from left to right and from top to bottom.

3.5.1 Expression Statement

Some statements are expression statements, which have the form

```
      expression
```

Usually expression statements are assignments or function calls.

3.5.2 Compound statement

The compound statement is in this form:

```
      compound-statement:
            statement-list
      statement-list:
            statement
            statement statement-list
```

Where a statement-list is a group of statements.

3.5.3 Conditional statement

The conditional statement has these forms:

```
      if expression:
            statement

      if expression:
            statement
      else statement
```

Where the notation (...)* means 0 or more of the same statements.

3.5.4 While statement

The while statement has this form:

```
      while(expression):
            statement-list
```

The statement is executed repeatedly until the value of the expression becomes false.

3.5.5 For statement

The for statement has this form:

```
      for(expression-opt; expression-opt; expression-opt):
          Statement-list
```

The first expression is to assign initial value to the variable that is being checked. The second expression is the condition that is being checked for termination. The loop will terminate when the value of this expression is false. The third is the change applied to the variable in every iteration.

3.5.6 Return statement

The return statement can be:

```
     return
     return expr
```

In the first case, no value will be returned. In the second case, the value of the expression is returned to the caller of the function. The type of expr should match the return type in the function definition.

3.6 Function Definition

All function declarations are by default and the only type of external definitions for JQER, which means there must be an external definition for the given identifiers somewhere outside the function in which they are declared.

Function definitions have the form:

```
     function-definition:
          type-specifier function-declarator function-body
```

Type-specifier is the same as in 5.1 and represents the return value of the function.

```
     function-declarator:
          declarator (parameter-list)
```

Parameter-list is optional.

```
     parameterlist:
          parameter , parameterlist
          empty-list

     parameter:
          type-specifier identifier
```

Thus the parameters would be declared in function-declarator.

```
     function-body:
          declaration-list-opt statement-list

     declaration-list-opt:
          declaration, declaration-list-opt
```

```
statement-list:
	statement, statement-list
```

A simple example of a complete function definition is below, with indentation as tab:

```
int def max(int a, int b, int c):
	int m
	if a > b:
		m = a
	else m = b
	if m > c:
		return m
	else return c
```

3.7 Scope Rules

JQER has not yet supported or been supported by any external files or libraries, mainly because of the lack of I/O system. Accordingly, all declarations, definitions and operations would be done as a unit of a single file. All declarations and definitions at the top level of the file would be considered "global" thus extending themselves from their definition through the end of the file in which they appear.

It is an error to redeclare identifiers already declared in the current context, unless the new declaration specifies the same type and storage class as already possessed by the identifiers.

Since we use tab indentations to represent the beginning of three cases: a function definition, a conditional flow and a loop; with cancelling the tab indentation to represent the end of them, we consider JQER a block-structured language. All declarations and definitions when one or more indentations exist are only valid (considered declared as "local") within the current level block, which is represented by matching indentation levels before them. Multiple declarations on the same identifier is allowed between two parallel blocks on the same level or blocks within them with a lower level, but redeclare identifiers already declared in a higher level is an error.

3.8 Built-in Module

3.8.1 print(var)

print(var) is used to print the str representation of the variable to standard output. The nature of that representation depends on the type of variable. If variable is one of the primary expressions, there is a built-in print for it.

For example:

```
for i in 4:
	print(i) # prints 1 2 3 4
```

3.9. Examples

3.9.1 Fibonacci

This function returns the integer value of n-th element of the Fibonacci Sequence.

```
int def fib_recur(int n):
    if n <= 1:
        return n
    return fib_recur(n-1) + fib_recur(n-2)
```

4. Project Plan

4.1 Planning & Development Process

In general, JQER group members met every week; the day of the week varied based on availability, but most often we met on Wednesday and Sunday. As deliverable deadlines approached, we met more times a week to complete the work.

We used Zoom to hold meetings and used WeChat to communicate with each other, and any deadlines or responsibilities that we decided on at meetings were put in the WeChat. To write the language reference manual, we split up the different sections amongst the 4 of us. To develop our language, the system architect came up with the whole architecture, and the language guru developed the starting template of the whole structure. The coding of different functionalities was splitted up, and every one was writing tests suite to cover their implementations, and the tester was making sure all functionalities were tested comprehensively.

We spent a lot of time trying to implement the tree type as we were initially designed, but we didn't finish it because we did not have enough resources to complete it after we implemented other important parts (indentation/grouping and special primitive types), even though we did a lot of experiments on it.

4.2 Timeline

Milestone	**Due Date**
Deciding on language	Jan 27
Initial proposal written	Jan 31
Proposal Submission	Feb 3
Split up work on language reference manual	Feb 10
LRM, Parser Submission	Feb 24
Ast, Sast	Mar 3
Codegen, tests, makefile	Mar 21
Hello World Submission	Mar 24
Basic operators and control flows	April 1
Indentation and Grouping	April 4

String, Tuple and tree	April 14
Char	April 21
Testing and clean-up	April 24
Demo	April 25

4.3 Roles and Responsibilities

Eurey Noguchi - Manager:

As a manager, I reminded my group when the meetings were happening and kept track of deadlines approaching. For the scanner, parser, and jqer.ml files from what I received from Jiaxuan as boilerplate, I worked on implementing the Python-like syntax of grouping with indentation through using a stack. The next task for me was to make print statements be able to work with any primitive type, which required me to change the scanner, parser, ast, sast, and codegen. Since there were many errors and warnings I also worked on resolving these issues throughout the project. I also rewrote many of the test cases from microC so that it matched our syntax. Finally, at the end of the project, I worked on implementing the char type.

Jiaxuan (Percy) Pan - Language Guru:

I was responsible for the language design, reference searching and coming up with the starting point of the project, including parser, scanner. I was also responsible for Tuple and String implementation and some pieces of the ast, sast, codegen and makefiles.

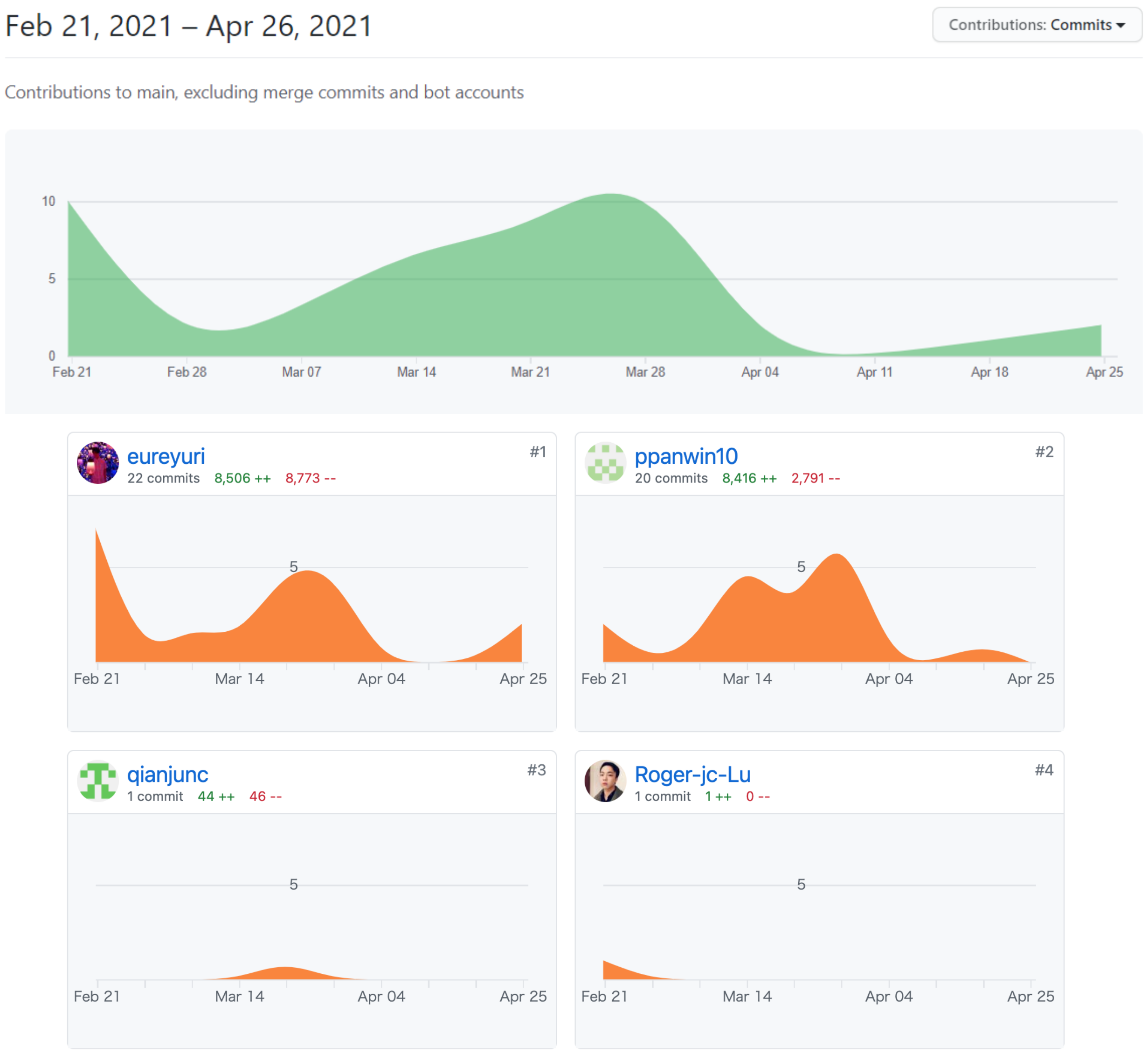
Qianjun Chen - System architect:

I was responsible for doing research and implementing tree structure and solving some of the warnings.

Roger Lu - :

I was responsible for test suite merging, bug fixing and language structure modification.

4.4 Project Commits


Feb 21, 2021 – Apr 26, 2021

Contributions: Commits

Contributions to main, excluding merge commits and bot accounts

eureyuri #1
22 commits 8,506 ++ 8,773 --

ppanwin10 #2
20 commits 8,416 ++ 2,791 --

qianjunc #3
1 commit 44 ++ 46 --

Roger-jc-Lu #4
1 commit 1 ++ 0 --

4.5 Commit log

```
commit e48fafe0682d37a9f32fc76791f288735000cb44
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sun Apr 25 15:57:43 2021 -0400

    clean

commit 6e75503c54ecac57aaca638cd7c57784b75724a0
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sun Apr 25 15:56:44 2021 -0400
```

```
    fix: failed test

commit ee3502607967864981f9bbe79fa1c428b53b6f0f
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sat Apr 24 21:13:37 2021 -0400

    clean up

commit aed72b7bf8375ce69860d0a7b26e006df8113c7d
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sat Apr 3 21:37:59 2021 -0700

    update tuple test

commit 06092dbb2af87ff60354ffad4c834f7db3046436
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sat Apr 3 20:58:14 2021 -0700

    clean up.

commit 8abda365eced31194139790c7fa4bc6ec2d1bc2f
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sat Apr 3 20:36:46 2021 -0700

    clean up the unused list definition.

commit a82fc4bfc675c0dcb5b5a635ec0cdcec891cf540
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sat Apr 3 20:24:08 2021 -0700

    add tuple type

commit a7a12fc4de98ba1870c2a79e1c36e4b1b367be70
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Fri Apr 2 08:21:32 2021 -0700

    resolved the diff in fail test

commit c17f9c0fa91636291f31b9e5aa0a6033e373e904
Merge: c476fbf c484933
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Wed Mar 31 04:16:01 2021 +0900

    Merge pull request #11 from eureyuri/indentation

    Indentation

commit c48493346477ff17d12c8afc94da8237c03417ff
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Tue Mar 30 15:13:46 2021 -0400

    fix fail test cases

commit 5d97d0e63508cbd1ee780419000d2ce3d8a75fe4
```

```
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Tue Mar 30 15:00:10 2021 -0400

    add for loop; fix success tests

commit c476fbfa00b03064199403072e939a6add8d677d
Merge: ee562c5 2201298
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Tue Mar 30 07:28:13 2021 +0900

    Merge pull request #10 from eureyuri/indentation

    adjust test syntax

commit 22012989a655fbef0c918aaa4e24e8af3ff22da8
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Mon Mar 29 18:27:29 2021 -0400

    adjust test syntax

commit ee562c59dbfef3bf649a7cdd013064eea640525f
Merge: a87675b 97f826f
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Tue Mar 30 07:03:02 2021 +0900

    Merge pull request #9 from eureyuri/indentation

    more like python syntax

commit 97f826fcbcd8b85a3646d789c0fff8a741388f63
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Mon Mar 29 18:01:47 2021 -0400

    more like python syntax

commit a87675bcd35d9e0ee0e39f9d97fde9cfe1fdebba
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sun Mar 28 20:44:50 2021 -0700

    add tests files

commit 56727b726335522e9f7057f83711ea4381f37c25
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sun Mar 28 17:50:51 2021 -0700

    clean up.

commit 50d0b1c91c1f9947983a285420b0ae0abf2c4a09
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sun Mar 28 17:45:52 2021 -0700

    update the docker file

commit eed187b6cafcea1f4978414d2be3351472915091
Merge: 73d95bf d2b1c8a
```

```
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Mon Mar 29 09:31:36 2021 +0900

    Merge pull request #8 from eureyuri/print-refactor

    delete float

commit d2b1c8a8daa0d79e409837d73057e9703fcdc7a5
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sun Mar 28 20:30:17 2021 -0400

    delete float

commit 73d95bf8aa8242837b5798ff11c4929c2e137f3a
Merge: a3b97f8 2093472
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Sun Mar 28 07:13:22 2021 +0900

    Merge pull request #7 from eureyuri/print-refactor

    fix warnings

commit 20934720cfb8ca686af1312bb7df5958e5af650f
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sat Mar 27 18:12:39 2021 -0400

    remove files

commit 2703816224cbfdc04af192f7f5cdab690c415bf3
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sat Mar 27 18:12:03 2021 -0400

    fix all warnings

commit a3b97f8d2d083e59455cb4d223f7f7c6069d9a4f
Merge: abfc253 c4fc2e4
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Sat Mar 27 12:51:42 2021 +0900

    Merge pull request #6 from eureyuri/print-refactor

    fix warnings

commit c4fc2e48b23b5adbe9d1ee19a0387ae459cba636
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Fri Mar 26 23:51:04 2021 -0400

    fix warnings

commit abfc2539fb71e1f54d7e185da4316e6e33f067b1
Merge: 2ab41de aed07bc
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Sat Mar 27 11:33:01 2021 +0900

    Merge pull request #5 from eureyuri/print-refactor
    modify and or

commit aed07bc20f183dd3ce0a919c87100b3bdfae924b
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Fri Mar 26 22:27:14 2021 -0400

    modify and or

commit 2ab41de93912f48f5beeb86e5e871b198fa2bc91
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Fri Mar 26 08:45:34 2021 -0700

    rename pyl to jqer

commit 77ac2935702987bc62b1fd2a84dccbd654382430
Author: Qianjun Chen <qianjunc@gmail.com>
Date:   Wed Mar 24 21:55:36 2021 -0400

    fix some warning

commit 39ad4d28a80df88007066fdd43efeccdb516685f
Merge: 287bfd8 4137293
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Wed Mar 24 03:31:11 2021 +0900

    Merge pull request #4 from eureyuri/print-refactor

    feat: refactor print so it is similar to Python

commit 4137293649e874aefaf30cc4325a671193eb6042
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Tue Mar 23 14:29:30 2021 -0400

    feat: refactor print so it is similar to Python

commit 287bfd81d7105d5be895206889b5d778f9cc3498
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Mon Mar 22 19:26:34 2021 -0700

    change the ref to pylit

commit 4278a24a775e5077feb7dcbd63fc0de0777e33cf
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Fri Mar 19 11:10:08 2021 -0400

    renaming cleaning

commit 6a86ba7b41a92f6bad8d0c055907484fa95dadac
Merge: e0efe0d 9e455c6
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 17 22:48:25 2021 -0400

    Merge branch 'main' of https://github.com/eureyuri/JQER
```

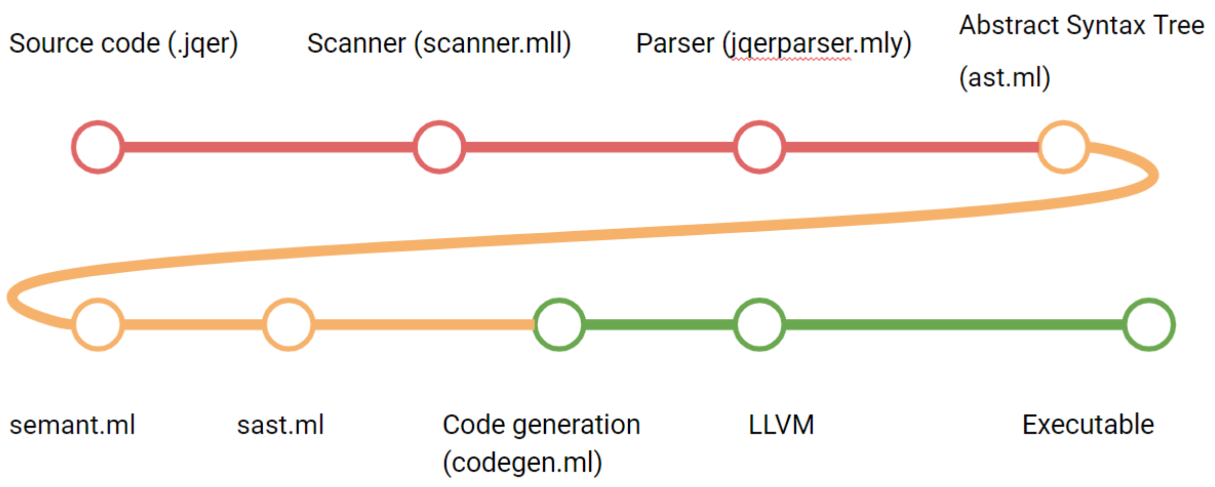
```
commit e0efe0d06216477e8fd9f1f1268aefb60abae005
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 17 22:48:18 2021 -0400

    clean up lexer

commit 9e455c63e784ebc70bbbec8b3df5936be3acd379
Merge: d809a6c a5b9370
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Thu Mar 18 11:22:51 2021 +0900

    Merge pull request #3 from eureyuri/parser-eurey

    feat: connecting parser, scanner, and ast with addition of type decla…

commit a5b9370da04164f8c6255f016aed752cf6a4a474
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Wed Mar 17 22:21:58 2021 -0400

    feat: connecting parser, scanner, and ast with addition of type declaration

commit d809a6cc52df1a2f6ff3a3971cb09880fcf637b7
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 17 15:35:07 2021 -0400

    clean up

commit 5a9b7e866c592c6feeb2765b52c619ccb21a6ace
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 17 15:21:26 2021 -0400

    clean up and simplied the lexer

commit ecb3fca4107eecb11be6a75b7280af97bb1b11e1
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sun Mar 14 23:10:41 2021 -0400

    replicate mini py

commit 96c8373c7d736418e3bec3189eaef1c189fbe17c
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Sun Mar 14 23:10:08 2021 -0400

    replicate the mini python

commit dd6f571e817ec58b95fcb1ea6bdd2b612350def5
Merge: 69ceb6d 7216ad2
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Sat Mar 13 01:43:18 2021 +0900

    Merge pull request #2 from eureyuri/parser-eurey

    fix: reduce/reduce conflicts for NONE

commit 7216ad23d5342181a4167168c53786ced784457f
```

```
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Fri Mar 12 11:42:13 2021 -0500

    fix: reduce/reduce conflicts for NONE

commit 69ceb6ddef62e5422b5148f3e6b7473093dac042
Merge: eb0cdd7 67ef986
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Fri Mar 12 13:25:08 2021 +0900

    Merge pull request #1 from eureyuri/parser-eurey

    addition to parser

commit 67ef98645756c06898128854ef37faaa3aeb7ea1
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Thu Mar 11 23:24:21 2021 -0500

    addition to parser

commit eb0cdd718f0ba20ed5af9239ea0844136e778f53
Merge: 17c9b02 ab6b762
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 10 22:52:54 2021 -0500

    Merge branch 'main' of https://github.com/eureyuri/JQER

commit 17c9b025e548d007a946f5342eaeeb8021b3e4ba
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Mar 10 22:51:47 2021 -0500

    Refine the scanner based on the ref language

commit ab6b762759dcd1450c9789c60fc79763c64ab8ae
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Sat Feb 27 04:40:04 2021 +0900

    Update 2-26.md

commit 941c5585b984e8b37e07a25d3c6f6e90c125cc6c
Merge: a3a9e23 132afd2
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Fri Feb 26 14:37:51 2021 -0500

    Merge branch 'main' of github.com:eureyuri/JQER into main

commit a3a9e237a2540b15a6174ade4134633e4d625a14
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Fri Feb 26 14:36:51 2021 -0500

    notes: 2/26

commit 132afd2620fe08d6f5db9ed4826a09414dae4f26
Author: RogerLu <ljctbs520@gmail.com>
Date:   Wed Feb 24 23:40:02 2021 -0500
```

```
    to do

commit 9132e14762d0bc9ae93e11f39ecd1fc3449e7203
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Feb 24 12:22:24 2021 -0500

    Refined the parser.

commit 0bdfb60387a84e723814c2798e801d9c2eadbf89
Author: ppanwin10 <percyjxp@gmail.com>
Date:   Wed Feb 24 00:40:56 2021 -0500

    Refined the Tree expr

commit ca6d2ab1ca1f5321944f319391992e675f2440a5
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Mon Feb 22 20:58:44 2021 -0500

    feat: parsing statements

commit 2bcbbab8c1d7621c629cf0dbab046fbbba1f780c
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Mon Feb 22 16:59:45 2021 -0500

    feat: parser expressions; Makefile

commit 695a6949b9e1f96e07208ccac3bc86797a028471
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sun Feb 21 23:28:05 2021 -0500

    feat: defining tokens

commit 998be57b739e613564e769138e5e6e9d777825c4
Author: eureyuri <eureynoguchi@gmail.com>
Date:   Sun Feb 21 22:50:30 2021 -0500

    feat: parser and scanner from calculator

commit 603626cd665428a91c7393e7f5ebc8423891debc
Author: Eurey (Yuri) Noguchi <eureynoguchi@gmail.com>
Date:   Mon Feb 22 10:07:10 2021 +0900

    Initial commit
```

5. Architectural Design

5.1 Overall Flow



Scanner: the scanner leverages ocamllex to produce a lexical analyzer from a set of regular expressions with associated semantic actions. It takes a jqer source file as it's input and generates tokens based on acceptable values defined in the scanner source code. In order to implement grouping by indentation we used a stack to keep track of indentation level and insert our own tokens to indicate the beginning and end of a grouping.

Parser and AST: the parser takes in tokens generated from the lexer. Using ocamlyacc, a parser is produced from the jqer production rules specified in the source code. An abstract syntax tree is constructed given the data types defined in the AST source code.

Semantic check: the semantic module recursively traverses the AST and converts it into a semantically checked abstract syntax tree. The sast adds type annotations to expressions. The sast is needed because the code generator needs type information to perform correctly.

Code Generation: after the AST is checked for semantic accuracy, the codegen.ml converts the AST into LLVM IR. This intermediate representation is assembly language that can be used for code generation of Intel X86 binary, ARM, etc.

6. Test Plan

6.1 Arithmetic

```
int def main():
 print(20 + 5)
```

Output:

```
25
```

6.2 Variable declaration

```
none def foo(bool i):
 int i
 i = 9
 print(i + i)

int def main():
 foo(true)
```

Output:

```
18
```

6.3 Function declaration

```
int def add(int a, int b):
 return a + b

int def main():
 int a
 a = add(7, 2)
 print(a)
```

Output:

```
9
```

6.4 Control flow - if/else

```
int def main():
 if (true):
   print(19)
 else:
   print(8)
 print(17)
```

Output:

```
8
17
```

6.5 Control flow - for

```
int def main():
 int i
 for (i = 0 ; i < 5 ; i = i + 1):
   print(i)
 print(12)
```

Output:

```
0
1
2
3
4
12
```

6.6 Control flow - while

```
int def main():
 int i = 0
 while (i < 5):
   print(i)
   i = i + 1
```

Output:

```
0
1
2
3
4
```

6.7 Multi-level grouping

```
int def gcd(int a, int b):
 while (a != b):
   if (a > b):
     a = a - b
   else:
     b = b - a
 return a

int def main():
   print(gcd(14,21))
   print(gcd(8,36))
   print(gcd(99,121))
```

Output:

```
7
4
```

```
11
```

6.8 Tuple

```
int def main():
 tuple t
 int a
 int b

 a = 1
 b = 2
 t = (a , b)
 print(t.0)
    print(t.1)
```

Output:

```
1
2
```

7. Reflection

Roger

Aim low at first to build up on the work, instead of cutting down features later. I also feel that we spent too much time restrained by Ocaml where we could refer to a lot of external libraries for features when came to LLVM to save time.

Eurey Noguchi

It is important to determine how to correctly divide work. For example, it is not possible for one person to finish the scanner entirely and another person to complete the parser. Therefore, we should be taking very small steps where each person touches on all files to implement one feature at a time. I believe that testing played a key role in the project since we would immediately know if we broke something with the new implementation and also easily verify if our program is working as we expected. Lastly, dealing with LLVM was difficult for me and found the importance of understanding documentation precisely.

Qianjun Chen

The most important lesson I learned is to always start and plan early, both in terms of the overall project and each individual task. This project is complicated and it takes a lot of time to absorb the new information and do research for our project specifically after lectures, before we are able to actually use it to build something with this scale. Also, when trying to add a new feature to the project, it might involve big changes to many parts of the code, so starting and planning early is crucial.

The other important thing I should always remind myself is that it is important to stick with schedule and try to finish tasks as soon as possible even when the time left seems abundant. If one task is not completed on time, it is possible that all other tasks that have been scheduled will be delayed. This is something I didn't do well in this project.

Jiaxuan Pan

Designing a language is interesting but challenging. Working with a group is fun and necessary for a big project like this, but it is not easy to make everyone on the same page or make everyone happy all the time. Will not lose faith in humanity, but will keep cautious.

8. Source code:

8.1 scanner.mll

```
(* Ocamllex scanner for JQER *)

{
  open Jqerparse
  open Lexing

  let unescape s =
      Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -> x)

   (* Referenced for using stack:
https://github.com/LibAssignment/INF564-assignment2 *)
  let stack = ref [0]  (* indentation stack *)
  let rec unindent n = match !stack with
    | m :: _ when m = n -> []
    | m :: st when m > n -> stack := st; END :: unindent n
    | _ -> raise (Failure "bad indentation")
  let update_stack n =
    match !stack with
    | m :: _ when m < n ->
      stack := n :: !stack;
      [NEWLINE; BEGIN]
    | _ ->
      NEWLINE :: unindent n
 }

let letter = ['a'-'z' 'A'-'Z']
let digit = ['0'-'9']
let ident = letter (letter | digit | '_')*
let digits = digit+
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let string = '"' ( (ascii | escape)* as s) '"'
let space = ' ' | '\t'
let comment = "#" [^'\n']*
```

```
rule token = parse
| '\n'    { new_line lexbuf; update_stack (indentation lexbuf)
}
| '\r' { token lexbuf } (* Whitespace *)
(* | '#'     { comment lexbuf }           Comments *)
| (space | comment)+ { token lexbuf }
| '('      { [LPAREN] }
| ')'      { [RPAREN] }
| '{'      { [LBRACE] }
| '}'      { [RBRACE] }
| '['      { [LSQUARE] }
| ']'      { [RSQUARE] }
| ':'      { [COLON] }
| ','      { [COMMA] }
| '+'      { [PLUS] }
| '-'      { [MINUS] }
| '*'      { [TIMES] }
| '/'      { [DIVIDE] }
| '%'      { [MODULUS] }
| '='      { [ASSIGN] }
| "=="     { [EQ] }
| "!="     { [NEQ] }
| "<"      { [LT] }
| "<="     { [LEQ] }
| ">"      { [GT] }
| ">="     { [GEQ] }
| "and"     { [AND] }
| "or"     { [OR] }
| "!"      { [NOT] }
| ";"      { [SEMI] }
| "."      { [DOT] }
| "def"    { [DEF] }
| "if"     { [IF] }
| "else"   { [ELSE] }
| "for"    { [FOR] }
| "while"  { [WHILE] }
| "return" { [RETURN] }
| "int"    { [INT] }
| "bool"   { [BOOL] }
| "none"   { [NONE] }
| "str"    { [STRING] }
| "true"   { [BLIT(true)]  }
| "false"  { [BLIT(false)] }
| "print"  { [PRINT] }
| "tuple"  { [TUPLE] }
| digits as lxm { [INT_LITERAL(int_of_string lxm)] }
| ident as lxm { [ID(lxm)] }
| string            { [STRING_LITERAL( (unescape s) )] }
| eof { [EOF] }
| _ as char { raise (Failure("illegal character " ^
Char.escaped char)) }

(* and comment = parse
  '\n' { token lexbuf }
| _    { comment lexbuf } *)

and indentation = parse
  | (space | comment)* '\n'
      { new_line lexbuf; indentation lexbuf }
  | space* as s
      { String.length s }


{
  let next_token =
    let tokens = Queue.create () in (* next tokens to send back
*)
    fun lb ->
      if Queue.is_empty tokens then begin
     let l = token lb in
     List.iter (fun t -> Queue.add t tokens) l
      end;
      Queue.pop tokens
}
```

8.2 jqerparse.mly

```
/* Ocamlyacc parser for JQER */

%{
open! Ast
%}

%token COLON LPAREN RPAREN LBRACE RBRACE LSQUARE RSQUARE COMMA
PLUS MINUS TIMES DIVIDE MODULUS ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR EOL DOT
%token DEF BEGIN END NEWLINE RETURN IF ELSE FOR WHILE INT BOOL
TUPLE NONE STRING PRINT SEMI
```

```
%token <int> INT_LITERAL
%token <bool> BLIT
%token <string> ID STRING_LITERAL
%token <int * int> TUPLE_LITERAL
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MODULUS
%right NOT

%%

program:
  decls EOF { $1 }

decls:
   /* nothing */ { ([], [])               }
 | decls vdecl { (($2 :: fst $1), snd $1) }
 | decls fdecl { (fst $1, ($2 :: snd $1)) }

fdecl:
   typ DEF ID LPAREN formals_opt RPAREN COLON NEWLINE BEGIN
vdecl_list stmt_list END
     { { typ = $1;
       fname = $3;
       formals = List.rev $5;
       locals = List.rev $10;
       body = List.rev $11 } }

formals_opt:
    /* nothing */ { [] }
  | formal_list   { $1 }

formal_list:
    typ ID                   { [($1,$2)]     }
  | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:
    INT   { Int   }
  | BOOL  { Bool  }
  | NONE  { None  }
  | STRING { String }
  | TUPLE { Tuple }

vdecl_list:
    /* nothing */    { [] }
  | vdecl_list vdecl { $2 :: $1 }

vdecl:
   typ ID NEWLINE { ($1, $2) }

stmt_list:
    /* nothing */  { [] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    expr NEWLINE                              { Expr $1
}
  | RETURN expr_opt NEWLINE                   { Return $2
}
  | COLON NEWLINE BEGIN stmt_list END                    {
Block(List.rev $4)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5,
Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)
}
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN stmt
                                            { For($3, $5, $7,
$9)   }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)
}
  | PRINT LPAREN expr RPAREN NEWLINE { Print($3) }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

expr:
    INT_LITERAL          { IntLiteral($1)            }
  | BLIT             { BoolLit($1)            }
  | STRING_LITERAL   { StringLit($1) }
  | ID               { Id($1)                  }
  | expr PLUS   expr { Binop($1, Add,   $3)    }
  | expr MINUS  expr { Binop($1, Sub,   $3)    }
  | expr TIMES  expr { Binop($1, Mult,  $3)    }
  | expr DIVIDE expr { Binop($1, Div,   $3)    }
  | expr MODULUS expr { Binop($1, Mod,   $3)    }
  | expr EQ     expr { Binop($1, Equal, $3)    }
  | expr NEQ    expr { Binop($1, Neq,   $3)    }
  | expr LT     expr { Binop($1, Less,  $3)    }
  | expr LEQ    expr { Binop($1, Leq,   $3)    }
  | expr GT     expr { Binop($1, Greater, $3)  }
  | expr GEQ    expr { Binop($1, Geq,   $3)    }
  | expr AND    expr { Binop($1, And,   $3)    }
  | expr OR     expr { Binop($1, Or,    $3)    }
  | MINUS expr %prec NOT { Unop(Neg, $2)       }
  | NOT expr         { Unop(Not, $2)           }
  | ID ASSIGN expr   { Assign($1, $3)          }
  | ID LPAREN args_opt RPAREN { Call($1, $3)   }
  | LPAREN expr RPAREN { $2                    }
  | ID DOT INT_LITERAL  { TupleAccess($1, $3)}
  | LPAREN expr COMMA expr RPAREN { TupleLiteral($2,$4) }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

args_list:
    expr                    { [$1] }
  | args_list COMMA expr { $3 :: $1 }
```

8.3 ast.ml

```
(* Abstract Syntax Tree *)

type op = Add
        | Sub
        | Mult
        | Div
        | Mod
        | Equal
        | Neq
        | Less
        | Leq
        | Greater
        | Geq
        | And
        | Or

type uop = Neg
         | Not

type typ = Int
         | Bool
         | None
         | String
         | Tuple

type bind = typ * string

type expr =
    IntLiteral of int
  | BoolLit of bool
  | TupleAccess of string * int
  | TupleLiteral of expr * expr
  | Id of string
  | StringLit of string
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | Print of expr

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

type program = bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "and"
  | Or -> "or"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let rec string_of_expr = function
    IntLiteral(l) -> string_of_int l
  | StringLit s -> "\"" ^ s ^ "\""
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | TupleLiteral(e1, e2) -> "(" ^ string_of_expr e1 ^ ", " ^
string_of_expr e2 ^ ")"
  | TupleAccess(s1, s2) -> s1  ^ "." ^ string_of_int s2
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^
")"
  (* | Seq(a) -> string_of_list a *)
  | Noexpr -> ""
```

```
let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_stmt stmts)
^ "}\n"
  | Expr(expr) -> string_of_expr expr ^ ";\n";
  | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
  | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2
^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^
string_of_stmt s
  | Print(e) -> string_of_expr e

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | None -> "none"
  | String -> "str"
  | Tuple -> "tuple"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^
";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " def" ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd
fdecl.formals) ^
  ")\n:\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "\n" ^ "end"

let string_of_program (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

8.4 semant.ml

```
(* Semantic checking for the JQER compiler *)

open! Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =

  (* Verify a list of bindings has no none types or duplicate
names *)
  let check_binds (kind : string) (binds : bind list) =
    List.iter (function
      (Ast.None, b) -> raise (Failure ("illegal none " ^ kind ^
" " ^ b))
      | _ -> ()) binds;
    let rec dups = function
        [] -> ()
      |    ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (**** Check global variables ****)

  check_binds "global" globals;

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no
bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = None;
      fname = name;
      formals = [(ty, "x")];
      locals = []; body = [] } map
    in List.fold_left add_bind StringMap.empty [
                                            ("printbig", Int) ]
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    let built_in_err = "function " ^ fd.fname ^ " may not be
defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions
of built-ins *)
         _ when StringMap.mem n built_in_decls -> make_err
built_in_err
       | _ when StringMap.mem n map -> make_err dup_err
       | _ ->  StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  let function_decls = List.fold_left add_func built_in_decls
functions
  in

  (* Return a function from our symbol table *)
  let find_func s =
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function "
^ s))
  in

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    (* Make sure no formals or locals are none or duplicates *)
    check_binds "formal" func.formals;
    check_binds "local" func.locals;

    (* Raise an exception if the given rvalue type cannot be
assigned to
       the given lvalue type *)
    let check_assign lvaluet rvaluet err =
       if lvaluet = rvaluet then lvaluet else raise (Failure
err)
```

```
    in

    (* Build local symbol table of variables for this function
*)
    let symbols = List.fold_left (fun m (ty, name) ->
StringMap.add name ty m)
                     StringMap.empty (globals @ func.formals @
func.locals )
    in

    (* Return a variable from our local symbol table *)
    let type_of_identifier s =
      try StringMap.find s symbols
      with Not_found -> raise (Failure ("undeclared identifier
" ^ s))
    in

    (* Return a semantically-checked expression, i.e., with a
type *)
    let rec expr = function
        IntLiteral  l -> (Int, SIntLiteral l)
      | BoolLit l  -> (Bool, SBoolLit l)
      | Noexpr     -> (None, SNoexpr)
      | StringLit s -> (String, SStringLit s)
      | TupleLiteral (x, y) -> let t1 = expr x and t2 = expr y
in
        (Tuple, STupleLiteral (t1, t2))
      | TupleAccess (s1, s2) ->  (Int, STupleAccess(s1, s2))
      | Id s       -> (type_of_identifier s, SId s)
      | Assign(var, e) as ex ->
          let lt = type_of_identifier var
          and (rt, e') = expr e in
          let err = "illegal assignment " ^ string_of_typ lt ^
" = " ^
            string_of_typ rt ^ " in " ^ string_of_expr ex
          in (check_assign lt rt err, SAssign(var, (rt, e')))
      | Unop(op, e) as ex ->
          let (t, e') = expr e in
          let ty = match op with
            Neg -> t
          | Not when t = Bool -> Bool
          | _ -> raise (Failure ("illegal unary operator " ^
                                 string_of_uop op ^
string_of_typ t ^
                                 " in " ^ string_of_expr ex))
          in (ty, SUnop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
          let (t1, e1') = expr e1
          and (t2, e2') = expr e2 in
          (* All binary operators require operands of the same
type *)
          let same = t1 = t2 in
          (* Determine expression type based on operator and
operand types *)
          let ty = match op with
              Add | Sub | Mult | Div | Mod when same && t1 =
Int   -> Int
            | Add                          when same && t1 =
String -> String
          | Equal | Neq            when same              ->
Bool
          | Less | Leq | Greater | Geq
                   when same -> Bool
          | And | Or when same && t1 = Bool -> Bool
          | _ -> raise (
           Failure ("illegal binary operator " ^
                     string_of_typ t1 ^ " " ^ string_of_op op
^ " " ^
                     string_of_typ t2 ^ " in " ^
string_of_expr e))
          in (ty, SBinop((t1, e1'), op, (t2, e2')))
      | Call(fname, args) as call ->
          let fd = find_func fname in
          let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise (Failure ("expecting " ^ string_of_int
param_length ^
                          " arguments in " ^ string_of_expr
call))
          else let check_call (ft, _) e =
            let (et, e') = expr e in
            let err = "illegal argument found " ^ string_of_typ
et ^
              " expected " ^ string_of_typ ft ^ " in " ^
string_of_expr e
            in (check_assign ft et err, e')
          in
          let args' = List.map2 check_call fd.formals args
          in (fd.typ, SCall(fname, args'))
    in
    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^
string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing
sexprs *)
    let rec check_stmt = function
        Expr e -> SExpr (expr e)
      | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1,
check_stmt b2)
      | For(e1, e2, e3, st) ->
       SFor(expr e1, check_bool_expr e2, expr e3, check_stmt
st)
      | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
      | Print(e) -> SPrint(expr e)
      | Return e -> let (t, e') = expr e in
        if t = func.typ then SReturn (t, e')
        else raise (
       Failure ("return gives " ^ string_of_typ t ^ " expected
" ^
             string_of_typ func.typ ^ " in " ^ string_of_expr
e))

         (* A block is correct if each statement is correct and
nothing
            follows any Return statement.  Nested blocks are
flattened. *)
      | Block sl ->
          let rec check_stmt_list = function
              [Return _ as s] -> [check_stmt s]
            | Return _ :: _   -> raise (Failure "nothing may
follow a return")
            | Block sl :: ss  -> check_stmt_list (sl @ ss) (*
Flatten blocks *)
            | s :: ss         -> check_stmt s ::
check_stmt_list ss
            | []              -> []
          in SBlock(check_stmt_list sl)

    in (* body of check_function *)
    { styp = func.typ;
```

```
      sfname = func.fname;
      sformals = func.formals;
      slocals  = func.locals;
      sbody = match check_stmt (Block func.body) with
     SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't
become a block?"))
    }
  in (globals, List.map check_function functions)
```

8.5 sast.ml

```
(* Semantically-checked Abstract Syntax Tree *)

open Ast

type sexpr = typ * sx
and sx =
    SIntLiteral of int
  | SBoolLit of bool
  | STupleLiteral of sexpr * sexpr
  | STupleAccess of string * int
  | SId of string
  | SStringLit of string
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of string * sexpr
  | SCall of string * sexpr list
  | SNoexpr

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt
  | SPrint of sexpr

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }

type sprogram = bind list * sfunc_decl list

(* Pretty-printing functions *)

let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
    SIntLiteral(l) -> string_of_int l
  | SBoolLit(true) -> "true"
  | SBoolLit(false) -> "false"
  | SStringLit(s) -> s
  | STupleLiteral(e1, e2) -> "(" ^ string_of_sexpr e1 ^ ", " ^
string_of_sexpr e2 ^ ")"
  | STupleAccess(s1, s2) -> s1 ^ "." ^ string_of_int s2
  | SId(s) -> s
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^
string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr
el) ^ ")"
  | SNoexpr -> ""
                       ) ^ ")"

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "{\n" ^ String.concat "" (List.map string_of_sstmt stmts)
^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n";
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n";
  | SIf(e, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr
e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^
string_of_sstmt s
  | SPrint(expr) -> "print(" ^ string_of_sexpr expr ^ ");"

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " def" ^ " " ^
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd
fdecl.sformals) ^
  ")\n:\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "\n"

let string_of_sprogram (vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

8.6 codegen.ml

```
(* Code generation: translate takes a semantically checked AST
and
   produces LLVM IR
*)

module L = Llvm
open Sast
open! Ast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "JQER" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context
  and string_t   = L.pointer_type (L.i8_type context)
  and none_t     = L.void_type   context
    in

  let tuple_t = L.named_struct_type context "tuple_t" in
    L.struct_set_body tuple_t [| i32_t; i32_t|] false;

  (* Return the LLVM type for a JQER type *)
  let rec ltype_of_typ = function
      Int   -> i32_t
    | Bool  -> i1_t
    | None  -> none_t
    | String -> string_t
    | Tuple -> tuple_t
  in

  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    let global_var m (t, n) =
      let init = L.const_int (ltype_of_typ t) 0
      in StringMap.add n (L.define_global n init the_module) m
in
    List.fold_left global_var StringMap.empty globals in

  let printf_t : L.lltype =
    L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
  let printf_func : L.llvalue =
    L.declare_function "printf" printf_t the_module in
  let string_concat_t : L.lltype =
    L.function_type string_t [| string_t; string_t |] in
  let string_concat_f : L.llvalue =
    L.declare_function "string_concat" string_concat_t
the_module in

  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let function_decls : (L.llvalue * sfunc_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types =
        Array.of_list (List.map (fun (t,_) -> ltype_of_typ t)
fdecl.sformals)
      in let ftype = L.function_type (ltype_of_typ fdecl.styp)
formal_types in
```

```
      StringMap.add name (L.define_function name ftype
the_module, fdecl) m in
    List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname
function_decls in
    let builder = L.builder_at_end context (L.entry_block
the_function) in

    let int_format_str = L.build_global_stringptr "%d\n" "fmt"
builder
    and string_format_str = L.build_global_stringptr "%s\n"
"fmt" builder in

    (* Construct the function's "locals": formal arguments and
locally
       declared variables.  Allocate each on the stack,
initialize their
       value, if appropriate, and remember their values in the
"locals" map *)
    let local_vars =
      let add_formal m (t, n) p =
        L.set_value_name n p;
        let local = L.build_alloca (ltype_of_typ t) n builder
in
        ignore (L.build_store p local builder);
        StringMap.add n local m

      (* Allocate space for any locally declared variables and
add the
       * resulting registers to our map *)
      and add_local m (t, n) =
        let local_var = L.build_alloca (ltype_of_typ t) n
builder
        in StringMap.add n local_var m
      in

      let formals = List.fold_left2 add_formal StringMap.empty
fdecl.sformals
          (Array.to_list (L.params the_function)) in
      List.fold_left add_local formals fdecl.slocals
    in
    (* Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let lookup n = try StringMap.find n local_vars
      with Not_found -> StringMap.find n global_vars
    in

    (* Construct code for an expression; return its value *)
    let rec expr builder ((_, e) : sexpr) = match e with
        SIntLiteral i  -> L.const_int i32_t i
      | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
      | SStringLit s -> L.build_global_stringptr s "str"
builder
      | SNoexpr     -> L.const_int i32_t 0
      | STupleLiteral (x, y) ->
        let x' = expr builder x
        and y' = expr builder y in
        let t_ptr = L.build_alloca tuple_t "tmp" builder in
        let x_ptr = L.build_struct_gep t_ptr 0 "x" builder in
        ignore (L.build_store x' x_ptr builder);
        let y_ptr = L.build_struct_gep t_ptr 1 "y" builder in
        ignore(L.build_store y' y_ptr builder);
        L.build_load (t_ptr) "t" builder
      | SId s       -> L.build_load (lookup s) s builder
      | SAssign (s, e) -> let e' = expr builder e in
        ignore(L.build_store e' (lookup s) builder); e'
      | SBinop ((String,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
           Add     -> L.build_call string_concat_f [| e1'; e2'
|] "string_concat" builder
         | _ -> raise (Failure ("operation " ^ (string_of_op
op) ^ " not implemented")))
      | SBinop (e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
           Add     -> L.build_add
         | Sub     -> L.build_sub
         | Mult    -> L.build_mul
         | Div     -> L.build_sdiv
         | Mod     -> L.build_srem
         | And     -> L.build_and
         | Or      -> L.build_or
         | Equal   -> L.build_icmp L.Icmp.Eq
         | Neq     -> L.build_icmp L.Icmp.Ne
         | Less    -> L.build_icmp L.Icmp.Slt
         | Leq     -> L.build_icmp L.Icmp.Sle
         | Greater -> L.build_icmp L.Icmp.Sgt
         | Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
      | SUnop(op, ((_, _) as e)) ->
        let e' = expr builder e in
        (match op with
         | Neg                  -> L.build_neg
         | Not                  -> L.build_not) e' "tmp"
builder
      | STupleAccess (s1, s2) ->
        let t_ptr = (lookup s1) in
        let value_ptr = L.build_struct_gep t_ptr s2 ( "t_ptr")
builder in
        L.build_load value_ptr "t_ptr" builder
      | SCall (f, args) ->
        let (fdef, fdecl) = StringMap.find f function_decls in
        let llargs = List.rev (List.map (expr builder)
(List.rev args)) in
        let result = (match fdecl.styp with
              None -> ""
            | _ -> f ^ "_result") in
        L.build_call fdef (Array.of_list llargs) result builder
    in

    (* LLVM insists each basic block end with exactly one
"terminator"
       instruction that transfers control.  This function runs
"instr builder"
       if the current block does not already have a terminator.
Used,
       e.g., to handle the "fall off the end of the function"
case. *)
    let add_terminal builder instr =
      match L.block_terminator (L.insertion_block builder) with
        Some _ -> ()
      | None -> ignore (instr builder) in

    (* Build the code for the given statement; return the
builder for
       the statement's successor (i.e., the next instruction
will be built
       after the one generated by this call) *)
    let rec stmt builder = function
        SBlock sl -> List.fold_left stmt builder sl
      | SExpr e -> ignore(expr builder e); builder
      | SReturn e -> ignore(match fdecl.styp with
          (* Special "return nothing" instr *)
            None -> L.build_ret_void builder
          (* Build return statement *)
          | _ -> L.build_ret (expr builder e) builder );
        builder
      | SIf (predicate, then_stmt, else_stmt) ->
        let bool_val = expr builder predicate in
        let merge_bb = L.append_block context "merge"
the_function in
        let build_br_merge = L.build_br merge_bb in (* partial
function *)

        let then_bb = L.append_block context "then"
the_function in
        add_terminal (stmt (L.builder_at_end context then_bb)
then_stmt)
          build_br_merge;

        let else_bb = L.append_block context "else"
the_function in
        add_terminal (stmt (L.builder_at_end context else_bb)
else_stmt)
          build_br_merge;

        ignore(L.build_cond_br bool_val then_bb else_bb
builder);
        L.builder_at_end context merge_bb

      | SPrint e -> let (ty, _) = e in (match ty with
        | Int | Bool -> ignore(L.build_call printf_func [|
int_format_str ;
                  (expr builder e) |] "printf" builder);
builder
        | String -> ignore(L.build_call printf_func [|
string_format_str ;
                  (expr builder e) |] "printf" builder);
builder
        (* TODO: Temporary fix but Should deal with FLoat,
None, List *)
```

```
        | _ -> ignore(L.build_call printf_func [|
string_format_str ;
        (expr builder e) |] "printf" builder); builder
      )
      | SWhile (predicate, body) ->
        let pred_bb = L.append_block context "while"
the_function in
        ignore(L.build_br pred_bb builder);

        let body_bb = L.append_block context "while_body"
the_function in
        add_terminal (stmt (L.builder_at_end context body_bb)
body)
          (L.build_br pred_bb);

        let pred_builder = L.builder_at_end context pred_bb in
        let bool_val = expr pred_builder predicate in

        let merge_bb = L.append_block context "merge"
the_function in
        ignore(L.build_cond_br bool_val body_bb merge_bb
pred_builder);
        L.builder_at_end context merge_bb

      (* Implement for loops as while loops *)
      | SFor (e1, e2, e3, body) -> stmt builder
                                     ( SBlock [SExpr e1 ;
SWhile (e2, SBlock [body ; SExpr e3]) ] )
    in

    (* Build the code for each statement in the function *)
    let builder = stmt builder (SBlock fdecl.sbody) in

    (* Add a return if the last block falls off the end *)
    add_terminal builder (match fdecl.styp with
          None -> L.build_ret_void
        | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
  in

  List.iter build_function_body functions;
  the_module
```