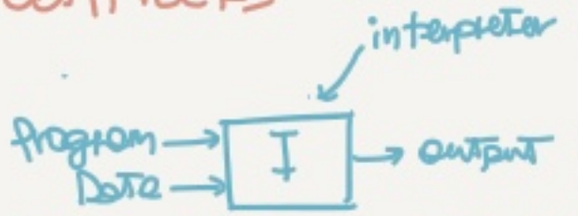
COMPILERS

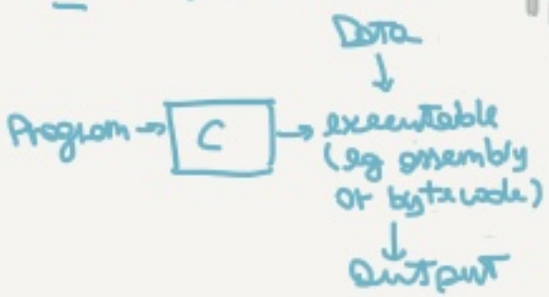


- Part to develop the program
- slower than compiled
- Takes more space

It doesn't do any processing of the program before it executes. It immediately begins running on input

We can say it is on-line, meaning the work it does it's on part of running your program

—



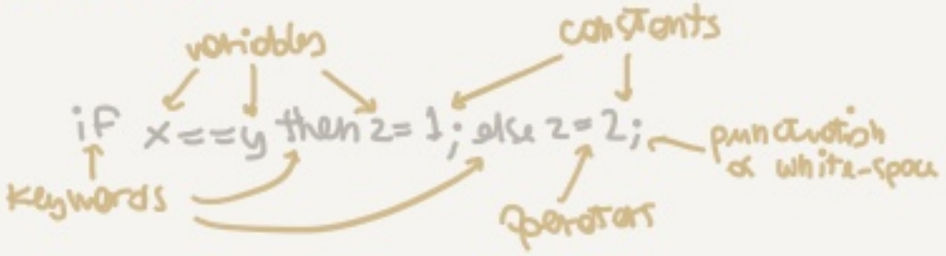
"Formulas translated, or FORTRAN", Backus
- First compiler
- Modern compilers preserve the outlines of their compiler

A compiler is off-line. It pre-processes the program first that produces an exec that can be run on many different inputs without having to recompile or do any other processing of the program

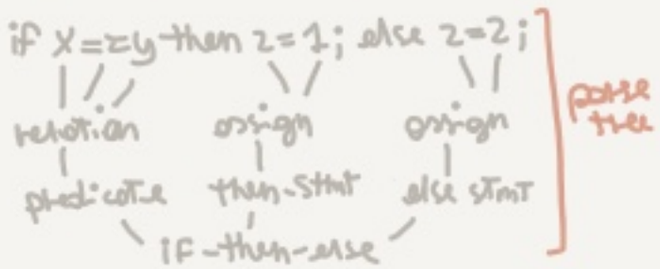
—

Lexical analysis] Syntactic
Parsing]
Semantic analysis — Semantic (eg types, scope)
Optimisation
Code generation — translation

Lexical analysis, divide the program into tokens



Parsing, understand the structure of the sentence



Then we want to understand the meaning, and this is hard!

Compilers can only do very limited semantic analysis. It generally tries to catch inconsistences

Programming languages define rules to avoid ambiguities

```
{ joe=1
  { joe=3
    print joe }
}
3
```

it will print 3 as the outer definition is hidden

Optimisation, to use less of some resource
- Run faster
- Less memory
- Power
- Network usage
- ...

$X = Y * 0$ ⇓ $X = 0$

This is NOT a correct rule. It's not always obvious when to do optimisation.

This is valid for integers only but not for floating points ($NAN * 0 = NAN$)

Code generation (or code gen.)

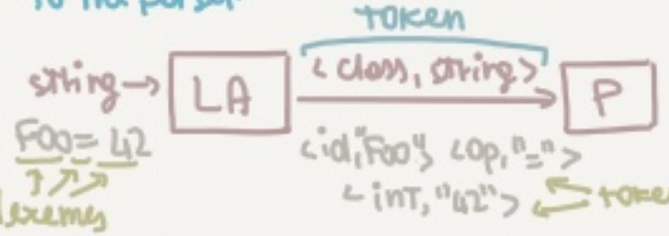
It usually produces assembly code. But in general is a translation into some other language

Lexical Analysis

```
if(i==j)
  z=0;
else
  z=1;
```
⇒ `\tif (i==j)\n\t\tz=0;\n\t`
`... else\n\t\tz=1;`
↑ what the lexer receives

It divides the string into tokens and classifies them into token classes
- identifiers, string of letters or digits starting with a letter
- integer, a non-empty string of digits
- keywords, reserved words such as "if", "begin"
- whitespace, a non-empty sequence of blanks, newlines, and tabs

It will then communicate the tokens to the parser



`\tif_(i==j)\n\t\tz=0;\n\telse\n\t\tz=1;`
↑ one token!

operator | ( ) : ; =
whitespace
keywords
identifiers
numbers
↑ classes by themselves

Lexical analysis examples

FORTRAN: whitespace is insignificant ↩

`DO 5 I=1,25` ← FORTRAN loop
`DO 5 I=1.25` ← variable assign.
VA R1 is the same as VAR1

To understand the role of DO we **lookahead** until there is no ambiguity (. or , in that case)

Lookaheads complicates the implementation of lexical analysis: we want to minimize them when building a compiler

When partitioning the string from left to right, lookahead may be required to decide where one token ends and the next begins

```
if (i==j)
  z=0;
else
  z=1;
```
is 'i' a variable ← is this an assignment

Regular languages

which set of strings belongs to each token is specified by a regular language

To define a regular language we use regular expressions.

$'c' = \{'c'\}$
$\varepsilon = \{''\}$ ← language with only an empty string. It is NOT an empty language

$A+B = \{a \mid a \in A\} \cup \{b \mid b \in B\}$ union

$AB = \{ab \mid a \in A \wedge b \in B\}$ concat.

$A^* = \bigcup_{i \geq 0} A^i \quad A^i = A \dots A$ (i times) $\quad A^0 = \varepsilon$

$\Sigma = \{0,1\}$

$1^* = \bigcup_{i \geq 0} 1^i = \underbrace{1 + \dots 1}_{i}$ ← all strings of 1's

$(1+0)1 = \{ab \mid a \in 1+0 \wedge b \in 1\} = \{11, 01\}$

$0^* + 1^* = \{0^i \mid i \geq 0\} \cup \{1^i \mid i \geq 0\}$

$(0+1)^* = \bigcup_{i \geq 0} (0+1)^i$ ↳ $(0+1)^0 = \varepsilon, 0+1, (0+1)(0+1), \dots$

$R = \varepsilon \mid 'c' \; c \in \Sigma \mid R+R \mid RR \mid R^*$ ← grammar for a RE

also denoted as Σ^*

Formal languages

A formal language has an alphabet Σ

A language over Σ is a set of strings of characters drawn from Σ

eg Σ = ASCII Language = C programs

A meaning function maps syntax to semantics

$L(e) = M$ (reg exp → set of strings)

$L('c') = \{'c'\}$
$L(A+B) = L(A) + L(B)$

We use meaning function to
- Makes clear what is syntax, what is semantics
- Allow us to consider notation as a separate issue
- Because expressions and meaning are not in one to one correspondence

Meaning is many to one, never one to many. The meaning of expressions in our programming language must be unambiguous.

lexical specifications

keyword: "if" or "else" or "then" or...

concat: `'i'+'f'+'e'+'l'+'s'+'e'+'t'+'h'+'e'+'n'+...` / `'if'+'else'+...`

integer: non-empty string of digits

digit = '0'+'1'... '9' ← most tools allow naming

digit digit* = digit+ ← digits

identifier: strings of letters or digits starting with a letter

letter = 'a'+'b'+'c'... 'z'+'A'... 'Z' → [a-zA-Z] ← character range

letter (letter + digit)*

whitespace: non-empty sequence of blanks, newline and tabs

('' + '\n' + '\t')+

eg PASCAL

digit = '0'+... '9'
digits = digit+
opt_fraction = ('.' digits) + ε → ('.' digits)? ← that means optional
opt_exponent = ('E' ('+'+'-' + ε) digits) + ε
num = digits opt_fract opt_exponent

Lexical specification

A^+, AA^*

$A|B$, $A+B$

'a'+'b'...'z', [a-z]

[^a-z] — any character except that range

complement of [a-z]

1 number = digit+
keyword = ...
identifier = letter(letter + digit)*
openpar = '('
...

first step of lexical spec. is to write a RE for each syntactic category in the language

2 R = keyword + identifier + ...

then we construct R, matching all lexemes for all tokens. This is just the union of the previous regular expressions

3 Let input be $x_1 \dots x_n$

For $1 \le i \le n$ check $x_1 \dots x_i \in L(R)$

take the input, for every pre-fix of that input we check whether it is in the language of R

4 $x_1 \dots x_i \in L(R_j)$ for some j

...If it is in the language, we know that it is in the language of one of the constituent RE.

5 Remove $x_1 \dots x_i$ from input and go to step 3

Remove that prefix and look for the next one, until the string is empty

In the case where
$x_1 \dots x_i \in L(R)$
$x_1 \dots x_i \in L(R_j)$
$x_1 \dots x_i \in L(R_k)$
if $\in L(\text{keywords})$
if $\in L(\text{identifiers})$

Choose the one listed first. They have priority ordering

In the case where
$x_1 \dots x_i \in L(R)$ and $x_1 \dots x_j \in L(R)$ and $j > i$

Maximal munch, always take the longer one.

IF no rule matches
$x_1 \dots x_i \notin L(R)$
Don't let this to happen!
ERROR = all strings not in the lexical spec
put it last in priority (so that it can overlap other RE's in the language)

Finite automata

QUICK RECAP!

is a 5-tuple with
- Σ alphabet
- S, states
- n, start state
- $F \subseteq S$ final states
- set of transitions

$s_1 \to^a s_2$ in s_1 with input a go to s_2

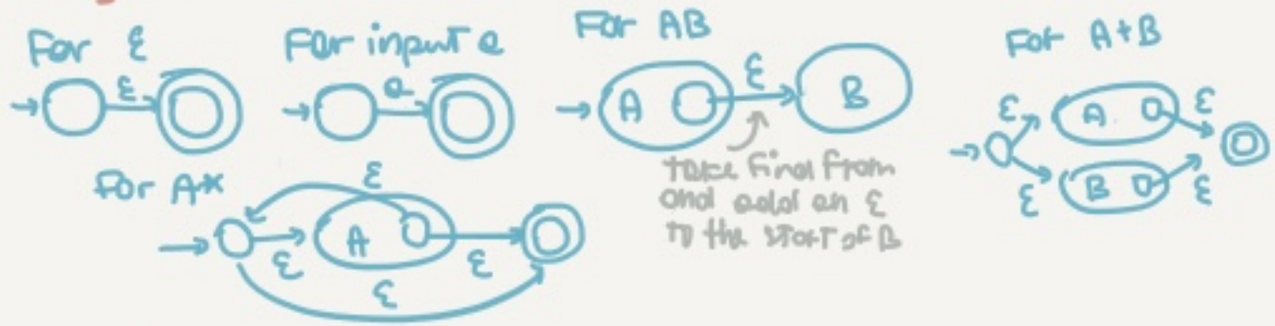
If after reading the string you're in a final state, accept, the string is in the language, else reject.
If the machine gets stuck (no transitions on the input), also reject.

NFA's have ε-moves, meaning that the machine can either change state without consuming any input, or to not move. More transitions per state makes the automata non-determ.

An NFA then, can wind up in any number of states. We accept if an accept state is reached after reading the string

An NFA is not more powerful than a DFA

Regex TO NFA



NFA to DFA

ε-closure, starting from a state, it is all the states we can reach by only following ε-moves.

ε-closure(A) = {A, B, D}

An NFA can be in many states at the same time. But how many?

N states
$|S| \le N$

$2^N - 1$ possible subsets of n states

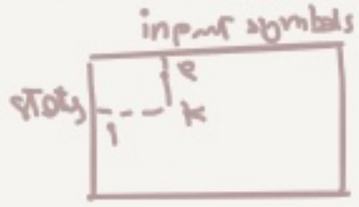
NFA
states S
start $s \in S$
final $F \subseteq S$

$a(X) = \{y \mid x \in X \wedge x \xrightarrow{a} y\}$
for a set of states X, show me all reachable states on input a

DFA
states, subsets of S
start, ε-closure(s)
Final, $\{x \mid x \cap F \neq \emptyset\}$
$X \xRightarrow{a} Y$ if
$Y = \varepsilon\text{-clos}(a(X))$

Implementing DFA

A DFA can be implemented as a two-dimensional array
- one dimension is the state
- other is input symbol
- for every transition $S_i \xrightarrow{a} S_k$ define $T[i,a] = k$




```
// start input and
   state index 0
i=0;
state=0;
while (input[i])
    state= A[state,
             input[i++]]
```

With that representation there are lots of repetitions
To avoid that we could..

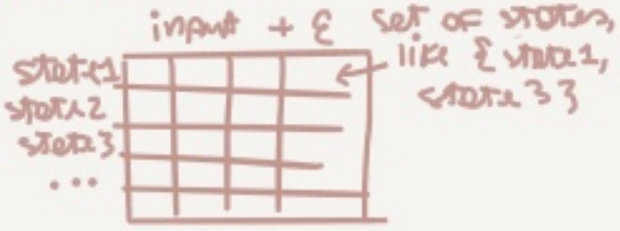
there's a pointer to the input rows. That way you can share duplicated rows Pointers to other table, will make lookup a bit slower.

Lexical analyser construction

What we covered so far!

What if we want from NFA directly to table, maybe because our specification would make the DFA table huge.
We can do that but...

Now the inner loop is gonna be much more expensive.



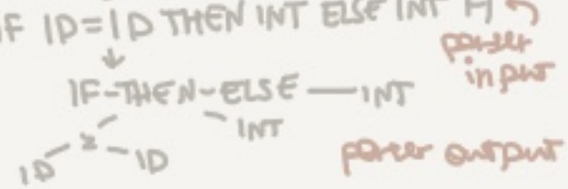
Parsing

Regular languages can't express all languages
ie $\{(^i)^i \mid i \geq 0\}$ language of balanced parentheses

It can only count mod k where k is the number of states
It cannot count to an arbitrary i, with a finite set of states

A parser takes a sequence of tokens from the lexer and it produces a parse tree of the program

cool, if x=y then 1 else 2 fi



In some compilers, the parser is powerful enough to explore lexical analysis in addition to parsing

Context-free grammars

We need a way to describe valid strings of tokens and a method to distinguish valid ones

Programming languages have a recursive structure.

cool, - EXPR ← can be if while...
- if EXPR then EXPR else EXPR fi

These expressions are recursively composed of other expressions

Context-free grammars are used to describe such recursive structures.

A CFG consists of
- set of terminals T
- set of non-terminals N
- start symbol S
- set of productions

$X \to y_1 \dots y_N$ ← production
$X \in N$, $y_i \in N \cup T \cup \{\varepsilon\}$

eg $S \to (S)$ $S \to (S)$
$S \to \varepsilon$ $S \to ((S))$

- Begin with S, top symbol usually
- Replace any non-terminal X by the right hand side of some production
- Repeat until there are no terminals

Let G be a CFG, L(G) is the language of G

- Terminals are so called because there are no rules for replacing them. Once generated, they are permanent
- Terminals ought to be the tokens

eg. cool

```
EXPR → if EXPR then EXPR
       else EXPR fi |
       while EXPR loop
       EXPR pool | id ...

E → E+E | E*E | (E) | id
```

- A CFG G just tells us whether the string is in L(G) or not
- We want to handle these errors gracefully

Derivation

$S \to \ldots \to \ldots \to \ldots$ ← series of productions

can be drawn as a tree

$E \to E+E \mid E*E \mid (E) \mid id$

$E \to E+E \to$
$E*E+E \to$
$id*E+E \to$
$id*id+E \to$
$id*id+id$

parse tree

Parse trees have terminals at the leaves and non-terminals at the interior nodes

An in order traversal of the leaves is the original input

It shows associations of operations. In the example the "*" binds more tightly than the +, as it is at a lower level, we do * first before +.

The example shows a leftmost derivation where at each step, we replace the left-most non-terminal

Right-most and left-most derivations have the same parse tree

Ambiguity

$E \to E+E \mid E*E \mid (E) \mid id$
$id*id+id$

There are two parse trees for this grammar

Each of this parse trees have many derivations. This is not about derivations.

This grammar is ambiguous, it has more than one parse tree for some string

Ambiguity is bad, leaves meaning of some programs ill-defined

To eliminate ambiguity, write a new grammar that is unambiguous

$E \to E'+E \mid E'$
$E \to id*E' \mid id \mid (E)*E' \mid (E)$

Enforces precedence of * over +!

---

if E_1 then if E_2 then E_3 else E_4

"else" must match with the closest unmatched "then"

$E \to MIF$ /* all "then" are matched
$\mid UIF$ /* some "then" are matched"

$MIF \to$ if E then MIF else MIF | OTHER
$UIF \to$ if E then E | if E then MIF else UIF

It's impossible to automatically convert an ambiguous grammar to an unambiguous one

Used with care, ambiguities can simplify the grammar
- Allows more natural definitions
- We need some disambiguation method

Most practical parsing tools use this approach, the tool provide disambiguating declarations

Most tools allow precedence and associativity declarations

$E \to E+E \mid int$

left associativity declaration

we declare + to be left associative

%left + "Notation used in bison"

$E \to E+E \mid E*E \mid int$

precedence declarations

%left +
%left *

the order gives precedence, * has higher precedence

Top-down parsing

A compiler
- translates valid programs
- detect non-valid programs

Many kinds of errors, lexical, syntax, semantic or correctness errors

An error handler should...
- Report error accurately and clearly
- Recover from an error quickly
- Not slow down compilation of valid code

Panic mode is the simplest, most popular method

When an error is detected
- Discard tokens until one with a clear role is found
- continue from there

Look for synchronizing tokens, to identify where we are.

eg (1++2)+3 ← ++ not in the language, just skip it

$E \to \ldots \mid error\ int \mid (error)$

Bison uses the special terminal error to describe how much input to skip

They inserted last so that they'll run only if the previous ones didn't match

●●●

Error productions

Specify known common mistakes in the grammar

eg - E x instead of 5 * x
- Add $E \to EE$
- Complicates the grammar
  Harder to understand, maintain...

People do it in practice, warn things you're not supposed to do, but accept them anyway.

Error correction (rather than just detecting as previous methods do)

Find a correct "nearby" program
- Try token insertions and deletions
- Exhaustive search (edit distance)

Disadvantages
- Hard to implement
- Slows down parsing
- "Nearby" is not necessarily the intended program

Abstract syntax tree

A parser traces the derivation of a series of tokens.

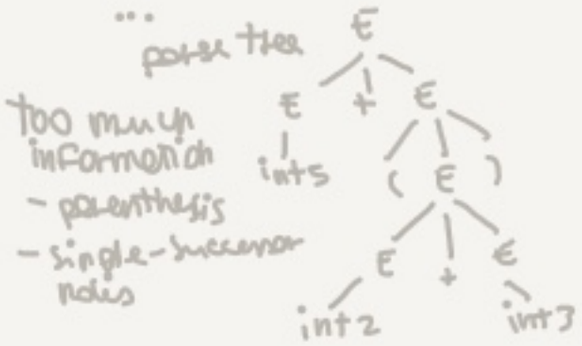
But the rest of the compiler needs a structural representation

Abstract syntax tree, AST
- Like parse trees but ignore some details

eg $E \to int \mid (E) \mid E+E$ $5+(2*3)$
int_5 '+' '(' int_2 '*' int_3 ')' ← lexical analysis

...

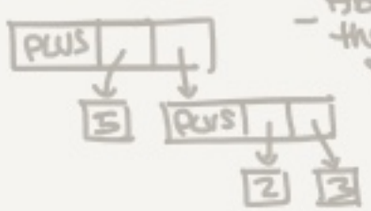
... parse tree



too much information
- parenthesis
- single-successor nodes

↓

An AST representing the same thing



- Abstracts from the concrete syntax
- More compact and easier to use

Recursive descent parsing

It is a top-down parsing algorithm

the parse tree is constructed
- From top
- From left to right

terminals are seen in order of appearance in the token stream

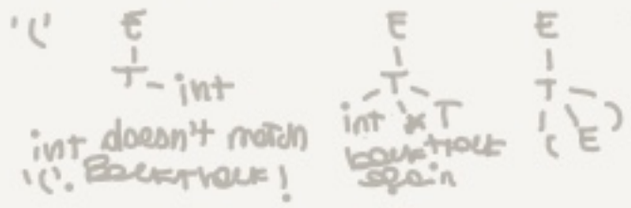
$t_2\ t_5\ t_6\ t_8\ t_9$

Consider this grammar

$E \to T \mid T+E$
$T \to int \mid int*T \mid (E)$

token stream is (ints)

- start with top-level non-terminal
- Try rules for that terminal in order



Recursive descent algorithm

- Let TOKEN be the type of tokens
- Let the global next point to the next input TOKEN

First, define boolean functions that check for matches
- A given token terminal
  `bool term(TOKEN tok) {return *next++ == tok;}`
- the nth production of S
  `bool Sn() {...}`
- try all productions of S
  `bool S() {...}`

For production $E \to T$
`bool E1 {return T();}`

For production $E \to T+E$
`bool E2() {return T() && term(PLUS) && E();}`

evaluates in left to right order

for all E productions
```
bool E(){
  TOKEN *save=next
  return (next=save, E1()) || (next=save, E2());}
```
↑ to allow backtracking

not really necessary

first try E_1; if that fails, E_2 is not evaluated

Functions for non-terminal T

`bool T1() {return term(INT);}` $T \to int$

`bool T2() {return term(INT) && term(TIMES) && T();}` $T \to int*T$

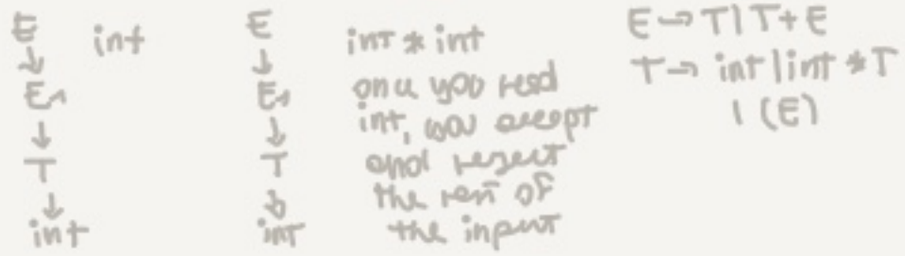
`bool T3() {return term(OPEN) && E() && term(CLOSE);}` $T \to (E)$

```
bool T(){ TOKEN *save=next
  return (next=save, T1())
      || (next=save, T2())
      || (next=save, T3())
```

restore input pointer before trying next alternative

To start the parser
- Initialise next to point to the first token
- Invoke E(), to match everything derivable from the start symbol

Recursive descent algorithm limitations



While there is backtracking for a non-terminal, there is no backtracking once we found a production that succeeds for a non-terminal.

General descent algorithms have that kind of backtracking

This algorithm however
- is easy to implement by hand
- sufficient for grammars where for any non-terminal, at most one production can succeed
- Grammars can be rewritten to work with this algorithm.

Left recursion

S→Sa bool S₂(){return S() && term(a);}
bool S(){return S₂();}

S() goes in an infinite loop

A left recursive grammar has a non-terminal S
$S \to^{+} Sd$ for some d

S→Sα|β

Recent parsing wants to see the first part of the input first. In left recursions we produce the string right to left.

We can fix it by using right recursion

S→βS'
S'→αS'|ε

—

S→Sd₁|...|Sdn|β₁|...|βm More general form

S→β₁S'|...|βmS'
S'→α₁S'|αnS'|ε

—

S→Aα|δ ← still left recursive grammar!
A→Sβ

This grammar cannot be predicted. We need to left_factor the grammar

E→T+E|T
T→int|int*T
|(E)

E→Tx
x→+E|ε
T→int y|(E)
y→*T|ε

—

Method similar to recursive descent
- For the leftmost non-terminal S
- We look at the next input Token a
- look up in the LL(1) table the production to use for [S,a]

A stack of records to record the Frontier
- Non-terminals that have yet to be expanded
- Terminals that have yet to be matched against input
- Left most pending terminal or non-terminal is at the top of the stack

Bottom-up Parsing

Predictive parsing

Like recursive descent but parser can "predict" which production to use
- By looking at the next few tokens ← lookahead
- No backtracking

Predictive parsers accept LL(k) grammars
- L: Left to right read input
- L: Left-most derivation
- (k): k tokens of lookahead (usually 1)

—

In LL(1), at each step, only one production
No backtracking, every other production is guaranteed to be incorrect.

E→T+E|T
T→int|int*T
|(E)

- T has two productions starting with int
- E // with T.

Reject on reaching error state
Accept on end of input and empty stack.

stack: bottom ↓

initialize stack = <S$>
repeat
  case stack of
  <x, rest>:
  if T[x, next] = y1...yn
  then stack ← y1...yn rest
  <t, rest>:
  if t == next++
  then stack ← <rest>
  else error();
until stack == <>

First sets

consider $A, A \to \alpha$, and token t

$T[A,t] = \alpha$ in two cases:

if $\alpha \to^* t\beta$
- α can derive t in the first production
- we say $t \in \text{first}(\alpha)$

if $A \to \alpha$ and $\alpha \to^* \varepsilon$ and $S \to^* \beta A t \delta$
- Useful if stack has A, input is t, and A cannot derive t
- In this case the only option is to get rid of A (by deriving ε)
  Can work only if t can follow A in at least one derivation
- We say $t \in \text{follow}(A)$

—

- $\text{first}(x) = \{t \mid x \to^* t\alpha\} \cup \{\varepsilon \mid x \to^* \varepsilon\}$

Algorithm

1. $\text{first}(t) = \{t\}$
2. $\varepsilon \in \text{first}(x)$
   - if $x \to \varepsilon$
   - if $x \to A_1 \dots A_n$ and $\varepsilon \in \text{first}(A_j)$ for $1 \le i \le n$
3. $\text{first}(\alpha_i) \subseteq \text{first}(x)$ if $x \to A_1 \dots A_n$ and $\varepsilon \in \text{First}(A_i)$ for $1 \le i \le n$

$E \to TX \quad X \to +E \mid \varepsilon \quad T \to (E) \mid \text{int}\, Y \quad Y \to *T \mid \varepsilon$

$\text{First}(E) \supseteq \text{first}(T)$

$\text{First}(T) = \{(, \text{int}\}$
$\text{First}(E) = \{(, \text{int}\}$
$\text{First}(X) = \{+, \varepsilon\}$
$\text{first}(Y) = \{*, \varepsilon\}$

$\text{first}(+) = \{+\}$
$\text{First}(*) = \{*\}$
$\vdots \quad \{(\}$
$\{)\}$
$\{\text{int}\}$

Follow sets

$\text{Follow}(x) = \{t \mid S \to^* \beta x t \delta\}$

t is follow(x) if there is some derivation where t appears right after x

IF $X \to AB$ then

$\text{First}(B) \subseteq \text{Follow}(A)$ $X \to AB \to^* AtB$
$\text{Follow}(X) \subseteq \text{Follow}(B)$ $S \to \dots Xt \dots \to ABt$

if $B \to^* \varepsilon$ then
$\text{Follow}(X) \subseteq \text{Follow}(A)$ $S \to \dots Xt \dots \to ABt \to At$

IF S is the start symbol then $\$ \in \text{Follow}(S)$ (end of the input symbol)

Algorithm

1. $\$ \in \text{follow}(S)$
2. $\text{First}(\beta) - \{\varepsilon\} \subseteq \text{Follow}(x)$
   - For each production $A \to \alpha x \beta$
3. $\text{Follow}(A) \subseteq \text{follow}(x)$
   - For each production $A \to \alpha X \beta$ where $\varepsilon \in \text{First}(\beta)$

$E \to TX \quad X \to +E \mid \varepsilon$
$T \to (E) \mid \text{int}\, Y \quad Y \to *T \mid \varepsilon$

equal $\begin{cases} \text{Follow}(X) \subseteq \text{Follow}(\varepsilon) & \text{because } X \to +E \\ \text{Follow}(E) \subseteq \text{Follow}(X) & \text{because } E \to TX \end{cases}$
$\text{Follow}(E) \subseteq \text{follow}(T)$ $E \to TX \to T$
equal $\begin{cases} \text{Follow}(Y) \subseteq \text{follow}(T) & Y \to *T \\ \text{Follow}(T) \subseteq \text{follow}(Y) & T \to \text{int}\, Y \end{cases}$

by definition

$\text{follow}(E) = \{\$,)\}$
$\text{Follow}(X) = \{\$,)\}$
$\text{Follow}(T) = \{+, \$,)\}$
$\text{Follow}(Y) = \{+, \$,)\}$

$\text{follow}('(') = \{(, \text{int}\}$
$\text{follow}(')') = \{+, \$,)\}$
$\text{follow}('+') = \{(, \text{int}\}$
$\text{follow}('*') = \{(, \text{int}\}$
$\text{follow}(\text{int}) = \{*, +, \$,)\}$

LL1 Parsing Table

Construct parsing table T for CFG G

For each production $A \to \alpha$ in G do
- For each terminal $t \in First(\alpha)$ do
  $T[A,t] = \alpha$
- If $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do
  $T[A,t] = \alpha$
- If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do
  $T[A,\$] = \alpha$

$E \to TX$ $\quad X \to +E \mid \varepsilon$
$T \to (E) \mid int\ Y$ $\quad Y \to *T \mid \varepsilon$

we need to know Follow(x) in order to know when to use $X \to \varepsilon$

$Follow(x) = \{\$,)\}$
$Follow(Y) = \{+, \$,)\}$

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

blank entries are parsing errors

— =

$S \to Sa \mid b$

$First(S) = \{b\}$
$Follow(S) = \{\$, a\}$

	a	b	$
S		b/Sa	

multiply defined!

If any entry is multiply defined, then G is not LL1
- not left factored
- left recursive
- ambiguous
- other grammars are not LL1

Most programming language CFGs are not LL1

Bottom-up parsing

Bottom up parsing is more general than top-down parsing

- Just as efficient
- Builds on ideas in top-down parsing

Bottom-up is the preferred method!

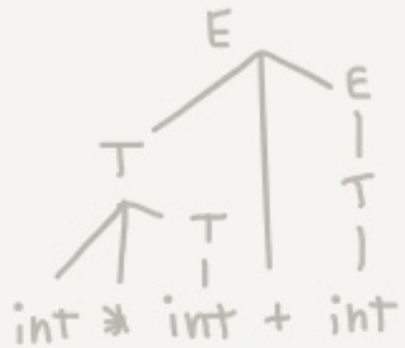
- For our example, we no longer need left factored grammars

E → T+E | T
T → int*T | int | (E)

Bottom-up parsing reduces a string to the start symbol by inverting productions

input string
int * int + int
int * T + int
T + int
T + T
T + E
E ← start symbol

Parse tree produced...



A bottom-up parser traces a rightmost derivation in reverse

IMPORTANT FACT #1

Shift reduce parsing

- Let αβw be a step of a bottom-up parse
- Assume the next reduction is by X → β
- Then w is a string of terminals

αXw → αβw is a step in a right-most derivation

Split string into two substrings

- Right substring is the unexamined input
- Left substring has terminals and non-terminals
- Split marked by a |

Bottom-up parsing uses only two kinds of actions. Those are shift and reduce

- Shift move | one place to the right

ABC|xyz ⇒ ABCx|yz

- Reduce is to apply an inverse production at the right end of the left string

if A → xy
Cbxy|ijk ⇒ CbA|ijk

|int * int + int shift
int |* int + int shift
int * |int + int shift
int * int |+ int reduce
int * T |+ int reduce
...

...
T |+ int reduce
T + |int shift
T + int| reduce
T + E | reduce
T + T | reduce

Left string can be implemented by a stack
- Top of the stack is 1

Shift pushes a terminal onto the stack

Reduce pops symbols from top of the stack and pushes one terminal on the stack

—=

IF it is legal to both shift and reduce, we say there's a shift-reduce conflict

IF it is legal to reduce by two productions, then there's a reduce-reduce conflict
↑ BAD

Handlers

How to decide when to shift/reduce?

$E \to T+E \mid T \quad T \to int * T \mid int \mid (E)$

...

$int \mid * int + int$

↑ we could decide to reduce, that would be a mistake!

We want to reduce only if the result can still be reduced to the start symbol

Assume $S \to^* \alpha X w \to \alpha \beta w$ (reduction)

Then $\alpha\beta$ is a handle of $\alpha\beta w$. A handle is a reduction that also allows further reductions back to the start symbol

We only want to reduce handles!

IMPORTANT FACT #2

In shift-reduce parsing handles appear only at the top of the stack!

Recognising handles

There are no efficient algorithms to recognise handles. But there are good heuristics for guessing them and for some CFG they always guess correctly

—=

α is a viable prefix if there is an w (rest of input) such that $\alpha | w$ is a valid configuration of a shift-reduce parser
↑ stack

- A viable prefix does not extend past the right end of a handle
- It is viable because it is a prefix of the handle
- As long as the parser has viable prefixes on the stack, no error has been encountered

For any grammar, the set of viable prefixes is a regular language

IMPORTANT FACT #3

Recognizing viable prefixes

1 Add a dummy production $S' \to S$ to G

2 The NFA states are the items of G
- The NFA reads the stack and it is either gonna say 'yes' it's valid, or no.

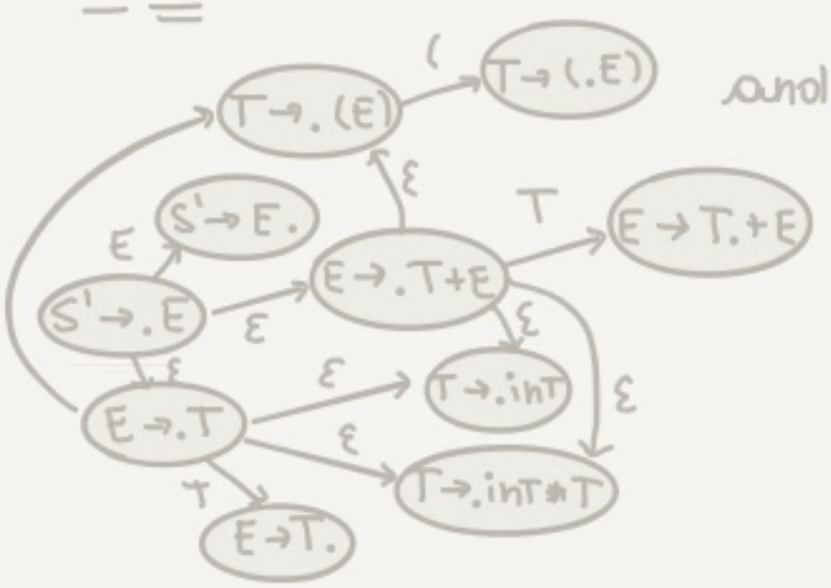
3 For item $E \to \alpha . X \beta$ (seen α on stack) add transition
$E \to \alpha X . \beta \to^{X} E \to \alpha X . \beta$

4 For item $E \to \alpha . X \beta$ (non-terminal) and production $X \to \gamma$ add
$E \to \alpha . X\beta \to^{\varepsilon} X \to . \gamma$

5 Every state is an accepting state

6 Start state is $S \to S'$

— =



and so on...

Valid items

Using the standard subset of states construction, we can build a DFA that is equivalent to the previous NFA

Each state is a set of items

Those states are called canonical collection of LR(0) items

— =

Item $X \to \beta . \gamma$ is valid for a viable prefix $\alpha\beta$ if
$S' \to^{*} \alpha X w \to \alpha\beta\gamma w$
by a rightmost derivation

After parsing $\alpha\beta$, the valid items are the possible tops of the stack items

SLR parsing

LR(0) parsing
- state contains α
- next input is t
- DFA on input α terminates in state s

Reduce by $X \to \beta$ if
- s contains item $X \to \beta .$

Shift if
- s contains item $X \to \beta . t\, w$
- equivalent to saying s has a transition labeled t

•••

• • •

LR(0) has a reduce-reduce conflict if
- Any state of the DFA has two reductions

LR(0) has a shift-reduce conflict if
- Any state has a reduce item and a shift item

—=

SLR stands for simple LR

SLR improves LR(0) by adding some heuristics that will refine when we shift and when we reduce, so that fewer states have conflicts

We add one condition to the reduction case in LR(0)

...
- $t \in$ follow(X) ← additional rule

—=

If there are conflicts under these rules, then the grammar is not SLR

—=

These rules amount to a heuristic for detecting the handles

—=

Algorithm

1 Let M be a DFA for viable prefixes of G
2 Let $|x_1 \dots x_n\$$ be initial configuration
3 Repeat until configuration is $S|\$$
- Let $\alpha|w$ be current configuration
- Run M on current stack α
- If M accepts α with items I, let 'a' be the input
  - Shift if $X \to \beta.a\gamma \in I$
  - Reduce if $X \to \beta. \in I$ and $a \in$ follow(X)
  - Report parsing error if neither applies

In SLR(k), k is the amount of lookaheads, in practice k=1

—=

SLR Improvements

Rerunning the viable prefix automaton at the top of the stack is wasteful
- Most work is repeated

To avoid repeating, we want to remember the state of the automaton at each prefix

Change stack to contain pairs ⟨symbol, state⟩

The state is the result of running the DFA on all symbols to its left

At the bottom of the stack we will have ⟨any, start⟩
- 'any' is any dummy symbol
- 'start' is the start state

—=

Define a table goto that maps a symbol to another state

This is just the transition functions of the DFA written as an array

—=

Our algorithm moves are
- Shift; push ⟨a, x⟩ on the stack
  - a is the current input
  - x is a DFA state
- Reduce $X \to \alpha$ (just as before)
- Accept and Error moves

—=

Define an action table which tells us which move to make in every possible state

The action table is indexed by the state and the next input symbol. It returns one of the moves above

• • •

For each state s_i and terminal a

- IF s_i has item $X \to \alpha . a \beta$ and goto$[i,a] = j$ then action$[i,a]$ = shift j
- IF s_i has item $X \to \alpha$ and $a \in$ follow(X) and $X \neq S'$ then action$[i,a]$ = reduce $X \to \alpha$
- IF s_i has item $S' \to S.$ then action$[i,\$]$ = accept
- Otherwise action$[i,a]$ = error

Full SLR algorithm

Let I = w$ be the initial input
Let j = 0
Let DFA state 1 have item S' → .S
Let stack = <dummy, 1>
repeat
  case action[top_state(stack), I[j]] of
    shift k: push <I[j++], k>
    reduce X → A:
      pop |A| pairs,
      push <X, goto[top_state(stack), X]>
    accept: halt normally
    error: halt and report error

Note that this algorithm only uses the DFA states and the input. stack symbols are never used

We still need stack symbols for later compiler stages

Widely used bottom-up parsing algorithms are based on a more powerful class of grammars, the LR(1) where
- Lookahead is built into items
- An item is a pair LR(0) item and lookahead
- More accurate than just using follow sets

Semantic Analysis

Lexical analysis
- Detects inputs with illegal tokens

Parsing
- Detects inputs with ill-formed parse trees

Semantic analysis
- Catches all remaining errors

Checks of many kinds ... code checks:
- All identifiers are declared
- Types
- Inheritance and relationships
- Reserved identifiers not misused
- Classes and methods defined only once
- others...

The requirements depend on the language

Scope

Matching identifier declarations with uses

let y: String ← "abc" in y+3
  y cannot be added

let y: Int in x+3 ← no definition of x

The scope of an identifier is the portion of a program in which the identifier is accessible

The same identifier may refer to different things in different parts of the program
- Different scopes for the same name can't overlap

An identifier can have restricted scope

Most programming languages today have static scope

- The scope of a variable depends only on the program text, not run-time behaviour

A few languages are dynamically scoped
Like LISP, SNOBOL

- Scope of a variable depends on the execution behaviour of the program

```
let x : Int <- 0 in
{ x
  let x int <- 1 in
    x;
  x; }
```

Same scope

A variable binds to the definition that is most closely enclosing it of the same name

In COOL...

class definitions:

- cannot be nested
- are globally visible

A class name can be used before it is defined

Attribute names are global within the class in which they are defined

Methods need not to be defined in the class in which it is used, but in some parent class

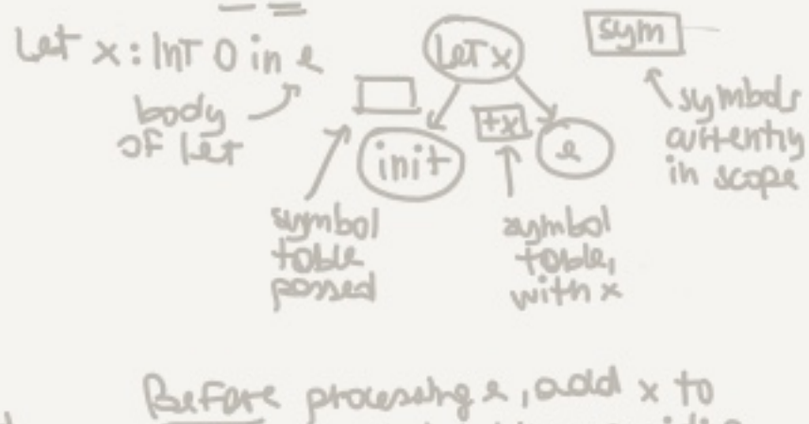
A method can be overridden

Symbol table

Much of semantic analysis can be seen as a recursive descent of an abstract syntax tree

Before: Process on AST node n (recursive)
Recurse: Process the children of n
After: Finish processing node n

When performing semantic analysis on a portion of the AST, we need to know which identifiers are defined



Before processing e, add x to symbol table overriding any existing definition

Recurse

After processing e, remove x from table and restore old definition of x

Symbol table tracks the current binding of identifiers

For a simple symbol table we can use a stack

- add_symbol(x), push x and associated info
- find_symbol(x), return first x occurrence, or NULL
- remove_symbol(), pop

When using a stack, declarations are perfectly nested

Consider f(x:Int, x Int){}
that is not legal; Functions introduce multiple names at once and the stack will not detect any error

We change stack operations slightly

- enter_scope(), start a new nested scope
- exit_scope(), exit current scope
- check_scope(x), true if x defined in current scope
- Find_symbol(x), find current x or null
- add_symbol(x), add x to the table

This is a stack of scopes!

Class names can be used before being defined

- Pass through code before and gather all class names
- Pass again and do the check

Semantic analysis requires multiple passes

Types

Notion varies from language to language

A set of values
A set of operations on those values

—=

Doesn't make sense to add pointer and integer

Does make sence to add int with int

Both have the same assembly implementation

A language's type system specifies which operations are valid for which types

Type checking ensures operations are used only with the correct types

—=

Statically typed, All or almost all checking of types is done as part of compilation

Dynamically typed, Almost all checking of types is done as part of program execution

Untyped, No type checking

Static typing
- Static checking catches many program errors at compile time
- Avoids overhead of runtime type checks

Dynamic typing
- Static types are restrictive
- Rapid prototyping difficult within a static type system

cool types are
- class names
- SELF_TYPE

The user declares types for identifiers

The compiler infers types for expressions

—=

Type checking is the process of checking a fully typed program

Type inference is the process of filling in missing types

Those terms are different, but they're usually used interchangeably

Type checking

We've seen two examples of formal notation for specifying parts of a compiler. RE's and CFG's

The formalism for type checking is logical rules and rules of inference

—=

- Inference rules have the form:
  IF Hypothesis is true, then Conclusion is true

- Type checking computes by reasoning
  IF E_1 and E_2 have certain types, then E_3 has certain type

- Rules of inference are a compact notation for "if-then" statements

Start with simple symbols and gradually add features

Building blocks
- Symbol ^ is "and"
- Symbol ⇒ is "if-then"
- x:T is "x has type T"

—=

IF e_1 has type int and e_2 has type int, then e_1+e_2 has type int

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

—=

The statement above is a special case of an inference rule

↓

hypothesis$_1$ ^ ... hypothesis$_n$ ⇒ conclusion

By tradition, inference rules are written $\frac{\vdash \text{hyp}_1 \quad \dots \quad \vdash \text{hyp}_n}{\text{conclusion}}$

called turnstiles, means "It is provable that..."

—=

$$\frac{i \text{ is integer literal}}{\vdash i : \text{Int}} \ [\text{Int}]$$

$$\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \ [\text{Add}]$$

A type system is sound if
- Whenever e:T
- Then e evaluates to a value of type T

We only want sound rules
- Some sound rules are better than others

$$\frac{\text{i is int}}{\vdash i : \text{Object}}$$ BAD RULE

—=

TO summarise

Type checking proves facts e:T
- Proof on the structure of the AST
- Proof has the same shape of the AST

$$\frac{\vdash 1 : \text{Int} \quad \vdash 2 : \text{Int}}{\vdash 1+2 : \text{Int}}$$ childs, root

In the type rule used for node e:
- Hypothesis are the proofs of types of e's subexpressions
- Conclusion of type e

Types are computed bottom-up pass over the AST

Type environments

(cool rules)

$$\frac{}{\vdash \text{false} : \text{Bool}}\ [\text{False}] \qquad \frac{s \text{ is a string}}{s : \text{String}}\ [\text{String}]$$

new T produces an object of type T

$$\frac{\vdash e : \text{Bool}}{\vdash !e : \text{Bool}}\ [\text{Not}]$$

$$\frac{}{\vdash \text{new } T : T}\ [\text{New}]$$

$$\frac{\vdash e_1 : \text{Bool} \quad \vdash e_2 : T}{\vdash \text{while } e_1 \text{ loop } e_2 \text{ pod} : \text{Object}}\ [\text{Loop}]$$

(condition, body)

—=

$$\frac{x \text{ is a variable}}{\vdash x : ?}\ [\text{Var}]$$ What is the type of x?

Put more information in the rule!

A type environment gives types to free variables
- Free variables are not defined within the expression
- A type environment is a function from variable names to types

Let O be a type environment

$O \vdash e : T$

under the assumption that variables have the types given by O, it is provable that expression e has type T

Type environment is added to previous rules

$$\frac{i \text{ is an integer literal}}{O \vdash i : \text{Int}}\ [\text{Int}] \qquad \frac{O \vdash e_1 : \text{Int} \quad O \vdash e_2 : \text{Int}}{O \vdash e_1 + O \vdash e_2 : \text{Int}}\ [\text{Add}]$$

$$\frac{O(x) = T}{O \vdash x : T}\ [\text{Var}]$$ ← If we want to know the type of x, we look it up in O

—=

$O[T/x]$ ← is the function O, modified modified at x to return T

$O[T/x](x) = T$

$O[T/x](y) = O(y)$

$$\frac{O[T/x] \vdash e_1 : T_1}{O \vdash \text{let } x : T_0 \text{ in } e_1 : T_1}\ [\text{Let-No-Init}]$$

↓

symbol table

—=

Type environments gives types to the free identifiers in the current scope

Type environment is passed down the AST Through the root towards the leaves

Types are computed up the AST from the leaves towards the root

Subtyping

Define a relation $\leq$ on classes

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

$$\frac{O(x) = T_0 \quad O \vdash e_1 : T_1 \quad T_1 \leq T_0}{O \vdash x \leftarrow e_1 : T_1} \text{ [Assign]}$$

—=

if e_0 then e_1 else e_2

We don't know which branch is taken. The best we can do is the smallest supertype (greater than type of e_1 or e_2)

lub(X,Y), the least upperbound of X and Y is Z if

- $X \leq Z \wedge Y \leq Z$ (Z is an upper bound)
- $X \leq Z' \wedge Y \leq Z' \Rightarrow Z \leq Z'$ (Z is least among upper bounds)

The least upper bound is the least common ancestor in the inheritance tree

$$\frac{O \vdash e_0 : Bool \quad O \vdash e_1 : T_1 \quad O \vdash e_2 : T_2}{O \vdash \text{IF } e_0 \text{ then } e_1 \text{ else } e_2 \text{ fi} : lub(T_1, T_2)} \text{ [IF-Then-Else]}$$

A method 'foo' and an object 'foo' can coexist in the same namespaces

In the type rules, this is reflected by a separate mapping 'M' for method signatures

$M(C,F) = (T_1 \ldots T_n, T_{n+1})$ ← result

Means in class C there's a method F

$F(x_1 : T_1, \ldots, x_n : T_n) : T_{n+1}$

—=

Add method environment to all rules

$$\frac{O, M \vdash e_1 : Int \quad O, M \vdash e_2 : Int}{O, M \vdash e_1 + e_2 : Int} \text{ [Add]}$$

—=

The full type environment for COOL

- A mapping O giving types to object id's
- A mapping M giving types to methods
- The current class C

—=

The full form of a sentence in the COOL sentence logic is

$O, M, C \vdash e : T$

Implementing type checking

COOL type checking can be implemented in a single traversal over the AST

Top-down phase, type environment is passed down the tree

bottom-up phase, types are passed back up

$$\frac{O, M, C \vdash e_1 : Int \quad O, M, C \vdash e_2 : Int}{O, M, C \vdash e_1 + e_2 : Int} \text{ [Add]}$$

```
TypeCheck(Environment, e1 + e2) = {
                O, M, C
  T1 = TypeCheck(Environment, e1);
  T2 = TypeCheck(Environment, e2);
  Check T1 == T2 == Int;
  return Int;} // check succeeds
```

—=

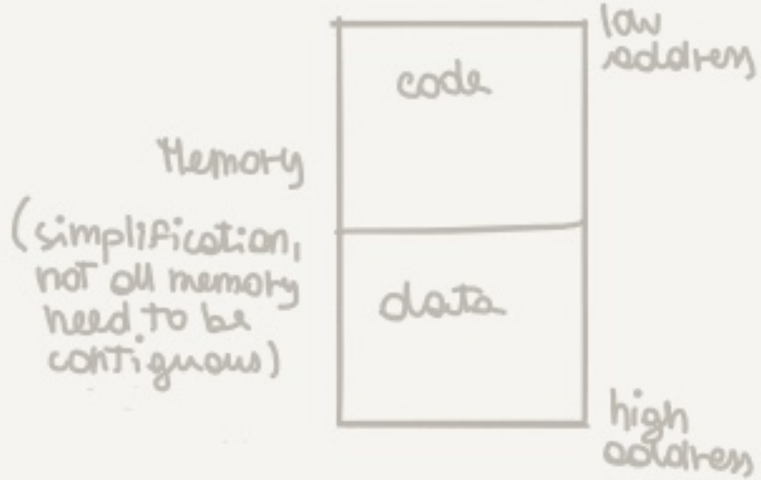
$$\frac{O, M, C \vdash e_0 : T_0 \quad O[T/x], M, C \vdash e_1 : T_1 \quad T_0 \leq T}{O, M, C \vdash \text{let } x:T \leftarrow e_0 \text{ in } e_1 : T_1} \text{ [Let-Init]}$$

```
TypeCheck(Environment, let x:T ← e0
                                in e1) = {
  T0 = TypeCheck
          (Environment, e0);
  T1 = TypeCheck(Environment.
          add(x:T), e1);
  Check subtype(T0, T1);
  return T1; }
```

Runtime organization

Before discussing code generation, we need to understand what we're trying to generate

When a program is invoked
- The OS allocates space for the program
- The code is loaded into that space
- The OS jumps to the entry point (ie main)



Compiler is responsible for generating code, but also has to decide what the layout of the data is going to be and then generate code that correctly manipulates that data

Activation

Complications in code generation come from trying to be fast as well as correct

Assumptions
- Execution is sequential (...in a well defined order)
- After a procedure is called, control returns to the point immediately after

We'll call an invocation of procedure P, an activation of P

The lifetime of P is all the steps in P and all the steps of the procedures that P calls

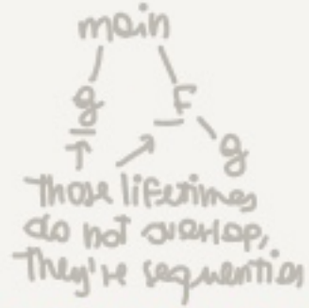
The lifetime of a variable x is the portion of the execution in which x is defined

—=

When P calls Q, Q returns before P

Activation lifetimes can be depicted as a tree

```
f():Int{g()};
main():Int
{{g();f();}};
...
```



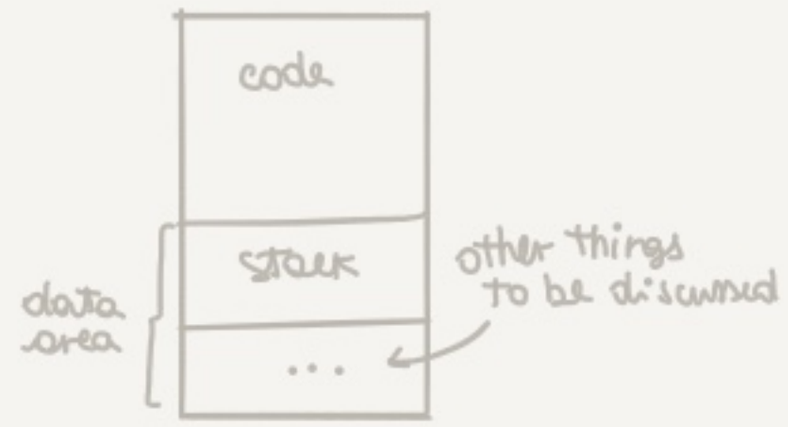
Those lifetimes do not overlap, they're sequential

Since activations are properly nested, we can use a stack to implement the currently active activations.

For the above example

The stack will at any time, contain the active procedures

stack=
push main
push g
pop g
push f
...
push g
pop g
pop f
pop main



Activation records (also called frames)

An activation record is all the information that's needed to manage the execution of one procedure activation

If procedure F calls procedure G, then G's activation record contains a mix of info about F and G

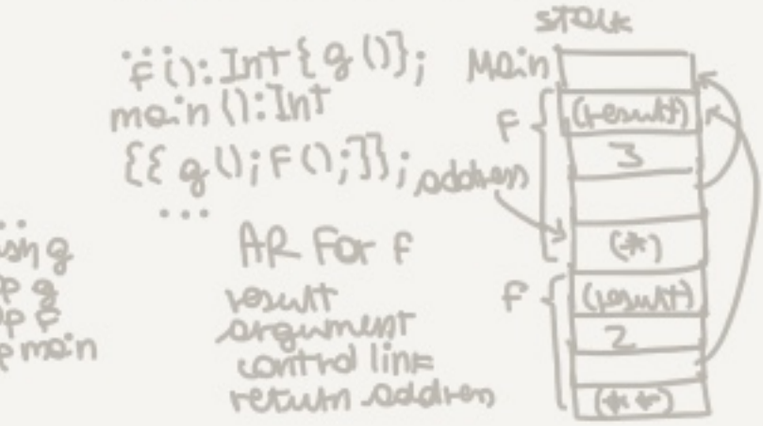
—=

F calls G

F is suspended until G completes, at which point F resumes

G's activation record contains information needed to complete the execution of G and resume F

```
f():Int{g()};
main():Int
{{g();f();}};
...
```



Compilers hold as much of the frame as possible in registers
- Especially method result and arguments

The compiler has to determine at compile-time the layout of activation records and generate code that correctly accesses locations in the activation records

AR layout and code generator must be designed together

Globals and heaps

All references to a global variable point to the same object
- We can't store a global in an activation record

Globals are assigned a fixed address once
- Said to be "statically allocated" (at compile time)

A value that outlives the procedure that creates it cannot be kept in the AR

```
method foo() { return new Bar }
```

'Bar' must survive the deallocation of Foo's AR

Language with dynamically allocated data use a heap to store dynamic data

Different kinds of data that a language implementation has to deal with
- The code area: fixed size and read only (in many languages)
- The static area contains data with fixed addresses (global data); fixed size, may be readable or writable
- Stack contains an AR for each currently active procedure; each AR has fixed size and contains all the local information needed to execute a particular activation
- Heap contains all other data; in C, heap is managed by malloc and free

Both heap and stack can grow!

We don't want them to grow into each other

Solution, start heap and stack at opposite ends of memory and let them grow into each other

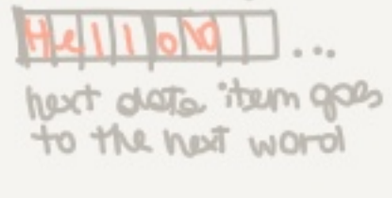
If these pointers ever become equal, the program is out of memory and the runtime system will take some sort of action

Alignment

Most modern machines are 32 or 64 bits
- 4 or 8 bytes in a word
- Machines are either byte or word addressable

Data is word aligned if it begins at a word boundary

Most machines have some alignment restrictions
- Or performance penalties for poor alignment



Stack machine

Only storage is a stack

$r = F(a_1 \dots a_n)$:
- pops n operands
- Computes F using the operands
- push the result r on the stack

↓

Contents prior to r are preserved

—=

Addresses are not explicitly stated
- Always refer to the top of the stack

More compact programs
(...than register machine)

Java bytecode uses stack evaluation

—=

n-register stack machine
- Keep to n stack locations in registers

1-register stack machine
- Also called accumulator

↙

Take ADD, in a stack machine
- Pop two arguments
- Add them
- Put result back

In a 1-register stack machine
→ acc ← acc + top_of_stack

accumulator register

—=

Consider an expression $op(e_1 \dots e_n)$
- $e_1 \dots e_n$ are subexpressions

For each e_i $(0 < i < n)$
- compute e_i ← result in acc
- push result

e_n → just evaluate (don't push result)

Pop n-1 values from the stack, compute op

Store result in accumulator

After evaluating e, the accumulator holds the value of e and the stack is unchanged

—=

3 + (7 + 5)

code	acc	stack
acc ← 3	3	<init>
push acc	3	3, <init>
acc ← 7	7	3, <init>
push acc	7	7, 3, <init>
acc ← 5	5	7, 3, <init>
acc ← acc + top_of_stack	12	7, 3 <init>
pop	12	3, <init>
acc ← acc + top_of_stack	15	3, <init>
pop	15	<init>

Code generation

We focus on generating code for a stack machine with accumulator
(simplest code gen strategy)

We use the MIPS processor to run the resulting code on a real machine

Simulate stack machine instructions using MIPS (simulator)

—=

The accumulator is kept in MIPS register $a0

The stack is kept in memory
- Grows towards lower addresses

The address of the next stack location is kept in MIPS register $sp (stack pointer)
- Top of the stack is $sp + 4

MIPS architecture
- Prototypical RISC
- Most operations use registers for operands and results

—=

lw reg1 offset(reg2)
- Load 32 bit word from reg2 + offset into reg1

add reg1 reg2 reg3; reg1 ← reg2 + reg3

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

addiu reg1 reg2 imm; reg1 ← reg2 + imm

li reg imm; reg ← imm

—=

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

temporary register

Code generation I

language with integers and operations

P → D; P | D

D → def id (ARGS) = E;

ARGS → id, ARGS | id

E → int | id | if $E_1 = E_2$ then E_3 else E_4

| $E_1 + E_2$ | $E_1 - E_2$ | id ($E_1, \dots, E_n$)

- The first function definition f is the entry point

—=

For each expression e we generate MIPS code that

- Compute e and store result in $a0
- Preserve $sp and stack contents

cgen(e) generates code for e

—=

To evaluate an expression, which is just a constant, load that constant into the accumulator

cgen(i) = li $a0 ; compile time runtime

```
cgen(e1+e2) =
    cgen(e1)
    sw $a0 0($sp)      push e1 on the stack
    addiu $sp $sp-4    bump the stack pointer
    cgen(e2)
    lw $t1 4($sp)      load e1 in t1 from stack
    add $a0 $t1 $a0
    addiu $sp $sp 4    pop stack
```

Assembly is printed into a file

Put the result of e_1 directly in t_1

```
cgen(e1+e2) =          WRONG
    cgen(e1)           Values are overridden in t1
    move $t1 $a0
    cgen(e2)
    add $a0 $t1 $a0
```

—=

The code for + is a template with holes for evaluating e_1 and e_2

Stack machine code gen is recursive

—=

```
beq reg1 reg2 label
  - Branch if reg1 = reg2
b label
  - Jump to label
```

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

Code generation II

Code for function calls and function definitions depends on the layout of the AR

A very simple AR suffices for this language

- The result is always in the accumulator
  No need to store result in AR

- AR holds actual parameters
  For $f(x_1, \dots, x_n)$ push $x_n, \dots, x_1$
  These are the only variables in this language

—=

Stack pointer is preserved across function calls
- No need for a control link in AR

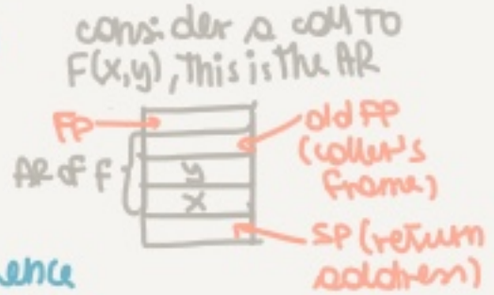
We need return address in AR

A pointer to the current activation is useful
- This pointer is in register $Fp (frame pointer)

—=

So we need an AR with
- Frame pointer
- Actual parameters
- return address

—=



The calling sequence is the sequence of instructions to set up a function invocation

—=

jal label; jump to label and save address of next instruction in $ra (return address)

cgen(f(e1...en)) =

caller side:
- sw $fp 0($sp) — save fp in stack (start building AR)
- addiu $sp $sp -4
- cgen(en)
- sw $a0 0($sp) — push the result
- addiu $sp $sp -4
- ... for all arguments
- cgen(e1)
- sw $a0 0($sp)
- addiu $sp $sp -4
- jal f_entry

The AR so far is 4*n+4 bytes; frame pointer

jr reg; jump to address in reg

cgen(def(x1...xn) = e) =

f_entry:
- move $fp $sp
- sw $ra 0($sp)
- addiu $sp $sp -4

At this point, the AR is completely set up. Next we generate code for the function body

- cgen(e)
- lw $ra 4($sp)
- addiu $sp $sp 4*n+8 — with n arguments + return address + old fp
- lw $fp 0($sp)
- jr $ra

Variable references are the last construct

The variables of a function are just its parameters
- They're all in the AR

But the stack grows when intermediate results are saved. The variables are not at a fixed offset from $sp

Solution, we use the frame pointer
- Always points to the return address of the stack
- Since it doesn't move, it can be used to find the variables

Let xi be the ith (i=1...n) argument of the function

cgen(x) = lw $a0 4*i($fp)

Production compilers do different things
- Emphasis on keeping values on registers
- Intermediate results are laid out in the AR, not pushed and popped from stack

Temporaries

Keep temporaries in the AR

IDEA → ... rather than pushing and popping

The code generation must assign a fixed location in the AR for each temporary

Let NT(e) be the # of temporaries needed to evaluate e

NT(e1 + e2)
- Needs at least as many temporaries as NT(e1)
- Needs at least as many temporaries as NT(e2) + 1

NT(e1+e2) = max(NT(e1), 1+NT(e2)) — hold the result of e1

NT(e2+e2) = same

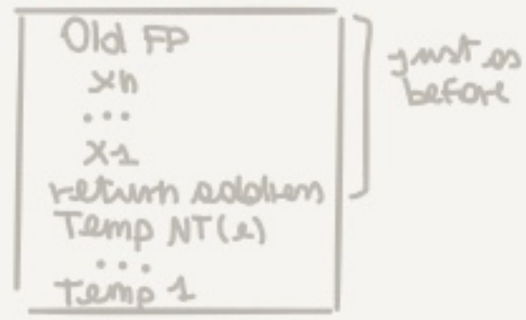
NT(if e1=e2 then e3 else e4) = max(NT(e1), 1+NT(e2), NT(e3), NT(e4))

NT(id(e1,...,en)) = max(NT(e1),...,NT(en)) — no additional ones for result because result is saved in the AR

NT(int) = 0

NT(id) = 0 — variable reference

For a function definition f(x1...xn) = e the AR is 2+n+NT(e) elements
- Return address
- Frame pointer
- n arguments
- NT(e) locations for intermediate results



Code generation must know how many temporaries are in use at each point

Add a new argument to code generation - the position of the next available temporary

cgen(e1 + e2, nt) =
- cgen(e1, nt)
- sw $a0 nt($fp) — we have already saved a temporary value; use next available temporary
- cgen(e2, nt+4)
- lw $t1 nt($fp)
- add $a0 $t1 $a0

Object layout

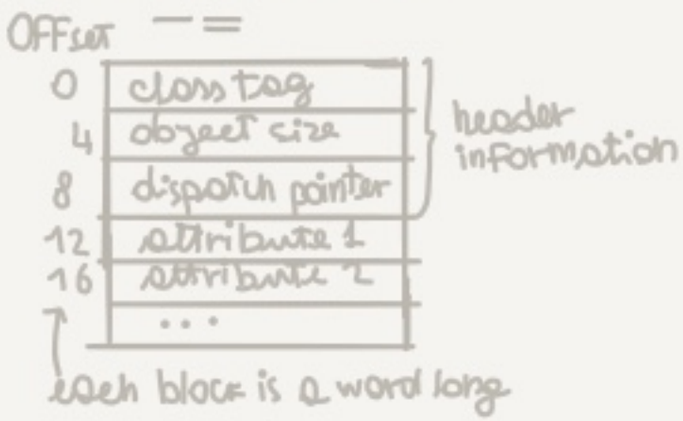
```
class A {
  a: Int ← 0;
  d: Int ← 1;
  f(): Int {a ← a + d};
};

class C inherits A {
  c: Int ← 3;
  h(): Int {a ← a * c};
};

class B inherits A {
  b: Int ← 2;
  f(): Int {a};
  g(): Int {a ← a - b};
};
```

— =

Objects are laid out in contiguous memory

Each attribute is stored at a fixed offset in the object
- The attribute is in the same place for every object of that class

When the method is invoked, the object is 'self' and the fields are the object's attributes (like 'this' in java)



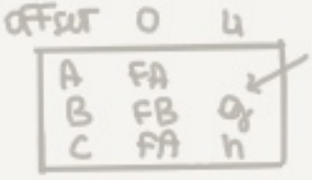
The class tag is an integer
- Identifies the class of the object

The object size is an integer
- size of the object in words

Dispatch pointer is a pointer to a table of methods

Attributes in subsequent slots

— =



F B is method f overridden in B
The method f is always found at the same offset

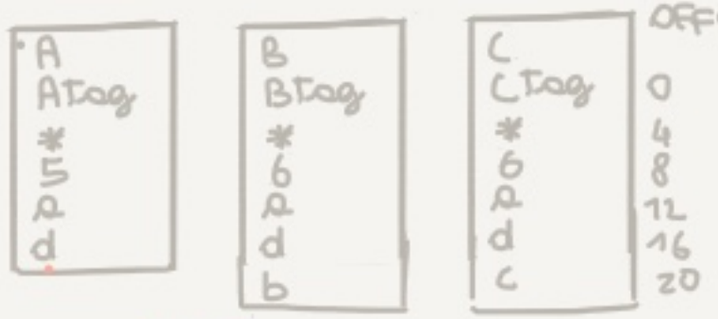
The dispatch pointer in the object header points to the table of method entries for that class

We use a separate table because that way it can be shared between all objects of that class

— =

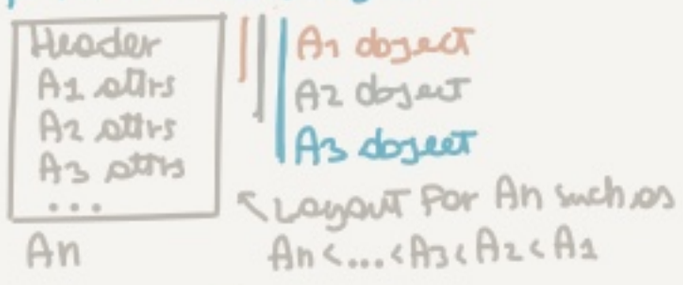
Given a layout for a class A, a layout for a subclass B can be defined by extending the layout of A
- Layout of A is unchanged (OFFset for A's attributes is the same for all subclasses)
- B is an extension

To implement a dynamic dispatch e.f()
- evaluate e, giving an object x
- call D[Of]
  - D is the dispatch table for x
  - In the call, self is bound to x



— =

IF we have a chain of inheritance relationships each prefix is a valid object



— =

A dispatch table indexes a class' methods
- An array of method entry points
- A method F lives at fixed offset in a dispatch table for a class and all of its subclasses

Operational semantics

We must specify for every cool expression what happens when it's evaluated

- The "meaning" of an expression

— =

Quick recap

In defining a programming language
- The tokens → lexical analysis
- the grammar → syntactic analysis
- The typing rules → semantic analysis
- The evaluation rules → code generation and optimisation

— =

We have specified evaluation rules indirectly
- Compilation of COOL to a stack machine
- evaluation rules of the stack machine

But assembly language descriptions of language implementations have a lot of irrelevant detail
- Whether to use a stack machine
- Which way the stack grows
- How ints are represented
- ...

We need a complete description instead
- But not overly restrictive specification

— =

There are many ways to specify semantics, we're going to use operational semantics
- Describes a program evaluation via execution rules
- Most useful for specifying implementations

Other ways to specify semantics are Denotational sem and Axiomatical sem

We're going to be using logical rules of inference, as in type checking

In type checking they had the form
context ⊢ e : C
In a given context expression 'e' has type C

For evaluation
context ⊢ e : v
In the given context expression e evaluates to v

$$\frac{\text{context} \vdash e_1 : 5 \qquad \text{context} \vdash e_2 : 7}{\text{context} \vdash e_1 + e_2 : 12}$$

— =

Consider the evaluation of y ← x + 1
To evaluate it have to know
- environment, where in memory a variable is stored
- store, what is in the memory

— =

A variable environment maps variables to locations

E = [a : l1, b : l2] list of variable and location pairs

↑ Keep track of which variables are in scope

— =

Store maps locations to values

S = [l1 → 5, l2 → 7]

S' = S[12/l1] takes S and updates the value at l1 to be 12 by defining a new store S'

Stores are just functions

X = (a1 = l1 ..., an = ln) is a COOL object where
- X is the class of the object
- ai are the attributes (including inherited)
- li is the location of the value of ai

— =

special cases (classes without attributes)
Int(5) ; Bool(true) ; String(4, "cool") ← length

There is a special value void of type Object
- No operations can be performed on it
- Except for the test is void
- Concrete implementations might use NULL to represent void

— =

The evaluation judgement is
$$S_0, E, S \vdash e : v, S'$$
- S_0 is the self object
- E is the environment
- S is the store

} context

- In that context an expression 'e' will
- ... produce a value 'v'
- ∴ produce a modified store S' only IF e evaluates

— =

Result of an evaluation is a value in a new store

The environment E and the self object don't change during the evaluation

NOTE: Contents of the self might change, but the layout of the object and its location doesn't change

Intermediate code

It is a language between the source language, and the target language

- More details than source ← eg you want to optimise register usage. A source lang has no notion of registers so you need interm. lang
- Fewer details than target

↗ Above a particular instruction set of a machine. So it's easier to retarget to lots of different kinds of machines

source lang → IL → target lang

Some compilers have more than one interm. lang

—=

Intermediate language = high-level assembly
- Uses register names, an unlimited # (eg jump, labels)
- Uses control structures like assembly
- Uses opcodes but some are higher level

Each instruction is of the form

$x := y \ op \ z$ — y,z are either registers or constants
$x := op \ z$

Really common form! → — Three address code, it has three addresses in it:
- at most two arguments
- a destination

The expression $x + y * z$ is

$t_1 := y * z$ ← each subexpression has a "name"
$t_2 := x + t_1$

—=

igen(e,t); code to compute 'e' in register 't'

```
igen(e1+e2, t) =
    igen(e1, t1)
    igen(e2, t2)
    t := t1 + t2
```

Unlimited registers leads to very simple code generation of intermediate code

Optimisation

Lexical Analysis → Parsing → Semantic Analysis ... → Optimisation → Code generation

Most complex part of the compiler

—=

We optimise intermediate language
- Machine independant
- Exposes optimisation opportunities

—=

```
P → S P | S
S → id := id op id | id := op id |
    id := id | push id | id := pop |
    if id relop id goto L | L: | jump L
```

A basic block is a maximal sequence of instructions
- No label, except first instruction
- No jumps, except last instruction

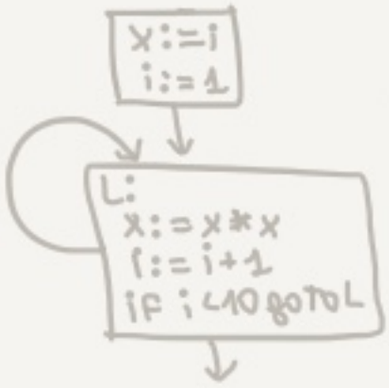
Idea, execution is guaranteed to proceed from the first to the last block statement

```
1 L:
2   t := 2 * x
3   w := t + x
4   if w > 0 goto L
```

(3) executes only after (2)
- (3) can be $w := 3 * x$
- We could eliminate (2) after that optimisation if 't' is not used elsewhere

=—

A control-flow graph is a directed graph of basic blocks. There's an edge from node A to B if the execution can pass from last A's instruction to B's first instruction



- The body of a method can be represented as a control flow graph
- There is an initial node
- All return nodes have no edges coming out

—=

Optimisation improves resource utilisation
- execution time
- code size
- memory usage and so on

... without altering what the program produces

—=

(1) Local optimisation
applies to basic blocks

(2) Global optimisation
applies to control-flow graphs

(3) Inter-procedural optimisation
applies across method boundaries

Most compilers do (1), many do (2), few do (3)

—=

- Some optimisations are hard to implement
- Some are costly in compilation time
- ... Or have low-payoff
- Many fancy optimisations have all three

Goal: Maximum benefit for minimum cost!

Local optimisation

Optimise one basic block

x: Int enables other optimisations

can be deleted
- x := x+0
- x := x*1 → can be simplified x := x

x := x*8 → x := x<<3
bit shifts are usually faster than *

x := x*0 → x := 0
y := y**2 → y := y*y
** is not a built-in machine instruction

Algebraic simplifications

Compute operations on constants at compile time — Constant folding — REALLY COMMON!

x := 2+2 ⇒ x := 4

IF 2<0 jump L //deleted

Eliminate unreachable basic blocks

- Makes the code smaller

It is quite common

eg libraries. Not used methods can be removed to make the code smaller

debugging. Not running a piece of code while debugging

result of other optimisations. They may create unreachable blocks

Some optimisations are simpler to express if each register occur only once on the left-hand side of an assignment

Single assignment form

x := z+y ⇒ b := z+y
a := x → a := b
x := 2*x → x := 2*b

eg
- if basic block in single a.. form then all assignments with the same right hand side compute the same value ↙ common subexpression optimization
- If w := x appears in a block, then all subsequent w's can be replaced with x ↖ copy propagation

if w := rhs and w is never used then the statement can be deleted

Typically one optimisation enables another

— repeat optimisation until no improvement is possible

a := x**x
b := 3
c := x
d := c*c
e := b+2
f := a+d
g := a*f

algebraic opt. →

a := x*x
b := 3
c := x
d := c*c
e := b<<1
f := a+d
g := a*f

⇒ copy propag.

a := x*x
b := 3
c := x
d := x*x
e := 3<<1
f := a+d
g := a*f

⇒ constant folding

a := x*x
b := 3
c := x
d := x*x
e := 6
f := a+d
g := a*f

⇒ common subexp.

a := x*x
b := 3
c := x
d := a
e := 6
f := a+d
g := a*f

⇒ copy propag.

a := x*x
b := 3
c := x
d := a
e := 6
f := a+a
g := 6*f

⇒ dead code elim.

a := x*x
f := a+a
g := 6*f

↳ There are other optimisations that are harder to find

a := x*x
f := 2*a ⇒ g := 12*a
g := 12*a

a := x*x
g := 12*a

Peephole optimisation

Optimise assembly code directly

The "peephole" is a short sequence of

Replace the sequence with another equivalent one (but faster)

Peephole optimisations are written as replacement rules $i_1,\dots,i_n \to j_1,\dots,j_m$

move $a $b, move $b $a → move $a $b
addiu $a $a i, addiu $a $a j →
addiu $a $a i+j

Many optimisations from local optimisations can be used as peephole optimisations

Just like locals, they must be applied repeatedly

Code produced by "optimisers" is not optimal, it is just improved

Dataflow analysis

Local optimisations can be extended to an entire control flow graph

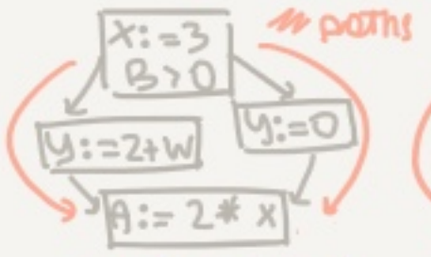
—=

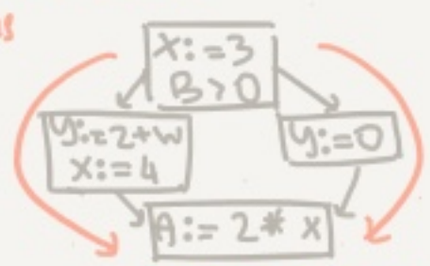
Global dataflow analysis is a technique to solve problems with these characteristics

- The optimisation depends on knowing a property x at a particular point in the program
- Proving x at any point requires knowledge of the entire program
- It is ok to be conservative. Either x is true or we don't know (and don't do the optimisation)

Constant propagation

To replace a use of x with a constant k if
- on every path to the use of x, the last assignment to x is x := k





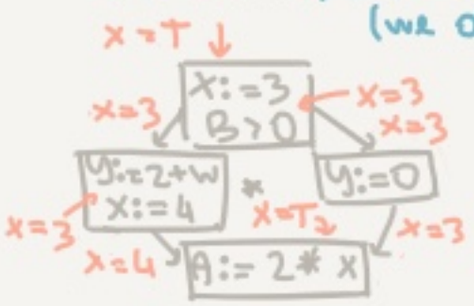
—=

For statements with x, we associate

T "top"; x is not a constant (we can't figure out if x is a constant)

C ; x is a constant

⊥ "bottom"; this statement never executed (we don't know if it executes)



*we don't know which path we're coming from

The analysis of a complicated program can be expressed as a combination of simple rules relating to the change of information between adjacent statements

C(x,s,in) = value of x before s
C(x,s,out) = value of x after s

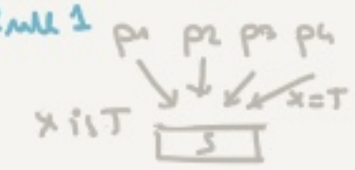
"C stands for constant information"

A transfer function transfers information from one statement to another

Every statement s has a set of predecessors (p1...pn)

—=

Rule 1

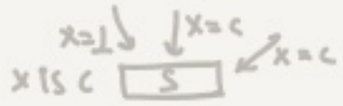


if C(pi,x,out)=T for any i, then C(x,s,in)=T

Rule 2

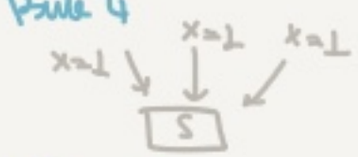
if C(pi,x,out)=c & C(pj,x,out)=d & d <> c not equal
then C(s,x,in)=T

Rule 3



if C(pi,x,out)=c or ⊥ for all i
then C(s,x,in)=c

Rule 4



if C(pi,x,out)=⊥ for all i
then C(s,x,in)=⊥

Rules 1-4 are for out-in statements
next we see for in-out

Rule 5

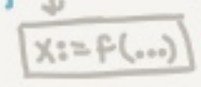
C(x:=c,x,out)=⊥
if C(s,x,in)=⊥

Rule 6

C(x:=c,x,out)=c
if c is a constant

There may be a conflict between rule 5 and 6. So we say that rule 5 has higher priority than 6

Rule 7



rule 5 has precedence over rule 7

Rule 8

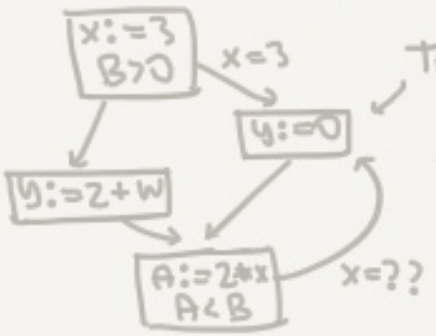
C(y:=...,x,out)=
C(y:=...,x,in)
if x<>y

—=

Algorithm

1. For every entry s to the program, set C(s,x,in)=T
2. Set C(s,x,in)=C(s,x,out)=⊥ everywhere else
3. Repeat until all points satisfy 1-8
   - Pick s not satisfying 1-8 and update using appropriate rule

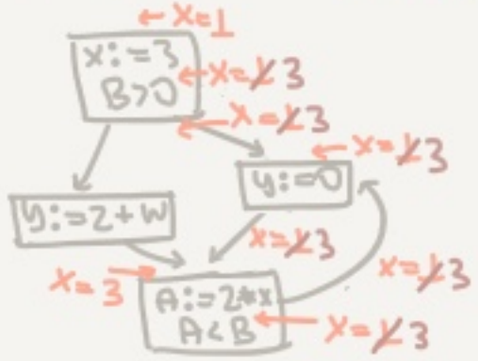
Analysis of loops



To compute whether x is constant we check the predecessor.
But A:=2*x depends on its predecessors including y:=0. They depend on each other

Break the cycle by starting with an initial guess ⊥.
This allows us to make progress

By following the algorithm previously described (which also assigns initial guesses):



Orderings

⊥, c, T are called abstract values

We can simplify things by ordering the abstract values

⊥ < c < T

We can now define lub (least upper bound) operation. lub takes the smallest element that is bigger than anything in the least upper bound

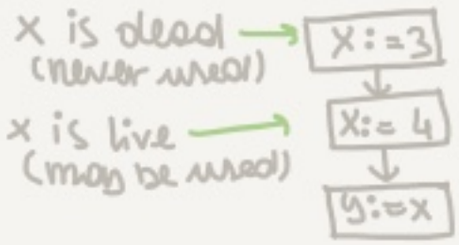
lub(⊥,⊥) = ⊥
lub(T,⊥) = T
lub(1,2) = T
↑ Then we're uncomparable (both c)

Rules 1-4 can be written as
C(s,x,in) = lub { C(p,x,out) | p predecessor of P }

lub explains why the algorithm terminates
- values start at ⊥ and only increase
- ⊥ can change to a constant, and a constant to T
- Thus C(s,x,_) can change at most twice

Constant propagation then is linear in program size
steps = C(...) values computed * 2
= # of program statements * 4

Liveness analysis

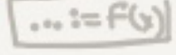


A variable is live at statement s if
- There exist a statement s_i that uses x
- There is a path from s to s_i
- The path have no assignments to x

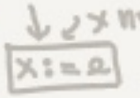
A statement x:=... is dead code if x is dead after the assignment (not live)
- That statement can be deleted

Rule 1 

L(p,x,out) = ∨{L(s,x,in) | s successor of p}

Rule 2 

L(s,x,in) = True if s refers to x on the rhs

Rule 3 

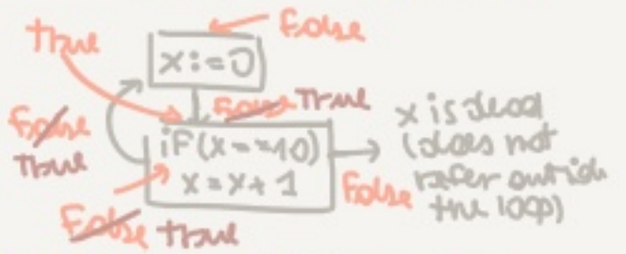
L(x:=a,x,in) = false if a does not refer to x

Rule 4 

L(s,x,in) = L(s,x,out) if s does not refer to x

Algorithm

1. Let all L(...) = false initially
2. Repeat until all statements s satisfy rules 1-4
- Pick s where one of 1-4 does not and update using appropriate rule



Each value can only increment (ordering false < true)
Since it can only change once, termination is guaranteed

Constant propagation is forward analysis
- Information pushed from inputs to outputs

Liveness is backward analysis
- From outputs back towards inputs

Register allocation

Intermediate code uses unlimited temporaries
- Simplifies code gen and optimisation
- But translation to assembly is harder

Problem:
Rewrite intermediate code to use no more temporaries than there are registers

Solution: Assign multiple temporaries to one register (without changing program's behaviour)

```
a := c+d      r1 := r2+r3
e := a+b      r1 := r1+r4
f := e-1      r1 := r1-1
```

- Assume a and e are dead after their use
- Allocate a, e and f all to one register

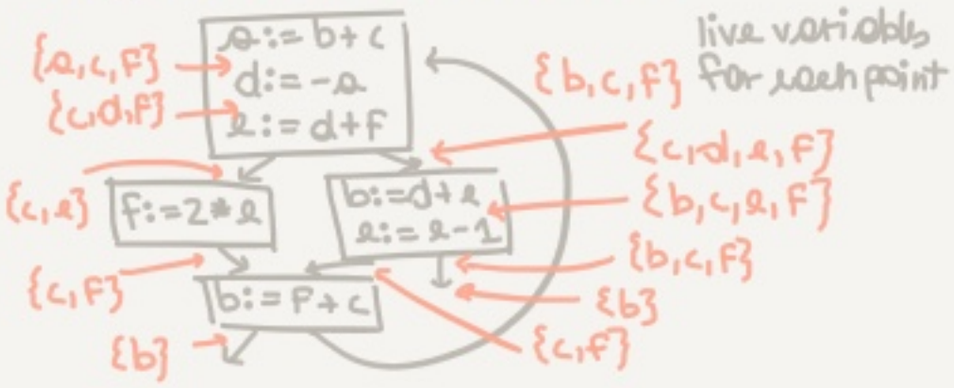
—=

Register allocation in the original FORTRAN project was a bottleneck in the quality of the compiler

The register allocation scheme based on graph coloring solved the issue years later

—=

Temporaries t_1 and t_2 can share a register if they are not live at the same time



Construct an undirected graph with a node for each temporary and an edge between if the temporaries are live at the same time at some point in the program

This is called register interference graph (RIG)
- Two temporaries can be allocated to the same register, if there is no edge between

RIG for our example

the RIG gives a global picture of the register requirements

After RIG, the register allocation algorithm is architecture independent

Graph coloring

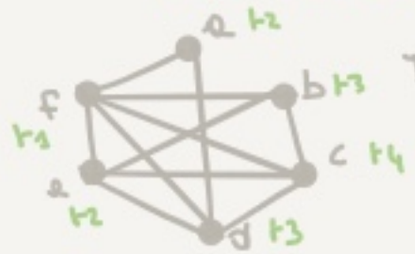
Assign colors to nodes such that nodes such that nodes connected by an edge have different color

A graph is k-colorable if it has a coloring with k colors

—=

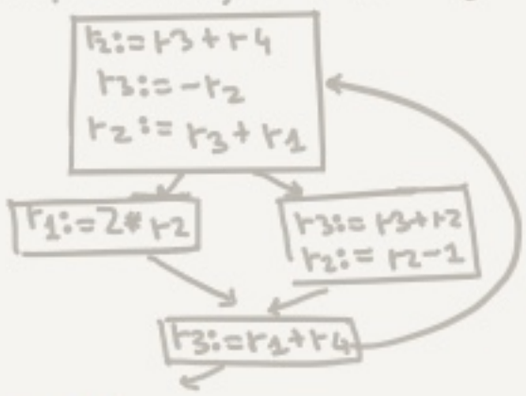
Let k = number of registers

If the RIG is k colorable, there is a register assignment that uses no more than k registers



The graph is 4-colorable

Control flow graph for RIG.
- Replace temporaries wh registers



—=

Graph coloring is NP-hard
- compilers usually use heuristics

Observation: DIVIDE AND CONQUER
- pick a node wh fewer than k neighbours in RIG
- Eliminate t and its edges in RIG
- If result is k-colorable, so is the original graph

Let $c_1 \dots c_n$ be the colors assigned to to t neighbours, since $n < k$ we can pick some color for t that is different from those of its neighbours

Algor. —=
- Pick node t wh fewer than k neighbours
- Put t on the stack and remove it from RIG
- Repeat until graph is empty
- start with last node added
- At each step, pick a color from those assigned to already colored neighbours

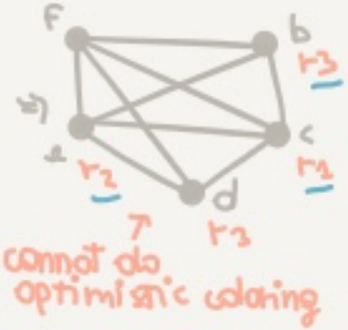
Spilling

IF all nodes of the RIG have k or more neighbours. Then there are not enough registers and some values are spilled in memory

remove a since has <k edges

No other node has <k
Pick a node for spilling, pick F

← can continue with usual algorithm

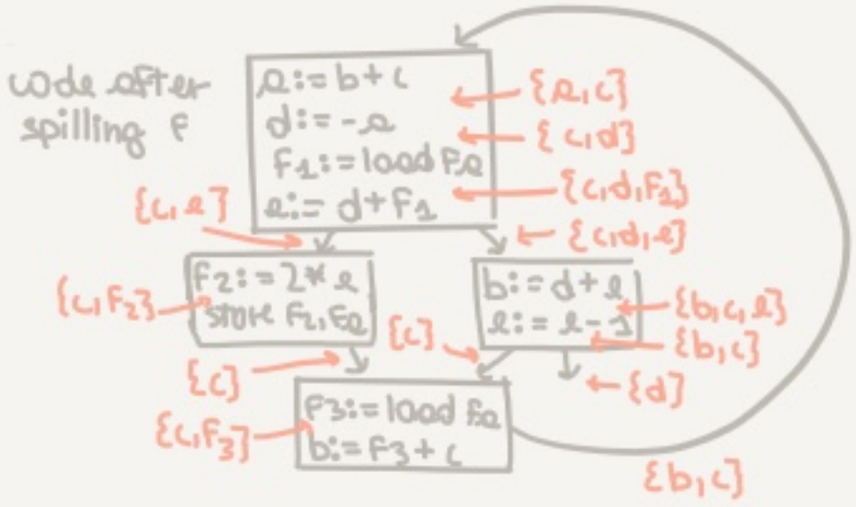


cannot do optimistic coloring

We must assign a color for F.
It could be that among among F's neighbours we use less than 3 (which is k) colors.
This is called optimistic coloring

IF optimistic coloring fails we spill F
- Allocate a memory location for F
  - Typically in the current stack frame
  - Call this Fa

Before each operation that reads F insert
F := load Fa
After each operation that writes F
store F, Fa



Fi is live only
- Between Fi = load Fa and the next instruction
- Between a store Fi, Fa and the preceding instr.

Spilling reduces live range of F
- Fewer RIG neighbours

This new RIG is 3-colorable!

= =

Additional spills may be required before a coloring is found

Some spilling choices lead to better code but any choice is correct
- spill temporary with most conflicts
- spill temporaries with few uses
- avoid spilling inner loops

possible heuristics

Managing caches

Power usage limits size and speed of registers and caches

The cost of a cache miss is really high and typically requires more than one level of cache to bridge a fast processor with large main memory

= =

Compilers are really good at managing registers but not so good at managing caches
- Up to the programmer to make good use of it

Consider the loop

```
for (j := 1; j < 10; j++)
  for (i := 1; i < 1000000; i++)
    a[i] *= b[i]
```

Every iteration will be a cache miss

a[1] b[1] a[2] b[2] ...

```
for (i := 1; i < 1000000; i++)
  for (j := 1; j < 10; j++)
    a[i] *= b[i]
```

- Computes the same thing
- But with much better cache behaviour
- More than 10x faster

A compiler can perform this **loop interchange** optimization

NOT all compilers implement this because it might be hard to decide whether you can do it
Usually it's up to the programmer

Garbage collection

C and C++ have manual memory management
Many storage bugs
- Forget to free a used memory
- dereferencing dangling pointers
- overriding parts of a data structure
- others...

Those bugs are hard to find!

—=

This problem has been studied since 1950s for LISP

Become mainstream with Java

—=

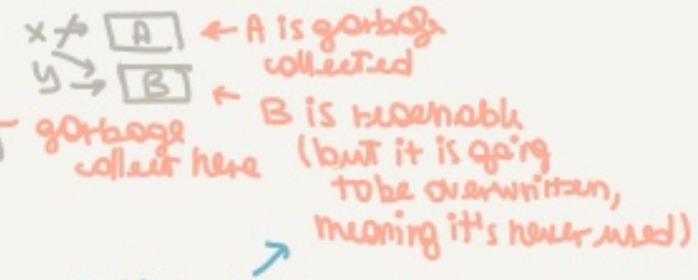
A program can only use the objects that it can find

An object x is reachable if
- a register contains a pointer to x
- another reachable object y contains a pointer to x

Reachable objects can be found by starting from registers and following all the pointers

Unreachable objects can never be used and are called garbage

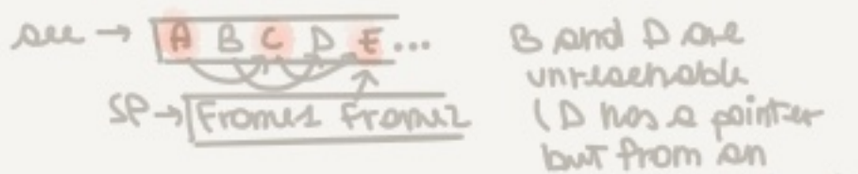
```
x ← new A;
y ← new B;
x ← y;
if alwaysTrue()
 then x ← new A
 else x.foo() fi
```



reachability is an approximation, just because it is reachable it doesn't guarantee that it will be used

Cool uses an accumulator
- points to an object which may point to others

... and a stack pointer
- each stack frame contains pointers
  eg method parameters
- each stack frame also contains non-pointers
  eg return address



B and D are unreachable (D has a pointer but from an unreachable object)

—=

Each garbage collection scheme has these steps

1 Allocate space for new objects

2 When space runs out
- compute what objects might be reused
- Free space used by objects not found in (a)

Some strategies perform garbage collection before the space runs out

Java arrays

Assume B < A

```
B[] b = new B[10];   b → [B's]
A[] a = b;           a ↗
a[0] = new A();
b[0].aMethodNotDeclaredInA();
```
Runtime error!

Having multiple aliases to updatable locations with different types is unsound

This problem is common among programming languages

Standard solution: B[] < A[] if B=A
↑ subtyping rule for arrays

Java checks each array assignment at runtime

Adds overheads on array computations
primitive type arrays are not affected

Java exceptions

In a section of your code, you find an error
- out of memory
- a list is supposed to be sorted but is not
- etc.

Add a new type (class) of exceptions

Add new forms
try {...} catch (x) {...}
throw exception

T(v) = exception thrown with value v

$$\frac{E \vdash e_1 : v_1}{E \vdash \text{try}\{e_1\}\ \text{catch}(x)\{e_2\} : v_1}$$

$$\frac{E \vdash e_2 : T(v_1) \qquad E[x \leftarrow v_1] \vdash e_2 : v_2}{E \vdash \text{try}\{e_1\}\ \text{catch}(x)\{e_2\} : v_2}$$

$$\frac{E \vdash e : v}{E \vdash \text{throw}\ e : T(v)}$$

$$\frac{E \vdash e_1 : T(v_1)}{E \vdash e_1 + e_2 : T(v_1)}$$

When we encounter try
- Mark current stack position

When we throw an exception
- Unwind the stack to the first try
- Execute the corresponding catch

More complex techniques reduce the cost of try and catch

—=

In Java, exceptions are part of the method interface and checked by the compiler

public void x() throws MyException

In the original Java project they observed that there are many exceptions that could be raised by Java programs and people lost track of exceptions to handle

Some exceptions do not need to be part of the method signature
- dereferencing null
- others ...

throw must be applied to an object of type Exception

Interfaces

Specify relationships between classes without using inheritance

```
interface PointInterface { void move(int dx, int dy); }
class Point implements PointInter.
  { void move(int dx, int dy) {...} }
```

Inheritance is more efficient and should be used where possible
- Classes implementing interfaces need not be at fixed offsets

```
class Point implements PointInterface {
  void move(int dx, int dy) {...} }

class Point2 implements PointInterface {
  void dummy() {...}
  void move(int dx, int dy) {...} }
```

—=

Dispatch e.f() where e has an interface is more complex than usual

One approach:
- Each class implementing an interface has a lookup table method names → methods
- Hash method name for faster lookup

from the Java language manual

"Java programs can use interfaces to make it unnecessary for related classes to share a common abstract superclass or to add methods to Object"

i.e. they play the same role as multiple inheritance in C++, because a class can implement multiple interfaces

Java coercions

Java allows primitive types to be coerced in certain contexts

1 + 2.0 int 1 is widened to float 1.0

A coercion is a primitive function that the compiler inserts for you

Java distinguishes coercions and casts in
- Widening, always succeeds (int → float)
- Narrowing, may fail if data can't be converted to desired type (float → int)

Narrowing casts must be explicit
widening casts/coercions can be implicit

Coercions can lead to surprising behaviour

Java threads

Each thread in Java has its own program counter and stack (local variables and activation records)

Threads are objects of type Thread
- start and stop methods

synchronized (x) {e}
- locks x, evaluates e and unlocks x

or can be put on methods (more common)

synchronized f () {...}
- object implicitly locked

```
class Simple {
    int a=1, b=2;
    void synchronized to() {...}   <- WRONG! Both methods must have the lock. That does nothing!
    void fro() {...}
- void synchronized to()            <- This is synchronized instead
    void synchronized fro()
```

A variable should only hold values written by some thread
- Writes of values are atomic
  cannot switch thread while writing and only write some bits
- Java guarantees that, except for doubles
  It consumes two words and translates into two machine instructions
  Values created from two different threads are called "out of thin air values"

By declaring double volatile, then the writes will be atomic

Other Topics

Java allows classes to be loaded at runtime
- Type checking of the source takes place at compile time
- Bytecode verification takes place at runtime

Loading policies are handled by the ClassLoader

—=

A class is initialised when a symbol in the class is first used
- Not when the class is loaded
- Delays initialisation errors to a predictable point

1 Lock the object for the class
2 Check if it is already initialised
3 If initialised, return
4 Else mark initialisation as in progress by this thread and unlock the class
5 Initialise superclass (textual order)
- static and final fields first
- Give every field default value before init.
6. Any errors result in an incorrectly initialised class, mark as erroneous
7. If no errors, lock class, label class as initialised, notify threads waiting on class object, unlock class