G53CMP Revision Notes

Craig Knott
Version 1.0
December 28, 2013

Contents

1 What is a compiler? 4
- 1.1 Compilers vs Interpretors . . . 4
- 1.2 Structure of a compiler . . . 4
- 1.3 Syntax . . . 4
- 1.4 Semantics . . . 5

2 Defining Programming Languages 5
- 2.1 Context-Free Grammars . . . 5
  - 2.1.1 Mathematical Style . . . 6
  - 2.1.2 Programming Language Specification Style . . . 6

3 Bottom-Up Parsing 6
- 3.1 Shift Reduce Parsing . . . 6
- 3.2 Shift-Reduce Parsing Theory . . . 7
  - 3.2.1 Items . . . 7
  - 3.2.2 Handles . . . 7
  - 3.2.3 Viable Prefix . . . 7
  - 3.2.4 Validity of items . . . 8
  - 3.2.5 Completeness of items . . . 8
  - 3.2.6 Why knowing valid items is useful... . . . 9

4 LL, LR, and LALR parsing 9
- 4.1 LR(0) Grammars . . . 9
- 4.2 LR(1) Grammars . . . 10
- 4.3 LR(k) Grammars . . . 11
- 4.4 LALR Grammars . . . 11

5 Parser Generators 11
- 5.1 Ambiguous Grammars . . . 11

6 Contextual Analysis 12
- 6.1 Limitations of CFGs . . . 12
- 6.2 Scope . . . 13
  - 6.2.1 Symbol Tables . . . 13

6.3 Type . . . 14

7 Versatile Pattern Design **14**

8 Types & Type Systems **15**
8.1 Statically Typed . . . 15
8.2 Dynamically Typed . . . 15
8.3 Conservatism . . . 16
8.4 Run time errors . . . 16
8.5 Language Safety . . . 16
8.6 Typing Advantages . . . 16
8.7 Disadvantages of Typing . . . 17

9 Type System Example **17**
9.1 Example Evaluations . . . 19
9.1.1 First Example . . . 19
9.1.2 Second Example . . . 19
9.2 Types . . . 20
9.3 Exceptions . . . 22

10 Code Generation **22**
10.1 Run-Time Organisation . . . 22
10.2 Specifying Code Selection . . . 23
10.2.1 Code Selection Example . . . 23
10.3 A problem approaches... . . . 24
10.4 Symbols . . . 24

11 Run Time Organisation **25**
11.1 Storage Areas . . . 25
11.1.1 Life Time . . . 25
11.2 Storage Allocation . . . 26
11.3 Stack Frames . . . 26
11.3.1 Stack Frame Example . . . 27
11.4 Non-Local Variable Access . . . 28

12 Data Representation **29**
12.1 Direct and Indirect Representations . . . 30
12.2 Representing Primitive Types . . . 30
12.3 Representing Records (Object fields) . . . 30
12.4 Alignment . . . 30
12.5 Representing Arrays . . . 31
12.5.1 Static arrays . . . 31
12.5.2 Dynamic Arrays . . . 31
12.6 Representing Disjoint Unions . . . 31
12.7 Representing Recursive Types . . . 31
12.8 Uniform Representation . . . 32

13 Register Allocation **32**
13.1 Naive Code Generation . . . 33
13.2 Registers across calls . . . 34
13.3 Liveness . . . 34
13.4 Graph Colouring . . . 35

13.4.1 Interference Graph Example . . . 35
13.5 Register Spilling . . . 35

14 Optimisation **36**
14.1 Intermediate Level Optimizations . . . 38
14.2 High Level Optimizations . . . 39
14.2.1 Constant Folding . . . 39
14.2.2 Common Sub-expression Elimination . . . 40
14.2.3 Algebraic Identities . . . 41
14.2.4 Copy Propagation . . . 42
14.2.5 Dead Code Elimination . . . 42
14.2.6 Strength Reduction . . . 43
14.2.7 Code Motion . . . 43
14.2.8 Loop Unrolling . . . 43
14.2.9 In-lining . . . 44
14.3 Interactions between Optimisations . . . 44
14.4 Time Vs Space . . . 45
14.5 Compile Time Vs Run Time . . . 45

1 What is a compiler?

A compiler is a "program translator", it takes source code written in some language, and converts it to either some other language, or some executable code, producing error messages as a side-effect. The purpose of compilers to allow for a higher level of abstract, making programming computers much simpler. They have a large spectrum of uses, for instance programming language implementations, document processing, database optimization, hardware design, modelling and simulation, and even in web browsers to speed up execution of embedded code.

1.1 Compilers vs Interpretors

An interpretor is another class of translator. The difference between the two is that a compiler will take a program and translate it into target language once for each time the compiler is used. An interpreter on the other hand will translate the used part of a source program every time the program is ran.

1.2 Structure of a compiler

A compiler is traditionally split into five distinct parts, these are:

- Scanner
  Takes care of lexical analysis, in which the code is read and divided into tokens, each of which corresponds to a symbol in the programming language (variable names, keywords, numbers, etc). This stage is where white space and comments will be discarded (white space is only useful to initially separate tokens).
- Parser
  Takes care of syntactic analysis, in which the list of tokens produced in the previous stage are arranged into a tree-structure (called a syntax tree) reflecting the structure of the program.
- Type Checker
  Takes care of contextual analysis, in which the syntax tree is analysed to determine if the program violates certain consistency requirements (for instance, if a variable is used but has no been declared, or a variable is used outside of the scope it is declared in).
- Optimizer
  In this stage, the code is optimized and improved so that it will run faster, use less space/memory, and other various attributes (this is almost always a heuristic improvement, as it is difficult to guarantee an optimal result).
- Code Generation
  Finally, the optimized code will be converted into the target language.

1.3 Syntax

Syntax is the *form* of a program, of which there are two different types: concrete syntax (or surface syntax), is what the program looks like, which are usually strings of characters or symbols; or abstract syntax, which is are trees representing the essential structure of syntactically valid programs. Concrete syntax is generally defined at two levels: the lexical syntax, representing the language symbols or tokens, white space, and comments; and the context-free syntax. The concrete syntax is specified by the lexical grammar, and the context-free grammar, but if both grammars produce a similar looking output, why do we not just combine them? The reasons are simplicity, dealing with white space and comments in a context-free grammar becomes very difficult, and efficiency, working on classified groups of tokens facilitates parsing, potentially allowing us to use a simpler parsing algorithm.

The key difference between the concrete syntax and the abstract syntax is that the concrete syntax will generally be a string generated from the lexical and context-free grammars, where as the abstract syntax will be represented as a tree. You can also map an abstract syntax tree to an algebraic datatype where by:

1. Each non-terminal is mapped to a type
2. Each label is mapped to a constructor of the corresponding type
3. The constructors get one argument for each non-terminal and "variable" terminal in the RHS of the production
4. Sequences are represented by lists
5. Options are represented by values of type *Maybe*
6. Literal terminals are ignored

1.4 Semantics

Semantics are the actual meaning of programs, of which there are also two different types: static semantics, which are the static compile-time meaning of programs or program fragments; and dynamic semantics, which are what the programs and program fragments mean, or do, when executed at run time.

2 Defining Programming Languages

In order to develop a compiler (or other language process), it is necessary to define the syntax and semantics of both the source language and the target language. These definitions can either be *formal* or *informal*, but usually they are somewhere in between. The reason that is it important to precisely define the source and target language is that: the scanner and parser cannot be designed unless the source language is known, the code generator cannot be designed unless the target language is known, and the semantics of both languages must be known to ensure that the translation preserves the meaning of the source program (compiler correctness).

In any language definition, informal or formal, a careful distinction must be made between: the *Object Language*, the one being defined, and the *Meta Language*, which is the language definition itself. It is important for fully understand the meta language.

In an informal specification, the meta language is generally a natural language, such as English. However, it is important to note that "informal" does not, in this case, mean "lack of rigour", it is still possible to be precise.

Formal languages are mathematically precise, and usually a formal meta-language is used, for example: EBNF, for specifying context- free syntax, inference rules and syntax, for specifying static and/or dynamic semantics, and denotational semantics, for specifying dynamic semantics.

2.1 Context-Free Grammars

A Context-Free Grammar (CFG) formally describes some Context-Free Language (CFL). Context-Free Language capture common programming language ideas, such as: nested structure, balanced parentheses and matching of keywords (such as *begin* and *end*). Most "reasonable" CFLs can be recognised by a simple machine: a deterministic pushdown automata. There are two different styles of CFGs used in G53CMP.

2.1.1 Mathematical Style

Generally used for small, abstract examples at a meta level, when talking about grammars. Simple naming conventions are used to distinguish between terminal symbols, and non-terminal symbols. Generally, terminal symbols are lower-case characters or digits, and non-terminal symbols are upper-case characters. The start symbol is generally "S". (This is the style used in G52MAL).

2.1.2 Programming Language Specification Style

Used for larger, and more realistic examples. Generally, typographical conventions are used to distinguish between terminals and non-terminals. For instance, non-terminals will be italicized, terminals will be bolded, and terminals with variable spellings, for instance Identifiers, will be italicized and underlined. The start symbol for this style is generally implied by the context. For instance:

AssignStmt → *Identifer* := *Expr*

Where *Identifer* is generally defined somewhere else in the program.

The CFGs we have been looking at have been in Backus-Naur Form. But we can use Extended Backus-Naur Form, to conveniently describe more complex grammars (note that, EBNF is no more powerful than BNF, they can represent the exact same set of languages). The extra EBNF constructs allow for: parentheses for grouping, '|' for alternatives within parentheses, and the Kleene Star (*) for iteration.

3 Bottom-Up Parsing

There are two basic strategies for parsing:

1. Top-down Parsing
   This method attempts to construct the parse tree from the root downward (the root being the initial node). This traces out a *leftmost derivation*, and an example of such is Recursive-Descent Parsing, from G52MAL.

$$S \to aABe \qquad A \to bcA \mid c \qquad B \to d$$

$$\begin{aligned} S &\Rightarrow aABe \\ &\Rightarrow abcABe \\ &\Rightarrow abccBe \\ &\Rightarrow abccde \end{aligned}$$

Figure 1: Example of top-down parsing

2. Bottom-up Parsing
   This method attempts to construct the parse three from the leaves, working up toward the root. This traces out a *rightmost derivation*, in reverse.

3.1 Shift Reduce Parsing

Shift-reduce parsing is a general style of bottom-up suntax analysis. This works from the leave toward the root of the parse tree, and has two basic actions: Shift, which reads the next terminal symbol; and reduce, which takes a sequence of read terminals and previously reduced nonterminals corresponding to the RHS of a production, to the LHS nonterminal of that production. Basically, we look at the terminals, and attempt to find a corresponding nonterminal for the productions of that terminal.

$$S \to aABe \qquad A \to bcA \mid c \qquad B \to d$$

$$\begin{array}{rl} abccde & reduce\ by\ A \to c \\ abcAde & reduce\ by\ A \to bcA \\ aAde & reduce\ by\ B \to d \\ aABe & redude\ by\ S \to aABe \\ S & \end{array}$$

Figure 2: Example of bottom-up, shift-reduce parsing

This is a good parsing method, but how would a compiler know which terminal to reduce first? This is where the theory of shift-reduce parsing comes in.

3.2 Shift-Reduce Parsing Theory

3.2.1 Items

An *item* for a CFG is a production with a dot anywhere in the RHS, as shown below.

For example, the grammar:
$S \to aAC \qquad A \to Ab \mid \varepsilon$
Has the following items

$$\begin{array}{ll} S \to \cdot\, aAc & A \to \cdot Ab \\ S \to a \cdot Ac & A \to A \cdot b \\ S \to aA \cdot c & A \to Ab \cdot \\ S \to aAc \cdot & A \to \cdot\, \varepsilon \end{array}$$

Figure 3: Example of items in a simple grammar

Given a CFG $G = (N, T, P, S)$, a string $\phi \in (N \cup T)^*$, is a sentential form for G if, and only if $S \underset{G}{\overset{*}{\Rightarrow}} \phi$.

3.2.2 Handles

A *right-sentential* form is a sentential form that can be derived by a right most derivation. A *handle* of a right-sentential form, ϕ, is a substring, α, of ϕ such that $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A \omega \underset{rm}{\Rightarrow} \delta\alpha\omega$ and $\delta\alpha\omega = \phi$, where $\alpha, \delta, \phi \in (N \cup T)^*$, and $\omega \in T^*$. In simple terms, it is the symbol(s) on the RHS that replace the replaced LHS non-terminal.

For an unambiguous grammar, the rightmost derivation is unique. Thus we can talk about "the handle" rather than merely "a" handle.

3.2.3 Viable Prefix

A viable prefix of a right-sentential form ϕ is any prefix γ of ϕ ending no father right than the right end of the handle ϕ. In simple terms, a viable prefix are all sequential combinations of symbols that extend no further than the end of the handle. As in, if *abcAde* had handle *bcA*, the viable prefixes of *abcAde* are: $\varepsilon, a, ab, abc, abcA$.

For example, consider the grammar:
$S \to aABe \qquad A \to bcA \mid c \qquad B \to d$

And the following rightmost derivation of this grammar:
$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} abcAde$$

Because $aABe$, $aAde$ and $abcAde$ were derived from a rightmost derivation, they are right-sentential forms; with the following handles:

RHSProduction	*Handle*	*Explanation*
$aABe$	$aABe$	*S is replaced by aABe*
$aAde$	d	*B is replaced by d*
$abcAde$	bcA	*A is replaced by bcA*

Figure 4: Identification of handles in a rightmost derivation

Consider the grammar:
$S \to aABe \qquad A \to bcA \mid c \qquad B \to d$

And the rightmost derivation:
$$S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} abcAde$$

Knowing that abcAde has handle bcA, it has the following viable prefixes:
$\varepsilon, a, ab, abc, abcA$

Figure 5: Identification of viable prefixes in a rightmost derivation

3.2.4 Validity of items

An item $A \to \alpha \cdot \beta$ is valid for a viable prefix γ, if there is a rightmost derivation $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A\omega \underset{rm}{\Rightarrow} \delta\alpha\beta\omega$ and $\delta\alpha = \gamma$. In simpler terms, they are a set of productions, for each non ε viable prefix of a RHS, that display all symbols to the left of the handle.

Consider the grammar and derivation from figure 5. The last derivation step was $aAde \underset{rm}{\Rightarrow} abcAde$, by production $A \to bcA$, meaning the handle is bcA. This means that the valid items for the non-ε prefixes are:

Viable Prefix	Valid Item
a	$A \to \cdot bcA$
ab	$A \to b \cdot cA$
abc	$A \to bc \cdot A$
$abcA$	$A \to bcA\cdot$

Figure 6: Identification of valid items

3.2.5 Completeness of items

An item is *complete*, if the dot is the rightmost symbol in the item.

3.2.6 Why knowing valid items is useful...

Knowing the valid items for viable prefix allows a right most derivation in reverse to be found. If $A \to \alpha \cdot$ is a complete valid item for a viable prefix $\gamma = \delta\alpha$ of a right-sentential form $\gamma\omega$ ($\omega \in T^*$), then it appears that $A \to \alpha$ can be used at the last step, and that the previous right-sentential form is $\delta A\omega$. If this indeed always the case for a CFG G, then for any $x \in L(G)$, since x is a right-sentential form, previous right-sentential forms can be determined until S is reached, giving a right-most derivation of x. Of course, if $A \to \alpha\cdot$ is a complete valid item for a viable prefix $\gamma = \delta\alpha$, in general, we only know it *may be possible* to use $A \to \alpha$ to derive $\gamma\omega$ from $\delta A\omega$. For example:

1. $A \to \alpha\cdot$ may be valid because of *different* rightmost derivation $S \underset{rm}{\overset{*}{\Rightarrow}} \delta A\omega' \underset{rm}{\Rightarrow} \phi\omega'$
2. There coudl be *two or more complete items* valid for γ
3. There could be a handle of $\gamma\omega$ that *includes symbols of* ω

4 LL, LR, and LALR parsing

There are three main methods of Parsing; *LL(k)*, *LR(k)* and *LALR(k)*.

1. *LL(k)*
   Input is scanned from Left to Right, uses Leftmost derivation, k symbols of look-ahead.
2. *LR(k)*
   Input is scanned from Left to Right, uses Rightmost derivation, k symbols of look-ahead.
3. *LALR(k)*
   Stands for Look-Ahead Left-Right, and is a simplified version of LR parsing.

By extension, we name the classes of grammars that each of these methods can handle by the same name (for instance, a *LL(k)* parser can handle *LL(k* grammars).

The reason we study LR and LALR parsing is that these methods handle a wide class of grammars, handle left- and right-recursive grammars, and LALR is a good compromise between expressiveness and space cost of implementation.

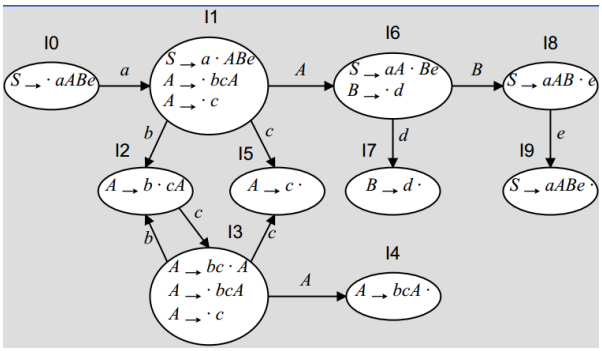
4.1 LR(0) Grammars

A CFG for which knowing a complete valid item is enough to determine the previous right-sentential form is called *LR(0)*. For *any* CFG, the set of viable prefixes is regular; thus, and efficient parser can be developed for an *LR(0)* CFG based on a DFA for recognising viable prefixes and their valid items. The states of the DFA are *sets* of items valid for a recognised viable prefix. It is worth noting that, due to convention, all states on a LR-DFA are considered accepting, and error transitions/states are not drawn. Given a DFA recognising viable prefixes, an *LR(0)* parser can be constructed as follows:

- In a state *without complete items:* Shift.
  Read next terminal symbol and push it onto an internal parse stack. Move to a new state by following the edge labelled by the read terminal.
- In a state with a *single complete item*: Reduce.
  The top of the parse stack contains the handle of the current right-sentential form (since we have recognised a viable prefix for which a single complete item is valid). The handle is just the RHS of the valid item. Reduce to the previous right-sentential form by replacing the handle on the parse stack with the LHS of the valid item. Move to the state indicated by the new viable prefix on the parse stack.

- For other states..
  If there state contains both complete and incomplete items, or if a state contains more than one complete item, then the grammar is not *LR(0)*.

One such example of a DFA (of the grammar we have been using throughout these notes) is below:



The complete sequence of derivation for the word *abccde* is the following:

State	Stack (γ)	Input (ω)	Move
i0	ε	abccde	Shift
i1	a	bccde	Shift
i2	ab	ccde	Shift
i3	abc	cde	Shift
i4	abcc	de	Reduce by $A \to c$
i5	abcA	de	Reduce by $A \to bcA$
i6	aA	de	Shift
i7	aAd	e	Reduce by $B \to d$
i8	aAB	e	Shift
i9	aABe	e	Reduce by $S \to aABe$
	S	ε	Done

Which gives us the rightmost derivation: $S \underset{rm}{\Rightarrow} aABe \underset{rm}{\Rightarrow} aAde \underset{rm}{\Rightarrow} abcAde \underset{rm}{\Rightarrow} abccde$

4.2 LR(1) Grammars

In practice, LR(0) tends to be a bit too restrictive. If we add one symbol of "lookahead" by determining the set of *terminals that possible could follow a handle* being reduced by a production $A \to \beta$, then a wider class of grammars can be handled. Such grammars are called $LR(1)$ (as mentioned previously, the "1" means one symbol of look ahead). The basic idea behind the *LR(1)* grammar is:

1. Associate a lookahead set with items:
   $A \to \alpha \cdot \beta, \{a_1, a_2, ..., a_n\}$

2. On reduction, a complete item is only valid if the next input symbol belongs to its lookahead set

3. Thus it is OK to have two or more simultaneously valid complete items, as long as their lookahead sets are disjoint.

4. This is similar to predictive recursive-descent parsing.

4.3 LR(k) Grammars

Naturally is is possible to have more than one symbol of lookahead. In general a a grammar that may be parsed with k symbols of lookahead is called $LR(k)$. However, $k > 1$ does not add to the class of languages that can be defined.

4.4 LALR Grammars

A problem with *LR(k)* parsers is that the DFAs become very large. *LALR* is a simplified construction that leads to much smaller DFAs. The basic idea is to reduce the number of states by merging sets of *LR(1)* items that are "similar". *LALR* places some additional constants on a grammar, but those constraints are not too severe in practice. Most programming languages have *LALR* grammars.

5 Parser Generators

A parser generator is a program that generates a parser for a compiler (sort of like a compiler compiler). The reason that these exist is that it is tedious and time consuming to write a parser by hand. In particular, this is true for LR(k) and LALR parsers: constructing the corresponding DFAs is extremely laborious. For even simple grammars, the DFAs can be very large (the DFA in the previous section being an example of this).

Parser construction is, in many ways, a very mechanical process, so why would we not write a program to do this for us? A parser generator takes a grammar as input, and outputs a parser for the grammar. The input grammar is augmented with "semantic actions": code fragments that get invoked when a derivation step is performed. These semantic actions typically construct an AST.

Consider the following LR shift-reduce parser, in which some of the actions whilst parsing *abccde*.

$State$	$Stack(\gamma)$	$Input(\omega)$	$Move$
...	...	...	...
$i6$	aA	de	$Shift$
$i7$	aAd	e	$Reduce\ by\ B \to d$
$i8$	aAB	e	$Shift$
$i9$	$aABe$	ε	$Reduce\ by\ S \to aABe$
	S	ε	$Done$

A reduction corresponds to a derivation step in the grammar (an LR parser performs a rightmost derivation in reverse).

At reduction, the terminals and non-terminals of the RHS of the production (the handle) are on the parser stack, associated with semantic information (the corresponding AST). You can think of RHS symbols as variables whose values are the corresponding semantic information. If the goal is to construct as AST, the parser has access to the sub-ASTs and can construct an AST for the present derivation step.

5.1 Ambiguous Grammars

Context-free grammars are foten initially ambiguous; for instance, consider the following:

$$\begin{aligned} Cmd &\to \ ... \\ &\mid \ if\ Exp\ then\ Cmd \\ &\mid \ if\ Exp\ then\ Cmd\ else\ Cmd \end{aligned}$$

According to this grammar, a program fragment:

$$if\ e_1\ then\ if\ e_2\ then\ c_1\ else\ c_2$$

Can be parsed in two ways, with very different meanings (this is know as the dangling else problem).

There are two problems that can occur through the use of ambiguous grammars can cause: shift/reduce conflicts, where some states have mixed complete and incomplete items; and reduce/reduce conflicts, where some states have more than one complete item. Generally shift/reduce will resolve by shifting, however there is no go-to option to resolve reduce/reduce conflicts, they must be manually disambiguated.

6 Contextual Analysis

Contextual Analysis (also known as checking static semantics) is the third stage of compiler design, and it is tasked with: resolving the meaning of symbols, reporting undefined symbols, and type-checking. It is used to check that a program is "statically well-formed". There are two important kinds of contextual constraints: scope rules, and type rules, which will both be covered after looking at the limitations of CFGs.

6.1 Limitations of CFGs

You could argue that context-free grammars could be used for contextual analysis, however it becomes quickly apparent that the size of these grammars grows exponentially with the number of variables being defined. for instance, see the grammar defined below.

$$\begin{array}{lcl} Prog & \rightarrow & DeclA\ ProgA \\ ProgA & \rightarrow & StmtA\ ProgA \mid \varepsilon \\ DeclA & \rightarrow & int\ a; \\ StmtA & \rightarrow & a = ExprA; \\ ExprA & \rightarrow & a \mid ExprA + ExprA \mid Expr \\ Expr & \rightarrow & LitInt \mid Expr + Expr \end{array}$$

For set of n variables, $V = \{a_i \mid 1 \leq i \leq n\}$, we get 2^n non-terminals. This makes it impractical to use a context-free grammar, especially considering there is no limit to the number of variables that can be used.

The grammar below shows a new way of approaching this problem.

$$\begin{array}{lcl} IntExpr & \rightarrow & LitInt \\ & \mid & IntVar \\ & \mid & IntExpr + IntExpr \\ BoolExpr & \rightarrow & false \\ & \mid & true \\ & \mid & BoolVar \\ & \mid & IntExpr < IntExpr \\ & \mid & not\ BoolExpr \\ & \mid & BoolExpr\ \&\&\ BoolExpr \end{array}$$

At first, this one seems much more reasonable. However, it is still not viable. The scheme hinges on partitioning the variables by name into two groups: integer variables (IntVar) and boolean variables (BoolVar). But in most languages, the type of a variable is given by the context, not its name. In fact distance variables with in the same name can be used at different types in one program. we should no expect to be able to capture such *context-sensitive* information using a *context-free* grammar.

These two examples do not *prove* that it is impossible to achieve contextual analysis using CFGs, however it *can* be proved. Context constraints result in context sensitive or even recursively enumerable languages; and of course, such languages cannot be described by CFGs.

6.2 Scope

Scope rules dictate the visibility of declarations and where they take effect. We use identification to apply the scope rules in order to relate each applied identifier occurrence to its declaration. For instance, observe the following block of Java code:

$$
\begin{array}{ll}
\textit{public} & \textit{class } C \ \{ \\
& \textit{int } x,\ n; \\
& \textit{void set } (\textit{int } n) \ \{\ x = n;\ \} \\
\}
\end{array}
$$

The variables x and n have both been defined as instance variables (fields). So what would happen when the *set*() method is called? This is where scoping take effect, to deal with ambiguities like these. In the case of Java, the "closest" declaration is the one that is used, meaning the calling of the *set*() method would use the argument n, over the field n. How this is dealt with in any programming language is through the governing scope rules.

Two key terms to remember for this section are:

1. Scope
   The portion of a program over which a declaration takes effects
2. Block
   A program phrase that delimits the scope of declarations within it.

In addition to deciding the range of declarations, the scope rules also deal with issues like whether explicit declarations are required, whether multiple declarations at the same level are allowed, and whether shadowing or hiding are allowed.

6.2.1 Symbol Tables

A symbol tables, also called an identification table, or environment, is used during identification to keep track of symbols and their attributes, such as: kind of symbol (class name, local variable, etc), scope level, type and source code position. The organisation of this table depends on the source language's block structure, of which there are three main possibilities: Monolithic, where there is one global scope; flat, where there are blocks with local scope enclosed in a global scope; and nested, where blocks can be nested to arbitrary depths (this is the most common). Generally, monolithic and flat block structure can just be considered special cases of the nested block structure.

The steps to using a symbol table are the following:

1. Initialise the table (enter the standard environment)
2. When a declaration is encountered
   (a) check if declared identifier clashes with existing symbol
   (b) Report error if it does
   (c) If not, enter declared identifier into table along with its attributes

3. When an applied identifier is encountered
   (a) Look up identifier in table, taking scope rules into account
   (b) Report error if not found
   (c) If found, annotate applied occurrence with symbol attributes from table
4. When entering a new block, arrange so that subsequently entered symbols become associated with the scope corresponding to this new block (open scope)
5. When leaving a block, either remove or make inaccessible, symbols declared in that block (close scope)

6.3 Type

Type rules dictate the internal consistency of the program; they ensure that every expression computes a value of an acceptable form. We use type-checking to apply the type rules to infer the type of each expression, allowing us to compare it with the expected type.

7 Versatile Pattern Design

The big advantage with functional programming is that everything is explicit, which makes large programs much easier to understand, but means that there are no side effects. However, this can often be functional programming's downfall, because large programs can very quickly become very cluttered, and have much repition. There are three major ways that we can attempt to capture patterns in our code and help to decrease redundancy:

1. Observing the sequence of evaluations
2. Note that, if one evaluation fails, all should fail
3. Otherwise, make the result available to following evaluations

The Maybe Monad
Consider a value of type $Maybe\ a$ as denoting a computation of a value of type a that may fail. When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails. This means that a failure is an *effect*, implicitly affecting subsequent computations.

Stateful Computations
A stateful computation consumes a state and returns a result along with a possible updated state. For instance, the expression:

$$type\ S\ a = Int \rightarrow (a,\ Int)$$

Captures this idea, by taking a value of type a, and returning a value with both this new a, and some new variable value. A value (function) of type $S\ a$ can now be viewed as denoting a stateful computation of value type a. When sequencing stateful computations, the resulting state should be passed onto the next computation. This means that state updating is an effect, implicitly affecting subsequent computations.

Monads
A moand is represented by:

1. A type constructor:
   $M :: * \to *$
   In which $M\ T$ represents omputations of a value of type T.

2. A polymorphic function:
   $return :: a \to M\ a$
   For lifting a value to a computation.

3. A polymorphic function:
   $(>>=) :: M\ a \to (a \to M\ b) \to M\ b$
   For sequencing computations.

In Haskell, the notion of a monad is captured by a $TypeClass$.

$$\begin{array}{lll} class & Monad & m\ where \\ & return & :: \quad a \to M\ a \\ & (>>=) & :: \quad m\ a \to (a \to m\ b) \to m\ b \end{array}$$

This means that the names of common functions can be overloaded, and derived derivations can be shared.

8 Types & Type Systems

Type systems are an example of "lightweight formal methods"; this is because they are highly automated, but have limited expressive power. A plausible definition for type systems is the following:

"A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute".

8.1 Statically Typed

This definition implies "static checking", since the goal is to prove absense of certain errors, and this is achieved by classifying syntactic phrases (or terms) according to the kinds of value they computre. A type system computers a static approxiamation of the run-time behaviour. For instance, if it is known that two program fragments exp_1 and exp_2 both compute integers (this is the "classification" part of the system); then it is safe to add those numbers together (this is the "absence of errors" segment). The addition of these two expresseion is known to be an integer; even though we do not know exactly which integers are involved (this is the "static approximation" segment of the system).

8.2 Dynamically Typed

"Dynamicaly typed" languages do not have a type system according to the previous definition; they should really be called "dynamically checked". For example, in a dynamically checked language, $exp_1 + exp_2$ would be evaluated as follows:

Evaluate exp_1 and exp_2
Add results together in a manner depending on their types (whether this be integers or floats..) or singla an error is this is not possible.

8.3 Conservatism

A type system is necessarily conservative. This means that even some well-behaved programs will be rejeceted. For example:

if *complextest* then S else *typeerror*
will be rejeceted as "ill-typed", even if *complextest* always evaluates to true. Since the static checker cannot know this.

8.4 Run time errors

A type system checks for certain kinds of bad program behaviour, known as "run-time errors". Exactly which, depends on the type system and the language design. For example, current main-stream type systems typically:

1. Check that arithmetic operations are only done on numbers - these will cause compile errors.
2. Do not check that the second operand of division is not zero, or that array indices are within bounds - this will not cause compile errors.

The safety (or *soundness*) of a type system must be judged with respect to its own set of run-time errors.

8.5 Language Safety

Language safety is a contentious notion. One possible definition is that:

"A safe language is one that protects its own abstractions"

For instance, a Java object should behave as an object. It would be bad if it was destroyed by the creation of some other object. Or if indexing one array overwrote values in another.

Language safety is not the same as static typing, instead safety can be achieved through static typing (and/or dynamic run-time checks). For instance, "Scheme" is a dynamically checked safe language. It's worth noting that even statically typed language usually use some dynamic checks, for instance checking array bounds, down-casting, checking for division by zero, or even pattern matching failures.

8.6 Typing Advantages

The advantages of typing and type systems are the follow:

1. Detecting errors early
   Program in richly typed languages often "just work", but why is this? Simple, common mistakes very often lead to type inconsistencies; forcing programmers to think a bit harder.
2. Enforcing disciplined programming
   Type systems are the backbone of modules and classes.
3. Documentation
   Unlike comments, type signatures will always remain current.
4. Efficiency
   First use of types in computing was to distinguish between integer and floating point numbers. Elimination of many of the dynamic checks that otherwise would have been needed to guarantee safety.

8.7 Disadvantages of Typing

However, type systems can sometimes get in the way. For instance, simple things can often become quite complicated if you have to work around the type system (for instance, heterogeneous list in Haskell[1]). Sometimes this even becomes inpossible; or at least not without loss of efficiency.

Increasingly sophisticated type systems, which keep track of more invarients can help this. But this can make the type systems harder to understand and less automatic.

9 Type System Example

Take the following type system:

$$
\begin{array}{lll}
t \rightarrow & & terms: \\
 & true & constant\ true \\
| & false & constant\ false \\
| & if\ t\ then\ t\ else\ t & conditional \\
| & 0 & constant\ zero \\
| & succ\ t & successor \\
| & pred\ t & predecessor \\
| & iszero\ t & zero\ test
\end{array}
$$

The *values* of a language are a subset of the terms that are possible results of evaluation. As in, the values are the meaning of terms according to the dynamic semantics of the language. The evaluation rules are going to be such that: no evaluation is possible for values, a term to which no evaluation rule applies is a normal form, and all values are normal forms.

This means that the values for the example type system are the following:

$$
\begin{array}{lll}
v \rightarrow & & values: \\
 & true & true\ value \\
| & false & false\ value \\
| & nv & numeric\ value
\end{array}
$$

$$
\begin{array}{lll}
nv \rightarrow & & numericvalues: \\
 & 0 & zero\ value \\
| & succ\ nv & successor\ value
\end{array}
$$

Evaluation Relations
We must now consider the evaluation relations on terms. The expression $t \rightarrow t'$ is an evaluation relation, and it read as "t evaluates to t' in one step. The evaluation relation constitutes an operational (dynamic) semantics for the example language.

All the relations are listed below:

$$\frac{}{if\ true\ then\ t_2\ else\ t_3 \rightarrow t_2}\ \text{E-IFTRUE}$$

$$\frac{}{if\ false\ then\ t_2\ else\ t_3 \rightarrow t_3}\ \text{E-IFFALSE}$$

[1] Lists with elements of differing types

$$\frac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ then\ t_2\ then\ t_3}\ \text{E-IF}$$

It is worth remembering at this point that a mathematical relation can be understood as a (possible infinite) set of pairs of "related things". The idea of our "one step evaluation relation" is that the "related things" are terms and that one term is related to another if and only if the first evaluates to the second in one step. For example, the expression: if true then succ 0 else 0, evaluates to succ 0 in one step. This could be formally specified as:

(if true then succ 0 else 0, succ 0) $\in (\rightarrow)$

As mentioned before, evaluation relations can be infinite, so enumerating all pairs is not appropriate. Instead, schematic inference rules are used to specify relations, like so:

$$\frac{Premise_1\ \ Premise_2\ \ ...\ \ Premise_n}{Conclusion}$$

Schematic means that universally quantified variables are allowed in the rules, for example:
if true then t_2 else $t_3 \rightarrow t_2$.
Such a rule schema actually stands for an infinite set of rules.

The *domain* of a variable is often specified by naming convention. For example, the name of a variable may indicate some specific syntactic category, such as t, v or nv:

$$\frac{t_1 \rightarrow t_1'}{if\ t_1\ then\ t_2\ else\ t_3 \rightarrow if\ t_1'\ then\ t_2\ then\ t_3}\ \text{E-IF}$$

$$\frac{}{pred(succ\ nv_1) \rightarrow nv_1}$$

The other relation rules for our example language are:

$$\frac{t_1 \rightarrow t_1'}{succ\ t_1 \rightarrow succ\ t_1'}\ \text{E-SUCC}$$

$$\frac{}{pred\ 0 \rightarrow 0}\ \text{E-PREDZERO}$$

$$\frac{}{pred\ (succ\ nv_1) \rightarrow nv_1}\ \text{E-PREDSUCC}$$

$$\frac{t_1 \rightarrow t_1'}{pred\ t_1 \rightarrow pred\ t_1'}\ \text{E-PRED}$$

$$\frac{}{iszero\ 0 \rightarrow true}\ \text{E-ISZEROZERO}$$

$$\frac{}{iszero\ (succ\ nv_1) \rightarrow false}\ \text{E-ISZEROSUCC}$$

$$\frac{t_1 \rightarrow t_1'}{iszero\ t_1 \rightarrow iszero\ t_1'}\ \text{E-ISZERO}$$

9.1 Example Evaluations

9.1.1 First Example

Consider the term:

$$\text{pred(pred 0)}$$

To begin evaluation, we must find a term from our rules that encompasses the entire term. For this reason, we cannot evaluate "pred 0", because this would ignore the first "pred". We must use the E-PRED evaluation term.

$$\frac{pred\ 0 \rightarrow}{pred\ (pred\ 0) \rightarrow}\ \text{E-PRED}$$

Here we have only one term to evaluate, the pred(0), and thus can use E-PREDZERO. This returns the value for our expression.

$$\frac{\dfrac{}{pred\ 0 \rightarrow 0}\ \text{E-PREDZERO}}{pred\ (pred\ 0) \rightarrow}\ \text{E-PRED}$$

This value is then propagated down the rest of the expression; we replace all occurrences of "pred 0" with the value 0.

$$\frac{\dfrac{}{pred\ 0 \rightarrow 0}\ \text{E-PREDZERO}}{pred\ (pred\ 0) \rightarrow pred\ 0}\ \text{E-PRED}$$

At this point, we have reached the bottom expression, but this is not yet a value, and must move to a second step of computation:

$$pred\ 0$$

Must like the top expression of the last step, it is possible to evaluate this expression using E-PREDZERO, and this is what we do.

$$\frac{}{pred\ 0 \rightarrow 0}\ \text{E-PREDZERO}$$

This now evaluates to a value; and this evaluation is complete.

9.1.2 Second Example

Consider the term:

$$if\ (iszero\ (pred\ (succ\ 0)))\ then\ (pred\ 0)\ else\ (succ\ 0)$$

To begin evaluation, we must find a term from our rules that encompasses the entire term. As we do not know what the IF expression will evaluate to yet, we must use the generic IF expression. We then begin evaluating the condition of the IF statement, as this will allow us to determine if we should use the E-IFTRUE or E-IFFALSE later.

$$\frac{iszero\ (pred\ (succ\ 0)) \rightarrow}{if\ (iszero\ (pred\ (succ\ 0)))\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow}\ \text{E-IF}$$

This obviously still needs to be evaluated, so we keep applying expressions until we can evaluate one, in this case we apply E-ISZERO and E-PREDSUCC.

$$\dfrac{\dfrac{\dfrac{}{pred\ (succ\ 0) \rightarrow}\ \text{E-PREDSUCC}}{iszero\ (pred\ (succ\ 0)) \rightarrow}\ \text{E-ISZERO}}{if\ (iszero\ (pred\ (succ\ 0)))\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow}\ \text{E-IF}$$

At this point, we can evaluate the expression *pred* (*succ* 0), and begin replacing the values further down the expression tree.

$$\dfrac{\dfrac{\dfrac{}{pred\ (succ\ 0) \rightarrow 0}\ \text{E-PREDSUCC}}{iszero\ (pred\ (succ\ 0)) \rightarrow iszero\ 0}\ \text{E-ISZERO}}{\begin{array}{c}if\ (iszero\ (pred\ (succ\ 0)))\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow \\ if\ (iszero\ 0)\ then\ (pred\ 0)\ else\ (succ\ 0)\end{array}}\ \text{E-IF}$$

We have now reached the bottom expression but have no value, so we begin step two. Again, we have to evaluate the generic IF, as we still do not know the value of (*iszero* 0) is.

$$\dfrac{iszero\ 0 \rightarrow}{if\ (iszero\ 0)\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow}\ \text{E-IF}$$

However, now we *can* evaluate (iszero 0), and replace all it's occurrences down the expression tree.

$$\dfrac{iszero\ 0 \rightarrow true}{if\ (iszero\ 0)\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow if\ (true)\ then\ (pred\ 0)\ else\ (succ\ 0)}\ \text{E-IF}$$

Once again we have reached the bottom expression with no value, so we continue on to step 3. In this step, we now *can* evaluate the IF expression, because we know the value of the condition, in this case, *true*, so we use E-IFTRUE. This evaluates straight away to (*pred* 0).

$$\dfrac{}{if\ (true)\ then\ (pred\ 0)\ else\ (succ\ 0) \rightarrow pred\ 0}\ \text{E-IFTRUE}$$

Still, with no value, we continue to step four, in which we simply evaluate *pred* 0.

$$\dfrac{}{pred\ 0 \rightarrow 0}\ \text{E-PREDZERO}$$

Stuck terms and Values
As mentioned previously, values are terms that cannot be evaluated further. Stuck terms are similar, but they cannot be evaluated further because they are in obviously nonsensical states - it cannot be evaluated further, but it is also not a value (if 0 then pred 0 else 0).

We let the notion of getting stuck be model our run-time errors. The goal of a type system is thus to guarantee that a program never get stuck.

9.2 Types

In our example type system, we know that we have but two types, booleans and natural numbers:

$$\begin{array}{rcll} T & \rightarrow & & types: \\ & & Bool & type\ of\ booleans \\ & | & Nat & type\ of\ natural\ numbers \end{array}$$

We type a *typing relation* between terms, and their types. This is written as $t : T$ and is read as "t has Type T". A term that has a type (i.e, has a typing relation), is said to be well typed. These typing relations are normally defined by schematic typing rules, in the same way we defined evaluation relations earlier.

$$\frac{}{true : Bool}\ \text{T-TRUE}$$

$$\frac{}{false : Bool}\ \text{T-FALSE}$$

$$\frac{t_1 : Bool,\ t_2 : \tau,\ t_3 : \tau}{if\ t_1\ then\ t_2\ else\ t_3 : \tau}\ \text{T-IF}$$

$$\frac{}{0 : Nat}\ \text{T-ZERO}$$

$$\frac{t_1 : Nat}{succ\ t_1 : Nat}\ \text{T-SUCC}$$

$$\frac{t_1 : Nat}{pred\ t_1 : Nat}\ \text{T-PRED}$$

$$\frac{t_1 : Nat}{iszero\ t_1 : Bool}\ \text{T-ISZERO}$$

The most basic property of a type system is "safety"; and by this we mean "well typed programs do not go wrong"; where wrong means entering a stuck state. This term then splits down into two further parts: Progress, where a well-typed term is not stuck; and Preservation, where, if a well-typed term takes a step of evaluation, then the resulting term is also well typed. Together, these two properties say that a well-typed term can never reach a stuck state during evaluation.

Formally, these can be expressed as:

Lemma Progress:
$$\forall t,\ t : T \Rightarrow value(t) \vee \exists t', such\ that\ t \rightarrow t'$$

Lemma Preservation:
$$\forall t,\ t : T \wedge t \rightarrow t' \Rightarrow t' : T$$

Now we will look at the effects of the typing rules of a conditional expression. Below is a typical case when proving progress by induction on a derivation of $t : T$.

Case T-IF:
$$t = if\ t_1\ then\ t_2\ else\ t_3$$
$$t_1 : Bool,\ \ t_2 : T,\ \ t_3 : T$$

By induction hypothesis; either t_1 is a value, or else there is some t_1' such that $t_1 \rightarrow t_1'$. If t_1 is a value, then it must be either true or false, in which case either E-IFTRUE or E-IFFALSE applies to t. On the other hand, if $t_1 \rightarrow t_1'$, then by E-IF, $t \rightarrow if\ t_1'\ then\ t_2\ else\ t_3$

9.3 Exceptions

What happens in type systems when impossible actions are performed? For instance, division by zero, or attempting to index into a position of an array that is out of bounds? These situations can still occur, even in a well-typed system. If the type system does not rule them out, we need to differentiate them from stuck terms, or we can no longer claim that our well typed program does not go wrong.

The basic idea is to allow for *exceptions* to be raised, and make it well-defined what happens when they are. For example, we introduce a new term, *error*, introduce new evaluations rules, such as $head[\,] \rightarrow error$, and the typing rule, $error : T$.

We also have to introduce new *propagation rules*, to ensure that once an error has been raised, the entire program evaluates to *error* - as we do not have an exception handling mechanism. For instance, $pred\ error \rightarrow error$. This also means that our definition of the Progress theorem above, is no longer adequate, and must cope with errors as well.

Lemma Progress-With-Errors:
$$\forall t,\ t : T \Rightarrow value(t) \vee error(t) \vee \exists t', such\ that\ t \rightarrow t'$$

10 Code Generation

Code generation (along with Optimisation) is the final stage of the compiler. The code generator must address the following issues: Code Selection, which code sequence do we generate for each source code phrase; Storage Allocation, where and how do we store data like global and local variables; and Register Allocation, how to allocate registers for variables and other purposes.

10.1 Run-Time Organisation

Code generation is intimately related to the Run-Time Organisation, which, deals with:

1. Memory Organisation
   How to organise memory in data structure for different kinds of storage (stacks, heaps, global statics)

2. Calling Conventions
   Protocols for procedure/function/method calls and returns, including how to pass arguments and how to return results.

3. Data Representation
   How to represent high-level data types, such as integers, arrays, and objects, as sequences of bits

A very basic example of Code Selection (using the TAM language) is shown below:

MiniTriangle Code
$$x := x * 2$$

TAM Code (assuming SB+2 refers to x)
```
LOAD [SB + 2]
LOADL 2
MUL
STORE [SB + 2]
```

10.2 Specifying Code Selection

Code selection is specified inductively over the phrases of a source language. For instance:

$$\begin{array}{rcl} Command & \rightarrow & Identifier := Expression \\ & | & Command; Command \end{array}$$

A *code function* maps a source phrase to an instruction sequence, for example:

$$\begin{array}{lcl} execute & : & Command \rightarrow Instruction* \\ evaluate & : & Expression \rightarrow Instruction* \\ elaborate & : & Declaration \rightarrow Instruction* \end{array}$$

These code function do not, however, actually execute, evaluate, or elaborate the expressions or commands they are applied to. Instead they generate the code for these actions to be undertaken.

Code functions are specified by means of code templates. An example code template for the execute command has been given below. The double square brackets $[\![$ and $]\!]$, enclose pieces of concrete syntax (like the semi colon), and meta variables (like C_1 and C_2). It is worth mentioning that these expressions are evaluated recursively, and that they are inductive defined over the underlying phrase structure.

$[\![\ ]\!]$ can be thought of as a map from concrete to abstract syntax, as specified by the abstract syntax grammars.

$$\begin{array}{l} execute\ [\![\ C_1\ ;\ C_2\ ]\!] = \\ \quad execute\ C_1 \\ \quad execute\ C_2 \end{array}$$

10.2.1 Code Selection Example

Given the following two templates:

$$\begin{array}{l} execute\ [\![\ C_1\ ;\ C_2\ ]\!] = \\ \quad execute\ C_1 \\ \quad execute\ C_2 \end{array} \qquad \begin{array}{l} execute\ [\![\ I\ := E\ ]\!] = \\ \quad evaluate\ E \\ \quad STORE\ addr(I) \end{array}$$

Given that $addr$(f) $= [SB + 11]$ and $addr$(n) $= [SB + 17]$. Generate code for the following MiniTriangle fragment, and expand as far as the above templates allow.

$$\begin{array}{l} f\ := f\ *\ n; \\ n\ := n\ -1 \end{array}$$

$$\begin{array}{ll} = & \{\quad \text{Substitute Values into suitable instruction} \quad \} \\ & execute\ [\![\ f\ := f\ *\ n;\ n\ := n\ -1\ ]\!] = \\ = & \{\quad \text{Definition of } execute\ [\![\ C_1\ ;\ C_2\ ]\!] \quad \} \\ & \quad execute\ [\![\ f\ := f\ *\ n;\ n\ := n\ -1\ ]\!] = \\ & \qquad execute\ [\![\ f\ := f\ *\ n\ ]\!] \\ & \qquad execute\ [\![\ n\ := n\ -1\ ]\!] \end{array}$$

At this point, we have applied our rule according to the template. But it should be obvious that this has not been expanded as far as possible - there are still matchable patterns. So we continue our computation.

$$
\begin{array}{ll}
 & \mathit{execute}\ [\![\ f\ :=\ f\ *\ n;\ n\ :=\ n\ -1\ ]\!] = \\
 & \quad \mathit{execute}\ [\![\ f\ :=\ f\ *\ n\ ]\!] \\
 & \quad \mathit{execute}\ [\![\ n\ :=\ n\ -1\ ]\!] \\
= & \quad \{\quad \text{Definition of } \mathit{execute}\ [\![\ I\ :=\ E\ ]\!] \quad \} \\
 & \mathit{evaluate}\ [\![\ f\ *\ n\ ]\!] \\
 & \quad \text{STORE } \mathit{addr}([\![\ \text{f}\ ]\!]) \\
 & \mathit{evaluate}\ [\![\ n\ -1\ ]\!] \\
 & \quad \text{STORE } \mathit{addr}([\![\ \text{n}\ ]\!])
\end{array}
$$

Alternatively, we can include the addressing rules that we specified:

$$
\begin{array}{ll}
= & \{\quad \text{Definition of } \mathit{execute}\ [\![\ I\ :=\ E\ ]\!] \text{; and } \mathit{addr} \quad \} \\
 & \mathit{evaluate}\ [\![\ f\ *\ n\ ]\!] \\
 & \quad \text{STORE [SB + 11]} \\
 & \mathit{evaluate}\ [\![\ n\ -1\ ]\!] \\
 & \quad \text{STORE [SB + 17]}
\end{array}
$$

10.3 A problem approaches...

In our typing rules, we state that *addr* (type, $Identifier \rightarrow Address$) maps our identifier to an address. However, how can this function do this? You simply cannot get a stack location from an identifier.

This means that: *elaborate* is responsible for assigning addresses to variables and a function like *addr* needs access to the addresses assigned by *elaborate*. However, the type signature for the code functions do not permit this communication. As a direct result of this: the code functions need to be extended to allow for an additional stack environment argument, associating variables with addresses; the code function *elaborate* must return; and we now need to keep track of the current stack depth (with respect to LB) to allow *elaborate* to determine the address for a new variable. We will also need to keep track of the current scope level, as the difference of current level and the scope level of a variable is needed, in addition to its address, to access it.

10.4 Symbols

Symbols are different to identifier, as they carry semantic information along with them (for instance, type and scope level). This information is then readily available for the compiler when it needs it, like when generating code. There are two main types of symbols:

1. External - defined outside the current compilation unit (like a library)
   External symbols are known entities, and thus can be looked up once and for all (during identification). They have some value, such as an address.

2. Internal - defined in the current compilation unit (like a *let*)
   Internal symbols do not carry any value, because this is not computed until the time of code generation. This information has to be looked up in the code generation environment.

11 Run Time Organisation

11.1 Storage Areas

There are three main varieties of storage in the context of compilers and computer architecture.

1. Static Storage
   Storage for entities that live throughout an execution
2. Stack Storage
   Storage allocated dynamically, but deallocation must be carried out in the opposite order to allocation
3. Heap Storage
   Region of the memory where entities can be allocated and deallocated dynamically as needed, in any order

11.1.1 Life Time

Lifetime is the scope under which a variable is accessible during run time. Consider the following code fragment:

Example 1

```
var x, y, z: ...

proc P()
        var p1, p2: ...
        begin ... end

proc Q()
        var q1, q2: ...
        begin ... if ... Q(); ... end

proc R()
        var r1, r2: ...
        begin ... Q() ... end

begin ... P() ... R() ... end
```

The life time for the different variables is:

Example 2
For another example, consider the following Java code

```
private static Integer foo(int i){
        Integer n = new Integer(i);
        return n;
}
```

The lifetimes of the argument i and the local variable n coincide with the invocation of foo. The lifetime of the integer *object* created by *new* starts when *new* is executed and ends when the last reference to the objects disappears.

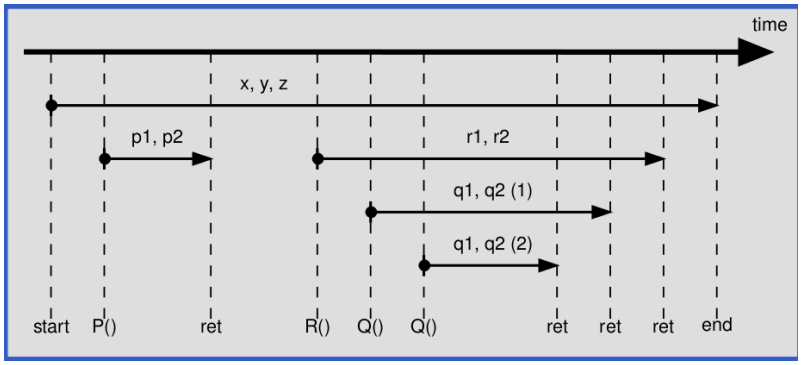


Figure 7: Life time of variables from the example code

Thus, the lifetime of the integer object goes past the end of the invocation of foo.

11.2 Storage Allocation

Global variables exist throughout the program's run time. This means that the location to store this variable can be decided statically, at compile (or link) time - once and for all.

For example:

```
private static String[] tokenTable = ...
```

Arguments and local variables, on the other hand, only exist during a function (or procedure or method) invocation. Function calls are properly nested, and, in the case of recursion, a function may be re-entered any number of times. Each function activation needs a private set of arguments and local variables. These observations suggest that storage for arguments and local variables should be allocated on a stack.

When the lifetime does not coincide with procedure/function invocations, heap allocation is needed. For example: objects in object-oriented languages, function closures, and storage allocated by procedure like *malloc* in C. Such storage is either explicitly deallocated when no longer needed, or automatically reclaimed by the garbage collector.

11.3 Stack Frames

One stack frame, or activation record, is needed for each currently active function/procedure/method. These contain: arguments to the function, book keeping information like the return address, dynamic link and static link, and the local variables of the function. A temporary workspace is also included.

The stack is usually defined by a handful of registers, dictated by the CPU architecture and/or conventions. For example: SB meaning stack base, ST, meaning stack top, and LB meaning local base. These names can, obviously, vary between implementation. Often SP, the stack pointer is used in stead of the Stack top, and FP the frame pointer, is used instead of local base.

The general structure of a stack frame is as follows:

Address	**Contents**
LB - $argOffset$	arguments
...	...
LB	static link
LB + 1	dynamic link
LB + 2	return address
LB + 3	local variables
...	...
LB + $tempOffset$	temporary storage

Where $argOffset$ is number of arguments, and $tempOffset$ is 3+ the number of local variables.TAM uses this convention (addresses are offset in *words* - 4 bytes).

11.3.1 Stack Frame Example

Imagine the following pseudo-MiniTriangle code:

```
var n : Integer;
...
fun f (x, y: Integer) : Integer =
        let
                z : Integer
        in begin
                z := x * x + y * y;
                return n * z
        end
```

A call for $f(3,7) * 8$, would look like:

```
2015  LOADL 3 ; 1st arg (x)
2016  LOADL 7 ; 2nd arg (y)
2017  Call f
2018  LOADL 8
2019  MUL
```

As you can see, the address of each instruction is explicitly indicated on the left. The address of f here is given by a symbolic label, which corresponds to the address at which the code for f is defined (say, 2082).

The stack layout for this would look like:

Address	**Contents**
...	
SB + 42	n : n
...	
LB - 2	x: 3
LB - 1	y: 7
LB	static link
LB + 1	dynamic link
LB + 2	return address = 2018

The return address is the previous program counter, which is the address of the instruction immediately after the function call. The new program counter is then set to be the first instruction of the function f (2082).

The associated TAM code for this would be:

```
LOADL 0                          ADD
LOAD [LB - 2]; x                 STORE [LB + 3]; z
LOAD [LB - 2]; x                 LOAD [SB + 42]; n
MUL                              LOAD [LB + 3]; z
LOAD [LB - 1]; y                 MUL
LOAD [LB - 1]; y                 POP 1 1
MUL                              RETURN 1 2
```

RETURN replaces the activation record (the frame) of f, by the result, restores the local base, and jumps to the return address (2018). It is worth noting that all variable offsets here are static.

Dynamic Links
Value to which LB (local base) is restored by RETURN when exiting a procedure - it is the address of the calling frame. It is known as dynamic because it depends on the function it was called from.

Static Links
Base of underlying frame function that immediately lexically encloses this one. They are known as static because depends on the programs structure, and not on its execution. They are used to determine addresses of variables of lexically enclosing functions.

11.4 Non-Local Variable Access

Consider the nested procedures:

```
proc P()
        var x, y, z: Integer

        proc Q()
                ...
                begin ... if ... Q() ... end

        proc R()
                ...
                begin .. Q() ... end

        begin ... Q() ... R() ... end
```

P's variables are in scope also Q and R. But they are neither global, nor local, so how can we access them? Luckily, they belong to the lexically enclosing procedure.

In particular, we cannot access x, y, or z relative to the stack base (SB) since we cannot (in general) know if P was called directly from the main program or indirectly via one or more other procedures. For example, there could be arbitrarily many stack frames below P's frame.

However: the static links in Q's and R's frames are set to point to P's frame on each activation. The static link in P's frame is set to point to the frame of its closest lexically enclosing procedure, and so on and so on. Thus, by following the chain of static links, one can access variables at any levels of a nested scope.

As another example, imagine this:

```
proc P()
        var x, y, z : Integer

        proc Q()
                proc R()
                        ...
                        begin ... if ... R() ... end
                ...
                begin ... R() ... end
        begin ... Q() ... end
```

In this case, Q's variables are now in scope in R. To access them, we copute the difference in scope levels between the accessing procedure and the accessed variable (this is static information), and follow that many static links.

12 Data Representation

The object of data representation is the store various kinds of data (integers, characters, strings, arrays, etc). To do this, we use the program memory. Each individual address in the memory can have some value, but we need to encode the data to be stored. However, we must be wary of the two following facts:

1. Nonconfusion: Different values of a given type must have different representations
2. Uniqueness: Each value should have exactly one representation.

Obviously, if two different values are represented the same way, they cannot be told apart. In dynamically checked languages, every possible value must have a distinct representation. Statically typed languages, however, only require values of the same type to have distinct representations - the same representation may be reused for values of different types.

For an example, imagine that characters and small integers were represented by 8-bit bytes. In this case, the character 'A' and the integer 65 would both be represented by 01000001. If we were to print this value, how would be know whether to print the integer or the character? There is no way to tell the representation of 'A' and 65 apart in a dynamically checked setting. In static languages, however, we can use the type to disambiguate.

Now, consider the two enumerations:

$$\begin{aligned} &data\ Color = Red \mid Green \\ &data\ Size = Small \mid Large \end{aligned}$$

Intuitively, it is always the case that the way *Red* is represented must be different to the way that *Green*. Likewise, *Small* and *Large* must be represented differently. In addition to this, in a dynamically checked setting: The representations of *Red* and *Green* must not be the same as either *Small* or *Large*. Depicted by:

$$\{repr(Red),\ repr(Green)\} \cap \{repr(Small),\ repr(Large)\}$$

Other issues in data representation include: constant-size representation (representation of all values of a given type occupy the same amount of space). This enables the compiler to statically plan storage allocation (since the type and, hence, size is known statically). However, this can be wasteful.

12.1 Direct and Indirect Representations

When specifying the representation of a piece of data can be done in one of two ways: direct, or indirect. In direct representation, the representation of some value, is simply the binary representation of that value. This is efficient to access, and means no heap allocation or deallocation overhead is present.

In an indirect representation, a handle is used, which points to a binary representation of the value on a stack or in a heap. This allows the support of varying size data (like dynamic arrays), recursive types (like linked lists and trees), and facilitates the implementation of parametric polymorphism (as handles can be uniform).

12.2 Representing Primitive Types

Primitive types are often supported directly by the underlying hardware. For instance, 32-bit machines generally support the addressing of 8-bit bytes and 32-bit words, 32-bit twos-complement integer arithmetic, and 64-bit floating point operations. There are also standard encoding conventions, such as the 7-bit ASCII or 8-bit ISO character codes, or the Unicode standard. Adopting such conventions facilitates interoperability and communication.

12.3 Representing Records (Object fields)

Records like the example below, are represented in memory by a sequence of individual fields.

$$
\begin{array}{l}
type\ Date = record \\
\quad y:\ Integer, \\
\quad m:\ Integer, \\
\quad d:\ Integer, \\
end; \\
type\ Details = record \\
\quad female:\ Boolean, \\
\quad dob:\ Date, \\
\quad status:\ Char \\
end;
\end{array}
$$

The problem with this approach is alignment restrictions - where restrictions are present from the underlying architecture. We could “relax” this; but this may require extra work

12.4 Alignment

An address, a, is n-byte aligned if, and only if, $a \equiv 0 (mod\ n)$ A variable/field is n-byte aligned if and only if it is stored starting at an n-byte aligned address. To satisfy alignment requirements of its components, a variables of aggregate type like a record is often aligned according to the maximum alignment of its components. Padding is often needed between variables/components to ensure the alignment requirements of each is met. As an example, if we think back to the records we defined in the previous section: assuming that 1 word = 1 byte = 32 bit Integer, 1 byte = 8 bit Boolean and Char, and that Integers must be word aligned. What would alignment and size of the Date type be?

As all the variables are Integers, and they all have a size of 32 bits, and we have 3. The size of the Date type would be 96 bits (3 * 32).

The size of the Details data type, would be 96 bits (the size of the Date type), plus 16 bits (because the single Char is 8 bits, and the single Boolean is 8 bits). However, because the Integer needs to be word aligned, we require some padding, which increases the size of the type to be 160 bits. The example shows the addresses of the variables in relation to some Details type x. You'll see that the padding is added so that any integer coincides with a word boundary.

variable	address	contents
x.female	addr(x)	1 (Boolean true)
padding	addr(x) + 1	padding
padding	addr(x) + 2	padding
padding	addr(x) + 3	padding
x.dob.y	addr(x) + 4	1984
x.dob.m	addr(x) + 8	7
x.dob.d	addr(x) + 12	25
x.status	addr(x) + 16	117 ('u').

12.5 Representing Arrays

Arrays are represented by a sequence of representations of individual array elements. We have two cases to consider when representing our arrays, both of which are discussed below. In either case, When accessing array elements, we must ensure indices are within the bounds of the array. This is computed from the base address of the array, the index of the element to be accessed, and the size of the elements in the array.

12.5.1 Static arrays

, A static array is one which the number of elements is known at compiler time. Given some static array $T[n]$ x (named x, with n elements, of type T). The required storage will be $n * sizeOf(T)$, and the processing of accessing some elemtn $x[i]$ is to first verify that $0 \leq i \leq (n-1)$, then to compute the address of the desired element, which would be represented by $addr(x[0]) + (i * sizeOf(T))$, and then fetch/store the value at that address.

12.5.2 Dynamic Arrays

A dynamic array is one which the number of elements is determined at run time. To represent a dynamic array, we use an indirect representation approach, and access via a handle. The handle itself has a fixed size, and contains a pointer to an "array proper" (the storage of which is allocated at run time), and the bounds of the array. Index is checked by comparing it with the array bounds stored in the handle.

12.6 Representing Disjoint Unions

A disjoint union consists of two parts: a tag, and a variant. The value of the tag determines the type of the variant. Disjoint unions occurs as: variant records in Pascal and Ada; algebraic data types in Haskell and ML; and as object types in OO languages like Java and C#. Disjoint unions can be represented like a record in which the value of the tag field determines the layout of the rest of the record. If a constant size is necessary, we allow the size to be the maximal size over the various possible layouts.

12.7 Representing Recursive Types

A recursive type is one defined in terms of itself, for example, linked lists, and trees. Recursive types are usually represented indirectly, since this allows values of arbitrary size to be referenced through a fixed size handle. In languages like C or Pascal, the programmer needs to introduce the indirect representation explicitly through pointer types.

12.8 Uniform Representation

Languages like Haskell and ML adopt a uniform data representation: all values (Even primitive ones) have an indirect representation with a pointer. This facilitates parametric polymorphism, this means functions such as the identity function (id x = x), can be compiled to a single piece of code that works for values of any type, because all values are represented in the same way.

Recursive types are supported automatically, because everything is already a pointer. Many OO languages adopt a *mostly* uniform representation in which all objects are represented by pointers, meaning an object of some class is also an object of any of the superclasses.

13 Register Allocation

A register is one of a small number of very fast storage elements internal to a CPU. Register allocation is the process of deciding which registers to use for what purpose, and when to do this. Most computers nowadays are register machines, and most instructions target certain registers, which are specified as arguments to the instruction. For instance, the instruction 'ADD R3, R1, R2', takes the value of R1, and the value of R2, adds these together, and stores it in R3. Generally, there are very few registers (typically 8-32), which is problematic when most modern programs often use hundreds of Megabytes of memory. In addition to this, some registers are special purpose registers, that are used for program structure.

Obviously, there are not enough registers to keep all data in registers all the time, which means most of the data must be stored in the main memory. However, we still need to use some registers, because of the ways the instruction set is designed. We try to optimise the program by using as many registers as possible, because resisters are very fast, and it is preferable to use registers for frequently used data, instead of seldom used data.

This means, as mentioned, that register allocation is an optimisation problem. We have to minimise the memory traffic (loads and stores to memory) by using registers, but we have to not exceed the available number of registers, which is further limited by some of them being reserved as special purpose registers. It is often the case that minimising the number of load and store instructions usually reduces the execution time of the program.

Assume some simple register machine, in which there are n general purpose registers, SB for the Stack Base, LB for the Local Base and ST for the stack top. If we have the following instructions (where the offset d, is in bytes):

load Ri, [R$j + d$]
store Ri, [R$j + d$]
add Ri, Rj, Rk
mul Ri, Rj, Rk

Given the following:

Variable	Address
x	SB + 0
y	SB + 4
z	SB + 8

And some general purpose registers, R0, R1, ..., R9, the code for the expression $z := z * (x + y)$, would be:

```
load R0, [SB + 0] ; this is x
load R1, [SB + 4] ; this is y
add R2, R0, R1
load R3, [SB + 8] ; this is z
mul R4, R3, R2
store R4, [SB + 8] ; store in z
```

But it should be obvious that this can be optimised to use less registers, like so:

```
load R0, [SB + 0] ; this is x
load R1, [SB + 4] ; this is y
add R0, R0, R1
load R1, [SB + 8] ; this is z
mul R1, R1, R0
store R1, [SB + 8] ; store in z
```

13.1 Naive Code Generation

We can implement a code generation function in a relatively simple way, that is similar to the stack machine code generator. The only difference is that in register code generation, the result of our operation must have a specified register, in which the result will be stored in. Imagine we had the following code to generate the "Add" instruction:

$$
\begin{aligned}
&evaluate\ [\![E_1 + E_2]\!] = do \\
&\quad r_1 \leftarrow evaluate\ E_1 \\
&\quad r_2 \leftarrow evaluate\ E_2 \\
&\quad r \leftarrow freeReg \\
&\quad emit\ (Add\ r\ r_1\ r_2) \\
&\quad return\ r
\end{aligned}
$$

In which the function $freeReg$, returns some previously unused register.

If we were to generate code for the following pseudo-MiniTriangle snippet:

$$
\begin{aligned}
&z\ :=\ x * x + y * y; \\
&return\ n * z
\end{aligned}
$$

This would be generated to something like...

```
load R0, [LB-8]
load R1, [LB-8]
mul R2, R0, R1
load R3, [LB-4]
load R4, [LB-4]
mul R5, R3, R4
add R6, R2, R5
store R6, [LB+12]
load R7, [SB+168]
load R8, [LB+12]
mul R9, R7, R8
```

We know that reading and writing to memory is extremely slow, when compared to reading/writing to register. From this fact, it is obvious that this code is extremely inefficient, because there are many unnecessary memory accesses. The naive code generator could also very easily run out of registers (this example has already used 10), as it assumes there is always some free register, which there obviously is not.

Instead, it would be much better to have some sort of ad-hoc register allocation. In our example, this would mean allocating a specific register for x and y, meaning the reading of them only needs to happen once each, and to allocate a register for z, so we don't have to keep writing it to memory. These techniques would allow us to have fewer loads and stores, shorter code overall, and fewer registers used.

13.2 Registers across calls

Assume that we have some calling convention, in which the first three arguments of a function are passed in registers R0, R1, and R2, and the result is returned in R0. Imagine the following code fragment.

```
add   R5, R6, R7      R5 = x + y
load  R0, [SB+168]    R0 = n
call  factorial       R0 = n!
mul   R0, R5, R0      R0 = (x+y) * n!
```

Problems start to arise if the factorial function uses R5 across the call. However, there are two basic approaches to solving this problem: caller saves, and callee saves. In caller saves, the caller saves registers that are in use; although this has the potential of saving registers that the callee never attempts to use. In callee saves, the callee saves the registers it will use, but this have the potential of saving registers that were never in use in the caller. In practice, a mixture of both of these methods if often adopted, where some registers are designated as caller saves, and others as callee saves. Assuming that R5 is our caller-saves register, and the only register in use across the call, our above code fragment becomes (assuming LB+30 is some address of free space):

```
add   R5, R6, R7      R5 = x + y
load  R0, [SB+168]    R0 = n
store R5, [LB+30]     Save R5
call  factorial       R0 = n!
load  R5, [LB+30]     Restore R5
mul   R0, R5, R0      R0 = (x+y) * n!
```

We now need to worry about how to automatically decide which registers to use, and how we can decrease the total number of registers used (as there is only a limited number). The number of registers used by a code fragment is known as *register pressure*, and it is, intuitively, desirable to keep register pressure as low as possible. Minimising the pressure maximises the size of the code for which no auxiliary storage is needed, and is means fewer registers are needed to be preserved across subroutine calls.

13.3 Liveness

The liveness of a variable and intermediate results are important to make it possible to use one register for many purposes. If a result is no longer live, there is no need to keep it in registers. We say that a variable, v, is live at point, p, if there is an execution path from p to a use of v along which v is not updated. For example:

```
1  x := 3 * m;
2  y := 42 + x;
3  z := y * x;
4  if z > 0 then u := x else u := 0;
5  y := u;
6  return y;
```

In this case, x is live immediately before line 4 because it *may* be used in line 4. On the other hand, y is dead immediately before line 4, because y is updated before being updated again. Variable u is dead before line 4 because it is updated in both branches of the if-statement in line 4.

13.4 Graph Colouring

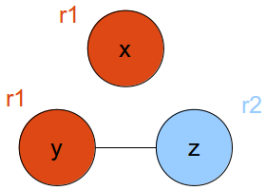
Graph colouring is a common approach for register allocation, of which the basis idea is to represent each variable by a node in a graph, called an interference graph. If two variables are live simultaneously an edge is added between the nodes representing them. The graph is then coloured so that no two adjacent nodes get the same colour, using as few colours as possible. Each of these colours then corresponds to a register. Bare in mind this is a very difficult optimization problem (but is NP-complete!).

13.4.1 Interference Graph Example

Consider the following code fragment:

```
1  y := x * x;
2  z := y + 42;
3  return y * z;
```

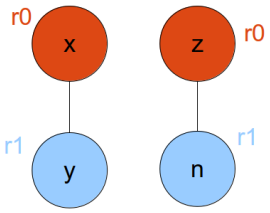
The interference graph would thus have y and z connected, as they are live simultaneously. The variable x is connected to no other nodes, as this is not live at the same time as any other value. This means our graph can have only two colours, as it is only y and z that cannot be the same, whereas x can take on the colour of either y or z. We can then use this to allocate registers, like so:




```
1  R1 := R1 * R1;
2  R2 := R1 + 42;
3  return R1 * R2;
```

As another example:

$$z := x * x + y * y;$$
$$return\ n * z$$




```
1  R0 := (R0 * R0) + (R1 * R1);
2  load R1, n;
2  return R1 * R0;
```

Code generation generally proceeds in to passes: firstly, we generate code assuming arbitrarily many virtual registers (essentially the naive approach we looked at earlier); followed by, use graph colouring to bind each virtual register to a physical register.

13.5 Register Spilling

What happens when the number of registers used, exceeds the number of available registers - we go through a process called register spilling. This is the process of storing the contents of a register into memory, to free it and reduce the register pressure. Intermediate results are stored into the "temporary" storage areas of stack frame/activation record. This is another hard optimization problem, as we have to decide which registers to spill.

14 Optimisation

The code generated by a compiler must be correct, and should also run fast, be small, and use as little space as possible. Code improvement is the process of improving the time and/or space behaviour of generated code, without changing its functional behaviour (correctness must be preserved with all the optimisations made). Looking at the code example; it can be said, that, in the inner loop, the value of i is constant - it never changes in that loop. Therefore, we could optimise this code by moving the expression $x := f(w * 10) * a[i]$; outside of this inner loop. This optimises our code and decreases the number of calculations from 20,000 calculations, to only 100 (associated with that expression, at least).

```
w := 42;
i := 0;
while ( i < 100 ) do begin
    j := 0;
    while ( j < 200 ) do begin
        x := (w * 10) * a[i];
        y := y + x + b[j];
        j := j + 1;
    end;
    i := i + 1;
end
```

Looking at a different type of optimisation: if a function is called twice, and the result of this function is added to itself (for instance, f(x) + f(x)); would it not be possible to simply call the function and double the result? (2 * f(x)). Unfortunately, optimisations are not always this simply, and we must inspect the code itself before making any large changes like this. Especially if a function has side effects. Imagine if the function, f, from earlier, also updated a global variable? In our second code snippet, the global variable would only be updated once, instead of twice as in the original code. Side effects can include a number of different things, including: updating of variables or data structures, I/O and other changes to the system state, exceptions and even non-termination of programs. This shows us that optimisation is a careful and delicate process, and is not always easy, or obvious.

Code improvement is generally referred to as optimisation, but there are a few key flaws with it, that should be made known. For instance, it is hardly ever possible to guarantee optimally under any mathematical measure. The optimised code is not necessarily an improvement on the original code, as it is not known what is going to happen at run-time, instead, "optimising" deals with the average expected case. The only way to truly test our optimisations is through the use of careful and extensive benchmarking. This means that optimisation is not an entirely formal process, and contains a lot of heuristics, and "best-guesses".

Code improvements can be carried out at multiple different levels

1. High Level
   Source-to-source (AST) transformations, looking at the source code of the program to compile, and looking for any potential optimisations here (like the looping example earlier)

2. Intermediate Level
   Transformations in intermediate representation, such as the "bare-bones" high level language, and graph representations of control and data flow

3. Low Level
   Transformations that take place on machine code

Consider the following code fragment:

```
if x then
    if y then
        putint(1)
    else
        putint(2)
else
    putint(3)
```

In this particular source code, there are no optimisations that can be put in place. The code has no *obvious* mistakes or optimisation issues. So, we compile this all the way to machine code (in this case, TAM).

```
LOAD [SB + 12]          #2: LOADL 2
JUMPIFZ #0                  CALL putint
LOADL [SB + 13]         #3: JUMP #1
JUMPIFZ #2              #0: LOADL 3
LOADL 1                     CALL putint
CALL putint             #1: EXIT
JUMP #3
```

In this code, a single *small* improvement can be made, in one of the jumps. We jump to label #3, which then jumps again directly to label #1. We can optimise this to be a single command, that jumps straight to #1. This shows us, that not all optimisations are huge, but are still helpful.

```
LOAD [SB + 12]          #2: LOADL 2
JUMPIFZ #0                  CALL putint
LOADL [SB + 13]         #3: JUMP #1
JUMPIFZ #2              #0: LOADL 3
LOADL 1                     CALL putint
CALL putint             #1: EXIT
JUMP #1
```

Different levels of code, make different levels of information available, which allows us different categories to optimise. For instance,

1. High Level
   Access to type information
2. Intermediate level
   Results of control/data flow analysis, and explicit index calculations and bound checking
3. Low level
   Results of code selection and register allocation

These differing levels are more or less suitable depending on the kind of optimisation in question. A good compiler will attempt to carry out optimisations on every level.

Low Level Optimizations

1. Delay Slots
   A delay slow is an instruction that gets executed without the effects of the preceding instruction. That is, a single arbitrary instruction location immediately after a branch will be executed, even if the preceding branch is taken. The makes the code appear to execute in an illogical or incorrect order. This is so that the goal of a pipelined architecture (to complete an instruction every clock cycle) is maintained, by ensuring the pipeline is full of instructions.

Original Code

```
add    %l1, 4, %l1
subcc  $l2, 1, %l2
bne    1b
nop
```

Delay Slot optimisations

```
subcc  $l2, 1, %l2
bne    1b
add    %l1, 4, %l1
```

2. Static instruction scheduling
   Deals with the ordering of instructions, to reach optimality. Specifically, we can "hide" read delays. In the example, we see how we put a read command as far prior to use of the read address as possible. This means, when the use of that read address is used, there is no delay to read it, as it has already been read.

Original Code

```
add     %l1, 4, %l1
ld      [%l2], %l0
add     %l1, %l0, %l0
```

Static instruction optimisations

```
ld      [%l2], %l0
add     %l1, 4, %l1
add     %l1, %l0, %l0
```

3. Peep hole optimisations
   Strictly local improvements of short instruction sequences.

Original Code

```
add     %l1, 4, %l1
sub     %l1, 4, %l1
```

Static instruction optimisations

```
nothing!
```

None of these optimisations can be expressed at a higher level, but do not forget to be very careful when applying these optimisations!

14.1 Intermediate Level Optimizations

Information that was implicit in the high level representation of the code, might become explicit at the intermediate level. This enables us to apply a different set of optimisations. One of these optimisations, in array indexing. Consider the following code:

```
var x, y: array[1..100] Integer;
...
a := x[i] + y[i];
```

This code itself, cannot be optimised, but we *can* optimise the intermediate-level representation, by manipulation of pointer arithmetic.

```
if (i < 1 || i > 100) then raise index_bounds;
t1 := (x + 4 * (i - 1));
if (i < 1 || i > 100) then raise index_bounds;
t2 := (y + 4 * (i - 1));
a := t1 + t2;
```

This code now makes two things explicit: the first is that each time we index into an array, we check we are not going out of bounds; and, the computation of the address of the array variables. Both of these actions take place for the indexing into each of the arrays, but this can obviously be optimised by combining the operations, and abstracting out the pointer arithmetic, this produces the following optimised code:

```
if (i < 1 || i > 100) then raise index_bounds;
t0 := 4 * (i - 1);
t1 := (x + t0);
t2 := (y + t0);
a := t1 + t2;
```

In this new code, we only check the index bounds once (because the arrays are both of the same size), and abstract the calculation of the address, so that the number of calculations that take place is decreased.

14.2 High Level Optimizations

14.2.1 Constant Folding

The basic idea of constant folding is to evaluate expressions/sub expressions at compile time, where possible, especially where constants are involved. For instance, if the value of π is used in code, any expressions that involve π (excluding variables), can be evaluated at compile time; as shown in the example.

Original Code

```
const pi: Double = 3.1416;
var volume, radius: Double;
...
volume := 4/3 * pi * radius^3;
```

Constant folded code

```
const pi: Double = 3.1416;
var volume, radius: Double;
...
volume := 4.1888 * radius^3;
```

This helps to reduce the number of calculations necessary at run time.

This technique, however, is not necessarily only limited to declared constants - sometimes constants[2] emerge in ordinary variables. That is, when a variable is assigned, and then used by other variables (without changing its value), it can be considered a "compile time constant" for the duration of those operations. This allows us to do the following optimisations;

Original Code

```
x := 3;
y := x + 1;
x := x * 2;
```

Constant folded code

```
x := 3;
y := 4;
x := 6;
```

That last example was very clear in what could be optimised, but more complex examples are not so clear, and require flow analysis. If you look at the code optimisations below, you see that only a very small optimisation can be made. This is because we have a loop, meaning that x is no longer a constant at compile time. Some extra optimisations may be possible if the value of z is known, but that is not the case in this example.

Original Code

```
x := 3;
y := x + 1;
while ( x < z ) begin
    x := x * 2;
end
```

Constant folded code

```
x := 3;
y := 4;
while ( x < z ) begin
    x := x * 2;
end
```

[2]Not strictly constants, but constant for the purpose of the code block

It is important, like in all optimisations, to avoid changing the semantics of the program. For instance, the implementation of compile time floating point/integer precision, must agree with that of the run time precision. We have to worry about cross compilation issues, and potential rounding errors that can occur from this. We also have to worry about arithmetic exceptions, like division by zero.

14.2.2 Common Sub-expression Elimination

The idea behind this optimisation technique is to avoid expressions that are repeated, and the minimisation of this repetition. We generally abstract out the repeated expressions, and apply these more generally. We saw an example of this in the address computation of intermediate code in the previous section, and this is a common place for it to occur (as you would generally hope that a good programmer would not write code that had these obvious optimisation issues).

Original Code

```
x1 := y1 + 7 * z + 42;
x2 := y2 + 7 * z + 42;
```

Common Sub-expression Eliminated code

```
t := 7 * z + 42;
x1 := y1 + t;
x2 := y2 + t;
```

Bare in mind, however, that the expression must not only be syntactically the same, but they must also mean the same thing. For instance, scope rules must be taken into account. In the example below, we see two expressions of $y * 17$, but due to their scoping, they are different, and cannot be abstracted.

```
let x = y * 17 in
    let y = 13 in
        let z = y * 17
```

It is also important to take side effects into account (which is where the flow analysis comes in). In the two examples below, we cannot apply common sub-expression elimination, as the sub expression do not compute to the same value.

Example 1

```
x := y * 17 + 3;
y := y + 1;
z := y * 17 + 3;
```

Example 2

```
x := y++ * 17 + 3;
z := y++ * 17 + 3;
```

Example Below is a fragment of source code, what optimisations can be made? (Both Constant folding and common sub-expression elimination can be applied).

```
y := 20;
x := y - 10;
while ( y < 3 * x + z ) do begin
    z1 := (x + 10) * y + 5;
    z2 := ((x + 10) * y + (2+3)) * z1;
    w := w + z2 - z1;
    y := y + 1
end
```

Firstly, we will apply the simplest of compile-time constants to the program, to remove the run time calculation of y.

```
y := 20;
x := 10;
while ( y < 3 * x + z ) do begin
    z1 := (x + 10) * y + 5;
    z2 := ((x + 10) * y + (2+3)) * z1;
    w := w + z2 - z1;
    y := y + 1
end
```

After this, knowing that the value of x does not change during the loop, we can treat it as a compile time constant. We can also treat all arithmetic between constants as compile time constant, giving us the following code: (all values of x, and calculations involving x have been converted to constants)

```
y := 20;
x := 10;
while ( y < 30 + z ) do begin
    z1 := 20 * y + 5;
    z2 := (20 * y + 5) * z1;
    w := w + z2 - z1;
    y := y + 1
end
```

The final optimisation is one of common sub-expression elimination, in line 5, we see the use of (20 * y + 5), which just so happens to have been assigned to a variable in the line above. Because there are no side effects between lines 4 and 5, we can abstract the value of (20 * y + 5), giving us the final optimised code of:

```
y := 20;
x := 10;
while ( y < 30 + z ) do begin
    z1 := 20 * y + 5;
    z2 := z1 * z1;
    w := w + z2 - z1;
    y := y + 1
end
```

14.2.3 Algebraic Identities

An algebraic identity is an relation that states that both elements of the relation are equal (for instance, the identity of x, is x, or that $1 * x - 0 \Rightarrow x$). In optimisation, this refers to expressions that evaluate to the same thing, algebraically (this is similar to common sub-expression elimination). The following optimisation details this:

Original Code

```
x := (z + 2) * i;
y := (2 + z)  * j;
```

Algebraically Identified Code

```
t := z + 2;
x := t * i;
y := t * j;
```

In this example, we see how the symmetry of $2 * z = z * 2$ is exploited, with help from common sub-expression elimination.

However, it important to realise that sometimes, computer arithmetic does not always obey the general algebraic laws. For instance, in mathematics, it is intuitively obvious, because of the transitivity of plus, that $x + (y + z) = (x + y) + z$. However, in computer arithmetic, this is not always the case. If x and y are large position numbers, and z is a large negative number, then $(x + y) + z$ might result in an overflow/underflow trap, whereas $x + (y + z)$ would not. We also must consider when floating point numbers are added, because the order of this *does* matter.

14.2.4 Copy Propagation

This is the idea that, after an assignment that copies a value (such as $x := y$), you will be able to use value you are copying from (y), in place of the value you are assigning (x). This is best demonstrated by the following optimisation:

Initial code

```
x := y;
v := x * 17;
w := x + 19;
```

Copy Propagation Optimisation

```
x := y;
v := y * 17;
w := y + 19;
```

This is not necessarily a big optimisation, however, it could turn out that the variable we assign (x), is never used. In which case the assignment is dead code, and can be eliminated.

Initial code

```
x := y;
v := y * 17;
w := y + 19;
```

Copy Propagation Optimisation

```
v := y * 17;
w := y + 19;
```

These copies generally appear as the result of other optimisations (such as common sub expression elimination).

14.2.5 Dead Code Elimination

Dead code elimination is the idea that it is statically possible to determine certain parts of the code are either unreachable, or have no effect (dead code). These are both examples of useless code, that can be removed without changing the meaning of the program. For instance, we can optimise the following code as such:

Initial code

```
debug = false;
if ( debug ) {
    print("debug");
}
```

Constant folded

```
if ( false ) {
    print("debug");
}
```

Dead code eliminated

```
nothing!
```

We can also use dead code elimination to remove assignment to variables that are never used, just like we saw in the copy propagation example above.

It is important to note that there are some edge cases to this. Such as division by zero, which can produce side effects, which potentially has some effect on other code. In the following example, the value of z is never actually used, but the potential for x/y to be a division by zero, means that this assignment cannot be eliminated.

```
fun f (x, y: Integer): Integer =
    let
        var z : Integer := x / y
    in
        return x * y
```

14.2.6 Strength Reduction

Strength reduction is a slightly lower level optimisation, and the idea is that we can replace expensive operations, by cheaper ones. For instance, adding and shift is potentially cheaper than multiplication, and multiplication may be cheaper than exponentiation (given be know the integral power). A loop may have a number of induction variables that remain in lock step, for instance, i and t in the below snippet. We can remove the more computationally expensive multiplication inside the loop, and replace this with a simple subtraction (as all that is happening in this code is the decrease of t).

Original code

```
i := 10;
while ( i > 0 ) do begin
    i    := i - 1;
    t    := 4 * i;
    a[i] := b[t]
end
```

Strength reduction optimisation

```
i := 10;
t := 4 * i;
while ( i > 0 ) do begin
    i    := i - 1;
    t    := t - 4;
    a[i] := b[t]
end
```

14.2.7 Code Motion

The idea that if a piece of code is loop-invariant (evaluates to the same value at each iteration of the loop), it should be moved to outside of a loop. In the following code, $m - 1$ and $n - 1$ are invariants in the outer loop, and $i * 10$ is invariant in the inner loop. This allows us to transform the code to:

Original code

```
for ( i := 0 ; i <= m - 1 ; i++ ) do
    for ( j := 0 ; j <= n - 1 ; j++ ) do
        x := x + a[i * 10 + j]
```

Code Motion optimisation

```
t1 := m - 1;
t2 := n - 1;
for ( i := 0 ; i <= t1 ; i++ ) do
    t3 = i * 10;
    for ( j := 0 ; j <= t2 ; j++ ) do
        x := x + a[t3 + j]
```

Of course, we have to be careful if there are side effects. For instead, consider the code:

```
for ( i := 0 ; i < n ; i++ ) do
    x := x + f(17);
```

The function call $f(17)$ might look like a loop invariant code at first, but it could have side effect, in which case it is wrong to move it out of the loop.

14.2.8 Loop Unrolling

As loop carry certain overhead (evaluation of the condition, and jumps), it can be beneficial to "unroll" loops that are known to be short, for instance, consider:

Original code

```
for (i := 0; i < 5; i++) do
    a[i] := b[4 - i] * 2^i;
```

Unrolled loop

```
a[0] := b[4 - 0] * 2^0;
a[1] := b[4 - 1] * 2^1;
a[2] := b[4 - 2] * 2^2;
a[3] := b[4 - 3] * 2^3;
a[4] := b[4 - 4] * 2^4;
```

We can improve our unrolled loop, with constant folding:

```
a[0] := b[4] * 1;
a[1] := b[3] * 2;
a[2] := b[2] * 4;
a[3] := b[1] * 8;
a[4] := b[0] * 16;
```

However, we should be aware of the caveats of unrolling: code can grow considerably (we must consider the space/time trade off, discussed later), and the cache can be impacted greatly.

14.2.9 In-lining

The final optimisation method is in-lining. The idea of which is to avoid the overhead of function and procedure calls, by instantiating the body with the actual parameters and copy the result to the call site. Also known as procedure integration, for this reason. Inlined procedures/functions should be small, or the size of code might blow up, and we have to be careful with recursion, or the compiler can get stuck in a loop. An example of inlining is shown below:

Original code

```
fun f (x: Integer): Integer =
    begin
        return (x+17) * 123;
    end
end

x := f(a + 3);
y := f(x * 3);
```

In-lining optimisation

```
x := ((a + 3) + 17) * 123;
y := ((x * 3) + 17) * 123;
```

14.3 Interactions between Optimisations

Sometimes, when one optimisation has been applied, we are then able to apply a second optimisation, to the newly optimised code. Here, we will go through a complete example of that, with multiple levels.

Original code

```
const level: Integer = 4;
const debugging: Boolean = true;
func debug (severity: Integer) =
begin
    return debugging && severity > level
end
...
x := 10;
if ( debug(3) ) then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

In-lining optimisation

```
const level: Integer = 4;
const debugging: Boolean = true;
...
x := 10;
if ( debugging && 3 > 4 ) then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

Constant folding

```
const level: Integer = 4;
const debugging: Boolean = true;
...
x := 10;
if ( false ) then begin
    print "Oops! Well, got here.";
    x := x + 1
end;
y := x + 10;
```

Dead code elimination

```
x := 10;
y := x + 10;
```

Constant folding

```
x := 10;
y := 20;
```

If x never used:

```
y := 20;
```

14.4 Time Vs Space

Time and space optimizations are often in conflict; emphasized by the following example. Imagine an array of booleans, in which each boolean is represented by *one* machine *word*. This allows for fast access into our array, but wastes space. We could instead imagine that each boolean was represented by a single bit, but then accessing into this array requires extra operations (shifting and masking), which takes more times, and instruction space. These trades off are further complicated by cache effects (a cache is a method to speed up memory access); such that compact representations will allow us to fit a larger part of the array into the cache.

We are sometimes lucky in that we can consider small operations as *fast* as well. For instance, accessing memory is slow, thus the fewer instructions, pieces of data, and memory accesses, the faster the execution of the program. It is also highly desirable to keep inner loops small, so that the can fit in the first level of the instruction cache, likewise, it is desirable to keep the set of "currently-accessed" memory locations small, so that they fit in the first level of the data cache.

But then again, since memory access is very slow, avoiding memory access could sometimes be worth a few extra instructions. This is because instruction fetching is much faster than data fetching, as it is more predictable. In practice, it is often the case that we make an educated guess, and then verify this guess by benchmarking - the trade off between time and space is a highly complicated issue!

14.5 Compile Time Vs Run Time

Generating highly optimised code can take a long time. Typically, when we are developing code, we only tell the compile to perform a small subset of optimisations, and when the code is compiled for the release version, we allow the full set of optimisations to take place. However, sometimes code must satisfy certain time and space constraints, in which case it may always be required to be compiled with optimizations turned on. Many optimisation techniques have quadratic, or worse, time complexity. As such, programs are usually optimised one part at a tiem, even through global optimisation would yield better results. In some cases, spending days or weeks optimizing can make sense (for instance, when compiling to hardware).