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
Table of Contents

1. Introduction

2. Language Tutorial
   2.1 Setup
   2.2 Getting Started

   3.1 Lexical Conventions
   3.2 Character Set
   3.3 Line Terminators
      3.3.1 Physical Lines
      3.3.2 Logical Lines
      3.3.3 Explicit Line Joining
      3.3.4 Indentation
   3.4 Tokens
   3.5 Comments
   3.6 Keywords
   3.7 Strings
   3.8 Syntax and Semantics
   3.9 Statements
      3.9.1 Expression Statements
      3.9.2 Declaration Statements
      3.9.3 Control Flow Statements
      3.9.4 Loop Statements
   3.10 Types
   3.11 Primitive Data Types
      3.11.1 bool
      3.11.2 char
      3.11.3 num
      3.11.4 string
   3.12 Lists
   3.13 Type Inference
   3.14 Automatic Initialization
   3.15 Operators and Expressions
   3.16 Assignment
   3.17 Operators
      3.17.1 Arithmetic Operators
      3.17.2 Logical Operators
      3.17.3 String Operators
      3.17.4 Relational Operators
      3.17.5 List Operators
4. Project Plan
   4.1 Planning Process
   4.2 Project Timeline
   4.3 Roles and Responsibilities
   4.4 Software Development Environment
   4.5 Project Log
5. Architectural Design
   5.1 Block Diagram
   5.2 Components
      5.2.1 Scanner
      5.2.2 Parser
      5.2.3 Ast
      5.2.4 Sast
      5.2.5 Semantic Checker
      5.2.6 Codegen
6. Test Plan
   6.1 Test Suites
   6.2 Test Automation
7. Lessons Learned
   7.1 John Anukem
   7.2 Clyde Bazile
   7.3 Braxton Gunter
   7.4 Terence Jacobs
   7.5 Sebastien Siclait
   7.6 Future Teams
8. Appendix
   8.1 Makefile
   8.2 newbie.ml
   8.3 ast.ml
   8.4 sast.ml
   8.5 scanner.ml
   8.6 parser.mly
   8.7 semant.ml
   8.8 codegen.ml
   8.9 exceptions.ml
   8.10 node.ml
   8.11 Tests
      8.11.1 testall.sh
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.

```bash
$ 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.


3.1 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:

<table>
<thead>
<tr>
<th>and</th>
<th>If</th>
<th>While</th>
<th>Params</th>
</tr>
</thead>
<tbody>
<tr>
<td>or</td>
<td>break</td>
<td>not</td>
<td>no</td>
</tr>
<tr>
<td>class</td>
<td>continue</td>
<td>equals</td>
<td>each</td>
</tr>
<tr>
<td>return</td>
<td>null</td>
<td>print</td>
<td>in</td>
</tr>
<tr>
<td>true</td>
<td>def</td>
<td>for</td>
<td>to</td>
</tr>
<tr>
<td>False</td>
<td>Else</td>
<td>With</td>
<td>from</td>
</tr>
<tr>
<td>Greater</td>
<td>less</td>
<td>than</td>
<td></td>
</tr>
</tbody>
</table>

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']
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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>False</td>
</tr>
<tr>
<td>char</td>
<td>null</td>
</tr>
<tr>
<td>num</td>
<td>0</td>
</tr>
<tr>
<td>string</td>
<td>null</td>
</tr>
<tr>
<td>List</td>
<td>[ ]</td>
</tr>
</tbody>
</table>

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

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

*Access* - returns the element at an index

```plaintext
set a to [4, 5, 6]
a[0] // 4
```

*Update* - updates the element at an index

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

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

```plaintext
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

```plaintext
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

```plaintext
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

```plaintext
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:

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

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.
<table>
<thead>
<tr>
<th>Team Member</th>
<th>Responsibilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>John Anukem</td>
<td>Tester/AST, Standard Library, Codegen</td>
</tr>
<tr>
<td>Clyde Bazile</td>
<td>Language Guru/Codegen</td>
</tr>
<tr>
<td>Braxton Gunter</td>
<td>System Architect/Scanner, Type Inference, sast, semant</td>
</tr>
<tr>
<td>Terence Jacobs</td>
<td>Project Manager/Parser</td>
</tr>
<tr>
<td>Sebastien Siclait</td>
<td>Tester/Testing</td>
</tr>
</tbody>
</table>

### 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

e5b3625bca1a6973f59d1890d24da1802f115ecc Merge pull request #42 from terencej23/dev  
1266c40126e5acc4fa41d66200aa8c530ca846 Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
085dec572a1d72a9b313e96a7e3ac64dd9691 Add binary search  
ee7d46a710b7024a6348fcc80a162b0d55cc4b4 all tests functional  
5af662c1f105b1457b753de96ce751c0880db all but while loop tests updated  
cc39c127fca674af1d2f235cad4e7b317c163 Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
d741f0602ae1b7691eb83151dbca01b0c240a00 new tests (precedence+parameters) dropped old tests  
39ce81a4a6fd25deddfc7329b23ca91da5be79 Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
13a0223d804c124913208e76510a476898ba1249 Add recursive binary search for presentation  
cc22d9f40ce7b78102ee1a1ae496472815c273 Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
bcf43ee10ffea8dbb37e02f0a0ecbae6b8e27c2 formatted rec_lib and cleaned up test script  
5f90deb19aabbc75b74d5f50a6429765b2cbb848 Merge pull request #41 from terencej23/dev  
f51bd9018c2ee11315fdcbcf3d6689b8e27c23a91da5be79 Merge pull request #40 from braxeatssnacks/master  
79a495997d32761cf66222cc240943b971c9839e list can initialize  
36c3352c76508f09d90cac57546d84d0c5f03f05 Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
40fa1d5e8c1e6ebebc854b37b2c0a1120aaba4c Merge branch 'dev' of https://github.com/terencej23/newbie into dev  
837b3a8066b9629b6d95b7160695092b2b22f0 Merge pull request #39 from terencej23/dev  
ccd2f69c021710ff8b987d9d57c81c66c3e901dd Merge pull request #38 from braxeatssnacks/master  
f17db5d21801a05d37c8097f133ba3f3ce1349a7 further progress on codegen for list
https://github.com/terencej23/newbie
84657fcd563d5dc7948f40d3324055e2bb5408a Merge branch 'master' of
github.com:braxeatssnacks/newbie
da309db42898e668f8d83c6ede2113e08fcf76f Prints hello world!
4c07a5692516f1292c09759528dee02e4c7cc199 Merge pull request #23 from terencej23/dev
4b56deaa25c409545a16175a5f3dd043c3716db98 newbie.ml update
35b0533bc8464ca7335883f33e9e028db769ba Print LLVM code for 'hello world'
9cd205687978e504e2c349ceef13064a428e91b0 Merge pull request #22 from terencej23/master
d142701920b0c1de4a8a90079a2f153200ed2269 Merge pull request #21 from braxeatssnacks/master
2cf117a247f2e6255b4ded97913f8af7a9e8fee Merge branch 'dev' into master
d0331c1e17ba7f4855c851b4f41cf0cd7277ca5 Merge branch 'dev' into master
be48f06472f4775fb30fe7f422a24db3af793d codegen (sast)
c8833e1564a0f715876819879b959204e9c0580 prime top-level for new codegen
cba90c5fd430b8819cdd2da0a2a6ab6d6e2b3be2d add clyde's codegen
256a26cefe06288170560a26fcb1b68886258a3 Merge pull request #20 from braxeatssnacks/master
ad161fb8f063748801c4b6a1da9e975a23c8ca comment out prime for iteration
89b22119b315e2b91b9a501c881d0b596762a158 Build function body
5d6663c89c70e4e13746fdeba319550307e9b9a Compiles successfully! Adds global and local vars.
9cf2aee30a114a5657c37ba361e32a89a9608240 Print LLVM
c43379ea7931111b7a761f8bf8a8c9c54dc3 Update
a5524792a3122d053493812ce4b7bab1cbbdd9e0 Print LLVM code
9508ef7e8a3d522e65c9f307a6efbb838e07ca0f remove extraneous file
ff92401d4363dacc1ca4e576d28b9601279b54 Merge pull request #19 from braxeatssnacks/master
82679268379a19d9c3ea9c728402066a8e0315d resolve conflicts
9ee8872d71f58c4cadcc8185bcd2e902ae6a5cad3 resolve conflicts
f383d3d03f706e05c4a4d4be4e102f159a78b84001 fix entrypoint
ba826a14045bf4a11f36b452ff14c60f8510cc3 added sformals
2dad3b2f017b95f64dc852957279485e9454138 naming module fix -- segmentation fault in codegen
dba530a989e9c0955f51d361a3c2439da2395e Added Node Class to Makefile
d3150840c9900af3566a31b422794b04c66a2d semantic checking and type inference
b0214d836615e0c7de93573508856d0cadb type inference sfdecl updating done
0e79e2b54e18479e5b2df0dcc12615152a9ad32 WIP semantic checking won't build functions other than main
c49381ff53ec70a969f398a28bb12a6a871c442 semantic checking and type inference
0a51b7a560a552845ae3f9c61ce6f4225a4289ecfix scanner newline
32c4a2f5d607acc993ffd8c424adfe45f1179a5f added sformals
e4f869128480ce34ee2ee28bba98c0119dc59aae naming module fix -- segmentation fault in codegen
a258ce78071861ce6355562003911dc9b2ca32e interim edits -- need free parser
3a46f3ee5480bd63c32042d7ae7475898711757 fix entrypoint
3f9da2a693096a0da1c5b5fca60df7d53cd232815 Merge branch 'master' of
github.com:braxeatssnacks/newbie
5114c010df019636147942e035b1f9462f2a886 added sformals
8f6c7e868f8d822d6936506bf126d81b9214ed8 naming module fix -- segmentation fault in codegen
3a2962cbf0ea3a3d3029ad57277c40a8c4f36e interim edits - need free parser
62c477e42a25bb37e189009a12951f362162b4 Merge branch 'master' of
github.com:braxeatssnacks/newbie
5312a064a46e6f1147b930223870806333ab00 Merge pull request #18 from braxeatssnacks/dev
831e7060e9d8b77878460524e42722ebc358c1a1b58 Merge branch 'dev' into dev
3a3d223f0ec8fd7d7577f6c60619257950eb25b7 Merge branch 'dev' into dev
semantic checking and type inference

d2a6ea02f93eb93a52da1406b594d85c0f160e Merge pull request #17 from braxeatssnacks/dev
e431c87c33498e87998835b5aea2a43e52f876073 type inference sfdecl updating done
f57f9785a6158ea6cf727d0d30d53801fbfcf3 Merge pull request #16 from braxeatssnacks/dev
bf4966da99e1f26c5d0fbaabbb4af732b1b5680 WIP return type not bounced up in fdecl_to_sfdecl
13c26ae76b82ca80ef21bf410e9a44af2c096541 Added Node Class
760d7b9a42628c7fda257578941d7e50ba0f7d2 Merge pull request #15 from braxeatssnacks/dev
00d587d15e15a246c31f7423079432f31d4b4ebb WIP semantic checking won't build functions other than main
489c0dea4a1d21d643b22cfd44f294cac6469988 semantic checking and type inference
29b669280805e078417e612afbe2e787aa0230e fix scanner newline
d4434382119ba8220935da45083cc55a78aaccb added sformals
0489a4d4c089a396519b26a1a6a151729ef64e428 naming module fix -- segmentation fault in codegen
f6ace3d62db055204d41f14f86c0efef85f3f3 interim edits -- need free parser
fbd6c7f7e1f18621d7b59292c9e8a4a09a3ed0aa Merge pull request #14 from terencej23/dev
4e28c151ceb65e987ab7a12fd163880f872ed0 Merge pull request #13 from braxeatssnacks/master
288e3db350e02a2929293f7d3c7ee5e0efb89de edits
eff486af703699a4ce732308a383f33ce17d044f6 added sformals
9d93432e4614e6414695076ee0395951de2a9a fix segmentation error
e66285b7f538ec529ff29c4a54e2b568c0e2cb3b naming module fix -- segmentation fault in codegen
337773c4d7fb0a78d7377b3d36f821776169ce added sformals
ff3a1d6b348ed92c23e9e8ce739efb02010603 Merge branch 'master' of github.com:braxeatssnacks/newbie
045ad84ed20bc8a55ed3d81b12d389daf7fad1 interim edits -- need free parser
6db80d4e2ed882e65a5aad19e3d3f216680c29c interim edits -- need free parser
3571eb4d484859c843a512c1d16c87ec629641 Merge pull request #12 from braxeatssnacks/master
87e18e336621be01fc3640067bccccd9b97f693 parser requires main method
a9bbd175f478b151913f94c668b8b2c11dfdc93 Merge branch 'master' of github.com:braxeatssnacks/newbie
c511b70ab14f57a260771e7f21f2cb9f8344d2ce AST IS FUNCTIONAL
6f282d0138e333d5a386f5e4afda641a1961ccf196c one bug left; commented out 'for' stmt in ast.ml
23b3d7f7e707014afbf1102b4e093de819459 unfinished - code-gen piece for hello world
868da4d1e2024af1c41d6865c4be5eb581a18b8f9e AST IS FUNCTIONAL
a4bb8fd7d1e74d7781202d86667c8422c91081 one bug left; commented out 'for' stmt in ast.ml
a707e1644f170ac82d816b28bc0de2d30e1d217 Merge pull request #11 from braxeatssnacks/integrate-codgen
c49df262471b3430b04eddf89a506b18e0dfe33 Merge branch 'dev' into integrate-codgen
becc829a3394a595595047bf1f055958148b0eb9 Merge branch 'dev' into integrate-codgen
79538a9504c6362246a457373ba1323f2ec56c unfinished - code-gen piece for hello world
11ca8ff0e0c2df6d80074c9e305523153629f6a4 Merge pull request #10 from braxeatssnacks/master
f4b69057085e4a8cddf80a4e281440bfc6b9c6e add intermediary cache for scanner token list to parser
57e26325e4e8c5c6b8bba17415b3767f10f4ac54 add intermediary cache for scanner token list to parser
ad99a5226897e7878142e7711343240e754c36 Merge pull request #9 from clyde-bazile/master
51a3c83f3b97b35b371afa2cd7ea7d55a624abad9 init codegen
3d1516ff0f5a576205a8cad148769b78d2d4d7c Merge pull request #8 from braxeatssnacks/master
265939d5e05933caee7b9472d6e232e82152c2 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

1c522a1bd13d38421fa50fd9a52b2d5997978ef9 get rid of shift/reduce conflicts with associativity

a77e72e925e45d17b07d26186d6ff121ff9c42 add newline character

52840ea7776c9a9d19c4283beee36f06365146a5 parser file added

23a73270452621c046558b2aae05aee48b90e0 Merge pull request #6 from braxeatssnacks/master

fb8542830943b54d047da682971d00b8d64faa0 add overarching entrypoint newbie.ml -- makefile not currently working

23d4252ab23dd3521ae9c800df49ce13e0126255 Deleted something

c78f7d19c9df7f7b9d814a99c26a20312658f1 Added something
d046b46476003202750767b9bb3a071ec49ca6 Merge pull request #5 from johnanukem/master
e7b29f640f39de5583cc76defd51ed4f9c17 Added ast

5037e57f6989d4a3e576ad68752e0dd17889cc5 added microrc directory

8d10a334cbac569117952560cf4d3589d78b0e8a add print keyword

f0610f56a9b4c29d671ab43a82d28a4262bf52 token resolution

29ed12b72a71d5293f39f212914147a6f5f9d3 Merge pull request #1 from

braxeatssnacks/semantic_setup

413d330de73eb3ec694091e299353a633d08097 resolve conflicts

f4b992be9e24ec12251b046d69b1e2aa28daaef make literals distinct

1c0afa85cd4cc2deb890c6f02b5792e46a681c Merge branch 'master' of

github.com:braxeatssnacks/newbie

7462e390e29f4d974a36c1f4a3d5d6b16cc1eeec8 Update README.md

2b0551d8f57d5aaee42a9893b247116b5faabc48 add comment functionality

f8d6eda81b1a3d46b3e0c050e683585d211eab20 add comment functionality

c141ca6e05c4217163a177c3dcab8abb52565689 Merge branch 'master' of

https://github.com/terencej23/newbie

b5dd73aeeef32620f8ce2755a045d48f6b6ecc66e Merge branch 'master' of

https://github.com/terencej23/newbie

289614cf609c5c688c0c78f56efe6bb2d1d978e Added ast

31875d8cf6101b6d2b76477ab50d15b81ae1a5c7 Merge pull request #4 from

braxeatssnacks/semantic_setup

0aee3c782cf654e946f8bd220abd194915630b67 fix exhaustive pattern matching

b2523478d6263f5d0c7339b02ea9d6b92b0232 Merge branch 'semantic_setup' of

github.com:braxeatssnacks/newbie into semantic_setup

f1e790882d3727a32084c18adeaa7272f52d481 read up young champs

320ace658e06027139608860d7d5a9435047dee18 added keyword

33c741757d81f1edebca24666edc7d3c4097f05 add delimit tests
e074058a6c6da05e49f9ac5673ff6609c723a66 generate indent and delent tokens based off \t

6d021780473e571b59325e578ccc4313b6eacaa generic makefile for scanner compilation

bd2deee39543ce0e6939f6ba77265ed785f927e add few testcases
c6d83ac7b41a11e7f17871ab13c2e0f2b2bb4fa2 allow passing of cmd_line arg path/to/file for testing
eaafaf56c7289d21cbe85e0f1d1c9ac0e2d9e5 add more token recognition; prime INDENT/DEDENT

a939cd59793e20b4b2f42b6c7bed43a60a789b add string support

922d9e4f1538c3ea6b8551b74fb73666dcf149 add output capacity

1460d797ca7475e95d3827019ff24c5bf093c9 add basic tokens and matching

5e2deee28992ba259473a015b800b36a78b4e98 add verbose err reporting functions

2021ff0d7f4f9cc4a3ce301d060ec564cad336 setup keyword scanning

f0f1f726af9b6daa092f478c28845519adb574a read up young champs

baa58cd17000bef25eb92a2ad54defea2dbe3386 added keyword

64cf4f8a83b2f7344dcf6b20152edd0f4ea3fa add delimit tests
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.cmo
exceptions.cmo:
exceptions.cmx:

node : node.c
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 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 | COMPIL | 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", LLVM_IR) ; (* generate, do NOT check *)
            (*-c", COMPIL) (* 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 =
  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)
  LLVMIR-> let the_module = Codegen.translate gen_sast in
            print_endline (Llvm.string_of_llmodule the_module)
  COMPIL-> 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 ll_fname =
              let basename = Filename.basename fname in
              (Filename.remove_extension basename) ^ "\n"
in
let outFile = open_out (ll_fname) in
  Printf.fprintf outFile "%s\n" (llvm_code) ; close_out outFile ;
  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\n\n" (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\nt%s"
                        (string_of_expr e) (string_of_stmt s)
| If(e, s1, s2) -> Printf.sprintf "if (%s)\n\nt%s\n\nelse\n\nt%s"
               (string_of_expr e) (string_of_stmt s1) (string_of_stmt s2)
| While(e, s) -> Printf.sprintf "while (%s)\n\nt%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;"

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\nt%s"
                          (fdecl.fn
                           ) (String.concat ", " (List.map (fun x -> x) fdecl.formals))
                          (String.concat "\n\n" (List.map string_of_stmt fdecl.body))

let string_of_program (vars, funcs) = Printf.sprintf "%s\n\n%s"
                                    (String.concat "\n\n" (List.map string_of_assign vars))
                                    (String.concat "\n\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 *)
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"
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_suop 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\n\telse\n\t%s"
    (string_of_sexpr se) (string_of_sstmt ss1) (string_of_sstmt ss2)
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) \n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
\n\t%<%s>
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 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 = [' ' '
' ' ' '	']

(* 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 }
(* 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"
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 */
%
ten 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
%
ten <int> INTLIT
%token <float> FLOATL
%token <string> STRLIT
%token <string> ID
%
%nonassoc NOELSE
%nonassoc ELSE
%left OR
%left AND
%left EQUALS
%left LT GT LEQ GEQ
%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:
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
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;
}

(* 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 locals_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 locals_map) then
      StringMap.remove name map
    else
      map
    in
    List.fold_left remove_formals env.env_flocals (StringMap.bindings env.env_flocals)
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 statements *)
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
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)

(* 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
let env_ref = ref(env) in
let convert_stmt stmt =
    let (new_stmt, env) = stmt_to_sstmt stmt env 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 (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)
  else
    let new_env = {
      env_fmap = env.env_fmap;
      env_fname = env.env_fname;
      env_return_type = typ;
      env_globals = env.env_globals;
      env_focals = env.env_focals;
      env_in_loop = env.env_in_loop;
      env_set_return = true;
      env_sfmap = env.env_sfmap;
    }
    in
    (SReturn(sexpr, typ), new_env)

(* if return type has been set, check new return type *)
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;
          }
          (SAssign(var, sexpr, new_typ), new_env)
    else
      let msg =
        Printf.sprintf "var %s has been redeclared with listtype %s" var (string_of_typ (Datatype(t2)))
      in
      (raise (E.TypeMismatch(msg)))
  else if (new_typ = 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;
    }
    (SAssign(var, sexpr, new_typ), new_env)
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 binary 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_sexp (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_sexp 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_sexp expr env in
            let typ = sexpr_to_type sexpr in
            (typ :: l)
in
            List.fold_left get_type [] e_l
        |> List.rev
    in
    (* check for correct number of args *)
    let fdecl = StringMap.find id env.env_fmap in
    if (List.length e_l <> List.length fdecl.formals) then
        (raise (E.WrongNumberOfArguments))
    else
        (Scall(new_s, [se], Datatype(Int)), env)
else if (id = env.env_fname) then
  let (e_l, env) = expr_list_to_sexp_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_sexp_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_sexp_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"]);
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_sexp 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)
and check_list e_l env =
  if (List.length e_l = 0) then
    (SList([], Listtype(Int)), env)
  else
    let (first_sexp, env) = expr_to_sexp (List.hd e_l) env in
    let target_typ = sexpr_to_type first_sexp 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_sexp 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_sexp :: se_l in
    ((SList(final, Listtype(list_typ))), !env_ref)

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 = expr_to_sexp e1 env in
    let se2 = expr_to_sexp 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_sfrmap = StringMap.empty;
}

(* 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
 receptions
 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_sfrmap = 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"]

(* 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

(* 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))) |

(* 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

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("printf", [e], _)  ->
  L.build_call print_func [] str_format_str builder; (expr builder e)]
  "printf" builder
| S.SCall("println", [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 [] float_format_str builder; (expr builder e)]
  "printf" builder
| S.SCall("printbool", [e], _) ->
  L.build_call print_func [] float_format_str builder; (expr builder e)]
  "printf" builder
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

(* | A.Neq   -> L.build_fcmp L.Fcmp.One *)

| A.Deq   -> L.build_fcmp L.Fcmp.Oeq  
| A.Leq   -> L.build_fcmp L.Fcmp.Ole  
| A.Geq   -> L.build_fcmp L.Fcmp.Oge  

_ -> 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 struct_ptr = L.build_struct_gep struct_ptr 0 "access2" builder

  "idl"

  builder

  in

  let res = L.build_grep arr [] idx [] "access3" builder in
  let res = 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 struct_ptr = L.build_grep struct_ptr 0 "access2" builder

  "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_grep 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.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 !curr
| 
| match curr_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 )
| 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.SWWhile (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.buildalloca (ltype_of_typ t) n builder in
StringMap.add n local_var m
| _ ->
let local_var = L.buildalloca (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
#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
#!/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
  if diff -b "$1" "$2" "$3" 2>&1 |
    then
    SignalError "$1 differs"
    echo "FAILED $1 differs from $2" 1>&2
  fi
}

# 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/.*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"
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