Compiler Construction Project

Diego Compiler

Dan Sebastian Thrane
<dathr12@student.sdu.dk>
Lars Thomasen
<latho12@student.sdu.dk>
Group 4

Spring 2015

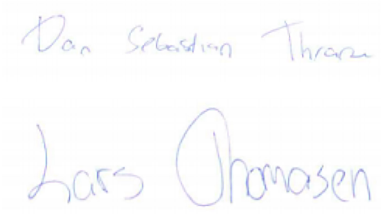
Department of Mathematics and Computer Science
University of Southern Denmark, Odense

Spring 2015
KSL

CC, Spring 2015
Exam Project, part 4

Group	4

Date	05/05/15

Name	Dan Sebastian Thrane Lars Thomasen
Birthday	090693 040485
Logins	dathr12 latho12
Signature	

This report contains a total of 83 pages.

Please write very clearly. Under Logins, give your student (`student.sdu.dk`) login. If you have an IMADA login that is different from your student login, give that in parenthesis.

Compiler Construction

Diego Compiler

Dan Sebastian Thrane
Lars Thomasen

IMADA
Institute for Mathematics and Computer Science

Abstract

Covers the construction of a compiler, as a bachelor project. This report serves to cover the design choices and any issues solved while development was ongoing. Tools as flex and bison has been used to handle the front-end. The programming language is C, and runs on any linux environment.

The resulting code won "Compiler of the Year 2015" which was measured on elegance, speed, and the amount of additions created.

Ref: http://www.imada.sdu.dk/~kslarsen/Courses/CompilerBachelor-2015-spring/Konkurrence/winner.html

Table of Contents

Chapter 1

Introduction

Introduction

This report is documenting the progress and result of creating a compiler for the Diego language. The base language has been supplied at the beginning of the project and has since been extended to reflect new features.

Each chapter in the report reflect a different phase of the compiler, documenting design choices made, implementation decisions, testing, and outcome of that phase. It is expected that the reader already knows the theoretical aspects of compiler construction before reading this.

The work in this project has been based heavily on the work done in the course book [**?**]. Along with the information available from the course website [**?**], and the related compiler construction course [**?**].

1.2 The phases

There are five phases in this compiler, each shown in the figure below. Some phases of the compiler has been combined to a single phase in the report as they complement each other directly.

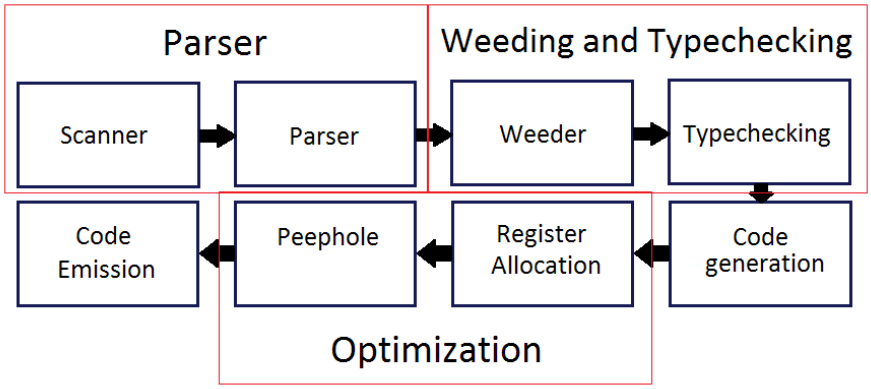


Figure 1.1: The phases, the red line indicates they are combined in the report.

1.3 Usage guideline

The compiler takes the following optional flags:

```
Usage: kitty -flag

-x:
    Turns of runtime safety improvements in order to reach better
    performance.
-r | --run:
    Automatically compiles and executes the Diego program.
-v | --verbose:
    Activates debug print-outs of each phase to stderr.
--simplereg:
    Turns off liveness analysis and peep-hole optimization.
    Instead every temporary is pushed to the stack.
--nopeephole:
    Turns off peep-hole optimization.
--no:
    Turns off the initial welcome message.
--pretty:
    Pretty prints the AST.
--tree:
    Pretty prints the AST in JSON format.
--symboltable | --earlysymboltable | --latesymboltable:
    Prints the symbol table at certain points during compilation.
--dumpgraph:
```

```
    Prints out the flow graph.
--exit:
    Stops compilation once a debug print has finished.
```

1.4 Additions

The following additions has been made, extending the basic requirement of the project.

Language Extensions

- Standard library and including other source files.
- Strings with various string operations implemented in the standard library.
- Java-style annotations.

Runtime Safety Improvements

Run-time checks for:

- Array index values (return value 2).
- Division by zero (return value 3).
- Positive argument for array allocation (return value 4).
- Dereferencing of null pointers (return value 5).
- Out-of-memory (return value 6).

Extra Extensions

- Peep-hole Optimization.
- Advanced register allocation (Liveness Analysis).
- Constant folding in weeding phase.

1.5 Implementation Status

The compiler is feature-complete. Meaning that all the features are implemented and in a working condition. Garbage collection was planned but due to time constraints, was never implemented.

A single bug remained in the advanced register allocation, excluding the register EBX from being enabled in the allocation phase. This is due to three tests were causing an undetected register conflict between special usage of EBX and the allocation (missing interference edge in analysis). The cause of the bug is known, but time constraints prevented it from being resolved.

Chapter 2

Introduction to Diego

Diego is a simple statically typed, imperative programming language. The language is very young, for that reason it supports relatively few constructs, when compared to more mature languages. In this section we will give a brief introduction to Diego, present its core grammar, and highlight any extensions made to the language. Only a brief introduction will be given here, further discussion will come later in the report.

2.1 Grammar

The core language grammar can be seen in figure 2.1 and 2.2. Should the reader not feel like parsing the entire grammar in their head, then listing 1 showcases most of the core features of the language.

Diego as a language supports only two primitive types, those are integers and booleans. It has support for type aliases, arrays, and records (often known as structs). Amongst its more notably features are nested functions, as presented in the hello world example. More on how this is implemented in the type chapter, as well as the code generation chapter.

Also noteworthy are the restrictions put on declarations and statements. In most languages these can be mixed liberally, in this language, however, there is a strict requirement that declarations must come before statements, and that they may not be mixed. There are upsides to doing things this way, mainly that this enforces a nice coding style. These constraints can however be considered too strict, and could be subject to change in a future version of the language.

⟨function⟩ : ⟨head⟩ ⟨body⟩ ⟨tail⟩
⟨head⟩ : **func id (** ⟨par_decl_list⟩ **) :** ⟨type⟩
⟨tail⟩ : **end id**
⟨type⟩ : **id**
| **int**
| **bool**
| **array of** ⟨type⟩
| **record of** { ⟨var_decl_list⟩ }
⟨par_decl_list⟩ : ⟨var_decl_list⟩
| ε
⟨var_decl_list⟩ : ⟨var_decl_list⟩ **,** ⟨var_type⟩
| ⟨var_type⟩
⟨var_type⟩ : **id :** ⟨type⟩
⟨body⟩ : ⟨decl_list⟩ ⟨statement_list⟩
⟨decl_list⟩ : ⟨decl_list⟩ ⟨declaration⟩
| ε
⟨declaration⟩ : **type id** = ⟨type⟩ **;**
| ⟨function⟩
| **var** ⟨var_decl_list⟩ **;**
⟨statement_list⟩ : ⟨statement⟩
| ⟨statement_list⟩ ⟨statement⟩
⟨statement⟩ : **return** ⟨expression⟩ **;**
| **write** ⟨expression⟩ **;**
| **allocate** ⟨variable⟩ ⟨opt_length⟩ **;**
| ⟨variable⟩ = ⟨expression⟩ **;**
| **if** ⟨expression⟩ **then** ⟨statement⟩ ⟨opt_else⟩
| **while** ⟨expression⟩ **do** ⟨statement⟩
| { ⟨statement_list⟩ }
⟨opt_length⟩ : **of length** ⟨expression⟩
| ε
⟨opt_else⟩ : **else** ⟨statement⟩
| ε
⟨variable⟩ : **id**
| ⟨variable⟩ **[** ⟨expression⟩ **]**
| ⟨variable⟩ **. id**

Figure 2.1: Grammar, part 1.

```
type myType = int;
var myVariable: myType, myInt: int;

func hello(): bool
    func world(number: int): int
        return myVariable + number;
    end world

    myVariable = 10;
    myVariable = world(32);
    return myVariable == 42;
end hello

write hello();
```

Listing 1: Hello Kitty!

2.2 Extensions to the Language

Load Directives

Allows for Diego code to be separated into multiple files with an almost C like load statement. The grammar for this feature is:

```
#load FILE_NAME
```

Strings

Strings have been added to the language. With this a new primitive type has been added to the grammar. This uses the `char` keyword. Similarly this supports both character literals and string literals, this uses the C syntax. No special characters (escaped) are supported in strings or literals. The only exception to this rule is new-line, which is supported using the traditional C syntax (`'\n'`).

Notes/Annotations

This language has support for what in the Java world is known as annotations. These are small notes, that may be put on different parts of the program, annotating some extra information to it. The annotations are structured, which means that they are allowed have literal values associated with them. An annotation does not require any

⟨expression⟩ : ⟨expression⟩ **op** ⟨expression⟩
| ⟨term⟩
⟨term⟩ : ⟨variable⟩
| **id** **(** ⟨act_list⟩ **)**
| **(** ⟨expression⟩ **)**
| **!** ⟨term⟩
| | ⟨expression⟩ |
| **num**
| **true**
| **false**
| **null**
⟨act_list⟩ : ⟨exp_list⟩
| ε
⟨exp_list⟩ : ⟨expression⟩
| ⟨exp_list⟩ **,** ⟨expression⟩

Figure 2.2: Grammar, part 2.

values. This can be used for various different things, which are discussed further in the annotation section. Here the syntax for annotations are exemplified:

```
@AnnotationNoValues
@AnnotationWithValues(param1 = 10, someOther = false, param2 = 42)
```

The annotations can currently only be placed on bodies, and come before the declaration list.

Chapter 3

Parsing

Lexical Analysis with Flex

Lexical analysis is used to recognize patterns in an input stream of characters. The Flex tool[**?**] is used to generate the actual lexer component of our compiler. Flex takes as input a description of the tokens. These descriptions are expressed as regular expressions for each token that needs to be recognized. For each token, we can specify what to pass onto the parser. Here we can for example pass extra data about the token, for identifier tokens, this might for example be what the actual identifier is.

Most of the Flex syntax is trivial, and are simply taken directly from the language grammar. Ways to handle comments has some design choices, and are thus of interest.

Diego supports both single line comments, and multi-line comments. Single line comments are started with the "`#`" token. Everything on the right of that token, should be ignored on that line.

Multi-line comments use "`(*`" to start, and "`*)`" to end multi-line comments. They are allowed to be nested, a multi-line comment must be fully closed, otherwise an error is reported.

To implement these comments, we have used the following states:

1. `INITAL`[1]
2. `COMMENT`
3. `MULTICOMMENT`

Each state indicates a transition as shown in the figure below. When in either comment state, all symbols but "`\n`" (and "`*)`", for multi-line) are ignored.

[1]The INITIAL state is a built-in state in Flex, which indicates the initial state.

Nested comments has been implemented by keeping track of how many levels deep the nesting is. Only if this count reached zero, is the state set back to `INITIAL`.

The DFA shown in figure 3.1 shows how comments (without nested comments) has been implemented:

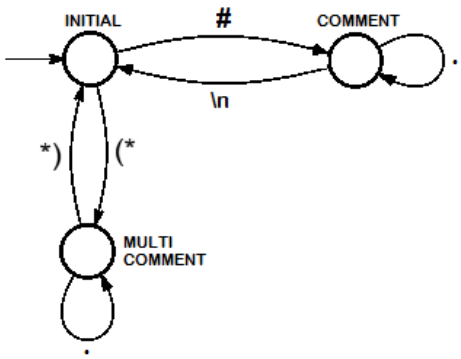


Figure 3.1: The DFA for implementing comments in Diego

3.1 Parsing with Bison

The main purpose of the parser, is to take tokens from the lexer, and determine if these tokens, together, produce valid sentences in the language. The parser should also output an abstract syntax tree. These are regular syntax trees with detail unnecessary for the remaining phases thrown away.

The parser is generated with a tool called Bison[**?**]. Bison works a lot like Flex, this time it uses a context-free grammar to describe the languages structure and the Lexical Analysis generated by Flex to retrieve a sequence of tokens from the input. Bison then returns a C-program which will parse a program using the context-free grammar.

Semantic records are specified in the union type at the top of the Bison file, here we attach a data-type that each record may have.

```
1 %union {
2     int32_t token;
3     int32_t number;
4     char *string;
5     struct AST_FUNCTION *function;
6     struct AST_HEAD *head;
7     struct AST_BODY *body;
8     ...
```

```
}
```

When declaring a type to a non-terminal symbol, these are then connected to the semantic record specified in the union-type so that a value may be stored.

```
%type <function> function
%type <head> head
%type <body> body
...
```

For each rule in the grammar, a function is called for the appropriate right hand side, which returns a node in the AST.

Unless extra actions are taken, then the grammar will produce conflicts. One of the conflicts we had to resolve had to with operator precedence. The following shows the grammar:

```
expression      :   expression T_MULT expression
                |   expression T_DIV expression
                |   expression T_ADD expression
                |   expression T_SUB expression
```

This would cause a case where the following has to be determined in order to know when to reduce a series of tokens to the left hand side of the grammar:

$$\begin{aligned} \text{expression} + \text{expression} * \text{expression} &= (1 + 2) * 3 \\ &= 1 + (2 * 3) \end{aligned}$$

We would like Bison to create an AST, which sets the same precedence as in mathematics. This is done by setting precedences with the operators in question, as seen below.

```
%left T_AND T_OR
%left T_EQUAL T_CNE
%left T_CEQ T_CLE T_CRE T_LEQ T_GEQ T_BANG
%left T_ADD T_SUB
%left T_MULT T_DIV T_PER
```

Setting a precedence tells Bison whether to reduce using the available rule, or shift to another token. The `%left` precedence rule here indicates that Bison should always prefer the reduction. If more reductions are available, the last mentioned precedence weighs the most.

Limitations

There are 21 shift/reduce[2] conflicts when parsing the grammar in our compiler, the default, when no precedence are specified, is to shift. Because of this default, the conflicts are resolved as expected and no errors are caused. Optimally specifying a shift rule precedence for these conflicts is optimal just to get rid of the warnings, but this was never done.

An example of one of these conflicts, is the dangling else, when the parser have read a statement, it can either shift into an opt_else, or it could reduce what has already been parsed. The correct way here is to shift and include the else.

```
1 T_IF expression T_THEN statement . opt_else
```

Building the Abstract Syntax Tree

Correctly determining if sentences are valid is a big part of parser. But the parser has another big job, that is, passing a tree on to the other modules of the compiler. This tree represents the logical structure of the program.

A large portion of the code necessary to produce the AST in Bison, is nothing but busy work. Large portions were directly generated with a small Groovy script, in the following section we will describe the convention that this was generated by.

Structs for AST Nodes

There is for every non-terminal generated a single struct. It uses the following template:

```
1 struct AST_RULE_NAME {
2     int lineNumber;
3     enum { K_RULE_NAME_K1, K_RULE_NAME_K2 } kind;
4     union {
5         struct { /* data for kind 1 */ } eDeclListK1;
6         struct { /* data for kind 2 */ } eDeclListK2;
7     } val;
8 };
```

Each node contains some meta-data about where it comes from, and any other information that may be needed further on in development. This could for example be for

[2] A position where both shifting and reducing is possible.

error reporting purposes. Currently each node has a line number from where it was reduced.

For every rule there exists, for that non-terminal a new kind is created along with any data that it may contain. The data contained within each kind depends on the rule. All non-terminals are included as pointers to their respective nodes. The associated values of terminals are also included, if they contain any additional data. An example of this could be the `T_ID` token, which contains the actual name of the identifier.

For consistency it was decided that non-terminals, that only has one associated rule, should still contain a `kind` enum. This allows for all non-terminals to be expanded, without breaking existing code. Similar idea for generating the value structs were used. Thus meaning that structs that would only have one associated value, would still have an enclosing struct. Note that structs in C do not have any memory overhead.

3.1.1 Manual Tweaking

In some cases the tree resulting from this generation is inconvenient to work with, and may contain irrelevant parsing details. In this section the major changes going from a parser tree to an AST is described.

The first major change that was done were for list types. Consider the natural way of expressing a list type in the grammar:

```
1 item_list     :   item
2               |   item_list T_COMMA item
3               ;
```

This will result in a tree which has two unfortunate properties:

1. The node doesn't use any list type. First of all this makes the intention less clear. It will also be harder to manipulate the actual list, due to lacking a uniform manipulation API.
2. The resulting node(s) are dependent on how the grammar is actually implemented (left/right recursion).

To fix this issue all nodes that represented a list type were replaced with a new linked list structure. This list then contains the actual elements. The implemented list, is a simple linked list, which contains two structs. An overall "list" struct, which contains a pointer to the head and last, it also contains the size. The second struct then contains pointer to next and previous element, as well as the actual data. Having a pointer to both head and last element of the list. Having this allows the grammar to use either of left and right recursion, without introducing any performance overhead.

Next consider this small piece of Diego code:

```
1 var total: int, k: int, f: int;
2 var t: int;
3 var running: bool;
```

The corresponding parser tree would contain three declarations. The first containing the node which declares all three of `total`, `k`, and `f`. This is purely a parsing detail, ideally we would have our AST contain five separate declarations. One for each variable declaration. This change required "pushing" the var types into the declaration node. This was done simply by adding a new kind, and associated value. Next relevant flattening code was added to declaration reduction in Bison. Which would then take a var decl list (the previous type) and flatten it, to produce the new nodes.

Finally there were a few grammatical rules that only existed to allow certain lists to be empty. These nodes were removed and allowed for an empty case, were an empty list would be pushed onto the stack.

3.2 Tests

For testing of the Flex scanner and Bison parser, additional code have been written. This involves a pretty printer function, which takes the input and uses parenthesis to encapsulates nodes and their children. This pretty printer can be invoked using the `-pretty` flag, but is for the time being also the default output.

Testing the scanner and parser has mainly been done by validating the code from a pretty printer. Two different pretty printers are implemented, one printing a textual representation of the program, the other printing it in JSON.

These tests were combined with the overall testing framework used throughout the process. Since lexical analysis and parsing is crucial to the compiler, we would have seen tests not being able to complete, if there were any failures in the parser.

A single test failed, which is directly related to the parser. The compiler fails to recognize nested absolute operators, and mistakes them as the token for boolean or.

Chapter 4

Weeding

4.1 Traversing the AST

Several components of the compiler needs to access, and traverse the AST. Just to name a few: the weeder, the typechecker, extensions. Adhering to software engineering best practices, a small component, named `ast_visitor`, was made to more easily facilitate the development of components that must traverse the AST.

The design choice behind this lies in abstracting away the actual traversal of the tree and isolate any logic regarding the weeding and type-checking phases. This keeps the source code clean and readable, allowing for easier coding and understanding of the code.

The visitor has a function for visiting every kind of node, which exists in the AST. As parameters it takes the node, and also a list of `visitor_delegate`s. These are structs which holds pointers to functions, for visiting and exiting a specific type of node. Each `astVisit` function is then responsible for making any recursive calls, to traverse the tree, and calling the delegate functions. The following code snippets shows the function for visiting `HEAD` nodes works.

```
1  void astVisitHead(AST_HEAD *p, linked_list *del) {
2      // Visit delegates
3      FOR_EACH(del, visitor_delegate, {
4          it->visitHead(p);
5      });
6      // Recursive calls for tree traversal
7      astVisitParDeclList(p->params, del);
8      astVisitType(p->returnType, del);
9      // Exit delegates
10     FOR_EACH(del, visitor_delegate, {
11         it->exitHead(p);
```

```
12      });
13  }
```

Having both visit and exit functions allow for great flexibility, and so far proven to be all that is needed, for the development of visitor components thus far. For example using exit functions only allows a component to traverse the tree bottom-up. Using both visit and exit functions were helpful when creating scopes in the symbol gatherer.

The components using the `ast_visitor` only need define functions for the nodes, that it actually needs to do work on. Making it easy to create small components that works on a very specific piece of the tree. Giving a list of delegates, instead of just one, then allows us to easily define the components of a single pass. One example of this, is the following snippet, which sets up part of a single pass:

```
1  linked_list *pass0Delegates = makeLinkedList();
2  addElement(pass0Delegates, makeReturnCheckDelegate());
3  addElement(pass0Delegates, makeFunctionCheckDelegate());
4  addElement(pass0Delegates, makeDivByZeroDelegate());
5  // ...
```

4.2 Components of the Weeder

In this section, the components that are part of the weeder are described. The weeder is a convenience phase. It allows us a compiler programmers, to clean-up in the AST, and perform some extra error-checks, that aren't directly related to the type phase, and too complex for the parser to handle.

The actual weeding phase is done in a single recursive traversal of the AST using the visitor pattern mentioned in the previous section.

4.2.1 Function Check

In Diego a function is declared using the following syntax:

```
1  func foobar(): bool
2      return true;
3  end foobar
```

The purpose of this component is to ensure that the identifier used in the `HEAD` node (`foobar`) matches the one in the `TAIL`. The design choices in this step lies in where to implement the logic that enforces this. It could also have been implemented straight into the grammar.

Instead this was moved to the weeding phase, in order to keep the different compiler phases as modular as possible. This keeps maintainability and readability of the compiler high, and helps in any further development on this project.

4.2.2 Return Check

In the language all functions are required to return some value. The exception to this rule is the global root function, which isn't allowed to return. The purpose of this component is to ensure that these constraints are enforced.

It is not practical to check if a function will always provide a return value at run-time. The reason for this is, that it requires us to know exactly what will happen at run-time. Consider for example the following program:

```
1 var a: int;
2 a = 0;
3
4 while true {
5     if (a == 10) then return true;
6     a = a + 1;
7 }
```

Listing 2: A Diego program that will always return, although hard to validate

This program will always return, but it requires to know that it will reach the correct condition. For this reason a more conservative approach is used. A block is considered to provide a return value, if it holds any of the following conditions:

1. The block ends with a `return` statement
2. The block consists of an `if` and an `else` branch, that both provide a `return` statement (recursively checked).

While this approach will reject some programs, which actually would run, it should not cause any problems when writing programs.

4.2.3 Division by Zero Check

Division by zero is undefined. The purpose of this component is to check that, in the simplest of cases, no division by zero is made.

This component looks at all expressions, which performs a division. If the right hand side of the expression is a literal 0, then an appropriate error code is returned. This could be expanded to also deal with more advanced cases. Such as using a constant variables set to 0. Although this would currently require an expansion in the language,

to allow for such constants. Alternatively some kind of analysis would be required, to check that a variable is guaranteed to be 0, but this would require a lot of work for very little payoff.

4.2.4 Sorting Records by Name

The language should allow for the following situation:

```
type my_type1_t = record of { a: bool, b: int };
type my_type2_t = record of { b: int, a: bool };

var my_var_1: my_type1_t;
var my_var_2: my_type2_t;
# Allocate the records
my_var_1 = my_var_2;
```

That is the language should consider the order of fields in a record to be irrelevant. The only thing that should matter to the typechecker is that the two records, contain exactly the same fields, of the same type.

From this an interesting issue arises. How should records by laid out in memory. In C, a struct is laid out in memory, following the order in which the fields are defined. This becomes problematic when generating code for actually accessing these records. Because if `my_var_1` previously was laid on in memory like this:

```
false, 42
```

Then the code generator would, likely, assume, from its type, that the fields would always be in this order. But after the assignment to `my_var_2`, it would look like this:

```
1337, true
```

To deal with this issue, it has been decided that records are laid out in memory, sorted on the field's name, in this case simply implemented by using `strcmp` from the standard library. Having the fields sorted, also made it easier to implement typechecking between record types.

4.2.5 Constant Folding

The purpose of the constant folding is to evaluate expressions consisting of literal constants, at compile-time as opposed to run-time. This will save a small amount of time.

This means that if a constant is present in both expressions in the case of the following, it will be replaced by a single expression instead.

```
constant OP constant
```

This is placed in the weeder, to make it run before typecheckes, such that a simpler/smaller AST is worked on. The implementation simply looks at all expressions, and if it can be evaluated early, then it is. The result of this is that the previous children of this expression are deleted, and expression node is replaced with a new literal, containing the actual result. This is done in a bottom-up fashion, such that all the literals that can be folded, are folded.

This step required additional error handling as performing operations at this level could leads to overflow values. Example would be addition of two integers, who when added together would be bigger than the value an integer can hold. This would lead to overflow and therefore changing the sign-bit[1] resulting in a completely different value.

It was chosen to simply warn the user of this at compile time, instead of halting the execution of the compiler.

Additionally boolean expressions are folded to a single node, much in the same manner as integer folding.

[1]Since we are saving the values as a signed integer, had it been unsigned the value would actually instead be reduced to zero and incremented from there.

Chapter 5

Type System

5.1 Introduction to Diego's Type System

At the core of Diego's type system is a rather traditional type system, as would be seen in for example C.

5.1.1 Simple Types

Diego supports several simple types, these are:

- **Integers.** A simple integer type useful for computations
- **Booleans.** A simple boolean type used primarily for logic
- **Character.** A simple character type, representing a single ASCII character.

Variables of simple types are initial assigned using literals, and can typically be changed using several of the built-in operators. For an example of this see listing 3.

```
1 var a: int;
2 var b: bool;
3 var c: string;
4
5 a = 10;
6 a = a + 10 - 5 % 3;
7 b = false;
8 b = b && true || false;
9 c = "Hello world!";
```

Listing 3: Creating and using variables of simple types

5.1.2 Heap Allocated Types

Diego also supports two types, that are allocated on the heap.

The first which are arrays. Arrays are fixed-size and allocated in a contiguous memory segment on the heap. In general they function much like arrays does in C. Usage of arrays and how they are stored can be seen in listing 4.

```
1 var myArray: array of int;
2 allocate myArray of length 100;
3
4 myArray[0] = 42;
5 myArray[2] = 10;
6 myArray[3] = 1337;
```

Listing 4: How arrays are created and used.

Note that the elements not defined and undefined contents. This is different from how, many high-level languages does it. It languages like Java, for example, each element is initialized to some default element. In this language, however, no elements are initialized, and are simply kept at whatever bit pattern was present at that location. The main reason for doing this is performance. If the programmer requires default values, then these could simply be filled by the programmer.

The second type are “records”. A comparable concept in C would be structs. A record consists of several fields, each field having a name and a type. An example of usage and storage can be seen in 5.

```
1 var myRecord: { b: bool, c: int, a: char };
2
3 allocate myRecord;
4
5 myRecord.a = 'H';
6 myRecord.b = false;
7 myRecord.c = 42;
```

Listing 5: How records are created and used.

Note that both records and arrays share the same allocate syntax. The length is required for arrays and explicitly not allowed for records, and vice versa.

5.1.3 The `null` type

A generic null type, this is compatible with any heap allocated object, and is used to indicate the lack of a value.

5.1.4 Function type

The type of a function. Contains type information about its parameters and return type.

5.1.5 Type Definition

In addition to these primitive types, Diego also supports type definitions. Type definitions are created using the syntax as seen in 6.

```
1 type myType = ...;
```

Listing 6: The syntax for type definitions

On the right hand side of the definition can be any other type supported by the system. This allows for creating aliases for other types. Most importantly of all type definitions allow for recursive records. As without type definitions, it would be impossible to refer back to itself. For example a simple linked list implementation could be created with the snippet seen in 7.

```
1 type linked_list = record of { data: int, next: linked_list };
```

Listing 7: A simple linked list record in Diego

5.2 Type Compatibility

In this section a brief overview of which types are compatible will be introduced. This will be used by the type-checker to determine if a program is type-correct by this definition. A quick overview of type compatibility in Diego can be seen in table 5.1.

Simple types are only compatible with themselves. It is perhaps however noteworthy to mention that this language does not coerce integers into booleans or vice versa. This is done to make it more clear, what each variable's intention is. We believe that this decision makes programs more readable.

Table 5.1: Type Compatibility in Diego

	int	bool	char	array	record	type definition	null
int	X					(3)	
bool		X				(3)	
char			X			(3)	
array				if types match (1)		(3)	X
record					if all types match (2)	(3)	X
type definition	(3)	(3)	(3)	(3)	(3)	(3)	(3)
null						(3)	X

For heap allocated objects, it should be noted that they are compatible with `null`. This does not go the other way, since it does not make sense within the system to assign `null` to anything.

(1): Two arrays are only compatible iff their element's types are also compatible.

(2): As mentioned in the weeder section, records are in this version of Diego considered sets. That is two records, A and B are considered equal iff the following holds:

1. The number of fields in A is equal to the number of fields in B
2. Both records contain the same set of names
3. If f is a field of A and B, then $type(A.f) = type(B.f)$.

Note that this means that the order of which the fields are declares, is irrelevant.

(3): Two type definitions are only compatible if they originate from the same declaration. This is perhaps the most surprising part of this type system. We will now take a closer look at what this means.

First of all this means that two typedefs that are typed to the same type are not equal. Consider for example the program shown in listing 8.

```
1 type first_t = int;
2 type second_t = int;
3
4 var a: first_t;
5 var b: second_t;
6
7 // ...
8
9 a = b; // Not allowed!
```

Listing 8: `first_t` and `second_t` are not compatible!

This might at first glance seem like a poor design choice. The rationale behind this

is choice, is to allow the programmer to create types, that are compatible solely with themselves. This makes the type system more strict and can eliminate bugs, where two types might be internally compatible, but they are designed to be interpreted in different ways. Take for example the program shown in listing 9. In this example there are two typedefs, that represent the two units: miles and kilometers. In both cases it makes sense to represent them using some numeric type. However, it can be dangerous for functions, designed to work with kilometers to compile even if it is given a variable of type miles. NASA for example learned this the hard way, on the Mars Climate Orbiter[1] mission. This mission failed due to a similar bug, resulting from mixing different kinds of units. Total mission cost were around $327.6 million USD.

```
1  type miles_t = int;
2  type kilometers_t = int;
3
4  var someMiles: miles_t;
5  var someKilometers: kilometers_t;
6
7  func performCalculation(param: miles_t): ...
8      ...
9  end performCalculation
10
11 result = performCalculation(someMiles); // Will compile
12 result = performCalculation(someKilometers); // Won't compile!
```

Listing 9: Performing calculations with two radically different types, that internally look alike.

Finally, there are some complications with assigning typedefs primitive values, and assigning typedefs to their primitive variant. To deal with this, we will attempt to resolve the typedef, and then compare the resolved type with the other type. This resolving will only occur if one of the two are typedefs. This resolving will be applied recursively, thus allowing typedefs to be hierarchical. To ensure that the resolve algorithm terminates, we will have to guard against empty types. Empty types are shown in listing 10.

```
1  type a = b;
2  type b = c;
3  type c = a;
```

Listing 10: Empty types should not be legal in Diego

To guard against these, we simply keep track of all types met during the resolving. If we attempt to resolve some type already resolved, we can be certain that we're in a cycle,

[1] http://en.wikipedia.org/wiki/Mars_Climate_Orbiter

and thus have detected an empty type.

5.3 Scoping Rules

The scoping rules of this language are inspired by the ones used in JavaScript. Every Diego program has a global scope. This is the scope in which the main program may be written.

New scopes are then created for functions and for records. These scopes may then be further nested as new functions and records are declared inside of these.

The system will search the scopes, starting at the current scope, moving upwards, until it reaches the root scope. If the symbol is found in any of the scopes, then the search can stop. Note that the system will look in the entire scope, even in the declarations that are done at a later point in the code. That is the compiler supports forward references.

Chapter 6

Type-checker

Type-checking is structurally organized in three different phases: Symbol gathering, type resolving, and finally type-checking.

6.1 Gathering Symbols (for the Symbol Table)

To perform the type-check, we must first have an idea of which symbols are known, and to which scope they belong. To store this information we will use a special data-structure, a symbol table. The symbol table can be thought of as a tree of tables. Each table represents a single scope in our Diego program. This table will contain a mapping between a symbol identifier and its corresponding type. Listing 11 shows an example Diego program and the corresponding table for that scope.

It should be noted that the variables inside of `hello` are not included. Instead these live in another table (scope). This scope then points back up to this symbol table (its parent), eventually creating a tree like structure.

It should also be noted that function parameters are visible both from the outer scope, and from within the function body's scope. This were for example needed to validate function calls.

It should also be noted that we decided to create new scopes for records. Where there exists a symbol for every field the record contains. This was needed to perform validation on field lookup.

6.1.1 Implementing the Symbol table

As a basis for our implementation we will be using a linked hash table. They provide an efficient implementation for the dictionary mapping needed for representing a single

```
+-----------------------------------+ +-------------------------------+
  Diego Program                       Outer Symbol table (A)
+-----------------------------------+ +-------------------------------+
var a: int, b: bool, c: array of int;   a: int
func hello(p: int): bool                b: bool
    var helloA: int:                    c: array of int
    #// Function body                  hello: function(p: int) -> bool
end hello
                                      +-------------------------------+
                                        Inner Symbol table (B)
                                      +-------------------------------+
                                        p: int
                                        helloA: int
```

Listing 11: To the left: A Diego program. To the right: The symbol table for the outer scope

scope. To use these as symbol tables, several modification were required to including the scoping rules that the system has.

To implement the tree structure, we let each table point to its parent. We chose to point back to parents, as opposed to pointing to children, for an important reason. When we must search for a symbol in our program, we are given an identifier, and a scope. The scoping rules state that we must start at this scope, and start moving up through the parents, until the symbol is either found, or there are no more scopes to search in.

6.1.2 Collecting Symbols

Collecting the symbols is done using an AST traversal, using the same delegate mechanism as described earlier. The reader should also recall that the visitor goes through the nodes in a depth-first fashion.

It is the job of the component to collect all symbol declarations. These come from the nodes, thus the visitor component simply needs to be configured, such that it inserts these symbol definition on the relevant nodes.

For the component to know where to insert the symbol, we must keep track on which scope it belongs to. When the component starts, it creates a root scope. New scopes can be made, when an appropriate node is visited. This scope will then point back to its parent. Whenever the visitor exits that node, we may leave and go back to the parent scope.

6.1.3 Performing the Type-check

The third phase of the type-checker involves checking whether the expressions are of the correct type, in the situation they are used. This is done using another iteration of the recursive run-trough of the abstract syntax tree.

Function calls

When performing function calls in the language, there are several things that the type-checker needs to confirm:

- Make sure the function is actually a function.
- Check the number of parameters match.
- Type-check each parameter.

Each item needs to hold or the compiler rejects the program and informs the user of the error, and the line number it was detected on.

Checking whether function head and tail id matches, is handled in the weeding phase.

Arithmetic Operators

When doing arithmetic, it should be checked m whether both operators are integers. Using any of the following symbols `(+, -, *, /, %)` should only be possible when and integer is present on both sides of the expressions.

At a later point in time, this could be expanded to allow for new types, or even allow full operator overloading.

Logic Operators

Likewise to arithmetic operators, logic operators need to be performed on a correct set of symbols. The type-checker requires that both operands of logic operators evaluate to boolean types. No implicit casting from any other type to `bool` is made. In the future such casts could be made. For example an implicit cast from record types to a `null`-check could be made. But at the moment no such implicit casts are made.

Statements

Write: The output for write[1] must be legal, currently this is restricted to `int, bool`, and `char`.

Flow Control (While, If, Else): The expression, controlling if/while statements, must evaluate to a boolean type.

Return: The type returned in a function must match the return type specified in the function head.

Alloc (of length): When including the optional "of length" expression, only an integer is permitted. Allocations are only allowed on records and arrays. Records require that the of length expression is not included, arrays require that it is included.

Absolute Operator

The "absolute operator" (that is `|expression|`) is defined for both integers, where it will take the absolute value of the expression. It is also defined for arrays, where it will return the length of the array. The typechecker will reject any expression not of one of these types.

Dealing with Typedefs

As the type system indicates, we deal with type definitions in a slightly complex way. When two types are compared, and they are both typedefs then they should be compared by pointer equivalence. However, if one of the types are primitive, then we should check to see if the primitive type is compatible with the RHS of the type definition.

[1] Write in the Diego language works as a print function.

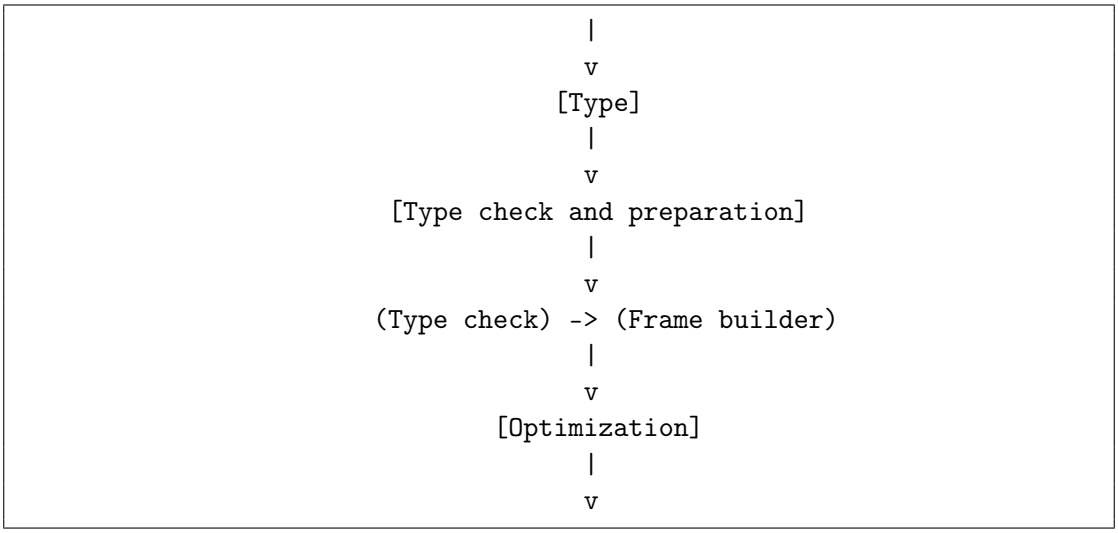
Chapter 7

Code Generation

7.1 Preparing for Code Generation

In this section we will discuss how the compiler prepares itself for the code generation phase. From having just finished the type-checking phase, and now needs to being the first stage of actual code generation.

We will return to looking at the phases of our compiler:



Listing 12: The phases of the compiler, focused on preparation for code generation

7.1.1 Frame Building

For this part of our compiler, we can consider the memory as being split up into two parts: the heap, and the stack.

The stack may consist of several stack frames, where as each frame represents a function in the Diego code. This frame holds information we want to store about each function, so we can find parameters along with any local variables we might need when working with them. The generated frame is stored in functions in the AST.

A frame consists of a name, list of associated temporaries, and the current offset from the base pointer where the latest temporary is to be stored.

To summarize, the frame builder must:

1. **Create a stack frame for each function.**
2. **Insert all in-going parameters as part of the frame.**
3. **Allocate space for variables.**
4. **Prepare temporaries.**

In order to maintain the stack, whenever we visit a new function, we store the base pointer for the previous stack and move the stack pointer into the the base pointer to resemble the beginning of the new stack.

Upon leaving a function we then restore the base pointer to the old value. Both these cases are handled in addFunctionPrologue and addFunctionEpilogue in `ir_emit.c`.

Using this technique we can always refer to parameters, local variables, or the static link at an specific offset from the base pointer.

7.2 Introduction to the Internal IR Representation

In the initial code generation phase, code is not generated directly to machine language. It is instead generated to an intermediate representation (IR). We will in this section discuss the important design choices made with respect to representation of IR code.

When generating code, we do not wish to emit machine language directly. Instead we want to use some intermediate representation, which is an abstraction over a generalized machine language. This makes it possible for a compiler to support multiple backends (architectures) as well as several different frontends (programming languages).

IR-code is traditionally represented as a tree. This tree can be created either by directly translating the AST, but could also be implemented by creating a new IR-programming language. It is then the job of the backend to translate this tree into machine language.

In this compiler a more direct approach has been taken. The primary reason behind this choice were the time constraints of the project.

Diego internally generates code to a form of structured x86 assembler (GAS syntax). These instructions are then kept in a linked list. Figure 7.1 presents how these instructions are structured.

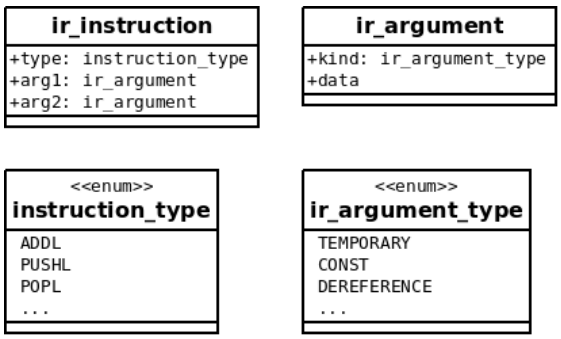


Figure 7.1: Class diagram of the structured assembler used in Diego

At the core of this system we have the type `ir_instruction`. These represents a typical x86 instruction, it has a type, which is the actual instruction, as well as up to two operands.

The arguments can be of different types. The need for these are quite obvious, the syntax for passing a dereference and a simple constant are very different, and thus needs to be modelled. The arguments do however also serve yet another purpose.

The temporary kind is an abstraction over all kinds of memory access. A "temporary" can thus be thought of as a register-type, in our imaginary machine, that runs our IR-code. This machine has infinitely many temporaries available. This means that when performing code-generation, we do not have to deal with actual register allocation, we may instead post-pone this to a later phase, this greatly reduces the complexity of the code required.

7.3 Generating IR Code

In this section the overall algorithms along with some strategies for generating IR code will be discussed.

The Diego compiler generates code primarily using a simple AST traversal, as explained in previous sections. Code is generated in a 1:1 fashion, that is every statement in our program has a corresponding IR-code representation.

The relative simplicity of the Diego language, allows us to generate code in a straight forward fashion. Listing **??** shows the overall structure of our generated IR-code.

```
.data
"Standard data required by Diego"
"Potential constants and other static data created by the program"

.text
func1:
"Code for function 1"

"..."

funcn:
"Code for function n"

.globl main
main:
"Initialization required by all Diego programs"
"Code for root scope"
```

Listing 13: The overall structure of the IR-code

For the data section, we will have to collect this data, in some previous phase. Depending on the information this needs, it can be done either when calculating resources, or in the weeder phase. At the start of code generation, this data can be filled in.

The AST traversal will need to generate entry points for each function, other than that it will simply need to visit and generate code for every statement.

To generate code for a single statement, we will first determine which kind of statement it is. Lets for example assume, that we're working with the write statement. Going back in time, our grammatical rule for write statements look like this:

```
T_WRITE expression T_SCOLON // For example could be: "write 2 * a + 14;"
```

The expression represents what we need to write to the terminal. Thus before we can begin printing anything, we will first need to calculate the result of the expression. We can do this by making a recursive call, to generate code for a single expression. The result of this expression, will then be made available in the temporary, that we allocated during frame-building.

When inspecting the contents of an expression node, we quickly see that these themselves may contain more expressions or terms. Terms may also contain references to even more nodes. Thus it is clear, that we cannot just in the body of the write statement, generate code for an entire expression at the time. We will instead make recursive calls, that generate the relevant code from a single AST node. We will continue with putting the results of a single node, in the relevant temporaries, allocated to us in the frame-building

phase.

Before going into the code templates required, we will make a quick de-tour. To explain how constructing this assembler internally, was made easier.

7.4 Making Code Generation Easier

The structured IR-code makes it very easy to work with. As we won't have any parsing to do on our IR-code, to make any sense of it.

Writing code for IR quickly became hard to understand, verbose to write, and as a result harder to maintain. This is nicely illustrated in listing 14, which is a translation of the assembler shown in 15.

```
temporary *t1;

ir_instruction *i1 = makeInstruction(MOVL, makeConstArgument(100),
    makeTempArgument(t1));
ir_instruction *i2 = makeInstruction(ADDL,
    makeTempArgument(makePhysicalRegisterTemporary(REG_EAX)),
    makeTempARgument(t1));
ir_instruction *i3 = makeInstruction(MOVL,
    makeTempArgument(t1),
    makeDereferenceArgument(0, makePhysicalRegisterTermpory(REG_EBX)));

addElement(instructions, i1);
addElement(instructions, i2);
addElement(instructions, i3);
```

.

Listing 14: C code required for codegen. This generates code for a very small sample template

```
MOVL 100, T1  // T1 being a temporary
ADDL %EAX, T1
MOVL T1, (%EBX)
```

Listing 15: A small sample code template, in pseudo-assembler

While this code can certainly be read, and isn't too hard to write, it still contains a lot of information which feels relatively redundant. To fix this problem, we created a few macros, whose main purpose was to create something that felt more like writing pseudo-assembler, which is what our IR-code essentially is. We won't go into the implementation

details, but simply refer to the source code in `emit/ir_emit.h`, which can be found in the appendix.

With these macros we managed to turn listing 14 into what is shown in 16.

```
1 ADD_INSTR(MOVL, CONST(100), makeTempArgument(t1));
2 ADD_INSTR(ADDL, REG(EAX), makeTempArgument(t1));
3 ADD_INSTR(MOVL, makeTempArgument(t1), DEREF(0, REG_TEMP(EBX)));
```

Listing 16: C code required for codegen, with our small framework added.

7.5 The Calling Convention of Diego

When functions are called, there must be a convention in place as to where arguments are stored, and how to bring the program back to consistent state after a function has finished executing. It is important that both the calling function and the function being called both agree on which convention to use, as we otherwise cannot make any guarantees that everything comes back consistently, or that the correct data is being used.

The calling convention used in Diego is heavily inspired by the one used in C. We have chosen this convention for several reasons. The convention is very familiar, and has been good enough for C. Secondly we will need to interface several times with C, for example to make calls to write to standard out. We will now present the calling convention used, and highlight the differences.

At the calling site:

1. Must save caller registers. The registers should, but don't have to, be saved on the stack. The registers to be saved are: `ECX, EDX, EDI, ESI`. This also means that the function being called, cannot rely on these being present on the stack. They are always generated, but later removed in the peep-hole phase if not needed.
2. Include all function arguments on the stack. The arguments are pushed on the stack in a right-to-left manner. As a result argument 1 will be closest to the base pointer.
3. Include the static link. This is not a part of the C calling convention. When interfacing with foreign code this should not be included.
4. Once the called function has returned, the caller registers should be restored if saved.

At the receiving site:

1. Must saver all callee registers. Same procedure as with the caller registers. The registers to be saved are: `EBX, ESI, EDI`.

2. Before returning, the callee registers must be restored.

7.6 Static Links

As an introduction to static links, we will first present the following, legal, Diego code:

```
1  var a: int, _: bool;
2
3  func fA(): bool
4    var b: int;
5    func fB(): bool
6      b = a + 10;
7      return fC();
8    end fB
9    func fC(): bool
10     return false;
11   end fC
12   a = 32;
13   return fB();
14 end fA
15
16 _ = fA();
```

The reader should note the use of nested-functions, and the use of variables in different scopes. Some of these variables, for example `b`, is residing on `fA`'s stack. To be able to access this from within `fB` we will need knowledge of where this stack is located. To solve this problem, we will be using static links.

A static link, is a pointer to the static link, residing in the stack frame, belonging to the enclosing scope. Thus both `fB` and `fC` would receive a pointer to the stack frame of `fA` that they belong to. This also means that `fA` would receive a static link to the root stack frame. As mentioned in the calling convention section, all functions have their static link passed as argument "0". To make this work with the same code, we have had to inject a static link into the root frame. This static link simply points back to itself. Thus if any site would require the enclosing frame of root, a pointer to the root static link would be passed.

When we need a pointer to a static link, we will need to find which stack frame it belongs to. To calculate this, we will use the symbol table, as it can tell us who the enclosing scope is, relative to the scope we're standing in. We can find this by counting the number of times we go to the parent scope, when searching for the symbol.

For example, when we call `fB`, we will see that we had to go to the parent scope 0 times. And when we access `b` from within `fB`, we have to access it 1 time.

These numbers represent exactly the number of times, we have to follow our own static link. Once we have followed the static link, we will be placed at the static link, of the frame we wish to access.

7.7 Memory Allocator

The memory allocation model in Diego is a very simple one. All local variables and parameters will live on the stack. There are however two types of objects (records and arrays) that will always live on the heap. It is not possible to put these on the stack, just as it is not possible to put any of the primitive types on the heap. This design choice has been made solely to simplify the memory management of Diego, allowing flexibility in where objects can be allocated, could have many benefits. It is for example not possible to ask another function to update an integer, unless it is wrapped in a record. Similarly if records and arrays were allowed to live on the stack, this this would greatly simplify cleaning-up, since this is automatically handled when a function returns.

The heap is currently implemented as a fixed size chunk of memory, which is allocated at start-up. The allocator contains 3 global variables that keep track of the state of the heap. None of these global variables can be accessed from within a Diego program. The variables are:

- `heapNext`: Contains a pointer to where memory should be allocated next.
- `heapLimit`: Contains a pointer to the end of the heap.
- `heapRemaining`: The remaining memory left in the heap (stored in bytes). This is defined as `heapLimit - heapNext`, it is used solely to simplify some of the error checking code.

The size of the heap is managed using the UNIX system call `sbrk`. This call takes a single integer parameter, and expands the program "break" with that many bytes, and returns a pointer to the previous program break. The program break is essentially the end of the program's data-segment, thus increasing it, will increase the amount of memory our program is allowed to access.

At the start of every Diego program, code is generated, which initializes all the global variables, and increases the program break. This is shown in listing 17. Lines 1-2 initializes `heapNext` by retrieving the pointer to the start of the pool, by calling `sbrk(0)`. Which will expand the break by 0 bytes, and thus return a pointer to the start of the segment. The rest of the code simply expands the program break, and initializes all the global variables.

Every object that is allocated to the heap contains a small piece of memory reserved for meta-data. This is intended to be used by sub-routines, that will work the internals of a Diego program. This can have many different purposes. If an OO model were to be

```
generateSystemCall("sbrk", 1, CONST(0));
ADD_INSTR(MOVL, REG(EAX), RAW("heapNext"));
generateSystemCall("sbrk", 1, CONST(DEFAULT_HEAP_SIZE));
ADD_INSTR(ADDL, CONST(DEFAULT_HEAP_SIZE), REG(EAX));
ADD_INSTR(MOVL, REG(EAX), RAW("heapLimit"));
ADD_INSTR(MOVL, CONST(DEFAULT_HEAP_SIZE), RAW("heapRemaining"));
```

Listing 17: Initializing the program's heap

introduced into Diego, for example, then this segment might also include information about its own type. This segment could also be used to expose additional information to a debugger.

We currently store the length of all objects in bytes. This information is used to calculate the length of arrays, and would also prove essential to almost any GC[1] system that could run on top of this.

Even though the heap doesn't dynamically expand, this could easily be added in. The system currently does checks, to see if it has run out of memory. Instead of always failing with an error code, we could try to expand the heap size first.

7.8 Code Templates

This section covers over some of the interesting parts of the code templates used in generation. We will highlight any interesting design choices, and try to exemplify how the compiler actually generates IR code.

7.8.1 Notation Used for Code Templates in this Report

At the beginning of every code template, we will provide a quick reminder of the associated grammar, and a specification of how this should work. This is followed by a brief discussion on the implementation details. Any extensions to the language that may have been made are left out of this discussion, to focus on the core implementation details.

The code templates will use a pseudo-assembler like syntax. It will occasionally refer to other sub-routines that have been discussed.

`body`: The current function body, which we're generating code for. Is never `NULL`.

[1] Garbage Collection

7.8.2 Statements

Statements are the very heart of our code generation. These control almost all of the code being generated. We will go through the more interesting ones.

Returning Values

```
return <expression>;
```

Places the result of `<expression>` into `%EAX` and jumps to the end of the function.

This was achieved using the following template:

```
GENERATE expression INTO expression.temp
MOVL expression.temp, %EAX
JMP body.endLabel
```

This is very straight forward. The contents of the expression is calculated, and its result is put into a temporary. This result is then moved into the return register, and we jump to the end of our function.

Write Statements and System Calls

```
write <expression>;
```

Writes the contents of an expression to standard out, in an appropriate format.

To write to standard out, we will have to make some system call. For ease of implementation we have chosen to simply depend on the C standard library. Should we ever want to minimize dependencies on the C library, then we could call the appropiate system calls directly. Thus the code generated for write statements become relatively simple:

```
1 GENERATE_SYSTEM_CALL(printf, formatForType(expression.type),
2   expression.temp)
```

Making calls to call the standard C library, requires us to follow the C calling convention. Luckily this is almost what the Diego language already does. One notable exception however is that Diego has static links, which C does not. For that reason we do not have to include this.

Allocation Statements

```
allocate <variable> [of length <expression>];
```

Allocates an object on the heap. The supported object types are records and arrays. If the length is given, then the object must be an array, this will be the length of the array (in elements), otherwise it is a record. A pointer to the beginning of the newly created heap object, is placed in the variable.

It is quite clear that the implementation of this statement will branch out a bit. The overall implementation of the allocate statement will however look like this:

```
1 CALCULATE total size OF heap object
2 ALLOCATE total size HEAP MEMORY INTO stmt.alloc_temp
3 ASSIGN variable TO temp
```

Arrays: The size of an array is potentially not known until run-time. For this reason we will always calculate the size of the array. Optimizations such as constant propagation could then later eliminate the code, if the size of the array is constant, and thus known at compile-time.

The number of elements, is trivially calculated from the expression. The size of each element is always 4 bytes, thus we must simply multiply the result of this with 4.

Records: In Diego, the sizes of records are always known at compile-time. This is mainly due to the fact that all fields that can be stored inside a record will be of a fixed 4 bytes. One might think that arrays would be larger. However in Diego arrays are stored in the records only by pointer, and not directly in the record.

Thus we will need to lookup the fields of the record. This information is stored on the variable's type. Due to the work done by the type-phase, this can accessed directly, and move the appropriate constant directly into the size temporary.

If statements

```
if <expression> then <statement1> [else <statement2>]
```

Evaluates `<expression>`, if `true` then `<statement1>` is executed. If the `<expression>` evaluates to false, and the else block is present, execute `<statement2>`. If the else block is not present jump to the end of statement.

To implement an if-statement, we will need several labels to jump to. First jumping to the else block (`L1`), and the second label jumping to the end of the entire statement (`L2`).

```
1  GENERATE expression INTO expression.temp
2  CMPL true, expression.temp
3  JNE L1
4
5  GENERATE statement1
6  JMP L2
7
8  INSERT L1
9  GENERATE statement2
10
11 INSERT L2
```

Lines 1-3 deviate slightly from how if-statements are traditionally implemented. To see how, consider the sample program shown in listing 18.

```
1  if a != 20 then (* body *)
```

Listing 18: A simple program using an if-statement

A straightforward translation is shown in listing 19.

```
1  CMPL 20, a
2  JE L1
```

Listing 19: A simple program using an if-statement

This should be contrasted against our way of always generating a boolean value from the expression. Then performing an extra load, with the constant true. Followed by yet another compare.

This means, that whenever the comparison consists of a single comparison, we're wasting several instructions worth of unnecessary work. This work is however only wasted, because all the work is already done by the comparison itself. While we have not done any testing, to see the cost of this, or any implementation, this could be a clear candidate for further optimization.

We have chosen our approach, as it for all cases will generate valid code. Any optimization that can be done, should instead be postponed for a later phase, the important thing in the initial code generation is valid and predictable code. Predictable code makes it easier to predict which templates that may generate non-optimal code.

Also worth noting, is that the jump on line 6, will in the case of an empty block produce a useless jump to the next instruction. This pattern is caught by one of the peephole patterns in the optimization phase.

While Loops

```
while <expression> do <statement>
```

Evaluates `<expression>`, if it is `true` then execute `<statement>`, and repeat. If it evaluates to `false` then jump to the end of the while-statement.

Again the implementation will require two labels. The first label will be put at the start of the while-loop (`L1`). The second label will be put directly after the while-loop (`L2`).

```
INSERT L1
GENERATE expression INTO expression.temp
CMPL true, expr.temp
JNE L2
GENERATE statement
JMP L1
INSERT L2
```

The code template produces a very traditional loop. Not many design choices to discuss about the while-loop. The statements made about the if-statement, and conditional jumping however still holds for the while-loop.

7.8.3 Expressions

Expressions always result in some value, which should be calculated into some temporary, specified by the requesting sub-routine. We will in this section refer to this temporary as `out`.

Comparisons (`<`, `>`, `<=`, `>=`, `==`)

```
<left_expression> <op> <right_expression>
```

Where `<op>` is any of the comparison operators, for example "==".

Compares two expressions of compatible types using one of the operators.

```
1 GENERATE left- and right_expression INTO their respective temporaries
2 CMPL right_expression.temp, left_expression.temp
3 SETX %AL // least-significant byte of %EAX
4 MOVL %AL, out
```

To implement the boolean expressions we use the `SETX` instruction. This is a family of instructions that sets a register, depending on the result of some comparison. This family of instruction works much like the `JMP` family. Thus for example `SETE` will set the register to `true` iff the comparison indicated that the objects were equal.

As has been mentioned before, in Diego all types are of 4 bytes. This includes booleans, and means that booleans are 32-bits wide. This clashes somewhat with the instructions available in our target architecture, as all the `SETX` instructions work only on a single byte. It should also be noted that the `SETX` family only works on register that are a single byte wide.

To get around this issue, we will have to change our template slightly, as we cannot just move a single byte into our 4-byte location. The changes are shown below:

```
1 MOVL 0, %EAX                                                          (1)
2 GENERATE left- and right_expression INTO their respective temporaries
3 CMPL right_expression.temp, left_expression.temp
4 SETX %AL // least-significant byte of %EAX
5 MOVL %EAX, out                                                        (2)
```

In (1) we start out by clearing the contents of `EAX`. This is done to make sure that the register is completely empty. In (2) we know move `EAX` into `out` as opposed to only the least-significant byte of `EAX`.

Boolean `AND` and `OR` (`&&`, `||`)

```
<left_expression> <op> <right_expression>
```

Where `<op>` is any of the allowed operations, for example "`||`".

Performs a boolean operation. First the left expression is evaluated, then the right. If the result of the expression can be determined after only evaluating the left, then the right hand side won't be evaluated (short-circuiting). For example, if our operator is the boolean or. Then if the left expression evaluates to true, then we won't need to evaluate the right hand side, as we already know the result of the expression will be `true`.

To implement short-circuiting, we will need a bit of branching, and labels. The first label (`L_SC`) will be used in case of a short-circuit. The second label (`L_END`) will indicate the end of the expression, and can be used to skip the short-circuit section. We will refer to the instruction corresponding to the operator as `<or>`. Also the value indicating a short-circuit will be called `sc_val` (true for `||` and false for `&&`).

```
1 GENERATE left_expression INTO left_expression.temp
2 CMPL sc_val, left_expression.temp
3 JE L_SC
```

```
4 MOVL right, out
5 <op> left, out
6 JMP L_END
7 INSERT J_SC
8 MOVL sc_val, out
9 INSERT L_END
```

Lines 1-3 is used to evaluate the left expression, and determine if a short-circuit should occur. If execution reaches line 4, then no short-circuit has occurred, and it must evaluate the right side, and perform the actual operation. In line 6, it will jump past the short-circuit section. In case a short-circuit did happen, then in line 8, we will move the short-circuited value to the output temporary.

Finally we feel it is relevant to briefly touch on the performance of short-circuiting. To implement short-circuiting we have had to implement several jumps. Jumps have several performance impacts associated with them, for example branch-prediction. Thus if both the expressions are easy to evaluate (computationally) then we risk the extra jumps actual end up costing more. However if one of the two expressions are very expensive to evaluate, then we may save a lot of work.

Operations Related to Division (%, /)

```
<left_expression> <op> <right_expression>
```

Where <op> is any of the allowed operations, for example "/".

Performs the mathematical operation related to the operator. For "/" that would be integer division. For "%" that would the modulo operation.

To understand how these two operations are implemented, we must first take a look at how the integer division instruction, IDIVL, works.

Like one would imagine the instruction will need to take a dividend and a divisor. It takes as its first argument the divisor. The instruction expects the dividend to be placed in EDX:EAX. This notation means register concatenation. Thus the dividend's most significant bits will be placed in EDX and its least significant bits in EAX.

The instruction will then put the resulting remainder in EDX, and the resulting quotient in EAX.

```
1 GENERATE left- and right_expression INTO their respective temporaries
2 MOVL left_expression.temp, EAX
3 MOVL EAX, EDX
4 SARL 31, EDX
```

```
5 IDIVL right_expression.temp
6 MOVL output_register, out
```

Where the `output_register` is the register containing the information we want, as dictated by the operator.

Lines 2-4 moves the dividend into place, and performs what is known as sign extension. That is the operation of increasing the number of bits, while preserving the sign and value of the number.

Lines 5-6 performs the actual division, and moves the relevant information, from either `EDX` or `EAX` into the output temporary.

7.8.4 Terms

Terms, like expression, are given an output temporary by the requesting sub-routine. Again we will refer to this as `out`.

Primitive Values (booleans, integers, characters, null)

All primitive values are relatively easy to handle, and should simply move their value into the output temporary. We will now go through the specification for how each type is represented internally.

Booleans: Booleans are represented internally as either `0` for `false` or `1` for true. Stored as 32-bits.

Integers: Integers are stored as 32-bit signed integers.

Characters: Characters are stored as 32-bit signed integers, with a value corresponding to the ASCII value. This representation is rather wasteful, but allows us to make several simplifications in the code, as we do not have to handle any other size that 32-bits. This technically also gives us enough space to use other encodings that allow for more that just ASCII values. A perfect fit for this would be UTF-32, as it would allow more characters, while having exactly the same size requirements as the current Diego characters.

Null: Stored in a 32-bit value, always has the value `0`.

Negation

```
!<term>
```

Negates a boolean-term.

The relative simple representation of booleans, allows us to implement negation using `XOR`.

Parenthesis

```
(<expr>)
```

Parenthesis are used to determine precedence in the AST. Should simply evaluate to the result of the expression.

This is implemented simply by evaluating the expression, and putting the result in the output temporary. This will generate extra unnecessary moves, and should be caught in a later phase. Ideally these nodes could have been removed from the AST in an earlier phase.

Absolute Operator

Syntax:

```
|<expr>|
```

Specification:

Returns the absolute value of an integer-expression.

Implementation:

This feature was implemented in a special way, taking advantage of one of our language extensions. The Diego language ships with a standard library that is always included. One of the functions exposed in the standard library is the `abs` function doing exactly what this operator should do. Thus the absolute operator is implemented simply as a call to the `abs` function. This re-uses the same code, already available, to generate code for standard function calls.

One of the benefits of this is that the code for this operator will automatically benefit from all the optimizations in the later phase. If this were written directly in IR-code. Then there is a higher chance that this won't be caught, since this might not be as predictable as the other code templates, and as such not be caught by the optimizer.

One disadvantage to this approach is however that we know directly rely on the standard library always being present. This might not always be wanted. For example we might want a large standard library, being able to perform many different operating systems. If we then ever were to run Diego on bare metal (no OS), then we would likely have to discard this standard library, and write our own. This would then mean that either the absolute operator wouldn't work, or we would have to implement our own.

Array Length Operator

Syntax:

```
|<expr>|
```

(Shares the syntax with the absolute operator)

Specification:

Returns the length of an array expression.

Implementation:

According to the specification, the length of an array (in bytes) is stored at the start of the heap object. The length is stored in a signed 4 byte number. We know that the size of each element will be 4 bytes, as such we will need to access this field, and divide it by 4.

```
1 GENERATE expr INTO expr.temp
2 MOVL expr.temp, %EBX
3 MOVL (%EBX), %EBX
4 MOVL %EBX, out
5 MOVL 4, EBX
6 GENERATE DIVISION (out / %EBX) INTO out
```

Line 1, loads in the address of the array. Lines 2-3 then dereferences this address, and moves the length in bytes into out. Lines 4-6 then prepares and performs the division, needed to get the number of elements.

Function Calls

Syntax:

```
<functionIdentifier>(<expr1>, <expr2>, ..., <exprN>);
```

Specification:

Performs a call to a native Diego function, using the Diego calling convention. Function arguments are guaranteed to be evaluated left-to-right.

Implementation:

```
1 GENERATE CALLER SAVE
2 EVALUATE expr[1], ..., expr[N] INTO their respective temporaries
3 FOR p IN expr[N], expr[N-1], ..., expr[1] {
4   PUSHL p
```

```
5  }
6  PUSHL static_link
7  CALL function.begin_label
8  ADDL (param_count * 4 + 4), ESP
9  GENERATE CALLER RESTORE
10 MOVL EAX, out
```

Lines 1-5 do exactly as specified in the calling convention. It saves all the caller registers, evaluates the parameters in the correct order, and puts them on the stack.

The static link, which is pushed in line 6, is found using the technique described in the static link section.

Line 8-9 performs clean up. It removes everything pushed onto the stack in a single operation, by simply moving the stack pointer.

Specification for Variable Access

All variables access, should move the value stored in the variable to the given output temporary. If the corresponding object is an heap allocated object, then this should be a pointer to said object.

Whenever any variable access is generated, it will return an additional value, indicating if it contains a pointer to the actual value, or if it is the value. This will be needed to determined if extra dereferences are needed.

Raw Variables

Syntax:

```
<identifier>
```

Specification:

Follows the direct specification variable access.

Implementation:

Uses a static link to find the variable in the correct frame. An optimization has been made, such that if no static links are needed, then the variable is accessed directly from the frame.

Note that "frame" is used a loose sense, the variable might be allocated either in a register or on the stack.

Array Access

Syntax:

```
<var>[<expr>]
```

Specification:

Follows the specification for variable access on the object stored at the calculated index (given by `<expr>`) in the array.

Implementation:

```
1  GENERATE expr INTO expr.temp
2  IMUL 4, expr.temp
3
4  GENERATE VARIABLE ACCESS OF var INTO %EAX
5  IF %EAX IS A POINTER {
6    MOVL (%EAX), %EAX
7  }
8
9  ADDL expr.temp, %EAX
10 ADDL 4, %EAX
11 MOVL %EAX, out
```

Lines 1-2 generate the the offset into the array. Here we use the the constant size of all elements

Lines 4-7 generates the base address of the array. If the variable needs to be further dereferenced it is.

Finally the last two lines add the offset to the base address, and we arrive at the actual element.

This calculation could also have been done using `LEAL`, and would likely have been faster.

Record Access

Syntax:

```
<var>.<field_name>
```

Specification:

Follows the specification for variable access on the object stored in the field (given by `field_name`) in the record.

Implementation:

To figure out the offset into the record, we will have to lookup the actual type, and search for the field key. Assuming that the typechecker has done its job, we can be certain that this key will be found. Lets assume that this lookup has already taken place and the final offset into the record has been found, and put into a compiler variable called: `fieldOffset`

```
1 GENERATE VARIABLE ACCESS OF var INTO %EAX
2 IF %EAX IS A POINTER {
3   MOVL (%EAX), %EAX
4 }
5
6 ADDL fieldOffset, %EAX
7 MOVL %EAX, out
```

Works almost just like variable access, instead the field offset if calculated at compile-time, as opposed to run-time. This address calculations could also have been done using `LEAL` for a performance boost.

Chapter 8

Optimization

8.1 Register Allocation

Register allocation can be approached in several ways. The simplest approach would be to push everything onto the stack. We can also be more smart about this, and attempt to find a good allocation of temporaries into registers.

Liveness Analysis is a massive optimization factor that can be performed in the compiling process. Accessing registers is nearly instantaneous where as accessing the main memory (which the stack is a part of) is several factors slower.

The goal of the register allocation is to minimize the number of temporaries that gets spilled to main memory. To do this we must determine which registers are in use at a given point in the program. This analysis is called a liveness analysis.

The approach used in this compiler, is heavily inspired from the course book[1] with a few simplifications taken, such as no coalescing/spill priority.

8.1.1 Flow Control Graph

The flow control graphs, shows the different paths the compiled program can take (through the instructions). The following algorithm shows the approach taken to create

[1] Modern Compiler Implementation in C by Andrew W. Appel

a flow control graph:

```
Data: IR code as a linked list
Result: Control Flow Graph
while next IR instruction exists do
    if jump instruction then
        add edge to jump destination instruction;
        continue;
    end
    if jump compare OR call then
        add edge to jump/call destination instruction;
    end
    add edge to next instruction;
end
```

Algorithm 1: Flow Graph algorithm

For each instruction, an edge is added to the next instruction in the list given - if it is possible to fall through. The only case where we cannot fall though is in a pure jump statement, where we only add an edge to the destination of the jump. In the case of jump-on-compare or function call statements, we add an additional edge to the destination of the jump/call along with the already added edge to the next instruction in the list.

As described in the book[2], the flow graph is traverse in reverse when setting libe-in and live-out information, this greatly reduces the number of iterations needed to be done.

It was necessary to determine which instructions would define a temporary, and which ones was using a temporary. For most isntructions this was straightforward, as a MOVE instruction clearly is using one, and defining another. The table below shows how this was done. Note that a CALL instruction defines EAX, as it is always expected that a function call has a return value (enforced in the Diego language).

[2]Modern Compiler Implementation in C, by Andrew W. Appel

Instruction	Operator 1	Operator 2	Other
`STRING_DATA, LOCAL_DATA,` `COMM_DATA, GLOBAL,` `START_FRAME, END_FRAME,` `TRACE`			
`JMP, JNE, JE, JGE, JLE, JL, JG, JZ`			
`IDIVL`	Use	Use	Define EAX
`SARL`	Use		
`CMPL, IMUL, ANDL, ORL`	Use	Use	
`MOVL, SUB, ADDL, XORL, LEAL`	Use	Define	
`POPL, PUSHL, NEGL`	Use		
`CALL`			Define EAX
`RET`			

Table 8.1: Use and Define registration. Other column is pre-coloured nodes.

It was chosen to build the graph with each node specifying a single instruction, rather than an entire basic block[3]. The reason for this lies in the simplicity, and the fact that no apparent advantages other than runtime for the compiler itself could benefit us in using basic blocks.

This also allowed us to store the information for each node in the node itself, rather than using a void pointer to a data structure. While the data structure allows for more a modular graph structure, the graph itself is not going to be used outside liveness analysis, and the speed/simpleness offered in having direct access to the data rather than having to access a separate data structure was more fitting for this project.

The size of a node is a bit excessive as it was preferred to use the additional memory rather than doing destructive updates on the flow-graph when working in the interference graph.

8.1.2 Interference Graph

The interference graphs shows conflicts between two temporaries. If there is an edge between two temporaries, then they cannot be allocated to the same register.

The graph structure is used to create the interference edges between defining nodes and any temporaries which is live at the same point in the program. Adding an interference edge is done by cycling through all the nodes which defines a temporary, and adding an

[3]A basic block is a set of instructions that has no conditions or jump statements.

```
struct node {
        ir_instruction *instruction;
        linked_list *prev;
        linked_list *next;
        int key;
        int visited;
        temporary *def;
        temporary *use1;
        temporary *use2;
        linked_list *liveIn;
        linked_list *liveOut;
        linked_list *inteference;
        bool popped;
};
```

Listing 20: A single graph node.

edge whenever a conflict is there.

Data: Flow Control Graph
Result: Interference Graph
while next graph node exists **do**
  if instruction is a MOVE **then**
    add egde from define to each live-out temporary that is not the same as the define.
  else
    add egde from defined to all live-out temporaries.
  end
end

Algorithm 2: Adding interference edges

8.1.3 Allocating registers

With the previous two data-structures in place, we can now finally perform the actual allocation of registers onto temporaries. This is done using a simple graph-colouring algorithm.

The build phase, created the data-structured described in the earlier sections. We now move on to the remaining phases.

This section varies the most from the course book, in that a simpler approach was used. The flow can be described as:

```
build -> simplify -> select -> spill
```

It iterates the interference graph and pushing the node with the lowest degree of edges to the stack. The following shows the algorithm used:

```
Data: Inteference Graph
while node remains do
    Find node with smalles number of egdes;
    if temporary is escaped then
        Spill the temporary;
    end
    Push onto the stack;
end
while node remains do
    Pop newest element off stack;
    Assign color();
end
```

Algorithm 3: Allocating registers

A temporary is escaped if it is used outside the current scope. Spilling this was a design choice for easily handling that the value stored should be found outside the scope it was created, had it been assigned a register and then used within a nested function, the value would not be saved at a fixed point like when saved in the stack frame. It also wouldn't be present in the register, as liveness-information never carries across function calls.

Assigning the colour to a specific temporary is described below.

```
Data: Given temporary
for each register enabled do
    for each conflicting edge do
        if conflict node is using proposed register then
            continue to next register;
        end
    end
    Assign register to temporary;
    return;
end
All in use: spill temporary;
```

Algorithm 4: Assigning colours to a specific temporary

Note that the algorithm above, will always try to allocate a register, even if the number of edges are bigger than the number of registers. This was done as some of the edges might be using the same registers, therefore it could still be possible to allocate the temporary. A register is only assigned if no conflicting temporary is using that register.

If a temporary is spilled, an offset from the frame pointer is set into the temporary and its type is changed to reflect the spill. The code emitter will then handle the proper code generation from this. Likewise if a register is assigned, the argument type is changed to

reflect a physical register and the register name is stored in the argument, the emitter will then handle generating the correct arguments to the instruction.

8.1.4 Performance and Testing

The performance gain has been measured running the knapsack problem[4] with the allocation process having 0 and 5 registers[5].

An average runtime with everything pushed to the stack yields an average runtime of 1m16.577s while with liveness enabled, the average runtime is 27.848s. This is a speed increase of 274% of the emitted code. Both tests had all run-time checks enabled.

Testing and debugging the liveness analysis very often comes down to an missing interference edge, as it is this information that is used to allocate registers. Therefore it is possible to printout the graph with the interference information available using the `--dumpgraph` flag. This can also confirm that the liveness analysis was performed correctly by doing the same process by hand and comparing.

8.1.5 Missing features

The implementation of the register allocation itself has simplified several steps, which the referenced book has done. This section serves to explain what these choices are, and what the consequences are.

Priority Spilling

The spilling process has no priority implemented at all, if a node is found which cannot be given a register because all conflicting edges already uses all the registers, it is simply spilled. The consequence of this is that we might spill a node which is accessed often in a short amount of time, where as a conflicting node that is only accessed very little could have been spilled instead.

Coalescing

Coalescing move instructions into a single node, if no interference edge exists between them, into a single node, has been omitted. The consequence of this is that we may have several unnecessary move instructions generated, which could have been eliminated had this been implemented.

Register EBX

[4] See `cctests/all_tests/O_KnapsackNoComments.die`

[5] This can be changed on the define `NR_OF_REG` in `register_allocator.h`.

The register EBX has been deactivated in allocation[6] due to unseen conflicting edges in a few test cases. The reason for this lies in how the spilled temporaries is handled, which uses the `EBX` register to store the spilled value for when they are used. This has not been accounted for when calculating the liveness analysis.

8.1.6 Eliminating dead code

Combined with the fact that we always implements the standard library, but often never use any methods in it, it seemed appropriate to eliminate these instructions from being emitted into the outputted assembly code.

The implementation for this is very simple, and only serves to remove code which is never visited in a control flow graph. Therefore it does not remove the code as seen below, since the flow graph does traverse this.

```
1 if (false) then
2   #// Some code
3 else
4   #// More code
```

After doing liveness analysis, the flow graph is quickly traversed, and any node which has not been visited is removed from the list of instructions that we want to emit. This is checked by a single integer `visited` in the node struct, which counts how many times it was visited while defining live-in and live-out sets.

This will also remove functions from the Diego code which never gets called. The only effect is therefore to clean up the emitted assembly code to help the human reader in understanding and debugging it while at the same time reducing the amount of physical storage required for the program.

8.2 Peep-hole

8.2.1 Introduction

The peep-hole optimizing phase runs after register allocation is performed. This was chosen so that it could utilize the fact that some liveness information was available. Its flow consists of a single while loop that runs until no change has been performed, i.e. no pattern has matched a set of instructions in the current iteration.

A peep-hole pattern, is a pattern of certain instructions. An algorithm has been made to match for these patterns, once a pattern is matched, it is replaced with a more optimized

[6] Determined by setting `NR_OF_REG` to 5 instead of 6 in `register_allocator.h`.

version. Each pattern is analyzed regarding its impact on performance, and that the optimization phase terminates (no never-ending loops).

Nine peep-hole patterns have been created (with some overlap), this list is by no means complete, and should be a good candidate for further optimization.

8.2.2 Algorithm

The algorithm for applying peep-hole is fairly simple. The component takes as input the entire IR-representation, and starts iterating through the list. It passes the the current element. If a pattern decides that it can optimize, then it is allowed to do so, and we start from the next element.

Each pattern is matched using a simple state machine. The state machine is internally implemented through a set of macros. Each macro allows for the state machine to match a template of instructions, and push it onto the next state.

The philosophy with this system was, that writing peephole patterns should be no different that writing a code template for the generator.

8.2.3 Patterns

Unnecessary Caller and Callee

This pattern recognizes that a register is pushed and popped to and from the stack, without it being needed. This pattern often arises with repeated calls, as the caller registers are repeatedly saved and restored between each call.

The patterns recognizes a caller/callee set of PUSH and POP instructions. Using liveness information it checks whether each register is live at that current point in the program, this can lead to two outcomes:

1. It is live: Pattern resets, no change is performed.
2. Is is not live: Patterns removes both the push and pop of the actual register in the caller/callee sequence.

This pattern removes two memory access instructions for each register, which is not live. The pattern became necessary because caller/callee instructions are always created around a function call, without regard for them actually being necessary or not. An alternative solution caption could have been to use the liveness analysis to check if performing caller/callee was necessary, in that a register would be live past a call instruction.

In the end the outcome would be the same.

Unnecessary Move At Dereference

The code generated for static link traversal is a bit excessive, as it has to allow for that temporary to be spilled. This caused unnecessary move sequences if the temporary was already in a register.

```
1 MOVL %TMP, %ebx
2 MOVL (%ebx), %ebx
3 MOVL %ebx, %TMP
```

Assuming that `%TMP` is a register, then Rather than moving the temporary into register EBX first, we can simply dereference it directly.

```
1 MOVL (%TMP), %TMP
```

Unnecessary Move To Self

If a move instruction is a move with the same origin and destination, it can be removed. Depending on the Diego code supplied, and the amount of variable used - this pattern is applied a lot.

Unnecessary Jump To Next Instruction

If an instruction is off type jump, and the label is an immediate successor in the natural flow of instructions, it can be removed entirely.

This implemented as a pattern that detects a JUMP, and if no other instructions types other than LABEL or TRACE appear before the LABEL that is jumped to, the JUMP instruction is removed. This maintains the semantics of the program, as neither LABEL or TRACE instruction types performs any semantic value.

Unnecessary Add/Sub

Subtracting or adding a zero from a value. This has no semantic value of the program and can be completely removed. This could easily have been removed in the weeding phase, since only constants are checked, but this was never corrected.

8.2.4 Performance and Termination

The performance is measured in the table below, this is done by running the Knapsack problem 10 times, and then dividing the total time by 10. The `Time Included` represents

the runtime of the program with all patterns enabled, the `Time Excluded` represents the program running with that pattern disabled.

Pattern	Time Included	Time Excluded	Yield	Yield %
CallerCallee	22.93s	24.88s	1.95s	7.83%
MoveAtDeRef	22.93s	23.58s	1.02s	4.32%
MoveToSelf	22.93s	26.83s	3.90s	10.80%
JumpToNextInstruction	22.93s	23.38s	0.45s	1.92%
Sub/unnessecaryAdd	22.93s	22.93s	0.00s	0.00%

Table 8.2: Performance variation when deactivation certain patterns. Note that the performance gain on the last pattern is too small to be measured.

The termination function for all the peep-holes can be shown in the same proof. Since all the patterns remove at least one instruction, and we have some number

$$t = \#instructions$$

then if some pattern is applied we have

$$t - \#removed\_instructions$$

It can then be argued that all of the patterns will terminate at some point, since we constantly reduce the amount of instructions t, that are present for each iteration.

If at some point no instructions are removed in an iteration, then no patterns have been applied, and the loop terminates.

8.2.5 Limitations

The algorithm for applying peep-hole is very simple, this simplicity can cause troubles as the number of patterns is expanded.

Since all patterns are matched simultaneously, should one pattern be a sub-set of another pattern, problems could arise. This could be avoided by expanding the algorithm to start over whenever a pattern is applied, rather than continuing its iteration.

Whether such a problem is the fault of a bad pattern, which directly conflicts with another pattern, or that the algorithm should be improved, results more in an software engineering aspect. Following that, code should be as robust as possible, making it hard for small changes to completely break the code.

Chapter 9

Code Emission

9.1 Converting IR to Assembly

The final stage of the compiler consists of converting the IR code to proper assembly code. Since the intermediate representation was created to resemble the assembly language as closely as possible, this phase is straightforward. A single problem which had to be solved was the handling of instructions involving spilled temporaries.

Most assembly instructions cannot reference to the memory, and has to involve the usage of registers. This was handled by explicit usage of the EBX register, so when an instruction to emit is involving spilled temporaries in a incorrect way, the spilled temporary is moved into the EBX register immediate before the instruction, and then moved back into the frame immediately after.

Instruction	Operator 1	Operator 2
MOVL, ADDL, ANDL, SUB, ORL, CMPL	No	No
IMUL	No	Yes
LEAL	-	No

Table 9.1: Instructions which cannot handle memory references.

The emission process itself simply consists of iterating trough the IR linked list, converting each instruction into assembly instructions using `printPrettyInstruction` and `prettyArgument`, each can be found in the file in `ir_emit.c`.

These two functions simply reads the IR instructions and converts it to proper assembly format, which is printed to either a file or stdout.

9.2 Factorial output

```
# standard # emit/ir_emit.c:42
intformat:
.string "%d\n"
stringformat:
.string "%s\n"
charformat:
.string "%c"
.local heapNext # emit/allocator.c:8
.comm heapNext,4,4 # emit/allocator.c:9
.local heapLimit # emit/allocator.c:10
.comm heapLimit,4,4 # emit/allocator.c:11
.local heapRemaining # emit/allocator.c:12
.comm heapRemaining,4,4 # emit/allocator.c:13
# end standard # emit/ir_emit.c:50
L302:
PUSHL %ebp
MOVL %esp, %ebp
PUSHL %ebx
PUSHL %esi
PUSHL %edi
MOVL 12(%ebp), %edi
MOVL $0, %esi
MOVL $0, %eax
CMPL %esi, %edi
SETE %al
MOVL %eax, %ecx
CMPL $1, %ecx
JE L313
MOVL 12(%ebp), %edi
MOVL $1, %esi
MOVL $0, %eax
CMPL %esi, %edi
SETE %al
ORL %ecx, %eax
JMP L314
L313:
MOVL $1, %eax
L314:
MOVL $1, %ebx
CMPL %eax, %ebx
```

```
41 JNE L444
42 MOVL $1, %eax
43 JMP L303
44 JMP L445
45 L444:
46 MOVL 12(%ebp), %esi
47 MOVL 12(%ebp), %eax
48 MOVL $1, %edi
49 SUB %edi, %eax
50 PUSHL %esi
51 PUSHL %eax
52 LEAL 8(%ebp), %ebx
53 MOVL (%ebx), %ebx
54 PUSHL %ebx
55 CALL L302
56 ADDL $8, %esp
57 POPL %esi
58 IMUL %esi, %eax
59 L445:
60 L303:
61 POPL %edi
62 POPL %esi
63 POPL %ebx
64 MOVL %ebp, %esp
65 POPL %ebp
66 RET
67 .globl main
68 main:
69 PUSHL %ebp
70 MOVL %esp, %ebp
71 PUSHL %ebx
72 PUSHL %esi
73 PUSHL %edi
74 LEAL 8(%ebp), %eax
75 MOVL %eax, 8(%ebp)
76 PUSHL $0
77 CALL sbrk
78 ADDL $4, %esp
79 MOVL %eax, heapNext
80 PUSHL $1048576
81 CALL sbrk
82 ADDL $4, %esp
83 ADDL $1048576, %eax
```

```
84  MOVL %eax, heapLimit
85  MOVL $1048576, heapRemaining
86  MOVL $5, %eax
87  PUSHL %eax
88  LEAL 8(%ebp), %ebx
89  PUSHL %ebx
90  CALL L302
91  ADDL $8, %esp
92  PUSHL %eax
93  PUSHL $intformat
94  CALL printf
95  ADDL $8, %esp
96  POPL %edi
97  POPL %esi
98  POPL %ebx
99  MOVL %ebp, %esp
100 POPL %ebp
101 PUSHL $0
102 CALL exit
103 ADDL $4, %esp
104 RET
```

Chapter 10

Language Extensions

10.1 Load Directive

The load directive allows for Diego programs to include other Diego source files into it. All programs have the Diego standard library included by default. The syntax for creating a load is:

```
#load FILE_NAME
```

Loads are allowed from anywhere in the Diego program, and the results of a load are not constrained to any scope.

The load directive has been designed to work a bit like PHP's `include_once`. The `include_once` function works by simply dumping the contents of another file where the function is called, and then continuing onwards to work with the new source. This approach wouldn't work in Diego, due to the constraints put on when declarations and statements are allowed to appear. Instead we adopted a slightly modified design. In this design, whenever a new file is included, its annotations, declarations, and statements are appended to their respective lists in a "mother" program.

This function also differs in another way from PHP's version. This does not include code in at run-time, due us not wanting it, and the many complexities associated with it. Instead all load directives are collected in a pre-parsing phase.

The pre-parsing phase created the "mother" program, on which the parser may run. Thus the load directives are not a part of the grammar[1]. Since the pre-parsing phase does not have any knowledge of the structure of the program (or if it is even correct), we will not do the ordering just yet. We will instead modify our grammar slightly, to allow multiple bodies, each body will contain the contents of a single file. To allow the grammar to

[1]The grammar will however silently ignore them, as they will appear as comments to it

see these multiple bodies, a separator token has been introduced. Whenever we reduce this rule in the parser, we can perform the appending step, thus creating an AST with the same structure as before, but now allowing multiple files. The algorithm for the pre-parsing phase is shown in algorithm 5.

```
Data: stdin: A Diego source file from standard in
Result: out: One large Diego source file, ready to be parsed
begin
    Q ⟵ [stdin, standard library];
    S ⟵ [];
    while Q is not empty do
        Write NEW_FILE token to out;
        F ⟵ popHead(Q);
        while EOF not reached for F do
            L ⟵ next line;
            if L matches load directive then
                N ⟵ file referenced by L;
                if N not in S then
                    addElement(S, N);
                    addElement(Q, N);
                end
            end
            else
                Write L to out;
            end
        end
    end
end
```

Algorithm 5: The pre-parser algorithm for implementing the load directive

The algorithm for pre-parsing relatively simple. It goes through all the files to be loaded, and outputs the contents of the file, with a `NEW_FILE` token between each file. The output of this file can easily be parsed, with the small changes already described. The only problem left now is the error reporting. The error reporting will now be relative to the large "mother" file. To correct this information, we have introduced some extra meta-data to the `NEW_FILE` token. It contains the origin, such that the correct file can be reported. Whenever a `NEW_FILE` token is read, the line count can be reset to 0. This last change is done already in the lexing phase, in which the line numbers were already being controlled.

10.1.1 The Standard Library

To make real use of the Load Directive feature, the Diego compiler ships with a standard library. The standard library is relatively small, but showcases some of the things it could be used for. In the code generation chapter, we for example showed how the absolute operator is implemented by making a simple call to the standard library.

The standard library also comes with a hash table implementation, based off a C version used internally in the compiler. It was put there primarily to showcase that the compiler was stable enough to support it. While also demonstrating what would make sense to put in the standard library. The annotation system were also meant to use this, however never fully implemented, more on that later.

10.1.2 Limitations

The loading system implemented here is relatively simple. While it might seem like its fine, there are however several crucial features that are missing for it to be ready for real-life production environments.

The biggest problem with the loading system, is that all paths are currently relative to the directory from which the compiler is loaded[2]. This limitation makes it very hard to write any re-usable libraries for the language. Optimally we would want paths to be relative to themselves, just as they are in C. The main reason for this being difficult is that one of the files will come from stdin, and we do not know the actual location of this file. This cannot just be changed, as it is part of the compiler specification, and would make this a breaking change as opposed to a language extension.

Another issue to be vary of are the lack of namespaces. This especially becomes important when we consider that built-in operators rely on the standard library. It is currently possible for a program to override the behavior of for example the absolute operator, by providing a function with the same head, as the one in the standard library. We will not go further into a discussion of namespaces, other than saying they could fix this issue. By allowing the caller to be specific about which version of a function to call.

10.2 Annotations

This language has support for what in the Java world is known as annotations. These are small notes, that may be put on different parts of the program, annotating some extra information to it.

Annotations can be put on most nodes, to put additional information to it, we however feel that they're most useful only on functions and on field declarations. This language

[2]This is not a problem for the standard library, as the pre-parser gives an absolute path to it

only supports them in functions (technically on the body, to allow the main function to have annotations).

Listing 21 shows a small test program, where a function body has two annotations, namely a `NormalAnnotation` and `StructuredAnnotation`. The structured annotation has two values associated with it.

```
1 func helloAnnotations(): bool
2     @NormalAnnotation
3     @StructuredAnnotation(avalue = 42, abool = false)
4     #// ... Normal body of function ...
5 end helloAnnotations
```

Listing 21: Hello annotations!

Annotations are meant to be used both by the compiler, the program, and other tools that may work with the language (for example IDEs). These tools may use the associated data to perform extra work, for example code generation. The result of this is allowing for a more declarative way of programming, that wouldn't be as easy to get without.

We will now present implementation details, how they are used in the current state of Diego, how we see them being used, along with a discussion on its current limitations.

Implementing

The annotations have several limitations on which values it may take. Annotations are supposed to be used already at compile-time, for this reason it cannot take any values, dynamically calculated by a program. This means that the values used by annotations must be known at compile-time.

By only allowing structured annotations to take literals, we can ensure this. This is however also a very conservative approach. It could, with the help of the constant folder, for example easily be expanded to also allow simple expressions, that may be evaluated at compile-time. We however feel that this is an unnecessary feature, and as such not implemented. Due to the way strings currently are implemented, this also means that they are not supported as values in annotations.

We feel that at this point it should be clear how one could add annotations into the language's grammar. For this reason we will not discuss how annotations, that are not kept until run-time, were implemented.

It should be possible for functions to inspect which annotations are available for which function. Optimally we would like some dictionary, in which we can lookup and get

all annotations available on a function. Unfortunately the type system does not currently support, function pointers as values. Which would be required in such a mapping. Instead we have implemented a small proof-of-concept, which allows a function to access the values of its own annotations. This system could easily be expanded at a later point.

The values of a structured run-time annotation, should be kept in a record, of the same name, with the same fields. It should also only be possible to reach it from within the body's scope. An example of this is shown in listing 22.

```
1 func outer(): bool
2     @MyAnnotation(value1 = 10, value2 = true)
3     func inner(): bool
4         write MyAnnotation.value2;
5         return true;
6     end inner
7     write MyAnnotation.value1;
8     return inner();
9 end outer
```

Listing 22: Using annotations at run-time

To implement this, we will use a technique we haven't yet discussed, which is by transforming the AST. We will do this in a weeder component, which will insert variables, and initialization code directly into the AST. Implementing features by transforming the AST has several advantages.

In this case, we were able to implement an entire feature without having to touch the back-end or even the code generation component of the compiler. This way the code created will also automatically benefit from the work of other components. We automatically get the scoping from symbols. It also makes it more likely that the back-end can optimize, since the code generated, will be identical to code it already handles. These things wouldn't happen if they were treated as separate entries.

Thus the job of this component, is actually to take the code like shown in 22 and transform it into something like listing 23.

One concern we had with this design is that it leaks to several variables, meaning that the compiler will throw errors, should the program already contain variables of that name. This could be minimized by using a dictionary approach, like described earlier.

The component was implemented in the weeder phase. The AST transformation itself were relatively easy, and involves only creating new nodes and inserting them into the tree where relevant.

```
func outer(): bool
    @MyAnnotation(value1 = 10, value2 = true)

    var MyAnnotation: record of { value1: int, value2: bool };  #

    func inner(): bool ... end inner

    allocate MyAnnotation;                                      #
    MyAnnotation.value1 = 10;                                   #
    MyAnnotation.value2 = true;                                 #

    write MyAnnotation.value1;
    return inner();
end outer
```

Listing 23: The code after being transformed by the component

10.2.1 Annotations Supported by Diego

Currently the language supports two annotations.

`@asmtrace`: Enables debugging trace in the generated assembly code, will only be enabled for that single function, does not extend into nested functions.

`@skipsafety`: Skips also safety checks for that single function body, does not extend into nested functions.

10.3 Strings

Diego supports a simple version of dynamic strings. The goal of this string implementation is not to create an efficient string implementation, but to show how they could be implemented. We will discuss any interesting implementation details, along with discussing how they could be improved.

Internally the system does not know of strings, the Diego compiler only knows of the primitive type, that is `char`. Strings are implemented in a literal sense as arrays of characters. As a result this also means that the built-in write keyword does not support strings, this has instead been implemented in the standard library along with other useful string utilities.

Even though the system does not know of strings, a string literal is still supported. It is implemented as a shorthand for creating an array of characters and initializing that array with the characters contained in the string.

Just like with annotations, this has been implemented through an AST transformer. In the development of this component we encountered a few interesting problems.

The first problem we encountered is that not all string literals are used in assignment. Some literals are passed directly into a function, and as such does not have a variable on which we can insert allocation statements. To fix this problem a new variable is inserted for every string literal there is. This in itself introduces yet another problem, which is the leaking of variables into the scope. We do not wish these to be able to clash with any existing variables. To solve this problem we used a technique that we also used in our type-system, which is to create variables that contain characters that are invalid in the language grammar. This way the system can continue to treat these variables like any other variable, but without us having to worry about name clashes.

10.4 Run-time Security Checks

Several run-time security checks has been added to the language. These are implemented by emitting extra code, which checks that the operation is valid, before performing it. If it is illegal then a call to `exit` is made with an appropriate return code. Currently code is emitted for each exit directly at the check, as opposed to jumping to where the exit is actually made. This would have decreased the size of the binaries by a bit.

10.5 Improved Loops for Arrays (Not complete)

One of the features planned for Diego, of which some of the work was done, was an improved way of writing for loops. This feature would allow for a programmer, to write for-loops with arrays, using the following syntax:

```
+------------------------------+   +-----------------------------------+
 Improved syntax                  Traditional syntax
+------------------------------+   +-----------------------------------+
var myArray: array of int;         var myArray: array of int;
var e: int;                        var idx: int;
                                   var element: int;
for (e in myArray) do {            idx = 0;
    #/* Do something with e */     while (idx < |myArray|) do {
}                                      element = myArray[idx]
                                       #/* Do something with element */
                                       idx = idx + 1;
                                   }
```

We had two primary reasons for wanting this feature. The first reason being a more easy to use syntax. We have personally seen, that a lot of the time we write loops, we use them simply to enumerate over some collection. This feature would allow, to do the same in less code, along with more precisely communicating the intent of the code.

Additionally, we believe that it would be easier to generate efficient code for the new improved syntax. The reason we can improve the code generated, is because we know for a fact that we will be enumerating over an array. Before going into a discussion on what to do, we will first present the steps involved in every single iteration, of the traditional syntax.

1. Calculate the array index
2. Compare the index against the array index
3. Find the element
   (a) Find the variable (possibly following several static links)
   (b) Check that the variable is not null (involves a conditional jump)
   (c) Check that the index is non-negative. Check that the index is within bounds (involves two conditional jumps, and several memory look-ups)
   (d) Add the index to the base address
4. Do work
5. Increment the index

As we can see, there are many steps, that do not directly involve the work we actually wish to do. We will now see how our knowledge of the loop's intent, can help us eliminate several "useless" instructions, that would otherwise be needed in general-purpose code.

Right off the bat, we can see several improvements that may be made. In the traditional syntax we would repeatedly calculate the same array length. Obviously the argument could be made, that the version on the right, could still cache the length, and as a result only have to calculate it once. We may also assume this since Diego doesn't support any threads currently. It is also potentially dangerous to enumerate over a collection which is being changed as we enumerate it. Thus given the introduction of multiple threads, this boiler-plate would have to be changed, in the case of multiple threads working on the array. Overall we would like no changes to occur on the array. It is however relatively hard to implement such a restriction at compile-time, and any run-time solution would likely be more expensive that what we could save. Thus the results of changing the array while enumerating, should simply be undefined.

Now that we can assume that the array won't change, we can do yet another optimization. We know that all indices that we check will be within bounds. This is because the

condition, on which our loop terminates, will be that the index is no longer within bounds. For this reason we can turn all of the index checks off.

We also see that a null check is performed in every iteration. We cannot turn this completely off, since the array may still be null. We can however move this check out of the loop, and only do the check once, before we begin iterating.

This brings us to our final optimization. Note that we calculate the base address of our array in every single loop, we do not need to do this. We can instead only calculate the base address once. At the end of every iteration we simply have to add the offset required to reach the next element (in this case always 4 bytes). This way we skip a lot of work, especially in the case of the variable being several scopes further up, thus requiring several layers of static links.

Chapter 11

Conclusion

The kitty compiler, using the supplied Diego language with additions, has been created and tested extensively.

Each phase has been structured into an separate phase which always ended by testing if the current phase had a working state. Several additions were created in the final phase, by example this includes advanced register allocation, peep-hole optimization, strings, run-time safety-checks, annotations.

The final result was tested using the supplied testing framework with slight modifications, along with several tests which were aimed at additions and design choices performed. A single issue remained in the parsing phase, along with register allocation only utilizing 5 of the 6 general purpose registers.

The remaining tests were all completed and adhered to the expected result.

Chapter 12

Appendix

12.1 Source code

12.2 Parsing Tests

Testing Multi- and Single-line Comments

Status: PASS

Test file: `comment_test.diego`

Description:

1. Testing a single line comment. This should result in the line being ignored.
2. Tests that nested comments work across several lines, and that the line on which it is ended does not matter.
3. In-line comment with code.
4. Allowing unclosed multi-line comments to cause the parser to ignore the rest of the file. Note that some compilers would consider this to be invalid code. This is allowed here as a design choice, since it allows for the programmer to quickly comment out the rest of a file.

Expected Result: A single statement, returning 1 + 1.

Motivation: Assuring that the design choices made in the Flex file for comments work as intended.

Testing Operators

Status: PASS

Test file: `operators_test.diego`

Description: Tests that all the operators may be used, and that their precedence work correctly. Also tests that spaces aren't needed between operators and its operands.

Expected Result: An AST describing correct execution of the statement.

Motivation: Assuring that the operators work as intended.

Testing Statements and Control Blocks

Status: PASS

Test file: `statement_test.diego`

Description: Tests different kinds of statements and blocks, and checks that they work with and without any optional features.

Expected Result: An AST describing correct execution of the statements.

Motivation: Assuring that the operators work as intended.

Testing Declaration Lists and Function Calls

Status: PASS

Test file: `estimate_pi.diego`

Description: Tests declaration lists and function calls in a small code snippet intended to estimate π.

Expected Result: An AST describing correct execution of the statements. This includes that the declarations lists are correctly flatten as described in "Building the Abstract Syntax Tree".

Motivation: Assuring that the lists and functions works as intended.

Testing Typedefs, Records, and Heap Allocation

Status: PASS

Test file: `alloc_test.diego`

Description: Tests typedefs, records and the `allocate` keyword.

Expected Result: An AST describing correct execution of the statements.

Motivation: Assuring that the syntax works as intended.

12.3 Type-checking Tests

Array Access Test

Status: PASS

Test file: `comment_test.diego` and `bad_array_access_test.diego`

Description: Tests that array access works as expected. Also tests that multi-dimensional arrays work.

Expected Result: The bad test should fail on line 4-6. The good test should not complain about any of the types.

Assignment Test

Status: PASS

Test file: `assignment_test.diego`

Description: Tests that assignment works as it should. It tests int and bool. It is also tested that null is allowed to be assigned to records and arrays.

Expected Result: That the compiler validates, as noted in the comments of the file.

Negation Test

Status: PASS

Test file: `bad_negation_test.diego`

Description: Tests that negation is only allowed boolean types.

Expected Result: Should fail on the last line.

Bad Symbol Test

Status: PASS

Test file: `bad_symbol.diego` and `bad_symbol2.diego`

Description: Tests that variables aren't allowed to be overriden.

Expected Result: The compiler should fail to compile.

Constant Folding Test

Status: PASS

Test file: `constant_folding_test.diego`

Description: Tests that constant folding works.

Expected Result: The new AST should now only have a single node, correctly stating the folded value.

Nested Functions

Status: PASS

Test file: `nested_functions.diego`

Description: Tests taht the symbol gatherer will correctly fill out the scopes, when using nested functions.

Expected Result: That the symbol table is correctly filled.

Null Test

Status: PASS

Test file: `null_test.diego`

Description: Tests that null is allowed where arrays and records are expected. Also tests on edge cases, such as function parameters and in nested records.

Expected Result: That the test fails on the lines annoted in the test file.

Record Test

Status: PASS

Test file: `record_test.diego`

Description: Tests different types of records and their use in functions.

Expected Result: That the test fails on the lines annoted in the test file.

Return Test

Status: PASS

Test file: `return_test[2,3].diego`

Description: Tests both correct and incorrect code to see if the return checker will catch any errors.

Expected Result: Return test 1 and 2 should succeed, while test 3 should fail.

Typedef Test

Status: PASS

Test file: `typedef_test.diego`

Description: Tests different kinds of typedefs, and checks if the compiler will correctly resolve the types.

Expected Result: The test program should succeed.

Write Test

Status: PASS

Test file: `write_test.diego`

Description: Tests that the write statements work correctly.

Expected Result: The test program should succeed

Linked List Test

Status: FAIL

Test file: `linked_list_test.diego`

Description: Tests that it is possible to create a linked list structure.

Expected Result: The test should succeed.

12.4 Additional tests