Expertise
and insight
for the future

ANH NGUYEN

Compiler design and implementation in OCaml with LLVM framework

Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Bachelor's Thesis
10th March 2019

Author	ANH NGUYEN
Title	Compiler design and implementation in OCaml with LLVM framework
Number of Pages	103 pages + 1 appendices
Date	10th March 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Software engineering
Instructors	Jarkko Vuori, Principal Lecturer Minna Paananen-Porkka, Language Supervisor

The past several decades constantly witnessed the noticeable growth in the quantity as well as the performance of compilers for high-level programming languages due to the high demand for increasingly intricate computer programs.

The objective of this thesis is to explore the feasibility of adopting Low Level Virtual Machine (LLVM) framework which is a set of well-optimised, reusable tools for constructing modern compilers. Specifically, the thesis focuses on employing LLVM framework as the Intermediate Representation (IR) code generator and as the back-end compiler infrastructure to rapidly construct compilers. Along the way, this thesis depicts the fundamental structure of a modern compiler as well as the techniques to apply compiler theoretical concepts into practice.

In order to achieve the goal of demonstrating those concepts, practices and the effectiveness of LLVM framework, the thesis project was to design and implement a compiler for a simple, imperative programming language known as Tiger. During the process of developing this compiler, several commonly used libraries for building compiler including Lex, Yacc were leveraged to solve domain specific problems.

The final outcome of the project is a compiler written in a strongly-typed, general purpose, functional programming language known as Objective Categorical Abstract Machine Language (OCaml). This compiler can translate the Tiger programming language to LLVM

IR and subsequently to any architecture-dependent native code supported by LLVM. As a result, the project analyzes and emphasizes the robustness and effectiveness of LLVM framework in the process of constructing compilers. Additionally, the operational compiler serves as concrete examples as well as proofs for the correctness of the theories and skills discussed in this thesis.

Keywords	Compiler, LLVM, OCaml, Lex, Yacc

Contents

List of Abbreviations

AST	Abstract syntax tree
ARM	Advanced RISC Machine
DFA	Deterministic Finite Automata
FSA	Finite State Automata
GCC	GNU Compiler Collection
Lex	Lexical Analyzer Generator
LLVM	Low Level Virtual Machine
LLC	Low Level Virtual Machine Static Compiler
ML	Meta Language
MIPS	Microprocessor without Interlocked Pipelined Stages
NFA	Nondeterministic Finite Automata
OCaml	Objective Categorical Abstract Machine Language
SSA	Static Single Assignment
YACC	Yet Another Compiler-Compiler

1 Introduction

Compiler development is an interesting computer science field that empowers countless modern technological advancements in information technology industry. As the demand for robust, complex computations in various IT fields is booming, the increasing number of new programming languages such as Swift, Go, Rust and Elixir have quickly emerged in the last few decades. In fact, the process of constructing the compiler often requires the equal understandings from both computation theories and programming practices. As a result, developing compilers not only strengthens programmers' theoretical foundation of computer science but also boosts their problem-solving, programing skills. More importantly, the process of implementing compilers often provides the insights into how popular programming languages operates behind the scenes. Indeed, the curiosity about the underlying implementation of modern programming languages is the biggest motivation behind this project.

Tiger is a simple, general-purpose, statically-typed and procedural programming language designed by Andrew Appel in his book "Modern Compiler Implementation" [1, p. 2]. It has been commonly used for teaching compiler design principles at many universities such as Princeton, Columbia. As a result, there are many existing implementations of Tiger back-end compilers, each of which targets merely a single architecture either MIPS, x86 or ARM. However, this project employs the power of LLVM Intermediate Representation (IR) and its industrial-strength static compiler LLC to target multiple computer architectures at once. In fact, LLVM infrastructure is selected for this project because of its wide adoptions by many recent compiler projects either to create new programming languages such as Swift, Rust, Julia or to enhance the development process of the existing ones such as Glasgow Haskell Compiler (GHC) [2].

The purposes of this thesis were to give an overview of modern compiler development process and to analyze the benefits of using LLVM framework. Those two goals were achieved by constructing the compiler’s front-end components that collaboratively translate the Tiger language to LLVM IR code before adopting LLVM back-end infrastructure to produce architecture-dependent, decently optimized native code.

1.1 Definitions

1.1.1 Computer programming languages

A computer program is a set of instructions following rules of a specific programming language. Once a program is executed, it instructs the computer to perform actions and achieve outcome. In order to construct executable software, programmers are required to use specific vocabularies and to follow a set of grammatical rules which are formulated in programming languages specifications. [3, p. 42]

Programming languages can be categorized into two groups: low-level languages and high-level languages.

Machine languages are the lowest level programming languages as they can be directly executed by processors. Those languages can be used directly to build software yet the development process is usually inconvenient, time-consuming and error-prone. Moreover, machine languages are architecture-dependent since different processors requires different types of machine code. [4, p. 9]

On the contrary, high-level programming languages are human-readable thanks to their high abstractions that prioritize the easiness of expression and readability [4, p. 9]. As a matter of fact, the use of high-level programming languages noticeably eases the pain of building software as they give programmers the ability to express complicated ideas with more concise, readable instructions. In fact, instructions in high-level languages are relatively similar to English, which brings the easiness when translating human's ideas into code. Additionally, compilers for high-level languages are often more intelligent in detecting programming errors and they also give programmers more informative error messages. As a result, it is easier to spot the mistakes and correct them during the software development process. [5, p. 1]

1.1.2 Compiler software

Implementing software in high-level programming languages is apparently more productive compared to programming in raw machine code. However, computers can not directly execute instructions written in high-level programming languages. As a result, code in high-level languages must be translated to machine languages before being

processed by the processors. This translation procedure is often time-consuming, repetitive; yet it can fortunately be automated by translation software known as compiler. [6, p. 1]

Formally, a compiler is a software that is responsible for translating a sequence of instructions written in one language (source language) to the corresponding version written in another language (target language). Usually, the source language tends to be a high-level language while the target language is low-level one. During the course of compiling, overly apparent programming defects are often detected and reported [5, p. 1].

1.2 Technologies

1.2.1 Tiger programming language

Tiger is a simple, statically-typed, procedural programming language of the Algo family. Tiger has 4 main types: integer, string, array and record. The language has support for control flow with if statement, while loop, for loop, functions and nested functions. The syntaxes of Tiger are similar to languages in Meta-language (ML) family. [7, p2]

The sample program in Tiger language is shown in Figure 1:

test ▸ demo.tig

```
let
  type record_type = { input: string, name: string }
  var record := record_type { input = 5, name = "Demo program" }

  type arrary_type = array of record_type
  var array_size := 5
  var array: array_type := array_type [array_size] of record

  function fib (n: int): int =
    if n = 0 then 0
    else if n = 1 then 1
    else fib (n - 1) + fib (n - 2)

  var counter := array[0].input
in
  while counter > 0 do
    (
      printInt (fib(counter));
      print (record.name);
      counter := counter - 1
    )
end
```

Figure 1. Sample program in Tiger

1.2.2 OCaml programming language

OCaml is a statically typed, general purpose, and functional programming language that has been in the industry for more than 20 years [8]. It is a strongly typed language with support for polymorphic type checking which gives flexibility to the type checking mechanism. Another innovative functionality is type inference, which exempts programmers from explicitly declaring types for every single variable and function-parameters as long as those missing types are automatically deductable from the context. As a result, this functionality allows programmers to write more concise, reusable code since the compiler can usually infer the types from programming context. [8]

In addition, OCaml promotes functional programming principles with full support for mathematical lambda functions, immutable data structure, recursion tail-call optimization, algebraic data structures and pattern matching. Nonetheless, due to the requirement of expressing idea easily in some specific programming tasks, the language also supports imperative paradigm, mutable data types such as array, hash table and a complex exception handling system. [8]

In particular, OCaml provides automatic memory management which mitigates the risk of memory corruption. In fact, this feature allows programmers to focus solely on the structure of data computation, rather than the manual memory deallocation process [8]. Therefore, implementing compilers in OCaml is more convenient compared to using languages without garbage collector such as C or C++.

Finally, OCaml offers well-supported tools for writing compilers, such as Lexical Analyzer Generator (Lex), Yet Another Compiler-Compiler (Yacc) and LLVM bindings that interact with C++ API of LLVM. As a result, the pain of writing this compiler is eased significantly thanks to those toolkits.

1.2.3 LLVM compiler framework

LLVM is a compiler toolkit implemented in C++ that offers a rich set of commonly used modules and reusable toolchains for constructing modern compilers in a timely manner. In fact, LLVM provides developers tools to programmatically generate instructions in LLVM IR - a statically typed, architecture independent abstraction of assembly code. LLVM IR bitcode can further be optimized in a sequence of phrases and consequently

compiled into native machine code in various target architectures such as MIPS, x86, ARM. As a result, this advantage boosts the portability of the source languages to multiple machine platforms. In addition, LLVM can also perform Just-In-Time (JIT) compilation on the IR code in the context of another program as an interpreter. [2]

Initially, LLVM was a created by Chris Latter, who is also the father of Swift programming language, as a research project at the University of Illinois in 2000. In fact, LLVM framework was created because other existing open source C compilers such as GNU Compiler Collection (GCC) had become stagnated. [2]

Indeed, GCC aging codebase often poses a steep learning curve for new developers. GCC was implemented in a monolithic mindset which means that every component is tightly coupled with each other causing the poor reusability when integrating with other software. Furthermore, GCC did not provide support for modern compiling techniques such as JIT code generation, cross-file optimization at that time. [9]

As a result, LLVM took a completely different approach by employing a modular, reusable architecture in which each compiler component is constructed following the single responsibility principle and they are a loosely coupled. Therefore, those components can be composed together in order to build a full compiler. Furthermore, the implementation of LLVM involves best known modern techniques such as Single Static Assignment (SSA) compilation mechanism with the ability of supporting both static and runtime compilation of any programming languages with high performance. [2] [10]

Over the last two decades, LLVM project has significantly gained its popularity in the field of compiler development since the framework is not only useful for developing new programming languages but also for extending and enforcing the back-end development process of the existing ones. Hence, the adoptions of LLVM tools are visible in multiple industrial-strength projects including Apple's Swift programming language, Rust programming language, Clang compiler, Glasgow Haskell Compiler (GHC), Kotlin native, and WebAssembly. [2]

1.3 Structure of the implementation process

The process of constructing a compiler is conventionally separated into two parts (front-end and back-end) in order to boost the modularity and reusability of each part [5, p. 2].

In practice, the front-end part of the compiler consists of several phases: lexical analysis, syntax analysis, semantic analysis, and Intermediate Representation (IR) translation [5, p. 3]. In details, the lexical analysis phase is responsible for deconstructing the input program written in source language into a sequence of valid words. The next phase is syntax analysis, in which the structure of the source program is analyzed and programmatically captured by the data structure known as Abstract Syntax Tree (AST). Next, the semantic analysis stage mainly conducts type-checking and determines variables scoping. The last front-end step is to transform abstract syntax tree to an Intermediate Representation (IR) code that acts as a bridge connecting front-end and back-end. [11, p. 6]

On the other hand, the main task of back-end is to efficiently map the generated IR code to the corresponding set of instructions in target language. In fact, the compiler back-end usually consists of several phrases: instruction selection, control flow analysis, dataflow analysis and code emission. During the process, the back-end also performs optimization by eliminating redundant computations. [11, p. 6]

In practice, if a compiler is developed from scratch without using any framework, the compiler development process tends to follow a sequence of steps starting from lexical analysis to linking phrase as can be seen in Figure 2. In that linear process, each phase consumes the outcome of the previous step and computes the output which can be used by the next step [5, p. 2]. However, this project takes a shortcut path by building the front-end part of the compiler and then leveraging tools provided by LLVM back-end infrastructure to do the heavy-lifting back-end work. By taking this approach, the project can rapidly produce a decently optimized, multiple-target-oriented compiler within a short period of time.

In detail, the thesis project implements all front-end parts of the compiler including lexical analysis, syntax analysis, semantic analysis and IR translation. Once the IR code is generated by the front-end compiler, this project simply leverages the LLVM static back-end compiler (LLC) which takes LLVM IR as the input and subsequently emits assembly code for the specified architecture as the output [12]. After that, the

generated assembly code are processed by the assember and the linker in order to yield the executable binary code [12]. Furthermore, the design of this project boosts the extensibility of the Tiger language by allowing the Tiger programs to call native C functions. As a result, those associated C functions must also be compiled and linked to the compiled code of the Tiger program. Fortunately, this step could simply be performed by using the C compiler (Clang) which is one of the most popular tools based on LLVM framework. Specifically, Clang compiler is the C languages family (C, C++, Objective-C) front-end compiler that is built on top of the LLVM back-end infrastructure [9]. It is responsible for translating programs written in C languages family into LLVM IR code before using LLVM back-end compiler to further compile the generated IR code to assembly. Finally, the compiled assembly of the C functions is linked to the assembly version of Tiger program before the executable binary is yield.

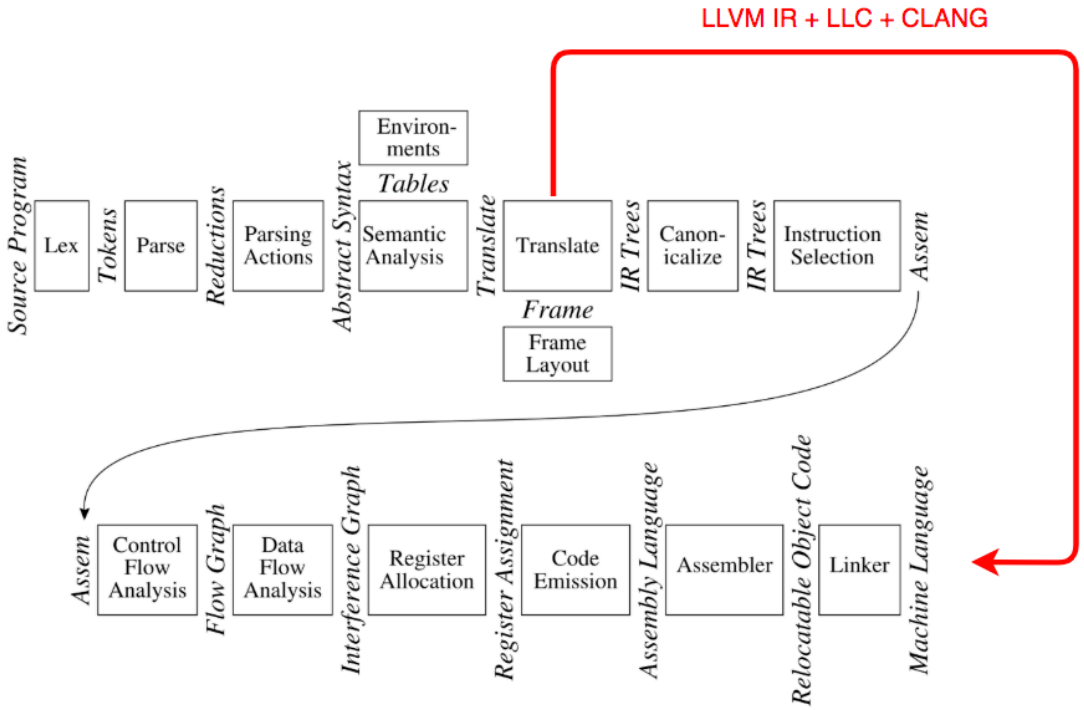

Figure 2. Compiler development flow [11, p.4]

2 Lexical analysis

Lexical analysis is a phrase in which string input is partitioned into a set of individual valid words which are often refered to as lexical tokens. Each lexical token is traditionally regarded as a unit constructing the grammar of that programming language. They are commonly divided into several types such as keywords (IF, THEN, WHILE), variable names (foo, bar), numbers (1, 2), and strings ("foo", "bar"). In fact, input stream is scanned one character at a time from left to right order. During the process of scanning, the lexical analyzer accomplishes two tasks. Its first task is to perform omission of white spaces, new line symbols and comments so as to minimize the responsibility of the subsequent stage - syntax analysis. In fact, this task is one of the primary reasons why lexical analysis should be separated into a different stage from syntax analysis. The second task of the lexer is to group sequences of characters into tokens by pattern-matching them against a collection of predefined token rules. Finally, those matched tokens are sent to syntax analysis phase for processing. [5, p. 10] [11, p. 16]

In order to specify rules of lexical tokens, a formal algebraic language known as Regular Expression is commonly used to describe the pattern of tokens.

2.1 Regular expression

A language is constructed from a collection of words in which each individual letter is taken from a set of characters. For instance, integers can be represented by a set of digits from 0 to 9 while variable names (identifiers) often consists of letters, digits and several special characters such as star, underscore. [5, p.10]

Regular expression (Regex) is an algebraic, human readable notation that can be employed to provide specifications for languages [5, p.10]. A regex is constructed by 2 parts. The first component is the set of all valid characters in the languages; for example, all alphabet characters, digits, special characters. The second component is a set of Regular expression operators which are often known as meta characters. For example, **|, (,), *, + .** Each meta character has a special meaning described in Table 1.

One important characteristic of regular expression is composability. This means that regular expression fractions that specify patterns for simple strings can be combined together into a larger expression that describes a more complicated set of strings. [5, p. 10]

Table 1. Common regular expressions and meanings

Regular expres-sion	Formal description	Example	Meaning
a	A simple alphabet character	a	single letter string ‘a'
E	Empty string	""	empty string
M|N	Alternation expression composed of expres-sion M and N. This expression matches strings that satisfy either expression M or N	a | b	Matched string is either ‘a’ or ‘b'
M . N or MN	Concatenation expression composed of ex-pression M and N. This expression matches strings that is the concatenation of 2 sub strings a and b with a satisfies M constraints and b satisfies N constraints	a.b	Matched string is “ab"
M*	Concatenation of non-negative occurence of string that satisfy expression M	a*	Matched string is ‘’, ‘a' or “aaaaaa..."
M+	Concatenation of non-zero occurence of string that satisfies expression M	a+	Matched string is ‘a' or “a..."

M?	Optional expression that indicates that the occurrence of M is either 0 or 1	a?	Matched string is '' or 'a'
[a-zA-Z]	Any single character in the set	[ab]	Matched string is ‘a' or ‘b'
.	Any single character except new line	.+	Any character that is not “\n”
"a+*."	Quotation, string between 2 quotes literally describe itself	"literal string"	Matched string is literally “literal string"

However, there are situations where one string can match multiple tokens. For instance, string **“ifabc"** can be recognized as **if** *abc* or variable name **ifabc**. As a result, rules for resolving ambiguities should be applied in those cases. In practice, when a string matches multiple tokens, the **longest matching token** is usually selected. Therefore, “ifabc” is recognized as variable's name **ifabc** in the previous example. Furthermore, if there are more than one longest matching tokens, the scanner should select the one that is specified first in token list. Thus, the order of specified rules of tokens plays an important part in resolving conflicts. [11, p. 20]

Regular expression is a useful specification language for lexical analysis. However, to programmatically implement regular expressions, the concept of finite state machine (finite automata) needs to be explored. [11, p. 20]

2.2 Finite automata

2.2.1 Deterministic finite automata (DFA)

Formally speaking, a finite automaton is an abstracted machine that consists of a limited number of states (nodes) and a set of transitions (edges) leading from one state to another. In addition, a symbol is usually assigned to each edge. [5, p.16]

Finite automata can be utilized to verify the validity of the input string in a language. In practice, finite automaton starts from one node regarded as starting state. From starting state, the scanner reads one input character at a time then it compares this character against the symbol of each edge coming from that current state. If the scanned character matches the label of an edge X, the scanner follows that edge X to navigate to the next state. This process repeats until all input characters are read. Finally, the last state is checked if it belongs to a group of accepted states (final states). If the last state is accepted, the input string is considered as valid token of the specified language [5, p.16]. In contrast, if the last state is not accepted or if there is no matched transition to follow at any point during the process, the automaton simply marks the input as invalid [5, p.22].

Figure 3 depicts the simple automata for a language that consists of tokens IF, INTEGER, REAL and ID. Each state is illustrated by a circle, and it is often numbered for the easiness of identification. The starting state has the label 1 and it is usually the target of an arrow without any label starting from outside of the DFA. Nodes with double circles are accepted states or final states. The arrows connecting two states denotes transitions from one state to another. Additionally, edges with multiple characters are indeed used to concisely illustrate a group of parallel transitions.

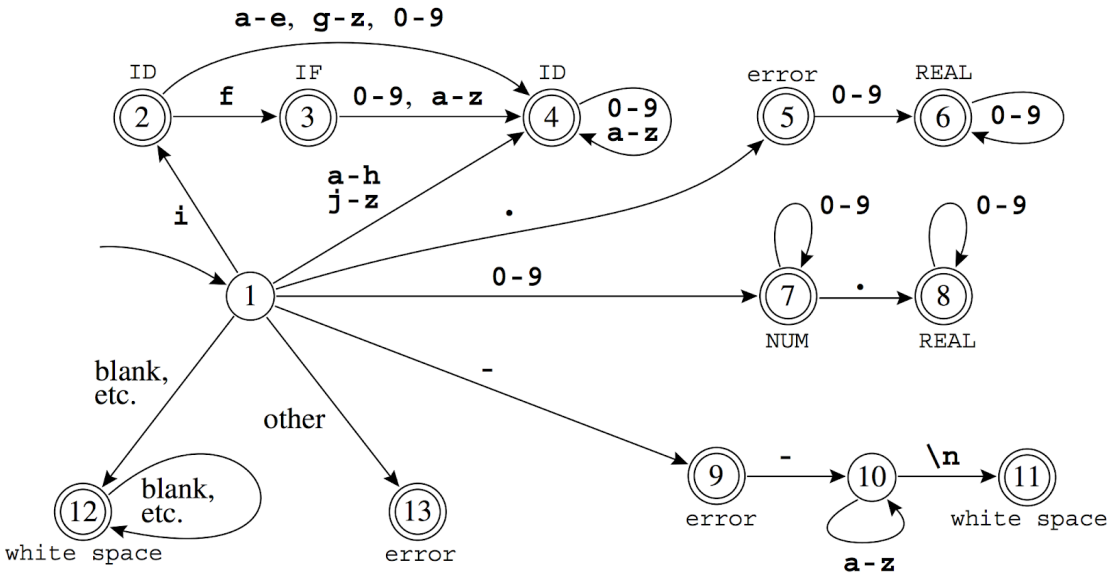

Figure 3. Visualization of deterministic finite automata [11, p.22]

One example is to use the finite automaton in Figure 3 to recognize input string “if”. Starting from State 1, the character *‘i’* is scanned and compared with all the labels of

edges coming out from State 1. Apparently, the character *'i'* matches the label of Edge 1-2 which means that the scanner follows the Edge 1-2 to navigate from State 1 to State 2. Next, the same process is executed with the character 'f' and the lexer moves from State 2 to State 3 this time. At this point, the finite automaton checks if the current state (State 3) is the final state since all the input characters were scanned. Fortunately, State 3 is accepted which means that the input string “if" is valid and token “IF" is returned. Another example is when running the same algorithm against the input string “ifabc”, the result State is 4 thus “ifabc" is recognized as token ID (variable name). In contrast, if the input string is **“a}”**, the execution is stuck at 4 since there is no edge whose label is **“}”** coming out of State 4. Hence, **“a}”** is not an acceptable token in this language.

This finite automaton is categorized as deterministic finite automaton (DFA) since it satisfies two conditions. Firstly, at any state in the DFA, a single input character navigates the program from one state to at most one new state. Secondly, there is no empty-labeled transition (epsilon edge) in the graph. [5, p. 22]

DFA characteristic ensures that there is at most one edge to follow from State **A** with the given input character **k**. This nature prevents the situation when computers have to decide which path to follow from a given state. However, the process of converting regular expression to DFA directly is usually more complicated than translating regular expressions to another type of finite automata known as nondeterministic finite automata (NFA) and subsequently convert NFA to DFA. [5, p.18] [11, p.25]

2.2.2 Nonedeterministic finite automata (NFA)

Nondeterministic finite automata is a type of automata that allows multiple transitions coming from a state to share the same character label or empty character label. This means that at some particular states, computers might have to decide which path to follow out of multiple paths with the same label or it may follow epsilon transition without consuming any input character at all. Hence, computers cannot always rely on the current state and the given input character when selecting the new state. In reality, not all selectable transitions from one state lead to an accepted state. Due to this nondeterministic characteristic, an input string is regarded as valid token in NFA if there exists at least one possible path from starting state to an accepting state when following its characters [5, p.16]. An example of NFA is illustrated in Figure 4.

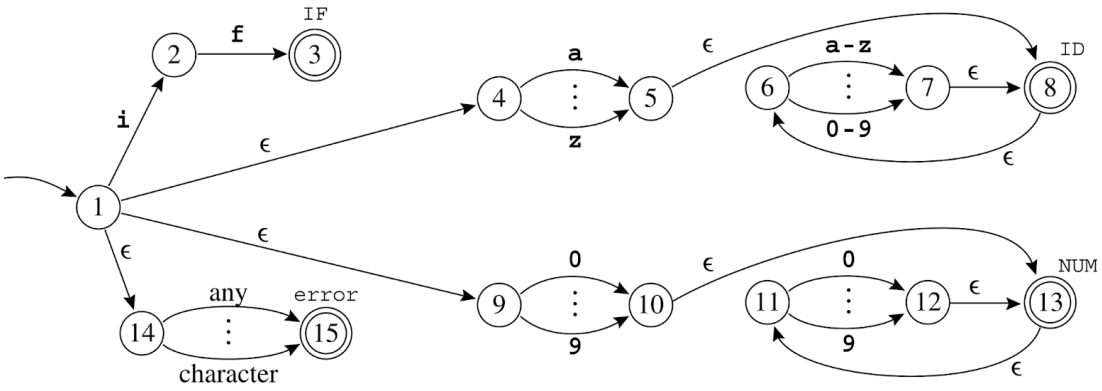

Figure 4. Visualization of nondeterministic finite automata [11, p.27]

An example of NFA, constructed from the regular expression *a*(a|b)*, is illustrated in Figure 5. In this case, starting state is labeled 1 and accepting state has label 3. The edge from State 1 to State 2 is epsilon edge which means that it does not consume any input character. Given the input string "aab", one possible path that accepts this input is 1-2-1-2-1-3. In contrast, if the program takes path 1-2-3, process is stuck at State 3 when the remaining input character is still {a, b}. As mentioned in the previous paragraph, any accepting path that begins from starting state and ends at final state is sufficient to consider string "aab" as a valid token according specification of regex *a*(a|b)*.

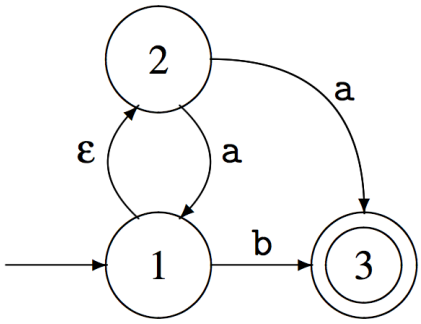

Figure 5. NFA of regular expression a*(a|b)

It is obvious that NFA closely resembles regular expression, which brings the convenience in the translation process from regular expression to NFA. However, employing NFA as an implementation for regular expression might not be an efficient approach since this solution requires examining the all possible paths or perform back-tracking

until one accepting path is found. Therefore, NFA, which is translated from Regex, needs to be subsequently converted to DFA for efficient execution. [5, p.18]

2.2.3 Translation algorithm from NFA to DFA

In short, the algorithm converts NFA to DFA by grouping a set of NFA states into an equivalent DFA state. Then it adds the transitions between DFA states. Finally, the algorithm assigns a valid token to be recognized to each newly created DFA state.

To formally define algorithm that translates NFA to DFA, several notations have to be used:

- **S** is a set of states **{s1, s2, s3,...}**
- **edge(s, c)** is a set of all NFA states that can be reached by following **only** the character **'c'** from State **s**
- **e-closure** or **closure(S)** is a set of all possible states that are reachable from each State s1, s2, s3... in set S by following **only** epsilon transitions. Take Figure 4 for example, if **S1 = { 1 }**, **closure(S1) = { 1, 4, 9, 14 }**. Similarly if **S2 = { 5 }**, **closure(S2) = { 5, 8, 6 }**. As a result, if **S3 = S1 U S2 = { 1, 5 }**, **closure(S3) =**

  $$T = S \cup \left(\bigcup_{s \in T} \mathbf{edge}(s, \epsilon) \right)$$

  { 1, 4, 9, 14, 5, 8, 6 }. Formal definition of **closure(S)**:
- **DFAedge(S, c)** is a set of all possible NFA states that are reachable from each State s1, s2, s3 in set S by following either character 'c' or epsilon transitions. For instance, if **S1 = { 1 }** then **DFAedge(S1, 'i') = { 1, 2, 4, 9, 14 }**. Another interesting example is when **S2 = {4}**, **DFAedge(S2, 'a') = { 4, 5, 8, 6 }**. Notice that closure({5}) is also a sub set of **DFAedge(S2, a)** because State 8 is reachable from State 5 without consuming any character. Formal definition of **DFAedge(S, c)** is as follows:

  $$\mathbf{DFAedge}(d, c) = \mathbf{closure}(\bigcup_{s \in d} \mathbf{edge}(s, c))$$

- **Σ** is the set of all valid characters

The implementation of this algorithm is shown in Figure 6 and it requires usage of two lists. Firstly, the list **states** is used to store created DFA states. Secondly, the list **trans** is a two-dimensional list used for storing DFA edges.

```
states[0] ← {};      states[1] ← closure({s1})
p ← 1;      j ← 0
while j ≤ p
  foreach c ∈ Σ
    e ← DFAedge(states[j], c)
    if e = states[i] for some i ≤ p
      then trans[j, c] ← i
      else p ← p + 1
           states[p] ← e
           trans[j, c] ← p
  j ← j + 1
```

Figure 6. Pseudo instructions of algorithm that converts NFA to DFA [11, p.29]

Once the complete DFA is constructed, each DFA state needs to be assigned with a token following the priority rule. This rule specifies that the algorithm scans through every token corresponding to each NFA node member of a DFA state and choose the one with the lowest index in the list of specified tokens. Finally, the result produced by this algorithm is visualized in Figure 7:

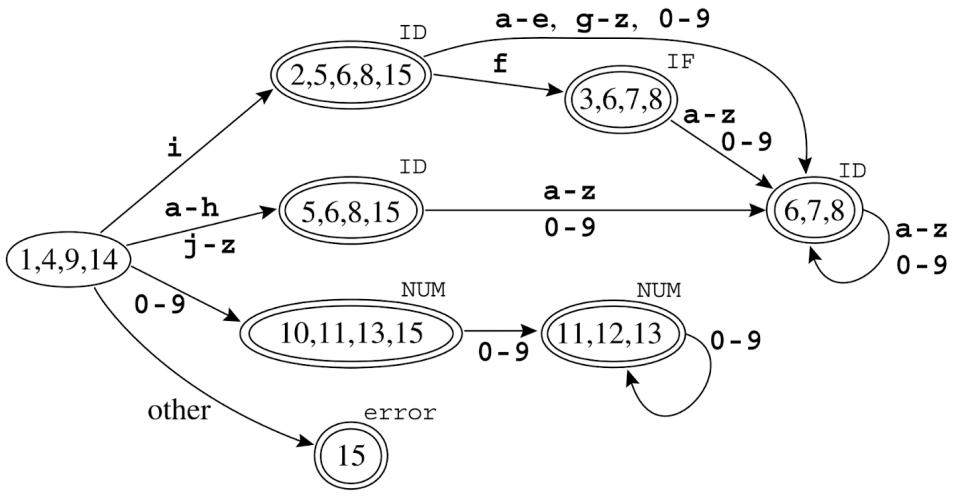

Figure 7. Visualization of the result DFA [11, p.29]

Once the scanner successfully partitions input string into a set of valid tokens, it sends those tokens together with their associated information such as values, positions in the source file to the next phrase - syntax analysis.

2.3 Lexing implementation

2.3.1 OCamllex

OCamllex is a tool to conduct lexical analysis in this project. Its task is to scan the stream of input character from the given file and recognize individual tokens based on regular expressions that are specified in file lexer.mll. Once a sequence of consecutive input characters matches a regular expression, Ocamllex executes the corresponding Ocaml code associated with that regular expression. This executed piece of code yields the type of token together with its associated value. Once the project is compiled, Ocamllex implicitly translates code in lexer.mll to a native Ocaml source file lexer.ml. Finally, the file lexer.ml is compiled and linked to the production binary. [13, p.1]

Ocamllex file is constructed by 4 parts: header, definitions, rules and trailer section. They are divided into mandatory category (header, rules) and optional one (definitions and trailer) [13, p.4]. The structure of file lexer.mll is illustrated in Figure 8:

```
(* header section *)
{ header }

(* definitions section *)
let ident = regexp
let ...

(* rules section *)
rule entrypoint [arg1... argn] = parse
  | pattern { action }
   | ...
   | pattern { action }
and entrypoint [arg1... argn] = parse
    ...
and ...

(* trailer section *)
{ trailer }
```

Figure 8. Structure of OCamllex file [13, p.4]

a. Header section. The header usually contains OCaml code to import from other modules, libraries. In addition, helper functions are usually defined in this section. When generating output file lexer.ml, this header section is copied to the beginning of the output file first. Similarly, trailer section also contains native OCaml code. However, this part is appended to the end of the output file only after all other parts of file *lexer.mll* have been processed. [13, p.4]

b. Definitions section. The definitions section is not compulsory since it is the place where regular expressions are given alias if needed. Those aliases can be used in rule section as compact substitutions for their corresponding regular expressions.

c. Rules section. The rules section is the most essential part since it contains a set of rules that specify the language's tokens. Each rule in this section has the form shown in Figure 9. In fact, token rules defined in this part are converted to native OCaml functions in the output file. [13, p.4]

```
rule entrypoint [arg1... argn] = parse
  | pattern { action }
   | ...
   | pattern { action }
and ...
```

Figure 9. Format of token rule [13, p.4]

The translated OCaml function that corresponds to entrypoint rule is depicted in Figure 10:

```
let entrypoint [arg1... argn] lexbuf =
  ...
and ...
```

Figure 10. Native OCaml function corresponding to entrypoint [13, p.10]

where lexbuf is the parameter for input stream.

Each rule consists of a sequence of patterns and their associated action in the following form:

```
| pattern { action }
```

where patterns are the placeholders for either regular expressions or their aliases. On the right-hand side of a pattern is an action section wrapped in curly braces. This action is the place for OCaml code that is executed when its corresponding pattern is matched.

2.3.2 Tiger tokens handling

a. Regular expressions of tokens:

Basic regular expressions for Tiger tokens are summarized in Table 2:

Table 2. Common regular expressions for Tiger's tokens

Regular expression pattern	**Meaning**
‘“’	Starting/ Ending character of a Tiger string
‘\n’	New line character
_	Any character
eof	End of file
“function”	Literal string for the keyword “function”
['0'-'9']+	Integer token
['a'-'z' 'A'-'Z']+(['a'-'z' 'A'-'Z'] | ['0'-'9'] | ‘_’)*	ID (variable name, type name)
“/*”	Starting comment
“*/”	Ending comment
“>=”	Greater than or equal operator

One observation is that all keywords of Tiger language (“function”, “type”, “if”, etc...) are specified using string literal regex. Another interesting token is ID token which contains a set of alphabet characters, digits and underscore character. However, the first character of ID token has to be an alphabet character.

b. Action:

Once a pattern is matched, the action code next to that pattern is executed as a result. In lexer.mll, most normal actions only return Tokens types, defined in Parser module, and their associated values. For instance, item | *"if" { P.IF }* returns *IF* token defined in Parser module *P*. Another interesting example is item | *id as value { P.ID(value) }* returns ID token which also contains a string value. This means that once the input string *"variable_name"* matches Regex of ID, a token *P.ID("variable_name")* is returned. In fact, most tokens in Tiger do not contain any value except for *ID(string), INT(integer), String(string)*. The list of tokens defined in Parser are shown in Figure 11:

```
%token <int>    INT
%token <string> STRING
%token <string> ID
%token          FOR WHILE BREAK LET IN NIL TO END
%token          FUNCTION VAR TYPE ARRAY IF THEN ELSE DO OF
%token          LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE
%token          DOT COLON COMMA SEMI
%token          PLUS MINUS TIMES DIVIDE UMINUS
%token          EQ NEQ LT LE GT GE
%token          AND OR
%token          ASSIGN
%token          EOF
```

Figure 11. Tiger's token declarations

c. White spaces and comments:

As discussed in theory part, the first responsibility of scanner is to eliminate white spaces and comments. Therefore, the actions corresponding to whitespace and comment patterns do not return any token. In practice, once the lexer encounters a white space character, it just ignores that character and jump to the next one. This could be simply implemented by recursively calling function **token(lexbuf)** to move to the next character when a white space character is matched. For instance, by calling function **token(lexbuf)** with the given input string **" if"**, the parser receives token **IF**. This is because character ' ' is matched in the first place but its own action recursively calls **token(lexbuf)** one more time which yields the **IF** token.

Comments in Tiger are enclosed between "/*" and "*/". However, one comment can be nested inside another. For instance, */* outer comment /* inner comment */ */* is valid in

Tiger. In addition, all characters in comments are discarded, which means that they do not yield any result token. This could be implemented using finite automata in Figure 12.

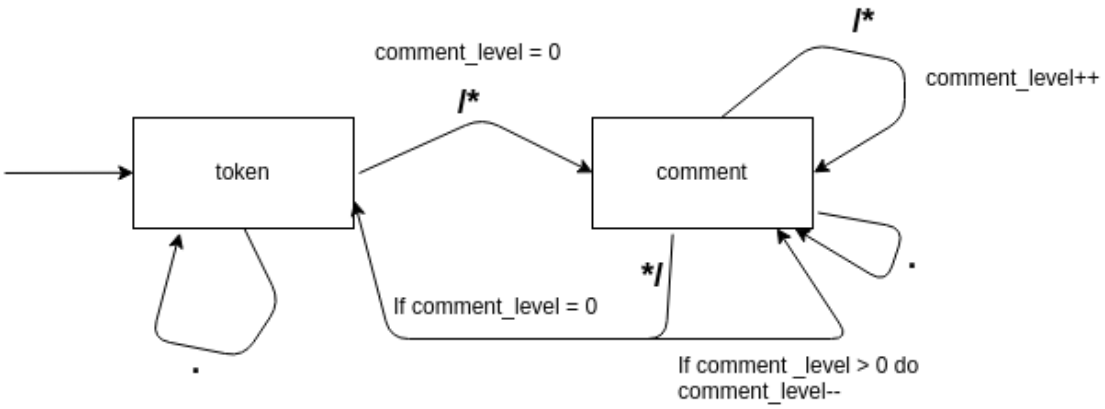

Figure 12. Finite automata to handle comments

Initially, the parser calls function *Lexer.token()* to scan for the next input token. In function *token()*, if '/*' is matched, its action code is executed. This action calls function ***comment()*** which is a new rule defined in rule section. At this point, the scanner changes its state from 'token' to 'comment' by following edge '**/***'. In order to implement nested comment, a stack-based solution is used to keep track of the current nested level of comment. In fact, when function *comment()* is called within the body of function *token()*, the comment's nested level is initialized to 0 and passed to function *comment()*. Once the scanner is in comment state, every time that it encounters '/*', the nested level of comment is incremented by one. On the other hand, the scanner decreases the nested level of comment by one when '*/' is matched. Once the comment level reaches zero, function *token()* is called to escape from comment state. Apart from '/*' and '*/', every other pattern encountered in comment rule is simply ignored. If the scanner reaches EOF character while the nested level is not 0, this means that the comment state does not escape thus an error exception needs to be raised.

In general, each rule (function) declared in the rule section of lexer.mll can represent a distinct DFA state. The transition from State *A* to State *B* is performed by calling the function *B()* when a certain regular expression pattern in State *A* is matched.

Similar to comment processing approach, the scanner enters state **string** when it matches character **"** and escapes this state by matching the same character. There-

fore, a similar rule (state/ function) **string** is added to the set of rules in order to process string. Unlike comment handling approach, once the scanner escapes *string* mode, it has to return token *STRING(value)* containing the whole scanned string. As a result, every character scanned in string mode has to be stored and finally returned on this state's transition.

Another special pattern in every state is new line character. In this project, every time a \n character is scanned, the function *incr_linenum()* is called to increase the line number and save the start position of the current line. In practice, those data are useful when producing informative error messages.

At the end of the process, the rule section of *lexer.mll* contains 3 rules: comment, string and token. Rule **token** is the main entrypoint of the lexer that is exposed directly to the Parser. This mean that function *token()* is called whenever Parser needs to scan for the next token in the input stream. The interaction between Parser and lexer was made by calling function *Parser.prog()* in the file *parse.ml*:

In fact, the type definition of Parser.prog is
Parser.prog: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> AST_value

This means that function *Parser.prog()* takes 2 arguments: the function closure *Lexer.token* from *lexer.mll* and *lexbuf*. The first argument *Lexer.token* is the entry function of Lexer module while the second argument is the stream of input characters.

3 Syntax analysis

As discussed in the previous chapter, the result of the lexical analysis phrase is a list of valid words specified in the language. However, the definition of a language is more than just words. Similar to natural languages, programming languages are constructed from a set of words (tokens) put together to form meaningful instructions by following a set of rules called grammars. Defining those grammatical rules is the core task of syntax analysis (Parsing) phase in compiler development process. Specifically, parsing is a procedure that transforms a linear list of tokens produced by the lexer into a hierarchical and meaningful data structure named Syntax tree for the future analysis. [5, p.54] [11, p.40]

3.1 Context-free grammar (CFG)

The grammar that is used to defines a programming language is called context-free grammar (CFG). It is a collection of rules that describe the hierarchical structure of a programming language. Each rule of CFG can be described in a form:
A -> aBdc...

The section on the left hand side of the '->' character is refered to as the head of the production. The head section of a CFG production contains at most one character. This character is in uppercase and also known as non-terminal symbol.

In contrast, the tail of the production is on the right-hand side of the arrow symbol. This part contains zero or more symbols. Each symbol in the tail of the production is either a terminal symbol (lowercase character) or non-terminal symbol (uppercase character). In practice, terminal symbols are the language's tokens returned by the scanner. Additionally, a special non-terminal symbol S is usually used as the starting symbol of the grammar [11, p.40]. One real example of CFG is

S -> if T then E else E
T -> true | false
E -> 1 | 0

In this example, S denotes the starting symbol of the grammar. Tokens **if, then, else, true, false, 1, 0** are terminal symbols. **T, E, S** are non-terminal symbols. Another interesting example that illustrates the recursive expressiveness of CFG is

S -> aS
S -> b

This grammar can equivalently express the regular expression **a*b**. The appearance of non-terminal symbol S in the tail of the first rule shows the recursive power of CFG. This means that the matched string could be "ab", "aab", "aaa...b".

By using the set of grammatical rules (CFG) for programming language, the parser can determine if a sentence is a valid sentence in that language. This process is called derivation which can be implemented by starting at the start symbol of CFG and repeatedly substituting any nonterminal symbol *X* with the tail of the production *X->tail*. Derivation can be represented as a sequence of products or as a parse tree. For instance, given the input string ***num * num + num*** and the grammar of an arithmetic expressions is:

S -> E
E -> num
*E -> E * E*
E -> E + E

One possible way to generating the string using this grammar is to start with the starting symbol *S*. First, *S* is replaced with its tail *E*. Next, *E* is expanded to *E + E* before the first *E* is consequently substituted with *E * E*. Finally, the expansion process is terminated by replacing all *E* symbols in the current result with *num*. The representation of this derivation in sequence:

S
-> E
-> E + E
*-> E * E + E*
*-> num * num + num* (This is the input string)

Alternatively, this derivation can be represented by syntax tree in Figure 13:

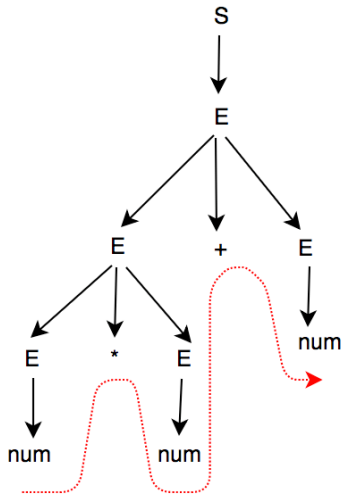

Figure 13. Derivation tree

3.2 Syntax tree

The syntax tree has several attributes. Firstly, its root node has the label **S** which is the starting symbol of the grammar. Secondly, each leaf of the tree is a terminal symbol which is a token return by the lexer. Another interesting characteristic of the syntax tree is that the sequence, generated by in-order traversing **leaf** nodes, is actually the original input string [5, p.60]. Finally, the syntax tree adds the association of operation to the derived string.

From the observation, an input string is grammatically correct if there exists at least one CFG derivation tree that yields the exact same string when performing in-order traversing on its leaf nodes. [5, p.60]

However, some grammar rules can produce more than one parse tree for a given input string. The grammar that produces those parse trees is ambiguous grammar [11, p.42]. In fact, some ambiguities are trivial as they do not adversely affect the parsing result while others might yield different results for compiling process. Thus, those adversely ambiguous grammars must be resolved appropriately. For instance, given the input string 1+ 2 * 3 and the grammar:

S -> E

E -> num

*E -> E * E*

E -> E + E

can yield two syntax tree shown in Figure 14. Each syntax tree produce different result when being evaluated: the left tree yields 9 while the right tree yields 7. Therefore, this grammar is adversely ambiguous.

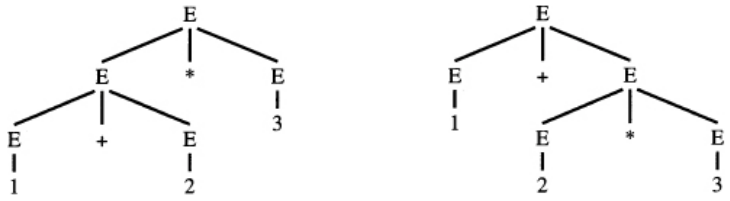

Figure 14. Two derivation trees caused by ambiguous grammar [11, p.43]

However, the primary objective of syntax analysis stage in compiler development is not checking the validity of a sentence in the given language. Instead, the goal of the parsing phrase is to build a syntax tree from the given sequence of input tokens and grammars of a language. [5, p.60]

3.3 Parsing algorithms

This part of the thesis demonstrates the implementation of the LR(1) algorithm. However, the two algorithms NULLABLE and FIRST have to be explored first since they are used in LR(1) algorithm.

3.3.1 NULLABLE, FIRST algorithms

NULLABLE is a function that take a string X and returns true if there exists at least one empty string derived from X

FIRST is a function that takes a string input and returns the set of all possible terminal symbols that can potentially appear as the first character in any output string derived from that input. [5, p.48]

For instance, given a set of grammar rules:

S -> aAb
A -> Bc
B -> '' | d
D -> Ef
E -> eG
G -> g

Obviously, FIRST (S) = {a} because 'a' is the first terminal character that appears in any string, derived from 'aAb', such as 'acb'. Another interestingly different case is when the non-terminal symbol *B* appears as the first character in the tail of the second rule. In this case, *nullable(B) = true* since *B* could derive either an empty string or token *d*. Therefore, those two cases need to be taken into consideration. In case that *B = ''*, *FIRST(A) = {c}* because B is empty thus ignored. In contrast, the case *B = d* yields the result *FIRST(A) = FIRST(B) = {d}*. Hence, *FIRST(A) = {c, d}* as it is the result of combining the previous two cases.

Pseudo code of this algorithm is depicted in Figure 15:

```
for each terminal symbol Z
    FIRST[Z] ← {Z}
repeat
    for each production X → Y1Y2 ··· Yk
        if Y1 ... Yk are all nullable (or if k = 0)
          then nullable[X] ← true
        for each i from 1 to k, each j from i + 1 to k
            if Y1 ··· Yi−1 are all nullable (or if i = 1)
              then FIRST[X] ← FIRST[X] ∪ FIRST[Yi]
```

Figure 15. Pseudocode of the algorithm computing FIRST [11, p.48]

3.3.2 LR(1) algorithm

Parsing algorithms can be categorized into two groups based on their parsing philosophies. The first group is top-down parsing algorithms which generates the parse tree from the starting symbol down to the leaves of the tree. In contrast, the bottom up parsing algorithms construct the parse tree by starting at the leaves and gradually

merge sub-trees into the bigger tree. In fact, the bottom up parsing techniques are more powerful than the former one since more types of grammar can be successfully parsed using bottom-up parsing algorithms [5, p.88]. The scope of this thesis focuses only on the bottom-up approach by examining the fundamental concepts and implementation of LR(1) algorithm.

LR(1) stands for left-to-right, rightmost-derivation, one-token lookahead algorithm. Fundamentally, LR(1) algorithm employs the power of deterministic finite automata (DFA) to construct its parsing table and uses stack to parse the input string. Each DFA state contains a set of grammar rules in the form of ***A -> a.B, N***. Specifically, each production in a DFA state consists of 2 parts separated by a comma: grammar rule ***A -> a.B*** and the set ***N*** of lookup characters. The dot character in grammar part ***A -> a.B*** indicates the current processing position of the parser. This means that character **'a'** is already processed and pushed to the stack while *B* has not been processed yet [5, p.63]. In addition, the character ***$*** is used to represent End-Of-File character. Shifting *$* means that the parser stops the parsing process successfully. [11, pp.55-56]

There are 2 main types of operations in LR(1): shift and reduce. Shift action pushes the input token to the top of the stack. Reduce action selects reducible grammar rule based on the tokens on top the stack, discards tokens in the tail of that rule from top of the stack and pushs the head token of that rule to the stack.

For instance, the grammar below is used to demonstrate the fundamental concept of this algorithm:

S -> AA
A -> aA | b

The first step of the algorithm is to add an augmented production ***S' -> .S, $*** to the initial DFA State **I0**. Next, the algorithm computes **Closure(I)** - set of all grammar productions that belongs to a DFA State **I** and corresponding look-ahead symbols. The algorithm to compute the **Closure(I)** is shown in Figure 16:

Closure(I) =
  repeat
    for any item $(A \to \alpha.X\beta, z)$ in I
      for any production $X \to \gamma$
        for any $w \in \text{FIRST}(\beta z)$
          $I \leftarrow I \cup \{(X \to .\gamma,\ w)\}$
  until I does not change
  return I

Goto(I, X) =
  $J \leftarrow \{\}$
  for any item $(A \to \alpha.X\beta,\ z)$ in I
    add $(A \to \alpha X.\beta,\ z)$ to J
  return Closure(J).

Figure 16. Algorithms to compute Closure and Goto [11, p.63]

At this point, the algorithm yields the following result (State I0 of DFA)

S' -> .S, $
S -> .AA, $
A -> .aA | b, a/b

Then, the algorithm uses function *GOTO(I, X)* to compute new states of DFA by moving the dot pass the following symbol *X* in every production of State I0 and computing the closure of the new states.

The result DFA at this point is shown in Figure 17:

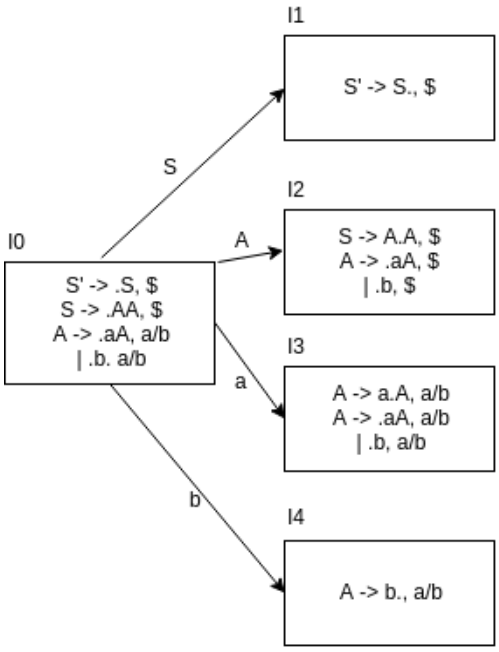

Figure 17. Current DFA states

This process is repeated until the algorithm reaches all final states in which all the dots are at the final position in the tail of each production. The final result is depicted in Figure 18:

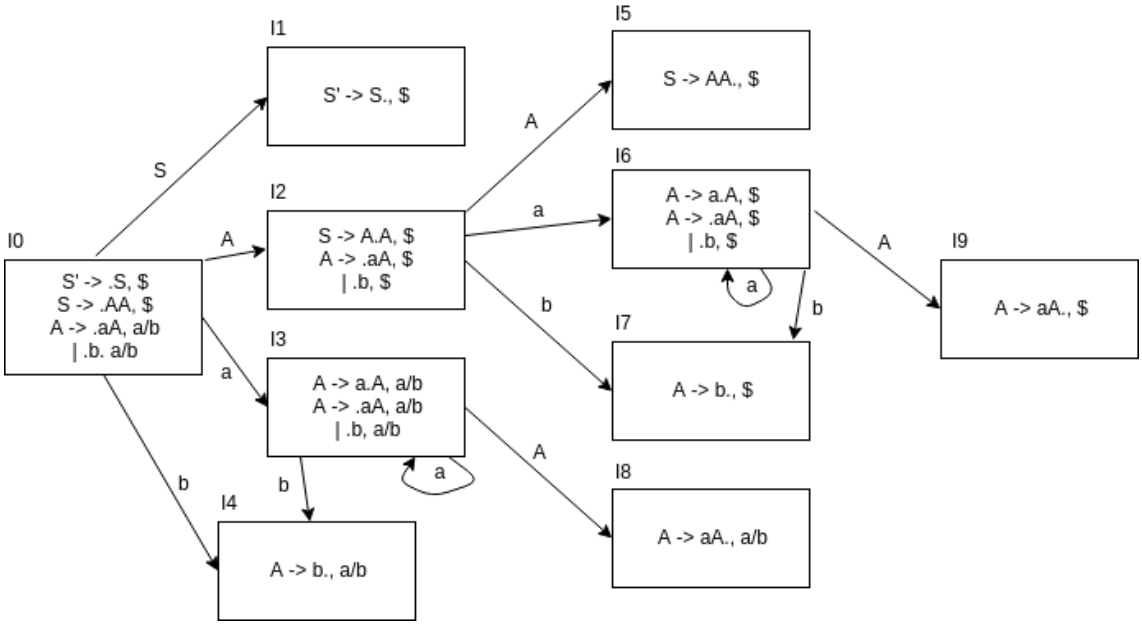

Figure 18. The complete DFA

Using this DFA, the parsing table that describes the transitions between DFA states can be seen in Table 3. In the parsing table, LR(1) uses information of the current state (row) and symbol (column) to look for correct actions to perform. There are several kinds of action in the table. Firstly, **s + <new state number>** denotes shift action that instructs the program to scan the next terminal character input and move to new state. Secondly, **g + <new state number>** denotes *GOTO* action which tells the program to follow non-terminal symbol *X* in order to reach new state. Thirdly, **r + <rule number>** means that the program should reduce the rule with that number. One crucial observation is that the reduced actions are only placed in final states (State 4, 5, 7, 8, 9) and only under the columns of their look-ahead symbols. For instance, reduce action **r1** is placed in State **I5** under **$** column because the item **S -> AA., $** is reduced. Finally, accept action indicates the successful termination of the parsing process. [5, p.89]

Table 3. The LR(1) parsing table computed from DFA

State\Symbol	a	b	$	S	A
0	s3	s4		g1	g2
1			accept		
2	s6	s7			g5
3	s3	s4			g8
4	r3	r3			
5			r1		
6	s6	s7			g9
7			r3		
8	r2	r2			
9			r2		

The next step is to use this parsing table and a stack to parse the input. For the convenience of demonstrating, the information about the current input token is not saved on the demo stack. This means that the stack in this example needs to store only the DFA state instead of the combination of DFA state and token as in some implementation. For instance, given the input string **"bab"**, the result in each step when parsing the input string is shown in Table 4:

Table 4. Inputs, actions perform in each LR(1) parsing step

Step	The remaining input at the beginning of each step	DFA state stack (bottom -> top)	Rule reduced in previous step	Action to perform in each step
1	bab$	0		s4
2	ab$	0-4		r3

3	ab$	0	A -> b	g2
4	ab$	0-2		s6
5	b$	0-2-6		s7
6	$	0-2-6-7		r3
7	$	0-2-6	A -> b	g9
8	$	0-2-6-9		r2
9	$	0-2	A -> aA	g5
10	$	0-2-5		r1
11	$	0	S -> AA	g1
12	$	0-1		accept

In each step of the parsing process, LR(1) algorithm usually uses the state on the top of the stack and the first character of the remaining input to perform a lookup in the parsing table. Based on the result of the lookup, LR(1) conducts shift/reduce action with that input character. Each time the algorithm conducts shift action, it consumes an input character and push the new state onto the stack. When performing a reduce action, **N** states on the top of the stack are removed with **N** equals to the number of tokens in the tail of the reduced grammar rule. For example, the rule **A -> aA** is reduced in step 8 causing the removals of **two** States 6 and 9 on the stack as can be seen in step 9. Simultaneously, parts of the parse tree are also constructed as a result of reduce actions in step 2-6-8-10. Right after each reduction, the program moves to the new state by performing lookup using state from the top of the current stack and the non-terminal symbol on the left-hand side of the reduced rule. For example, in step 9, the program moves to State 5 by using lookup information in the parsing table of State 2 and symbol A. Finally, the program successfully parses the string “bab” after 12 steps by performing accept action. Snapshots of the parse tree after each reduction are depicted in Figure 19:

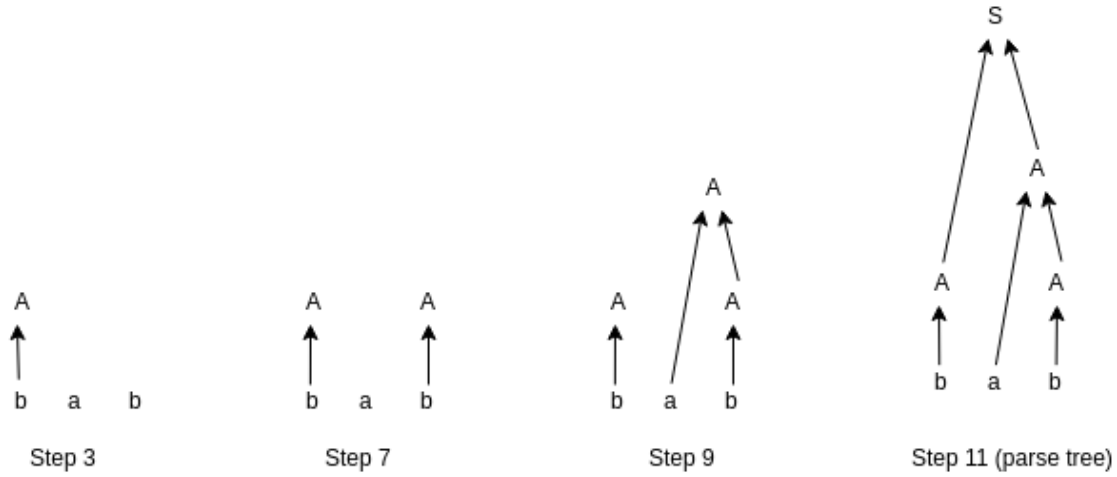

Figure 19. Snapshots of the syntax tree after each reduce action

Resolve conflicts:

Unfortunately, there are situations in which one cell of the parsing table contains more than one action [5, p.94]. For instance, it may contain 2 reduce actions (reduce-reduce conflicts) or a shift and a reduce action at once (shift-reduce conflicts). Those cases are usually the result of parsing ambiguous grammars. In order to yield correct output, the parser needs to devise strategy for choosing which action to perform in such situations. One technique is using rule of precedence to resolve trivial conflicts. One example is when parsing string *"3 * 4 + 5"*, the program certainly reaches the position *"3*4.+5"* at some points. At that point, the parser faces shift-reduce conflict, in which it has to choose whether to reduce *3*4* or shift state with character '+'. In fact, Shifting and reducing produces 2 different parse trees in this case. As a result, reducing yields 17 as an evaluated result while shift action produces 27. Therefore, the parser needs to specify that the character ***** has higher priority than the character **+** thus it prefers reducing 3*4 over shifting. Another common case when parsing the grammar of a programming language is the If-then-else statements:

S -> if E then E
S -> If E then E else E

In this case, the parser should give the token **else** higher precedence so as to perform shift action instead of reduce action. [5, p.97]

3.4 Abstract syntax tree (AST)

As discussed previously, each terminal input token is represented by a leaf node in the syntax tree. While some terminal tokens such as parentheses are useful in the parsing process since they convey special meanings, the corresponding nodes of those tokens are completely redundant for the analyses in other upcoming phrases. Therefore, those irrelevant nodes should be eliminated in the syntax tree. Such tree is called Abstract Syntax Tree (AST). In fact, Abstract syntax tree is a refined concrete syntax tree that contains only useful information for other future phrases in compiling process. [5, p.99]

For instance, parsing input *3 * (4 +5)* yields the concrete syntax tree in Figure 20. Then this syntax tree can be converted to the equivalent abstract syntax tree in the same figure without losing any important semantic meaning in the future analyses.

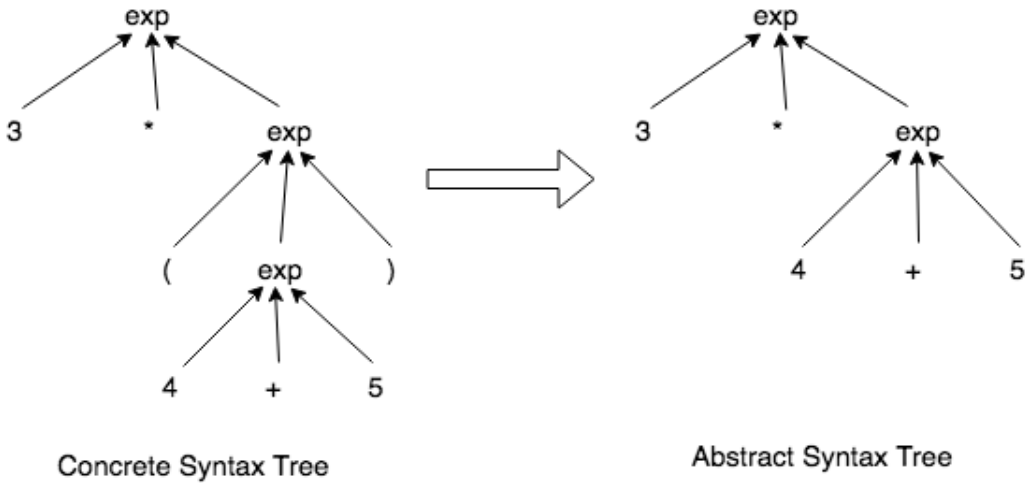

Figure 20. Concrete Syntax Tree and equivalent Abstract Syntax Tree

3.5 Parser implementation

3.5.1 Menhir

The process of parsing is actually tedious and repetitive. Fortunately, this process can be simply automated by using parser generator such as YACC or BISON [14][15]. This project uses Menhir - a successor parser tool of Ocamlyacc. Menhir employs **LR(1)** algorithm as its core parsing algorithm. [16, p.4]

The logic of the parser is specified in the file **parser.mly**. This file contains 3 parts: header, tokens and rules as shown in Figure 21:

```
%{
  (* header *)
%}
  (* token declarations *)
%%
  (* rules *)
%%
```

Figure 21.Structure of file parser.mly

Header. The header section of Menhir is enclosed between **%{** and **%}**. This section has the similar responsibility as the header of Ocamllex since both of them contains native OCaml code which imports dependencies and declares utility functions, local variables. [16, p.7]

Token. The token section is constructed from 3 parts: a set of token declarations and a set of tokens' priorities - associativity rules and the starting symbol declaration shown in Figure 22:

```
(* token declarations *)
%token <int>    INT
%token <string> STRING
%token <string> ID
%token          FOR WHILE BREAK LET IN NIL TO END
%token          FUNCTION VAR TYPE ARRAY IF THEN ELSE DO OF
%token          LPAREN RPAREN LBRACK RBRACK LBRACE RBRACE
%token          DOT COLON COMMA SEMI
%token          PLUS MINUS TIMES DIVIDE UMINUS
%token          EQ NEQ LT LE GT GE
%token          AND OR
%token          ASSIGN
%token          EOF

(* Rules of precedence, associativity *)
%nonassoc ASSIGN
%left OR
%left AND
%right THEN
%right ELSE

%nonassoc EQ NEQ LT LE GT GE
%left PLUS MINUS
%left TIMES DIVIDE
%left UMINUS

(* Starting symbol declaration *)
%start prog
%type <Absyn.exp> prog
```

Figure 22. Token declarations of Tiger

In the token declarations part, each token declaration has the following format:
%token <ocaml_type> token_name,
where <*ocaml_type*> denotes the type of the semantic value carried by token *token_name*. For instance, *%token <int> INT* means that the token *INT* contains an integer value. If the token does not contain any value, <*ocaml_type*> can be omitted. [16, p.9]

Tokens' priority rules define the level of precedence and associativity for tokens. This information is useful when resolving simple shift/reduce conflicts. The template for this section is:

%nonassoc A B ...
%left C D ...
%right E ...,

where **%left X** indicates that operation on token X is left associative (reduce preference) while **%right X** means that X is right associative (shift preference). For example, the parser certainly faces shift-reduce conflicts when parsing ambiguous expression **5 - 4 - 3**. In fact, this expression can be interpreted as **(5 - 4) - 3** if **%left MINUS** is declared since **%left MINUS** prefers reducing to shifting. In contrast, if **%right MINUS** is in use, the evaluated result would be **5 - (4 - 3)** since **%right MINUS** prefers shift action over reduce action. In addition, **%nonassoc X** is used to indicate that there is no association in the grammar *exp X exp X exp*. For instance, applying **%nonassoc ASSIGN** throws syntax error when parsing expression **a := b := 5**.

Nevertheless, even if **%left PLUS MULTIPLY** is defined on the same line, the parser still could not decide whether to shift or reduce when parsing the expression 5 * 4 + 3. In fact, the shifting on character '+' yields the result of 35 while 23 is the result of reducing on character '*'. Therefore, Menhir provides a way to set the priority values of tokens by the order of priority declaration. In the example above, token C and D have higher priority value than token A or B since rules of A, B are declared prior to the rule declarations of C and D. Thus, the declarations

%left PLUS
%left MULTIPLY

give **MULTIPLY** token higher priority compared to **PLUS**. When rules of two tokens are declared on the same line, they have equal priority values. For example, the rule declaration **%left PLUS MULTIPLY** gives **PLUS** and **MULTIPLY** the same priority value.

The last part of token section is to define the starting symbol of the parser. The declaration **%start < OCaml type > entrypoint** specifies the entrypoint function of the parser (starting symbol) which is called outside Parser module in order to generate the Abstract Syntax Tree. In this project, **%start <Absyn.exp> prog** is used to specify that function *prog()* is exposed to other modules and this function call returns AST node of type *Absyn.exp*. As discussed in implementation of lexical analysis, the type definition of function *prog()* in the output file:

Parser.prog: (Lexing.lexbuf -> token) -> Lexing.lexbuf -> Absyn.exp

Rules section. The rules section is surrounded by **%** characters and it is the most important parts in file *parser.mly*. In general, the anatomy of this section resembles the rule section in lexical analysis. This section is constructed from a set of rules, each of which is a group of all Context-free grammar productions for a given non-terminal symbol. Next to each production is the corresponding semantic action. This action is executed once the grammar production is reduced into the non-terminal symbol in its head. In fact, the OCaml semantic action of one production can have access to the computed semantic values of symbols in its production's tail. Semantic value of symbol X can be accessed using the query $<index of symbol X in production's tail> [16, p.11]. One example of a parser rule is:

```
exp:
   INT            { Absyn.IntExp $1 }
 | exp PLUS exp   { Absyn.OpExp({
                                left=$1; oper= Absyn.PlusOp;
                                right= $3; pos= $startofs}) }
```

In the semantic action of the first production, **$1** is the reference to the semantic value of token *INT* returned by the lexer. In fact, when symbol *INT* is reduced into non-terminal symbol *exp*, its semantic value *Absyn.IntExp($1)* is returned. Similarly, the semantic values of two *exp* symbols in the second production can be extracted by using keywords ***$1*** and ***$3***. Actually, **1** and **3** are the indexes of the first and second symbol *exp* respectively in the tail of the second production. In addition, *$startofs* returns the start position of the first symbol on the right-hand side of production. This position is useful

when generating error messages in the following phases of compiler. From the observation of this example, the returned value of the second semantic action can be constructed by combining some semantic values of its matched symbols.

Specifically, the sole responsibility of the rule section in *parser.mly* is to construct the AST tree from CFG. This could be implemented by mapping each grammar production to its corresponding AST node. Therefore, the returned values of most semantic actions in this project are indeed AST tree nodes. Once the starting production

prog -> exp EOF is reduced, the root node of the complete AST is returned.

To understand the mapping logic between Tiger's grammars and the data structures of Tiger's Abstract Syntax Tree, those two subjects have to be examined.

3.5.2 Context-free grammar of Tiger

CFG of Tiger is shown in Figure 23, in which *dec*+ means that non-terminal symbol *dec* repeats at least once. Additionally, $exp^{*;}$ denotes that *exp* repeats zero or moretime, and each repetition of *exp* is immediately followed by *semicolon* character.

$$
\begin{array}{lcl}
program & \rightarrow & exp \\
exp & \rightarrow & letExp \mid seqExp \mid ifThenElse \mid whileExp \mid forExp \\
 & \mid & assign \mid lValue \mid callExp \mid infixExp \mid neg \mid arrCreate \mid recCreate \\
 & \mid & \texttt{nil} \mid \textbf{INT} \mid \textbf{STRING} \mid \texttt{break} \\
 & & \\
letExp & \rightarrow & \texttt{let}\ dec^{+}\ \texttt{in}\ exp^{*;}\ \texttt{end} \\
seqExp & \rightarrow & \texttt{(}\ exp^{*;}\ \texttt{)} \\
ifThenElse & \rightarrow & \texttt{if}\ exp\ \texttt{then}\ exp\ [\ \texttt{else}\ exp\ ] \\
whileExp & \rightarrow & \texttt{while}\ exp\ \texttt{do}\ exp \\
forExp & \rightarrow & \texttt{for}\ \textbf{ID}\ \texttt{:=}\ exp\ \texttt{to}\ exp\ \texttt{do}\ exp \\
 & & \\
dec & \rightarrow & funDec \mid varDec \mid tyDec \\
funDec & \rightarrow & \texttt{function}\ \textbf{ID}\ \texttt{(}\ fieldDec^{*,}\ \texttt{)}\ [\ \texttt{:}\ \textbf{TYID}\ ]\ \texttt{=}\ exp \\
varDec & \rightarrow & \texttt{var}\ \textbf{ID}\ [\ \texttt{:}\ \textbf{TYID}\ ]\ \texttt{:=}\ exp \\
tyDec & \rightarrow & \texttt{type}\ \textbf{TYID}\ \texttt{=}\ ty \\
 & & \\
ty & \rightarrow & \textbf{TYID} \mid arrayTy \mid recTy \\
arrayTy & \rightarrow & \texttt{array of}\ \textbf{ID} \\
recTy & \rightarrow & \texttt{\{}\ fieldDec^{*,}\ \texttt{\}} \\
fieldDec & \rightarrow & \textbf{ID}\ \texttt{:}\ \textbf{TYID} \\
 & & \\
assign & \rightarrow & lValue\ \texttt{:=}\ exp \\
lValue & \rightarrow & \textbf{ID} \mid subscript \mid fieldExp \\
subscript & \rightarrow & lValue\ \texttt{[}\ exp\ \texttt{]} \\
fieldExp & \rightarrow & lValue\ \texttt{.}\ \textbf{ID} \\
callExp & \rightarrow & \textbf{ID}\ \texttt{(}\ exp^{*,}\ \texttt{)} \\
infixExp & \rightarrow & exp\ \textbf{OPER}\ exp \\
neg & \rightarrow & \texttt{-}\ exp \\
 & & \\
arrCreate & \rightarrow & \textbf{TYID}\ \texttt{[}\ exp\ \texttt{]}\ \texttt{of}\ exp \\
recCreate & \rightarrow & \textbf{TYID}\ \texttt{\{}\ fieldInit^{*,}\ \texttt{\}}
\end{array}
$$

Figure 23. Context-free grammar of Tiger programming language [1, p.1]

3.5.3 Abstract syntax tree of Tiger

Figure 24 depicts the data types for Tiger's AST specified in file absyn.ml.

```
structure Absyn = struct
type pos = int   and   symbol = Symbol.symbol

datatype var = SimpleVar of symbol * pos
             | FieldVar of var * symbol * pos
             | SubscriptVar of var * exp * pos
and exp =
   VarExp of var
 | NilExp
 | IntExp of int
 | StringExp of string * pos
 | CallExp of {func: symbol, args: exp list, pos: pos}
 | OpExp of {left: exp, oper: oper, right: exp, pos: pos}
 | RecordExp of {fields: (symbol * exp * pos) list, typ: symbol, pos: pos}
 | SeqExp of (exp * pos) list
 | AssignExp of {var: var, exp: exp, pos: pos}
 | IfExp of {test:exp, then': exp, else': exp option, pos:pos}
 | WhileExp of {test: exp, body: exp, pos: pos}
 | ForExp of {var: symbol, escape: bool ref,
                    lo: exp, hi: exp, body: exp, pos: pos}
 | BreakExp of pos
 | LetExp of {decs: dec list, body: exp, pos: pos}
 | ArrayExp of {typ: symbol, size: exp, init: exp, pos: pos}

and dec = FunctionDec of fundec list
        | VarDec of {name: symbol, escape: bool ref,
                     typ: (symbol * pos) option, init: exp, pos: pos}
        | TypeDec of {name: symbol, ty: ty, pos:pos} list

and ty = NameTy of symbol * pos
       | RecordTy of field list
       | ArrayTy of symbol * pos

and oper = PlusOp | MinusOp | TimesOp | DivideOp
         | EqOp | NeqOp | LtOp | LeOp | GtOp | GeOp

withtype field = {name: symbol, escape: bool ref, typ: symbol, pos: pos}
   and   fundec = {name: symbol, params: field list,
                   result: (symbol * pos) option,
                   body: exp, pos: pos}
end
```

Figure 24. Data structures for Tiger's Abstract Syntax Tree [17]

In details,

- *VarExp* is data structure of AST node for any variable in Tiger. There are three kinds of variable in Tiger: *SimpleVar* (variable x), *FieldVar* (object.field) and *SubscriptVar* (array[index]). Notice that the first argument of *FieldVar* and *SubscriptVar* is also a *var* due to the recursive nature of this data structure, which allows a chain of data extraction from record and array. For example, The

expression *object.array[0].field* is converted to

```
FieldVar(
    SubscriptVar(
        FieldVar(
            SimpleVar("object", pos),
            "array",
            pos
        ),
        IntExp(0),
        pos
    ),
    "field",
    pos
)
```

- *CallExp* is AST node for function call. This node contains the function's name and a list of argument expressions. For instance, *sum(1, 2)* is translated to

```
CallExp({
  func = "sum";
  args = [ IntExp(1); IntExp(2) ];
  pos = pos_sum })
```

- *OpExp* is AST for arithmetic operator between two expressions such as *1 + 2* or comparison operator such as *1 < 2*.

- *RecordExp* contains a list of expression fields declared in that record and their types.

- *SeqExp* contains a list of expressions.

- *AssignExp* contains two expressions: left-hand side expression of the assignment (VarExp) and a value expression on the right-hand side of the assignment

- *IfExp* contains a condition expression, a *then* expression and an optional *else* expression. For instance, *if (1 < 2) then 1* is translated to

```
IfExp({
        test = OpExp({
                        left = IntExp(1);
                        oper = LtOp;
                        right = IntExp(2);
                        pos = position
                    });
        then' = IntExp(1);
        else' = None
})
```

- *ArrayExp* contains the type of the array's elements, the size of the array and the initial value of all array's elements.

- *LetExp* is special since it contains a list of variable declarations (*VarDec*), function declarations (*FunctionDec*) and type declarations (*TypeDec*). Interestingly, functions declared adjacent to each other are treated as mutually recursive functions in Tiger. This means that one function in the set of mutually recursive functions can call any other function within that set regardless of their declaration's order. For instance, given the program

```
let
  function a(): int = b() + 1
  function b(): int = 2
in
  a()
end
```

Function *a()* and *b()* are mutually recursive when function *b()* is callable in the body of function *a()* even though function *b()* is declared after the declaration of function a.

In order to support mutual recursion, *FunctionDec* has to contains a list of consecutive function declarations instead of a single function. Similarly, *TypeDec* also contains a list of consecutive type declarations to support mutually recursive type declarations. One such example is

```
let
  type a = b
  type b = c
  type c = int
in
  ...
end
```

Additionally, *exp1 OR exp2* is translated to *if exp1 then exp2 else 0*. Similarly, *exp1 AND exp2* is translated to *if exp1 then 1 else exp2*.

Moreover, almost all AST nodes has field ***'pos'*** containing their approximate positions in the original source file. This is because position information is useful when generating informative error messages in other upcoming phrases of the compiler (tiger 93)

Once having the understanding about data structure of AST nodes, the challenge is to map each production to its corresponding AST node.

3.5.4 Tiger mapping logic between CFG and AST

The implementation of most productions in the file parse.ml are fairly straight forward; each semantic action returns an AST node. One such example are semantic actions in the example x. However, there are cases where semantic action returns function instead. For example, this grammar rule that parses three types of variable in Tiger known as *SimpleVar*, *FieldVar* and *SubscriptVar:*

```
lvalue:
  ID lvaluetail                    { $2(Absyn.SimpleVar(Symbol.symbol($1), $startofs)) }

lvaluetail:
/* empty */                        { fun x -> x }
| DOT ID lvaluetail                { fun var -> $3(Absyn.FieldVar(
                                                   var, Symbol.symbol($2), $startofs)) }
| LBRACK exp RBRACK lvaluetail     { fun var -> $4(Absyn.SubscriptVar(var, $2, $startofs)) }
```

where each semantic action of *lvaluetail* returns a chain of lambda functions. Each function in the chain takes an AST node *Absyn.var* as argument. The function in in the first production returns the same *Absyn.var* node. In the second or third production of *lvaluetail*, the returned function returns AST node *FieldVar* or *SubscriptVar* that wraps

its parameter *Absyn.var* node. Finally, once the chain is called in the semantic action of *lvalue*, the complete AST subtrees are built.

In addition, there are also semantic actions that returns list as value. For example, the rules in Figure 25 are used to parse the arguments of Tiger functions. The result of parsing arguments of a function is a list of corresponding AST nodes.

```
args:
  /* empty */                          { [] }
| exp                                  { $1 :: [] }
| exp COMMA args_rest                  { $1 :: $3 }
;

args_rest:
  exp                                  { $1 :: [] }
| exp COMMA args_rest                  { $1 :: $3 }
```

Figure 25. Rules to parse the arguments of Tiger functions

In summary, the syntax analysis phrase is a process that take a stream of input tokens, perform parsing based on context-free grammar rules to produce a complete Abstract syntax tree. This syntax tree abstractedly represents the structure of the original source program; thus, it can be used for the further analysis in future phases of compiling process.

4 Semantic analysis

Previously, the syntax analysis phase detects syntactical errors and yields an abstract syntax tree. However, the error checking mechanism in syntax analysis has its own limitation since Context-free grammars cannot naturally handle semantic errors [11, p.76]. For instance, the following Java program is syntactically correct:

int a = 1;

String b = "foobar";

int c = a - b;

However, this program raises a semantic error as the type of variable *a* is *integer* and variable *b* has type *string* while "-" operator only accepts numerical types.

As a result, an additional phase known as semantic analysis is conducted on the parsed AST tree to detect semantic errors during compile time. One responsibility of this process is to ensure that a variable is only used within its scope - a portion of the program (AST sub-tree) in which that variable is accessible. Additionally, semantic analysis also performs type-checking on expressions and subsequently calls other modules to generate intermediate code. In practice, semantic analysis can be conducted when pre-orderly traversing the AST tree from root node to leaves. [11, p.103]

4.1 Rules of variable scope in Tiger

Variables, types and functions in Tiger are declared in the **D** section of the expression *let **D** in **body** end*

One example of Tiger declarations is shown in Figure 26:

```
1  let
2    var a := 1       (* a is accessible until the programm reach line 17 *)
3
4    function f(b: int): int =
5      let
6        var c := 3
7        function g(): int = c - a
8      in
9        g();
10       a + b + c
11     end
12
13   var d := f(2)
14 in
15   print(a);        (* This line prints 1 *)
16   print(d);        (* This line prints 6 *)
17   print(b)         (* This line throws error since b is not accessible *)
18   print(c)         (* This line throws error since c is not accessible *)
19 end
```

Figure 26. Let expression in Tiger

In Figure 26, the variable *a*, declared on line number 2, is accessible in any subsequent declarations, expressions until the end of the let expression's body on line 19. Hence, the scope of variable *a* is between line 2 and line 19. Similarly, the scope of variable *d* is 13-19.

Besides that, the argument variable *b* of a function *f()* can only be accessed within the body of that function. For instance, variable ***b*** can only be used from line 5 to line 11. Interestingly, Tiger has support for nested function declarations, which means that the body of a nested function has access to the variables previously declared in the outer environment [11, p.117]. For instance, the body of the nested function ***g()*** on line 7 can access variable ***c*** declared in the body of function ***f()***, and variable ***a*** declared in the outer scope.

4.2 Symbol table

Generally, scoping rules can be implemented using symbol mapping table known as bindings environment. This table is a key-value based data structure that maps the names of variables to their corresponding values. First, the algorithm traverses the AST tree from root node to leaf nodes, if it encounters an AST node ***VarDec("x", ...)*** which represents variable declaration ***var x := value***, it adds the binding ***x -> value*** to the value bindings environment ***v_env***. Similarly, the program also adds the argument bindings ***x -> value*** to the environment ***v_env*** if the AST node ***FunctionDec***, representing function declaration ***f(x,...)***, is detected. During the course of traversing, if

the AST node SimpleVar(x) is detected, meaning that variable *x* is used, the algorithm looks for the value of *x* in table *v_env*. If value of x is not found in the lookup table, this means that variable *x* has not been declared or it is used outside of its scope; thus, the algorithm throws an error exception. [5, p.114]

However, the mapping *x -> value* must be discarded from the binding environment after its scope ends to ensure that variable x can not be used outside of its scope. In Tiger, if a variable *x* is declared in the **D** section of the expression ***let*** *D* ***in*** *body* ***end***, the scope of *x* terminates after evaluating ***body*** expression. Additionally, if *x* is the argument variable in function ***f(x, …) = body*** , the mapping of x has to be cleared after evaluating the body of function *f()*. There are two practical implementations of the binding environment: imperative hash table and functional map. [11, p.106]

4.2.1 Imperative hash table

In an imperative implementation, the compiler often uses a chaining hash table to store the mapping between a variable and its value [11, p.106]. In fact, when the mapping *x -> value* is added to the table, a hash function takes string *"x"* and generates the corresponding index ***i*** in the bucket array. Then the value of *x* is added to the head of the linked list at bucket ***ith*** in the array while the remaining nodes of the linked list is kept intact. Similarly, the same hashing process occurs when retrieving value of variable *x* from the hash table. However, the function *search(x)* only returns the first matched value of variable *x* that it finds when scanning the linked list from head to tail. This search functionality allows variables declared in the inner scopes to shadow their previous declaration in outer scopes [5, p.114]. When the scope of the variable *x* ends, the program search for only the first occurrence of *x* in the hash table and remove this value from the linked list. For instance, consider the program shown in Figure 27:

```
1  let
2    var a := 5
3    function f() = let var a := 6 in print (a) end  (* This line prints 6 *)
4  in
5    f();
6    print (a)        (* This line prints 5  *)
7  end
```

Figure 27. Variable, function declarations in Tiger

The scoping hash table implementation of the program in Figure 27 is illustrated in Figure 28:

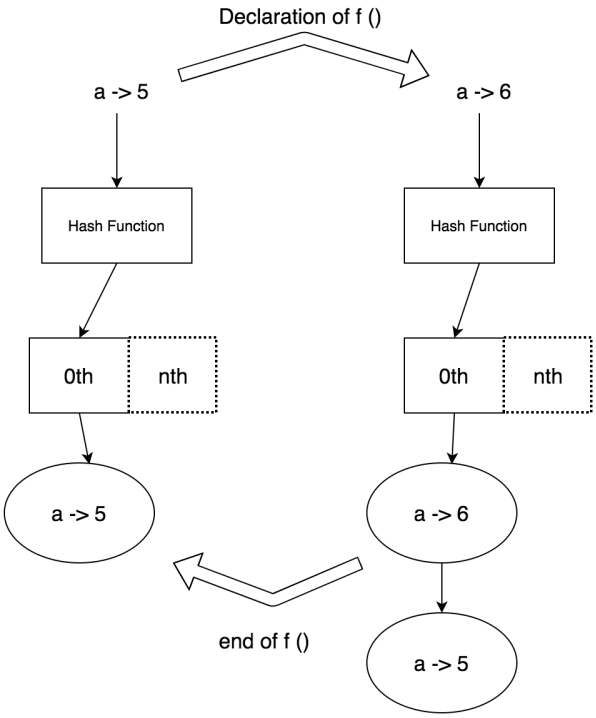

Figure 28. Example of scoping hash table implementation

In Figure 28, the binding *a -> 5* is added to the head of the linked list at array's bucket 0th once the program executes the declaration of variable *a* in line 2. Then, the let expression in the body of function *f()* is evaluated on line 3 which adds the binding *a -> 6* to the head of the old linked list while the mapping a -> 5 is still untouched. After evaluating the body of the let expression in the body of function *f()*, the binding *a -> 6* is removed from the head of the linked list. As a consequence, variable *a* is evaluated to 5 from line 4 to line 7.

In order to remember which variable mappings must be removed from the hash table when a scope exits, a helper stack is used to keep track of the added variables of that scope. In practice, a special marker is pushed to the top of the stack so as to mark the beginning of a new scope. Then, the algorithm adds the variables declared in that new scope to the top of the stack. After the scope has ended, the algorithm pops every variable off the stack until it removes the starting marker of that scope. Simultaneously, all the popped variables are discarded from the hash table so as to restore the state of the scoping table as it was before entering that scope. Therefore, the variables declared within inner scopes are not accessible in outer scopes. [5, p.115]

4.2.2 Functional map

In functional approach, when the program enters a new scope with the declaration for variable *x*, a binding *x -> value* is also added to the binding environment *X*. However, a new binding environment *X' = X + (x -> value)* is created while *X* is not mutated [11, p.107]. As a result, the environment *X'* can then be used to perform scope-checking until the current scope terminates. Because the original environment *X* of the outer scope remains intact, the program does not need to perform table restoration as it does in the imperative implementation when exiting a scope. One efficient implementation of this immutable data structure is to use a self-balanced binary search tree such as red-black tree that can perform insertion and search with average time complexity of O(log n) [11, p.108]. One simple example of this immutable data structure is demonstrated in Figure 29:

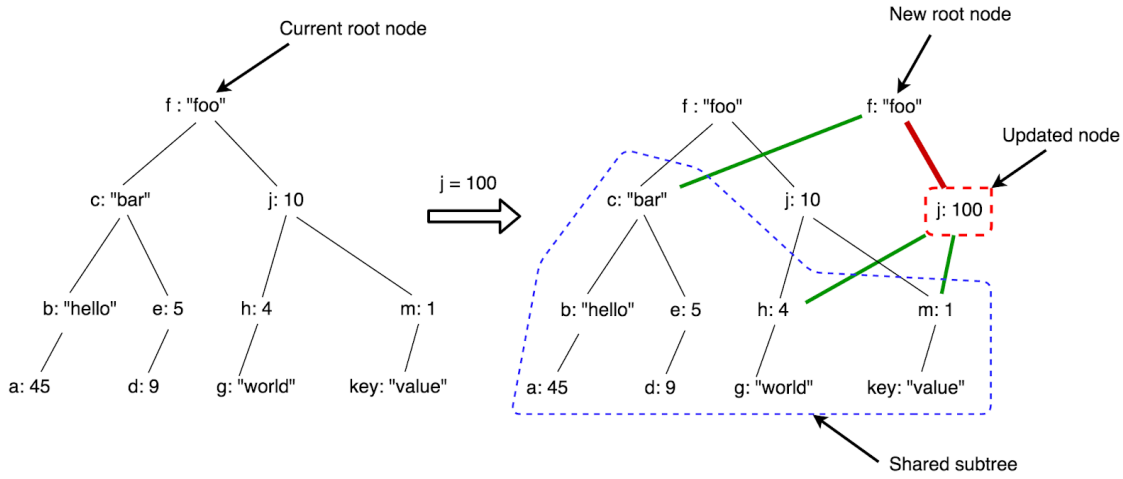

Figure 29. Immutable map

Initial, the program points to the root node f -> "foo" of the binary tree. When a binding *j -> 100* is added to the environment, the algorithm compares the key *j* with the key of the root node *f*. If the value of key *j* is greater than that of key *f*, the binding *j -> 100* is certainly added to the right subtree, which means that the right subtree has to be re-constructed. As all the nodes of the original tree are immutable, a new root node *f -> "foo"* must be created and this newly cloned node shares the same left-child node with the original root node. This means that the whole original left subtree is reused in the new tree as this part remains unchanged during the whole insertion process. On the contrary, the right part of the subtree needs to be reconstructed and the function traverse to the right child node of the original root node. It then compares the key *j* with the key of the right child node *j*. Fortunately, they are equal this time; thus, the program has found the right place to add the new binding *j -> 100*. As a consequence, the new

node *j -> 100* is created sharing the children with the original node *j -> 10*. Finally, the new tree which contains the binding *j -> 100* is created and the function returns the root node of the newly created tree. From the observation, if the updated node is at depth ***d*** from the root, ***d*** new nodes along the traverse path are created in the new tree, thus the algorithm can achieve an average time complexity of **O(log n)** if the tree of n nodes is balanced. In addition, the new tree can potentially share a part of its subtree with the original tree as shown in the same figure. This mechanism allows the algorithm to efficiently reduce the memory that the new tree occupies. [11, p.108]

The semantic analysis phase of this project leverages the functional map implementation to handle the scoping rules of variables, functions and types declaration.

4.3 Type system in Tiger

There are 4 basic types in Tiger: INT, STRING, RECORD and ARRAY. Those basic types can be composed together to construct more complex types [17]. In details, type RECORD contains a list of pair name-type of its fields. Additionally, each type RECORD also contains a unique id which is generated on its declaration. This unique id is used to distinguish each RECORD type when performing type checking. For example, given the Tiger program in Figure 30:

```
let
  type record_a = {foo: int}
  type record_b = {foo: int}
  type record_c = record_a

  var a := record_a {foo = 5}
  var b := record_b {foo = 6}
  var c := record_c {foo = 7}
in
  a := c          (* This is legal since record_c is just an alias for record_a *)
  a := b          (* This is illegal since unique id of record_a != unique id of record_b *)
end
```

Figure 30. Record declaration in Tiger

Despite the fact that types *record_a* and *record_b* carries similar fields, they are not equal since the unique id of *record_a* is different from the unique id of *record_b*. In contrast, *record_c* is an alias type of type *record_a*, thus type *record_c* and *record_a* are equal. In addition, the expression with type *NIL* can be assigned to any *RECORD* type [17].

In addition, type ARRAY contains the type of its element and a unique id for each Array type declaration. The unique id of ARRAY type works exactly the same way as that of RECORD type.

Similar to variables, types in Tiger also have a scope. Therefore, the program needs to maintain a separate type bindings environment ***t_env*** in order to check for the scope of types. In fact, each type declaration adds the binding *t -> type* to the type binding environment *t_env*. However, Tiger types system has support for type forwarding as discussed in syntax analysis chapter. Therefore, the algorithm scans for a list of types declared adjacent to each other and adds the dummy type *NAME(type_name, mutable_real_type)* to the environment before evaluating the right hand side of each type declaration. For instance, given the program in Figure 31:

```
let
  type a = b
  type b = c
  type c = int
in
  ...
end
```

Figure 31. Type forwarding declarations in Tiger

Firstly, the type checker scans for a list of adjacent type declarations [a, b, c] and adds dummy bindings:

a->NAME("a",None)
b->NAME("b",None)
c->NAME("c",None)

to the type binding environment *t_env*. Next, the type cheker uses the updated *t_env* to evaluate the right-hand side of each declaration. As a result, those dummy bindings *b -> NAME("b", None)* and c -> *NAME("c", None)* are already available in the environment *t_env* before evaluating any type on the right-hand side. This approach prevents the type checker from throwing any unnecessary type scoping error. Once the type checker finishes evaluating the right-hand side of type *c*, the binding *c -> INT* is added to the environment while *a, b* still maps to dummy types. Thus, the program uses new environment *t_env'* and loops through all type declarations in the list again to resolve dummy types. Finally, the type binding environment *t_env* contains the bindings:

a -> INT, b -> INT, c -> INT.

Regarding recursive type declaration, consider the type declaration:

type node = { data: int, next: node }

After the type evaluation, the type binding environment contains the mapping:
node -> RECORD([(“data”, INT), (“next”, NAME(“node”, None))], unique_id)

The next step is to add the real type to the dummy type *NAME(“node”, None)* to turn it into *NAME(“node”, RECORD(...))*. The type NAME is still kept at this point as it has a special meaning in the upcoming phase - IR translation.

In addition, circular type declaration such as *a -> b -> c -> a* must be rejected by the type checker.

4.4 Functions in Tiger

Unlike type declarations which require a different binding environment ***t_env***, function declaration mappings are added to the same binding environment ***v_env*** as variable declaration. Thus, the environment ***v_venv*** stores 2 distinct types of value: *VarEntry* for variable declarations and *FunEntry* for function declarations.

On the other hand, Tiger also has support for mutually recursive function declarations as discussed in syntax analysis chapter. Therefore, the type checker handles mutual recursive function declarations in the similar approach as adjacent type declarations. In fact, the type checker adds header bindings

f -> FunEntry(parameter_types, result_type)

to the environment *v_env* for all functions *f()* declared next to each other before evaluating the body of each function *f()*. This design allows the Tiger program to call function *g()* within the body of function *f()* even though function *g()* has not been declared during the declaration of function *f()*.

4.5 Type checking implementation

Type checking in Tiger mostly happens in: variable declarations, function call, assign expression, arithmetic operation, if expression, loop expressions, record and array creations.

The logic of the type checker in this project can be found in the module **semant.ml**. This implementation consists of three mutually recursive OCaml functions shown in

Figure 32. Those three functions recursively traverse the AST tree from root node to leaves to perform type-checking and scope-checking:

```
type v_env = Env.enventry Map
type t_env = Types.ty Map

trans_dec: v_env * t_env * Absyn.dec list -> {v_env: v_env; t_env: t_env}
trans_var: v_env * t_env * Absyn.var -> Types.ty
trans_exp: v_env * t_env * Absyn.var -> Types.ty
```

Figure 32. Three main functions that performs semantic analysis in semant.ml

4.5.1 Tiger declarations

In the implementation of semantic analysis phase, the responsibility of function ***transDec()*** is to add variable, function, type bindings to the existing environment. This function takes the current value bindings environment ***v_env***, type binding environment ***t_env*** and a list of AST node ***Absyn.dec*** which can either be *FunctionDec*, *VarDec* or *TypeDec*. The returned value of this function is a pair of new binding environemt ***v_env'*** and ***t_env'*** containing new bindings of declarations caused by the list of the AST nodes *Absyn.dec*. The function *transDec()* handles the AST node *Absyn.dec* differently based on its concrete type *FunctionDec*, *VarDec* or *TypeDec.* In details:

- ***FunctionDec(adjacent_fundec_list)*** node contains a list of adjacent function declarations that should be treated as mutually recursive functions. As discussed in the previous section, a set of mapping *f -> FunEntry(param_types, result_type)* is added to *v_env* before the program evaluates the body of each function. If the result_type is not found, type NIL is added to the mapping instead. In addition, before the body of function *f(a)* is evaluated, parameter variable mappings *a -> VarEntry(...)* are added to *v_env* so that the use of variable a within the body does not throw scoping error.
- ***TypeDec(adjacent_typedec_list)*** is handled by adding a list of dummy types *NAME* to the type bindings environment *t_env* before evaluating the right-hand side of each type declaration. Finally, the type checker replaces those NAME types with their actual types.
- ***VarDec({name, type, ...})*** adds the binding *name -> VarEntry(type)* to the value bindings environment *v_env*.

4.5.2 Tiger variables

Function ***transVar()*** checks for the inapropriate use of variables in Tiger. This function takes a value binding environment *v_env*, a type binding environment *t_env* and an AST node *Absyn.var* (SimpleVar/ FieldVar/ SubscriptVar) as its arguments.

- For ***SimpleVar("x")*** node, the type checker looks for the declaration of variable ***x*** in the value binding environment *v_env*. If the lookup result is not found, the type checker throws an error exception since variable *x* has not been declared previously. In contrast, if the binding *x -> VarEntry(type)* is available in *v_env*, the program returns the associated type of variable *x*.
- For ***FieldVar(record, field_name, pos)*** node, the function *transVar()* recursively calls itself with *transVar(v_env, t_env, record)*. If the result returned by that function call is equal to type *RECORD(name_type_field_list)*, the type checker looks for the type associated with name *"field_name"* in *name_type_field_list* and returns this type.
- **SubscriptVar(array, index)** is handled similarly as *FieldVar*. However, the function call *transVar(v_env, t_env, rec)* is expected to return type *ARRAY(element_type,...)*. In addition, index expression must be an integer; thus, the function call *transExp(v_env, t_env, index)* must return type INT

4.5.3 Tiger expressions

Function ***transExp()*** is responsible for checking types of other expressions in Tiger. Similar to *transVar()*, this function also takes *v_env*, *t_env* as the first two arguments and returns a Tiger type. However, the third argument is an AST node Absyn.exp. Based on the type of Absyn.exp, the function *transExp()* returns the corresponding Tiger type:

- ***VarExp(var)*** is handled by calling function *transVar(v_env, t_enc, var)* and returning the result type of that function call
- ***NilExp*** is handled by returning type *NIL*
- ***IntExp*** is handled by returning type *INT*
- ***StringExp*** is handled by returning type *STRING*
- ***CallExp({ function_name; arguments; ... })*** is handled by first looking for the declaration of function *"function_name"* in *v_env*. If the lookup result is

FunEntry({param_types; result_type}), the type checker compares the type of each argument expressions against the each type of *param_types*. Then this function returns the *result_type* from the lookup result. If the lookup result is not *FunEntry* or it is not found, the type checker throws error exception.

- ***OpExp(left_exp, operator, right_exp)*** represents arithmetic operator(plus, minus, divide, multiply), comparison operator(equal, less than, etc) or boolean operator (and, or) in Tiger. In all three cases, both function call *transExp(v_env, t_env, left_exp)* and *transExp(v_env, t_env, right_exp)* are expected to return *INT*. As a result, the returned type of the whole operation must also be *INT*.

- ***RecordExp({fields; type; ...})*** represents Tiger record creation which contains a list of fields expressions and type's name of the record. Firstly, the type checker searches for the type of record in environment *t_env* using the name of its type Secondly, the type checker checks if the type in the lookup result is equal to type *RECORD([(field_name, field_type), ...])*. Then the type of each *field* expression in the AST node *RecordExp({fields;...})* is computed by calling *transExp(v_env, t_env, exp)* and subsequently compared to the corresponding *field_type* in type *RECORD([(field_name, field_type), ...])*. If the type checker does not detect any type error, this function returns type *RECORD*.

- ***ArrayExp({type; size_exp; init_exp; …})*** is first checked by looking up the type of array in *t_env* and the lookup result must be type *ARRAY(element_type)*. Next, the type of *size_exp* must be evaluated and compared against type *INT*. The next step is to compute the type of *init_exp* before comparing this type with the type of array's element. Finally, type *ARRAY(element_type)* is returned

- ***SeqExp([exp,..., last_exp])*** represents a sequence of Tiger expressions separated by the character ";". The type of each expression in the expression list is evaluated by calling *transExp(v_env, t_env, exp)*. However, only the type of the last expression ***last_exp*** in the sequence is returned as the final type for the entire sequence *SeqExp([exp,...])*.

- ***AssignExp({var; exp; ...})*** represents an assignment expression with *VarExp* on the left side of character ":=" and *exp* on the right. The type checker calls function *transVar(v_env, t_env, var)* to compute the type of the left-side *VarExp*.

Next, the computed result is compared against the returned type of *transExp(v_env, t_env, exp)*

- ***IfExp({test_exp; then_exp; else_exp})*** is type checked by computing type of *test_exp*, *then_exp* and *else_exp*. The computed type of *test_exp* must be INT as the number 0 represents *false* value while other numbers represent *true* value in Tiger. Next, the type checker checks if *else_exp* is available. If the *IfExp* contains ***else*** clause, the type checker evaluates types of *then_exp* and *else_exp*. In fact, the evaluated types of *then_exp* and *else_exp* must be equal and this type is the type of the whole *IfExp*. If the expression *IfExp* has no *else* clause, the function simply returns type *NIL* as the final type of the whole *IfExp*.

- ***WhileExp({test_exp; body_exp; ...)*** is evaluated similarly to *IfExp*. In the AST node *WhileExp*, the type of *test_exp* must be *INT*, *body_exp* can have any type since *WhileExp* does not return any value. Hence, the returned type of *WhileExp* is always *NIL*
- ***ForExp({var; low; high; body; ...})*** is handled by evaluating type of *var, low, high* expression and the computed results must be equal to type *INT*. Then the type of the *body* is evaluated before being discarded since *ForExp* does not return any value.
- ***BreakExp*** returns type *NIL* since *BreakExp* does not have any useful semantic meaning
- ***LetExp({declarations; body; …})*** is the most special expression in semantic analysis phase since the type checkers computes new environments *v_env'* and *t_env'* by calling *transDec(v_env, t_env, declaration)* for each declaration in the chain of declarations. The evaluated result of the whole list of declarations is the pair of new value environment *v_env'* and the new type environment *t_env'* that contain all mappings declaration in *letExp*. Finally, the program computes the type of the *body* by calling *transExp(v_env', t_env', body)* and returns this type.

5 Activation record

This chapter briefly covers the storage organization aspect of programming languages during the execution process. In fact, the execution process of a program involves different types of memory allocations: stack allocation, register allocation, heap allocation.

5.1 Stack frames

During the execution process of a program, each time a function is called, a chunk of memory is reversed by the computer for saving execution data such as local variables, return address of that function. This piece of data is often known as function's frame or record [11, p.116]. Once the function returns, its data frame is automatically deallocated.

In fact, a function may call itself or call other functions before exiting. As a consequence, multiple frames of the called functions can exist simultaneously at run time. The computers leverage stack data structure to store the data frames of those functions' calls. Stack data structure is selected for this task due to the Last-In-First-Out nature of function calls, in which a function call can return only after all of the called functions in its body exit. [11, p.116]

Specifically, the frame stack has two main operations: push and pop [11, p.118]. In order to illustrate the working mechanism behind the frame stack, the execution process of the program in Figure 33 is analyzed.

```
let
  function f() = g()
  function g() = print ("function g exits")
in
  f()
end
```

Figure 33. Simple function calls in Tiger program

When this program is executed, the state of the frame stack in each step is illustrated in Figure 34:

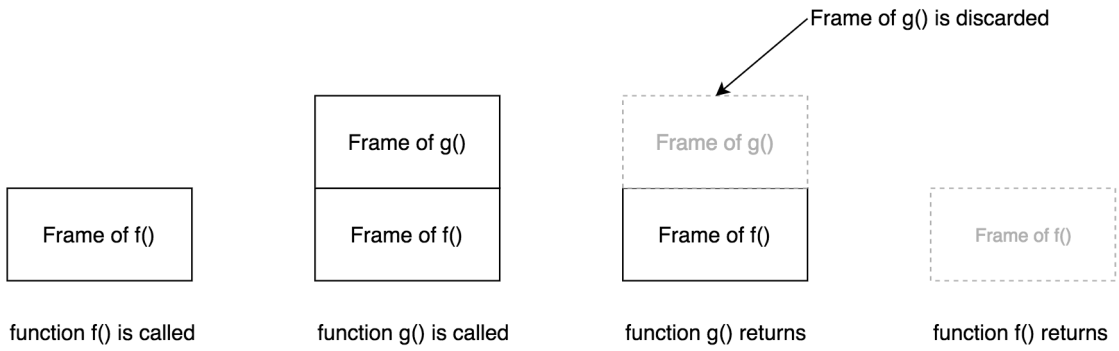

Figure 34. Visualization of frame stack during execution of program in Figure 33

In details, the frame of function *f()* is pushed on top of the stack once the function *f()* is called. In contrast, that frame is popped off the stack after function *f()* has returned. In the middle of the process, as the function *f()* calls function *g()* in its body, the frame of function *f()* is kept on hold while the frame of function *g()* is put on top of it. After function *g()* has returned, the frame of *g()* is removed from the stack, making the frame of function *f()* active again. However, there is no other execution in the body of function *f()* at this point. Hence, the function *f()* returns, its frame is eliminated and the whole program terminates.

5.2 Registers

The main task of the processor is to process data. Apparently, the processor can generate load and save instructions to access and modify data allocated in stack frame. However, loading data from memory and saving data to memory can significantly decrease the computational speed of the processor. In order to speed up the computations, the short-lived data such as short-lived variables, arguments of functions and temporary values tend to be saved in a finite set of registers, which is the short-term, smallest and fastest memory unit built-in in the CPU [11, p.120]. Nevertheless, due to the limited number of registers in the processor and the high demand for quick variables access, the compiler has to occasionally spill data from the registers to the frame stacks and load the data back when necessary.

5.3 Frame layout

In general, the stack frame of a function is used to store local variables, function's arguments and return addresses. The frames of all function calls usually share the common layout [5, p.210]. However, this frame structure may differ depending on the calling

convention specified by the producer of the machine [11, p.118]. By following the same calling convention, a program written in one language can call functions written in others. The general layout of a frame is illustrated in Figure 35:

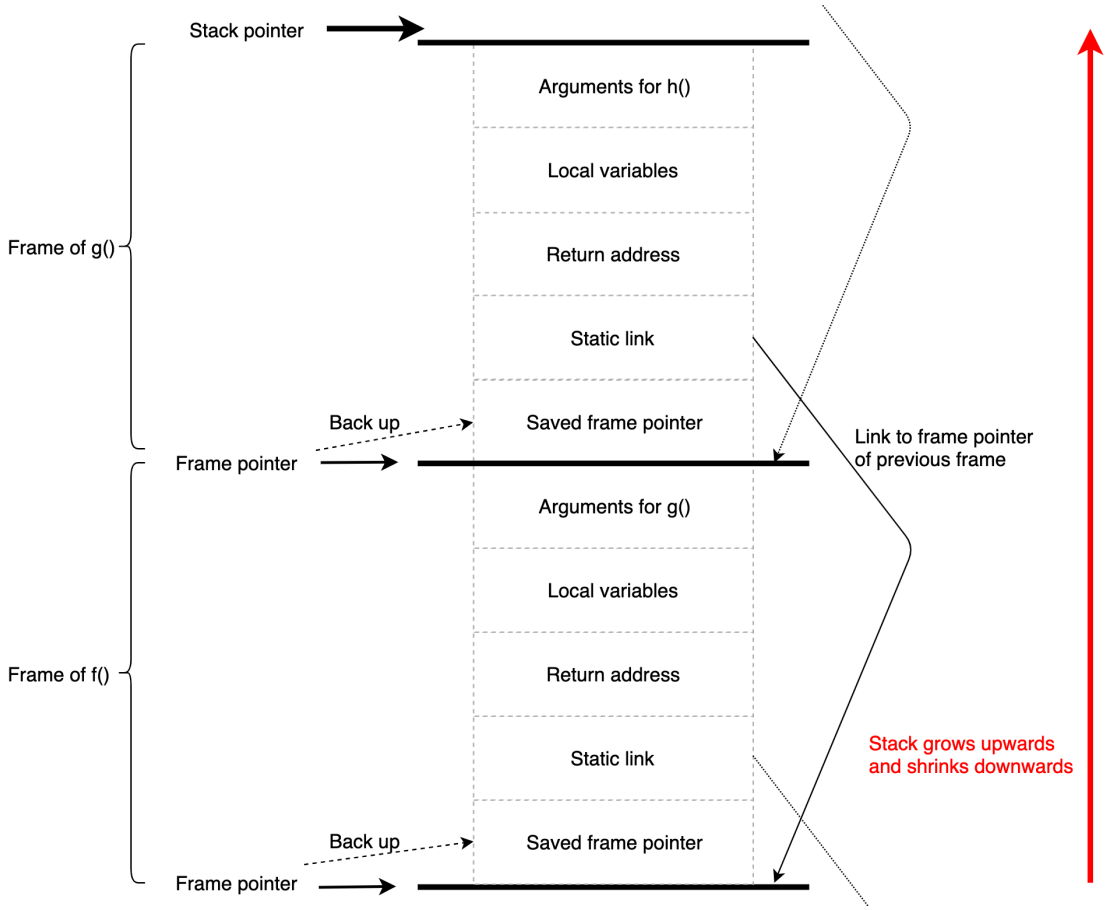

Figure 35. General frame layout in MIPS function call convention

To understand the frame layout of a function call, there are several concepts that need to be explored: stack pointer, frame pointer, return address, local variables, arguments of function calls, and static link.

5.3.1 Stack pointer

Stack pointer (SP) is a pointer that always point at the top of the frame stack. When a function *f()* is called, the size of the stack grows by the same volume as the frame's size of function *f()*. This is to ensure that the frame can sufficiently store necessary data of function *f()* [11, p.118]. Therefore, each time a function *f()* is called, the stack pointer must be recalculated following the formula:

SP = SP + <frame's size of function f()>

Similarly, the size of the stack frame declines by the same amount when function *f()* returns. Any data stored by the program at addresses beyond the stack pointer is treated as garbage data which needs to be collected [11, p.118].

5.3.2 Frame pointer

Stack pointer always points to the top of the stack, which means that the stack pointer address might increase as the size of the stack grows. As a result, accessing local variable using offset from the stack pointer can become tricky, confusing and error-prone [11, p.120]. For example, a variable *x* was allocated at the offset *-8* from the stack pointer at one point in the process. However, when the stack pointer address increases by the size of *4*, the exact same variable *x* has to be accessed at the offset *-12* from the stack pointer. Therefore, several assembly implementations leverage frame pointer to easily access frame-allocated variables.

Frame pointer (FP) is a pointer that points to the start of the active frame on the stack. Hence, the frame pointer address does not change during the time the frame of function *f()* is active. In the implementation using the frame pointer, when the function *f()* calls function *g()*, the current location of the frame pointer is saved to the stack frame of function *f()*. Next, the frame pointer points to the same location as the stack pointer before the stack pointer address is increased by the size of frame *g()*. In order to access variable *x* allocated on the frame of function *g()* when the its frame is active, the program adds the offset of variable *x* to the current frame pointer address and uses this calculated address to load *x*. When the function *g()* exits, the stack pointer points to the same location as the frame pointer. Next, the previously saved frame pointer address is loaded from the frame back to the current frame pointer. Therefore, the locations of frame pointer and stack pointer are restored as they were before the function *g()* was called. [11, p.120]

5.3.3 Return addresses

The return address of a function call is the location in the program where the execution continues right after that function call has returned [11, p.123]. For instance, when function *f()* calls function *g()*, the call to function *g()* assigns the location of the next instruction in the body of *f()* to the current return address. Therefore, the previous return address has to be saved in the frame of function *f()* before it is changed. Once the function *g()* has return, the program jumps back to the return address of the function

g(), which is a location in the body of function *f()*, and continue executing the following instruction. Finally, the previously back-up return address of function *f()* is restored.

5.3.4 Static links

Tiger supports nested function declarations. This means that if a function *g()* is defined in the body of function *f()*, function *g()* can have access to variables previously declared in the scope of function *f()*. An example of nested function declarations can be seen Figure 36:

```
let
  function f() =
    let
      var a := 5
      function g(): int = a
      function h() = g() + a
    in
      h()
    end
in
  f()
end
```

Figure 36. Nested function declarations in Tiger

In this program, function *g()* and function *h()* are declared nested in the body of function *f()* after the declaration of variable *a*. As a result, function *g()* and *h()* can access variable *a*. However, Tiger does not provide support for function closure mechanism which allows function to be treated as first-class object, to be passed around or to be returned like any other values. Therefore, a function in Tiger can only be called within its scope. [11, p.125]

In order to support nested function declarations, the frame pointer of function *f()* is passed as the first argument to the every nested function call *g()* and *h()*. The passed frame pointer is often refered to as **static link**. Once having access to the frame pointer address of function *f()*, the body of *g()* and *h()* can use this static link and the offsets of variables to access frame-allocated variables declared in *f()*. As a result, the frame pointer address of outer function *f()* needs to be saved on the stack frame of *g()* and *h()* at a known offset from the frame pointer for easy access. [11, p.131]

Interestingly, when function *g()* is called within the body of function *h()*, the program must pass the frame pointer of function *f()* instead of the frame pointer of function *h()*. This is because function *g()* is declared inside the body of function *f()* rather than the body of function *h()*. At this point, the frame of function *h()* is active; thus, the program uses the static link value saved in the frame of *h()* to obtain the frame pointer address of its parent function. Luckily, the function *f()* is the shared parent function of both function *h()* and function *g()*. As a result, the frame pointer address of function *f()* is passed as the first argument to the function call of *g()*. However, there are cases where parent function of function *h()* is not the direct parent of function *g()*. In those cases, the program repeats the process of following static links until it reaches the frame of the parent of function *g()*. [11, p.126]

5.3.5 Local variables

Short-lived local variables of a function are preferably stored in processor's registers for quick access. Typically, those variables are the ones that are not accessed by any nested functions (unescaped variables) or they do not out-live the frame of the functions that they are directly declared within. [11, p.124]

However, there exists exceptional cases where short-lived, unescaped variables are saved to the stack frame. One such example is when the size of a variable exceeds the capacity of a single register; thus, it has to be saved into stack-frame instead. In addition, if the number of local variables exceeds the number of available registers, some variables have to be spilled in the frame. [11, p.124]

In order to access the frame-resident variable *x* located at the offset *X* from the frame pointer, the compiler calculates the address of *x* by adding the offset *X* to the frame pointer address and uses the calculated result to load *x*. If a frame-allocated variable is accessed within nested functions, the program follows the static-link until it reaches the frame pointer address of function *f()* where x is declared and load variable *x* using offset *X*.

5.3.6 Function arguments

There are many conventions defining how argument values are passed into a function when that function is called. One such convention is MIPS function call standard which is demonstrated by examining the program in Figure 37:

```
let
  function f() = g (1, 2, 3, 4, 5, 6)
  function g(a: int, b: int, c: int, d: int, e: int, z: int): int = a + b + c + d + e + z
in
  f()
end
```

Figure 37. Function call in Tiger

In this example, function *g()* is invoked in the body of function *f()*. Hence, function *f()* is refered to as a **caller** while function *g()* is refered to as a **callee**. When function *f()* calls functions *g()*, the first 4 arguments *a, b, c, d* of the function *g()* are passed via registers while the rest *(e, f)* are saved at the end of the caller function *f()* 's frame[11, p.121]. Figure 38 shows the stack frame when function *g()* is called:

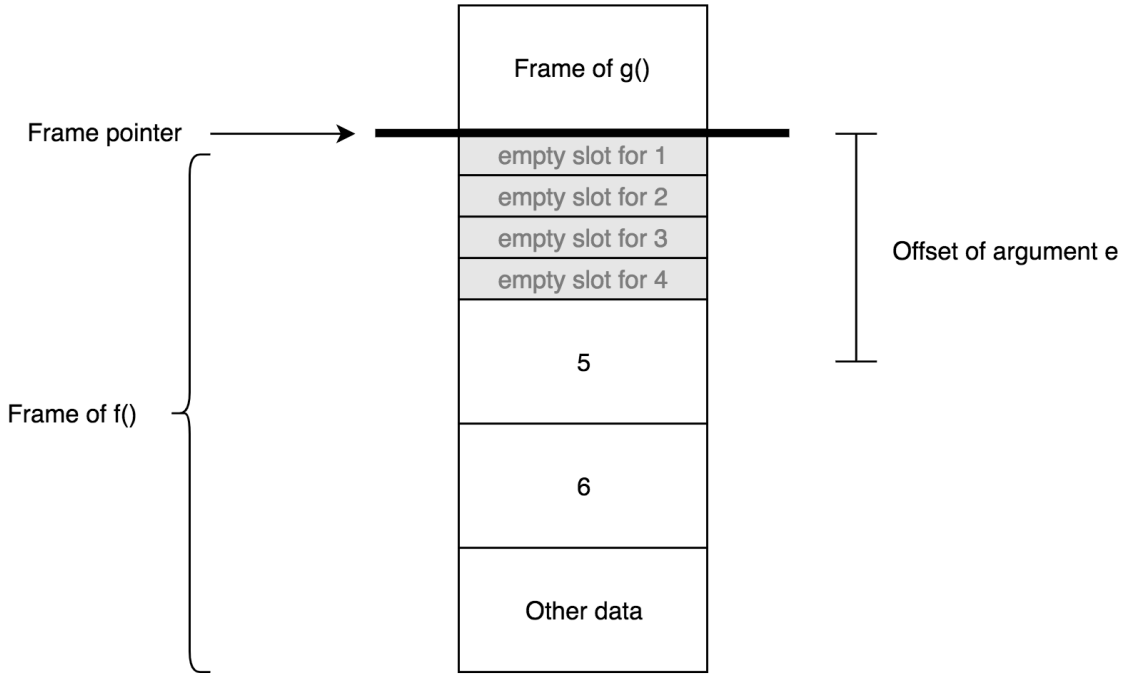

Figure 38. Snapshot of frame stack when function g() is called

Despite the fact that the values for the first 4 arguments of *g()* are passed via registers, function *f()* still reserves 4 empty slots in its frame for those arguments in case function *g()* needs to save those values [11, p.123]. Additionally, function *f()* saves the values 5, 6 for arguments *e, z* in its frame. Those frame-allocated arguments *e, z* can be accessed inside function *g()* by substracting the frame pointer address with arguments' offsets.

5.4 Heap allocations

Obviously, frame-resident variables are allocated on the entry of the function and deallocated when that function returns. As a result, the program can not save variables such as array or record in the stack frame as they might outlive the frame of the function that they are defined in [5, p.259]. For example, given the program in Figure 39:

```
let
  type arr = array of int
  function create_array(): arr = array [5] of 1
  var out_live_array := create_array()
in
  out_live_array[0]
end
```

Figure 39. Tiger array that outlives the frame

In this program, the array, created in the body of function *create_array()*, has to be returned and used after the frame of that function is removed from the stack. Therefore, this array can not be allocated in stack frame of function *create_array()*.

In practice, Tiger arrays and records are allocated in heap which is a part of computer's memory that might require manual memory management from programmers or automatic memory management by garbage collectors [5, p.259]. In C programming language, native functions such as *malloc()* and *calloc()* can be called to allocate data to the heap while function *free()* can be used to deallocate obsolete heap-allocated memories [5, p.260]. If obsolete heap memory is not efficiently released, the program may run into memory leak problem.

5.5 Implementation

As this project leverages LLVM IR and LLVM compiler (LLC) to implement the back-end of the compiler, the project does not directly implement the frame stack from scratch or perform registers allocations. However, the emulation of frame pointer has to be implemented to support nested function declarations in LLVM IR. In fact, local variables of a Tiger function are separated into 2 categories: escaped and un-escaped variables. Un-escaped variables are either allocated in registers or in the stack frames. They usually do not require any special treatment from the compiler. On the other hand, escaped variables are the ones that are declared in the parent functions and accessed within the bodies of the children functions. Those escaped variables have to be allocated at the offset X from the frame pointer of the function frame as they have to be

accessed within the nested functions using static links. As a result, the compiler has to detect those escaped variables at compile time so as to handle them differently.

The process of detecting escaped variables are implemented in the file **escape.ml**. The structure of this module is summarized in Figure 40:

```
type depth = int
type esc_env = (depth * bool ref) Map

find_escape: Absyn.exp -> unit (* entry function *)

traverse_var: esc_env * depth * Absyn.var -> unit
traverse_exp: esc_env * depth * Absyn.exp -> unit
traverseDecs: esc_env * depth * Absyn.dec list -> esc_env
```

Figure 40. Structure of file escape.ml

Similar to type checking, the entry function *find_escape()* of module **escape.ml** takes the root node of the AST tree as its input. Next, function *find_escape()* recursively traverses the AST tree from root to every leaves in order to detect the uses of escaped variables within the nested functions bodies. In practice, this process could be implemented by calling function *trans_var()*, *trans_exp()* and *traverseDec()* in the similar fashion as type-checking in module **semant.ml**.

In addition, the AST node of variable declaration ***Absyn.Vardec*** is extended to contain a new field **escape** which is a boolean value marking the escaping status of that declared variable. Similarly, boolean **escape** property is also added to the AST node ***Absyn.field*** which represents function arguments.

However, the value binding environment *v_env* and type binding environment *t_env* are unnecessary when detecting escaped variables. Instead, module escape.ml uses a new binding environment ***esc_env***, which maps the name of variable to its corresponding pair of declaration depth level and escaping status. The process of detecting escaped variables goes as follow:

Firstly, the compiler sets the initial declaration depth level to zero and traverses the AST tree from root to leaves. During the process of traversing, whenever the program encounters AST node ***FunDec*** representing function declarations, it creates the new depth level by incrementing the current level by one before adding the binding for each function's argument ***argument_name -> (new_level, false)*** to the environment *esc_env*. After that, the program uses the new depth level to traverse the body of that

function. Whenever the program processes the AST node ***VarDec("a", escape)*** representing variable declarations, it adds the mapping ***a -> (current_level, false)*** to the environment *esc_env*. Next, the program continues traversing the following nodes of the AST tree using the newly updated environment *esc_env'*. [17]

If the program finds the AST node ***SimpleVar("a")*** at any point, it looks for the declaration level of variable *"a"* in the *esc_env* and compares this result with the current depth level where variable *"a"* is used. If the used level of variable *"a"* exceeds its declaration level, this means that the variable *"a"* is used within a nested function. As a result, the ***escape*** field of AST node ***VarDec("a", escape)*** is set to *true*. It is essential that the process of detecting escaped variables must be conducted **prior** to the IR translation processes in the module **semant.ml** since IR translation need to use information regarding escaped variable. [17]

6 Intermediate Representation (IR)

Compiler is a software that translates the program written in the source language to the target language. Target languages are usually low-level languages such as assembly. However, assembly instructions are dependent on the architectures of the processors which means that different architectures require different types of assembly [5, p.147]. Hence, a compiler needs to translate the program written in a source language to **M** different types of assembly code in order to target **M** different architectures. As a result, the process of directly translating **N** source languages to **M** types of assembly requires **N x M** implementations as shown in Figure 41 [11, p.136]. Thus, the direct translation from the source language to the target assembly usually results in poor reusability.

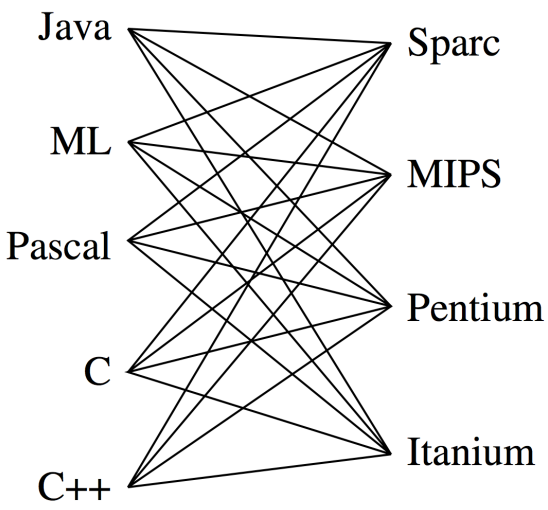

Figure 41. Direct programming languages translation graph [11, p.137]

In order to improve the portability, most compilers leverage an intermediate representation (IR) as a bridge language between the source language and the target machine language. In fact, IR is an abstracted, architecture-independent version of assembly language which usually contains more details than the source language and fewer details than the actual assembly language [11, p.136].

By using the IR as a middle layer, the compiling process can be divided into two separate parts: front-end and back-end. The task of the compiler front-end is to translate the source language to IR language. Once the IR code is available, the back-end of the compiler is responsible for compiling this IR code to **M** distinct versions of

assembly in order to target **M** architectures. Fortunately, by using the common IR language as an interface between front-end and back-end, the compilers for **N** source languages need to implement only **N** separate front-ends as they can reuse those **M** existing back-end implementations. As a result, the approach of sharing IR language between compilers enables the reduction in the total number of translations down to **M + N** when targeting **M** architectures for **N** source languages, as can be seen in Figure 42 [5, p.148].

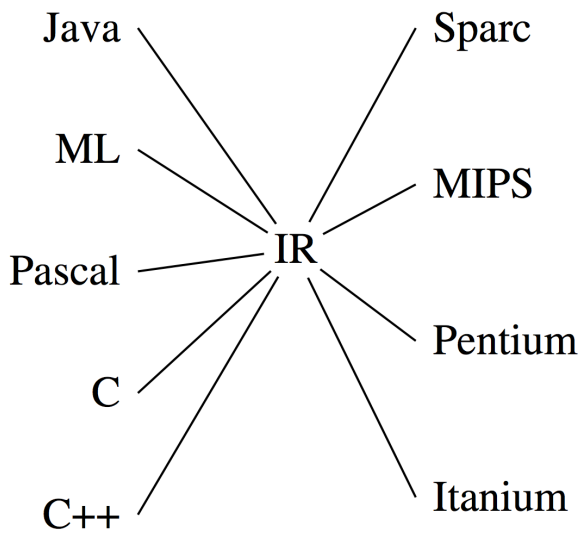

Figure 42. Indirect programming language compilation graph via IR [11, p.137]

This thesis project uses LLVM IR as an interface between Tiger programming language and assembly code.

6.1 LLVM IR

LLVM IR is a low-level, assembly-like intermediate language with a strongly type system. Unlike assembly, LLVM IR has infinite number of temporary registers. However, LLVM registers can only be assigned at most once since LLVM IR is a Static-single-assignment (SSA) based language [18]. The list of basic LLVM IR instructions can be observed in the Appendix 1.

An LLVM program is a collection of LLVM compilation units known as Modules. Each module contains a set of global variables declarations, named type declarations and function declarations. Those modules can be linked together so that function definition can be merged, forward declarations are resolved. [18]

6.1.1 Variables

Names of global variables in LLVM IR always starts with the character ***'@'***. Global variables are treated as pointers. Therefore, a ***load*** instruction must be generated to load values of global variables while saving the values of global variables is performed by the ***store*** instruction. [19, p.9]

On the other hand, names of local variables start with character ***'%'***. There are 2 types of local variable in LLVM IR: stack-based variables and register-based variables. [19, p.9]

Stack-based variables are allocated on the function's frame using the ***alloca*** instruction. The result of the ***alloca*** instruction is the pointer address to the frame location where that variable is allocated. Similar to global variable, the program must generate ***store*** instruction in order to save the value of the frame-allocated variable to its pointer address. In addition, a ***load*** instruction is needed to read the value of the frame-allocated variable at its pointer address [19, p.9]. For instance, given then C program in Figure 43:

```
int a = 5;

int main()
{
  int b = 3;
  int sum = b + a;
  return 0;
}
```

Figure 43. Simple program written in C

The equivalent LLVM IR translation of the program in Figure 43 is shown in Figure 44, in which the global variable *a* is represented as @*a* in LLVM IR while *%b* and *%sum* are the pointer addresses for the stack-allocated variables *b* and *sum* respectively. In order to obtain the values of variables *a, b* and *sum*, the *load* instructions are performed. On the other hand, *store* instructions are used to save values of variable *b, sum*.

```
@a = global i32 5, align 4

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
  %b = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 3, i32* %b, align 4
  %1 = load i32, i32* %b, align 4
  %2 = load i32, i32* @a, align 4
  %3 = add nsw i32 %1, %2
  store i32 %3, i32* %sum, align 4
  ret i32 0
}
```

Figure 44. Unoptimized LLVM translation of the C program in Figure 43

Register-based variables has the form of *%<name>* = *<expression>* in LLVM IR. For instance, *%b, %sum, %1, %2, %3* in Figure 44's program are all register-based variables. In fact, register-based variables in LLVM IR must be assigned only once because LLVM IR are in SSA form.

6.1.2 Static Single Assignment (SSA)

LLVM variables has to follow SSA form meaning that a variable is statically assigned at most once in the entire program [19, p.24]. Therefore, the conventional program such as the one on the left side of Figure 45 must be converted to the program on the right side in order to conform to the requirements of SSA form.

```
1   a = 3                                   a1 = 3
2   b = 5 + a                               b1 = 5 + a1
3   c = pow(b, 1000)        SSA transform   c1 = pow(b1, 1000)
4   a = c + b               ============>  a2 = c1 + b1
5   d = a * 100                             d1 = a2 * 100
6   ...                                     ...
60  e = pow(b, 1000)                        e1 = pow(b1, 1000)
```

Figure 45. SSA transformation conducted on a straight-line program

The transformation in Figure 45 could be performed by giving a unique name to each variable on the left-hand side of the assignment expression. For instance, variable *a* on the line 4 of the original program is renamed to *a2* in the SSA form in order to be different from the variable *a1*. After renaming the variable on the left-hand side of each

assignment expression, the algorithm updates the name of each corresponding variable on the right-hand side of the assignment to the most recent updated name [11, p.400]. For instance, variable *a* in line 5 of the original program is renamed to *a2* in the SSA form as a consequence of the variable renaming step in line 4. Finally, the transformed program on the right side of Figure 45 meets the requirements of SSA form in which any variable is never assigned twice.

In fact, IR implementation in SSA form can potentially simplify and aid various compiler optimizations [19, p.24]. One optimization that SSA enables in this example is constant propagation. Apparently, SSA form ensures that all variables are immutable. Hence, it is obvious that the variable *b1* in line 60 in Figure 45 has never been changed since its declaration in line 2. On top of that, variable *c1* and *e1* has the same right hand side expression of *pow(b1, 100)*. As a result, the optimizer can safely infer that ***e1 = c1*** without checking the code between line 3 and line 60. This inference means that the expression *pow(b1, 100)* is evaluated only once in line 3 and is reused in line 60.

However, the complexity of the SSA transformation process is significantly increased due to the existence of conditional branches in the program. For instance, given the program in Figure 46:

```
1   a = 5                                  a1 = 5
2   if (cond) {       SSA tranform         if (cond) {
3     a = 6           ============>         a2 = 6
4   }                                      }
5   print (a)                              print (a ????)
```

Figure 46. SSA transformation when the program contains conditional branches

The variable *a* in line 5 of the left program should be renamed to either *a1* or *a2* based on the condition of the if expression. If the condition is false, the variable *a* should be renamed to *a1*, otherwise it should be renamed to *a2*. Fortunately, this problem is solvable by using *phi* function which selects value for a variable based on the predecessor blocks of the current block in the Control Flow Graph (CFG). For instance, the CFG of the program in Figure 46 is illustrated in Figure 47.

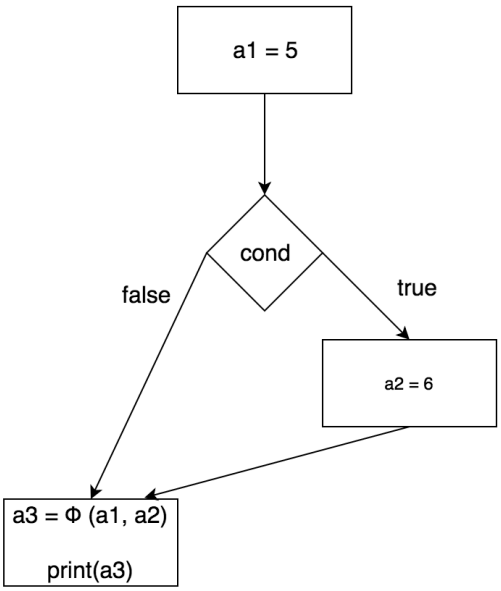

Figure 47. Control flow graph of the program in Figure 46 with phi node

In Figure 47, the program assigns the a merge node ***Φ(a1, a2)*** to new variable *a3* at the merge point of two blocks. This phi node can choose either the value of *a1* or *a2* to assign to *a3* based on the previously executed block leading to the merge block. Finally, the variable *a* in line 5 is renamed to *a3*.

6.1.3 LLVM Types

LLVM is a strongly typed IR language which has integer types, floating types, pointer types, array types, structure types and function types. [18]

- Integer types has the form of ***i<N>***, in which *N* denotes the number of bits occupied by the integer. The lower bound of *N* is 0 while its upper bound is n^23-1. For example, a 64-bit integer is shown as *i64* in LLVM. [18]

- Float types can be represented with ***float*** (32-bit float value), ***double*** (64-bit float value), ***fp128*** (128-bit float value), ***x86_fp80*** (80-bit float value) and ***ppc-fp128*** (128-bit float value). [18]

- Pointer types can be seen in in the form of ***<type>****, where <type> is the type of value saved at the address of that pointer. For example, *i8** represents a pointer address of the 8-bit integer. [18]

- Array types represent types for frame-allocated arrays. Array types can be seen in the form of ***[<size> <element_type>]*** in LLVM IR where <size> is the pre-defined size of the array, and <element_type> is the type of the array's elements. On the other hand, heap-allocated arrays in LLVM are usually represented by the pointer types. [18]

- Structure types represent a composition of different member types [18]. Structure types in LLVM are in the form of ***{<type_1>, <type_2>, ...}*** where <type_1>, <type_2>, ... are types of that structure's members in the order of declarations.

- Function types are the combination of the return types and argument types of LLVM functions. Function types can be seen in the format: ***<return_type> (<argument_type_1>, <argument_type_2>, …)***

6.1.4 Function definition and control flow

Function definitions in LLVM IR has the basic format:

define <return_type> @<function_name> (<argument_type1>, …) { <body> },

where the <body> section of functions is a sequence of basic blocks which construct the control flow of that function in LLVM [18]. Each basic block contains a list of instructions starting with a label and ending with a terminator which is either a branch instruction ***br*** or a return instruction ***ret*** [19, p.22]. In fact, the first block of the function is executed immediately once the function is called. In addition, this entry block must not be the successor of any other blocks, which means that there is no branch instruction to this entry block [18]. One example of the factorial function is shown in Figure 48:

```
define i32 @factorial(i32) {
entry:
  %eq_tmp = icmp eq i32 %0, 1
  %bool_tmp = zext i1 %eq_tmp to i32
  %cond = icmp eq i32 %bool_tmp, 1
  br i1 %cond, label %then, label %else

then:                                             ; preds = %entry
  br label %merge

else:                                             ; preds = %entry
  %minus_tmp = sub i32 %0, 1
  %1 = call i32 @factorial(i32 %minus_tmp)
  %mul_tmp = mul i32 %0, %1
  br label %merge

merge:                                            ; preds = %else, %then
  %if_result_addr.0 = phi i32 [ 1, %then ], [ %mul_tmp, %else ]
  ret i32 %if_result_addr.0
}
```

Figure 48. Factorial function in LLVM IR

In Figure 48, the function *factorial* takes a 32-bit integer as a parameter input and returns a 32-bit integer as a result. The body of this function consists of 4 blocks: *entry, then, else, merge*. As the block ***entry*** is the first block of the function, it is executed immediately on the entry of the function *factorial*. At the end of the *entry* block is a conditional branch with the format:

br i1 <cond>, label <then_label>, label <else_label>,

where if *<cond>* is evaluated to be true, the program jumps to the block whose label is *<then_label>*, otherwise it jumps to block labeled with *<else-label>*.

On the other hand, ***br*** instruction can also perform unconditional jump to a single target block in the format ***br <label>*** [18]. For instance, unconditional ***br*** instructions at the end of the *then* and *else* block both instruct the program to jump to the same *merge* block.

At the end of the *merge* block exists a return statement of the function *factorial*. This return instruction has the form ***ret <type> <value>*** if the function returns a value or ***ret void*** if the function has no return value [18].

Between the label and terminator of each block stands a sequence of zero or more instructions. Those instructions can be arithmetic operations such as *add, sub, mul, sdiv* etc or comparisons such as *icmp eq, icmp slt*. Besides that, those instructions can also

be a *load*, *store* instructions or function call such as the one in line 13. Last but not least is the appearance of the *phi* instruction that conditionally merges the value created by the *then* block or the *else* block. This *phi* instruction has the general format:

phi <type> [value1, label1], [value2, label2], …

Where <type> is the type of the result value. On the right side of <type> stands a list of [value, label] pair which represents the value created by each predecessor block of the merge block. [18]

6.2 Implementation

Obviously, LLVM IR program can becomes relatively verbose as the complexity of the program grows. Therefore, a utility package named IR builder is leveraged in order to abstractedly and programmatically generate LLVM IR code. In fact, LLVM IR builder is written in C++ but it fortunately offers API bindings in many languages including OCaml. Specifically, the IR builder offers the ability to change the code insertion location within a LLVM module. For example, the translation process can create functions, append basic blocks to the body of those functions, and freely jump between different blocks to emit code instructions at any time. Therefore, code instructions are not necessarily generated in sequential order thanks to the IR builder insertion pointer. [20]

The IR translation code can be found in file **translate.ml**. Firstly, the LLVM module for the program has to be created and given a name. This module will contain all the translated IR code including global variables declarations, named types declarations and functions declarations. Secondly, the LLVM IR builder object is instantiated to maintain the location of the insertion pointer within the global context. This builder object also contains build-in functions to generate LLVM instructions [20]. The code for this section is shown in Figure 49:

```
let context = L.global_context ()
let the_module = L.create_module context "Tiger"
let builder = L.builder context
```

Figure 49. LLVM IR builder bootstraping instructions

The next step is to extend the function *trans_var* and *trans_exp* of the module semant.ml to return LLVM IR value along with the type of each AST node at the same

time. After the extenstion, the signatures of core functions in module semant.ml are shown in Figure 50:

```
type v_env = Env.enventry Map
type t_env = Types.ty Map

trans_dec: v_env * t_env * Absyn.dec list -> {v_env: v_env; t_env: t_env}
trans_var: v_env * t_env * Absyn.var -> {type: Types.ty; exp: LLVM.llvalue}
trans_exp: v_env * t_env * Absyn.exp -> {type: Types.ty; exp: LLVM.llvalue}
```

Figure 50. Types of core functions in semant.ml after the extenstion

In fact, function *trans_var* and *trans_exp* returns the type of AST node and the LLVM value for that node simultaneously. This means that the type-checking process and IR translation process are conducted at the same time when traversing the AST tree from root node to leaves.

6.2.1 Primitive value translations

Integers in Tiger are translated to LLVM 32-bit integer constants in this project. The translation is done by calling the partial function *const_int(i32_type)(number)* provided by LLVM IR builder when traversing the AST node *IntExp(number)*.

Tiger strings can be represented by 8-bit integer pointer in LLVM IR. The compiler translates the AST node *StringExp("foobar")* to LLVM IR by calling the built-in function *build_global_stringptr("foobar").* The result of this translation is a global string constant variable in LLVM IR.

6.2.2 Records translations

Tiger records are converted to literal struct pointer in LLVM IR by calling the function ***record_exp()*** of module *translate.ml*. An example of translating the Tiger record *{ name = "foo", age = 1 }* to LLVM IR can be used to demonstrate each step of the translation process:

Firstly, a list of the record's fields' types (string, integer) must be translated to their corresponding LLVM types (i8*, i32). This type conversion logic is handled within the function *get_llvm_type()* in *translate.ml*.

Once the type conversion process has finished, the ***malloc({i8*, i32})*** instruction is generated to allocate the record to the **heap** as discussed in the previous chapter. As a result, this *malloc* instruction returns a pointer address, whose type is ***{i8*, i32}****, to that heap-allocated struct.

The next step is to save the value of each struct's field to their allocated addresses. In fact, the fields of LLVM struct do not possess any name [19, p.10]. This means that those fields must be accessed by their indices (starting from 0) within that LLVM struct. In order to compute the addresses of those fields, LLVM offers the instruction Get Element Pointer (GEP) which takes the struct's pointer address and then computes the memory address of a member element at index X [19, p.11]. After calculating the addresses of struct's fields (name, age), the program stores the value ***"foo"***, value ***1*** to the first element's address (index 0) and the second element's address (index 1) respectively in allocated struct. Finally, the function *record_exp()* returns the pointer address of the allocated LLVM struct.

One crucial attribute of GEP instruction is that this instruction does not automatically perform memory dereferencing. For instance, consider the simple C program in Figure 51:

```
1   struct Node {
2     int data;
3   };
4
5   int main() {
6     struct Node *node = NULL;
7     // allocate node to heap
8     node = (struct Node*)malloc(sizeof(struct Node));
9     node->data = 1;
10    return 0;
11  }
```

Figure 51. Simple C program

The equivalent LLVM version of this C version is shown in Figure 52:

```
%struct.Node = type { i32 }

; Function Attrs: nounwind ssp uwtable
define i32 @main() #0 {
  %1 = alloca %struct.Node*, align 8
  store %struct.Node* null, %struct.Node** %1, align 8
  %2 = call i8* @malloc(i64 4)
  %3 = bitcast i8* %2 to %struct.Node*
  store %struct.Node* %3, %struct.Node** %1, align 8
  %4 = getelementptr inbounds %struct.Node, %struct.Node* %3, i32 0, i32 0
  store i32 1, i32* %4, align 4
  ret i32 0
}
```

Figure 52. Translated LLVM IR version of the program in Figure 51

In Figure 52, the instruction ***node -> data*** in C is translated to the LLVM instruction: *getelementptr inbounds %struct.Node, %struct.Node* %3,* ***i32 0, i32 0***
In fact, when C compiler executes the instruction *node -> data*, it assumes that the index of *node* is 0. As a result, the instruction *node -> data* is assumingly equivalent to ***node[0].data***. Unfortunately, the same assumption cannot be made in LLVM IR since the LLVM IR instruction ***getelementptr*** explicitly requires 2 indices ***i32 0, i32 0*** in order to access the *data* field of the struct *Node* [21]. In details, the first index 0 is the index of the first struct *Node* from the pointer address while the second one is the index of the field *data* within that struct Node. The result of the GEP instruction is the memory address of the field *data* within the struct Node. Finally, the value 1 is saved to the memory address returned by the GEP instruction.

6.2.3 Array translations

Similar to record, Tiger array is also heap-allocated as an array may outlive the frame of the function in which it is defined. It is also translated to LLVM structure type pointer ***{ i32, <array_element>* }****, where the first field of the struct stores the pre-defined length of the array while the second field stores the pointer address to the actual heap-allocated array. An example of translating the Tiger array in Figure 53 can be used to analyze the translation process.

```
type arr = array of int
function create_array(): arr = arr [8] of 1
```

Figure 53. Array initialization in Tiger

The translated LLVM of the program in Figure 53 is shown in Figure 54:

```
define { i32, i32* }* @create_array() local_unnamed_addr gc "ocaml" {
entry:
  %malloccall = tail call i8* @malloc(i32 32)
  %array_init = bitcast i8* %malloccall to i32*
  br label %test

test:                                             ; preds = %loop, %entry
  %i.0 = phi i32 [ 0, %entry ], [ %add_tmp, %loop ]
  %lt_tmp = icmp slt i32 %i.0, 8
  %bool_tmp = zext i1 %lt_tmp to i32
  %cond = icmp eq i32 %bool_tmp, 1
  br i1 %cond, label %loop, label %end

loop:                                             ; preds = %test
  %Element = getelementptr i32, i32* %array_init, i32 %i.0
  store i32 1, i32* %Element
  %add_tmp = add i32 %i.0, 1
  br label %test

end:                                              ; preds = %test
  %malloccall4 = tail call i8* @malloc(i32 ptrtoint ({ i32, i32* }* getelementptr ({ i32, i32* }, { i32, i32* }* null, i32 1) to i32))
  %array_wrapper = bitcast i8* %malloccall4 to { i32, i32* }*
  %array_length = getelementptr { i32, i32* }, { i32, i32* }* %array_wrapper, i32 0, i32 0
  store i32 8, i32* %array_length
  %array_ptr = getelementptr { i32, i32* }, { i32, i32* }* %array_wrapper, i32 0, i32 1
  store i32* %array_init, i32** %array_ptr
  ret { i32, i32* }* %array_wrapper
```

Figure 54. Tiger array translation in LLVM IR

Firstly, the pre-defined length of the Tiger array is converted to LLVM Integer constant by calling *int_exp(length)*. Then the type of Tiger array's element is converted to equivalent LLVM type by calling function *get_llvm_type().* As a result, the struct pointer, representing the created array, has the type *{ i32, i32* }**.

The next step is to generate ***malloc*** instruction to allocate the heap memory occupied by that array. The result of *malloc* function call is the pointer address *%array_init* of the heap-allocated array. Once the array is allocated, the translator fills all the slots in that array with the initial value. This step could be implemented by creating a loop with 3 blocks (*test, loop, end*) in LLVM IR in order to iterate from index *0* to index *length - 1*. In the body of this loop exists a GEP instruction so as to compute the memory address of the array's element at the current index of the loop. Once the address of an array's element has been calculated, the instruction ***store*** is executed to save the initial value to that address.

After the program has exited the loop by jumping to the *end* block, the array wrapper struct *%array_wrapper*, containing the length and the pointer address of the array, is

saved to heap in the same fashion as discussed in the Record section. Finally the function *array_exp()* returns the pointer address to the wrapper struct.

6.2.4 Variables

The constraints, imposed by the SSA form of LLVM IR, might increase the complexity when implementing mutable variables in Tiger [20]. One common solution to tackle this problem is to allocate all variables to the stack frame with the instruction ***alloca***. This *alloca* instruction returns the pointer address of the allocated variable, which could later be used to save the value with the ***store*** instruction or to read the value with the ***load*** instruction [19, p.9]. For instance, given the simple Tiger program in Figure 55:

```
1  function main (): int =
2    let
3      var a := 5
4    in
5      a
6    end
```

Figure 55. Simple use of variable in Tiger language

The equivalent LLVM version of the Tiger program in Figure 55 is depicted in Figure 56:

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 5, i32* %1, align 4
  %2 = load i32, i32* %1, align 4
  ret i32 %2
}
```

Figure 56. Translated LLVM IR of the program in Figure 55

The LLVM IR translation process of the Tiger program in Figure 55 involves several steps. First, an ***alloca*** instruction is generated to allocate frame-memory for variable *a* (left-hand value) before returning the allocated memory address. This memory address is then used to store the value of 5 with the instruction ***store***. When the value of variable a (right-hand value) is used, a ***load*** instruction is created to load the value

from the frame-address of variable a. The diagram in Figure 57 illustrates this translation process:

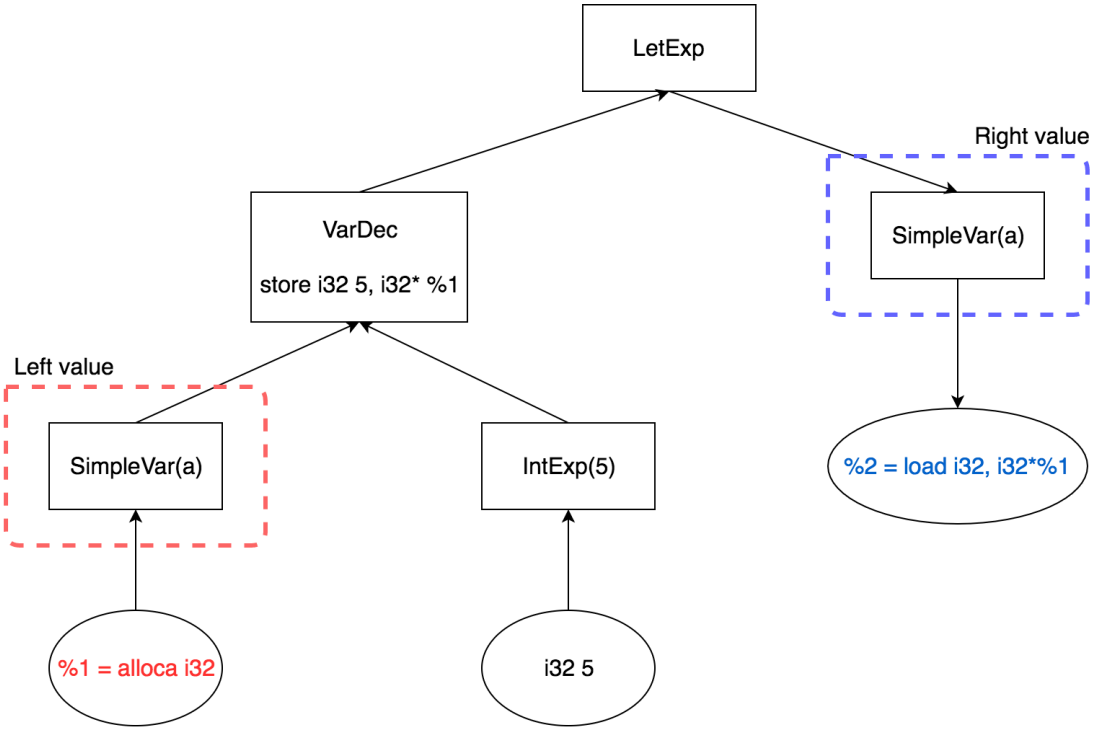

Figure 57. LLVM IR translation process of the program in Figure 55

From the observation, if a variable stands on the left side of the assignment expression in Tiger (**left-value**), it is translated to the frame-allocated **pointer address** returned by the *alloca* instruction. On the other hand, if a variable is not the target of the assignment statement (**right-value**), it is translated by loading the **value** from its known pointer address with the load instruction.

In fact, saving and loading unnecessary data to/from stack frames might adversely affect the computation speed of the compiled program. Fortunately, LLVM framework provides a built-in optimization pass ***mem2reg*** which efficiently converts the use of frame-allocated variables to register-allocated variables in SSA form. The only constrain that ***mem2reg*** imposes is that all the ***alloca*** instructions for frame-allocated variables in a function must happen in the ***entry*** block (first block) of that function. [22]

Besides that, the AST node *FieldVar* representing a specific property of the Tiger record is translated by computing the allocated address of that property with 2 known

values: the pointer address of the whole record and the index of that property. This field's address computation can be performed using ***getelementptr*** instruction (GEP) as discussed in the Record translation section. Once the memory address of the record's field is computed, the program simply returns the **pointer address** of that field if *FieldVar* node appears to be **left-value**. On the other hand, the program returns the **value** loaded from that computed address if *FieldVar* node is **right-value**. The similar translation process also applies for *SubscriptVar* node which represents the element members of the Tiger arrays.

6.2.5 Arithmetic and comparison expressions

Tiger arithmetic operations (add, subtract, multiply, divide) can easily be translated to LLVM IR using OCaml IR builder functions (build_add, build_sub, build_mul, build_sdiv). For instance, the Tiger arithmetic operations *a + 6*, *6 - a*, *a * 6*, *6 / a* are correspondingly translated to the LLVM IR instructions shown in Figure 58

```
add i32 %a, 6
sub i32 6, %a
mul i32 %a, 6
sdiv i32 6, %a
```

Figure 58. Basic arithmetic instructions in LLVM IR

Similarly, Tiger comparison operations (equal, not equal, less than, less than or equal, greater than, greater than or equal) are translated to LLVM instructions by calling IR builder function (Icmp.Eq, Icmp.Ne, Icmp.Slt, Icmp.Sle, Icmp.Sgt, Icmp.Sge). For instance, the Tiger integer comparison operations a = 6, a <> 6, a < 6, a <= 6, a > 6, a >= 6 are respectively translated to the LLVM IR instructions shown in Figure 59:

```
icmp eq i32 %a, 6
icmp ne i32 %a, 6
icmp slt i32 %a, 6
icmp sle i32 %a, 6
icmp sgt i32 %a, 6
icmp sge i32 %a, 6
```

Figure 59. Comparison instructions in LLVM IR

However, the type of the results, returned by LLVM IR comparison instructions, is 1-bit integer (*i1*) while the default type of integers in the rest of this project is *i32*. This means that the compiler has to generate an additional instruction: ***zext i1 %result to i32*** to cast type *i1* to *i32.*

6.2.6 If expression

Similar to assembly languages, control flow in LLVM IR can be constructed by navigating from one basic block to another [19, p.22]. An example program in Figure 60 could be used to illustrate how an If expression in Tiger is translated to LLVM IR:

```
function max (a: int, b: int) =
  if a >= b
  then a
  else b
```

Figure 60. If expression in Tiger program

The translated LLVM IR version of the program in Figure 60 can be seen in Figure 61:

```
1  define i32 @max(i32, i32) gc "ocaml" {
2  entry:
3    %max = alloca i32
4    br label %test
5
6  test:                                             ; preds = %entry
7    %cmp_result = icmp sge i32 %0, %1
8    %casted_cmp_result = zext i1 %cmp_result to i32
9    %cond = icmp eq i32 %casted_cmp_result, 0
10   br i1 %cond, label %else, label %then
11
12 then:                                             ; preds = %test
13   store i32 %0, i32* %max
14   br label %merge
15
16 else:                                             ; preds = %test
17   store i32 %1, i32* %max
18   br label %merge
19
20 merge:                                            ; preds = %then, %else
21   %max.1 = load i32, i32* %max
22   ret i32 %max.1
23 }
```

Figure 61. Translated version of the Tiger program in Figure 60

From the observation, the If expression in LLVM is implemented using 4 blocks: test, then, else, merge.

Block ***test*** is optional since it is used only to enhance the readability of LLVM IR translation. In fact, the optimizer usually merges block *test* with block *entry* when the IR code is optimized. The block *test* contains instructions that evaluate the condition of the *If* expression [23]. In fact, the value 0 represents *false* in Tiger while other numbers are considered to be *true*. Therefore, the result value of the comparison expression *a* >= *b* has to be evaluated before being compared against the value 0 in the test block. If the evaluated value of the condition expression is equal to 0, the program jumps to block *else*, otherwise it jumps to block *then*.

Block ***then*** contains instructions that are executed when the condition of the *If* expression is true. On the other hand, the program jumps from the block ***test*** to block ***else*** and executes instructions within block ***else*** if the condition is false. At the end of both blocks ***then*** and *else* **exist** the unconditional jumps to the same block ***merge*** which contains the Tiger instructions right after the if expression in the Tiger program. The visualization of this control flow graph is depicted in Figure 62:

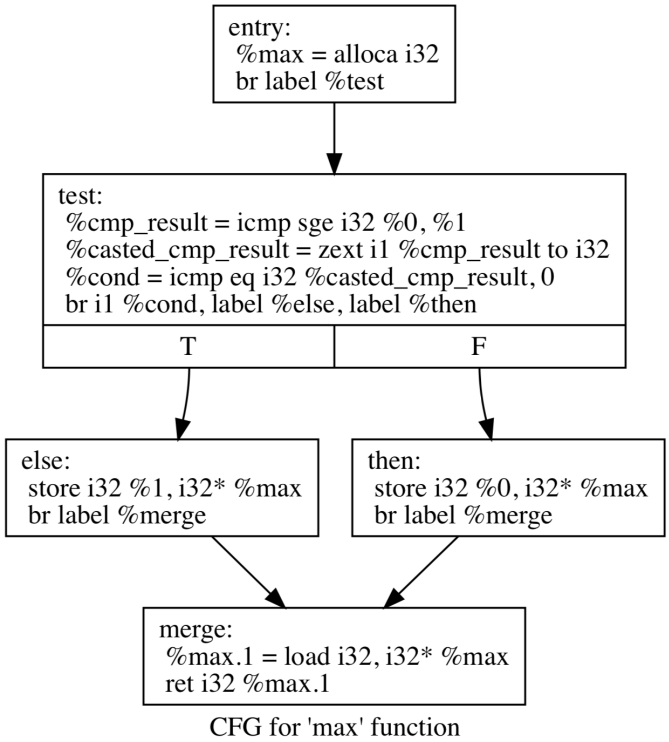

Figure 62. Control flow graph for the max function in Figure 60

In order to translate the AST node *IfExp* to LLVM IR using the IR builder, the function ***insertion_block()*** is called in order to get the current insertion block that the IR builder pointer is pointing at. Subsequently, the translator grabs the instance of the LLVM function that the current insertion block belongs to. The next step is to create and append 4 blocks ***test***, ***then***, ***else***, and ***merge*** to the instance of the grabbed function by calling the built-in function ***append_block()***. [23]

Next, the function ***build_br()*** is used to build the unconditional jump from current insertion block to the ***test*** block. After that, the insertion location is changed to the newly created ***test*** block before the program emits IR instructions for evaluating the condition expression of the *IfExp* node. The following step is to call function ***build_cond_br()*** in order to create the conditional jump instruction:
br <cond> <true_block> <false_block>. The code section for creating, appending basic blocks and evalutating condition expression is shown in Figure 63:

```
385     let current_block = L.insertion_block builder in
386     let function_block = L.block_parent current_block in
387     let test_block = L.append_block context "test" function_block in
388     let then_block = L.append_block context "then" function_block in
389     let else_block = L.append_block context "else" function_block in
390     let merge_block = L.append_block context "merge" function_block in
391
392     ignore(L.build_br test_block builder);
393     L.position_at_end test_block builder;
394     let test_val = gen_test_val () in
395     let cond_val = L.build_icmp L.Icmp.Eq test_val (int_exp 0) "cond" builder in
396     ignore(L.build_cond_br cond_val else_block then_block builder);
```

Figure 63. Translated version of the Tiger program in Figure 60

Similarly, the IR builder pointer is set to point to the block *then* before emitting IR instructions for that block. Subsequently, the partial function:
build_br(merge_block)(builder)
is called to create the unconditional branch from the current block to the ***merge*** block. The same process occurs when generating instructions in the ***else*** block. Finally, the IR builder points to the ***merge*** block where other following instructions in the program are soon be inserted into. [23]

6.2.7 Loops

There are 2 types of loop in Tiger: while loop and for loop. This project takes the approach of **replacing** the AST node *ForExp* with the AST node *WhileExp* nested inside *LetExp*. One example, in which the compiler implicitly replaces the AST node ForExp with WhileExp and LetExp, is shown in Figure 64:

```
for i := 1 to 5 do          AST rewrite          let
  print(i)                ================>        var i := 1
                                                 in
                                                     while (i <= 5) do
                                                       (print(i); i := i + 1)
                                                 end
```

Figure 64. Conversion from ForExp to WhileExp in Tiger

As a result, only the AST node ***WhileExp*** must be translated to LLVM IR to represent loop in Tiger.

The while loop in Tiger is translated to LLVM IR by appending 3 basic blocks (***test, loop, end***) to the body of the function [23]. Next, an unconditional ***br label %test*** instruction is emitted to jump from the current block to ***test*** block. The ***test*** block contains instructions that evaluate the condition to enter the loop. Then, the conditional branch instruction is generated at the end of the ***test*** block so that if the condition value is different from 0, the compiled program jumps to the ***loop*** block otherwise it jumps to ***end*** block. Next, the IR builder points to ***loop*** block and emits instructions within the body of the loop. At the end of the block ***loop*** exists an unconditional jump instruction to the ***test*** block in order to repeat the process. The LLVM IR flow graph of the program in Figure 64 is shown in Figure 65:

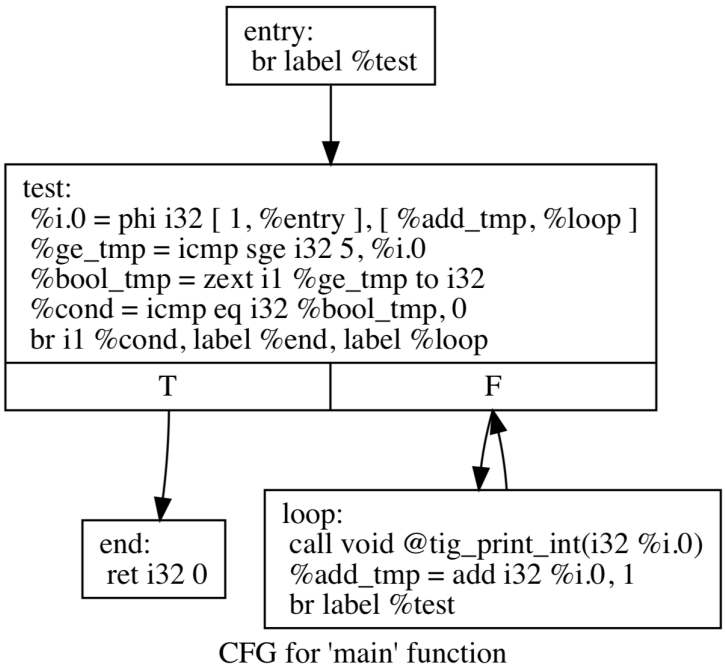

Figure 65. LLVM IR flow graph of the Tiger while loop in Figure 64

In order to implement the break statement (AST node *BreakExp*) in the body of the loop, the instance of the block ***end*** is passed down the AST tree as an argument to function *trans_exp()* when translating the body of the loop. Once the AST node BreakExp is encountered in the sub-tree that represents the body of the loop, it is translated to the unconditional jump instruction ***br label %end*** so that the compiled program can terminate the the execution of the loop early.

6.2.8 Functions

Function declaration in LLVM IR has the general format shown in Figure 66:

```
define <return_type> @<function_name> (<argument_type1>, …) {
entry:
  <instructions for entry>
; other blocks...
  ret <return_type> <return_value>
}
```

Figure 66. General format of function declarations in LLVM IR

In Figure 66, the declaration section of a function starts with the keyword ***define***. Next to the keyword *define* stands the return type of the function and the name of the function which always has the suffix '@'. On the right hand side of the function's name is the list of zero or more arguments' types enclosed between 2 parentheses. The body of the function is wrapped between 2 characters **{** and **}** in which the LLVM IR instructions are usually grouped into basic blocks. At the end of the function's body section stands the return statement ***ret*** marking the end of the function's body. One example LLVM IR function can be observed in Figure 67:

```
define i32 @f() {
entry:
  ; <body instructions...>
  ret i32 7
}
```

Figure 67. Simple function declaration in LLVM IR

In order to support mutually recursive function declarations (adjacent function declarations) in Tiger, the compiler first has to generate a list of LLVM function headers before emitting the body instructions for each adjacent function. Specifically, LLVM function header can be created with the IR builder by calling the built-in function: ***declare_function (function_name) (function_type) (module)***

The result of this function call is an IR function header in the format: ***declare*** *<return_type> @<function_name>(argument_type1,…)*

Once the LLVM IR function's headers of all mutually recursive functions are generated, the body of each function is translated to LLVM IR in sequential order. Firstly, the compiler looks for the defined function header using function name as a key. After the correct function header is found, the basic block ***entry*** is created and appended to that function. Then, the pointer to current insertion block is kept so that the IR builder can jump back to that block after finishing emiting code for the current function. This step is crucial to maintain the correct IR insertion point when translating nested functions. For instance, given the function inner() nested inside function outer() in Figure 68:

```
1  function outer() =
2    let
3      function inner() = 1
4    in
5      5 + 6
6    end
```

Figure 68. Nested function in Tiger

Before the translation process of function *inner()* occurs, the IR builder points to block *entry* of function *outer()*. When translating the function *inner()*, the IR insertion point is changed to the block *entry* of function ***inner()*** in order to generate instructions for the body of that function. After the function *inner()* is translated, the insertion point of the IR builder is still in the block *entry* of function *inner()*. Therefore, the IR builder has to point back to its previous insertion location in the body of the function ***outer()*** before generating code for the instruction *5 + 6*.

During the process of translating each function body, the IR builder points to the block *entry* of that function and emits IR instructions for the body of that function. Finally, the return instruction ret is emitted to mark the end of the function's body.

6.2.9 Nested functions

In fact, variables declared on the outer most level of the Tiger program are not actually translated to LLVM IR global variables in this project. In truth, they are mapped to the local variables of the LLVM function ***main()*** which is also the entry point of any LLVM program. This feature could be implemented by implicitly generating the LLVM IR function header and block *entry* for function *main()* before the AST tree is translated to LLVM IR. Next, the IR builder is set to point to the *entry* block of function main()before the IR instructions of the compiled Tiger program are inserted to that location. As a result, global" variables in Tiger are indeed translated to frame-allocated variables in the block *entry* of the LLVM function *main()*. Unfortunately, the concept of nested functions are unavailable in LLVM IR, meaning that local variables in one LLVM IR function are inaccessible in the bodies of other LLVM IR functions by default. For instance, the LLVM IR translation of the Tiger program in Figure 68 can be observed in Figure 69:

```
define i32 @outer() {
entry:
  ret i32 11
}

define i32 @inner() {
entry:
  ret i32 1
}
```

Figure 69. LLVM IR translation of program with nested functions

In fact, the function *inner()* and function *outer()* are completely independent in the translated LLVM IR eventhough function inner() is the nested inside function outer in the original Tiger program in Figure 68. Therefore, local variables of function *outer()* can not be accessed within the body of function *inner()* in LLVM IR by default.

One possible solution to this problem is to implicitly pass the addresses of the escaped variables of function *outer()* as arguments to function *inner()* when it is called within the body of function *outer()* [19, p.45]. For example, consider the Tiger program in Figure 70:

```
function outer() =
  let
    var a := 5
    function inner() = a + 1
  in
    5 + 6
  end
```

Figure 70. Tiger program with nested functions

In order to make variable *a* accessible in the body of function *inner()*, function *inner()* is automatically extended to carry one argument variable of type *i32**. This argument variable is reserved for passing the address of variable *a* to function *inner()*. In practice, the call expression *inner()* in the Tiger program is automatically translated to ***inner(address_of_a)*** as can bee noticed in Figure 71. As a result, variable *a* is accessible in the nested function inner().

```
define i32 @outer() {
entry:
  %a = alloca i32, align 4
  store i32 5, i32* %a, align 4
  call i32 @inner(i32* %a)
  ret i32 11
}

define i32 @inner(i32*) {
entry:
  %a = load i32, i32* %0
  %sum = add i32 %a, 1
  ret i32 %sum
}
```

Figure 71. LLVM IR translation of the program in Figure 70

However, there are certain cases where multiple variables of the function *outer()* are accessed in the body of function *inner()*. If every escaped variable reserves one argument variable of the function *inner()*, the generated IR code of that function would become increasingly confusing especially when function *inner()* also has some explicit arguments. Therefore, all escaped variables of the function *outer()* should be packed and allocated into one struct in order to emulate the frame pointer as mentioned in the Record activation chaper [19, p.45]. Subsequently, the address of this struct is always passed as the first argument to function *inner()* when it is called.

The whole process of implementing nested function in LLVM IR can be divided into 3 steps: extending function definitions, passing frame pointer address to function calls, and back tracing static link to access variables.

a. Extending function definitions:

The objective of this step are to compute the type for the frame pointer struct of each function and to implicitly extend the LLVM IR function definitons so that frame pointer address of parent function can be passed to its children functions.

In order to compute the type of the frame pointer struct, the type of each escaped variable of the function *outer()* are added to a list before the translation process for that

function's declaration occurs. This computed list of types is then used to compute the type the frame pointer struct of the function *outer()*. In details, the structure of the frame pointer struct of a function has the following form:

{ <frame pointer address of parent function>, <escaped variable>,… },

where the first element of the function's frame pointer struct stores the frame pointer address of its parent function when that address is passed as the first argument to the function. For instance, the frame pointer address of function *outer()* is passed as the first argument to function *inner()* when function *inner()* is called. This address is then saved to the first element of the frame pointer struct of function *inner()*. Other followings element of the frame pointer structs contains the values of the function's escaped variables. For example, the frame pointer address of function *outer()* in Figure 71 has type {void, i32}* while that of function *inner()* has type { {void, i32}* }* as shown in Figure 72:

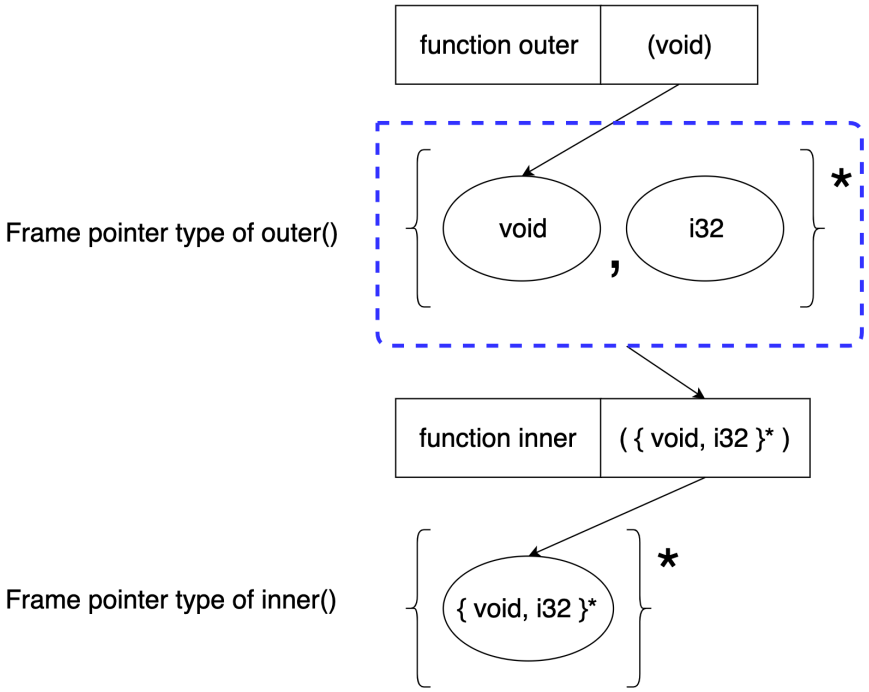

Figure 72. Frame pointer address passing for nested functions in Figure 70

Another interesting Tiger program in Figure 73 can be used to analysed how nested functions are compiled to LLVM IR.

```
let
  function grand_parent(): int =
    let
      var a := 1
      var b := "foo"
      var c := 2
      function uncle(n: int): int = c + n
      function parent(): int =
        let
          var d := 3
          var e := b
          function child(): int = uncle(5) + d
        in
          child() + a
        end
    in
      parent()
    end
in
  grand_parent()
end
```

Figure 73. Tiger program with nested function

Before the compiler starts generating code for the function *grand_parent()*, it scans the whole AST sub-tree representing the body of that function in order to collect the types of escaped variables. The result of the scanning process is the list of ***[i32, i8*, i32]*** as both variable ***a***, ***b*** and ***c*** are used in the nested functions (uncle, parent, child). Besides that, the function *main()* has an empty frame pointer struct since there is no global variable in this program. This means that the frame pointer address of the function *grand_parent()* has the type **{void, i32, i8*, i32}***. As a result, the function definitions of *uncle()* and *parent()* are implicitly extended to carry an extra argument of type *{void, i32, i8*, i32}** so that the frame pointer of function *grand_parent()* can be passed via that extra argument to function *uncle()* and *parent()*. Thus, the function *uncle()* has type i32 (**{ void, i32, i8*, i32 }***, i32) while the function parent() has type i32 (**{ void, i32, i8*, i32 }***) at this point.

Another interesting example is the translation process of function *parent()*, in which the compiler detects that the variable ***d*** is escaped. Hence, the frame pointer address of the function *parent()* has the type { **{ void, i32, i8*, i32 }***, i32 }* since the type of the struct's first element is the frame pointer's type of function *grand_parent()* while the

type of the second element is the type of variable *d*. The frame pointer passing process of the program in Figure 73 is shown in Figure 74:

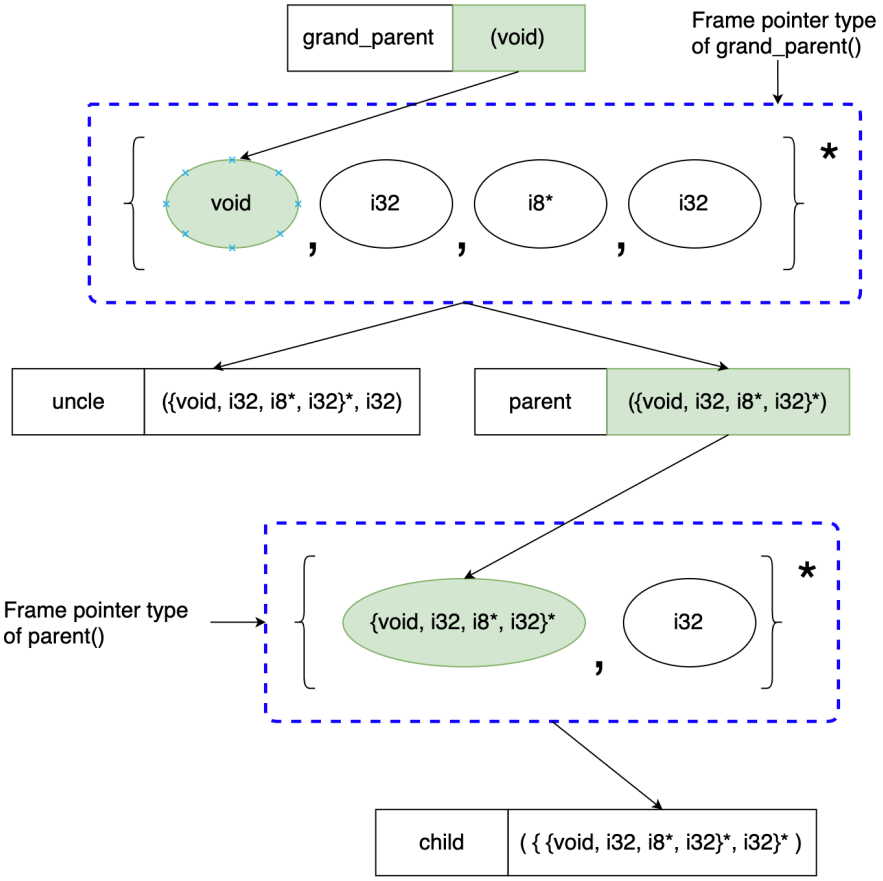

Figure 74. Frame pointer address passing for the program in Figure 73

In order to implicitly extend each the function definition to accomodate an extra frame pointer's address argument (the first argument), a stack data structure is used to store frame pointer's type of the current function. In practice, the frame pointer's type of a function is put on top of the stack when the body of that function is being translated. In constrast, that type is popped off the stack once the translation process for that function's body finishes. Figure 75 depicts the states of the stack during the translation process of each function in Figure 73, where the type in the top frame of the stack at each stage is actually the frame pointer's type of the function that are being generated at that point.

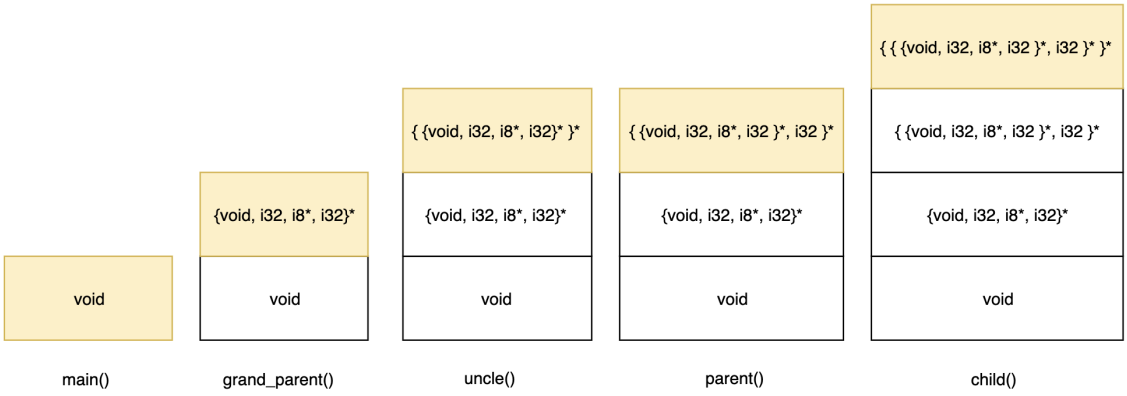

Figure 75. States of stack used for extending function definitions for the program in Figure 73

Initially, the stack contains only the type *void* which is the frame pointer's type of function *main()*. When the header of function *grand_parent()* is about to be generated, the compiler looks for the type on top of the current stack and uses this type as the first argument's type of function *grand_parent()*. As a result, the type of function *grand_parent()* is *i32 (void)*. Next, the frame pointer's type of function *grand_parent()* is computed from the type *void* on top of the stack and the types of escaped variable *a, b, c*. Therefore, the frame pointer's type of function *grand_parent()* is *{void, i32, i8*, i32}** and this computed type is put on top of the stack so that it could be used as the type of the first argument for both function *uncle()* and *parent()*. When the header of function *uncle()* is generated, its frame pointer's type *{ {void, i32, i8*, i32}* }** is put on top of the stack. Subsequently, this type is removed from the stack when translation process for function *uncle()* has terminated. The similar translation process occurs for function *parent()* and function *child()*. As a result, the frame pointer's type of function *parent()* is { { void, i32, i8*, i32 }*, i32 }* which is also the type of the first argument of function *child()*. After that, there is no other function to translate which means that the stack is cleared at this point.

b. Passing relevant frame pointer address to function call:

The next problem to solve is how to programatically pass the correct frame pointer address as the first argument of the function call. Firstly, the static nested level of each function defintions are determined as illustrated in Figure 76 where level 0 is the body level of function *main()* and it is also the definition level of function *grand_parent()*.

Similarly, level 1 is the defintion level of function *uncle()* and function *parent()* while function *child()* is defined in level 2.

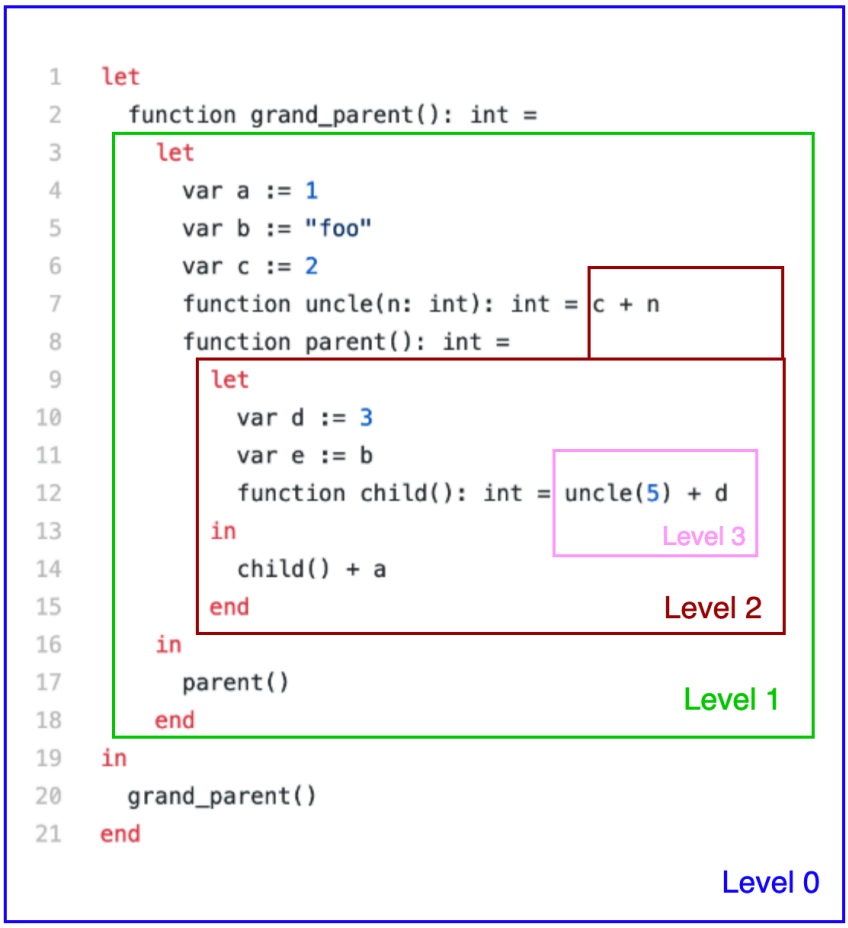

Figure 76. Static nested level of the program in Figure 73

The nested levels of the program in Figure 73 can be summarised in Table 5:

Table 5. Nested levels of program in Figure 73

Level	**Body level of function**	**Definition level of function(s)**
0	*main()*	*grand_parent()*
1	*grand_parent()*	*uncle(), parent()*

2	*parent(), uncle()*	*child()*
3	*child()*	

If a function is called in the same level as its definition level, the frame pointer address of the function, whose body corresponds to that level, is passed as the first argument to the callee function. For instance, function *grand_parent()* is both defined and called in level 0 which is the body's level of function *main()*. As a result, the frame pointer of function *main()* is passed as the first argument to function *grand_parent()*. Similarly, as level 1 is both the definition level and called level of function *parent()*, the frame pointer of function *grand_parent()* is passed as first argument to the call.

Interestingly, function *uncle()* is called in level 3 while its definition level is level 1 which is the body level of function *grand_parent()*. Therefore, the compiler needs to trace the static link from the body of function *child()* in order to obtain the frame pointer address of function *grand_parent()*. In fact, function *uncle()* is called in the body of function *child()* whose first argument variable is the frame pointer address of function *parent()*. As a result, the instructions:
%grand_parent_fp_address_pointer = getelementpointer %parent_fp, i32 0, i32 0
%grand_parent_fp_address = load %grand_parent_fp_address_pointer
generated to get the first element in the frame pointer struct of function *parent()*. This first element is actually the frame pointer's address of function *grand_parent()*. As a result, the tracing process terminates at this point and *%grand_parent_fp_address* is passed as first argument to the call of function *uncle()*. The tracing process can be observed in Figure 77:

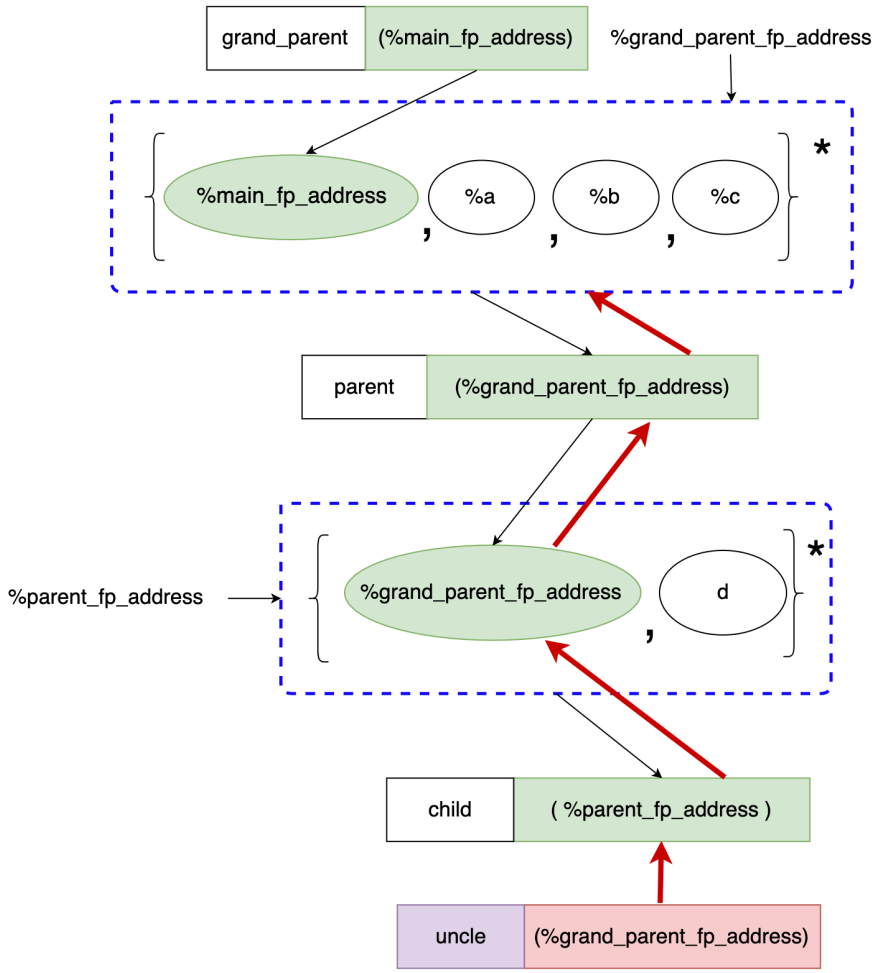

Figure 77. Static link tracing for function call of the program in Figure 73

c. Back tracing static link to access variables defined in outer scopes:

When variables, defined in the outer functions, are accessed within the nested functions, they are always allocated in the frame pointer structs of the outer functions. This means that at least one *getelementptr* instructions is needed to access those escaped variables. In practice, the static link is used in similar fashion, as described in the previous section, in order to obtain the address of frame pointer struct where those variables are stored. One solid example of this case is the variable ***a*** defined in the body of function *grand_parent()* and used in the body of function *parent()*. In order to access variable *a*, the frame pointer address of function *grand_parent()* must be obtained since variable *a* is allocated in the second element of that frame pointer struct. Once the correct frame pointer address is obtained, the compiler access the escaped

variable from the struct by generating the *getelementptr* instruction with the corresponding index.

At this point, variables defineds in outer scopes become accessible in inner scopes thanks to this three-step stactic link implementation.

6.2.10 External functions

In fact, LLVM IR function headers (function definitions with empty bodies) are treated as placeholders for external functions. During the linking phrase, those functions' headers are eventually linked to their associated implementations in other modules. As a result, this LLVM feature potentially brings the possibility in which the Tiger program can call C functions. For instance, one commonly-used external function in Tiger is function *print()*, which is actually an alias for the C function *tig_print()* defined in file **bindings.c**.

In order to implement external functions for the Tiger language, the list of functions' header mapping *alias -> FunEntry({label = "external_function_name_in_C", …})* are added to the value bindings environment *v_env* in the file **env.ml** at the beginning of the compilation process. As a result, those aliased functions are treated as global functions in Tiger since their mappings are available in the environment *v_env* from the beginning. For example, the function mapping:
"print" -> FunEntry({ label = "tig_print"; formals=[T.STRING]; result=T.NIL })
is added to environment *v_env* before the Tiger program is type-checked. Therefore, when the function *print()* is called in that Tiger program, the lookup result with the key *"print"* is available from the environment *v_env*. [17]

Similarly, the LLVM IR function headings for those external functions has to be created at the beginning of the translation process before the program is translated to LLVM IR. Finally, the generated LLVM IR file of the Tiger program is linked with the file **bindings.c** using ***clang*** compiler. As a result, external functions, writen in C such as *tig_print()*, *tig_random()*, *size()* and *exit()*, can be called within Tiger programs making Tiger language become extensible. [17]

6.3 Optimizations

LLVM framework offer its built-int LLVM optimiser **opt** which reads the code from LLVM IR input file and sequentially performs various types of optimization known as LLVM asses [24]. LLVM IR code can be optimised using built-int optimisation passes with the command:
opt -f -S <input_file>.ll -o <output_file>.ll -<pass_name>

LLVM offers various built-in optimisation passes which are trivial to include in optimization script. However, this thesis briefly mentions only a few fundamental optimisation passes: *mem2reg* pass, *constprop* pass, *die* pass.

6.3.1 Mem2reg Pass

Many LLVM-based compilers implement mutable variables by storing and loading those variables from the stack frame in order to bypass the constrains imposed by SSA form [22]. As a result, this approach usually generates a huge number of *store* and *load* instructions which notoriously slows down the computation process. Therefore, those variables need to be allocated on LLVM temporary registers in order to speed up the execution by eliminating unnecessary *store* and *load* instructions. Moreover, register-based variables must be in SSA form which enables multiple subsequent optimizations such as constant propagation.

Mem2reg pass is one of the most crucial built-in optimizer that enables other LLVM passes in the optimisation process. It is responsible for detecting frame-allocated variables and converting them to register-allocated variables in SSA form using the iterated dominance frontier algorithm [22]. For example, Figure 78 depicts 2 LLVM IR versions of the same program. The version on the left side relies on frame allocation solution while the one on the right side is its mem2reg-optimised version.

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 5, i32* %2, align 4
  %4 = load i32, i32* %2, align 4
  %5 = sub nsw i32 %4, 2
  store i32 %5, i32* %3, align 4
  %6 = load i32, i32* %3, align 4
  ret i32 %6
}
```

mem2reg ------------------>

```
define i32 @main() #0 {
  %1 = sub nsw i32 5, 2
  ret i32 %1
}
```

Figure 78. The LLVM IR program before and after running mem2reg pass

In fact, mem2reg pass can only detect the *alloca* instruction in the first block (entry block) of the function. Therefore, alloca instructions of the function must be available only in the first block of the function so as to leverage the power of mem2reg pass. [22]

6.3.2 Constant propagation Pass

Constprop is an optimisation pass that performs simple constant propagation. In this optimisation, arithmetic instructions, which involves constants, are replaced by their evaluated constant results at compile time [24]. For instance, when the *constprop* optimizer detects the arithmetic instruction *%a* = ***mul nsw i32 5000, 200***, it replaces that instruction with the instruction *%a* = ***100000*** at compile-time. As a result, the compiled program can directly use the evaluated value 100000 at run-time without having to compute the original multiply instruction. Therefore, this constant propagation optimisation can boost the run-time performance of the compiled program.

6.3.3 Dead instruction elimination Pass

This dead instruction elimication (die) pass is responsible for removing instructions that are apparently unreachable in the LLVM IR program. For example, consider the Tiger program and its optimised translation in Figure 79:

```
if 2 < 0                                          define i32 @main() #0 {
then 10 * 5 + 18 / (1 + 2 + 1 + 2)     =========>     ret i32 5
else 5                                            }
```

Figure 79. Tiger program and its dead-code eliminated LLVM translation

In fact, this optimization pass detects that the condition of the *If* expression in Figure 79 is always false. This means that the instructions inside the ***then*** clause is never executed at run time. Therefore, the optimiser simply discards the unreachable instructions in the compiled program. As a matter of fact, eliminating dead code does not necessarily increase the execution speed of the compiled program. Nevertheless, it can reduce the size of the compiled program. [24]

6.4 Assembly emission and linking

Once the LLVM IR translation process of the Tiger program has ended, the LLVM IR file **<file_name>.ll** is generated. Then, the LLVM optimizer **opt** performs analyses and optimisations on the generated file <file_name>.ll in order to produce an optimized LLVM IR file **<file_name>-opt.ll**. The next step is to instruct LLVM back-end compiler LLC to generate assembly code from LLVM IR file <file_name>-opt.ll.

Apparently, the power of LLVM framework lies in its industrial-strength back-end compilers which takes valid LLVM IR programs as input and generates well-optimised assembly code. As a result, the process of constructing compiler can be cut in half as the LLVM back-end compiler **LLC** does all the back-end heavy lifting tasks such as assembly instruction selection, controll flow analysis, data flow analysis and register allocation out of the box. The LLVM compiler **LLC** can be used to compile LLVM IR program to assembly with the command:

llc <file_name>.ll

The result of executing that command is an assembly file **<file_name>.s** based on the architecture of the computer that runs the command. Finally, the assembly output file is linked with the file **bindings.c** which contains the C functions that can be called in the Tiger program. This linking step could easily be achieved by running the command:

clang <file_name>.s bindings.c -o <program_name>

The output of which is the native compiled program **<program_name>** which can be executed directly on computers. Hence, the compilation process of the Tiger program successfully terminates at this point.

7 Evaluations

This short chapter reflects the strength as well as the drawbacks of this project's implementation.

7.1 Strength

Obviously, the implementation of this compiler relies heavily on automated toolings such as **Lex** for scanning, **Yacc** for parsing and LLVM compiler **LLC** for compiling IR

code to assembly. As a result, the development process was eased significantly since the most complicated tasks are handled out of the box by those well-design, industrial-strength tools. In fact, this approach allows programmers to concentrate solely on the operations involving Abstract syntax tree such as semantic analysis and IR translation. Therefore, the narrow scope and the level of difficulty of this project are well-suited for the scope of the bachelor thesis.

Specifically, LLVM IR is a well-design, well-supported and popular intermediate representation language. In fact, since LLVM IR is a strongly typed low-level language that abstracts assembly, it has both the strong power of low-level language as well as the expressiveness of high-level language at the same time. Hence, the process of debugging, analysing the translated program was more pleasant than conducting the same tasks over raw assembly code. In addition, the **Clang** compiler can emit LLVM IR code for C programs which depicts the internal implementations of C programming language's features. Therefore, those internal implementations could be used as a references when implementing similar features for the compiled language. For instance, the implementations of array, record in Tiger could be similar to the counterpart in C. As a result, the LLVM IR program that implements array, struct in C could be used as references when implementing array, record in Tiger.

In addition, LLVM built-in optimisations and static back-end compiler (LLC) are perfor mant, reliable tools which are responsible for the most complicated tasks in constructing compiler. By relying on these tools, a decent compiler that targets broad range of computer architectures can be rapidly constructed.

7.2 Drawbacks

As every coin has two sides, the implementation, which is dependent on readily-made tools, has its drawbacks. Firstly, some of the most challenging yet rewarding parts of compiler programming such as register allocation, assembly emission and optimisation are automatically taken care of by LLVM. While this approach might significantly shorten the development time of the working compiler, it also takes away the back-end customizablity from the programmers at the same time.

Additionally, the design of the semantic analysis and IR translation phrase in this project was originally meant for translating Tiger to a dynamically typed IR language

which closely resembles assembly. Therefore, the process of emitting strongly typed LLVM IR code and maintaining the insertion point of the IR builder require several tricky modifications in the existing implementation of semantic analysis. These unexpected changes added the complexity to the code making it less functional and less clear.

However, the advantages, which LLVM brings to this project, clearly outweight its shortcommings. Thus, using LLVM framework as the back-end compiler infrastructure is a decent option for rapid compiler development.

8 Conclusion

The first goal of this paper, which is to give an overview over the compiler development process, was achieved by introducing, analysing and implementing multiple computational concepts in each stage of the compiling process. Most importantly, the second goal of the paper, which is to explore the possibility of using LLVM framework as a back-end compiler infrastructure, was also attained when the operational compiler was successfully created in the thesis project. As a result, this paper not only covers the theoretical aspects of creating the front-end compiler but also gives solid examples on how those theoretical concepts are used in the real implementation.

By and large, LLVM framework has both strengths and weaknesses. Nevertheless, this framework inevitably plays an important role in reducing the scope of compiler development from the full process, involving both front-end and back-end compiling, down to barely front-end compiling. As a result, the efficient use of LLVM framework in constructing compilers could remarkably cut the compiler development time while the output compiler can still be reasonably performant.

References

1 Demaille Akim, Levillain Roland. Tiger Compiler Reference Manual [Internet] 2018 [cited 2019 January 16]. Available from: https://www.lrde.epita.fr/~tiger//tiger.pdf.

2 Lattner Chris. LLVM [Internet] [cited 2019 January 16]. Available from: https://www.aosabook.org/en/llvm.html.

3 Dave Nell, Weems Chip. Turbo Pascal [Internet] 4th edition. London; Jones and Barlett Publishers: 1998 [cited 2019 January 16]. Available from: https://books.google.fi/books?id=XFaB8rDpUjYC.

4 Hernyák Zoltán. High-Level Programming Languages II [Internet] 2014 [cited 2019 January 17]. Available from: http://aries.ektf.hu/~hz/pdf-tamop/pdf-xx/MProg-II-OOP-Lecture.pdf.

5 Mogensen Torben. Basics of Compiler Design [Internet]. Copenhagen; University of Copenhagen: 2010 [cited 2019 September 17]. Available from: http://hjemmesider.diku.dk/~torbenm/Basics/basics_lulu2.pdf.

6 Wirth Niklaus. Compiler Construction [Internet]. Zurich; Addison-Wesley: 2005 [cited 2019 January 18]. Available from: http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf.

7 Hirzel Martin, Rose Kristoffer. Tiger Language Specification [Internet] 2013 [cited 2019 January 30]. Available from: https://cs.nyu.edu/courses/fall13/CSCI-GA.2130-001/tiger-spec.pdf.

8 Madhavapeddy Anil, Hickey Jason, Minsky Yaron (2013). Real World Ocaml [Internet] 2013 [cited 2019 January 30]. Available from: https://v1.realworldocaml.org/v1/en/html/prologue.html.

9 Clang vs Other Open Source Compilers [Internet] [cited 2019 January 30]. Available from: https://clang.llvm.org/comparison.html.

10 Lattner Chris. Introduction to the LLVM Compiler System [Internet] [cited 2019 February 4]. Available from: http://llvm.org/pubs/2008-10-04-ACAT-LLVM-Intro.pdf.

11 Appel Andrew, Palsberg Jens. Modern Compiler Implementation in Java [Internet] 2nd edition. Cambridge University Press: 1998 [cited 2019 February 5]. Available from: https://drive.google.com/file/d/1dlMEKdzNeazPuNwU1dx7XJkgE5eOcVi9/view?usp=sharing.

12 LLVM documentation. LLVM static compiler [Internet] [cited 2019 February 5]. Available from: https://llvm.org/docs/CommandGuide/llc.html.

13 Oh SoHyoung. Ocamllex Tutorial [Internet] 2004 [cited 2019 February 5]. Available from: https://courses.softlab.ntua.gr/compilers/2015a/ocamllex-tutorial.pdf.

14 Johnson Stephen. Yacc: Yet Another Compiler-Compiler [Internet] [cited 2019 February 10]. Available from: http://dinosaur.compilertools.net/yacc/.

15 Corbett Charles, Stallman Richard. Bison: The YACC-compatible Parser Generator [Internet] 1995 [cited 2019 February 10]. Available from: http://dinosaur.compilertools.net/bison/bison_1.html#SEC1.

16 Pottier Francois, Régis-Gianas Yann. Menhir Reference Manual [Internet] [cited 2019 February 10]. Available from: http://gallium.inria.fr/~fpottier/menhir/manual.pdf.

17 Appel Andrew, Palsberg Jens. Modern Compiler Implementation in ML 2nd edition. Cambridge University Press: 1998 [cited 2019 February 10].

18 LLVM Language Reference Manual [Internet] [cited 2019 February 15]. Available from: https://releases.llvm.org/2.2/docs/LangRef.html.

19 Mapping High Level Constructs to LLVM IR Documentation [Internet] [cited 2019 February 15]. Available from:

https://media.readthedocs.org/pdf/mapping-high-level-constructs-to-llvm-ir/latest/mapping-high-level-constructs-to-llvm-ir.pdf.

20 LLVM documentation. Kaleidoscope: Code generation to LLVM IR [Internet] [cited 2019 February 15]. Available from: https://llvm.org/docs/tutorial/OCamlLangImpl3.html.

21 LLVM documentation. The Often Misunderstood GEP Instruction [Internet] [cited 2019 February 15]. Available from: https://llvm.org/docs/GetElementPtr.html.

22 LLVM documentation. Kaleidoscope: Extending the Language: Mutable Variables [Internet] [cited 2019 February 20]. Available from: https://llvm.org/docs/tutorial/OCamlLangImpl7.html.

23 LLVM documentation. Kaleidoscope: Extending the Language: Control Flow [Internet] [cited 2019 February 20]. Available from: https://llvm.org/docs/tutorial/OCamlLangImpl5.html.

24 LLVM documentation. LLVM's Analysis and Transform Passes [Internet] [cited 2019 February 20]. Available from: https://llvm.org/docs/Passes.html.

Appendix 1. List of basic LLVM IR instructions

a. Instructions to load and store global variables:

```
@global_var = global i32 2
define i32 @main() {
  %1 = load i32, i32* @global_var ; load global variable's value
  store i32 1, i32* @global_var ; store 1 to global variable
  ret i32 %1 ; return %1
}
```

b. Instruction to allocate and store stack-allocated local variables:

```
define i32 @main() {
  %a = alloca i32; allocate variable a on stack
  store i32 1, i32* %a; store value 1 to address %a
  ret i32 1
}
```

c. Arithmetic instructions:

```
define i32 @main() {
  %1 = alloca i32
  store i32 1, i32* %1
  %a = load i32, i32* %1
  %sum = add i32 %a, 6; add instruction
  %sub = sub i32 6, %a; subtract instruction
  %product = mul i32 %a, 6; multiply instruction
  %division = sdiv i32 6, %a; division instruction
  ret i32 1
}
```

d. Comparison instructions:

```
define i32 @main() {
  %1 = alloca i32
  store i32 1, i32* %1
  %a = load i32, i32* %1
  %equal = icmp eq i32 %a, 6
  %not_equal = icmp ne i32 %a, 6
  %less_than = icmp slt i32 %a, 6
  %less_than_or_equal = icmp sle i32 %a, 6
  %greater_than = icmp sgt i32 %a, 6
  %greater_than_or_equal = icmp sge i32 %a, 6
  ret i32 1
}
```

e. Branching instructions:

```
define i32 @max(i32 %a, i32 %b) {

entry:
  %0 = icmp sgt i32 %a, %b ; compare %a and %b
  br i1 %0, label %true_block, label %false_block ; conditional branching

true_block: ; preds = %entry
  br label %merge_block ; unconditional branching

false_block: ; preds = %entry
  br label %merge_block ; unconditional branching

merge_block: ; preds = %true_block, %false_block
  ; phi instruction selects return_val based on the previously executed block in
the control flow
  %return_val = phi i32 [%a, %true_block], [%b, %false_block]
  ret i32 %1
}
```

f. Function call instruction:

```
define i32 @main() {
  %result = call i32 @do_nothing(i32 1) ; call function do_nothing() with value 1
  ret i32 %result
}

define i32 @do_nothing (i32 %a) {
  ret i32 %a
}
```

g. Struct related instructions:

```
%Foo_struct = type { i32, i32 } ; %Foo_struct = {a: int, b: int}

declare i8* @malloc(i32) ; external function to allocate heap memory

define %Foo_struct* @create_foo() nounwind {
  ; allocate heap memory for Foo_struct
  %foo_address = call i8* @malloc(i32 8)
  %foo = bitcast i8* %foo_address to %Foo_struct*

  ; Compute the address of the first element of %Foo_struct with instruction
getelementptr then save value 3 to that address
  %a = getelementptr %Foo_struct* %foo, i32 0, i32 0
  store i32 3, i32* %a

  ; Compute the address of the second element of %Foo_struct with instruction
getelementptr then save value 4 to that address
  %b = getelementptr %Foo_struct* %foo, i32 0, i32 0
  store i32 4, i32* %b

  ; Load values of the first and second element of struct from their addresses
  %a_val = load i32, i32* %a, align 4
  %b_val = load i32, i32* %b, align 4
  %sum = add i32 %a_val, %b_val
  ret %foo
}
```

h. Array related instructions:

```
declare i8* @malloc(i32) ; external function to allocate on heap
define i32* @create_array() nounwind {
  ; allocate heap memory for array of size 3
  %array_address = call i8* @malloc(i32 12)
  %array = bitcast i8* %array_address to i32*

  ; Compute the address of the second element of the array and save value 3 to
that address
  %a = getelementptr i32* %array, i32 1
  store i32 3, i32* %a

  ; Compute the address of the third element of the array and save value 4 to that
address
  %b = getelementptr i32* %array, i32 2
  store i32 4, i32* %b

  ; Load values of the second and third element of struct from their addresses
  %a_val = load i32, i32* %a, align 4
  %b_val = load i32, i32* %b, align 4
  %sum = add i32 %a_val, %b_val
  ret %array
}
```