Contents

1 Introduction (1)

2 Lexical Analysis (2)

2.1 General

- The analysis of a program is usually broken into
  - **Lexical analysis:** breaking the input into individual words or "tokens"
  - **Syntax analysis:** parsing the phrase structure of the program
  - **Semantic analysis:** calculating the programs meaning
- The lexical analyzer takes a stream of characters and produces a stream of names, keywords and punctuation marks
  - It discards white space and comments between tokens

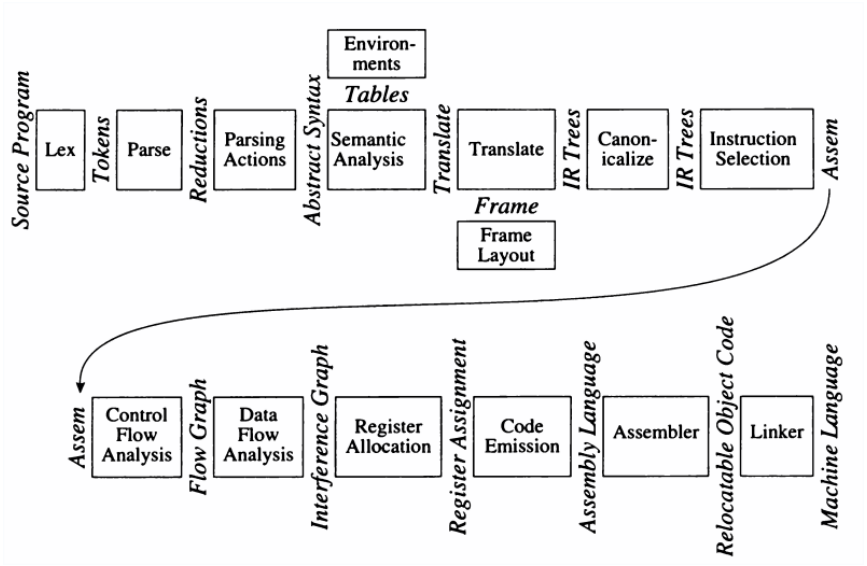

Figure 1: Phases of a compiler and interfaces between them

2.2 Lexical Tokens

- A **lexical token** is a sequence of character that can be treated as a unit in the grammar of a programming language.
  - A programming language classifies lexical tokens into a finite set of token types

- Punctuation tokens such as `IF`, `VOID`, `RETURN` constructed from alphabetic characters are called reserved words and can in most languages not be used as IDs

2.3 Regular Expressions

- A lexer is defined in the book using regular expressions and a FA.
- There are two important disambiguation rules used by Lex, ML-Lex and other similar lexical analyzer generators
  - **Longest match:** The longest initial substring of the input that can match any regular expression is takes as the next token

Type	Examples
ID	foo n14 last
NUM	73 0 00 515 082
REAL	66.1 .5 10. 1e67 5.5e-10
IF	if
COMMA	,
NOTEQ	!=
LPAREN	(
RPAREN	)

Figure 2: Examples of token types

comment	/* try again */
preprocessor directive	#include<stdio.h>
preprocessor directive	#define NUMS 5 , 6
macro	NUMS
blanks, tabs, and newlines	

Figure 3: Examples of nontokens

  - **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

2.4 FA

- To recognize the longest match just means remembering the last time the automaton was in a final state with two variables, `Last-Final` (the state number of the most recent final state encountered) and `Input-Position-at-Last-Final`
  - Every time it enters a final state it updates the variables
  - When a *dead* state (a nonfinal state with no output transitions) is reached the variables tell that the token was matched and where it ended

2.5 ML-Lex: A Lexical Analyser Generator

- ML-Lex is a lexical analyser generator that produces a ML program from a lexical specification
- For each token type in the programming language to be lexical analysed the specification contains a regular expression and an action
  - The action communicates the token type to the next phase of the compiler
  - The output of ML-Lex is a program in ML that interprets a DFA
- The first part of the specification contains functions and types written in ML
  - These must include the type `lexresult` which is the result type of each call to the lexing function
  - It must also include the function `eof`, which the lexing engine will call at end of file
- The second part of the specification contains regular-expression abbreviations and state declarations

```
(* ML Declarations: *)
    type lexresult = Tokens.token
    fun eof() = Tokens.EOF(0,0)
    %%
(* Lex Definitions: *)
    digits=[0-9]+
    %%
(* Regular Expressions and Actions: *)
    if                  => (Tokens.IF(yypos,yypos+2));
    [a-z][a-z0-9]*      => (Tokens.ID(yytext,yypos,yypos+size yytext));
    {digits}            => (Tokens.NUM(Int.fromString yytext,
                                yypos,yypos+size yytext));
    ({digits}"."[0-9]*)|([0-9]*"."{digits})
                        => (Tokens.REAL(Real.fromString yytext,
                                        yypos, yypos+size yytext));
    ("--"[a-z]*"\n")|(" "|"\n"|"\t")+
                        =>    (continue());
                        =>    (ErrorMsg.error yypos "illegal character";
                               continue());
```

Figure 4: An example of an ML-Lex specification

  - e.g. `digits=[0-9]+ allows the name ~{digits}` to stand for a nonempty sequence of digits within regular expressions allows

- The third part of the specification contains regular-expression abbreviations and state declarations
  - The actions are fragments of ordinary ML code
  - Each action return a value of type `lexresult`
  - In the action fragments, several special variables are available
    - The string matched by the regular expression is `yytext`
    - The file position of the beginning of the matched string is `yypos`
    - The function `continue()` calls the lexical analyser recursively

- It is possible to declare a set of *start states*
  - Each regular expression can be prefixed by the set of start states in which it is valid
  - The action statements can explicitly change the start states

```
      the usual preamble ...
%%
%s COMMENT
%%
<INITIAL>if        => (Tokens.IF(yypos,yypos+2));
<INITIAL>[a-z]+    => (Tokens.ID(yytext,yypos,
                                 yypos+size(yytext)));
<INITIAL>"(*"      => (YYBEGIN COMMENT; continue());
<COMMENT>"*)"      => (YYBEGIN INITIAL; continue());
<COMMENT>.         => (continue();)
```

Figure 5: An example of explicity declaring startes

3 Parsing (3)

3.1 Context-Free Grammars

1 $S \rightarrow S$; S	4 $E \rightarrow$ id	8 $L \rightarrow E$
2 $S \rightarrow$ id := E	5 $E \rightarrow$ num	9 $L \rightarrow L$, E
3 $S \rightarrow$ print (L)	6 $E \rightarrow E + E$	
	7 $E \rightarrow (S , E)$	

GRAMMAR 3.1. A syntax for straight-line programs.

- Parsers must read not only terminal symbols such as `+`, `-`, `num` and so on, but also the end-of-file marker
  - `$` is used to represent the end of file

3.2 Predictive Parsing

3.2.1 General

- Some grammars are easy to parse using an algorithm known as *recursive descent*
  - Each grammar production turns into one clause of a recursive function
  - Works only on grammars where the first terminal symbol of each sub-expression provides enough information to choose which production to use
  - The advantage of this is that it can be constructed by hand without the need for automatic tools

3.2.2 First And Follow Sets

- Given a string γ of terminal and nonterminal symbols, FIRST(γ) is the set of all terminal symbols that can begin any string derived from γ
  - If two different productions $X \to \gamma_1$ and $X \to \gamma_2$ has overlapping FIRST sets the grammar cannot be parsed using predictive parsing
- With respect to a particular grammar, given a string γ of terminals and nonterminals
  - 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
- The following is true for the FIRST relation to strings of symbols
  - FIRST($X\gamma$) = FIRST[$X\gamma$] if not nullable[X]
  - FIRST($X\gamma$) = FIRST[X] $\cup$ FIRST(γ) if not nullable[X]

3.2.3 Constructing a predictive parser

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

FIGURE 3.14. Predictive parsing table for Grammar 3.12.

- A predictive parsing table is a table that is indexed by nonterminals X and terminals T

  - Constructed by entering a production $X \to \gamma$ in row X, column T of the table for each $T \in \text{FIRST}(\gamma)$
  - If γ is nullable enter the production in row X, column R for each $T \in \text{FOLLOW}[X]$
  - If one of the entries contain more than one production predictive parsing will not work on the grammar
  - Grammars whose predictive parsing table tables contain no duplicate entries are called LL(1)

- $LL(k)$ parsing tables is a table whose rows and columns are every sequence of k terminals
  - Rarely used in done, because the of the large tables

- Grammars parsable with $LL(k)$ parsing tables are called $LL(k)$ grammars
  - Any $LL(k-1)$ grammar is also a $LL(k)$ grammar

3.2.4 Error recovery

- When during error recovery one must:
  - Print out a meaning full message
  - Delete the thing causing the error to avoid running forever

3.3 LR Parsing

3.3.1 General

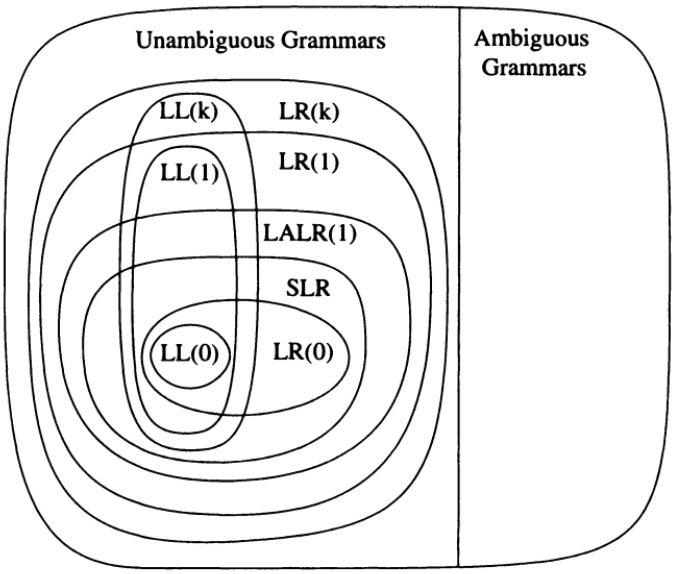

FIGURE 3.29. A hierarchy of grammar classes.

- A powerfull technique for parsing is called LR(k) parsing
  - It is able to postpone the decision until it has seen input tokens corresponding to the entire right-hand side of the production in question
    - k more input tokens beyond
- The parser has a stack and an input
- The first k tokens of the input are the **lookahead**
- Based on the contents of the stack and the lookahead the parser performs two kinds of actions
  - `Shift`: move the first input token to the top of the stack
  - `Reduce`: Choose a grammar rule $X \rightarrow A\ B\ C$; pop C, B, A
- The action of shifting the end-of-file marker `$` is called **accepting** and cause the parser to stop successfully

3.3.2 LR Parsing Engine

- The LR parser know when to shift and reduce by using a FA
  - It is not applied to the input but to the stack
  - The edges are labeled by symbols that can appear on the stack

	id	num	print	;	,	+	:=	(	)	$	S	E	L
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9	s20	s10						s8				g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

TABLE 3.19. LR parsing table for Grammar 3.1.

- The elements in a transition table are labeled with four kinds of actions:
  - ***s****n* Shift into state n;
  - ***g****n* Goto state n;
  - ***r****k* Reduce by rule k;
  - ***a*** Accept;
  - Error is denote by a blank entry in the table
- To use a transition table in parsing treat the shift and goto actions as edges of the DFA and scan the stack
- Rather than rescan the stack again the parser can remember the state reached for each stack element.

```
Look up top stack state, and input symbol, to get action;
If action is
   Shift(n):    Advance input one token; push n on 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 stack, look up X to get "goto n";
                Push n on top of stack.
   Accept:      Stop parsing, report success.
   Error:       Stop parsing, report failure.
```

Figure 6: The parsing algorithm

3.3.3 $LR(0)$ Parser generation

- An $L(k)$ parser uses the contents of its stack and the next k tokens of the input to decide which action to take
  - In practice $k > 1$ is not used for compilation
  - Grammars which has $k = 0$ is too weak to be very useful

3.4 Using Parser Generators

3.4.1 ML-Yacc General

- ML-Yacc is a parser generator
- An ML-Yacc specification is divided into three sections separated by %% marks

```
user declarations
%%
parser declarations
%%
grammar rules
```

- The **user declarations** are ordinary ML declarations usable from the semantic actions in later sections
- The **parser declarations** include a list of the terminal symbols non-terminals and so on

```
%%
%term ID | WHILE | BEGIN | END | DO | IF | THEN | ELSE | SEMI | ASSIGN | EOF

%nonterm  prog | stm | stmlist

%pos int
%verbose
%start prog
%eop EOF %noshift EOF

%%

prog: stmlist                    ()

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

stmlist : stm                    ()
        | stmlist SEMI stm       ()
```

Figure 7: Example of ML-Yacc without Semantic Actions

- The **grammars rules** are productions of the form

```
exp: exp plus exp (semantic action)
```

- Where `exp` is a nonterminal producing a right hand side of `exp+exp` and `PLUS` is a terminal symbol
  - The semantic action is written in ordinary ML and will be executed whenever the parser reduces using this rule

3.4.2 Conflicts

- ML-Yacc reports shift-reduce and reduce-reduce conflicts
  - A **shift-reduce conflict** is a choice between shifting and reducing
  - A **reduce-reduce conflict** is a choice between reducing and reducing
  - By default ML-Yacc resolves shift-reduce conflicts by shifting and reduce-reduce conflicts by using the rule that appears earlier in the grammar

- Most shift-reduce conflicts and reduce-reduce conflicts are serious problems and should be eliminated by rewriting the grammar

3.4.3 Precedence Directives

- ML-Yacc has precedence directives to indicate the resolution of the class of **shift-reduce conflicts** that are caused by ambiguity in the grammar

```
%nonassoc EQ NEQ
%left PLUS MINUS
%left TIMES DIV
%right EXP
```

Figure 8: Example of precedence directives that are used to indicate that + and - are left-assciative and bind equally tightly and that * and / are left-assciative and bind more tightly than +, that $= \neq$ are nonassociative and binds more weekly than + and that ^ is rightassociative and bind most tightly.

3.4.4 Syntax Versus Semantics

- When given an identifier which can have multiple types e.g. numbers and booleans, one must change the grammar to make the two identifiers equal and let the semantic part of the compiler handle it

3.5 Error Recovery

3.5.1 Recovery Using The Error Symbol

- Local error recovery mechanisms work by adjusting the parse stack and the inputs *where the error was detected* in a way that will allow parsing to resume
  - Many versions of the Yacc parser generator uses a special *error* symbol to control the recovery process
  - Can be done by adding error-recovery productions
- When the LR parser reaches an error state it does not following actions

1. Pop the stack (if necessary) until a state is reached in which the action for the *error* token is *shift*
2. Shift the *error* token
3. Discard input symbols (if necessary) until a state is reached that has a non-error action on the current lookahead token
4. Resume normal parsing

3.5.2 Global Error Repair

- **Global error repair** finds the smallest set of insertions and deletions that would turn the source string into a syntactically correct string
  - Even if the insertions and deletions are not at a point where an LL or LR parser would first report an error
- **Burke-Fisher error repair:** Tries every possible single-token insertion, deletion or replacement at every point that occurs no earlier that K tokens before the point where the parser reported the error
  - The correction that allows the parser to parse furthest past the original reported error is taken as the best error repair
  - Generally if a repair carries the parser $R = 4$ tokens beyond where it originally got stuck it is "good enough"
  - The advantage of this technique is that the grammar is not modified at all, nor are the parsing tables modified, only the parsing engine
  - The parsing engine must be able to back up K tokens and reparse
    - Needs to remember what the parse stack looked like K tokens ago.
    - The algorithm maintains two parse stack the *current* stack and the *old* stack
    - Queue of K tokens is kept, as a new token is shifted it is pushed on the current stack and put onto the tail of the queue and the head is pooped
- **Semantic actions:** Shift and reduce actions are tried repeatedly and discarded during the search for the best error repair
  - The Burke-Fisher parser does not execute any of the semantic actions as the reductions are performed on the current stack

    - Waits until the same reductions are performed on the *old* stack

- **Semantic value for insertions**: In repairing an error by insertion the parser needs to provide a semantic value for each token it inserts
  - Done in ML-Yacc by using the `%value` directive

```
%value ID  ("bogus")
%value INT (1)
%value STRING  ("")
```

Figure 9: Value directive example

- **Programmer-specified substitutions:** Some common kinds of errors cannot be repaired by the insertion or delection of a single token
  - Should be tried first
  - In the ML-Yacc grammar specification the probrammer can use the `%change` directive to suggest error corrections to be tried first, forfore the default "delete or insert each possible token" repairs

```
%change     EQ -> ASSIGN  |  ASSIGN -> EQ
       |   SEMICOLON ELSE -> ELSE    |    -> IN INT END
```

Figure 10: Change directive example

4 Abstract Syntax (4)

4.1 Semantic Actions

- For a rule $A \rightarrow B\ C\ D$, the semantic action must return a value whose type is the one associated the nonterminal A
  - It can build this value from the values associated with the matched terminals and nonterminal B, C, D

- In a recursive-descent parser, the semantic actions are the values returned by the parsing functions, or the side effects of those functions, or both

```
%%
%term INT of int | PLUS | MINUS | TIMES | UMINUS | EOF
%nonterm  exp of int
%start exp
%eop EOF

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

exp :  INT              (INT)
    |  exp PLUS exp     (exp1 + exp2)
    |  exp MINUS exp    (exp1 - exp2)
    |  exp TIMES exp    (exp1 * exp2)
    |  MINUS exp    %prec UMINUS  (~exp)
```

Figure 11: ML-Yacc grammar with semantic actions

- A parser specification for ML-Yacc consists of a set of grammar rules each annotated with a semantic action that is an ML expression
  - When ever the generated parser reduces by a rule it will execute the corresponding semantic action fragment
  - The semantic rules can refer to the semantic value of right-hand-side symbols
  - The value produced by every semantic expression must match the nonterminal
  - Implement semantic values by keeping a stack of them parallel to the state stack
    - Where each symbol would be on a simple parsing stack the now is a semantic value

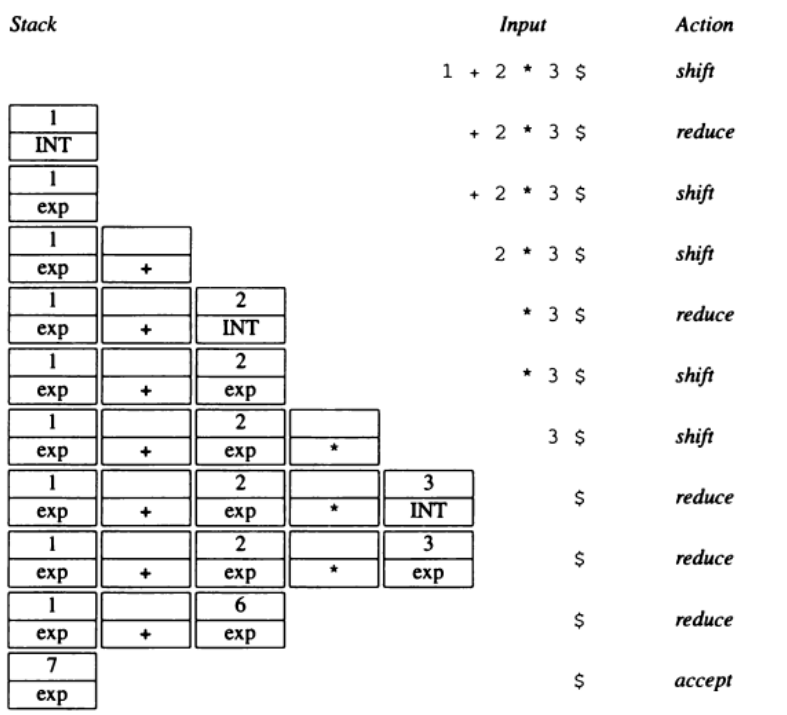

FIGURE 4.3. Parsing with a semantic stack.

4.2 Abstract Parse Trees

```
structure Absyn = struct
   datatype binop = ···  (see Program 1.5)
   datatype stm =  ···
        and exp =  ···
end
%%
%term INT of int | ID of string | PLUS | MINUS | TIMES | DIV
     | ASSIGN | PRINT | LPAREN | RPAREN | COMMA | SEMICOLON | EOF
%nonterm exp of exp | stm of stm | exps of exp list | prog of stm
%right SEMICOLON
%left PLUS MINUS
%left TIMES DIV
%start prog
%eop EOF
%pos int
%%
prog: stm                        (stm)

stm : stm SEMICOLON stm          (Absyn.CompoundStm(stm1,stm2))
stm : ID ASSIGN exp              (Absyn.AssignStm(ID,exp))
stm : PRINT LPAREN exps RPAREN   (Absyn.PrintStm(exps))

exps: exp                        ( exp :: nil )
exps: exp COMMA exps             ( exp :: exps )

exp : INT                        (Absyn.NumExp(INT))
exp : ID                         (Absyn.IdExp(ID))
exp : exp PLUS exp               (Absyn.OpExp(exp1,Absyn.Plus,exp2))
exp : exp MINUS exp              (Absyn.OpExp(exp1,Absyn.Minus,exp2))
exp : exp TIMES exp              (Absyn.OpExp(exp1,Absyn.Times,exp2))
exp : exp DIV exp                (Absyn.OpExp(exp1,Absyn.Div,exp2))
exp : stm COMMA exp              (Absyn.EseqExp(stm,exp))
exp : LPAREN exp RPAREN          ( exp )
```

PROGRAM 4.7. Abstract-syntax builder for straight-line programs.

- A way to separate the issues of parsing from the issues of semantics is to produce a **parse tree**
  - A data structure that later phases of the compiler can traverse
  - It has exactly one leaf for each token of the input and one internal node for each grammar rule reduced during phase
    - Is called a **concrete parsing tree** for concrete syntax
    - Is inconvenient to use directly
    - Depends too much on the grammar
- An **abstract syntax** makes a clean interface between the parser and the later phases of the compiler

  - The compiler will need to represent and manipulate abstract syntax trees as data structures (in ML using `datatype`)

- Since a grammar using abstract syntax tree often separates the different compiler operations one need to remember the position for reporting failures
  - To remember positions accurately the abstract-syntax data structures must be sprinkled with `pos` fields
  - The ML-Yacc parser makes the beginning and end positions of each token available to the parser

5 Semantic Analysis (5)

5.1 General

- The **semantic analysis** phase of a compiler
  - connects variable definitions to their uses
  - checks that each expression has a correct type
  - translates the abstract syntax into a simpler representation suitable for generating machine code

5.2 Symbol Tables

5.2.1 General

- This phase is characterized by the maintenance of **symbol tables** mapping identifiers to their types and locations
  - Also called **environments**
  - As the declarations of types, variables and functions are processed, these identifiers are bound to "meanings" in the symbol tables
  - When **uses** (non-defining occurrences) of identifiers are found, they are looked up in the symbol tables
  - 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 identifier bindings local to that scope are discarded

- An **environment** is a set of bindings

  - Denoted by the $\mapsto$ arrow
  - e.g. an environment σ_0 which contains the bindings $\{g \mapsto \text{string}, a \mapsto \text{int}\}$ meaning that a is an integer variable and g is a string variable
  - When two environments are added the new variables, i.e. the left hand side, has precedence over the existing types
  - Can be implemented in to ways
    * **Functional style:** where the original environment are kept in pristine condition
    * **Imperative style:** where the environment is modified to become a new environment and a undo stack is kept

- In some languages there can be several active environments at once
  - Each module or class or record in the program has a symbol table σ of its own

5.2.2 Efficient Imperative Symbol Tables

```
val SIZE = 109   should be prime
type binding = ···
type bucket = (string * binding) list
type table = bucket Array.array
val t : table = Array.array(SIZE, nil)

fun hash(s: string) : int =
    CharVector.foldl (fn (c,n)=> (n*256+ord(c)) mod SIZE) 0 s

fun insert(s: string, b: binding) =
   let val i = hash(s) mod SIZE
    in Array.update(t,i,(s,b)::Array.sub(t,i))
   end

exception NotFound

fun lookup(s: string) =
   let val i = hash(s) mod SIZE
       fun search((s',b)::rest) = if s=s' then b
                                          else search rest
         | search nil = raise NotFound
    in search(Array.sub(t,i))
   end

fun pop(s: string) =
   let val i = hash(s) mod SIZE
       val (s',b)::rest = Array.sub(t,i)
    in assert(s=s');
       Array.update(t,i,rest)
   end
```

PROGRAM 5.2. Hash table with external chaining.

- Because a large program may contain thousands of distinct identifiers symbol tables must permit efficient lookup
  - Imperative-style environments are usually implemented using hash table, which are very efficient
  - The operation $\sigma' = \sigma + \{a \mapsto \tau\}$ is implemented by inserting τ in the hash table with key a
  - A simple **hash table with external chaining** works well and supports deletion easily
    - When we will need to delete $\{a \mapsto \tau\}$ to recover σ at the end of the scope of a

5.2.3 Efficient Functional Symbol Tables

- In the functional style, we wish to compute $\sigma' = \sigma + \{a \mapsto \tau\}$ in such a way that we still have σ available to look up identifiers
  - Instead of altering a table we create a new table by computing the "sum" of an existing table and a new binding
  - By using binary search tree we can perform functional additions to search trees efficiently

6 Activation Records (6)

6.1 Higher order functions

- In some languages such as C the local variables are destroyed when a function returns
- In languages supporting both nested functions and function valued variables, it may be necessary to keep local variables after a function has returned
  - It is the combination of nested functions and functions returned as results that cause the local variables to have longer lifetimes than their enclosing function invocations
  - Pascal (and Tiger) has nested functions but do not have functions as return variables, C has functions as returnable variables but not nested functions
    - These languages can use stacks to hold local variables
  - ML, Scheme and several other languages have both nested functions and functions as returnable values
    - This combination is called higher-order functions
    - They cannot use a stack to hold all local variables

6.2 Stack Frames

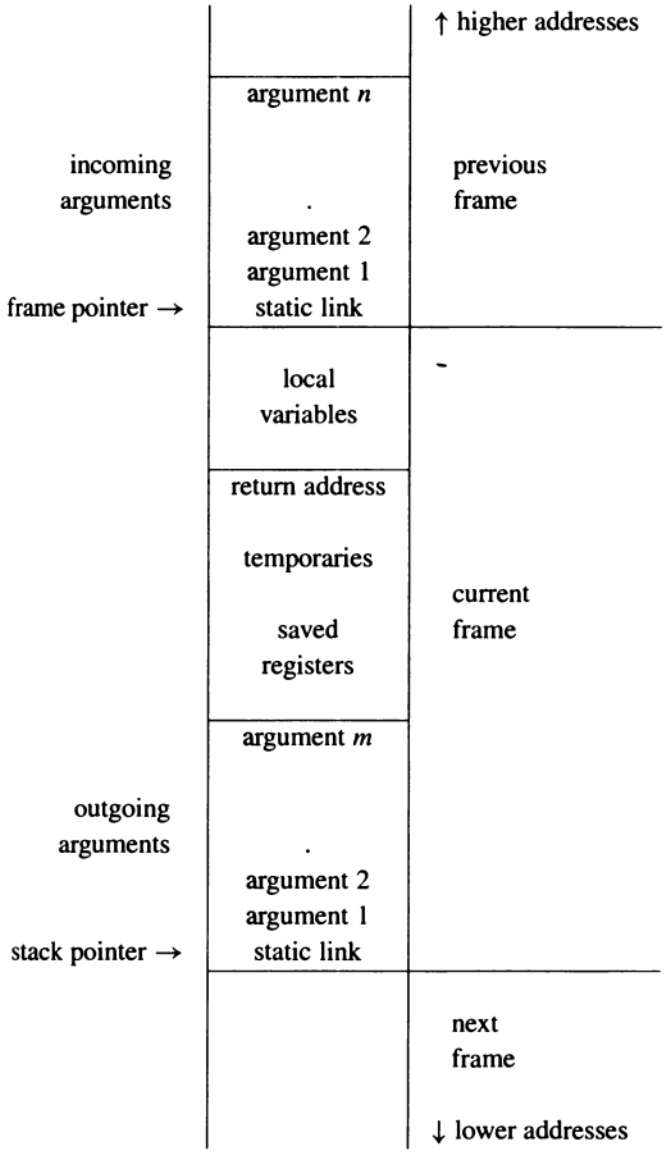

FIGURE 6.2. A stack frame.

6.2.1 General

- Since we need to push and pop in large batches and access variables deep within the stack the standard stack is not suitable for storing local variables
- The stack is treated as a big array with a special register, the *stack pointer* that points to some location
  - All locations beyond the stack pointer are considered **garbage**
  - All locations before the stack pointer are considered **allocated**
  - It usually grows only at the entry to a function, by an increment large enough to hold all the variables for that function
    - It shrinks at by the same amount when exiting the function
  - The area on the stack devoted to the local variables, parameters, return address and other temporaries for a function is called the functions **activation record** or **stack frame**
  - Run-time stacks usually start at a high memory address and grow toward smaller addresses
  - The design of the frame layout takes into account the particular features of an instruction set architecture and the programming language being compiled
    - A manufacture of a computer often prescribes a "standard" frame layout to be used on that architecture where possible by all compilers for all programming languages
    - By using the standard layout we gain the considerable benefit that functions written in one language can call functions written in another language
  - The *return address* is created by the `CALL` instruction and tells where control should return upon completion of the current function
  - Some local variables are kept in the frame others are kept in machine registers
    - Sometimes a local variable kept in the registers needs to be saved into the frame to make room for other uses of the register
    - There is an area in the frame for this purpose
  - When the current function calls other functions it can use the *outgoing argument* space to parse parameters

6.2.2 The frame pointer

- If a function $g(\dots)$ calls the function $f(a_1, \dots, a_n)$ we say g is the caller and f is the callee
  - On entry to f the stack pointer points to the first argument that g passes to f
  - On entry, f allocates a frame by simply subtracting the frame size from the stack pointer
    - The old `SP` becomes the current frame pointer `FB`
  - In some frame layouts `FP` is a separate register
    - The old value of `FP` is saved in memory
    - The new `FP` becomes the old `SP`
    - When f exits it just copies `FP` back to `SP` and fetches back the save `FP`
    - This is useful if the frame size can vary or if frames are not always continuous on the stack
  - If the frame size is fixed it is not necessary to use a register for `FP` at all
    - For each function f the `FP` will always differ from `SP` by some fixed amount
    - `FP` is a "fictional" register whose value is always `SP` + *frame-size*
  - Since the frame size is not know until quite late in the compilation process
    - Therefore it is convenient to talk about a frame pointer
    - Also since we put the formals and locals right near the frame pointer at offsets that are known early
    - Temporaries and saved registers go farther away at offsets that are known later

6.2.3 Registers

- A modern machine has a large set of registers
  - To make compile programs run fast, it's useful to keep local variables, intermediate results of expressions and other values in registers instead of in the stack frame

  - Registers can be directly accessed by arithmetic instruction
    * On most machines it requires separate *load* and *store* instructions
    * Even on machines whose arithmetic instructions can access memory it is faster to access registers

- A machine usually has only one set of registers but many different procedures and functions need to use registers
  - If a function f is using register r to hold a local variable and calls procedure g which also uses r for its own calculations
    * Then one must save r into the stack frame before g uses it
    * Both f and g could have that responsibility
  - r is a **caller-save** register if the caller must save and restore the register
  - r is a **callee-save** register if the callee must save and restore the register
  - On most machine architecture the notion of caller-save and callee-save registers is not something built into the hardware but it is a convention described in the machine's reference manual
  - Some times saves and restores are unnecessary
    * e.g. if the caller knows a variable will no be needed by the callee
  - One will rely on our register allocator to choose the appropriate kind of register for each variable and temporary value

6.2.4 Parameter Passing

- In the calling conversions for machines that where designed in the 1970s the arguments where passed on the stack
- Since most functions use less than four arguments some of the arguments are typically passed through the registers
  - It specifies that the first k arguments (typically 4 or 6) of a function are passed in register $r_p, \ldots, r_{p+k-1}$ and the rest are passed in memory
  - If $f(a_1, \ldots, a_n)$ calls $h(z)$ it must pass the argument z in r_1 so f saves the old contents of r_1 (the value a_1) become calling h

- The reasons passing argument in registers saves time is
  1. Some procedure don't call other procedures
     - They are called **leaf** procedures
     - Leaf procedures needs not write their incoming arguments to memory
     - They do not need to allocate a stack frame at all
  2. Some optimizing compilers use *interprocedural register allocation*
     - It analyses all functions in an entire program at once
     - They assign difference procedures different register in which to receive parameters and hold local variables
  3. Even if f is not a leaf procedure it might be finished with all its use of argument x by the time it calls h
     - f can overwrite r_1 without saving it
  4. Some architecture have *register windows*
     - Each function invocation can allocate a fresh set of registers without memory traffic
- f needs to write an incoming parameter into the frame
  - Ideally its frame layout should matter only in the implementation of f
  - A straightforward approach would be for the caller to pass argument $a_1, \ldots, a_k$ in registers and $a_{k+1}, \ldots, a_n$ at the end of its own frame
  - In the standard calling convention of many modern machines the calling function reserves space for the register arguments in its own frame next to the place where it writes argument $k+1$
    * The caller does no write anything there
    * The space is written into by the called function
  - Another way is to take the address of a local variable and use *call-by-reference*
    * The programmer does not explicitly manipulate the address of a variable x
    * If x is passed as the argument to $f(y)$ where y is a "by reference" parameter the compiler generates code to pass the address of x instead of the contents of x

6.2.5 Return Addresses

- When a function g calls 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 to return to is $a + 1$
    - This is called the return address
  - On some old machines the return address was pushed on the stack by the call instruction
  - Modern science has shown that it is faster and more flexible to pass the return address in a register
  - On modern machine the *call* instruction merely puts the return address in a designated register
    - On non leaf procedure would have to write it to the stack

6.2.6 Frame Resident Variables

- Values are only written to memory for one of these reasons
  - The variable will be passed by reference, so it must have a memory address
  - The variable is accessed by a procedure nested inside the current one
  - The value is too big to fit into a single register
  - The variable is an array, for which address arithmetic is necessary to extract components
  - The register holding the variable is needed for a specific purpose
    - e.g parameter passing
    - may be moved by the compiler to other registers instead of storing them in memory
  - There are so many local variables and temporary values that they won't all fit in registers
    - Some of them are "spilled into the frame"
- A variable **escapes** if it is passed by reference, its address taken or it is accessed from a nested function

- When a formal parameter or local variable is declared it's convenient to assign it a location either in registers or in the stack frame, right at that point in processing the program
  - The occurrences of that variable are translated into machine code that refers to the right locations
  - A good compiler must assign provisional location to all formals and locals and decide later which of them should really go in registers

6.2.7 Static Links

- In languages that allow nested function declarations (e.g. ML and Tiger) the inner functions may use variable declared in outer functions which is called a **block structure**
- To accomplish at block structure there are several methods
  - Whenever a function f is called it can be passed a pointer to the frame of the function statically enclosing f
    - Called a static link
    - In each procedure call or variable access, a chain of zero or more fetches is required
  - 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
    - Called a **display**
  - When g calls f, each variable of g that is actually accessed by f is passed to f as an extra argument
    - Called lambda lifting

7 Liveness Analysis (10)

7.1 General

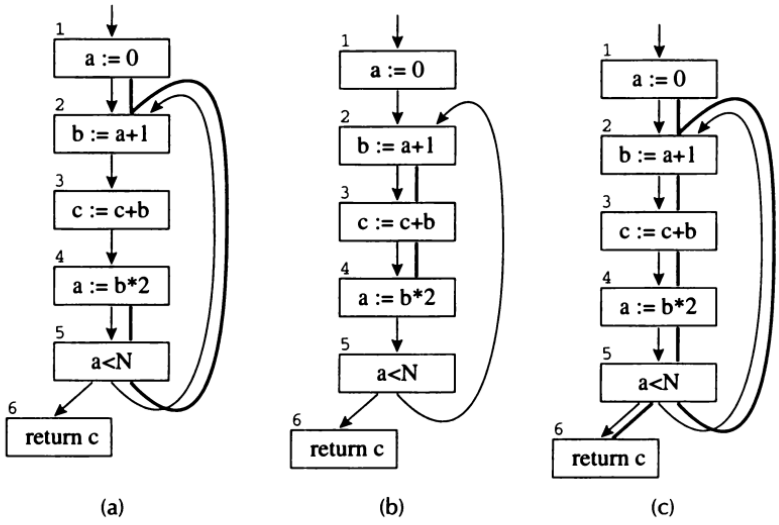

FIGURE 10.2. Liveness of variables *a*, *b*, *c*.

- To decide which registers are safe to use a **liveness analysis** is performed on the IR
  - We say that a variable is **live** is it holds a value that may be needed in the future
  - A control flow graph is often used

7.2 Solution of Dataflow Equations

7.2.1 Terminology

- Determining the live range of each variable is an example of a **dataflow** problem
- A flow-graph node has
  - **out-edges** that lead to **successor** nodes
  - **in-edges** that come from **predecessor** nodes
  - the set $pred[n]$ is all the predecessor of node n
  - the set $succ[n]$ that is all successors of node n

- An assignment to a variable or temporary **defines** that variable
  - An occurrence of a variable on the right-hand side of an assignment or other expressions **uses** that variable
  - The def of a variable is the set of graph nodes that define it
  - The def of the graph node is the set of variables that it defines
  - The use of a variable is the set of graph nodes that uses it
  - The use of the graph node is the set of variables that it uses
- A variable is **live** on an edge if there is a directed path from the edge to a use of that variable that does not flow through any def
  - A variable is **live-in** a node if it is live on any of the in-edges of that node
  - A variable is **live-out** a node if it is live on any of the out-edges of that node

7.2.2 Calculation of Liveness

for each n
  $in[n] \leftarrow \{\}$; $out[n] \leftarrow \{\}$
repeat
  for each n
    $in'[n] \leftarrow in[n]$; $out'[n] \leftarrow out[n]$
    $in[n] \leftarrow use[n] \cup (out[n] - def[n])$
    $out[n] \leftarrow \bigcup_{s \in succ[n]} in[s]$
until $in'[n] = in[n]$ and $out'[n] = out[n]$ for all n

ALGORITHM 10.4. Computation of liveness by iteration.

- Liveness information can be calculated from use and def as follows
  1. If a variable is in $use[n]$ the it is *live-in* at node n
     - If the statements uses a variable the variable is live on entry to that statement
  2. If a variable is *live-in* at node n, then it is *live-out* at all nodes m in $pred[n]$
  3. If a variable is *live-out* in node n, and not in $def[n]$, then the variable is also *live-in* at n

  - If someone needs the value of a at the end of statement n and n does not provide that value, then a's value is needed even on entry to n

- Flow-graph nodes that only have one predecessor and one successor are not very interesting
  - They can be merged with their predecessor and successors
  - This results in a graph with fewer nodes
  - Faster running time
- It can be practical to compute dataflow for one variable at a time as information for that variable is needed
  - This would mean repeating the dataflow traversal once for each temporary
  - Starting from each *use* site of a temporary t and tracing backward using depth-first search
  - The search stops at definition of the temporary
  - Many temporaries have short live range and the searches would terminate quickly

7.2.3 Representation of Sets

- There are two good ways to represent sets for data flow equations
  1. As arrays of bits
     - If there are N variables in the program, the bit-array representation uses N bits for each set
     - Calculating the union of two sets is done by or-ring the corresponding bits at each positionn
     - It takes N/K operations if there is K bits
  2. As sorted list of variables
     - Represented as a linked list of its members, sorted by any totally ordered key e.g. variable name
     - Calculating the union is done by merging the list
     - It takes time proposition to the size of the sets being unioned
- When the sets (fewer than N/K elements the sorted-list representation is asymptotically faster
- When the sets are dense the bit-array representation is better

7.2.4 Time Complexity

- The worst case running time of the algorithm is $O(N^4)$
  - Ordering the nodes using depth-first-search usually bring the number of iteration down to two or three with an algorithm that runs between $O(N)$ and $O(N^2)$ is practice

7.2.5 Least Fixed Points

- Any solution to the dataflow equations is a *conservative approximation*
  - It can be assured that if a is needed at some node n then it can be assured that a is live-out at node n in any solution to the equations
  - We might calculate that d is live-out but it doesn't mean its value is really used
- **Theorem.** Equations 10.3 have more than one solution
- **Theorem.** If $in_X[n]$ and $in_Y[n]$ are the live-in sets for some node n in solution X and Y, then $in_X[n] \subseteq in_Y[n]$
- Algorithm 10.4 always computes the least fixed points

7.2.6 Static vs. Dynamic Liveness

- **Theorem.** There is no program H that takes as input any program P and input X and (without infinite-looping) returns true if $P(X)$ halts and false if $P(X)$ infinite loops
- **Corollary.** No program $H'(X, L)$ can tell, for any program X and label L within X whether the label L is ever reached on an execution of X
- Because of the halting problem there does not exists any general algorithm that can tell if a variable is truly need in a specific place at run time
- **Dynamic liveness** A variable a is dynamically live at node n if some execution of the program goes from n to a use of a without going through any definition of a

- **Static liveness** A variable a is statically live at node n if there is some path of control-flow edges from n to some use of a that does not go through a definition of a

7.2.7 Interference Graphs

1

FIGURE 10.9. Representations of interference.

- Liveness information is used for several kinds of optimization in a compiler
  - For some optimizations we need to know which variables are live at each node in the flow graph
- One of the most important applications of liveness analysis is for register allocation
  - We have a set of temporaries $a, b, c, \ldots$ that must be allocated to registers $r_1, \ldots, r_k$
  - A condition that prevents a and b being allocated to the same register is called an **interference**
  - The most common kind of interference is caused by overlapping live ranges when a and b are both live at the same program point
    - Then they cannot be put in the same register
    - There are other causes e.g. when a must be generated by an instruction that cannot address register r_1 then a and r_1 interfere
- To add interference edges for each new definition considering a move instruction

1. At any nonmove instruction that *defines* a variable a, where the *live-out* variables are $b_1, \ldots, b_j$, add interference edges $(a, b_1), \ldots, (a, b_j)$.
2. At a move instruction $a \leftarrow c$, where variables $b_1, \ldots, b_j$ are *live-out*, add interference edges $(a, b_1), \ldots, (a, b_j)$ for any b_i that is *not* the same as c.

8 Register Allocation (11)

8.1 General

- The job of the register allocator is
  - To assign the many temporaries to a small number of machine registers
  - Where possible to assign the source and destination of a `MOVE` to the same register e.g. deleting the `MOVE`
- From an examination of the control and dataflow graph an **interference graph** can be derived
  - Each node in the interference graph represents a temporary value
  - Each edge (t_1, t_2) indicate a pair of temporaries that cannot be assigned to the same register
- The interference graph is colored
  - As few colors as possible should be used
  - No pair of nodes connected by an edge may be assigned the same color
  - The "colors" correspond to registers
  - If the target machine has K registers we can K color the graph
    - Then coloring is a valid register assignment for the interference graph
  - If there is no K coloring some of variables and temporaries should be kept in memory instead
    - This is called **spilling**

8.2 Coloring by Simplification

- Register allocation is an *NP*-complete problem
- Graph coloring is also *NP*-complete
- The following is a linear-time approximation algorithm with good results with the following phases
  - **Build:** Construct the interference graph
    - Dataflow analysis is used to compute the set of temporaries that are simultaneously live at each program point
    - We add an edge to the graph for each pair of temporaries in the set
    - Repeated for all program points
  - **Simplify:** The graph is colored using a simple heuristic
    - K is the number of registers in the machine
    - Suppose the graph G contains a node m with fewer than K neighbors
    - Let G' be the graph $G - \{m\}$ obtained by removing m
    - If G' can be colored. then so can G
    - This leads naturally to a stack-based/reserves algorithm for coloring
      - We repeatedly remove nodes of degree less than K
      - Each simplification will decrease the degrees of other nodes
  - **Spill:** Suppose at some point during simplification the graph G has nodes only of significat degree, that is nodes of degree $\geq K$
    - The simplify heuristic fail and we mark some node for spilling
    - We choose some node in the graph and decide to represent it in memory during program execution
    - An optimistic approximation to the effect of spilling is that the spilled node does not interfere with any of the other remaining in the graph
    - It can be removed the node and pushed on the stack as the simplify process continuous
  - **Select:** Colors are assigned to the nodes in the graph
    - Starting with the empty graph

        * The original graph is rebuild by repeatedly adding a node from the top of the stack
        * When we add a node to the graph there must be a color for it
        * When potential spill node n that was pushed using the Spill heuristic is popped there is no guarantee that it will be colorable
            · In this case we have an **actual spill**
            · No color is assigned and the Select phase is continued to identify other actual spils
            · If not we can color n which is known as **optimistic coloring**
    - **Start over:** If the **Select** phase is unable to find a color for some node(s)
        * The Program must be rewritten to fetch them from memory just before each use and store them back after each definition
        * A spilled temporary will turn into several new temporaries with tiny live range
        * These will interfere with other temporaries in the graph
        * The algorithm is repeated on this rewritten program
        * This process iterates until simplify succeeds with no spills
            · Typically one or two iteration almost always suffice

8.3 Coalescing

8.3.1 Algorithm

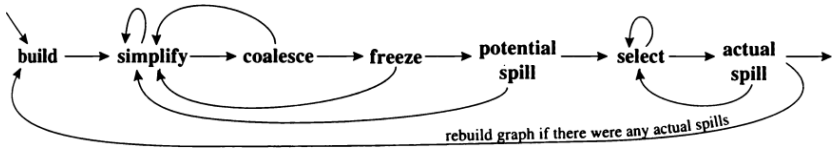

FIGURE 11.4. Graph coloring with coalescing.

- If there is no edge between the source and destination of a move instruction the move can be eliminated
    - The source and destination nodes are **coalesced** into a new node whose edges are the union of those of the nodes being replaced

  - In principle any pair of nodes not connected by an interference edge could be coalesced
  - The node being introduced is more constrained than those being remove
    * Since it contains a union of edges
  - Could make a K colorable graph no longer K colorable

- Strategies to coalesce a graph that is safe which does not render the graph uncolorable
  - **Briggs:** Nodes a and b can be coalesced if the resulting node ab will have fewer than K neighbors of significant degree
  - **George:** Nodes a and b can be coalesced if, for every neighbor t of a either t already interferes with b or t is of insignificant degree

- Phases of register allocator with coalescing

Build: Construct the interference graph, and categorize each node as either *move-related* or *non-move-related*. A move-related node is one that is either the source or destination of a move instruction.

Simplify: One at a time, remove non-move-related nodes of low ($< K$) degree from the graph.

Coalesce: Perform conservative coalescing on the reduced graph obtained in the simplification phase. Since the degrees of many nodes have been reduced by *simplify*, the conservative strategy is likely to find many more moves to coalesce than it would have in the initial interference graph. After two nodes have been coalesced (and the move instruction deleted), if the resulting node is no longer move-related it will be available for the next round of simplification. *Simplify* and *coalesce* are repeated until only significant-degree or move-related nodes remain.

Freeze: If neither *simplify* nor *coalesce* applies, we look for a move-related node of low degree. We *freeze* the moves in which this node is involved: that is, we give up hope of coalescing those moves. This causes the node (and perhaps other nodes related to the frozen moves) to be considered non-move-related, which should enable more simplification. Now, *simplify* and *coalesce* are resumed.

Spill: If there are no low-degree nodes, we select a significant-degree node for potential spilling and push it on the stack.

Select: Pop the entire stack, assigning colors.

Consider Graph 11.1; nodes `b`, `c`, `d`, and `j` are the only move-related nodes. The initial work-list used in the simplify phase must contain only non-move-

8.3.2 Spilling

- If spilling is necessary build and simplify must repeated on the whole program
  - The simplest version of the algorithm discards any coalescence's found if build must be repeated
  - A more efficient algorithm preserves an coalescences done before the first potential spill was discovered and discards the rest
- The algorithm for coalescing of spill is as follows

1. Use liveness information to construct the interference graph for spilled nodes.
2. While there is any pair of non-interfering spilled nodes connected by a move instruction, coalesce them.
3. Use *simplify* and *select* to color the graph. There is no (further) spilling in this coloring; instead, *simplify* just picks the lowest-degree node, and *select* picks the first available color, without any predetermined limit on the number of colors.
4. The colors correspond to activation-record locations for the spilled variables.

- Should done before generating the spill instructions and regenerating the the register-temporary interference graph

8.4 Precolored nodes

8.4.1 General

- Some temporaries are precolored since they represent machine registers
  - e.g. function arguments
  - The *select* and *coalesce* operations can give an ordinary temporary the same color as a precolored as long as they don't interfere
  - For a K register machine, there will be K precolored nodes that all interfere with each other
  - Those of the precolored nodes that are not used explicitly will not interfere with any ordinary nodes
  - A machine register used explicitly will have a live range that interferes with any other variable that might happen to be live at the same time
  - A precolored node cannot be simplified and they should not be spilled

8.4.2 Temporary Copies of Machine Registers

- The coloring algorithm works by calling *simplify*, *coalesce* and *split* until only the precolored node remain
  - Then the *select* phase can start adding the other nodes (and coloring them)
  - Since precolored nodes do not spill, the front end must be careful to keep their live range short
  - It can be done by generating MOVE instruction to move values to and from precolored nodes

8.4.3 Caller-Save and Callee-Save Registers

- A local variable or compiler temporary that is not live across any procedure call should usually be allocated to a caller-save register
  - In this case no saving and restoring of register will be necessary at all
- Any variable that is live across several procedure calls should be kept in a callee-save register
  - Since then only one save/restore will be necessary
- The register allocator should allocate variable to registers using this criterion
  - To do this with a graph-coloring allocator the `CALL` instructions in the `Assem` language have been annotated to interfere with all the caller-save registers

9 Garbage Collection (13)

9.1 General

- Heap-allocated records that are not reachable by any chain of points from program variables are **Garbage**
  - The memory occupied by garbage should be reclaimed for use in allocating new records
  - This process is called **Garbage Collection**
  - It is not performed by the compiler but by the runtime system
  - We will require the compile to guarantee that any live record is **reachable**
  - The number of reachable records that are not live should be minimized

9.2 Mark-And-Sweep Collection

9.2.1 General

```
Mark phase:                          Sweep phase:
  for each root v                      p ← first address in heap
      DFS(v)                           while p < last address in heap
                                           if record p is marked
                                               unmark p
                                           else let f1 be the first field in p
                                               p.f1 ← freelist
                                               freelist ← p
                                           p ← p+(size of record p)
```

ALGORITHM 13.3. Mark-and-sweep garbage collection.

- Program variables and heap-allocated records form a directed graph
  - The variables are roots of this graph
  - A node n is reachable if there is a path of directed edges $r \to \cdots \to n$ starting at some root r
  - A graph-search algorithm such as DFS can mark all reachable nodes
  - Any node not marked must be garbage and should be reclaimed
    - It can be done by a **sweep** of the entire heap from the first address to the last looking at nodes that are not marked
    - These nodes are garbage and can be linked together in a linked list (the freelist)
  - The sweep phase should also unmark all marked nodes in preparation for the next garbage collection
  - After the garbage collection the program resumes execution
  - Whenever the program wants to heap-allocate a new record it gets a record from the freelist
    - When it becomes empty it is a good time to do another garbage collection
  - If there are R words of reachable data in a heap of size J the cost of a garbage collection is $O(R + H)$

9.2.2 Using an explicit stack

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← 1
  stack[t] ← x
  while t > 0
        x ← stack[t];   t ← t − 1
        for each field f_i of record x
            if x.f_i is a pointer and record x.f_i is not marked
              mark x.f_i
              t ← t + 1;   stack[t] ← x.f_i
```

ALGORITHM 13.5. *Depth-first search using an explicit stack.*

- The DFS algorithm is recursive and the maximum depth of its recursion is as long as the longest path in the graph of reachable data
- There could be a path of length H in the worst case
  - Meaning the stack of the activation records would be larger than the entire heap
- To solve this problem an explicit stack is used instead of recursion
  - The stack could still grow to size H
  - This is H words and not H activation records

9.2.3 Pointer reversal

```
function DFS(x)
 if x is a pointer and record x is not marked
  t ← nil
  mark x;  done[x] ← 0
  while true
        i ← done[x]
        if i  <  # of fields in record x
          y ← x.f_i
          if y is a pointer and record y is not marked
            x.f_i ← t;   t ← x;   x ← y
            mark x;  done[x] ← 0
          else
            done[x] ← i + 1
        else
          y ← x;   x ← t
          if x = nil then return
          i ← done[x]
          t ← x.f_i;   x.f_i ← y
          done[x] ← i + 1
```

ALGORITHM 13.6. Depth-first search using pointer reversal.

- After the contents of field $x.f_1$ has been pushed to the stack the algorithm will never look at the original location $x.f_i$
  - It means we can use $x.f_i$ to store one element of the stack itself
  - $x.f_i$ will be made to point back to the record from which x was reached
  - When the stack is popped the field $x.f_i$ will be restored to its original value
  - The algorithm requires a field in each record called *done*
    - It should indicate how many fields in that record have been processed
    - This only takes a few bits per record
  - The variable t servers as the top of the stack
    - Every record x on the stack is already marked and it $i =$ done$[x]$ then $x.f_i$ is the stack link to the next node down
    - When popping the stack, $x.f_i$ is restored to its original value

9.2.4 An array of freelists

- The sweep phase is the same no matter which marking algorithm is used
  - It just puts the unmarked records on the freelist, and unmarks the marked records
- If records are of many different sizes, a simple linked list will not be very efficient for the allocator
  - Since when allocating a record of size n, we may have to search a long way down the list for a free block of that size
- A good solution is to have an array of several freelist
  - freelist[i] is a linked list of all records of size i
- The program can allocate a node of size i just by taking the head of freelist[i]
- The sweep phase of the collector can put each node of size j at the head of freelist[j]
- If the program attempts to allocate from an empty freelist[i],it can try to grab a larger record from freelist[j] (for $j > i$) and split it
  - Putting the unused potion back on freelist[$j - i$]
  - If this fails it is time to call the garbage collector to replenish the freelists

9.2.5 Fragmentation

- I can happen that the program want allocate a record of size n but there are many smaller that n
  - This is called **external fragmentation**
    - **Internal fragmentation** occurs when the program uses a too-large record without spliting it

9.3 Reference Counts

- Garbage collection can be done directly by keeping track of how many pointers point to each record
  - This is the **reference count** of the record and it is stored with each record
  - The compiler emits extra instructions so whenever p is stored into $x.f_i$
    - The reference count of p is incremented
    - The reference count of what $x.f_i$ previously pointed to decremented
  - If the decremented reference count of some record reaches zero then r is put on the freelist and all other records that r points to have their reference counts decremented
  - Instead of decrementing the counts of $r.f_i$ when r is put on the freelist it is better to do recursive decrementing when r is removed from the freelist for two reasons
    1. It breaks up the "recursive decrementing" work into shorter pieces
       - This makes the program run more smoothly
    2. The compiler must emit code (at each decrement) to check whether the count has reached zero and put the record on the freelist
       - The recursive decrementing will only be done in one place the allocator
  - There are two major problems with reference counting
    1. Cycles of garbage cannot be reclaimed
       - e.g. a loop of list cells that are not reachable from program variables but each has a reference count of 1
    2. Incrementing the reference counts is very expensive
       - Since in place of the single machine instruction $x.f_i \leftarrow p$ the execute

```
z       ← x.f_i
c       ← z.count
c       ← c − 1
z.count ← c
if c = 0 call putOnFreelist
x.f_i   ← p
c       ← p.count
c       ← c + 1
p.count ← c
```

- A naive reference counter will increment and decrement the counts on every since assignment to a program variable
  - Since this would be very expensive many increments and decrements are eliminated using dataflow analysis
    - As a pointer value is fetched and then propagated through local variables, the compile can aggregate the many change in the count to a since increment
  - Even with this technique there are many ref-counts increments and decrements that remain and their cost is very high
  - There are two possible possible solutions to the cycles problem
    1. Simply require the programmer to explicitly break all cycles when she is done with the data structure
       - It is less annoying than putting explicit free calls
       - It is hardly elegant
    2. Coming reference counting with an occasional mark-sweep collection
  - As a whole the problems with reference counting outweigh its advantages
    - It is rarely used

9.4 Copying Collection

9.4.1 General

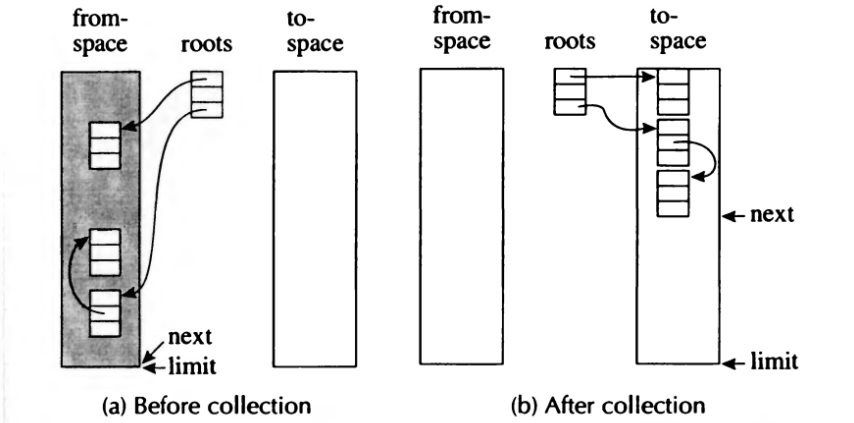

FIGURE 13.7. Copying collection.

- The reachable part of the heap is a directed graph with records as nodes and pointers as edges and program variables as roots
- **Copying garbage collection** traverses this graph in a part of the heap (called **from-space)** building a isomorphic copy in a fresh area of the head (called **to-space**)
  - The to-space copy is **compact**, occupying contiguous memory without fragmentation
  - The roots are made to point at the to-space copy
  - Then the entire from-space is unreachable
    - Garbage plus the previously reachable graph
  - It does not have a fragmentation problem
  - Eventually the program will allocate enough that `next` reaches `limit`
    - The another garbage collection is needed
    - The roles of to and from space are swapped
- **Initiating a collection:** to start a new collection, the pointer `next` is initialized to point at the beginning of to-space

– As a reachable record is found in the from space it is copied to to-space at position `next`, and `next` incremented by the size of the record

9.4.2 Forwarding

```
function Forward(p)
  if p points to from-space
    then if p.f1 points to to-space
         then return p.f1
         else for each field fi of p
                  next.fi ← p.fi
              p.f1 ← next
              next ← next+ size of record p
              return p.f1
    else return p
```

ALGORITHM 13.8. Forwarding a pointer.

- The basic operation of copying collection is forwarding a pointer
  - That is, given a pointer p that points to from-space, make p point to to-space
  - There are three cases:
    1. If p points to a from-space record that has already been copied
       * Then $p.f_1$ is a special *forwarding pointer* that indicates where the copy is
       * The forwarding pointer can be identified just by the fact that it points within the to-space, as no ordinary from-space field could point there
    2. If p points to a from-space record that has not yet been copied
       * It is copied to location `next`
       * The forwarding pointer is installed into $p.f_1$
       * It is all right to overwrite the f_1 field of the old record because all the data have already been copied to the to-space of `next`
    3. If p is not a pointer at all, or if it points outside from space, then forwarding p does nothing

9.4.3 Cheney's algorithm

```
scan ← next ← beginning of to-space
for each root r
    r ← Forward(r)
while scan < next
    for each field f_i of record at scan
        scan.f_i ← Forward(scan.f_i)
    scan ← scan+ size of record at scan
```

ALGORITHM 13.9. Breadth-first copying garbage collection.

- The simplest algorithm for copying collection uses BFS to traverse the reachable data
  - The roots are forwarded
    - This copies a few records to to-space thereby incrementing `next`
  - The area between `scan` and `next` contains records that have been copied to to-space
    - The fields has not yet been forwarded
    - These fields point in general to from-space
  - The area between the beginning of to-space and `scan` contains records that have been copied and forwarded
    - All pointers in this area point to to-space
  - The while loop of algorithm moves `scan` toward next
    - Copying records will cause next to move also
    - Eventually `scan` catches up with next after all reachable data are copied to to-space
  - It requires no external stack and no pointer reversal
    - It uses the to-space area between `scan` and `next` as the queue of its BFS
    - It makes it simpler to implement than DFS with pointer reversal

9.4.4 Locality of reference

- Pointer data structures copied by BFS have poor locality of reference
  - Since the records near each other are those whose distance from the roots are equal
  - Record near each other are not likely to be related
- In a computer system with virtual memory or memory a memory cache good locality of reference is important
  - After the program fetches address a then the memory subsystem expects addresses near a to be fetched soon
  - This ensures that the entire page or cache line containing nearby addresses can be quickly accessed

```
function Forward(p)
   if p points to from-space
     then if p.f1 points to to-space
            then return p.f1
            else Chase(p); return p.f1
     else  return p

function Chase(p)
   repeat
     q ← next
     next ← next+ size of record p
     r ← nil
     for each field fi of record p
         q.fi ← p.fi
         if q.fi points to from-space and q.fi.f1 does not point to to-space
         then r ← q.fi
     p.f1 ← q
     p ← r
   until p = nil
```

ALGORITHM 13.11. Semi-depth-first forwarding.

- Depth-first copying given better locality, since each object a tend to be adjacent to its first child b
  - This is unless b is adjacent to another "parent" a'

- A hybrid partly depth first and partly breadth first algorithm can provide acceptable locality
    - The basic idea is to use breadth-first copying but whenever an object is copied see if some child can be copied near it

9.5 Generational Collection

- Since in many programs new objects are likely to die soon whereas an objects still reachable after many collections will probably survive for many more collections
    - The collector should concentrate its effort on "young data"
    - Since there is a higher proportion of garbage
    - A heap is divided into **generations**
        - The youngest objects in generation G_0
        - Ever object in generation G_1 is older than any object in G_0
        - Everything in G_2 is older than G_1 and so on
    - To collect just G_0 just start from the roots and either depth-first marking or breadth-first copying
        - The roots are not just program variables the include any pointer within $G_1, G_2, \ldots$
        - If there are too many of these then processing the roots will take longer than traversal of reachable objects within G_0
        - Its rare for an older object to point to a much younger object
        - To void searching all of $G_1, G_2, \ldots$ for root of G_0 we make the compiled program remember where there are pointers from old objects to new once
    - There are several ways of remembering
        - **Remembered list:** The compiler generates code, after each *update* store of the form $b.f_i \leftarrow a$ to put b into a vector of updated object
            - At each garbage collection the collector scans the remembered list loking for old objects b that point into G_0
        - **Remembered set**: Like the remembered list, but uses a bit within object b to remember to record that b is already in the vector
            - The code generated by the compiler can check this bit to avoid duplicate reference to b in the vector

    - **Card marking:** Divide the memory into logical "cards" of size 2^k bytes
        - An object can occupy part of a card or start in the middle of one card and continue onto the next
        - Whenever address b is updated the card containing that address is marked
        - There is an array of bytes that server as marks
        - The byte index can be found by shifting address b right by k bits
    - **Page marking:** Is like card marking, but if 2^k is the page size the computers virtual memory system can be used instead of extra instructions generated by the compiler
        - Updating an old generation sets a dirty bit for that page
- When a garbage collection begins the remembered set tells which objects of the old generation can possible contains pointer into G_0 are scanned for roots
    - When using the copying collection only G_0 are copied
    - The marking algorithm does not mark old generation records
    - After several collections of G_0, generation G_1 may have accumulated a lot of garbage
        - Since G_0 may contain many pointers into G_1 it is best to collect G_0 and G_1
        - The remembered set must be scanned for roots contained in $G_2, G_3, \ldots$
    - Each older generation should be exponentially bigger than the previous one
    - An object should be promoted from G_i to G_{i+1} when it survives two or three collections of G_i
    - If the program does many more updates than fresh allocations generational collection may be more expensive than non generation collection

9.6 Incremental Collection

- Even if the overall garbage collection time is only a few percent of the computation time, the collector will occasionally interrupt the program for long periods

- For interactive or real-time programs this is undirable
- Incremental or concurrent algorithm s interleave garbage collection work with program execution to avoid long interruptions

```
while there are any grey objects
  select a grey record p
  for each field f_i of p
      if record p.f_i is white
        color record p.f_i grey
  color record p black
```

ALGORITHM 13.13. Basic tricolor marking.

- Terminology
  - The **collector** tries to collect garbage
  - The compiled program keeps changing the graph of reachable data so it is the **mutator**
  - An **incremental** algorithm is one in which the collector operates only when the mutator requests it
  - A **concurrent** algorithm is one where the collector can operate between or during any instructions executed by the mutator
- **Tricolor marking.** In a mark-sweep or copying garbage collection, there are three classes of records:
  - **White** objects are not yet visited by the depth-first or breadth-first search
  - **Grey** objects have been visited (marked or copied), but their children have not yet been examined
    - In mark-sweep collection, these objects are on the stack
    - In Cheney's copying collection they are between `scan` and `next`
  - **Black** objects have been marked an their children also marked
    - In mark-sweep collection, they have already been popped of the stack
    - In Cheney's copying collection they have already been scanned
- The collection starts with all objects white

  - The collector executes the basic tricolor marking algorithm
  - Blackening gray objects and graying their white children
  - In changing an object from gray to black is removing it from the stack or queue
  - When there are no gray objects then all the white objects must be garbage

- All the algorithms preserve two natural invariants

  1. No black object points to a white object
  2. Every gray object is on the collector's data structure
     - Called the gray set

- While the collector operates the mutator creates new object and updates pointer fields of existing objects

  - If the mutator breaks one of the invariants the collection algorithm will not work
  - Most incremental and concurrent collection algorithm are based on techniques which allows the mutator to get work done while preserving invariants

- Examples of incremental and concurrent collection algorithms:

Dijkstra, Lamport, et al. Whenever the mutator stores a white pointer *a* into a black object *b*, it colors *a* grey. (The compiler generates extra instructions at each store to check for this.)

Steele. Whenever the mutator stores a white pointer *a* into a black object *b*, it colors *b* grey (using extra instructions generated by the compiler).

Boehm, Demers, Shenker. All-black pages are marked read-only in the virtual memory system. Whenever the mutator stores *any* value into an all-black page, a page fault marks all objects on that page grey (and makes the page writable).

Baker. Whenever the mutator fetches a pointer *b* to a white object, it colors *b* grey. The mutator never possesses a pointer to a white object, so it cannot violate invariant 1. The instructions to check the color of *b* are generated by the compiler after every fetch.

Appel, Ellis, Li. Whenever the mutator fetches a pointer *b* from any virtual-memory page containing any nonblack object, a page-fault handler colors every object on the page black (making children of these objects grey). Thus the mutator never possesses a pointer to a white object.

- The first three are **writer-barrier** algorithms
  - It means that each store by the mutator must be checked to make sure that an invariant is preserved
- The last two are **read-barrier** algorithms
  - It means that fetch instruction are the one that must be checked
- Any implementation of a write or read barrier must synchronize with the collector
  - Software implementations of the read or write barrier will need to use explicit synchronization which can be expensie
  - Implementations using virtual-memory hardware can take advantage of the synchronization implicit in a page fault
    * i.e. if the mutator faults on a page the operating system will ensure that no other process has access to that page before processing the fault

9.7 Backer's Algorithm

- **Backer's Algorithm** illustrates the details of instrumental collection
  - It is based on Cheney's copying algorithm
  - It forward reachable object from-space to to-space
  - It is compatible with generational collection
    * From-space and to-space might be for generation G_0, or might be $G_0 + \cdots + G_k$
  - To initiate a garbage the roles of the from-space and to-space are swapped and all the roots are forwarded this is called the flip
    * The mutator is then resumed
    * Each time the mutator calls the allocator to get a new record, a few pointers at `scan` are scanned, so that `scan` advances toward next
      · The new record is allocated at the end of the to-space by decrementing `limit` by the appropriate amount
    * The invariant is that the mutator has pointers only to to-space

        - Thus when the mutator allocates and initializes a new record that record need not to be scanned
    - When the mutator stores a pointer into an old record it is only storing the to-space pointer
    - If hte mutator fetches a field of a record it might invariant
        - Each fetch is followed by two or three instruction that check wether the fetched pointer points to from-space
        - If so, the pointer must be forwarded immediately using the standard forward algorithm
    - For every word allocated, the allocated must advance `scan` by at least one word
        - When `scan=next` the collection terminates until the allocator runs out of space
    - The largest cost of the Baker's algorithm is the extra instructions after every fetch

9.8 Interface to the Compiler

9.8.1 General

- The compiler for a garbage-collected language interacts with the garbage collector by generating code that allocates record
    - By describing locations of roots for each garbage-collection cycle
    - By describing the layout of data records on the heap
    - For some versions of incremental collection the compiler must also generate instructions to implement a read barrier or write barrier

9.8.2 Fast Allocation

- Some programming languages and some programs allocate heap data very rapidly
    - To minimize the cost of the garbage collector **copying collection** should be used so that the allocation space is a contiguous free region
    - The next free location is `next`
    - The end of the region is `limit`
    - To allocate one record of size N the steps are

1. Call the allocate function.
2. Test `next` + N < `limit` ? (If the test fails, call the garbage collector.)
3. Move `next` into `result`
4. Clear M[`next`], M[`next` + `1`], ..., M[`next` + `N` − `1`]
5. `next` ← `next` + N
6. Return from the allocate function.
A. Move `result` into some computationally useful place.
B. Store useful values into the record.

- Steps 1 and 6 should be eliminated by inline expanding the allocate function at each place where a record is allocated
- Step 3 can often by eliminated by combining it with step A
- Step 4 can be eliminated in facor of step B
- Steps 2 and 5 cannot be eliminated but if there is more than once allocation in the same basic block then the comparison and increment can be shared among multiple allocations
- By keeping `next` and `limit` in registers steps 2 and 5 can be done in a total of three instructions
- By using these techniques the cost of garbage collection can be reduced to 4 instructions

9.8.3 Describing Data Layout

- The collector must be able to operate on records of all types: `list`, `tree` or whatever the program has declared
  - It must be able to determine the number of fields in each record and whether each field is a pointer
  - In statically typed languages or object oriented the simplest way to identify heap objects is to have the first word of every object point to a special type or class descriptor record
    * For statically typed language is an overhead of one word
    * For object oriented languages this descriptor pointer needs to be in every object just to implement dynamic method lookup
      · No additional per-object overhead attributable to garbage collection

- The type- or class-descriptor must be generated by the compiler from the semantic analysis phase of the compiler
    - The descriptor-pointer will be the argument to the runtime systems `alloc` function
- The compiler must identify to the collector every pointer containing temporary and local variable
    - This is whether it is in a register or in an activation record
        - Since the set of live temporaries can change at every instruction
        - The pointer map is different at every point in the program
        - It is simpler to describe the pointer map only at points where the garbage collector can begin
        - Calls to the `alloc` function
    - Any function might be calling a function that in turn calls `alloca`
    - The pointer map must described at each function call
- It is best keyed by return address
    - A function call at location a is best described by its return address directly after a
- The data structure maps return addresses to live-pointer sets
    - For each pointer that is live immediately after the call the pointer map tells its location
- To find all the roots the collector starts at the top of the stack and scans downward frame by frame
- Callee-save registers need special handling
    - They must be inherited from the calling function

9.8.4 Derived Pointers

- t_1 is derived from the base pointer a if it points to a place in that record
    - The pointer map must identify each derived pointer and tell the base pointer from which it is derived
    - When the collector relocates a to address a' it just adjeust t_1 to point to address $t_1 + a^{'} - a$

10 Exam

- Læs om recursive descent parsing