# NEWBIE FINAL REPORT

John Anukem (jea2161) - Systems Architect
Clyde Bazile (cb3150) - Language Guru
Braxton Gunter (beg2119) - Tester
Terence Jacobs (tj2316) - Project Manager
Sebastien Siclait (srs2232) - Tester

Programming, Languages and Translators
COMS 4115

## Table of Contents

# 1. Introduction

Traditional high-level programming languages are often too cryptic and difficult for new users to understand. The goal with Newbie is to create a pseudo-code like programming language aimed to simplify the programming experience for beginner developers. This will allow new coders the ability to design, implement  and better understand common algorithms without the frustration of learning specific programming syntax. Our standard library will specifically allow for easy implementation of basic algorithms involving linked lists, graphs, and trees.

# 2. Language Tutorial

## 2.1 Setup

Newbie has been developed in OCaml which needs to be installed to be able to use the compiler. The best way to install is through OPAM(OCaml Package Manager). Using OPAM, OCaml and related packages and libraries can be installed. Follow the below commands for the basic setup. *Note:* The version of the OCaml llvm library should match the version of the LLVM system installed on your system.

```
$ sudo apt-get install -y ocaml m4 llvm opam
$ opam init
$ opam install llvm.3.6 ocamlfind
$ eval 'opam config env'
```

## 2.2 Getting Started

Inside the *newbie* directory, type *make*. This creates the newbie to LLVM compiler, newbie.native, which takes as input a newbie file with .noob extension and outputs LLVM code.

# 3. Language Reference Manual

## 3.1 Lexical Conventions

This will describe how the lexical analyzer breaks a file into tokens.

### 3.1.1 Character Set

Newbie uses the 7-bit ASCII character set. If an 8-bit character set is recognized Newbie will throw an error.

### 3.1.2 Line Terminators

#### 3.1.2.1 Physical Lines

Programs are divided into lines by recognizing line terminators. Line terminators are any of the standard platform line terminations:

- Unix form, ASCII LF (newline)
- Macintosh form, ASCII CR (return)
- Windows form, ASCII CR followed by the ASCII LF

The two characters CR immediately followed by LF are counted as one line terminator and all three terminator sequences can be used interchangeably.

#### 3.1.2.2 Logical Lines

The end of a logical is represented by the NEWLINE token. Statements cannot cross logical line boundaries except for when using specified explicit line joining rules.

#### 3.1.2.3 Explicit Line Joining

Two or more physical lines may be joined into logical lines using a single backslash (\) character per line. The backslash must not be in a string literal or comment. Blank lines, lines without whitespace or a comment terminates multi-line statements.

#### 3.1.2.4 Indentation

Leading whitespace (spaces and tabs) at the beginning of a logical line is used to compute the indentation level of the line, which determines the the grouping of statements.

Tabs are replaced by four spaces and the total number of characters up to and including the replacement characters must be a multiple of four even if a mixture of tabs and spaces are used. The total number of spaces preceding the first non-blank character determines the level of indentation. Indentation cannot be split over multiple physical lines. The whitespace up to the first backslash determines the indentation.

The indentation level of consecutive lines are used to generate INDENT and DEDENT tokens, using a stack as follows:

Zero is pushed to the stack before any line is read and will not be popped off. The numbers pushed onto the stack will always be strictly increasing from bottom to top. At the beginning of each logical line, the line's indentation level is compared to the top of the stack. If it is equal, nothing happens. If it is larger, the indentation level is pushed to the top of the stack and one INDENT token is generated. If it is smaller, the indentation level must be one of the numbers occurring on the stack. All larger indentation levels are popped off and a DEDENT token is generated for each. At the end of the file, a DEDENT token is generated for each number remaining on the stack that is larger than zero.

***Correctly formatted example, *=space, <tab>=tab*

```
define foo with params bar
* * * * if len(bar) equals 1
* * * * * * * * return bar
<tab>set new_bar to new [ ]
* * * * for i from 0 to len(bar)
* * * * <tab> for j from 0 to len(bar)
* * * * * * * * * * * * new_bar = new_bar + bar[ i : j ]
* * * * * * * * <tab>new_bar = new_bar + bar[ j : ]
* * * * * * * * new_bar = new_bar + bar[ i : ]
<tab>return new_bar
```

### 3.1.3 Tokens

There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. Spaces, tabs, and newlines can be used interchangeably to separate tokens. Some whitespace is always required to separate tokens.

### 3.1.4 Comments

Single line comments will be signified by two backslashes like //. Multiline comments will be enclosed by the following characters /* … */. Comments will not nest, and they will not occur within string or character literals. Comments are ignored by the syntax; they are not tokens.

### 3.1.5 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

| | | | |
|---|---|---|---|
| *and* | *If* | *While* | *Params* |
| *or* | *break* | *not* | *no* |
| *class* | *continue* | *equals* | *each* |
| *return* | *null* | *print* | *in* |
| *true* | *def* | *for* | *to* |
| *False* | *Else* | *With* | *from* |
| *Greater* | *less* | *than* | |

#### 3.1.6 Strings

Strings are represented by a sequence of characters surrounded by double quotes (”) and are immutable.

## 3.2 Syntax and Semantics

### 3.2.1 Statements

Newbie supports expression, declaration, control flow and loop statements.

#### 3.1.1 Expression Statements

An expression statement is a statement which must be evaluated.

#### 3.1.2 Declaration Statements

Variables in Newbie utilize type-inference (4.3) and are statically-typed. Must assign some variable to any given new variable.

set team_size to 5

#### 3.1.3 Control Flow Statements

The if statement is used to execute the block of statements in the if-clause when a specified condition is met. If the specified condition is not met, the statement is skipped over until any of the condition is met. If none of the condition is met, the expressions in the else clause (when specified) will be evaluated. Keywords are in bold.

**if** *expr*
  *statement*
**else if** *expr*
  *statement*
**else**
  *statement*

#### 3.1.4 Loop Statements

The while statement is used to execute a block of code continuously in a loop until the specified condition is no longer met. If the condition is not met upon initially reaching the while loop, the code is never executed. 'for each' can only be used on iterable structures, i.e., strings and list. Example:

**while** *expression*
    **statement**

**for each** *variable* **in** *list*
    *statement*

**for** *idx* **from** 1 **to** 100
    *statement*

## 3.3 Types

### 3.3.1 Primitive Data Types

Newbie will have 4 primitive data types: *bool, char, num, string*

#### *3.3.1.1 bool*

bool represents a simple boolean value, either true or false. They can be declared as follows:

**set** *PLT_is_great* **to true**
**set** *Edwards_is_boring* **to false**

#### *3.3.1.2 char*

*char* are single ASCII characters or an escape sequence followed by a character contained in single quotes. They can be declared as follows:
**set** p **to** ‘P’
**set** l **to** ‘l’
**set** t **to** ‘t’

It is a compile-time error for the character following the single character or escape sequence to be anything other than a '.

#### *3.3.1.3 num*

*num* represents both integers and floating point numbers. num types are 32-bits and follow IEEE 754 standard. Because there is no distinguishing factor between integers and floating point numbers, it is acceptable to declare numerics in a variety of ways:

**set** *plt* **to** 4115
**set** *edwards_age* **to** 57
**set** *grade* **to** 57.0

Given that *num* represents both integers and floats, boolean operations will ignore decimals as well. For example:

*grade* **equals** *edwards_age* // true

If at least one of the operands are floating point then the result will also be a floating point.

#### *3.3.1.4 string*

A *string* consists of a collection of characters enclosed in double quotes, such as ". . . ". It is possible to index through a statically declared string, but it is not possible to manipulate the data contained in the string. The string datatype supports all ASCII characters. To insert the " character in a string, use \" to avoid ending the string. Strings are iterable.

**set** *plt* **to** "COMS 4115"
**set** *hello* **to** "Hello world, PLT is a \"great\" class"

### 3.3.2 Lists

In newbie, we will have only one type of collection: Lists. A List is represented by a sequence of comma-separated elements enclosed in two square brackets [...]. Elements can be accessed by their positions in the list, beginning with the zero index. The List is a mutable data structure, which means that it supports functions to append, remove, or update its values. Lists can contain primitives or objects, but not a mix of both. Within a List of primitives, each element must be of the same type – for example, a List may not hold a collection of both *num* and *string* elements. Within a List of objects, all elements must be of the same type.

**set** L **to** ['N','E', 'W', 'B']
**set** L **to** L + 'I'    // ['N', 'E', 'W', 'B', 'I']
**set** L[4] **to** 'E'    // ['N', 'E', 'W', 'B', 'E']
L[7] // error

### 3.3.3 Type Inference

Newbie will contain a robust type inference system. Given an expression, it will be determine variable type at compile time. This will make it easier for users as they no longer need to declare parameter or variable types. As such, if we declare:

**set** coms **to** 4115
**set** class **to** "plt"
**set** class_is_great **to true**
**set** grade **to** 'p'

the compiler will interpret these variables as a *num*, *string*, *bool*, and *char* respectively. This will extend to more advanced data types,such as Lists, as well. For example, in the expression:

**set** team **to** ["Brax", "Clyde", "John", "Sebas", "TJ"]

team will be interpreted as a list of strings. This type inference will be done using the hindley milner method with a standardized notation for common data types.

### 3.3.4 Automatic Initialization

During compile-time, we will be identify all variables and their corresponding types. These variables will be automatically initialized to predictable default values. This means that variables do not need to be explicitly declared or initialized. Primitives are automatically initialized to a default value depending on their type. Lists are automatically initialized to their empty states.

| Type | Default Value |
|---|---|
| *bool* | False |
| *char* | null |
| *num* | 0 |
| *string* | null |
| *List* | [ ] |

## 3.4 Operators and Expressions

### 3.4.1 Assignment

The = operator or ( to ) can be used to assign the value of an expression to an identifier.

**set** x = 5
**set** x **to** 5 // same as above

With type inference, the variable x is automatically declared without having to declare the type. Assignment is right associative, allowing for assignment chaining.

a = b = 10 // Set both a and b to 10
**set** a **to** b **to** 10 // same as above

### 3.4.2 Operators

3.4.2.1 **Arithmetic Operators**

The arithmetic operators consist of +, - ,*, /, ^, and %. The order of precedence from highest to lowest is the ^ exponentiation operator, the unary - followed by the binary * and / followed by the binary + and -.

#### 3.4.2.2 Logical Operators

The logical operators consist of the keywords *and*, *or*, and *not*. The negation operator *not* keyword inverts true to false and vice versa. The logical operators can only be applied to boolean operands. The *and* keyword joins two boolean expressions and evaluates to true when both are true. The *or* keyword joins two boolean expressions and evaluates to true when both are true.

#### 3.4.2.3 String Operators

String access is denoted by square brackets enclosing an integer in the range of the length string. It returns the String indexed by the integer.

**set** a **to** "Hello world!"
**print** a[0] // prints "H"

String concatenation is denoted by the binary + operator.

**set** a **to** "Hello"
**set** b **to** " world!"
**set** c **to** a + b // "Hello world!"

#### 3.4.2.4 Relational Operators

Relational operators consist of >, <, >=, <=, == and != which have the same precedence. For primitive types, the equality comparison compares by value. == compare structurally while the is keyword compares physically. The == and != operators are valid for primitives and lists containing primitives.

**set** a = 1
**set** b = 1
**set print** a == b // True
**set print** a < b // False
**set print** a >= b //True

The above could be written as:

**set** a **to** 1
**set** b **to** 1
**print** a **equals** b
**print** a **less than** b
**print** a **greater than or equal to** b

#### 3.4.2.5 List Operators

Lists support the following operations:

*Length* - returns the length of the list

**set** a **to** [4, 5, 6]
length(a) // 3

*Access* - returns the element at an index

**set** a **to** [4, 5, 6]
a[0] // 4

*Update* - updates the element at an index

**set** a **to** [4, 5, 6]
**set** a[1] **to** 7
a[1] // 7

*Insertion* - inserts an element at an index and return it

**set** a **to** [4, 5, 6]
insert(a, 1, 8) // a == [4, 8, 5, 6]

*Removal* - removes the element at an index and return it

**set** a **to** [4, 5, 6]
remove(a, 0) // a == [5, 6]

*Push/Enqueue* - inserts an element at the end of the list and return it

**set** a **to** [4, 5, 6]
push(a, 7)  // a == [4, 5, 6, 7]
enqueue(a, 8) // a == [4, 5, 6, 7, 8]

*Pop/Dequeue* - removes the last element and returns it

**set** a **to** [4, 5, 6]
pop(a) // a == [4, 5]
dequeue(a) // a == [5]

# 4. Project Plan

## 4.1 Planning Process

Our team met twice a week: (1) on Sunday evenings to work on and plan tasks for the upcoming week ; and (2) on Friday afternoons to check in about the past couple days and to continue coding. We often had a number of milestones that we set to accomplish each week because different members of the team were involved in various aspects of the project. As we approached the project deadline, subsets of the team often met more frequently to make sure that pending tasks would be completed on time.

## 4.2 Project Timeline

The timeline of our project can be seen below:

| Date | Task |
|---|---|
| September 26th | Project Proposal |
| October 8th | Github Repo Created w/ Initial Commits |
| October 16th | Language Reference Manual |
| October 22nd | Scanner, Parser, AST for initial end-to-end |
| October 29th | Tokenizer w/ indent/dedent created |
| November 3rd | AST functional |
| November 8th | Codegen |
| November 17th | Hello World w/o Type Inference |
| December 1st | Semantics |
| December 3rd | Type Inference |
| December 17th | Language Complete |
| December 20th | Final Report |

## 4.3 Roles and Responsibilities

Throughout the semester, subsets of the team branched out to become more heavily involved in different components of the Compiler. We did a lot of pair-programming and helped each other to make sure that any changes made by a single team member made would be properly integrated with the rest of the code base. Because there was such significant overlap, there was a fluid division of responsibilities, although different members would oftentimes be more ”specialized” in an aspect of the language than another.

| Team Member | Responsibilities |
|---|---|
| John Anukem | Tester/AST, Standard Library, Codegen |
| Clyde Bazile | Language Guru/Codegen |
| Braxton Gunter | System Architect/Scanner, Type Inference, sast, semant |
| Terence Jacobs | Project Manager/Parser |
| Sebastien Siclait | Tester/Testing |

## 4.4 Software Development Environment

Operating Systems: Mac OS Systems
Languages: OCaml (used OPAM to install), C (runtime libraries)
Text Editor: Sublime, Vim
Version Control: Git, GitHub
Documentation: Google Docs

## 4.5 Project Log

e5b3625bca1a697f359d1890d24da1802f115ecc Merge pull request #42 from terencej23/dev
1266c40126e5accc4fa41d66200aa8c530c2a846 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
085decd5782d1072a9b313e96a7e3ac64ddf9691 Add binary search
ee7d4c6a710b7024a6348fcc80a162b0d55cc4b4 all tests functional
5af662c11f3051f1457bb753de9e6ce751c880db all but while loop tests updated
cc39c127fcfa674afaf1df235cadc4e7b317c163 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
d741f0602ae1b7691ebd83151dbca01b0cb4da00 new tests (precedence+parameters) dropped old tests
39ce81a4a6fd25deddcff7329b23ca9c1da5be79 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
13a0223d804c124913208e76510a476898ba1429 Add recursive binary search for presentation
ccc22d9f40ce7b78d102eaa1ae4964727815c273 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
bcf43ee10ffea8db37e821f490ecbae6b8e2d7c2 formatted rec_fib and cleaned up test script
5f90deb19aabbc75b74d5f50a6429765b2cbb848 Merge pull request #41 from terencej23/dev
f51bd9018c2ee11315fdbcbf3d6689bdd86fbf19 Merge pull request #40 from braxeatssnacks/master
79a495997d32761cf66422cc240943b971c9839e list can initialize
36c3352c76505f90d90cac57546d84d0c5f03f05 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
40fa1d5e8c1e6ebbefc854b3f7b2c0a1120aba4c Merge branch 'dev' of https://github.com/terencej23/newbie into dev
837b3a8066b9629b6d95b7160695092b2b022fef Merge pull request #39 from terencej23/dev
ccd2f69c0217f10ffb987d9d57c81c6dc3e901dd Merge pull request #38 from braxeatssnacks/master
f17db5d21801a05d37c8097f133ba3f3ce1349a7 further progress on codegen for list

e5842f3efc6c69d8c27a355027dc1f6946e2337a remove old test files
7e37535732d177ed9f2875829ea0c1e8f45f45b7 Merge pull request #37 from braxeatssnacks/master
0a047bf1b200093f76ea9aa049d8e7d8f2edb84d Merge branch 'dev' into master
225fe0d7e6a8aa761af5dbdc3cc32925d3396d17 Merge branch 'dev' into master
f29b4042682c8dfef8b2a5e489e292d74d7d64f4 DO NOT MERGE - list codegen doesn't work
aa761edc14c5234b7335978de9c33a4c9f87df8e Added test cases and test script. Changed format of test cases to start with 'test-*'
4100f7cb3c6477db640399a62848827904c125e0 Add recursive fib for testing
98c517f98e64ad458f8b712d1a6189ddd7b1df32 Add test for parameters
30afd54fabf7b4d71eef6bca46429adb0a6fdc4e Add return type to functions
c5f24946a4b27726c7185fc41dbc41954c8d395d add break for loop
09cdffcfb9a89114a073beb27a6621570c091d19 Merge pull request #36 from terencej23/dev
acc331ebe086b3c56a0cb545ce54e418a1b7b97e Merge pull request #35 from braxeatssnacks/master
40673999ac49c3bf629e6b7e43409addcb1667bd add while loop to parser
d053a733bca0b858a591d124f5d5e1af49277857 Merge branch 'master' of github.com:braxeatssnacks/newbie
60e91057419dbad3aaa16e0f73c931dbbe67b872 add while loop test file
a1261567aeb1ca7a0a8489c384ed5890132f4d6c Merge pull request #34 from terencej23/dev
f78937a2713df35e35e23e3274927628a1fba5d2 Merge pull request #33 from braxeatssnacks/master
b6e1d638485056dc6363810ead18c7ba4fc99acf add semantically checked while loop
b28d4f9ae7071a3e6d46b082ed43ed5f8f6fa313 Merge pull request #32 from terencej23/dev
8a4b3707023c029e54076a13a654e4b5139fd4e2 Merge pull request #31 from braxeatssnacks/master
16b7f437d0fc283c3527308189fc87c2393dcb9a merge codegen
57ed1657301eb0282f9b84034434189aac15c541 resolve binop edits
7340b620573bddf984199029be9284b584fdaaa7 add conditional statement support
f95fb0db71622cc93e13fc3357545a2d1a5b0cbb add conditional statement support
48f660d5ae041393f20159621683396a2a81ec01 updated test for binop
184eca61018cdc035bd30268ceb66aa85ede3bf9 Merge branch 'dev' of https://github.com/terencej23/newbie into dev
18f01f7ddef683af64b62be1634b78f59fd2eb10 Add binary operators
91a0010a23b2018329aacde56165761793446152 Merge pull request #30 from terencej23/dev
10d3861a5ed0bd68a7bcc6bfe300341f90d103c8 Merge pull request #29 from braxeatssnacks/master
78652dfb86c84994593ef72763fcb801a12cfcf6 add unop
054a1a5a5a0f16450ceb41378de69b864dab4c9f Merge pull request #28 from terencej23/dev
5ae2e1438385d466c9b0171a0a00230fc4bcc302 Add assignments
70ea976b6af8a6cfb92c59bd80de0be5c7d187d9 Merge branch 'master' of https://github.com/terencej23/newbie
e58ad4a0f8b80047b4709e277620a345f5e209b0 Merge pull request #27 from terencej23/master
0ed8e40b9614094b39a6e4c9913964f7c951a711 Merge pull request #26 from braxeatssnacks/master
60ad5d6faca5cf868d617a731bfc0dee943c541e remove makefile ifs, add toplevel definition of path to lli
d61c3b9a6c20cbfa02452e609cf920b9bbb71310 Merge branch 'master' of https://github.com/terencej23/newbie
3a65ba25660854b85a1e8cf1506d2794a7baad93 Merge pull request #25 from terencej23/dev
8c0457b5fbfe5ae21e0f282f9bc3cae40fa97997 Merge pull request #24 from braxeatssnacks/master
8c672a76d7435b6cff7c96c7864d42375ca99176 default command now builds and executes input file as llvm
09dacc64103c254f5a369a31be26b7cc6671cbc7 Add assignment test
ea71bed492223389a1636058e04a76602daa9e4e Print bools, floats, ints
1fa16ee2784c529224b0b8c9c6c82329d1379206 Merge branch 'master' of

https://github.com/terencej23/newbie
84657fcd563fd5dc7948f40d3324055e2bb5408a Merge branch 'master' of github.com:braxeatssnacks/newbie
da309db42898e668fd88c36ede2113e08fc3f76f Prints hello world!
4c07a56925161f292c09759528dee02e4c7cc199 Merge pull request #23 from terencej23/dev
4b56dea25c409545a16175a53fd0d43c3716db98 newbie.ml update
35b0533bc8646cda7335883f333e9e028db769ba Print LLVM code for 'hello world'
9cd205687978e504e2c349ceef13064a428c91b0 Merge pull request #22 from terencej23/dev
d124701920b0c12dea8a90079a2f153200ed2269 Merge pull request #21 from braxeatssnacks/master
2cff17a247fe26c55b4ded97913f8af7a9e8feee Merge branch 'dev' into master
d0331dc1e7baf7485fc851b4f41cf0cd7277fca5 Merge branch 'dev' into master
be48f0647c2f4755fba30fe7f422a24db3af793d codegen (sast)
c8833e1564af07f587681987b9b959204e9c0580 prime top-level for new codegen
ca90c05fd430b8819cdd2a6a02a6abd6e2b3be2d add clyde's codegen
256a26cefe06288170560a26fcbb1b68886258a3 Merge pull request #20 from braxeatssnacks/master
a61dbf8bf0f63748801c4b6abe1dae975a23c8ca comment out prime for iteration
89b22119b315e2b91b9a501c881d0b596762a158 Build function body
5d66638c98b70eb4e413746deba319550307e9ba Compiles successfully! Adds global and local vars.
9fc2aee30a114a565c37ba361de32a89a9608240 Print LLVM
c43379ea7931311b7a7b618f1bff8fa8c9c54dc3 Update
a5524792a3122d053493812ce4b7bab1cbfdd9e0 Print LLVM code
9508ef2e78a3d522e5c9f307a6efbb838e07ca0f remove extraneous file
ff92401d436d3acc1ca48e576d28b960127f9b54 Merge pull request #19 from braxeatssnacks/master
826792683179a1d9cb3ea9c728402606de0315d3 resolve conflicts
9ee8872d71f58c4cadcb8185bcd2e902ae6acad3 resolve conflicts
f638d3d03f706e05ca4d4b4e102f159a7b848001 fix entrypoint
ba826a14045bf4a1fb36b452ff14cf6078510cc3 added sformals
2dad3b2f071b95fe4dc8529572794b85e9445138 naming module fix -- segmentation fault in codegen
cba530af982e9c0955f51d361a3c24349da2395e Added Node Class to Makefile
d3150840c9900afd3566a31b422794b40dce6a2d semantic checking and type inference
b0214d8362fe151e0c72de9357f3508856d0cabd type inference sfdecl updating done
0e79e2b54e18479e5b2df0dcc12615152ac9ad32 WIP semantic checking won't build functions other than main
c49381ff53ec70a9f69f398a28bb12a6a871c442 semantic checking and type inference
0a51b7a560a5528454ef39c61ce6f4225a4289ec fix scanner newline
32c44a2f5d607acc9938fdc842adfe45f1798a5f added sformals
e4f8691284808ce34eee281bba98c0119dc59aae naming module fix -- segmentation fault in codegen
a258ce78701861ce63555162003911dc9b2ca32e interim edits -- need free parser
3a46f3ee5480bd63c32042d7ae7475898871f757 fix entrypoint
3f9daf2a69309c60a1cdb5fca60d7d35cd232815 Merge branch 'master' of github.com:braxeatssnacks/newbie
5114c01d0f019636147942e035b1f9462dfa2886 added sformals
8f6c7e86f88d822d6936f506bf126d81b9214e8d naming module fix -- segmentation fault in codegen
3a2962cbff0ea3a3d38925ad57277c40a8c4436e interim edits -- need free parser
62c477e42a25bb37ea189009a12951f362162b44 Merge branch 'master' of github.com:braxeatssnacks/newbie
5312a0a64a46e6f1147b9302283780086333ab00 Merge pull request #18 from braxeatssnacks/dev
831e7060e9d8b77846052e42722ebc358c1a1b58 Merge branch 'dev' into dev
3a3d223f0ec8fd7d7577f6c60619257950be25b7 Merge branch 'dev' into dev

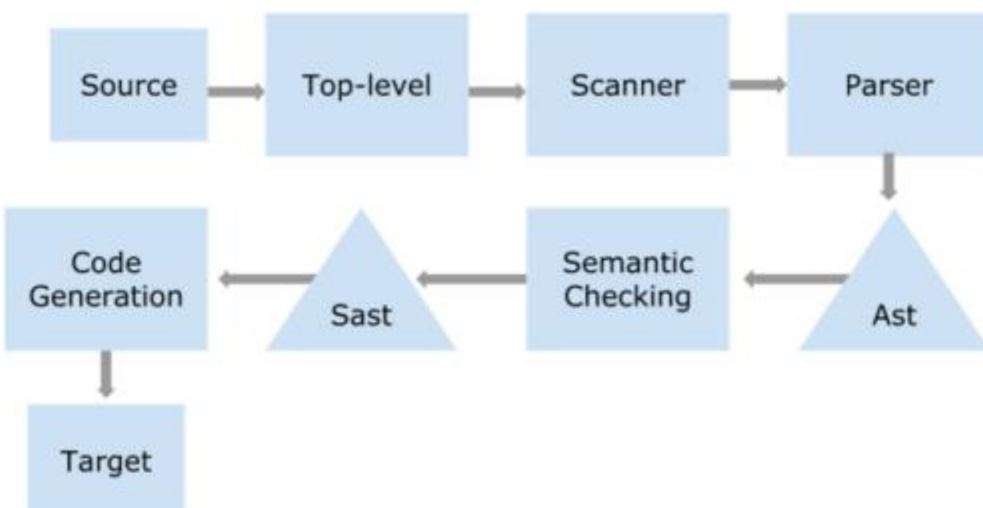
4c2744ff5df1133b12365c37b5933afa6a5a48e7 semantic checking and type inference
d2a6ea02ff93eb93a52da1406b594d85c0f9160e Merge pull request #17 from braxeatssnacks/dev
e431c87c33489e8799883b5aea2a43e52f876073 type inference sfdecl updating done
f57f9785a6a158ea6cf727d0d30d53801dfbbcf3 Merge pull request #16 from braxeatssnacks/dev
bf4966da999e1f26c5d0fbaabbb4fe732bd15680 WIP return type not bounced up in fdecl_to_sfdecl
13c26ae76b82ca80ef21b4f40e9a44af2c096541 Added Node Class
760d7b90a42628cfda25b7589f4d17e50baf07d2 Merge pull request #15 from braxeatssnacks/dev
00d587d15e15a246c31f7423079432f31d4b4ebb WIP semantic checking won't build functions other than main
489c0dea4a1d21d643b22cfd44f294cac6469988 semantic checking and type inference
29b669280805e5078417e612afbe2e787aa0230e fix scanner newline
b2419828cfb51163878442f1be208c19430b284a edits
4b4434832119ba8220935da45083fc55a78a4ccb added sformals
0489a4d4c809a396519b26a1aa615729ef64e428 naming module fix -- segmentation fault in codegen
f6ace3d626dbb05520d4af1bf4fb86c0efe85ff3 interim edits -- need free parser
fbdfcf7e1f18621db75d9292c9e8a4a09a3ed0aa Merge pull request #14 from terencej23/dev
4e28c151ceb6d5e987ab7a12fd163880f872edb0 Merge pull request #13 from braxeatssnacks/master
28be3db3530efa2299293f97d3c7ee5e0efb89de edits
eff486af70369ac4e732308a383f33ce17d044f6 added sformals
9d9342d34e6414e64416495076ee039561dea29a fixed sformals error
e66285b75f38ec529ff29c4a54e28b568c0ecb3b naming module fix -- segmentation fault in codegen
33778c3d4f7b0ac78d73f7bc3df582d1776169ce added sformals
ff3a1d6b348ed92c23ec9e8ce739ebf02010f603 Merge branch 'master' of github.com:braxeatssnacks/newbie
045ad84ed20bc8a55ed3d81b12d389daff78fad1 interim edits -- need free parser
6db80d4ee2d882ef65aad19e3dc3f216680cb29c interim edits -- need free parser
3571eb4d48485c9c8c43a512c1d16c87ec629641 Merge pull request #12 from braxeatssnacks/master
87e18e336621be01cf3640067bccdfd9b97ff693 parser requires main method
a9bbd175f478b151913f94c66b8b2c1f1ddc9d3c Merge branch 'master' of github.com:braxeatssnacks/newbie
c511b70ab14f57a260771e7f21f2cb9d8344d2ce AST IS FUNCTIONAL
6f282d0138e33d5ba386f5edaf64a1961cc19f6c one bug left; commented out 'for' stmt in ast.ml
23bfdd3d78e507701a4fbf1102b4e093deb19459 unfinished - code-gen piece for hello world
868da4d1e2024af1c41d6865c4bebe581a18b8fa AST IS FUNCTIONAL
a4bbd8f7d11e7d4fb771802df8667c4822c9f081 one bug left; commented out 'for' stmt in ast.ml
a707e1644f170ac82d816b28bc0de2dd30e1d217 Merge pull request #11 from braxeatssnacks/integrate-codegen
c49df262471b3430b0e4dd08f9a506b18e0dfe33 Merge branch 'dev' into integrate-codegen
becc829a3394b59535047fbf1f055958f48b0eb9 Merge branch 'dev' into integrate-codegen
79538a9504c65362246af457373ba1332ffec5c6 unfinished - code-gen piece for hello world
11caff8e0ec2d6da80074c9e3055231536296fa4 Merge pull request #10 from braxeatssnacks/master
f4b69057085e4a8cddf80a4e281440bfc6b96cce add intermediary cache for scanner token list to parser
57e26325efebc5c6cb8ba17415b376df10f4ac54 add intermediary cache for scanner token list to parser
ad99a52268b97e7878142e7d11343240e754c360 Merge pull request #9 from clyde-bazile/master
51a3c8f3b97b35b371afa2cdb7ea7d55a624bad9 init codegen
3d1516ff40f5a576205ae8cad148769b78d24d7c Merge pull request #8 from braxeatssnacks/master
265939d5ef05933caeee7b9472d6e232e82152c2 we are in a bind -- output of scanner shouldnt be a list of tokens -> what is it?
027bfdf80600dfeba59ad213b00c4d04a3105917 files are primed for compilation --> scanner needs to

output Lexing.lexbuf function -> Parser.tokens instead of list of Parser.tokens
1c522a1bd13d38421fa50fd9a52b26d599978ef9 get rid of shift/reduce conflicts with associativity
a77e72e925ee45d137b07d2618d6dff121ff9c42 add newline character
52840ea7776c9a9d19c4283beee36f06365146a5 parser file added
23a732704526221c046558b2aae05aee848b90e0 Merge pull request #6 from braxeatssnacks/master
fb85428309434b54d047da682971d00b8d64faa0 add overarching entrypoint newbie.ml -- makefile not currently working
23d4252ab23dd3521ae9c800df49ce13e0126255 Deleted something
c78f7d19c9df97fd7b9d814a99c26a20312658f1 Added something
d04f6b4647600f3202750767b9bb3a071ec49ca6 Merge pull request #5 from johnanukem/master
ef7b29f640fa39de5583cc76dedfd51ed4fdbf17 Added ast
5037e576f9894da3de576ad68752e0dd17889cc5 added microc directory
8db10a34bcad569117952560cf4d3589d78b0e8a add print keyword
f0610f56a99b4c29d671ab43a82db28a4262bf52 token resolution
29ebd1272a71d5929f39f291291c14e7a6f5fd93 Merge pull request #1 from braxeatssnacks/semantic_setup
413d330de73eb3ec694091e299353a633d008097 resolve conflicts
f4b992be9e24ecc12251b046d69b1e2aa28daafc make literals distinction
1c0afa85cd4cc2deb890cf6f02bf5792e46a681c Merge branch 'master' of github.com:braxeatssnacks/newbie
7462e390e29f4d974a36c1fa4d35d6b16cc1eec8 Update README.md
2b0551d8f57d53ae42a9893b2471f16b5faabc48 add comment functionality
f8d6edac81ba1340b3e0c050e863858d21feab20 add comment functionality
c141ca4e05c4217163a177c3dcab8abb52566589 Merge branch 'master' of https://github.com/terencej23/newbie
b5ddd73aeef32620f8ce2755a045d48f6bec666e Merge branch 'master' of https://github.com/terencej23/newbie
289614cdf609c5c688c0c78f56efe6bb2d1d978e Added ast
31875d8cf6101b6d2b76477ab50d15b81ae1a5c7 Merge pull request #4 from braxeatssnacks/semantic_setup
0aec378c2f654e946fbbd4220abd194915630b67 fix exhaustive pattern matching
b2523478d626f335d0c7339bd2ea9db6a9b02b32 Merge branch 'semantic_setup' of github.com:braxeatssnacks/newbie into semantic_setup
f1e790882d3727aa32084c18adeaa2772f252481 read up young champs
320ac658e06027139608860d7d5a9435057dee18 added keyword
33c741757d81f1debc6a2466edc7d3ccc4097f05 add delimit tests
e0740858a6cfda05e49f9ac5673ff6609c723af6 generate indent and dedent tokens based off \t
6d0217880473e571b593525e578ccc4331b6ecaa generic makefile for scanner compilation
bd2dee3954f33e0e6939feba77265ed7858f927e add few testcases
c6d83afc7be1a11ef71871ab13c2e0f282bb4fa2 allow passing of cmd_line arg path/to/file for testing
eaafaf546cf7289d21cbe85e0ffd1c9ac0e2d6e5 add more token recognition; prime INDENT/DEDENT
a939cd59793e20b4b2f242b6c6bed43a06aa789b add string support
9d92eddfe41538c3ea6b8551b74fb7366dfc1649 add output capacity
1460d79c7a47d5ea9e53d827019f224c5f0d93c9 add basic tokens and matching
5e2deece2899ba259473ab01bd800b36a78b4e98 add verbose err reporting functions
2021ff0d7f44f9ccc4a3ce301d060ec564cad336 setup keyword scanning
f0f1f726afb9b6daa092f478c28845519d8b574a read up young champs
baa58cd17000bef25eb92a24df54dfefa2db3386 added keyword
64cf4f4b8a83b2f7344dcf6b20152edd04fea3fa add delimit tests

f05a4c7c818abcf3bd362c937c1e129710d16458 generate indent and dedent tokens based off \t
120e351214842b4607c0a95c622ef4dd423412a5 Update README.md
7ba7dea1f984c2c5f6320c3784149be05d98409d Merge pull request #2 from johnanukem/master
62d10be01050229ef672fa71f9d6a17bb40d9d38 hello
53e81389ff8fc9b02fec07960ac2104423c50b50 Delete Test.txt
3fa0d40cd6d114950f064497e0255a36db3bd5f4 Merge pull request #1 from clyde-bazile/master
96a559493a5d683b6226f4ba4ef34dbdf937cc1b Create Test.txt
0e43e76b1d5e0981bcb7d821a9f96348c9741359 generic makefile for scanner compilation
986462e79c04b27638df8bdd49a8ac9891458398 add few testcases
3595180c39c9abfd7046fbdcddf6c5588a4c1c0c allow passing of cmd_line arg path/to/file for testing
369f58b14188e84e836c3155677181e692cfbe91 add more token recognition; prime INDENT/DEDENT
f513ba5684d8989f8d3e45b945f32527a07d095c add string support
6629e47a3e0df1049e0aee0e2a69746e118ca500 add output capacity
9cd7b0ac365906733bb58aab287a47c653a9fd81 add basic tokens and matching
e1f1d56afdf5242a5fd99df481581d0229ce30f2 add verbose err reporting functions
d64af6d1216b0833be3384796a334a6b57423638 setup keyword scanning
5fba701256b84cefc066accdc80e10d2b64f0e2d Update README.md
d2ff15343dbd58056198becb80c37468f0a9f3c6 Update README.md
40d8401b4c3e361d73c6fe77a5ef1c6bc5a48fd7 Add files via upload
2c6f7179a2289d05eb8cb65d1a19538f340e80aa Initial commit

# 5. Architectural Design

## 5.1 Block Diagram




## 5.2 Components

### 5.2.1 Scanner (Braxton)

The scanner takes the input file and tokenizes it into keywords, identifiers and literals. It also ignores and removes all comments in the input file. If there is anything in the input file that is not syntactically valid, the scanner will throw an error.

### 5.2.2 Parser and AST (John, Terence)

The parser takes a series of tokens generated by the scanner and and generates an Abstract Syntax Tree (AST). If the code is successfully parsed, then it is syntactically correct.

### 5.2.3 Semantic Checker and SAST (Braxton, Sebastien)

The semantic checker takes the Abstract Syntax Tree generated by the parser and, with the new types defined in the sast.ml, translates the AST into a Semantically Checked Abstract Syntax Tree (SAST). The semantic checker is responsible for resolving all types of the previously typeless functions and variables from the program in the input file. It is also responsible for making sure that all parts of the program (expressions, statements, and functions) are semantically valid. The semantic checker works by using an environment variable, an aggregator argument to every function within the semant file, that tracks global variables, functions within the file, functions that have been semantically checked, the current local variables for a function, whether the current function's return type has been set, and the function's return type. This environment variables allowed me to perform full type inferencing

### 5.2.4 Codegen (Clyde)

The code generator takes the SAST produced by the semantic checker, and generates LLVM IR. This file is mainly responsible for generating LLVM code for all parts of the program.

## 6. Test Plan

We modeled our tests off of the microc tests. The automated test suite was critical as our compiler had many parts with dependencies on each other. As these components all had to be worked on in parallel with one another and the test suite was crucial for ensuring that any changes to one part did not also break another part

### 6.1 Test Suite

We had several tests found in our tests directory. After every compile, the test script was run to make sure that the latest compile did not break any previous functionality.

### 6.1.1 Unit Testing

As the compiler was developed, tests were written every time a new feature or component was added to verify that it worked correctly. For each new feature, 1-2 tests were written and added to the tests directory.

### 6.1.2 Integration Testing

Integration testing involved running programs written in the Newbie language and comparing it to the expected output. This runs through the entire pipeline. Our testing here focused on actual algorithms that the programmer may write.

### 6.2 Test Automation

The testall.sh test script in the tests directory compiles, runs, and links all the files within the tests directory. The files must begin with either ”test-” or ”fail-” and must have the extension ”.n00b”. They must also be accompanied by a file with the same base name and the extension ”.out”.

# 7. Lessons Learned

## 7.1 John Anukem

I think that I realized that not all programming languages are created equally. While I might have a grasp on imperative programming languages, functional programming proved to be an extremely difficult concept to grasp. If someone could've told me advice that would matter more than anything else, it would be that making a compiler is much different than making any other programming project, because it requires you to test in full rather than in parts.

## 7.2 Clyde Bazile

The most important lesson I learned was  how to tackle a large project using unfamiliar concepts and technologies. I think often times, we were focusing on trying to meet deadlines which caused us to try to take shortcuts and setting us back even further instead of taking our time and focusing on understanding and planning out the implementation. I also learned not to underestimate the difficulty of writing a language, but also not to underestimate my ability to do so. Adding simple features like function parameters and recursion aren't as easy as it may seem at first, but you shouldn't be daunted by the challenge. It be a lot of work but in the end, it'll be very rewarding.

## 7.3 Braxton Gunter

One of the most important lessons I learned was how difficult it is to build a multi-level application like a compiler, especially in a new language. For this project there were so many different files all operating in sync with each other, and oftentimes it was very frustrating to follow an error through across the different classes. I also learned how to effectively organize my workspace on the computer to efficiently debug errors.

## 7.4 Terence Jacobs

I think the most important lesson I learned was that as a manager, I need to be on top of scheduling and planning out work for everyone in the group. I considered myself and my colleagues to all be responsible for the whole project, but I think that as a manager I should have taken more ownership of the project at the start. Then it would be my job to ensure that we all fairly split the work.

## 7.5 Sebastien Sicalit

I spent way too much time looking into continuous integration and I was unable to get it going. I had used a continuous integration system in a previous internship but setting one up was an entirely different beast. In big part because many of the continuous integration systems I looked into were set up to be optimized for certain languages and OCaml isn't one of them. A lesson learned for me from this experience is to give respect to big problems and to be comfortable pivoting when things are going south.

## 7.6 Future Teams

Start early……. very early.
Don't underestimate how long something will take, even if you think it'll be easy.
Every line of OCaml counts….. EVERY LINE
If you fail to test, your tests will fail.

# 8. Appendix

## 8.1 Makefile

---

```
OBJS = exceptions.cmx ast.cmx sast.cmx semant.cmx codegen.cmx parser.cmx scanner.cmx
newbie.cmx

.PHONY: all
all: newbie print.o

newbie: $(OBJS)
	ocamlfind ocamlopt -linkpkg -fPIC -package llvm -package llvm.analysis $(OBJS) -o
newbie

scanner.ml: scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli: parser.mly
	ocamlyacc parser.mly


%.cmo: %.ml
	ocamlc -c $<
%.cmi: %.mli
	ocamlc -c $<
%.cmx: %.ml
	ocamlfind ocamlopt -c -package llvm $<


newbie.cmo: semant.cmo scanner.cmo parser.cmi codegen.cmo sast.cmo ast.cmo
exceptions.cmo
newbie.cmx: semant.cmx scanner.cmx parser.cmx codegen.cmx sast.cmo ast.cmx
exceptions.cmx
scanner.cmo: parser.cmi exceptions.cmo
scanner.cmx: parser.cmx exceptions.cmx
parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
parser.cmi: ast.cmo
ast.cmo:
ast.cmx:
sast.cmo: ast.cmo
```

```
sast.cmx: ast.cmx
semant.cmo: sast.cmo ast.cmo exceptions.cmo
semant.cmx: sast.cmx ast.cmx exceptions.cmx
codegen.cmo: semant.cmo sast.cmo ast.cmo
codegen.cmx: semant.cmx sast.cmo ast.cmx
exceptions.cmo:
exceptions.cmx:

node : node.c
	cc -o strcmp -DBUILD_TEST node.c

.PHONY: clean
clean:
	ocamlbuild -clean
	rm -rf *.diff newbie scanner.ml parser.ml parser.mli
	rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.s *.ll *.out *.exe
	rm -rf print

print: print.c
	gcc -o print print.c
```

---

## 8.2 newbie.ml

```
(* top-level of newbie compiler *)

type actions = TOKEN  | AST | SAST | LLVIM_IR | COMPILE | DEFAULT

let lli_exec = "/usr/local/opt/llvm/bin/lli"

let main () =
  let is_tag str =
    String.get str 0 = '-'
  in
  let action =
    if (Array.length Sys.argv = 3 && is_tag Sys.argv.(1)) then                  (* tag arg *)
      List.assoc Sys.argv.(1) [
        ("-t", TOKEN)     ; (* output tokens only *)
        ("-a", AST)       ; (* output ast only*)
        ("-s", SAST)      ; (* output sast only *)
        ("-l", LLVIM_IR)  ; (* generate, do NOT check *)
        ("-c", COMPILE)     (* generate, check LLVM IR *)
      ]
    else if (Array.length Sys.argv = 2 && not (is_tag Sys.argv.(1))) then       (* no tag *)
```

```
    DEFAULT
  else                                                            (* error *)
    (raise (Exceptions.InvalidExecFormat("invalid format ./newbie [-t] path_to_file")))
in
let fname =
  match action with
    DEFAULT     -> Sys.argv.(1)
  | _           -> Sys.argv.(2)
in
let lexbuf = Lexing.from_channel (open_in fname) in
let tokens = List.rev (Scanner.token [] lexbuf) in
let cache =                                                        (* finesse *)
  let l = ref [] in
  fun lexbuf ->
    match !l with
    | hd::tl  -> l := tl ; hd
    | []      ->
      match (tokens) with
      | hd::tl  -> l := tl ; hd
      | []      -> (raise (Exceptions.MalformedTokens))
in
let gen_ast = Parser.program cache lexbuf in
let gen_sast = Semant.check gen_ast in
match action with
  TOKEN          -> print_endline (Scanner.string_of_tokens tokens)
| AST            -> print_endline (Ast.string_of_program gen_ast)
| SAST           -> print_endline (Sast.string_of_sprogram gen_sast)
| LLVIM_IR       -> let the_module = Codegen.translate gen_sast in
                    print_endline (Llvm.string_of_llmodule the_module)
| COMPILE        -> let the_module = Codegen.translate gen_sast in
                    Llvm_analysis.assert_valid_module the_module ;
                    print_string (Llvm.string_of_llmodule the_module)
| DEFAULT        -> let the_module = Codegen.translate gen_sast in
    Llvm_analysis.assert_valid_module the_module ;
    let llvm_code = Llvm.string_of_llmodule the_module in
    let ll_fname =
      let basename = Filename.basename fname in
      (Filename.remove_extension basename) ^ ".ll"
    in
    let outFile = open_out (ll_fname) in
    Printf.fprintf outFile "%s\n" (llvm_code) ; close_out outFile ;          (* write out to file *)
    ignore(Sys.command (Printf.sprintf "%s %s" lli_exec ll_fname))          (* run llvm interpreter
- set var in makefile *)
```

```
let _ = Printexc.print main ()
```

## 8.3 ast.ml

```
type binop = Add | Sub | Div | Mod | Mult | Or | And | Lt | Leq | Gt | Geq | Eq
type typ = Int | Void | String | Float | Bool
type unop = Neg | Not
type datatype = Datatype of typ | Listtype of typ

type expr =
  | StrLit of string
  | IntLit of int
  | FloatLit of float
  | BoolLit of bool
  | Binop of expr * binop * expr
  | Unop of unop * expr
  | Id of string
  | Call of string * expr list
  | Noexpr
  (* list *)
  | List of expr list
  | ListAccess of string * expr
  | ListSlice of string * expr * expr

type stmt =
  | Block of stmt list
  | If of expr * stmt * stmt
  | While of expr * stmt
(*  | For of expr * expr * expr  *)
(*  | Iter of expr * expr *)
  | Break
  | Expr of expr
  | Return of expr
  | Assign of string * expr
  (* list *)
  | ListReplace of string * expr * expr

type fdecl =  {
  fname : string;
  formals : string list;
  body : stmt list;
}

type global =  string * expr
```

```
type program = global list * fdecl list


(* Pretty-printing functions *)

let string_of_op = function
    Add   -> Printf.sprintf "+"
  | Sub   -> Printf.sprintf "-"
  | Mult  -> Printf.sprintf "*"
  | Div   -> Printf.sprintf "/"
  | Mod   -> Printf.sprintf "%%"
  | Eq    -> Printf.sprintf "="
  | Lt    -> Printf.sprintf "<"
  | Leq   -> Printf.sprintf "<="
  | Gt    -> Printf.sprintf ">"
  | Geq   -> Printf.sprintf ">="
  | And   -> Printf.sprintf "and"
  | Or    -> Printf.sprintf "or"

let string_of_uop = function
    Neg   -> Printf.sprintf "-"
  | Not   -> Printf.sprintf "!"

let rec string_of_expr = function
    IntLit(d)              -> Printf.sprintf "%d" d
  | FloatLit(f)            -> Printf.sprintf "%f" f
  | StrLit(s)              -> Printf.sprintf "\"%s\"" s
  | BoolLit(true)          -> Printf.sprintf "true"
  | BoolLit(false)         -> Printf.sprintf "false"
  | Id(s)                  -> Printf.sprintf "%s" s
  | Binop(e1, o, e2)       -> Printf.sprintf "%s %s %s"
                              (string_of_expr e1) (string_of_op o) (string_of_expr e2)
  | Unop(o, e)             -> Printf.sprintf "%s %s"
                              (string_of_uop o) (string_of_expr e)
  | Call(f, e)             -> Printf.sprintf "%s(%s)"
                              f (String.concat ", " (List.map string_of_expr e))
  | Noexpr                 -> Printf.sprintf "noexpr"
  (* list *)
  | ListAccess(s, e)       -> Printf.sprintf "%s[%s]" s (string_of_expr e)
  | ListSlice(s, e1, e2)   -> Printf.sprintf "%s[%s:%s]"
                              s (string_of_expr e1) (string_of_expr e2)
  | List(e_l)              -> Printf.sprintf "[%s]"
                              (String.concat ", " (List.map string_of_expr e_l))
```

```
let rec string_of_stmt = function
    Block(s)                 -> Printf.sprintf "%s"
                             (String.concat "\n\t" (List.map string_of_stmt s))
  | Expr(e)                  -> Printf.sprintf "%s"
                             (string_of_expr e)
  | Return(e)                  -> Printf.sprintf "return %s"
                             (string_of_expr e)
  | If(e, s, Block([]))        -> Printf.sprintf "if (%s)\n\t%s"
                             (string_of_expr e) (string_of_stmt s)
  | If(e, s1, s2)            -> Printf.sprintf "if (%s)\n\t%s\nelse\n\t%s"
                             (string_of_expr e) (string_of_stmt s1) (string_of_stmt s2)
  | While(e, s)                -> Printf.sprintf "while (%s)\n\t%s"
                             (string_of_expr e) (string_of_stmt s)
  | Assign(s, e)               -> Printf.sprintf "set %s to %s"
                             s (string_of_expr e)
  | ListReplace(s, e1, e2)      -> Printf.sprintf "set %s[%s] to %s"
                             s (string_of_expr e1) (string_of_expr e2)
  | Break                    -> Printf.sprintf "break\n;"

let string_of_assign (s, e) = Printf.sprintf "set %s to %s" s (string_of_expr e)

let string_of_fdecl fdecl = Printf.sprintf "define function %s with params (%s)\n\t%s"
  (fdecl.fname)
  (String.concat ", "   (List.map (fun x -> x) fdecl.formals))
  (String.concat "\n\t" (List.map string_of_stmt fdecl.body))

let string_of_program (vars, funcs) = Printf.sprintf "%s\n\n%s"
(String.concat "\n" (List.map string_of_assign vars))
(String.concat "\n" (List.map string_of_fdecl funcs))
```

---

## 8.4 sast.ml

```
open Ast

type sexpr =
    SStrLit of string * datatype
  | SIntLit of int * datatype
  | SFloatLit of float * datatype
  | SBoolLit of bool * datatype
  | SBinop of sexpr * binop * sexpr * datatype
  | SUnop of unop * sexpr * datatype
  | SId of string * datatype
  | SCall of string * sexpr list * datatype
  | SNoexpr
  (* list *)
```

```
  | SList of sexpr list * datatype
  | SListAccess of string * sexpr * datatype

type sstmt =
    SBlock of sstmt list
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt
(*
  | SFor of sexpr * sexpr * sexpr * stmt
  | SIter of sexpr * sstmt
*)
  | SAssign of string * sexpr * datatype
  | SExpr of sexpr * datatype
  | SReturn of sexpr * datatype
  (* list *)
  | SListReplace of string * sexpr * sexpr * datatype
  | SBreak

type sfdecl = {
  styp: datatype;
  sfname: string;
  slocals: (string * datatype) list;
  sformals: (string * datatype) list;
  sbody: sstmt list;
}

type sglobal = string * sexpr * datatype
type sprogram = sglobal list * sfdecl list


(* pretty printing functions *)

let string_of_sop = function
    Add   -> Printf.sprintf "+"
  | Sub   -> Printf.sprintf "-"
  | Mult  -> Printf.sprintf "*"
  | Div   -> Printf.sprintf "/"
  | Mod   -> Printf.sprintf "%%"
  | Eq    -> Printf.sprintf "="
  | Lt    -> Printf.sprintf "<"
  | Leq   -> Printf.sprintf "<="
  | Gt    -> Printf.sprintf ">"
  | Geq   -> Printf.sprintf ">="
  | And   -> Printf.sprintf "and"
```

```
  | Or    -> Printf.sprintf "or"

let string_of_suop = function
    Neg   -> Printf.sprintf "-"
  | Not   -> Printf.sprintf "!"

let rec string_of_typ = function
    Datatype(String)       -> Printf.sprintf "str"
  | Datatype(Int)          -> Printf.sprintf "int"
  | Datatype(Float)        -> Printf.sprintf "float"
  | Datatype(Bool)         -> Printf.sprintf "bool"
  | Datatype(Void)         -> Printf.sprintf "void"
  | Listtype(typ)          -> Printf.sprintf "%s" (string_of_typ @@ Datatype(typ))

let rec string_of_sexpr = function
    SIntLit(d, _)            -> Printf.sprintf "%d" d
  | SFloatLit(f, _)          -> Printf.sprintf "%f" f
  | SStrLit(s, _)            -> Printf.sprintf "\"%s\"" s
  | SBoolLit(true, _)        -> Printf.sprintf "true"
  | SBoolLit(false, _)       -> Printf.sprintf "false"
  | SId(s, _)                -> Printf.sprintf "%s" s
  | SBinop(se1, so, se2, _)   -> Printf.sprintf "%s %s %s"
                             (string_of_sexpr se1) (string_of_sop so) (string_of_sexpr se2)
  | SUnop(so, se, _)         -> Printf.sprintf "%s %s"
                             (string_of_suop so) (string_of_sexpr se)
  | SCall(s, se, _)          -> Printf.sprintf "%s(%s)"
                             s (String.concat ", " (List.map string_of_sexpr se))
  | SNoexpr                  -> Printf.sprintf "noexpr"
  (* list *)
  | SListAccess(s, se, typ)    -> Printf.sprintf "%s[%s] -> %s"
                              s (string_of_sexpr se) (string_of_typ typ)
  | SList(se_l, typ)          -> Printf.sprintf "<%s>[%s]"
                              (string_of_typ typ) (String.concat ", " (List.map string_of_sexpr se_l))

let rec string_of_sstmt = function
    SBlock(ss)                -> Printf.sprintf "%s"
                               (String.concat "\n\t" (List.map string_of_sstmt ss))
  | SExpr(se, _)              -> Printf.sprintf "%s"
                               (string_of_sexpr se)
  | SReturn(se, _)             -> Printf.sprintf "return %s"
                               (string_of_sexpr se)
  | SIf(se, ss, SBlock([]))     -> Printf.sprintf "if (%s)\n\t%s"
                               (string_of_sexpr se) (string_of_sstmt ss)
  | SIf(se, ss1, ss2)          -> Printf.sprintf "if (%s)\n\t%s\nelse\n\t%s"
```

```
                              (string_of_sexpr se) (string_of_sstmt ss1) (string_of_sstmt ss2)
  | SWhile(se, ss)               -> Printf.sprintf "while (%s)\n\t%s"
                               (string_of_sexpr se) (string_of_sstmt ss)
  | SAssign(ss, se, _)          -> Printf.sprintf "set %s to %s"
                              ss (string_of_sexpr se)
  | SListReplace(s, se1, se2, _)   -> Printf.sprintf "set %s[%s] to %s"
                               s (string_of_sexpr se1) (string_of_sexpr se2)
  | SBreak                        -> "break;\n"

let string_of_sassign (s, se, _) = Printf.sprintf "set %s to %s" s (string_of_sexpr se)

let string_of_sfdecl sfdecl = Printf.sprintf "define function %s with params (%s) -> <%s>\n\t%s"
  (sfdecl.sfname)
  (String.concat ", "   (List.map (fun (name, typ) -> Printf.sprintf "<%s>: %s" (string_of_typ typ)
name) sfdecl.sformals))
  (string_of_typ sfdecl.styp)
  (String.concat "\n\t" (List.map string_of_sstmt sfdecl.sbody))

let string_of_sprogram (vars, funcs) = Printf.sprintf "%s\n\n%s"
  (String.concat "\n" (List.map string_of_sassign vars))
  (String.concat "\n" (List.map string_of_sfdecl funcs))
```

## 8.5 scanner.mll

```
{
  module L = Lexing
  module B = Buffer
  module E = Exceptions
  open Parser

  let buf_size = 100 (* default buffer size *)

  let indent_stack = Stack.create ()
  let () = Stack.push 0 indent_stack

  let de_indent_gen str token_stream stack =
    let char_count str target = (* count target char occurences in str *)
      let explode str = (* explode str into list of char *)
        let rec helper i lst =
          if i < 0 then lst else helper(i-1) (str.[i] :: lst)
        in helper (String.length str - 1) []
      in
      let l = explode str in
      List.fold_left (fun acc elem -> if (elem = target) then (acc+1) else acc) 0 l
    in
```

```
    let curr_indent = char_count str '\t' in (* TODO: make tab or spaces *)
    let prev_indent = Stack.top stack in
    let cmp_tabs top curr = (* compare stack peek with with current num -> # tokens to gen and
of ? type *)
      if (curr != top) then
        if (curr > top) then
          let () = Stack.push curr stack in
          1
        else
          let rec dedent_track count check = (* give int < 0 -> # of DEDENT TOKENS TO
GENERATE *)
            if (Stack.is_empty stack) then
              raise (Failure("unexpected indentation"))
            else
              let popped = Stack.pop stack in
              if (check = popped) then
                count
              else
                dedent_track (count-1) popped
          in
          dedent_track (-1) (Stack.top stack)
      else
        0
    in
    let to_generate = cmp_tabs prev_indent curr_indent in
    let rec token_gen stream = function (* add appropriate tokens to stream based on int arg ->
updated stream *)
        0           -> stream
      | _ as int_     ->
        if (int_ > 0) then
          token_gen (INDENT :: stream) (int_ - 1)
        else
          token_gen (DEDENT :: stream) (int_ + 1)
    in
    token_gen token_stream to_generate

  let eof_dedent token_stream stack = (* add DEDENT tokens to stream at eof *)
    let peek = Stack.top stack in
    let rec token_gen stream = function
        0    -> EOF :: stream
      | _    ->
        let _ = Stack.pop stack in
        token_gen (DEDENT :: stream) (Stack.top stack)
    in
```

```
    token_gen token_stream peek
}

let digit = ['0'-'9']
let alpha = ['a'-'z' 'A'-'Z']
let id = (alpha | '_') (alpha | digit | '_')*

let nl = '\n' | ('\r' '\n')
let tab = '\t'
let ws = [' ' '\t']

(* TODO: pass in stack arg for tracking indentation *)
rule token stream = parse
    (nl+ tab*)+ as delimit    {
                              L.new_line lexbuf ;
                              let toks = de_indent_gen delimit (NEWLINE :: stream) indent_stack in
                              token toks lexbuf
                            }
  | ws+                     { token stream lexbuf }
  | "if"                    { let toks = IF     :: stream in token toks lexbuf }
  | "break"                 { let toks = BREAK  :: stream in token toks lexbuf }
  | "else"                  { let toks = ELSE   :: stream in token toks lexbuf }
  | "true"                  { let toks = TRUE   :: stream in token toks lexbuf }
  | "false"                 { let toks = FALSE  :: stream in token toks lexbuf }
  | "for"                   { let toks = FOR    :: stream in token toks lexbuf }
  | "while"                 { let toks = WHILE  :: stream in token toks lexbuf }
  | "each"                  { let toks = EACH   :: stream in token toks lexbuf }
  | "in"                    { let toks = IN     :: stream in token toks lexbuf }
  | "and"                   { let toks = AND    :: stream in token toks lexbuf }
  | "or"                    { let toks = OR     :: stream in token toks lexbuf }
  | "no"                    { let toks = NO     :: stream in token toks lexbuf }
  | "not"                   { let toks = NOT    :: stream in token toks lexbuf }
  | "function"              { let toks = FUNC   :: stream in token toks lexbuf }
  | "return"                { let toks = RETURN :: stream in token toks lexbuf }
  | "set"                   { let toks = ASSIGN :: stream in token toks lexbuf }
  | "define"                { let toks = DEF    :: stream in token toks lexbuf }
  | "with"                  { let toks = WITH   :: stream in token toks lexbuf }
  | "params"                { let toks = PARAMS :: stream in token toks lexbuf }
  | "to"                    { let toks = TO     :: stream in token toks lexbuf }
  | '['                     { let toks = LBRACK :: stream in token toks lexbuf }
  | ']'                     { let toks = RBRACK :: stream in token toks lexbuf }
  | ':'                     { let toks = COLON  :: stream in token toks lexbuf }
  | ','                     { let toks = COMMA  :: stream in token toks lexbuf }
  | '('                     { let toks = LPAREN :: stream in token toks lexbuf }
```

```
  | ')'                      { let toks = RPAREN :: stream in token toks lexbuf }
  | ('+' | "plus")           { let toks = PLUS   :: stream in token toks lexbuf }
  | ('-' | "minus")          { let toks = MINUS  :: stream in token toks lexbuf }
  | ('*' | "times")          { let toks = MULT   :: stream in token toks lexbuf }
  | ('/' | "divided by")     { let toks = DIVIDE :: stream in token toks lexbuf }
  | ('%' | "modulo")         { let toks = MOD    :: stream in token toks lexbuf }
  | ('=' | "equals")         { let toks = EQUALS :: stream in token toks lexbuf }
  | ('>' | "greater than")   { let toks = GT     :: stream in token toks lexbuf }
  | ('<' | "less than")      { let toks = LT     :: stream in token toks lexbuf }
  | (">=" | "greater than or equal to")
                             { let toks = GEQ    :: stream in token toks lexbuf }
  | ("<=" | "less than or equal to")
                             { let toks = LEQ    :: stream in token toks lexbuf }
  | '"'                      { let toks = STRLIT(str (B.create buf_size) lexbuf) :: stream in token toks
lexbuf }
  | '#'                      { comment stream lexbuf }
  | "/*"                     { multi_comment stream lexbuf }
  | digit+ as num            { let toks = INTLIT(int_of_string num) :: stream in token toks lexbuf }
  | digit+ '.' digit* as num  { let toks = FLOATLIT(float_of_string num) :: stream in token toks
lexbuf }
  | "'s"                     { let toks = ATTR   :: stream in token toks lexbuf }
  | id as ident              { let toks = ID(ident) :: stream in token toks lexbuf }
  | eof                      { eof_dedent stream indent_stack }
  | _ as char                { raise (E.SyntaxError((Printf.sprintf "unrecognized char '%c'" char))) }

and str buf = parse
  | [^ '"' '\r' '\n' '\\' ]+ as text
                             { B.add_string buf text ; str buf lexbuf }
  | nl as newline            { B.add_string buf newline ; L.new_line lexbuf ; str buf lexbuf }
  | ('\\' '"')               { B.add_char buf '"' ; str buf lexbuf }
  | ('\\' 'n')               { B.add_char buf '\n' ; str buf lexbuf }
  | ('\\' 'r')               { B.add_char buf '\r' ; str buf lexbuf }
  | ('\\' 't')               { B.add_char buf '\t' ; str buf lexbuf }
  | '\\'                     { B.add_char buf '\\' ; str buf lexbuf }
  | '"'                      { B.contents buf } (* return *)
  | eof                      { raise (E.SyntaxError("eof within str")) }
  | _ as char                { raise (E.SyntaxError((Printf.sprintf "unrecognized char '%c'" char))) }

and comment stream = parse
  | nl+                      { token stream lexbuf }
  | eof                      { eof_dedent stream indent_stack }
  | _                        { comment stream lexbuf }

and multi_comment stream = parse
```

```
  | "*/"                     { token stream lexbuf }
  | eof                      { raise (E.SyntaxError("eof within comment")) }
  | _                        { multi_comment stream lexbuf }

{

 (* pretty printing functions *)

 let string_of_token = function
    STRLIT(str)     -> Printf.sprintf "STRLIT(%s)" (String.escaped str)
  | INTLIT(num)     -> Printf.sprintf "INTLIT(%d)" num
  | FLOATLIT(num)   -> Printf.sprintf "FLOATLIT(%f)" num
  | ID(str)         -> Printf.sprintf "ID(%s)" str
  | ATTR            -> Printf.sprintf "ATTR"
  | ASSIGN          -> Printf.sprintf "ASSIGN"
  | DEF             -> Printf.sprintf "DEF"
  | WITH            -> Printf.sprintf "WITH"
  | PARAMS          -> Printf.sprintf "PARAMS"
  | EQUALS          -> Printf.sprintf "EQUAL"
  | GT              -> Printf.sprintf "GT"
  | LT              -> Printf.sprintf "LT"
  | GEQ             -> Printf.sprintf "GEQ"
  | LEQ             -> Printf.sprintf "LEQ"
  | PLUS            -> Printf.sprintf "PLUS"
  | MINUS           -> Printf.sprintf "MINUS"
  | NEG             -> Printf.sprintf "NEG"
  | MULT            -> Printf.sprintf "MULT"
  | DIVIDE          -> Printf.sprintf "DIVIDE"
  | MOD             -> Printf.sprintf "MOD"
  | IF              -> Printf.sprintf "IF"
  | ELSE            -> Printf.sprintf "ELSE"
  | TRUE            -> Printf.sprintf "TRUE"
  | FALSE           -> Printf.sprintf "FALSE"
  | FOR             -> Printf.sprintf "FOR"
  | WHILE           -> Printf.sprintf "WHILE"
  | IN              -> Printf.sprintf "IN"
  | EACH            -> Printf.sprintf "EACH"
  | TO              -> Printf.sprintf "TO"
  | AND             -> Printf.sprintf "AND"
  | OR              -> Printf.sprintf "OR"
  | NOT             -> Printf.sprintf "NOT"
  | NO              -> Printf.sprintf "NO"
  | FUNC            -> Printf.sprintf "FUNC"
  | RETURN          -> Printf.sprintf "RETURN"
```

```
  | COMMA          -> Printf.sprintf "COMMA"
  | COLON          -> Printf.sprintf "COLON"
  | LBRACK         -> Printf.sprintf "LBRACK"
  | RBRACK         -> Printf.sprintf "RBRACK"
  | LPAREN         -> Printf.sprintf "LPAREN"
  | RPAREN         -> Printf.sprintf "RPAREN"
  | INDENT         -> Printf.sprintf "INDENT"
  | DEDENT         -> Printf.sprintf "DEDENT"
  | NEWLINE        -> Printf.sprintf "NEWLINE"
  | EOF            -> Printf.sprintf "EOF"
  | BREAK          -> Printf.sprintf "BREAK"

let string_of_tokens tokens =
  List.map (fun tok -> string_of_token tok) tokens
    |> String.concat " "

}
```

## 8.6 parser.mly

```
%{
  open Ast
%}

/* TODO: lists, iterations, breaks */

%token LPAREN RPAREN LBRACK RBRACK COLON COMMA ATTR
%token NEWLINE INDENT DEDENT
%token RETURN DEF FUNC WITH NO PARAMS IF ELSE FOR WHILE EACH IN
%token BREAK
%token PLUS MINUS NEG MULT DIVIDE MOD EQUALS ASSIGN TO
%token GT LT GEQ LEQ AND OR NOT TRUE FALSE
%token EOF

%token <int>    INTLIT
%token <float>  FLOATLIT
%token <string> STRLIT
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE
%left OR
%left AND
%left EQUALS
%left LT GT LEQ GEQ
```

```
%left PLUS MINUS
%left MULT DIVIDE MOD
%right NOT NEG

%start program
%type <Ast.program> program

%%


program:
    decls EOF               { $1 }
  | NEWLINE decls           { $2 }

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

fdecl:
    fdecl_params      { $1 }
  | fdecl_noparams    { $1 }

fdecl_params:
    DEF FUNC ID WITH PARAMS LPAREN params_opt RPAREN NEWLINE INDENT stmt_list
DEDENT
    {{
      fname = $3 ;
      formals = $7 ;
      body = List.rev $11 ;
    }}

fdecl_noparams:
  DEF FUNC ID WITH NO PARAMS NEWLINE INDENT stmt_list DEDENT
  {{
    fname = $3 ;
    formals = [] ;
    body = List.rev $9
  }}

params_opt:
    /* nothing */   { [] }
  | param_list      { List.rev $1 }
```

```
param_list:
   ID                  { [$1] }
 | param_list COMMA ID   { $3 :: $1 }

iteration_stmt:
 WHILE LPAREN expr RPAREN NEWLINE compound_stmt  { While($3, $6) }

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

stmt:
   expr_stmt     { $1 }
 | select_stmt   { $1 }
 | assign_stmt   { $1 }
 | compound_stmt { $1 }
 | jump_stmt     { $1 }
 | iteration_stmt{ $1 }

expr_stmt:
   expr NEWLINE  { Expr $1 }

select_stmt:
   IF LPAREN expr RPAREN NEWLINE compound_stmt %prec NOELSE
     { If($3, $6, Block([])) }
 | IF LPAREN expr RPAREN NEWLINE compound_stmt ELSE NEWLINE compound_stmt
     { If($3, $6, $9) }

assign_stmt:
   ASSIGN ID TO expr NEWLINE                    { Assign($2, $4) }
 | ASSIGN ID LBRACK expr RBRACK TO expr NEWLINE  { ListReplace($2, $4, $7) }

compound_stmt:
   INDENT stmt_list DEDENT        { Block(List.rev $2)}

jump_stmt:
 | BREAK NEWLINE { Break }
 | RETURN expr NEWLINE     { Return($2) }

expr:
   literals                    { $1 }
 | expr PLUS expr              { Binop($1, Add, $3) }
 | expr MINUS expr              { Binop($1, Sub, $3) }
 | expr MULT expr              { Binop($1, Mult, $3) }
```

```
  | expr DIVIDE expr              { Binop($1, Div, $3) }
  | expr MOD expr                 { Binop($1, Mod, $3) }
  | expr EQUALS expr              { Binop($1, Eq, $3) }
  | expr LT expr                  { Binop($1, Lt, $3) }
  | expr GT expr                  { Binop($1, Gt, $3) }
  | expr LEQ expr                 { Binop($1, Leq, $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 NEG          { Unop(Neg, $2) }
  | NOT expr                      { Unop(Not, $2) }
  | ID LPAREN actuals_opt RPAREN  { Call($1, $3) }
  | LPAREN expr RPAREN            { $2 }
  | list_expr                     { $1 }

literals:
    INTLIT      { IntLit($1) }
  | FLOATLIT    { FloatLit($1) }
  | STRLIT      { StrLit($1) }
  | TRUE        { BoolLit(true) }
  | FALSE       { BoolLit(false) }
  | ID          { Id($1) }

list_expr:
    LBRACK expr_list_opt RBRACK       { List($2) }
  | ID LBRACK expr RBRACK             { ListAccess($1, $3) }
  | ID LBRACK expr COLON expr RBRACK  { ListSlice($1, $3, $5) }

expr_list_opt:
    /* nothing */     { [] }
  | expr_list         { List.rev $1 }

expr_list:
    expr                      { [$1] }
  | expr_list COMMA expr      { $3 :: $1 }

vinit:
    ASSIGN ID TO expr NEWLINE   { ($2, $4) }

actuals_opt:
    /* nothing */     { [] }
  | actuals_list      { List.rev $1 }

actuals_list:
```

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

## 8.7 semant.ml

```
(* type inference and semantic checking *)

open Ast
open Sast
module E = Exceptions

module StringMap = Map.Make(String)

type env = {
  env_fmap: fdecl StringMap.t;
  env_fname: string;
  env_return_type: datatype;
  env_globals: datatype StringMap.t;
  env_flocals: datatype StringMap.t;
  env_in_loop: bool;
  env_set_return: bool;
  env_sfmap: sfdecl StringMap.t;
}

let rec expr_to_sexpr expr env =
  match expr with
    IntLit(d)                -> (SIntLit(d, Datatype(Int)), env)
  | FloatLit(f)              -> (SFloatLit(f, Datatype(Float)), env)
  | StrLit(s)                -> (SStrLit(s, Datatype(String)), env)
  | BoolLit(b)               -> (SBoolLit(b, Datatype(Bool)), env)
  | Id(s)                    -> (check_scope s env, env)
  | Noexpr                   -> (SNoexpr, env)
  | Unop(op, e)              -> (check_unop op e env)
  | Binop(e1, op, e2)        -> (check_binop e1 op e2 env)
  (* built-in functions *)
  | Call("print", e_l)       -> (check_print e_l env)
  | Call(s, e_l)             -> (check_call s e_l env)
  (* list functionality *)
  | List(e_l)                -> (check_list e_l env)
  | ListAccess(s, e)         -> (check_access s e env)
  | ListSlice(s, e1, e2)     -> (check_slice s e1 e2 env) (* returns func call *)

and sexpr_to_type = function
    SIntLit(_, typ)          -> typ
  | SFloatLit(_, typ)        -> typ
```

```
  | SStrLit(_, typ)           -> typ
  | SBoolLit(_, typ)          -> typ
  | SId(_, typ)               -> typ
  | SBinop(_, _, _, typ)      -> typ
  | SUnop(_, _, typ)          -> typ
  | SCall(_, _, typ)          -> typ
  | SNoexpr                   -> Datatype(Void)
  (* list functionality *)
  | SList(_, typ)             -> typ
  | SListAccess(_, _, typ)    -> typ

and expr_list_to_sexpr_list e_l env =
  let env_ref = ref(env) in
  match e_l with
    hd :: tl  ->
      let (se, env) = expr_to_sexpr hd !env_ref in
      env_ref := env ;
      let (l, env) = expr_list_to_sexpr_list tl !env_ref in
      env_ref := env ;
      (se :: l, !env_ref)
  | []        -> ([], !env_ref)

and stmt_to_sstmt stmt env =
  match stmt with
    Block sl                -> check_sblock sl env
  | Expr e                  -> check_expr e env
  | Assign(s, e)            -> check_assign s e env
  | Return e                -> check_return e env
  | If(e, s1, s2)           -> check_if e s1 s2 env
  | While(e, s)             -> check_while e s env
  | Break                   -> check_break env
  (* list functionality *)
  | ListReplace(s, e1, e2)  -> check_replace s e1 e2 env

and fdecl_to_sfdecl fname arg_type_list env =
  let fdecl = StringMap.find fname env.env_fmap in

  (* make params and their types local vars *)
  let flocals = (
    List.fold_left2 (fun map name typ -> StringMap.add name typ map)
      StringMap.empty fdecl.formals arg_type_list
  )
  in
```

```
(* start env variable *)
let env = {
  env_globals = env.env_globals;
  env_fmap = env.env_fmap;
  env_fname = fname;
  env_return_type = Datatype(Void); (* placeholder *)
  env_flocals = flocals;
  env_in_loop = false;
  env_set_return = false;
  env_sfmap = env.env_sfmap;
}
in

(* semantically check body statments *)
let (sstmts, env) = stmt_to_sstmt (Block fdecl.body) env in

(* create semantically checked formals for fname *)
report_duplicate(fdecl.formals) ;

let sformals =
  let get_args l name typ =
    try
      let found_typ = StringMap.find name env.env_flocals in
      (name, found_typ) :: l
    with Not_found ->
      (name, typ) :: l
  in
  List.rev (List.fold_left2 get_args [] fdecl.formals arg_type_list)
in

(* create semantically checked locals for fname *)
let formals_map = (
  List.fold_left (fun map (name, typ) -> StringMap.add name typ map)
    StringMap.empty sformals
)
in
let locals_map =
  let remove_formals map (name, typ) =
    if (StringMap.mem name formals_map) then
      StringMap.remove name map
    else
      map
  in
  List.fold_left remove_formals env.env_flocals (StringMap.bindings env.env_flocals)
```

```
  in
  let locals = StringMap.bindings locals_map in
  let slocals = List.fold_left (fun l (name, typ) -> (name, typ) :: l) [] locals
    |> List.rev
  in

  (* semantically check body statments *)
  let (sstmts, env) = stmt_to_sstmt (Block fdecl.body) env in

  (* create semantically checked func *)
  let sfdecl = {
    styp = env.env_return_type;
    sfname = fdecl.fname;
    slocals = slocals;
    sformals = sformals;
    sbody = match sstmts with SBlock(sl) -> sl | _ -> [];
  }
  in

  (* return the env with updated semantic func map *)
  let new_env = {
    env_fmap = env.env_fmap;
    env_fname = env.env_fname;
    env_return_type = env.env_return_type;
    env_globals = env.env_globals;
    env_flocals = env.env_flocals;
    env_in_loop = env.env_in_loop;
    env_set_return = env.env_set_return;
    env_sfmap = StringMap.add fname sfdecl env.env_sfmap;
  }
  in
  new_env

(* report duplicate variables *)
and report_duplicate list =
  let rec helper = function
      n1 :: n2 :: _ when n1 = n2  -> (raise (E.DuplicateVariable n1))
    | _ :: t                      -> helper t
    | []                          -> ()
  in
  helper (List.sort compare list)

and change_sexpr_type sexpr typ env =
  match sexpr with
```

```
    SId(var, _)     -> (
      let flocals = StringMap.add var typ env.env_flocals in
      let new_env = {
        env_globals = env.env_globals;
        env_fmap = env.env_fmap;
        env_fname = env.env_fname;
        env_return_type = env.env_return_type;
        env_flocals = flocals;
        env_in_loop = env.env_in_loop;
        env_set_return = env.env_set_return;
        env_sfmap = env.env_sfmap;
      } in
      (SId(var, typ), new_env)
    )
  | _             -> (sexpr, env)

(* check conditionals *)
and check_if expr s1 s2 env =
  let (sexpr, env) = expr_to_sexpr expr env in
  let typ = sexpr_to_type sexpr in
  let (if_body, env) = stmt_to_sstmt s1 env in
  let (else_body, env) = stmt_to_sstmt s2 env in
  if (typ = Datatype(Bool) ) then
    (SIf(sexpr, SBlock([if_body]), SBlock([else_body])), env)
  else
    (raise (E.InvalidIfStatementType))

and check_break env =
  if env.env_in_loop then
    (SBreak, env)
  else raise E.BreakOutsideOfLoop

(* check block for validity *)
and check_sblock sl env =
  (* make sure nothing follows a return *)
  let rec check_block_return = function
      Return _ :: _ :: _      -> (raise (E.NothingAfterReturn))
    | Block sl :: ss          -> check_block_return (sl @ ss)
    | _ :: ss                 -> check_block_return ss
    | []                      -> ()
  in check_block_return sl ;

  (* check all statements within block *)
  match sl with
```

```
  | []    -> (SBlock([SExpr(SNoexpr, Datatype(Void))]), env)
  | _     ->
    let env_ref = ref(env) in
    let convert_stmt l stmt =
      let (new_stmt, env) = stmt_to_sstmt stmt !env_ref in
      env_ref := env ; (new_stmt :: l)
    in
    let (block, _) = (List.rev @@ (List.fold_left convert_stmt [] sl), !env_ref) in
    (SBlock(block), !env_ref)

(* check and validate while loops *)
and check_while expr stmt env =
  (* create env for loop context *)
  let prev_context = env.env_in_loop in
  let env = {
    env_globals = env.env_globals;
    env_fmap = env.env_fmap;
    env_fname = env.env_fname;
    env_return_type = env.env_return_type;
    env_flocals = env.env_flocals;
    env_in_loop = true;
    env_set_return = env.env_set_return;
    env_sfmap = env.env_sfmap;
  }
  in

  (* semantically check condition & body *)
  let (sexpr, env) = expr_to_sexpr expr env in
  let typ = sexpr_to_type sexpr in
  let (body, env) = stmt_to_sstmt stmt env in

  (* revert env *)
  let env = {
    env_globals = env.env_globals;
    env_fmap = env.env_fmap;
    env_fname = env.env_fname;
    env_return_type = env.env_return_type;
    env_flocals = env.env_flocals;
    env_in_loop = prev_context;
    env_set_return = env.env_set_return;
    env_sfmap = env.env_sfmap;
  }
  in
```

```
  (* check condition *)
  if (typ = Datatype(Bool)) then
    (SWhile(sexpr, SBlock([body])), env)
  else
    (raise E.InvalidCondition)

(* check and verify return type *)
and check_return expr env =
  let (sexpr, env) = expr_to_sexpr expr env in
  let typ = sexpr_to_type sexpr in

  (* if return type has been set, check new return type *)
  if (env.env_set_return) then
    let () = ignore(
      if (env.env_return_type <> typ) then
        Printf.printf "WARNING: function %s has expected return type of %s but is type %s ..."
        env.env_fname (string_of_typ env.env_return_type) (string_of_typ typ)
    )
    in
    (SReturn(sexpr, typ), env)

  (* no return type set, so set it *)
  else
    let new_env = {
      env_fmap = env.env_fmap;
      env_fname = env.env_fname;
      env_return_type = typ;
      env_globals = env.env_globals;
      env_flocals = env.env_flocals;
      env_in_loop = env.env_in_loop;
      env_set_return = true;
      env_sfmap = env.env_sfmap;
    }
    in
    (SReturn(sexpr, typ), new_env)

(* check expression *)
and check_expr expr env =
  let (sexpr, env) = expr_to_sexpr expr env in
  let typ = sexpr_to_type sexpr in
  (SExpr(sexpr, typ), env)

(* check access to var *)
and check_scope var env =
```

```
  try
    let typ = StringMap.find var env.env_flocals in
    SId(var, typ)
  with Not_found -> (
    try
      let typ = StringMap.find var env.env_globals in
      SId(var, typ)
    with Not_found ->
      (raise (E.UndefinedId var))
  )

(* check variable assignment *)
and check_assign var expr env =
  let sexpr, env = expr_to_sexpr expr env in
  let new_typ = sexpr_to_type sexpr in

  (* var has been declared, check new type *)
  if (StringMap.mem var env.env_flocals || StringMap.mem var env.env_globals) then
    let old_typ =
      try
        StringMap.find var env.env_flocals (* try local first *)
      with Not_found ->
        StringMap.find var env.env_globals (* query global *)
    in
    match (old_typ, new_typ) with
      (* check list types *)
      Listtype(t1), Listtype(t2)  ->
        if (t1 <> t2 && t2 <> Void) then      (* type mismatch - for now TODO *)
          let msg =
            Printf.sprintf "var %s of listtype %s has been redeclared with listtype %s - lists must be
homogenous"
            var (string_of_typ (Datatype(t1))) (string_of_typ (Datatype(t2)))
          in
          (raise (E.TypeMismatch(msg)))
        else if (t2 = Void) then              (* new type is void return old type *)
          (SAssign(var, sexpr, old_typ), env)
        else
          let flocals = StringMap.add var new_typ env.env_flocals in
          let new_env = {
            env_fmap = env.env_fmap;
            env_fname = env.env_fname;
            env_return_type = env.env_return_type;
            env_globals = env.env_globals;
            env_flocals = flocals;
```

```
          env_in_loop = env.env_in_loop;
          env_set_return = env.env_set_return;
          env_sfmap = env.env_sfmap
        }
        in
        (SAssign(var, sexpr, new_typ), new_env)

      (* check all other types *)
    | _                           -> (
        if (old_typ <> new_typ) then
          Printf.printf "WARNING: var %s of type %s has been redeclared with type %s ..."
            var (string_of_typ old_typ) (string_of_typ new_typ)
      ) ; (SAssign(var, sexpr, new_typ), env)

  (* var not declared, bind typ in map *)
  else
    let flocals = StringMap.add var new_typ env.env_flocals in
    let new_env = {
      env_fmap = env.env_fmap;
      env_fname = env.env_fname;
      env_return_type = env.env_return_type;
      env_flocals = flocals;
      env_globals = env.env_globals;
      env_in_loop = env.env_in_loop;
      env_set_return = env.env_set_return;
      env_sfmap = env.env_sfmap;
    }
    in
    (SAssign(var, sexpr, new_typ), new_env)

(* check unop operations *)
and check_unop op expr env =
  let check_bool_unop = function
      Not       -> Datatype(Bool)
    | _         -> (raise (E.InvalidUnaryOperation))
  in
  let check_int_unop = function
      Neg       -> Datatype(Int)
    | _         -> (raise (E.InvalidUnaryOperation))
  in
  let check_float_unop = function
      Neg       -> Datatype(Float)
    | _         -> (raise (E.InvalidUnaryOperation))
  in
```

```
  let (sexpr, env) = expr_to_sexpr expr env in
  let typ = sexpr_to_type sexpr in
  match typ with
    Datatype(Int)     -> (SUnop(op, sexpr, check_int_unop op), env)
  | Datatype(Float)   -> (SUnop(op, sexpr, check_float_unop op), env)
  | Datatype(Bool)    -> (SUnop(op, sexpr, check_bool_unop op) , env)
  | _                 -> (raise (E.InvalidUnaryOperation))

(* check binop operations *)
and check_binop e1 op e2 env =
  let (se1, env) = expr_to_sexpr e1 env in
  let (se2, env) = expr_to_sexpr e2 env in
  let typ1 = sexpr_to_type se1 in
  let typ2 = sexpr_to_type se2 in
  match op with
    Eq ->
      if (typ1 = typ2 || typ1 = Datatype(Void) || typ2 = Datatype(Void)) then
        if (typ1 = Datatype(String)) then
          (SCall("strcmp", [se1 ; se2], Datatype(Bool)), env)
        else
          (SBinop(se1, op, se2, Datatype(Bool)), env)
      else
        (raise (E.InvalidBinaryOperation))
  | And | Or ->
      if (typ1 = Datatype(Bool) && typ2 = Datatype(Bool)) then
        (SBinop(se1, op, se2, Datatype(Bool)), env)
      else
        (raise (E.InvalidBinaryOperation))
  | Lt | Leq | Gt | Geq ->
      if (typ1 = typ2 && (typ1 = Datatype(Int) || typ1 = Datatype(Float))) then
        (SBinop(se1, op, se2, Datatype(Bool)), env)
      else
        (raise (E.InvalidBinaryOperation))
  | Add | Mult | Sub | Div | Mod ->
      if (typ1 = typ2 && (typ1 = Datatype(Int) || typ1 = Datatype(Float))) then
        (SBinop(se1, op, se2, typ1), env)
      else
        (raise (E.InvalidBinaryOperation))

(* check built-in print type inference *)
and check_print e_l env =
  if ((List.length e_l) <> 1) then
    (raise (E.WrongNumberOfArguments))
  else
```

```
    let (se, env) = expr_to_sexpr (List.hd e_l) env in
    let typ = sexpr_to_type se in
    let new_s = match typ with
        Datatype(Int)           -> "printint"
      | Datatype(String)        -> "printstr"
      | Datatype(Bool)          -> "printbool"
      | Datatype(Float)         -> "printfloat"
      | _                       -> (raise E.CannotPrintType)
    in
    (SCall(new_s, [se], Datatype(Int)), env)

(* check function call *)
and check_call id e_l env =
  if (not (StringMap.mem id env.env_fmap)) then
    (raise (E.FunctionNotDefined id))
  else
    let env_ref = ref(env) in

    (* semantically check args *)
    let _ =
      let check l expr =
        let (sexpr, env) = expr_to_sexpr expr env in
        env_ref := env; (sexpr :: l)
      in
      List.fold_left check [] e_l
    in

    (* list argument types *)
    let arg_type_list =
      let get_type l expr =
        let (sexpr, _) = expr_to_sexpr expr env in
        let typ = sexpr_to_type sexpr in
        (typ :: l)
      in
      List.fold_left get_type [] e_l
        |> List.rev
    in

    let fdecl = StringMap.find id env.env_fmap in

    (* check for correct number of args *)
    if (List.length e_l <> List.length fdecl.formals) then
      (raise (E.WrongNumberOfArguments))
```

```
    (* recursive *)
    else if (id = env.env_fname) then
      let (e_l, env) = expr_list_to_sexpr_list e_l env in
      (SCall(id, e_l, env.env_return_type), env)

    (* called by another function and already called/defined (lib) *)
    else if (StringMap.mem id env.env_sfmap) then
      let called_fdecl = StringMap.find id env.env_sfmap in
      let check new_typ (s, old_typ) = ignore(
        if (new_typ <> old_typ) then
          let msg = Printf.sprintf "argument %s should be of type %s not %s"
            s (string_of_typ old_typ) (string_of_typ new_typ)
          in
          (raise (E.IncorrectArgumentType msg))
      )
      in
      List.iter2 check arg_type_list called_fdecl.sformals ;
      let (se_l, env) = expr_list_to_sexpr_list e_l env in
      (SCall(id, se_l, called_fdecl.styp), env)

  (* called for the first time - semantically check *)
  else
    let (se_l, env) = expr_list_to_sexpr_list e_l env in
    let new_env = fdecl_to_sfdecl id arg_type_list env in
    let env = {
      env_fmap = env.env_fmap;
      env_fname = env.env_fname;
      env_return_type = env.env_return_type;
      env_globals = env.env_globals;
      env_flocals = env.env_flocals;
      env_in_loop = env.env_in_loop;
      env_set_return = env.env_set_return;
      env_sfmap = new_env.env_sfmap;
    }
    in
    let called_fdecl = StringMap.find id env.env_sfmap in
    (SCall(id, se_l, called_fdecl.styp), env)

(* create function declaration map *)
and build_fdecl_map functions =

  (* reserved built-ins *)
  let builtin_decls = (StringMap.add "print"
  { fname = "print"; formals = [("x")];
```

```
  body = []; } (StringMap.add "strcmp"
  { fname = "strcmp"; formals = ["x"; "y"];
  body = []; } (StringMap.add "str"
  { fname = "str"; formals = [("x")];
  body = []; } (StringMap.add "num"
  { fname = "num"; formals = [("x")];
  body = []; } (StringMap.add "bool"
  { fname = "bool"; formals = [("x")];
  body = []; } (StringMap.singleton "printstr"
  { fname = "print" ; formals = [("x")];
  body = []; }) )))))

  in
  (* make sure no functions have reserved/duplicated names *)
  let check map fdecl =
    if (StringMap.mem fdecl.fname map) then
      (raise (E.DuplicateFunctionName fdecl.fname))
    else if (StringMap.mem fdecl.fname builtin_decls) then
      (raise (E.FunctionNameReserved))
    else
      StringMap.add fdecl.fname fdecl map
  in
  List.fold_left check builtin_decls functions

(* list - check __get_item__ *)
and check_access id expr env =
  let _ = check_scope id env in
  let typ =
    try
      StringMap.find id env.env_flocals
    with Not_found ->
      StringMap.find id env.env_globals
  in
  let elem_typ =
    match typ with
      Listtype(t) -> t
    | _           -> (raise (E.IndexError))
  in
  let (se, env) = expr_to_sexpr expr env in
  let access_typ = sexpr_to_type se in
  if (access_typ <> Datatype(Int)) then
    (raise (E.IndexError))
  else
    (SListAccess(id, se, Datatype(elem_typ)), env)
```

```
(* list - check init *)
and check_list e_l env =
  if (List.length e_l = 0) then
    (SList([], Listtype(Int)), env)
  else
    let (first_sexpr, env) = expr_to_sexpr (List.hd e_l) env in
    let target_typ = sexpr_to_type first_sexpr in
    (* TODO: for now - assume all list elements have same primitive type *)
    let list_typ =
      match target_typ with
        Datatype(typ)     -> typ
      | _                 -> (raise (E.InvalidListElementType))
    in
    let env_ref = ref (env) in
    let check_element_type se_l expr =
      let (sexpr, env) = expr_to_sexpr expr !env_ref in
      env_ref := env ;
      let typ = sexpr_to_type sexpr in
      if (typ <> target_typ) then
        (raise (E.ListElementsOfVariantType))
      else
        (sexpr :: se_l)
    in
    let se_l = List.rev @@ (List.fold_left check_element_type [] (List.tl e_l)) in
    let final = first_sexpr :: se_l in
    ((SList(final, Listtype(list_typ))), !env_ref)

(* list - check slice operation *)
and check_slice id e1 e2 env =
  let sid = check_scope id env in
  let typ =
    try
      StringMap.find id env.env_flocals
    with Not_found ->
      StringMap.find id env.env_globals
  in
  let elem_typ =
    match typ with
      Listtype(t) -> t
    | _           -> (raise (E.IndexError))
  in
  let (se1, env) = expr_to_sexpr e1 env in
  let (se2, env) = expr_to_sexpr e2 env in
```

```
  let typ1 = sexpr_to_type se1 in
  let typ2 = sexpr_to_type se2 in
  if (typ1 <> Datatype(Int) || typ2 <> Datatype(Int)) then
    (raise (E.IndexError))
  else
    (* TODO: build these functions *)
    match elem_typ with
      Int     -> (SCall("sliceint", [sid; se1; se2], typ), env)
    | String  -> (SCall("slicestr", [sid; se1; se2], typ), env )
    | Float   -> (SCall("slicefloat", [sid; se1; se2], typ), env)
    | Bool    -> (SCall("slicebool", [sid; se1; se2], typ), env)
    | _       -> (raise (E.InvalidSliceOperation))

(* list - check replace assignment *)
and check_replace id e1 e2 env =
  (* verify index is accessible *)
  let _ = check_access id e1 env in
  let typ =
    try
      StringMap.find id env.env_flocals
    with Not_found ->
      StringMap.find id env.env_globals
  in
  let elem_typ =
    match typ with
      Listtype(t) -> t
    | _           -> (raise (E.InvalidListAssignmentOperation))
  in
  let (se1, env) = expr_to_sexpr e1 env in
  let (se2, env) = expr_to_sexpr e2 env in
  let access_typ = sexpr_to_type se1 in
  let input_typ = sexpr_to_type se2 in

  (* index must be int & TODO: for now - list type must be homogenous *)
  if (access_typ <> Datatype(Int) || input_typ <> Datatype(elem_typ)) then
    (raise (E.InvalidListAssignmentOperation))
  else
    (SListReplace(id, se1, se2, input_typ), env)

(* convert ast to sast *)
and ast_to_sast (globals, functions) fmap =
  (* temp env *)
  let tmp_env = {
    env_fmap = fmap;
```

```
  env_fname = "main";
  env_return_type = Datatype(Int);
  env_globals = StringMap.empty;
  env_flocals = StringMap.empty;
  env_in_loop = false;
  env_set_return = true;
  env_sfmap = StringMap.empty;
}
in

(* semantically check globals *)
let env_ref = ref tmp_env in
let sglobals =
  let check l (name, expr) =
    let (sexpr, env) = expr_to_sexpr expr !env_ref in
    let typ = sexpr_to_type sexpr in
    env_ref := env ; (name, sexpr, typ) :: l
  in
  List.fold_left check [] globals
    |> List.rev
in

(* create globals map *)
let globals_map =
  let add_to_map map (name, _, typ) = StringMap.add name typ map in
  List.fold_left add_to_map StringMap.empty sglobals
in

(* create new env for semantic check *)
let env = {
  env_fmap = fmap;
  env_fname = "main";
  env_return_type = Datatype(Void);
  env_globals = globals_map;
  env_flocals = StringMap.empty;
  env_in_loop = false;
  env_set_return = false;
  env_sfmap = StringMap.empty;
}
in

(* check that they are no duplicate functions *)
report_duplicate (List.map (fun fd -> fd.fname) functions);
```

```
  (* convert all fdecls to sfdecls through main *)
  let env = fdecl_to_sfdecl "main" [] env in

  (* return all checked functions & globals *)
  let sfdecls =
    let add_sfdecl l (_, sfdecl) = sfdecl :: l in
    List.fold_left add_sfdecl [] (StringMap.bindings env.env_sfmap)
      |> List.rev
  in
  (sglobals, sfdecls)

and check (globals, functions) =
  let fmap = build_fdecl_map functions in
  (* return SAST *)
  ast_to_sast (globals, functions) fmap
```

---

## 8.8 codegen.ml

```
module L = Llvm
module A = Ast
module S = Sast
module E = Exceptions
module Semant = Semant

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module  context "Newbie"

      and i32_t  = L.i32_type     context
      and i1_t   = L.i1_type      context
      and void_t = L.void_type    context
      and float_t = L.double_type context
      and str_t  = L.pointer_type   (L.i8_type context)
    in

    let br_block    = ref (L.block_of_value (L.const_int i32_t 0)) in

    let global_vars = ref (StringMap.empty) in
    let current_f = ref (List.hd functions) in
    let local_vars = ref (StringMap.empty) in
    (* list lookup *)
    let list_lookup = ref (StringMap.empty) in
```

```
(* pointer wrapper - map of named struct types pointers *)
let pointer_wrapper =
  List.fold_left
  (fun map name -> StringMap.add name (L.named_struct_type context name) map)
  StringMap.empty ["string"; "int"; "float"; "void"; "bool"]
in

(* set struct body fields for each of the types *)
let () =
  let set_struct_body name l =
    let t = StringMap.find name pointer_wrapper in
    ignore(
      L.struct_set_body t (Array.of_list(l)) true
    )
  in
  let named_types = ["string"; "int"; "float"; "void"; "bool"] in
  let llvm_types = [
    [L.pointer_type str_t; i32_t; i32_t]    ;
    [L.pointer_type i32_t; i32_t; i32_t]    ;
    [L.pointer_type float_t; i32_t; i32_t]  ;
    [L.pointer_type void_t; i32_t; i32_t]   ;
    [L.pointer_type i1_t; i32_t; i32_t]     ;
  ]
  in
  List.iter2 set_struct_body named_types llvm_types
in

(* Format strings for printing *)
let int_format_str builder = L.build_global_stringptr "%d\n" "fmt" builder
and str_format_str builder = L.build_global_stringptr "%s\n" "fmt" builder
and float_format_str builder = L.build_global_stringptr "%f\n" "fmt" builder in

(* get struct pointer *)
let lookup_struct typ =
  let s = S.string_of_typ typ in
  StringMap.find s pointer_wrapper
in

let ltype_of_typ = function
    A.Datatype(A.Int)     ->  i32_t
  | A.Datatype(A.Bool)    ->  i1_t
  | A.Datatype(A.Void)    ->  void_t
  | A.Datatype(A.String)  ->  str_t
  | A.Datatype(A.Float)   ->  float_t
```

```
  | A.Listtype(t)         ->  L.pointer_type (lookup_struct (A.Datatype(t)))
in

(* Declare print *)
let print_t = L.var_arg_function_type i32_t [| str_t |] in
let print_func = L.declare_function "printf" print_t the_module in

(* Define each function (arguments and return type) so we can call it *)
let function_decls =
  let function_decl m fdecl =
    let name = fdecl.S.sfname
    and formal_types =
    Array.of_list(List.map
      (fun (_, t) -> ltype_of_typ t) fdecl.S.sformals)
    in
    let ftype = L.function_type (ltype_of_typ fdecl.S.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


let rec add_terminal builder f =
    match L.block_terminator (L.insertion_block builder) with
    Some _  -> ()
    | None    -> ignore (f builder)

and expr builder = function
    S.SBoolLit(b, _)    -> L.const_int i1_t (if b then 1 else 0)
  | S.SStrLit (s, _)    -> L.build_global_stringptr s "string" builder
  | S.SNoexpr           -> L.const_int i32_t 0
  | S.SFloatLit(f, _)   -> L.const_float float_t f
  | S.SIntLit (i, _)    -> L.const_int i32_t i
  (* print built-in *)
  | S.SCall("printstr", [e], _)   ->
      L.build_call print_func [| str_format_str builder; (expr builder e)|]
      "printf" builder
  | S.SCall("printint", [e], _)   ->
      L.build_call print_func [| int_format_str builder; (expr builder e)|]
      "printf" builder
  | S.SCall("printfloat", [e], _) ->
      L.build_call print_func [| float_format_str builder; (expr builder e)|]
      "printf" builder
  | S.SCall("printbool", [e], _)  ->
```

```
    L.build_call print_func [| int_format_str builder; (expr builder e)|]
    "printf" builder
  (* function refs *)
  | S.SCall(f, act, _ ) ->
    let (fdef, fdecl) = StringMap.find f function_decls in
    let actuals = List.rev (List.map (expr builder) (List.rev act)) in
    let result =
      match fdecl.S.styp with
        A.Datatype(A.Void)  -> ""
      | _                   -> f ^ "_result"
    in
    L.build_call fdef (Array.of_list actuals) result builder
  | S.SUnop(op, e, _) ->
    let e' = expr builder e in
    let llvm_build = function
      | A.Neg       -> L.build_neg e' "tmp" builder
      | A.Not       -> L.build_not e' "tmp" builder
    in
    llvm_build op
  | S.SId (s, _)    -> L.build_load (lookup s) s builder
  | S.SBinop (e1, op, e2, _) ->
    let e1' = expr builder e1
    and e2' = expr builder e2 in
      let typ = Semant.sexpr_to_type e1 in
      (match typ with
          A.Datatype(A.Int) |  A.Datatype(A.Bool) ->  (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
        | A.Div     -> L.build_sdiv
        | A.Mod     -> L.build_srem
        | A.Eq   -> L.build_icmp L.Icmp.Eq
        | A.Lt    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Gt -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        (* | A.Neq     -> L.build_icmp L.Icmp.Ne *)
        ) e1' e2' "tmp" builder
        | A.Datatype(A.Float) -> (match op with
        A.Add     ->   L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mult    -> L.build_fmul
```

```
        | A.Div     -> L.build_fdiv
        | A.Mod     -> L.build_frem
        | A.Eq   -> L.build_fcmp L.Fcmp.Oeq
        | A.Lt    -> L.build_fcmp L.Fcmp.Ult
        (* | A.Neq     -> L.build_fcmp L.Fcmp.One *)
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Gt -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
        | _ -> raise E.InvalidBinaryOperation
        ) e1' e2' "tmp" builder
        | _ -> raise E.InvalidBinaryOperation)
(* list expr *)
| S.SListAccess(s, se, t) ->
    let idx = expr builder se in
    let idx = L.build_add idx (L.const_int i32_t 1) "access1" builder in
    let struct_ptr = expr builder (S.SId(s, t)) in
    let arr =
      L.build_load
        (L.build_struct_gep struct_ptr 0 "access2" builder)
        "idl"
        builder
    in
    let res = L.build_gep arr [| idx |] "access3" builder in
    L.build_load res "access4" builder
| S.SList(se_l, t)        ->
    let it =
      match t with
        A.Listtype(it)  -> it
      | _               -> (raise (E.InvalidListElementType))
    in
    let struct_ptr = L.build_malloc (lookup_struct t) "list1" builder in
    let size = L.const_int i32_t ((List.length se_l) + 1) in
    let typ = L.pointer_type (ltype_of_typ (A.Datatype(it))) in
    let arr = L.build_array_malloc typ size "list2" builder in
    let arr = L.build_pointercast arr typ "list3" builder in
    let values = List.map (expr builder) se_l in
    let buildf index value =
      let arr_ptr =
        L.build_gep arr [| (L.const_int i32_t (index+1)) |] "list4" builder
      in
      ignore(L.build_store value arr_ptr builder)
    in
    List.iteri buildf values ;
    ignore(
```

```
        L.build_store
          arr
          (L.build_struct_gep struct_ptr 0 "list5" builder)
          builder
      ) ;
      ignore(
        L.build_store
          (L.const_int i32_t (List.length se_l))
          (L.build_struct_gep struct_ptr 1 "list6" builder)
          builder
      ) ;
      ignore(
        L.build_store
          (L.const_int i32_t 0)
          (L.build_struct_gep struct_ptr 2 "list7" builder)
          builder
      ) ;
      struct_ptr

and stmt builder =
  let (the_function, _) = StringMap.find !current_f.S.sfname function_decls
  in function

  | S.SBlock sl           -> List.fold_left stmt builder sl ;
  | S.SExpr (e, _)        -> ignore (expr builder e) ; builder
  | S.SReturn (e, _) -> ignore (
      match !current_f.S.styp with
        A.Datatype(A.Void) -> L.build_ret_void builder
      | _                  -> L.build_ret (expr builder e) builder
    ); builder
  | S.SAssign (s, e, _)   ->
      let expr_t = Semant.sexpr_to_type e in (
      match expr_t with
        A.Listtype(A.Void)  ->
          if (StringMap.find s !list_lookup = A.Void) then
            builder
          else
            let typ = StringMap.find s !list_lookup in
            let struct_ptr =
              L.build_malloc (lookup_struct (A.Datatype(typ))) "voidassign1" builder
            in
            let typ = L.pointer_type (ltype_of_typ (A.Datatype(typ))) in
            let arr = L.const_pointer_null typ in
            ignore(
```

```
          L.build_store arr
            (L.build_struct_gep struct_ptr 0 "voidassign2" builder)
            builder
        ) ;
        let size = L.const_int i32_t 0 in
        ignore(
          L.build_store size
            (L.build_struct_gep struct_ptr 1 "voidassign3" builder)
            builder
        ) ;
        ignore(L.build_store struct_ptr (lookup s) builder) ;
        builder

    | _                   -> ignore (
      let e' = expr builder e in
      L.build_store e' (lookup s) builder
    ) ; builder
    )
| S.SIf(condition, if_stmt, else_stmt) ->
    let bool_val = expr builder condition in
    let merge_bb = L.append_block context "merge" the_function in

    let then_bb = L.append_block context "then" the_function in
    add_terminal (stmt (L.builder_at_end context then_bb) if_stmt)
    (L.build_br merge_bb);

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

    ignore (L.build_cond_br bool_val then_bb else_bb builder);
    L.builder_at_end context merge_bb
| S.SWhile (predicate, body) ->
    let pred_bb = L.append_block context "while" the_function in
    let body_bb = L.append_block context "while_body" the_function in

    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

      br_block  := merge_bb;

      ignore(L.build_br pred_bb builder);
```

```
        add_terminal (stmt (L.builder_at_end context body_bb) body)
          (L.build_br pred_bb);

      ignore (L.build_cond_br bool_val body_bb merge_bb pred_builder);
      L.builder_at_end context merge_bb
  | S.SBreak ->  ignore (L.build_br !br_block builder);  builder

 (* Lookup gives llvm for variable *)
and lookup n  =
  try
    StringMap.find n !local_vars
  with Not_found ->
    StringMap.find n !global_vars
in

(* Declare each global variable; remember its value in a map *)
let _global_vars =
    let (f, _) = StringMap.find "main" function_decls in
    let builder = L.builder_at_end context (L.entry_block f) in
    let global_var m (n, e, _) =
        let init = expr builder e
        in StringMap.add n (L.define_global n init the_module) m in
    List.fold_left global_var StringMap.empty globals
in global_vars := _global_vars;

let build_function_body fdecl =
    let (the_function, _)  = StringMap.find fdecl.S.sfname function_decls in
    let builder = L.builder_at_end context (L.entry_block the_function) in
    current_f := fdecl;

    (*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 (n, t) 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
        in
        let add_local m (n, t) =
          match t with
          | A.Listtype(it)  ->
```

```
            ignore(list_lookup := StringMap.add n it !list_lookup) ;
            let local_var = L.build_alloca (ltype_of_typ t) n builder in
            StringMap.add n local_var m
          | _              ->
            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.S.sformals (Array.to_list (L.params the_function))
        in
        List.fold_left add_local formals fdecl.S.slocals
      in
      local_vars := _local_vars;

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

      (* Add a return if the last block falls off the end *)
      add_terminal builder (match fdecl.S.styp with
        A.Datatype(A.Void) -> 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
```

---

## 8.9 exceptions.ml

```
exception CannotPrintType
exception DuplicateFunctionName of string
exception DuplicateVariable of string
exception FunctionNameReserved
exception FunctionNotDefined of string
exception IncorrectArgumentType of string
exception IndexError
exception InvalidBinaryOperation
exception InvalidCondition
exception InvalidExecFormat of string
exception InvalidIfStatementType
exception InvalidListAssignmentOperation
exception InvalidListElementType
exception InvalidSliceOperation
exception InvalidUnaryOperation
exception ListElementsOfVariantType
exception MalformedTokens
```

```
exception NothingAfterReturn
exception SyntaxError of string
exception TypeMismatch of string
exception UndefinedId of string
exception WrongNumberOfArguments
exception BreakOutsideOfLoop
```

---

## 8.10 node.c

```c
#include <stdio.h>
#include <stdlib.h>
struct NumNode {
	int val;
	struct NumNode* next;
};

struct StringNode {

	char* val;
	struct StringNode* next;
};


void print_list_num(struct NumNode* head){
	// prints the list of items in LinkedList
	while(head){
		printf("%d\n", head->val);
		head = head->next;
	}
}

void print_list_string(struct StringNode* head){
	// prints the list of items in LinkedList
	while(head){
		printf("%s\n", head->val);
		head = head->next;
	}
}
```

---

## 8.11 Tests

### 8.11.1 testall.sh

```sh
#!/bin/sh

# Regression testing script for Newbie
```

```
# Step through a list of files
#  Compile, run, and check the output of each expected-to-work test
#  Compile and check the error of each expected-to-fail test

# Path to the LLVM interpreter
LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

# Path to the LLVM compiler
LLC="llc"

# Path to the C compiler
CC="cc"

# Path to the newbie compiler.  Usually "./newbie.native"
# Try "_build/newbie.native" if ocamlbuild was unable to create a symbolic link.
NEWBIE="./newbie"
#NEWBIE="_build/newbie.native"

# Set time limit for all operations
ulimit -t 30

globallog=testall.log
rm -f $globallog
error=0
globalerror=0

keep=0

Usage() {
    echo "Usage: testall.sh [options] [.n00b files]"
    echo "-k    Keep intermediate files"
    echo "-h    Print this help"
    exit 1
}

SignalError() {
    if [ $error -eq 0 ] ; then
    echo "FAILED"
    error=1
    fi
    echo "  $1"
}
```

```
# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to difffile
Compare() {
   generatedfiles="$generatedfiles $3"
   echo diff -b $1 $2 ">" $3 1>&2
   diff -b "$1" "$2" > "$3" 2>&1 || {
   SignalError "$1 differs"
   echo "FAILED $1 differs from $2" 1>&2
   }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
   echo $* 1>&2
   eval $* || {
   SignalError "$1 failed on $*"
   return 1
   }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
   echo $* 1>&2
   eval $* && {
   SignalError "failed: $* did not report an error"
   return 1
   }
   return 0
}

Check() {
   error=0
   basename=`echo $1 | sed 's/.*\\///
                         s/.n00b//'`
   reffile=`echo $1 | sed 's/.n00b$//'`
   basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

   echo -n "$basename..."

   echo 1>&2
   echo "###### Testing $basename" 1>&2
```

```
   generatedfiles=""

   generatedfiles="$generatedfiles ${basename}.ll ${basename}.s ${basename}.exe
${basename}.out" &&
   Run "$NEWBIE" $1 > "${basename}.out" &&
   Compare ${basename}.out ${reffile}.out ${basename}.diff

   # Report the status and clean up the generated files

   if [ $error -eq 0 ] ; then
   if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
   fi
   echo "OK"
   echo "###### SUCCESS" 1>&2
   else
   echo "###### FAILED" 1>&2
   globalerror=$error
   fi
}

while getopts kdpsh c; do
   case $c in
   k) # Keep intermediate files
      keep=1
      ;;
   h) # Help
      Usage
      ;;
   esac
done

shift `expr $OPTIND - 1`

LLIFail() {
  echo "Could not find the LLVM interpreter \"$LLI\"."
  echo "Check your LLVM installation and/or modify the LLI variable in testall.sh"
  exit 1
}

which "$LLI" >> $globallog || LLIFail

if [ ! -f printbig.o ]
then
```

```
    echo "Could not find printbig.o"
    echo "Try \"make printbig.o\""
    exit 1
fi

if [ $# -ge 1 ]
then
    files=$@
else
    files="tests/test-*.n00b"
fi

for file in $files
do
    case $file in
    *test-*)
        Check $file 2>> $globallog
        ;;
    *)
        echo "unknown file type $file"
        globalerror=1
        ;;
    esac
done

exit $globalerror
```