50006 - Compilers - (Dr Dulay) Lecture 1

Oliver Killane

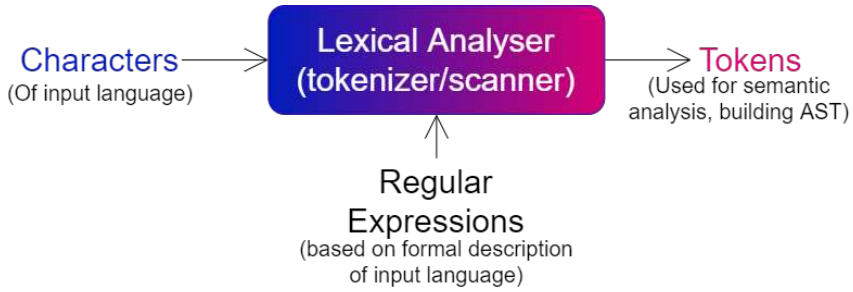
09/01/22

Lecture Recording

Lecture recording is available here

Lexical Analysis

Transforming a stream of characters into a stream of tokens, based on the formal description of the tokens of the input language.



Identifier Tokens

Lexical analyser needs to identify keywords quickly, so often fast string lookup is achieved using a perfect hash function (hash with no possible collisions).

- **Keyword Identifiers**
  Have special meaning in a programming language and are normnally represented by their own token.
  e.g 'class' → CLASS, 'package' → PACKAGE, 'while' → WHILE, etc.
- **Non-Keyword Identifiers**
  Programmer defined identifiers, such as variable names. Typically a generic token is used that uses the provided string name.
  e.g 'var1' → IDENT("var1")

Literal Tokens

Literals (constant values embedded into the input program)

- **Unsigned Integers**
  Represented as a literal token for integers, containing the value used. Tokenizer needs to account for negative integers, as well as varying integer sizes (e.g larger than typical default of 4 bytes) to prevent overflow and correctly assign the value from the input program.
  e.g '123' → INTEGER(123), '1e400' → BIGINTEGER(1e400), '0x11' → INTEGER(17), etc.
- **Unsigned Reals**
  Represented by a literal token for floating-point values, which containg the value. Tokenizer must take into account large and negative floats.
  e.g '17.003' → FLOAT(17.003)
- **Strings**
  Represented by string tokens containing the string (much like non-keyword identifiers). Tokeinzer

Needs to account for input language characters are encoding (e.g unicode, ascii etc), as well as escape characters (backslash)
e.g '"hello, world!"' → STRING("hello, world!")

Other Tokens

- **Operators**
  Usually one or two characters, with their own token. e.g $+, -, *, /, ::, <=$
- **Whitespace**
  Normally removed unless inside a string literal. Some information may be included as metadata for later stages of the compiler (e.g for nice error handling). Normally needed to separate identifiers ('pub mod' → [PUBLIC, MODULE] but 'pubmod' → IDENT("pubmod"))
- **Comments**
  Normally Removed, have no effect on program logic.
- **Pre-processing directives & Macros**
  Most languages remove/process before lexical analysis either by an external tool or an earlier stage of the compiler (e.g pre-processor). One exception to this is **Rust's** macros system, which allows macros to process tokens and is incredibly powerful as a result.

Regular Expressions

Expressions to match strings, cannot be recursive.

a	Match symbol	x matches 'x' only.
$\backslash symbol$	Escape regex char	\(matches '(' only.
ϵ	Match empty string.	ϵ matches '' only.
R_1R_2	Match adjacent.	$ab89$ matches 'ab89' only.
$R_1\|R_2$	Alternation (or), match either regex	$abc\|1$ matches 'abc' and '1'.
(R)	Group regexes together	$(a\|b)c$ matches 'ac' and 'bc'.
$R+$	One or more repetitions of expression R	$(a\|b\|j)+$ matches all non-empty strings of a,b & j (e.g 'abbjab'
$R*$	Zero or more repetitions	$R*$ is equivalent to $(R+)\|\epsilon$

Precedence (highest → lowest) grouping, repetition, concatenation, alternation.

The following can be derived from the previous rules.

$R?$	Optional, zero or one occurence	$a?$ matches '' and 'a'.
.	wildcard, match any character	$.*$ matches every possible string.
$[abcd]$	Character Set, match any character in the set	$[abc]$ matches 'a', 'b' and 'c'.
$[0-9]$	Match characters in range	$[a-zA-Z]$ matches all single alphabet character
$[\wedge abc]$	Match all characters except those in the character set.	$[^0-9]*$ matches all strings with no numbers.

These expressions can be used in production rules:

$$
\begin{array}{lcl}
Digit & \rightarrow & [0-9] \\
Int & \rightarrow & Digit+ \\
Signedint & \rightarrow & (+|-)?Int \\
Keyword & \rightarrow & 'if' \mid 'while' \mid 'do' \\
Letter & \rightarrow & [a-zA-Z] \\
Identifier & \rightarrow & Letter(Letter|Digit)*
\end{array}
$$

Disambiguation Rules

When more than one expression matches, choose the longest matching character sequence. Otherwise assume regular expression rules are ordered (textual precedence, earlier rule takes precedence).

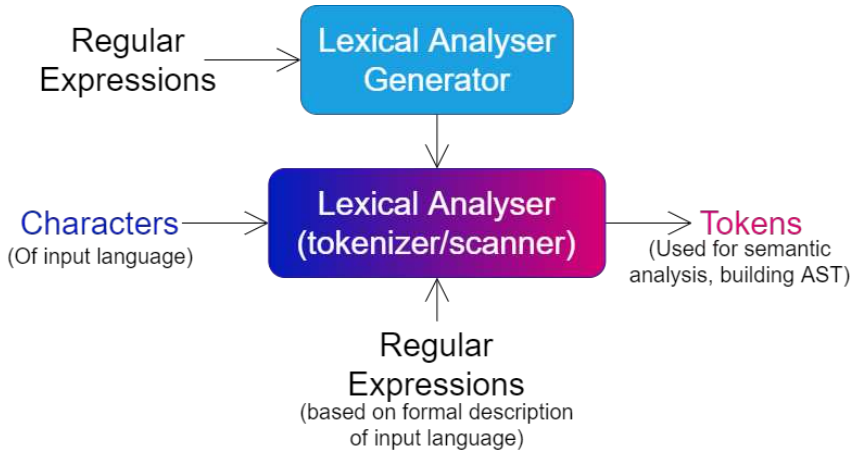
$$Keyword \rightarrow while \mid if \mid do \quad \text{and} \quad Identifier \rightarrow Letter(Letter|Digit)*$$

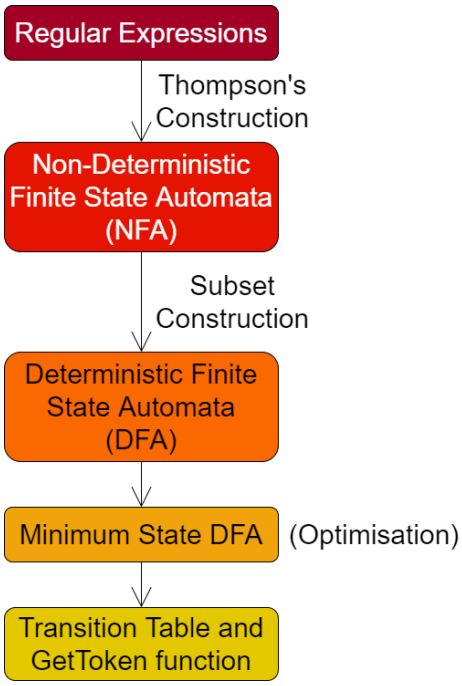
'whileactive' matches $Identifier$ and $Keyword$, however $Identifier$ matches all of the string (longer) so is chosen.

Lexical Analyser Implementation

- **Manually Implement**
  Relatively easy to write from scratch, however can become very difficult to change to change rules for tokens, and many rules implemented in an ad-hoc fashion.
- **Lexical Analyser Generators**
  For example **Logos** which take programmer defined token structures & functions to consume matches specified by regular expressions. They generate a tokenizer convert inputs into the defined tokens.

  An example is included in the **code** directory of this lecture.

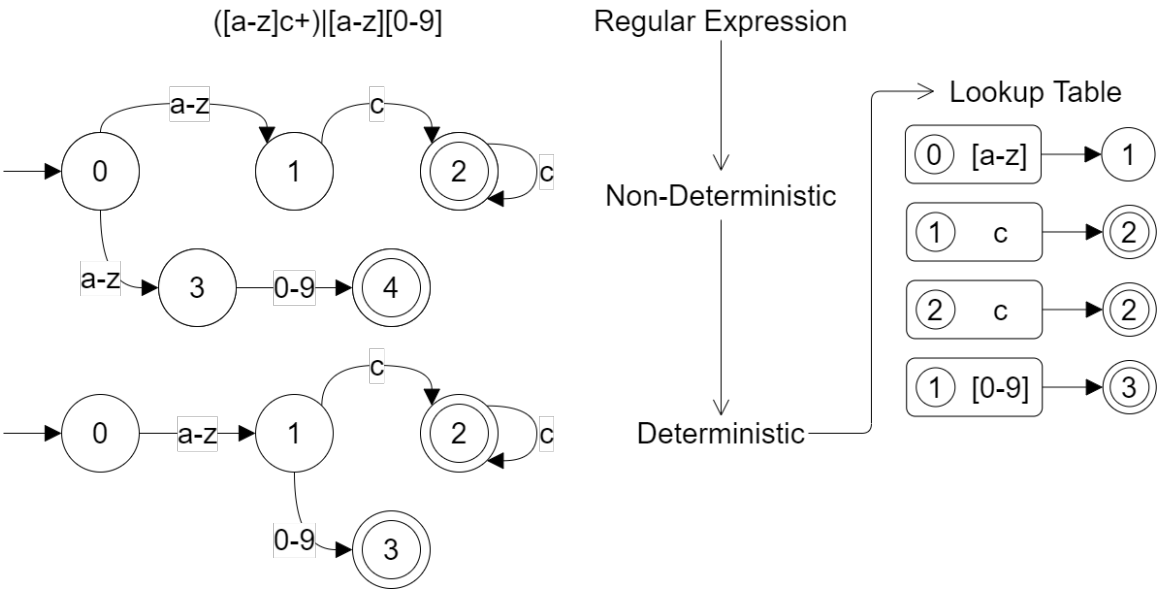




Finite Automata

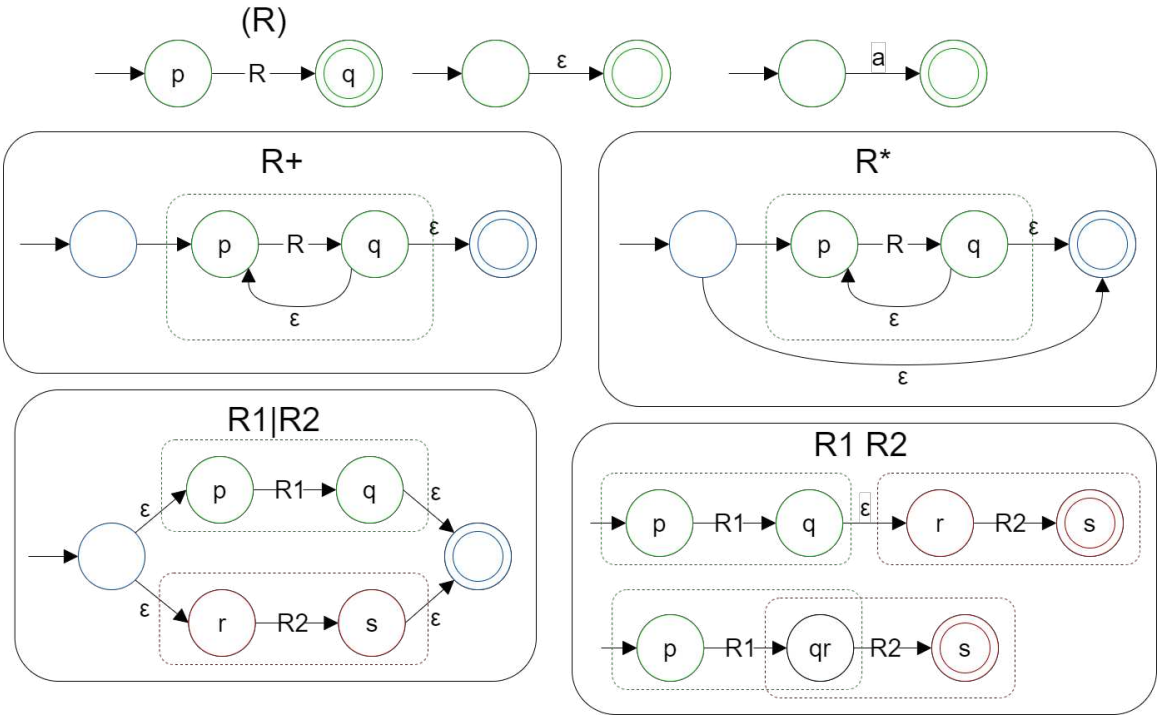
Also called finite state machines.

- Arrows denote treansitions between states.
- Start state has an unlabelled transition to it.
- Accepting states are double circles.
- Technically each non-accepting state should have a transition for every symbol (potentially to an error state), however these are ommitted from the diagram for conciseness.
- Will stop when no transition can be made (as a result matches the longest string possible to a state).



Regular Expressions to Nondeterministic Finite Automata

Thompson's construction is used to translate, it uses ϵ transitions to stick NFAs together.



Subset Construction

- NFA traversal requires backtracking (for a state, input must try every branch for that input).
- Backtracking is slow.
- DFA traversal is faster (no backtracking) so compilers convert to DFA by removing all ϵ transitions instances of multiple paths for an input.
- DFAs often require more memory than NFAs (up to 2^n states for an n state NFA) so some application such as regex searches in some editors use them.

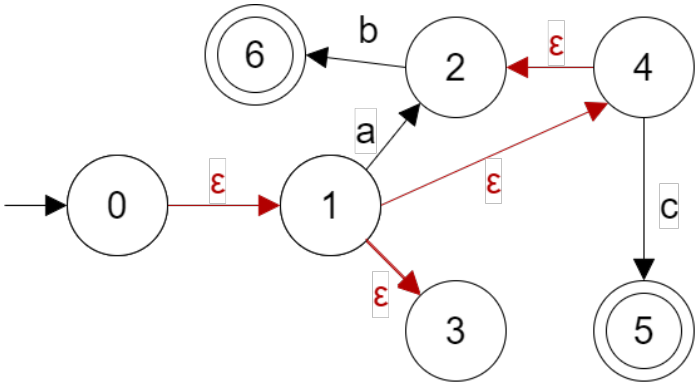
Subset Construction eliminates ϵ transitions, converts to a DFA. States of DFA are subsets of states in the NFA, hence the name.

	Space	**Time**
NFA	$O(len\ R)$	$O(len\ R \times len\ X)*$
DFA	$O(2^{len\ R})$	$O(len\ X)$

It is very rare for the space of the DFA to be an issue with compilers, so they are always used for the increased speed.

ϵ Closures

ϵ-**Closure**(s)	Set of all states that can be reached through only ϵ transitions (including itself).
ϵ-**Closure**$(s_1, \ldots, s_n)$	union of the closures for $s_1, \ldots, s_n$.



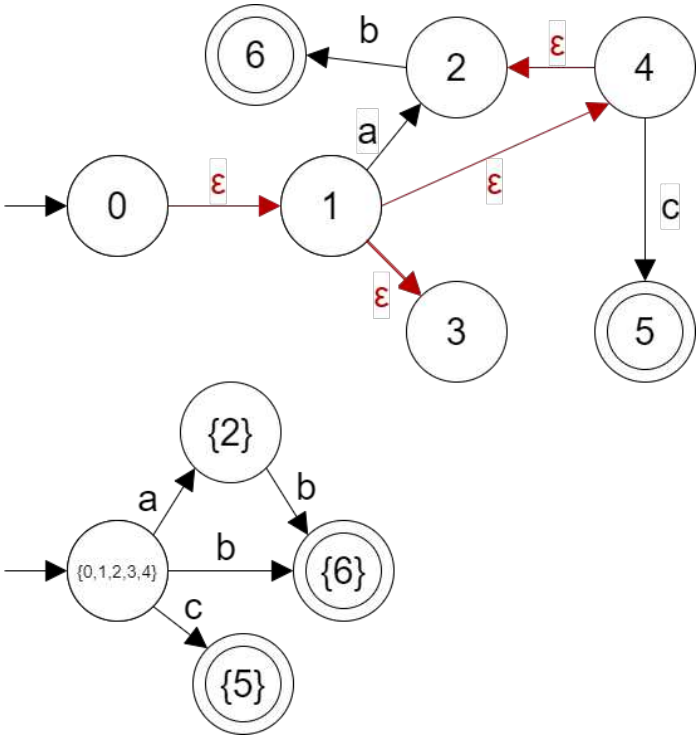
ϵ-Closure(0) = $\{0, 1, 2, 3, 4\}$
ϵ-Closure(1) = $\{1, 2, 3, 4\}$
ϵ-Closure(2) = $\{2\}$
ϵ-Closure(3) = $\{3\}$
ϵ-Closure(4) = $\{2, 4\}$
ϵ-Closure(5) = $\{5\}$

Generating Subset Construction

1. Start at NFA start state.
2. Get the ϵ-Closure (all states the machine could possibly be in)
3. For each possible transition (not including erroneous) create a transition to the possible states (e.g if in the current ϵ-Closure 'a' goes to states 7 and 10, then 'a' should now transition to state $\{7, 10\}$).

4. COntinue step 2 for the states transitions have been created to.

A good example walkthrough can be found here.



50006 - Compilers - (Dr Dulay) Lecture 2

Oliver Killane

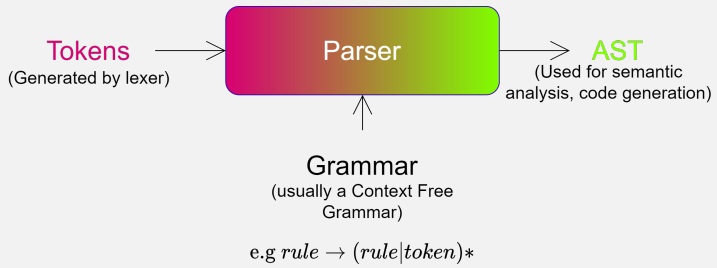
07/04/22

Lecture Recording

Lecture recording is available here

Parsing

Definition: Parsing (Static Analysis)



Transforming a sequence of tokens into an **abstract syntax tree** using a language's **grammar**.

Chomsky Hierarchy

A hierarchy of grammars and the machines required to parse them.

Given that:

R	non-terminal
t	sequence of tokens
α, β, φ	sequences of terminals and non-terminals

We can express the rules as:

Type 3	$R \rightarrow t$	Regular	DFA
Type 2	$R \rightarrow \alpha$	Context Free	Pushdown automata (DFA with a stack and read-only tape)
Type 1	$\alpha\ R\ \beta \rightarrow \alpha\ \varphi\ \beta$ where $R \rightarrow \varphi$	Context Sensitive	Linear Bounded Automata (Turing machine with limited tape)
Type 0	$\alpha \rightarrow \beta$	Unrestricted/recursively enumerable	Turing Machine

regular $\subset$ context-free $\subset$ context-sensitive $\subset$ recursively enumerable

Parsers for Context Free Grammars

Context-free grammars are parsed with complexity $O(n^3)$, **LL** and **LR** grammars are subsets of **CFG** that can be parsed in $O(n)$. Note that **LL(n)** is a subset of **LR(n)**.

Higher lookahead allows for more powerful & complex grammars, at the cost of performance (particularly memory).

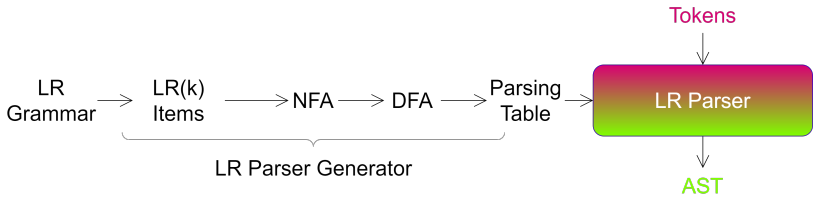
LL/Top Down Parsers (Builds **AST** from root to leaves)	
LL(k)	**L**eft-to-right scan, **L**eftmost-derivation, **k**-token look-ahead.
LL(0)	Does not exist (needs to have lookahead of at least 1).
LL(1)	Most Popular. Can be implemented using recursive descent, r as an automaton.
LL(2)	Sometimes useful.
LL(k > 2)	Rarely Needed.
LR/Bottom-Up Parsers (Builds **AST** from leaves to root)	
LR(k)	**L**eft-to-right scan, **R**ightmost-derivation (in reverse) with **k**-token lookahead.
LR(0)	Weak & used in education.
SLR(1)	Stronger than **LR(0)**, but superseded by **LALR(1)**
LALR(1)	Fast, popular and comparable power to **LR(1)**
LR(1)	Powerful but has high memory usage.
LR(k ≥ 2)	Possible but rarely used.

Programming Language Grammars

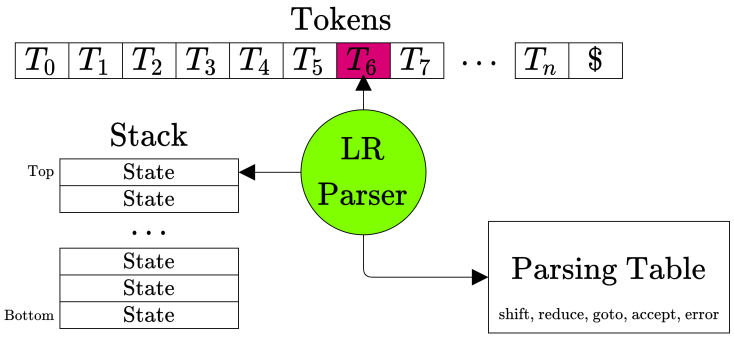
Most programming languages' syntaxs are described using **Context-Free Grammars**. Almost all **context-free** programming language grammars can be described as an **LR grammar** and implemented using an **LR parser**.

C++ is much more complex, as it is not a context-free grammar (very well explained here.)

LR Parser



LR Parser Operation



shift Sn	Push state n onto the stack and move to the next token.
reduce Rn	1. Use rule n to remove m states from the stack (where m = length of RHS of rule n). 2. Get the rule associated with the new-top of stack (i.e perform the goto action). 3. Push back the LHS of rule n. 4. Generate the new AST node for the rule.
accept a	Parse was successful
error	Report an error. Note that errors may be added to the AST under construction so multiply syntax errors can be reported.
goto Gn	Used in the reduce case, not directly.

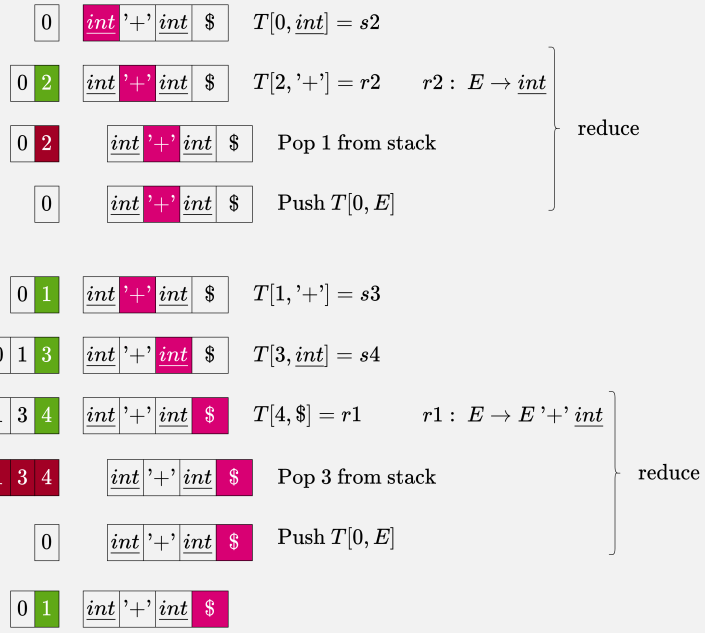
Example: Basic Parser Operations

With a basic grammar:

$$
\begin{aligned}
& E' \rightarrow E\ \$ \\
r1: \quad & E \rightarrow E\ \text{'+'}\ \underline{int} \\
r2: \quad & E \rightarrow \underline{int}
\end{aligned}
$$

We have an **LR(1) Parse Table**:

State	Action			GOTO
	int	'+'	$	*E*
0	s2			g1
1		s3	a	
2		r2	r2	
3	s4			
4		r1	r1	



End of Input

For **LR** parsers, an end of input symbol $ rule is always required.

$$E' \rightarrow E\ \$$$

LR(0) Parsers

Definition: LR(0) Items

Instances of the rules of the grammar with $\bullet$ (representing the current position of the parser) in some position on the right hand side of the rule.

$$X \to \underbrace{AB\ldots}_{\text{Has been matched}} \bullet \underbrace{\ldots YZ}_{\text{To be matched}}$$

$X \to \bullet ABC$	**Initial Item** (has not yet matched any part of the rule)
$X \to A \bullet BC$	
$X \to AB \bullet C$	
$X \to ABC\bullet$	**Complete Item** (Has matched the whole rule)

We can use the **LR(0) items** as states in our **NFA** as they track thr progress of the parser through a rule.

We can create transitions between **LR(0) Items**:

$$\underbrace{(X \to A \bullet BC)}_{\text{current state}} \overbrace{\to_B}^{\text{If encountering a } B} \underbrace{(X \to AB \bullet C)}_{\text{new state}}$$

If B is a non-terminal, for each rule $B \to \bullet D$ we add a ϵ transition (no token needed to match):

$$(X \to A \bullet BC) \to_\epsilon (B \to \bullet D)$$

Definition: LR(0) Parsing Table

The parsing table describes the rules to apply, and states to move to when a given item is encountered.

It is generated from a **DFA** using the rules:

Terminal translation $X \rightarrow_T Y$	Add $P[X, T] = sY$ (shift Y, we cannot reduce yet)
Non-terminal translation $X \rightarrow_N Y$	Add $P[X, N] = gY$ (Goto Y)
State X containing item $R' \rightarrow \cdots \bullet$	Add $P[X, \$] = a$ (accept) (end of input)
State X containing item $R \rightarrow \cdots \bullet$	Add $P[X, T] = rN$ (reduce) (use rule rN to reduce)

State	**Action**			**GOTO**
	int	'+'	$	*E*
0	s3			
1		s4	a	
2	r1	r1	r1	
3	r3	r3	r3	
⋮				

-

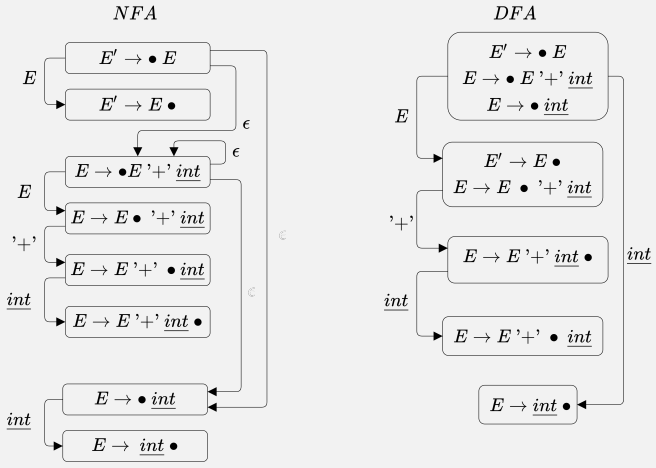
To create an **LR(0)** parser the steps are:

1. To generate an **LR(0)** parser the context free grammar is converted into **LR(0) items**.
2. The items are used as states in an **NFA**, with ϵ transitions for non-terminals.
3. The **NFA** is converted to a **DFA**.
4. Generate a **parsing table** from the **DFA**

Example: Basic **LR(0)** Example

For a language allowing only addition expressions.

Grammar	LR(0) Items
$E' \to E\ \$$	$E' \to \bullet E$ (Initial Item, the \$ is implied) $E' \to E\bullet$
$E \to E\text{'+'}\underline{int}$	$E \to \bullet E\text{'+'}\underline{int}$ (Initial Item) $E \to E \bullet \text{'+'}\underline{int}$ $E \to E\text{'+'} \bullet \underline{int}$ $E \to E\text{'+'}\underline{int} \bullet$ (Complete Item)
$E \to \underline{int}$	$E \to \bullet\underline{int}$ (Initial Item) $E \to \underline{int} \bullet$ (Complete Item)



State	Action			GOTO
	int	'+'	$	*E*
0	s3			g2
1		s4	a	
2	r1	r1	r1	
3	r3	r3	r3	
⋮				

LR(1) Parsers

Lecture Recording

Lecture recording is available here

Notation

$\rightarrow^*$	Zero or more derivations.
$\rightarrow^+$	One or more derivations.
$A \rightarrow \epsilon$	Non-terminal A is **nullable**.
$A \rightarrow AB \vert BC$	AB and BC are **alternatives** of A.

Definition: First Set

The **first set** for a sequence of rules (non-terminals) and tokens (terminals) α is the set of all tokens that could start a derivation α.

$$\text{first}(\alpha) = \{t : \alpha \rightarrow^* t\beta\} \cup [\begin{cases} \{\epsilon\} & \text{if } \alpha \rightarrow *\epsilon \\ \{\} & otherwise \end{cases}]$$

- For a token T the $\text{first}(T) = \{T\}$
- $\text{first}(\epsilon) = \{\epsilon\}$

We can construct **first set** from the rule by going through each alternative. In each alternative we iterate through each terminal/non-terminal B, if $\epsilon \in \text{first}(B)$ then we include $\text{first}(B) \setminus \{\epsilon\}$ in the **first set** and move on to the next in the sequence.

```
1  def get_first_set(non_terminal):
2      first_set = []
3      # go through each alternative
4      for alternative in non_terminal:
5          # iterate through all terminal/non-terminals in the alternative until
6          # one does not have an epsilon.
7
8          # If it has an epsilon, it means we could potentially skip through to
9          # the next production (e.g in "i*j" the first if i has epsilon, as we
10         # can skip it, hence we must also consider j).
11
12         for b in alternative:
13             b_first_set = get_first_set(b)
14             first_set += b_first_set - EPSILON
15
16             # if no epsilon in the next terminal/non-terminal, we stop adding
17             # to the first set.
18             if EPSILON not in get_first_set(b):
19                 break
```

Definition: Follow Set

The follow set for a non-terminal rule is the set of all tokens (terminals) that could immediately follow, and \$ if the end of the input could follow.

$$\text{Follow Set}(A) = \{t : S \to^+ \alpha A t \beta\} \cup \begin{cases} \{\$\} & \text{if } S \to *\alpha A \\ \{\} & otherwise \end{cases}$$

Hence we must check each rule which contains the non-terminal.

$$\text{Given rule } B \to C\ A\ D \text{ we have follow}(A) = \text{first}(D) \setminus \{\epsilon\} \cup \begin{cases} \text{follow}(B) & \text{if } D \to^* \epsilon \\ \{\} & otherwise \end{cases}$$

- C can be empty here, since it is at the start of the rule, it has no bearing on the **follow set**.
- If A can end the input, we include \$ in the **follow set**.

Definition: LR(1) Items

An **LR(1) Item** is a pair:

$$[\text{LR(0) item},\ \text{lookahead token } t]$$

Hence given an **LR(1) item**

$$[X \to A\ \bullet B\ C,\ t]$$

- A is on top of the stack.
- We only want to recognise B if it is followed by a string derivable from Ct.
- B is followed by Ct if the current token $\in$ first(Ct).

NFA Transitions

Given a state $[X \to A\ \bullet\ B\ C,\ t]$ we add the transition:

$$[X \to A\ \bullet\ B\ C,\ t] \to_B [X \to A\ B\ \bullet\ C,\ t]$$

If B is a rule/non-terminal then for every rule of form $B \to D$ we add an ϵ transition and a new state for every token u in first(Ct) (Add a transition iff B derivable from Ct).

$$[X \to A\ \bullet\ B\ C,\ t] \to_\epsilon [B \to \bullet\ D,\ u]$$

We also need an initial item for the rule $[X' \to \bullet\ X, \$]$ (start of text, has end of input token \$).

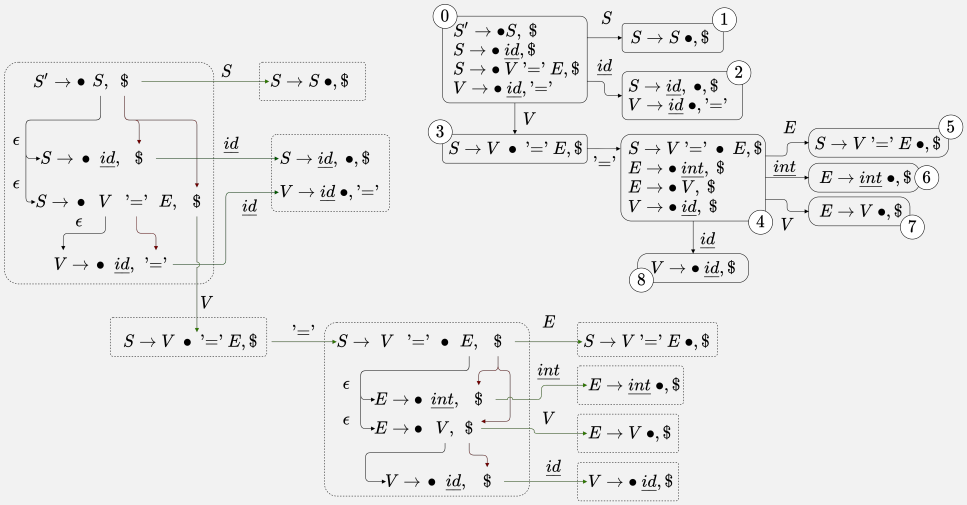
Definition: LR(1) Parsing Table

Our parsing table can now contain several different rules per row (same state, different current token).

We only perform reduction of a rule $[A \to A\ \bullet,\ t]$ when the current token is t.

Example: Basic LR(1) Grammar

Grammar	Expanded
	$S' \rightarrow S\ \$$
$S \rightarrow \underline{id} \mid V\text{'='}E$	$r1 : S \rightarrow \underline{id}$
	$r2 : S \rightarrow V\text{'='}E$
$V \rightarrow \underline{id}$	$r3 : V \rightarrow \underline{id}$
$E \rightarrow V \mid \underline{int}$	$r4 : E \rightarrow V$
	$r5 : E \rightarrow \underline{int}$



State	Action				GOTO		
	id	*int*	'='	$	E	V	S
0	s2					g3	g1
1				a			
2			r3	r1			
3			s4				
4		s8	s6			g5	g7
5				r2			
6				r5			
7				r4			
8				r3			

LALR(1) Parsers

Definition: LALR(1) Items

LALR(1) parsers are similar to **LR(1)** parsers, however **LR(1)** states that have the same **LR(0) items** and only differ in the lookahead token are merged. This reduces the memory usage of **LALR(1) Parsers**.

e.g LR(1) items $\begin{matrix}[R \to A\ B\ \bullet\ C, t_0] \\ [R \to A\ B\ \bullet\ C, t_1] \\ \vdots \\ [R \to A\ B\ \bullet\ C, t_n]\end{matrix}$ become LALR(1) Item $[R \to A\ B\ \bullet\ C, \{t_0, t_1, \ldots, t_n\}]$

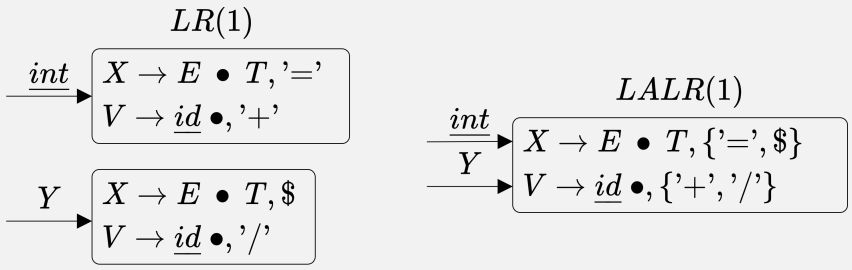
Due to this merging, some reductions can occur before an error is detected, which would've been immediately detected by an **LR(1)** parser.

$$\text{Grammar } \begin{matrix}A \to \text{'('}\ A\ \text{')'} \\ A \to +\end{matrix} \text{ given input string "+)"}$$

With **LR(1)** as soon as we get to + and shift it, we immediately have an error.

However for **LALR(1)** as we would two states $A \to$ '+'$\bullet$, \$ and $A \to$ '+'$\bullet$, ')' merged, we would detect the error after reaching the ')'.

Example: LALR(1) Items



Conflicts and Ambiguity

Definition: Ambiguous Grammars

Grammars for which more than one parse tree can be created for some input.

Ambiguous grammars cannot be **LR(k)** as during parsing it will reach a state where it cannot decide wether to shift or reduce.

Ambiguity can be resolved two ways:

1. Rewrite the grammar to remove ambiguity.
2. Augment grammar with additional rules (e.g associativity or precedence)

For **LR** parser generators shifting is given priority, when a shoft-reduce confligt occurs, a warning is typically given.

Definition: Shift-Reduce Conflict

Caused by an ambiguity where the parser cannot decide if to reduce the tokens to the LHS of a rule, of continue to shift another token.

For example the grammar below would cause a **shift-reduce conflict**.

$$Expr \rightarrow \underline{Num} \quad \text{and} \quad Expr \rightarrow \underline{Num} \text{ '+' } \underline{Num}$$

Example: Shift-Reduce Conflict

Given the grammar:

$$S \rightarrow \text{'if' } E \text{ 'then' } S \text{ 'else' } S$$
$$S \rightarrow \text{'if' } E \text{ 'then' } S$$
$$S \rightarrow \underline{id}$$

We can generate **LR(1) items**:

$$[S \rightarrow \text{'if' } E \text{ 'then' } S \bullet, \text{'else'}]$$
$$[S \rightarrow \text{'if' } E \text{ 'then' } S \bullet \text{ 'else' } S, \underline{any}]$$

Given some input *"if a then if b then c else d"* we can parse:

$$\underset{\text{Shift}}{if\ a\ then\ \underbrace{if\ b\ then\ c\ else\ d}_{\text{inner if}}} \qquad \underset{\text{reduce}}{if\ a\ then\ \underbrace{if\ b\ then\ c}_{\text{inner if}}\ else\ d}$$

We can resolve this using a modified grammar:

$$S \rightarrow MatchedS$$
$$MatchedS \rightarrow \text{'if' } E \text{ 'then' } MatchedS \text{ 'else' } MatchedS$$
$$MatchedS \rightarrow other$$

$$S \rightarrow UnMatchedS$$
$$UnMatchedS \rightarrow \text{'if' } E \text{ 'then' } S$$
$$UnMatchedS \rightarrow \text{'if' } E \text{ 'then' } MatchedS \text{ 'else' } UnMatchedS$$

This grammar ensures only matched statements can come before the 'else' in an if statement, hence the ambiguity is resolved.

Definition: Reduce-Reduce Conflict

Caused by two rules having the same RHS. Cannot determine which rule should be applied.

For example the following grammar has a **reduce-reduce conflict**.

$$Expr \rightarrow \underline{id} \quad \text{and} \quad Var \rightarrow \underline{id}$$

A common disambiguation rule added for LR parser generators is to prioritise the earliest rule from the grammar source.

Example: Conflict with Precedence

For example the grammar:

$$Expr \rightarrow Expr \text{ '+' } Expr \mid Expr \text{ '-' } Expr \mid Expr \text{ '*' } Expr \mid \text{'(' } Expr \text{ ')'} \mid \underline{int}$$

Hence if given a source *"3 + 2 * 5"* we have a conflict

$$(3 + 2) * 5 \text{ or } 3 + (2 * 5)$$

Hence we can diambiguate by using several rules.

$$\begin{aligned} Expr &\rightarrow Expr \text{ '+' } Term \mid Term \\ Term &\rightarrow Term \text{ '*' } Factor \mid Factor \\ Factor &\rightarrow \text{'(' } expr \text{ ')'} \mid \underline{int} \end{aligned}$$

Parse Tree

A parse tree is produced by the parser. It is broadly similar to the AST, but can contain much redundant information associated with the grammar.

- Leaf nodes built on a shift operations
- Non-Leaf nodes created on a reduce.

To produce the AST we can either build it in a pass over the completed parse tree, or associate AST construction code with rules in the grammar to build the AST during parsing.

50006 - Compilers - (Dr Dulay) Lecture 3

Oliver Killane

09/04/22

Lecture Recording

Lecture recording is available here

LL Parsing

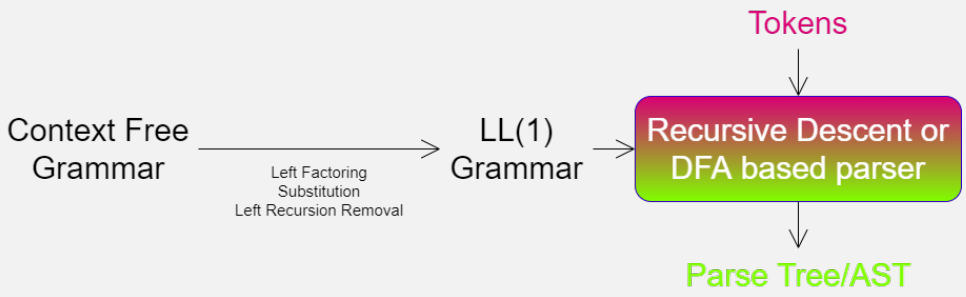
Top-down parsing using either recursive descent (can be hand-coded) or an **LL(1)** pushdown automaton (generated by an **LL(1)** parser-generator (e.g ANTLR)).

Definition: LL(k) Grammar

A grammar is **LL(k)** if a k-token lookahead is sufficient to determine which alternative of a rule to use when parsing.

- **LL(1)** uses the current token only.
- **LL(0)** does not exist (would be deciding based on 0 tokens).
- When using **LL/Top Down Parsing** leaf nodes are constructed from the root.

The parser created for the grammar can be implemented as either a **DFA** or to parse by **recursive descent**.



Definition: LL(1) Grammar

A grammar is **LL(1)** if for a rule $A \to \alpha \mid \beta$ (A is non-terminal).

$$
\begin{aligned}
&first(\alpha) \cap first(\beta) = \emptyset \\
\wedge \epsilon \in first(\alpha) &\Rightarrow (first(\beta) \cap follow(A) = \emptyset) \\
\wedge \epsilon \in first(\beta) &\Rightarrow (first(\alpha) \cap follow(A) = \emptyset)
\end{aligned}
$$

$$A \to \underbrace{\alpha}_{t \in first(\alpha)} \mid \overbrace{\beta}^{t \in first(\beta)}$$

If both could start with t, if we get t as the current token, we cannot determine if we are parsing a or b.

$$A \to \underbrace{\alpha}_{\epsilon \in first(\alpha)} \mid \overbrace{\beta}^{t \in first(\beta)}$$

$follow(A) = \{\ldots, t, \ldots\}$

If a contains e, then we could potentially skip over this rule to whatever follows A.

Here with token t we cannot determine if we are skipping past this rule, or parsing b.

And the other order, and with many alternatives

This extends to more than two alternatives.

Definition: Extended Backus-Naur Form (BNF)

Used for writing context free grammars, and includes several useful features:

$\{\alpha\}$	0 or more occurrences of α
$[\alpha]$	0 or 1 occurrences of α
$(\ldots)$	A way to group elements together

For example we can left factor rules with alternatives that have intersecting first sets.

$$Expr \to Term \text{ '+' } Expr \mid Term$$

then becomes

$$Expr \to Term \text{ ['+' } Expr]$$

Or we can remove left recursion:

$$Sequence \to Sequence \text{ ';' } Statement \mid Statement$$

then becomes

$$Sequence \to Statement \text{ \{';' } Statement\}$$

Definition: Recursive Descent Parser

Consists of a set of parse functions for each rule which take in some tokens, and return remaining tokens and a generated **AST**.

We can use a basic function for matching **terminal** tokens:

```
# Get the next token. If it matches the expected, pop it from
# the list of tokens, else throw an error.
def match(expected: token):
    if lexical_analyser.next_token() == expected:
        lexical_analyser.pop_token()
    else:
        raise error("Unexpected token")
```

We can apply some basic rules for the main patterns:

```
# A B
A()
B()

# A | B (to be an LL(1) grammar, first sets must be disjoint)
if next_token() in first(A):
    A()
elif next_token() in first(B):
    B()

# {A}
while next_token() in first(A):
    A()

#[A]
if next_token() in first(A):
    A()
```

Example: Statements

$$Stat \rightarrow IfStat \mid BeginStat \mid PrintStat$$
$$IfStat \rightarrow \text{'if'}\ Expr\ \text{'then'}\ Stat\ [\text{'else'}\ Stat]\ \text{'fi'}$$
$$BeginSTat \rightarrow \text{'begin'}\ Stat\ \{\text{';'}\ Stat\}\ \text{'end'}$$
$$PrintStat \rightarrow \text{'print'}\ Expr$$

```
1  def Stat() -> Statement:
2      if next_token == IF:
3          return IfStat()
4      elif next_token == BEGIN:
5          return BeginStat()
6      elif next_token == PRINT:
7          return PrintStat()
8      else:
9          raise error("Expected Statement Starting Token")
10
11 # Parse an if statement, returning the statement if parse succeeds.
12 def IfStat() -> Statement:
13     match(IF)
14     cond = Expr()
15     match(THEN)
16     if_branch = Stat()
17     else_branch = None
18     if next_token() == ELSE:
19         match(ELSE)
20         else_branch = Stat()
21     match(FI)
22     return IfStatement(cond, if_branch, else_branch)
23
24 # Parse a block of statements, returning the block statement if
       ↪ successful
25 def BeginStat() -> Statement:
26     stats = []
27     match(BEGIN)
28     stats.append(Stat())
29     while next_token() == SEMICOLON:
30         match(SEMICOLON)
31         stats.append(Stat())
32     match(END)
33     return Block(stats)
34
35 # Parse a print statement, returning the statement if successful.
36 def PrintStat() -> Statement:
37     match(PRINT)
38     return PrintStatement(Expr())
```

AST Construction

Class hierarchies can be used to create organise nodes into variants of a given type (e.g if statements, print statements and assignments are all statements, so implement/inherit from some STatement class/interface).

Other languages

While in languages such as Java or Python, class hierarchies are used to implement ASTs other languages have cleaner representations.

- In **Haskell** the data keyword can be used to build an **ADT** for an **AST**
- Rust supports enums similar to the haskell data keyword.

CFG → LL(1) Conversion

Transformations must be applied to a non-LL(1) context free grammar, which cannot always be automated.

- Left Recursion Removal and Substitution are usually applied first.
- The semantics of the grammar must be maintained.
- Readability is a key consideration, expecially if the language is to be extended in future.

Definition: Left Factorisation

Where two or more alternatives of a rule have a common prefix (first element to be parsed), factor this to be parsed before determining which alternative to parse.

non-LL(1)	**EBNF LL(1)**	**BNF LL(1)**
$A \to B\ C \mid B\ D$	$A \to B\ (C \mid D)$	$A \to B\ X$ $X \to C \mid D$
$A \to B\ C \mid B$	$A \to B\ [C]$	$A \to B\ X$ $X \to C \mid \epsilon$

Definition: Substitution

Substituting a rule/non-terminal with its alternatives.

- Can be used to find indirect conflicts that may require left-factoring.
- Can produce grammars ecoding more information in a rule (makes it easier to produce the required AST)

$$A \to B \mid C$$
$$B \to \text{'hello'}$$
$$C \to \text{'hello' 'there'}$$

Here we have an indirect conflict as both alternatives for C start with 'hello'.

$$A \to \text{'hello'} \mid \text{'hello' 'there'}$$

Now we can left-factor.

$$A \to \text{'hello' ['there']}$$

Definition: Left Recursion Removal

Grammars cannot be **LL(1)** with left-recursion. We can use **direct left recursion removal** to eliminate left recursion from rules while leaving the grammar mostly intact.

$$A \to X \mid A\ Y \Rightarrow A \to X\{Y\}$$

For example with left-associative arithmetic expressions:

$$\begin{aligned} Expr &\to Expr\ (\text{'+'} \mid \text{'-'})\ Term \mid Term \\ Term &\to Term\ (\text{'*'} \mid \text{'/'}) \mid Factor \\ Factor &\to \text{'('}\ Expr\ \text{')'} \mid \underline{int} \end{aligned}$$

The **parse tree** may no longer represent the associativity, hence we will need to ensure the arithmetic is still associative when we construct the **AST**:

$$\begin{aligned} Expr &\to Term\ \{(\text{'+'} \mid \text{'-'})\ Term\} \\ Term &\to Factor\ \{(\text{'*'} \mid \text{'/'})\ Factor\} \\ Factor &\to \text{'('}\ Expr\ \text{')'} \mid \underline{int} \end{aligned}$$

Error Recovery

When detecting an error in parsing...

- Useful error messages need to contain information collected during parsing.
- Error recovery can be used to allow further parse errors to be detected, with as little code as possible being skipped but avoiding nonsense/spurious error messages being created as a result.
- Error correction can be attempted to allow for semantic checks to be performed. The corrections must attempt to emulate what the semantics of the erroneous code likely was.

Definition: Panic Mode Recovery

Each parse function has a **syncset** of tokens. When an error occur, the parser skips forwards until it encounters one of these tokens.

- Additional tokens can be provided as arguments to the parsing function (e.g add **follow set** of outer non-terminal to inner parse)
- The **follow set** of a rule is often used as the **syncset**
-

```
1  # match a token, if necessary report and error, return boolean if the
       ↪ token matched expected.
2  def match(expected) -> bool:
3      if next_token() == expected:
4          pop_token()
5          return True
6      else:
7          add_error(next_token(), parser_pos(), [expected],
               ↪ INCORRECT_TOKEN)
8          return False
9
10 # Skip until at the end of the file, or token matches the sync set
11 def skipto(syncset):
12     while next_token() not in syncset and next_token() != EOF:
13         pop_token()
14
15
16 def check(expectset, syncset, error):
17     if next_token() not in expectset:
18         add_error(next_token(), parser_pos(), expectset, error)
19         skipto(expectset + syncset)
```

50006 - Compilers - (Dr Dulay) Lecture 4

Oliver Killane

10/04/22

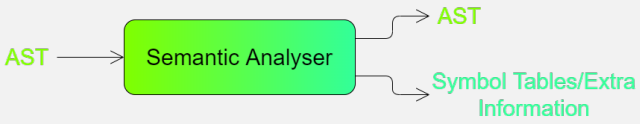
Semantic Analysis Overview

Lecture Recording

Lecture recording is available here

Definition: Semantic Analysis/Context Analysis

Compiler phase which checks (statically at compile time) if the program is semantically valid within the rules of the language. It also provides information for the next phases (code generation) such as creating symbol tables of identifiers and their types.



Some languages perform very few checks (e.g Python is highly dynamic, e.g types checked at runtime), to many (e.g Rust - among the most extreme) .

Semantic analysers can be generated from **attribute grammars** (which associate semantic rules with AST node types), however they are typically hand-written.

Typical Semantic Checks

variable Declaration

$$< Type >=< Id >$$

- **Check the type is in scope (the type is declared)**
  e.g $someclass = somevar$ the $someclass$ must be defined, if imported from another module/file further checks for qualified names, overloading etc must be done.
- **Check a declaration of the type is valid**
  The language may prevent some declarations, e.g $void\ a$;, or where the type is an abstract class.

  Furthermore some langagues place constraints on the order of declarations (e.g forward declaration in C).
- **Check the idenfitier in the current scope**
  If the language does not allow variable shadowing (renaming in inner scopes).
- **and more...**

Assignment

$$< Id >=< Expr >$$

- **Check the identifier is valid**
  Includes checking the scope to ensure it has been declared, as well as rules on variable types (e.g parameters).
- **Check the expression**
  The inner expression must be semantically analysed using rules for expressions.

- **Check the types**
  Given the expression is valid, it will have a type (or range of possible types). The semantic analyser must check it can coalesce this type into that of the assignment.

  It may also need to check the ranges, e.g assigning a constant larger than 255 to an unsigned byte-size type (e.g $u8$ in Rust or *unsigned char* in C).

Array Declaration

$$< Type > < Id > [< Size >] \dots$$

- **Check the declared Type is valid**
  Normal checks for if the type is valid, declared/in-scope. It may not be valid to create arrays of a given type (e.g $voidarray[3]$ is invalid, though $void * array$ is valid in C)
- **Check the size is valid**
  Type check the size expression, potentially only allowing constants (e.g for stack allocated arrays a size may be required).
- **Checking for scoping**
  Check for rules including variable shadowing, just as with declarations.
- **Array Size Warnings**
  Some array sizes may be too large (e.g large stack allocated arrays risking stack overflow in a recursive function), so a warning may need to be created.

Array Element Assignment

$$< Id > [< Index >] \cdots =< Expr >$$

- **Variable assignment scope checks**
  To check if the variable identifier is in scope.
- **Index and Type Check**
  We must check that the idenfifer's type allows it to be indexable (either it is an array or potentially implements some interface allowing indexing).

  The type of the assigned expression must match the de-indexed type of the variable (e.g $int\ a[][]; a[3] = \{1, 2, 3\}$ is correct as $a[i]$ is an integer array).
- **Bounds Checks**
  For each index we may need to check bounds (e.g index is within the bounds)

Function Declarations

$$< Ret\ Type > < Name > (< Param\ Type > < Param\ Id >, \dots)\{\dots\}$$

- **Checking types of return and the parameters**
  Normal variable declaration checking.
- **Checking the function name declared**
  The language may allow for function overloading, or defining functions in nested scopes (modules).
- **Checking scoping rules**
  Some languages may have rules for parameter scopes, for example in WACC the parameters exist in a scope enclosing the function body.

- **Checking for returns**
  Some languages require the return type of the body to be correct (and have no incorrect implicit returns - i.e path through function can end without a return).

Function Call

$$< Fun\ Name > (< Expr >, \dots)$$

- **Checking arguments are parameter compatible**
  Check the expressions provided are valid expressions, and if so that they can be matched with the type of the parameters.
- **Checking the Return Type**
  The expression will evaluate to the type of the function call. Hence if there is no return type (e.g returns *void* in C) this must be considered (e.g would be fine in a call statement, but not as part of an integer expression).

Type Checking in Languages

Type systems highly vary between languages:

- **Static vs Dynamic Typing** If types are checked at compile time, vs at runtime.
- **Strong Vs Weak Typing** If types can be cast and coerced easily.
  For example Python is dynamically and strongly typed, so when type checking occurs (at runtime) types must be explicity converted.

  However C is statically weak, meaning types are checked at compile time, but can be easily cast and converted (e.g cast any pointer type to any other pointer type)
- **Type Inference**
  Some languages can use values provided, functions used, return types etc to infer the type of a variable at compile time without it being explicitly stated.

  An example of this is Rust which has one of the most powerful such systems.
- **Function Overloading** Multiple different functions with the same name in the same scope. If function signatures differ, the compiler can determine which function to use in calls.

  In statically typed languages this is simpler (as types are known), some dynamically typed languages also support this.
  Python allows overloading, but the number of arguments must differ (as the type system is not yet powerful enoguh to distinguish between identical signatures).

  Despite its powerful type system, Rust avoids this (a language design choice) as overloading is considered unhelpful & to potentially make code more difficult to discipher by programmers.
- **Polymorphic Typing** Some languages allow for types to be based on interfaces, parent classes, etc. Another form is generics.
- **Assignment-Compatability and Type-Equivalence** Differs between languages, rules concering how types are matched, and what types can be matched
- **Type Coercion/casting** Some languages allow implicit casting (e.g C) while others require explicit (e.g Rust, Python).

- **Primitive Sizes** Basic data types such as integers, floats are implemented differently between languages.
  For example some languages allow for arbitrary size integers as a language feature (e.g Python). This has negative performance implications, so high performance languiages such as Rust and C use fixed size integers (as represented in the architectures they compile to).

Symbol Tables

Lecture Recording

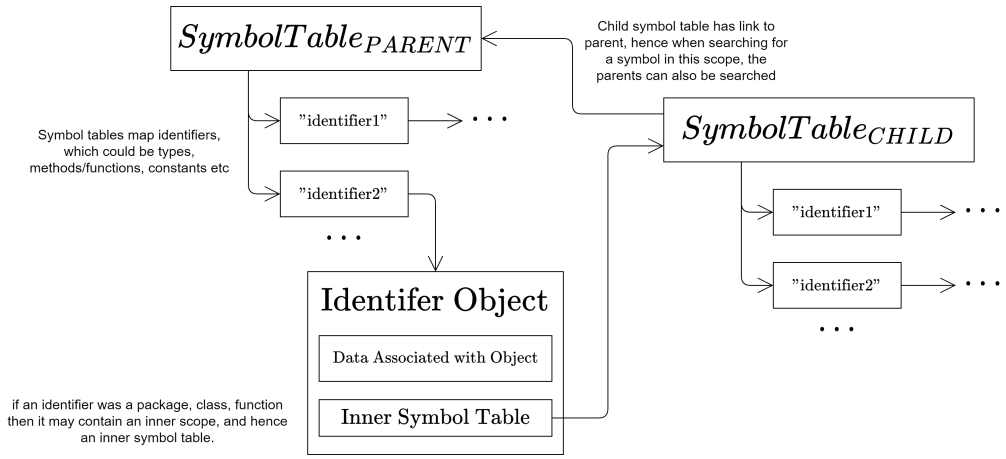
Lecture recording is available here

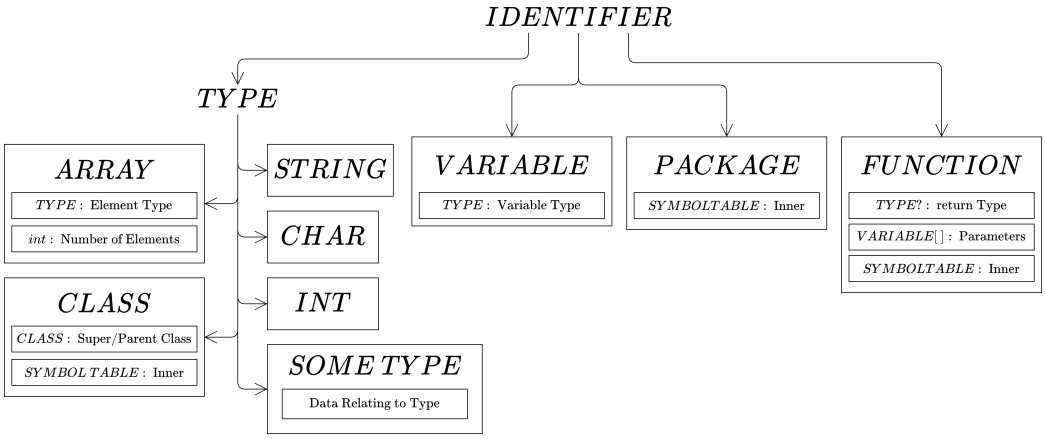
Definition: Symbol Table/Identifier Table

Rather than store semantic information for identifiers in AST nodes (which would require lengthly lookup during code generation) a structure is used to identify identifies with semantic information.

- For scoping a tree of symbol tables can be used. Alternatively a flat symbol table can be generated with identifiers being translated to some non-shadowed form (no name conflicts).
- In many languages identifiers are declared before use. This allows a single pass to build the symbol table.
- Map data structures are used for the tables to reduce the time required for identifier lookup.

Example Symbol Table Implementation





Lecture Recording

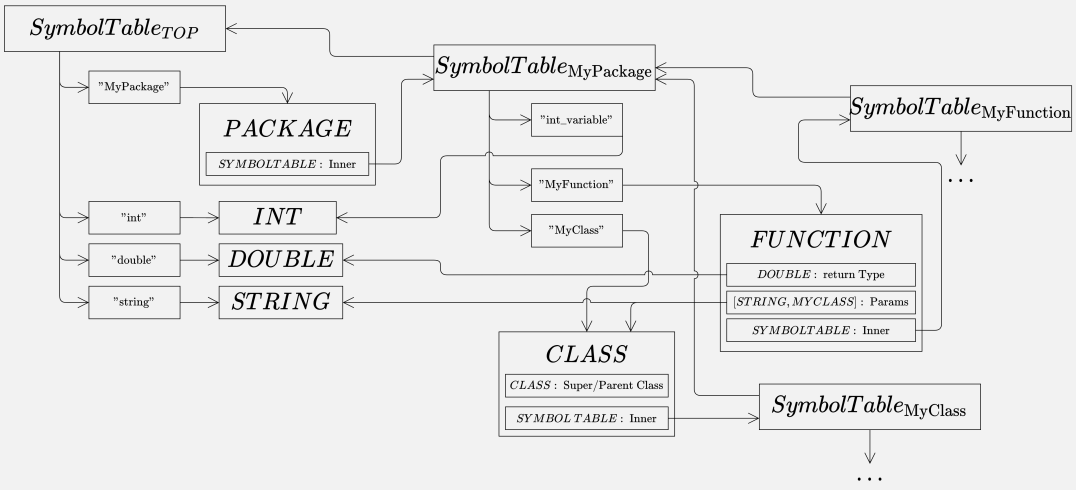
Lecture recording is available here

We can embed pointers the identifier objects inside the AST also.

When checking types, we can use ad-hoc rules for coercion (e.g can assign an *int* to a *double*). We may also want to use the class hierarchy (if no match, check if any super classes can match)

Example: Basic Package

```
1 package MyPackage {
2     int int_variable;
3     class MyClass { ... }
4     double MyFunction(String param_1, MyClass param_2) { ... }
5 }
```



We can then put our basic types, standard types, functions etc (default includes) in the top level symbol table.

50006 - Compilers - (Dr Dulay) Lecture 5

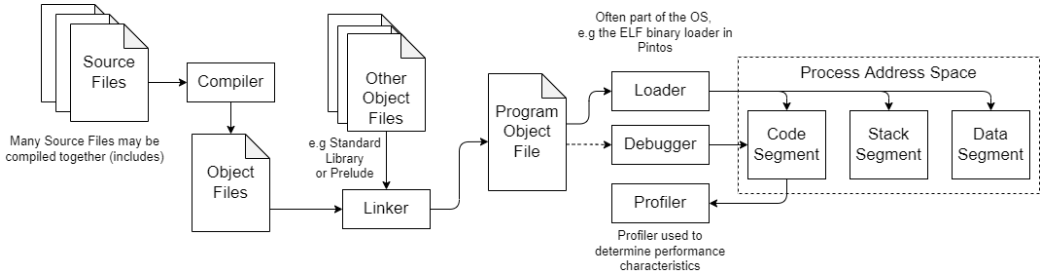
Oliver Killane

14/04/22

Lecture Recording

Lecture recording is available here

Classic Approach to Compilers



Data Representation

Definition: Primitive Types

The most basic types, supported by the architecture the compiler targets.

Type	Bytes	Representation
Boolean	1	0 for false and 1 for true.
Integer	1,2,4,8	Typically 2s-complement.
Unsigned Integer	1,2,4,8	Basic unsigned binary.
Float/Real	4,8,16	IEEE Floating Point.
Char	1,2	For ascii (byte) and unicode.

- Compilers may align types for more optimal memory access, sometimes this is enforced by the target architecture.
- Some languages support pointer types (e.g C), however this makes type checking difficult (consider pointer arithmetic). As a result more modern language use type-checkable object references.

Definition: Alignment

Typically some address is considered n-byte aligned when it is a multiple of n bytes.

- Often data is aligned by the size of data the processor can load in a single instruction. This is done to ensure loading some data requires the minimal number of memory accesses.
- Some architectures enforce alignment for this reason, particularly for the stack.
- When aligned to a multiple of two 2^k, the first k bits of the address will be zero. For example alignment of pages in memory allow for page table entries to use the bits that will always be zero (due to alignment) to be used for other information (e.g reference, supervisor, read/write)

Definition: Structs/Records

Data types consisting of groups of fields/members of other types.

- Fileds typically allocated in a contiguous block of memory.
- Alignment often used to space fields.
- Some languages order fields by their position in code (e.g C) while others can reorder and optimise.
- Tuples are effectively anonymous structs.

If no padding/alignment is used, and the struct/record is kept in order as expressed by the source code (e.g typical of C):

$$\begin{aligned} Record &= (Field_1, Field_2, \ldots, Field_n) \\ Size(Record) &= Size(Field_1) + Size(Field_2) + \cdots + Size(Field_n) \\ Address(Field_k) &= StartAddress(Record) + Size(Field_1) + \cdots + Size(Field_{k-1}) \end{aligned}$$

Struct Representation

As previously mentioned some languages reorder and optimise structs/records.

Rust not only does this, but allows the programmer to specify different representation schemes through macros. (See Rust alternative representations.)

Definition: Arrays

A contiguous section of memory populated by n variables (elements) of the same type.

- Can have elements aligned
- Some languages associated arrays with auxiliary data (e.g length for bounds checking).

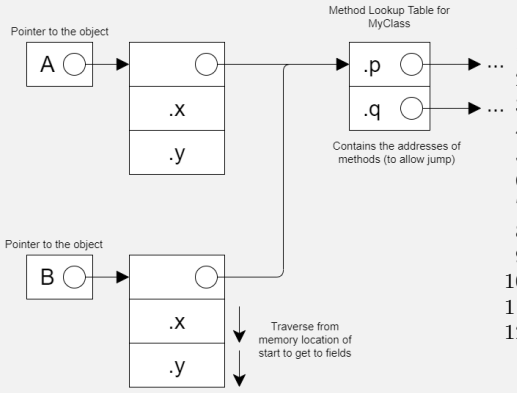
A basic array implementation (elements not aligned/padded):

$$Array[Type] = Element[Type]_1, \ldots, Element[Type]_n \text{ (Indexed as 0 to } n-1)$$
$$Size(Array) = Size(Type) \times n$$
$$Address(Element_k) = StartAddress(Array) + k \times Size(Type)$$

Definition: Objects

Implemented as a reference to a record, but with a pointer to a **method lookup table**(**MLT**) for that class (needs to consider inheritance).

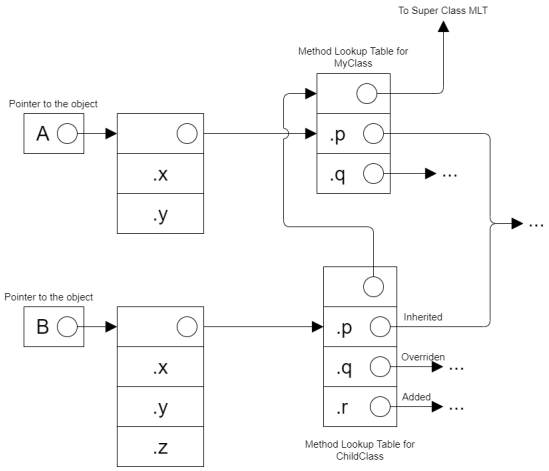
- When calling a method of an object, traverse to the object's **MLT**. Then jump to the method, having placed the first argument as a pointer to the object.
- Access fields just like a struct.
- The indirection required for accessing values, calling methods adds overhead.




```
1  class MyClass {
2      int x;
3      int y;
4
5      method ... p(...) {...}
6      method ... q(...) {...}
7  }
8
9  MyClass A = new MyClass();
10 MyClass B = new MyClass();
11
12 B.q(...) // MyClass.q(&B, ...)
```

Class Inheritance

For single inheritance we include the parent's fields and methods in the struct and **MLT**.




```
1  class MyClass {
2      int x;
3      int y;
4
5      method ... p(...) {...}
6      method ... q(...) {...}
7  }
8
9  class ChildClass {
10     int z;
11
12     override method ... q(...) {...}
13
14     method ... r(...) {...}
15 }
16
17 MyClass A = new MyClass();
18 MyClass B = new ChildClass();
```

- Pointers in the **MLT** go to the method (potentially a method also pointed to by the parent if no overriden)
- New added methods are in the **MLT**
- We chain the child/subclass's **MLT** to the parent/superclass' so that we can check for types (e.g is this an *instanceof* another class, a child etc) at runtime using the **MLT**. We could also have used type descriptors or other implementations.
- We preserve the layout of the parent/super class in the child (only appending methods and fields).

Dynamic Binding

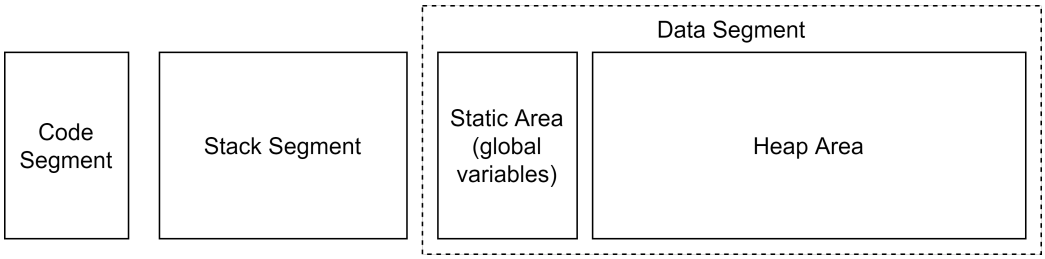
As we only append fields and entries to the **MLT** when inheriting, it is possible to set a variable of type parent class to a value of type child class.

In the example below, we can see that a new field is added in B, (the inherited $A.n$ is still available). The **MLT** of the B class contains the overriden *get* function.

```
1  public class OOPEXample {
2      public static void main(String args[]) {
3          A a = new B();
4          // As get returns the class's n, and the get method is B's we get 2.
5          // However a.n refers to A.n, which is 1. The field n is inherited from
6          // A, B adds another field called n.
7
8          // Outputs: "a = new B(); a.get() -> 2, a.n -> 1"
9          System.out.printf("A a = new B(); a.get() -> %d, a.n -> %d", a.get(), a.n);
10     }
11 }
12
13 /**
14  * A:
15  * -> MLT: A.get (defined)
16  * - int A.n     (defined)
17  */
```

```
18 class A {
19     public int n = 1;
20     public int get() { return n; }
21 }
22
23 /**
24  * A:
25  * -> MLT: B.get (override)
26  * - int A.n     (inherited)
27  * - int B.n     (added)
28  */
29 class B extends A {
30     public int n = 2;
31     public int get() { return n; }
32 }
```

Program Address Space



Typically programs divide their address space into several segments (each potentially multiple pages large).

- By placing static data, code and stack space in different pages (in different segments/parts of the address space) they can have different permissions (e.g code is read-execute only, stack cannot be executed, etc)
- Global variables can be placed in a fixed location in the static area.
- Constants, **MLT**s and other global data can be kept in the static area.
- Local variables are stored on the stack in the stack frame of the function they are local to. They are allocated for the duration of the call.
- The stack can also contain arguments sent to the function, as well as information such as the address of the object (for methods) and slots for temporarily storing registers.
- Some architectures such as IA32 have a special register for the start address of the current stack frame (in IA32 it is the EBP register).

Heap Management

Lecture Recording

Lecture recording is available here

We can heap allocate variables (heap variables, also called dynamic variables). Here they can remain for any period of time, and be referenced from anywhere within the program (unlike local variables which are bound to a given function).

Heap variables must be allocated (e.g by *new* in java, or *malloc* in C/C++)

Programming languages manage memory in 4 different ways:

Explicit	C, ASM	Programmer determines when memory is freed, and how it is accessed. This is difficult, but allows for very fast code as the programmer can highly optimize their memory usage.
Garbage Collection	Java, Haskell, Go	The compiler adds code to check what objects are no longer in use, and to free them. This is done at runtime and has considerable overhead.

Rust Superiority

Rust's main memory management tool is Lifetimes (though it does use smart pointers with reference counting, and allow for explicit memory management also).

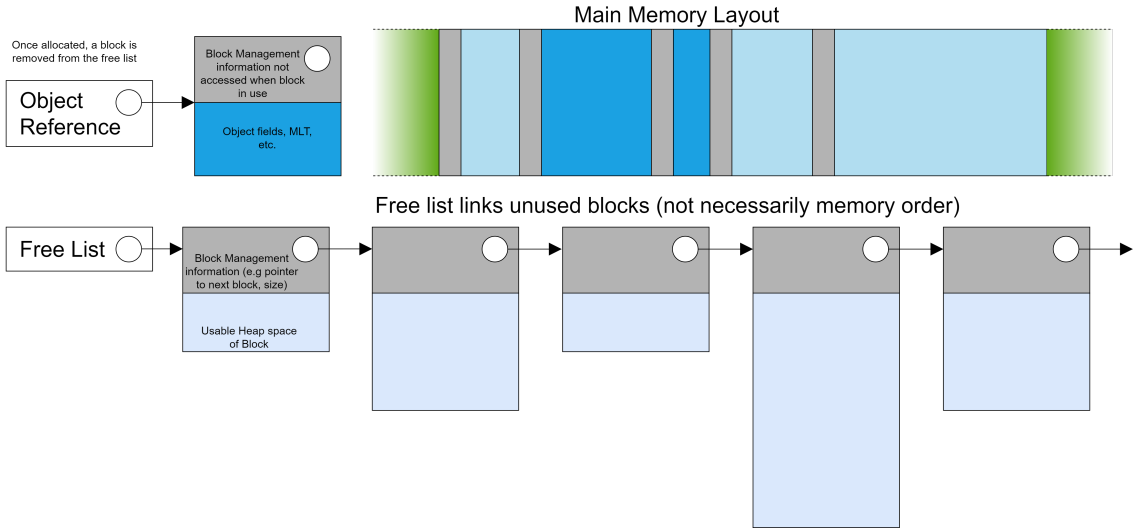
All references are qualified with information on mutability, and where the reference will be valid. The compiler uses these to ensure memory safety and insert required frees at compile time.

This makes the compiler more complex, but is safer than a GC (fewer bugs, will refuse to compile unsafe code (e.g with data races)), with performance almost identical to explicit memory management.

Explicit Heap Allocation

We maintain a structure containing the free blocks and information associated with each block. Here we will consider a simple linked list.

- Maintaining several lists for different sizes can reduce the search time.
- Should allocate blocks just large enough.
- For extensible arrays (vectors, mutable strings) we should allocate with the expectation that more memory will be used (allocate more than currently required).
- Some architectures require alignment of values, this must be considered. Requirements aside, as previously mentioned not-aligning can reduce performance.
- Heap allocation is worse for locality (e.g than the stack) and hence can potentially have worse cache locality, affecting performance.




```
1  def allocate(size: int) -> Address:
2      # Search through the blocks, can use a better algorithm:
3      block = free_blocks.search(size)
4      if block.size == size:
5          # found exactly the size needed
6          free_blocks.remove(block)
7          return block.start_address
8      elif block.size > size:
9          # split the block
10         (size_block, other) = free_blocks.split(size)
11         free_blocks.remove(size_block)
12         return block.start_address
13     else:
14         # we need more memory
15         if os.give_me_more_mem():
16            # try again with allocation
17         else:
18             # OS could not provide
19             return NULL
20
21 # Can be implemented to takes the block size as a parameter also
22 def deallocate(ptr: Address):
23     # Check free is valid
24     if free_blocks.check(address):
25         # return block to free blocks, coalesce/merge with another block to
26         # reduce fragmentation
27         free_blocks.insert_and_coalesce(ptr)
```

Garbage Collection

Definition: Garbage Collector

A part of the language's runtime that ensures heap allocated variables are properly de-allocated when they are no longer used. Many different **GC** algorithms are employed by many different languages.

GCs have several requirements:

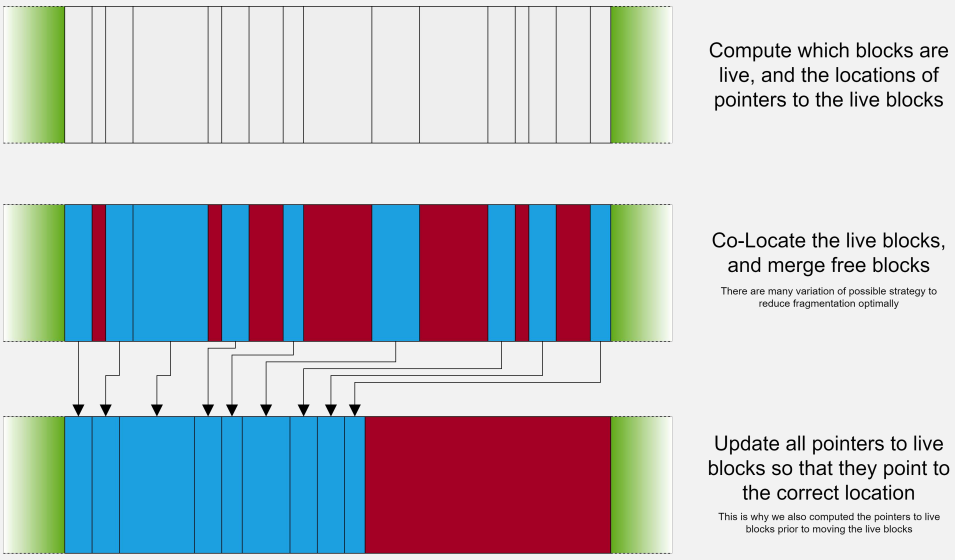
- **Correctness** The **GC** must never collect (de-allocate) live data (in use).
- **Performance** Must be fast & have low memory overhead to reduce the impact on program performance.
- **Compiler Supported** In order to function the compiler needs to provide the garbage collector information for:
  - Which variables point to the heap.
  - The pointers contained in each block (e.g an object contains a reference to an object).

  The compiler can either provide this data directly (type description data for objects), or generate special subroutines for the **GC** to call.

Garbage collectors can be run as **1-shot** (pause the program, run the **GC** and then resume), **on-the-fly** (as the program is run) or concurrent (runs on a different thread).

Definition: Heap Compaction

After **GC** de-allocates many blocks, the heap may be fragmented (lots of small free blocks). Hence much memory may not be free, but it may not be possible to allocate for an object as there is no block large enough.



Definition: Reference-Counting Garbage Collector

The block's management/housekeeping information contains a **reference count**. When a reference is made to an object, the reference count of the block containing it is increased, when that reference is removed, this is decreased.

When the reference count is decremented to zero, we know to free the block.

- Simple and efficient (garbage collection done as the program runs)
- Requires the compiler to generate the code to correctly track references.
- In reference cycles (e.g A references B which references A, even when no-one references them (they should both be collected) they keep eachother alive) blocks may be kept live indefinitely, the **GC** needs to avoid this.

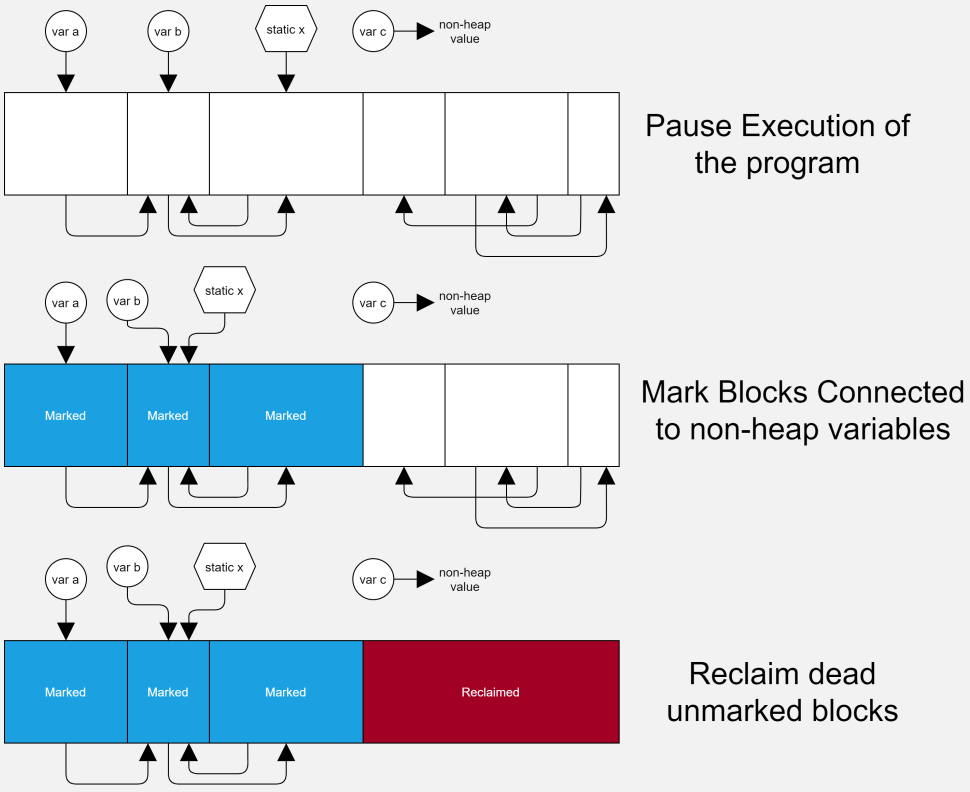
```
1  # used for assigments such as a = b
2  def assign(lval, rval):
3      if rval.on_heap():
4          rval.references += 1
5
6      if lval.on_heap():
7          lval.references -= 1
8          if lval.references == 0:
9              gc.reclaim(lval.block)
10
11     lval.value = rval.value
```

Definition: Mark-Sweep Garbage Collector

Pause the program, and collect all blocks that are not pointed to (potentially indirectly) by a non-heap value (e.g local or static variable).

This will find all dead blocks/garbage (unlike a **reference counting GC**) however it has considerable overhead. Each block's maintenance/housekeeping information must also contain space for a mark bit.

- **Mark** For all stack variables, if they point to the heap then recursively traverse, marking each block they point to as live.
- **Sweep** Go through each block, if it is not marked as live, then collect and deallocate it.



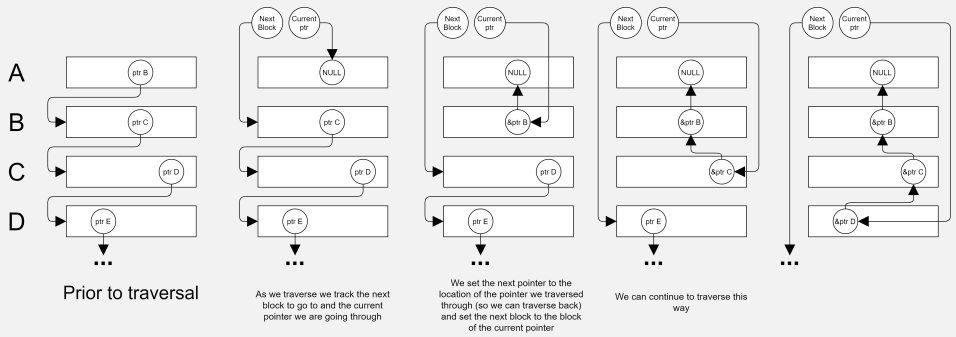
Definition: Pointer-Reversal Marking

Recursive traversal of pointers requires a potentially large amount of stack space (e.g going through a linked list of throusands of elements).

A garbage collector needs to work when memory is low. Hence we can use **Pointer-Reversal Marking** to store the pointers to return to (previous in traversal) in place.

1. When visiting a block, we select a pointer from within the block. We then set that pointer as the current, and place the address of the previous current pointer into it (hence now reversed).
2. We can then use the current pointer value to traverse to the next block, and continue.
3. When we find an already marked block, or no pointer in the block top traverse to, we can retreat back by using the pointer held in the location of the "current pointer" as this points back to the parent. We can set this
4. Note we need to store the number of marked children in the block's maintenance/-housekeeping data.

By storing the previously visited in the pointer we are traversing through, we require no extra stack space.




```c
// Variables used to keep track of location
void **current_ptr;
void **next_ptr;
void* current_block;

// traverse downwards
next_ptr = find_ptr_in_block(current_block);

current_block = *next_ptr; // current_block = address of next block
*next_ptr = current_ptr; // next_ptr now contains the address of where the old current_ptr was stored
current_ptr = next_ptr;

// we can mark the block we are current at
mark_block(current_block);

// traversing upwards (assuming no next_ptr)
next_ptr = *current_ptr; // the next pointer is now previous (traversing back)
*current_ptr = current_block; // set the current ptr back to being the next block
current_ptr = next_ptr; // current pointer moves to the previous
current_block = block_from_ptr(*current_ptr); //get the block we are now in
```

Definition: Two Space Garbage Collector

The heap is split into two, a **from-space** and a **two-space**. Blocks are allocated from the **from-space**. When there are no more free blocks, all live (reachable from a non-heap pointer) blocks are copied to the **to-space** and then the **to-space** becomes the **from-space** and vice-versa.

- Fast, there are few complex pointer manipulations.
- Automatically compacts memory when copying.
- Can place linked blocks close together in memory when copying to allow for better cache locality.

Definition: Generational Garbage Collector

The heap is split into several areas based on the age of blocks.

- Allocate new blocks from the youngest, if full, move the oldest young blocks to an older area.
- **GC** is used more frequently on the younger areas.
- Can apply different **GC** techniques to the different areas.

Java 13

Java 13 has 5 different garbage collectors, you can read more about there here.

- Serial
- Parallel
- Concurrent Mark-Sweep (CMS)
- Garbage First (G1),
- ZGC

Other Compiler Components

Definition: Debugger

Useful tools for inspecting the behaviour of programs.

- Program behaviour must be maintained when being debugged (execution time can change, but correctness must be consistent).
- Debugging information (e.g function and variable names, source line mappings) may be embedded in the program.
- Highly optimising compilers and debuggers are mostly incompatible (as a useful debugger can map behaviour back to source code, and no such relation exists once complex optimisations are done). Hence debuggers usually compile with optimisations off.

There are two main kinds of debuggers:

- **Interactive** e.g GDB Allow users to inspect the state of the program, variables, functions, memory etc and to pause execution of the program.

  For natively compiled (e.g Rust, C, Go) the architecture and OS must support breakpoints. For bytecode compiled (e.g Java) the interpreter supports this.
- **Post-Mortem / Core Dump** e.g Rust backtrace Provides debugging information upon failure by doing a reverse lookup from the program counter.

  – Stack TraceBack shows the methods called, their local variables, arguments etc.
  – Contents of global and dynamic/heap variables.

  This requires the debugger to work out where variables are stored (register, stack, heap) and to map them back to names from the source using debugging information embedded at compile time.

Definition: Profilers

Provide performance information on a program.

- Typically used once code is correct, algorithmically optimal and optimised by the compiler.
- Can determine the time spent in functions, or how much of a program is spent is certain sections of code.

For example a profiler could interrupt execution periodically (some number of ms), identify the method being used, and increment a counter for it. Hence after profiling we can get a breakdown of the proportion of program time spent in each function.

50006 - Compilers - (Prof Kelly) Lecture 1

Oliver Killane

05/01/22

Lecture Recording

Lecture recording is available here

Course Introduction

Lecturers

- **Paul Kelly**
  - Introduction to syntax analysis & toy compiler
  - Code Generation
  - Generating better code using registers
  - Register Allocation
  - Optimisation and data flow analysis
  - Loop optimisations
- **Naranker Dulay**
  - Lexical Analysis (characters to tokens)
  - LR (Bottom-up) parsers (tokens to AST)
  - LL (top-down parsing)
  - Semantic Analysis, checking program validity
  - Runtime organisation (memory)

Materials

- Scientia
- Course Textbook
- Course Webpages for Paul Kelly and Naranker Dulay

Recommended Textbooks:

- **The Dragon Book: 'Compilers: Principles, Techniques and Tools'**
  The definitive book on compilers. Called the dragon book due to its front illustration (allusion to taming the dragon of compiler complexity).

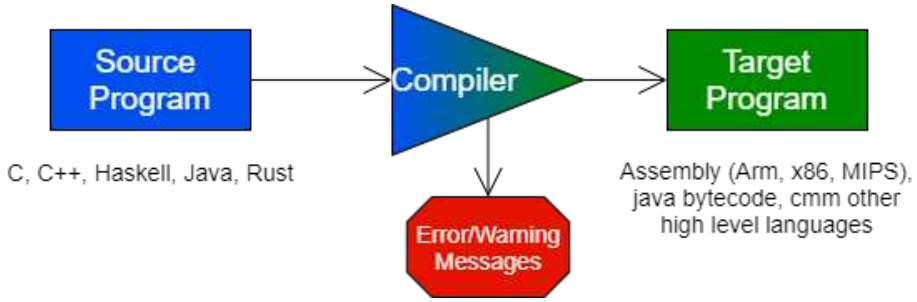
  Has all required information and lots more (very thick book). (Amazon link)
- **'Modern Compiler Implementation in Java'**
  Details a java compiler implementation, with reasoning behind design choiced. (Amazon link)
- **Most Recommended: 'Engineering a Compiler'**
  Most closely follows course & well reviewed. (Amazon link)

Compilers Introduction

A particular class of program called **language processors**. Compilers can be:

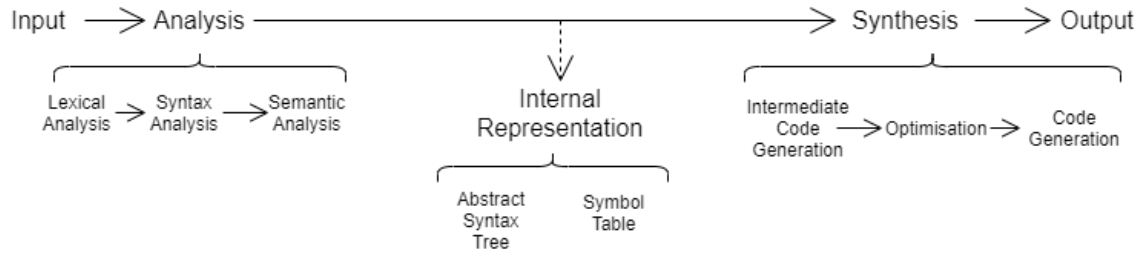
- **Processes** programs written in some language.
- **Writes** programs in some language.
- **Translates** programs from one language to an equivalent program in another. (e.g from hier to lower level language)

A tool to enable programs written using high-level concepts to be translated into a low-level implementation.



- Typically from a high level language to a lower level one.
- Error messages ensure source program can be fixed easily when failing to compile.
- Can compiler from & to a high-level language (e.g compile to C, or between languages)
- Analyse the source program to provide the programmer useful information.

Basic Compiler Structure



- **Input** Take program written in some language.
- **Analysis**
  Create an internal representation (IR) of the source which encodes its meaning (semantics).

  This often starts with a tree (abstract syntax tree), but graphs may be used to represent control flow.

  - **Lexical Analysis**
    Tokenization, converting text to a series of tokens representing keywords, identifiers, etc.
  - **Syntax Analysis**
    Analysis of the nested structure of a program (e.g if-else within if-else within loop block).
  - **Semantic Analysis**
    Analysing the meaning of a program such as type checking, determining overloading (e.g is + between variables correct & what action (e.g add integers, or concantenate strings)).

- **Synthesis**
  Use the IR to create the program in the target language (retaining the same meaning/semantics).

  The IR may be analysed and transformed to provide extra information to the programmer

(e.g warnings) as well as to optimise (e.g inlining, loop hoisting, loop unrolling, removing dead code).

  - **Intermediate Code Generation**
    Encodes more information, and is used in sphisticated compilers for optimisation. For example determinign loop invariant code (code that can be hoisted out of the loop & only be evaluated once).
  - **Optimisation**
    Improving code performance without altering its meaning.

- **Output** Output the target program. (e.g x86 assembly or Java bytecode)

Symbol Table

Contains information on all declared identifiers, and is used to check its use (e.g type checking) and generate code for access.

- Name
- Type of identifier (function, class, variable, constant)
- Size (Memory to reserve & where (e.g stack, data segment, heap))

Code

```c
#include <stdio.h>
#include <stdlib.h>


int a = 6;

void pretty_print(int num) {
    printf("The number is: %d", num);
}

int another_cool_function(int* num) {
    *num++;
    return *num - 1;
}

int main(int argc, char **argv) {
    int b = 7;

    pretty_print(a + b);

    return EXIT_SUCCESS;
}
```

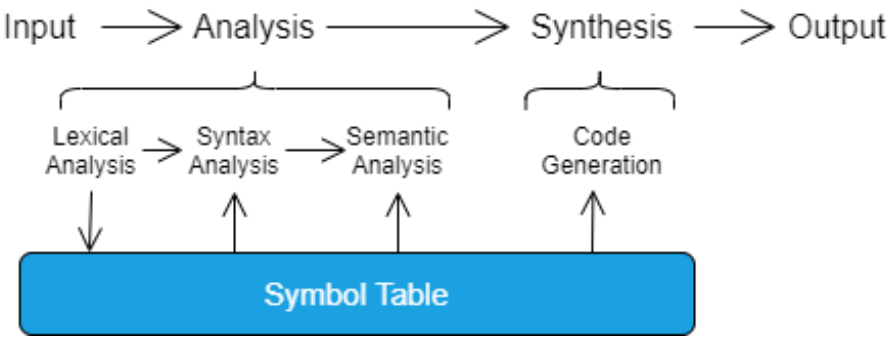
GCC symbol table

```
0000000000003dc8 d _DYNAMIC
0000000000003fb8 d _GLOBAL_OFFSET_TABLE_
0000000000002000 R _IO_stdin_used
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
00000000000021b4 r __FRAME_END__
0000000000002018 r __GNU_EH_FRAME_HDR
0000000000004018 D __TMC_END__
0000000000004014 B __bss_start
                 w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000001100 t __do_global_dtors_aux
0000000000003dc0 d
    ↪ __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003db8 d
    ↪ __frame_dummy_init_array_entry
                 w __gmon_start__
0000000000003dc0 d __init_array_end
0000000000003db8 d __init_array_start
0000000000001240 T __libc_csu_fini
00000000000011d0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2
                     ↪ .2.5
0000000000004014 D _edata
0000000000004018 B _end
0000000000001248 T _fini
0000000000001000 t _init
0000000000001060 T _start
0000000000004010 D a
0000000000001171 T another_cool_function
0000000000004014 b completed.8060
0000000000004000 W data_start
0000000000001090 t deregister_tm_clones
0000000000001140 t frame_dummy
0000000000001194 T main
0000000000001149 T pretty_print
                 U printf@@GLIBC_2.2.5
00000000000010c0 t register_tm_clones
```

GCC Symbol Table

Simply compile, and then use **nm** on the resulting binary to get a basic representation of GCC's symbol table (that is included with the executable).

The symbol table is used in different ways through different phases of the compiler:



Code

```
1 int a;
2 int b;
3
4 void test_fun(void) {
5     a = b+42;
6 }
```

Assembly

```
1 test_fun:
2         movl    b(%rip), %eax
3         addl    $42, %eax
4         movl    %eax, a(%rip)
5         ret
6 b:
7         .zero   4
8 a:
9         .zero   4
```

Boilerplate

Not lots of boilerplate assembly is omitted, if you try this locally you will see this.

GodBolt

The compiler explorer is a great tool for generating & exploring assembly generated by compilers.

Compiler Phases

We can show the phases of the compiler through an example. Converting **C** code into **AT&T/GAS** assembly.

$int_{space}a;_{newline}\, int_{space}b;_{newline}\ _{newline}void_{space}text\_fun(void)_{space}\{_{newline}a_{space} =_{space} b{+}42;_{newline}\}$

[INTtok,
IDENTtok(pointer to 'a' entry in symbol table),
SEMICOLONtok,
INTtok,
IDENTtok(pointer to 'b' entry in symbol table),
SEMICOLONtok,
VOIDtok,
IDENTtok(pointer to 'test_fun' entry in symbol table),
LBRACKETtok,
VOIDtok,
RBRACKETtok,
LCURLYBRACKETtok,
IDENTtok(pointer to 'a' entry in symbol table),

EQtok,
IDENTtok(pointer to 'b' entry in symbol table),
PLUStok,
CONSTINTtok(123),
SEMICOLONtok,
RCURLYBRACKETtok]

Lexical Analysis

Also known as tokenization. The input sequence of characters is tokenized and any new identifiers added to the symbol table during tokenization.

Tokens to identifiers include pointers to their corresponding entry in the symbol table.

Syntax Analysis

Also known as parsing. Using a grammatical structure to generate an **Abstract Syntax Tree** from the sequence of tokens provided by the previous stage.

1. Ensure the structure is correct, report an error otherwise.
2. Build **Abstract Syntax Tree** from tokens to represent program.

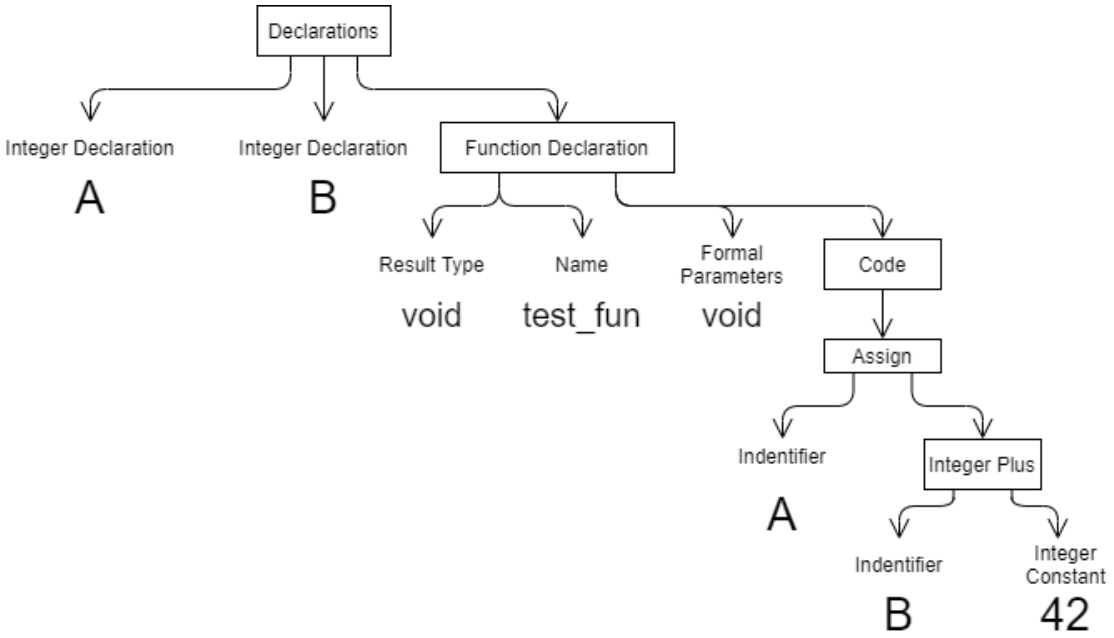
The grammar of the language can be expressed in a notation such as **Backus-Naur Form**. This is much the same as the definitions used for the simple while language in **Models of Computation**.

A rudimentary example could be:

$$\begin{array}{rcl} Cond & ::= & 'true' \mid 'false' \mid Expr == Expr \mid Expr < Expr \mid Cond \& Cond \mid \neg Cond \ldots \\ Expr & ::= & Var \mid Const \mid Expr + Expr \mid Expr \times Expr \ldots \\ Stat & ::= & x := Expr \mid 'if'\ Cond\ 'then'\ Stat\ 'else'\ Stat \mid Stat;\ Stat \mid 'skip' \mid 'while'\ Cond\ 'do'\ Stat \end{array}$$

The **Abstract Syntax Tree** is implemented as a tree of linked objects. It must be designed to be efficient to construct & contain all require information for subsequent stages.

The **Abstract Syntax Tree** for the basic C program would be something like:



Domain Specific Languages

Many programming languages are **domain specific**, meaning they are designed for a specific use.

- **Latex** Creating documents
- **Markdown** Simple text files with image, font support (e.g readmes)
- **YAML & TOML** Configuration files
- **Verilog** Digital Circuit Design
- **R** Statistics
- **Simulink** Dynamic System Modelling
- **Yacc** (Yet another compiler-compiler) parser generator language

There are also many language processors that are not compilers:

- **FindBugs** Finds bugs in java code
- **Pylint** Linter for python
- **Valgrind** Detecting bugs & memory leaks

50006 - Compilers - (Prof Kelly) Lecture 2

Oliver Killane

05/01/22

Lecture Recording

Lecture recording is available here

Syntax Analysis

- **Syntax** The grammatrical structure of the language expressed through rules.
  The compiler must determine if the program is syntactically correct.

  Parser Generators tools used to generate the code to perform the analysis phases of a compiler from the language's formal specification (usually similar to **Bakus-Naur Form**).

- **Semantics** meaning associated with program.
  For example type-checking, or checking for memory safety.

  Compiler Generators/Compilers are an active area of research. They generate the synthesis phase from a specification of the semantics of the source & target language.

  These tools are promising but usually the code is written manually instead.

Bakus-Naur Form

Also called **Backus Normal Form** is a context-free grammar used to specify the syntactic structure of a language.

$$stat \rightarrow \text{'if'} \ \text{'('} \ expr \ \text{')'} \ stat \ \text{'else'} \ stat$$

- **Context Free Grammar**
  A context free grammar is a set of **Productions**. Associated with a set of tokens (terminals), a set of non-terminals and a start (non-terminal) symbol.

  Each production is of the form:

  $$\text{single non-terminal} \rightarrow \text{String of terminals \& non-terminals}$$

  The simple LHS makes it a context-free grammar, more complex LHSs are possible in context-sensitive grammars.

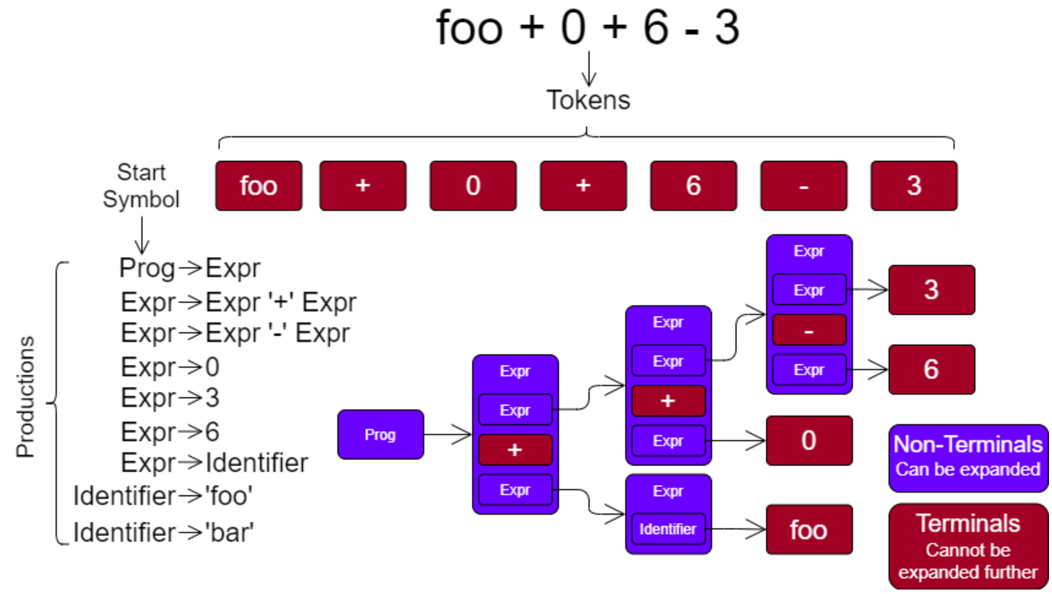
- **Production**
  Shows one valid way to expand a non-terminal symbol into a string of terminals & non-terminals.

  $expr \rightarrow$ '0'
  $expr \rightarrow$ '1'
  $expr \rightarrow$ $expr + expr$
  $expr \rightarrow$ '0' | '1' | $expr + expr$ Can combine two productions for more concise representation.

- **Terminals & Non-Terminals**
  Symbols that cannot be further expanded, these are the tokens generated from lexical analysis (e.g brackets, identifiers, semicolons).

- **Parse Tree**
  Shows how the string is derived from the start symbol.

  This tree is a graphical proof that a given sentence is within the grammar. Parsing is the process of generating this.

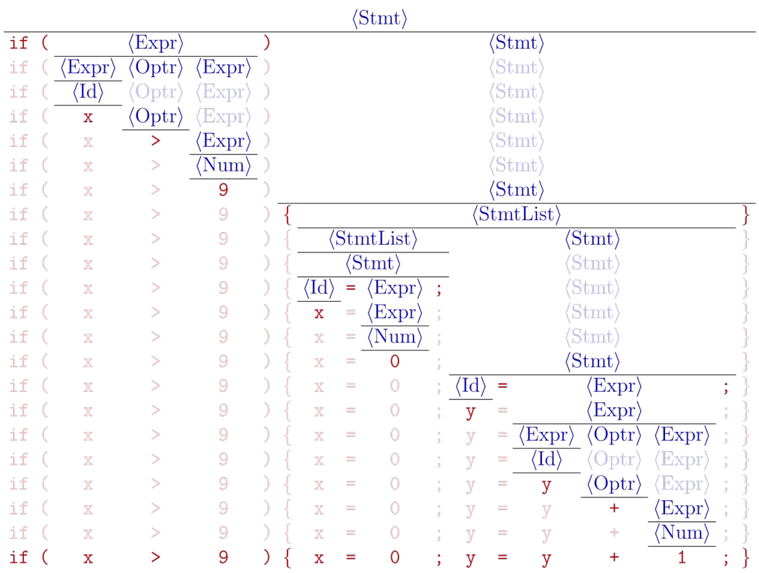


We can express the grammar as a tuple:

$G = (S, P, t, nt)$ where S - start symbol, P - productions, t - terminals, nt - nonterminals, and $S \in nt$

The input is entirely terminals, we use productions & pattern matching to analyse.

⟨Stmt⟩	→	⟨Id⟩ = ⟨Expr⟩ ;
⟨Stmt⟩	→	{ ⟨StmtList⟩ }
⟨Stmt⟩	→	if (⟨Expr⟩) ⟨Stmt⟩
⟨StmtList⟩	→	⟨Stmt⟩
⟨StmtList⟩	→	⟨StmtList⟩ ⟨Stmt⟩
⟨Expr⟩	→	⟨Id⟩
⟨Expr⟩	→	⟨Num⟩
⟨Expr⟩	→	⟨Expr⟩ ⟨Optr⟩ ⟨Expr⟩
⟨Id⟩	→	x
⟨Id⟩	→	y
⟨Num⟩	→	0
⟨Num⟩	→	1
⟨Num⟩	→	9
⟨Optr⟩	→	>
⟨Optr⟩	→	+



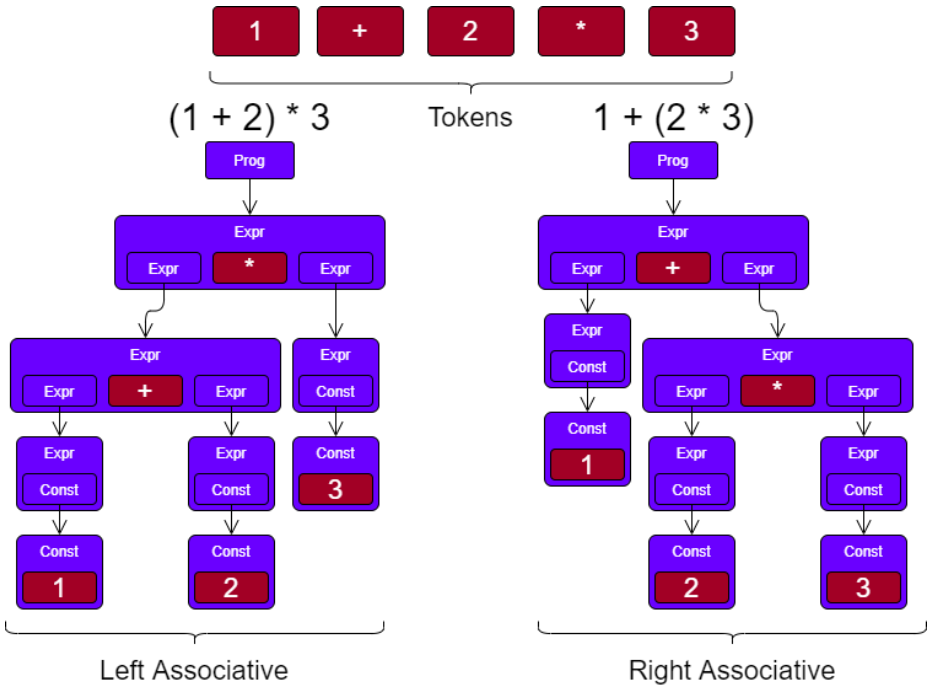
An example with a basic C-style if statement (sourced from wikipedia)

- Starting with the start symbol we can use the productions to replace each non-terminal with some string of terminals and non-terminals, continually expanding the non-terminals.
- A string dervbied that only consists of terminals is a **sentence** (cannot derive any further string of symbols).
- The **language** of a grammar is the set of all sentences that can be derived from the start symbol.

Grammar Ambiguity

In some grammars there may be ambiguity (e.g multiple different productions can be applied to the same string, or the same production in different ways).

For example $3 - 2 - 1$ can be $(3 - 2) - 1$ or $3 - (2 - 1)$. This ambiguity results in multiple possible parse trees.



Often the language designer will specify how to deal with ambiguities (assigning operator precedence & associativity) using the grammar.

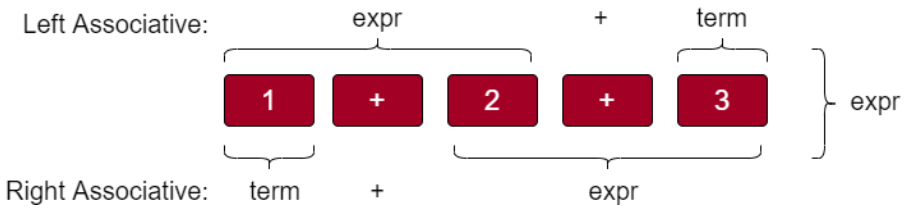
Precedence and Associativity

Precedence determines which operators are applied first, and associativity how operators of the same precedence are applied.

grammar for Associativity

Associativity can be enforced by using left or right recursive productions.

$term \rightarrow$	$const \mid ident$	Define a base term.
$expr \rightarrow$	$expr + term$	Left associative, the split is on the final $+$.
$expr \rightarrow$	$term + expr$	Right associative, the split is on the first $+$.

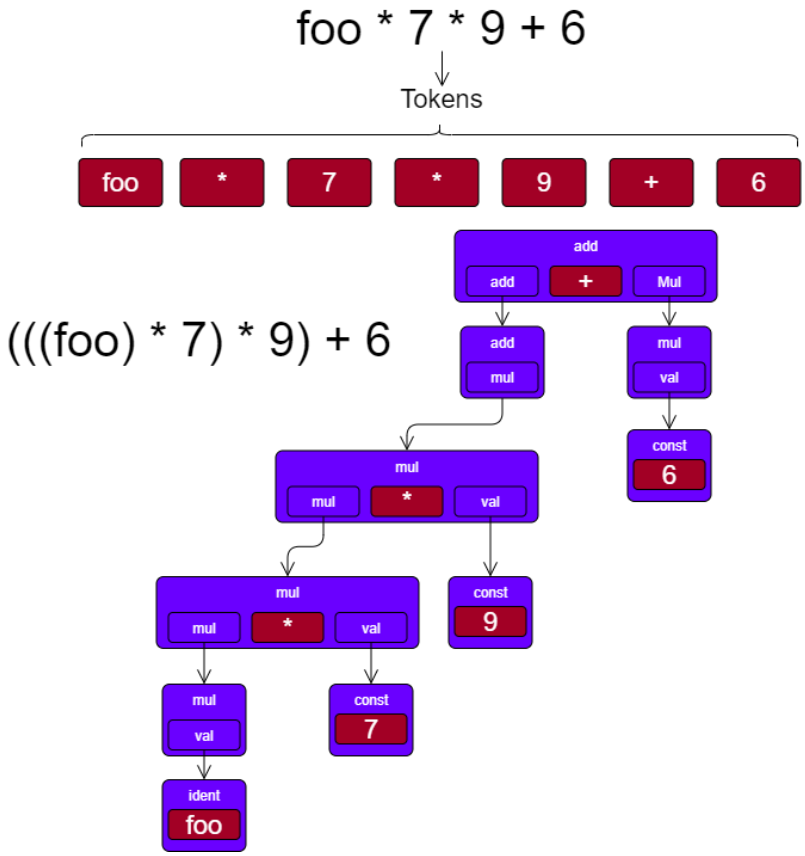


Grammar for Precedence

We can *layer* our grammar such that some symbols are parsed first.

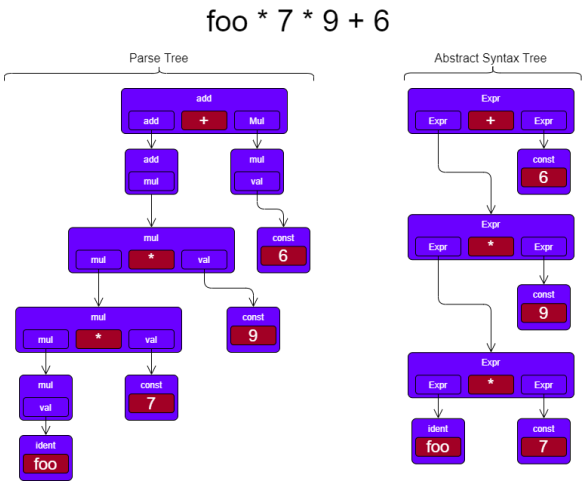
$$\begin{aligned} add &\rightarrow add + mul \mid add - mul \mid mul \\ mul &\rightarrow mul * val \mid mul/val \mid val \\ val &\rightarrow const \mid ident \end{aligned}$$

By splitting the expression into an add and multiply stage (both left associative), the second layer (mul) has higher precedence. To add more levels of precedence we can use more layers.



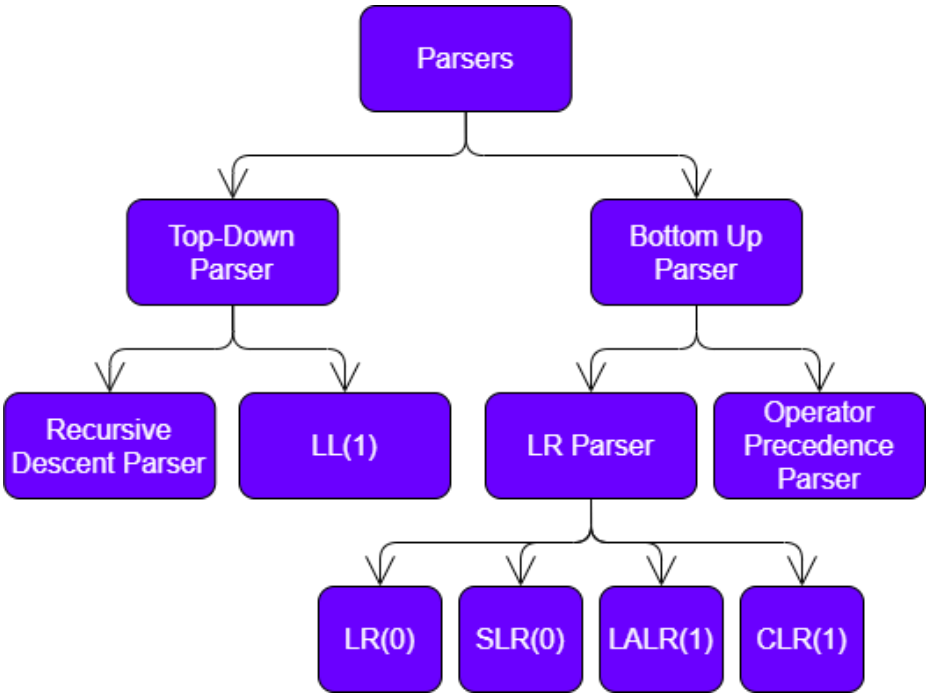
Parse Tree vs Abstract Syntax Tree

The abstract syntax tree has similar structure, but does not need much of the extra information (layers for expressions used to enforce precedence for example).



Parsers

Parsers check the grammar is correct & construct an **AST**.



Top-Down Parsing

Also called predictive parsing.

- Input is derived from a start symbol.
- Parser takes tokens from left → right, each only once.
- **For each step**
  In each step the parser uses:

  – the current token
  – the current non-terminal being derived
  – the current non-terminal's production rules

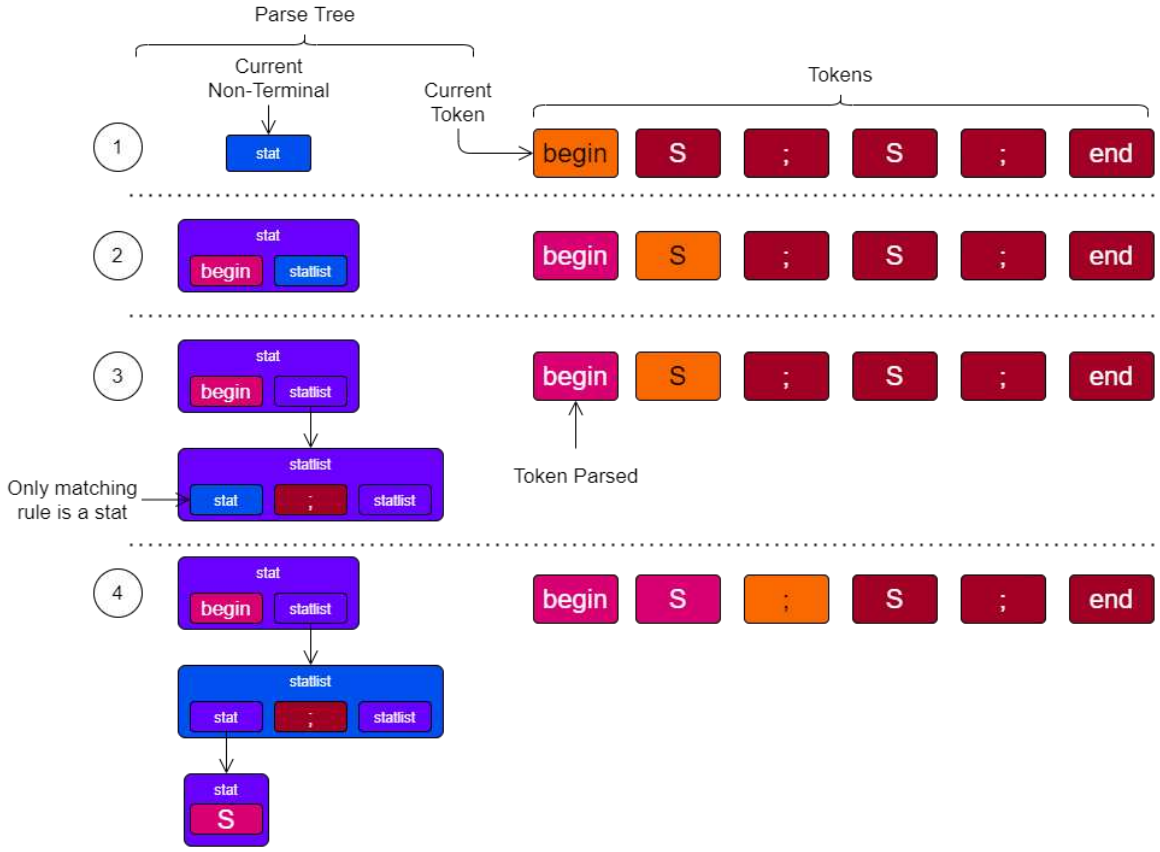
  By using the production rules & the current token we can predict the next production rule, and use this to either:

  1. Get another non-terminal to derive from, and potentially others for subsequent steps
  2. Get a terminal which should match the current token (or else an error has occured/the program is syntactically invalid)
- We are using the grammar *left* → *right*.

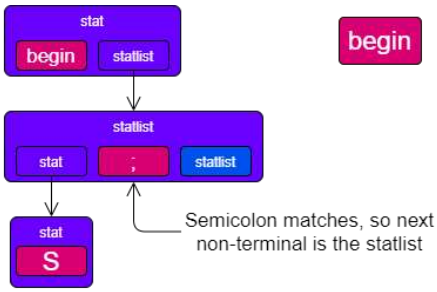
For example with the grammar:

$$
\begin{aligned}
\textit{stat} &\rightarrow \text{'begin'}\ \textit{statlist} \\
\textit{stat} &\rightarrow \text{'S'} \\
\textit{statlist} &\rightarrow \text{'end'} \\
\textit{statlist} &\rightarrow \textit{stat}\ \text{';'}\ \textit{statlist}
\end{aligned}
$$

Start symbol is *stat*.

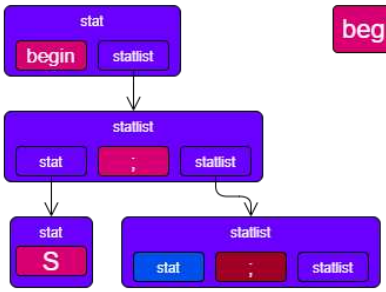


5


stat
begin
statlist
statlist
stat
;
statlist
stat
S
Semicolon matches, so next
non-terminal is the statlist

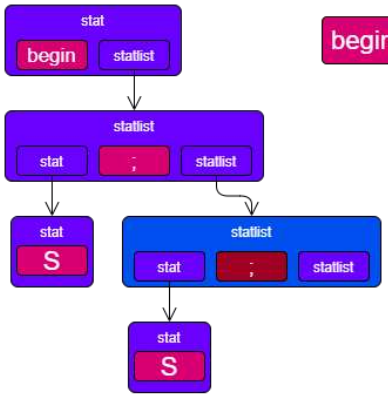
begin
S
;
S
;
end

6


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist

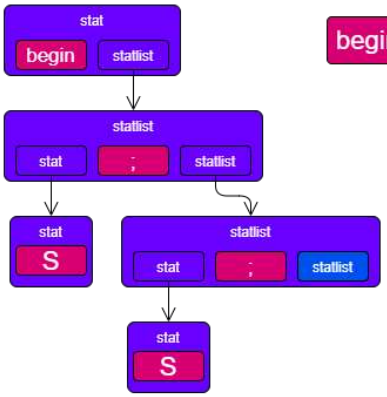
begin
S
;
S
;
end

7


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S

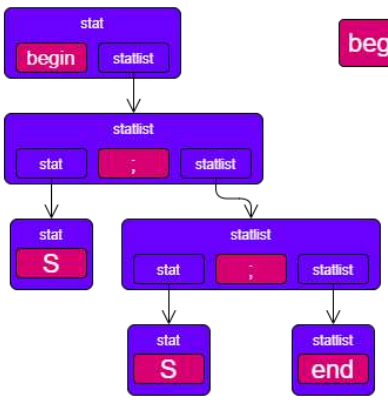
begin
S
;
S
;
end

8


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S

begin
S
;
S
;
end

9


stat
begin
statlist
statlist
stat
;
statlist
stat
S
statlist
stat
;
statlist
stat
S
statlist
end

begin
S
;
S
;
end

Production Choice

We may have a grammar where we cannot determine which production for a non-terminal token to use based on the first symbol.

$$\begin{aligned} stat \rightarrow &\quad \text{'loop'}\ statlist\ \text{'until'}\ expr \\ stat \rightarrow &\quad \text{'loop'}\ statlist\ \text{'while'}\ expr \\ stat \rightarrow &\quad \text{'loop'}\ statlist\ \text{'forever'} \end{aligned}$$

When we have token 'loop' we cannot determine which production to use. There are two methods to deal with this:

- **Delay the choice**
  Delay creating this tree (from stat) until it is known which production matches.

  It is still possible to create the statlist inside while doing so.
- **Modify the grammar**
  Change the grammar to factor out the difference.

$$\begin{aligned} stat \rightarrow &\quad \text{'loop'}\ statlist\ loopstat \\ loopstat \rightarrow &\quad \text{'until'}\ expr \\ loopstat \rightarrow &\quad \text{'while'}\ expr \\ loopstat \rightarrow &\quad \text{'forever'} \end{aligned}$$

However there are more difficult problems, which can be more easily fixed with bottom-up parsing.

Left recursion

Right recursive grammars produce right recursive parse trees:

$$\begin{aligned} add \rightarrow &\quad mul\ \text{'+'}\ add \\ add \rightarrow &\quad mul\ \text{'-'}\ add \\ add \rightarrow &\quad mul \\ mul \rightarrow &\quad val\ \text{'*'}\ mul \\ mul \rightarrow &\quad val\ \text{'/'}\ mul \\ mul \rightarrow &\quad val \\ val \rightarrow &\quad integer \\ val \rightarrow &\quad identifier \end{aligned}$$

We consider this right-recursive as the recursion on productions is right of the symbol.

Top-down parsing implemented through **Recursive Descent Parsers** cannot do left recursive grammars. This is as it will result in an infinite recursion.

$$add \rightarrow \quad add\ \text{'+'}\ mul$$

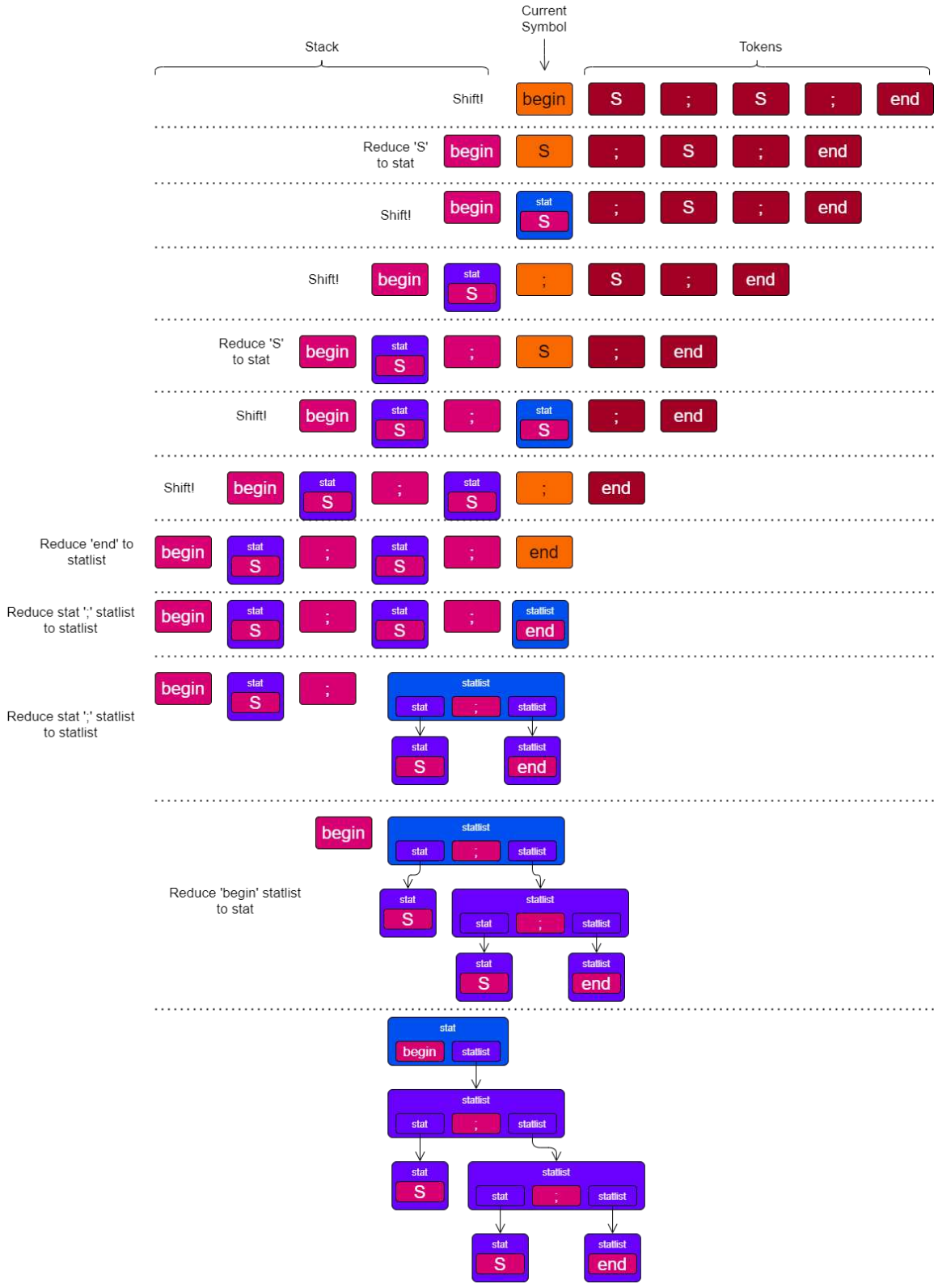
If attempting to parse this production:

```
1 def parse_add(input):
2     (add_parse_tree, rest) = parse_add(input)
3     rest = parse_plus(rest) # infinite recursion
4     (mul_parse_tree, rest) = parse_mul(rest)
5     return (tree(add_parse_tree, mul_parse_tree), rest)
```

Bottom-up Parsing

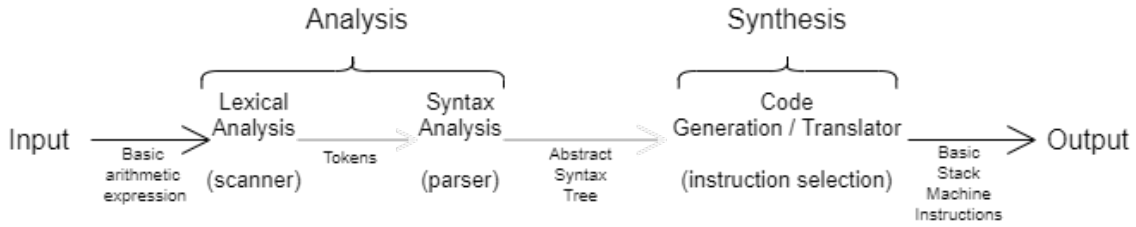
- The grammar's productions are used *right* $\rightarrow$ *left*.
- Input is compared against the right hand side to produce a non-terminal on the left.
- Parsing is complete when the whole input is replaced by the start symbol.

Bottom up parsers are difficult to implement, so parser generators are recommended.


Stack
Current Symbol
Tokens
Shift!
Reduce 'S' to stat
Shift!
Shift!
Reduce 'S' to stat
Shift!
Shift!
Reduce 'end' to statlist
Reduce stat ';' statlist to statlist
Reduce stat ';' statlist to statlist
Reduce 'begin' statlist to stat
begin
S
;
end
stat
statlist

Simple Complete Compiler

A very basic compiler written is haskell to convert basic arithmetic expressions into instructions for a basic stack machine.




```
{-
Simple compiler example:
Arithmetic Expressions -> Stack Machine Instructions

Original Description:
"Compiling arithmetic expressions into code for a stack machine.  This is not a
solution to Exercise 2 - it's an executable version of the code generator for
expressions, which is given in the notes.  Build on it to yield a code generator
for statements.

Paul Kelly, Imperial College, 2003

Tested with Hugs (Haskell 98 mode), Feb 2001 version"

Changes:
- This version has been updated to work with Haskell version 8.6.5
- Use of where over let & general refactoring
- Fixed bug with execute (missing patterns for invalid stack instructions)
- New grammar to support multiplication and division

Grammar:
 add    -> mul + add | mul - add | mul
 mul    -> factor * mul | factor / mul | factor
 factor -> number | identifier

+- (right associative, low precedence)
*/ (right associative, high precedence)

Eg:
> tokenise "a+b*17"
[IDENT a,PLUS,IDENT b,MUL,NUM 17]

> parser (tokenise "a+b-17")
Plus (Ident a, Minus (Ident b, Num 17))

> parser (tokenise "3-3-3*77*a-4")
Minus (Num 3, Minus (Num 3, Minus (Mul (Num 3, Mul (Num 77, Ident a)), Num 4)))

> compile "a+b+c*7-3"
[PushVar a,PushVar b,PushVar c,PushConst 7,MulToS,PushConst 3,SubToS,AddToS,AddToS]

> translate (parser (tokenise "a+b/17"))
[PushVar a,PushVar b,PushConst 17,DivToS,AddToS]

```

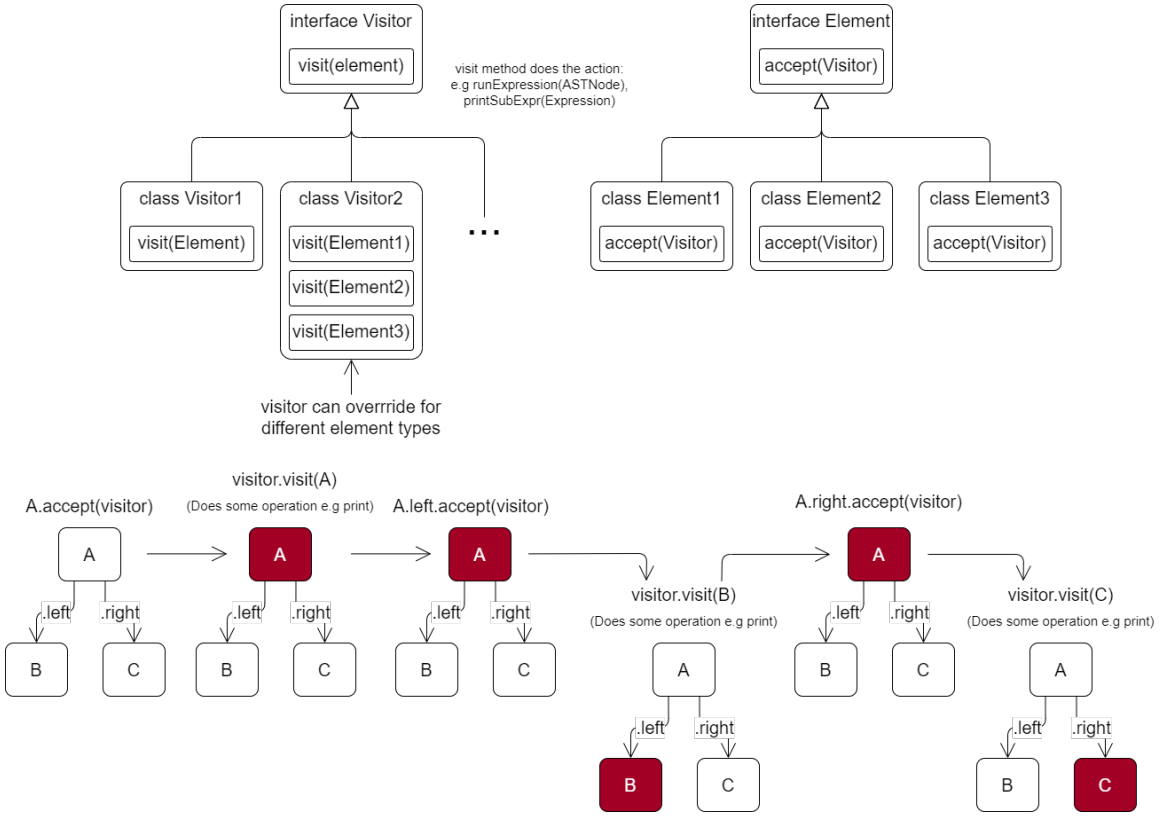
```
> putStr (runAnimated [("a", 9)] [] (translate (parser (tokenise "100+a*3-17"))))
[100]
[9,100]
[3,9,100]
[27,100]
[17,27,100]
[10,100]
[110]
[110]

-}

import Data.Char ( isDigit, isAlpha, digitToInt )
import Text.Parsec (tokens, Stream (uncons))

-- Token data type
data Token
  = IDENT [Char] | NUM Int | PLUS | MINUS | MUL | DIV

-- Ast (abstract syntax tree) data type
data Ast
  = Ident [Char] | Num Int | Plus Ast Ast | Minus Ast Ast | Mul Ast Ast | Div Ast Ast

-- Instruction data type
--
-- PushConst pushes a given number onto the stack; AddToS takes the top
-- two numbers from the top of the stack (ToS), and them and pushes the sum.
-- (We have to invent new names to avoid clashing with MUL, Mul etc above)
data Instruction
  = PushConst Int | PushVar [Char] | AddToS | SubToS | MulToS | DivToS

instance Show Token where
  showsPrec p (IDENT name) = showString "IDENT " . showString name
  showsPrec p (NUM num) = showString "NUM " . shows num
  showsPrec p (PLUS) = showString "PLUS"
  showsPrec p (MINUS) = showString "MINUS"
  showsPrec p (MUL) = showString "MUL"
  showsPrec p (DIV) = showString "DIV"

instance Show Ast where
  showsPrec p (Ident name) = showString "Ident " . showString name
  showsPrec p (Num num) = showString "Num " . shows num
  showsPrec p (Plus e1 e2) = showString "Plus (" . shows e1 . showString ", " . shows
      ↪  e2 . showString ")"
  showsPrec p (Minus e1 e2) = showString "Minus (" . shows e1 . showString ", " .
      ↪ shows e2 . showString ")"
  showsPrec p (Mul e1 e2) = showString "Mul (" . shows e1 . showString ", " . shows
      ↪ e2 . showString ")"
  showsPrec p (Div e1 e2) = showString "Div (" . shows e1 . showString ", " . shows
      ↪ e2 . showString ")"

instance Show Instruction where
  showsPrec p (PushConst n) = showString "PushConst " . shows n
  showsPrec p (PushVar name) = showString "PushVar " . showString name
  showsPrec p AddToS = showString "AddToS"
  showsPrec p SubToS = showString "SubToS"
  showsPrec p MulToS = showString "MulToS"
  showsPrec p DivToS = showString "DivToS"

-- Parse the tokens (top-down) by parsing each expression to get a new parse
```

```haskell
-- tree and the rest of the tokens. No tokens should remain after parsing.
parser :: [Token] -> Ast
parser tokens
  | null rest = tree
  | otherwise = error "(parser) excess rubbish"
  where
    (tree, rest) = parseAdd tokens

parseAdd :: [Token] -> (Ast, [Token])
parseAdd tokens
  = case rest of
      (PLUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Plus multtree
          ↪ subexptree, rest3)
      (MINUS : rest2) -> let (subexptree, rest3) = parseAdd rest2 in (Minus multtree
          ↪ subexptree, rest3)
      othertokens -> (multtree, othertokens)
  where
    (multtree, rest) = parseMul tokens

parseMul :: [Token] -> (Ast, [Token])
parseMul tokens
  = case rest of
      (MUL : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Mul factortree
          ↪ subexptree, rest3)
      (DIV : rest2) -> let (subexptree, rest3) = parseMul rest2 in (Div factortree
          ↪ subexptree, rest3)
      othertokens -> (factortree, othertokens)
  where
    (factortree, rest) = parseFactor tokens

parseFactor :: [Token] -> (Ast, [Token])
parseFactor ((NUM n):restoftokens) = (Num n, restoftokens)
parseFactor ((IDENT x):restoftokens) = (Ident x, restoftokens)
parseFactor [] = error "(parseFactor) Attempted to parse empty list"
parseFactor (t:_) = error $ "(parseFactor) error parsing token " ++ show t


-- Lexical analysis - tokenisation
tokenise :: [Char] -> [Token]
tokenise [] = []                        -- (end of input)
tokenise (' ':rest) = tokenise rest      -- (skip spaces)
tokenise ('+':rest) = PLUS : (tokenise rest)
tokenise ('-':rest) = MINUS : (tokenise rest)
tokenise ('*':rest) = MUL : (tokenise rest)
tokenise ('/':rest) = DIV : (tokenise rest)
tokenise (ch:rest)
  | isDigit ch  = (NUM dn):(tokenise drest2)
  | isAlpha ch  = (IDENT an):(tokenise arest2)
  where
    (dn, drest2) = convert (ch:rest)
    (an, arest2) = getname (ch:rest)
tokenise (c:_) = error $ "(tokenise) unexpected character " ++ [c]

getname :: [Char] -> ([Char], [Char]) -- (name, rest)
getname = flip getname' []
  where
      getname' :: [Char] -> [Char] -> ([Char], [Char])
      getname' [] chs = (chs, [])
      getname' (ch : str) chs
        | isAlpha ch = getname' str (chs++[ch])
```

```
157         | otherwise  = (chs, ch : str)
158
159 convert :: [Char] -> (Int, [Char])
160 convert = flip conv' 0
161   where
162     conv' [] n = (n, [])
163     conv' (ch : str) n
164       | isDigit ch = conv' str ((n*10) + digitToInt ch)
165       | otherwise  = (n, ch : str)
166
167 -- Translate - the code generator
168 translate :: Ast -> [Instruction]
169 translate (Num n) = [PushConst n]
170 translate (Ident x) = [PushVar x]
171 translate (Plus e1 e2) = translate e1 ++ translate e2 ++ [AddToS]
172 translate (Minus e1 e2) = translate e1 ++ translate e2 ++ [SubToS]
173 translate (Mul e1 e2) = translate e1 ++ translate e2 ++ [MulToS]
174 translate (Div e1 e2) = translate e1 ++ translate e2 ++[DivToS]
175
176 compile :: [Char] -> [Instruction]
177 compile = translate . parser . tokenise
178
179 -- Execute, run - simulate the machine running the stack instructions
180 --
181 -- Note that this simple machine is too simple to be realistic;
182 -- (1) 'execute' doesn't return the store, so no instruction can change it
183 -- (2) 'run' forgets each instruction as it is executed, so can't do loops
184
185 -- The state of the machine consists of a store (a set of associations
186 -- between variable and their values), together with a stack:
187
188 type Stack = [Int]
189 type Store = [([Char], Int)]
190
191 -- 'run' executes a sequence of instructions using a specified
192 -- store, and starting from a given stack
193 --
194 run :: Store -> Stack -> [Instruction] -> Stack
195
196 run store stack [] = stack
197 run store stack (i : is) = run store (execute store stack i) is
198
199 -- 'execute' applies a given instruction to the current state of the
200 -- machine - ie the store and the stack
201 --
202 execute :: Store -> Stack -> Instruction -> Stack
203 execute store (a : b: rest) AddToS        = ( (b+a) : rest )
204 execute store (a : b: rest) SubToS        = ( (b-a) : rest )
205 execute store (a : b: rest) MulToS        = ( (b*a) : rest )
206 execute store (a : b: rest) DivToS        = ( (b `div` a) : rest )
207 execute store rest          (PushConst n) = ( n : rest )
208 execute store rest          (PushVar x)   = ( n : rest )
209   where n = valueOf x store
210 execute store [x] instr = error $ "(execute) attempted to run " ++ show instr ++ "
      ↪ with only" ++ show x ++ " on the stack"
211 execute store []  instr = error $ "(execute) attempted to run " ++ show instr ++ "
      ↪ with an empty stack"
212
213 valueOf x [] = error ("no value for variable "++show x)
214 valueOf x ( (y,n) : rest ) = if x==y then n else valueOf x rest
```

```haskell

-- runAnimated does what run does but shows the stack after each step:
--
runAnimated :: Store -> Stack -> [Instruction] -> [Char]
runAnimated store stack [] = show stack
runAnimated store stack (i : is) = show newstack ++ "\n" ++ runAnimated store
    ↪ newstack is
  where
    newstack = execute store stack i
```

Visitor Pattern



The advantage of this pattern is that the data structure and operations are separated. This means new operations can be added easily by simply creating and passing a new visitor, which is then able to operate on the structure (e.g a tree as in the diagram).

Example usage could be: turtle operations on an **abstract syntax tree** of a turtle program (visitors for different colour, text styles, languages).

50006 - Compilers - (Prof Kelly) Lecture 3

Oliver Killane

10/01/22

Lecture Recording

Lecture recording is available here

Simple Programming Language

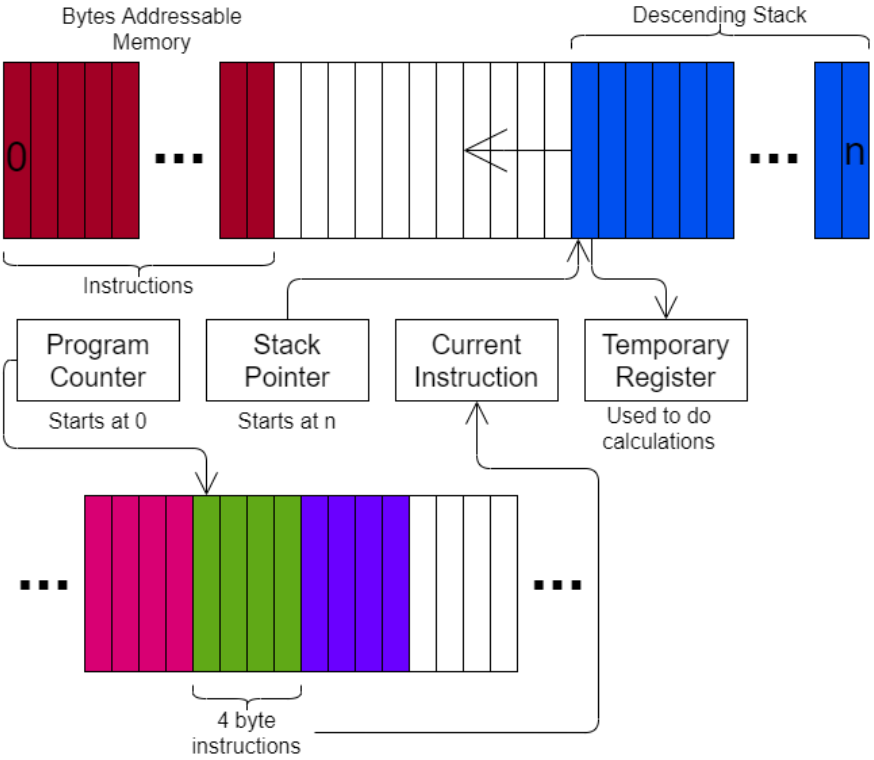
The grammar is expressed as:

$$
\begin{aligned}
\textit{stat} \rightarrow &\quad \textit{ident}\ \text{':='}\ \textit{exp} \mid \textit{stat}\ \text{';'}\ \textit{stat} \mid \text{'for'}\ \textit{ident}\ \text{'from'}\ \textit{exp}\ \text{'to'}\ \textit{exp}\ \text{'do'}\ \textit{stat}\ \text{'od} \\
\textit{exp} \rightarrow &\quad \textit{exp}\ \textit{binop}\ \textit{exp} \mid \textit{unop}\ \textit{exp} \mid \textit{ident} \mid \textit{num} \\
\textit{binop} \rightarrow &\quad \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \mid \\
\textit{unop} \rightarrow &\quad \text{'-'}
\end{aligned}
$$

And abstract syntax tree as:

```haskell
data Stat =
    Assign Name Exp |
    Seq Stat Stat |
    ForLoop Name Exp Exp Stat
    deriving (Show)

data Exp =
    BinOp Op Exp Exp |
    Unop Op Exp |
    Ident Name |
    Const Int
    deriving (Show)

data Op =
    Plus |
    Minus |
    Times |
    Divide
    deriving (Show)

type Name = [Char]
```

Target Stack Machine



Assembly instructions (Some are directives/pesudoinstructions):

```
data Instruction
    = Add | Sub | Mul | Div
    | PushImm Int    -- Push an immediate value
    | PushAbs Name   -- push variable at given location on the Stack
    | Pop Name       -- remove the top of the stack and store at location name
    | CompEq          -- Subtract top two elements of the stack, replace with a
                     -- 1 is the result was zero, zero otherwise
    | Jump Label     -- Jump to the label
    | JTrue Label    -- Remove top item from stack, if 1 jump to label
    | JFalse Label   -- Remove top item from stack, if 0 jump to label
    | Define Label   -- Set destination for jump (An assembler directive,
                     -- not instruction).
```

Pesudocode for execution behaviour:

```
ADD / MINUS / MUL / DIV /:
    T := store[SP]
    SP := SP + 4
    T := store[SP] [+=*/] T
    store[SP] := T

PUSHIMM:
    SP := SP - 4
    store[SP] := operand(IR)

```

```
PUSHABS:
    T := store[operand(IR)]
    SP := SP - 4
    store[SP] := T

POP:
    T := store[SP]
    SP := SP + 4
    store[operand(IR)] := T

COMPEQ:
    T:= store[SP]
    SP := SP + 4
    T := store[SP] - T
    store[SP] = T=0 ? 1 : 0

JTRUE:
    T := store[SP]
    SP := SP + 4
    PC := T=1 ? operand(IR) : PC

JFALSE:
    T := store[SP]
    SP := SP + 4
    PC := T=0 ? operand(IR) : PC
```

Typical Assembly

```
start:
    PushAbs i
    PushImm 1
    Sub
    Pop i
    PushAbs i
    PushImm 100
    CompEq
    JTrue start
```

Compiler Code Generated

```
[
    Define "start",
    PushAbs "i",
    PushImm 1,
    Sub,
    Pop "i",
    PushAbs "i",
    CompEq,
    Jtrue "Start"
]
```

The define directive (and assembly label) are directives to make the linker/assembler convert jumps to the label into jumps to the memory address of the instruction immediately after the label.

Translation (Naive Implementation)

```
data Stat =
    Assign Name Exp |
    Seq Stat Stat |
    ForLoop Name Exp Exp Stat
    deriving (Show)

data Exp =
    BinOp Op Exp Exp |
    Unop Op Exp |
    Ident Name |
    Const Int
    deriving (Show)

data Op =
```

```
    Plus |
    Minus |
    Times |
    Divide
    deriving (Show)

type Name = [Char]

data Instruction
    = Add | Sub | Mul | Div
    | PushImm Int    -- Push an immediate value
    | PushAbs Name   -- push variable at given location on the Stack
    | Pop Name       -- remove the top of the stack and store at location name
    | CompEq          -- Subtract top two elements of the stack, replace with a
                     -- 1 is the result was zero, zero otherwise
    | Jump Label     -- Jump to the label
    | JTrue Label    -- Remove top item from stack, if 1 jump to label
    | JFalse Label   -- Remove top item from stack, if 0 jump to label
    | Define Label   -- Set destination for jump (An assembler directive,
                     -- not instruction).

type Label = [Char]

transExp :: Exp -> [Instruction]
transExp (BinOp op e1 e2)
    = transExp e1 ++ transExp e2 ++ [case op of
        Plus -> Add
        Minus -> Sub
        Times -> Mul
        Divide -> Div]
transExp (Unop Minus e)
    = transExp e ++ [PushImm (-1), Mul]
transExp (Unop _ _)
    = error "(transExp) Only '-' unary operator supported"
transExp (Ident id) = [PushAbs id]
transExp (Const n) = [PushImm n]

transStat :: Stat -> [Instruction]
transStat (Assign id exp) = transExp exp ++ [Pop id]
transStat (Seq s1 s2) = transStat s1 ++ transStat s2


{-
for x:e1 to e2 do
    body
od

x := <e1>
loop:
    if <e2> then goto break
    <body>
    x := x + 1
    goto loop
break:


<transExp e1>
Pop x
define "loop"
<tranEval e2>
```

```
75 CompEq
76 JTrue "break"
77 <transStat body>
78 PushImm 1
79 Add
80 Pop x
81 Jump "loop"
82 Define "break"
83 -}
84
85 -- assumes the labels will be unique, this almost always not the case
86 transStat (ForLoop x e1 e2 body)
87  = transExp e1 ++ Pop x:Define "loop":transExp e2 ++ CompEq:JTrue "break":transStat
      ↪ body ++ [PushImm 1, Add, Pop x, Jump "loop", Define "break"]
```

Intermediate Representations

- **Abstract Syntax Tree** Usually the first intermediate representation. Can include statements, operations and expressions in a uniform way (simple data structure)
  Useful for sophisticated instruction selections and register allocation.
- **Flattened Control Flow Graph** Represents assembler-level code
  Order of opeerations defines control flow, useful for loop-invariant code motion.
- **Dependency Based Graphs** More complex, used by most modern compilers.
  Used for optimisations, can create 'static single assignment' graphs to deal with dependencies on mutable data.

50006 - Compilers - (Prof Kelly) Lecture 4

Oliver Killane

11/01/22

Lecture Recording

Lecture recording is available here

Unbounded Register Use

We will generate code for arithmetic expressions that:

- Assumes there will always be enough registers.
- Handles the case when we run out of registers.
- Translates expressions while minimising number of registers required.

```
data Instruction
   = Add Reg Reg | Sub Reg Reg
     | Mul Reg Reg | Div Reg Reg -- Op r1 r2 -> r1 := r1 <Op> r2
     | AddImm Reg Int | SubImm Reg Int
     | MulImm Reg Int | DivImm Reg Int -- <Op>Imm r c -> r := r <Op> c
     | Load Reg Name    -- Load r1 n    -> r1 := mem[n]
     | LoadImm Reg Int  -- LoadImm r1 i -> r1 := i
     | Store Reg Name   -- Load r1 n    -> mem[n] := r1
     | Push Reg         -- Push r1      -> SP++; mem[SP] := r1
     | Pop Reg          -- Pop r2       -> r1 := mem[SP]; SP--
     | CompEq Reg Reg   -- CompEq r1 r2 -> r1 := r1 - r2 = 0 ? 1 : 0
     | JTrue Reg Label  -- JTrue r1 l   -> IF r1 = 1 THEN JUMP TO l
     | JFalse Reg Label -- JFalse r1 l  -> IF r1 = 0 THEN JUMP TO l
     | Define Label     -- Assembler directive to set up label
```

We can take an input such as:

$$(100 * 3) + ((200 * 2) + 300) + (400 + (500 * 3))$$

	Instruction	Stack Slot			
		3	2	1	0
0	PushImm 100				100
1	PushImm 3			3	100
2	Mul			3	300
3	PushImm 200			200	300
4	PushImm 2		2	200	300
5	Mul		2	400	300
6	PushImm 300		300	400	300
7	Add		300	700	300
8	Add		300	700	1000
9	PushImm 400		300	400	1000
10	PushImm 500		500	400	1000
11	PushImm 3	3	500	400	1000
12	Mul	3	1500	400	1000
13	Add	3	1500	1900	1000
14	Add	3	1500	1900	2900

We can use the placement of values in the stack (relative to the initial stack pointer) to assign registers. Assigning each slot to a register.

We want to provide the translator with the register to place the result in, it can use any higher registers.

	Instruction	Register R3	R2	R1	R0
0	LoadImm R0 100				100
1	LoadImm R1 3			3	100
2	Mul R0 R1			3	300
3	LoadImm R1 200			200	300
4	LoadImm R2 2		2	200	300
5	Mul R1 R2		2	400	300
6	LoadImm R2 300		300	400	300
7	Add R1 R2		300	700	300
8	Add R0 R1		300	700	1000
9	LoadImm R1 400		300	400	1000
10	LoadImm R2 500		500	400	1000
11	LoadImm R3 3	3	500	400	1000
12	Mul R2 R3	3	1500	400	1000
13	Add R1 R2	3	1500	1900	1000
14	Add R0 R1	3	1500	1900	2900

Register Improvements

We could improve our generated code if there were instructions available for in place constant application.

IA32 with Immediate Operand

```
1 movl $3, %eax
2 imull $3, %eax
3 addl $4, %eax
```

Our Simple Assembly

```
1 PushImm R0 3
2 PushImm R1 3
3 Mul R0 R1
4 PushImm R1 4
5 Add R0 R1
```

	Instruction	Register R3	R2	R1	R0
0	LoadImm 0 3				3
1	MulImm 0 100				300
3	LoadImm 1 2			2	300
4	MulImm 1 200			400	300
5	AddImm 1 300			700	300
6	Add 0 1			700	1000
7	LoadImm 1 3			3	1000
8	MulImm 1 500			1500	1000
9	AddImm 1 400			1900	1000
10	Add 0 1			1900	2900

Code for Translation

```haskell
translateOp :: Op -> (Int -> Int -> Instruction)
translateOp Plus = Add
translateOp Minus = Sub
translateOp Times = Mul
translateOp Divide = Div

translateOpImm :: Op -> (Int -> Int -> Instruction)
translateOpImm Plus = AddImm
translateOpImm Minus = SubImm
translateOpImm Times = MulImm
translateOpImm Divide = DivImm

transExp :: Exp -> Reg -> [Instruction]
transExp (Const n) r = [LoadImm r n]
transExp (Ident id) r = [Load r id]

-- Only allow for - unary operato (e.g -3)
transExp (Unop Minus e) r = transExp e r ++ [MulImm r (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- As * and + are commutative, the order does not matter
transExp (BinOp Times (Const n) e) r = transExp e r ++ [MulImm r n]
transExp (BinOp Plus (Const n) e) r = transExp e r ++ [AddImm r n]

-- Can run left hand, then do right hand with immediate operand
transExp (BinOp op e (Const n)) r = transExp e r ++ [translateOpImm op r n]

-- General case for two expressions
transExp (BinOp op e1 e2) r
  = transExp e1 r ++ transExp e2 (r+1) ++ [translateOp op r (r+1)]
```

Bounded Number of Registers

Accumulator Machine

Has a single register (**Accumulator**) upon which arithmetic instructions can be applied.

```haskell
data Instruction = Add | Sub | Mul | Div --
  | AddImm Int | SubImm Int | MulImm Int | DivImm Int   --
  | CompEq          -- CompEq   -> Acc :=
  | Push            -- Push     -> SP--; mem[SP] := Acc
  | Pop             -- Pop      -> Acc := mem[SP]; SP++
  | Load Name       -- Load n   -> Acc := mem[n]
  | LoadImm Int     -- Load i   -> Acc := i
  | Store Name      -- Store n  -> mem[n] := Acc
  | Jump Label      -- Jump l   -> PC := l
  | JTrue Label     -- JTrue l  -> IF Acc = 1 THEN JUMP TO l
  | JFalse Label    -- JFalse l -> IF Acc = 0 THEN JUMP TO l
  | Define Label    -- Assembler directive to set up label
```

	Instruction	Acc	Stack 1	Stack 0
0	LoadImm 500	500		
1	MulImm 3	1500		
3	AddImm 400	1900		
4	Push	1900		1900
5	LoadImm 200	200		1900
6	MulImm 2	400		1900
7	AddImm 300	700		1900
8	Push	700	700	1900
9	LoadImm 100	100	700	1900
10	MulImm 3	300	700	1900
11	Add	1000	700	1900
12	Add	2900	700	1900

```haskell
transOpImm :: Op -> (Int -> Instruction)
transOpImm Plus = AddImm
transOpImm Minus = SubImm
transOpImm Times = MulImm
transOpImm Divide = DivImm

transOp :: Op -> Instruction
transOp Plus = Add
transOp Minus = Sub
transOp Times = Mul
transOp Divide = Div

transExp :: Exp -> [Instruction]
transExp (Const n) = [LoadImm n]
transExp (Ident x) = [Load x]
transExp (Unop Minus e) = transExp e ++ [MulImm (-1)]

-- Can only use Minus unary operator (e.g -3)
transExp (Unop _ _)
  = error "(transExp) Only '-' unary operator supported"

-- If constant on the left, can use immediate operand
transExp (BinOp op e (Const n)) = transExp e ++ [transOpImm op n]

-- With commutative operator, can switch order to use immediate operand
transExp (BinOp Times (Const n) e) = transExp e ++ [MulImm n]
transExp (BinOp Plus (Const n) e) = transExp e ++ [AddImm n]

-- General case for two expressions
transExp (BinOp op e1 e2) = transExp e2 ++ Push : transExp e1 ++ [transOp op]
```

Limited Register Set

One solution is to combine the register and accumulator strategies

```haskell
data Instruction
   = Add Reg Reg | Sub Reg Reg
     | Mul Reg Reg | Div Reg Reg -- Op r1 r2 -> r1 := r1 <Op> r2
     | AddImm Reg Int | SubImm Reg Int
     | MulImm Reg Int | DivImm Reg Int -- <Op>Imm r c -> r := r <Op> c
```

```
    | AddStack Reg | SubStack Reg
    | MulStack Reg | DivStack Reg -- <Op>Imm r c -> r := r <Op> mem[SP]; SP--
    | Load Reg Name     -- Load r1 n     -> r1 := mem[n]
    | LoadImm Reg Int   -- LoadImm r1 i  -> r1 := i
    | Store Reg Name    -- Load r1 n     -> mem[n] := r1
    | Push Reg          -- Push r1       -> SP++; mem[SP] := r1
    | Pop Reg           -- Pop r2        -> r1 := mem[SP]; SP--
    | CompEq Reg Reg    -- CompEq r1 r2  -> r1 := r1 - r2 = 0 ? 1 : 0
    | JTrue Reg Label   -- JTrue r1 l    -> IF r1 = 1 THEN JUMP TO l
    | JFalse Reg Label  -- JFalse r1 l   -> IF r1 = 0 THEN JUMP TO l
    | Define Label      -- Assembler directive to set up label
```

- When free register sremain, use the register machine strategy.
- When the limit is reached (one register left to use as accumulator), switch to accumulator strategy.

This results in most expressions using full benefit of registers, while very large expressions can still be correctly executed.

```
translateOp :: Op -> (Int -> Int -> Instruction)
translateOp Plus = Add
translateOp Minus = Sub
translateOp Times = Mul
translateOp Divide = Div

translateOpImm :: Op -> (Int -> Int -> Instruction)
translateOpImm Plus = AddImm
translateOpImm Minus = SubImm
translateOpImm Times = MulImm
translateOpImm Divide = DivImm

translateOpStack :: Op -> (Int -> Instruction)
translateOpStack Plus = AddStack
translateOpStack Minus = SubStack
translateOpStack Times = MulStack
translateOpStack Divide = DivStack

maxReg :: Int
maxReg = 10

transExp :: Exp -> Reg -> [Instruction]

-- No need to check maxreg as we only use one register (a reg or the last one)
transExp (Const n) r = [LoadImm r n]
transExp (Ident id) r = [Load r id]
transExp (Unop Minus e) r = transExp e r ++ [MulImm r (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- As * and + are commutative, the order does not matter, only use one register so no
    ↪ need to check maxReg
transExp (BinOp Times (Const n) e) r = transExp e r ++ [MulImm r n]
transExp (BinOp Plus (Const n) e) r = transExp e r ++ [AddImm r n]

-- Can run left hand, then do right hand with immediate operand
transExp (BinOp op e (Const n)) r = transExp e r ++ [translateOpImm op r n]

-- General case for two expressions, we need to take into account the registers used.
transExp (BinOp op e1 e2) r
```

```
40    | r == maxReg = transExp e2 r ++ Push r : transExp e1 r ++ [translateOpStack op r]
41    | otherwise = transExp e1 r ++ transExp e2 (r+1) ++ [translateOp op r (r+1)]
```

50006 - Compilers - (Prof Kelly) Lecture 5

Oliver Killane

12/01/22

Register Usage

Lecture Recording

Lecture recording is available here

Register usage has several advantages:

- Registers are very fast to read from and write to.
- Registers are multi-ported (two or more registers can be read per clock cycle).
- Registers are specified by a small field of the instruction (leaves room for immediate operands and other data).
- CPUs can optimise at runtime and use register accesses & data dependencies to optimise instruction ordering among other techniques.

Hence we should attempt to use as few registers as efficiently as possible and keep as little as possible in the rest of the memory hierarchy.

Order Does Matter!

Example below does not have immediate operand instructions.

$$x + (3 + (y * 2))$$

	Instruction	Register			
		R3	R2	R1	R0
0	LoadAbs R0 "x"				x
1	LoadImm R1 3			3	x
3	LoadAbs R2 "y"		y	3	x
4	LoadImm R3 2	2	y	3	x
5	Mul R2 R3	2	2*y	3	x
6	Add R1 R2	2	2*y	3 + (2*y)	x
7	Add R0 R1	2	2*y	3 + (2*y)	x+(3+(y*2))

$$((y * 2) + 3) + x$$

	Instruction	Register	
		R1	R0
0	LoadAbs R0 "y"		y
1	LoadImm R1 2	2	y
3	Mul R0 R1	2	2 * y
4	LoadImm R1 3	3	2 * y
5	Add R0 R1	3	3 + (2 * y)
6	LoadAbs R1 "x"	x	3 + (2 * y)
7	Add R0 R1	x	x +(3 + (2 * y))

Subexpression Ordering Principle

Given an expression E_1 *op* E_2 always evaluate the subexpression that uses the most registers first. This is as while the second expression is evaluated we must also store the result of the first in a

register. This is called **Sethi-Ullman Numbering**.

If E_1 evaluated first, registers needed is $max(E_1, E_2 + 1)$
If E_2 evaluated first, registers needed is $max(E_1 + 1, E_2)$

Given n_A registers to evaluate A and n_B for B:

	Action	**Registers**
(1)	Evaluate A	n_A
(2)	Result of A stored in a reg	1
(3)	Evaluate B while storing result of A	$n_B + 1$
(4)	Result of B stored in a reg	2
(5)	Operate on subexpression results	1

Hence we use a weight function to compute the number of registers required before translating the code.

```
1  weight :: Exp -> Int
2  -- Base cases, registers required to hold values
3  weight (Const _) = 1
4  weight (Ident _) = 1
5
6  -- Can use immediate operand multiply so no extra registers required
7  weight (Unop Minus e) = weight e
8  weight (Unop _ e) = error "(weight) can only use unary operator -"
9
10 -- As we can target registers, if either is a constant we can use immediate operands
11 weight (BinOp Plus (Const _) e)  = weight e
12 weight (BinOp Times (Const _) e) = weight e
13 weight (BinOp _ e (Const _))     = weight e
14
15 -- Use maximum of either
16 weight (BinOp _ e1 e2)
17     = min e1first e2first
18     where
19         e1first = max (weight e1) (weight e2 + 1)
20         e2first = max (weight e2) (weight e1 + 1)
```

Non commutative operations such as $-$ or $/$ need ordering to be maintained.

We can fix this by switching the order of registers for the operation (e.g $Div\ r1\ r$) instead, however this breaks our invariant (that the higher number registers can be used) as when we run the expression using r it can overwrite $r + 1$.

Register Targeting

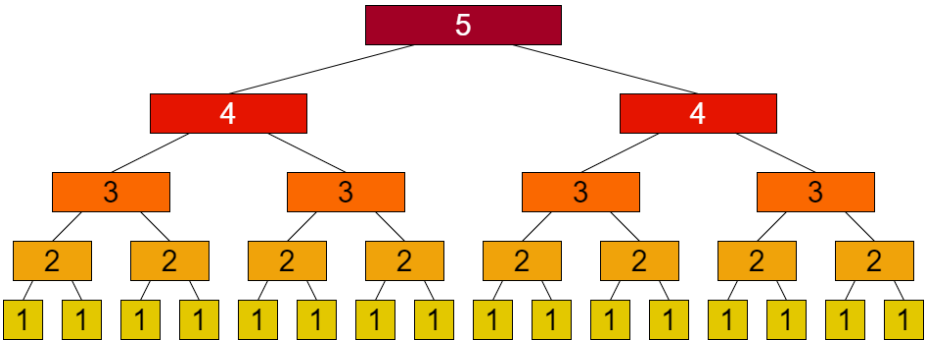
We want specify which registers can be used, must be preserved.

Here we also include the immediate operations. One complex part of compiler design and optimization is that instruction selection effects register usage (**weight** must take into account **transExp**).

```
1  transExp :: Exp -> [Register] -> [Instruction]
2  transExp (Const n) (dest:rest) = [LoadImm dest n]
3  transExp (Ident x) (dest:rest) = [Load dest x]
4
```

```haskell
-- Get result into dest register, the negate
transExp (Unop Minus e) reg@(dest:_) = transExp e reg ++ [MulImm dest (-1)]
transExp (Unop _ _) _
  = error "(transExp) Only '-' unary operator supported"

-- If the constant is on the right, we can use all operations
transExp (BinOp op e (Const n)) reg@(dest:_) = transExp e reg ++ [transOpImm op dest
    ↪ n]

-- If on the left, we can use the commutative operations
transExp (BinOp Plus (Const n) e) reg@(dest:_)  = transExp e reg ++ [Add dest n]
transExp (BinOp Times (Const n) e) reg@(dest:_) = transExp e reg ++ [Mul dest n]

-- If we are on the last register, default to accumulator scheme
-- Else we use the weight function to determine which path to follow
-- e1 <- dest and e2 <- next and Instr dest next
transExp (BinOp op e1 e2) [dest]
  = transExp e2 [dest]
  ++ Push dest : transExp e1 [dest]
  ++ [transOpStack op dest]
transExp (BinOp op e1 e2) (dest:next:rest)
  | weight e1 > weight e2
    = transExp e1 (dest:next:rest)
    ++ transExp e2 (next:rest)
    ++ [transOp op dest next]
  | otherwise
    = transExp e2 (next:dest:rest)
    ++ transExp e1 (dest:rest)
    ++ [transOp op dest next]
```

Effectiveness of Sethi-Ullman Numbering



The worst case is a perfectly balanced tree.

- k values
- $\frac{k}{2} - 1$ operators
- $\lceil \log_2 k \rceil$ registers required

Hence in the worst case N registers can support 2^N terms.

In this restricted setting, though does not account for reused variables & values and fails to put user variables in registers.

Was used in C compilers for many years, though optimising compilers commonly use more sophisticated techniques such as graph colouring in addition.

Register Allocation for Function Calls

Lecture Recording

Lecture recording is available here

$$x + a * getValue(b + c, 3)$$

- Need to know where parameters are when passed (stack, registers)
- Changing order of evaluation (can change result due to side effects)
- Register Targeting (ensure arguments are computed in the right registers)
- At point of call several registers may already be in use and can be different from different **call-sites**
-

Function Call Evaluation Order

$$f(a) + f(b) + f(c) \quad \text{or } g(f(a), f(b))$$

- In C++ the order is undefined, the compiler can choose any order, even if there are side-effects.
- In Java order is left $\rightarrow$ right.

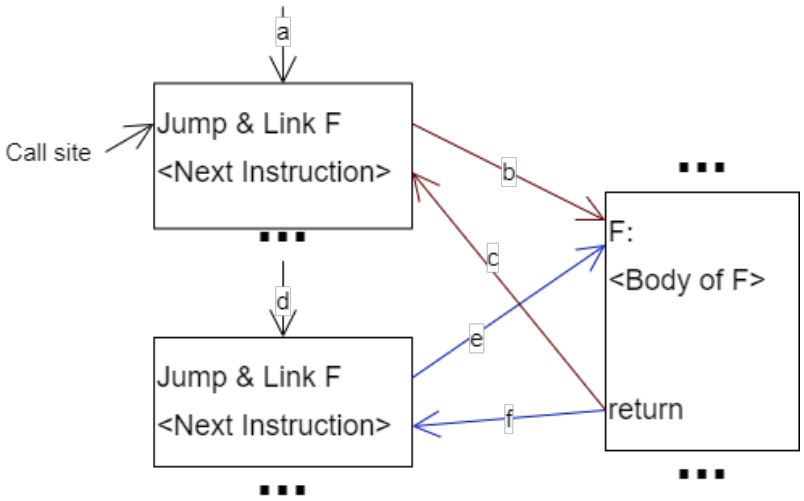
We must also consider register use, for example:

$$(f(x) + 1) + (1 * (a + j))$$

Which side of the + should be evaluated first depends on the context (e.g registers that need to be saved at the call site, and registers used by the callee, calling convention).

Calling a subroutine

- Functions can be called from many different places (call sites).
- Must go to the correct position in the program on return.
- Address of next instruction is saved and stored.



We care about the **feasible** paths:

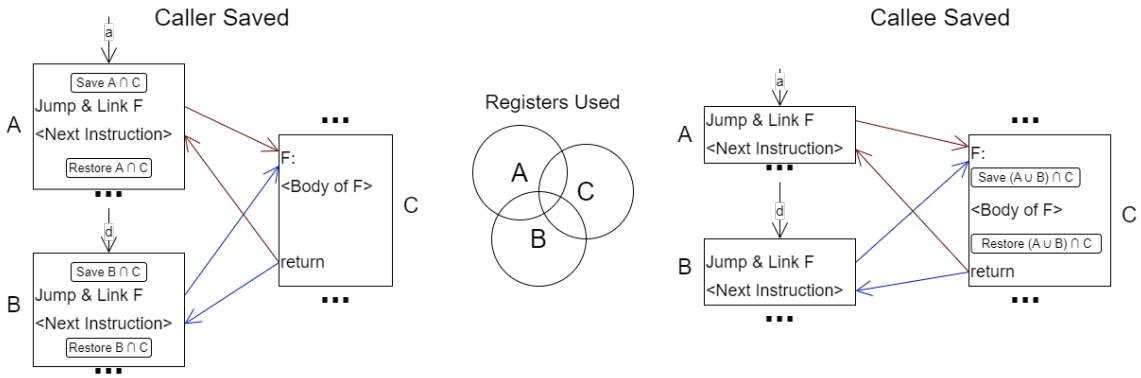
a,b,c,d,e,f,g	Feasible Path
a,b,f,g	A valid path in the graph, but not a feasible path (we understand how jumps & returns work)

The **infeasable control flow graph problem** becomes a difficult issue with lots of call sites.

Saving Registers

We must enforce a calling convention to ensure registers are not mangled (e.g non-argument or return registers are changed).

Caller Saved	Caller saved registers it is using to preserve them (callee cannot clobber).	Caller may save registers the callee does not use (redundant).
Callee Saved	Callee saves the registers it needs to use.	Callee saves registers that caller does not care use (redundant).



Another issue is **separate compilation**, we may want to compile a library, and link it later. Or use a shared library that is dynamically linked. Hence for caller saved there is no knowledge of callee register use, and if callee saved, it cannot know what registers callers are using.

Hence the compiler has to make a conservative assumption of the register usage, which inevitably results in redundant saves (two memory accesses required).

Alternatives

There have been architectures that solve this problem by making register preservation decisions be done at runtime.

The **VAX** architecture's call instructions use a small bitmap to provided by the caller to determine which registers to automatically save to the stack. More modern systems (RISC and CISC) do not employ such schemes, and simply have a calling convention for binary interfaces that the compiler optimises around.

The VAX call instructions are explained on page 88 of this manual (register save mask).

To solve this for a given architecture an **Application Binary Interface** is defined. This ensures linked libraries callers and callee's save the correct registers. Outside of interfaces the compiler can use any scheme it wants (not interacting with other binaries).

Intel IA32 register saving convention

Caller-Saved			Callee-Saved			Stack Pointer	Frame Pointer
%eax	%edx	%ecx	%ebx	%esi	%edi	%esp	%ebp

There are many more rules for the stack frame layout, arguments on the stack and parameter passing (registers to use).

ARM register saving convention

Caller Saved	r0	a1	Argument/Result/Scratch Register 1
	r1	a2	Argument/Result/Scratch Register 2
	r2	a3	Argument/Result/Scratch Register 3
	r3	a4	Argument/Result/Scratch Register 4
Callee Saved	r4	v1	Variable Register 1
	r5	v2	Variable Register 2
	r6	v3	Variable Register 3
	r7	v4	Variable Register 4
	r8	v5	Variable Register 5
Depends	r9	v6	Variable Register 6 or otherwise platform defined
Callee Saved	r10	v7	Variable Register 7
	r11	v8	Variable Register 8
	r12	IP	Intra Procedure call scratch register
Callee Saved	r13	SP	Stack pointer
	r14	LR	Link Register (used for jump to location, return)
	r15	PC	Program Counter
Callee Saved	r16-31		

MIPS32 register saving convention

	r0	zero	Constant register (0)
Caller Saved	r1	at	Temporary values in pesudo commands (e.g **slt**)
	r2	v0	Expression evaluation and results of a function
	r3	v1	Expression evaluation and results of a function
	r4	a0	Argument 1
	r5	a1	Argument 2
	r6	a2	Argument 3
	r7	a3	Argument 4
	r8	t0	Temporary
	r9	t1	Temporary
	r10	t2	Temporary
	r11	t3	Temporary
	r12	t4	Temporary
	r13	t5	Temporary
	r14	t6	Temporary
	r15	t7	Temporary
Callee Saved	r16	s0	Saved temporary
	r17	s1	Saved temporary
	r18	s2	Saved temporary
	r19	s3	Saved temporary
	r20	s4	Saved temporary
	r21	s5	Saved temporary
	r22	s6	Saved temporary
	r23	s7	Saved temporary
Caller Saved	r24	t8	Temporary
	r25	t9	Temporary
	r26	k0	Reserved for OS kernel
	r27	k1	Reserved for OS kernel
	r28	gp	Pointer to global area
	r29	sp	Stack pointer
	r30	fp or s8	Frame pointer
	r31	ra	Return address (used by function call)

Register Allocation By Graph Colouring

Lecture Recording

Lecture recording is available here

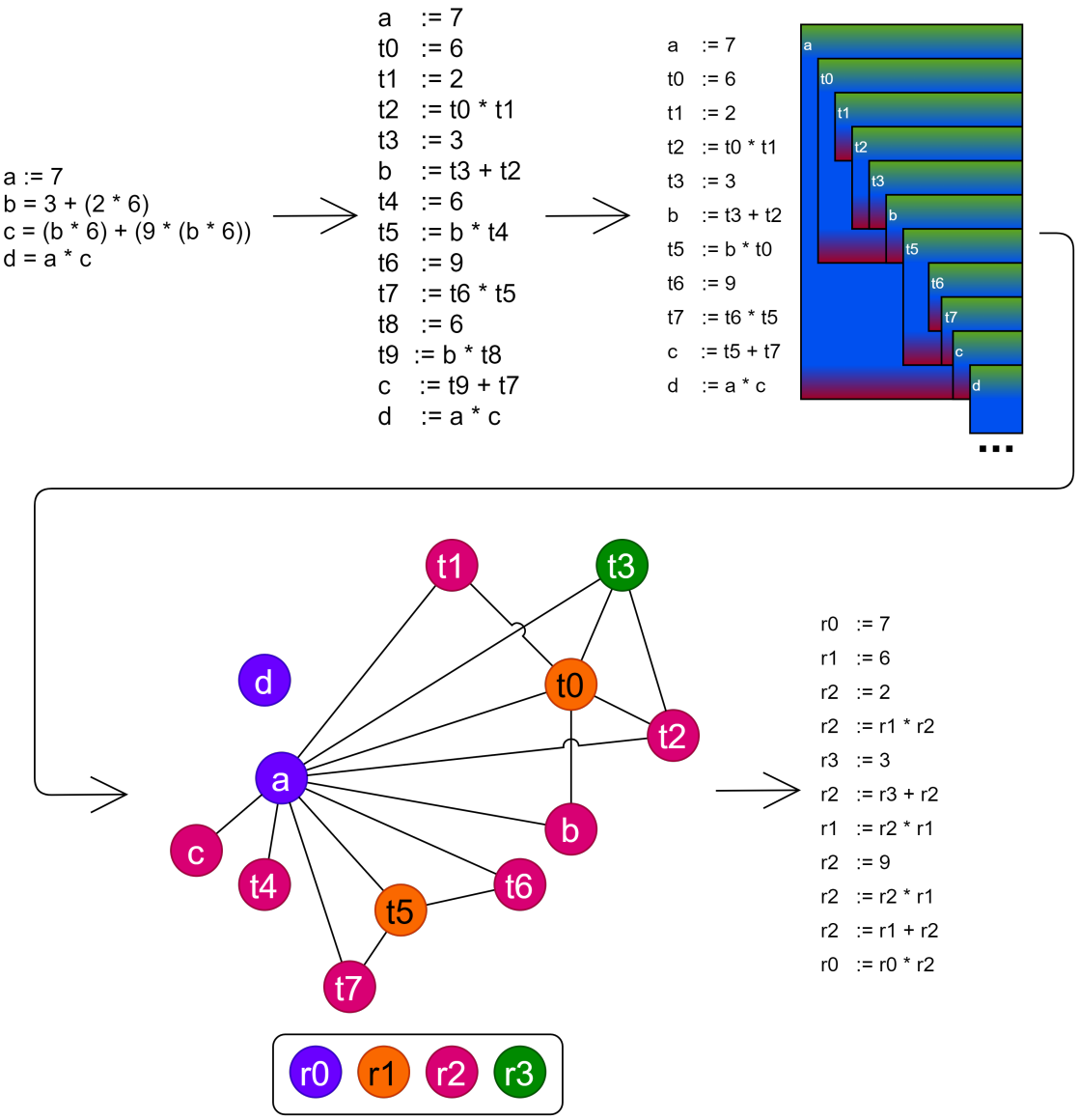
Moore's Law

An observation by **Intel** co-founder Gordon Moore that every two years the density of transistors in integrated circuits doubles.

This has been twisted into performance doubling every two years. Which has slowed significantly in the past two decades due to the limits of physics and chip fabrication technology. As a result the benefit of developing far more advanced optimising compilers has increased dramatically.

- The tree weighted translator simply traversed our expression's tree.
- Context (e.g locations of variables in registers) were not used.
- Repeated uses of a variable are not handled.
- Exploitation of context of generated code separated straightforward from optimising compilers.

Optimising compilers will attempt to use register instructions (faster).



1. **Use simple traversal to generate intermediate code**
   Temporary values are always saved in a named location. (e.g $t0 \ldots$). This way we can consider **all** values including intermediate ones.

2. **Construct an Inference Graph**
   each node is a temporary location, each edge connects simultaneously live locations.

   Registers that need to simultaneously store values must be different colours (different registers).

3. **Attempt To Colour Nodes**

If colouring is not possible **spilling occurs**.

(a) Find an edge, to remove it either split the live range (e.g temporarily put to memory).
(b) Redo the analysis to determine if the graph can now be coloured.

When choosing which values to spill it is important to consider how often a variable is used. e.g avoid spilling from innermost loop.

50006 - Compilers - (Prof Kelly) Lecture 6

Oliver Killane

07/02/22

Introduction to Optimisation

Lecture Recording

Lecture recording is available here

GCC -O

- **Without Optimisations**
  Program is slower, however easier to debug as generated code represents a simple translation.
- **With Optimisation**
  Various optimisations such as loop hoisting (moving loop invariant code out of the loop), constant propagation, inlining and smart register allocation make the program execute more quickly, but at the cost of the debugger.

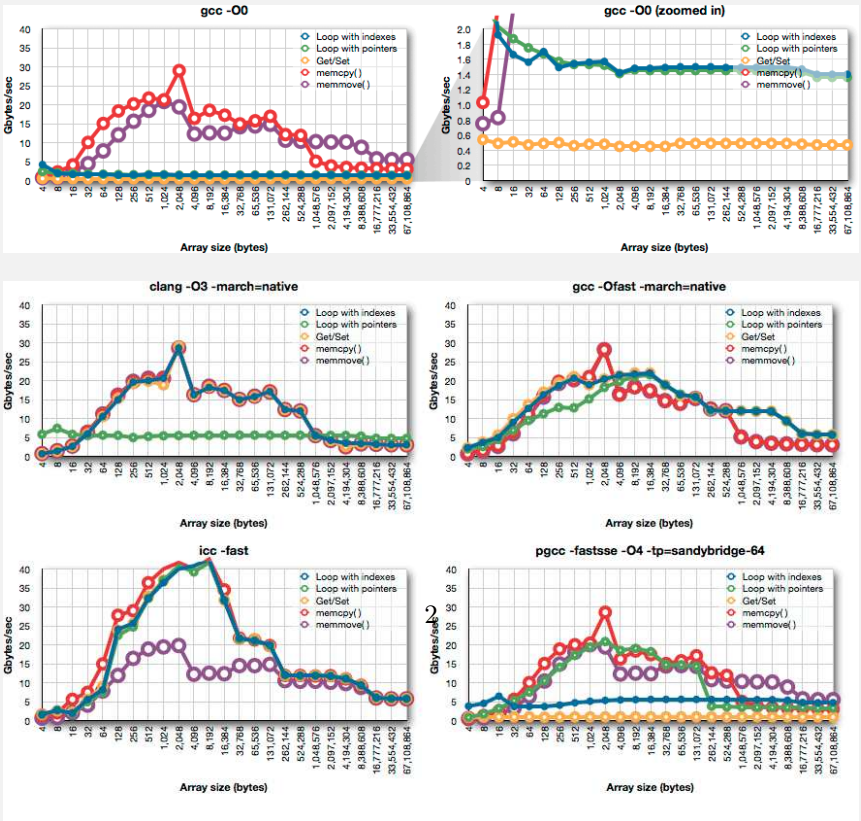
  Debuggers now much less useful as statements are reordered or completely changed from the source, & degugger tools such as setting variable values, jumping to lines and breakpoints will often not have the intended effect.

Example: Copying Arrays

When copying an array of values the compiler can optimise in several areas (index calculations, highly optimised copying from libraries such as **memcpy**) including in the most advanced case using vectorised instructions which can use a single instruction on multiply sections of data in an array (SIMD).

An example of this is here:

```
1  /* Loop with array indexes */
2  for ( size_t i = 0; i < length; ++i )
3          dst[i] = src[i];
4
5  /* Loop with pointers */
6  const int* s = src;
7  int* d = dst;
8  const int*const dend = dst + n;
9  while ( d != dend )
10         *d++ = *s++;
11
12 /* Loop with get/set function calls */
13 int get( const int*const src, const size_t index ) { return src
       ↪ [index]; }
14 int set( int*const dst, const size_t index, const int value ) {
       ↪  dst[index] = value; }
15
16 for ( size_t i = 0; i < n; ++i )
17         set( dst, i, get( src, i ) );
18
19 /* Memcpy (two regions are disjoint) */
20 memcpy( (void*)dst, (void*)src, n * sizeof(int) );
21
22 /* Memmove (for when regions may overlap) */
23 memmove( (void*)dst, (void*)src, n * sizeof(int) );
```



Definition: High Level Optimisation

Using high-level information encoded in the program (types, function analysis).

Example: Function Inlining

Replace the call site with a copy of the body of the function.

- Avoids call/return overhead.
- Creates further oportunities to optimise (e.g part of function result not used, or removal of call allowing for better register allocation).
- Can require static analysis (e.g call to a virtual or overloaded function requires information about types)

Definition: Low-Level Optimisations

Using low-level information (instruction types, the ISA, the order of instructions in the IR, etc) to optimise the output.

Note that we can loose information from high level representations as we get lower level.

Example: Instruction Scheduling

In pipelined architectures instruction order can impact the speed of processing, (e.g instructions immediately after a conditional branch may be waiting after a pipeline stall).

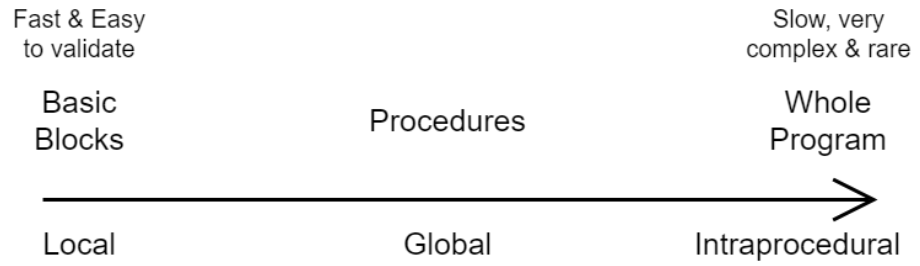
- Re-Order instructions to better allow parallel processing.
- Requires some dependency analysis (e.g switching order of storing to an array - check indexes, do $A[i]$ and $A[j]$ refer to the same location in a given context).

The spectrum of Optimisations

Definition: Peephole Optimisation

Scan through the assembly in order, looking for obvious cases to optimise.

- Can catch some of the worst cases (e.g store follwed by load of the same location).
- Very easy to implement (at smallest just consider two adjacent instructions).
- *Phase ordering problem* in what order should the optimisations be applied to get the best result?



Local	Optimisation on the level of basic blocks (single entry adn exit points) (e.g expressions)	Runs quickly and easy to validate.
Global	Optimisation on the scale of a whole procedure	Can have worse than $O(n)$ (Linear) complexity given n represents instructions, basic blocks or variables.
Intraprocedural	Optimisation over the whole program	Rare and hard to avoid excessive compile times.

Loop Optimisations

- **Loop Invariant Code Motion**
  An instruction is loop-invariant if its operands (inputs) only arrive from outside the loop (moving it out of the loop does not change the semantics).

  Hence the loop iteration has no effect on the value, so we can move this computation from within the loop (being redundantly run with the same inputs many times) to the loop pre-header (compute value just before loop and then use).

  More generally **strength reduction** is replacing a compelx operation, with a smaller simpler one.

```
1 int a = 7;
2 for (int b = 0; b < 4; b++) {
3     int c = a + 3 * 2; // invariant code
4     printf("%d", c);
5 }
```

```
1 int a = 7;
2 int c = a + 3 * 2; // code hoisted out of loop
3 for (int b = 0; b < 4; b++) {
4     printf("%d", c);
5 }
```

- **Strength Reduction**
  An **induction variable** is one which increases/decreases by a **loop invariant** amount each loop iteration.

  Once we detect **induction variables** we can use less costly instructions to compute their value.

```c
int a = 7;
for (int b = 0; b < 4; b++) {
    a += 4;
    printf("added 4 to a");
}
```

```c
int a = 23; // 7 + 4 * 4
for (int b = 0; b < 4; b++) {
    printf("added 4 to a");
}
```

Or a far, far more complex example:

```c
int a = 9;
int b = 7;
while (some_predicate(a)) {
    b = some_function(b);
    for (int i = 0; i < b; b++) {
        a++
    }
}
```

```c
int a = 9;
int b = 7;
while (some_predicate(a)) {
    b = some_function(b);
    a += b
}
```

- **Control Variable Selection**
  Replace the loop control variable with one of the induction variables that is used. The exit condition for the loop must be changed to work with the variable chosen.

```c
int* a = malloc(15 * sizeof(int));
for (int i = 0; i < 15; i++) {
    a[i] = 5;
}
```

```c
int* a = malloc(15 * sizeof(int));
int* end = a + 15;
for (; a != end; a++) {
    *a = 5;
}
```

- **Dead Code Elimination**
  Code that does not produce a used result can be eliminated.

  many other optimisations result in dead code (e.g inlining a function where not all the function's returned values or optional arguments are used.)

Intermediate Representation

An **intermediate representation** (**IR**) of the program used in code synthesis and optimisation.

- Must represent all primitive operations needed for execution.

- Be easy to analyse and manipulate.
- Independent of the target instruction set (allowing for many different ISAs to be targeted by backends using the **IR**).
- Uses temporary variables to store intermediate values to allow for some optimisations and register assignment (e.g via graph colouring).
- Still an area of discussion & active research, many different approaches including multiple IRs for different stages are used.

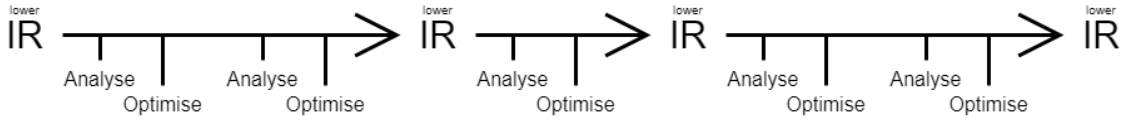
Definition: Lowering Representation

Taking high-level features and converting them into lower-level representations.

For example taking arrays and converting them into pointer arithmetic/address calculation.

When lowering you loose high-level information (e.g that values are part of an array), but can optimise the lower level representation (optimise address calculations).

Optimising Compiler synthesis can be described as several stages of analysis, optimisation and lowering of **IR**.



Data Flow Analysis for Live Ranges

Lecture Recording

Lecture recording is available here

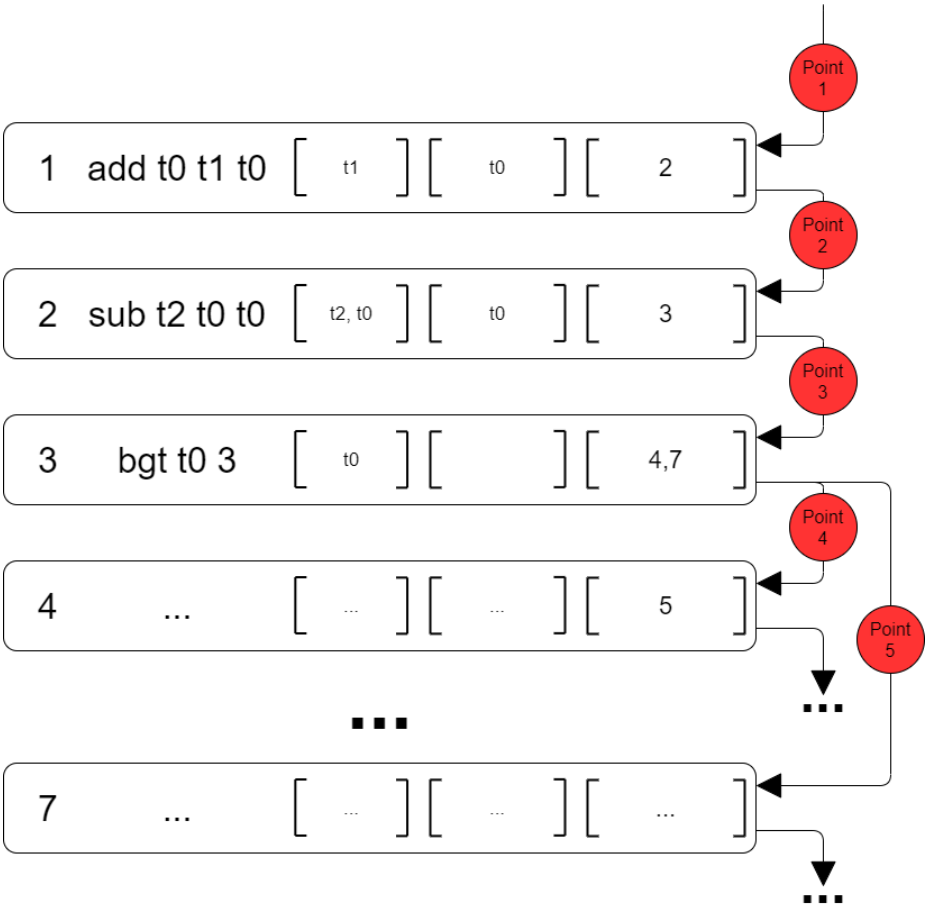
A live range is the range of instructions for which a temporary value must be maintained.

A live range starts at a definition, and ends when either the variable is used, or immediately if the value is never used.

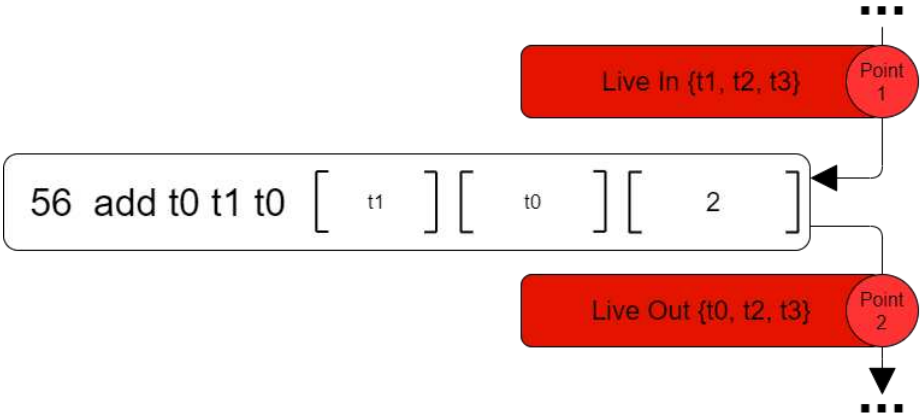
Control Flow Graph

We can generate a graph of simple **IR** (ideally 3 code) instructions in the program:

Live Range Definitions



- A **point** is any location between adjacent nodes.
- A path is a sequence of points traversing through the control flow graph.
- A variable can be *live* at a point if it may be used along some path that goes through the point.



- **Live Out** Live immediately after the node if it is live before any successors.

$$LiveOut(n) = \bigcup_{s \in succ(n)} LiveIn(s)$$

- **Live In** If it is live after the node (unless overwritten by the node) or it is used by the node.

$$LiveIn(n) = uses(n) \cup (LiveOut(n) - defines(n))$$

Iterative Method to get Live Ranges

```
for node in CFG {
    LiveIn(node) = {}
    LiveOut(node) = {}
}

repeat {
    for node in CFG {
        LiveIn(node) = uses(node) set-or (LiveOut(node) - defines(node))
        LiveOut(node) = set-or of LiveIn(all successors to node)
    }
} until LiveIn and LiveOut do not change
```

Once the iteration stops, the assignments will hold as predicates, meaning the definitions for *LiveIn* and *LiveOut* will be true.

Given there are limited nodes, and the size of the *LiveIn* and *LiveOut* can only increase, we know it will terminate.

Improvements to the Iterative Method

To reduce required iterations it is best to update the nodes from last → first, as generally the program will be connected to go from first instruction to last (cycles can exist which affects this). As the definitions are dependent on successor nodes.

Hence we consider it a *backwards analysis*.

The time complexity is dependent on the number of instructions and temporaries/registers.

50006 - Compilers - (Prof Kelly) Lecture 7

Oliver Killane

15/04/22

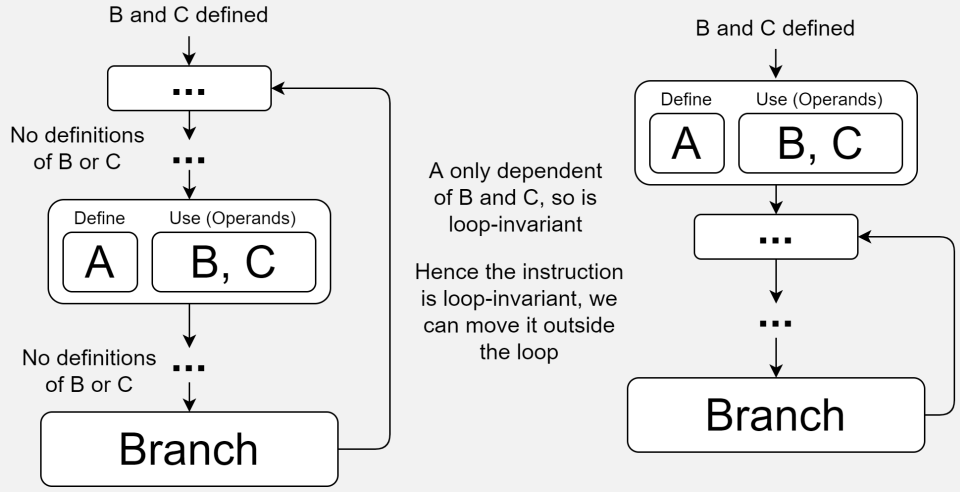
Loop Invariant Code Motion

Lecture Recording

Lecture recording is available here

Definition: Loop Invariant

An instruction is **loop-invariant** if its operands are only defined outside of the loop. Hence the value it defines is not loop-invariant (same for every iteration) and hence it may be possible to move the instruction outside the loop.



Finding Loop-Invariant Instructions

Using a control flow graph, each node is an instruction. We attempt to find definition nodes of form:

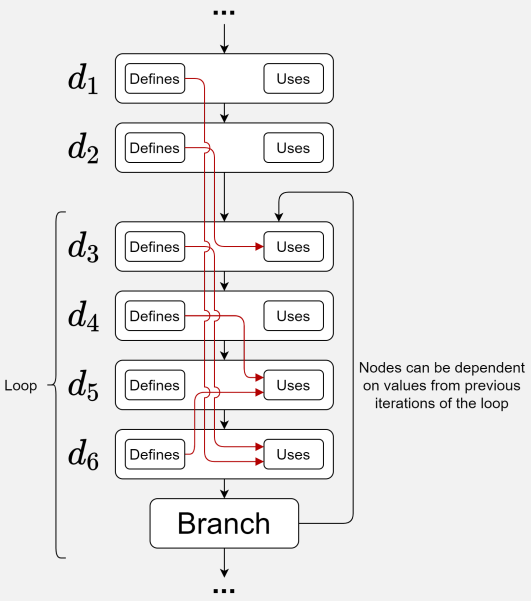
$$\underbrace{d}_{\text{Node ID}} \; : \; t_d := (\underbrace{u_1 \bullet u_2}_{\text{Binary Op}} \mid \underbrace{u_1}_{\text{Copy Op}} \mid \underbrace{c}_{\text{Constant}})$$

Where t_d is the destination, and u_i is for temporary variables used.

d is **loop-invariant** if every definition of a $u_i \in uses(d)$ that reaches d is outside the loop.

Example: Basic Loop

We can see in this loop that some definitions are used as operands. Including from previous iterations of the loop.

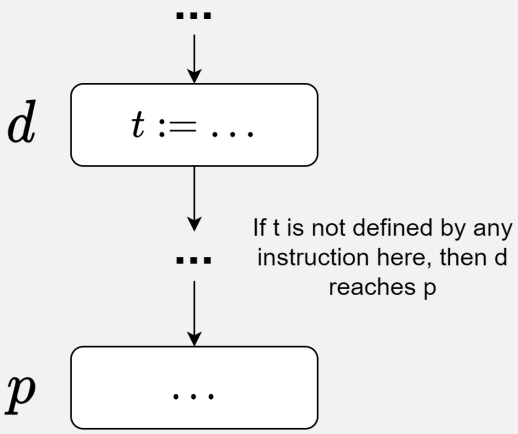


Definitions

We refer to a node/instruction defining some variable t as a definition of t.

Definition: Reach

A **definition** d reaches p if there is a path from $d \to p$ where d is not killed.



For a node n we have several sets containing other nodes/instructions:

$$Gen(n) = \{n\} = \text{the set of definitions generated by the node}$$
$$Kill(n) = \text{Set of all definitions of } t \text{ except for } n$$
$$ReachIn(n) = \text{Set of definitions reaching up to } n$$
$$ReachOut(n) = \text{Set of definitions reaching after } n$$

We can calculate the **reach-in** and **reach-out** sets as:

$$ReachIn(n) \triangleq \bigcup_{p \in Pred(n)} ReachOut(p)$$
$$ReachOut(n) \triangleq Gen(n) \cup (ReachIn(n) \setminus Kill(n))$$

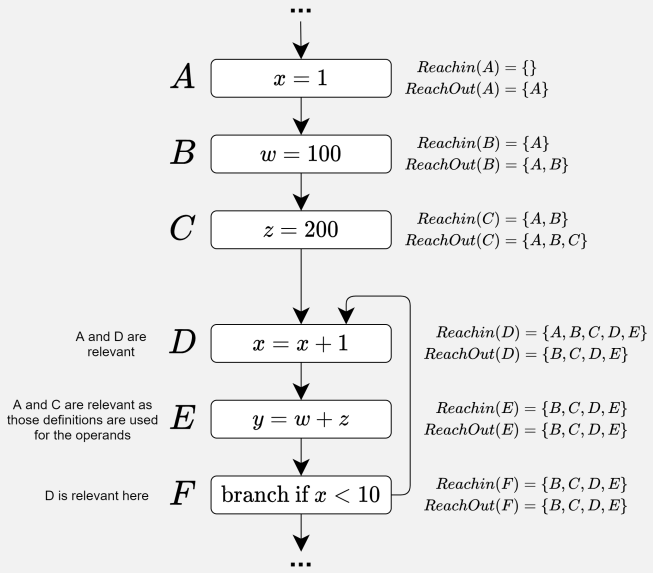
This is a **forward analysis** as definitions reach forwards.

```
1  def get_reach_sets(program):
2
3      # set all reachin, reachout sets to {}
4      for instr in program:
5          instr.reachin = []
6          instr.reachout = []
7
8      changed_set = True
9      while changed_set:
10         # until there is no change, continue to update the sets. This eventually
11         # terminates as the sets are finite, and at each step they can only remain
12         # the same size (no change) or grow.
13
14         changed_set = False
15
16         for instr in program:
17             # Reachin(n) = all predecessor's reachouts
```

```
18                new_reachin = union([pred.reachout for pred in instr.preds])
19                if new_reachin != instr.reachin:
20                    changed = True
21                    instr.reachin = new_reachin
22
23                # Reachout(n) = Gen(n) U (Reachin(n) \ Kill(n))
24                new_reachout = instr.gen + (instr.reachin - instr.kill)
25                if new_reachout != instr.reachout:
26                    changed = True
27                    instr.reachout = new_reachout
```

Form the **reaching definitions**, we can reduce each set to the **relevant reaching definitions**, by considering only the reachins that are actually used by the instruction (for operands).

Example: Basic Reaching Definitions

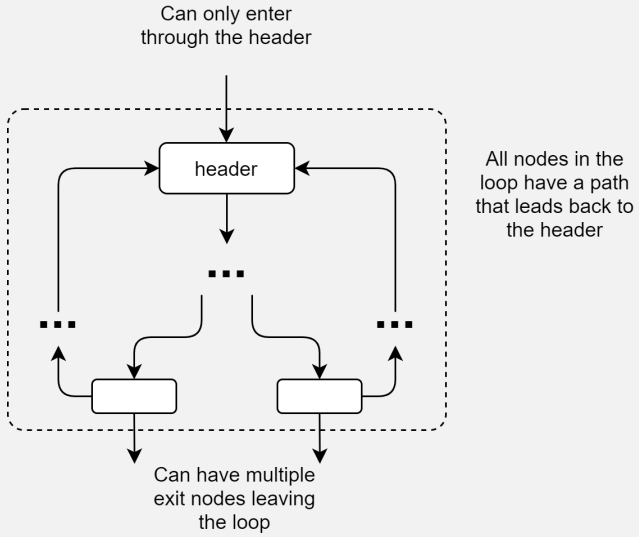


Identifying Loops

Definition: Loop

Given a set S of nodes that are part of a loop.

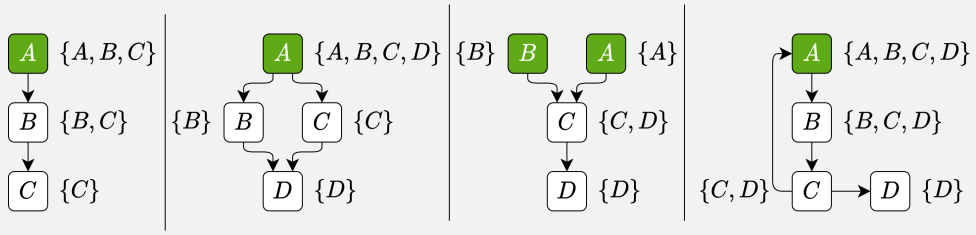
- There is a single header node $h \in S$.
- There is a path from h to any other node in S.
- There is a path from any node in S to h.
- All non-loop nodes can only directly connect to the header h, and no other nodes in the loop.



Definition: Dominator

Node d dominates node n if every path from the start node to n goes through d (Note that every Node dominates itself).

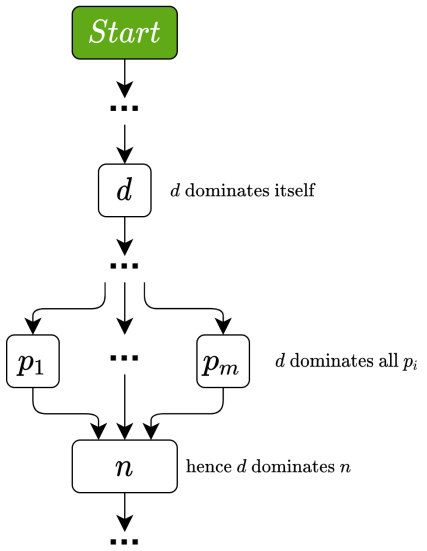
Hence for each node we can get the set of nodes it **dominates** (as shown below), or the set of nodes that **dominate** it.



To find all the nodes that **dominate** a given node:

1. Set all *Dom* sets to the set of all nodes.

2. Apply the rules (as below), as the start node will have a set of $\{start\}$ this will propagate, reducing the sizes of the sets for other nodes.
3. Once the sets stop changing, we have our solution.

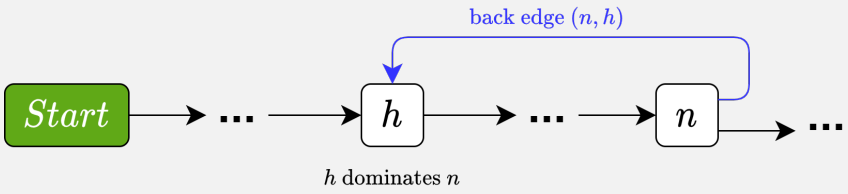


$$Doms(start) = \{start\}$$

$$Doms(n) = \{n\} \cup \left(\bigcap_{p \in preds(n)} Doms(p) \right)$$

Definition: Back Edges

An edge in the **control flow graph** from $n \to h$, where h dominates n is a **back edge**.



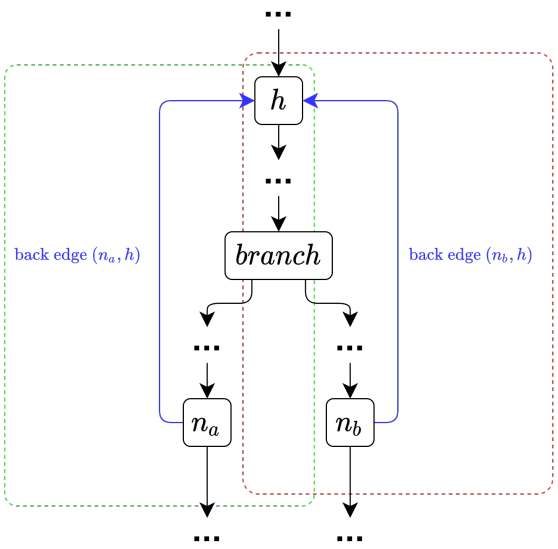
Definition: Natural Loop

The **natural loop** of a **back-edge** (n, h) is the set of nodes S such that:

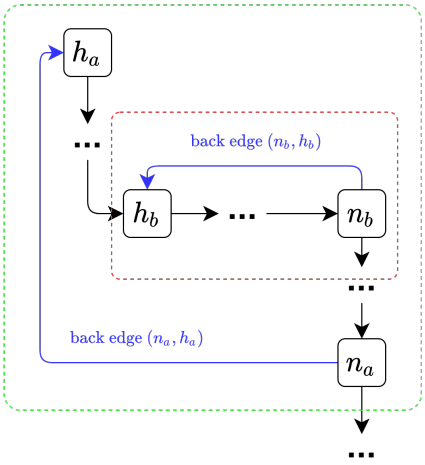
- All nodes $x \in S$ are dominated by h
- For all nodes $x \in S$ (except h), there is a path from $x \to n$ that does not contain h.

This represents a loop, with the header node h.

Multiple Loops



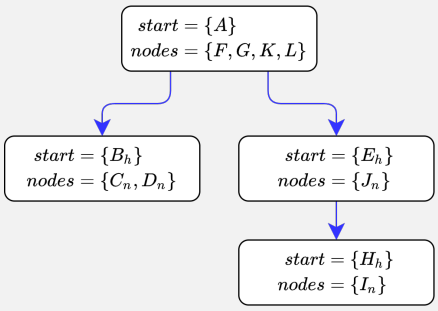
Could be one or two loops, we interpret it as a single loop.

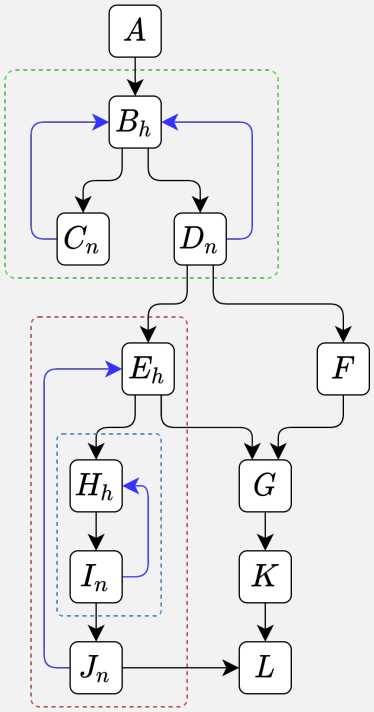


Nested Loops

Definition: Control Tree

We can construt a tree to show which loops are nested, what the headers and final nodes in each loop are.

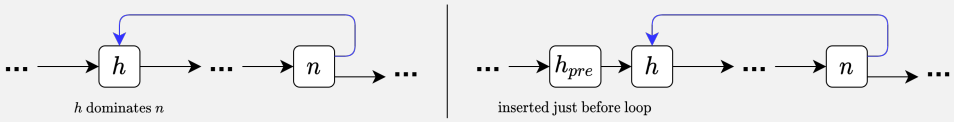




Hoisting Instructions

Definition: Pre-Header

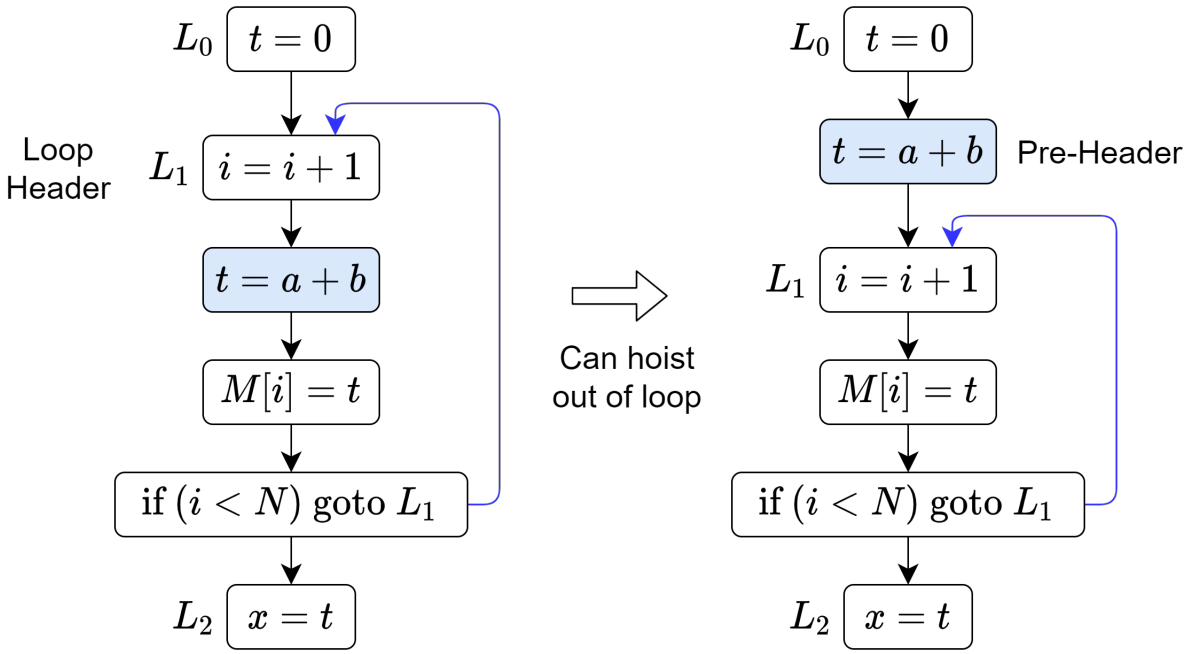
A node inserted immediately before the **header** node of a **natural loop**.

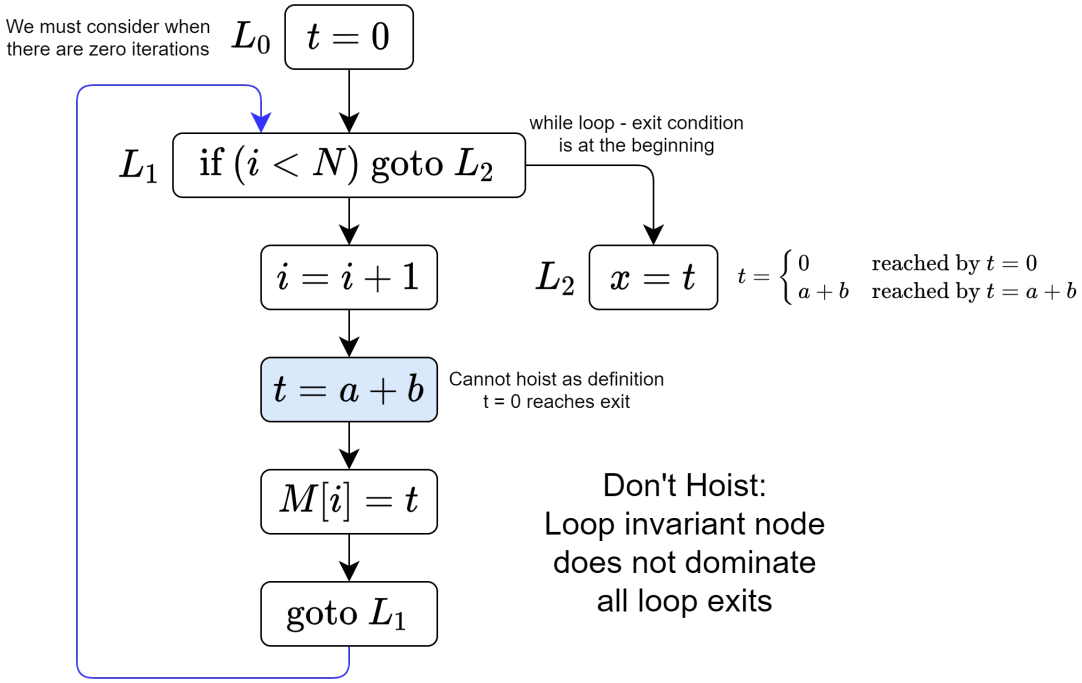


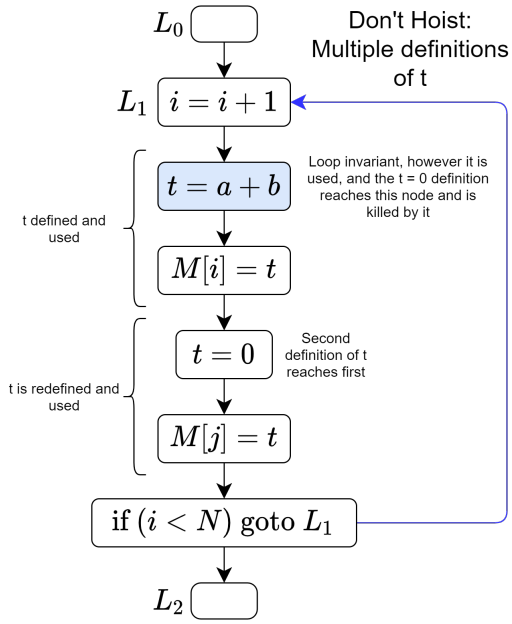
Lecture Recording

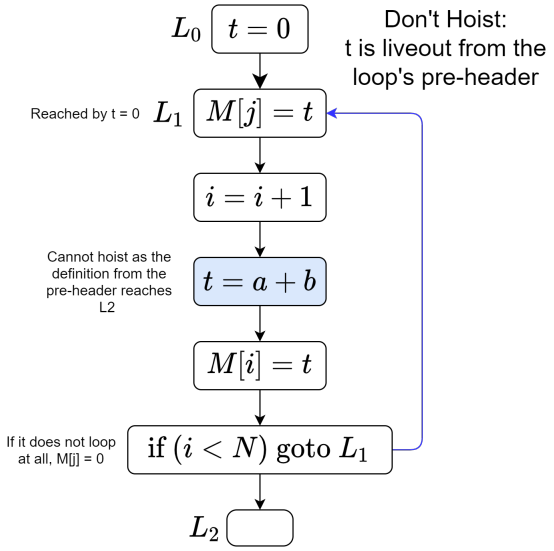
Lecture recording is available here

Consider the following examples:


L_0 $t = 0$
Loop Header
L_1 $i = i + 1$
$t = a + b$
$M[i] = t$
if $(i < N)$ goto L_1
L_2 $x = t$
Can hoist out of loop
L_0 $t = 0$
$t = a + b$
Pre-Header
L_1 $i = i + 1$
$M[i] = t$
if $(i < N)$ goto L_1
L_2 $x = t$


We must consider when there are zero iterations
L_0 $t = 0$
L_1 if $(i < N)$ goto L_2
while loop - exit condition is at the beginning
$i = i + 1$
L_2 $x = t$
$t = \begin{cases} 0 & \text{reached by } t = 0 \\ a + b & \text{reached by } t = a + b \end{cases}$
$t = a + b$
Cannot hoist as definition t = 0 reaches exit
$M[i] = t$
goto L_1
Don't Hoist:
Loop invariant node does not dominate all loop exits





Given a node $d: \ t = a \bullet b$ the conditions for hoisting are:

1. **All reaching definitions used by d occur outside the loop**
   We must not use data defined inside the loop (otherwise not invariant). Use **reaching definition** analysis for this.
2. **Loop invariant node must dominate all loop exits**
   The pre-header dominates all exits, hence in order to be moved to the pre-header, the invariant node must also. Use **dominators** analysis for this.
3. **There can only be one definition of t**
   Count the definitions.
4. **t cannot be liveout from the loop's pre-header**
   If t is live out from the pre-header, it means that t is used somewhere in the loop, and t is also defined in the loop. Hence we cannot hoist it. Use **live range** analysis for this.

The basic process of hoisting loop-invariant instructions out of a loop is:

1. Compute **dominance** sets for each node.
2. Use **dominance** sets to identify natural loop and their headers.
3. Compute the reaching sets for nodes.
4. Use relevant reaching definitions to identify loop-invariant code.
5. Attempt the loop invariant code to a **pre-header**.
6. Check that the semantics of the program are not altered.

This process can potentially be repeated (e.g hoisting out of several layers of nested loops).

Static Single Assignment

Lecture Recording

Lecture recording is available here

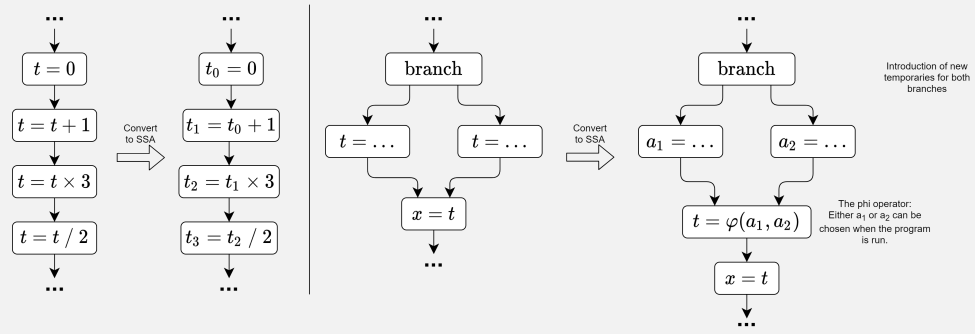
Definition: Static Single Assignment (SSA)

A representation avoiding side conditions by allowing only a single assignment to each temporary (temporaries are immutable).

- Simplifies many optimisation problems
- Requires a potentially complex translation (to and from SSA)
- Makes many optimisation passes more efficient.
- Widely used in compilers.

Each Reassignment of a variable is renamed, this splits all live ranges.

- Live ranges split (temporary is live from definition to last use, with no gaps in liveness).
- Each variable has only one reaching definition.

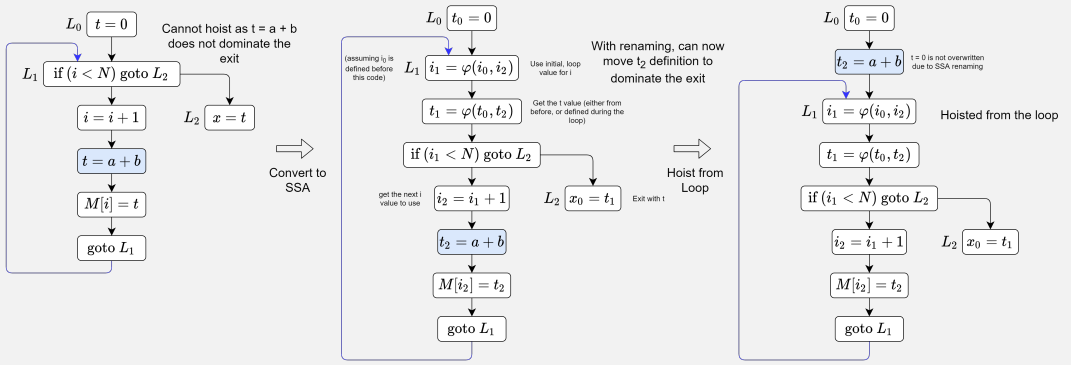


The **phi function** is used for branching. A **phi** statement $\varphi(a_1, a_2)$ means either a_1 or a_2 could be used.

By renaming variables, we can eliminate many of the loop hoisting issues (particularly for variable redefinitions in the loop)

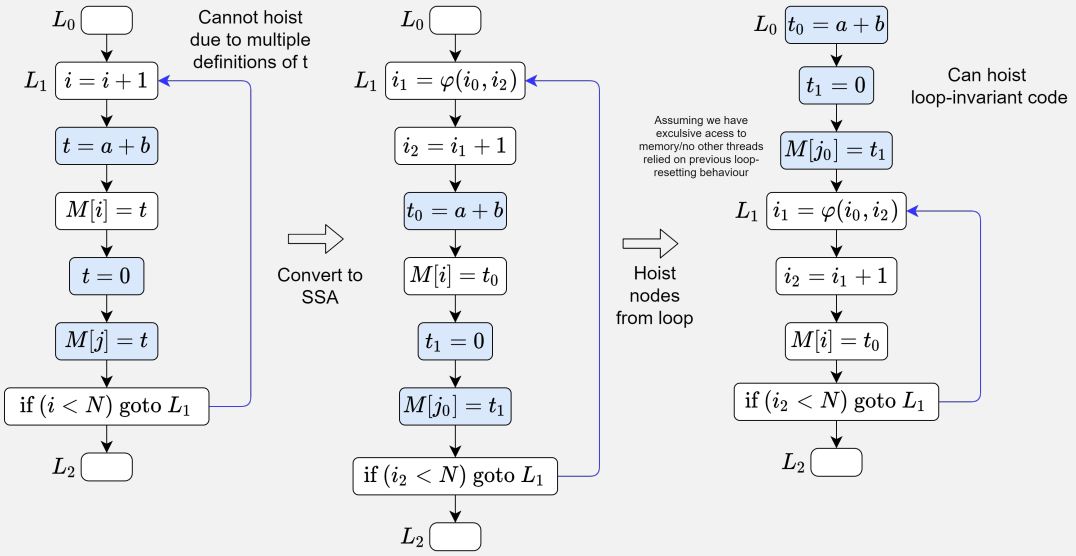
Example: Hoist a node that does not dominate the exit

The renaming done by SSA allows us to move a node so that it dominates the exit (allowing us to hoist it from the loop), without altering the semantics of the program.

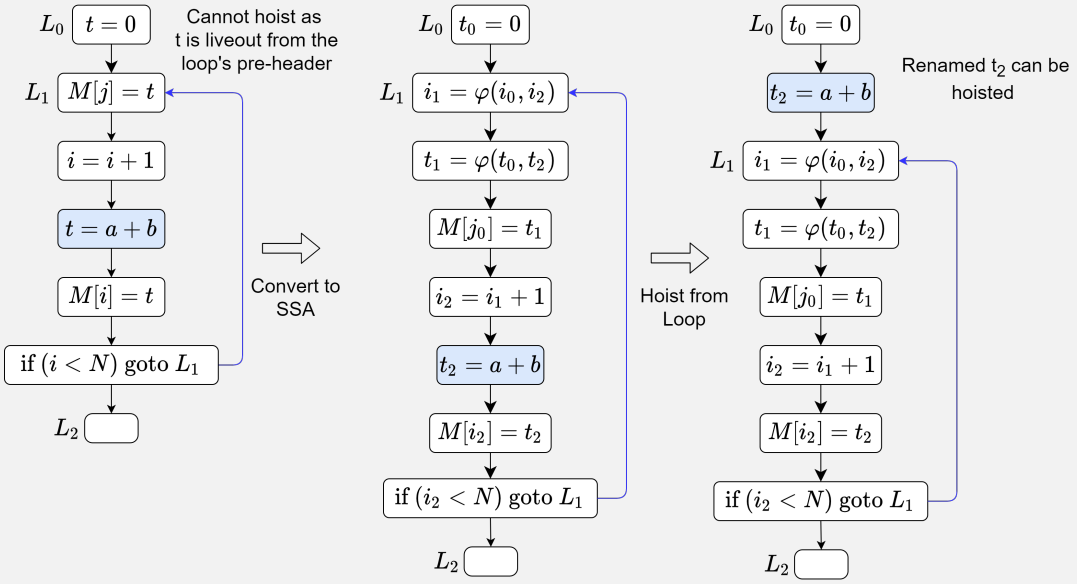


Example: Hoist with multiple definitions of t

Again, renaming from SSA allows us to hoist.



Example: Hoist when t is liveout from the Pre-Header



Once in **SSA** form, we can reassess the requirements for hoisting:

1. **All reaching definitions used by d occur outside the loop**
   (Same as prior to **SSA**). Use **reaching definitions** analysis for this.
2. **Loop invariant node must dominate all loop exits**
   No longer an issue.
3. **There can only be one definition of t**
   Guarenteed by **SSA** form.
4. **t cannot be liveout from the loop's pre-header**
   Cannot occur with **SSA** due to single assignment.

Other Compiler Optimisations

There is a wide selection of optimisation techniques that can be applied, some general (e.g inlining), while others are architecture specific (optimising instruction selection).

Induction Variables	Strength Reduction	Induction variable Elimination
Rewriting Comparisons	Array Bounds Check Elimination	Common Sub-Expressions
Partial Redundance Elimination	Loop Unrolling	Inlining
Dead Code Elimination	Rematerialisation	Tail-Call Optimisation

While conventional compilers attempt to reduce work to be done at runtime, **restructuring compilers** determine the optimal order to do a computation (hence re-structuring the computation).

For example adding parallelism to loops, or changing the order nesting of loops (e.g to allow for SMID instructions, or better cache locality).

Higher-Level Optimisations

Higher level programming language features require care to be optimised.

- **Subtype Polymorphism**
  Given some $x.f()$ we may be able to determine the method called at compile time (allowing inlining, removing loads required to go to x's **Method Lookup Table**) or be able to inline selection (if statements/switches) to determine what the type is and run code accordingly.
- **Pattern Macthing**
  A feature of languages such as Haskell, Prolog and Rust. Compile should determine the optimal order of tests (that still respects the language semantics - i.e matching order)
- **Memory Management**
  Managing heap/dunamic variables, either not doing management (left to the programmer - C), compiler using programmer provided information (and inferring as in Rust), or garbage collectors at runtime.
- **Lazy Evaluation**
  Moving evaluations of expressions to their uses (so no redundant computation is done), or constructing closures to represent complex computations.
- **Arrays**
  Optimising overloads (e.g + to concatenate arrays), and the underlying structure (e.g determining optimal array/list structure at compile time for the programmer). Allowing for efficient access to arrays containing types (e.g arrays of objects - array of pointers, or array of structs).

  Allowing array slicing (e.g $A[3..10]$) without requiring lots of copying of data, or restricting usage of the array the slice is taken from.