Rochester Institute of Technology

Independent Study Report

Compiler Construction II

Ross Bayer

December 2016

Supervised by Professor Matthew Fluet

Contents

Introduction

A number of semesters ago I had the fortune of taking the graduate Compiler Construction course taught by Professor Fluet which ignited my continuing interest in Programming Language Theory and language implementation. The class takes a project-based approach to learning about compilers focusing on a series of five projects that implement parts of a compiler for the small language *LangF*. The projects focus on the five main stages of compilation: scanning (also known as lexing), parsing, type-checking, optimization and code generation. Each project builds upon the work and understanding of previous projects and directly on the lecture material. The *LangF* language is relatively small and syntactically resembles ML. Notably there is no module system and no operator overloading, however the language does include parametric polymorphism and type inference.

This independent study is meant to act as an extension of the Compiler Construction course and allow me to delve into more advanced topics that were not included in the curriculum. More specifically this independent study focuses on the Static Single Assignment (SSA) Intermediate Representation and related optimizations. To facilitate my studies of SSA I spent the majority of my work implementing an extension to the *LangF* compiler that introduces a new SSA IR. The new IR includes a conversion step from ANF IR, an interpreter, type-checker and a handful of optimization passes. Professor Fluet and I met with every week to review my implementation progress, answer questions and discuss any problems I encountered during my research or implementation.

Intended Work

The intended work for this independent study was tentatively broken down into three sections. The first section was to be focused on researching Static Single Assignment and then implementing a new intermediate representation in the *LangF* source code that takes place after the ANF IR. Research was to consist of reading chapters from a selected textbook and by examining actual implementations of SSA in existing compilers, such as MLton. The new SSA IR was intended to include an associated converter, type-checker and a few optimization passes.

The second section of work was to focus on researching the Low-Level Virtual Machine project (LLVM) and then to implement a translation phase from my SSA IR implementation to LLVM IR. A back-end for LLVM would replace the current stack-based VM that exists and provide a more "native" code-generation facilitated by the LLVM project. The LLVM project is a toolkit for language development that is used as a generic language back-end, capable of targeting a large variety of machine architectures from their single intermediate representation LLVM IR. LLVM IR acts a lot like a portable assembly language with very granular control over low-level details. In reality LLVM IR is a form of SSA.

The third section of work was vaguely mapped out to be a whirlwind tour of garbage collection. I was to read through portions of "*The Garbage Collection Handbook: The Art of Automatic Memory Management*" by Richard Jones, Anthony Hosking and Eliot Moss and then implement a very small and probably naive garbage collector and associated runtime for the LLVM back-end. The LLVM back-end would initially have included stubs for a runtime and garbage collector that would later be replaced if time permitted. It was likely that the LLVM back-end would have initially allocated memory without any intention to free unused resources.

The proposed timetable for this independent study was as follows:

Weeks	**Goals**
1-2	Research SSA
2-3	Implement SSA IR structure in *LangF*
4-5	Research and implement SSA optimization passes
6-10	Research LLVM and implement SSA IR to LLVM translation pass
10-15	Research garbage collection schemes and implement primitive garbage collector in *LangF*

Actual Work

In this section I summarize the actual work completed over the course of this independent study. Summaries are provided on a weekly or sometimes bi-weekly manner. The focus will be on discussing implementation progress, roadblocks and a general summary of what was completed during the course of the semester. For a more detailed look of Static Single Assignment and discussion of my implementation refer to section *Static Single Assignment*.

Week 1

There was not a lot accomplished the first week as it was primarily spent constructing the independent study proposal, then meeting with Professor Fluet to finalize the document and submitting it to the CS office. I did have time to begin reading chapter 5 in *Engineering a Compiler* which discusses different intermediate representations and provides a quick introduction to SSA.

Professor Fluet provided me with the source code for the final version of the *LangF* compiler project. With Professor Fluet's permission I used the initial source to create a private github repository and added Professor Fluet as a contributor. A small amount of infrastructural modifications were made to the initial project source using my previous work on the MLton project to convert the LangF build system from GNU make to CMake. Originally I stripped out support for SML NJ, which I did not intend to use at the time. However around week 11 I decided that using MLton to incremental development was slowing me down, especially for quick code modifications, thus I restored support for SML/NJ builds. The final CMake build system supports targets for building the `langfc` and `langfi` executables using either MLton or SML/NJ.

Weeks 2 and 3

Weeks 2 and 3 were spent researching SSA starting with selected readings from "*Engineering a Compiler*". I completed my reading of chapter 5 which is mainly an overview of different intermediate representations. Primarily the chapter served to clarify a number of terms used commonly when discussing IRs. The majority of the chapter is devoted to discussing graphical, linear and hybrid IRs. Graphical IRs encode a program in terms of a graphical

object, which could be nodes, edges, tree or lists. Linear IRs are simply linear sequences of instructions, similar to assembly language. Lastly there is hybrids, like SSA which are use elements of both. SSA is a directed-graph of linear sequences (basic blocks).

Chapter 9 is devoted data-flow analysis, it explores SSA form, dominance and constant propagation. There is a lot of discussion on the structure of SSA and how it behaves, as well as how to translate from a first-order language to SSA. However, this does not quite fit my use-case as *LangF* is a higher-order polymorphic language and my SSA will remain as such. The other sections on dominance frontiers and constant propagation are still mostly relevant.

Week 4

My progress in week 4 was less optimal than I had initially hoped. Sunday and Monday were spent sick in bed with a cold.

I had initially wished to start implementing the SSA IR conversion pass only to realize that while my previous research into SSA provided a basic understanding of the IR and how to construct it from a relatively simple first-order source language, it did not detail how to convert from a higher-order language. After some quick searching through the MLton docs I found my answer, I need to write my own form of their *defunctionlization* and *closure-conversion* to lift lambda/closures into the top level. I read through the paper "*Flow-directed Closure Conversion for Typed Languages*" by Henry Cejtin, Suresh Jagannathan, and Stephen Weeks which outlines how MLton solved this exact issue. I'll be honest, there is a lot to process in the paper and it did not all appear relevant to LangF, which will not have a simply-typed SSA, rather it requires parametric polymorphism as the language does not enforce that recursive calls have exactly the same type arguments.

After spending hours reading this paper and working on the general structure and signature definitions for my SSA implementation I was left with a lot of questions. Rather than continue down the road of futility I went and worked on implementing the skeleton for the ANF to SSA conversion pass, type-checker and SML structure and signature for the SSA IR.

I went to the weekly meeting with Professor Fluet and after discussing my questions on defuctionalization and closure conversion it was made clear that such topics were beyond the original scope of this study.

Week 5

Week 5 was spent working on the conversion phase from ANF to SSA, which was proving to be much trickier than expected. Originally I thought that translation would be rather straightforward, however I was still uncertain of what the final SSA signature would look like which made it difficult to progress. I did alter the structure and signature to incorporate my discussion with Professor Fluet in our previous meeting. Now the IR is higher-order and polymorphic.

In our weekly meeting we discussed my troubles with the program representation. Originally I used the MLton definition of a program, which is a distinguished main function and separate datatype declarations and function declarations. MLton can accomplish this because it is first-order, meaning that all function and datatype declarations live at the top level. To accommodate for the design of *LangF* I changed my `SsaIR.Prog` structure to represent a program as a single distinguished main function. Functions, which are really just a collection of basic blocks with a fixed start, have local datatypes and function declarations introduced in their scope. Previously in ANF and earlier IRs a program was considered as a set of declarations and a return value. This wouldn't work with SSA translation, converting top level declarations would yield blocks which must exist inside a function. Rather than making a program an implicit function body, I've decided to make it explicit. We also discussed some of the type signatures and potential strategies for conversion functions.

Week 6

Much like in week 5, week 6 was devoted to the ANF to SSA conversion algorithm. After some time banging my head on the wall hoping to grok some of the particulars of translation, I decided to sit down an spend time to write out ANF programs on paper and translate them to SSA by hand. Once I felt more confident in the translation steps I continued implementing, only to realize that there is a lot of foundational code provided with previous IRs that seems useful. So I switched gears and began implementing some of the same sorts of functionality, this includes some type comparison functionality. Later I found out that my implementation of type equality was incorrect as it did not account for type variable bindings being positional.

I also created a few helper functions for certain structures to make constructing typical use-cases simpler. Such as a function to create a nullary block,

which takes no arguments and only has a list of statements and some transfer. The rest of my time this week was limited due to the career fair and a number of midterms.

Week 7

A large portion of code was written during week 7 for the ANF to SSA conversion pass. A lot of the stubbed out functions were given concrete definitions, including much of the code to traverse the ANF tree and slowly build up blocks from individual components.

The SSA IR signature and structure saw some large changes as well, I put into practice what Professor Fluet and I discussed in our last weekly meeting and now statements can hold not only variable bindings, but also mutually recursive datatype declarations and function declarations. I also had the chance to use a little know feature of SML, the `sharing` clause, which allowed me to declare my many structures in the SSA IR as mutually recursive. The program representation is broken down into a number of sub-structures that define parts of an SSA program. There is the main program structure that hold the main function; functions hold a list of basic blocks, which are sequences of statements which can be declarations for mutually recursive functions. It is the inherent nature of my higher-order SSA to be mutually recursive, if it were first-order then it would be very linear without loops.

Work began implementing some supporting utility functions for the SSA IR, including many layout functions used to output a human-readable format for the program. Equality functions were implemented for a number of types and the start of full program cloning utilities were in the works towards the end of the week.

At the close of the week there is still a few small bugs and a large portion of the RHS conversion that needs to be implemented.

Week 8

Week 8 saw the end of the ANF to SSA converter pass, finally! The RHS conversion functions were implemented and some small bugs with converting lambdas were fixed. I had one particularly challenging section of logic to write, when converting ANF declarations it is necessary to not only convert each set of declarations individually, but also ensure that the environment

for each knows about all the previous declarations. To solve this problem it required two passes, the first to construct the basic blocks for each declaration without much care for the environment, except for collecting each new declaration name. While converting in the first pass a thunk was created in place of each block transition, the thunk would allow for a label to be specified at a later time. The next step was to traverse the list of newly generated blocks backwards to connect each together using the thunks. A lot of temporary blocks are generated because of this step.

Week 9

During week 9 I made fixes for a few bugs in the SSA to ANF converter which caused the compiler to fail on a few of the examples. Most of the bugs were directly related to mishandling the environment. Specifically in the `convertDecl` and `convertPat` the `env'` value ought be just the newly introduced values in the environment and not an extension of the initial environment.

I also modified the layout code to make the output more readable, before it was a bit of a jumbled mess for the sake of debugging. I made some modifications to the structure of the SSA IR a bit, specifically changing the return transfer case to only have a single value and the corresponding change to the return in types, which is only a single value.

In my meeting with professor Fluet it was decided that progress was not sufficient to continue on to LLVM and GC work. I had a lot of incremental trouble getting the ANF to SSA conversion step implemented and did not have any of the other tooling around the IR to justify moving on. Thus the focus of this independent study was adjusted to focus more in depth on my SSA implementation and create optimization passes.

Week 10

In week 10 I worked on implementing more foundational structures and functions for manipulating the SSA IR. In total more than 500 lines of code added to `ssa-ir.sml` that provides equality comparison, type substitution, free variables, etc... to types and restructured parts of SSA. The main SSA structure now includes a few sub-structures such as the Bind/Binds and Case/Cases for handling opaque types in the graph. Another portion of my time was spent implementing structures for cloning the SSA graph and for

collecting free IDs. These new structures appear similar to their counterparts in ANF IR and Core IR. Free IDs include not only free variables in parts of the program, but also free type variables, labels (for blocks), dacons (data constructors) and tycons (type constructors).

Equality for types was a massive pain, specifically the case of comparing function types has to be handled very carefully to ensure that positional type variables are respected, even if their names differ. The naive approach would be to walk over all parts of the type and compare equality for each, if any portion fails then the types are not equal. However the naive approach does not handle cases for types such as the following:

$$\forall\alpha.\forall\beta.(\alpha \to \beta) \to \text{List } \alpha \to \text{List } \beta$$
$$\forall\gamma.\forall\xi.(\gamma \to \xi) \to \text{List } \gamma \to \text{List } \xi$$

The above types are equivalent because the type not only has the same structure, but also because the type variables α and β are bound in the same positions as γ and ξ. In order to correctly handle type comparisons that contian type variables it is imperative to consider not only the ordering of elements, but also the bindings of type variables. My implementation uses a similar approach as the ANF type equality, it loops over the type and creates a mapping from type variable to position index and then uses the index to compare if type variables are equal rather than a direct comparison of their names.

The SSA interpreter is still ongoing, this week there were some major implementation milestones. The foundational code to include the interpreter into the `langfi` pipeline was written, however now `langfi` will only run the SSA interpreter rather than the original ANF. This is a small trade-off for the moment, in the future it would be nice to add a command-line option that allows the user to select which interpreter they wish to use.

Lastly I decided that development time was rather slow using only MLton for incremental compilation, thus I re-added support for SML/NJ builds of `langfc` and `langfi`. There are now CMake generated Make targets `langfc-smlnj` and `langfi-smlnj`.

Week 11

The start of week 11 focused on modifying the CMake build system to include some recent fixes to various CMake modules I created for MLton. The SML/NJ build targets now actually create a heapfile using the name given to the command, rather than generating a heapfile with the special SML/NJ platform dependent heap suffix. A handful of new targets were added for convenience's sake, there is now targets for type-checking `langfc` and `langfi` using MLton and targets to generate "def-use" files for both `langfc` and `langfi`.

Later in the week I continued to work on the SSA interpreter which successfully compiles and runs some of the simpler example programs. The interpreter is relatively straight-forward in design, it mimics a few key components of the ANF interpreter, such as the representation of heaps and values. However, interpreting SSA is substantially different than ANF because it is a directed graph rather than a tree. In ANF it was sufficient to walk over the tree and evaluate each branch, with SSA each function is made up of an unordered set of blocks with a distinguished starting label. Needless to say there is a lot of searching through the list of blocks for each transfer case.

I also spent a lengthy amount of time hunting down two major bugs with the ANF to SSA conversion pass. The first bug occurred when converting RHS values that held lambdas, the converter would always generate new names for the lambdas causing later code that relied upon their original names to fail at execution time. To fix this issue the function for converting lambdas now also accepts an optional name parameter, which it will respect. The second bug was hiding in the expression conversion function. Expressions in ANF are essentially `let` expressions with a set of declarations and a final result variable and type. Originally this function would convert the return variable and type before converting the inner declarations into blocks, however this caused the inner blocks to generate a completely different return variable than the one actually returned. Now the declarations are converted first and the resulting environment is used to lookup the return value.

By the end of the week the interpreter worked for all the example programs. Professor Fluet and I went over the code during our weekly meeting and I asked a few questions regarding some particulars of the type-checker.

Week 12

Week 12 did not see a lot of progress on the implementation front. I started working on the type-checker, stubbing out various functions and using the ANF implementation as a basis for the structure.

Week 13

Thanksgiving break and snow day, nothing accomplished.

Weeks 14 and 15

In weeks 14 and 15 I spent wasted a number of hours researching directed-graph libraries for Standard ML only to become frustrated at their complexity. It was then that I decided to work on my own late one evening and after hours of tweaking I was still having troubles getting it to compile. I swiftly abandoned the directed-graph and instead spent the rest of my time this semester implementing four optimization passes for the SSA IR.

The first optimization pass is called "Goto Shrink" and it's job is relatively simple. My SSA IR implementation is essentially a directed graph of basic block that hold code where each graph makes up a function. While converting from ANF IR to SSA IR a large number of blocks is created for the sake of constructing the graph. Many of the blocks in the graph don't actually contain any code and instead have a `goto` transfer at the end. These blocks exist as a side effect of conversion and can be eliminated safely using a simple algorithm. Firstly if a block has only a single child and that child has only a single parent then it is safe to join the blocks. When joining it is important to consider cases where the parent has arguments passed to the child. In this case it is necessary to avoid capturing variables by creating temporary variables with the original values from the parent and then introducing variables with the same names as the arguments for the child block that hold the values save in the temporary variables.

The second optimization pass is the well know copy propagation. Copy propagation is used to remove unnecessary indirection in variables throughout the program. A variable alias is a binding from one variable to another variable in-scope. By traversing the program it is possible to construct a mapping of these aliases and then to reduce them such that they have at most a single layer of indirection. Using the reduced mapping it is a simple matter to walk

over the program and replace uses of variable aliases with their now more direct values.

The third optimization implemented was tail-call elimination, which reduces recursive tail-calls to tight loops inside a function. Using loops in place of a recursive call means that new stack frames do not need to be introduced and thus avoids the problem of stack overflows. Tail-calls are exceptionally important in all functional languages as the only looping construct is a recursive function. For functional languages to be usable they must implement this optimization, otherwise loop functions would frequently overflow the stack, rendering the language mostly useless. Implementing this optimization with SSA is incredibly simple because SSA is a directed graph. Tail-calls inside a function will look like a recursive call transfer case followed immediately by another block with a return transfer. Eliminating the tail-call is as simple as swapping out the recursive call for a `goto` transfer that jumps to the start of the function with the caller's arguments.

The final optimization pass implemented was rather small and it's only purpose was to clean-up after tail-call elimination. This pass removes all blocks in a program that have no parent blocks and unsurprisingly it's called `unused-blocks`.

Conclusion

At the close of this semester I ended short of the (adjusted) course goals. I wished to have the SSA type-checker complete and to fix the goto-shrink optimization pass which still has some unknown bug that causes interpreter errors. Overall I would say that my progress during the first third of this semester was slower than I would have liked, a large portion of time was dumped into designing the SSA structure and getting the ANF to SSA conversion completed. What was initially thought to be a relatively simple conversion pass was instead a fair bit more complex than I initially thought and took substantially longer than anticipated. Looking back at the original course goals I would say they were a bit of a stretch. Spending a measly 5 weeks on SSA was not sufficient for me to implement the IR and all related code.

I learned a lot from this independent study, creating an intermediate representation from scratch was rather challenging. The training wheels were taken off this time around, unlike when I worked on the *LangF* projects in

Compiler Construction. Each project came bundled with a large portion of supporting code and scaffolding, it clearly labeled where students were expected to fill in their code and that was that. Exploration of the entire project source was not necessary to understand and implement each sub-component and all the major design decisions were already made. This independent study was illuminating, it allowed me to better grasp the amount of effort and thought required to construct a compiler from scratch. I also substantially improved my understanding of Standard ML.

Further work on this project would include finishing the type-checker and some of the supporting type substitution code. Introducing a proper directed-graph library would also be a substantial improvement to the code-quality of this project, especially when implementing optimization passes. Having an efficient graph library would enable some duplicated code in the optimization passes to be abstracted out. Successors and predecessor lookups would become constant or linear time operations, which should make goto-shrink and tail-call elimination faster (and easier to implement). The graph library would also make calculating dominator trees for each function possible. Naturally more optimization passes would be added such as: constant propagation, dead code elimination and value numbering.

Appendix A

This appendix discusses Static Single Assignment in more detail and then delves into the design of the new *LangF* SSA IR.

Static Single Assignment

The Static Single Assignment intermediate representation was introduced in the paper "*Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*" in 1991. SSA intermediate representations are used in a wide variety of compilers, typically more for first-order imperative languages like C or C++, however there is a use-case for SSA in functional language compilers. MLton for example has multiple SSA forms, but it requires some extra work to transform an inherently higher-order polymorphic language into a first-order simply typed language.

Now you might be asking yourself, why bother with SSA? What makes it useful? To answer that question, let's first understand what SSA is. The hallmark of an SSA form is that it requires all variables in a program to be dominated by their definition and have exactly one assignment. More simply all uses of a variable in a program must come after it's been defined and it cannot ever be re-defined. This simple requirement enhances the assumptions that can be made when optimizing a program.

There are a number of optimization passes that become trivial or at least much simpler when performed on an SSA representation rather than a tree-like structure such as the abstract syntax tree. For example a typical constant propagation optimization is exceptionally easy under an SSA. Copy propagation finds variable aliases and eliminates extra layers of indirection. For example imagine you have the following variable bindings:

$$
\begin{aligned}
x &= 1 \\
y &= x \\
z &= y
\end{aligned}
$$

It is clear to see that y is an alias to x and by extension z is also an alias to x. Once it has been optimized with copy propagation then the definition of z become simply $z = x$.

SSA forms are typically represented as a control-flow graph made up of basic blocks. A basic block is represented by some unique identifier, typically called a label and a sequence of code without branches. At the end of each block is a branching mechanism, we'll call them transfers. Transfers dictate the edges between blocks.Inherently these control-flow graphs can be cyclic to allow for more natural encodings of looping constructs.

A common functionality of many SSA implementations is the ability to construct what is known as a *dominator tree*. A dominator tree is an encoding for dominance between blocks. We say the a block A *strictly dominates* a block B if in the control flow graph it is impossible to reach B except by passing through A. Furthermore we can say that A *dominates* B if and only if either A *strictly dominates* B or $A = B$. Using these definitions it is possible to walk over the graph and construct a tree representing this relationship. This tree is the dominator tree which can be used to compute a minimal SSA which is better pruned from the outset and will most-likely have fewer blocks total.

In the original paper the authors proposed a construct used in their SSA form which they dubbed the ϕ-function to join possible variable definitions that might have more than one potential dominating definition.

One might ask, how do ϕ-functions work? Well they are magical existences from a higher plane which use their all-knowing powers to determine where a particular definition originates. All joking aside, this detail is really left up to the implementer to decide. For some it might be as simple as enforcing a particular register for a variable definition and ensuring that all possible values are placed in that register before entering the block. However, some very clever people working on the MLton compiler came up with a more elegant solution. Rather than trying to be an oracle at compile time and placing the burden of proof on the definition point it is possible to flip the roles. MLton defined basic blocks to act more like functions that accept arguments. Now it is the "caller's" responsibility to pass in the correct value and then the block will expect it's arguments in a specific order.

SSA IR Design

LangF is different from Standard ML is a number of ways. Firstly it is less restrictive on recursive uses of polymorphic functions. It is possible to recursively call a function with a different polymorphic value. SML does not allow this behavior. Another simple difference is the lack of a module system,

which drastically simplifies the language.

The SSA design used in my project is based heavily on the SSA implementation in MLton with a few distinct differences. MLton sidesteps a lot of design difficulties with SSA by removing the need for ϕ-functions by using basic blocks that accept arguments. I use a very similar design, but *LangF* cannot eschew type information at this stage in the compiler, thus the language is still higher-order and polymorphic. As a result my basic blocks also accept type arguments in addition to value arguments.

Another stark difference from MLton is that my SSA is higher-order. MLton has a wide variety of intermediate representations that uses during compilation. One particular pass, dubbed `ClosureConvert`, performs a very interesting transformation know in MLton as *defunctorization* which remove removes closures and transforms the program to a first-order representation. This transformation is also called *lambda lifting* by some sources. While SSA forms are typically used for first-order languages it is not a requirement. My SSA implementation remains higher-order, which simplifies the definition drastically.

Below is the source for a simplified Standard ML structure describing my implementation of SSA. The `ID` signature provides some nice functionality for equality comparison and unique name generation, but the details are not important in this context. The important bits of this implementation are located in the mutually recursive types `stmt`, `block` and `func`, which represent statements, blocks and functions respectively. These types define the main structure of the directed-graph.

Let's break down each component bit-by-bit. First off a function is composed of the fields `name`, `tyargs`, `args`, `start`, `blocks` and `return`. The first field `name` is a unique name for the function that is used when calling the function. The `tyargs` field is a list of type arguments to the function and directly after that the `args` field is a list of types that denote the expected types for the function arguments. Next is `start` which is the label that uniquely identifies the starting block for the function that ought exist in the `blocks` list for the function. Lastly is the return type `return`. Take note that functions do not care about argument names, it's up to the starting block to ascribe names to the values, but the types ought match. Each function is really just a container for the blocks which hold the real code.

Next up is blocks, they are composed of the fields `label`, `tyargs`, `args`, `stmts` and `transfer`. The field `label` is a unique identifier for the block in the program. The `tyargs` field is a list of type arguments to the block and

the `args` field is a list of argument names and associated types. Next is the `stmts` field which holds a list of statements making up a linear sequence of code. Lastly is `transfer` which describes where the block should go next. For example a transfer could be a simple **goto** that takes a label and moves execution to the block with the given label. Transfers also encode branches using **case** expressions where each arm of the expression is substituted for a **goto** that leads to the actual code.

Lastly let's look at the statement type which has three possible variants. The first is `Stmt` with the record `{var: Var.t, ty: ty, exp: exp}`. The record is encoding a binding from an expression to a variable with an explicit type. It might look something like: `x : int = 0`. The second variant is `DataTys` which holds a list of mutually recursive datatypes. Lastly is `Funcs` which similarly holds a mutually recursive list of functions delcarations.

Expressions are not exceptionally interesting, they hold values like primitive applicatons, integers, strings, variables, etc. The last really interesting type is `prog` which describes an SSA program. While it may look simple, holding only a single main function, it is important to recognize that ANF programs held a list of declarations and a result variable. Where did those go in this IR? To answer simply the main function is explicity wrapping the previously top-level declarations and explicitly returning the final result variable.

```
signature SSA_IR =
sig
  structure DaCon : ID  (* data constructor *)
  structure TyCon : ID  (* type constructor *)
  structure TyVar : ID  (* type variable *)
  structure Var   : ID  (* variable *)
  structure Label : ID  (* label *)

  (* primitive operations *)
  structure Prim :
  sig
    datatype t = ...
  end

  ...

  datatype ty =
    T_Func of {tyvars: TyVar.t list, args: t list, return: t}
  | T_TyCon of {tycon: TyCon.t, tyargs: t list}
  | T_TyVar of {tyvar: TyVar.t}
```

```
type pat = {dacon: Dacon.t, tyargs: ty list}

datatype transfer =
  Case of {test: Var.t,
           cases: {pat: pat, label: Label.t} list}
| Goto of {tyargs: ty list,
           args: Var.t list,
           dest: Label.t}
| Call of {func: Var.t,
           tyargs: ty list,
           args: Var.t list,
           return: Label.t}
| Return of {arg: Var.t}

type dacon = {dacon: DaCon.t, tyargs: ty list}

datatype dataty =
  DataTy of {tycon: TyCon.t,
             tyvars: TyVar.t list,
             dacons: dacon list}

datatype exp =
  E_Integer of IntInf.int
| E_String of String.string
| E_Var of Var.t
| E_DaCon of {dacon: DaCon.t,
              tyargs: ty list,
              args: Var.t list}
| E_PrimApp of {prim: Prim.t,
                tyargs: ty list,
                args: Var.t list}

datatype stmt =
  Stmt of {var: Var.t, ty: ty, exp: exp}
| DataTys of {datatys: dataty list}
| Funcs of {funcs: func list}

and block =
  Block of {label: Label.t,
            tyargs: TyVar.t list,
            args: {var: Var.t, ty: ty} list,
            stmts: stmt list,
            transfer: transfer}
and func =
  Func of {name: Var.t,
           tyargs: TyVar.t list,
           args: ty list,
           start: Label.t,
           blocks: block list,
```

```
              return: ty}

  datatype prog = Prog of {main: func}

  ...
end
```

Appendix B

This appendix includes a list of major tasks completed over the course of this independent study and the approximate number of hours spent.

Hours	**Task**
6	Readings from "*Engineering a Compiler*"
2	Reading "*Flow-directed Closure Conversion for Typed Languages*"
3	Reading MLton SSA source
2	*LangF* CMake build system and general infrastructure enhancements
7	SSA IR design
22	ANF IR to SSA IR converter
8	SSA IR interpreter
5	SSA IR type-checker
4	SSA IR optimizer structure and integration to `langfc` and `langfi`
2	SSA IR goto-shrink optimization pass
1	SSA IR copy propagation optimization pass
0.5	SSA IR unused blocks optimization pass
1	SSA IR tail-call elimination optimization pass
10	Independent study report
73	Total