Using Haskell to Implement a diML-to-LLVM Compiler

Thomas Dietert

Honors Thesis - Summer/Fall 2015
Advisor: Dr. Jarred Ligatti
Committee: Dr. Adriana Iamnitchi

Chapter 1

Introduction

The purpose of this thesis is to provide an introduction to the design and implementation of a compiler for a simple functional programming language, diML. Furthermore, this thesis aims to present what I learned during the process in such a way that the average undergraduate interested in compiler design and implementation could spend considerably less time implementing a language of equal or greater size. The reader is assumed to have little to no working knowledge of compilers, along with little to no familiarity with the Haskell programming language.

This thesis is a culmination of many months of interest in the design and implementation of programming languages, and is supported by the sources cited in the body, as well as class notes from theoretical programming languages and compilers courses I have taken the past year.

1.1 Motivation

The motivation behind the implementation of this compiler results from my interest in learning more about the overlap between Programming Languages theory and its accompanying application. Programming Languages theory focuses on the design and implementation of programming languages through characterization of the static and dynamic behavior of programming languages with the use of theory originating from formal logic, linguistics, and abstract branches of mathematics. The theories present in the study of Programming Languages are elegant, and often seem natural after the notational overhead is overcome; however, analysis of programming languages from a theoretical perspective is essentially useless unless someone somewhere is going to put theory to practice.

The practical application of Programming languages theory lies in the implementation of a *compiler*, a piece of software that translates a program written in a high level programming language into a semantically equivalent program adhering to the syntactic and semantic definition of a low level programming language. Such software is *necessary* for programmers to write programs efficiently because low level program-

ming languages are often much more difficult to reason about. High level languages allow humans to specify what actions they wish the computer processor to perform at a level of abstraction that is comfortable to read and write. Without a compiler implementation for a programming language, the language would only exist conceptually, in the same way that mathematical theorems exist before they are applied to the design and implementation of real world systems. Programs could be written and evaluated in such a language, but only by humans, counteracting the reasons for defining a programming language—to instruct a computer how to operate. This is the distinction between theory and engineering: without theory, the engineering of complex systems would be disorganized and likely impossible, but without the engineering effort to apply theory to the physical world, much of modern technology would not exist. The implementation of a compiler brings a programming language into existence.

After studying programming languages passionately for two semesters, I felt it was necessary to understand the practical application of my studies. During the two semesters a simple language named "diML", which stands for "diminished ML", was defined simply at first, introducing the basics of programming language specification such as the structured operational semantics and syntactic definition and then repetitively expanded so that at the end of two semesters, diML could hardly be referred to as *diminished*. However, the compiler implementation outlined by this thesis respects diML's prefix and only implements a compiler recognizing a subset of the features and constructs defined during those two semesters. In the following sections, the definition of diML's syntax and type-system are presented, along with an overview of what the thesis aims to cover.

1.2 diML Language Specification

The syntax of diML is highly similar to ML, hence the name "diminished ML", though it does have some distinct differences. For isntance, diML (this implementation, at least) only supports integer literals and variables recognized by the regular expression given in Figure 1.1 to preserve triviality to some extent—the aim of this text is not to present an industrial strength compiler. For the purpose of this thesis, the *1st order abstract syntax* for expressions is presented as the specification of the language's syntax.

$$\begin{array}{rl} (variables) & x, f \ \in \ [a\text{–}zA\text{–}Z][a\text{–}zA\text{–}Z0\text{–}9'] \\ (integers) & n \ \in \ \mathbb{Z} \end{array}$$

Figure 1.1: Valid diML literals

The type system for diML is presented alongside all valid expressions within the language since type annotations are allowed on many diML expressions; in the case that an expression is within parentheses, type annotations are allowed on *all* diML

expressions. It is worth noting that the type system displayed in Figure 1.2 represents a naive version of diML's type system, expanded later in Chapter 3, where the implementation of type inference for the language is demonstrated. Two forms of expressions are presented: unannotated expressions **u**, and annotated expressions **e**. It is worth noting that every expression that can be annotated can also be written without a type annotation and is still a valid expression in diML—the programmer is not required to put type annotations on any expressions due to the power of *Hindley Milner Type Inference* discussed in Section 3.3.

$$
\begin{aligned}
\tau &:= Int \mid Bool \mid \tau_1 \ \rightarrow \ \tau_2 \mid \tau_1 \ \times \ \tau_2 \\
u &:= x \mid n \mid true \mid false \mid e_1 \ + \ e_2 \mid e_1 \ - \ e_2 \mid e_1 \ \times \ e_2 \mid e_1 \ / \ e_2 \mid \\
&\quad\ \ if \ e_1 \ then \ e_2 \ else \ e_3 \mid e_1 \ e_2 \mid (e_1, e_2) \mid let \ x = e_1 \ in \ e_2 \mid printInt(e) \\
e &:= \backslash x : \tau \rightarrow e \mid let \ fun \ f(x : \tau_1) : \tau_2 = e_1 \ in \ e_2
\end{aligned}
$$

Figure 1.2: 1st order abstract syntax for diML types and expressions

Several possibly unclear aspects of the abstract syntax denoted by Figure 1.2 must be further explained. Two forms of let expressions are listed, one for declaring variables to be used in the body of the let expression, and another for declaring functions able to be called in the body of the let expression. Also, there is syntactic sugar on let expressions, or a syntactic feature that makes programs deliberately more readable or clear to the programmer, in which variable or function declarations can be chained through the use of a comma. However, this sugar simply gets converted to a nesting of let expressions in which the variable or function declarations are defined:

$$
\begin{aligned}
&let < declaration >_1 \ , \ \ldots , \ < declaration >_n \ \ in \ e \\
&\qquad\qquad \Rightarrow \\
&let < declaration >_1 \ in \\
&\qquad let < declaration >_2 \ \ in \\
&\qquad\qquad \ldots \\
&\qquad\qquad let < declaration >_n \ in \ e
\end{aligned}
$$

Differing in from the usual notation, the *lambda abstraction* (an expression representing an anonymous function) is denoted $\backslash x : \tau \rightarrow e$. The anonymous functions in diML are defined and behave analogous to the lambda abstractions found in any other language. Another expression worth noting is the function application expression $e_1 \ e_2$; since diML is a functional programming language and function application is the crux of functional computation, it makes sense to limit the syntactic noise and to denote function application simply, with a space between the function and its argument. In a more advanced implementation of diML, functions with multiple arguments and, likewise, partial function application would be implemented.

1.3 Overview

The purpose of this thesis is to show that implementing a non-trivial sized functional programming languages such as diML can be accomplished with the effort of one programmer using the Haskell language. There are many languages used in the implementation of compilers, and this thesis and respective compiler implementation aims to express the ease in which such a feat can be accomplished. The complexity of implementing a compiler can be greatly simplified through the used of tools such as Haskell's Parsec library, and the LLVM project (discussed below). Haskell itself has many benefits with respect to implementation, though many are shared with other functional programming languages; as a result, this thesis also argues that compilers are more easily implemented in functional languages.

The body of this thesis is composed of several chapters: Chapter 2 outlines the general structure and design of a modern compiler, whereas Chapter 3 hopes to convey the implementation of the compiler in such a way that the implementation of another simple language similar to diML could be implemented by the reader after reading the thesis in its entirety and consulting the various bodies of information suggested throughout. My bias towards the Haskell programming language, along with my excitement with respect to many of the theoretical concepts may be noticeable at some points. Aside from the inescapable bias, I hope that the myriad of helpful features the Haskell language supports, outlined in several sections throughout this thesis, adequately shows how the complexity of the implementation is reduced in contrast to compiler implementations using other languages. This thesis aims to provide its own argument, and I hope the astute reader is able to discern the objective complexity reduction and underlying beauty of theory without influence of my own interest. In Chapter 4, the evaluation of the outcome of this thesis and compiler implementation is discussed, preceding the conclusion. Reasons for choosing Haskell as the implementation language, along with why this compiler implementation targets the LLVM IR are outlined below.

1.3.1 Haskell

The language in which the diML compiler will be implemented is Haskell. This decision results from several considerations with respect to the implementation of a compiler. Haskell is a strong, statically typed, *pure* functional programming language possessing several features that make the implementation of a compiler for a functional programming language simpler and more elegant in many ways. One of these features is the ease in which *abstract data types* (ADT) can be defined; ADTs make it easy to define recursive data structures that can represent many useful data structures (e.g. Abstract Syntax Trees). These ADTs can be defined with polymorphic fields such that the structure of the ADT is preserved throughout all instances while the underlying type of some or all of the fields of the data type may be different. Haskell code is also more consise, more readable (for those familiar with the syntax), and generally much

shorter than imperative programming languages. This combination yields an implementation of a compiler that is manageable when implemented by one person, and frees the programmer of the headache of defining complex and un-intuitive data structures to represent the different structures necessary during implementation. However, it must be noted that these advantages are present in languages of the ML family, and encompass the usual benefits of programming in a functional programming language.

Perhaps the most interesting reason to use Haskell is its purity—the way in which Haskell separates side-effectful code from non-side-effectful code makes reasoning about and debugging of programs easier. The purity is accomplished through a complex construct originating in category theory: the monad. At a high level, monads are "frozen computations" in which side-effectful computations are contained, and their use results in the types of functions denoting whether or not the respective function has side effects or not. It is common to refer to this idea as "the types guide the programmer", since Haskell programmers spend much more time reasoning about the types of functions and the general type structure of the program compared to programming in other languages. In Haskell, monads are implemented using a typeclass, or a way of categorizing types based on the functions that are defined for them. Types of the monad typeclass must have certain functions defined for them and these functions must adhere to a certain set of behaviors such that the programmer can reason about the behavior of all monadic values in a similar way. It is the general observation of many functional programmers that monads make programs more concise and more understandable, allowing stateful programming within an inherently stateless programming paradigm without compromising purity.

1.3.2 LLVM Backend

LLVM is a compiler tool chain that provides a robust and comprehensive back-end for modern compiler implementations, saving implementers time and effort. Generally, the back-end of a compiler is the most *dirty*, rather, it relies the least on interesting theory and most on engineering prowess. The LLVM project provides a set of tools necessary to implement a full-fledged code-generation module without having to spend much time learning a specific architecture's assembly language; LLVM's main contribution is a statically typed hybrid intermediate representation that is assembly-like in many ways while also resembling C code with functions, global variables, and main function. Once LLVM IR is generated, the full power of the LLVM project can be utilized—a wide variety of architecture dependent and independent optimizations can be performed, and the resulting optimized LLVM IR can be compiled to run on virtually any contemporary processor architecture.

LLVM IR is a RISC style assembly language in static single assignment (SSA) form supporting a typed, infinite register set. SSA form refers to the notion that all variables in the IR must be declared before they are assigned, and each variable is assigned once throughout the entire generated program. LLVM IR provides several functions (such as *phi* nodes, mentioned later) to accomplish SSA elegantly, as code

in SSA form can be more readily optimized by the LLVM tools that translate the IR into machine dependent assembly languages. On average, LLVM IR has a 80% the compile and execution time x86 code generated by GCC [8] and is generally human readable—if not, the LLVM compiler tool chain has several compilation options to produce increasingly readable LLVM IR for easier debugging of the code generation module. Another reason one might choose to target LLVM IR as the backend for a compiler is that the register set is abstracted in such a way that it seems infinite. All store instructions are assigned to a "new register", waiting for compilation from LLVM IR to LLVM bit code during which register numbers are resolved depending on the native machine on which the program is compiled [9].

Haskell LLVM Bindings The LLVM Core library supplies a set of functions and data types necessary for generation of LLVM IR; unfortunately, the library is implemented using C++, for C++ programmers. Luckily, the Haskell community has seen the benefits in using LLVM IR as a target language for a compiler and two packages LLVM-general and LLVM-general-pure have been written that wrap the LLVM Core functionality with a Haskell library; the LLVM-general-pure exporting a data type `Module` such that LLVM module construction can be implemented seamlessly in Haskell. LLVM-general is the *foreign function interface* needed for Haskell to call LLVM Core components from the Codegen.hs module, and LLVM-general-pure is a *functionally pure* wrapper for the constructs available in LLVM-general. By constructing an LLVM module using the constructs exported by LLVM-general-pure, making function calls to the LLVM Core library via LLVM-general preserves type safety and purity because the construction of the code representation in the *pure* LLVM-general-pure module makes it so. Below is the data type used in diML's implementation defining an LLVM module, represented as a stateful computation using the State Monad, using *record syntax* such that the function `unLLVM`, when applied to a value of type `LLVM a`, results in the State Monad representing the LLVM module being constructed.

Thesis Outline The body of this thesis is composed of two chapters; Chapter 2 outlines the high level design and structure of a modern compiler, whereas Chapter 3 presents every phase of the implementation of the diML programming language. Essentially, Chapter 2 introduces what a compiler is and Chapter 3 describes how to build one. Chapter 4 is the conclusion of the thesis, where a short summary of the work is delivered along with an evaluation of potential extensions of the diML language, theoretically interesting features omitted, and several closing remarks.

Chapter 2

Compiler Architecture

A compiler is generally composed of many different phases in which the data received by each phase is either transformed into a representation suited for latter phases, or statically analyzed to validate that the previous transformations or input program was valid. As mentioned earlier, the phases necessary to implement a compiler are the lexer, parser, semantic analyzer, and code generation. These phases can be further broken down into more distinct phases depending on the tools and methods chosen by the implementer of the compiler, and in some cases condensed into fewer. In the implementation of the diML Compiler, the parser and lexer are combined and will be analyzed in more depth later. The phases of compilation are quite distinct, and can be defined independent of each other as long as adjacent phases have shared interfaces. In this sense, a compiler can be viewed as a pipeline of transformations of the original program written in the source language, into a functionally equivalent program in the target language. For readers interested in a more complete overview of compiler fundamentals, please consult *Compiler Construction, Principles and Practice* [11] as it contains much of the information included in this chapter but goes into much more depth .

2.1 Lexical Analysis

Lexical analysis is the first step in building a compiler, and generally describes the act of transforming the input string into a stream of lexical *tokens*. The *tokens* outputted by the lexer are the smallest groupings of characters that have meaning when structuring the intermediate abstract representation of the source program. Often referred to as *tokenizing* the input, the transformation of the source program into a stream of tokens is the first step in compilation and is defined by a fairly intuitive algorithm:

> **Repeat:** Read characters from the input string until a delimiter (in most cases, a whitespace character) or EOF is read.

- If the characters read match a token defined for the language, add the token to the token stream and resume reading input.
- Else, if the characters do not match a valid token, throw an error.

For an example, consider the following statement found in most programming languages and its resulting token stream output from the lexer:

if true then 0 else 1 ⇒
IF TRUE THEN INT(0) ELSE INT(1)

Figure 2.1: Transformation of if statement into token stream

Since tokens are defined to be the smallest elements of the program that have meaning, this statement must be tokenized such that the token represent the exact meaning of the initial program. The syntax does not remain exactly the same, but the resulting token stream very clearly represents all elements of the input program from which it was derived. The values associated with the INT tokens are referred to as *semantic values*; certain tokens must be parameterized by a semantic value because the token can take many forms. If these tokens were not parameterized by a semantic value, either all instances of integers must be enumerated in token form (e.g. ONE, TWO, THREE...), or else there would be no other way to distinguish any integer from any other integer. Another example of a token with a semantic value is the token for identifiers (ID(*variable name*)), in which every occurrence of the ID token in a token stream must be parameterized by the identifier it represents.

2.2 Parsing

When referred to in the context of linguistics, parsing means to break down a sentence into distinguishable and sensible groupings of tokens such that the meaning of the sentence becomes clearer. However, with respect to Computer Science, parsing generally refers to the decomposition of large strings of formatted data into the smallest groups such that each grouping has meaning. It is the first phase in which the compiler attempts to derive meaning from the input program by constructing logical groupings of tokens such that the syntactic specification of the language is adhered to.

When defined in the context of compiler implementation, parsing the input program refers to making a pass over the token stream output from the lexer such that the token stream is transformed into a sequence of nested statements and expression, constructing an *Abstract Syntax Tree* (AST) during traversal by attempting to construct valid groupings of tokens. After a grouping is determined to be the only valid grouping of tokens that the parser can construct (it is certain there will be only one, as mentioned in the next paragraph), a *semantic action* is performed. A semantic action is a piece of code that is executed after the parser successfully recognizes a valid token grouping to add to the AST the parser is constructing. The parser only adds the token grouping to the AST after every sub-expression or nested statement is correctly parsed.

2.2.1 Context Free Grammars

The first step in the parsing phase is to define a *Context Free Grammar* (CFG) that recognizes the language the compiler is implementing. CFGs are widely used to define programming languages because they easily and concisely represent the recursive nature of nested statements and expressions that valid programs are composed of. An example of a CFG for the language recognizing addition expression composed of natural numbers is show below:

$$\begin{aligned} \text{G: } & S \rightarrow S + n \\ & S \rightarrow n \end{aligned}$$

CFGs are composed of a series of *non-terminals* on the left hand side of the arrows, and *terminals* which are terms that do not expand any further (terminals are often the tokens in the language recognized by the lexer). In the addition grammar above S is a non-terminal, whereas $+$ and n are terminals. The recursive structure is apparent in the *production* $S \rightarrow S + n$ (where $S \rightarrow S + n$ and $S \rightarrow n$ are both productions) which states that an expression S can be composed of another expression derivable from S followed by the sequence of terminals $+$ n. Repeatedly expanding non-terminals within the right hand side of a production is called *deriving* an expression within the language; All expressions recognized by this CFG are able to be derived from a sequence of derivations using productions.

There is a caveat, however, that a compiler implementer must take care to avoid when defining the grammar for the programming language being implemented, and that is the grammar must not be *ambiguous*. A CFG is said to be ambiguous if there exists two parse trees for any valid program in the language defined by the CFG. At a high level, the construction of two distinct parse trees that represent the same expression is akin to there existing two sequences of derivations to arrive at the same expression. Succinctly, it is desired that every expression recognized by a CFG be constructed by only a single sequence of derivations. For example, the previous addition grammar could be written in a similar way that is ambiguous:

$$\begin{aligned} \text{G': } & S \rightarrow S + S \\ & S \rightarrow n \end{aligned}$$

With respect to *G'*, the expression $n * n + n$ could be derived just as it could from G, but in this case there could be more than one way to derive it. Since both the S on the left and right side of the $+$ terminal can be decomposed further into $S \rightarrow S + S$, it is unclear which should be expanded first. In the previous grammar, there was a single non-terminal S on the right hand side of the first production, so there was indeed only one way to derive the expression $n * n + n$. Unfortunately, constructing unambiguous grammars is an art in itself—no algorithm exists to transform an ambiguous grammar into a unambiguous grammar.

Due to the complexity of many constructs within non-trivial programming languages, parser tools exist to help the implementer build a CFG that is not ambiguous. For instance, compiler writers don't construct unambiguous grammars themselves but instead provide a partially ambiguous grammar (the ambiguity is usually contained within only a few non-terminals). Common parser generator tools to aid the design of the lexer and parser modules of a compiler are yacc, bison (along with flex), and ANTLR. Each of these tools has a unique interface for the programmer to specify the CFG of the programming language, along with other necessary specifications such as valid tokens and auxiliary code to be executed during the phases such as error reporting. The programmer will generally define the precedence of the operators supported in the language and define the productions representing those expressions ambiguously, letting the parser tool do all of the heavy lifting to construct an unambiguous CFG; However, these tools cannot do all of the work, and with some tools it still takes a clever mind to write an unambiguous grammar. Luckily for diML's implementation, the Parsec library in Haskell is very powerful and does most of the work.

2.2.2 Abstract Syntax Trees

ASTs are the output of the parsing phase of the compiler and likewise, the input to the semantic analyzer, the next phase compilation. The purpose of the parsing phase is to extract meaning from the input program without needing to keep track of extraneous symbols that may aid both the programmer and the parser; The former in defining clear and readable programs, and the latter by making it clearer which tokens should be grouped together as expressions. ASTs are dramatically smaller than their alternative, Concrete Parse Trees (which do not forgo unneeded symbols), and make the program much easier to manipulate in the semantic analysis and code-gen phases.

The reason for constructing unambiguous CFGs when designing a programming language is to avoid the case where a valid program recognized by the CFG is interpreted to have multiple meanings. When a program can be interpreted multiple ways, then multiple ASTs can be constructed to represent the program and the compiler is left uncertain which program the programmer was trying to write. Since one of the requirements of an adequate compiler for a programming language is to produce a semantically equivalent program in the target language, ambiguity proves to not meet this requirement. ASTs can be constructed for the expression $n * n + n$ using both grammars G and G'; Where G will generate a single parse tree, G' will result in two possible parse trees (ignoring operator precedence, as grammar G' does not account for it).

As illustrated in Figure 2.2, grammar G can only produce one AST from parsing the expression $n * n + n$. There is no other way to derive the expression from G and as a result, there is only one way in which the compiler can understand the meaning of the expression: $(n * n) + n$. Likewise, in Figure 2.3 the idea that a single expression can have multiple meanings is quite apparent: Does the compiler evaluate the expression $(n * n) + n$ or $n * (n + n)$, which clearly do not evaluate to the same

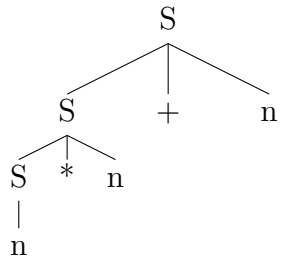

Figure 2.2: AST for $n * n + n$ with grammar G

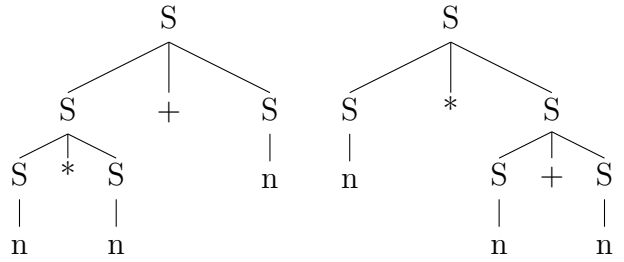

Figure 2.3: ASTs for $n * n + n$ with grammar G'

result. The necessity to define unambiguous grammars when constructing the CFG for a programming language should now be apparent.

2.3 Semantic Analysis

After the parser module constructs an AST, the semantic analysis phase begins. Often referred to as type-checking, this phase will either report whether the input program (now in the form of an AST) is well typed or not. Semantic analysis is often aided by the use of a *symbol table*, a data structure constructed during the type checking process that maps variables to their types in a given context. The symbol table can contain other information about variables declared and used within the source program such as array bounds for array type values, and object super-classes when the language being type checked supports Object-Oriented programming. The type-checking implementation for a compiler can get a bit complicated in languages supporting a wide variety of features, but has a beautiful theory behind it.

2.3.1 Static Semantics

In Programming Languages theory, the static semantics are deductive systems defining how expressions are well typed with the help of a construct called a *judgement form*, which states how each rule of the deductive system will look in structure. The deductive system consists of recursively defined *inference rules*; each rule containing either a

premise or premises above the horizontal line and a conclusion below it, or just a conclusion that is a tautology. The former are referred to as inference rules whereas the latter are known as axioms; if we define n to be any integer, is it a tautology that n has type *Int* in all contexts so this rule is an axiom. In all of the instances of the judgement form $\Gamma \vdash \mathrm{e} : \mathrm{T}$, there is a symbol Γ that represents a variable typing context (symbol table). The main reason for carrying around this context is for the rule *T-Var*, which defines variables to be well typed with type T if within the context Γ variable x has type T. Below are several typing rules that are used to define well typed expressions within diML.

$\boxed{\Gamma \vdash \mathrm{e} : \tau}$

$$\frac{}{\Gamma \vdash \texttt{n} : \mathit{Int}}\ \text{T-Int} \qquad \frac{}{\Gamma \cup \{x : \tau\} \vdash \texttt{x} : \tau}\ \text{T-Var}$$

$$\frac{}{\Gamma \vdash \texttt{True} : \mathit{Bool}}\ \text{T-True} \qquad \frac{}{\Gamma \vdash \texttt{False} : \mathit{Bool}}\ \text{T-False}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Int} \quad \Gamma \vdash e_2 : \mathit{Int}}{\Gamma \vdash e_1 \ \texttt{>}\ e_2 : \mathit{Bool}}\ \text{T-Greater} \qquad \frac{\Gamma \vdash e_1 : \mathit{Int} \quad \Gamma \vdash e_2 : \mathit{Int}}{\Gamma \vdash e_1 \ \texttt{+}\ e_2 : \mathit{Int}}\ \text{T-Add}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \text{T-If}$$

Figure 2.4: Some of diML's static semantics

To give more context to the rules above, rule T-If can be explained as follows: In order for the expression *if* e_1 *then* e_2 *else* e_3 to be well typed in context Γ, the premise e_1 must have type *Bool*, e_2 must have type T, and e_3 must have type T. To clarify further, the reason both e_2 and e_3 must both have type T, is that both of an if-statement's branches must have the same type in order for the program to be well typed. T is an arbitrary type variable that stands in for the actual result type of the two branch expressions and is only used to demonstrate that the two branches must both have the same type. The reason there are no occurrences of T in the other rules like given is that there is more restriction on the types for T-Greater and T-Add; the expressions e_1 and e_2 *must* have type *Int*, as binary arithmetic expressions are only valid if each subexpression is of type *Int*.

These rules provide an intuitive basis for the implementation of the type-checker, and likewise, a good approach to defining and implementing a robust and correct type-checker is to first define the static semantics for all expressions that are valid in the language the compiler is defining. However, in order to fully determine the static semantics' correctness, there are several theorems that require rigorous and exhaustive proofs that are out of the scope of this thesis. Accompanying static semantics, dynamic semantics can be defined for a programming language which illustrate the nature in which expressions are evaluated. The judgment form is $e \rightarrow e'$, showing that all rules must illustrate how some expression e evaluates to *e'*, based on a premise or premises written above the horizontal line, similar to how the inference rules in the

static semantics are defined. For more information on this topic, the curious reader can consult *Types and Programming Languages* by Benjamin Pierce for a more in depth explanation of static and dynamic semantics.

2.4 Type-checking

Type checking is has deep theoretical roots and is predicated by the static semantics of a programming language. To implement a type-checker for a compiler means a few things: deciding upon the types that can exist in the language, whether your language will be *statically* or *dynamically* typed, whether it will be *strongly* or *weakly* typed, and the code to satisfy the type-system decided upon. These decisions along with the implementation of the type-checker itself play a large role in compilation speed, execution speed, complexity of the type-checker, ease of programming, along with many more, less significant subtle features regarding the programming language defined by the compiler. Type checking is essentially the safeguard a programmer has

The implementation of type checking follows naturally from the static semantics defined for a programming language; For every inference rule, the conclusion (the statement below the horizontal line) is well typed if the premises are well typed. Given a valid expression in the language, whether it is well typed or not can be discovered from the fitting the premises of one inference rule to the conclusions of others, each branch terminating when an axiom is matched to a premise. The resulting tree-like structure serves as a proof and a derivation of such a well typed or ill-typed expression (in the case that there are no inference rule conclusions to match a given premise such that the derivation cannot arrive at an axiom).

$$\dfrac{\dfrac{\cdot}{\Gamma \cup \{x : Int\} \vdash \texttt{x} : Int}\ \text{T-Var} \qquad \dfrac{\mathbf{D} \qquad \mathbf{E}}{\Gamma \cup \{x : Int\} \vdash \texttt{2 + 7} : Int}\ \text{T-Int}}{\Gamma \cup \{x : Int\} \vdash \texttt{x > (2 + 7)} : Bool}\ \text{T-Greater}$$

$$\textbf{D:}\ \dfrac{\cdot}{\Gamma \cup \{x : Int\} \vdash \texttt{2} : Int}\ \text{T-Int} \qquad \textbf{E:}\ \dfrac{\cdot}{\Gamma \cup \{x : Int\} \vdash \texttt{7} : Int}\ \text{T-Int}$$

Figure 2.5: A derivation of a well-type expression in diML

The pseudo code for the type checking algorithm can be written intuitively when using the static semantics as a reference; Just as the inference rules recursively determine an expression to be well typed or not, in each case of the type-checker (each case a valid expression) the sub expressions of the current expression are type-checked just as the premises were checked in Figure 2.5. If at any point in the type checking of subexpressions and expression is found to be ill typed, and error should be reported to alert the programmer of the ill-typed expression.

The static semantics are presented for several diML expressions in Figure 2.4 is represented in Figure 2.6 in only 25 lines of code and serves to illustrate the ease in

```
typecheck :: Context -> DimlExpr -> Type
typecheck ctxt e =
   case e of
       IntExpr n -> Int
       TrueExpr  -> Bool
       FalseExpr -> Bool
       VarExpr x -> case lookup x ctxt of
                Just type -> type
                Nothing -> error "Var x not in scope"
       GreaterExpr e1 e2 ->
           t1 = typecheck ctxt e1
           t2 = typecheck ctxt e2
           if t1 == t2 && t1 == Int then Bool
           else error "Greater-Than expr is ill-typed"
       AddExpr e1 e2 ->
           t1 = typecheck ctxt e1
           t2 = typecheck ctxt e2
           if t1 == t2 && t1 == Int then Int
           else error "Add expr is ill-typed"
       IfExpr e1 e2 e3 ->
           case typecheck ctxt e1 of
               Bool -> t2 = typecheck ctxt e2
                       t3 = typecheck ctxt e3
                       if t2 == t3 then t2
                       else error "If branches are ill-typed"
               _ -> error "If condition is ill-typed"
```

Figure 2.6: Haskell-esque pseudo code for type-checking several diML exprs

which the static semantics of a language can be translated to the implementation of the type-checker. The majority of the semantic analysis phase is comprised of the type-checker, and in as a language becomes more complex, the type-system more robust, the type-checker grows equally in complexity. However, having a good theoretical foundation with respect to defining provably correct static semantics for a language can dramatically decrease implementation time and effort.

2.5 Code Generation

The code-generation phase of a compiler can easily be the largest and most difficult phase to implement. A true compiler translates a high level language into a lower level, assembly-like language that supports significantly fewer abstractions. This process in-

volves the deconstruction of many high level abstractions programmers use to simplify code and aid readability; ideally, the parser module produces an AST that has removed some abstraction such that code-generation is easier. In high level languages, the abstractions used by programmers are often referred to as *syntactic sugar*, a term used to describe an abstraction that can be represented by a more primitive yet semantically equivalent language construct. Removing this syntactic sugar at some point between the parsing and code-gen phases greatly simplifies the code-gen phase because it reduces the number of forms an expression for which code must be generated can take. Code generation is quite tedious and requires an understanding of the target assembly language positively correlated with the complexity of the source language being compiled; the more complex the source language, the more difficult the implementation of the code-gen phase will be.

2.5.1 Assembly Languages

Assembly languages usually consist of few abstractions and instead explicit op-codes (instructions) that operate on values residing in CPU registers, moving values to and from memory locations to registers or vice-versa, and branching or jumping to different lines of the code. To translate the AST produced by the parser into assembly code requires a comprehensive understanding of what the expressions that make up the AST represent in terms of CPU operations; for example, a variable declaration is, at a low level, a store instruction in which the CPU names a certain location of memory the variable name being assigned and subsequently stores the value the variable is being declared as into that memory location.

Choosing the target language for a compiler depends greatly on the type of machine the programs written in the source language will be executed on. For a language to be executable on most modern computers, a suitable target language would be x86 as a majority of personal computers have processors using x86 architecture; if the code-gen module outputs x86 code, the resulting assembly program will be executable on any machine using an x86 architecture. Another example of a suitable and portable target language is the LLVM Intermediate Representation (LLVM IR). LLVM IR is an assembly-like intermediate representation of code that can target almost every modern architecture due to the large number of code-gen implementations that translate LLVM IR into various assembly languages. The LLVM IR project tries to fix the notion that for every architecture a compiler designer would like to target, a distinct code-gen phase would have to be implemented that produces the respective assembly language for that architecture. Instead, the LLVM project maintainers have implemented many code-gen modules that translate the LLVM IR into the assembly languages of virtually every contemporary computer architecture. For these reasons, the diML code-gen phase targets the LLVM IR and will be discussed in greater detail in Chapter 3.

Chapter 3

Implementation

This chapter provides a comprehensive overview of the implementation of the diML Compiler. This compiler is implemented using the Haskell programming language, arguably one of the best languages for the job (along with SML and related languages), as strongly and statically typed languages lend themselves well to capturing the structure of CFGs and ASTs in a way that dramatically reduces the amount of code necessary for implementation.

Compiler implementations can have between four and six distinct phases depending on the tools the implementer chooses to use, but most consist of at least four: lexing, parsing, type-checking, and code generation. However, the implementation of the diML compiler makes several changes to the usual compiler structure: the lexer and parsing phases are combined into one phase resulting from the use of a powerful and expressive library called Parsec. Also, instead of type-checking programs based on explicit type-annotations required of the user to define when declaring functions, a type-inference module is implemented using the Hindley-Milner type inference algorithm **CITE!!**. Using this algorithm, type-checking as discussed in Chapter 2 of this thesis is subsumed, and the implementation of this algorithm will be discussed in depth in Section 2 of this chapter. The code generation phase of diML is fairly standard, but targets the LLVM IR such that the code generated by this compiler can be executed on most modern computer systems. The implementation of the diML code-gen phase differs from the norm by targeting LLVM IR, as LLVM is a fairly new technology in the scope compiler implementation.

3.1 Monads

Normally, a discussion about monads might not be included when presenting the implementation of a compiler, but since a majority of the code that comprises the diML compiler is monadic, it seems necessary to introduce the concept at a high level. In essence, monads are a way to encapsulate or separate code that has side-effects from code without side-effects; this distinction is generalized by referring to the former as

impure code and the latter as *pure* code. Since functional programs are essentially a series of function applications to the return values of previously applied functions (function composition), functions are the basic building block that the adjectives *pure* and *impure* modify. Essentially, pure functions always return the same result when applied to the same argument/s, whereas impure functions are not guaranteed to always return the same result.

In Haskell, the type signatures of functions denote whether the function is pure or impure; functions with monadic type signatures are impure. A simple example of this distinction can be illustrated with the function `getLine` that lives inside the IO monad, defined in Haskell's Prelude (similar to C's stdlib). The type signature of `getLine` is `IO String`, conveying that whenever `getLine` is called, the return value is not guaranteed to be the same—the function is *impure* (denoted by `IO` in the type signature). In Figure 3.1, the bind operator `>>=` can be used to chain together monadic computations, first executing the monad on the left, and then applying the function on the right of the operator to the result (to the reader familiar with UNIX systems, this behavior is similar to the pipe "|" operator). In Haskell, the type signature of `main` is required to be `IO ()`, which describes its behavior as a function that can perform input and output, but requires the last expression in the function to result in a value of type `IO ()`. Here, the function `putStrLn` has type `String -> IO ()`, taking a `String` as an argument and performing the print computation, returning the value unit inhabiting the IO monad.

```
exclaim :: String -> String
exclaim s = s ++ "!"

main :: IO ()
main = getLine >>= (\input -> putStrLn $ exclaim input)
```

Figure 3.1: A Haskell program that appends "!" to a string read from stdin.

The `exclaim` function in Figure 3.1, however, is a pure function, as can be derived from the type signature: it takes a `String` as an input and returns a `String` as a result. Pure functions that do not manipulate state or have side-effects can still be used inside of a monadic function, but not the other way around; a pure function cannot use a monadic (impure) function in the body without also adopting the monad in the type signature. Using this distinction, side-effectful functions are explicitly annotated as such by their type signatures. Providing a clear distinction between which functions perform side-effects and which don't can greatly ease the burden on programmers during the debugging phase; the only places in which your program can manipulate state are clearly defined by type signatures.

Polymorphism Before further explanation of monads, however, there is an idea that is fairly integral to the use of monads within the diML compiler implementation. In a

functional programming medium like Haskell, polymorphism refers to the notion that functions can be defined to operate on varying data types without having to define such a function for every data type. Take, for instance, the `length` function in Haskell; its type signature is `[a] -> Int`, where `a` is a *type-variable* denoting that the length function can take a list of any type. Many monads defined in the Parsec library as well as elsewhere in this implementation have type signatures of the form *`Monad`* `a`, where `a` denotes the type that parameterizes the monad. It can be understood at a high level as the monad *wrapping* a value of any type in a context; the type of `getLine` in the example above shows exactly that: `IO String` says that the resulting type of the `getLine` function is of type `String`, but that the value resulted from a computation within the IO context.

In addition to denoting impurity and programatically separating side-effectful code from side-effectless code, monads also provide a means in which to increase a program's modularity and compositionality. Every data type that is made an instance of the monad typeclass must have several functions defined for it, the most important of being the `bind` and `return` functions:

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
(>>) :: (Monad m) => m a -> m b -> m b
return :: (Monad m) => a -> m a
```

The `return` function *lifts* a value into the respective monad, where `return 5` within a function in the IO Monad will result in the value `5` with type `IO Int`. The `>>=` `(bind)` function is used to compose monadic functions together. Many of the modules composing the diML compiler use these functions extensively so that aspects of the diML compiler implementation which are inherently stateful can still be implemented with elegance. The `>>` operator is similar to `>>=` except that when the first monadic argument is executed, the result is thrown away, as opposed to being passed as an argument to the second monadic computation (refer to the type signatures). Below is an example of chaining monadic computations with the `bind (>>=)` operator in which the program prints the string input by the user followed by a "?!":

```
ask :: String -> IO String
ask s = return (s ++ "?")

exclaim' :: String -> IO String
exclaim' s = return (exclaim s)

main :: IO ()
main = readLine >>= ask >>= exclaim' >>= putStrLn
```

Some of the more complex monadic data-types used in the implementation of this compiler make use of the State Monad. The State Monad is a monad defined in Haskell's **Control.Monad** package and exports several functions that make it easy to

modify the state of a persistent, stateful data structure, like symbol tables. Often, these monads are combined with a construct called a *monad transformer*, discussed in the subsequent section.Having a construct that combines monads can greatly simplify code as functions can interact with several stateful computations at a time. However, a full understanding of the implementation of monads and monad transformers is unnecessary to understand the implementation of the diML compiler at a high level. To summarize, monadic code provides a way to compose inherently stateful computations in within a programming language that is inherently stateless.

Monad Transformers Monad transformers are an advanced topic even within the Haskell community, and an overview of their complete definition is outside the scope of this thesis. The extent of knowledge of monad transformers necessary to understand the implementation of several phases of the diML compiler is that monad transformers simply wrap several monads together into a single data type in which all monadic functions related to each individual monad are callable inside functions of the monad transformer data type in which they are wrapped. In the type inference section of this thesis, a Reader Writer State monad transformer is used such that the TypeInfer.hs module has access to a symbol table (via the Reader monad), a persistent, stateful set of constraints able to be appended to (via the Writer monad), and a stateful functions generating new type variable names (via the State monad). If the reader is interested more in how these complex yet useful constructs are implemented and how to write your own, a comprehensive overview of monad transformers is available the FPComplete website.

3.2 Parsec - Parsing and Lexing

Parsec is a robust and efficient parser combinator library in Haskell that has a monadic interface along with many well crafted functions that make composing the lexer and parser modules of a compiler fairly simple. Like most Haskell code, the Parsec library is built with the idea of compositionality and reusability in mind; Parser combinators are defined as *polymorphic* monadic functions (section 3.1) that are parameterized by the type of data that the parser combinators should output after a successful parse. The monadic parser type, exported by the Parsec package, is of the form '`Parser a`', where `a` denotes the data-type that running the parser will result in, usually the expression type the AST will take the form of. In the implementation of the diML compiler, the type of parser combinators that parse valid diML expression have type `Parser DimlExpr`, since `DimlExpr` is the ADT representing nodes in the diML AST (Figure 3.2). One of the reasons the '`Parser a`' type is a monad is that monadic types can be constructed in such a way that failure is handled gracefully, returning an value of a type representing a failed computation instead of throwing an exception; under the hood, the '`Parser a`' has a failure type associated with it, a type that is returned when that parser combinator fails: `ParseError`. On an unsuccessful parse, computations of type

'`Parser a`' result in value of type `ParseError`, pointing to what the parser expected as the next character in a sequence, and what character it found in place of the one/s expected.

```
data DimlExpr
    = Lit DLit
    | Var Name
    | BinOp Name DimlExpr DimlExpr
    | Lam Name Annot DimlExpr
    | Fun Name Name Annot Annot DimlExpr
    | If DimlExpr DimlExpr DimlExpr
    | Apply DimlExpr DimlExpr
    | Decl Name DimlExpr
    | Let DimlExpr DimlExpr
    | Tuple DimlExpr DimlExpr Annot
    | Parens DimlExpr Annot
    | PrintInt DimlExpr
```

Figure 3.2: Haskell ADT representing diML syntax

The parsing method that the Parsec package uses is LL(k) where n is an arbitrary number of token *look-ahead*. LL(k), also known as predictive parsing or top down parsing, constructs the AST by means of the leftmost derivation. Other parsing methods construct the AST from the bottom up and are known as LR(n) or bottom-up parsing, where the AST is built from the rightmost derivation. Practical parsers often involve some amount of *look-ahead*, meaning that parsing decisions (deciding which tokens to group together) is determined by which tokens come after the current token being processed. The parser module for diML is implemented as a combination of individual parser combinators defined to parse each individual expression form, composed with the functions `(<|>)` and `try`. Together, these functions (discussed in 3.2.2) are the reason that the DimlExpr ADT (Figure 3.2) can be interpreted as an ambiguous CFG that, with the help of Parsec, produces unambiguous ASTs. One of the key ideas illustrated in this section is that although the Parsec library is quite complex, to use the library effectively does not require a deep understanding of the implementation. Only a trivial understanding of Haskell, monads, and a few examples should be enough for those who wish to implement a parser in Haskell to accomplish what they desire in a short time with fewer lines of code than in most other languages. For the curious reader wanting to know more about how Parsec is implemented and how to use it in more detail, consult *Parsec, a Fast Combinator Parser* [10].

3.2.1 Lexer

The lexer module of diML makes use of the `Text.Parsec.Token` package, which contains the simple yet powerful function `makeTokenParser`. This function makes it straightforward to define a several key features of the language: reserved keywords, reserved operators, and the form of identifiers. The parser combinators `letter`, `alphaNum`, and `char` (referenced in Figure 3.3) operate in a fairly intuitive way, parsing any upper or lower case letter, alphanumeric characters, or whatever character supplied as an argument, respectively. Keywords and operators are defined by the implementer as a list of strings, each string in the list becoming either a reserved word or reserved operator in the language.

```
lexer :: Token.TokenParser ()
lexer = Token.makeTokenParser style
    where style = emptyDef {
                    Token.identStart = letter
                  , Token.identLetter = alphaNum <|> char '_'
                  , Token.reservedNames = keyWords
                  , Token.reservedOpNames = ops
                  , Token.commentStart = "(*"
                  , Token.commentEnd = "*)"
                }
```

Figure 3.3: Definition of lexer for diML

The `lexer` is a value of type `TokenParser` that represents the definition of all valid tokens in the language being defined can be used to build parser combinators that recognize valid tokens in diML; in conjunction with functions from the `Text.Parsec.Token` package that take a lexer (value of type `TokenParser`, the language definiton) as an argument, the implementer can create tokenizing functions that recognize the tokens specified in the `lexer` definition distinct to diML. The functions found in the `Text.Parsec.Token` module help the implementer construct parser combinators that recognize keywords, reserved operators, and literals specified in the language definition used to construct the `lexer` TokenParser value (Figure 3.3). By defining the token parser combinators in this way, these token parsers fit seamlessly into the diML expression parser combinators defined in the Parser.hs module as they result in values inhabiting the `Parser a` monad.

The `lexer` defined in Figure 3.3 and the token parser combinators defined in Figure 3.4 (excluding several token parsers defined in a similar manner that recognize other valid token in diML) encompass everything necessary to construct a complete lexer that recognizes and parses all valid tokens defined for diML. The following section on constructing the parser for diML shows how easily these tokenizing parsers fit into

```
integer :: Parser Integer
integer = Token.integer lexer

identifier :: Parser String
identifier = Token.identifier lexer

reservedOp :: String -> Parser ()
reservedOp = Token.reservedOp lexer

whitespace :: Parser ()
whitespace = Token.whiteSpace lexer
```

Figure 3.4: Examples of token parsing parser combinators in Lexer.hs

the complete parser definition and just how powerful the Parsec library is once a fairly rudimentary understanding of Haskell syntax and monads is achieved.

3.2.2 Parser

The expression parser combinators for diML are defined in the Parser.hs module and use several non-trivial aspects of the Parsec library that simplify the implementation of the parser module. The parser module for diML is structured by defining parser combinators for every valid expression diML expression defined by the DimlExpr ADT in Figure 3.2. The `DimlExpr` data-type represents every possible return value a parser combinator defined in the diML parser module will have, and likewise, a parser combinator is defined for every form a DimlExpr can take; a literal, variable, binary operations, lambda expression, etc. An example of a parser combinator defined for diML can be found in Figure 3.5. Please note the `do` operator and lack of `>>=` operators in this figure. "Do" notation supplants the need to chain together monads with the `>>=` operator, using the `<-` operator to "bind" results of monadic functions to variables equivalent to how the `input` argument of the lambda expression in Figure 3.1 captured the result of `getLine`. On the lines where no `<-` operator assigns the result of the computation to a variable, the function is executed and the result is thrown away (equivalent to the `>>` operator). Also note that `expr` is the principal parser combinator recognizing all valid diML expression, discussed later in this section.

As previously mentioned, Parsec provides many functions that aid in parsing ambiguously defined grammar such as diML's. The most important functions that provide the functionality of combining parser combinators easily and effectively without conflicts are the `choice` and `try` functions, where `choice` is a binary operator of the form `(<|>)` (in Haskell, all operators are equivalent to functions and can be written in prefix notation with parentheses surrounding them, or infix without parentheses).

To understand how `choice` works, first consult the type signature (Figure 3.6);

```
ifExpr :: Parser DimlExpr
ifExpr = do
    reserved "if"
    e1 <- expr
    reserved "then"
    e2 <- expr
    reserved "else"
    e3 <- expr
    return (If e1 e2 e3)
```

Figure 3.5: The parser combinator for diML `if then else` expressions.

```
(<|>) :: Parser a -> Parser a -> Parser a
try :: Parser a -> Parser a
```

Figure 3.6: Type signatures of choice (`<|>`) and `try`

the type signature says that `(<|>)` is a function that takes two parser combinators as arguments, and returns a parser combinator as a result. From the name and the type signature, the behavior of the function does not come as a surprise—the `choice` function tries to apply the first parser and if at any point the first parser fails, `choice` will apply the second parser. It should be noted that if both fail, the function returns a value of type `ParseError`, the error type encapsulated in the '`Parser a`' monad. This function provides the glue necessary to compose all individual parser combinators defined for all valid diML expressions; however, there is one caveat: if the first parser fails it may have still consumed input, and the second parser will begin attempting to parse the input from where the first parser left off. This is a problem because the tokens consumed by the first parser are important to the meaning of the program and without considering them, the second parser may also fail to parse the input even though the input string represents a valid program.

One solution to this problem would be to define the parser combinators in such a way that no parser combinators for diML expressions share any token prefixes. This is difficult to do, however, and a better option would be to use the conveniently defined `try` function. The `try` function takes a parser combinator as an argument and returns a parser combinator as a result. The result of `try` applied to a parser combinator is one of two values: a parser combinator that has succeeded and is parameterized by the `DimlExpr` successfully parsed (with respect to the diML compiler implementation) or a `ParseError` identical to that of a `ParseError` resulting from failed parser combinators *not* prefixed by `try`, except that a backtrack is performed and no tokens from the input string have been consumed. Effectively, the use of `try` in conjunction with `(<|>)` results in the resolution of token overlap when combining parser combinators that share several tokens as a prefix. The `try` function provides *infinite lookahead* for the parser

```
factor :: Parser DimlExpr
factor =  try applyExpr
      <|> try declExpr
      <|> funExpr
      <|> lamExpr
      <|> boolExpr
      <|> ifExpr
      <|> letExpr
      <|> intExpr
      <|> prIntExpr
      <|> varExpr
      <|> try tupleExpr
      <|> parensExpr
```

Figure 3.7: The expression parser combinator for diML expressions.

combinator it is applied to, and as such should be used sparingly—infinite lookahead is computationally inefficient and care should be taken to define parser combinators that do not overlap in more places than necessary.

A balance between carefully crafted parser combinators and the use of `try` is essential to maintaining the speed the Parsec library delivers. The use of these functions in constructing a parser combinator that parses all valid diML expressions aside from binary and unary operator expressions is illustrated in Figure 3.7.

Along with `try` and `(<|>)`, there are a few other functions needed to complete the parser module for diML, and likewise, almost any programming language. The first is `buildExpressionParser`, which takes an operator table as the first argument, a parser combinator recognizing all non-operator expressions within the language as the second, and returns a parser combinator that recognizes the entirety of the language being implemented. In parsec, an operator table is defined as a list of lists in which the order of the sub-lists define precedence in decreasing order and each sub-list is a list of operator definitions of the same precedence. This greatly simplifies the usual difficulty in constructing a grammar that obeys all operator precedence rules desired, as the implementer need not make any changes to the ambiguous grammar that has been used for the entirety of the parser module so far. Applying `buildExpressionParser` to the operator table and parser combinator recognizing all non-operator expressions in diML results in the *principal parser combinator*, a parser combinator recognizing every possible valid expression within the language being defined, that is, as long as every parser combinator comprising the principal parser combinator implemented correctly.

The last function necessary to complete the parser definition for diML is `parse` (used within the function `parseExpr` on the last line of Figure 3.8). The `parse` function does most of the heavy lifting—it takes the principal parser combinator, a label naming the input string that is being parsed, and the input string as arguments, re-

```
opTable :: OperatorTable
opTable = [ [ binary "*" Expr.AssocLeft
            , binary "/" Expr.AssocLeft]
          , [ binary "+" Expr.AssocLeft
            , binary "-" Expr.AssocLeft ]
          , [ binary "<" Expr.AssocLeft
            , binary ">" Expr.AssocLeft
            , binary "==" Expr.AssocLeft ]
          ]

-- principal parser combinator
expr :: Parser DimlExpr
expr = buildExpressionParser opTable factor

parseExpr :: String -> Either ParseError DimlExpr
parseExpr str = parse expr "<stdin>" str
```

Figure 3.8: diML's operator table and principal parser combinator.

turning either a ParseError or a DimlExpr. If the parse results in an error, an error message is constructed from within the '`Parser a`' monad and should be printed to the screen alerting the user of an unsuccessful parse, and which expression is syntactically invalid. In the case of a successful parse resulting in a DimlExpr (an AST semantically equivalent to the input program), the DimlExpr should be passed as an argument to the next phase of the compiler—the type-checker, or, with respect to diML, the type inference module.

3.3 Type Inference

Related to type-checking outlined in a previous section (2.4), type inference is a compilation phase in which the types of expressions are inferred through a sequence of type-variable constraint generation, construction of type substitutions, and type-unification; furthermore, type-annotations on expressions that were once required of the user, e.g. function definitions, are now optional. Type inference, more suitably known as type reconstruction, is accomplished by the implementation of the Hindley-Milner type inference algorithm, created by Luis Damas and Robin Milner in their famous paper *Principal type-schemes for functional programs* that appeared in ACM's SIGPLAN Journal in 1982 [3]. However, the improperly titled *Hindley-Milner* type inference algorithm should be called the *Damas*-Milner type inference algorithm, as the misconceived *Hindley-Milner* title originates from the Hindley-Milner polymorphic type system for functional programming languages introduced in 1978 in the paper *A theory*

of type polymorphism in programming [12], providing the basis for the Damas-Milner type inference algorithm discovered a few years later. A comprehensive overview of the Damas-Milner type inference algorithm is outside the scope of the thesis, but since its implementation subsumes that of the type-checking module, a high level explanation of the implementation will be discussed in the sections below. The discussion will leave out many details, and the deductive systems describing various aspects of the type system and type inference algorithm will be omitted aside from a few examples for only experienced programming languages students would benefit.

To implement the type inference algorithm, a change to the type system must be made: The introduction of a typing structure known as a *type scheme* represented by σ, where a type-scheme is a closure of a set of type variables over a type τ.

$$\begin{aligned} \tau &:= \alpha \mid Int \mid Bool \mid \tau \rightarrow \tau' \mid \tau \times \tau' \\ \sigma &:= \forall \overrightarrow{\alpha}.\tau \end{aligned}$$

This new type systems differs only slightly from the original diML type system's definition, adding type variables α and type schemes, σ. With these two additions along with an alteration of the original `DimlExpr` ADT and parser module to recognize annotated or unannotated expressions, all the constructs necessary for the implementation of type inference for diML exist. Several variations of the Damas-Milner type inference algorithm exist; this thesis loosely follows the algorithm outlined in [7] with some of the implementation details originating in Stephen Diehl's tutorial on how to implement a small subset of the Haskell language [4]. The following sections explain the algorithm at a high level; more interested readers who wish to implement this algorithm for a compiler written in Haskell should consult Stephen Diehl's tutorial.

3.3.1 Implementation Details

Generalization To generalize a type τ with respect to a typing environment Γ entails the quantification of type variables that are free in τ but do not occur in Γ, i.e.:

$$\begin{aligned} generalize(\Gamma, \tau) &= \forall \overrightarrow{\alpha}.\tau \\ where\ \overrightarrow{\alpha} &= FV(\tau) - FV(\Gamma) \end{aligned}$$

To generalize a type with respect to a given typing context is isomorphic to converting a nested function into a closure as described in Section 3.4. Generalizing is important when adding a function type to a typing environment Γ after inferring it's type; generalization closes the function type with the types of otherwise free type variables in the inferred type of the function.

Instantiation To *instantiate* a type scheme $\forall \overrightarrow{\alpha}.\tau$ means to replace every quantified type variable in $\overrightarrow{\alpha}$ by a fresh type variable with respect to the typing context Γ (where

β_n/α_n is read as "substitute type variable β_n for type variable α_n in type τ"):

$$instantiate(\forall\alpha_1, \ldots \alpha_n) \ = [\beta_1/\alpha_1, \ldots \beta_n/\alpha_n] \ \tau$$
$$where \ \beta_1, \ldots \beta_n \ are \ fresh$$

Instantiation is necessary when looking up the type of a variable in a context that may be different from the context in which the type scheme of the variable was inferred. Instantiations of types yields a type that is isomorphic to the original type, equivalent structurally but not nominally.

RWST Monad For the implementation of diML's type inference module, a Reader-Writer-State monad transformer is used. At a high level, the RWST monad gives access to an instance of each respective monad: The reader monad helps carry around a typing context, the writer monad keeps track of constraints as they are generated (using function `tell`), and the state monad gives access stateful counter that can be incremented each time a fresh variable is generated—the counter is used to index an infinite list of unique variables names. The monad `Infer a` used in the TypeInfer.hs is defined below.

```
type Infer a = (RWST
                    TypeEnv             -- Typing environment
                    [Constraint]        -- Generated constraints
                    InferState          -- Inference state
                    (Except TypeError)  -- Inference errors
                a)                      -- Result
```

3.3.2 Constraint Generation

The Damas-Milner type inference algorithm begins with generating a set of typing *constraints*, requiring recursive traversal of the AST similar by nature to the type checking algorithm presented in Chapter 2 Section 4. A constraint is a relationship between two type schemes (primitive types are type schemes closed by an empty type environment), and is of the form $\tau \times \tau'$, a pair of types that must be *unified* in a later phase of the algorithm—unification is discussed in Section 3.3.3.

Generating constraints is perhaps the most distinct part of the type inference algorithm and to stay true to theory [7] without taking any shortcuts, the substitution and unification phases of the algorithm occur once all constraints for an entire program are generated. As such, constraint generation can be conveniently represented with a judgement form that encapsulates the recursive algorithm, similar to how diML's static semantics were presented earlier in Chapter 2. As in Figure 2.4, Figure 3.9 depicts the constraint generation deductive system containing only several of the inference rules for expressions in diML. Since both annotated and unannotated expressions are possible in diML, Figure 3.9 depicts only the constraint generation judgement form and deductive system for unannotated expressions.

$$\boxed{\Gamma \vdash u \rightsquigarrow \mathrm{e} : \tau \mid \mathrm{C}}$$

$$\frac{}{\Gamma \vdash \texttt{n} \rightsquigarrow \texttt{n} : Int \mid \emptyset}\ \text{C-Int} \qquad \frac{}{\Gamma \vdash \texttt{True} \rightsquigarrow \texttt{True} : Bool \mid \emptyset}\ \text{C-True}$$

$$\frac{}{\Gamma \cup \{\texttt{x} : \tau\} \vdash \texttt{x} \rightsquigarrow \texttt{x} : \tau \mid \emptyset}\ \text{C-Var}$$

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_1 \mid C_1 \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2 \mid C_2}{\Gamma \vdash u_1 \ \texttt{>}\ u_2 \rightsquigarrow e_1 \ \texttt{>}\ e_2 : \tau_3 \mid C_1 \cup C_2 \cup \{\tau_1 = Int, \tau_2 = Int, \tau_3 = Bool\}}\ \text{C-GT}$$

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_1 \mid C_1 \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2 \mid C_2}{\Gamma \vdash u_1 \ \texttt{+}\ u_2 \rightsquigarrow e_1 \ \texttt{+}\ e_2 : \tau_3 \mid C_1 \cup C_2 \cup \{\tau_1 = Int, \tau_2 = Int, \tau_3 = Int\}}\ \text{C-Add}$$

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_1 \mid C_1 \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2 \mid C_2 \qquad (\texttt{fresh}\ \alpha)}{\Gamma \vdash u_1\ u_2 \rightsquigarrow (e_1\ e_2) : \tau_3 \mid C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to \alpha\}}\ \text{C-Apply}$$

Figure 3.9: Several Inference rules of the constraint generation judgement form

From these inference rules, the algorithm can be easily implemented using Haskell using the `Infer a` monad transformer, as it encapsulates every parameter necessary for the algorithm. The implementation of the `infer` function depicted in Figure 3.10 follows naturally, where `fresh` generates a fresh type variable, `addConstr` adds a constraint to the list of constraints generated by the algorithm, and `return` lifts the type variable back into the `Infer` monad, such that the result type of `infer` is `Infer Type`.

3.3.3 Unification

Unification refers to the process of resolving constraints produced by the constraint generation algorithm by yielding substitutions that *unify* two types such that when the substitution applied to each type, the types are equivalent. A *substitution* is represented in the type inference module as a mapping of type variables to types, which may also contain type variables. If no such substitution exists and the types can't be unified, then unification fails within the `Infer` monad transformer with a `TypeError`. Reading $\sim$ as "unifies", the implication below states that if τ and τ' are unified by substitution S, then substitution S applied to τ is equivalent to S applied to τ'.

$$\tau \sim \tau' : S \quad \Longrightarrow \quad [S]\tau = [S]\tau'$$

In the implementation of the unification algorithm for diML, two functions are used: The `solve` function, which unifies a single constraint, composes the resulting substitution with a *principal substitution* (the substitution that will be returned as

```
addConstr :: Type -> Type -> Infer ()
addConstr t1 t2 = tell [(t1, t2)]

infer :: DimlExpr -> Infer Type
infer expr =
    case expr of
        Lit (DInt _) -> return tInt
        Lit (DBool _) -> return tBool
        Var x -> lookupVarType x
        BinOp ">" e1 e2 ->
            t1 <- infer e1
            t2 <- infer e2
            tv <- fresh
            let infType = TArr t1 (TArr t2 tv)
                opType  = TArr tInt (TArr tInt tBool)
            addConstr infType opType
            return tv
        BinOp "+" e1 e2 ->
            t1 <- infer e1
            t2 <- infer e2
            tv <- fresh
            let infType = TArr t1 (TArr t2 tv)
                opType  = TArr tInt (TArr tInt tInt)
            addConstr infType opType
            return tv
        Apply e1 e2 -> do
            t1 <- infer e1
            t2 <- infer e2
            tv <- fresh
            addConstr t1 (TArr t2 tv)
            return tv
```

Figure 3.10: The constraint generation algorithm cases implied by the constraint generation judgement form in Figure 3.9

the result of the `solve` function when there are no more constraints to unify), and then applies the current substitution over the remaining constraints; and function `unify`, which takes a constraint as an argument and attempts to unify the two types that comprise the constraint. As usual, the unification algorithm can be concisely summarized by a deductive system, found in Figure 3.11. The **occurs check** premise of rule `U-TVar` states that type variable α must not appear in the free type variables of τ, because when substituting τ for α in both the principal substitution and the set of remaining constraints, such a substitution would result in non-termination; as τ is substituted for α, one if not more α variables would appear in the types in which τ is

being substituted for α such that the algorithm enters an infinite loop.

$$\boxed{S \mid C \mapsto S' \mid C'}$$

$$\frac{}{S \mid C \cup \{\tau \ = \ \tau\} \mapsto S \mid C} \text{ U-Eq}$$

$$\frac{}{S \mid C \cup \{\tau_1 \to \tau_2 = \tau_3 \to \tau_4\} \mapsto S \mid C \cup \{\tau_1 = \tau_3, \tau_2 = \tau_4\}} \text{ U-Fun}$$

$$\frac{\alpha \notin FV(\tau) \quad (\textit{occurs check})}{S \mid C \cup \{\alpha = \tau\} \mapsto ([\tau/\alpha]S) \cup \{\alpha = \tau\} \mid [\tau/\alpha]C} \text{ U-TVar}$$

Figure 3.11: Several Inference rules of the constraint generation judgement form

Once the principal substitution is found, it is applied to the resulting type of the `infer` function. This reinforces the choice to use the convenient RWST monad transformer: running the `infer` function on the `DimlExpr` AST results in a monadic computation in which the set of constraints and principal type constructed by the algorithm can be extracted. For sake of brevity, the code used in the implementation of the `unify` and `solve` functions, as well as the monadic helper functions that provide the interface between the type inference module and the parser and codegen modules is omitted, and can be found at `https://github.com/tdietert/dimlCompiler/blob/master/src/TypeInfer.hs` starting around line 260 and ending at the end of the file. Now that the principal type is found and simplified from application of the principal substitution, the AST representing the input program can be transformed such that the codegen phase of the compiler is easier to implement.

3.4 diML IR

An interesting realization occurs when attempting to generate LLVM IR from the `DimlExpr` AST constructed in the parsing phase of the compiler; diML is a functional programming language, yet LLVM IR is not. This means that LLVM IR does not support the nesting of functions in let expressions nor function bodys as diML does. In diML, it is possible to nest functions as deeply as the programmer desires but in LLVM IR, all functions must be defined top-level, as their own module. An LLVM IR file (denoted by a .ll file extension) is composed of a set of modules that define globally scoped functions and variables. Since this code structure is not supported by LLVM IR, it is complicated to provide a concise and readable function that generates LLVM IR from the `DimlExpr` IR unless a transformation of the `DimlExpr` AST is performed. Since most assembly languages do not allow nested functions, this AST transformation has been well studied and is known as *lambda lifting*, or *closure conversion*. Fortunately, this transformation can be done with one traversal of the initial AST and allows for several other minor transformations during the lambda lifting process. The Haskell ADT representing the program in its new form is found below in Figure 3.12.

```
data IExpr = IInt Integer
           | ITrue
           | IFalse
           | ITupl (IExpr,IExpr)
           | IVar Name
           | IBinOp Name IExpr IExpr
           | IEq IExpr IExpr
           | IIf IExpr IExpr IExpr
           | IApp Name [IExpr]
           | IDec Name IExpr
           | ILet IExpr IExpr
           | ITup IExpr IExpr
           | IClosure Name Arg SymbolTable IExpr
           | ITopLevel IExpr IExpr
           | IPrintInt IExpr
           | Empty
```

Figure 3.12: diML IR to represent functions as top level closures

Uniquification of Variables A part of the transformation of the `DimlExpr` IR into the new IR that defines functions and lambda expressions at the top level, a renaming of all variables is performed. In an implementation of a compiler, the *symbol table* can map variables to their types or their values. With respect to this phase of the compiler, it is of concern that function definitions have access to the values of the variables that were in their lexical scope at the time of definition. By constructing the symbol table with these uniquely named variables, the code generation module is greatly simplified, as having unique variables names for all lexically scoped variables means that when looking up the values of variables in a symbol table, the lookup is always guaranteed to return the value of that unique variable. For instance, in the expression `let x = 1, fun f(x) = x + 1 in f(x)`, the x that is an argument to the function f, without renaming, could pose a conflict when constructing a symbol table: the first x would be assigned to the literal `IInt 1` whereas the argument x to the function f would be renamed to $x1$ via a stateful renaming function. This process guarantees that when the symbol table is queried for the value of x, the correct value is returned. In the code generation module, had the argument x to the function f not been renamed, a conflict would occur when the body of the function f wanted to look up the value of x, depending on the implementation, it would be uncertain as to what variable value would be returned.

Lambda Lifting First, a new IR is designed such that the resulting transformation of the `DimlExpr` IR is semantically equivalent, but converts every lambda expression and function definition in a let declaration to a `ITopLevel` expression, forcing code to

```
-- Initial DimlExpr IR
Let (Decl "x" (Lit (DInt 1)))
    (Let (Fun "f" "x" Nothing Nothing
              (BinOp "+"
                  (Var "x")
                  (Lit (DInt 1))))
         (Apply (Var "f") (Var "x")))

-- New IR
ITopLevel (IClosure "f" "x1" [("x",IInt 1)]
              (IBinOp "+"
                  (IVar "x1")
                  (IInt 1)))
          (ILet (IDec "x" (IInt 1))
              (ILet Empty
                  (IApp "f" [IVar "x",IVar "x"])))
```

Figure 3.13: A lambda lift transformation from `DimlExpr` IR to the new IR

be generated for all `ITopLevel` expressions before generating code for expressions that may call the hoisted functions, aside from the recursive calls in the body of the function definitions. Along with hoisting all function definitions and lambda expressions to a top level expression, the lambda lifting function (defined in IR.hs) performs action necessary for the code generation module to have access to the correct values of *free variables* in the body of a function. Free variables are defined to be variables that are not defined as arguments to a function, but rather variables that are in the local scope of the function where it is initially defined. Every function definition and lambda expression is converted into an abstraction known as a *closure*. A closure is simply a function definition closed over all lexically scoped variables existing in the context in which the function is initially defined. Converting functions to closures is necessary as the function definitions need to retain access to the values of lexically scope variables. An example of this transformation is found in Figure 3.13, using the example defined previously in this section regarding the renaming of variables.

3.5 Codegen to LLVM

The implementation of the diML compiler targets the LLVM Intermediate Representation (LLVM IR), an assembly-like intermediate representation that can generate competitively fast assembly code for nearly all contemporary computer architectures in use. However, LLVM is not only an IR, but is a set of sub-projects encompassing many mod-

ular and reusable compiler technologies including Clang, a native C++/C/Objective-C compiler that produces more verbose and accurate error messages along with producing code that executes faster than most code output by GCC (3x faster in the case of Objective-C compilation) [1]. The LLVM project was first started at the University of Illinois - Urbana Champaign in 2000 by Vikram Adve and Chris Lattner; since LLVM was open sourced in 2004, progress in many parts of the project has accelerated. Along with the statically typed, sophisticated type system directing the LLVM IR, the IR can capture high level language constructs easier than an assembly language like x86. Along with LLVM IR, the LLVM Core library supplies a large number of helper functions that aid in generating the LLVM IR, greatly simplifying sometimes complicated code generation cases. Because of the caliber of researchers dedicated to the project along with a deep interest in the LLVM project in the open source community, it seemed that LLVM IR would be the best choice to target for the implementation of the backend of the diML compiler [9].

3.5.1 Code Structure

The structure of the LLVM IR is C-like, differing strongly from that of x86 or RISC languages like ARM and MIPS. A program written in LLVM IR (denoted by the .ll file extension) is known as a *module*, where modules are linked at compile time (the compile time of the LLVM IR, not the compilation time of diML IR to LLVM IR) using the LLVM linker (lld project) that resolves conflicting function names and global variables along with other tasks performed by linkers. Each module is composed of global variables, top level functions, symbol table references, and an optional main block. Inside each function is a set of blocks, each block optionally denoted by a *name*. Similar to labels in other assembly languages, block names are used for control flow; control flow instructions in the LLVM IR use these block names to specify which memory location to jump to to continue program execution.

Taken from the LLVM Language Reference Manual, the code shown in Figure 3.14 displays a canonical "Hello, World" program used to introduce many programming languages' syntax. In this piece of code, the syntax and structure of LLVM IR modules can be more easily understood: The first line of code illustrates global variable declaration of the "Hello, World" string, named `@.str` for use in the function body of `@main`. In LLVM IR, all global variables and function declarations are known as global references and are always prefixed by the `@` symbol. The `;` symbol denotes a comment and the `%` denotes a store instruction where the name following the symbol names the register. Furthermore, the `define` keyword is reserved for function definitions, where the `declare` keywords signifies the function will be found in another module linked at compiler time. The keywords of the form `i`N, where N is a natural number, represents an integer type in the LLVM IR—LLVM IR is strongly typed and the types of values, both the argument and return type of functions along with the types of global variables must be annotated. Several other LLVM IR constructs are illustrated, including function attributes (e.g. `nocapture`), the syntax of the `getelementptr` function, etc,

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"Hello, World\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {
  ; Convert [13 x i8]* to i8  *...
  %cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}
```

Figure 3.14: "Hello, World" program written in LLVM IR [2]

but these are outside of the scope of this thesis and the structure and syntax outlined so far is enough to understand most of the code generated by diML code generation module. The interested reader should consult the LLVM Language Reference Manual for a more detailed overview of the LLVM IR syntax and code structure [2].

```
newtype LLVM a = LLVM { unLLVM :: State Module a }
```

3.5.2 Implementation

The implementation of the code generation module for diML is found in the Codegen.hs file located in the src directory of the github repository mentioned previously in this thesis. The AST arrives at the code generation module in the form of an `IExpr`, representing the lambda lifting transformation of the original `DimlExpr` AST; a function `codegenTop` is applied to the IExpr, resulting in an LLVM module executable by a series of LLVM backend transformations. This is the last phase of the compiler, and the compilation, optimization, and execution of the resulting LLVM module can be handled entirely by code from the LLVM Core library.

First, several data types are defined such that the stateful nature of code generation is represented; this necessitates the use of the State Monad such that code in the `codegenTop` and `codegen` functions can generate code in an imperative style using the *seemingly* imperative chaining monadic computations (i.e. the code is still pure, but imitates stateful computation with the State Monad's implementation of the `>>=` operator). The `CodegenState` data type represents the state of the current of the code generation phase for each function—all code instruction blocks are generated within functions. Likewise, the `BlockState` data type represents the code generation state of

the current instruction block for which instructions are being generated. By wrapping these data types in a State Monad `Codegen a`, the stateful nature of code generation is emulated. In the definition of these two data types, found in Figure 3.15, *record syntax* is used, a feature of the Haskell language that automatically generates functions with the name of each *field* defined in the record. The type annotation of each field expresses the return type of the function, the argument being a value of whatever type the record field is defined for. For instance, the full type signature of the function `currentBlock` defined in the data type `CodegenState` is `CodegenState -> Name`.

```
type SymbolTable = [(String, Operand)]
type Blocks = Map.Map Name BlockState

data CodegenState
  = CodegenState {
    currentBlock :: Name         -- Name of the active block to append to
  , blocks       :: Blocks             -- Blocks for function
  , symtab       :: SymbolTable        -- Function scope symbol table
  , blockCount   :: Int                -- Count of basic blocks
  , count        :: Word               -- Count of unnamed instructions
  , names        :: Names              -- Name Supply
  } deriving Show

-- CodeGenState wrapped in state monad
newtype Codegen a = Codegen { runCodegen :: State CodegenState a }
  deriving (Functor, Applicative, Monad, MonadState CodegenState )

-- for blocks within functions
data BlockState
  = BlockState {
    idx   :: Int                      -- Block index
  , stack :: [Named Instruction]      -- Stack of instructions
  , term  :: Maybe (Named Terminator) -- Block terminator
  } deriving Show
```

Figure 3.15: Two data types representing top-level LLVM function generation and instruction blocks.

Types To greatly simplify the implementation of the code generation module, all values have type double, except for the representation of the tuple data type which is represented by a vector with two fields of type double. The reason this simplifies code is that all function definitions can assume that both the argument and return type will be of type double; if this wasn't the case, another field may have to be added to one or several of the data types defined in Figure 3.15 to track the return type of the

```
import LLVM.General.AST
import qualified LLVM.General.AST.Constant as Const
import qualified LLVM.General.AST.Float as F

false :: AST.Operand
false = ConstantOperand $ Const.Float (F.Double 0)

true :: AST.Operand
true = ConstantOperand $ Const.Float (F.Double 1)

int :: Integer -> AST.Operand
int = ConstantOperand . Const.Float . F.Double . fromIntegral
```

Figure 3.16: Primitive value representations in Codegen.hs

value returned by a block or function definition. Primitive values such as integers and booleans are converted to their respective forms as a double; integers are converted to their respective double representation (e.g. IInt 7 $\implies$ 7.000000e+00), whereas true and false values are converted to 1.000000e+00 and 0.000000e+00 respectively. Even if different data types were supported in the code generation module, booleans would still have to be converted to a number representation, as computers can only understand the notion of *true* and *false* by numerical representation. The Haskell code that generates the LLVM IR with respect to these types is shown in Figure 3.16.

Instructions One of the critical parts of code generation is to generate the proper assembly code instructions to perform the high level action specified by the programmer of the original input program. In assembly code, these high level actions are generally more complex than their high level abstraction. To generate LLVM IR that accomplishes these actions, a composition of several basic operations at the register level is necessary. A function is defined to add an instruction to the current block that code is being generated for, named `instr`, such that applying `instr` to an instruction simply appends the instruction to the end of the current block being generated and assigns its result to a register. All instructions necessary for the implementation of diML's back-end are represented as functions that take several arguments depending on the instruction, and return a representation of the instruction using the bindings in LLVM-general. The construction of each instruction passes through several functions defined in LLVM-general-pure such that if the code being generated has a type error at the LLVM IR level it can be caught before generating the LLVM IR—this technique allows the implementer to catch type errors during compilation of the compiler such that the implementer can debug Haskell, arguably easier than debugging LLVM IR. Several important instructions and the `instr` function's implementation are shown in Figure 3.17.

```
import qualified LLVM.General.AST.CallingConvention as CC

instr :: Instruction -> Codegen (Operand)
instr ins = do
    n <- fresh
    let ref = (UnName n)
    blk <- current
    let i = stack blk
    -- appends the '%register = instruction' to the end of the block
    modifyBlock (blk { stack = i ++ [ref := ins] } )
    return $ local ref

-- subtraction
fadd :: Operand -> Operand -> Codegen Operand
fadd a b = instr $ FAdd NoFastMathFlags a b []

-- function call
call :: Operand -> [Operand] -> Codegen Operand
call fn args = instr $ Call False CC.C [] (Right fn) (toArgs args) [] []

-- allocates memory for a variable
alloca :: Type -> Codegen Operand
alloca ty = instr $ Alloca ty Nothing 0 []

-- stores a value in a variable mem loc
store :: Operand -> Operand -> Codegen Operand
store ptr val = instr $ Store False ptr val Nothing 0 []

-- branch instruction
br :: Name -> Codegen (Named Terminator)
br val = terminator $ Do $ Br val []

-- return from an instruction block
ret :: Operand -> Codegen (Named Terminator)
ret val = terminator $ Do $ Ret (Just val) []
```

Figure 3.17: Examples of functions constructing instructions in LLVM IR

Codegen Functions Finally, after presenting the necessary data types, how they can be wrapped in a State Monad to emulate stateful computation, how primitive values are represented, and how instructions are added to instruction blocks inside function definitions, the `codegenTop` and `codegen` functions can be described. The `codegenTop` function is defined for only the values that can be found at the top-level of the AST; this includes `ITopLevel` expressions, `ILet` expressions, and `IClosure` expressions. If

the lambda lifting transformation did not construct the new AST correctly, potentially different expressions could occur at the top level, as an `ITopLevel` expression contains two fields that could be any form of an `IExpr`. The purpose of `codegenTop` is to construct function definitions with the help of several helper functions and append them to the LLVM module (program) that is being generated. There will only be one top level `ILet` expression, named "main" as required by LLVM IR, as a diML program is represented by a single let expression allowing arbitrary nesting of both functions and other let expressions; since functions nested inside let expression will have all been converted to closures and hoisted to the top level, a case for `IClosure` in the `codegen` function is not necessary. Conveniently, LLVM-general exports a function `functionDefaults` and data constructor `GlobalDefintion` that allows for the easy construction of function definitions by passing in the proper arguments constructed by generating the instruction blocks for function bodies in the `codegenTop` function.

```
addDefn :: Definition -> LLVM ()
addDefn d = do
  defs <- gets moduleDefinitions
  modify $ \s -> s { moduleDefinitions = defs ++ [d] }

define :: Type -> String -> [(Type, Name)] -> [BasicBlock] -> Linkage -> LLVM ()
define retty label argtys body linkage = addDefn $
  GlobalDefinition $ functionDefaults {
    linkage     = linkage
  , name        = Name label
  , parameters  = ([Parameter ty nm [] | (ty, nm) <- argtys], False)
  , returnType  = retty
  , basicBlocks = body
  }
```

Figure 3.18: Functions used to construct and append function definitions to the current `LLVM a` monad representing the program module

The two functions defined in Figure 3.18 make up most of the body of the `codegenTop` function, as `ITopLevel e1 e2` expressions simply call `codegenTop` on e1 and then e2. Both the `IClosure` and `ILet` cases use the `define` function, differing only in the passing of argument `argtys` as let expressions take no arguments. The implementation of the `codegenTop` function is a bit more complex than what is illustrated here, but through careful reading and consultation of the source code of the Codegen.hs module, the astute reader should be able to derive an implementation of such a function for a compiler for a language other than diML.

The `codegen` function differs slightly from it's parent function `codegenTop` in the sense that its role is not to append function definitions to the LLVM module definition under construction, but to append instruction blocks to the current function being defined. After generating code to define and allocate memory for the arguments to a

function, `codegen` is called on the body of the function to begin construction of the respective instruction blocks. The `codegen` function is defined for every form of an `IExpr` aside from `IClosure`, as every other `IExpr` is allowed to be in the body of a function, including arbitrarily nested let expressions. To illustrate an interesting case of the `codegen` function, the case for `IIf` expressions is defined below. The reason this case is chosen to illustrate code generation is that it encompasses several non-trivial parts of the code generation implementation– creating blocks, control flow instructions, and phi nodes.

```
codegen :: IExpr -> Codegen Operand
codegen (IIf cond tru fals) = do
    ifthen <- addBlock "if.then"
    ifelse <- addBlock "if.else"
    ifexit <- addBlock "if.exit"
    condval <- codegen cond
    test <- fcmp ONE false condval
    cbr test ifthen ifelse
    setBlock ifthen
    trueBrVal <- cgen tru
    br ifexit
    ifthen <- getBlock
    setBlock ifelse
    falsBrVal <- cgen fals
    br ifexit
    ifelse <- getBlock
    setBlock ifexit
    phi double [(trueBrVal, ifthen), (falsBrVal, ifelse)]
```

Figure 3.19: Code generation for `IIf` expression

First, three new blocks are generated—one for the case in which the condition evaluates to true, the "then" branch, another for the "else" branch, and the last for the instruction block to jump to at the end of either the then or else branches. Next, code is generated for the condition of the if statement (a recursive call to codegen). After returning an operand (either a true or false value, as dictated by the type-inference module), the returned operand is checked to see if it's equivalent to 0 or not, the former resulting in a jump to the `ifthen` block and the latter to the `ifelse` block. Next, the block for the `ifthen` begins generation; this starts with a function call `setBlock ifthen` because the `codegen` call to test the condition of the `IIf` expression most likely set the current block to the block of code generated for the condition. Code is then generated for the case in which the condition is true (the `tru` field in the `IIf` expression), and after, a branch instruction to the `ifexit` block is generated. The codegen code for the `ifelse` branch is analogous to the `ifthen` block. The last line

of code illustrated in this case of `codegen` is the generation of a `phi` node—an LLVM instruction that results from the SSA form LLVM is designed to have. Essentially, the code states "if the `ifthen` block was executed, return the value `trueBrVal`, else return the value `falsBrVal`". The `phi` node sees from where it was jumped to, and returns the value associated by the result of the instruction block as a result.

In this section, the implementation of the Codegen.hs module has been described to the fullest extent with respect to the scope of this thesis. For a more comprehensive overview of LLVM IR and the components of the LLVM Core module, please consult the LLVM Language reference manual [2]. For a more complete overview of implementing a back-end for a compiler written in Haskell using the LLVM bindings for Haskell, the interested reader can consult Stephen Diehl's Kaleidescope tutorial [5] which outlines the implementation of a small, imperative language originating from a tutorial available on the LLVM web-page. If more context is necessary for the reader to understand the implementation described in this section, the code is available online in a public github repository [6].

Chapter 4

Conclusion

4.1 Summary

In this thesis, an overview of the design and implementation of a compiler for the language diML targeting LLVM IR using the Haskell programming language was presented. Although not all the code necessary for a complete implementation is presented, an overview of the process along with specifics relating to particularly difficult phases is given in detail. Also, resulting from my interest in Programming Languages (PL) theory, several deductive systems that beautifully characterize different aspects of the underlying theory of programming language implementation are displayed (e.g. Figure 2.4 on page 12 depicting the static semantics for diML). Though these are not intuitively read if the reader is unfamiliar with PL theory, the sections in which these are presented aim to adequately explain how the systems translate easily into understandable and intuitive functions that compute the various results needed by the respective phases.

In Chapter 1, the formal specification of diML's syntax and type system was introduced, as well as the motivation for the implementation of the compiler and the technologies involved during the implementation process. For the reader unfamiliar with the different phases involved in the design and structure of a compiler, a detailed overview of the different components of which compilers are generally comprised was given in Chapter 2. In Chapter 3, the implementation of the diML compiler along with relevant Haskell constructs relating to the implementations of the different phases were presented; the parser combinator library Parsec, the complex yet beautiful theory of type-inference, and the use of the LLVM-general and LLVM-general-pure libraries with respect to translating diML IR into LLVM IR were all covered in enough detail such that the astute reader, along with the help of several resources mentioned in the respective sections, could implement a compiler for a functional language using Haskell as the tool for implementation. The subject of compilers can be overwhelming to students unfamiliar with the body of information associated with the design and implementation of such complex pieces of software.

This thesis covered several topics in more depth than a research paper of a similar nature would, but research was never the expected outcome of this thesis. It was written with the intention that the reader would not feel overwhelmed, providing a structured and easy to understand overview of the process of designing such a large program. Furthermore, this thesis aimed to provide reiteration of the fact that using Haskell to implement modern compilers for newly invented programming languages is relatively simple and arguably much easier than other implementation languages such as C. Due to robust and powerful libraries such as Parsec and LLVM-General-Pure, along with language level constructs such as monad transformers, compiler modules can be implemented with fairly few lines of code while leaving the code readable and understandable to those even just slightly familiar with Haskell syntax and monads.

As presented, the diML compiler is a working piece of software with a single form of I/O: the `printInt` function. As a result, diML programs cannot perform sophisticated effectful computations such as interacting with a data base, reading and writing to files, or reading user input from the console. However, the `printInt` function is enough such that diML, given the current state of the compiler, can implement mathematical functions such as the fibonacci function to calculate the n^{th} fibonacci number.

4.2 Project History

During the past five months I was able to implement a working compiler for diML, though it is not the diML I aimed to implement. Along the way I was hindered by several flaws in the design of my implementation that I was not able to quickly resolve, and as a result, could only implement code for all phases for a subset of expressions with respect to the diML constructed during the Fall and Spring semseters of Dr. Ligatti's Programming Languages courses. Although the expressions comprising this implementation of diML seem lacking to me, I was able to accurately implement several non-trivial phases that are not necessary for all compilers: I discovered the necessity for a transformation of the original diML AST called *closure conversion* such that the code generation module could be more succinctly defined, implemented a type inference module using *Generalized Damas-Milner Type Inference*, and reinforced the notion that LLVM IR is not only a good choice for a target language, but could potentially subsume any other target language when using Haskell as the language of implementation.

This discovery of the necessary *lambda lifting* transformation was particularly exciting because it came after trying to implement the code generation module for diML following the base design of a code generation module for a toy language called *Kaleidescope*, implemented by Stephen Diehl [5]. I noticed that all functions defined in LLVM were top level, and that in order for code generation to work using monad transformers defined similarly to Kaleidescope's, the diML AST could not contain nested functions. After some research, I realized that closure conversion had a well established theory supporting it and I was able to solve my problem after several hours of coding.

The other aspects of this implementation of the diML compiler that I consider worth noting are type inference and compilation to LLVM IR, easily the two most interesting parts of the project from my perspective. Type inference influenced optional type annotations, yet another interesting feature of diML. However, I must admit that I relied on Stephen Diehl's tutorials heavily for a sophisticated code structure and boilerplate monad transformers that I might use for ever subsequent compiler implementation using Haskell. This allowed me to focus on the high level implementation aspects differing from the imperative language outlined in several very detailed tutorials [4] [5]. I do not regret this, however, because if I didn't have such an organized, well developed collection of information regarding the implementation process of a compiler written in Haskell targeting the LLVM IR, the diML implementation presented in this thesis would be even smaller; in hindsight, I'm not sure I would have completed such a project as I started with virtually no knowledge of what the implementation process of a compiler entails.

Future Work I plan to continue working on this compiler, implementing several expressions and language features that I did not have time to implement and complete my thesis. Perhaps the first addition to the diML compiler would be to add a `readInt` function such that programmer writing in diML can construct more useful programs. Other ideas considered as additions during the implementation process are pattern matching, lists, case expressions, sum types, references and arrays, along with several more found on the project webpage [6]. These ideas have varying levels of difficulty with respect to the amount of effort necessary for their implementation, yet choosing just several to implement would greatly expand the power of the diML language. This implementation of diML is "open source" in the sense that I would be happy to accept pull requests and new features implemented by others interested in further developing diML.

In the field of programming languages and compilers, there are almost endless features and language constructs worth implementing, each teaching the implementer something new with respect to how programming languages *work* under the hood. I plan to keep studying and adding to diML, hoping that this compiler can become an ongoing exercise for me for the next few years.

4.3 Closing Remarks

Much of the time spent on the implementation of the diML compiler was spent during the late hours of the night. This is not because of procrastination or poor time management, but rather the result of an inability to stop writing code when it became far too late due to the state of mind that results from hours of deep thought. Often, I would begin working at a reasonable hour and continue to work deep into the night as the more I thought about the problems or particular function I was trying to implement, it became more difficult to escape such a state of focus and going to sleep became a

chore. The insights into the intricacies of the engineering process gained during these periods are perhaps some of the most valuable. I would recommend the design and implementation of a compiler to any student desiring to gain a deeper understanding the tools (programming languages) programmers use on a daily basis—the tools used to build the software systems that humans rely on to manage everything from personal email to an entire country's health-care system.

The implementation and writing process of this thesis has fostered my interest in compilers and programming languages as much as the classes I have taken on the subject matter have; the longer I worked on the various phases of the compiler, the more appreciation I developed for the theory underlying such a complex piece of software. The field of Programming Languages and the practice of implementing a compiler are deeply connected, and through this process I have grown an immense appreciation for both. As a result, I will continue to spend time after graduation researching and learning more about the design of programming languages along with the piece of software that brings them into the real world—the compiler.

Bibliography

[1] *Clang/LLVM Maturity Report* (Moltkestr. 30, 76133 Karlsruhe - Germany, June 2010). *See* `http://www.iwi.hs-karlsruhe.de`.

[2] LLVM Language Reference Manual. `http://llvm.org/docs/LangRef.html`, 2015. [Online; accessed 17-November-2015].

[3] Damas, L., and Milner, R. Principal Type-Schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (1982), ACM, pp. 207–212.

[4] Diehl, S. Write you a Haskell - Hindley Milner Type Inference. `http://dev.stephendiehl.com/fun/006_hindley_milner.html`, 2014. [Online; accessed 15-November-2015].

[5] Diehl, S. Implementing a JIT Compiled Language with Haskell and LLVM. `http://www.stephendiehl.com/llvm/`, 2015. [Online; accessed 15-November-2015].

[6] Dietert, T. dimlCompiler. `http://www.github.com/tdietert/dimlCompiler`, 2015.

[7] Heeren, B., Hage, J., and Swierstra, S. D. Generalizing Hindley-Milner Type Inference Algorithms. *Technical report UU-CS* (2002).

[8] Lattner, C. Introduction to the LLVM Compiler System. `http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf`, November 2008.

[9] Lattner, C., and Adve, V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization* (Washington, DC, USA, 2004), CGO '04, IEEE Computer Society, pp. 75–103.

[10] Leijen, D. Parsec, a Fast Combinator Parser. `http://research.microsoft.com/en-us/um/people/daan/download/parsec/parsec-letter.pdf`, 2001.

[11] Louden, K. C. *Compiler Construction, Principles and Practice*. Course Technology, 1997.

[12] Milner, R. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences 17*, 3 (1978), 348–375.