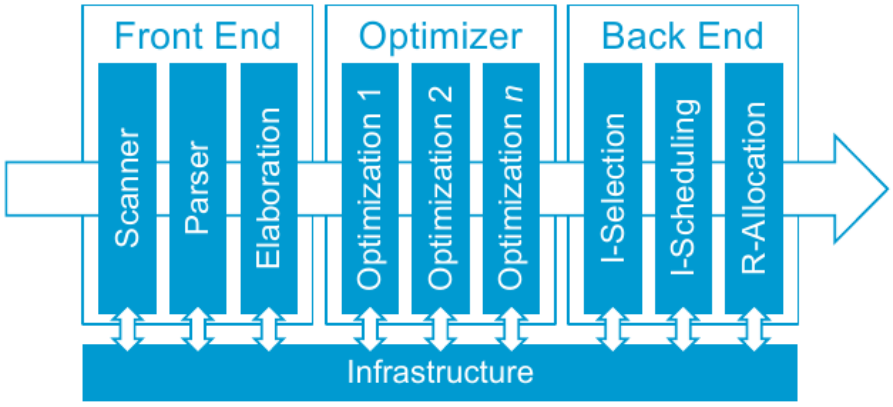
Typical Structure of a Compiler



Week	Date	Topic
1	22.10.2018	Introduction and Overview
2	29.10.2018	Scanners 1: From Regular Expression to Scanner
3	05.11.2018	Scanners 2: Implementation Techniques
4	12.11.2018	Parsers 1: Top-Down Parsing
5	19.11.2018	Parsers 2: Bottom-Up Parsing
6	26.11.2018	Elaboration: Type Checking
7	03.12.2018	Intermediate Representations
8	10.12.2018	Code Shape 1: Procedures
9	17.12.2018	Code Shape 2: Operators, Complex Values, and Control Flow
10	07.01.2019	Instruction Selection
11	14.01.2019	Instruction Scheduling
12	21.01.2019	Register Allocation
13	28.01.2019	Optimization 1: Introduction
14	04.02.2019	Optimization 2: Data-Flow Analysis
15	11.02.2019	Optimization 3: Scalar Optimizations

Contents

1 Front-End **3**
- 1.1 Scanner . . . 3
  - 1.1.1 From Regular Expressions to Scanners . . . 3
  - 1.1.2 Implementing Scanners . . . 4
- 1.2 Parser . . . 4
  - 1.2.1 Expressing Syntax: Contect-Free Grammars . . . 4
  - 1.2.2 Primitive Top-Down Parsing . . . 5
  - 1.2.3 Bottom-up aka LR(1)-Parser . . . 6
- 1.3 Elaboration: Language-specific Semantic Analysis . . . 6
  - 1.3.1 Type Systems . . . 6
  - 1.3.2 Attribute-Grammar Framework . . . 6

2 Core: Intermediate Representation and Code Shape **7**
- 2.1 Graph-based IRs . . . 7
  - 2.1.1 Parse Tree . . . 7
  - 2.1.2 Abstract Syntax Tree . . . 7
  - 2.1.3 Directed Acyclic Graph . . . 7
  - 2.1.4 Control Flow Graph . . . 7
  - 2.1.5 Dependence Graph . . . 7
  - 2.1.6 Call Graph . . . 7
- 2.2 Linear IRs . . . 7
  - 2.2.1 Stack Machine Codes . . . 7
  - 2.2.2 Three-Address Code . . . 8
- 2.3 SSA . . . 8
- 2.4 Procedures . . . 8
  - 2.4.1 Calls . . . 8
  - 2.4.2 Name Spaces . . . 8
  - 2.4.3 Addressability . . . 8
- 2.5 Storage Locations . . . 9
- 2.6 Arithmetic Expressions . . . 9
- 2.7 Boolean Expressions . . . 10
  - 2.7.1 Numerical Encoding . . . 10
  - 2.7.2 Positional Encoding . . . 10
- 2.8 Arrays . . . 10
  - 2.8.1 Vectors aka One-dimensional arrays . . . 10
  - 2.8.2 Storage Layout . . . 11
  - 2.8.3 Referencing . . . 11
- 2.9 Strings . . . 11
- 2.10 Structure References . . . 11
  - 2.10.1 Structure Layout . . . 11
  - 2.10.2 Runtime Dependent Types . . . 12
- 2.11 Control Flow . . . 12
  - 2.11.1 Conditional Execution . . . 12
  - 2.11.2 Loops . . . 12
  - 2.11.3 Case Statements . . . 12
  - 2.11.4 Procedure Calls . . . 13

3 Back End **14**
- 3.1 Instruction Selection . . . 14
  - 3.1.1 Tree Pattern-Matching . . . 14
  - 3.1.2 Peephole Optimization . . . 14
- 3.2 Instruction Scheduling . . . 15
  - 3.2.1 Local List Scheduling . . . 15
  - 3.2.2 Regional List Scheduling . . . 16
- 3.3 Register Allocation . . . 16
  - 3.3.1 Local Register Allocation . . . 16
  - 3.3.2 Global Register Allocation . . . 17

Chapter 1

Front-End

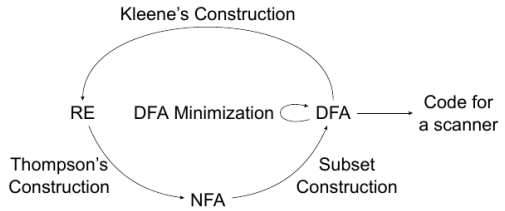
1.1 Scanner

A **Finite Automaton** (FA) is a five-tuple $(S, \Sigma, \delta, s_0, S_A)$, where
- S is the finite set of states in the recognizer, along with an error state s_e.
- Σ is the finite alphabet used by the recognizer. Typically, Σ is the union of the edge labels in the transition diagram.
- $\delta(s, c)$ is the recognizer's transition function. It maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition $s_i \xrightarrow{c} \delta(s_i, c)$.
- $s_0 \in S$ is the designated start state.
- S_A is the set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

An **Regular Expression** (RE) describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ϵ that represents the empty string. For a given RE, r, we denote the language that it specifies as $L(r)$. A RE is built up from three basic operations:
1. The **alternation**, or union, of two sets of strings, R and S, denoted $R \mid S$, is $\{ x \mid x \in R \vee x \in S \}$.
2. The **concatenation** of two sets R and S, denoted RS, contains all strings formed by prepending an element of R onto one from S, or $\{ xy \mid x \in R \wedge y \in S \}$.
3. The **Kleene closure** of a set R, denoted R^*, is $\bigcup_{i=0}^{\infty} R^i$. This is just the union of the concatenations of R with itself, zero or more times.

Finite Closure R^i: one to i occurrences of R, *e.g.*, $R^3 = (R \mid RR \mid RRR)$
Positive Closure R^+: one or more occurrences of R, *i.e.*, RR^*
Ranges $[x_0 \ldots x_n]$: abbreviate ranges *e.g.*, $[0 \ldots 3] = (0 \mid 1 \mid 2 \mid 3)$
Complement $\hat{}c$: complement of c with respect to Σ, *i.e.*, $\{\Sigma - c\}$

1.1.1 From Regular Expressions to Scanners

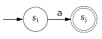


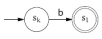
The construction of a Deterministic FA (DFA) from an RE follows three steps.
1. **Thompson's construction** derives a Nondeterministic FA (NFA) from a RE
2. **Subset construction** builds a DFA that simulates the NFA
3. **Hopcroft's algorithm** minimizes the DFA

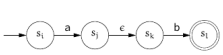
RE to NFA: Thompson's Construction

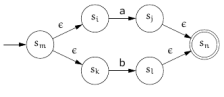
As a first step in moving from an RE to an implemented scanner, we must derive an NFA from the RE.

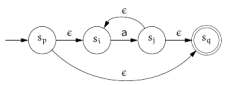
Thompson's construction builds an NFA from an RE in a straightforward way
- one template for building each NFA that corresponds to a single-letter RE
- NFA transformations for RE operators: concatenation, alternation, and closure


NFA for *a*


NFA for *b*

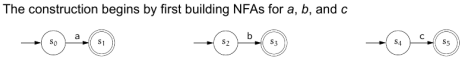

NFA for *ab*


NFA for *a | b*

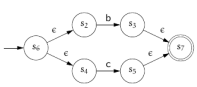

NFA for *a**

As an example, we use Thompson's Construction to derive the NFA for the RE *a (b | c)**.

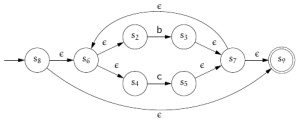
1. The construction begins by first building NFAs for *a*, *b*, and *c*



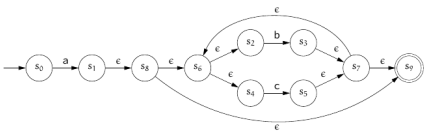
2. Since parentheses have highest precedence, it then builds the NFA for the expression enclosed in parentheses, *b | c*



3. Since closure has higher precedence than concatenation, the construction next builds the closure, *(b | c)**



4. Finally, it concatenates the NFA for *a* to the NFA for *(b | c)**



NFA to DFA: Subset Construction

The **Subset Construction** takes as input an NFA $(N, \Sigma, \delta_N, n_0, N_A)$ and produces a DFA $(D, \Sigma, \delta_D, d_0, D_A)$
- the NFA and the DFA use the same alphabet Σ
- the DFA's start state d_0 and its accepting states D_A will emerge from the construction
- the **complex part** of the construction is
  - the derivation of the set of DFA states D from the NFA states N, and
  - the derivation of the DFA transition function δ_D

```
1  q0 ← ε-closure({n0})
2  Q ← q0
3  WorkList ← {q0}
4  while WorkList ≠ ∅ do
5  |  remove q from WorkList
6  |  foreach character c ∈ Σ do
7  |  |  t ← ε-closure(Delta(q,c))
8  |  |  T[q,c] ← t
9  |  |  if t ∉ Q then
10 |  |  |  add t to Q
11 |  |  |  add t to WorkList
```

Algorithm constructs a set Q whose elements q_i are each a **subset** of N, *i.e.*, each $q_i \in 2^N$

`ε-closure(S)` examines each state $s_i \in S$ and adds to S any state reachable by following one or more ϵ-transitions from s_i

`Delta(q,c)` applies the transition function of the NFA to each state s in q and returns

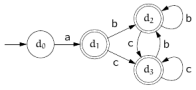
$$\bigcup_{s \in q} \delta_N(q, c)$$

The algorithm takes the following steps
1. The initialization sets q_0 to `ε-closure`$(\{n_0\})$, which is just n_0
2. The 1st iteration computes `ε-closure(Delta(`q_0`,a))`, which contains six NFA states, and `ε-closure(Delta(`q_0`,b))` and `ε-closure(Delta(`q_0`,c))`, which are empty
3. The 2nd iteration examines q_1 and produces two configurations, named q_2 and q_3
4. The 3rd iteration examines q_2 and constructs two configurations, which are identical to q_2 and q_3
5. The 4th iteration examines q_3 and it reconstructs q_2 and q_3 (like the 3rd iteration)

The following table sketches the steps that the subset construction algorithm follows.

Set Name	DFA States	NFA States	`ε-closure(Delta(`$\circ$`))` a	b	c
q_0	d_0	n_0	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	$\emptyset$	$\emptyset$
q_1	d_1	$\{n_1, n_2, n_3, n_4, n_6, n_9\}$	$\emptyset$	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$
q_2	d_2	$\{n_5, n_8, n_9, n_3, n_4, n_6\}$	$\emptyset$	q_2	q_3
q_3	d_3	$\{n_7, n_8, n_9, n_3, n_4, n_6\}$	$\emptyset$	q_2	q_3

Finally, we can construct the resulting DFA as follows
- the states correspond to the DFA states from the table
- the transitions are given by the `Delta` operations that generate those states
- since the sets q_1, q_2, and q_3 all contain n_9 (the accepting state of the NFA), all three become accepting states in the DFA



DFA to minimal DFA: Hopcroft Algorithm

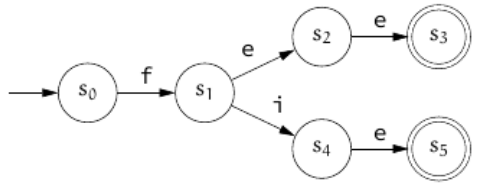
```
1  T ← {D_A, {D − D_A}}
2  P ← ∅
3  while P ≠ T do
4  |  P ← T
5  |  T ← ∅
6  |  foreach set p ∈ P do
7  |  |  T ← T ∪ Split(p)
```

```
1  Split(S)
2  |  foreach c ∈ Σ do
3  |  |  if c splits S into s1 and s2 then
4  |  |  |  return {s1, s2}
5  |  return S
```

The algorithm constructs a set partition $P = \{p_1, p_2, \ldots, p_m\}$ of the DFA states by grouping together DFA states that have the **same behavior**.

Two DFA states $d_i, d_j \in p_s$ have the same behavior in response to all input characters

$$\forall\, p_s \in P, d_i, d_j \in p_s; c \in \Sigma : d_i \xrightarrow{c} d_x, d_j \xrightarrow{c} d_y \implies d_x = d_y$$



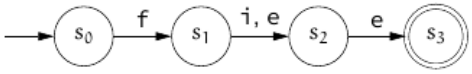
The algorithm takes the following steps

1. Initialization separates accepting from nonaccepting states: $\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$
2. Step 1 examines $\{s_3, s_5\}$: neither state has an exiting transition, *i.e.*, no split occurs
3. Step 2 examines $\{s_0, s_1, s_2, s_4\}$: on charachter `e`, it splits $\{s_2, s_4\}$ out of the set
4. Step 3 examines $\{s_0, s_1\}$: on charachter `f`, it splits $\{s_1\}$ out of the set
5. Step 4 makes a final pass over the current partition $\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$: no more splits occur and, therefore, the fix-point is reached

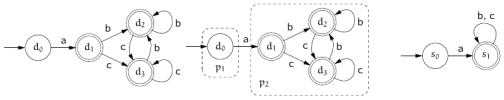
Note We assume that the various loops in the algorithm iterate over the sets of `P` and over the characters in $\Sigma = \{e, f, i\}$ in order.

The following summarizes the significant steps that occur in minimizing this DFA.

Step	Current Partition	Examines Set	Examines Char	Action
0	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	—	—	—
1	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_3, s_5\}$	*all*	*none*
2	$\{\{s_3, s_5\}, \{s_0, s_1, s_2, s_4\}\}$	$\{s_0, s_1, s_2, s_4\}$	`e`	*split* $\{s_2, s_4\}$
3	$\{\{s_3, s_5\}, \{s_0, s_1\}, \{s_2, s_4\}\}$	$\{s_0, s_1\}$	`f`	*split* $\{s_1\}$
4	$\{\{s_3, s_5\}, \{s_0\}, \{s_1\}, \{s_2, s_4\}\}$	*all*	*all*	*none*



Final DFA with renumbered states



Original DFA — Initial Partition — Minimal DFA

1.1.2 Implementing Scanners

To find the **longest word** that matches one of the REs, the DFA should run until it reaches the point where the current state s has no outgoing transition on the next character.

Which RE was matched at this point?

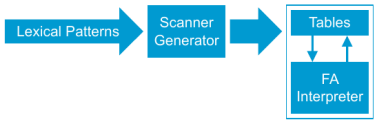
1. if s is an **accepting state**: report the recognized word and its syntactic category
2. if s is an **nonaccepting state**
   - if possible, roll back to most recent accepting state to match **longest valid prefix**
   - otherwise, report an **error** since no prefix of the input string is a valid word

Scanner generator

1. creates NFA for each RE defined by compiler writer for syntactic classes
2. joins all resulting NFAs by ϵ-transitions
3. translates NFA into a corresponding DFA
4. minimizes DFA
5. converts DFA into executable code

Having covered Steps 1 to 4 already, we now look at three implementation strategies for Step 5: **table-driven scanners**, **direct-coded scanners**, and **hand-coded scanners**.

Table-Driven Scanner



The table-driven approach uses a generic **skeleton scanner** for control and a set of **three generated tables** that encode language-specific knowledge

- classifier table `CharCat` maps each $c \in \Sigma$ to a category
- transition table δ encodes the transitions of the underlying DFA
- token type table `Type` maps each state s_i of the DFA to a token type

The code generated for the skeleton scanner, *i.e.*, algorithm `NextWord()` on Slide 103, is divided into **four** sections.

- initializations
- a scanning loop that models the DFA's behavior
- a roll back loop in case the dfa overshoots the end of the token
- a final section that interprets and reports the results

Algorithm `NextWord()`

```
1  state ← s0
2  lexeme ← ""
3  clear stack
4  push(bad)
5  while state ≠ se do
6  |  NextChar(char)
7  |  lexeme ← lexeme + char
8  |  if state ∈ SA then
9  |  |  clear stack
10 |  push(state)
11 |  cat ← CharCat[char]
12 |  state ← δ[state,cat]
13 while state ∉ SA ∧ state ≠ bad do
14 |  state ← pop()
15 |  truncate lexeme
16 |  RollBack()
17 if state ∈ SA then
18 |  return Type[state];
19 else
20 |  return invalid
```

Notice the similarity between this code and the fragment shown on Slide 57.

Table-Driven Scanners

Tables `CharCat` and δ encode the DFA. To update the state, a **two-step translation** is performed.

1. character → category
2. current state and category → new state

This approach lets the scanner use a compressed transition table.

`r`	`0, ..., 9`	`EOF`	**Other**
Register	*Digit*	*Digit*	*Other*

Classifier Table `CharCat`

	Register	Digit	Other
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Transition Table δ

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

Token Type Table `Type`

Larger character sets, *e.g.*, Unicode, may need a more complex data structure to represent `CharCat`.

The so-called **Maximal Munch Scanner** avoids excess roll back by marking dead-end transitions as they are popped from the stack.

It differs from the scanner on Slide 103 in **three** important ways.

1. a global counter `InputPos` to record position in the input stream
2. a two-dimensional bit-array `Failed` to record dead-end transitions
3. additional routine `InitializeScanner()` to initialize these two variables

⟶ Over time, the maximal munch scanner records specific ⟨state, input position⟩-pairs that cannot lead to an accepting state.

Direct-Coded Scanner

What could we do to **improve the performance** of a table-driven scanner?

- read a character
- compute the next DFA transition

Direct-coded scanners reduce the cost of computing DFA transitions by replacing the explicit representation of the DFA's state and transition graph with an implicit one.

- simplifies the two-step, table-lookup computation
- eliminates the memory references entailed in that computation
- allows other specializations

⟶ The resulting scanner has the same functionality as the table-driven scanner, but with a lower overhead per character.

At the heart of a direct-coded scanner is an alternate implementation of the central `while` loop of the table-driven scanner.

```
1 while state ≠ se do
2 |  ...
3 |  cat ← CharCat[char]
4 |  state ← δ[state,cat]
```

For each character, the table-driven scanner does **two** table lookups, involving address computations.

- `CharCat`: $@\texttt{CharCat}_0 + \mathrm{i} \times w$
- δ: $\delta_0 + (\texttt{state} \times |\text{columns in } \delta| + \texttt{cat}) \times w$

Even though both lookups take $O(1)$ time, they impose **constant-cost overheads** that a direct-coded scanner can avoid.

⟶ Rather than representing the current DFA state and the transition diagram explicitly, a direct-coded scanner has a specialized code fragment to implement each state.

```
  // State sinit
1  lexeme ← ""
2  clear stack
3  push(bad)
4  goto 5
  // State s0
5  NextChar(char)
6  lexeme ← lexeme + char
7  if state ∈ SA then
8  |  clear stack
9  push(state)
10 if char = 'r' then
11 |  goto 14
12 else
13 |  goto 32
  // State s1
14 NextChar(char)
15 lexeme ← lexeme + char
16 if state ∈ SA then
17 |  clear stack
18 push(state)
19 if '0' ≤ char ≤ '9' then
20 |  goto 23
21 else
22 |  goto 32
  // State s2
23 NextChar(char)
24 lexeme ← lexeme + char
25 if state ∈ SA then
26 |  clear stack
27 push(state)
28 if '0' ≤ char ≤ '9' then
29 |  goto 23
30 else
31 |  goto 32
  // State sbad
32 while state ∉ SA ∧
       state ≠ bad do
33 |  state ← pop()
34 |  truncate lexeme
35 |  RollBack()
36 if state ∈ SA then
37 |  return Type[state];
38 else
39 |  return invalid
```

Hand-Coded Scanner

Many compilers use hand-coded scanners. A hand-coded scanner can further reduce the overhead of the interfaces between the scanner and the rest of the system.

In the following, we study two possible optimizations

- buffering the input stream
- generating lexemes on-the-fly

In order to reduce the I/O cost per character, we can use buffered I/O

- each read operation returns a longer string of characters or buffer
- the scanner then indexes through the buffer by maintaining a pointer into the buffer
- `NextChar` fills the buffer and tracks the current location in the buffer

The cost of reading a full buffer of characters has two components

- a large fixed overhead
- and a small per-character cost

A buffer and pointer scheme **amortizes** the fixed costs of the read over many single-character fetches.

1.2 Parser

Parsing is finding a derivation of a Grammar G that produces the input token stream s.

1.2.1 Expressing Syntax: Contect-Free Grammars

Context-Free Grammars

Formally, a **Context-Free Grammar** G is a quadruple (T, NT, S, P) where

- T is the set of terminal symbols, or words, in the language $L(G)$.
- NT is the set of nonterminal symbols that appear in the productions of G.
- S is a nonterminal designated as the **goal symbol** or **start symbol** of the grammar. S represents the set of sentences in $L(G)$.
- P is the set of productions or rewrite rules in G. Each rule in P has the form $NT \rightarrow (T \cup NT)^+$, *i.e.*, it replaces a **single nonterminal** with a string of one or more grammar symbols.

Ambigous Grammars: A grammar G is ambiguous if there exists a sentence which has more than one rightmost or leftmost derivation. If a Grammar is ambiguous it needs to be altered such that it is not, as programs shall be unambiguous.

Encoding meaning: Besides ambiguity, other rules are needed to encode e.g. precedence rules fot arith. expressions, in array subscripts, for type casts and assignment.

1.2.2 Primitive Top-Down Parsing

A **top-down parser** begins with the root of the parse tree and systematically extends the tree downward until its leaves match the classified words returned by the scanner.

General top-down parsing algorithm

1. select a nonterminal symbol on the lower fringe of the partially built parse tree
2. replace symbol with children corresponding to right-hand side of one of its productions
3. repeat this process until
   a) fringe only contains terminal symbols and input stream has been exhausted
      → parsing succeeds
   b) clear mismatch occurs between fringe and input stream
      → backtrack and try another production
      → if there are no more possible productions, report an error

```
1  root ← node for the start symbol S
2  focus ← root
3  push(null)
4  word ← NextWord()
5  while true do
6  |  if focus is a nonterminal then
7  |  |  pick next rule to expand focus (A → β1, β2, ..., βn)
8  |  |  build nodes for β1, β2, ..., βn as children of focus
9  |  |  push(βn, βn−1, ..., β2)
10 |  |  focus ← β1
11 |  else if word matches focus then
12 |  |  word ← NextWord()
13 |  |  focus ← pop()
14 |  else if word = eof and focus = null then
15 |  |  accept the input and return root
16 |  else
17 |  |  backtrack
```

If the focus is a terminal symbol that does not match the input, the parser must backtrack.

The implementation of "*backtrack*" is straightforward

1. set `focus` to its parent in the partially-built parse tree and disconnects its children
2. if an untried rule remains with `focus` on its left-hand side
   - perform Lines 7 to 10 of algorithm on Slide 146
3. if no untried rule remains
   - move up another level and try again
   - if out of possibilities, report a syntax error and quit

Backtracking increases the asymptotic cost of parsing. In practice, it is an expensive way to discover syntax errors.

One key insight makes top-down parsing efficient: a large subset of the context-free grammars can be parsed **without** backtracking!

Eliminating Left-Recursion

$$\begin{array}{rcl} A & \to & A\alpha \\ & | & \beta \end{array} \qquad\qquad \begin{array}{rcl} A & \to & \beta A' \\ A' & \to & \alpha A' \\ & | & \epsilon \end{array}$$

The translation from (direct) left recursion to right recursion is mechanical

- introduce a new nonterminal A' and transfer the recursion onto A'
- add a rule $A' \to \epsilon$, where ϵ represents the empty string

So far, we have only tackled **direct** left recursion. There can also be **indirect** left recusion, which is caused by chains of "transitive" productions.

$$\alpha \to \beta, \beta \to \gamma, \text{and } \gamma \to \alpha\delta \implies \alpha \to^+ \alpha\delta$$

Indirect left recursion can be obscured by a long chain of productions. Therefore, we need a **more systematic approach** to convert indirect left recursion into right recursion.

We can eliminate all left recursion from a grammar using **two** simple techniques

- forward substitution to convert indirect left recursion into direct left recursion
- rewriting direct left recursion as right recursion

```
1 impose an arbitrary order on nonterminals A1, A2, ..., An
2 for i ← 1 to n do
3 |  for j ← 1 to i − 1 do
4 |  |  if there is a production Ai → Aj γ and Aj → δ1|δ2|...|δn then
5 |  |  |  replace Ai → Aj γ with a set of productions Ai → δ1γ|δ2γ|...|δnγ
6 |  rewrite the productions to eliminate any direct left recursion on Ai
```

Note This algorithm assumes that the original grammar has no cycles ($A \to^+ A$) and no ϵ-productions.

Backtrack-Free Parser: A Parser is backtrack free if the underlying context-free grammar is constructed such that the leftmost top-down derivation can always be predicted with a lookahead of one.

FIRST

For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

The domain of FIRST is the set of grammar symbols, $T \cup NT \cup \{\epsilon, \texttt{eof}\}$ and its range is $T \cup \{\epsilon, \texttt{eof}\}$.

$\alpha \in T \cup \{\epsilon, \texttt{eof}\}$ FIRST(α) has exactly one member α
$A \in NT$ FIRST(A) contains all terminal symbols that can appear as the leading symbol in any sentential form derived from A

```
1  foreach α ∈ T ∪ {ε, eof} do
2  |  FIRST(α) ← α
3  foreach A ∈ NT do
4  |  FIRST(A) ← ∅
5  while FIRST sets are still changing do
6  |  foreach p ∈ P of the form A → β1β2...βk, where βi ∈ T ∪ NT do
7  |  |  rhs ← FIRST(β1) − {ε}
8  |  |  i ← 1
9  |  |  while ε ∈ FIRST(βi) and i ≤ k − 1 do
10 |  |  |  rhs ← rhs ∪ (FIRST(βi+1) − {ε})
11 |  |  |  i ← i + 1
12 |  |  if i = k and ε ∈ FIRST(βk) then
13 |  |  |  rhs ← rhs ∪ {ε}
14 |  |  FIRST(A) ← FIRST(A) ∪ rhs
```

Dealing with ϵ-productions

- parser should apply the ϵ-production if the lookahead symbol is **not a member** of the FIRST set of any other alternative
- to differentiate between **legal input** and **syntax errors**, it needs to know which words can appear as the leading symbol after a valid application of an ϵ-production

FOLLOW

For a nonterminal A, FOLLOW(A) contains the set of words that can occur immediately after A in a sentence.

```
1  foreach A ∈ NT do
2  |  FOLLOW(A) ← ∅
3  FOLLOW(S) ← {eof}
4  while FOLLOW sets are still changing do
5  |  foreach p ∈ P of the form A → β1β2...βk, where βi ∈ T ∪ NT do
6  |  |  lhs ← FOLLOW(A)
7  |  |  for i ← k down to 1 do
8  |  |  |  if βi ∈ NT then
9  |  |  |  |  FOLLOW(βi) ← FOLLOW(βi) ∪ lhs
10 |  |  |  |  if ε ∈ FIRST(βi) then
11 |  |  |  |  |  lhs ← lhs ∪ (FIRST(βi) − {ε})
12 |  |  |  |  else
13 |  |  |  |  |  lhs ← lhs ∪ FIRST(βi)
14 |  |  |  else
15 |  |  |  |  lhs ← FIRST(βi)          // Note that since βi ∉ NT, FIRST(βi) = {βi}
```

Backtrack-Free Grammar

For a production $A \to \beta$, we define its augmented FIRST set, FIRST$^+$.

$$\text{FIRST}^+(A \to \beta) = \begin{cases} \text{FIRST}(\beta) & \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

A grammar is **backtrack-free** if the following property holds for any nonterminal A with multiple right-hand sides, *i.e.*, $A \to \beta_1|\beta_2|\ldots|\beta_n$.

$$\forall\, 1 \le i, j \le n, i \ne j : \text{FIRST}^+(A \to \beta_i) \cap \text{FIRST}^+(A \to \beta_j) = \emptyset$$

We can left factor any set of rules that has alternate right-hand sides with a common prefix.

$$A \to \alpha\beta_1|\alpha\beta_1|\ldots|\alpha\beta_n|\gamma_1|\gamma_2|\ldots|\gamma_m$$

The transformation introduces a new nonterminal B to represent the alternate suffixes for α and rewrites the original productions according to the following pattern.

$$\begin{array}{rcl} A & \to & \alpha B|\gamma_1|\gamma_2|\ldots|\gamma_m \\ B & \to & \beta_1|\beta_1|\ldots|\beta_n \end{array}$$

To left factor a complete grammar, we must inspect each nonterminal, discover common prefixes, and apply the transformation in a **systematic** way.

Top-Down aka LL(1)-Parser:

1. scan input **L**eft to right
2. construct **L**eftmost derivations
3. use a lookahead of **1** symbol

Recursive descend Parser:

1. for each non-terminal: Construct procedure to recognize rhs
2. in each procedure call other procedures to derivate until terminal(s)
3. recognize terminals by direct string matching

Most common variant: table-driven skeleton parser:

```
1  word ← NextWord()
2  stack.push(eof)
3  stack.push(S)
4  focus ← stack.peek()
5  loop
6  |  if focus = eof and word = eof then report success and exit the loop
7  |  else if focus ∈ T or focus = eof then
8  |  |  if focus matches word then
9  |  |  |  stack.pop()
10 |  |  |  word ← NextWord()
11 |  |  else report an error looking for symbol at top of stack
12 |  else
13 |  |  if table[focus, word] is A → B1B2...Bk then
14 |  |  |  stack.pop()
15 |  |  |  for i ← k down to 1 do
16 |  |  |  |  if Bi ≠ ε then stack.push(Bi)
17 |  |  else report an error expanding focus
18 |  focus ← stack.peek()
```

Given a nonterminal A and a lookahead symbol w, `table[`A`,`w`]` specifies the correct expansion.

The algorithm to build `table` (shown on the right) is straightforward.

If the grammar meets the backtrack-free condition, the algorithm will produce the correct `table` in $\mathcal{O}(|P| \times |T|)$ time.

If the grammar is not backtrack-free, the algorithm will try to assign more than one production to some elements of `table`.

```
1  build FIRST, FOLLOW, and FIRST+ sets
2  foreach A ∈ NT do
3  |  foreach w ∈ T do
4  |  |  table[A, w] ← error
5  |  foreach p ∈ P with form A → β do
6  |  |  foreach w ∈ FIRST+(A → β) do
7  |  |  |  table[A, w] ← p
8  |  |  if eof ∈ FIRST+(A → β) then
9  |  |  |  table[A, eof] ← p
```

Another Variant: direct-coded Parser:

1. build $FIRST$, $FOLLOW$ and $FIRST^+$ sets
2. iterate trough grammar as in the table driven variant
3. for each non-terminal, generate procedure that recognizes rhs

1.2.3 Bottom-up aka LR(1)-Parser

1. scan input **L**eft to right
2. construct reversed **R**ightmost derivations
3. use a lookahead of **1** symbol

A **bottom-up parser** builds a parse tree starting from its leaves and working toward its root.

General bottom-up parsing algorithm

1. find a handle $\langle A \to \beta, k \rangle$ on the upper frontier of this partially completed parse tree
2. replace the occurence of β at k with A
3. repeat this process until
   a) frontier is reduced to a single node that represents the grammar's goal symbol
      → parser has found a derivation
      → parsing succeeds, if there are no more words in the input stream
   b) cannot find a handle
      → parser cannot build a derivation for the input stream
      → report an error

The critical step in a bottom-up parser is to find the next handle efficiently.

Table-driven LR(1) parser

- uses a handle-finding automaton, encoded into two tables called `Action` and `Goto`
- shifts symbols onto the stack until the automaton finds the right end β of a handle
- reduces by the production $A \to \beta$ in the handle
  - pops the symbols in β from the stack
  - pushes the corresponding lefthand side A onto the stack

The `Action` and `Goto` tables thread together **shift** and **reduce** actions in a grammar-driven sequence that finds a reverse rightmost derivation, if one exists.

Using a stack lets the LR(1) parser make the position k in the handle $\langle A \to \beta, k \rangle$ be constant and implicit.

```
stack.push($)
stack.push(s0)
word ← NextWord()
loop
  state ← stack.peek()
  if Action[state, word] = "reduce A → β" then
    stack.pop(2 × |β| symbols)
    state ← stack.peek()
    stack.push(A)
    stack.push(Goto[state, A])
  else if Action[state, word] = "shift si" then
    stack.push(word)
    stack.push(si)
    word ← NextWord()
  else if Action[state, word] = "accept" then
    break()
  else
    report an error
report success
```

Again most common variant: table-driven skeleton parser:

LR(1) parsers take time proportional to…

- the length of the input (one shift per word returned from the scanner) **and**
- the length of the derivation (one reduce per step in the derivation)

In general, we cannot expect to discover the derivation for a sentence in any fewer steps.

Building LR(1) Tables

To construct the `Action` and `Goto` tables, the parser generator builds a model of the handle-recognizing automaton, called the **canonical collection of sets of LR(1) items**.

The model represents all of the possible states of the parser and the transitions between those states. It is reminiscent of the subset construction (*cf.* Slide 81).

1.3 Elaboration: Language-specific Semantic Analysis

1.3.1 Type Systems

Type: A type specifies a set of properties held in commin by all objects of that type. Types can be specified by membership (e.g. defining the set of interges by specifying the range) or by rules (e.g. the class keyword defines a type).

Purpose: The set of types in a programming language along with the rules that use types to specify program behaviuor are called type system. It can be seen as a second kind of vocabulary besides the syntactic grammar to describe form and behaviour of a valid program.
This information is used for ensuring runtime safety, correct usage and connection between functions, modules and (algebraic) data types, improving expressiveness (e.g. operator overloading) and certain optimizations like vectorization. An example for runtime safety is implicit conversion of compatible types (float and int). A language is calles **strongly typed** if every expression can be assigned an unambiguous type, else it's either **untyped or weakly typed**. If every expression can be typed at compile time and is checked at compile time it's called **statically typed**. If an expression can only be typed and checked at runtime it's called **dynamically typed**. From a perfomance perspective, dynamic typing is really bad.

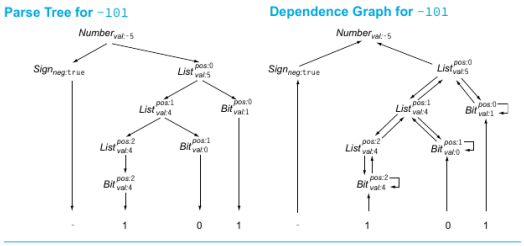
Components

1. Basic/Primitive Types
   Mostly contains numbers (int, float), characters (ASCII/UTF-8), booleans (T&F), often also Pointers
2. Composite/User-defined Types
   Like Arrays, Maps, Lists, Tuples, Objects, Trees, Stacks, Queues, Enums, Structs, Unions
3. Type Equivalence, Compatibility operators
   Two general approaches: name equivalence, structural equivalence
4. Type Inference Rules
   Specify the mapping and compatibilities between operand types and the result type for each operator, function, … If some information is unknown at compile time it needs to be deferred to runtime, e.g. if it's unsure wether a certain expression is compatible depending on the state of the system.
   Especially function calls which often require a type signature depend on the type system and force to programmer to obey reasonable rules without complete prior knowledge

1.3.2 Attribute-Grammar Framework

- Formalism to perform contect-sensitive analysis
- Consists of a context-free grammar augmented by a set of rules
  - A rule defines an attribute of a Symbol (node in the parse tree)
  - Each rule specifies the value of one attribute in terms of literals and other attributes
  - rules are functional, i.e. dont imply a specific order
  - All rules together form a set of dependencies which form an attribute-dependence graph

E.g. Terminal Number with Attribute value



The bidirectional flow of values that we noted earlier shows up in the dependence graph. We distinguish between attribute attributes based on the direction of value flow.

Synthesized Attributes

- defined by bottom-up information flow
- draws values from the node itself, its descendants, and constants

Inherited Attributes

- defined by top-down and lateral information flow
- draws values from the node itself, its parent and siblings, and constants

Note The syntax of the attribution rules allows a rule to reference its own result, either directly or indirectly. Such **circular** attribute grammars are ill-formed.

Evaluators for the AGF can be of the following approaches:

- Dynamic: Use the structure of a particular attributed parse tree to determine evaluation order e.g. if tree is left deep do bottom-up DFS, if tree is bushy do top-down BFS or generating folds depending on the attribute-dependence graph
- Rule-based: Rely on static analysis to execute evaluation, e.g. extracting variables which always take on the same value
- Oblivious: consider neither the grammar nor the parse tree to determine order of execution

Circular attribute grammars can give rise to cyclic attribute-dependence graphs. Our models for evaluation fail when the dependence graph contains a cycle.

A compiler that uses attribute grammars can use **two** approaches to handle circularity.

Avoidance

- restrict to a class of attribute grammars that cannot give rise to circular dependences
- for example, restricting the grammar to use only synthesized and constant attributes
- more general classes exist, *e.g.*, strongly noncircular attribute grammars,

Evaluation

- assign a value to every attribute, even those involved in cycles
- evaluator might iterate over the cycle and assign appropriate or default values
- this approach avoids the problems associated with a failure to fully attribute the tree

In practice, most attribute-grammar systems are restrict to **noncircular** grammars.

Attribute grammar are a **general** framework that has many applications in a compiler.

- **evaluation rules** as used in the example with signed binary numbers
- **type inference rules** to identify type-related errors
- rules to **estimate execution time** in cycles
- rules for **ad-hoc syntax-directed translation**

Oberon-0 Compiler Project

The Oberon-0 compiler gives rise to the **first two** applications. Evaluation rules are used to calculate the value of arithmetic expressions over constants. Type inference rules are used to enfore the language rules for typing in Oberon-0.
The **visitor** pattern can be used to annotate the abstract syntax tree with attributes.

Chapter 2

Core: Intermediate Representation and Code Shape

There are many ways to map source code to a sequence of operations in a certain target ISA, resulting in variations in speed, memory utilization, register usage, energy consumption, binary size, . . .

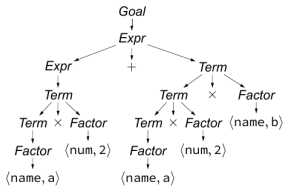
An Intermediate Representation conserves knowledge and informations about the code to compile. One annotated or multiple IRs may be used during compilation. IRs differ in structure and level of abstraction.

2.1 Graph-based IRs

2.1.1 Parse Tree

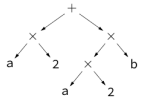
The **parse tree** is a graphical representation for complete derivation, with a node for each grammar symbol in the derivation.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* − *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name



2.1.2 Abstract Syntax Tree

The **abstract syntax tree** (AST) is a contraction of the parse tree that omits most nodes for nonterminals, but retains its essential structure.



The AST is a near-source-level representation. Because of its rough correspondence to a parse tree, the parser can build an AST directly.

ASTs have been used in many practical compiler systems.

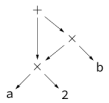
2.1.3 Directed Acyclic Graph

A **directed acyclic graph** (DAG) is a contraction of the AST that avoids this duplication. Identical subtrees are instantiated once, with multiple parents.

DAGs are used in real systems for **two** reasons
- reduce the memory footprint of compiler
- expose and prevent redundancies in compiled code

However, the compiler must **prove** that a shared expression's value cannot change between uses.

If the expression contains neither assignment nor calls to other procedures, the proof is easy.



2.1.4 Control Flow Graph

The simplest unit of control flow in a program is a **basic block**
- maximal length sequence of straightline, or branch-free, code
- sequence of operations that always execute together, unless an exception is raised
- control always enters a basic block at its first operation and exits at its last operation

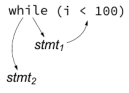
Control-Flow Graph

A control-flow graph (CFG) models the flow of control between the basic blocks in a program. A CFG is a directed graph $G = (N, E)$.
- each node $n \in N$ corresponds to a basic block
- each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j

We assume that each CFG has a unique entry node n_0 and a unique exit node n_f.

Example Consider the following CFG for a `while` loop.

```
while (i < 100)
  begin
    stmt1
  end
stmt2
```



The edge from $stmt_1$ back to the loop header creates a **cycle**. The AST for this fragment would be acyclic.

Compilers typically use a CFG **in conjunction** with another IR
- graph represents the relationships among blocks
- operations inside a block are represented with another IR

Using **single-statement blocks**, *i.e.*, code blocks that corresponds to a single source-level statement, as nodes can simplify algorithms for analysis and optimization.

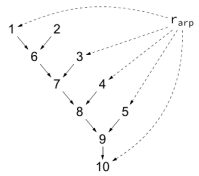
Many parts of the compiler rely on a CFG, either explicitly or implicitly
- control-flow analysis
- instruction scheduling
- global register allocation

2.1.5 Dependence Graph

- nodes represent operations including definitions, directed edges indicate definition and usages of values (from definition to use)
- represents constraints on the sequence and possibility
- used for e.g. instruction scheduling, parallelization, optimizing memory (e.g. array but also I/O) access patterns

Dependence Graph

```
1  loadAI   r_arp, @a ⇒ r_a
2  loadI    2         ⇒ r_2
3  loadAI   r_arp, @b ⇒ r_b
4  loadAI   r_arp, @c ⇒ r_c
5  loadAI   r_arp, @d ⇒ r_d
6  mult     r_a, r_2  ⇒ r_a
7  mult     r_a, r_b  ⇒ r_a
8  mult     r_a, r_c  ⇒ r_a
9  mult     r_a, r_d  ⇒ r_a
10 storeAI  r_a       ⇒ r_arp, @a
```



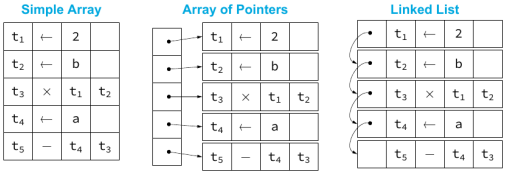
Note Uses of r_{arp} (starting address of the local data area) refer to its implicit definition at the start of the procedure and are shown with dashed lines.

2.1.6 Call Graph

- each node is a procedure
- each edge is a procedure call

2.2 Linear IRs

As compiler spends a lot of time calculating with IR, data structure storing linear IR shall be compact, efficient, e.g. array, array of pointers, linked list



2.2.1 Stack Machine Codes

Stack-machine code, a form of one-address code, assumes the presence of a stack of operands.

The stack creates an **implicit name space** and eliminates many names from the IR
- shrinks the size of a program in IR form
- all results and arguments are transitory, unless the code explicitly moves them to memory

Stack-machine code is simple to generate and to execute.

Stack-machine code for expression a − 2 × b

```
push 2
push b
multiply
push a
subtract
```

Bytecode

Bytecode is an IR designed specifically for its compact form. Typically it represents code for an abstract stack machine.

The name derives from its limited size: opcodes are limited to one byte or less.

→ compact form of the program for distribution
→ reasonably simple scheme for porting the language to a new target machine

2.2.2 Three-Address Code

In **three-address code**, most operations have the form `i ← j op k`
- one operator (`op`)
- two operands (`j` and `k`)
- one result (`i`)

Note Some operators, such as an immediate load and a jump, will need fewer arguments

Three-address code for expression $a - 2 \times b$

```
t1 ← 2
t2 ← b
t3 ← t1 × t2
t4 ← a
t5 ← t4 − t3
```

Advantages of three-address code
- reasonably compact: an operation (1 or 2 bytes) and three names (4 bytes)
- freedom to control the reuse of names and values
- many modern processors implement three-address operations

2.3 SSA

Static single-assignment form (SSA) is a naming scheme that many modern compilers use to encode information about both the control and data flow in the program.

A program is in SSA form when it meets two constraints
- each definition has a distinct name
- each use refers to a single definition

In SSA form, names correspond uniquely to specific **definition points** in the code. Each name is defined by one operation, hence the name static single assignment.

To reconcile this single-assignment naming scheme with the effects of **control flow**, a special operation, called ϕ-functions, is inserted at points where control-flow paths meet.

```
x ← ···
y ← ···
while (x < 100)
    x ← x + 1
    y ← y + x
```

```
      x0 ← ···
      y0 ← ···
      if (x0 ≥ 100) goto next
loop: x1 ← ϕ(x0, x2)
      y1 ← ϕ(y0, y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < 100) goto loop
next: x3 ← ϕ(x0, x2)
      y3 ← ϕ(y0, y2)
```

ϕ-function

A ϕ-function takes several names and merges them, defining a **new** name.

The ϕ-function's behavior **depends on context**: it defines the name with the value of its argument that corresponds to the edge along which control entered the block.

On entry to a basic block, all of its ϕ-functions execute **concurrently**, before any other statement
- first, they **all** read the values of the appropriate arguments
- then they **all** define their target SSA names

Note ϕ-functions do not comform to the three-address model!

2.4 Procedures

- callee is invoked procedure, caller is the procedure which invoked the called procedure
- each procedure introduces a new namespace
- a name declared as argument of a function is called formal parameter, a variable passed as argument to the function is called actual parameter
- The linkage convention is a standard used by compiler and OS that defines rules like register mapping for parameters, preserving the callers runtime and setting up the calles

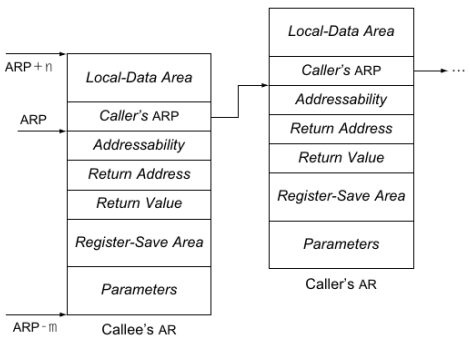
2.4.1 Calls

- A procedure call (**Activation**) transfers control from the call of the caller to the start of the callee
- on exit the callee returns to the instruction following the call instruction of the caller
- on call the caller pushes the return address on the stack, on return the callee pops this address

2.4.2 Name Spaces

A **Name Space** maps a set of names to a set of values and procedures, may inherit things from other name spaces and can create names that are inacessible for the outside name space. A **Scope** refers to a Name Space. **Lexical scopes** are nested name spaces that nested as they are encountered dureing program execution. In a given scope **each name refers to its lexically closest scope**. The outermost scope contains the **global variables**. The **static coordinate** $< l, o >$ is a pair with l lexical scope identifier, o offset from the beginning of l.

Stack Frame aka Activation Record



The compiler needs to **reserve space for local data** items in the AR
- assign each such item an appropriately sized area
- record its offset from the ARP in the symbol table

Together with its lexical level the item's offset becomes the its static coordinate.

If the compiler **cannot know the size** of a local variable at compile time
1. leave space for a pointer to the actual data or to a descriptor for an array
2. arranges to allocate the actual storage elsewhere at runtime
3. fill the reserved slot with the address of the dynamically allocated memory

If the source language allows the program to specify an initial value for a variable, the compiler must arrange for that **initialization** to occur.

Static and Global Variables
- lifetime independent of any procedure
- data can be inserted directly into the appropriate locations by the loader
- for example, use a **data section** (`.data`) in assembly code

Local Variables
- must be initialized at runtime as a procedure may be invoked multiple times
- generate instructions that store the initial values to the appropriate locations

When p calls q, one of them must **save the register values** that p needs after the call
- it may be necessary to save all the register values or a subset may suffice
- on return to p, these saved values must be restored
- saved registers are stored in the AR of either p or q, or both

If the **callee saves a register**, its value is stored in register save area of the callee's AR. Similarly, if the **caller saves a register**, its value is stored in the caller's register save area.

Calling Convention

Who saves what is defined by the **calling convention** used by the operating system. Solaris, Linux, FreeBSD, and macOS follow the System V AMD64 ABI calling convention. Windows follows the Microsoft x64 calling convention.

Allocating Activation Records

When p calls q at runtime, the code that implements the call must allocate an AR for q and initialize it with the appropriate values.

Fields of AR are stored in **memory**
- caller needs to have access to callee's AR to store parameters, return address, *etc.*
- this forces allocation of callee's AR into caller, who might not know the local data size

Values of AR passed by **registers**
- allocation of the ar can be performed in the callee, who knows the local data size
- callee may store into its ar some of the values passed in registers after allocation

The compiler can allocate ARs on the **stack** or on the **heap**. In certain cases, it can allocate a single **static** AR for a procedure or **coalesce** ARs of a set of procedures.

If procedure calls and returns are balanced, they follow a last-in, first-out (LIFO) discipline. Since the ARs also follow this LIFO ordering, they can be allocated on the **stack**.

Keeping activation records on the stack has several advantages
- inexpensive allocation and deallocation: simply move the top-of-stack pointer
- separation of concerns between caller and callee
  - caller allocates space for address parameters, return address, *etc.*
  - callee can extend AR to include local data area and variable-sized objects
- debugger can walk the stack to produce a snapshot of the currently active procedures

Example Pascal, C, and Java are typically implemented with stack-allocated ARs

Certain language features prevent ARs from being allocated on the stack
- if a procedure can outlive its caller
- if a procedure can return an object that includes references to its local variables

In these situations, ARs can be kept on the **heap**. With heap-allocated ARs, variable-size objects can be allocated as separate objects on the heap.
- explicit deallocation: procedure return code frees the AR and variable-size objects
- implicit deallocation: garbage collector frees them when they are no longer useful

Example Implementations of Scheme and ML typically use heap-allocated ARs

A **leaf procedure** is a procedure that calls no other procedures. The compiler can allocate ARs for leaf procedures **statically**. This eliminates the runtime costs of AR allocation.

Example If the calling convention requires the caller to save its own registers, then the AR of a leaf procedure needs no register save area.

Under certain circumstances the compiler can do even better! Assuming that callees cannot outlive callers, **only one** leaf procedure can be active at any point during execution.
- allocate a **single** static AR for use by **all** leaf procedures
- AR must be large enough to accommodate any of the program's leaf procedures
- static variables of all leaf procedures can be laid out together in that single AR

Using a single static AR for leaf procedures reduces the space overhead of separate static ARs for each leaf procedure.

2.4.3 Addressability

Each procedure encapsulates common operations relative to a small set of names, i.e. parameter values, return values, global variables and local variables, which must be bound by the compiler. As the procedure is called from many different contexts which might not be known when writing, binding is curcial.
Scalar values like addresses and numbers are stored in registers or the parameter section of the callees AR.
Large values like arrays, structs, objects, ... add significant overhead when copied and should be passed by reference (e.g. const enables automatic reformulation of function signature).

Call by Value (evaluates possibly an expression and) binds the concrete value to the formal parameter of the callee by copying it into a specific register or parameter slot in the callees AR. The value is only bound to the formal parameter name at procedure start, i.e. the value beeing bound only to the formal parameter is an initial condition. A change of the value is only visible inside the function.

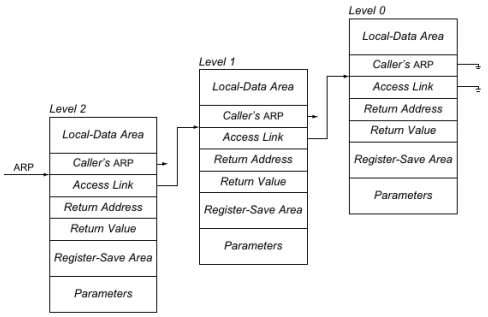
Call by Reference binds an address to the formal parameter of the callee. The content of the memory starting at this address have to be interpretable as the type specified by the formal parameters. A change to the contents of the memory starting at the passed address changes the state for all instances using the memory range. Thus Thus a level of indirection is introduced (**alias** for the instance of the referenced type).

Return value is stored outside the callees AR as it needs (eventually) to be used by the caller.
If the return value is small and has fixed size, it's stored in a designated register (rax in x64) or in the callers AR: Allocate space and write the address to the space into return slot. Let the callee access the pointer via the callers stack pointer.
If the return values size is unknown, the callee allocates space, stores the address into the callers AR return slot which transfers ownership to the caller.

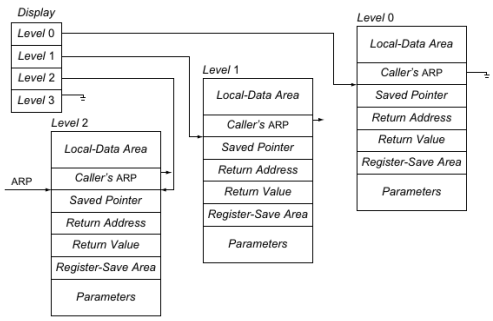
Addressing in a Stack Frame The memory region that holds the data for a scope is called **data area**. The address at the beginning of the data area is called **base address**. The base address can be static or dynamic (wrt. compile and run time).
For global and static data, the compiler arranges a data area with static base address and assigns it a label using name mangling (construction a unique string from a source-level name).
For procedures the compiler constructs an AR with a dynamic base address. Lexical scoping declares rules for which ARs shall be accessible from where.

Accessing Local variables by adding an offset to the base address (rbp) of the current AR or stored on the heap and accessed by indirection if their size is not known at compile time. Local variables of other procedures are accessed by a runtime data structure which the compiler must build. It shall map a static coordinate into a runtime address.

Access or Static Links extend each AR with a Pointer to the AR of its immediate lexical ancestor. Emit code that walks the linked list of ARs using the static coordinate.



Global Display instantiates a global array storing the rbps of the most recent procedure activations at each lexical level. Access local variables of other procedures by indirection.



Calling conventions aka ABI

System V AMD64 ABI

Mapping Actual to Formal Parameters
- integer arguments (including pointers) are placed in the registers `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9`, in that order
- floating point arguments are placed in the registers `%xmm0`-`%xmm7`, in that order.
- parameter values in excess of the available registers are pushed onto the stack
- if the function takes a variable number of arguments (like `printf`) then the `%eax` register must be set to the number of floating point arguments

Saving Registers
- the callee may use any registers, but it must restore the values of the registers `%rbx`, `%rbp`, `%rsp`, and `%r12`-`%r15`, if it changes them.

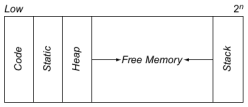
Returning Values
- the return value of a call is placed in `%rax`

2.5 Storage Locations

The compiler must assign storage locations to all variables. We can distinguish between **named** and **unnamed or anonymous** values. Named values follow the convention of the source language, while unnamed values must be handeled consistently with those, but effectively location and lifetime are managed by the compiler. Further the decision what values to keep in registers:

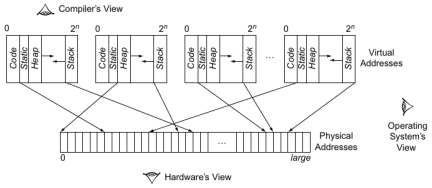
Memory-to-Memory Model
- compiler assumes that all values reside in memory
- values are loaded into registers as needed and stored to memory after each definition
- IR typically uses **physical register** names

Register-to-Register Model
- compiler assumes that it has enough registers to express the computation
- use a distinct **virtual register** for each value that can legally reside in a register
- store a virtual register's value to memory only when absolutely necessary



Typical layout for the address space used by a single compiled program
- **code** sits at the bottom of the address space
- **static area** holds both static and global data areas, along with any fixed-size data created by the compiler, *e.g.*, jump tables, debug information
- **heap** contains dynamically allocated data structures
- runtime **stack** is used to stack-allocate activation records, if possible

The operating system maps multiple **logical address spaces** into the single **physical address space** supported by the processor.



For convenience, the compiler groups together the storage for values with the **same lifetimes** and **visibility**. It creates distinct **data areas** for them.

Typical placement rules for Algol-like languages
- if x is declared locally in procedure p
  - if its value is not preserved across distinct invocations of p
    ⟶ assign it to procedure-local storage
  - if its value is preserved across invocations of p
    ⟶ assign it to procedure-local static storage
- if x is declared as globally visible
  ⟶ assign it to global storage
- if x is allocated under program control
  ⟶ assign it to the runtime heap

In local, static and global scopes, offsets have to be assigned to each name, conforming to the ISA and the calling conventions (ABI).

[1]The Intel386 ABI uses the term *halfword* for a 16-bit object, the term *word* for a 32-bit object, the term *doubleword* for a 64-bit object. But most IA-32 processor specific documentation define a *word* as a 16-bit object, a *doubleword* as a 32-bit object, a *quadword* as a 64-bit object and a *double quadword* as a 128-bit object.

Only unambigous value are allowed to be kept in registers across usages

Ambiguous and Unambiguous Values

Any value that can be accessed by multiple names is **ambiguous**. In contrast, a value that can be accessed with just one name is **unambiguous**.

Examples are

- class variables of objects
- array elements
- *dots*

2.6 Arithmetic Expressions

Lots of arithmetic operators, mappable by a tree walk for simple expressions:

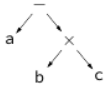
Arithmetic Operators

If the expression is represented in a tree-like IR, this process fits into a **postorder** tree walk.

`base`
- returns the name of a register holding the base address for an identifier
- if needed, it emits code to get that address into a register

`offset`
- returns the name of a register holding the identifier's offset
- offset is relative to the address returned by `base`

```
1  procedure expr(n)
2    if n ∈ {+, −, ×, ÷} then
3      t1 ← expr(n.left)
4      t2 ← expr(n.right)
5      r ← NextRegister()
6      emit(n, t1, t2, r)
7    else if n = ident then
8      t1 ← base(n)
9      t2 ← offset(n)
10     r ← NextRegister()
11     emit(loadAO, t1, t2, r)
12   else if n = num then
13     r ← NextRegister()
14     emit(loadI, n, _, r)
15   return r
```




```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
loadI   @c       ⇒ r5
loadAO  rarp, r5 ⇒ r6
mult    r4, r6   ⇒ r7
add     r2, r7   ⇒ r8
```

If parameters are operands:

1. Call-by-Value: if passed via stack, treat it as local variable (load), if passed in a register copy to appropriate operand register
2. Call-by-Reference: if passed via stack, load from the specified address.

If function is operand:

1. safe the registers to the stack
2. invoke the function call
3. registers
4. move the returned value into the operand register

For Mixed-Type expressions, the compiler must emit appropriate conversion code, applying the rules defined by the language specification and ABI for built-in types or the user provided conversions for programmer-defined types.

2.7 Boolean Expressions

2.7.1 Numerical Encoding

False $\equiv 0$, **True** $\equiv \neg$ **False or 1**. Comparison proceeds either by

- using a machine-provided operator that returns a boolean
- using condition codes and branch to write either 0 or 1

Condition Codes on Intel x86-64

Nearly all arithmetic instructions set condition codes based on their result.

- ZF result was **z**ero
- CF result caused **c**arry out of most significant bit
- SF result was negative (**s**ign bit was set)
- OF result caused (signed) **o**verflow

Based on these condition codes, standard condition suffixes *cc* are defined (*cf.* Slide 349) that modify the behaviur of three different kinds of instructions.

- `j`*cc* jumps to the specified label if *cc* holds
- `set`*cc* sets the given (single-byte) register to `1` or `0` depending on whether *cc* holds or not
- `cmov`*cc* performs the specified move only if *cc* holds

cc	Condition Tested	Meaning
e	ZF	equal to zero
ne	$\neg$ZF	not equal to zero
s	SF	negative
ns	$\neg$SF	not negative
g	$\neg$(SF $\oplus$ OF) $\wedge \neg$ZF	greater (signed $>$)
ge	$\neg$(SF $\oplus$ OF)	greater or equal (signed $\geq$)
l	SF $\oplus$ OF	less (signed $<$)
le	(SF $\oplus$ OF) $\vee$ ZF	less or equal (signed $\leq$)
a	$\neg$CF $\wedge \neg$ZF	above (unsigned $>$)
ae	$\neg$CF	above or equal (unsigned $\geq$)
b	CF	below (unsigned $<$)
be	CF $\vee$ ZF	below or equal (unsigned $\leq$)

2.7.2 Positional Encoding

In positional encoding the result is not stored but rather define the control flow. This and short-circuiting allow to generate much more efficient code

Short-Circuit Evaluation

In **short-circuit evaluation**, expressions are only evaluated until their final value is determined. Short-circuit evaluation relies on two Boolean identities.

$$\forall x \ \text{false} \wedge x = \text{false}$$
$$\forall x \ \ \text{true} \vee x = \text{true}$$

Some programming languages, *e.g.*, C and Java, **require** the compiler to use short-circuit evaluation.

```
1 if (a < b)
2 {
3     statement1;
4 }
5 else
6 {
7     statement2;
8 }
```

```
      comp   ra, rb ⇒ cc1
      cbr_LT cc1    → L1, L2
L1:   code for statement1
      jumpI         → L3
L2:   code for statement2
      jumpI         → L3
L3:   nop
```

Hardware Support for RelOps

- straight condition codes
- condition codes with conditional move
- boolean compare
- predicated operations

Source Code

```
if (x < y)
   then a ← c + d
   else a ← e + f
```

ILOC Code

Straight Condition Codes

```
      comp   rx, ry ⇒ cc1
      cbr_LT cc1    → L1, L2
L1:   add    rc, rd ⇒ ra
      jumpI         → Lout
L2:   add    re, rf ⇒ ra
      jumpI         → Lout
Lout: nop
```

Boolean Compare

```
      cmp_LT rx, ry ⇒ r1
      cbr    r1     → L1, L2
L1:   add    rc, rd ⇒ ra
      jumpI         → Lout
L2:   add    re, rf ⇒ ra
      jumpI         → Lout
Lout: nop
```

Conditional Move

```
comp   rx, ry ⇒ cc1
add    rc, rd ⇒ r1
add    re, rf ⇒ r2
i2i_LT cc1, r1, r2 ⇒ ra
```

Predicated Execution

```
      cmp_LT rx, ry ⇒ r1
      not    r1     ⇒ r2
(r1)? add    rc, rd ⇒ ra
(r2)? add    re, rf ⇒ ra
```

Source Code

```
x ← a < b ∧ c < d
```

ILOC Code

Straight Condition Codes

```
      comp   ra, rb ⇒ cc1
      cbr_LT cc1    → L1, L2
L1:   comp   rc, rd ⇒ cc2
      cbr_LT cc2    → L3, L2
L2:   loadI  false  ⇒ rx
      jumpI         → Lout
L3:   loadI  true   ⇒ rx
      jumpI         → Lout
Lout: nop
```

Conditional Move

```
comp   ra, rb ⇒ cc1
i2i_LT cc1, rT, rF ⇒ r1
comp   rc, rd ⇒ cc2
i2i_LT cc2, rT, rF ⇒ r2
and    r1, r2 ⇒ rx
```

Boolean Compare

```
cmp_LT ra, rb ⇒ r1
cmp_LT rc, rd ⇒ r2
and    r1, r2 ⇒ rx
```

Predicated Execution

```
cmp_LT ra, rb ⇒ r1
cmp_LT rc, rd ⇒ r2
and    r1, r2 ⇒ rx
```

Condition Codes
- code has at least one conditional branch per relational operator
- comparison operation may be omitted if condition codes are set by default

Conditional Move
- leads to faster code by avoiding branches
- safe as long as neither operation can raise an exception

Boolean Compare
- works without a branch and without converting comparison results to Boolean values
- a weakness of this model is that it requires explicit comparisons

Predicated Execution
- code is simple and concise
- predication can lead to the same code as the boolean-comparison scheme

2.8 Arrays

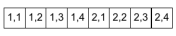
2.8.1 Vectors aka One-dimensional arrays

Vectors are stored in contiguous memory slots, so that the index together with the type information can be used to generate an offset which is added to the base address of the vector to obtain the elements address. If the Vector is specified over addresses or elements from $[\text{lower}, \text{upper}]$, the offset is calculated by $(i - \text{low}) \cdot \text{size\_t}$

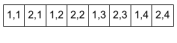
```
loadI  @V      ⇒ r@V   ...get V's address
subI   ri, 3   ⇒ r1    ...offset − lower bound
multI  r1, 4   ⇒ r2    ...× element length (4)
add    r@V, r2 ⇒ r3    ...address of V[i]
load   r3      ⇒ rV    ...value of V[i]
```

2.8.2 Storage Layout

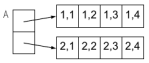
In **row-major order**, the elements of a are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations.



The obvious alternative to row-major order is **column-major order**. It keeps the columns of a in contiguous locations, producing the following layout.



A third alternative are **indirection vectors**. This scheme reduces all multi-dimensional arrays to a set of (one-dimensional) vectors.



Indirection vectors appear simple, but they introduce their own complexity
- need more storage than either of the contiguous storage schemes
- require that application initializes all of the indirection pointers at runtime

An advantage of indirection vectors is that they can easily represent **ragged arrays**.

2.8.3 Referencing

False Zero

The **false zero** of a vector `V` is the address where `V[0]` would be. In multiple dimensions, it is the location of a zero in each dimension.

Before deriving the corresponding formula, we introduce the following notation
- low_i and $high_i$ denote *low* and *high* of the i-th dimension, respectively
- $len_i = high_i - low_i + 1$ denotes the length of the i-th dimension

In row-major order, the address calculation must find the start of the row and then generate an offset within the row as if it were a vector.

To access element `A[i,j]` of a two-dimensional array the compiler therefore must emit code that computes the…
- address of row `i`: $(i - low_1) \times len_2 \times w$
- offset of element `j`: $(j - low_2) \times w$

Putting all parts together, the resulting address computation is as follows.

$$@A + (i - low_1) \times len_2 \times w + (j - low_2) \times w$$

With w sizeof array element, can be simplyfied to

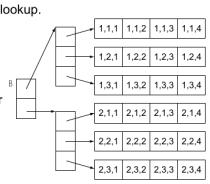
$$@\text{Base} + (i \cdot \text{len}_2 + j) \cdot w$$

```
loadI   @A0          ⇒ r@A0     …adjusted base address of A
multI   ri, len2     ⇒ r1       …i ×len2
add     r1, rj       ⇒ r2       …+ j
multI   r2, 4        ⇒ r3       …× element length (4)
loadAO  r@A0, r3     ⇒ ra       …value of A[i,j]
```

Indirection Vectors

Using indirection vectors simplifies the code generated to access an individual element. For an n-dimensional array, the first $n-1$ steps navigate the indirection vectors, while the last step performs a regular (one-dimensional) vector lookup.

Example Accessing element `B[i,j,k]` in the array `B[1...2,1...3,1...4]`
1. use `@B0`, `i`, and the length of a pointer, to find the vector for the subarray `B[i,*,*]`
2. use result, along with `j` and the length of a pointer to find the vector for the subarray `B[i,j,*]`
3. use base address in the vector-address computation with k and element length *w* to find the address of `B[i,j,k]`




```
loadI   @B0          ⇒ r@B0     …adjusted base address of B
multI   ri, 4        ⇒ r1       …pointer size (4)
loadAO  r@B0, r1     ⇒ r2       …load @B[i,*,*]
multI   rj, 4        ⇒ r3       …pointer size (4)
loadAO  r2, r3       ⇒ r4       …load @B[i,j,*]
multI   rk, 4        ⇒ r5       …element size (4)
loadAO  r4, r5       ⇒ rb       …value of B[i,j,k]
```

Dope Vector

A descriptor for an actual parameter array that contains both a pointer to the **start of the array** and the **necessary information for each dimension**. Dope vectors may also be used for arrays whose bounds are determined at runtime.

Caller
- builds the dope vector and stores it in callee's AR
- value passed in the array's parameter slot is a pointer to the dope vector

Callee
- loads values out of the dope vector as needed
- generates array reference using the same polynomial as for a local array

Dope vector and extended versions of it are used for boundary checking

2.9 Strings

- CISC machines support string operations, RISC machines rely on compiler to emit instructions for that
- C uses null termination whereas other langs use an explicit length field

The code is similar for longer strings. The following code for `a = b;` assumes explicit length representation and hardware support for character-sized memory operations.

```
        loadI    @b       ⇒ r@b      …load address of b
        loadAI   r@b, -4  ⇒ r1       …load b's length
        loadI    @a       ⇒ r@a      …load address of a
        loadAI   r@a, -4  ⇒ r2       …load a's length
        cmp_LT   r2, r1   ⇒ r3       …does b fit into a?
        cbr      r3       → Lsov, L1 …raise string overflow
L1:     loadI    0        ⇒ r4       …initialize counter
        cmp_LT   r4, r1   ⇒ r5       …more to copy?
        cbr      r5       → L2, L3   …jump
L2:     cloadAO  r@b, r4  ⇒ r6       …load character from b
        cstoreAO r6       ⇒ r@a, r4  …store character in a
        addI     r4, 1    ⇒ r4       …increment counter
        cmp_LT   r4, r1   ⇒ r7       …more to copy?
        cbr      r7       → L2, L3   …jump
L3:     storeAI  r1       ⇒ r@a, -4  …set length of a
```

In C, which uses null termination for strings, the same assignment would be written as a character-copying loop.

```
t1 = a;
t2 = b;
do {
   *t1++ = *t2++;
} while (*t2 != '\0')
```

```
        loadI   @b       ⇒ r@b      …load address of b
        loadI   @a       ⇒ r@a      …load address of a
        loadI   NULL     ⇒ r1       …load terminator
L1:     cload   r@b      ⇒ r2       …load character
        cstore  r2       ⇒ r@a      …store character
        addI    r@b, 1   ⇒ r@b      …move pointer
        addI    r@a, 1   ⇒ r@a      …move pointer
        cmp_NE  r1, r2   ⇒ r3       …more to copy?
        cbr     r3       → L1, L2   …jump
L2:     nop                         …next statement
```

Library routines are optimized to e.g. copy blocks, . . .

String Concatenation

Concatenation is simply a shorthand for a sequence of one or more assignments. It comes in **two basic forms**.

1. Appending string b to string a
   - compiler emits code to determine the length of a
   - space permitting, it assigns b to the space that immediately follows a
2. Creating a new string that contains a followed immediately by b
   - requires copying each character in a and each character in b
   - can be treated as a pair of assignments in code generation

The compiler should ensure that **enough space** is allocated to hold the result. If the lengths of a and b are unknown at compile time, runtime code needs to be generated
- to compute the lengths of the strings
- to perform corresponding test and branch

String Length

Programs that manipulate strings often need to compute a character string's length.

Different string representations lead to different costs for the length computation
- a **null-terminated string** requires time proportional to the length of the string
- with an **explicit length field** the cost is constant and small

Tradeoff Null termination saves a small amount of space, but requires more code and more time for the length computation. An explicit length field costs one more word per string, but makes the length computation take constant time.

Example With explicit length fields a statement `length(a + b)` can be optimized to two `loads` and one `add`.

String Length on Intel x86-64

The Intel x86-64 ISA provides several **dedicated operations** for string manipulation.
- `movs`, `lods`, `stos`: move, load, and store strings
- `cmps`: compare strings
- `scas`: scan strings

These instructions are combined with **repetition prefixes**: `rep`, `repe`, `repne`, `repz`, and `repnz`).

```
movq  -1, %rcx      # "clear" counting register
movq  %rdi, %rsi    # backup %rdi
mov   0, %al        # look for \0
repne scasb         # actually do the search
subq  %rsi, %rdi    # save the string length
decq  %rdi          # do not count the \0 in the string length
movq  %rdi, %rax    # save the return value
```

2.10 Structure References

2.10.1 Structure Layout

Compiler needs to emit code that safes offset and length/type of each structure member. Further offsets need to comply to the alignment rules of the target architecture.

Structure Layout Table

Name	Length	1st Element
node	8	•
...	...	•

Structure Element Table

Name	Length	Offset	Type	Next
node.value	4	0	int	•
node.next	4	4	struct node *	•
...	...	...	...	...

2.10.2 Runtime Dependent Types

Many languages support structures with **multiple**, **data-dependent** interpretations.

Unions and variants present one additional complication
- the possibility exists that element names are **not unique**
- compiler must resolve each reference to a unique offset and type in the runtime object

```
1 struct n1 {
2    int kind;
3    int value;
4 };
```

```
1 struct n2 {
2    int kind;
3    float value;
4 };
```

```
1 union one {
2    struct n1 inode;
3    struct n2 fnode;
4 } u1;
```

This problem has a **linguistic solution**: the programming language can force the programmer to make the reference unambiguous.

one.inode or one.fnode is a fully qualified name that resolves any ambiguity

Anonymous Values are only accessible through a reference, thus have no permanent name. An open problem is to keep track of all memory objects that are referencable as they may form an unbound set.

2.11 Control Flow

2.11.1 Conditional Execution

Conditional Execution

Most programming languages provide some version of an `if-then-else` construct.

```
if expr
   then statement1
   else statement2
statement3
```

The compiler needs to generate code that
- evaluates *expr* and branches to $statement_1$ or $statement_2$, accordingly
- implements the two statements and at the end jumps to $statement_3$

As we already saw on Slide 355, the compiler has many options for implementing `if-then-else` constructs.

```
if (a < b)
{
    statement1;
}
else
{
    statement2;
}
```

```
    comp    ra, rb ⇒ cc1
    cbr_LT  cc1    → L1, L2
L1: code for statement1
    jumpI          → L3
L2: code for statement2
    jumpI          → L3
L3: nop
```

Choosing between branching and predication to implement an `if-then-else` requires some care. Several issues need to be considered.

1. **Expected frequency of execution**: if one path executes significantly more often, techniques that speed up its execution (*e.g.*, branch prediction, speculative execution, and instruction reordering) may produce faster code
2. **Uneven amounts of code**: if one path contains many more instructions, this may weigh against predication or for a combination of predication and branching
3. **Control flow inside the construct**: if either path contains nontrivial control flow, in particular nested `if`-statements, then predication may be a poor choice

Predication in the Real World

Nvidia's CUDA parallel computing platform 32 threads are executed together as a **warp**. All threads in a warp execute the **same** instruction at the **same** time.

In order to support **different** threads doing **different** things, CUDA has predicated instructions that are executed only if a logical flag is `true`.

```
if (x < 0.0)
   r = -1.0;
else
   r = sqrt(x);
```

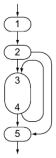
```
    p = (x < 0.0);
 p: r = -1.0;
!p: r = sqrt(x);
```

This is called **warp divergence**. The performance of the code generated by the `nvcc` compiler is impacted by the same issues as outlined on the previous slide.

2.11.2 Loops

The following schema shows how the compiler might lay out the code.

```
for (e1; e2; e3) {
    loop body
}
```



Step	Action
1	evaluate e_1
2	if ($\neg e_2$) then goto 5
3	*loop body*
4	evaluate e_3 if (e_2) then goto 5
5	*code following loop*

We will use this basic schema to explain the implementation of several kinds of loops.

The compiler can hide the latency of this branch in several ways

- **branch hint prefix**: mark the branch in Step 4 as likely to be taken
- **delay slot**: move instructions from loop body into the branch delay slot(s)
- **loop unrolling**: if number of iterations is predictable, combine multiple iterations

Delay Slot

Some computer architectures feature **delay slots** to mask the latency of time-consuming instructions, such as loads and branches. A delay slot is an instruction that gets executed without the effects of a preceding instruction.

Branch Prediction in Intel x86-64

In the past, Intel's ISA supported static branch prediction prefixes (`2EH` and `3EH`), a feature that has been abandoned since.

Modern Intel CPUs use a Branch Target Buffer (BTB) to manage branching history dynamically. If this information is not available, the CPU uses a simple heuristic:

- conditional forward branches are predicted as **not being taken**
- conditional backward branches are predicted as **being taken**

This heuristic fits well with the basic schema that we will use to implement loops.

For Loops

To map a `for` loop into code, the compiler follows the general schema from before.

```
for (i = 1; i <= 100; i++) {
    loop body
}
next statement
```

```
    loadI   1      ⇒ ri      …Step 1
    loadI   100    ⇒ r1      …Step 2
    cmp_GT  ri, r1 ⇒ r2
    cbr     r2     → L1, L2
L1: loop body                …Step 3
    addI    ri, 1  ⇒ ri      …Step 4
    cmp_LE  ri, r1 ⇒ r3
    cbr     r3     → L1, L2
L2: next statement           …Step 5
```

The compiler can also shape the loop to have only **one copy** of the test. In this form, Step 4 evaluates e_3 and then jumps to Step 2, *i.e.*, replace `cmp_LE` and `cbr` with `jumpI`.

While Loops

A `while` loop can also be implemented with the loop schema. Since it has no initialization, the code is even more compact.

```
while (x < y) {
    loop body
}
next statement
```

```
    cmp_LT  rx, ry ⇒ r1      …Step 2
    cbr     r1     → L1, L2
L1: loop body                …Step 3
    cmp_LT  rx, ry ⇒ r2      …Step 4
    cbr     r2     → L1, L2
L2: next statement           …Step 5
```

Replicating the test in Step 4 creates the possibility of a loop with a **single basic block**.

Until Loops

An `until` loop iterates as long as the controlling expression is `false`.

It checks the controlling expression **after** each iteration. Thus, it always enters the loop and performs at least one iteration and produces a particularly simple loop structure.

```
{
    loop body
} until (x < y)
next statement
```

```
L1: loop body                …Step 3
    cmp_LT  rx, ry ⇒ r1      …Step 4
    cbr     r1     → L1, L2
L2: next statement           …Step 5
```

Note The `do` loop known from C, C++, and Java is similar to the `until` loop with the difference that it iterates as long as the controlling expression is `true`.

2.11.3 Case Statements

Many programming languages include some variant of a case statement.

The basic strategy is straightforward

1. evaluate the controlling expression
2. branch to the selected case
3. execute the code for that case

Steps 1 and 3 are well understood. The complex part of case-statement implementation lies in choosing an efficient method to locate the designated case.

No single method works well for all case statements. We examine three strategies: a **linear search**, a **binary search**, and a **computed address**.

Linear Search

The simplest way to locate the appropriate case is to treat the case statement as the specification for a nested set of `if-then-else` statements.

```
switch(e) {
   case 0:   block0;
             break;
   case 1:   block1;
             break;
   case 3:   block3;
             break;
   default:  blockd;
             break;
}
```

```
t1 ← e
if (t1 = 0)
    then block0
    else if (t1 = 1)
        then block1
        else if (t1 = 3)
            then block3
            else blockd
```

The compiler should order cases according to estimated execution frequency.

Binary Search

```
switch(e) {
   case 0:   block0;
             break;
   case 15:  block15;
             break;
   case 23:  block23;
             break;
   …
   case 99:  block99;
             break;
   default:  blockd;
             break;
}
```

```
t1 ← e
down ← 0        …lower bound
up ← 10         …upper bound + 1
while (down + 1 < up) {
    middle ← (up + down) ÷ 2
    if (Value[middle] ≤ t1)
        then down ← middle
        else up ← middle
}
if (Value[down] = t1)
    then jump to Label[down]
    else jump to LBd
```

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

Computing the Address Directly

```
switch(e) {
   case 0:   block0;
             break;
   case 1:   block1;
             break;
   case 2:   block2;
             break;
   …
   case 9:   block9;
             break;
   default:  blockd;
             break;
}
```

```
t1 ← e
if (t1 < 0 or t1 > 9)
    then jump to LBd
    else
        t2 ← @Table + t1 × 4
        t3 ← memory(t2)
        jump to t3
```

Label
LB_0
LB_1
LB_2
LB_3
LB_4
LB_5
LB_6
LB_7
LB_8
LB_9

2.11.4 Procedure Calls

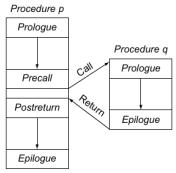
Procedure Calls

The implementation of procedure calls is, for the most part, straightforward.

Recall: a procedure call consists of

- a **precall** and a **postreturn** sequence in the caller
- a **prologue** and an **epilogue** sequence in the callee

In the following, we focus on issues that affect the compiler's ability to generate **efficient**, **compact**, and **consistent** code for procedure calls.



As a general rule, moving operations from the precall and postreturn sequences into the prologue and epilogue sequences should **reduce the overall size** of the final code.

Evaluating Actual Parameters

When it builds the precall sequence, the compiler must emit code to evaluate the actual parameters to the call. The compiler treats each actual parameter as an expression.

- **call-by-value parameters**: evaluate the expression and store its value in location designated for that parameter, *i.e.*, a register or callee's AR
- **call-by-reference parameter**: evaluate the parameter to an address and store it in the location designated for that parameter

Note If a call-by-reference parameter has no storage location, then the compiler may need to allocate space for that value so that it has an address to pass to the callee.

The compiler should use a consistent **evaluation order** for parameters, *i.e.*, either left to right or right to left, as evaluating parameters might have side effects.

As both the cost of memory operations and the number of registers have risen, the cost of **saving and restoring registers** has increased to the point that it needs careful attention.

1. **Using multi-register memory operations**: many ISAs support doubleword and quadword load and store operations for adjacent registers.
2. **Using a library routine**: replace the sequence of individual memory operations with a call to a compiler-supplied save or restore routine
3. **Combining responsibilities**: caller and callee pass a value back and forth that specifies which registers each must save

Chapter 3

Back End

Code Generation consists of Instruction Selection, Instruction scheduling and Register allocation. All of them are NP-complete for exact graph-based versions.

3.1 Instruction Selection

- The process of mapping IR code to target machine instructions
- pattern matching problem: Tree-pattern matching or Peephole optimization
- costs depend highly on ISA provided instructions (RISC, CISC, Stack machines)
- compiler writer should associate costs to instruction and built up a truly reflecting cost model for efficient code

3.1.1 Tree Pattern-Matching

1. Express target instruction set as a set of trees
2. Build AST from source
3. Find the minimal cost tiling, that implements every operation and all tiles connect to their neighbors (i.e. the AST node is covered by the current and a neighbour tile)

Tiling

A **tiling** is a collection of ⟨*ast-node*, *op-tree*⟩ pairs, where *ast-node* is a node in the AST and *op-tree* is an operation tree.

The presence of an ⟨*ast-node*, *op-tree*⟩ pair in the tiling means that the target-machine operation represented by *op-tree* **could** implement *ast-node*.

A tiling **implements** the AST if it implements every operation and each tile connects with its neighbors. A tile, ⟨*ast-node*, *op-tree*⟩, **connects** with its neighbors if *ast-node* is covered by a leaf in another *op-tree* in the tiling, unless *ast-node* is the root of the AST.

The relationships between operation trees and subtrees in the AST is encoded as a set of **rewrite rules**. The rule set includes **one or more** rules for every kind of node in the AST.

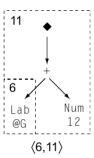
Rewrite Rules

A rewrite rule consists of a production in a tree grammar, a code template, and an associated cost.

	Production	Cost	Code Template
1	Goal → Assign	0	–
2	Assign → ←(Reg$_1$, Reg$_2$)	1	store r$_2$ ⇒ r$_1$
3	Assign → ←(+(Reg$_1$, Reg$_2$), Reg$_3$)	1	storeAO r$_3$ ⇒ r$_1$, r$_2$
4	Assign → ←(+(Reg$_1$, Num$_2$), Reg$_3$)	1	storeAI r$_3$ ⇒ r$_1$, n$_2$
5	Assign → ←(+(Num$_1$, Reg$_2$), Reg$_3$)	1	storeAI r$_3$ ⇒ r$_2$, n$_1$
6	Reg → Lab$_1$	1	load l$_1$ ⇒ r$_{new}$
7	Reg → Val$_1$	0	–
8	Reg → Num$_1$	1	load n$_1$ ⇒ r$_{new}$
9	Reg → ◆(Reg$_1$)	1	load r$_1$ ⇒ r$_{new}$
10	Reg → ◆(+(Reg$_1$, Reg$_2$))	1	loadAO r$_1$, r$_2$ ⇒ r$_{new}$
11	Reg → ◆(+(Reg$_1$, Num$_2$))	1	loadAI r$_1$, n$_2$ ⇒ r$_{new}$
12	Reg → ◆(+(Num$_1$, Reg$_2$))	1	loadAI r$_2$, n$_1$ ⇒ r$_{new}$
13	Reg → ◆(+(Reg$_1$, Lab$_2$))	1	loadAI r$_1$, l$_2$ ⇒ r$_{new}$
14	Reg → ◆(+(Lab$_1$, Reg$_2$))	1	loadAI r$_2$, l$_1$ ⇒ r$_{new}$
15	Reg → +(Reg$_1$, Reg$_2$)	1	add r$_1$, r$_2$ ⇒ r$_{new}$
16	Reg → +(Reg$_1$, Num$_2$)	1	addI r$_1$, n$_2$ ⇒ r$_{new}$
17	Reg → +(Num$_1$, Reg$_2$)	1	addI r$_2$, n$_1$ ⇒ r$_{new}$
18	Reg → +(Reg$_1$, Lab$_2$)	1	addI r$_1$, l$_2$ ⇒ r$_{new}$
19	Reg → +(Lab$_1$, Reg$_2$)	1	addI r$_2$, l$_1$ ⇒ r$_{new}$
20	Reg → −(Reg$_1$, Reg$_2$)	1	sub r$_1$, r$_2$ ⇒ r$_{new}$
21	Reg → −(Reg$_1$, Num$_2$)	1	subI r$_1$, n$_2$ ⇒ r$_{new}$
22	Reg → −(Num$_1$, Reg$_2$)	1	rsubI r$_2$, n$_1$ ⇒ r$_{new}$
23	Reg → ×(Reg$_1$, Reg$_2$)	1	mult r$_1$, r$_2$ ⇒ r$_{new}$
24	Reg → ×(Reg$_1$, Num$_2$)	1	multI r$_1$, n$_2$ ⇒ r$_{new}$
25	Reg → ×(Num$_1$, Reg$_2$)	1	multI r$_2$, n$_1$ ⇒ r$_{new}$

The diagram on the right summarizes this sequence

- dashed boxes show the specific right-hand sides that matched
- the rule number is recorded in the upper left corner of each box
- the list of rule numbers at the bottom indicates the sequence in which the rules were applied
- the rewrite sequence replaces the boxed subtree with the final rule's left-hand side.



⟨6,11⟩

Note Nonterminals ensure that the operation trees connect appropriately at the points where they overlap.

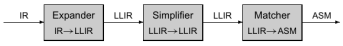
- Rule 6 rewrites a Lab as a Reg
- the left leaf in Rule 11 is a Reg

```
1  procedure Tile(n)
2  |  LABEL(n) ← ∅
3  |  if n is a binary node then
4  |  |  Tile(left(n))
5  |  |  Tile(right(n))
6  |  |  foreach rule r that matches n's operation do
7  |  |  |  if left(r) ∈ LABEL(left(n)) ∧ right(r) ∈ LABEL(right(n)) then
8  |  |  |  |  LABEL(n) ← LABEL(n) ∪ {r}
9  |  else if n is a unary node then
10 |  |  Tile(left(n))
11 |  |  foreach rule r that matches n's operation do
12 |  |  |  if left(r) ∈ LABEL(left(n)) then
13 |  |  |  |  LABEL(n) ← LABEL(n) ∪ {r}
14 |  else
15 |  |  LABEL(n) ← {all rules that match operation in n}
```

3.1.2 Peephole Optimization

Early peephole optimizers used a limited set of **hand-coded patterns**.

A modern peephole optimizer breaks the process into three distinct tasks: **expansion**, **simplification**, and **matching**.



Structurally, this looks like a compiler

- if the input and output languages are the same, this system is a peephole optimizer
- with different input and output languages, the same algorithms can perform instruction selection

The **expander** rewrites the IR into a sequence of lower-level IR (LLIR) operations that represents all the direct effects of an operation.

The **simplifier** makes a pass over the LLIR, examining the operations in a small window on the LLIR and systematically trying to improve them. The basic mechanisms are

- forward substitution,
- algebraic simplification,
- evaluating constant-valued expressions, and
- liminating useless effects, *e.g.*, creation of unused condition codes.

The **matcher** compares the simplified LLIR against the pattern library, looking for the pattern that best captures all the effects in the LLIR.

Example The next slides show these three steps for the low-level AST from Slide 440.

```
                                                    r10          ← 2
                                                    r11          ← @G
                                                    r12          ← 12
                                                    r13          ← r11 + r12
                                                    r14          ← M(r13)
                                                    r15          ← r10 × r14
expand                                              r16          ← -16          simplify
⟶                                                   r17          ← rarp + r16   ⟶
                                                    r18          ← M(r17)
                                                    r19          ← M(r18)
                                                    r20          ← r19 − r15
                                                    r21          ← 4
                                                    r22          ← rarp + r21
                                                    M(r22)       ← r20

r10          ← 2                                                r10          ← 2
r11          ← @G                                               r11          ← @G
r14          ← M(r11 + 12)                                      r14          ← M(r11 + 12)
r15          ← r10 × r14                                        r15          ← r10 × r14
r18          ← M(rarp − 16)                                     r18          ← M(rarp − 16)
r19          ← M(r18)                                           r19          ← M(r18)
r20          ← r19 − r15                                        r20          ← r19 − r15
M(rarp + 4)  ← r20                                              M(rarp + 4)  ← r20

r10          ← 2                                                loadI   2          ⇒ r10
r11          ← @G                                               loadI   @G         ⇒ r11
r14          ← M(r11 + 12)                                      loadAI  r11, r12   ⇒ r14
r15          ← r10 × r14                                match   mult    r10, r14   ⇒ r15
r18          ← M(rarp − 16)                             ⟶      loadAI  rarp, -16  ⇒ r18
r19          ← M(r18)                                           load    r18        ⇒ r19
r20          ← r19 − r15                                        sub     r19, r15   ⇒ r20
M(rarp + 4)  ← r20                                              storeAI r20        ⇒ rarp, 4
```

The next slides show the successive sequences that the peephole optimizer has in its window as it processes the low-level IR. Assume that it has a **three-operation window**.

```
r10 ← 2
r11 ← @G
r12 ← 12
```

Sequence 1

- no simplification is possible
- move window forward

```
r11 ← @G
r12 ← 12
r13 ← r11 + r12
```

Sequence 2

- forward substitute 12 into definition of r$_{13}$
- discard definition of dead register r$_{12}$

```
r11 ← @G
r13 ← r11 + 12
r14 ← M(r13)
```

Sequence 3

- forward substitute r$_{11}$ + 12 into definition of r$_{14}$
- discard definition of dead register r$_{13}$

```
r11 ← @G
r14 ← M(r11 + 12)
r15 ← r10 × r14
```

Sequence 4

- no simplification is possible
- move window forward

```
r14 ← M(r11 + 12)
r15 ← r10 × r14
r16 ← -16
```

Sequence 5

- no simplification is possible
- move window forward

```
r15 ← r10 × r14
r16 ← -16
r17 ← rarp + r16
```

Sequence 6

- forward substitute −16 into definition of r$_{17}$
- discard definition of dead register r$_{16}$

3.2 Instruction Scheduling

Terminology

A **stall** is the delay caused by a hardware interlock that prevents a value from being read until its defining operation completes.

An **interlock** is the mechanism that detects the premature issue and creates the actual delay.

A processor that relies on compiler insertion of `nops` for correctness is a **statically scheduled** processor.

A processor that provides interlocks to ensure correctness is a **dynamically scheduled** processor.

Commodity microprocessors often have operations that have **different latencies**. The following table summarizes typical values.

Cycles	Operation
1	integer add or subtract
3	integer multiply or a floating-point add or subtract
5	floating-point multiply
12–18	floating-point divide
20–40	integer divide

The latency of a load operation can range from a few cycles, *e.g.*, 1–5 cycles for the nearest cache, to tens or hundreds of cycles for values in main memory.

An Operation is a single opcode with operands. **An Instruction** is an aggregation of one or more operations that all issue in the same cycle.
Instruction scheduling is the process of reordering the code to decrease the running time. It must preserve the meaning of the code , it shall minimize the execution time by avoiding stalls or nops and it should avoid increasing value lifetimes past the point where additional register spills are necessary.

Given a dependence graph $\mathcal{D}$ for a code fragment, a schedule S maps each node $n \in N$ to a non-negative integer that denotes the cycle in which it should be issued, assuming that the first operation issues in cycle 1.

This provides a clear and concise definition of an instruction, namely, the i^{th} instruction is the set of operations $\{ n \mid S(n) = i \}$.

A schedule must meet **three** constraints
- $\forall n \in N : S(n) \geq 1$: operations that issue before execution starts are forbidden
- $\forall (n_1, n_2) \in E : S(n_1) + delay(n_1) \leq S(n_2)$: an operation cannot issue until its operands have been defined
- each instruction contains no more operations of each type t than the target machine can issue in a cycle

Given a well-formed schedule that is both correct and feasible, the **length** of the schedule is the cycle number in which the last operation completes, assuming the first instruction issues in cycle 1. Schedule length can be computed as follows.

$$L(S) = \max_{n \in N}(S(n) + delay(n))$$

With a notion of schedule length comes the notion of a **time-optimal schedule**.

Time-optimal Schedule

A schedule S_i is time optimal if $L(S_i) \leq L(S_j)$ for all other schedules S_j that contain the same set of operations.

Antidependence

Operation x is antidependent on operation y, denoted as $y \rightarrow x$, if x precedes y and y defines a value used in x.

Reversing their order of execution could cause x to compute a different value.

The scheduler can produce correct code in at least two different ways
- it can discover the antidependences that are present in the input code and **respect** them in the final schedule
- it can systematically **rename** the values in the block to eliminate antidependences before it schedules the code

3.2.1 Local List Scheduling

List scheduling is a **greedy, heuristic approach** to scheduling the operations in a basic block. It has been the dominant paradigm for instruction scheduling since the late 1970s.
- discovers reasonable schedules
- adapts easily to changes in computer architectures

To apply list scheduling to a block, the scheduler follows a **four-step plan**
1. rename to avoid antidependences
2. build a dependence graph $\mathcal{D}$
3. assign priorities to each operation
4. iteratively select an operation and schedule it

The heart of the algorithm, and the key to understanding it, lies in the final step, *i.e.*, the scheduling algorithm.

```
1 Cycle ← 1
2 Ready ← leaves of D
3 Active ← ∅
4 while Ready ∪ Active ≠ ∅ do
5   foreach op ∈ Active do
6     if S(op) + delay(op) < Cycle then
7       remove op from Active
8       foreach successor s of op in D do
9         if s is ready then
10          Ready ← Ready ∪ {s}
11  if Ready ≠ ∅ then
12    remove an op from Ready
13    S(op) ← Cycle
14    Active ← Active ∪ {op}
15  Cycle ← Cycle + 1
```

The algorithm maintains a simulation clock `Cycle` to track time.

`Ready` holds all the operations that can execute in the current cycle.

`Active` holds all operations that were issued in an earlier cycle but have not yet finished.

At each time step, it accounts for any operations completed in the previous cycle, it schedules an operation for the current cycle, and it increments `Cycle`.

Memory operations often have **uncertain** and **variable** delays.
- assuming the **worst-case** delay, risks idling the processor for long periods
- assuming the **best-case** delay, will stall the processor on a cache miss

The compiler can obtain good results by calculating an individual latency for each load based on the amount of instruction-level parallelism available to cover the load's latency.

This approach, called **balanced scheduling**, schedules the load with regard to the code that surrounds it rather than the hardware on which it will execute.
- mitigates cache misses by scheduling as much extra delay as possible for each load
- will not slow down execution in the absence of cache misses

```
1 foreach load operation l in the block do
2   delay(l) ← 1
3 foreach operation op in D do
4   let D_op be the nodes and edges in D independent of op
5   foreach connected component C of D_op do
6     find the maximal number of loads N on any path through C
7     foreach load operation l in C do
8       delay(l) ← delay(l) + delay(op)/N
```

Using this value as *delay*(l) produces a schedule that **shares the slack time** of independent operations evenly across all loads in the block.

The list-scheduling algorithm, as presented, makes several **assumptions** that may not hold true in practice.
1. most processors can issue **multiple operations per cycle**: scheduler needs to look for an operation for each functional unit in each cycle
2. some operations can execute on **multiple** functional units and others cannot: scheduler must chose an order for functional units
3. **partitioned** register set: scheduler may need to place an operation in the partition where its operands reside
4. **block boundaries**: invoke scheduler on blocks in reverse postorder on the CFG to understand when operands computed in predecessor blocks are available

In practice, list scheduling produces good results: it often builds optimal or near-optimal schedules. However, as with many greedy algorithms, its behavior is **not robust**.

The methodology used to **break ties** has a strong impact on the quality of schedules.

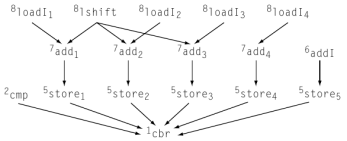
A typical scheduler has two or three tie-breaking **priority ranks** for each operation, which it applies in a consistent order.
- **node's latency-weighted path length**
- **number of node's immediate successors**: encourages the scheduler to pursue many distinct paths through the graph
- **total number of node's descendants**: amplifies the effect of the previous ranking
- **node's delay**: schedules long-latency operations as soon as possible
- **number of operands for which node is the last use**: moves last uses closer to their definitions

Critical operations occur near the leaves
- forward scheduling seems more likely to consider them together
- backward scheduling must work its way through the remainder of the block to reach them

Critical operations occur near the roots
- backward scheduling may examine them together
- forward scheduling sees them in an order dictated by decisions made starting at the other end of the block

Forward versus Backward List Scheduling

Example Below the dependence graph for a basic block found in the SPEC 95 benchmark program `go` is shown.



Opcode	Latency
loadI	1
lshift	1
add	2
addI	1
cmp	1
store	4

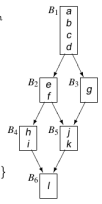
Note The compiler added dependences from the `store` operations to the block-ending branch to ensure that the memory operations complete before the next block begins.

	Integer	Integer	Memory
1	$loadI_1$	lshift	–
2	$loadI_2$	$loadI_3$	–
3	$loadI_4$	add_1	–
4	add_2	add_3	–
5	add_4	addI	$store_1$
6	cmp	–	$store_2$
7	–	–	$store_3$
8	–	–	$store_4$
9	–	–	$store_5$
10	–	–	–
11	–	–	–
12	–	–	–
13	cbr	–	–

	Integer	Integer	Memory
1	$loadI_4$	–	–
2	addI	lshift	–
3	add_4	$loadI_3$	–
4	add_3	$loadI_2$	$store_5$
5	add_2	$loadI_1$	$store_4$
6	add_1	–	$store_3$
7	–	–	$store_2$
8	–	–	$store_1$
9	–	–	–
10	–	–	–
11	cmp	–	–
12	cbr	–	–
13	–	–	–

3.2.2 Regional List Scheduling

An **extended basic block** (EBB) consists of a set of blocks $B_1, B_2, \ldots, B_n$ in which B_1 has multiple predecessors and every other block B_i has exactly one predecessor, some B_j in the EBB.

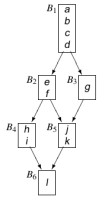


Example The CFG on the right has one large EBB, $\{B_1, B_2, B_3, B_4\}$, and two trivial EBBs, $\{B_5\}$ and $\{B_6\}$.

To obtain a **larger context** for list scheduling, the compiler can treat paths in an EBB, such $\{B_1, B_2, B_4\}$, as if they were single blocks.

When scheduling paths, the compiler needs to accounts for
- shared path prefixes, *e.g.*, B_1, which occurs in $\{B_1, B_2, B_4\}$ and $\{B_1, B_3\}$
- premature exits, *e.g.*, $B_1 \rightarrow B_3$ and $B_2 \rightarrow B_5$

To see how **shared prefixes** and **premature exits** complicate list scheduling, consider the possibilities for code motion in $\{B_1, B_2, B_4\}$.



Move an operation **forward**, *e.g.*, `c` from B_1 to B_2
- might speed execution along the path $\{B_1, B_2, B_4\}$
- changes the computation performed along the path $\{B_1, B_3\}$
- to fix this problem, the scheduler must insert a copy of `c` into B_3

Move an operation **backward**, *e.g.*, `f` from B_2 to B_1
- might speed execution along the path $\{B_1, B_2, B_4\}$
- inserts a computation of `f` into the path $\{B_1, B_3\}$, which might produce incorrect code along this path
- the scheduler must rewrite the code to undo that effect in B_3

Compensation Code

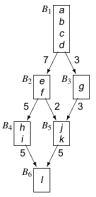
Code inserted into a block B_i to **counteract** the effects of cross-block code motion along a path that does not include B_i.

The issue of compensation code also makes clear the **order** in which the scheduler should consider paths in an EBB
→ since the first path scheduled receives little or no compensation code, the scheduler should choose paths in order of their **likely execution frequency**.

Trace scheduling constructs maximal-length acyclic paths through the CFG and applies the list-scheduling algorithm to those paths, or **traces**.

To build a trace, a simple **greedy approach** can be used that stops when it either runs out of possible edges or encounters a loop-closing branch.

Example The trace for the example on the right is $\{B_1, B_2, B_4, B_6\}$.



With respect to EBB scheduling, one **additional** opportunity for compensation code occurs: a trace may have interim entry points, *i.e.*, blocks in mid-trace that have **multiple** predecessors.

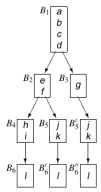
Note EBB scheduling can be considered a degenerate case of trace scheduling in which interim entries to the trace are prohibited.

Join points in the CFG **limit the opportunities** for either EBB scheduling or trace scheduling.

The compiler can **clone blocks** to create longer join-free paths.
- for EBB scheduling, it increases the size of the EBB and the length of some of the paths through the EBB
- for trace scheduling, it avoids the complications caused by interim entry points in the trace

After cloning, the entire graph forms **one single** EBB.

The compiler can combine blocks that are linked by an edge where the source has no other successors and the sink has no other predecessors.



3.3 Register Allocation

Register-to-register model
- knowledge about ambiguous memory references is encoded in the IR (*cf.* Slide 330)
- allocation is a necessary part of the process that produces legal code
- allocator inserts loads and stores to move some register-based values into memory

Memory-to-memory model
- allocator must determine which values can be kept safely in registers
- allocator must determine whether keeping values in registers is profitable
- allocation improves performance by eliminating some loads and stores

Allocation maps an unlimited name space onto the register set of the target machine
- **register-to-register model**: virtual registers are mapped to name set modelling physical register set and values that do not fit in the register set are spilled
- **memory-to-memory model**: a subset of the memory locations is mapped to a set of names that models the physical register set

Assignment maps an allocated name set to the physical registers of the target machine
- assumes that allocation has been performed
- produces the actual register names required by the executable code

Local allocation addresses the problems that arise in producing a good allocation for a **single** basic block.

To simplify discussion, we assume the following
- the block the is the **entire program**: it loads the values that it needs from memory and it stores the values that it produces to memory
- the block uses a **single class** of general-purpose registers
- the target machine provides a **single set** of k physical registers
- the code keeps any value that can legally reside in a register in a register, *i.e.*, it uses as many **virtual registers** as needed to encode this information
- the block contains a series of **three-address operations** $o_1, o_2, o_3, \ldots, o_N$ and each operation o_i has the form $op_i\ vr_{i_2}, vr_{i_2} \Rightarrow vr_{i_3}$

The allocator must ensure that the output code **has no more than** k values in registers at any point in the block.

Rather than allocating variables or values to registers, regional and global allocators compute a name space that is defined in terms of **live ranges**.

> **Live range**
>
> A closed set of definitions and uses that are related to each other because their values flow together.

The term live range, implicitly, relies on the notion of **liveness**.

> **Liveness**
>
> A variable v is live at point p if it has been defined along a path from the procedure's entry to p and there exists a path from p to a use of v along which v is not redefined.

In straighline code, we can represent a live range as an interval $[i, j]$ where operation i is its **definition** and operation j is its **last use**.

```
 1  loadI   ...          ⇒ r_arp
 2  loadAI  r_arp, @a    ⇒ r_a
 3  loadI   2            ⇒ r_2
 4  loadAI  r_arp, @b    ⇒ r_b
 5  loadAI  r_arp, @c    ⇒ r_c
 6  loadAI  r_arp, @d    ⇒ r_x
 7  mult    r_a, r_2     ⇒ r_a
 8  mult    r_a, r_b     ⇒ r_a
 9  mult    r_a, r_c     ⇒ r_a
10  mult    r_a, r_d     ⇒ r_a
11  storeAI r_a          ⇒ r_arp, @a
```

	Register	Interval
1	r_{arp}	$[1, 11]$
2	r_a	$[2, 7]$
3	r_a	$[7, 8]$
4	r_a	$[8, 9]$
5	r_a	$[9, 10]$
6	r_a	$[10, 11]$
7	r_2	$[3, 7]$
8	r_b	$[4, 8]$
9	r_c	$[5, 9]$
10	r_d	$[6, 10]$

At any point in the code, only live values need registers. LIVEOUT sets encode precisely this knowledge.

> **LIVEOUT**
>
> For each block b, the set LIVEOUT(b) contains all the variables that are live on exit from b.

Any value in LIVEOUT(b) must be **stored** to its assigned location in memory after its last definition in b to ensure that the correct value is available in a subsequent block.

3.3.1 Local Register Allocation

Top-Down

The **top-down local allocator** works from a simple heuristic: the **most heavily used** values should reside in registers.
- count the number of times (frequency) that each virtual register appears in the block
- allocates virtual registers to physical registers in descending order of frequency

If there are more virtual registers than physical registers, the allocator must reserve enough physical registers to load, store, and use the values that are not kept in registers.

> $\mathcal{F}$
>
> On any given ISA, $\mathcal{F}$ is the number of registers needed to generate code for values that live in memory. We pronounce $\mathcal{F}$ "feasible".

If the block uses **fewer than** k virtual registers, allocation is trivial and the compiler can simply assign each `vr` to its own physical register.

If the block uses **more than** k virtual registers, the compiler applies the following simple algorithm.
1. compute a priority for each virtual register
2. sort the virtual registers into priority order
3. assign the first $k - \mathcal{F}$ virtual registers to physical registers in priority order
4. rewrite the code
   - replace virtual register names with physical register names
   - replace virtual register names with no allocated physical register with code that uses one of the reserved register and performs the appropriate load or store

Example Assume a target machine with four physical registers r_1, r_2, r_3, and r_4 as well as two spill registers f_1 and f_2. Both r_{arp} and r_i are **live** upon entry into the block.

```
loadAI  r_arp, 12 ⇒ r_a
loadAI  r_arp, 16 ⇒ r_b
add     r_i, r_a  ⇒ r_c
sub     r_b, r_i  ⇒ r_d
mult    r_c, r_d  ⇒ r_e
multI   r_b, 2    ⇒ r_f
add     r_e, r_f  ⇒ r_g
storeAI r_g       ⇒ r_arp, 8
```

r_{arp}	r_a	r_b	r_c	r_d	r_e	r_f	r_g	r_i
3	2	3	2	2	2	2	2	2

```
loadAI  r_arp, 12 ⇒ r_1
loadAI  r_arp, 16 ⇒ r_2
add     r_4, r_1  ⇒ r_3
sub     r_2, r_4  ⇒ f_1
storeAI f_1       ⇒ r_arp, 20
mult    r_3, f_1  ⇒ f_1
storeAI f_1       ⇒ r_arp, 24
multI   r_2, 2    ⇒ f_1
storeAI f_1       ⇒ r_arp, 28
loadAI  r_arp, 24 ⇒ f_2
add     f_2, f_1  ⇒ f_1
storeAI f_1       ⇒ r_arp, 8
```

Bottom-Up

The primary weakness of top-down allocation is that it dedicates a physical register to one virtual register for the **entire** basic block.

The key idea behind the **bottom-up local allocator** is to focus on the details of how values are defined and used on an operation-by-operation basis.
1. start with an empty register set
2. allocate registers on demand
3. if no register is available, free one
   - keep values that will be "used soon" in registers
   - spill register whose next use is farthest in the future

```
1  foreach op_i vr_i1, vr_i2 ⇒ vr_i3, in order
     i = 1 ... N do
2    r_x ← Ensure(vr_i1, class(vr_i1))
3    r_y ← Ensure(vr_i2, class(vr_i2))
4    if vr_i1 is not needed after op_i then
5      Free(r_x, class(r_x))
6    if vr_i2 is not needed after op_i then
7      Free(r_y, class(r_y))
8    r_z ← Allocate(vr_i3, class(vr_i3))
9    rewrite op_i as op_i r_x, r_y ⇒ r_z
10   if vr_i1 is needed after op_i then
11     class.Next[r_x] ← Dist(vr_i1)
12   if vr_i2 is needed after op_i then
13     class.Next[r_y] ← Dist(vr_i2)
14   class.Next[r_z] ← Dist(vr_i3)
```

The bottom-up allocator iterates over the operations in the block, making allocation decisions on demand.

By considering vr_{i_1} and vr_{i_2} in order, the allocator avoids using two physical registers for an operation with a repeated operand.

Trying to free r_x and r_y before allocating r_z avoids spilling a register to hold an operation's result when the operation actually frees a register.

The function `Dist(vr)` returns the index in the block of the next reference to `vr`.

```
1  procedure Ensure(vr, class)
2    if vr is already in class then
3      result ← vr's physical register
4    else
5      result ← Allocate(vr, class)
6      emit code to move vr into result
7    return result
```

`Ensure` takes two arguments: a virtual register `vr` holding the desired value, and a representation for the appropriate register class `class`.
- if `vr` already occupies a physical register, `Ensure`'s job is done
- otherwise, it allocates a physical register for `vr` and emits code to move `vr`'s value into that physical register

In either case, it returns the physical register.

```
1  procedure Allocate(vr, class)
2    if class.StackTop ≥ 0 then
3      i ← pop(class)
4    else
5      i ← j that maximizes class.Next[j]
         store contents of j
6    class.Name[i] ← vr
7    class.Next[i] ← -1
8    class.Free[i] ← false
9    return i
```

```
struct Class {
  int Size;
  int Name[Size];
  int Next[Size];
  int Free[Size];
  int Stack[Size];
  int StackTop;
}
```

`Allocate` returns a physical register from the free list of class, if one exists.

Otherwise, it selects the value stored in class that is used farthest in the future, spills it, and reallocates the corresponding physical register to `vr`.

```
1  procedure Free(vr, class)
2    i ← push(class)
3    class.Name[i] ← -1
4    class.Next[i] ← ∞
5    class.Free[i] ← true
```

`Free` simply pushes the freed register onto the stack and resets its fields to their initial values.

Example Assume a target machine with four physical registers r_1, r_2, r_3, and r_4 as well as two spill registers f_1 and f_2. Both r_{arp} and r_i are **live** upon entry into the block.

```
loadAI  r_arp, 12 ⇒ r_a
loadAI  r_arp, 16 ⇒ r_b
add     r_i, r_a  ⇒ r_c
sub     r_b, r_i  ⇒ r_d
mult    r_c, r_d  ⇒ r_e
multI   r_b, 2    ⇒ r_f
add     r_e, r_f  ⇒ r_g
storeAI r_g       ⇒ r_arp, 8
```

```
loadAI  r_arp, 12 ⇒ r_1
loadAI  r_arp, 16 ⇒ r_2
add     r_4, r_1  ⇒ r_3
sub     r_2, r_4  ⇒ r_4
mult    r_1, r_4  ⇒ r_1
multI   r_2, 2    ⇒ r_2
add     r_1, r_2  ⇒ r_1
storeAI r_1       ⇒ r_arp, 8
```

Note Due to freeing operand registers before allocating result registers (*cf.* Lines 4–7 on Slide 516), the bottom-up allocator avoids all spills.

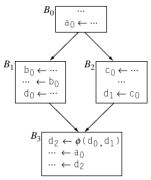
3.3.2 Global Register Allocation

To construct live ranges, the compiler must discover the **relationships** that exist among different definitions and uses, *i.e.*, group together into a single name
- all the definitions that reach a single use
- all the uses that a single definition can reach

The **SSA form** of the code (*cf.* Slide 265) provides a natural starting point
- each name is defined once, and each use refers to one definition
- ϕ-function record the fact that distinct definitions on different paths in the control-flow graph reach a single reference

To build live ranges from SSA form, the allocator uses the **disjoint-set union-find algorithm** and makes a single pass over the code
1. treat each SSA name, or definition, as a set in the algorithm
2. union sets of each ϕ-function parameter with the set for the ϕ-function result

Example Consider the code fragment in SSA form shown on the right. It involves source-code variables a, b, c, and d.



To find the live ranges, the allocator assigns each ssa name a set containing its name. It unions together the sets associated with names used in the ϕ-function: $\{d_0\} \cup \{d_1\} \cup \{d_2\}$.

This gives a final set of four live ranges: LR_a that contains $\{a_0\}$, LR_b that contains $\{b_0\}$, LR_c that contains $\{c_0\}$, and LR_d that contains $\{d_0, d_1, d_2\}$.

At this point, the allocator can either **rewrite** the code to use live-range names or it can create and maintain a **mapping** between SSA names and live-range names.

The cost of a spill has three components: the **address computation**, the **memory operation**, and an **estimated execution frequency**.

Address Computation
- to minimize the cost of the address computation spill into register-save area in AR
- use operations such as `loadAI` or `storeAI` relative to r_{arp} for the spill

Memory Operation
- the cost of the memory operation is, in general, unavoidable
- some embedded processors have scratchpad memory, *i.e.*, noncached local memory

Estimated Execution Frequency
- profile data
- heuristics, *e.g.*, loops execute 10 times, `if-then-else` decreases frequency by half

The spill cost of a live range can also be **negative** or **infinite**.

Any live range with a negative spill cost should be spilled
- decreases demand for registers and removes instructions from the code
- *e.g.*, a live range that contains a load, a store, and no other uses

Some live ranges are so short that spilling them does not help
- assigning a spill cost of infinity ensures that the allocator does not try to spill it
- a live range should have infinite spill cost if no other live range ends between its definitions and its uses

The fundamental effect that a global register allocator must model is the **competition** among values for space in the processor's register set.

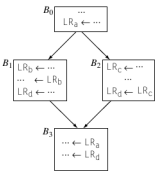
> **Interference**
> Two live ranges, LR_i and LR_j interfere if one is live at the definition of the other and they have different values.

To model the allocation problem, the compiler can build an **interference graph** $I = (N, E)$
- nodes in N represent individual live ranges
- edges in E represent interferences between live ranges

An undirected edge $(n_i, n_j) \in I$ exists if and only if the corresponding live ranges LR_i and LR_j interfere.

Interferences and the Interference Graph

Example On the left, the code (rewritten to use live ranges) from the example on Slide 529 is shown. The corresponding interference graph is shown on the right.



The interference graph shows that LR_a interferes with **each** of the other live ranges.

The rest of the live ranges, however, **do not** interfere with each other.

Top-Down

Once the allocator has built global live ranges and annotated each block in the code with its LIVEOUT set, it can construct the interference graph in a linear pass over each block.

```
1 foreach LR_i do
2   create a node n_i ∈ N
3 foreach basic block b do
4   LIVENOW ← LIVEOUT(b)
5   foreach operation op_n, op_{n-1}...op_1 in b with form op_1 LR_a, LR_b ⇒ LR_c do
6     foreach LR_i ∈ LIVENOW do
7       E ← E ∪ {(LR_c, LR_i)}
8     LIVENOW ← LIVENOW − LR_c
9     LIVENOW ← LIVENOW ∪ {LR_a} ∪ {LR_b}
```

Top-down allocators try to color live ranges in an order given by a **ranking function**.

The **priority-based**, **top-down allocators** assign each node a rank that is the estimated runtime savings that accrue from keeping that live range in a register.
1. consider live ranges in rank order and attempt to assign a color to each of them
2. if no color is available, invoke the spilling or splitting mechanism

To improve the process, the allocator can partition the live ranges into two sets: **constrained** live ranges (k or more neighbors) and **unconstrained** live ranges.
- first, constrained live ranges are colored in rank order
- then, unconstrained live ranges are colored in any order

Because an unconstrained live range has fewer than k neighbors, the allocator can always find a color for it: no assignment of colors to its neighbors can use all k colors.

To spill LR_i, the allocator inserts a store after **every definition** of LR_i and a load before **each use** of LR_i.

If memory operations need registers, the allocator can **reserve registers** to handle them.
- the number of registers needed is determined by the instruction set architecture
- reserving these registers simplifies spilling

An alternative to reserving registers is to **look for free colors** at each definition and use.
- if no color is available, retroactively spill a previously colored live range
- this scheme has the potential to spill previously colored live ranges recursively

This feature has led most implementors of top-down, priority-based allocators to reserve spill registers instead.

Another way to change the problem is to **split an uncolored live range** into new live ranges, *i.e.*, subranges that contain several references.
- live-range splitting can avoid spilling the original live range at every reference
- well-chosen split points, it can isolate the portions of the live range that must be spilled

Example The first top-down, priority-based coloring allocator, broke the uncolored live range into single-block live ranges.
- it counted interferences for each resulting live range, and then recombined live ranges from adjacent blocks if the combined live range remained unconstrained
- it placed an arbitrary upper limit on how many blocks a split live range could span
- it inserted a load at the starting point of each split live range and a store at the live range's ending point
- it spilled any split live ranges that remained uncolored

Bottom-Up

The major distinction between top-down and bottom-up allocators lies in the **mechanism used to order** live ranges for coloring.
- a top-down allocator uses high-level information to select an order for coloring
- a bottom-up allocator computes an order from detailed structural knowledge about the interference graph

To order the live ranges, a bottom-up, graph-coloring allocator relies on the fact that unconstrained live ranges are **trivial** to color.

It assigns colors in an order where every node has **fewer** than k colored neighbors.

The algorithm computes the coloring order for a graph $I = (N, E)$ as follows.

```
1 initialize stack to empty
2 while N ≠ ∅ do
3   if ∃n ∈ N : deg(n) < k then
4     node ← n
5   else
6     node ← n picked from N
7   remove node and its edges from I
8   push node onto stack
```

It uses two **distinct mechanisms** to select the node to remove next.
- the first clause takes a node that is **unconstrained** in the graph from which it is removed
- the second clause, invoked only when every remaining node is **constrained**, picks a node using some external criteria

The loop halts when the graph is empty. At that point, the stack contains all the nodes in order of removal.

To color the graph, the allocator rebuilds the interference graph in the order represented by the stack, *i.e.*, in **reverse order** of removing them from the graph.

```
1 while stack ≠ ∅ do
2   pop node from stack
3   insert node and its edges into I
4   color node
```

The allocator tallies the colors of node's neighbors in the current approximation to I and assigns node an unused color.

If no color remains, node is left uncolored.

When the stack is empty, I has been rebuilt.
- if every node has a color, the allocator declares success and rewrites the code
- if nodes remain uncolored, the allocator spills or splits the corresponding live range

At this point, the classic bottom-up allocators rewrite the code to reflect the spills and splits and **repeat** the entire process.

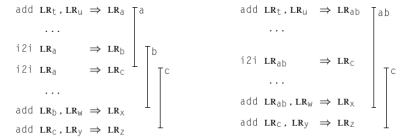
The compiler writer can use the interference graph to determine when two live ranges that are connected by a copy can be **coalesced**, or combined.

Combining two live ranges LR_i and LR_j has **several beneficial effects**
- it eliminates the copy operation, making the code smaller and, potentially, faster
- it reduces the degree of any LR_k that interfered with both LR_i and LR_j
- it shrinks the set of live ranges, making I and many related data structures smaller

Because these effects help in allocation, compilers often perform coalescing **before** the coloring stage in a global allocator.

Coalescing Copies to Reduce Degree

Example The original code appears on the left, with lines to the right that indicate the regions where each of the relevant values, LR_a, LR_b, and LR_b are live.



On the right, the result of coalescing LR_a and LR_b to produce LR_{ab} is shown. Since LR_c is defined by a copy from LR_{ab}, they do not interfere.

Comparing Top-Down and Bottom-Up Global Allocators

Both the top-down and the bottom-up coloring allocators have the **same basic structure**.