Contents

Chapter 1

Introduction

1.1 Interpreter and compiler

There are two approaches to implement a programing language: compilers and interpreters.

Interpreter is an online approach, i.e. the work done by the interpreter is part of running the program. The program we write and the data on which we wish to run the program are inputted into the interpreter, after which the output is produced by the interpreter.

Compiler is an offline approach, i.e. whatever the compiler does is the preprocessing of the program, and it does not take part in the actual execution of the program on the data. The program is translated into an executable by the compiler, and the data is passed to the executable, which then outputs the result.

1.2 History

In 1954, IBM developed the 704 machine. The customers found that the softwares cost more than the hardware, though the hardware already costs a lot. This inspired a lot of people to try to improve the productiveness of programing, among whom was John Backus. He developed "Speedcoding", which from today's the point of view is an interpreter. Speedcoding made it much faster to develop programs, but the programs developed with it ran much slower and also occupied too much memory. Backus continued to develop the FORTRAN project, which is an abbreviation for FORmula TRANslation. With FORTRAN I, he took a compiler approach: formulae written by programmers were translated into a form that could be understood by the machine. FORTRAN I was a successful project not only in the sense that it was soon adopted by most developers back in the 1950s, but also in the sense that its outline is still preserved by modern compilers. A compiler contains 5 phases:

Lexical analysis Syntactic.

Parsing Syntactic.

Semantic analysis Types, scopes, etc.

Optimization

Code generation Translation into another language.

Chapter 2

Lexical Analysis

An implementation of lexical analyzer must do two things:

1. Recognize substrings corresponding to tokens, i.e. the lexemes.
2. Identify the token class of each lexeme. A token is the <token class, lexeme> pair.

The goal of lexical analysis is to partition the string. It is implemented by reading left-to-right, recognizing one token at a time. "Lookahead" might be required to decide where one token ends and the next token begins.

2.1 Tools: regular expression and finite automata

2.1.1 Lexical specification & regular expression

1. Write a regular expression for the lexemes of each token class.
   - Number = digit+
   - Keyword = 'if' + 'else' + ...
   - Identifier = letter (letter + digit)*
   - Openpar = '('
   - ...
2. Construct R, matching all lexemes of all token classes.
   R = Keyword + Identifier + Number + ... $= R_1 + R_2 + \ldots$
3. Let input be $x_1 x_2 \ldots x_n$. For $1 \leq i \leq n$, check if $x_1 \ldots x_i \in L(R)$.
4. If yes, then we know that $x_1 \ldots x_i \in L(R_j)$ for some j.
5. Remove $x_1 \ldots x_i$ from input and go to 3.

Problems & ambiguities:

1. How much input is used? What if we have

$$x_1 \dots x_i \in L(R)$$
$$x_1 \dots x_j \in L(R)(i \neq j)$$

   at the same time?

   The answer is to choose the **maximal match**: match as long as possible.

2. Which token should be used if more than one token is matched, i.e. what if $x_1 \dots x_i$ simultaneously belongs to $L(R_j)$ and $L(R_k)$?

   A priority rule is set up to prevent such ambiguity. Typically, the rule is to **choose the one listed first**. For example, if should not be recognized as identifier because it belongs to the language of keyword.

3. What if no rule matches, i.e. what if $x_1 \dots x_i \notin L(R)$? This concerns the error handling of the compiler.

   The solution is not to let this happen. We will define one more class, the error class, that matches all strings not in the lexical specification, and put it last in priority.

2.1.2 Finite automata

Regex is the specification language of lexical analysis, and finite automata is an implementation mechanism of regex.

A finite automaton consists of

- An input alphabet Σ
- A set of states S
- A start state n
- A set of accepting states $F \subseteq S$
- A set of transitions state1 $\xrightarrow{\text{input}}$ state2

Transition $s_1 \xrightarrow{a} s_2$ is read "in state s_1 on input a go to state s_2". If the automaton is in accepting state at the end of the input, it will **accept** the string, meaning that the string is in the language of this machine. Otherwise it will **reject** the string. This is either because it terminates in state $s \notin F$, or because the machine gets stuck: there is no transition defined for the current state and input.

The language of a FA is equal to the set of its accepted strings.

It is possible that the machine changes its state without an input, when the transition is called an ε-move. Depending on whether ε-move is allowed, FA can be classified into two types. **Deterministic Finite Automata (DFA)**

make one transition per input and per state, which means for a certain state, an input can lead to at most one possible transition. No ε-move is allowed. **Nondeterministic Finite Automata** can have multiple transitions for one input at a given state, and can have ε-moves.

A DFA takes only one path through the state graph per input string, while an NFA can choose different paths. As long as some of the paths lead to accepting state, the NFA accept the input string.

DFAs and NFAs recognize the same set of languages, which is the set of regular languages. DFAs are faster to execute, while NFAs are in general much (exponentially) smaller.

2.2 Implementation

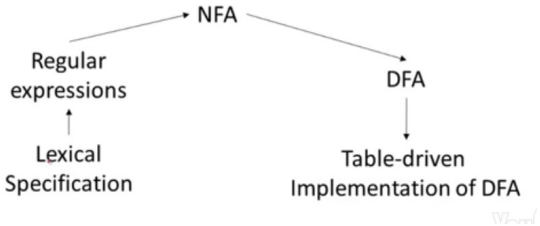


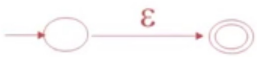
Figure 2.1: Pipeline of lexical analyser

We will follow the pipeline shown in Figure 2.1 to implement the lexical analyser step by step.

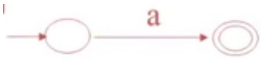
2.2.1 Regular expression into NFAs

Regex can be transformed into NFAs according to the following rules.

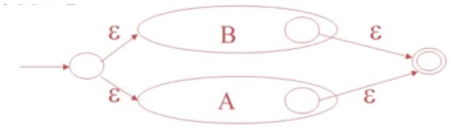
For the simplest ε regex:



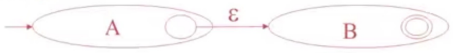
For regex with a single character a:



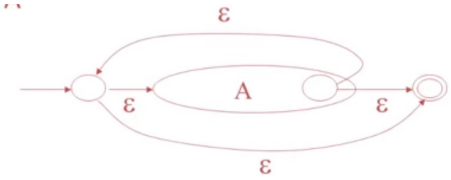
For union $A + B$:



For concatenation AB:



For iteration $A*$:



By combining the rules above, we can transform any regex into NFAs.

2.2.2 NFA to DFA

We introduce the idea of the **ε-closure** of state. The ε-closure of a state s is the set of the states that can be reached from s following only ε moves. In the NFA

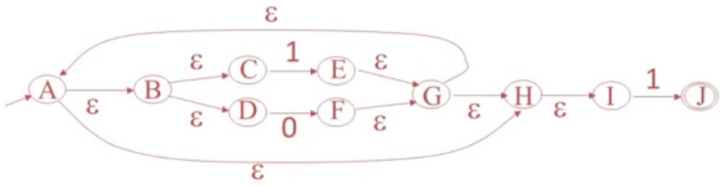


Figure 2.2: Idea of ε-closure

shown in Figure 2.2, the ε-closure of state B is $\{B, C, D\}$, while the ε-closure of state G is $\{A, B, C, D, G, H, I\}$. We also introduce the denotation $a(X)$, which is defined as

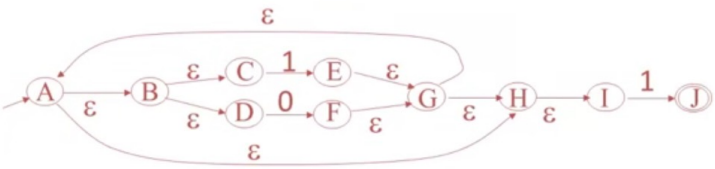
$$a(X) = \{y \in S \mid \exists x \in X \text{ s.t. } x \xrightarrow{a} y\}.$$

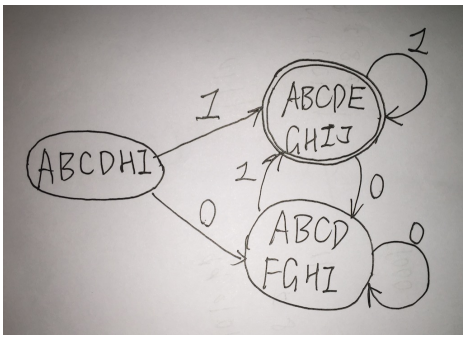
The NFA may be in many different states at any time. Suppose the NFA has N states, and it winds up in a subset of these states S. It is for sure that $|S| \leq N$. There exist totally $2^N - 1$ subsets. $2^N - 1$ is a big number, but still a finite one, which means that there exists a way to convert the NFA into a DFA. We set up a series of rules to convert an NFA to a DFA as shown in Table 2.1. The state of the DFA records the set of possible states that the NFA could

Table 2.1: NFA to DFA

	NFA	DFA
states	S	$P(S) - \emptyset$(subsets of S except $\emptyset$)
start	$s \in S$	ε-$clos(s)$
final	$F \subseteq S$	$\{X \in (P(S) - \emptyset) \mid X \cap F \neq \emptyset\}$
transition		$X \xrightarrow{a} Y$ if $Y = \varepsilon$-$clos(a(X))$

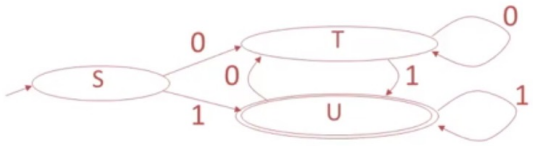
have gotten into with the same input. Below is an example demonstrating this conversion method.





2.2.3 Implementation of FA

A DFA can be implemented by a 2D **states-input symbol** table T. For every transition $s_i \xrightarrow{a} s_k$, there is $T(i, a) = k$. An example is shown below.



	0	1
S	T	U
T	T	U
U	T	U

Related C++ code to implement the state update:

```
int i = 0, state = 0;
while (i < len) {
  state = T[state][input[i++]];
}
```

It is also possible to implement the NFA directly without converting it to a DFA, considering that the conversion could be expensive (N states NFA $\rightarrow 2^N - 1$ states DFA). It is just a tradeoff between speed and space: DFA is faster but less compact, while NFA is slower but concise.

Chapter 3

Parsing

The lexer takes a string of characters as input and transforms it into a string of tokens. A parser takes the string of tokens as input, and produces a parse tree. Sometimes the parse tree is just implicit and is not actually built. Also, some compilers finish the two tasks in one phase.

3.1 Context free grammars

3.1.1 Introduction

Not all strings of tokens are valid programs, which calls for a language for describing valid strings of tokens as well as a method to distinguish valid strings of tokens from invalid ones.

Programming languages have recursive structures: an **expression** is often made up of other expressions. **Context free grammars** are a natural notation for such structure.

A context free grammar consists of

- a set of terminals (T)
- a set of non-terminals (N)
- a start symbol ($S \in N$)
- a set of productions (P)

A production is a relation of symbols:

$$X \to Y_1 Y_2 \dots Y_n$$

in which $X \in N$, and $Y_i \in T \cup N \cup \{\varepsilon\}$.

As an example, the language $\{(^i)^i\}, i = 0, \dots, N$ can be expressed by the CFG with $N = \{S\}, T = \{(,)\}$ and productions $\{S \to (S), S \to \varepsilon\}$.

Productions can be regarded as substitution rules. Terminals are so-called because there is no rule to replace them. Once generated, they are permanent. Terminals ought to be tokens of the programming language. Let G be a CFG with start symbol S. The language $L(G)$ of G is

$$\{a_1 \dots a_n | \forall a_i \in T, S \xrightarrow{*} a_1 \dots a_n\}$$

in which $S \xrightarrow{*} a_1 \dots a_n$ means that S can be rewritten into $a_1 \dots a_n$ in a few steps with the substitution rules defined by the productions.

In the definition of production, no precedence between operators or associativity of operator is assumed. For example, the grammar

$$E \to E + E | E - E | E * E | E/E | (E) | int \tag{3.1}$$

defines the normal $+ - \times \div$ operations of integers, but do not assume the precedence of $\times \div$ over $+-$, or the normal left associativity of these operators.

3.1.2 Derivations

With the help of CFG, we are able to figure out whether a string of tokens belongs to a language. However, we would also like to know the structure of the string, i.e. the parse tree, which calls for the help of derivations.

A sequence of productions is called a derivation. It can be drawn as a tree. The start symbol is the tree's root, and for each production $X \to Y_1 Y_2 \dots Y_n$, $Y_1 \dots Y_n$ is added as children of X. Such a tree drawn to describe an expression is called the **parse tree** of the expression.

A parse tree has terminals at the leaves and non-terminals at the interior nodes. An in-order traversal of the leaves is the original input. The parse tree shows the association of operations, while the input string does not.

According to the order of the non-terminals being replaced, a derivation can be left-most or right-most, or of a random order, which is rarely used. Both the left-most and the right-most derivation have the same parse tree.

3.1.3 Ambiguity

A grammar is **ambiguous** if it has more than one parse tree for some string. Equivalently, there is more than one left-most (right-most) derivation for this string. Ambiguity is a problem we strive to avoid. There are two solutions to the problem: either we directly rewrite the grammar unambiguously, or we enforce precedences on operations (e.g. * over +).

Here we give an example of rewriting grammar. The grammar

$$\mathrm{E} \to \mathrm{E} * \mathrm{E} | \mathrm{E} + \mathrm{E} | (\mathrm{E}) | id$$

is ambiguous. 1 + 2 + 3 can be generated by {1 + 2} + 3 or 1 + {2 + 3}. It can be written as

$$\begin{aligned} \mathrm{E} &\to \mathrm{E}' + \mathrm{E} \\ \mathrm{E}' &\to id * \mathrm{E}' | id | (\mathrm{E}) * \mathrm{E}' | (\mathrm{E}), \end{aligned}$$

which is no longer ambiguous. In the new grammar, E can generate sum(+), while E' can generate product(*). $\{1 + 2\} + 3$ is no longer legal in the new grammar.

Consider the grammar

$$\begin{aligned} \text{E} \rightarrow & \text{ if E then E} \\ | & \text{ if E then E else E} \\ | & \text{ OTHER} \end{aligned}$$

that describes an "if then else" relation in which "else" is optional. It is ambiguous because the expression "if E_1 then if E_2 then E_3 else E_4" has two parse trees because the "else" could correspond to both "then"s. The parse tree of the previous expression could be either "if E_1 then {if E_2 then E_3 else E_4}" or "if E_1 then {if E_2 then E_3} else E_4"We want to rewrite the grammar so that every "else" matches the closest "then". The new grammar is

$$\begin{aligned} \text{E} \rightarrow & \text{ MIF} \\ | & \text{ UIF} \\ \text{MIF} \rightarrow & \text{ if E then MIF else MIF} \\ | & \text{ OTHER} \\ \text{UIF} \rightarrow & \text{ if E then E} \\ | & \text{ if E then MIF else UIF} \end{aligned}$$

in which MIF means all "then"s are matched, while UIF means some "then"s are unmatched. In this new grammar, the second parse tree is no longer legal.

Unfortunately, it is impossible to automatically convert an ambiguous grammar to an unambiguous one. The manual conversion job is tedious, and the unambiguous grammar is often too complex to comprehend quickly. On the contrary, grammar with ambiguity is almost always tidy and more natural. But we need some mechanism to tackle the problem of ambiguity, which is the approach taken by most parsing tools. The most frequently used disambiguation mechanisms are precedence declarations and associativity declarations.

Note that the rewritten grammar no longer generates the same set of expressions from the point of view of semantic meaning. It only ensures that the same set of strings is generated. The grammar (3.1) can be rewritten as

$$\text{E} \rightarrow \text{E} + int | \text{E} - int | \text{E} * int | \text{E}/int | (E) | int$$

to remove the ambiguity. However, 3 * 5 + 2 - 6 / 2 will be parsed as and only as {{{3 * 5} + 2} - 6} / 2, which is against normal precedence and associativity assumptions. In order to make the grammar work correctly in the intended way, simply rewriting the grammar is far from enough. It must be combined with other measures such as precedence and associativity definitions.

3.2 Error handling

Compiler has two major tasks to complete: translating valid programs and detecting invalid ones. Different kinds of errors can be detected by different components of the compiler: lexical errors by lexer, syntactic errors by parser and semantic errors by type checker. There are also errors that are not within the scope of the programming language, and thus remain to be found by the programmer through tests.

An error handler within the compiler is expected to

- report errors accurately and clearly;
- recover from an error quickly;
- not slow down the compilation of valid codes.

There are three different approaches to implement the error handler: panic mode, error productions and automatic local/global correction. The first two are used in current compilers.

3.2.1 Panic mode

Panic mode is the simplest and thus the most popular method. When an error is detected, it discards tokens until one with a clear role is found, and resumes the compilation there. The "clear" token it looks for is usually the terminator of a statement or a function. In Bison (a popular parser generator), a special terminal error is used to describe how much input to skip in case of an error. For example,

$$\mathrm{E} \to \mathrm{E} + \mathrm{E} | (\mathrm{E}) | \text{error int} | \text{error}$$

specifies that an integer is regarded as a token to resume the compilation when an error occurs.

3.2.2 Error productions

Common errors made by programmers can sometimes be predicted. For example, 5 * x is often mistakenly written as 5 x. We can add new productions covering such typos into the grammar so that the compilation can continue, and the programmer receives warnings concerning the errors. This approach has an obvious disadvantage that it complicates the grammar.

3.2.3 Automatic correction

In the early years of programming, compilation is a time-consuming process. Programmers had to spent hours or even a whole day waiting for the compilation result. So compilers were expected to automatically correct probable errors in the program so that the time would not be totally wasted due to small mistakes such as typos, and more errors could be found in a single compilation cycle.

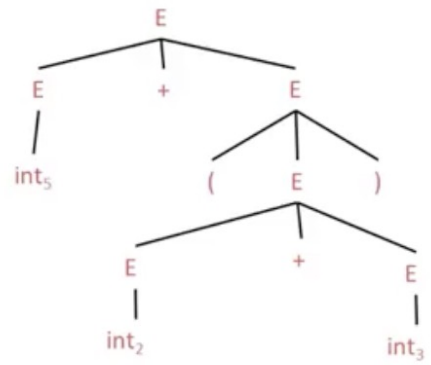


Figure 3.1: Parse tree of 5 + (2 + 3)

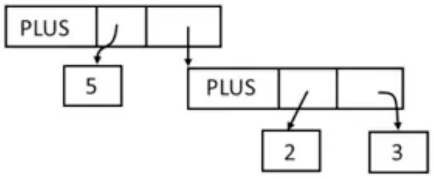


Figure 3.2: AST of 5 + (2 + 3)

This requirement makes it hard to implement a satisfactory compiler, and it also slows down the compilation of correct codes. Nowadays compilation is much more faster, and programmers tend to recompile everytime they fix an error. Thus complex error recovery becomes less compelling than a few decades ago.

3.3 Top down parsing

3.3.1 Abstract syntax tree

A parser traces the derivation of a sequence of tokens, but the rest of the compiler needs a structural representation of the program. **Abstract syntax tree**, or **AST** is the structure we use.

A parse tree traces operations of the parser and captures nesting structure of the token string being parsed. It contains a lot of verbose information, and the AST is an abstracted version that also captures the nesting structure, but is much more compact.

Consider the grammar E $\rightarrow$ int|(E)|E + E and the string 5 + (2 + 3). The parse tree and the related AST are shown in Figure 3.1 and Figure 3.2.

3.3.2 Recursive descent algorithm

Recursive descent parsing is a top-down parsing method, with which the parse tree is built from the top and from left to right. When building a parse tree, we start with the top-level non-terminal E, and try the rules of E in order. When another non-terminal is met, this process is carried out recursively. If a mis-match is found, we need to do the back tracking to find the correct rule to match.

Some rules in the illustration of the recursive descent parsing algorithm: TOKEN will be the type of tokens, and its instances will be INT (for int), OPEN (for '('), CLOSE (for ')'), TIMES (for '*'), etc; the global variable next will point to the next token in the input.

Some boolean functions need to be defined to check matches of token terminals and CFG rules. For a given token terminal tok, for the nth production of S, and for all productions of S, the functions are respectively

```
1  bool term(TOKEN tok) { return *next++ == tok; }
2  bool S_n() { ... }
3  bool S() { ... }
```

As an example, consider the grammar

$$\begin{aligned} &\mathrm{E} \rightarrow \mathrm{T} \mid \mathrm{T} + \mathrm{E} \\ &\mathrm{T} \rightarrow \mathrm{int} \mid \mathrm{int} * \mathrm{T} \mid (\mathrm{E}). \end{aligned} \tag{3.2}$$

We will have

```
1  bool E1() { return T(); }
2  bool E2() { return T() && term(PLUS) && E(); }
3  bool E() {
4    TOKEN *save = next;
5    return (next = save, E1()) || (next = save, E2());
6  }
7  bool T1 = { return term(INT); }
8  bool T2 = { return term(INT) && term(TIMES) && T(); }
9  bool T3 = { return term(OPEN) && E() && term(CLOSE); }
10 bool T() {
11   TOKEN *save = next;
12   return (next = save, T1()) ||
13        (next = save, T2()) ||
14        (next = save, T3());
15 }
```

To start the parser, next needs to be set to the first token, and E() should be invoked.

With the statement *save = next, the current position is saved in save. If the matching for a production succeeds, the matching for the whole non-terminal ends; otherwise next is restored with next = save, and the string continues to be matched with the next production. The use of the C++ comma operator is interesting.

Limitations

The recursive descent parsing algorithm just presented is easy to be implemented by hand, but it is not general. Specifically for this example, consider the matching process for the string int * int. We will go from E() to T() then to $T_1()$, which returns true and leaves us with the * int segment. This segment cannot be matched, forcing us to wrongly reject the whole string.

The problem is that with this algorithm, we have no way of back tracking and trying other productions once one production has been successfully matched. There exist a more general and yet more complex implementation that solves the problem by allowing "full back tracking", which will be covered later.

Left recursion

If a grammar has the form $S\rightarrow^{+}S\alpha$ for some α, then it has a left recursion problem because the recursive descent matching process will end up in an infinite loop.

Left recursion problem can be solved by writing the algorithm. The grammar $S\rightarrow S\alpha|\beta$ that generates the language $\beta\alpha*$ can be rewritten as $S\rightarrow \beta S'$, $S'\rightarrow \alpha S'|\varepsilon$. More generally, the grammar $S\rightarrow S\alpha_1|\ldots|S\alpha_n|\beta_1|\ldots|\beta_m$ that generates the language $(\beta_1|\ldots|\beta_m)(\alpha_1|\ldots|\alpha_n)*$ can be rewritten as $S\rightarrow \beta_1 S'|\ldots|\beta_m S'$, $S'\rightarrow \alpha_1 S'|\ldots|\alpha_n S'|\varepsilon$. The general rule is to recognize the terminal rather than the non-terminal first when matching a combination of the two.

There are algorithms that carries out the elimination of left recursion automatically (in the Dragon book).

3.3.3 Predictive parsing

Predictive parsing is another top-down parsing method. It is able to "predict" which production to use by looking at the next few tokens (lookahead) and no backtracking is needed. Predictive parser accepts so-called **LL(k)** grammars which means **left-to-right left-most-derivation** grammars requiring **k** lookahead tokens. Here we will focus only on the case with k=1. Recall that in recursive descent, many choices of productions could be used at each step, and we use backtracking to undo the bad choices. But with LL(1) grammar, at most one production is available at each step.

Consider the grammar (3.2). It is hard to predict the best production to use because for T, there are two productions that start with an int, and for E, both productions start with T so that it is not clear how to make the predication. The grammar needs to be **left-factored** in order to apply predicative parsing.

The new grammar is

$$\begin{aligned} &E \rightarrow TX \\ &X \rightarrow +E \mid \epsilon \\ &T \rightarrow \text{int } Y \mid (E) \\ &Y \rightarrow * T \mid \epsilon \end{aligned} \tag{3.3}$$

A **LL(1) parsing table** can be generated according to this grammar as shown in Table 3.1. The method to generate it will be covered later. $ is the end

Table 3.1: Parsing table of grammar (3.3)

	int	*	+	(	)	$
E	TX			TX		
X			+E		ϵ	ϵ
T	int Y			(E)		
Y		* T	ϵ		ϵ	ϵ

of the input. Rows of the table represent the current leftmost non-terminal, while columns of the table represent the next input token. The entry is the rhs of the production to be used according to the combination of the leftmost non-terminal and the next input token. As an example, we have [E, int] = TX, which means that when the current non-terminal is E and the next input token is int, we should use the production E $\rightarrow$ TX. Also, [Y, +] = ϵ means that when the current non-terminal is Y and the next input token is +, the production Y $\rightarrow$ ϵ should be used, i.e. Y should be got rid of. Finally, empty entry means that there is an error.

In order to carry out predictive parsing, we need to construct a stack that records the frontier of the parse tree. It contains non-terminals to be expanded as well as terminals to be matched. The top of the stack is always the leftmost pending terminal or non-terminal. The algorithm can be illustrated by Algorithm (3.1). If the stack is empty when we reach the end of the input, the string

Algorithm 3.1 Predictive parsing algorithm

```
Initialize stack = <S,$> and next
repeat
    switch stack
        case <X,rest>:
            if T[X, *next++] == Y_1 ... Y_n then
                stack←< Y_1 ... Y_n rest>
            else error()
        case <t,rest>:
            if t == *next++ then
                stack←<rest>
            else error()
until stack == <  >
```

is accepted. If any error state is reached, the string is rejected. A step-by-step predictive parsing process of the string int * int with the grammar (3.3) is shown in Table 3.2.

Now let's discuss the construction of LL1 parsing tables. For non-terminal A, production A$\rightarrow$ α and terminal t, we will have T[A,t] = α in and only in two

Table 3.2: Predictive parsing of int * int

Stack	Input	Action
E$	int * int$	TX
TX$	int * int$	int Y
intYX$	int * int$	terminal
YX$	* int$	* T
*TX$	* int$	terminal
TX$	int$	int Y
intYX$	int$	terminal
YX$	$	ϵ
X$	$	ϵ
$	$	accept

cases:

1. When $\alpha \xrightarrow{*} t\beta$, i.e. α can derive a t at the beginning. We say $t \in \text{First}(\alpha)$.
2. Otherwise when $\alpha \xrightarrow{*} \epsilon$ and $S \xrightarrow{*} \beta A t \delta$. We say $t \in \text{Follow}(\alpha)$.

First set

The formal definition of First(X) is

$$\text{First}(X) = \{t | X \xrightarrow{*} t\alpha\} \cup \{\epsilon | X \xrightarrow{*} \epsilon\}.$$

The algorithm to calculate First(X) is

1. For all terminal t, First(t) = {t}.
2. $\epsilon \in$ First(X) if $X \xrightarrow{*} \epsilon$, or if $X \xrightarrow{*} A_1 \dots A_n$ and $\epsilon \in \bigcap_j \text{First}(A_j)$.
3. $\text{First}(\alpha) \subseteq \text{First}(X)$ if $X \xrightarrow{*} A_1 \dots A_n \alpha$ and $\epsilon \in \bigcap_j \text{First}(A_j)$.

For grammar (3.3), we have

- For terminals, First(t) = t, t = +, *, (,), int.
- First(T) = {int, (}
- First(E) = {int, (}
- First(X) = {+, ϵ}
- First(Y) = {*, ϵ}

Follow set

The formal definition of Follow(X) is

$$\text{Follow}(X) = \{t | S \xrightarrow{*} \beta X t \delta\}.$$

The algorithm to calculate Follow(X) is

1. \$ $\in$ Follow(S)
2. For production $A \to \alpha X \beta$, First(β) $- \{\epsilon\} \subseteq$ Follow(X).
3. For production $A \to \alpha X \beta$ in which $\epsilon \in$ First(β), Follow(A) $\subseteq$ Follow(X).

For grammar (3.3), we have

- Follow(E) = {\$,)}
- Follow(X) = {\$,)}
- Follow(T) = {+,\$,)}
- Follow(Y) = {+,\$,)}
- Follow('(') = {int, (}
- Follow(')') = {+,\$,)}
- Follow(int) = {*,+,\$,)}
- Follow(+) = {int, (}
- Follow(*) = {int, (}

Parsing table

To build a parsing table, for each production $A \to \alpha$, we need to do:

1. For each terminal $t \in$ First(α), T[A, t] = α.
2. If $\epsilon \in$ First(α), for each terminal $t \in$ Follow(A), T[A, t] = α.
3. If $\epsilon \in$ First(α) and \$ $\in$ Follow(A), T[A, \$] = α.

If any entry of the parsing table is multiple defined, then the grammar is nor LL(1). In particular, if the grammar is

- not left factored
- left recursive
- ambiguous
- ...

then it is not LL(1). Actually most programming language CFGs are not LL(1). LL(1) grammar is too week to describe all the features required in these languages.

3.4 Bottom-up parsing

Bottom-up parsing is more general than deterministic top-down parsing. Actually it is as efficient, and builds on all the ideas we have discussed in top-down parsing. Bottom-up parsing is the preferable method in reality. Bottom-up parsing does not require left-factored grammar, thus we can revert to the natural grammar (3.2) in the following discussion. Nonetheless, bottom-up parsers do not deal with ambiguous grammars, thus we still have to enforce precedence and associativity rules.

Bottom-up parsing reduces a string to the start symbol by inverting productions. As an example, consider the string int * int + int. It can be reduced to the start symbol E via the following path:

int * int + int	T→int
int * T + int	T→int * T
T + int	T→ int
T + T	E→T
T + E	E→T + E
E	

Obviously, the left column is the rightmost derivation of the string written in reverse. This is actually always true for bottom up parsers. **Bottom-up parser traces a rightmost derivation in reverse.** It builds the parse tree from its leaves up towards the root, by combining smaller parse trees into larger ones.

3.4.1 Shift reduce parsing

Suppose $\alpha\beta\omega$ is a step of a bottom-up parse, and the next reduction to apply is $X \to \beta$. Since $\alpha X\omega \to \alpha\beta\omega$ is a step in a rightmost derivation, we can be sure that ω is a string of terminals. This inspires us of the idea of shift-reduce parsing. The string is split into two substrings, the right one unexamined by the parser, and the left one containing terminals and non-terminals. The dividing point is marked by a | sign. Two actions are needed to carry out bottom-up parsing: **Shift** and **Reduce**. Shift means moving | one place to the right, i.e. shifting one terminal to the left substring. Reduce means applying an inverse production at the right end of the left substring.

The left substring in shift-reduce string can be implemented by a stack, with the top of the stack being the | sign. Each shift action pushes a terminal on the stack. Each reduce action pops symbols (rhs of the production rule, terminals and nonterminals) out of the stack, and pushes a nonterminal on the stack.

In a given state, more than one action might lead to a valid parse. If it is legal to shift or reduce, there is a **shift/reduce** conflict. If there are two legal reduces, then there is a **reduce/reduce** conflict. Reduce/reduce conflicts are always bad, indicating some serious problem of the grammar. Shift/reduce conflicts can usually be removed by precedence definitions.

3.4.2 Handle & viable prefixes

Reducing whenever we meet a rhs of production will probably cause incorrect results. For example, we cannot reduce int * int to T * int after the first shift according to the grammar (3.2). We should only reduce when its result can still be reduced to the start symbol. A **handle** is a reduction that also allows further reductions back to the start symbol. For a rightmost derivation $S \overset{*}{\rightarrow} \alpha X\omega \rightarrow \alpha\beta\omega$, we say $\alpha\beta$ is a handle of $\alpha\beta\omega$. **In shift-reduce parsing, handles appear only at the top of the stack (never inside).** Handles are never to the left of the rightmost non-terminal. Thus shift-reduce moves are sufficient, and the | sign never has to move left. Bottom-up parsing algorithms are based on recognizing handles.

Recognizing handles

Unfortunately there exists no known efficient algorithm to recognize handles. Nonetheless, there are good heuristics for guessing handles, and for some fairly large classes of CFGs, these heuristics always identify the handles correctly. Figure 3.3 illustrates the relationship of different kinds of CFGs. Most of the

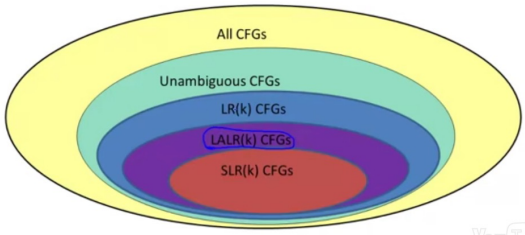


Figure 3.3: Relationship of different CFGs

time we will talk about SLR(k) CFGs.

It is not obvious how to detect handles. We should always keep in mind that at each step, the parser sees only the stack and not the entire input, which is the starting point of the whole discussion. First we define a **viable prefix**. α is a viable prefix if there is an ω such that $\alpha|\omega$ is a state of a shift-reduce parser. Here, α is the visible stack, while ω is the rest of the input. A viable prefix is a string that does not extend past the right end of the handle. It is called "viable" because it is a prefix of the handle. As long as the parser has viable prefixes on the stack, it means that no parsing error has been detected.

For any grammar, the set of viable prefixes is a regular language. As a result, the set of viable prefixes can be recognized by a finite automaton. We will shown how to compute the automata that accept viable prefixes.

First we introduce the idea of an **item**. An item is a production with a $\cdot$[1] somewhere in the rhs. For example, the production T→(E) produces 4 items:

[1] In the lectures . is used. I use $\cdot$(LaTeX\cdot) to avoid confusion with period.

T→ ·(E), T→(·E), T→(E·) and T→(E)·.For ϵ-productions, the only item for X→ ϵ is X→ ·. Items are usually called the LR(0) items. They provide a description of intermediate steps of shirt-reduce parsing. Consider the input (int) for grammar (3.2). (E|) is a state of a shift-reduce parse for it. (E is a prefix of the rhs of the production T→(E), and it is to be reduced if a) is recognized after the next shift. Item T→(E·) describes such situation: we have seen (E and hope to see).

The stack contains actually many prefixes of rhses of productions:

$$\text{Prefix}_1\text{Prefix}_2 \dots \text{Prefix}_{n-1}\text{Prefix}_n$$

Let Prefix_i be a prefix of rhs of $X_i \to \alpha_i$. Prefix_i will eventually reduce to X_i. In order that the parsing can continue, the missing part of α_{i-1} must start with X_i, i.e. there must exist a production $X_{i-1} \to \text{Prefix}_{i-1}X_i\beta$ for some β. Recursively $\text{Prefix}_{k+1} \dots \text{Prefix}_n$ eventually reduces to the missing part of α_k.

Consider (int * int) for grammar (3.2). (int * | int) is a state of a shift-reduce parse. We have the stack of items:

$$\begin{aligned} &\text{T} \to (\cdot\, \text{E}) \\ &\text{E} \to \cdot\, \text{T} \\ &\text{T} \to \text{int}^* \cdot \text{T} \end{aligned}$$

Recognizing viable prefixes

As concluded previously, to recognize viable prefixes, we must recognize a sequence of partial rhses of productions, where each partical rhs can eventually reduce to part of the missing suffix of its predecessor. We will build an NFA that takes the stack as an input and decides whether to accept or reject it.

1. Add a dummy production S'→S to the grammar G.
2. The NFA states are items of G, including the dummy production just added.
3. For item E→ $\alpha \cdot X\beta$, add transition

   $$\text{E} \to \alpha \cdot X\beta \xrightarrow{X} \text{E} \to \alpha X \cdot \beta$$

   Here X is either a terminal or a non-terminal. This rule extends a prefix or an rhs.
4. For item E→ $\alpha \cdot X\beta$ and production $X \to \gamma$, add transition

   $$\text{E} \to \alpha \cdot X\beta \xrightarrow{\epsilon} X \to \cdot\gamma$$

   Here X can only be non-terminals. This rule ends the current prefix and starts a new one.
5. Every state is an accepting state.

6. Start state is S'→S.

Consider grammar (3.2). After adding the dummy production, it becomes

$$\begin{aligned} &\text{S'} \rightarrow \text{E} \\ &\text{E} \rightarrow \text{T} \mid \text{T} + \text{E} \\ &\text{T} \rightarrow \text{int} \mid \text{int} * \text{T} \mid (\text{E}). \end{aligned}$$

By applying the algorithm above, we wind up with the NFA shown in Figure 3.4.

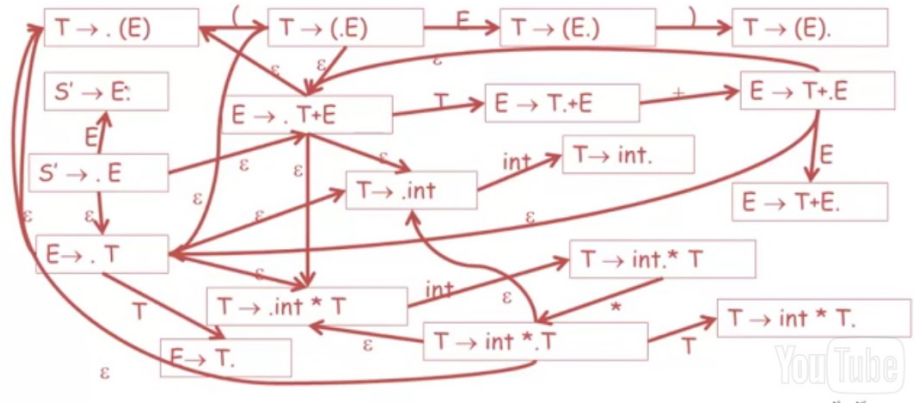


Figure 3.4: NFA of grammar (3.2)

The NFA gained using the algorithm above can be converted into a DFA. The states of the DFA are sets of items. The state of the DFA are called "canonical collections of items (or LR(0) items)"[2].

The item X→ $\beta \cdot \gamma$ is said to be **valid** for a viable prefix $\alpha\beta$ if

$$S' \xrightarrow{*} \alpha X \omega \rightarrow \alpha\beta\gamma\omega$$

by right-most derivation. After parsing $\alpha\beta$, the valid items are possible tops of the stack of items. An equivalent explanation is that for a given viable prefix α, the valid items are exactly the items in the final state of the DFA after it reads that prefix.

3.4.3 SLR parsing

In this section we will present the **SLR parsing** algorithm. SLR means "simple left-to-right rightmost".

First we will introduce a weak bottom-up parsing algorithm called LR(0) parsing. Suppose that the stack contains α and the next input is token t. The DFA on input α terminates in state s. Then the rules for shift-reduce actions are:

[2]Dragon book provides another way to define LR(0) items.

- Reduce by $X \to \beta$ if $X \to \beta\cdot \in s$
- Shift if $X \to \beta\cdot t\omega \in s$. This is equivalent to the fact that s has a transition labeled t.

LR(0) parsing has a reduce/reduce conflict if any state has two reduce items $X_1 \to \beta_1\cdot$ and $X_2 \to \beta_2\cdot$. It has a shift/reduce conflict if any state has a reduce item $X_1 \to \beta_1\cdot$ and a shift item $X_2 \to \beta_2 \cdot t\delta$.

SLR improves on LR(0) shift/reduce heuristics. As a result, fewer shift/reduce conflicts will happen. The only adjustment is on the reduce rule, which is changed to

- Reduce by $X \to \beta$ if $X \to \beta\cdot \in s$ and t$\in$Follow(X)

If any conflict still exists, then the grammar is not an SLR grammar. The handles of SLR grammars can be exactly detected by the heuristics. A lot of grammars are not SLR grammars. We can define define rules to resolve some conflicts for such grammars, which sometimes have the effect of precedence declaration.

Now we present the complete SLR parsing algorithm.

Algorithm 3.2 SLR parsing algorithm

Let M be the DFA for viable prefixes of G. and let $|x_1 \dots x_n\$$ be the initial configuration.
repeat
  Let $\alpha|\omega$ be the current configuration and run M on the current stack α.
  if M rejects α **then**
    report parsing error
  else
    M accepts α with items I, let a be the next input
    if $X \to \beta \cdot a\gamma \in I$ **then** shift
    else if $X \to \beta\cdot \in I$ **and** $a \in$Follow(X) **then** reduce
    else report parsing error
until configuration is $S|\$$

See video 8-6 for an exhaustive example of applying the algorithm.

Algorithm (3.2) can be improved because rerunning the viable prefixes automaton is wasteful: reduce or shift will only change a few symbols on the top of the stack, while the rest of the stack stays the same, and the work on them is just a repeat. If the repeat can be avoided, the algorithm can run much more quickly.

The state of the automaton on each prefix can be remembered. It will be more convenient to make the stack store <Symbol, DFA state> pairs rather than just symbols. We will have a stack

$$< \text{sym}_1, \text{state}_1 > \cdots < \text{sym}_n, \text{state}_n >,$$

in which state_n is the final state of the DFA on $\text{sym}_1 \ldots \text{sym}_n$. The bottom of the stack is now <any, start>, in which any is any dummy state, while start is the start state of the DFA. The algorithm has 4 possible moves:

- Shift x. <a,x> will be pushed on the stack, in which a is the current input, while x is a DFA state.
- Reduce X$\rightarrow \alpha$. As before, pop the rhs elements off the stack and push the lhs on.
- Accept.
- Error.

We will define two parsing tables.

- Define `goto[i,A] = j` if $\text{state}_i \xrightarrow{A} \text{state}_j$. `goto` is just the transition function of the DFA.
- Define `action[i,a]` for each DFA state s_i and the next input terminal a.
  - If s_i contains item $X \rightarrow \alpha \cdot a\beta$ and `goto[i,a] = j` then `action[i,a] = shift j`.
  - If s_i contains item $X \rightarrow \alpha \cdot$ and a$\in$Follow(X) and X$\neq$S' then `action[i,a] = reduce` $X \rightarrow \alpha \cdot$.
  - If s_i contains item S'$\rightarrow$S, then `action[i,$] = accept`.
  - Otherwise `action[i,a] = error`.

The improved algorithm is shown in Algorithm (3.3).

In practice, SLR parsing is too simple and sometimes naive. LR(1), or LALR(1), which is based on LR(1) and includes some optimisations, parsing is much more widely used. The main difference between LR(1) and SLR is that LR(1) builds lookahead into the items. An LR(1) items looks like [T$\rightarrow$ $\cdot$int * T, $], which means "after seeing int * T, reduce if lookahead is $". This mechanism is more accurate that using just the follow set to decide whether to reduce.

Algorithm 3.3 Improved SLR parsing algorithm

```
Let I = w$ be the initial input
Let j = 0.
Let DFA state 1 have item S'→S.
Let stack = <dummy, 1>
repeat
    switch action[top_state(stack), I[j]]
        case Shift k:
            Push <I[j++], k> on stack
        case reduce X → A:
            Pop |A| pairs
            Push <X, goto[top_state(stack), X]> on stack
        case accept:
            Halt normally
        case error:
            Halt with an error
until configuration is S|$
```

Chapter 4

Semantic Analysis

The lexer detects illegal tokens inside the input, while the parser detects ill-formed parse trees inside the input. Semantic analysis is the last "front end" phase of the compiler to catch errors. It is necessary because there are errors that cannot be caught by the parser and lexer, and some language constructs are not context free.

As a typical statically type checked object oriented language, Cool language requires its semantic analyser doing the following checks:

- All identifiers are declared
- Type checking (major function)
- Inheritance relationships
- Classes are defined only once
- Methods in a class are defined only once
- Reserved identifiers are not misused

This is not an exhaustive list.

4.1 Scope

There can be more than one definition of an identifier before it is used. In order to match the correct declaration of an identifier with its uses, we need to understand the conception of scope.

The **scope** of an identifier is the portion of a program in which that identifier is accessible. The same identifier may refer to different things in different parts of the same program. In such case, different scopes of the same identifier should not overlap. Programming languages can have either **static scope** or **dynamic scope**. Today most languages, including Cool, have static scope, which means

scopes of identifiers depend only on the program text, not the runtime behaviors. There are languages that are dynamically scoped, like SNOBOL and ancient Lisp, for which scopes depend on the execution process of the program. Generally speaking, static scoped language follows **the most closely nested rule**, meaning that the variable binds to the definition that is the most closely enclosing it. Dynamically scoped language follows **the most recent binding rule**, meaning that the variable binds to the most recent definition during the execution.

In Cool, identifier bindings are introduced by a lot of mechanisms:

- class definitions (class names)
- method definitions (method names)
- let expressions (object ids)
- formal parameters (object ids)
- attribute definitions (object ids)
- case expressions(object ids)

Not all identifiers in Cool follow the most closely nested rule. For example, class definitions in Cool cannot be nested, and they are globally visible throughout the program, which means that a class can be used before it is defined. Also, attribute names are globally visible within the class in which they are defined. What's more, method names have some complex rules, such as they can be defined in a parent class, and they can be overridden.

4.2 Symbol tables

Much of semantic analysis can be expressed as a recursive descent of an AST. In each step we do the following 3 things:

- Before: Begin processing an AST node n (preprocessing)
- Recurse: Process the children of n
- After: Finish processing node n (post-processing)

When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined. If we divide the processing of the expression $\mathsf{let}\ \mathsf{x}{:}\mathsf{T} \leftarrow e_0\ \mathsf{in}\ e_1$ into the 3 phases listed above, the preprocessing phase will add definition of x to the current definitions and override any previous definition of x, while the post-processing phase will remove definition of x and restore the old definition of x. A **symbol table** is the data structure used to track the current bindings of identifiers.

To implement a simple symbol table, we can use just a stack. It contains three operations:

- add_symbol(x): push symbol x and its associated info on the stack.
- find_symbol(x): search the stack from the top. Returns the first x found or NULL.
- remove_symbol(): pop the stack.

This simple implementation works for let expression because in let expressions, declarations are perfectly nested, and symbols are added to the symbol table one at a time. In other cases, the functionality of this implementation is not sufficient, e.g. in the definition of a method in which more than one symbols can be introduced each time. We need an implementation that covers the following operations:

- enter_scope(): start a new nested scope
- find_symbol(x): find current x (or NULL)
- add_symbol(x): add a symbol x to the table
- check_scope(x): true if x is defined in the current scope
- exit_scope(): exit current scope

Class names should be specially considered here because classes can be used before they are defined. Thus they cannot be checked using a symbol table, neither in one single pass. The solution is to complete two passes: gather all class names in the first one, and do the checking in the second one. In general, semantic analysis requires multiple passes. In the implementation of semantic analysis, a few simple passes is superior to one complex pass.

4.3 Type checking

4.3.1 Types

"What is a type" is a question worthy of asking because type is a notion varying from language to language. The consensus is that a type is a set of values and a set of operations on these values. In OO languages, classes are one instantiation of the modern notion of type, but types do not need to be associated with classes.

The goal of type checking is to ensure that operations are used only with the correct types. It is nonsensical to add a function pointer to an integer in C, but at assembly language level they share the same implementation. Type checking is intended to avoid such errors.

There are 3 kinds of languages:

Statically typed Almost or all type checking is done as part of compilation. e.g. C, java, cool.

Dynamically typed Almost all type checking is done at run time. e.g. Python, Lisp, Perl.

Untyped No type checking. e.g. machine code.

There have always been debates on the merits of static typing v.s. dynamic typing. Static typing proponents assert that static checking catches many errors at compile time, and it avoids overhead of runtime type checks. Dynamic typing proponents argue that static type systems are too restrictive, and it causes difficulty when it comes to rapid prototyping. In the end, we end up with compromises on both sides: static typed languages often provide an "escape" mechanism, e.g. casting in C-like languages; dynamic typed languages are often retrofitted for optimisation and debugging with static typing.

Types in Cool include class names and SELF_TYPE. User is supposed to declare types for identifiers, and the compiler will do the rest of the job: a type will be inferred for every expression.

4.3.2 Logical inference rules

We have seen two formal notations as the specification of parts of a compiler: regular expressions and context free grammars. Logical inference rules are the appropriate formalism for type checking.

Inference rules have the form

If Hypothesis is true, the Conclusion is true.

In the specific case of type checking rules, they often have the form

If E_1 and E_2 have certain types, then E_3 has a certain type.

In order to simplify the notation, we use $\wedge$ to denote "and", $\Rightarrow$ to denote "if-then", and x:T to denote "x has type T". Thus, the rule "if e_1 has type Int, e_2 has type Int, then $e_1 + e_2$ has type Int" is denoted as

$$(\texttt{e}_1 : \texttt{Int} \wedge \texttt{e}_2 : \texttt{Int}) \Rightarrow \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}$$

By convention, inference rules are written in the form

$$\frac{\vdash \text{Hypothesis}_1 \cdots \vdash \text{Hypothesis}_n}{\vdash \text{Conclusion}}$$

in which $\vdash$ is read "it is provable that". Here we give some rules as examples.

$$\frac{\vdash \texttt{i is an integer literal}}{\vdash \texttt{i} : \texttt{Int}}$$

$$\frac{\vdash \texttt{e}_1 : \texttt{Int} \vdash \texttt{e}_2 : \texttt{Int}}{\vdash \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}}$$

$$\frac{\dfrac{\texttt{1 is an int literal}}{\vdash \texttt{1:Int}} \quad \dfrac{\texttt{2 is an int literal}}{\vdash \texttt{2:Int}}}{\vdash \texttt{1} + \texttt{2} : \texttt{Int}}$$

$$\frac{}{\vdash \texttt{false} : \texttt{Bool}}$$

$$\frac{}{\vdash \texttt{new T} : \texttt{T}}$$

$$\frac{\vdash \texttt{e} : \texttt{Bool}}{\vdash \texttt{!e} : \texttt{Bool}}$$

$$\frac{\vdash \texttt{e}_1 : \texttt{Bool} \quad \vdash \texttt{e}_2 : \texttt{T}}{\vdash \texttt{while e}_1 \texttt{ loop e}_2 \texttt{ pool} : \texttt{Object}}$$

A type system is sound if whenever ⊢e:T, e evaluates to a value of type T. We only want sound rules, but some sound rules are better than others. For example, $\frac{\vdash \texttt{i:Int}}{\vdash \texttt{i:Object}}$ is sound but not helpful at all.

Type check proves facts in the form of e:T. The proof is on the structure of the AST. It actually has the shape of the AST, because one type rule is used for each AST node. In the rule used for an AST node e, Hypotheses are the proofs of the types of e's subexpressions, while the conclusion is the type of e. Types are computed in a bottom-up pass over the AST.

4.3.3 Type environment

For a variable, the local structural rule does not carry enough information to give it a type.

$$\frac{\text{x is a variable}}{\vdash \texttt{x} : ?}$$

More information should be put into the rules in such case. A **type environment** gives types to **free** variables. A variable is free if it is not defined within the expression. A type environment is a function from object identifiers to types. It is implemented by the symbol table.

Let O be a function from ObjectIdentifiers to Types, the sentence O⊢e : T is read: under the assumption that free variables in expression e have the type given by O, it is provable that e has type T. The type environment should be added to the earlier rules. For example now we have

$$\frac{\vdash \text{i is an integer literal}}{\texttt{O} \vdash \texttt{i} : \texttt{Int}}$$

$$\frac{\texttt{O} \vdash \texttt{e}_1 : \texttt{Int} \quad \texttt{O} \vdash \texttt{e}_2 : \texttt{Int}}{\texttt{O} \vdash \texttt{e}_1 + \texttt{e}_2 : \texttt{Int}}$$

And we can now write some new rules:

$$\frac{\texttt{O(x)} = \texttt{T}}{\texttt{O} \vdash \texttt{x} : \texttt{T}}$$

We use O[T/x] to denote the function that returns T for x, and O(y) for whatever y≠x. We can now define the rule for let expression:

$$\texttt{O} \vdash \texttt{e}_0 : \texttt{T}$$

$$\frac{\mathtt{O[T/x]} \vdash \mathtt{e_1 : T_1}}{\mathtt{O} \vdash \mathtt{let\ x : T} \leftarrow \mathtt{e_0\ in\ e_1 : T_1}} \tag{4.1}$$

The type environment is passed down the AST from the root to the leaves, while types are computed up the AST from the leaves to the root.

4.3.4 Subtyping

The rule (4.1) is not satisfactory in practice because It is not necessary that x:T. x can actually be of any subtype of T. In order to allow the use of subtypes, we introduce the $\leq$ relationship between classes. Its formal definition is

- $X \leq X$
- $X \leq Y$ if X inherits from Y
- $X \leq Z$ if $X \leq Y$ and $Y \leq Z$

With $\leq$ relationship added, rule (4.1) can be written as

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e_0 : T_0} \\ \mathtt{O[T/x]} \vdash \mathtt{e_1 : T_1} \\ \mathtt{T_0} \leq \mathtt{T}\end{array}}{\mathtt{O} \vdash \mathtt{let\ x : T} \leftarrow \mathtt{e_0\ in\ e_1 : T_1}}$$

Similarly, the rule of assignment can be written as

$$\frac{\begin{array}{c}\mathtt{O(x) = T_0} \\ \mathtt{O} \vdash \mathtt{e_1 : T_1} \\ \mathtt{T_1} \leq \mathtt{T_0}\end{array}}{\mathtt{O} \vdash \mathtt{x} \leftarrow \mathtt{e_1 : T_1}}$$

Attribute initialization inside a class also uses subtyping. $\mathrm{O}_C(\mathrm{x}) = \mathrm{T}$ means for all attributes x of class C we have x : T.

$$\frac{\begin{array}{c}\mathtt{O_C(x) = T_0} \\ \mathtt{O_C} \vdash \mathtt{e_1 : T_1} \\ \mathtt{T_1} \leq \mathtt{T_0}\end{array}}{\mathtt{O_C} \vdash \mathtt{x : T_0} \leftarrow \mathtt{e_1 : T_0}}$$

Consider the case of the `if` expression. The type of if e_0 then e_1 else e_2 fi can be either the type of e_1 or that of e_2, depending on whether the else clause or the then clause is executed at runtime. In this case, the best we can do is to use the smallest super type larger than both the types of e_1 and e_2, i.e. **their least upper bound**, which is denoted by $\mathsf{Z} = \mathsf{lub(X, Y)}$. Its formal definition is

- $X \leq Z \wedge Y \leq Z$

- if $X \leq Z' \wedge Y \leq Z', Z \leq Z'$

In Cool, the least upper bound of two types is their least common ancestor in the inheritance tree. Equipped with the definition of lub(X,Y), we can write the rule of the if expression:

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e_0} : \mathtt{Bool} \\ \mathtt{O} \vdash \mathtt{e_1} : \mathtt{T_1} \\ \mathtt{O} \vdash \mathtt{e_2} : \mathtt{T_2}\end{array}}{\mathtt{O} \vdash \mathtt{if}\ \mathtt{e_0}\ \mathtt{then}\ \mathtt{e_1}\ \mathtt{else}\ \mathtt{e_2}\ \mathtt{fi} : \mathtt{lub}(\mathtt{T_1}, \mathtt{T_2})}$$

The rule of case expression takes a similar but more complex form:

$$\frac{\begin{array}{c}\mathtt{O} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{O}[\mathtt{T_1}/\mathtt{x_1}] \vdash \mathtt{e_1} : \mathtt{T'_1} \\ \ldots \\ \mathtt{O}[\mathtt{T_n}/\mathtt{x_n}] \vdash \mathtt{e_n} : \mathtt{T'_n}\end{array}}{\mathtt{O} \vdash \mathtt{case}\ \mathtt{e_0}\ \mathtt{of}\ \mathtt{x_1} : \mathtt{T_1} \rightarrow \mathtt{e_1};\ \ldots;\ \mathtt{x_n} : \mathtt{T_n} \rightarrow \mathtt{e_n} : \mathtt{lub}(\mathtt{T'_1}, \ldots, \mathtt{T'_n})}$$

4.3.5 Methods type environment

In order to check the type of a method call, we need a mechanism similar to type environment O for variables. In Cool, method type rules are put in namespace M different from O, which means that a method and an object can share the same name. A rule

$$\mathtt{M}(\mathtt{C}, \mathtt{f}) = (\mathtt{T_1}, \ldots, \mathtt{T_n}, \mathtt{T_{n+1}})$$

means that in class C, there is a method f with signature $f(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1}$. Now we can write the rule of normal dispatch:

$$\frac{\begin{array}{c}\mathtt{O}, \mathtt{M} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e_1} : \mathtt{T_1} \\ \ldots \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e_n} : \mathtt{T_n} \\ \mathtt{M}(\mathtt{T_0}, \mathtt{f}) = (\mathtt{T'_1}, \ldots, \mathtt{T'_n}, \mathtt{T_{n+1}}) \\ \mathtt{T_i} \leq \mathtt{T'_i}\ \mathtt{for}\ 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O}, \mathtt{M} \vdash \mathtt{e_0.f}(\mathtt{e_1}, \ldots, \mathtt{e_n}) : \mathtt{T_{n+1}}}$$

Similarly, the rule of static dispatch is

$$\begin{array}{c}\mathtt{O}, \mathtt{M} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e_1} : \mathtt{T_1} \\ \ldots \\ \mathtt{O}, \mathtt{M} \vdash \mathtt{e_n} : \mathtt{T_n} \\ \mathtt{M}(\mathtt{T}, \mathtt{f}) = (\mathtt{T'_1}, \ldots, \mathtt{T'_n}, \mathtt{T_{n+1}})\end{array}$$

$$\frac{\begin{array}{c}\mathtt{T}_0 \leq \mathtt{T} \\ \mathtt{T}_i \leq \mathtt{T}'_i \quad \mathtt{for} \quad 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O}, \mathtt{M} \vdash \mathtt{e}_0@\mathtt{T}.\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{T}_{n+1}}$$

For some cases involving `SELF_TYPE`, we need to know the class in which an expression appears. Thus the full type environment of Cool contains 3 parts:

- A mapping O giving types to object identifiers.
- A mapping M giving types to methods.
- The current class C.

The whole environment must be added to all rules, although in most cases M and C are passed down but not actually used. For example, the rule for add of int is now

$$\frac{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 : \mathtt{Int} \quad \mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_2 : \mathtt{Int}}{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 + \mathtt{e}_2 : \mathtt{Int}} \tag{4.2}$$

4.3.6 Implementation

Cool type checking can be implemented in a single traversal over the AST. The type environment is passed down the tree from parent to child, while types are passed up the tree from child to parent.

The implementation of a rule is somewhat self-explaining. The addition rule for int (4.2) could be implemented with the following persudo-code.

```
1 TypeCheck(Environment, e1 + e2) {
2   T1 = TypeCheck(Environment, e1);
3   T2 = TypeCheck(Environment, e2);
4   Check T1 == T2 == Int; // report error if false
5   return Int;
6 }
```

The rule of let expression

$$\frac{\begin{array}{c}\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_0 : \mathtt{T}_0 \\ \mathtt{O}[\mathtt{T}/\mathtt{x}], \mathtt{M}, \mathtt{C} \vdash \mathtt{e}_1 : \mathtt{T}_1 \\ \mathtt{T}_0 \leq \mathtt{T}\end{array}}{\mathtt{O}, \mathtt{M}, \mathtt{C} \vdash \mathtt{let}\ \mathtt{x} : \mathtt{T} \leftarrow \mathtt{e}_0\ \mathtt{in}\ \mathtt{e}_1 : \mathtt{T}_1}$$

can be implemented as

```
1 TypeCheck(Environment, let x:T ← e0 in e1) {
2   TypeCheck(Environment, e0) = T0;
3   TypeCheck(Environment, e1) = T1;
4   Check subtype(T0, T); // report error if false
5   return T1;
6 }
```

4.3.7 Static v.s. dynamic typing

Static type checking systems detect common errors at compile time. Unfortunately, some correct programs from the perspective of runtime are disallowed by the type checker. In order to tackle this problem, some argue for dynamic type checking instead, while others want more expressive, but meantime more complex static type checking.

One of the ideas this discussion suggests is that there are two different notions of type: the dynamic type and the static type. Dynamic type is a runtime notion. The dynamic type of an object C is the class C used in the `new C` expression that created it. Static type is a compile notion. It captures all dynamic types the expression can have. For simple type systems, we have the soundness theorem, which asserts that for all expression E, dynamic_type(E) = static_type(E). However, for complex type systems like that of Cool, this is not always true. For example, if class A inherits class B, then `new A` can be assigned to a variable b that has static type B. The soundness theorem of Cool should be dynamic_type(E) $\leq$ static_type(E), $\forall$E, which implies that subclasses can only add attributes or methods. Methods can be overridden in subclasses, but their types should be observed.

4.4 Self type

4.4.1 Introduction

As an example of "more expressive static typing", we will discuss the type rule of self in Cool.

Consider the following class:

```
1 class Count {
2   i : int ← 0;
3   inc() : Count {{ i ← i + 1; self; }};
4 };
```

This class incorporates a counter and thus can serve as a base class for any class that needs this functionality. Consider one of such classes, the Stock class:

```
1 class Stock inherits Count {
2   name : String;
3 };
4 class Main {
5   main() : Object {
6     Stock a ← (new Stock).inc();
7     a.name = ...
8   };
9 };
```

If the Cool type checker that we have developed is employed on this piece of code, it will actually complain about a type mismatch error: a is expecting a value of type Stock, but `(new Stock).inc()` has static type Count as declared

in class Count. In order to use the functionality of the counter, all subclasses of Count have to do the tedious job of redefining `inc()`.

An extension of the current type system provides an elegant solution to the problem. Instead of specifying the return type of `inc()` as Count or any other subclass of Count, we require that `inc()` should return the type of self, which could be implemented by introducing a new keyword SELF_TYPE:

$$\texttt{inc}() : \texttt{SELF\_TYPE}\{\ldots\}$$

This mechanism allows the return type of `inc()` changing according to the dynamic type of the object calling it. The type checker can now prove

$$\texttt{O}, \texttt{M}, \texttt{C} \vdash (\texttt{newCount}).\texttt{inc}() : \texttt{Count}$$
$$\texttt{O}, \texttt{M}, \texttt{C} \vdash (\texttt{newStock}).\texttt{int}() : \texttt{Stock}$$

4.4.2 Self type operations

In order to make the mechanism of SELF_TYPE work, we need to fully incorporate it into the current type system. We have defined two operations on types: the subtype relationship ($\texttt{T}_1 \leq \texttt{T}_2$) and the least upper bound ($\text{lub}((\texttt{T}_1, \texttt{T}_2))$). They are extended to handle SELF_TYPE as follows.

For subtype relationship:

- $\text{SELF\_TYPE}_C \leq \text{SELF\_TYPE}_C$ (we never have to compare SELF_TYPE from different classes)
- $\text{SELF\_TYPE}_C \leq \text{T}$ if $\text{C} \leq \text{T}$ (which implies $\text{SELF\_TYPE}_C \leq \text{C}$)
- $\text{T} \leq \text{SELF\_TYPE}_C$ always false

For least upper bound:

- $\text{lub}(\text{SELF\_TYPE}_C, \text{SELF\_TYPE}_C) = \text{SELF\_TYPE}_C$
- $\text{lub}(\text{SELF\_TYPE}_C, \text{T}) = \text{lub}(\text{C}, \text{T})$
- $\text{lub}(\text{T}, \text{SELF\_TYPE}_C) = \text{lub}(\text{C}, \text{T})$

4.4.3 Self type usage

SELF_TYPE is not allowed everywhere a type can appear. Its usage is restricted by the following rules.

1. In an inheritance chain `class T inherits T'`, neither T nor T' can be SELF_TYPE.
2. In an attribute declaration `x:T`, T can be SELF_TYPE.
3. In a let expression `let x:T in E`, T can be SELF_TYPE.

4. In `new T`, T can be SELF_TYPE. It creates an object of the same dynamic type as self.

5. In a static dispatch $\texttt{m@T}(\texttt{E}_1 \ldots \texttt{E}_n)$, T cannot be SELF_TYPE.

6. In a method definition `m(x:T):T'{...}`, only T' can be SELF_TYPE. T cannot be SELF_TYPE because that would result in a $\texttt{T}_0 \leq \texttt{SELF\_TYPE}$ requirement in a dispatch, which is never true. Furthermore, take the following example.

```
1 class A { comp(x : SELF_TYPE) : Bool {...}; };
2 class B inherits A {
3   b : int;
4   comp(x : SELF_TYPE) { x.b <- 0 };
5 }
6 ...
7 let x : A ← new B in x.comp(new A);
8 ...
```

   Here x and new A both have static type A, thus there is no problem during type checking. But at runtime, since x has dynamic type B, it will try to access attribute b of new A when executing comp(), which causes undefined behavior, usually a crash.

4.4.4 Self type checking

A type checking rule $\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e} : \texttt{T}$ means that an expression e occurring in the body of class C has type T given the variable type environment O and method signatures M. Most type rules using SELF_TYPE remain just the same, except that the $\leq$ and `lub` are the new ones. There are some rules that need to be updated.

The old rule for dispatch requires that the return type is not SELF_TYPE:

$$\frac{\begin{array}{c}\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_0 : \texttt{T}_0 \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_1 : \texttt{T}_1 \\ \ldots \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_n : \texttt{T}_n \\ \texttt{M}(\texttt{T}_0, \texttt{f}) = (\texttt{T}'_1, \ldots, \texttt{T}'_n, \texttt{T}_{n+1}) \\ \texttt{T}_{n+1} \neq \texttt{SELF\_TYPE} \\ \texttt{T}_i \leq \texttt{T}'_i \text{ for } 1 \leq i \leq n\end{array}}{\texttt{O}, \texttt{M} \vdash \texttt{e}_0.\texttt{f}(\texttt{e}_1, \ldots, \texttt{e}_n) : \texttt{T}_{n+1}}$$

In case the return type is SELF_TYPE:

$$\begin{array}{c}\texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_0 : \texttt{T}_0 \\ \texttt{O}, \texttt{M}, \texttt{C} \vdash \texttt{e}_1 : \texttt{T}_1 \\ \ldots\end{array}$$

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_n} : \mathtt{T_n} \\ \mathtt{M}(\mathtt{T_0},\mathtt{f}) = (\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{SELF\_TYPE}) \\ \mathtt{T_i} \leq \mathtt{T'_i}\ \mathtt{for}\ 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O},\mathtt{M} \vdash \mathtt{e_0.f}(\mathtt{e_1},\ldots,\mathtt{e_n}) : \mathtt{T_0}}$$

Similarly for static dispatch:

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_1} : \mathtt{T_1} \\ \ldots \\ \mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_n} : \mathtt{T_n} \\ \mathtt{M}(\mathtt{T},\mathtt{f}) = (\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{T_{n+1}}) \\ \mathtt{T_0} \leq \mathtt{T} \\ \mathtt{T_{n+1}} \neq \mathtt{SELF\_TYPE} \\ \mathtt{T_i} \leq \mathtt{T'_i} \quad \mathtt{for} \quad 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_0@T.f}(\mathtt{e_1},\ldots,\mathtt{e_n}) : \mathtt{T_{n+1}}}$$

the rule becomes

$$\frac{\begin{array}{c}\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_0} : \mathtt{T_0} \\ \mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_1} : \mathtt{T_1} \\ \ldots \\ \mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_n} : \mathtt{T_n} \\ \mathtt{M}(\mathtt{T},\mathtt{f}) = (\mathtt{T'_1},\ldots,\mathtt{T'_n},\mathtt{SELF\_TYPE}) \\ \mathtt{T_0} \leq \mathtt{T} \\ \mathtt{T_i} \leq \mathtt{T'_i} \quad \mathtt{for} \quad 1 \leq \mathtt{i} \leq \mathtt{n}\end{array}}{\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{e_0@T.f}(\mathtt{e_1},\ldots,\mathtt{e_n}) : \mathtt{T_0}}$$

Note that we are returning type $\mathtt{T_0}$ rather than C, because the type of self (i.e. $\mathtt{T_0}$) can be a subtype of the type in which method f is defined (i.e. C).

There are two new rules due to the introduction of SELF_TYPE:

$$\frac{}{\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{self} : \mathtt{SELF\_TYPE_C}}$$

$$\frac{}{\mathtt{O},\mathtt{M},\mathtt{C} \vdash \mathtt{newSELF\_TYPE} : \mathtt{SELF\_TYPE_C}}$$

4.4.5 Error recovery

Detecting where errors occur during type checking is easier that during parsing because there is no need to skip over portions of code.

The main problem is what type should be given to an expression with no legitimate type. This type will influence the typing of the enclosing expression.

One choice is to assign type Object to ill-typed expressions. But usually this does not help much because Object does not conform to most of the type rules, thus the error will propagate up the AST, eventually escalating to a series of type errors.

A better approach is to introduce a new type `No_type` for use with ill-typed expressions. It has the property that `No_type`$\leq$`C` for any type C. As a result, every operation is defined for `No_type`. `No_type` will be propagated up the AST just like Object in the previous approach, but the cascading errors disappear. Nonetheless, `No_type` makes the class hierarchy no longer a tree structure. It actually becomes a DAG, which causes implementation difficulty.

Chapter 5

Runtime organization

Up to now we have completed the front-end phase of the compiler: lexical analysis, parsing and semantic analysis. These three phases intend to enforce the language definitions. If no errors were generated during the front-end phases, the program proves to be valid in the language, and we are ready to proceed to the backend phases: optimization and code generation. However, before we can talk about the backend phases, we need to talk about runtime organization, which is essential to help us understand what we are trying to generate.

The main topics of this section include run-time resources management, correspondence static(compile-time) and dynamic (run-time)structures, and storage organization.

Exectution of a program is initially under the control of the operating system. When a program is invoked, the OS allocates space for the program, the code is loaded into part of the space, and the OS jumps to the entry point of the program, i.e. the "main" function.

The memory space allocated for a program is not necessarily contiguous. Besides the part of space storing the code, the rest of the space stores data. The compiler is responsible not only for generating the code, but also for orchestrating the data area.

5.1 Activations

We have two goals in code generation: the correctness of the code in the sense that it correctly implements the program intended by the programmer, and the speed of the program. Complications in code generation originates from the need to solve the two problems simutaneously. Over history, an elaborate framework has been developped to ensure that the two goals can be achieved together. Activation is the first topic in our discussion of the framework.

We will assume that the programming languages for which we are trying to generate code satisfy:

1. Execution is sequential. Control moves from one point in a program to

another in a well defined order. This assumption is violated if a language supports concurrency.

2. When a procedure is called, control always returns to the point immediately after the call. This assumption is violated if the language supports advanced control mechanism such as exceptions and call/cc.

An invocation of procedure P is called an **activation** of procedure P. The **lifetime** of an activation of P is all the steps to execute P, including all steps in procedures called by P. Similarly, we can define the **lifetime** of a variable x as the portion of execution in which x is defined. Note that lifetime is a dynamic/runtime concept, while scope is a static concept.

From the definitions and our assumptions, it is clear that when procedure P calls procedure Q, Q must return before P returns. Thus lifetimes of procedure activations are properly nested, which makes them suitable to be depicted as a tree, i.e. the activation tree. The activation tree depends on runtime behavior, and can be different for different inputs. Since activations are nested, we can use a stack to track currently active procedures. The procedure stack comes after the part to store code in the memory. It grows when new procedure is called, and shrinks when the current procedure returns.

The information needed to manage one procedure is called an **activation record (AR)**, or a **frame**. Activation record keeps track of the information needed to properly execute a procedure. If procedure F calls G, the G's activation record contains a mix of information about F and G. In this case, F is suspended until G completes, at which point F resumes. G's AR contains information needed to complete the execution of G, and to resume execution of F. Condiser the following Cool procedure:

```
1 Class Main {
2   g():Int { 1 };
3   f(x:Int):Int { if x=0 then g() else f(x-1) fi};
4   main():Int { f(3) };
5 };
```

Main has no argument or local variables, and its result is never used. Thus its AR is not interesting. We will focus on the AR of f. The AR of f contains

- result of f (return value)
- argument
- control link (a pointer to the previous activation, i.e. the caller)
- return address (memory address of the instruction to jump to after f completes, i.e. where execution resumes after a procedure call finishes)

This is just one of many possible AR designs. It would also work for C, Pascal, FORTRAN, etc. The advantage of placing the return value at the 1st position in a frame is that the caller can find it at a fixed offset from its own frame. An AR design is better as long as it improves execution speed

or simplifies code generation. In practice, compilers hold as many frames in registers as possible, especially results and arguments of procedures.

The compiler must determine **at compile time** the layout of activation records and generate code that correctly accesses locations in the activation records. Thus, the AR layout and the code generator must be designed together.

5.2 Globals and heap

All references to a global variable point to the same object, thus they cannot be be stored in an activation record which is deallocated after an activation is completed. Globals are assigned a fixed address once, and we call these variables "statically allocated" because they are allocated during compile time. Depending on the language, there may be other statically allocated values.

Besides globals, a value that outlives the procedure that creates it cannot be kept in the AR either. For example, for the method `foo() {new Bar}`, the Bar value must survive the deallocation of foo's AR. Languages with with dynamically allocated data use a heap to store dynamic data.

Now we can summarize different kinds of data that a language implementation has to deal with.

- The code area contains object code. For many languages it is of fixed size and read-only.
- The static area contains data with fixed addresses, e.g. globals. This area is of fixed size, and can be read-only or writable.
- The stack contains an AR for each currently active procedure. Each AR is usually of fixed size, and contains the locals of the procedure.
- Heap contains all other data. In C, heap is managed by `malloc` and `free`, while in JAVA there is `new` for allocation and garbage collection mechanism takes care of reclaimation of heap space no longer to be used.

Both stack and heap grows. We should make sure that they do not grow into each other. A simple solution is to let them start at opposite ends of the memory and grow towards each other. We end up with the partition of the memory shown in Figure 5.1.

5.3 Alignment

Alignment is a very low-level but yet very important machine architecture detail for programmers trying to implement a compiler. Most modern machines are 32-bit or 64-bit, i.e. there are 4 or 8 bytes in a word. Machines are either byte or word addressable. A piece of data is said to be aligned if it begins at a word boundary. Most machines have some sort of alignment restrictions or performance penalties for poor alignment.

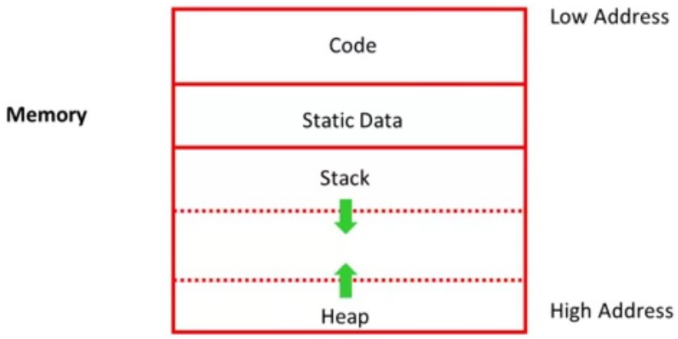


Figure 5.1: Partition of memory

Chapter 6

Code generation

6.1 Stack machines

A stack machine uses only a stack as storage. When executing an instruction $\mathtt{r} = \mathtt{F}(\mathtt{a_1}, \ldots, \mathtt{a_n})$, it pops n operands from the stack, computes the operation F using the operands, and pushes the result r back on the stack. For example, when computing 7 + 5, the stack will change from s-7-5 to s-12.

Consider two instructions: push i (push integer i on the stack) and add (add two integers). Then we have the program to compute 7 + 5:

push 7
push 5
add

An important property of stack machines is that location of the operands/result is not explicitly stated because they are always at the top of the stack. This is different from register machine in which the locations have to be specified. We have `add` instead of $\mathtt{add}\ \mathtt{r_1}, \mathtt{r_2}, \mathtt{r_3}$, which produces more compact programs. This is one of the reasons why JAVA bytecodes uses stack evaluation.

Stack machine produces compact programs, but register machine executes faster. There is an intermediate point between the two kinds of machines called an **n-register stack machine**. As the name reveals, the top n positions of the pure stack machine's stack are held in registers. It turns out that even one single register can provide considerable performance improvement, which is the case of **1-register stack machine**. The register is called the **accumulator**.

In a pure stack machine, an add does 3 memory operations: two reads and one write. But in a 1-register stack machine, what add does is acc←acc + top_of_stack. In general, consider an arbitrary expression $\mathtt{op}(\mathtt{e_1}, \ldots, \mathtt{e_n})$. For each subexpression $e_i (1 \leq i \leq n-1)$, we will compute e_i, store the result in acc and then push the result on the stack. For e_n, we will have its result remain in acc. Then we will pop n-1 values from the stack to compute op, and store the result in acc. If we follow this procedure, obviously after evaluating an

expression e, acc holds the value of e, and the stack is unchanged. In other words, **expression evaluation preserves the stack.**

Consider the calculation of 3 + (7 + 5). We will have the following process:

Code	Acc	Stack
acc←3	3	<init>
push acc	3	3,<init>
acc←7	7	3,<init>
push acc	7	7,3,<init>
acc←5	5	7,3,<init>
acc←acc + top_of_stack	12	7,3,<init>
pop	12	3,<init>
acc←acc + top_of_stack	15	3,<init>
pop	15	<init>

6.2 Basic MIPS instructions

In our discussion of code generation, we will focus on generating code for a stack machine with accumulator. The resulting code should be able to run on an MIPS processor (or simulator). Thus we have to simulate stack machine instructions using MIPS instructions and registers.

We choose to keep the accumulator in MIPS register $a0. The stack is kept in memory and grows towards lower addresses, which is a standard convention in MIPS. The address of the next location on the stack is kept in MIPS register $sp (which stands for stack pointer), and the top of the stack is at address $sp + 4.

MIPS is an old structure with a relatively simple instruction set (prototypical reduced instruction set computer, or RISC). Most MIPS operations use registers for operands and results. Load and store instructions are used to move values to and from memory. There are 32 general purpose registers (32 bits each) in MIPS, and we will use only $a0, $sp and $t1 (temp register used for operations that take two arguments).

Here are the first set of MIPS instructions that we introduce.

lw reg1 offset(reg2) : Load 32-bit word from address `reg2 + offset` into `reg1`.

sw reg1 offset(reg2) : Store 32-bit word in `reg1` at address `reg2 + offset`.

add reg1 reg2 reg3 : `reg1` ← `reg2 + reg3`

addiu reg1 reg2 imm : `reg1` ← `reg2 + imm`. `u` means that overflow is not checked.

li reg imm : `reg` ← `imm`.

move reg1 reg2 reg1 ← reg2.

The stack machine code for 7 + 5 is:

```
acc ← 7                      li $a0 7
push acc                     sw $a0 0($sp)
                             addiu $sp $sp -4
acc ← 5                      li $a0 5
acc ← acc + top_of_stack     lw $t1 4($sp)
                             add $a0 $a0 $t1
pop                          addiu $sp $sp 4
```

6.3 Code generation for a simple language

In this section we will take a look at code generation for higher level languages rather than a simple stack machine in the previous section.

Consider a language for integer operations. Its grammar depicts a list of function definitions:

$$\texttt{P} \rightarrow \texttt{D; P} \mid \texttt{D}$$
$$\texttt{D} \rightarrow \texttt{def id(ARGS)} = \texttt{E}$$
$$\texttt{ARGS} \rightarrow \texttt{id, ARGS} \mid \texttt{id}$$
$$\texttt{E} \rightarrow \texttt{int} \mid \texttt{id} \mid \texttt{if } \texttt{E}_1 = \texttt{E}_2 \texttt{ then } \texttt{E}_3 \texttt{ else } \texttt{E}_4 \mid \texttt{E}_1 + \texttt{E}_2 \mid \texttt{E}_1 - \texttt{E}_2 \mid \texttt{id}(\texttt{E}_1, \ldots, \texttt{E}_n)$$

The first function definition `f` is the entry point, i.e. the `main` routine. This language is enough to write a program that computes the Fibonacci numbers:

```
def fib(x) = if x = 1 then 0 else
             if x = 2 then 1 else
             fib(x - 1) + fib(x - 2)
```
(6.1)

For each expression `e`, we want to generate MIPS code that **compute the value of `e` in $a0** and **preserves $sp and the content of the stack**. We will define a function `cgen(e)` whose return value if the code generated for `e`.

6.3.1 Constants

For constants, we need to simply load it into the accumulator:

```
cgen(i) = li $a0 i
```

6.3.2 Addition

For addition:

```
cgen(e1 + e2) =
```

```
cgen(e1)
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Here we use different colors to emphasize the fact that MIPS code is **generated** at compile time and **executed** at run time. Code in red color is what happens at compile time: the generation, while code in black is what happens at run time: the execution. More precisely, we should write `sw $a0 0($sp)` as something like `print sw $a0 0($sp)`, which indicates that the MIPS code is generated at compile time and saved somewhere, maybe in an intermediate file, and does not get executed until run time.

It seems that the piece of code above could be optimized: why don't we save the value of e1 directly in $t1, rather than saving it in the memory and then retrieve it into $t1? The code will look like:

```
cgen(e1 + e2) =
        cgen(e1)
        move $t1 $a0
        cgen(e2)
        add $a0 $t1 $a0
```

Unfortunately, this neat piece of code is wrong. We can convince ourselves by simply considering the code generated for 1 + (2 + 3). In short, the value of $t1 will be modified during `cgen(e2)`, and is no longer the value of e1 when the execution comes to `add $a0 $t1 $a0`.

The simple example of code generation for addition demonstrates a few universal properties of code generation. The code generated for `e1 + e2` is a template with "holes" for code generated to evaluate e1 and e2. Stack machine code generation is actually recursive. The code generation process can be written as a recursive descent of the AST, at least for expressions.

6.3.3 Subtraction

By introducing another MIPS instruction sub:

sub reg1 reg2 reg3 reg1←reg2 - reg3

we can generate the code for `e1 - e2`:

```
cgen(e1 - e2) =
        cgen(e1)
```

```
sw $a0 0($sp)
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
sub $a0 $t1 $a0
addiu $sp $sp 4
```

6.3.4 If-then-else

In order to generate code for `if-then-else` expressions, we need to introduce a couple of new instructions:

beq reg1 reg2 label branch to label if reg1 = reg2

b label Unconditional jump to label

```
cgen(if e1 = e2 then e3 else e4) =
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    cgen(e2)
    lw $t1 4($sp)
    addiu $sp $sp 4
    beq $t1 $a0 true_branch
    false_branch:
      cgen(e4)
      b end_if
    true_branch:
      cgen(e3)
    end_if:
```

6.3.5 Function calls & definitions, variable references

Code for function calls and function definitions depends on the layout of the activation record, thus they need to be designed together. For this simple language, a very simple AR is sufficient.

- Since the result is always in the accumulator, there is no need to store the result in AR.
- The AR should hold the actual parameters, thus for $f(x_1, \ldots, x_n)$, we need to push $x_n, \ldots, x_1$ on the stack. These are the only variables in this language.

- The stack discipline guarantees that $sp is preserved after a function call. Thus there is no need for a control link: the previous activation can be found directly; and we do not need to look at another activation during the function call because there is no non-local variable.
- We need the return address.
- A pointer to the **current** activation is useful. It lives in register $fp (frame pointer).

To summarize, for this simple language, an AR will contain the caller's frame pointer, the actual parameters and the return address. The caller's frame pointer should be contained because $fp will be probably overwritten during the execution of the function (by other functions called during the execution). Consider a call to $f(x, y)$. Before its body gets executed, its AR looks like Figure 6.1. The **calling sequence** is the instructions (of both the caller and the

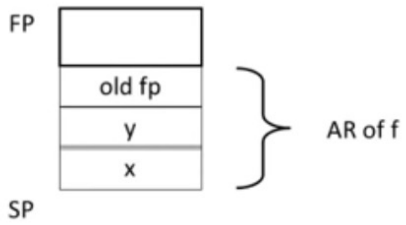


Figure 6.1: AR of f(x,y)

callee) to set up a function invocation. Here we introduce a new instruction

jal label Jump to label, and save the address of the next instruction in $ra (meaning return address). **jal** means jump and link.

Now we can actually generate the code for function call expressions.

```
cgen(f(e1,e2,...,en)) =
    sw $fp 0($sp)
    addiu $sp $sp -4
    cgen(en)
    sw $a0 0($sp)
    addiu $sp $sp -4
    ...
    cgen(e1)
    sw $a0 0($sp)
    addiu $sp $sp -4
    jal f_entry
```

For `f(e1,e2,...,en)`, we first save the frame pointer of the caller, then save the arguments one by one (from `en` to `e1`). Up to now we've completed the calling sequence on the caller side. Next we can use `jal` to jump to the entry point of function `f`. The return address is now in $ra. The AR so far is `4*n+4` bytes long.

We introduce another instruction

jr reg Jump to the address in register reg.

Now we can discuss the callee side of the calling sequence.

```
cgen(def f(x1,x2,...,xn) = e) =
    f_entry:
      move $fp $sp
      sw $ra 0($sp)
      addiu $sp $sp -4
      cgen(e)
      lw $ra 4($sp)
      addiu $sp $sp 4n + 8
      lw $fp 0($sp)
      jr $ra
```

First we set up the frame pointer by saving the current stack pointer into $fp. Then the return address is saved in memory. Now we can generate code for the function body. The stack pointer will be reserved, thus we can load the return address back into $ra. Next we can pop the return address, all arguments and the old fp out of the stack (totally 4*n+8 bytes). We restore the value of the old fp, and finally jump back to the return address.

Variables in this language are just the function arguments. They are all pushed into the AR by the caller. But since the stack grows when intermediate results are saved, the variables are not at fixed offsets from $sp. That's when $fp should be used. The offset of `xi` from $fp is 4*i. Thus we can generate the code for variable reference.

```
cgen(xi) = lw $a0 4*i($fp)
```

6.3.6 Summary

To summarize, the AR must be designed together with the code generator. Code generation can be completed by a recursive traversal of the AST. Such an approach using a stack machine is a wise choice to implement the COOL code generator.

Production compilers are for sure different from the example here. They emphasize keeping values in registers, especially the current stack frame, for the

code to run faster. Also, intermediate results are laid out in the AR rather than pushed and popped from the stack.

As an example, let's generate the code for the following program:

```
1 def sumto(x) = if x = 0 then 0 else x + sumto(x - 1)
```

```
sumto_entry:
  move $fp $sp          //set up frame pointer
  sw $ra $sp            //return address
  addiu $sp $sp -4
  lw $a0 4($fp)         //start generation for if-then-else. load x from AR.
  sw $a0 0($sp)         //save value of x (1st arg of comparison)
  addiu $sp $sp -4
  li $a0 0              //immediately load 0
  lw $t1 4($sp)         //load x (1st arg of comparison) into $t1
  addiu $sp $sp 4       //pop x
  beq $t1 $a0 true1     //compare and branch
false1:
  lw $a0 4($fp)         //load x from AR
  sw $a0 0($sp)         //save value of x (1st arg of addition)
  addiu $sp $sp -4
  sw $fp 0($sp)         //save frame pointer of caller
  addiu $sp $sp -4
  lw $a0 4($fp)         //load x from AR
  sw $a0 0($sp)         //save value of x (1st arg of subtraction)
  addiu $sp $sp -4
  li $a0 1              //immediately load 1
  lw $t1 4($sp)         //load x (1st arg of subtraction into $t1)
  sub $a0 $t1 $a0       //subtraction
  addiu $sp $sp 4       //pop x
  sw $a0 0($sp)         //save value of x-1 (arg of sumto(x-1))
  addiu $sp $sp -4
  jal sumto_entry       //jump and link
  lw $t1 4($sp)         //load x (1st arf of addition) into $t1
  add $a0 $t1 $a0       //addition (x+sumto(x-1))
  addiu $sp $sp 4       //pop x. up to now finished x+sumto(x-1)
  b end_if1
true1:
```

```
  li $a0 0            //immediately load 0
end_if1               //up to now finished if-then-else
  lw $ra 4($sp)       //load return address
  addiu $sp $sp 12    //pop ra, argument(x) and old fp
  lw $fp 0($sp)       //restore old fp
  jr $ra
```

6.4 Temporaries

One of the advantages of production compilers over the simple one we introduced is that temporaries are kept in the AR. In order to generate more efficient code, the code generator must assign a fixed location in the AR for each temporary. If we use NT(e) to represent the number of temporaries to evaluate expression e. We will have

$$\begin{aligned} NT(e_1 + e_2) &= \max(NT(e_1), 1 + NT(e_2)) \\ NT(e_1 - e_2) &= \max(NT(e_1), 1 + NT(e_2)) \\ NT(\text{if } e_1 = e_2 \text{ then } e_3 \text{ else } e_4) &= \max(NT(e_1), 1 + NT(e_2), NT(e_3), NT(e_4)) \\ NT(id(e_1, \ldots, e_n)) &= \max(NT(e_1), \ldots, NT(e_n)) \\ NT(int) &= 0 \\ NT(id) &= 0 \end{aligned}$$

The rule for $id(e_1, \ldots, e_n)$ is correct because the temps to store $e_1, \ldots, e_{i-1}$ are stored in the new AR, not the current AR.

Consider the Fibonacci function (6.1). With the rules above applied, 2 temps are enough to evaluate it.

For a function definition `def f(x1,...,xn) = e`, the AR has `2 + n + NT(e)` elements: return address, frame pointer, n arguments and NT(e) locations for intermediate results. The layout of the AR is shown in Figure 6.2.

Now that we have knowledge of how many temps are needed to evaluate a function and where they are going to be stored in the AR, what's left is to keep track of how many temps are in use at each point in the program. To achieve this, we will add a new argument to code generation: **the position of the next available temp**. The temp area will be used like a small, fixed-size stack.

Here is the old code generation for `e1 + e2`:

```
cgen(e1 + e2) =
        cgen(e1)
        sw $a0 0($sp)
```

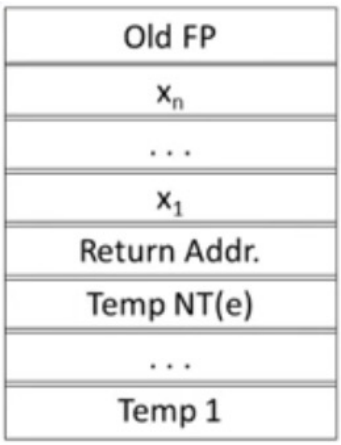


Figure 6.2: Layout of AR

```
addiu $sp $sp -4
cgen(e2)
lw $t1 4($sp)
add $a0 $t1 $a0
addiu $sp $sp 4
```

Under the new scheme, code generation will take a new argument:

```
cgen(e1 + e2, nt) =
        cgen(e1, nt)
        sw $a0 nt($fp)
        cgen(e2, nt + 4)
        lw $t1 nt($fp)
        add $a0 $t1 $a0
```

6.5 Object layout

In this section we will focus on code generation for a more advanced feature: objects.

In OO programming, if B is a subclass of A, the an object of class B can be used whenever an object of class A is used, which means that code generated for class A has to work without any modification for class B. To give a complete description of the code generation for objects, two questions need to be answered:

- How are objects represented in memory?
- How is dynamic dispatch implemented?

We will use the following COOL code to illustrate the code generation for objects.

```
1  class A {
2    a:Int <- 0;
3    d:Int <- 1;
4    f():Int { a <- a + d };
5  };
6  class B inherits A {
7    b:Int <- 2;
8    f():Int { a };
9    g():Int { a <- a - b };
10 };
11 class C inherits A {
12   c:Int <- 3;
13   h():Int { a <- a * c };
14 }
```

`A` is the base class while `B` and `C` inherits from it. Note that attributes `a` and `d` are inherited by `B` and `C`, and that `a` is used in all methods of the 3 classes. For these methods to work correctly, attribute `a` must be in the same "place" in each object. To ensure this, 2 conditions have to be satisfied:

- Objects are laid out in contiguous memory.
- Each attribute is stored at a fixed offset in the object.

Let's take a look at the layout of an object in COOL, as shown in Figure 6.3. The first 3 words are always header information. **Class tag** is an integer that

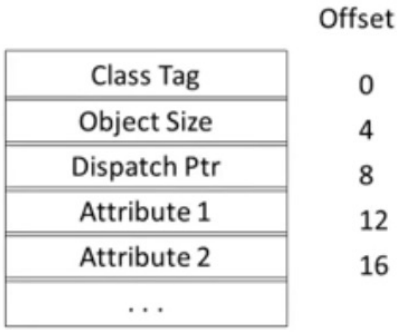


Figure 6.3: Layout of COOL object

identifies the class of the object. **Object size** is an integer that specifies the size of the object in words. **Dispatch pointer** is a pointer to a table of methods,

which will be explained in detail later. They are followed by attributes of the class in subsequent slots. All these are laid out in contiguous memory.

Given the layout of class A, the layout of its subclass B can be defined by extending the layout of A with additional slots for the additional attributes of B. Thus the layout of A is left unchanged.

In our example, the layout of the 3 classes are shown in Table 6.1. In general,

Table 6.1: Layout of classes `A,B,C`

class \ offset	0	4	8	12	16	20
A	Atag	5	*	a	d	
B	Btag	6	*	a	d	b
C	Ctag	6	*	a	d	c

if B is a subclass of A, then an A object is "nested" inside an B object.

Every class has a fixed set of methods, including inherited methods. A **dispatch table** indexes these methods. It is an array of method entry points. A method `f` lives at a fixed offset in the dispatch table for a class and all of its subclasses. The dispatch tables of classes `A,B,C` are shown in Table 6.2. The

Table 6.2: Dispatch tables of classes `A,B,C`

class \ offset	0	4
A	fA	
B	fB	g
C	fA	h

dispatch pointer in an object of class X points to the dispatch table of class X. The reason for which we use a pointer to the dispatch table rather than putting all the methods in each object directly is that each object can have its own copy of the attributes, but the methods are always the same.

Every method f of class X is assigned an offset O_f in the dispatch table at compile time. To implement a dynamic dispatch `e.f()`, we will evaluate `e` to get an object `x`, and then call $D[O_f]$, in which D is the dispatch table for `x`. In the call, `self` is bound to `x`.

6.6 Semantics

Semantics is intended to specify for every expression what happens when it gets evaluated. We have introduced a few formal notations to define a programming language:

- Tokens (regular expressions) $\Rightarrow$ lexical analysis

- Context free grammars ⇒ syntactic analysis
- Typing rules ⇒ semantic analysis
- **The evaluation rules ⇒ code generation & optimization**

Up to now we actually have specified indirectly a complete set of evaluation rules for COOL: we compile COOL to a stack machine, then we translate the actions of the stack machine into assembly code (i.e. evaluation rules of the stack machine). But this is not a good approach because such assembly-language descriptions of language implementations involve too many irrelevant details:

- Whether to use a stack machine or not.
- Which way the stack grows.
- How integers are represented.
- The particular instruction set of the architecture.

What we want is a complete but not overly restrictive specification that allows a variety of different implementations. There exist many ways to specify semantics:

Operational semantics Program evaluation is described via execution rules on an **abstract** machine. It is the most useful for specifying implementations.

Denotational semantics Program's meaning is a mathematical function from input to output. Elegant but complex.

Axiomatic semantics Program behavior is described via logical formulae. It is the foundation of many program verification systems.

6.6.1 Operational semantics

Operational semantics are a series of logical inference rules, as type rules in type checking. A type rule $\texttt{Context} \vdash \texttt{e} : \texttt{C}$ means that in the given context, expression `e` has type `T`. Similarly, for evaluation, the rule $\texttt{Context} \vdash \texttt{e} : \texttt{v}$ means that in the given context, expression `e` has value `v`.

We will illustrate the concept with the evaluation of $\texttt{y} \leftarrow \texttt{x} + \texttt{1}$. We track variables and their values with

An environment Where in memory a variable is. A map from variables to memory locations.

A store What resides in the memory. A map from memory locations to values.

A variable environment maps variables to locations. It keeps track of which variables are in scope and where those variables are. It is denoted as follow:

$$\mathtt{E} = [\mathtt{a} : \mathtt{l}_1, \mathtt{b} : \mathtt{l}_2]$$

In this example, the environment tells us that `a,b` are respectively at location $\mathtt{l}_1, \mathtt{l}_2$.

A store maps memory locations to values. It is denoted as follow:

$$\mathtt{S} = [\mathtt{l}_1 \to \mathtt{v}_1, \mathtt{l}_2 \to \mathtt{v}_2]$$

$\mathtt{S}'[\mathtt{v}'_1/\mathtt{l}_1]$ defines a new store $\mathtt{S}'$ such that $\mathtt{S}'(\mathtt{l}_1) = \mathtt{v}'_1$ and $\mathtt{S}'(\mathtt{l}) = \mathtt{S}(\mathtt{l})$ if $\mathtt{l} \neq \mathtt{l}_1$.

Given these notations, an actual evaluation rule looks like

$$\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e} : \mathtt{v}, \mathtt{S}'.$$

Its meaning is: given `so`, the current value of `self`, under the current variable environment `E` and the current store `S`, **if the evaluation of expression `e` terminates** then the value of `e` is `v` and the new store is `S'`. Note that the store might be modified as a side-effect of the evaluation of the expression, but the variable environment and the value of `self` do not change.

6.6.2 COOL semantics

Denotation

In COOL, all values are objects, i.e. instances of some class. We use the denotation

$$\mathtt{X}(\mathtt{a}_1 = \mathtt{l}_1, \ldots, \mathtt{a}_n = \mathtt{l}_n)$$

to represent an object of the class `X` of which $\mathtt{a}_i$ is an attribute and $\mathtt{l}_i$ is the location where the value of $\mathtt{a}_i$ is stored.

For basic classes without attributes(`Int, Bool, String`), they are specially denoted as

```
Int(5), Bool(true), String(4,"Cool")
```

The integer in the denotation of String is its length.

There is a special value `void` of type `Object`. No operations except for the test `isvoid` can be performed on it. Concrete implementation might use `NULL` here.

constants

$$\frac{}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{true} : \mathtt{Bool(true)}, \mathtt{S}}, \quad \frac{}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{false} : \mathtt{Bool(false)}, \mathtt{S}}$$

$$\frac{\mathtt{i\ is\ an\ integer\ literal}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{i} : \mathtt{Int(i)}, \mathtt{S}}$$

$$\frac{\texttt{s is a string literal} \quad \texttt{n is the length of s}}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{s} : \texttt{String}(\texttt{n}, \texttt{s}), \texttt{S}}$$

identifiers

$$\frac{\texttt{E}(\texttt{id}) = \texttt{l}_{\texttt{id}} \quad \texttt{S}(\texttt{l}_{\texttt{id}} = \texttt{v})}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{id} : \texttt{v}, \texttt{S}}$$

self

$$\frac{}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{self} : \texttt{so}, \texttt{S}}$$

assignment

$$\frac{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{e} : \texttt{v}, \texttt{S}_1 \quad \texttt{E}(\texttt{id}) = \texttt{l}_{\texttt{id}} \quad \texttt{S}_2 = \texttt{S}_1[\texttt{v}/\texttt{l}_{\texttt{id}}]}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{id} \leftarrow \texttt{e} : \texttt{v}, \texttt{S}_2}$$

addition

$$\frac{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{e}_1 : \texttt{v}_1, \texttt{S}_1 \quad \texttt{so}, \texttt{E}, \texttt{S}_1 \vdash \texttt{e}_2 : \texttt{v}_2, \texttt{S}_2}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{e}_1 + \texttt{e}_2 : \texttt{v}_1 + \texttt{v}_2, \texttt{S}_2}$$

expression block

$$\frac{\begin{array}{c}\texttt{so}, \texttt{E}, \texttt{S} \vdash \texttt{e}_1 : \texttt{v}_1, \texttt{S}_1 \\ \texttt{so}, \texttt{E}, \texttt{S}_1 \vdash \texttt{e}_2 : \texttt{v}_2, \texttt{S}_2 \\ \ldots \\ \texttt{so}, \texttt{E}, \texttt{S}_{\texttt{n}-1} \vdash \texttt{e}_\texttt{n} : \texttt{v}_\texttt{n}, \texttt{S}_\texttt{n}\end{array}}{\texttt{so}, \texttt{E}, \texttt{S} \vdash \{\texttt{e}_1; \ldots; \texttt{e}_\texttt{n}\} : \texttt{v}_\texttt{n}, \texttt{S}_\texttt{n}}$$

if-then-else

$$\frac{\begin{array}{c}\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e}_1 : \mathtt{Bool}(\mathtt{true}), \mathtt{S}_1 \\ \mathtt{so}, \mathtt{E}, \mathtt{S}_1 \vdash \mathtt{e}_2 : \mathtt{v}, \mathtt{S}_2\end{array}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3\ \mathtt{fi} : \mathtt{v}, \mathtt{S}_2}$$

$$\frac{\begin{array}{c}\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e}_1 : \mathtt{Bool}(\mathtt{false}), \mathtt{S}_1 \\ \mathtt{so}, \mathtt{E}, \mathtt{S}_1 \vdash \mathtt{e}_3 : \mathtt{v}, \mathtt{S}_3\end{array}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{if}\ \mathtt{e}_1\ \mathtt{then}\ \mathtt{e}_2\ \mathtt{else}\ \mathtt{e}_3\ \mathtt{fi} : \mathtt{v}, \mathtt{S}_3}$$

while loop

$$\frac{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e}_1 : \mathtt{Bool}(\mathtt{false}), \mathtt{S}_1}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{while}\ \mathtt{e}_1\ \mathtt{loop}\ \mathtt{e}_2\ \mathtt{pool} : \mathtt{void}, \mathtt{S}_1}$$

$$\frac{\begin{array}{c}\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e}_1 : \mathtt{Bool}(\mathtt{true}), \mathtt{S}_1 \\ \mathtt{so}, \mathtt{E}, \mathtt{S}_1 \vdash \mathtt{e}_2 : \mathtt{v}, \mathtt{S}_2 \\ \mathtt{so}, \mathtt{E}, \mathtt{S}_2 \vdash \mathtt{while}\ \mathtt{e}_1\ \mathtt{loop}\ \mathtt{e}_2\ \mathtt{pool} : \mathtt{void}, \mathtt{S}_3\end{array}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{while}\ \mathtt{e}_1\ \mathtt{loop}\ \mathtt{e}_2\ \mathtt{pool} : \mathtt{void}, \mathtt{S}_3}$$

let

$$\frac{\begin{array}{c}\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e}_1 : \mathtt{v}_1, \mathtt{S}_1 \\ \mathtt{l}_{\mathtt{new}} = \mathtt{newloc}(\mathtt{S}_1) \\ \mathtt{so}, \mathtt{E}[\mathtt{l}_{\mathtt{new}}/\mathtt{id}], \mathtt{S}_1[\mathtt{v}_1/\mathtt{l}_{\mathtt{new}}] \vdash \mathtt{e}_2 : \mathtt{v}_2, \mathtt{S}_2\end{array}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{let}\ \mathtt{id} : \mathtt{T} \leftarrow \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2 : \mathtt{v}_2, \mathtt{S}_2}$$

$\mathtt{l}_{\mathtt{new}} = \mathtt{newloc}(\mathtt{S}_1)$ means that $\mathtt{l}_{\mathtt{new}}$ is a location not already used in $\mathtt{S}$.

new T

Tasks to be completed:

- Allocation locations to hold all attributes of class T.
- Set attributes with their default values.
- Evaluate the initializers and set the resulting values.
- Return the newly allocated object.

For each class A there is a default value $\mathtt{D_A}$: $\mathtt{D_{int}} = \mathtt{Int(0)}$, $\mathtt{D_{bool}} = \mathtt{Bool(false)}$, $\mathtt{D_{string}} = \mathtt{String(0,"")}$, $\mathtt{D_A} = \mathtt{void}$ (for any other class A). In order to refer to the function's attributes in the rules, we define a function

$$\mathtt{class(A)} \;=\; (\mathtt{a_1} : \mathtt{T_1} \leftarrow \mathtt{e_1}, \ldots, \mathtt{a_n} : \mathtt{T_n} \leftarrow \mathtt{e_n})$$

in which $\mathtt{a_i}$ are the attributes (including inherited ones listed in greatest ancestor first order), $\mathtt{T_i}$ are their declared types, and $\mathtt{e_i}$ are the initializers. Finally we have the evaluation rule of `new T`.

$$\frac{\begin{array}{c}\mathtt{T_0} = \mathtt{if}\ (\mathtt{T} == \mathtt{SELF\_TYPE}\ \mathtt{and}\ \mathtt{so} = \mathtt{X}(\ldots))\ \mathtt{then}\ \mathtt{X}\ \mathtt{else}\ \mathtt{T} \\ \mathtt{class}(\mathtt{T_0}) = (\mathtt{a_1} : \mathtt{T_1} \leftarrow \mathtt{e_1}, \ldots, \mathtt{a_n} : \mathtt{T_n} \leftarrow \mathtt{e_n}) \\ \mathtt{l_i} = \mathtt{newloc(S)}\ \mathtt{for}\ \mathtt{i} = 1, \ldots, \mathtt{n} \\ \mathtt{v} = \mathtt{T_0}(\mathtt{a_1} = \mathtt{l_1}, \ldots, \mathtt{a_n} = \mathtt{l_n}) \\ \mathtt{S_1} = \mathtt{S}[\mathtt{D_{T_1}}/\mathtt{l_1}, \ldots, \mathtt{D_{T_n}}/\mathtt{l_n}] \\ \mathtt{E'} = [\mathtt{a_1} : \mathtt{l_1}, \ldots, \mathtt{a_n} : \mathtt{l_n}] \\ \mathtt{v}, \mathtt{E'}, \mathtt{S_1} \vdash \{\mathtt{a_1} \leftarrow \mathtt{e_1}; \ldots; \mathtt{a_n} \leftarrow \mathtt{e_n};\} : \mathtt{v_n}, \mathtt{S_2}\end{array}}{\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{new\ T} : \mathtt{v}, \mathtt{S_2}}$$

Note that `E'` is irrelevant to `E`. It's the environment in which the attributes of the new object is evaluated. Only the attributed are in scope in this environment. Also note that during the evaluation of the initializers, `self` is the current object `v`.

dynamic dispatch $\mathtt{e_0.f(e_1, \ldots, e_n)}$

Task to be completed:

- Evaluate arguments in order $\mathtt{e_1}, \ldots, \mathtt{e_n}$.
- Evaluate $\mathtt{e_0}$ to the targeted object.
- Let `X` be the dynamic type of the target object.
- Fetch from `X` the definition of `f`(with `n` arguments).
- Create `n` new locations and an environment that maps `f`'s formal arguments to those locations.
- Initialize the locations with the actual arguments.
- Set `self` to the target object and evaluate `f`'s body.

In order to look up a method in a class, we define a function

$$\mathtt{impl(A, f)} = (\mathtt{x_1}, \ldots, \mathtt{x_n}, \mathtt{e_{body}})$$

in which $\mathtt{x_i}$ are the names of the formal arguments, and $\mathtt{e_{body}}$ is the body of the method. Now we have the evaluation rule for dynamic dispatch as follow.

$$\mathtt{so}, \mathtt{E}, \mathtt{S} \vdash \mathtt{e_1} : \mathtt{v_1}, \mathtt{S_1}$$

$$\frac{\begin{array}{c}\mathtt{so},\mathtt{E},\mathtt{S}_1 \vdash \mathtt{e}_2 : \mathtt{v}_2, \mathtt{S}_2 \\ \ldots \\ \mathtt{so},\mathtt{E},\mathtt{S}_{n-1} \vdash \mathtt{e}_n : \mathtt{v}_n, \mathtt{S}_n \\ \mathtt{so},\mathtt{E},\mathtt{S}_n \vdash \mathtt{e}_0 : \mathtt{v}_0, \mathtt{S}_{n+1} \\ \mathtt{v}_0 = \mathtt{X}(\mathtt{a}_1 = \mathtt{l}_1, \ldots, \mathtt{a}_m = \mathtt{l}_m) \\ \mathtt{impl}(\mathtt{X}, \mathtt{f}) = (\mathtt{x}_1, \ldots, \mathtt{x}_m, \mathtt{e}_{body}) \\ \mathtt{l}_{x_i} = \mathtt{newloc}(\mathtt{S}_{n+1}) \,\mathtt{for}\, \mathtt{i} = 1, \ldots, \mathtt{n} \\ \mathtt{E}' = [\mathtt{a}_1 : \mathtt{l}_1, \ldots, \mathtt{a}_m : \mathtt{l}_m][\mathtt{x}_1/\mathtt{l}_{x_1}, \ldots, \mathtt{x}_n/\mathtt{l}_{x_n}] \\ \mathtt{S}_{n+2} = \mathtt{S}_{n+1}[\mathtt{v}_1/\mathtt{l}_{x_1}, \ldots, \mathtt{v}_n/\mathtt{l}_{x_n}] \\ \mathtt{v}_0, \mathtt{E}', \mathtt{S}_{n+2} \vdash \mathtt{e}_{body} : \mathtt{v}, \mathtt{S}_{n+3}\end{array}}{\mathtt{so},\mathtt{E},\mathtt{S} \vdash \mathtt{e}_0.\mathtt{f}(\mathtt{e}_1, \ldots, \mathtt{e}_n) : \mathtt{v}, \mathtt{S}_{n+3}}$$

Note the definition of E' is an update of $\mathtt{a}_i$ by $\mathtt{x}_i$ rather than the combination of them, because it is possible that some of the arguments are attributes of the object. In such case, the arguments are in scope.

Chapter 7

Optimization

7.1 Intermediate code

Intermediate code uses a language between the source language and the target language. It provides an intermediate level of abstraction: more details than the source and fewer details than the target. High-level source languages like `Cool` and `C` reveals less low-level conceptions such as registers, making it difficult to find room for optimization, while low-level languages like assembly language are often limited to a specific type of machine architecture.

We will introduce the conception with an intermediate language that can be called a "high level assembly language". It uses register names, but has an unlimited number of registers. It uses control structures like assembly language. Opcodes are used, but some of them are higher level (e.g. `push`, which will be translated into a few assembly instructions on a machine of a particular architecture). Each instruction is either binary ($\texttt{x} := \texttt{y}\ \texttt{op}\ \texttt{z}$) or unary($\texttt{x} := \texttt{op}\ \texttt{y}$). Arguments on the right are always registers or constants. This is actually a wide-used form of intermediate code called **three-address code**. In this language, every intermediate value will have its own name.

Generating intermediate code is similar to generating assembly code, except that unlimited number of registers can be used, which renders easier code generation. If we use $\texttt{igen}(\texttt{e}, \texttt{t})$ to denote the function that generates code for expression `e` and stores the result in register `t` , we will have

$$\begin{aligned}&\texttt{igen}(\texttt{e}_1 + \texttt{e}_2, \texttt{t}) = \\ &\quad \texttt{igen}(\texttt{e}_1, \texttt{t}_1) \\ &\quad \texttt{igen}(\texttt{e}_2, \texttt{t}_2) \\ &\quad \texttt{t} := \texttt{t}_1 + \texttt{t}_2\end{aligned}$$

7.2 Optimization overview

Optimizations can be performed at different times:

1. On AST

   Pro Machine independent

   Con Too high level

2. On assembly language

   Pro Exposes the most optimization opportunities

   Con Machine dependent. Has to be reimplemented when re-targeting to different architectures.

3. On intermediate language

   Pro Machine independent. Exposes optimization opportunities.

We will discuss optimizations performed on intermediate languages. The intermediate language we use can be described with the following CFG:

$$
\begin{aligned}
&\texttt{P} \rightarrow \texttt{SP} \mid \texttt{S} \\
&\texttt{S} \rightarrow \texttt{id} \coloneqq \texttt{id op id} \\
&\quad \mid \texttt{id} \coloneqq \texttt{op id} \\
&\quad \mid \texttt{push id} \\
&\quad \mid \texttt{id} \coloneqq \texttt{pop} \\
&\quad \mid \texttt{if id relop id goto L} \\
&\quad \mid \texttt{L :} \\
&\quad \mid \texttt{jump L}
\end{aligned}
$$

`Id`'s are registers, and can be substituted by constants when they serve as arguments. Typical operations are +,-,*,/.

A **basic block** is a maximal sequence of instructions with no labels (except for the first instruction) and no jumps(except for the last instruction). The execution can only jump into a basic block at the first instruction and jump out of it at the last instruction. There is no other way to jump into it, and all instructions inside the block are guaranteed to be executed sequentially before the execution jumps out. This property enables us to conduct a series of optimizations.

A **control flow graph** is a directed graph with basic blocks as nodes. An edge from block A to block B exists if the execution can pass from the last instruction in A to the first instruction in B. The body of a method can be represented as a control flow graph. There is one initial node, and all "return" nodes are terminals.

The purpose of optimization is to **improve a program's resource utilization**. Most of the time we try to make the program run faster, i.e. reduce the execution time. Other resources that optimization could be concerned about are code size, memory footprint, network messages sent, disk accesses, power,

etc. The bottom line is that optimization should not alter what the program computes.

For languages like `C` and `Cool` there are 3 granularities of optimizations:

Local optimization Applies to an isolated basic block.

Global optimization Applies to an isolated control flow graph (method body).

Inter-procedural optimization Applies across method boundaries.

Local optimizations are performed by most mainstream compilers. Many of them also perform global optimizations. Few compilers touch inter-procedural optimizations, not only because it's hard to implement, but also because it often does not provide as much improvement as the first two. In practice, it is usually a wise decision not to implement the fanciest optimizations because they tend to be hard to implement, costly in compile time while not much payoff can be gained.

7.3 Local optimization

Local optimization focuses on a single basic block. There is no need to analyze the entire method body.

7.3.1 Constant folding

For an instruction $\texttt{x} := \texttt{y}\ \texttt{op}\ \texttt{z}$ in which `y,z` are both constants, $\texttt{y}\ \texttt{op}\ \texttt{z}$ can be computed at compile time. E.g., $\texttt{x} := 2 + 2 \Rightarrow \texttt{x} := 4$.

Constants folding can be dangerous when the compiler and the target code it generates are run on different machines, which is not uncommon in reality, e.g. in most embedded platforms. The two machines might feature different round-offs of floating point numbers. If we do constant folding according to the floating point semantics of the compile machine directly at compile time, we may end up with unwanted result at runtime. An obvious solution is to keep full precision inside the compiler and represent floating pointer numbers as string literals, and leave it to the runtime machine to handle the round offs.

7.3.2 Eliminate unreachable basic blocks

By eliminating basic blocks that cannot be reached from the initial block, we can make the program smaller, and sometimes faster, because of cache effects.

7.3.3 Common subexpression elimination

Some optimizations can be simplified if each register occurs only once on the lhs of an assignment. Intermediate code can be rewritten into **single assignment form** by introducing new registers to substitute earlier appearances of registers that are assigned more than once.

If a basic block is in single assignment form and a definition `x :=` is the first use of `x` in a block, then when two assignments have the same rhs, they are guaranteed to compute the same value, which allows us saving the trouble of computing the same expression twice. We can simply substitute the second appearance of the rhs with the register assigned in its first appearance.

7.3.4 Copy propagation

if `w := x` appears in a block, subsequent use of `w` can be replaced with `x`.

7.3.5 Dead instruction elimination

if `w := rhs` appears in a basic block, and `w` does not appear anywhere else in the program, then the instruction is dead and can be eliminated.

Each local optimization does little by itself, but typically optimizations will interact with each other. Performing one optimization might enable another one. Thus the usual approach is to iterate until no more improvements can be made. The optimizer can also be stopped at any point to limit compilation time.

7.3.6 Peephole optimization

Local optimizations can be applied directly to assembly code rather than to intermediate code. **Peephole optimization** is effective for improving assembly code. The "peephole" is a short sequence of contiguous instructions. The optimizer replaces the sequence with an equivalent but faster sequence. The process is repeated for maximum effect.

7.4 Global optimization

7.4.1 Dataflow analysis

In order to replace a use of `x` by a constant `k` we must make sure that on every path to the use of `x`, the last assignment to `x` is `x := k`. This condition is not trivial to check, considering the existence of loops and conditional branches. Global **dataflow analysis** is required to check this condition.

Global optimization tasks share several common traits:

- The optimization depends on knowing a property X at a particular point in program execution.
- Proving X at any single point requires knowledge of the entire program.
- It is OK to be conservative. If the optimization requires X being true, then we want to figure out whether X is definitely true or we don't know if X is true. It is always safe to say "don't know".

7.4.2 Global constant propagation

We use the following denotations to note the status of `x` at each program point:

$\bot$: The statement never gets executed.

C : Constant C.

$\top$: `x` is not a constant.

Once we have the status of `x` at every program point, we can simply replace the use of `x` by the associated constant where the status of `x` is constant. In order to gain this information, we define a function $\texttt{C(x, s, in/out)}$ to compute information about the value of `x` before/after the statement `s`. In the following discussion, statement `s` has immediate predecessor statements $\texttt{p}_1, \ldots, \texttt{p}_\texttt{n}$. We have the following rules to infer the `in` status of one statement from the `out` of its predecessors.

1. If $\texttt{C}(\texttt{p}_\texttt{i}, \texttt{x}, \texttt{out}) = \top$ for any `i`, then $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \top$.
2. If $\texttt{C}(\texttt{p}_\texttt{i}, \texttt{x}, \texttt{out}) = \texttt{c}_\texttt{i}$, $\texttt{C}(\texttt{p}_\texttt{j}, \texttt{x}, \texttt{out}) = \texttt{c}_\texttt{j}$, $\texttt{c}_\texttt{i} \neq \texttt{c}_\texttt{j}$, then $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \top$.
3. If $\texttt{C}(\texttt{p}_\texttt{i}, \texttt{x}, \texttt{out}) = \texttt{c}$ for at least one `i` and $\texttt{C}(\texttt{p}_\texttt{i}, \texttt{x}, \texttt{out}) = \bot$ for all other `i`, then $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \texttt{c}$.
4. If $\texttt{C}(\texttt{p}_\texttt{i}, \texttt{x}, \texttt{out}) = \bot$ for all `i`, then $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \bot$.

Now we discuss the rules to infer the `out` status of a statement from its own `in` status.

5. If $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \bot$, then $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{out}) = \bot$.
6. $\texttt{C}(\texttt{x} := \texttt{c}, \texttt{x}, \texttt{out}) = \texttt{c}$ if `c` is a constant.
7. $\texttt{C}(\texttt{x} := \texttt{f}(\ldots), \texttt{x}, \texttt{out}) = \top$, if `f(...)` is not a constant.
8. If $\texttt{y} \neq \texttt{x}$, then $\texttt{C}(\texttt{y} := \ldots, \texttt{x}, \texttt{out}) = \texttt{C}(\texttt{y} := \ldots, \texttt{x}, \texttt{in})$.

We can gain information about status of `x` on all program points with the following algorithm:

1. For every entry `s` to the program, set $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in}) = \top$.
2. Set $\texttt{C}(\texttt{s}, \texttt{x}, \texttt{in/out}) = \bot$ everywhere else.
3. Repeat until the rules above are satisfied: pick statement `s` that does not satisfy the rules and update accordingly.

Special consideration needs to be taken when it comes to loops. If the `in` status of each statement relies on its predecessors, the reliance relation will form a cycle, making it impossible to obtain a result without assigning some initial values. The initial value $\bot$, which means "by far the execution hasn't reached this point", is obviously a wise choice.

The rules above can actually be simplified by the idea of ordering among the 3 values : $\bot < \mathtt{C} < \top$. Note that different constants are not comparable to each other. Rules 1-4 can actually be written as

$$\mathtt{C}(\mathtt{s}, \mathtt{x}, \mathtt{in}) = \underset{\mathtt{i}}{\mathtt{lub}}(\mathtt{C}(\mathtt{p_i}, \mathtt{x}, \mathtt{out}))$$

in which `lub` means least upper bound[1]. The use of `lub` actually explains why the algorithm is guaranteed to terminate. All values start as $\bot$ and can only increase, thus the status at any point can change at most twice, which implies that the constant propagation algorithm is linear in program size: total number of steps $\leq$ 2 * Number of $\mathtt{C}(\dots)$ values to compute = 4 * Number of statements.

7.4.3 Liveness analysis

Once constants have been propagated, we would like to eliminate dead code. This involves the analysis of **liveness**. A variable `x` is live at statement `s` if

- There exists a statement `s'` that uses `x`.
- There is a path from `s` to `s'`.
- That path has no intervening assignment to `x`.

A statement $\mathtt{x} := \dots$ is dead code and can be removed if `x` is dead after the assignment. The propagation of liveness obeys the following rules:

1. $\mathtt{L}(\mathtt{p}, \mathtt{x}, \mathtt{out}) = \bigcup_{\mathtt{i}} \mathtt{L}(\mathtt{s_i}, \mathtt{x}, \mathtt{in})$, in which $\mathtt{s_i}$ are successors of `p`.
2. $\mathtt{L}(\mathtt{s}, \mathtt{x}, \mathtt{in}) = \mathtt{true}$ if `s` refers to `x` on the rhs.
3. $\mathtt{L}(\mathtt{x} := \mathtt{e}, \mathtt{x}, \mathtt{in}) = \mathtt{false}$ if `e` does not refer to `x`.
4. $\mathtt{L}(\mathtt{s}, \mathtt{x}, \mathtt{in}) = \mathtt{L}(\mathtt{s}, \mathtt{x}, \mathtt{out})$ if `s` does not refer to `x`.

We can obtain the liveness information of variables using the following algorithm, just as we did for constant propagation:

1. Initialize all $\mathtt{L}(\dots) = \mathtt{false}$.
2. Repeat until all statements satisfy the rules above: pick `s` where the rules are not satisfied and update accordingly.

A value can change from false to true but not the other way around, thus can change only once, which guarantees the termination of the algorithm.

Constant propagation is a forward analysis (information pushed from input to output), while liveness analysis is a backward analysis (information pushed from output to input). There exist other global dataflow analysis, and most of them can be classified as either forward or backward analysis. Most of them follow the methodology of **local rules relating information between adjacent program points**.

[1] We used this concept in the discussion of type checking

7.5 Register allocation

Intermediate code uses unlimited temporaries, which simplifies code generation and optimization, but complicates final translation to assembly. The intermediate needs to be rewritten by assigning multiple temporaries to the same register without changing the behavior of the program, so that it uses no more temporaries than the number of machine registers. Register allocation is as old as compilers.

The basic principle of register allocation is that **temporaries t_1 and t_2 can share the same register if at any point in the program at most one of them is live**. In other words, **if t_1 and t_2 are both live at any time of the program, they cannot share a register**.

7.5.1 Register interference graph (RIG)

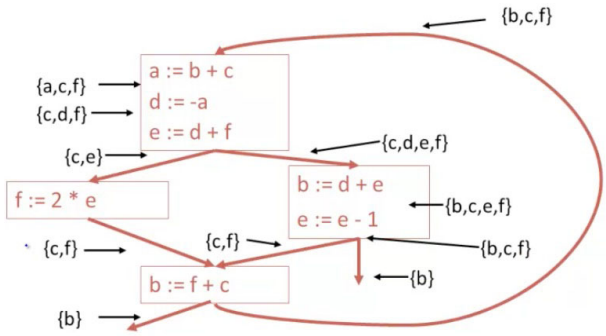


Figure 7.1: Liveness analysis of a program

Take the program in Figure 7.1 as an example. Its liveness analysis result is also shown. We can build an undirected graph according to this information. Each temporary is a node in the graph, and an edge is added between t_1 and t_2 if they are both live at some point in the program. The graph is called the **register interference graph (RIG)**. According to the principle above, two temporaries can share the same register as long as there is no edge between them in the RIG. The RIG of the program in Figure 7.1 is shown in Figure 7.2. In this example, `a` and `d` can use the same register since there is no edge between them.

RIG extracts exactly the information needed to carry out register allocation, and it provides a global (i.e. over the entire control flow graph) picture of the requirements for register allocation. After the construction of the RIG, the register allocation algorithm that we will present is architecture independent.

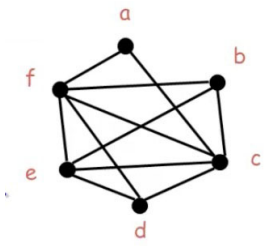


Figure 7.2: RIG

7.5.2 Graph coloring

A **coloring of graph** is an assignment of colors to nodes such that nodes connected by an edge have different colors. A graph is **k-colorable** if it can be colored using k or fewer colors. In our problem, if the RIG is k-colorable, then there must exist a register assignment that uses no more than k machine registers. The RIG shown in Figure 7.2 is 4-colorable.

We will take a divide-and-conquer approach to solve the problem.

- Pick a node `t` with **fewer** than `k` neighbors in the RIG.
- Eliminate `t` and its edges from the RIG.
- If the rest of the RIG is `k`-colorable, then so is the original RIG.

The logic is quite intuitive. Let $c_1, \ldots, c_n (n < k)$ be the colors assigned to the neighbors of `t` in the reduced graph, then we can pick a color from the $k - n$ colors unused by these neighbors to color `t` in the original RIG. The whole process can be described as below.

1. Build the stack of nodes.
   - Pick a node `t` with fewer than `k` neighbors.
   - Push `t` on the stack and remove it from the RIG.
   - Repeat until the RIG is empty.
2. Assign colors to nodes.
   - Pop the node on top of the stack.
   - Color the node with a color different from those assigned to its neighbors, which have been popped out of the stack and colored in previous steps.

7.5.3 Spilling

When the heuristic fails to find a coloring, it is impossible to hold all temporaries in the registers. Some of them have to be **spilled** to the memory.

When we get stuck in the heuristic, we will find that all nodes left in the RIG have `k` or more neighbors. We need to choose a node as a candidate for spilling. The associated value might have to live in the memory. The chosen node `f` and its edges will be removed from the RIG, and we continue the coloring with the rest of the RIG. If the coloring ends up with success, we eventually have to assign a color to `f`. If its (`k` or more) neighbors use fewer than `k` colors, a color can easily be assigned, which is the case of **optimistic coloring**. If optimistic coloring fails, `f` has to be spilled. We need to:

- Allocate a memory location, typically in the current frame, for `f`, noted as `fa`.
- Before each operation that reads `f`, insert `f` := `load fa`.
- After each operation that writes `f`, insert `store f, fa`.

Note that different `load`s and `store`s of `f` do not have to use the same register, thus occurrences of `f` should be indexed (`f1, f2,` etc.). The liveness information needs to be recalculated. The new liveness information is almost the same as the old one, except that `f` has been split into several different temporaries. `fi` is live only between a `fi` := `load fa` and the next instruction (which reads it), and between a `store fi, fa` and the preceding instruction (which writes it). Obviously, spilling reduces the interferences of `f` with other temporaries, resulting in a RIG in which each `fi` has fewer neighbors than the original `f`.

It is not uncommon that additional spilling is required before a coloring is found. The tricky part of spilling is the choice of the node to spill. There is no best choice, but it might well be helpful to spill the temporary with the most conflicts, which is the most promising choice for obtaining a colorable RIG, or the one with few definitions and uses so that the number of resulting memory operations can be minimized. Note that spilling temporaries in inner loops should be avoided for efficiency.

7.6 Cache management

Recall the memory hierarchy of modern computers: registers(KB) → cache(MB) → main memory(GB) → disk(TB). The access to caches is much faster than that to the main memory, but the cost of a cache miss is very high. Thus caches need to be properly managed. Compilers are good at managing registers, but there is little that they are able to do to manage caches, which is a job mostly left for the programmers.

Consider the following piece of code.

```
1 for(int i = 0; i < 10; ++i)
2   for(int j = 0; j < 1000000; ++j)
3     a[i] *= b[i];
```

Clearly every access to `a[i]` and `b[i]` is a cache miss. If we switch the two loops:

```
1 for(int j = 0; j < 1000000; ++j)
2   for(int i = 0; i < 10; ++i)
3     a[i] *= b[i];
```

The cache behavior becomes much better. Some, but not all, compilers implement this optimization to achieve better management of caches. Note that it's not always easy for the compiler to figure out whether the switch of loops is legal or not.

7.7 Automatic memory management (GC)

Storage management is still a hard problem in modern programming. C and C++ use manual storage management, which results in many storage bugs: memory leak, deference of dangling pointers, overwriting parts of a data structure by accident, etc. Such bugs are hard to find because they often do not show visible effect until far away in time and space. Heavy efforts have been made to develop a series of techniques for completely automatic memory management since the 1950s. But it didn't become mainstream until the popularity of JAVA in the 1990s.

The basic principle of **automatic memory management**, or **garbage collection** is simple. Memory is allocated when an object is created, and this piece of memory will be reclaimed for future allocation once the object can no longer be used again by the program.

Intuitively, a program can only use the objects that it can find. A formal definition is the **reachability** of an object. An object `x` is reachable if and only if a register contains a pointer to `x`, or another reachable object contains a pointer to `x`. All reachable objects can be found by starting from registers and following all the pointers, which requires knowledge of the AR. An unreachable object can never be used again, and is called **garbage**. Note that reachability guarantees the possibility of future uses rather than actual future uses. Thus reachable objects are actually an approximation of objects that will be used later.

We will present a few mainstream GC methods in the rest of this section.

7.7.1 Mark and sweep

The **mark and sweep** method executes 2 phases when memory runs out:

the mark phase traces all reachable objects;

the sweep phase collects garbage objects.

Every object will have an extra bit reserved for memory management called **the mark bit**, which is initialized to 0 and gets set to 1 during the mark phase if the object is reachable. The mark phase is shown in Algorithm 7.1.

The sweep phase scans the heap looking for objects with mark bit 0, who are then added to the free list as garbage to be collected later. Objects with

Algorithm 7.1 The mark phase

```
let todo = {all roots}
while todo ≠ ∅ do
    pick v ∈ todo
    todo ← todo - {v}
    if mark(v) = 0 then
        mark(v) ← 1
        let v_1, ..., v_n = the pointers contained in v
        todo ← todo ∪ {v_1, ..., v_n}
```

mark bit 1 have their mark bit reset to 0. The process is shown in Algorithm 7.2.

Algorithm 7.2 The sweep phase

```
p ← bottom of heap
while p < top of heap do
    if mark(p) = 1 then
        mark(p) = 0
    else
        add block [p... p + sizeof(p) - 1] to free list
    p ← p + sizeof(p)
```

The conception of the mark and sweep algorithm is simple, but lie behind the algorithm are a number of tricky details typical of GC algorithms.

The mark phase is invoked when we are out of memory, yet space is needed to construct the `todo` list, whose size is unbounded and thus for which space cannot be reserved in advance. A trick to solve the problem is **pointer reversal**: when a pointer is followed during reachability analysis, it is reversed to point to its parent. Essentially it helps to maintain the stack for a depth-first search of the graph. Similarly, the free list is stored in the free objects themselves.

The space for a new object is allocated from the free list. A block large enough is picked, an area of necessary size is allocated from it, and the leftover is put back in the free list. The algorithm can fragment the memory. It is necessary to merge the block when possible. Objects are not moved, thus there is no need to update pointers to objects, and it works for languages like C/C++, in which the literal value of a pointer is part of the language semantics.

7.7.2 Stop and copy

In **stop and copy**, memory is organized into two parts: the old space used for allocation and the new space used as a reserver for GC. The heap pointer points to the next available word in the old space, and an allocation does nothing but advancing the heap pointer.

When the old space is full, the program is suspended, and all reachable objects are copied from the old space into the new space. Garbage (unreachable objects) is left behind and no longer occupies any space in the new space. After the copy, the roles of the old and new spaces are switched and the program is resumed.

Since objects are physically moved, we have to fix all pointers to an object after copying it. As a solution, we store in the old object a **forwarding pointer** to the new object after it gets copied. When a pointer takes us to an object with a forwarding pointer, we can be aware of the fact that it has been moved, and fix the pointer accordingly.

The traversal can be implemented without using extra space. We partition the new space into three contiguous regions with two pointers: the `scan` pointer and the `alloc` pointer. The `alloc` pointer points to the first empty word that has not been occupied by the copied objects. The `scan` pointer is between copied objects whose pointers have been followed and those whose pointers haven't. The algorithm is shown in Algorithm 7.3. As with mark and sweep, we have

Algorithm 7.3 Stop and copy

```
while scan ≠ alloc do
    let O be the object at scan pointer
    for all pointer p contained in O do
        find O' that p points to
        if O' is without a forwarding pointer then
            copy O' to the new space (update alloc pointer)
            set a word of old O' to point to the new copy
            (set up the forwarding pointer)
            change p to point to the new copy of O'
        else
            set p in O equal to the forwarding pointer
    increment scan pointer to the next object
```

to have knowledge of how large an object is and where the pointers are stored inside it when we scan it. The latter can be unavailable for some languages, e.g. C++. In such case, we take a conservative approach: if a memory word looks like a pointer, which requires it being aligned and pointing to a valid address in the data segment, it is considered as a pointer. The set of reachable objects is overestimated. But still the objects cannot be copied and moved. And note that objects pointed to by the stack also need to be scanned and copied, which can turn out an expensive operation because the entire stack needs to be scanned.

Stop and copy is generally considered to be the fastest GC algorithm, because allocation is very cheap, and collection is relatively cheap, especially when there are a lot of garbage. But some language like C/C++ does not allow object copying, and this method cannot apply to them.

7.7.3 Reference counting

Rather than wait for memory to be exhausted, **reference counting** tries to collect an object immediately when there are no pointers to it. The number of pointers to an object (the reference count) is saved in it, and every assignment operation manipulates the reference count.

`new` returns an object with reference count equal to 1. Let `rc(o)` be the reference count of object `o`. Each assignment $\texttt{x} \leftarrow \texttt{y}$ becomes

$$\begin{aligned}
&\texttt{rc}(\texttt{o}_\texttt{y}) \leftarrow \texttt{rc}(\texttt{o}_\texttt{y}) + \texttt{1} \\
&\texttt{rc}(\texttt{o}_\texttt{x}) \leftarrow \texttt{rc}(\texttt{o}_\texttt{x}) - \texttt{1} \\
&\texttt{if}(\texttt{rc}(\texttt{o}_\texttt{x}) == \texttt{0})\, \texttt{free x} \\
&\texttt{x} \leftarrow \texttt{y}
\end{aligned}$$

Reference counting is easy to implement. There will not be long pauses in the execution for the sake of GC. But it cannot handle circular structures, and manipulating reference count at each assignment is quite slow.

Automatic memory management prevents storage bugs, but reduces programmer's control. The possible problematic pauses caused by GC are sometimes intolerable for real-time application. And memory leaks are still possible, or even likely even if GC techniques are used. A common type of error made by programmers, not a grammatical error but an error in terms of engineering, is to forget to set a pointer to `Null` when the pointer will clearly never be used again: the object won't be recognized as garbage until the pointer gets out of range, which is too late.

Chapter 8

Java

In this chapter we will apply what we have learned so far to analyze some features of Java. We will also touch some features of Java that are not included in COOL and thus haven't been covered by the course.

Java developed from the OAK project of SUN targeted at set-top devices, which never took off in the consumer electronics market. Nonetheless, Java became popular with the development of the Internet for guaranteeing better security. It was developed on the basis of several other languages: it took the type system of Modula-3, the OO design of C++/Objective C, the idea of interface in Eiffel, and the dynamic flavor of Lisp, etc.

8.1 Java arrays

Assume `B` is a subtype of `A`. If we take for granted that `B[]` is a subtype of `A[]` (i.e. array being **covariant**), we will have to face the following problem:

```
1 B[] b = new B[10];
2 A[] a = b;
3 a[0] = new A();
4 b[0].aMethodNotDeclaredInA(); //runtime error!
```

Note that the problem does not arise when `a,b` are not arrays, as shown below. Because here `a` and `b` are two different references of different types that cannot be retargeted at the same time, while in the case above, `a[0]` and `b[0]` are one reference represented by two aliases of different types. Retargeting one of the aliases (assignment to `a[0]`) will also retarget `b[0]`.

```
1 B b = new B();
2 A a = b;
3 a = new A();
4 b.aMethodNotDeclaredInA(); //no runtime error!
```

Such a type system is unsound. The standard solution is to disallow subtyping through arrays: `B[]` is subtype of `A[]` if and only if `B = A`, i.e. making

array **invariant**. Actually this solution is adopted by `ArrayList` of Java But Java solves the problem in a different way for arrays. Each assignment to an array element is checked at runtime for type correctness: is the type if the object being assigned compatible with the type of the array? The check is done against the type of the array itself rather than the declared type of the reference to the array that we are using. Consider the following example:

```
1 public class A {};
2 public class B extends A {};
3 public class C extends A {};
4 A[] as = new B[2];
5 as[0] = new A(); //compiles, but results in runtime error!
6
7 B[] bs = new B[2];
8 A[] as_for_bs = bs;
9 as_for_bs[0] = new C();//compiles, but results in runtime error!
```

8.2 Java exceptions