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

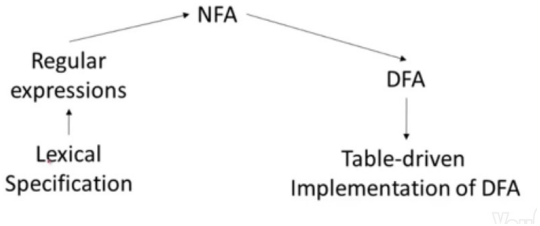


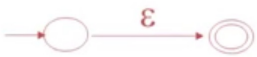
Figure 2.1: Pipeline of lexical analyser

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

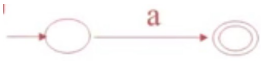
2.2.1 Regular expression into NFAs

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

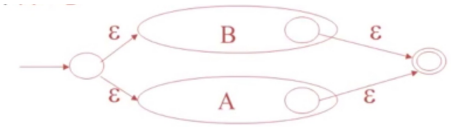
For the simplest ε regex:



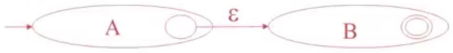
For regex with a single character a:



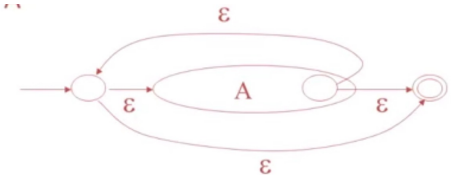
For union $A + B$:



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

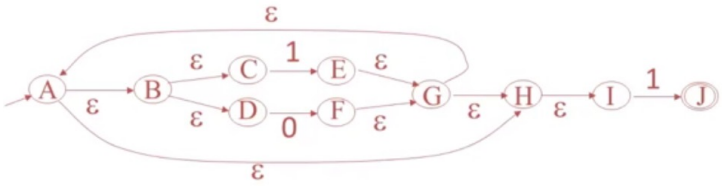


Figure 2.2: Idea of ε-closure

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

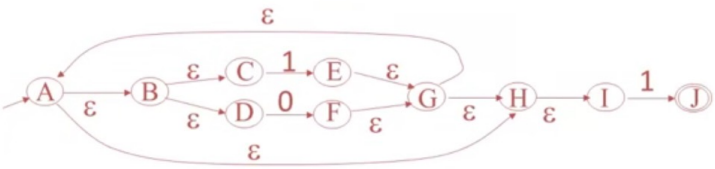
$$a(X) = \{y \in S \mid \exists x \in X \text{ s.t. } x \xrightarrow{a} y\}.$$

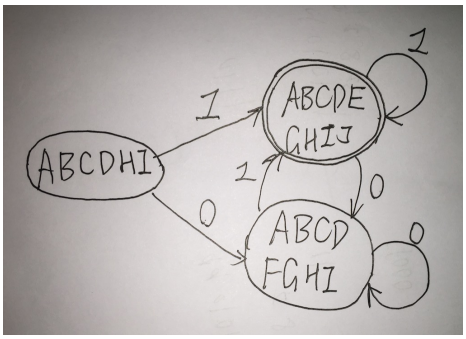
The NFA may be in many different states at any time. Suppose the NFA has N states, and it winds up in a subset of these states S. It is for sure that $|S| \leq N$. There exist totally $2^N - 1$ subsets. $2^N - 1$ is a big number, but still a finite one, which means that there exists a way to convert 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 $Y = \varepsilon$-$clos(a(X))$

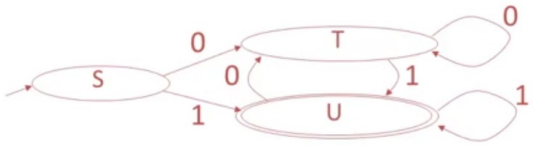
have gotten into with the same input. Below is an example demonstrating this conversion method.





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. Equivantly, 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 decribes 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 cloest "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: lexcial 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 funtion. 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 compilates 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.

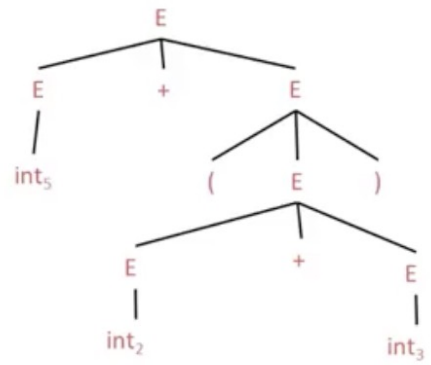


Figure 3.1: Parse tree of 5 + (2 + 3)

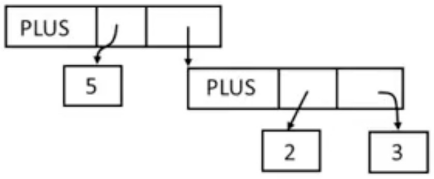


Figure 3.2: AST of 5 + (2 + 3)

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 algoruthm just presented is easy to be implemented by hand, but it is not general. Specificly 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,

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 == <  >
```

the string 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 grammer, 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 predixes

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

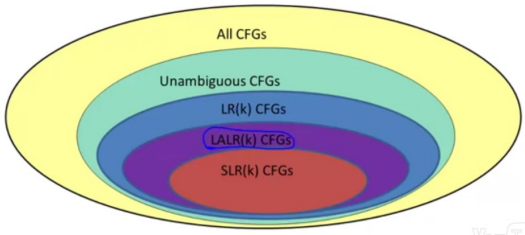


Figure 3.3: Relationship of different CFGs

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·) decribes 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.

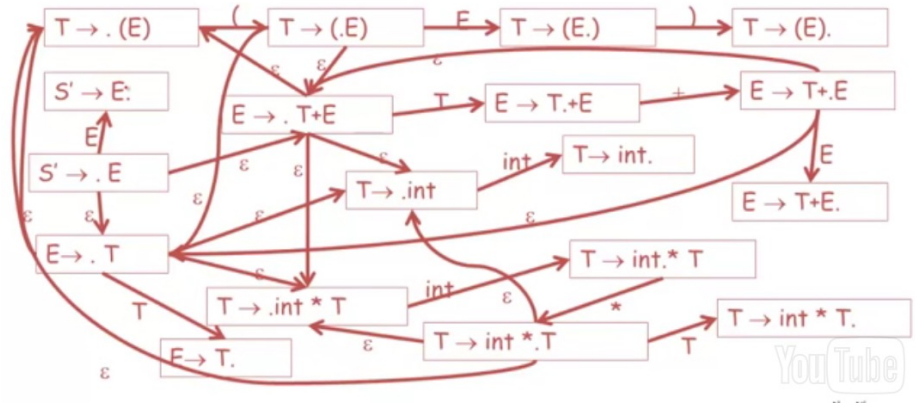


Figure 3.4: NFA of grammar (3.2)

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 trasition 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 confilicts 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 predixes 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 defintion 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 (postprocessing)

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 postprocessing 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 implemenation 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

$$(\mathsf{e_1} : \mathsf{Int} \wedge \mathsf{e_2} : \mathsf{Int}) \Rightarrow \mathsf{e_1} + \mathsf{e_2} : \mathsf{Int}$$

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.

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

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

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

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

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

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

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

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, $\frac{\vdash \mathsf{i}:\mathsf{Int}}{\vdash \mathsf{i}:\mathsf{Object}}$ 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.

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

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

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

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

And we can now write some new rules:

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

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:

$$\frac{\begin{array}{c}\mathsf{O} \vdash \mathsf{e}_0 : \mathsf{T} \\ \mathsf{O}[\mathsf{T}/\mathsf{x}] \vdash \mathsf{e}_1 : \mathsf{T}_1\end{array}}{\mathsf{O} \vdash \mathsf{let\ x} : \mathsf{T} \leftarrow \mathsf{e}_0\ \mathsf{in}\ \mathsf{e}_1 : \mathsf{T}_1} \tag{4.1}$$

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

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

Similarly, the rule of assignment can be written as

$$\frac{\begin{array}{c}\mathsf{O}(\mathsf{x}) = \mathsf{T}_0 \\ \mathsf{O} \vdash \mathsf{e}_1 : \mathsf{T}_1 \\ \mathsf{T}_1 \leq \mathsf{T}_0\end{array}}{\mathsf{O} \vdash \mathsf{x} \leftarrow \mathsf{e}_1 : \mathsf{T}_1}$$

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.

$$\frac{\begin{array}{c}\mathsf{O}_\mathsf{C}(\mathsf{x}) = \mathsf{T}_0 \\ \mathsf{O}_\mathsf{C} \vdash \mathsf{e}_1 : \mathsf{T}_1 \\ \mathsf{T}_1 \leq \mathsf{T}_0\end{array}}{\mathsf{O}_\mathsf{C} \vdash \mathsf{x} : \mathsf{T}_0 \leftarrow \mathsf{e}_1 : \mathsf{T}_0}$$

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 rumtime. 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:

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

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

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

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 differnt from O, which means that a method and an object can share the same name. A rule

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

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:

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

Similarly, the rule of static dispatch is

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

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

$$\frac{O, M, C \vdash e_1 : \text{Int} \quad O, M, C \vdash e_2 : \text{Int}}{O, M, C \vdash e_1 + e_2 : \text{Int}} \tag{4.2}$$

4.3.6 Implementation

Cool type checking can be implemented in a single travesal 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

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

can be implemented as

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

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 rumtime 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 overriden 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:

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

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

```
class Stock inherits Count {
  name : String;
};
class Main {
  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 funtionality 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:

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

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

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

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 ($\mathsf{T}_1 \leq \mathsf{T}_2$) and the least upper bound ($\mathrm{lub}((\mathsf{T}_1, \mathsf{T}_2))$). They are extended to handle SELF_TYPE as follows.

For subtype relationship:

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

For least upper bound:

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

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 $m@T(E_1 \dots E_n)$, 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 $T_0 \leq$ 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 $O, M, C \vdash e : T$ 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:

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

In case the return type is SELF_TYPE:

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

Similarly for static dispatch:

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

the rule becomes

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

Note that we are returning type T_0 rather than C, because the type of self (i.e. 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:

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

$$\frac{}{\mathsf{O}, \mathsf{M}, \mathsf{C} \vdash \mathsf{new SELF\_TYPE} : \mathsf{SELF\_TYPE_C}}$$

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 $\mathsf{No\_type} \leq \mathsf{C}$ 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.

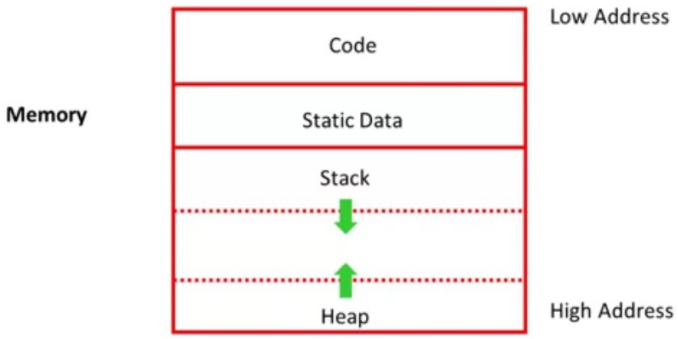


Figure 5.1: Partition of memory

Chapter 6

Code generation

6.1 Stack machines

A stack machine uses only a stack as storage. When executing an instruction $\mathsf{r} = \mathsf{F}(\mathsf{a}_1, \ldots, \mathsf{a}_n)$, 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 computer 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 add $\mathsf{r}_1, \mathsf{r}_2, \mathsf{r}_3$, 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 resgister 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 $\mathsf{op}(\mathsf{e}_1, \ldots, \mathsf{e}_n)$. 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

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.

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

P → D; P | D

D → def id(ARGS) = E;

ARGS → id, ARGS | id

E → int | id | if $E_1 = E_2$ then E_3 else E_4 | $E_1 + E_2$| $E_1 - E_2$| id($E_1, \ldots, E_n$)

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)
```

For each expression e, we want to generate MIPS code that **computers 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 condisering 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 Subtration

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, an AR will contain the caller's frame pointer, the actual parameters and the return address. Consider a call to $f(x, y)$. The AR is

```
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
```

```
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
```

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