Computer Science 3

Compiling Techniques

Samson Abramsky

samson@dcs

Why study compilers?

- To learn how to use them well.
- To learn how to write them.
- To illuminate programming language design.
- As an example of a large software system.
- To motivate interest in formal language theory.

Course bias

- *Not* a theory course.
- *Not* a hardware/assembler course.
- *Not* a superficial survey of techniques.
- Concentrate on important ideas.
- Examine the ideas in implementations.

Course text

- There are two editions of the course text; and there are three versions of each of them!
- You can choose either
  - "Modern Compiler Implementation", by Andrew Appel, Cambridge University Press, 1998.
    Price £27.95.
  - *or* "Modern Compiler Implementation: Basic Techniques", by Andrew Appel, Cambridge University Press, 1997.
    Price £19.95.
- There are versions for the languages C, ML and Java.
- We study Part One of the book. This is common to both editions.

Coursework

- Choose an implementation language (C, ML, Java, C++, ...)
- Obtain the (partial) source code and the *Tiger'99 Reference Manual* from the CS3 Compiling Techniques Web page.
- Working in groups or otherwise, develop a compiler for Tiger'99 as described in the reference manual.
- Groups can use the Compiling Techniques group accounts (with the user ids `ct00`, `ct01`, `ct02`, ...). Mail support@dcs telling them the names of the people in your group.
- Submit source, SUN Solaris executable and documentation (a README file)
- **Deadline:** End of Week 9 of this term.

Tiger and Tiger'99

- Tiger '99 is a dialect of the Tiger language described in Andrew Appel's textbook "Modern Compiler Implementation".
- Although Tiger '99 has many features in common with Tiger it adds some new syntax and concepts while taking away others. Thus it is neither a subset nor a superset of Tiger.
- However, the skills which are needed to know how to compile the language are those which can be learned from careful study of Appel's textbook.

2. Lexical analysis, LEX; basic parsing.
3. Predictive parsing; concepts of LR parsing.
4. YACC; Abstract syntax, semantic actions, parse trees.
5. Semantic analysis, tables, environments, type-checking.
6. Activation records, stack frames, variables escaping.
7. Intermediate representations, basic blocks and traces.
8. Instruction selection, tree patterns and tiling.
9. Liveness analysis, control flow and data flow.

Lexical analysis

- The first phase of compilation.
- White space and comments are removed.
- The input is converted into a sequence of lexical tokens.
- A token can then be treated as a unit of the grammar.

Lexical tokens

Examples:
`foo` (ID), `73` (INT), `66.1` (REAL), `if` (IF), `!=` (NEQ),
`(` (LPAREN), `)` (RPAREN)

Non-examples:
`/* huh? */` (comment),
`#define NUM 5` (preprocessor directive),
`NUM` (macro)


```
{if (!strncmp (s, "0.0", 3))
    return 0.;
}
```

⇓

VOID ID(`match0`) LPAREN CHAR STAR
ID(`s`) RPAREN LBRACE IF LPAREN
BANG ID(`strncmp`) LPAREN ID(`s`)
COMMA STRING(`0.0`) COMMA NUM(`3`)
RPAREN RPAREN RETURN REAL(`0.0`)
SEMI RBRACE EOF

LEX disambiguation rules

Longest match:
The longest initial substring of the input that can match any regular expression is taken as the next token.

Rule priority:
For a *particular* longest initial substring, the first regular expression that can match determines its token type.

This means that the order of writing down the regular expression rules has significance.

`if0` → ID(`if0`) *not* IF NUM(`0`)
`if` → IF *not* ID(`if`)

Context-free grammars

A *language* is a set of *strings*; each string is a finite sequence of *symbols* taken from a finite *alphabet*. A context-free grammar describes a language. A grammar has a set of *productions* of the form

$$symbol \rightarrow symbol \cdots symbol$$

where there are zero or more symbols on the right-hand side. Each symbol is either *terminal*, meaning that it is a token from the alphabet of strings in the language, or *non-terminal*, meaning that it appears on the left-hand side of a production. No terminal symbol can ever appear on the left-hand side of a production and there is only one non-terminal there (together these justify the name **context-free**). Finally, one of the productions is distinguished as the *start symbol* of the grammar.

A grammar for straight-line programs

1. $S \rightarrow S\ ;\ S$	(compound statements)
2. $S \rightarrow \text{id} := E$	(assignment statements)
3. $S \rightarrow \text{print}(L)$	(print statements)
4. $E \rightarrow \text{id}$	(identifier usage)
5. $E \rightarrow \text{num}$	(numerical values)
6. $E \rightarrow E + E$	(addition)
7. $E \rightarrow (S\ ,\ E)$	(comma expressions)
8. $L \rightarrow E$	(singleton lists)
9. $L \rightarrow L\ ,\ E$	(non-singleton lists)

Examples: `a := 7 ; b := c + (d := 5 + 6, d)`

Non-examples: `a := 7 ;` `b := (d, d)` `print ()`

A derivation

$\underline{S}$
$S\ ;\ \underline{S}$
$\underline{S}\ ;\ \text{id} := E$
$\text{id} := \underline{E}\ ;\ \text{id} := E$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := E + \underline{E}$
$\text{id} := \text{num}\ ;\ \text{id} := \underline{E} + (S\ ,\ E)$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\underline{S}\ ,\ E)$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E}\ ,\ E)$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := E + E\ ,\ \underline{E})$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \underline{E} + E\ ,\ \text{id})$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \underline{E}\ ,\ \text{id})$
$\text{id} := \text{num}\ ;\ \text{id} := \text{id} + (\text{id} := \text{num} + \text{num}\ ,\ \text{id})$

Ambiguity

Consider the following straight-line program.

```
a := b + b + c
```

- Does the right-hand side denote $(b + b) + c$ or $b + (b + c)$?
- Does it matter?

ence and *associativity* (left or right). The following grammar describes a language with *expressions* made up of *terms* and *factors*.

1. $E \to E + T$
2. $E \to E - T$
3. $E \to T$
4. $T \to T * F$
5. $T \to T \,/\, F$
6. $T \to F$
7. $F \to \text{id}$
8. $F \to \text{num}$
9. $F \to (\ E\ )$

Parsing by recursive descent

Consider the following grammar.

1. $S \to \text{if } E \text{ then } S \text{ else } S$
2. $S \to \text{begin } S\ L$
3. $S \to \text{print } E$
4. $L \to \text{end}$
5. $L \to ;\ S\ L$
6. $E \to \text{num} = \text{num}$

This grammar can be parsed using a simple algorithm which is known as *recursive descent*. A recursive descent parser has one function for each non-terminal and one clause for each production.


```
extern enum token getToken(void);

enum token tok;
void advance() {tok=getToken();}
void eat(enum token t) {if (tok==t) advance(); else error();}

void S(void) {switch(tok) {
        case IF:    eat(IF); E(); eat(THEN); S();
                                  eat(ELSE); S(); break;
        case BEGIN: eat(BEGIN); S(); L(); break;
        case PRINT: eat(PRINT); E(); break;
        default:    error();
       }}
void L(void) {switch(tok) {
        case END:  eat(END); break;
        case SEMI: eat(SEMI); S(); L(); break;
        default:   error();
       }}
void E(void) { eat(NUM); eat(EQ); eat(NUM); }
```

When recursive descent fails

$S \to E\ \$$	$E \to E + T$	$T \to T * F$	$F \to \text{id}$
	$E \to E - T$	$T \to T \,/\, F$	$F \to \text{num}$
	$E \to T$	$T \to F$	$F \to (\ E\ )$

```
void S(void) {  E(); eat(EOF);  }
void E(void) {switch(tok) {
        case ?: E(); eat(PLUS); T(); break;
        case ?: E(); eat(MINUS); T(); break;
        case ?: T(); break;
        default:   error();
       }}
void T(void) {switch(tok) {
        case ?: T(); eat(TIMES); F(); break;
        case ?: T(); eat(DIV); F(); break;
        case ?: F(); break;
        default:   error();
       }}
```


First and follow sets

Grammars consist of *terminals* and *non-terminals*. With respect to a particular grammar, given a string γ of terminals and non-terminals,

- nullable(X) is true if X can derive the empty string
- FIRST(γ) is the set of terminals that can begin strings derived from γ
- FOLLOW(X) is the set of terminals that can immediately follow X. That is, $t \in \text{FOLLOW}(X)$ if there is any derivation containing Xt. This can occur if the derivation contains $XYZt$ where Y and Z both derive ϵ.

Computing FIRST, FOLLOW and nullable

Initialise FIRST and FOLLOW to all empty sets
Initialise nullable to all *false*.
for each terminal symbol Z
  $\text{FIRST}[Z] \leftarrow \{Z\}$
repeat
  for each production $X \leftarrow Y_1 Y_2 \cdots Y_k$
    for each i from 1 to k, each j from $i+1$ to k
      if all the Y_i are nullable
        then $\text{nullable}[X] \leftarrow \mathit{true}$
      if $Y_1 \cdots Y_{i-1}$ are all nullable
        then $\text{FIRST}[X] \leftarrow \text{FIRST}[X] \cup \text{FIRST}[Y_i]$
      if $Y_{i+1} \cdots Y_k$ are all nullable
        then $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FOLLOW}[X]$
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
        then $\text{FOLLOW}[Y_i] \leftarrow \text{FOLLOW}[Y_i] \cup \text{FIRST}[Y_j]$
until FIRST, FOLLOW and nullable did not change in this iteration

Constructing a predictive parser

- The information which we need can be coded as a two-dimensional table of productions, indexed by non-terminals and terminals. This is a *predictive parsing table*.
- To construct the table, enter the production $X \to \gamma$ in column T of row X for each $T \in \text{FIRST}(\gamma)$. Also, if γ is nullable, enter the production in column T of row X for each $T \in \text{FOLLOW}(X)$.
- An ambiguous grammar will always lead to some locations in the table having more than one production.
- A grammar whose predictive parsing table has at most one production in each location is called LL(1). This stands for *Left-to-right parse, leftmost derivation, 1-symbol lookahead*.

Detecting ambiguity with a parsing table

$Z \to d$	$Y \to$	$X \to Y$
$Z \to XYZ$	$Y \to c$	$X \to a$

	nullable	FIRST	FOLLOW
X	*true*	$\{a, c\}$	$\{a, c, d\}$
Y	*true*	$\{c\}$	$\{a, c, d\}$
Z	*false*	$\{a, c, d\}$	

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

productions cannot be LL(1).

$$
\begin{array}{llll}
S \to E\ \$ & E \to E + T & T \to T * F & F \to \text{id} \\
 & E \to E - T & T \to T\ /\ F & F \to \text{num} \\
 & E \to T & T \to F & F \to (\ E\ )
\end{array}
$$

We can transform this grammar as shown below.

$$
\begin{array}{llll}
S \to E\ \$ & E \to T\ E' & T \to F\ T' & F \to \text{id} \\
 & & & F \to \text{num} \\
 & E' \to +\ T\ E' & T' \to *\ F\ T' & F \to (\ E\ ) \\
 & E' \to -\ T\ E' & T' \to /\ F\ T' & \\
 & E' \to & T' \to &
\end{array}
$$

Left factoring a grammar

Left recursion interferes with predictive parsing. A similar problem occurs when two productions for the same non-terminal start with the same symbols. For example:

$$
\begin{array}{lll}
S & \to & \text{if } E \text{ then } S \text{ else } S \\
S & \to & \text{if } E \text{ then } S
\end{array}
$$

In such a case we can *left-factor* the grammar to account for the optional else statement.

$$
\begin{array}{lll}
S & \to & \text{if } E \text{ then } S\ X \\
X & \to & \\
X & \to & \text{else } S
\end{array}
$$


```
        case ID:
        case NUM:
        case LPAREN: F(); Tprime(); break;
        default: printf("expected id, num or (")
    }}

int Tprime_follow [] = {PLUS, MINUS, RPAREN, EOF, -1};

void Tprime(void) { switch (tok) {
        case PLUS:   break;
        case MINUS:  break;
        case TIMES:  eat(TIMES); F(); Tprime(); break;
        case DIV:    eat(DIV); F(); Tprime(); break;
        case RPAREN: break;
        case EOF:    break;
        default:     printf("expected +, -, *, /, ) or EOF");
                     skipto(Tprime_follow);
    }}
```

LR Parsing

- The weakness of LL(k) parsing techniques is that they must *predict* which production to use, having seen only the first k tokens of the right-hand side.
- LR(k) parsing postpones the decision until it has seen the entire right-hand side of the production (and k more tokens beyond).
- The name LR(k) means *Left-to-right parse, rightmost derivation, k-token lookahead*.
- The parser has a *stack* and an *input*. The first k tokens of the input are the *lookahead*. The parser performs two kinds of actions:

  Shift: move the first input token to the top of the stack;

  Reduce: Choose a grammar rule $X \to A\ B\ C$; pop C, B, A off the stack, push X onto the stack.

- Shifting the end-of-file marker is called *accepting*.

LR transition tables

- LR parsers decide when to shift and when to reduce by consulting a *transition table* which is a two-dimensional array with rows indexed by state number and columns indexed by the terminals and non-terminals of the grammar.
- Elements in the table are labeled with four kinds of actions.

Shift(n)	Shift into state n;
Goto(n)	Goto state n;
Reduce(k)	Reduce by rule k; and
Accept	Acceptable input.

  Errors are denoted by blank entries in the table.
- An LR parser could scan the stack for each token to determine which state it is in but a useful optimisation is to record the state reached for each stack element. The stack now contains pairs of a terminal and a state or a non-terminal and a state.

An LR parsing algorithm

- Look up the top stack state and input symbol to get action;
- If action is:

  Shift(n): Advance input one token; push n onto stack.

  Reduce(k):

  - Pop stack as many times as the number of symbols on the right-hand side of rule k;
  - Let X be the left-hand-side symbol of rule k;
  - In the state now on top of the stack, look up X to get "**Goto**(n)";
  - Push n on top of stack.

  Accept: Stop parsing, report success.

  Error: Stop parsing, report failure.

Transition table generation

- An LR(k) parser uses the contents of its stack and the next k tokens of the input to decide which action to take. In practice the value of k is 1
  - because transition tables for $k > 1$ would be huge; and
  - most programming languages have LR(1) grammars.
- To construct a transition table for an LR(0) parser we need to explore the possible *states* of the parser. For this, we use the notion of an *item* (such as an LR(0) item or an LR(1) item). An item is a grammar rule, combined with a dot which indicates a position in its right hand side. For example:

  $$S' \to .S\$$$

  A state is just a set of items.

An example of LR(0) items and states

$$
\begin{array}{llll}
S' \to S\$ & S \to (L) & L \to S \\
 & S \to x & L \to L, S
\end{array}
$$

We build up states starting from the start symbol S'. There are nine states in total and the first six are these.

1
$$
\begin{array}{lll}
S' & \to & .S\$ \\
S & \to & .x \\
S & \to & .(L)
\end{array}
$$

3
$$
\begin{array}{lll}
S & \to & (.L) \\
L & \to & .L, S \\
L & \to & .S \\
S & \to & .(L) \\
S & \to & .x
\end{array}
$$

5
$$
\begin{array}{lll}
S & \to & (L.) \\
L & \to & L., S
\end{array}
$$

2
$$S \to x.$$

4
$$S' \to S.\$$$

6
$$S \to (L).$$

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

/ Moving past a non-terminal X */*
Goto(I, X) =
  set J to the empty set
  for any item $A \to \alpha.X\beta$ in I
    add $A \to \alpha X.\beta$ to J
  return Closure(J)

LR(0) parser construction

/ Compute the states traversed (T) and edge set (E) */*
Initialize T to {**Closure**($\{S' \to .S\$\}$)}
Initialize E to the empty set.
repeat
  for each state I in T
    for each item $A \to \alpha.X\beta$ in I
      let J be **Goto**(I, X)
        $T \leftarrow T \cup \{J\}$
        $E \leftarrow E \cup \{I \xrightarrow{X} J\}$
until E and T did not change in this iteration

/ Compute the set of LR(0) reduce actions (R) */*
$R \leftarrow \{\}$
for each state I in T
  for each item $A \to \alpha.$ in I
    $R \leftarrow R \cup \{(I, A \to \alpha)\}$

  - if X is a terminal, we put **Shift**(J) at position (I, X)
  - if X is a non-terminal, we put **Goto**(J) at position (I, X)
- For each state I containing $S' \to S.\$$ we put **Accept** at $(I, \$)$.
- For each state I containing $A \to \gamma.$ (production n with the dot at the end), we put a **Reduce**(n) action at (I, Y) for *every* token Y.

Parsers which are better than LR(0)

SLR (Simple LR) parsers

/ Compute the set of SLR reduce actions (R) */*
$R \leftarrow \{\}$
for each state I in T
  for each item $A \to \alpha.$ in I
    for each token X in FOLLOW(A)
      $R \leftarrow R \cup \{(I, X, A \to \alpha)\}$

LR(1) parsers

/ Items now include a one-symbol lookahead */*
$R \leftarrow \{\}$
for each state I in T
  for each item $(A \to \alpha., z)$ in I
    $R \leftarrow R \cup \{(I, z, A \to \alpha)\}$

Using parser generators

- The task of constructing LR(1) and LALR(1) parsing tables is simple enough to be automated.
- Many such parser generator tools exist (Yacc, Bison, ML-Yacc, CUP, JavaCC, ...)
- Grammar specifications are divided into sections (the following is for Yacc)

  parser declarations
  %%
  grammar rules
  %%
  programs

- The grammar rules are productions of the following form.

  `exp :  exp PLUS exp  { ` *semantic action* ` }`

An example grammar

1. $P \to L$ (programs)
2. $S \to$ id := id (assignment statements)
3. $S \to$ while id do S (while loops)
4. $S \to$ begin L end (compound statements)
5. $S \to$ if id then S (if-then statements)
6. $S \to$ if id then S else S (if-then-else statements)
7. $L \to S$ (singleton statement list)
8. $L \to L\ ;\ S$ (non-singleton statement list)

Implementation in Yacc/Bison

```
%{
int yylex(void);
void yyerror(char *s) { EM_error(EM_tokPos, "%s", s); }
%}
%token ID WHILE BEGIN END DO IF THEN ELSE SEMI ASSIGN
%start prog
%%

prog: stmlist

stm : ID ASSIGN ID
    | WHILE ID DO stm
    | BEGIN stmlist END
    | IF ID THEN stm
    | IF ID THEN stm ELSE stm

stmlist : stm
        | stmlist SEMI stm
```

Implementation in CUP

```
terminal ID, WHILE, BEGIN, END, DO, IF, THEN, ELSE, SEMI, ASSIGN;

non terminal prog, stm, stmlist;

start with prog;

prog ::= stmlist;

stm  ::= ID ASSIGN ID
      |  WHILE ID DO stm
      |  BEGIN stmlist END
      |  IF ID THEN stm
      |  IF ID THEN stm ELSE stm;

stmlist  ::= stm
          |  stmlist SEMI stm;
```

  - A shift-reduce conflict is a choice between shifting and reducing.
  - A reduce-reduce conflict is a choice of reducing by two different rules.
- Two conventions apply:
  - Resolve shift-reduce conflicts by shifting.
  - Resolve reduce-reduce conflicts by using the rule which appears earliest in the grammar.
- Causes of conflicts can be found by examining the *verbose description file*.

LR parsing of ambiguous grammars

Many programming languages have grammar rules such as

$S \rightarrow$ if E then S else S
$S \rightarrow$ if E then S

which allow programs such as

```
if a then if b then s1 else s2
```

Such a program could be understood in two ways

1. `if a then { if b then s1 else s2 }`
2. `if a then { if b then s1 } else s2`

The grammar leads to a shift/reduce conflict, resolved by shifting.

ambiguity with precedence directives.

$E \rightarrow \text{int} \quad E \rightarrow E + E \quad E \rightarrow E * E \quad E \rightarrow E - E \quad E \rightarrow -E$

```
%{ declarations of yylex and yyerror %}
%token INT PLUS MINUS TIMES UMINUS
%start exp

%left PLUS MINUS                                /* Left associativity. */
%left TIMES                      /* Left associativity, higher precedence. */
%left UMINUS                    /* Left associativity, highest precedence. */
%%

exp : INT
    | exp PLUS exp              /* By default, a rule has the precedence */
    | exp MINUS exp                 /* of its last (i.e. rightmost) token. */
    | exp TIMES exp
    | MINUS exp   %prec UMINUS        /* Explicitly setting precedence. */
```

Non-associative operators

- Some programming languages consider it unnecessarily confusing to allow some operators to be either left associative or right associative. Thus $a - b - c$ might be considered to be grammatically ill-formed.
- The Pascal programming language disallowed the `<=` operator on Booleans from being associative. The ordering which Pascal placed on Booleans was `false < true`. Because the operator is non-associative, the following Boolean expression is illegal.

  ```
  false <= false <= false
  ```

- Non-associativity can be specified with the `%nonassoc` directive.

Syntax versus semantics

The following grammar describes a language with assignment and arithmetic (AE) and Boolean expressions (BE).

$S \rightarrow \text{id} := AE$	$AE \rightarrow AE + AE$	$BE \rightarrow BE$ or BE
$S \rightarrow \text{id} := BE$	$AE \rightarrow \text{id}$	$BE \rightarrow BE$ and BE
		$BE \rightarrow AE = AE$
		$BE \rightarrow \text{id}$

The problem with a grammar such as this is that when the parser sees an identifier it does not know whether it is an arithmetic variable or a Boolean variable. The solution is to defer this kind of analysis to the "semantic" phase of the compiler. A more appropriate grammar is:

$S \rightarrow \text{id} := E$	$E \rightarrow E$ and E	$E \rightarrow E = E$
$E \rightarrow \text{id}$	$E \rightarrow E$ or E	$E \rightarrow E + E$

Semantic actions

- The *semantic actions* of a parser operate on the phrases which have been recognised as being produced by the productions of the grammar.
- Each terminal and non-terminal can be associated with its own type of *semantic value*, which must match the type that the lexer returns with that token.
- For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated with A. It can build this from the values associated with B, C and D.

A simple interpreter in Yacc/Bison

```
%{ declarations of yylex and yyerror %}
%union {int num; string id;}
%token <num> INT
%token <id> ID
%type <num> exp
%start exp

%left PLUS MINUS
%left TIMES
%left UMINUS
%%

exp : INT             {$$ = $1;}
    | exp PLUS exp    {$$ = $1 + $3;}
    | exp MINUS exp   {$$ = $1 - $3;}
    | exp TIMES exp   {$$ = $1 * $3;}
    | MINUS exp   %prec UMINUS  {$$ = -$2;}
```

Parse trees

- Compilers do not evaluate programs as interpreters do. A compiler subjects the program text to semantic analysis and then translates into a lower-level language. A *parse tree* is used to store the program text for analysis and translation purposes.
- A parse tree for the *concrete syntax* of the language would have exactly one leaf for each token of the input and one internal node for each grammar rule reduced during the parse.
- A parse tree for the *abstract syntax* retains the important content from the concrete parse tree but dispenses with the punctuation tokens which were needed to ensure the correct parse.
- The resulting language can be described by a smaller, simpler grammar.

2. $S \to \text{id} := E$ (assignment statements)
3. $S \to \text{print } L$ (print statements)
4. $E \to \text{id}$ (identifier usage)
5. $E \to \text{num}$ (numerical values)
6. $E \to E\ B\ E$ (binary operator use)
7. $E \to S\ ,\ E$ (comma expressions)
8. $L \to E$ (singleton lists)
9. $L \to L\ E$ (non-singleton lists)
10. $B \to +$ (addition)
11. $B \to -$ (subtraction)
12. $B \to *$ (multiplication)
13. $B \to /$ (division)

Error reporting and positional information

- In a one-pass parser, lexical analysis, parsing and semantic analysis are done simultaneously. If there is a type-checking error then the current position of lexer can be used in diagnostic error messages.
- A modern compiler makes many passes through the program so the parse tree must be sprinkled with `pos` fields. These indicate the position, within the original source file, of the characters from which the abstract syntax structures were derived.
- Ideally the parser should maintain a *position stack* along with the *semantic value* stack. Bison provides this; Yacc does not.

```
struct {
  int first_line, last_line;
  int first_column, last_column;
};
```



```
(a := 5; a+1)
```

translates into abstract syntax as shown below.

$A\_SeqExp(2,$
$\quad A\_Exp\_List(A\_AssignExp(4,$
$\quad\quad A\_SimpleVar(2, S\_Symbol(\texttt{"a"})),$
$\quad\quad A\_IntExp(7, \texttt{5})),$
$\quad A\_ExpList(A\_OpExp(11,$
$\quad\quad A\_plusOp,$
$\quad\quad A\_VarExp(A\_SimpleVar(10, S\_Symbol(\texttt{"a"}))),$
$\quad\quad A\_IntExp(12, \texttt{1})),$
$\quad NULL)))$

Derived forms/defined notation

- The abstract syntax of a language can be further compressed by defining some constructs of the language in terms of others.
  - e_1 `&` e_2 is translated to `if` e_1 `then` e_2 `else 0`
  - e_1 `|` e_2 is translated to `if` e_1 `then 1 else` e_2
  - $-i$ is translated as `(0 -` i`)`
- Derived forms lead to simplifications in the semantic analysis process but they make it harder to generate good diagnostic error messages for semantic errors.

Symbol tables

- The *semantic analysis* phase of a compiler is characterised by the maintenance of *symbol tables* (also called *environments*) which map identifiers used in the source program to their types and locations.
- As declarations of types, variables and functions are processed by the compiler the identifiers used are bound to "meanings" in the symbol table. An environment is a set of *bindings*.
- When *uses* (non-defining occurrences) of the identifiers are found, the symbol table is consulted to check the suitability of the use.
- Each local variable in a program has a *scope* in which it is visible. As the semantic analysis reaches the end of each scope, the bindings local to that scope are discarded.

Environments in Tiger

Consider the introduction and elimination of environments in the small Tiger program given below. This program is being compiled in the environment σ_0.

`let var a := 10`	$\sigma_1 = \sigma_0 + \{\ \texttt{a} \mapsto \texttt{int}\ \}$
`  var b := 12`	$\sigma_2 = \sigma_1 + \{\ \texttt{b} \mapsto \texttt{int}\ \}$
`in print_int(b);`	σ_2
`  let var j := a+b`	$\sigma_3 = \sigma_2 + \{\ \texttt{j} \mapsto \texttt{int}\ \}$
`  var a := "hello"`	$\sigma_4 = \sigma_3 + \{\ \texttt{a} \mapsto \texttt{string}\ \}$
`in print(a); print_int(j)`	σ_4
`  end;`	σ_2
`  print_int(a)`	σ_2
`end`	σ_0

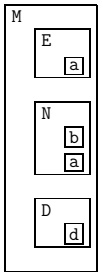
Multiple environments in ML and Java

```
structure M = struct
  structure E = struct
    val a = 5
  end
  structure N = struct
    val b = 10
    val a = E.a + b
  end
  structure D = struct
    val d = E.a + N.a
  end
end
```

```
package M;
class E {
  static int a = 5;
}
class N {
  static int b = 10;
  static int a = E.a + b;
}
class D {
  static int d = E.a + N.a;
}
```

Multiple symbol tables

With many environments active at once, we must associate names with environments. This leads to the following collection of symbol tables.



$$= \begin{cases} \sigma_1 &= \{\ \texttt{a} \mapsto \texttt{int}\ \} \\ \sigma_2 &= \{\ \texttt{E} \mapsto \sigma_1\ \} \\ \sigma_3 &= \{\ \texttt{b} \mapsto \texttt{int}, \texttt{a} \mapsto \texttt{int}\ \} \\ \sigma_4 &= \{\ \texttt{N} \mapsto \sigma_3\ \} \\ \sigma_5 &= \{\ \texttt{d} \mapsto \texttt{int}\ \} \\ \sigma_6 &= \{\ \texttt{D} \mapsto \sigma_5\ \} \\ \sigma_7 &= \sigma_2 + \sigma_4 + \sigma_6 \\ \sigma_8 &= \sigma_0 + \{\ \texttt{M} \mapsto \sigma_7\ \} \end{cases}$$

can convert each string to an (integer or pointer) symbol.

  - Comparing two symbols for equality is very fast.
  - Extracting an integer hash key is very fast (we just use the symbol pointer itself)
  - Comparing symbols for "greater than" is very fast (in case we wish to build binary search trees.)

- To implement the removal of entries in the table upon exiting a scope the symbol table must also maintain an auxiliary stack, showing the order in which the symbols were entered into the table. This can be integrated with the symbol table itself by adding an extra field in entries to point to the previously entered symbol.

Types in Tiger

- The primitive types of Tiger are `int` and `string`; all types are either primitive or constructed using records and arrays.
- Record types carry additional information: the names and types of the fields.
- Arrays work just like records with the modification that all of the elements of the array must have the same type.
- For array and record types there is another implicit piece of information which must be carried by the object: the address of the object itself. Thus two syntactically identical declarations can give rise to different types. We can use pointer equality to see if two record types are truly the same.


```
let type a = { x: int, y: int }
    type b = { x: int, y: int }
    var i : a := ···
    var j : b := ···
 in i := j
end
```

If we were to change the language to allow usage such as this we would have to examine record types field by field, recursively, when testing for equality. This would slow down the type-checking phase of the compilation considerably.

Record types in Tiger

The following program is legal.

```
let type a = { x: int, y: int }
    type c = a
    var i : a := ···
    var j : c := ···
 in i := j
end
```

It is the *type expression* which causes a new type to be made; not the *type declaration*.

Special cases

- In Tiger the expression *nil* belongs to any record type. We handle this by inventing a special 'nil' type.
- There are expressions in Tiger which return no value (such as the assignment expression) so we invent a 'void' type.
- When processing mutually recursive types we need a place holder for types whose name we know but whose definition we have not yet seen. We enter a null pointer for the type name to come and then later replace it.

Environments

Tiger maintains separate name spaces for types and values. Thus we will have a *type environment* and a *value environment*. The following progam demonstrates that one environment will not be enough.

```
let type a = int
    var a : a := 5
    var b : a := a
 in b+a
end
```

The symbol `a` denotes a type in contexts where a type is expected and a variable in contexts where a variable is expected. At no point are we ever unsure of the meaning from the context.

Environment entries

- For a type identifier we need to remember the type that it denotes. The type environment is initialised to include definitions for `int` and `string`.
- For each value identifier we need to know whether it is a variable or a function. If it is a variable we need to know its type. If it is a function we need to know its parameter and result types. The variable environment is initialised to include definitions of predefined functions.
- As types, variables and functions are declared, the type-checker augments the environments. The environments are consulted for each identifier found when processing expressions.

Type-checking expressions

Type-checking is the first part of the translation process which we consider. Our semantic analysis module performs type-checking of the

abstract syntax of the language. It contains these functions.

```
struct expty transVar(S_table venv, S_table tenv, A_var v);
struct expty transExp(S_table venv, S_table tenv, A_exp a);
void         transDec(S_table venv, S_table tenv, A_dec d);
struct Ty_ty transTy (              S_table tenv, A_ty a);
```

The `expty` type contains a translated expression and its Tiger-language type.

```
struct expty {Tr_exp exp; Ty_ty ty;};

struct expty expTy(Tr_exp exp, Ty_ty ty) {
  struct expty e; e.exp=exp; e.ty=ty; return e;
}
```

Type-checking arithmetic expressions (in ML)

```
fun transExp(venv, tenv,
             Absyn.OpExp{left,oper=Absyn.PlusOp,right,pos}) =
    let val {exp=_, ty=tyleft}  = transExp(venv, tenv, left)
        val {exp=_, ty=tyright} = transExp(venv, tenv, right)
     in case tyleft  of Types.INT => ()
                      | _ => error pos "integer required";
        case tyright of Types.INT => ()
                      | _ => error pos "integer required";
        {exp=(), ty=Types.INT}
    end
```

```
  switch (a->kind) {
      ⋮
    case A_opExp: {
      A_oper oper = a->u.op.oper;
      struct expty left = transExp(venv, tenv, a->u.op.left);
      struct expty right= transExp(venv, tenv, a->u.op.right);
      if (oper==A_plusOp) {
        if (left.ty.kind != Ty_int)
          EM_error(a->u.op.left->pos, "integer required");
        if (right.ty.kind != Ty_int)
          EM_error(a->u.op.right->pos, "integer required");
        return expTy(NULL, Ty_int());
      }
      ⋮
    }
  }
}
```

Type-checking variables (in Java)

```
ExpTy transVar(Absyn.SimpleVar v) {
  Entry x = (Entry)env.venv.get(v.name);
  if (x instanceof VarEntry) {
     /* found the variable in the symbol table */
     VarEntry ent = (VarEntry)x;
     return new ExpTy (null, ent.ty);
  }
  else {
     /* failed to find the variable */
     error (v.pos, "undefined variable");
     return new ExpTy(null, INT);
  }
}
```



```
switch (v->kind) {
  case A_simpleVar: {
    E_enventry x = S_look(venv,v->u.simple);
    if (x && x->kind==E_varEntry)
      /* found the variable in the symbol table */
      return expTy(NULL, actual_ty(x->u.var.ty));
    else
      /* failed to find the variable */
      { EM_error(v->pos, "undefined variable %s",
                         S_name(v->u.simple));
       return expTy(NULL, Ty_int());}
    }
  case A_fieldVar:
    ⋮
  }
}
```

Type-checking let-expressions

```
struct expty transExp(S_table venv, S_table tenv, A_exp a) {
  switch(a->kind) {
      ⋮
    case A_letExp: {
      /* let d in exp end */
      struct expty exp;
      A_declist d;
      S_beginScope(venv);
      S_beginScope(tenv);
      /* process the declarations in the list */
      for (d = a->u.let.decs; d; d = d->tail)
        transDec(venv, tenv, d->head);
      /* translate the expression in the body */
      exp = transExp(venv, tenv, a->u.let.body);
      S_endScope(tenv);
      S_endScope(venv);
      return exp;
    }
      ⋮
}
```


Type-checking variable declarations

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  switch(d->kind) {
    case A_varDec: {
      /* var id := exp */
      struct expty e = transExp(venv,tenv,d->u.var.init);
      S_enter(venv, d->u.var.var, E_VarEntry(e.ty));
    }
      ⋮
  }
}
```

If a type constraint is present it would also be necessary to check that the constraint and the expression are compatible. Initialisations to `nil` must be accompanied by a type constraint which is of a record type.

Type-checking function declarations

```
void transDec(S_table venv, S_table tenv, A_dec d) {
  switch(d->kind) {
    ⋮
    case A_functionDec: {
      A_fundec f = d->u.function->head;
      Ty_ty resultTy = tylook(tenv,f->result,f->pos);
      Ty_tyList formalTys = makeFormalTyList(tenv,f->params);
      /* for non-recursive functions only */
      S_enter(venv,f->name,E_FunEntry(formalTys,resultTy));
      S_beginScope(venv);
      { A_fieldList l; Ty_tyList t;
         for(l=f->params, t=formalTys; l; l=l->tail, t=t->tail)
           S_enter(venv,l->head->name,E_VarEntry(t->head));
      }
      /* should check compatibility with result type */
      transExp(venv, tenv, d->u.function->body);
      S_endScope(venv);
      break;
    }
```


Recursive declarations

- When type-checking a collection of mutually recursive declarations the compiler must make two passes through the declarations.
- In the first pass the aim is to collect all of the *headers* of the declarations.
  - The header of a type is its identifier.
  - The header of a function is its name, formal parameter list and return type.
- In the second pass the *bodies* of the declarations are processed in the environment extended with the header information.

Local variables

The following function has a parameter `x` and a local variable `y`.

```
function f(x : int) : int =
  let var y := x+x
   in if y < 10
        then f(y)
          else y-1
  end
```

When the function `f` is called a new instantiation of `x` is created and initialised by `f`'s caller. Because there are recursive calls to `f` many of these `x`'s exist simultaneously. Similarly, many `y`'s exist simultaneously. If local variables are created on function entry and destroyed on function exit, then we can use a stack to hold them.

valued variables it may be necessary to keep local variables after a function call has returned control to its caller. Consider the following Standard ML function.

```
fun f x =
  let fun g y = x+y
   in g
  end
```

When `f` returns it is too soon to destroy `x`. The value which `x` was given in the call to `f` will be needed when the function returned by `f` is called.

In order to be able to use the stack discipline to store local variables most programming languages provide either nested functions (Pascal and Tiger) or functions as returnable results (C) but not both.

Stack frames

- We will consider functions whose local variables can be stored on a stack. We treat the stack as a big array with a *stack pointer*. All locations past the stack pointer are considered to contain rubbish and all of the locations before it are considered to be allocated.
- The stack grows on entry to a function by an increment large enough to hold all of the local variables. On exit it shrinks by the same amount.
- The area on the stack devoted to local variables, parameters, return address and other temporary values is called the function's *activation record* or *stack frame*.
- Sometimes the manufacturer of a computer prescribes a "standard" frame layout. Using this allows calls to functions written in other programming languages.

say that g is the *caller* and f is the *callee*.

- On entry to f, the stack pointer points to the first argument which g passes to f. On entry, f allocates a frame by subtracting the frame size from the stack pointer SP. The old SP becomes the current *frame pointer* FP.
- If the frame size is fixed, then for each function f the FP will always differ from SP by a known constant, and it is not necessary to use a register for FP at all—FP is a 'virtual' register whose value is always SP + *framesize*.

Registers

- A modern machine has a large set of *registers* (typically 32 of them). For efficiency reasons, it is useful to keep local variables, intermediate results of expressions, and other values in registers instead of in the stack frame.
- The benefit of doing this is that registers can be directly accessed by arithmetic instructions whereas accessing memory typically requires *load* and *store* instructions.
- Even on those machines whose arithmetic instructions can access memory, it is still faster to access registers.

Function calling conventions

- A machine usually has only one set of registers, which many different functions need to use. Suppose f is using register r and calls g, which also uses r. Then r must be saved (into a stack frame) before g uses it and restored afterwards.
- If it is f's responsibility to do this then we say that r is a *caller-save* register. If it is g's responsibility then we say that r is a *callee-save* register.
- On most architectures this notion is a convention described in the machine's reference manual. On the MIPS architecture registers 16 to 23 are callee-save and all others are not.
- Sometimes the saves and restores are unnecessary. If f knows that the value of some local variable in not needed after the call it can put it in a caller-save register *and then not save it*. The selection of the appropriate kind of register for each variable is done by the *register allocator* routine.

Boring Fact about Programs #1

Studies of actual programs have shown that
very few functions have more than four arguments,
and almost none have more than six.

Therefore, parameter-passing conventions for modern machines specify that the first k arguments ($k = 4$ or $k = 6$, typically) of a function are passed in registers and the rest in memory.

However, suppose that $f(a\_1, \ldots, a\_n)$ calls $h(z)$. It must pass z in r_1 (say) and save a_1. How has the use of registers saved any time?

Boring Fact about Programs #2

Most functions which are called are *leaf functions*.
That is, they do not call any others.

1. Leaf functions need not store their incoming arguments in memory. In fact, they often do not need a stack frame at all.
2. Some optimising compilers use *interprocedural register allocation*, analysing all the functions in a program at once.
3. The threatened variable may now actually be dead anyway, and need not be saved.
4. Some architectures have *register windows* so that each function invocation can allocate a fresh set without memory traffic.

Address accesses and pointer arithmetic

- C allows programmers to take the address of a formal parameter and guarantees that all the formal parameters of a function are at consecutive addresses. The later fact is used by `printf`.
- Allowing programmers to take addresses can lead to *dangling references*.

```
int *f (int x) { return &x; }
```

- To resolve the contradiction that parameters are passed in registers but have addresses in memory too, the first k of them are passed in registers but any parameter whose address is taken must be written to a memory location on entry to the function.
- Because the memory locations into which register arguments are written must be consecutive if the addresses of parameters 1 and 5 are taken in the function then parameters 2, 3 and 4 must be written to memory on entry to the function also.

to use *call-by-reference*. With call-by-reference, the programmer does not explicitly manipulate the address of a variable, x. Instead, if x is passed to $f(y)$ where y is a "by reference" parameter, the compiler generates code to pass the address of x instead of its contents.

- At any use of y in the function body the compiler generates an extra pointer dereference.
- With call-by-reference, there are no dangling references because y must disappear when f returns.
- The Ada programming language has three types of parameter passing, *input*, *output* and *transput*. Requiring the programmer to be so discerning allows the compiler to generate good code for function calls with no copying of values.

Return addresses

- When function g calls function f, eventually f must return. It needs to know where to go back to.
- If the *call* instruction within g is at address a, then usually the right place is address $a + 1$. This is called the *return address*.
- The call instruction puts the return address in a designated register. A non-leaf function will then have to write it to the stack (unless interprocedural analysis says otherwise) whereas a leaf function will not.

traffic, values are written to memory (in the stack frame) only when necessary for one of these reasons.

1. The variable will be passed by reference, or the `&` operator applied.
2. The variable is accessed by a nested procedure.
3. The variable is too big to fit in a single register.
4. The variable is an array, necessitating address arithmetic.
5. The register holding the variable is needed for a particular purpose.
6. There are so many local variables and parameters that they must be "spilled" into the frame.

We say that a variable *escapes* if it is passed by reference, its address is taken (using `&`), or it is accessed from a nested function.

Static links

- In languages which allow nested functions, an inner function may use variables declared in an outer one. This language feature is called *block structure*.
- In such languages functions may have to have references to not only their own frame but also those of outer functions. There are several methods to accomplish this.
  - Whenever f is called it is passed a pointer to the frame of the function statically enclosing f. This pointer is the *static link*.
  - A global array can be maintained containing—in position i—a pointer to the frame of the most recently entered procedure whose *static nesting depth* is i. Such an array is called a *display*.
  - When g calls f, each variable of g that is actually accessed by f is actually passed as an extra argument. This is called *lambda lifting*.

A nested functions example

```
type tree = {key : string, left: tree, right: tree}

function prettyprint (tree: tree) : string =
let var output := ""

    function write (s: string) =
      output := concat (output, s)

    function show (n: int, t: tree) =
      let function indent (s: string) =
             (for i := 1 to n do write (" ");
               output := concat (output, s))
       in if t=nil then indent (".")
          else (indent (t.key);
                show (n+1, t.left);
                show (n+1, t.right))
      end
in show (0, tree); output
end
```

Abstracting from machine specific details

- Different target machine architectures will have different notions of stack frames. In order to produce a portable compiler we should *abstract away from* the details of the standard layout on a particular architecture.
- We define an interface for the routines which will process stack frames (`frame.h`). If machine-independent parts of the compiler only manipulate frames through this abstract interface then we will be able to replace one implementation of frames with another.

An interface to frames

```
/* frame.h */

typedef struct F_frame_ *F_frame;
typedef struct F_access_ *F_access; /* an abstract
        data type for parameters or local variables */

typedef struct F_accessList_ *F_accessList;
struct F_accessList_ { F_access head; F_accessList tail;};

F_frame F_newFrame(Temp_label name, U_boolList formals);
Temp_label F_name(F_frame f);
F_accessList F_formals(F_frame f);
F_access F_allocLocal(F_frame f, bool escape);
   ⋮
```

Frame creation and use

- The type `F_frame` holds information about formal parameters and local variables. To make a new frame for a function f with k formal parameters, call `F_newFrame(`f`,` l`)`, where l is a list of k booleans recording whether or not each parameter escapes. E.g.

```
F_newFrame(g, U_BoolList (TRUE,
                  U_BoolList (FALSE,
                       U_BoolList (FALSE, NULL))))
```

- The `F_formals` interface function returns a list of k *accesses* denoting the locations where the formal parameters will be kept at run time, as seen from inside the callee.
- The difference in the caller's method of access and the callee's is called the *view shift*.

which may be in the frame or in registers. The contents of `struct F_access_` are only visible inside the `Frame` module.

```
struct F_access_
    { enum { inFrame, inReg } kind;
      union {
         int offset;       /* InFrame */
         Temp_temp reg;    /* InReg */
      } u;
    };

static F_access InFrame(int offset);
static F_access InReg(Temp_temp reg);
```

`InFrame`(X) indicates offset X from the frame pointer; `InReg`(t_{84}) indicates that it will be held in register t_{84}.

The view shift

- Because the "view shift" (from the caller's perspective to the callee's perspective) depends on the calling conventions of the target machine it must be handled by the `Frame` module, starting with `newFrame`.
- For each formal parameter, `newFrame` must calculate two things:
  - How the parameter will be seen from inside the function (in a register, or in a frame location);
  - What instructions must be produced to implement the "view shift"
- For example, a frame-resident parameter will be seen as "memory at offset X from the frame pointer" and the view shift will be implemented by copying the stack pointer to the frame pointer on entry to the procedure.

On the MIPS architecture the parameters could be passed thus:

`InFrame`(0) `InReg`(t_{157}) `InReg`(t_{158})

- The view shift could be implemented in this way:

$$\begin{aligned}&\text{sp} \leftarrow \text{sp} - K\\ &M[\text{sp} + K + 0] \leftarrow \texttt{r2}\\ &t_{157} \leftarrow \texttt{r4}\\ &t_{158} \leftarrow \texttt{r5}\end{aligned}$$

- If the register allocator ultimately chooses `r4` for the second parameter and `r5` for the third then the unnecessary move instructions at the end of the view shift could be deleted at that time.

Local variables

- Some local variables are kept in the frame; others are kept in registers. To allocate a new local variable in a frame f the semantic analysis phase calls `F_allocLocal(`f`, TRUE)` where the boolean argument specifies whether the local variable escapes.
- Calls to `allocLocal` need not all come immediately after the frame is created. For example,

```
function f() =
let var v := 6
 in print(v); let var v := 7 in print(v) end;
    print(v); let var v := 8 in print(v) end;
    print(v)
end
```

In this example, the second and third v variables could be held in the same temporary location. A clever compiler might optimise the size of the frame to make use of this fact.

Calculating escapes

- Local variables that do not escape can be allocated in registers. Escaping variables must be allocated in the frame. A `FindEscape` function can look for escaping variables and record this information in the abstract syntax tree for the program. This processing must happen before semantic analysis begins.
- The traversal function for `FindEscape` will be a mutual recursion on abstract syntax `exp`'s and `var`'s, just like the type-checker.
- Again, like the type checker it will build up an environment which maps variables to bindings. Here the bindings simply record if the variable escapes.

Temporaries and labels

- The compiler's semantic analysis phase will choose registers for parameters and local variables, and also to choose machine-code addresses for procedure bodies. But it is too early to determine exactly which registers are available or exactly where a procedure body will be located. We use the word *temporary* to mean a value that is temporarily held in a register, and the word *label* to mean some machine-language location whose exact address is yet to be determined.
- The `Temp` module provides the following functions.
  - `Temp_newtemp()` return a new temporary
  - `Temp_newlabel()` return a new label
  - `Temp_namedlabel(`*string*`)` returns a new label whose assembly language name is *string*.

Constructors for temporaries and labels

```
Temp_temp Temp_newtemp(void);

Temp_tempList Temp_TempList(Temp_temp h, Temp_tempList t);

Temp_label Temp_newlabel(void);
Temp_label Temp_namedlabel(string name);
string Temp_labelstring(Temp_label s);

Temp_labelList Temp_LabelList(Temp_label h, Temp_labelList t);

Temp_map Temp_empty(void);
Temp_map Temp_layerMap(Temp_map over, Temp_map under);
void Temp_enter(Temp_map m, Temp_temp t, string s);
string Temp_look(Temp_map m, Temp_temp t);

Temp_map Temp_name(void);
```

Environment entries revisited

In the semantic analysis phase of the Tiger compiler, `transDec` creates a new "nesting level" for each function body. The nesting level is stored in the `FunEntry` data structure for the function. The `FunEntry` also needs the `label` of the function's machine code entry point.

```
/* New versions of VarEntry and FunEntry */
struct E_enventry_ {
  enum { E_varEntry, E_funEntry } kind;
  union { struct { Tr_access access; Ty_ty ty; } var;
          struct { Tr_level level; Temp_label label;
                   Ty_tylist formals; Ty_ty result; } fun;
        } u;
  };
```

language being compiled. Many source languages do not have nested functions so `Frame` should not know about static links.

- The `Translate` module knows that each frame contains a static link. The static link is passed in a register and stored in the frame. Since the static link behaves so much like a formal parameter we treat it as one, specifically one which escapes.
- Then, just before creating a new frame we add another formal parameter to the front for the static link.

```
F_newFrame(label, U_BoolList(TRUE, fmls))
```

What comes back is an `F_frame`. In this frame are three offset values, accessible by calling `F_formals(frame)`.

Translation to intermediate code

- The semantic analysis phase of a compiler must translate abstract syntax into machine code. It can do this at the same time as type-checking, or after.
- Although it would be possible to translate directly to real machine code this would hinder portability and modularity. An *intermediate representation* (IR) is an abstract machine language which can express target machine operations without too much machine-specific detail. It is independent of the details of the source language.
- Thus the *front end* of a compiler does lexical analysis, parsing, semantic analysis, and translation to an intermediate representation. The *back end* does optimisation of the intermediate representation and translation to machine language.

- It must be convenient for the semantic analysis phase to produce.
- It must be convenient to translate into real machine language, for all the desired target machines.
- Each construct must have a clear and simple meaning, so that optimising transformations that rewrite the intermediate representation can easily be specified and implemented.

The intermediate representation should have individual components which are extremely simple: a single fetch, store, add, move or jump. A piece of abstract syntax can be translated into a set of abstract machine instructions and groups of abstract machine instructions can be brought together to form genuine machine instructions.

Expressions in the IR language

CONST (i) The integer constant i.

NAME (n) The symbolic constant n, corresponding to an assembly language label.

TEMP (t) Temporary t. A temporary in the abstract machine is similar to a register in the real one.

BINOP (o, e_1, e_2) The application of operator o to e_1 and e_2, evaluated in that order. The operators are PLUS, MINUS, MUL, DIV, AND, OR, XOR, LSHIFT, RSHIFT, ARSHIFT.

MEM (e) The contents of *wordSize* bytes of memory from address e.

CALL (f, l) The application of function f to argument list l, evaluated in that order.

ESEQ (s, e) The statement s is evaluated for its side effect then e for its result.

Statements in the IR language

MOVE (**TEMP** t, e) Evaluate e and move it to temporary t.

MOVE (**MEM** e_1, e_2) Evaluate e_1, yielding address a. Then evaluate e_2 and store the result into *wordSize* bytes of memory starting at a.

EXP (e) Evaluate e and discard the result.

JUMP $(e, labs)$ Transfer control to address e. The list of labels specifies all possible locations that e can evaluate to, for dataflow analysis.

CJUMP (o, e_1, e_2, t, f) Evaluate e_1, then e_2. Compare the results using the relational operator o. If the result is `true`, jump to t; otherwise jump to f. The relational operators are EQ, NE, LT, GT, LE, GE, ULT, UGT, ULE, UGE.

SEQ (s_1, s_2) The statement s_1 followed by s_2.

LABEL (n) Define the constant value of name n to be the current machine code address.

Kinds of expressions

- The Tiger language contains expressions of different kinds: some produce values and some do not (assignments, loops and procedure calls). Expressions with Boolean values can be best represented by a conditional jump.
- We will make a union type, as usual, with a `kind` tag. This will model three kinds of expressions.

  Ex: This is a value-producing expression, represented as a `Tr_exp`.

  Nx: An expression with no result, represented as a `Tr_stm`.

  Cx: A conditional represented as a statement which may jump to a true-label or a false-label. These labels have yet to be filled in. The statement never "falls through".

Implementing expressions

```
/* in translate.h */
typedef struct Tr_exp_ *Tr_exp

/* in translate.c */
struct Cx { patchList trues; patchList falses; T_stm stm; };

struct Tr_exp_
   { enum { Tr_ex, Tr_nx, Tr_cx } kind;
     union { T_exp ex; T_stm nx; struct Cx cx; } u;
   };

static Tr_exp Tr_Ex(T_exp ex);
static Tr_exp Tr_Nx(T_stm nx);
static Tr_exp Tr_Cx(patchList trues, patchList falses, T_stm stm);
```

Translating an expression

The Tiger expression `a>b|c<d` might translate to the following.

```
Temp_label z = Temp_newlabel();
T_stm s1 =
   T_Seq(T_Cjump(T_gt, a, b, NULL_t, z),
       T_Seq(T_Label(z),
           T_Cjump(T_lt, c, d, NULL_t, NULL_f)));
```

The destinations t and f are not known yet, although `z` is. The statement contains `NULL`$_t$ and `NULL`$_f$, which are yet to be filled in.

```
typedef struct patchList_ *patchList
struct patchList_ { Temp_label *head; patchList tail; };

static patchList PatchList(Temp_label *head, patchList tail);
```

We can complete the translation of `a>b|c<d` as follows.

```
patchList trues =
   PatchList(&s1->u.SEQ.left->u.CJUMP.true,
     PatchList(&s1->u.SEQ.right->u.SEQ.right->u.CJUMP.true,
       NULL));

patchList falses =
   PatchList(&s1->u.SEQ.right->u.SEQ.right->u.CJUMP.false,
       NULL);

Tr_exp e1 = Tr_Cx(trues, falses, s1);
```

Modifying expressions

Sometimes we will have an expression of one kind (`Ex`, `Nx`, `Cx`) and we need to convert it to another kind. For example, the Tiger expression

```
flag := (a>b | c<d)
```

requires the conversion of a `Cx` into a `Ex` so that a 1 or 0 can be stored into `flag`. It is helpful to have three conversion functions.

```
static   T_exp   unEx(Tr_exp e);
static   T_stm   unNx(Tr_exp e);
static struct Cx unCx(Tr_exp e);
```

To implement these we need some utility functions on patch lists.

`trues` patch list with `t`. A similar call would fill in the `falses` list.

```
void doPatch(patchList tList, Temp_label label) {
   for (; tList; tList=tList->tail)
     *(tList->head) = label;
}
```

By calling `joinPatch(first,second)` we link two patch lists together.

```
patchList joinPatch(patchList first, patchList second) {
  if (!first) return second;
  /* go to end of list */
  for (; first->tail; first=first->tail);
  first->tail = second;
  return first;
}
```

The unEx conversion function

```
static T_exp unEx(Tr_exp e) {
  switch(e->kind) {
    case Tr_ex:
      return e->u.ex;
    case Tr_cx: {
      Temp_temp r = Temp_newtemp();
      Temp_label t = Temp_newlabel(), f = Temp_newlabel();
      doPatch(e->u.cx.trues, t);
      doPatch(e->u.cx.falses, f);
      return T_Eseq(T_Move(T_Temp(r), T_Const(1)),
               T_Eseq(e->u.cx.stm,
                 T_Eseq(T_Label(f),
                   T_Eseq(T_Move(T_Temp(r), T_Const(0)),
                     T_Eseq(T_Label(t),
                              T_Temp(r))))));
    }
    case Tr_nx:
      return T_Eseq(e->u.nx, T_Const(0));
  }
  assert(0); /* can't get here */
}
```


Simple variables

- The semantic analysis phase has a function which type-checks a variable in the context of a type environment `tenv` and a value environment `venv`. Previously the expression returned was just a place-holder but now we may return the intermediate representation of each Tiger expression.
- For the simple variable v declared in the current procedure's stack frame we translate it as

  $$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP } \texttt{fp}, \text{CONST } k))$$

  where k is the offset of v within the frame and TEMP `fp` is the frame pointer register.
- In Tiger this translation is simplified by the fact that all variables are the same size—the natural word size of the machine.

Interface between Translate and Semant

- The type `Tr_exp` is an abstract data type, whose `Ex` and `Nx` constructors are only visible within the `Translate` module.
- All of the manipulation of MEM nodes should be done within the `Translate` module, not within `Semant`.
- We add a function

  ```
  Tr_Exp Tr_simpleVar(Tr_Access a, Tr_Level l);
  ```

  to the `Translate` interface. Now `Semant` can pass in the `access` of x (obtained from `Tr_allocLocal`) and the `level` of the function in which x is used and get back a `Tr_Exp`.

Extending the Frame module

The `Frame` module holds all machine-dependent definitions; here we add a frame-pointer register `FP` and a constant whose value is the machine's natural word size.

```
/* frame.h */
   ⋮
Temp_temp F_FP(void);
extern const int F_wordSize;
T_exp F_Exp(F_access acc, T_exp framePtr);
```

Using the Frame module

- The function `F_Exp` is used by `Translate` to turn an `F_access` into a `Tree` expression. The `framePtr` argument is the address of the stack frame that the `F_access` is in.
- Thus for an access a such as `InFrame` k, we have

  ```
  F_Exp(a, T_Temp(F_FP()))
  ```

  returns the result

  $$\text{MEM}(\text{BINOP}(\text{PLUS}, \text{TEMP } \texttt{fp}, \text{CONST } k))$$

variables or formal parameters. A function might access a variable which was declared by its parent. When a variable is declared at an outer level of static scope like this we access it by following static links. in general, we have this:

$$\begin{array}{l}\text{MEM(BINOP(PLUS, CONST } k_n,\\ \quad\text{MEM(BINOP(PLUS, CONST } k_{n-1},\\ \quad\quad\cdots\\ \quad\quad\text{MEM(BINOP(PLUS, CONST } k_1,\\ \quad\quad\text{TEMP fp)) } \cdots \text{))))}\end{array}$$

- The constants $k_1, \ldots, k_{n-1}$ are the various static link offsets in nested functions and k_n is the offset of x in its own frame.

Translating complex logical expressions

- The result of a comparison operator will be a `Cx` expression; a statement s which will jump to any true-destination and false-destination which are specified.
- Making a `Cx` expression from a single use of the comparison operators of the Tiger language is relatively straightforward. For example, the expression `x<5` will be translated as a `Cx` with

$$\begin{array}{r}\texttt{trues} = \{t\}\\ \texttt{falses} = \{f\}\\ \texttt{stm} = \text{CJUMP (LT, } x\text{, CONST 5, } \boxed{\texttt{NULL}}_t\text{, } \boxed{\texttt{NULL}}_f\text{)}\end{array}$$

- However, `Cx` expressions can themselves form part of a larger truth-valued expression. The `&` (and) and `|` (or) operators of the Tiger language combine logical expressions using short-circuit evaluation. These have already been translated into `if`-expressions in the abstract syntax.

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

is to treat e_1 as a `Cx` expression, and e_2 and e_3 as `Ex` expressions. We need to fit the three sub-expressions together into a stream of statements into which we can insert labels for the **then** branch and the **else** branch.

- We proceed by applying `unCx` to e_1 and `unEx` to e_2 and e_3. We make two labels to which the conditional will branch, `t` and `f`. Allocate a temporary `r`, and after label `t` move e_2 to `r`; after label `f` move e_3 to `r`. Both branches should finish by jumping to a newly created "join" label.

Problem cases for conditional expressions (1)

Following our proposed translation would lead to inefficient code. A common use of an `if`-expression is to allow the selection between two "statements" (expressions which produce no value). In this case their representation will have been constructed using the `Nx` constructor. Applying `unEx` to them will work but it might be better to recognise this case specially.

Problem cases for conditional expressions (2)

Worse, if e_2 or e_3 is a `Cx` expression then the use of `unEx` produces even longer statement sequences. Again, we should recognise this case specially. For example, an `if`-expression such as the following

$$\textbf{if } (\underbrace{\textbf{if } x < 5 \textbf{ then } a > b \textbf{ else } 0}\_x < 5 \;\&\; a > b) \textbf{ then } \ldots$$

should be translated as

$$\text{SEQ}(s_1(z,\, f),\ \text{SEQ}(\text{LABEL } z,\ s_2(t,\, f)))$$

for some new label z. The notation $s_1(z,\, f)$ denotes the `Cx` statement s_1 with its `trues` labels filled in with z and its `falses` labels filled in with f.

Strings

- A string literal in the Tiger (or C) language is the constant address of a segment of memory initialised to the proper characters. In assembly language a label is used to refer to this address. At some place the *definition* of the label appears, followed by pseudo-instructions to reserve and initialise a block of memory.
- For each string literal `lit`, the `Translate` module make a new label `lab`, and returns the tree `T_Name(lab)`. It also puts the assembly-language fragment `F_String(lab,lit)` onto a global list of such fragments to be handed to the code emitter.
- All string operations are performed in functions provided by the runtime system.

Representing strings

- Different programming languages represent strings in different ways. These representations have different strengths and drawbacks.
- In Pascal strings are fixed-length arrays of characters; literals are padded with blanks to make them fit. The inflexibility in length is not very useful.
- In C, strings are pointers to variable-length sequences of characters terminated by a zero. Here, a string containing a zero byte cannot be represented.
- In Tiger we will have strings represented by a pair of a one-word integer containing the number of characters together with the characters themselves.

Record and array creation

- The Tiger language expression $t\{\ f_1 = e_1, f_2 = e_2, \ldots, f_n = e_n\ \}$ creates an n-element record of type t with fields f_i initialised to the values of expressions e_i. Such a record may outlive the instance of the function which created it, so it cannot be allocated on the stack. Instead it must be allocated in another memory area called the *heap*. Heap-allocated values must be reclaimed by a *garbage collector* when an analysis determines that they are no longer reachable.
- To create a record allocate an n-word memory area into a new temporary r (using `malloc`). A series of MOVE instructions can initialise offsets 0, W, $2W$, $\ldots$, $(n-1)W$ from r. The result of the whole expression is TEMP r.
- Array creation is very similar except that all locations are initialised to the same value.

Calling runtime-system functions

- To call an external function named `initArray` with arguments a and b, simply generate a call such as

```
CALL (NAME(Temp_namedlabel("initArray")),
    T_ExpList(a, T_ExpList(b, NULL)))
```

  This refers to an external function written in C or assembler.
- However, on some operating systems the C compiler puts an underscore at the beginning of each label; the calling conventions of C are different from those of Tiger functions; C functions don't expect to receive a static link, and so on. We encapsulate all of this in the `Frame` module, as usual.

```
/* frame.h */
   ⋮
T_exp F_externalCall(string s, T_expList args);
```

test:
    if not(*condition*) goto *done*
    body
    goto *test*
done:

If a **break** statement occurs within the body (and not nested within an interior `while` loop or a procedure body) then the translation is simply a jump to the label *done*.

- So that `transExp` can translate **break** statements, it will have a new formal parameter `break` that is the *done* label of the nearest enclosing loop.

For loops

A **for** statement can be expressed as a combination of **let** .. **in** .. **end** and a **while** loop.

for i := lo **to** hi **do** *body*

let var i := lo
    var *limit* := hi
in while i <= *limit*
    do (*body*; i := $i + 1$)
end

This is almost right. It fails when hi evaluates to the maximum representable integer. The final increment in the loop body will either overflow or wrap around. Neither behaviour is correct. A production quality compiler would generate an extra test for this.

Function calls

Translating a function call $f(e\_1, \ldots, e\_n)$ is relatively simple except that the static link must be added as an extra argument.

$$\text{CALL}(\text{NAME } l\_f, [sl, e\_1, \ldots, e\_n])$$

Here l_f is the label for f, sl is the static link and the e_i are the translations of the argument list a_i. To calculate the static link both

`Frame` module.

Variable definition

- The `transDec` function which we saw when considering type-checking updates the value environment and type environment. These are used in processing the body of a **let** expression.
- In Tiger, variables are always given their initial value at the time that they are declared and so a variable declaration must also produce a `Tree` expression that must be put just before the body of the **let**. Therefore `transDec` must also return a `Tr_exp` containing assignment expressions which accomplish these initialisations.
- If `transDec` is applied to function and type declarations, the `Tr_exp` must be a "no-op" expression such as `Tr_Ex(T_Const(0))`.

Function definition

Each Tiger function is translated into a *prologue* (1–5), a *body* (6) and an *epilogue* (7–11).

1. pseudo-instructions for the function beginning
2. a label definition for the function name
3. an instruction to adjust the stack pointer
4. instructions to save "escaping" arguments
5. store instructions to save any callee-save registers
6. the function *body*
7. an instruction to deliver the function result
8. load instructions to restore the callee-save registers
9. an instruction to reset the stack pointer
10. a jump to the return address
11. pseudo-instructions for the function end

Frames and function definitions

- Some of the parts of the translation of a function definition depend on knowledge of the frame size. In order to keep this information separated from the machine-independent parts of the compiler we find that we should add more definitions to the `Frame` module.
- For return values we need a distinguished return value location.

```
Temp_temp F_RV(void);
```

- Moving formal parameters and the saving and restoring of callee-save registers should also be handled by a function in the `Frame` module.

```
T_stm F_procEntryExit1(F_frame frame, T_stm stm);
```

Translating the `Tree` language to machine-language

Some features of the `Tree` language must be adapted before translation into machine language.

- The CJUMP instruction can jump to either of two labels but real machine's conditional jumps fall through if the condition is false.
- ESEQ nodes within expressions are inconvenient because they make different orders of evaluating subtrees yield different results.
- CALL nodes within expressions cause the same problem.
- CALL nodes within the argument-expressions of other CALL nodes cause problems when sharing formal parameter registers.

Rewriting trees

- We can take any tree and *rewrite* it into an equivalent tree without the problems due to ESEQ and CALL nodes. Without these cases we will have a simple collection of SEQ nodes clustered together at the top of the tree. This can easily be turned into a linear list of instructions.
- The transformation is done in three stages:
    1. A tree is rewritten into a list of *canonical trees* without SEQ or ESEQ nodes.
    2. This list is grouped into *basic blocks*, which contain no internal jumps or labels.
    3. The basic blocks are ordered into a set of *traces* in which every CJUMP is immediately followed by its `false` label.

Canonicalisation module

The `Canon` module has these tree rearrangement functions.

```
/* canon.h */
  ⋮
typedef struct C_stmListList_ *C_stmListList;
struct C_block { C_stmListList stmLists; Temp_label label; };
struct C_stmListList_ { T_stmList head; C_stmListList tail; };

T_stmList C_linearize(T_stm stm);
struct C_block C_basicBlocks(T_stmList stmList);
T_stmList C_traceSchedule(struct C_block b);
```

`Linearize` removes the ESEQs and moves the CALLs to top level. `BasicBlocks` groups statements into sequences of straight-line code. `TraceSchedule` orders the blocks so that each CJUMP is followed by its `false` label.

trees which obey the following two rules.

1. There is no use of the SEQ or ESEQ nodes in the tree.
2. The parent of each CALL is either of the form EXP(...) or of the form MOVE(TEMP t, ...). The former correspond to procedure calls (which do not return a result) and the latter to function calls (which do).

- ESEQ nodes are eliminated by moving them further and further up the tree until they become SEQ nodes. Many useful transformations on trees can be defined. These can be repeatedly applied to bring about the alterations which we want to make.

Commutativity

We wish to re-arrange intermediate representation trees so that all of the ESEQ nodes move up the tree. When we consider a subtree produced by `z+(x:=y,x)` such as

$$\text{BINOP}(\text{PLUS}, \text{MEM}(z), \text{ESEQ}(\text{MOVE}(\text{MEM}(x), y), \text{MEM}(x)))$$

with a subtree such as

$$\text{ESEQ}(\text{MOVE}(\text{MEM}(x), y), \text{BINOP}(\text{PLUS}, \text{MEM}(z), \text{MEM}(x)))$$

as would have been produced by `(x:=y,z+x)` we need to determine whether the `x:=y` and the `z` *commute*.

We cannot always determine this at compile time. Under some run-time evaluations `x` and `z` might be *aliases*. That is, they might be two different names for the same location. A situation such as this can be created by parameter passing where variables may be passed by reference.

and array types are passed by reference and thus inside the body of the `alias` function, `x` and `z` are two different names for the same record.

```
function eval () : int =
let
  type recvar = { val : int }
  var y: int := 10
  var z: recvar := recvar{ val = 0 }
  function alias (x: recvar) : int =
    (x.val := y, z.val + x.val)
in
  alias (z)
end
```

In C++ *reference declarations* give rise to aliasing. The example here could be coded as `int alias(int& x) {return (x=y, z+x);}`.

Conservatively estimating commutativity

- Although we cannot guarantee to always determine the validity of commuting a statement with an expression at compile-time we can at least *conservatively approximate* commutativity. This means that we say either "they definitely do commute" or "perhaps they don't commute".
- For example, we know that any statement will commute with any constant value (say, the expression CONST(n)) so we can always justify special cases such as the following.

$$\begin{aligned}&\text{BINOP}(op, \text{CONST}(n), \text{ESEQ}(s, e))\\ =\ &\text{ESEQ}(s, \text{BINOP}(op, \text{CONST}(n), e))\end{aligned}$$

General rewriting rules

Given a `Tree` statement or expression we first identify the subexpressions and extract these, as directed by our tree transformations. We then rebuild our statement or expression with the extracted subexpressions replaced in the right way.

```
typedef struct expRefList_ *expRefList;
struct expRefList_ { T_exp *head; expRefList tail; };

struct stmExp { T_stm s; T_exp e; };

/* Combine the statement parts from ESEQs */
static T_stm reorder(expRefList rlist);

static struct stmExp do_exp(T_exp exp);
static T_stm do_stm(T_stm stm);
```

Processing expressions

```
static struct stmExp do_exp(T_exp exp) {
switch (exp->kind) {
  case T_BINOP:                                  /* BINOP(o, e1, e2) */
    return StmExp(
      reorder(ExpRefList(&exp->u.BINOP.left,
        ExpRefList(&exp->u.BINOP.right,NULL))), exp);
  case T_MEM:                                                /* MEM(e) */
    return StmExp(
      reorder(ExpRefList(&exp->u.MEM,NULL)), exp);
  case T_ESEQ: {                                        /* ESEQ(s, e) */
     struct stmExp x = do_exp(exp->u.ESEQ.exp);
     return StmExp(seq(do_stm(exp->u.ESEQ.stm), x.s), x.e);
   }
  case T_CALL:                                          /* CALL(f, l) */
    return StmExp(reorder(get_call_rlist(exp)), exp);
  default:                                  /* CONST, NAME and TEMP */
    return StmExp(reorder(NULL), exp);
}}
```


Eliminating No-Ops

In the case that one of two adjacent statements is a No-Op we can omit it from the emitted instructions.

```
/* test for no-ops */
static bool isNop(T_stm x) {
  return x->kind == T_EXP && x->u.EXP->kind == T_CONST;
}

/* combine statements, omitting no-ops */
static T_stm seq(T_stm x, T_stm y) {
  if (isNop(x)) return y;
  if (isNop(y)) return x;
  return T_Seq(x,y);
}
```

Processing statements

```
static T_stm do_stm(T_stm stm) {
  switch (stm->kind) {
  case T_SEQ:
    return seq(do_stm(stm->u.SEQ.left), do_stm(stm->u.SEQ.right));
  case T_JUMP:
    return seq(reorder(ExpRefList(&stm->u.JUMP.exp, NULL)), stm);
  case T_CJUMP:
    return seq(reorder(ExpRefList(&stm->u.CJUMP.left,
                      ExpRefList(&stm->u.CJUMP.right,NULL))), stm);
  case T_MOVE:  ... to do ...
  case T_EXP:
    if (stm->u.EXP->kind == T_CALL)
         return seq(reorder(get_call_rlist(stm->u.EXP)), stm);
    else return seq(reorder(ExpRefList(&stm->u.EXP, NULL)), stm);
 default:
    return stm;
 }}
```

```
  switch (stm->kind) {
  ... other cases as before ...
  case T_MOVE:
    if (stm->u.MOVE.dst->kind == T_TEMP &&
        stm->u.MOVE.src->kind == T_CALL)
      return seq(reorder(get_call_rlist(stm->u.MOVE.src)), stm);
    else if (stm->u.MOVE.dst->kind == T_TEMP)
      return seq(reorder(ExpRefList(&stm->u.MOVE.src, NULL)), stm);
    else if (stm->u.MOVE.dst->kind == T_MEM)
      return seq(reorder(ExpRefList(&stm->u.MOVE.dst->u.MEM,
                        ExpRefList(&stm->u.MOVE.src, NULL))), stm);
    else if (stm->u.MOVE.dst->kind == T_ESEQ) {
      T_stm s = stm->u.MOVE.dst->u.ESEQ.stm;
      stm->u.MOVE.dst = stm->u.MOVE.dst->u.ESEQ.exp;
      return do_stm(T_Seq(s, stm));
    }
    assert(0); /* dst should be temp or mem only */
  ... other cases as before ...
}}
```

Commuting statements and expressions

With the assistance of `do_exp` and `do_stm`, the `reorder` function can pull the statement s_i out of each expression e_i on its list of references, going from right to left. Temporaries are introduced if the statement and expression do not commute. For example, there are three possible re-arrangements of $[e\_1, e\_2, \text{ESEQ}(s, e\_3)]$:

- $s; [e\_1, e\_2, e\_3]$
- $\text{SEQ}(\text{MOVE}(t\_1, e\_1), s); [\text{TEMP}(t\_1), e\_2, e\_3]$
- $\text{SEQ}(\text{MOVE}(t\_1, e\_1), \text{SEQ}(\text{MOVE}(t\_2, e\_2), s)); [\text{TEMP}(t\_1), \text{TEMP}(t\_2), e\_3]$

The following function estimates whether two expressions commute.

```
static bool commute(T_stm x, T_exp y) {
  return isNop(x) || y->kind==T_NAME || y->kind==T_CONST;
}
```


However, each function returns its result in the same return-value register TEMP(`RV`). Thus if we have

$$\text{BINOP}(\text{PLUS}, \text{CALL}(\dots), \text{CALL}(\dots))$$

then the second call will overwrite the `RV` register before the PLUS can be executed. As always, we can solve this by introducing a new temporary.

$$\text{CALL}(\mathit{fun}, \mathit{args}) \rightarrow \text{ESEQ}(\text{MOVE}(\text{TEMP } t, \text{CALL}(\mathit{fun}, \mathit{args})), \text{TEMP } t)$$

Linearising statements

Once an entire function body s_0 is processed with `do_stm`, the result is a tree s_0' where all the SEQ nodes are near the top. The `linearize` function repeatedly applies $\text{SEQ}(\text{SEQ}(a, b), c) = \text{SEQ}(a, \mathit{seq}\ (b, c))$.

```
static T_stmList linear(T_stm stm, T_stmList right) {
 if (stm->kind == T_SEQ)
   return linear(stm->u.SEQ.left,
            linear(stm->u.SEQ.right,
              right));
 else return T_StmList(stm, right);
}

T_stmList C_linearize(T_stm stm) {
    return linear(do_stm(stm), NULL);
}
```


Compiling conditional branches

- Another aspect of the `Tree` language which must be compiled down into simpler instructions is the two-way branch of the CJUMP operator. On a real machine the conditional jump either transfers control (on `true`) or "falls through" to the next instruction.
- To make the trees easy to translate into machine instructions we rearrange them so that every $\text{CJUMP}(\mathit{cond}, l\_t, l\_f)$ is immediately followed by $\text{LABEL}(l\_f)$, its "false branch".
- We make this transformation in two stages: we first take the list of canonical trees and form them into *basic blocks*; then we order the basic blocks into a *trace*.

Basic blocks

- In determining where the jumps go in a program we are analyzing the program's *control flow*. We consider only the sequencing of instructions, independent from the program's data. In this setting we cannot say whether the condition of a jump will evaluate to `true` or `false` and so we consider both possibilities.
- A basic block is a sequence of statements which is entered at the beginning and left at the end. That is:
  - the first statement is a LABEL;
  - the last statement is a JUMP or CJUMP; and
  - there are no other LABELS, JUMPS or CJUMPS.

Finding basic blocks

We divide a long sequence of statements into basic blocks as follows.

- The sequence is scanned from beginning to end.
- Whenever a LABEL is found, a new block is started.
- Whenever a JUMP or CJUMP is found, a block is ended.
- If this leaves any block not ending with a JUMP or CJUMP then a JUMP to the next block's label is added to the end of the block.
- If this leaves any block not beginning with a LABEL then a new label is invented and put there.

Traces

- Basic blocks can be arranged in any order and the result of executing the program will be the same. We take advantage of this fact in two ways:
  - we can move the false-labelled block immediately after its conditional branch; and
  - we can arrange that many of the unconditional jumps are followed by their target label.
- A *trace* is a sequence of statements which could be consecutively executed during the execution of the program. For our purposes we want to make a set of traces that exactly covers the program: each block must be in exactly one trace. To minimise the number of JUMPs from one trace to another, we would like to have as few traces as possible in our covering set.

```
while Q is not empty {
    Start a new (empty) trace, call it T.
    Remove the head element b from Q.
    while b is not marked {
        Mark b; append b to the end of T.
        Examine the successors of b (blocks to which b branches)
        if there is any unmarked successor c
            b ← c
    }
    /* All the successors of b are marked. */
    End the current trace T.
}
```

Instruction selection

- The `Tree` language expresses only one operation in each tree node: memory fetch or store, addition or subtraction, conditional jump, and so on. A real machine instruction can often perform several of these in the same instruction.
- Finding the appropriate machine instructions to implement a given intermediate representation tree is the job of the *instruction selection* phase of a compiler.
- We express a machine instruction as part of an IR tree, called a *tree pattern*. Then instruction selection becomes the task of *tiling* (or covering) the tree with a minimal set of non-overlapping tree patterns.

language expression using instructions from the *Jouette* instruction set.

```
a[i] := x
```

- We use the following facts about the variables in the expression and the *Jouette* architecture.
  - The variable `i` is a register variable.
  - The variables `a` and `x` are frame resident. (An array variable is really a pointer to an array.)
  - The wordsize W is 4 (bytes). Thus `a[i]` is represented by

    MEM(BINOP (+, a, BINOP(*, i, 4)))
  - Register r_0 always contains zero.

Optimal and optimum tilings

- The best tiling of a tree corresponds to an instruction sequence of least cost. If we assign a cost to each instruction then we could define an *optimum* tiling as one whose tiles sum to the lowest possible value.
- In contrast an *optimal* tiling is one where no two adjacent tiles can be combined into a single tile of lower cost.
- If there is some tree pattern that can be split into several tiles of lower combined cost then we should remove that pattern from our catalogue of tiles before we begin.
- Every optimum tiling is also optimal, but not vice versa.

The "Maximal Munch" algorithm

- The tiles which are used to tile the IR tree can be graded in terms of the number of their nodes. The tile for ADD has one node, the tile for SUBI has two and the tile for MOVEM has three. Loosely speaking, we can say that MOVEM is 'larger' than SUBI and that SUBI is 'larger' than ADD.
- The algorithm for optimal tiling is called *Maximal Munch* and proceeds in this way.
  - Starting at the root of the tree, find the largest tile that fits. (If two tiles of equal size match at the root then choose either one.)
  - Cover the root node (and others near it) with this tile, leaving several subtrees.
  - Repeat the algorithm for each subtree.

Dynamic programming

- Maximal Munch always finds an optimal tiling, but not necessarily an *optimum* one. A *dynamic programming* algorithm can find the optimum.
- The algorithm for optimum tiling proceeds in this way.
  - Work bottom-up by first finding the costs of all the children (and grandchildren, ...) of node n. Then each tile is matched against node n.
  - A tile has zero or more *leaves* where subtrees can be attached. For each tile t of cost c that matches at node n there will be subtrees $s_1, \ldots s_n$ of cost $c_1, \ldots, c_n$. So the cost of matching tile t is $c + c_1 + \cdots + c_n$.
  - Of all the tiles t_j which match at node n, the one with the minimum cost match is chosen and the (minimum) cost of node n is thus computed.

Instruction emission

- Once the cost of the root node (and thus the entire tree) is found, the *instruction emission* phase begins. The algorithm is as follows:

  Emission (node n): for each of the leaves l_i of the tile selected at node n, perform Emission(l_i). Then emit the instruction matched at node n.
- Note that Emission(n) does not apply itself recursively to the *children* of node n, but to the *leaves of the tile* which matched.

  For example, for MEM(+, CONST 1, CONST 2) we emit instructions for CONST 1 and for the node itself but not for any tile rooted at the + node.

Liveness analysis

- The front end of a compiler translates programs into an intermediate language with an unlimited number of temporary storage locations (*temporaries*). The translated program must ultimately run on a machine with a fixed number of registers.
- Even if their scopes overlap, two temporaries can fit into a single register if they are never in use at the same time. Following from this, many temporaries can fit into few registers; if they don't all fit, the excess temporaries can be kept in memory.
- A compiler needs to analyse the intermediate representation tree to determine which temporaries are in use at the same time. A variable is said to be *live* if it holds a value which may be needed in the future. This analysis is called *liveness analysis*.

control-flow graph. Each statement in the program is a node in the graph. If statement S_1 can be followed by statement S_2 then there is an edge from S_1 to S_2.

- In determining the *live range* of a variable we work backwards through the control-flow graph, examining statements.
  - If a variable b is *used* in a statement (i.e. its value is taken) then b must have been live on the edge leading to the statement.
  - If a variable is assigned into in a statement (i.e. its value is set) then it must be dead on the edge leading to the statement.

Utilising the results

The effect of understanding and making use of the results of liveness analysis is to allow us to see that two variables could be replaced by one.

	$a \leftarrow 0$	$a \leftarrow 0$	$b \leftarrow 0$
$L_1:$	$b \leftarrow a+1$	$L_1:$ $a \leftarrow a+1$	$L_1:$ $b \leftarrow b+1$
	$c \leftarrow c+b$	$c \leftarrow c+a$	$c \leftarrow c+b$
	$a \leftarrow b*2$	$a \leftarrow a*2$	$b \leftarrow b*2$
	if $a < N$ goto L_1	if $a < N$ goto L_1	if $b < N$ goto L_1
	return c	return c	return c

edges which come from *predecessor* nodes. The set $Pred[n]$ is all the predecessors of n and the set $Succ[n]$ is the set of all successors.

- An assigment to a variable or temporary *defines* that variable. An occurrence of a variable in an expression *uses* the variable. We can speak of the *def* of a variable as the set of node that use it, and conversely for a graph node. Similarly we can speak of the *uses* of a variable or graph node.
- A variable is *live* on an edge if there is a directed path from that edge to a *use* of the variable which does not go through any *defs*. A variable is *live-in* if it is live on any of the in-edges of a node. It is *live-out* if live on any out-edges.

Calculation of liveness by iteration

```
/* initialise the edge sets */
for each n
  in[n] ← {};   out[n] ← {};
repeat
  for each n
    /* record the old values */
    in'[n] ← in[n];      out'[n] ← out[n];
    /* compute the new values */
    in[n] ← use[n] ∪ (out[n] − def[n]);
    out[n] ← {};
    for each s in succ[n]
      out[n] ← out[n] ∪ in[s];
until in'[n] = in[n] and out'[n] = out[n] for all n
```


Index