The UNIVERSITY *of* EDINBURGH
SCHOOL *of* COMPUTER SCIENCE

Computer Science 3

Compiling Techniques

Martin Hofmann

mxh@dcs

OHP Slides based on Stephen Gilmore's slides from 98/99

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)
  - You will probably save time by using either ML or Java
  - Ask me if you would like to use a different language.
- Obtain the (partial) source code and the *Tiger'99 Reference Manual* from the CS3 Compiling Techniques Web page.
- Working in groups of 2-3, 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` and `mxh@dcs` telling them the names of the people in your group.
- Submit source, Linux x86 ELF executable and documentation (a README file)
- **Deadline:** End of Week 9 of this term.

Consulting

Each group is entitled to a minimum of 6 hours consulting time provided by the course tutors:

- Stephen Gilmore (`stg@dcs`, Ph.: 650 5189, JCMB Rm.: 1608)
- Björn Francke (`bf@dcs`, Ph. 650 5013, JCMB Rm.: 3311)

Tutors will be assigned to groups in week 3.

Of course, you can also talk to me after the lecture or send me mail as usual.

Tiger and Tiger'99

- Tiger '99 is a simplified 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.

Course outline by weeks

1. Introduction and concepts; structure.
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.

Structure of a compiler

Source program	→	Lexical analysis	→
→ Token stream	→	Syntax analysis	→
→ abstract syntax tree	→	Semantic analysis	→
→ AST+Environment	→	Translation	→
→ IR Tree	→	Instruction Selection	→
→ Assem	→	Optimisations	→
→ Assem	→	Code emission	→
→ Assembly L.	→	Linker	→
→ Executable			

Other compilers

- Java → JVM Bytecode
- Translating between high-level languages (ML→Java, Java→C, C++→C
- LaTeX→ DVI → PostScript
- XML, HTML → ...
- Scene description language → .gif, .ppm

Straight-line programming language

Stm→*Stm*;*Stm*	(CompoundStm)		
Stm→`id :=`*Exp*	(AssignStm)	*ExpList*→*Exp* `,` *ExpList*	(PairExpList)
Stm→`print (`*ExpList*`)`	(PrintStm)	*ExpList*→*Exp*	(LastExpList)
Exp→`id`	(IdExp)	*BinOp*→`+`	(Plus)
Exp→`num`	(NumExp)	*BinOp*→`-`	(Minus)
Exp→*Exp* *BinOp* *Exp*	(OpExp)	*BinOp*→`*`	(Times)
Exp→`(`*Stm* `,` *Exp*`)`	(EseqExp)	*BinOp*→`/`	(Div)

Abstract Syntax Trees in ML

```
type id = string

datatype binop = Plus | Minus | Times | Div

datatype stm = CompoundStm of stm * stm
             | AssignStm of id * exp
             | PrintStm of exp list

     and exp = IdExp of id
             | NumExp of int
             | OpExp of exp * binop * exp
             | EseqExp of stm * exp
val prog =
```

A concrete program

```
 CompoundStm(AssignStm("a",OpExp(NumExp 5, Plus, NumExp 3)),
  CompoundStm(AssignStm("b",
      EseqExp(PrintStm[IdExp"a",OpExp(IdExp"a", Minus,NumExp 1)],
           OpExp(NumExp 10, Times, IdExp"a"))),
   PrintStm[IdExp "b"]))
(* a := (5 + 3); b := (print(a, (a - 1)), (10 * a)); print(b) *)
```


Abstract Syntax Trees in C

```
/************************ slp.h *************************/
typedef struct A_stm_ *A_stm;
typedef struct A_exp_ *A_exp;
typedef struct A_expList_ *A_expList;
typedef enum {A_plus,A_minus,A_times,A_div} A_binop;

struct A_stm_ {enum {A_compoundStm, A_assignStm, A_printStm} kind;
             union {struct {A_stm stm1, stm2;} compound;
                    struct {string id; A_exp exp;} assign;
                    struct {A_expList exps;} print;
                   } u;
            };
A_stm A_CompoundStm(A_stm stm1, A_stm stm2);
A_stm A_AssignStm(string id, A_exp exp);
A_stm A_PrintStm(A_expList exps);
```



```
struct A_exp_ {enum {A_idExp, A_numExp, A_opExp, A_eseqExp} kind;
             union {string id;
                    int num;
                    struct {A_exp left; A_binop oper; A_exp right;}
                    struct {A_stm stm; A_exp exp;} eseq;
                   } u;
            };
A_exp A_IdExp(string id);
A_exp A_NumExp(int num);
A_exp A_OpExp(A_exp left, A_binop oper, A_exp right);
A_exp A_EseqExp(A_stm stm, A_exp exp);

struct A_expList_ {enum {A_pairExpList, A_lastExpList} kind;
                   union {struct {A_exp head; A_expList tail;} pair;
                          A_exp last;
                         } u;
```



```
                  };

A_expList A_PairExpList(A_exp head, A_expList tail);
A_expList A_LastExpList(A_exp last);

/************************ slp.c *************************/
#include "util.h"
#include "slp.h"

A_stm A_CompoundStm(A_stm stm1, A_stm stm2) {
  A_stm s = checked_malloc(sizeof *s);
  s->kind=A_compoundStm; s->u.compound.stm1=stm1; s->u.compound.stm2
  return s;
}

A_stm A_AssignStm(string id, A_exp exp) {
  A_stm s = checked_malloc(sizeof *s);
  s->kind=A_assignStm; s->u.assign.id=id; s->u.assign.exp=exp;
```

```
  return s;
}

A_stm A_PrintStm(A_expList exps) {
  A_stm s = checked_malloc(sizeof *s);
  s->kind=A_printStm; s->u.print.exps=exps;
  return s;
}

A_exp A_IdExp(string id) {
  A_exp e = checked_malloc(sizeof *e);
  e->kind=A_idExp; e->u.id=id;
  return e;
}

A_exp A_NumExp(int num) {
  A_exp e = checked_malloc(sizeof *e);
```



```
  e->kind=A_numExp; e->u.num=num;
  return e;
}

A_exp A_OpExp(A_exp left, A_binop oper, A_exp right) {
  A_exp e = checked_malloc(sizeof *e);
  e->kind=A_opExp; e->u.op.left=left; e->u.op.oper=oper; e->u.op.rig
  return e;
}

A_exp A_EseqExp(A_stm stm, A_exp exp) {
  A_exp e = checked_malloc(sizeof *e);
  e->kind=A_eseqExp; e->u.eseq.stm=stm; e->u.eseq.exp=exp;
  return e;
}

A_expList A_PairExpList(A_exp head, A_expList tail) {
```



```
  A_expList e = checked_malloc(sizeof *e);
  e->kind=A_pairExpList; e->u.pair.head=head; e->u.pair.tail=tail;
  return e;
}

A_expList A_LastExpList(A_exp last) {
  A_expList e = checked_malloc(sizeof *e);
  e->kind=A_lastExpList; e->u.last=last;
  return e;
}
```


A concrete program

```
#include "util.h"
#include "slp.h"

A_stm prog(void) {

return
 A_CompoundStm(A_AssignStm("a",
                A_OpExp(A_NumExp(5), A_plus, A_NumExp(3))),
  A_CompoundStm(A_AssignStm("b",
      A_EseqExp(A_PrintStm(A_PairExpList(A_IdExp("a"),
                A_LastExpList(A_OpExp(A_IdExp("a"), A_minus,
                                      A_NumExp(1))))),
             A_OpExp(A_NumExp(10), A_times, A_IdExp("a")))),
   A_PrintStm(A_LastExpList(A_IdExp("b")))));
}
```

Abstract Syntax Trees in Java

```
abstract class Stm {}

class CompoundStm extends Stm {
   Stm stm1, stm2;
   CompoundStm(Stm s1, Stm s2) {stm1=s1; stm2=s2;}
}

class AssignStm extends Stm {
   String id; Exp exp;
   AssignStm(String i, Exp e) {id=i; exp=e;}
}

class PrintStm extends Stm {
   ExpList exps;
   PrintStm(ExpList e) {exps=e;}
```



```
}

abstract class Exp {}

class IdExp extends Exp {
   String id;
   IdExp(String i) {id=i;}
}

class NumExp extends Exp {
   int num;
   NumExp(int n) {num=n;}
}

class OpExp extends Exp {
   Exp left, right; int oper;
   final static int Plus=1,Minus=2,Times=3,Div=4;
```



```
   OpExp(Exp l, int o, Exp r) {left=l; oper=o; right=r;}
}

class EseqExp extends Exp {
   Stm stm; Exp exp;
   EseqExp(Stm s, Exp e) {stm=s; exp=e;}
}

abstract class ExpList {}

class PairExpList extends ExpList {
   Exp head; ExpList tail;
   public PairExpList(Exp h, ExpList t) {head=h; tail=t;}
}

class LastExpList extends ExpList {
   Exp head;
```



```
   public LastExpList(Exp h) {head=h;}
}

class prog {
static Stm prog =
new CompoundStm(new AssignStm("a",new OpExp(new NumExp(5), OpExp.Plu
                                            new NumExp(3))),
 new CompoundStm(new AssignStm("b",
     new EseqExp(new PrintStm(new PairExpList(new IdExp("a"),
                       new LastExpList(new OpExp(new IdExp("a"), OpE
                                                 new NumExp(1))))),
             new OpExp(new NumExp(10), OpExp.Times, new IdExp("a")))
  new PrintStm(new LastExpList(new IdExp("b")))));
}
```

Homework assignment

- Implement interpreter for straight-line programs (Appel, Chap 1 "PROGRAM")
- Submit your solution as a UNIX tar file to `mxh@dcs` by 20 Oct 2000, 5pm.
- The solution should contain
  - A Linux x86 ELF executable
  - Source code
  - Instructions on how to build the source code
- The program to be interpreted should be "hardwired" into the source. Don't write a lexer / parser (yet)!

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)

Lexing

```
void match0(char *s) /* find a zero */
{if (!strncmp (s, "0.0", 3))
    return 0.;
}
```

$\Downarrow$

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

Using Regular Expressions and DFA's

`(` $\rightarrow$ LPAREN
$digit\, digit^*$ $\rightarrow$ INT(ConvertToInt("matched text"))
`print` $\rightarrow$ PRINT
$letter(letter + digit + \{\_\})^*$ $\rightarrow$ ID("matched text")
...
where $digit = \{0, \ldots, 9\}$ and $letter = \{a, \ldots, z, A, \ldots, Z\}$.

Motto: Use regular expressions to say what you want, use DFA to match. Automatic lexer generator (lex, flex, JLex, ML-lex) performs conversion, plus some more stuff.

LEX disambiguation rules

Longest match:
The longest initial substring of the input that can match any regular expression is taken as the next token. See [Appel] on how that's implemented.

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.

`print0` $\rightarrow$ ID(`print0`) *not* PRINT INT(`0`)
`print` $\rightarrow$ PRINT *not* ID(`print`)

Lexer generators

Generates a lexer from a *lexer specification* (`.lex`-file).

The *lexer* takes the form of a module (class) containing a function (method) `nextToken` which when called returns the *result* associated with the next matched substring of the input file according to the specification in the `.lex`-file.

Typically, the "result" is the token name plus its value, but it can be anything else as well: the `.lex`-file contains rules of the form

$$\textit{regexp} \rightarrow \{\textit{code}\}$$

where *code* is (C, Java, ML)-code producing the "result". The code can refer to the matched (against *regexp*) string via special functions and variables, e.g. `yytext()` and `yypos`.

See documentation for details.

States

- You can specify a set of states in the `.lex`-file.
- Rules may be prefixed by a state meaning that they're used only when you are in that state.
- the *code* section may contain special state changing commands (See documentation).
- This can be used for (nested) comments:
  - `/*` gets you into a special "comment state".
  - If `/*` occurs when in "comment state" you increment a depth counter.
  - Seeing `*/` decrements the depth counter or brings you back to the "ordinary state".
  - Generate error message when `/*` or `*/` occur unexpectedly.

Practical

Write Lexer for Tiger '99.

- Use skeleton files from Andrew Appel if you wish.
- Report errors with line / column number.
- Handle nested comments
- Handle end-of-file, illegal characters, etc.

Submission

Submit by Monday, 23.10.00, 5pm a pointer to a directory or a UNIX .tar file containing:

- a `tiger.lex` file
- a main program which reads a file, lexes it, and generates output corresponding to the sequence of tokens in the file.
- some test cases
- a Makefile or similar
- a `README` file with the usual, sufficiently detailed instructions

You may use Andrew Appel's files in `$TIGER/chap2` which contain useful skeleton files. You must **modify** some parts to cater for Tiger '99!

Error reporting

In particular they contain an `ErrorMsg` module (class) which provides functions (methods)

`err`(pos, msg)
`newline`(pos)

Calling `err`(pos, msg) prints a message of the form

$$filename{:}line.col{:}msg$$

where *line* and *col* are line No. and column No. corresponding to the absolute position *pos* (characters since beginning of file).

For this to work properly, you must have called `newline` each time a new line was encountered with the then absolute position as argument.

Of course, you also need to setup the filename. See source code for more detail.

Context-free grammars

$$\text{CFG} = \text{reg. exp.} + \text{recursive definitions}$$

Recall ARDEN's Lemma: If $X = AX + B$ then $X = A^*B$; if $X = XA + B$ then $X = BA^*$.

Reg. exp. subsume recursive definitions where the recursive variable appears *at most once* (linear recursion) and only at the left or at the right *left linear*, *right linear*.

Definitions like $X = \epsilon + (X)$ or $X = num + X\ op\ X$ do not define regular languages.

A *context free grammar* is nothing but a system of recursive equations between regular expressions.

Context-free grammars, formally

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

$$symbol \to symbol \cdots symbol$$

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

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

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

A derivation

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

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?

Removing ambiguity

To resolve ambiguity in a grammar we define for operators their *precedence* 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. $F \to \mathrm{id}$
8. $F \to \mathrm{num}$
9. $F \to (\ E\ )$

Parsing by recursive descent

Consider the following grammar.

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

A recursive descent parser

```
enum token {IF, THEN, ELSE, BEGIN, END, PRINT, SEMI, NUM, EQ};
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

$S \to E\ \$$	$E \to E + T$	$T \to T * F$	$F \to$ id
	$E \to E - T$	$T \to T\ /\ F$	$F \to$ 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, $t \in$ 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
  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 nullable $[X] \leftarrow$ *true*
      if $Y_1 \cdots Y_{i-1}$ are all nullable
        then FIRST $[X] \leftarrow$ FIRST $[X] \cup$ FIRST $[Y_i]$
      if $Y_{i+1} \cdots Y_k$ are all nullable
        then FOLLOW $[Y_i] \leftarrow$ FOLLOW $[Y_i] \cup$ FOLLOW $[X]$
      if $Y_{i+1} \cdots Y_{j-1}$ are all nullable
        then FOLLOW $[Y_i] \leftarrow$ FOLLOW $[Y_i] \cup$ FIRST $[Y_j]$
until FIRST, FOLLOW and nullable did not change in this iteration

Remark on FOLLOW sets

The above algorithm assumes that each nonterminal X is actually reachable from the start symbol, i.e. there exists a derivation $S \to \beta X \gamma$.

For a counterexample consider

$$S \to Yb$$
$$X \to Yc$$

Now FOLLOW$(X) = \{b\}$, but the algorithm gives $\{b, c\}$.

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 $T \in$ FIRST(γ). Also, if γ is nullable, enter the production in column T of row X for each $T \in$ 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*.

Summary on LL(1) parsing

- The first input symbol determines which production is to be used for it *and the following symbols*.
- The parse tree is generated starting from the start symbol ("top-down").
- Decision table on which production to use can be read off from FIRST and FOLLOW sets.
- If no decision is possible one can sometime tune the grammar by left factoring and similar techniques.
- The *big advantage* of LL(1) parsing is that it's easy to implement: It is the method of choice when no automatic parser generator is available, e.g. when programming in a scripting language.

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* σ which is a sequence of grammar symbols
  - and *remaining input* r which is a sequence of terminals.

  At any time we have $\sigma r \rightarrow w$ where w is the original input.

Actions of an LR Parser

- The parser performs the following two kinds of actions.

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

  Reduce: Choose a grammar rule $X \rightarrow \gamma$; pop the sequence γ off the stack, push X onto the stack.

  Notice: reducing in the middle of the stack is forbidden.
- The input is accepted when the stack contains only the start symbol and the remaining input is empty.
- An LR(k) parser decides which action to perform on the basis of the stack and the first k tokens of the remaining input. A grammar is LR(k) if that is possible. Only LR(1) is of practical relevance.

Miracle of LR - parsing

For each production $X \rightarrow \gamma$ and first (k) input token(s) a the set of those stacks in which one should reduce by $X \rightarrow \gamma$ is a regular language.

I.e. there is a finite automaton which by operating on the stack tells us which action to take; more precisely, the final state reached after reading the stack determines the action.

Example

(1)E →E + T (2)E→T
(3)T →T * F (4)T→F
(5)F →id (6)F→(E)

	0	1	2	3	4	5	6	7	8	9	10	11
0												
1	E											
2	T				T							
3	F				F		F					
4	(				(		(	(				
5	id				id		id	id				
6		+							+			
7			*							*		
8					E							
9							T					
10								F				
11								F	)			
	S	S	R2*	R4	S	R5	S	S	S	R1*	R3	R6

S = shift. Rx = reduce by (x). Rx* = reduce by (x) or shift *.

Optimisation: LR transition tables

- Annotate stack elements with the state reached at their position (One can then even remove the grammar symbols altogether and have a stack of states!)
- Set up 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	Accept input.

Errors are denoted by blank entries in the table.

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 (token and) 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.

Construction of automaton for LR(1) parsing

States: Pairs of productions with position and terminals or \$:

$(X \to \alpha.\beta, a)$ for instance $(E \to E. + T, *)$
$(X \to \alpha.\beta, \$)$ for instance $(E \to E + T., \$)$

The states are called *LR(1) items*.
Transitions:

- $X \to \alpha.s\beta, a) \longrightarrow^{s} (X \to \alpha s.\beta, a)$, s any grammar symbol,
- $(X \to \alpha.Y\beta, a) \longrightarrow^{\epsilon} (Y \to .\gamma, b)$, if $Y \to \gamma$ and $b \in \text{FIRST}(\beta a)$

Here $\text{FIRST}(\beta\$) = \begin{cases} \text{FIRST}(\beta) \cup \{\$\}, \text{ if } \beta \text{ is nullable} \\ \text{FIRST}(\beta), \text{ otherwise} \end{cases}$

When reaching state $(X \to \alpha., a)$ one reduces by $X \to \alpha$ provided the next input token is a.

Determinise using subset construction. States with containing conflicting items mean that grammar was not LR(1).

Verification of the construction

Define for each LR(1)-item $(X \to \alpha.\beta, a)$ a "language" $\mathcal{L}(X \to \alpha.\beta) \subseteq (N \cup T)^*$ by

$$\mathcal{L}(X \to \alpha.\beta, a) = \{\gamma\alpha \mid S \to_{\text{rm}} \gamma X a w \text{ for some } w \in T^*\}$$
$$\mathcal{L}(X \to \alpha.\beta, \$) = \{\gamma\alpha \mid S \to_{\text{rm}} \gamma X\}$$

Here $\gamma \in (N \cup T)^*$ and $\to_{\text{rm}}$ denotes *rightmost derivations*. (A derivation is rightmost if at each step the rightmost nonterminal gets expanded.)

Notice that $\mathcal{L}(X \to \alpha., a)$ is the set of those stacks which must be reduced by $X \to \alpha$ if the next token is a.

Proposition: Assume that there is only one production involving the start symbol. Then

$$\mathcal{L}(X \to \alpha s.\beta, a) = \mathcal{L}(X \to \alpha.s\beta, a)s$$
$$\mathcal{L}(S \to .\gamma, \$) = \{\epsilon\}$$
$$\mathcal{L}(X \to .\gamma, a) = \cup_{a \in \text{FIRST}(\beta b)} \mathcal{L}(Y \to \alpha.Y\beta, b)$$

with appropriate definition of $\text{FIRST}(\beta\$)$.

These equations are direct consequences from the definitions of the $\mathcal{L}(-)$ sets.

It follows directly from the definition of the automaton that if $(S \rightarrow .\gamma, \$) \rightarrow^{\sigma} (X \rightarrow \alpha.\beta, a)$ then $\sigma \in \mathcal{L}(X \rightarrow \alpha.\beta, a)$.

Therefore, if the automaton applied to stack σ lands in state $(X \rightarrow \gamma., a)$ and the next token is a one may reduce by $X \rightarrow \gamma$.

SLR(1) and LALR(1)

LR(1) parsing tables are very big (several 1000 states for a typical programming language).

LALR(1) is an optimisation which merges LR(1) states which differ only in the lookahead symbol. This reduces their number to several hundred for a typical programming language.

SLR dispenses with lookahead symbols in states altogether and uses FOLLOW sets to resolve conflicts.

Not every LR(1) grammar is LALR(1) or SLR(1). Most reasonable programming languages have a LALR(1) grammar.

Summary on LR parsing

- LR parsers generate a parse tree starting from the leaves ("bottom up") by *shifting* input symbols on a stack or *reducing* the beginning of a stack according to a production.
- The shift/reduce decision is determined by the state of a DFA operating on the stack and by the next input token.
- LR parsers are more general than predictive parsers (e.g. no problem with left recursion).
- LALR(1) and SLR(1) are optimisations of LR(1).

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

  user declarations (e.g. `package` statements, helper functions)
  %%
  parser declarations (e.g. sets of terminals / nonterminals)
  %%
  grammar rules

- The grammar rules are productions of the following form.

  `exp :  exp PLUS exp      (` *semantic action* `)`

An example grammar

Stm→*Stm*;*Stm*	(CompoundStm)		
Stm→`id :=`*Exp*	(AssignStm)	*ExpList*→*Exp* `,` *ExpList*	(PairExpList)
Stm→`print (`*ExpList*`)`	(PrintStm)	*ExpList*→*Exp*	(LastExpList)
Exp→`id`	(IdExp)	*BinOp*→`+`	(Plus)
Exp→`num`	(NumExp)	*BinOp*→`-`	(Minus)
Exp→*Exp* *BinOp* *Exp*	(OpExp)	*BinOp*→`*`	(Times)
Exp→`(`*Stm* `,` *Exp*`)`	(EseqExp)	*BinOp*→`/`	(Div)

Implementation in ML Yacc

```
<here go declarations + helper functions>
%%
%term
    EOF   | ID of string | NUM of int | COMMA | SEMI | LPAREN | RPAREN
  | PLUS | MINUS | TIMES | DIV | ASSIGN | PRINT

%nonterm  exp of exp | stm of stm | explist of exp list | prog of stm

%right SEMI COMMA
%left PLUS MINUS
%left TIMES DIV

%verbose %start prog %eop EOF %noshift EOF

%name Parser
```



```
%%

prog   : stm (stm)

stm    : stm SEMI stm ()
       | ID ASSIGN exp ()
       | PRINT LPAREN explist RPAREN ()

explist: exp ()
       | exp COMMA explist ()

exp:     NUM ()
       | ID  ()
       | exp PLUS exp ()
       | exp MINUS exp ()
       | exp TIMES exp ()
       | exp DIV exp ()
       | stm COMMA exp ()
       | LPAREN exp RPAREN ()
```


Implementation in CUP

```
package straight;
import java_cup.runtime.*;

action code  {: <here go helper functions for the actions> :}
parser code  {: <here go helper functions for the parse> :}
scan with    {: return lexer.nextToken(); :};

terminal String ID; terminal Integer INT;
terminal COMMA, SEMI, LPAREN, RPAREN, PLUS, MINUS,
         TIMES, DIVIDE, ASSIGN, PRINT;

non terminal Exp exp;
non terminal ExpList explist;
non terminal Stm prog;

precedence left SEMI;
precedence nonassoc ASSIGN;
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

```
start with prog;

prog ::= stm {: :}

exp ::= exp TIMES exp {: :}
       |exp DIVIDE exp {: :}
       |exp PLUS exp {: :}
       |exp MINUS exp {: :}
       |INT {: :}| ID {: :}
       |LPAREN stm COMMA exp RPAREN {: :}
       |LPAREN exp RPAREN {: :}

explist ::= exp {: :}
           |exp COMMA explist {: :}

stm ::= stm SEMI stm {: :}
       |PRINT LPAREN explist RPAREN {: :}
       |ID ASSIGN exp {: :}
```


Precedence directives

The above grammar is ambiguous, so there does not exist an LR(1)-parser!

Precedence directives are used to resolve the resulting conflicts.

The directive

```
%right SEMI COMMA
%left PLUS MINUS
%left TIMES DIV
```

says that `TIMES, DIV` bind stronger than `PLUS, MINUS` which bind stronger than `SEMI, COMMA`. `PLUS, MINUS, TIMES, DIV` associate to the left, `SEMI, COMMA` associate to the right.

This means, e.g., that $s1; s2; s3, x+y+z*w$ becomes $(s1; (s2; s3)), ((x+y) + (z * w))$

Unary minus

- Precedence directives apply to tokens; sometimes one wants to give precedence to a production.
- A typical example is $Exp \rightarrow -Exp$.
- We want to read $-e1 + e2$ as $(-e1) + e2$.
- So one introduces a dummy token `UMINUS` and gives it highest precedence. With

  ```
  exp: MINUS exp %prec UMINUS
  ```

  the corresponding production "inherits" the precedence from `UMINUS`.

Conflicts

The remaining conflicts are reported as *shift-reduce* and *reduce-reduce* conflicts.

- A shift-reduce conflict is a choice between shifting and reducing.
- A reduce-reduce conflict is a choice of reducing by two different rules.

By convention shift-reduce conflicts are resolved by shifting and reduce-reduce conflicts are resolved by using the rule which appears earliest in the grammar.

In most cases these conflicts highlight a problem with the grammar.

Only exception in Tiger'99: "dangling else."

Analyse all conflicts using the verbose description file (contains the LR(1) states).

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. (In CUP this type is always `Object`.)
- For a rule $A \to B\ C\ D$, 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.

Abstract syntax

The semantic actions can be used to perform arbitrary stuff, we could e.g. put the entire remainder of the compiler between the {: :} or ( ) delimiters.

However, it is more practical to use the semantic actions to generate a tree-like representation of the program known as *abstract syntax*.

$$\frac{\texttt{class Stm...}\quad\texttt{class Exp...}}{\text{straight line prog.}} = \frac{\text{abstract syntax for language } L}{\text{language } L}$$

Position arguments must be made part of the abstract syntax objects to enable subsequent error reporting.

```
class Stm {      class Exp {    datatype stm =
  int pos;         int pos;       CompoundStm of stm*stm*int
}                }                      ...
...                   ...
```


Semantic actions in ML-Yacc

```
open AbSyn;
%%
%nonterm exp of exp | stm of stm | explist of exp list | prog of stm
%%
prog   : stm (stm)
stm    : stm SEMI stm (CompoundStm(stm1, stm2, SEMIleft))
       | ID ASSIGN exp (AssignStm(ID, exp, ASSIGNleft))
       | PRINT LPAREN explist RPAREN (PrintStm(explist,PRINTleft))
explist: exp ([exp])
       | exp COMMA explist (exp :: explist)
exp:     NUM (NumExp(NUM, NUMleft))
       | ID  (IdExp(ID, IDleft))
       | exp PLUS exp (OpExp(exp1, Plus, exp2, PLUSleft))
       | exp MINUS exp (OpExp(exp1,Minus, exp2, MINUSleft))
       | exp TIMES exp (OpExp(exp1,Times, exp2, TIMESleft))
       | exp DIV exp (OpExp(exp1, Div, exp2, DIVleft))
       | stm COMMA exp (EseqExp(stm, exp, COMMAleft))
       | LPAREN exp RPAREN (exp)
```


Explanation

- Refer within action by `X` to the value of grammar symbol `X` in corresponding rule. Use `X1, X2, ...` to disambiguate.
- If `X` is a token then `Xleft`, `Xright` refer to position of beginning and end of token in input file.

  The parser generates a datatype of tokens which embodies this information. It's the responsibility of the lexer to fill it in.
- Definition of abstract syntax must go into a structure, here `AbSyn`.
- To use the parser one must invoke a magic incantation involving the "Compilation Manager", `structure`s, `functor`s, etc. See `parsetest.sml` and documentation.

  At the end, one gets a function `parse:string -> AbSyn.Stm` which takes a filename and returns abstract syntax.
- (In OCaml it's a lot simpler)

Semantic actions in CUP

```
exp ::= exp:e1 TIMES:t exp:e2
      {: RESULT= new OpExp(e1, OpExp.Times, e2, tleft); :}
...
stm ::= prog:s1 SEMI:t prog:s2
      {: RESULT = new CompoundStm(s1, s2, tleft); :}
```

Explanation

- You can choose the names by which to refer to values
- all values are `Objects`
- compiling the specification generates a `.java` file containing a `Parser` class parametrised by a lexer. It has a method `parse()` returning a symbol, i.e., an `Object` with positional information.

Abstract Syntax for Tiger '99

- To be adapted from abstract syntax for Tiger as provided.
- Identifiers are represented as symbols:

```
datatype var = SimpleVar of symbol * pos

and exp = VarExp of var
        | CallExp of {func: symbol, args: exp list, pos: pos}
          ...
```


Practical (Chap3 and 4 of [Appel])

- Define abstract syntax for Tiger '99. You only need to change the existing one for Tiger proper.
- Write parser for Tiger '99 returning abstract syntax.
- Combine parser with lexer (read documentation carefully!)
- Write test scaffolding, e.g., a function / method which prints abstract syntax. Again, you can build on the provided one for Tiger.
- Possible conflicts in your grammar should be carefully explained.
- No fancy error correction is required. You can stop the parse at the first syntax error.
- But make sure file positions are propagated properly.
- Submit your code by Friday, 10. November 2000, 5pm

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

Symbols and symbol tables

- To avoid unnecessary string comparisons in our hash tables we 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.)

Environment entries

- For each value identifier we need to know whether it is a variable or a function. If it is a variable we normally need to know its type (in Tiger '99 there is only one type, though). If it is a function we need to know its parameters and result types. The variable environment is initialised to include definitions of predefined functions.
- As variables and functions are declared, the type-checker augments the environments. The environments are consulted for each identifier found when processing expressions.
- In languages with user-defined types one also needs type environments. If the same symbol can be both a type and a variable according to context then a separate environment is needed. More generally, each namespace requires its own environment.
- Module identifiers (class names, structure names) get mapped to symbol tables themselves!

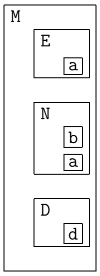
Multiple environments in ML and Java

```
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.



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

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.

```
type env = Env.enventry Symbol.table

transVar : env -> Absyn.var -> expty
transExp : env -> Absyn.exp -> expty
transDec : env -> Absyn.dec -> env
```

In C :

```
struct expty transVar(S_table env, A_var v);
struct expty transExp(S_table env, A_exp a);
void transDec(A_dec,  S_table env);
```

The `expty` type contains a translated expression and its Tiger-language type. For now `expty = {exp:unit, ty:Types.ty}`. Since there are no types in Tiger'99 you can even use `expty = unit`.

Functional symbol table

```
type 'a table
val empty : 'a table
val enter : 'a table * symbol * 'a -> 'a table
val look : 'a table * symbol -> 'a option
```

In Java (similarly in C):

```
Table empty();
static Table enter(Table env,
                          Symbol s,  Object b);
static Object look(Table env, Symbol s);
```

Can be implemented as persistent binary search tree, e.g., red-black tree (see Cormen et al.).

Caveat when using this with Java / C: After `s2 = enter(s1, x, b)` the tables `s1` and `s2` are *aliased*.

Imperative symbol table

```
type 'a table
val new : unit -> 'a table
val enter : 'a table * symbol * 'a -> unit
val look : 'a table * symbol -> 'a option
val beginScope : 'a table -> unit
val endScope : 'a table -> unit
```

In Java:

```
public class Table {
  public Table(); (* generate table *)
  public void put(Symbol key, Object value);
  public Object get(Symbol key);
  public void beginScope();
  public void endScope();
```

Can be implemented as hash table with stack.

Type-checking arithmetic expressions (in ML)

```
fun transExp(env,
               Absyn.OpExp{left,oper=Absyn.PlusOp,right,pos}) =
    let val {exp=_, ty=tyleft}  = transExp(env, left)
        val {exp=_, ty=tyright} = transExp(env, 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
```


Type-checking arithmetic expressions (in C)

```
struct expty transExp(S_table env,  A_exp a) {
  switch (a->kind) {
      ⋮
    case A_opExp: {
      A_oper oper = a->u.op.oper;
      struct expty left = transExp(env, a->u.op.left);
      struct expty right= transExp(env, 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.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);
  }
}
```


Type-checking variables (in C)

```
struct expty transVar(S_table venv, S_table tenv, A_var v) {
  switch (v->kind) {
    case A_simpleVar: {
      E_enventry x = S_look(env,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:
      ⋮
  }
}
```

Typechecking let expressions in ML

```
let fun transExp(env, a) = case a of
    letExp(_,d,e) => let env1 = transDec(env,d) in
            transExp(env1,e)
```


Type-checking let-expressions in C

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


Type-checking variable declarations in C

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


Type-checking declarations in ML

```
type dec = FunctionDec of {name: symbol, params: field list, body: exp,pos:pos}
       | VarDec of {name: symbol, init: exp, pos: pos}
and field = name: symbol

fun  transDec env Absyn.VarDec{name,init=e,pos} =
              let val {exp,ty} = transExp env e in
              Symbol.enter(env,name,Env.VarEntry{ty=ty})
    | transDec env Absyn.FunctionDec{name,params,body} =
              let fun add_field(f,env) =
                     Symbol.enter(env,f#name,Env.VarEntry{ty=Types.INT})
                  val newenv = foldr add_field env params
                  val _ = (* make sure params contains no duplicates *)
                  val e' = transExp env body
              in Symbol.enter(newenv,name,Env.FunEntry{
                  formals=map (fn f=>Types.INT) params,
                  result=Types.INT})
```

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.

Translation to intermediate code

Aim: make semantic analysis return intermediate code.

This will be achieved by the interplay of three modules:

- `Translate`: provides a function for each program construct which generates intermediate code for that construct. These functions are called in the various clauses of `transExp`.
- `Temp`: provides an interface to "temporaries" (abstract registers or memory locations) and "labels" (assembler-level jump addresses).
- `Frame`: provides an interface to the (machine-dependent) frame layout.

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.

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.

Higher-order functions and locality

In a language which supports both nested functions *and* function-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 or there is a *de facto* standard. Using this allows calls to functions written in other programming languages.

The frame pointer

- Suppose a function $g(\ldots)$ calls the function $f(a\_1, \ldots, a\_n)$. We say that g is the *caller* and f is the *callee*.
- On entry to f, the stack pointer SP points to the first free location on the stack. On entry, f saves the frame pointer FP on the stack (the *dynamic link*) and makes the current stack pointer the new frame pointer. On exit the effect of this is reversed: the contents of FP are moved to SP thus discarding SP; the contents of FP are restored by popping the stack.
- 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*.
- It will e.g. *n*ot be fixed if we support C's `alloca()`.

Registers

- Registers are extremely fast memory built into a processor.
- 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.
- Modern RISC machines such as Alpha, PowerPC, Sparc have a large set of general purpose 32-bit *registers* (typically 32 of them). The Pentium has only 8 integer registers (some of which are dedicated.)
- Often arithmetic operations are possible only on registers, so that data in memory must be loaded prior to and stored after an operation. The Pentium has instructions for arithmetic operations on memory contents but these take as many cycles as a load-op-store sequence (namely 3).

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 or there is a *de facto* standard.
- 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.
- On the Pentium parameters are usually passed on the stack.

Call-by-reference

- A more dignified way to take the address of a local variable is 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.

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. On the Pentium the return address is pushed on the stack.

Frame-resident variables

After applying considerable ingenuity to avoid unnecessary memory 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. In Tiger '99 no variable escapes!

Static links

- In languages which allow nested functions (Pascal, ML, Java, Tiger) (not Tiger '99!, not C), 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*, see Ex. 6.7.
  - 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*. Done, e.g., in the Pascal to C translator `p2c`.

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`). 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

```
signature FRAME =
sig type frame
    type access
    val newFrame : (name: Temp.label, formals: bool list) -> frame
    val name : frame -> Temp.label
    val formals : frame -> access list
    val allocLocal : frame -> bool -> access
end
```


Frame creation and use

- The type `frame` holds information about formal parameters and local variables. To make a new frame for a function f with k formal parameters, call `newFrame` $f\ \ l$, where l is a list of k booleans recording whether or not each parameter escapes.
- The type `access` implements abstract handles to variable references. The `Translate` module contains a function turning an "`access`" into actual code.
- The `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*.

The abstract access type

The `access` type describes formal parameters and local variables which may be in the frame or in registers. The contents of `access` are only visible inside the `Frame` module.

```
structure i386Frame : FRAME = struct
   datatype access = InFrame of int | InReg of Temp.temp
end
```

`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.

Frame descriptions

Suppose that g has three parameters.

On a Pentium and with a physical frame pointer the parameters would be passed thus:

`InFrame`(8) `InFrame`(12) `InFrame`(16)

The view shift would be implemented in this way:

$$M[\text{sp}] \leftarrow \text{fp}$$
$$\text{fp} \leftarrow \text{sp}$$
$$\text{sp} \leftarrow \text{sp} - K$$

Without physical frame pointer they would be passed thus

`InFrame`(4) `InFrame`(8) `InFrame`(12)

with view shift:

$$\text{sp} \leftarrow \text{sp} - K$$

If the first parameter would be passed via register t_{84} then the parameters would be accessed thus:

`InReg`(t_{84}) `InFrame`(4) `InFrame`(8)

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 `allocLocal(`f`, esc)` where the boolean argument `esc` specifies whether the local variable escapes. Tiger '99: no 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.

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.
  - `newtemp : unit->temp` return a new temporary
  - `newlabel : unit->label` return a new label
  - `namedlabel : string->label` returns a new label of a given assembly language name.

Environment entries revisited

During the semantic analysis we make calls to the `FRAME` module when we encounter variable or function declarations and store the results in the symbol table.

```
/* New versions of VarEntry and FunEntry */
datatype enventry =
    VarEntry of {access:access, ty:ty}
    FunEntry of {(*level: Translate.level *)
                 label: Temp.label,
                 formals: ty list, result: ty}
```

With nested functions we also need to maintain information on the nesting depth ("level"). Fortunately, don't have those in Tiger '99.

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.

Intermediate representation trees

A good intermediate representation has several qualities.

- 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. Used for string literals (not in T99), abbreviations such as frame size, and computed jump addresses.

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.

Remark: IR expressions are *untyped*. For "C" some type info. would be required.

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.

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

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

Kinds of expressions in ML

```
/* in translate.sml */
  datatype exp   = Ex of Tree.exp
                     | Cx of label * label -> Tree.stm

                     | Nx of Tree.stm
```

When using C or Java rather don't bother with different kinds of expressions, i.e., use `T.exp` throughout.

Conversion functions

```
fun unEx (Ex e) = e
  | unEx (Cx genstm) =
        let val r = newtemp() and t = newlabel() and f = newlabel()
         in T.ESEQ(seq[T.MOVE(T.TEMP r, T.CONST 1),
                       genstm(t,f),
                       T.LABEL f,
                       T.MOVE(T.TEMP r, T.CONST 0),
                       T.LABEL t],
                   T.TEMP r)
        end
  | unEx (Nx s) = T.ESEQ(s,T.CONST 0)



fun unCx(Ex(T.CONST 0)) = (fn (t,f) => T.JUMP(T.NAME f,[f]))
  | unCx(Ex(T.CONST _)) = (fn (t,f) => T.JUMP(T.NAME t,[t]))
  | unCx (Ex e) = (fn (t,f) => T.CJUMP(T.NE,e,T.CONST 0, t,f))
  | unCx (Cx c) = c
  | unCx (Nx s) = ErrorMsg.impossible "unCx(Nx _)"
```



```
fun unNx (Ex e) = T.EXP e
  | unNx (Cx genstm) = let val l = newlabel()
                        in T.SEQ(genstm(l,l),T.LABEL l) end
  | unNx (Nx s) = s
```

Translating an expression

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

```
let  val z = Temp.newlabel() in
  Cx(fun (t,f) =>
   Tree.Seq(Tree.Cjump(Tree.gt, a, b, t, z),
       Tree.Seq(Tree.Label(z),
           Tree.Cjump(Tree.lt, c, d, t, f))))
```

The destinations t and f are not known yet, although `z` is.

Alternatively, you can translate all logical expressions as integers and only use `Tree.eq`

Simple variables

- The semantic analysis phase has a function which type-checks a variable in the context of an environment `env`. 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

  MEM(BINOP(PLUS, TEMP `fp`, CONST k))

  where k is the offset of v within the frame (negative for local vars, positive for arguments) 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 `Translate.exp` is an abstract data type, whose `Ex`, `Nx`, and `Cx` 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

  ```
  simpleVar : access (* -> level*) ->  Translate.exp
  ```

  to the `Translate` interface. Now `Semant` can pass in the `access` of x (obtained from `allocLocal`) and get back a `Translate.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.sig */
   ⋮
 val FP : Temp.temp  (* = 2 *)
 val RV : Temp.temp   (* = 3 *)
 val wordSize : int      (* = 4 *)
 val exp : access -> Tree.exp -> Tree.exp
```

Accesses to `FP` can be translated into `SP`framesize+ during code generation.

Using the Frame module

- The function `exp` is used by `Translate` to turn an `access` into a `Tree` expression.
- Thus for an access a such as `InFrame` k, we have

  `Frame.exp(`a`, Tree.Temp(Frame.FP))`

  returns the result

  MEM(BINOP(PLUS, TEMP `fp`, CONST k))

Translating conditional expressions

```
fun ifThenElse(cond,a,b) =
    let val t = newlabel() and f = newlabel()
        val join = newlabel()
        val r = newtemp()
           in Ex(T.ESEQ(seq[unCx cond (t,f),
        T.LABEL t,
        T.MOVE(T.TEMP r, unEx a),
        T.JUMP(T.NAME join,[join]),
        T.LABEL f,
        T.MOVE(T.TEMP r, unEx b),
        T.LABEL join],
        T.TEMP r))
end
```

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.

We might also cater specially for the case where e_1 is a `Cx`.

Calling runtime-system functions

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

  CALL (NAME(`Temp_namedlabel("printch")`),
  `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. We encapsulate all of this in the `Frame` module, as usual.

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


While loops

- The general layout of a **while**-loop is this.

  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.
- a while loop with an explicit label gets that label attached to it. Improbable problem: that very label could already have been issued.
- labelled **break** statements get translated as unconditional jumps.

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:

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

Here l_f is the label for f and the e_i are the translations of the argument list a_i.

With nested functions we must also pass the static link.

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 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.

```
(* in frame.sig *)
val  RV : Temp.temp
```

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

```
(* in frame.sig *)
val procEntryExit1 : frame * Tree.stm -> Tree.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.sml *)
signature CANON =
sig
    val linearize : Tree.stm -> Tree.stm list
    val basicBlocks : Tree.stm list -> (Tree.stm list list * Tree.label)
    val traceSchedule : Tree.stm list list * Tree.label -> Tree.stm
    list
```

`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.

Canonical trees

- We define *canonical trees* as those intermediate representation 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

BINOP(PLUS, MEM(z), ESEQ(MOVE(MEM(x), y), MEM(x)))

with a subtree such as

ESEQ(MOVE(MEM(x), y), BINOP(PLUS, MEM(z), 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.

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.

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

Intel x86 Assembly Language: Registers

- The ix86 (in 32bit mode) has 8 registers: `%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp`
- `%esp` is the *stack pointer*. Don't use it for anything else.
- `%ebp` can be used as the *frame pointer*. If you don't need a frame pointer `%ebp` becomes general purpose.
- `%eax, %edx, %ecx` are *caller-save*
- `%ebx, %esi, %edi, %ebp` are *callee-save*

Operands

An *operand* is either

- a register
- a constant
- a memory location
- a *constant* is written as `$`n where n is a number. The *assembler* allows you to write an arithmetic expression involving decimal numbers and abbreviations instead of an explicit number.
- a *memory location* is written as n`(`*reg*`)` where n is a number or an arithmetic expression as above and *reg* is a register. It refers to the memory location "contents of register *reg* + n".

Move instructions

A *move instruction* takes the form

`movl` *src*,*dst*

Its effect is to move the contents of *src* to the *dst*. The source *src* may be any operand; the destination *dst* may not be a constant. Additionally, *src* and *dst* may not both be memory locations.

```
framesize = 48
     movl     %ebx,-4+framesize(%esp)
     movl     $33,%eax
     movl     (%ecx),%esi
```


Arithmetic instructions

- Addition, subtraction, and multiplication have the general format

  op src,*dst*

- Here *op* is one of `addl` (addition), `subl` (subtraction), `imull` (multiplication).
- The effect is

  $$src\ op\ dst \rightarrow dst$$

  so the result always ends up in the second operand.
- the operands *src* and *dst* undergo the same restrictions as in the move instructions.

Division (yuk)

The format of the *division instruction* is

```
cdq
idivl op
```

The effect of this is to place the result of placing the contents of `%eax` divided by *op* into `%eax`.

As a side-effect the register `%edx` becomes trashed (overwritten with garbage).

The operand *op* must be a register or a memory location (not a constant).

Load effective address

The following instruction

`leal` n_1(,*reg*,n_2), *dst*

moves $n_1 + reg_2 \times n_2$ into *dst*.

n_2 must be among $1, 2, 4, 8$.

The use of `leal` is to be preferred over `imull` and `addl` with constants.

Multiplication by 7 may be simulated by a `leal` followed by a `subl`, etc.

Shift instructions

- These have the format

  $sop\ op_1, op_2$

- *sop* is one of `sall` (left shift), `shrl` (right shift), `sarl` (arithmetic right shift)
- op_1 must be a constant
- op_2 must not be a constant.
- The effect is to shift the contents of op_2 by op_1.

Labels and unconditional jumps

- any instruction can be labelled by preceding it with an identifier followed by a colon

```
f:    subl    $main_framesize,%esp
L73:  movl    $35,%eax
```

- An *unconditional jump* instruction has the form

  `jmp` *label*

Conditional jumps

- A *conditional jump* takes the form

  `cmp` op_1, op_2
  jop label

- here *jop* is one of `je` (equal), `jne` (not equal), `jl` (less than), `jg` (greater than), `jle` (less or equal), `jge` (greater or equal), `jb` (unsigned less), `jbe` (unsigned less or equal), `ja` (unsigned greater), `jae` (unsigned greater or equal).
- the effect of the above instruction is to jump to *label* if the two operands stand in the relation denoted by *jop*. Otherwise execution continues with the following instruction (the jump "falls through").

Calling and returning

- The instruction

  `call` *label*

  jumps to *label* and pushes the return address (memory address of the following instruction) on the stack (thus subtracting 4 from `%esp`.
- The instruction

  `ret`

  jumps to the memory location on top of the stack, i.e., at location `(%esp)` and adds 4 to `%esp`.
- Return values should be passed in `%eax`

There are a few hundred other instructions...

Interfacing with the rest of the world

- a standalone assembler program needs a label `main` and the directive

  ```
  .globl main
  ```

- using `call` you can call other functions, e.g. written in `C`. Integer or pointer arguments to these must be placed in `(%esp)`, `4(%esp)`, `8(%esp)` in that order, i.e., the first argument is in `(%esp)`. The return value will be in `%eax`.
- to link your assembler program `prog.s` with a `C` program `library.c` defining these external functions you do

  ```
  cc -c library.c
  cc -o runme prog.s library.o
  ```

- The first of these commands compiles `library.c` to an object file `library.o`. The second one links `prog.s` with `library.o` to form an executable `runme` which you can run with

  ```
  ./runme
  ```


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.
- during this phase *machine instructions* are assembly language instructions with *temporaries* instead of real registers. Register allocation then takes care of mapping these onto the real registers or "spilling" them into the frame.

A tiling example

- We will consider tiling the following tree

  MOV(MEM(BINOP $(+, a,$ BINOP$(*, i, 4)))$,MEM(x))

  where i and a are temporaries.

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

  `movl` t_2,r
  `addl` t_1,r

  where r is a fresh temporary, has one node.
- The tile for

  `movl` t_2,r
  `addl` i+j(t_1),r

  has five nodes (two ADD, one MEM, two CONST).
- Additionally, we could take into account the number of cycles or the number of auxiliary registers.
- 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.

The structure Assem

- Serves as the target of instruction selection
- it provides a datatype

```
datatype instr = OPER of {assem: string,
                          dst: temp list,
                          src: temp list,
                          jump: label list option}
               | LABEL of {assem: string, lab: Temp.label}
               | MOVE of {assem: string,
                          dst: temp,
                          src: temp}
```

- The idea is that each of these results in a string, namely the contents of the `assem` field with `` `s0, `s1, ... `` and `` `d0, `d1, ... `` and `` `j0, `j1, ... `` replaced with the contents of the `dst`, `src`, `jump` fields, respectively.
- For instance

```
A.OPER{assem="\taddl\t`s0," ^ Int.toString i ^
       "+" ^ name ^ "(`d0)\n", src=[a],  dst=[b],jump=NONE}
```

  results in

```
addl   t134, 4+main_framesize(t67)
```

  in case `i` was 4, `a` was `t134`, `b` was `t67`, `name` was `main_framesize`.

A few words on registers

Define in `Frame` a list of designated temporaries which may be trashed, e.g., `EAX, EDX, ESI,\dots` and enter their corresponding strings, e.g., `%eax` in `Temp.table`.

Accordingly, no tile should return its result in one of these.

Define lists of caller save and callee save registers.

The callee save ones must be saved and restored on function entry/exit (by moving them to and from fresh temporaries).

The caller save ones are trashed by function calls, thus should appear in the `dst` list of call instructions.

Maximal munch by pattern matching

Write functions

```
munchExp : Tree.exp -> Temp.temp
munchStm : Tree.stm -> unit
```

As a side effect these functions generate a global list of instructions using `emit`

```
val ilist = ref (nil:Assem.instr list)
fun emit x = ilist:=x::ilist
```

`ilist` contains the instructions in reverse order.

```
 munchStm(T.SEQ(a,b)) =
|munchStm(T.JUMP(T.NAME lab, labs)) =
|munchStm(T.MOVE(T.TEMP t, T.CALL(e,el))) = (call(e,el); move(t,F.RV))
|munchStm(T.MOVE(T.MEM(e),b)) =
|munchStm(T.MOVE(T.TEMP t, e)) =
```



```
|munchExp(T.BINOP(T.PLUS,T.CONST i,T.TEMP t)) = (* Catch t=F.FP *)
|munchExp(T.BINOP(T.PLUS,T.TEMP t,T.CONST i)) =
|munchExp(T.BINOP(T.PLUS,MEM a,T.CONST i)) =
|munchExp(T.BINOP(T.PLUS,T.CONST i, T.MEM a)) =
|munchExp(T.BINOP(a,b)) = (* Should always work *)
|munchExp(T.MEM(T.TEMP(F.FP))) =
|munchExp(T.BINOP(T.DIV,a,b)) =
```

Try to identify as many tiles as possible. If fed up resort to general cases, i.e., small tiles.

Putting it all together

I will provide (in C/Java/ML) a function

```
type allocation = Frame.register Temp.Table.table
alloc : Assem.inst list * Frame.frame -> Assem.instr list * allocation
```

which maps all the temporaries to physical registers or "spills" them into memory in a semantics-preserving way.

We'll see in the next lectures how that can be done.

For each fragment (procedure definition) perform translation, canonicalisation, code generation, adding prologue/epilogue, allocation, printing.

The main block is also a function labelled `main`.

Variables in `main` should be accessible from anywhere. For this you need to store the `frame` allocated to `main` somewhere. This is possible since `main` is called only once.

Practical

- Implement instruction selection
- implement frame module
- Put things together to a working Tiger'99 compiler.
- Test your compiler with simple test programs and with `stars.tig`, `queens99.tig`.
- Submit documented source code by Friday 15th December 2000, 5pm.
- Little or no tutor support will be available in week 10 → start early!
- Working submissions will enter a contest with winner determined by shortest code & best runtime.

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 "pseudo assembly language" program 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

- To perform analyses on a program it is often useful to make a *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.

$$
\begin{array}{rl}
 & a \leftarrow 0 \\
L_1: & b \leftarrow a+1 \\
 & c \leftarrow c+b \\
 & a \leftarrow b*2 \\
 & \text{if } a < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
\qquad
\begin{array}{rl}
 & a \leftarrow 0 \\
L_1: & a \leftarrow a+1 \\
 & c \leftarrow c+a \\
 & a \leftarrow a*2 \\
 & \text{if } a < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
\qquad
\begin{array}{rl}
 & b \leftarrow 0 \\
L_1: & b \leftarrow b+1 \\
 & c \leftarrow c+b \\
 & b \leftarrow b*2 \\
 & \text{if } b < N \text{ goto } L_1 \\
 & \text{return } c
\end{array}
$$

Flow graph terminology

- A flow graph has *out-edges* which lead to *successor* nodes and *in-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.

Equations for in and out

$$in[n] = use[n] \cup (out[n] \setminus def[n])$$
$$out[n] = \bigcup_{s \in Suc[n]} in[n]$$

Calculation of liveness by iteration

$/*$ *initialise the edge sets* $*/$
for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$;

repeat
  for each n

    $/*$ *record the old values* $*/$
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$;

    $/*$ *compute the new values* $*/$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$;
    $out[n] \leftarrow \{\}$;
    for each s **in** $succ[n]$
      $out[n] \leftarrow out[n] \cup in[s]$;

until $in'[n] = in[n]$ **and** $out'[n] = out[n]$ **for all** n

Register allocation

- **Input:** Program containing machine registers and temporaries.
- **Output:** Equivalent program mentioning only the machine registers.
- **Method:** Assign temporaries to machine registers where possible (No two interfering temporaries can be mapped to the same machine register). Store temporaries in the frame otherwise.

Interference graph

- **Nodes:** Temporaries and actual machine registers.
- **Edges:** denote interference. Two adjacent nodes must not be assigned the same machine register.
- **How to determine edges:**
  - t_1, t_2 adjacent if live at the same time, i.e., $\exists n. t_1, t_2 \in \mathit{Out}[n] \vee t_1, t_2 \in \mathit{In}[n]$.
  - t_i is live at a def of t_{3-i}. (Better!)
  - No edge if t_1, t_2 simultaneously live *only* because of a —tt MOVE-instruction.
  - t_1 is a machine register and t_2 must not be assigned t_1.

Register allocation as graph colouring

- **Input:** Interference graph. Machine registers and temporaries as nodes. Edges denote interference.
- **Output:**
  - *Either* Assignment of temporaries to machine registers ("'colours"') such that no two connected temporaries are assigned the same machine register.
  - *Or "Can't do it".*

NP-Completeness of graph colouring

- P: Problems solvable in time $O(n), O(n^2), O(n^3), \ldots$ where n is the size of the input. E.g. Maximum flow, matching, liveness, interference graph, ...
- NP: Problems such that verification of an alleged solution is in P.
- NP-complete: NP-problem such that any other NP problem can be reduced to it. E.g. truth of propositional formulas, travelling salesman, graph colouring.
- Million-Dollar-Question: Is P=NP?

Spilling

- To *spill* a node (temporary) t means to replace the code by equivalent code in which t is held in memory (the frame) extra `MOVE` instructions are generated to ensure equivalence.
- Spilling different nodes may result in different runtime penalties.
- *Spill cost* approximates the runtime penalty of spilling a temporary.
- E.g. Spill cost$(t) = \Sigma_{\text{use or def } x \text{ of } t}\, 10^{\text{loop nesting depth of } x}$

  Or simply assume constant spill cost.
- The option of spilling means that we don't necessarily need optimal colouring.

Simple heuristics for graph colouring

K = # registers (colours). Degree of node = # adjacent nodes.

1. Successively remove nodes of degree $< K$ (means: push them on a stack). Rationale: if remaining graph can be coloured so can be the original one.
2. If all nodes have degree $\geq K$ remove (push on stack) arbitrary node in the hope that it can be coloured.
3. Repeat stage 1.
4. Try to colour the graph by successively popping nodes from the stack and selecting colours for them.
5. If this is impossible mark the node as "'spill candidate"' but continue popping nodes to find other spill candidates.
6. Rewrite program by spilling the spill candidates and start over. Usually terminates after 2-3 iterations.

Coalescing

We can unify ("coalesce") two nodes t_1, t_2 if

- code contains move instruction $t_2 \leftarrow t_1$,
- either the t_i do not interfere
- or no def of t_i occurs after the move

Coalescing is performed by

- Coalescing nodes in the interference graph
- eliminating move instruction from code
- updating any subsequent instructions accordingly, e.g. replace $u \leftarrow a \oplus t_2$ with $u \leftarrow a \oplus t_1$.

However, coalescing may make colouring more difficult: See [Appel] for heuristics when coalescing is safe in that respect.

Colouring heuristics with coalescing

- Build interference graph with move-related nodes marked.
- Simplify and coalesce; but don't simplify move-related nodes (so as not to destroy opportunity for coalescing)
- freeze a move related node and resume simplification
- Remove degree $\geq K$ node and resume simplification
- Identify spill candidates by popping
- Spill the candidates and start over.

Spilling

Use graph colouring again to minimise number of frame locations.

- Build interference graph for spill candidates
- Coalesce all move related spill nodes
- Push nodes on stack in the order of their degree (lowest degree first)
- Pop nodes and colour them (no limit on colours as these correspond to stack frame locations).
- Assign frame locations and alter code.
- Restart register allocation.

Precoloured nodes

- The nodes corresponding to machine registers are "'pre-coloured"'. They interfere with each other and can be neither simplified nor spilled.
- Live ranges of precoloured nodes should be kept short, e.g. by moving callee-save registers to temporaries at entry.

Global variables

Tiger '99 has no nested functions. However, it does have global variables.

How do we tell the callee where the global variables are?

We pass the frame pointer of the main frame as an extra argument. "static link light".

So, the base address of global variables can always be found in MEM(TEMP F.FP+wordsize), i.e., the position of the first argument.

The main function must place its frame pointer there initially; whenever you call a function you pass the contents of this location as the first argument.

The actual arguments are then of course to be found at TEMP.FP + 2*wordsize, etc.

Frame module

Variables can be in registers, in the current frame or global:

```
 datatype access = InFrame of int | InReg of Temp.temp | Global of int
```

A frame can either be the main frame or an ordinary frame

```
datatype frame = FRAME of {name: Temp.label,ismain: bool,
                         params: access list, locals: int ref}
    val link : frame -> Tree.exp (* base address of global variables:
             TEMP F.FP in the main frame; MEM(FP + wordsize) otherwise *)
    val exp : access -> Tree.exp
    val newFrame : {name: Temp.label, formals: int} -> frame
    val main : frame
    val formals: frame -> access list
    val name: frame -> Temp.label
    val allocLocal : frame -> bool -> access
           (* allocLocal f true = InFrame(...), allocLocal f false =
               InReg, however if f.ismain=true then
               allocLocal f b = Global(...) *)
    val procEntryExit1 : frame * Tree.stm -> Tree.stm
    val procEntryExit2 : frame * Assem.instr list -> Assem.instr list
```



```
    val procEntryExit3 : frame * Assem.instr list ->
                              {prolog: string, body: Assem.instr list,epilog: string}
   datatype frag = PROC of  {body: Tree.stm, frame: frame}
```

Translation again

Parametrise `transExp, transDec` by the frame:

```
transExp : Frame.frame * Env.enventry S.table * Tr.breakScope option
-> Absyn.exp -> Tr.exp
transDec : Frame.frame * Env.enventry S.table * Tr.breakScope option
-> Absyn.dec -> Env.enventry S.table * Tr.exp list
```

Use `allocLocal, access, exp, ...` from the frame module directly.

Putting it all together

Implement a Main module that generates ix86 assembler code!

The Monday lecture will be replaced by extra help session in Machine Hall.

Merry Christmas and Happy New Year!

Index

INDEX