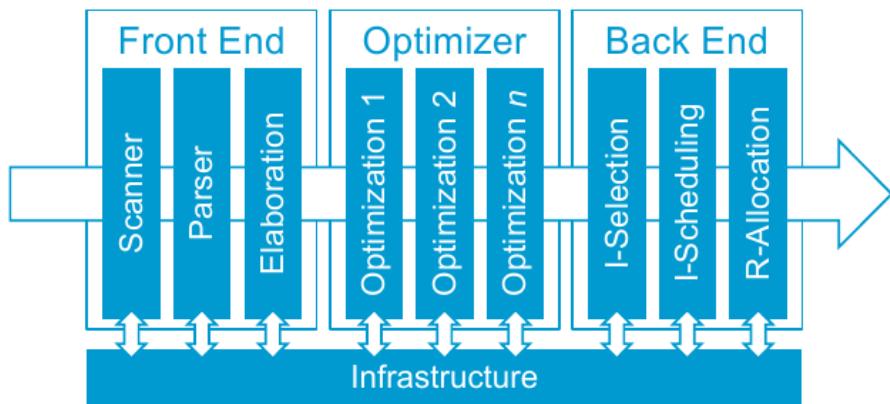


## Typical Structure of a Compiler



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

<b>1 Front-End</b>	<b>3</b>
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: Context-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
<b>2 Core: Intermediate Representation and Code Shape</b>	<b>7</b>
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 . . . . .	10
2.8.3 Referencing . . . . .	10
2.9 Strings . . . . .	11
2.10 Structure References . . . . .	11
2.10.1 Structure Layout . . . . .	11
2.10.2 Runtime Dependent Types . . . . .	11
2.11 Control Flow . . . . .	11
2.11.1 Conditional Execution . . . . .	11
2.11.2 Loops . . . . .	12
2.11.3 Case Statements . . . . .	12
2.11.4 Procedure Calls . . . . .	12
<b>3 Back End</b>	<b>13</b>
3.1 Instruction Selection . . . . .	13
3.1.1 Tree Pattern-Matching . . . . .	13
3.1.2 Peephole Optimization . . . . .	13
3.2 Instruction Scheduling . . . . .	13
3.2.1 Local List Scheduling . . . . .	13
3.2.2 Regional List Scheduling . . . . .	13
3.3 Register Allocation . . . . .	13
3.3.1 Local Register Allocation . . . . .	13
3.3.2 Global Register Allocation . . . . .	13

# 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  $s_e$ .
- $\Sigma$  is the finite alphabet used by the recognizer. Typically,  $\Sigma$  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.
- $S_A \subseteq S$  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,  $\Sigma$ , augmented with a character  $\epsilon$  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  $R | S$ , is  $\{x | x \in R \vee x \in S\}$ .
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  $\{xy | x \in R \wedge y \in S\}$ .
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.

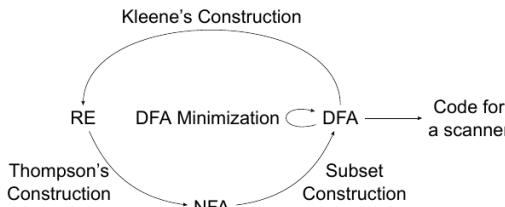
**Finite Closure**  $R^i$ : one to  $i$  occurrences of  $R$ , e.g.,  $R^5 = (R|RR|RRR)$

**Positive Closure**  $R^+$ : one or more occurrences of  $R$ , i.e.,  $RR^*$

**Ranges**  $[x_0 \dots x_n]$ : abbreviate ranges e.g.,  $[0 \dots 3] = \{0 | 1 | 2 | 3\}$

**Complement**  ${}^c\Sigma$ : complement of  $\Sigma$  with respect to  $\Sigma$ , 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.

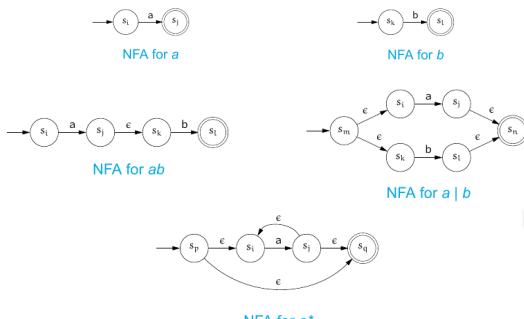
1. **Thompson's construction** derives a Nondeterministic FA (NFA) from a RE
2. **Subset construction** builds a DFA that simulates the NFA
3. **Hopcroft's algorithm** minimizes the DFA

#### 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.

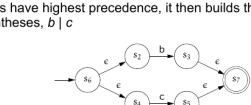
**Thompson's construction builds an NFA from an RE in a straightforward way**

- one template for building each NFA that corresponds to a single-letter RE
- NFA transformations for RE operators: concatenation, alternation, and closure

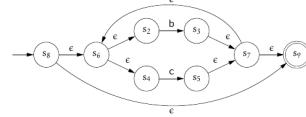


As an example, we use Thompson's Construction to derive the NFA for the RE  $a(b | c)^*$ .

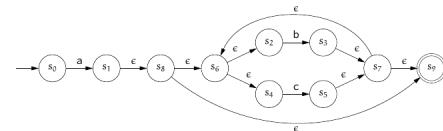
1. The construction begins by first building NFAs for  $a$ ,  $b$ , and  $c$
2. Since parentheses have highest precedence, it then builds the NFA for the expression enclosed in parentheses,  $b | c$



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  $(N, \Sigma, \delta_N, q_0, N_A)$  and produces a DFA  $(D, \Sigma, \delta_D, d_0, D_A)$

- the NFA and the DFA use the same alphabet  $\Sigma$
- the NFA's start state  $q_0$  and its accepting states  $N_A$  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  $\delta_D$

```

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(Δ(q,c))
8     if t ∉ Q then
9       add t to Q
10    add t to WorkList
11
  
```

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

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

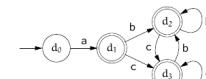
1. The initialization sets  $q_0$  to  $\epsilon$ -closure( $\{n_0\}$ ), which is just  $n_0$
2. The 1<sup>st</sup> iteration computes  $\epsilon$ -closure( $\Delta(q_0, a)$ ), which contains six NFA states, and  $\epsilon$ -closure( $\Delta(q_0, b)$ ) and  $\epsilon$ -closure( $\Delta(q_0, c)$ ), which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

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

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

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  $n_0$  (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)
  
```

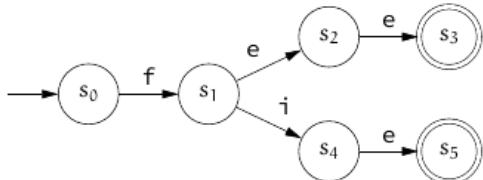
```

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  $P = \{p_1, p_2, \dots, p_m\}$  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

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



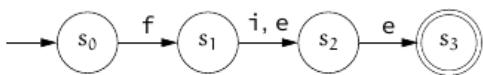
The algorithm takes the following steps

- Initialization separates accepting from nonaccepting states:  $\{(s_3, s_5), \{s_0, s_1, s_2, s_4\}\}$
- Step 1 examines  $\{s_3, s_5\}$ : neither state has an exiting transition, i.e., no split occurs
- Step 2 examines  $\{s_0, s_1, s_2, s_4\}$ : on character e, it splits  $\{s_2, s_4\}$  out of the set
- Step 3 examines  $\{s_0, s_1\}$ : on character f, it splits  $\{s_1\}$  out of the set
- 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

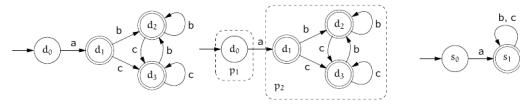
**Note** We assume that the various loops in the algorithm iterate over the sets of P and over the characters in  $\Sigma = \{e, f, i\}$  in order.

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

Step	Current Partition	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



### 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?**

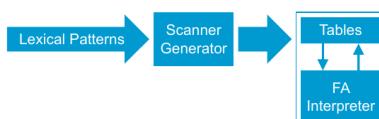
- if  $s$  is an **accepting state**: report the recognized word and its syntactic category
- if  $s$  is an **nonaccepting state**
  - if possible, roll back to most recent accepting state to match **longest valid prefix**
  - otherwise, report an **error** since no prefix of the input string is a valid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

#### Scanner generator

- creates NFA for each RE defined by compiler writer for syntactic classes
- joins all resulting NFAs by  $\epsilon$ -transitions
- translates NFA into a corresponding DFA
- minimizes DFA
- 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  $\text{CharCat}$  maps each  $\Sigma$  to a category
- transition table  $\delta$  encodes the transitions of the underlying DFA
- token type table  $\text{Type}$  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 ←  $s_0$ 
2 lexeme ← ""
3 clear stack
4 push("bad")
5 while state ≠  $s_A$ , do
6   NextChar(char)
7   lexeme ← lexeme + char
8   if state ∈  $S_A$  then
9     clear stack
10    push(state)
11    cat ← CharCat[char]
12    state ←  $\delta$ [state, cat]
13 while state ∉  $S_A$  ∧ state ≠ "bad" do
14   state ← pop()
15   truncate lexeme
16   Rollback()
17   if state ∈  $S_A$  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  $\text{CharCat}$  and  $\delta$  encode the DFA. To update the state, a **two-step translation** is performed.

1. character  $\rightarrow$  category
  2. current state and category  $\rightarrow$  new state
- This approach lets the scanner use a compressed transition table.

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

#### Classifier Table $\text{CharCat}$

Larger character sets, e.g., Unicode, may need a more complex data structure to represent  $\text{CharCat}$ .

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  $\text{InputPos}$  to record position in the input stream
2. a two-dimensional bit-array  $\text{Failed}$  to record dead-end transitions
3. additional routine  $\text{InitializeScanner}()$  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 ≠  $s_e$  do
2   ...
3   cat ← CharCat[char]
4   state ←  $\delta$ [state, cat]

```

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

```

- CharCat: @CharCat_0 + i × w
- δ: δ_0 + (state × |columns in δ| + cat) × 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  $s_{int}$ 
1 lexeme ← ""
2 clear stack
3 if state ∈  $S_A$  then
4   goto 5
5 // State  $s_A$ 
6 NextChar(char)
7 lexeme ← lexeme + char
8 if state ∈  $S_A$  then
9   push(state)
10  if '0' ≤ char ≤ '9' then
11    goto 23
12  else
13    goto 32
14  if state ∈  $S_A$  then
15    push(state)
16    if '0' ≤ char ≤ '9' then
17      goto 23
18    else
19      goto 32
20  else
21    if state ∈  $S_A$  then
22      push(state)
23  lexeme ← lexeme + char
24  if state ∈  $S_A$  then
25    return Type[state];
26  else
27    push(state)
28  if '0' ≤ char ≤ '9' then
29    goto 23
30  else
31    goto 32
32 // State  $s_A$ 
33 while state ∈  $S_A$  ∧ state ≠ bad do
34   state ← pop()
35   truncate lexeme
36   Rollback()
37 if state ∈  $S_A$  then
38   return Type[state];
39 else
40   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: Context-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.
- $P$  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.

**Ambiguous 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 for 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
  - c) 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 ( $A \rightarrow \beta_1, \beta_2, \dots, \beta_n$ )
8   |   | build nodes for  $\beta_1, \beta_2, \dots, \beta_n$  as children of focus
9   |   | push( $\beta_n, \beta_{n-1}, \dots, \beta_2$ )
10  |   | focus ←  $\beta_1$ 
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

$$\begin{array}{l} A \rightarrow A\alpha \\ | \quad \beta \\ \quad \quad \quad A' \rightarrow \beta A' \\ \quad \quad \quad | \quad \epsilon \end{array}$$

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  $A' \rightarrow \epsilon$ , where  $\epsilon$  represents the empty string

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

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

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  $A_1, A_2, \dots, A_n$ 
2 for i ← 1 to n do
3   | for j ← 1 to i - 1 do
4   |   | if there is a production  $A_i \rightarrow A_j\gamma$  and  $A_j \rightarrow \delta_1|\delta_2|\dots|\delta_n$  then
5   |   |   | replace  $A_i \rightarrow A_j\gamma$  with a set of productions  $A_i \rightarrow \gamma|\delta_2\gamma|\dots|\delta_n\gamma$ 
6   |   | rewrite the productions to eliminate any direct left recursion on  $A_i$ 

```

**Note** This algorithm assumes that the original grammar has no cycles ( $A \rightarrow^+ A$ ) and no  $\epsilon$ -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  $\alpha$ ,  $\text{FIRST}(\alpha)$  is the set of terminals that can appear at the start of a sentence derived from  $\alpha$ .

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

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

```

1 foreach  $\alpha \in T \cup \{\epsilon, \text{eof}\}$  do
2   | FIRST( $\alpha$ ) ←  $\alpha$ 
3 foreach  $A \in NT$  do
4   | FIRST( $A$ ) ←  $\emptyset$ 
5 while FIRST sets are still changing do
6   | foreach  $p \in P$  of the form  $A \rightarrow \beta_1\beta_2\dots\beta_k$ , where  $\beta_i \in T \cup NT$  do
7   |   | rhs ← FIRST( $\beta_1$ ) - { $\epsilon$ }
8   |   | i ← 1
9   |   | while  $\epsilon \in \text{FIRST}(\beta_i)$  and  $i \leq k - 1$  do
10  |   |   | rhs ← rhs ∪ (FIRST( $\beta_{i+1}$ ) - { $\epsilon$ })
11  |   |   | i ← i + 1
12  |   | if  $i = k$  and  $\epsilon \in \text{FIRST}(\beta_k)$  then
13  |   |   | rhs ← rhs ∪ { $\epsilon$ }
14  |   | FIRST( $A$ ) ← FIRST( $A$ ) ∪ rhs

```

### Dealing with $\epsilon$ -productions

- parser should apply the  $\epsilon$ -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  $\epsilon$ -production

### FOLLOW

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

```

1 foreach  $A \in NT$  do
2   | FOLLOW( $A$ ) ←  $\emptyset$ 
3 FOLLOW(S) ← {eof}
4 while FOLLOW sets are still changing do
5   | foreach  $p \in P$  of the form  $A \rightarrow \beta_1\beta_2\dots\beta_k$ , where  $\beta_i \in T \cup NT$  do
6   |   | lhs ← FOLLOW(A)
7   |   | for i ← k down to 1 do
8   |   |   | if  $\beta_i \in NT$  then
9   |   |   |   | FOLLOW( $\beta_i$ ) ← FOLLOW( $\beta_i$ ) ∪ lhs
10  |   |   |   if  $\epsilon \in \text{FIRST}(\beta_i)$  then
11  |   |   |   | lhs ← lhs ∪ (FIRST( $\beta_i$ ) - { $\epsilon$ })
12  |   |   |   else
13  |   |   |   | lhs ← lhs ∪ FIRST( $\beta_i$ )
14  |   |   |   else
15  |   |   |   | lhs ← FIRST( $\beta_i$ ) // Note that since  $\beta_i \notin NT$ , FIRST( $\beta_i$ ) = { $\beta_i$ }

```

### Backtrack-Free Grammar

For a production  $A \rightarrow \beta$ , we define its augmented FIRST set,  $\text{FIRST}^+$ .

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

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

$$\forall 1 \leq i, j \leq n, i \neq j : \text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset$$

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

$$A \rightarrow \alpha\beta_1|\alpha\beta_2|\dots|\alpha\beta_m$$

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

$$\begin{aligned} A &\rightarrow \alpha B|\alpha\beta_1|\dots|\alpha\beta_m \\ B &\rightarrow \beta_1|\beta_2|\dots|\beta_n \end{aligned}$$

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 Left to right
2. construct Leftmost 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 derive 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($)
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  $A \rightarrow B_1B_2\dots B_k$  then
14  |   |   | stack.pop()
15  |   |   | for i ← k down to 1 do
16  |   |   |   | if  $B_i \neq \epsilon$  then stack.push( $B_i$ )
17  |   |   | else report an error expanding focus
18  |   | focus ← stack.peek()

```

Given a nonterminal  $A$  and a lookahead symbol  $w$ ,  $\text{table}[A, w]$  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  $A \in NT$  do
3   | foreach  $w \in T$  do
4     |   | table[A, w] ← error
5   | foreach  $p \in P$  with form  $A \rightarrow \beta$  do
6     |   | foreach  $w \in \text{FIRST}^+(A \rightarrow \beta)$  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 Left to right
2. construct reversed Rightmost 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  $(A \rightarrow \beta, k)$  on the upper frontier of this partially completed parse tree
2. replace the occurrence of  $\beta$  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  $\beta$  of a handle
- reduces by the production  $A \rightarrow \beta$  in the handle
  - pops the symbols in  $\beta$  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  $(A \rightarrow \beta, k)$  be constant and implicit.

```

1 stack.push($)
2 stack.push(s0)
3 word ← NextWord()
4 loop
5 | state ← stack.peek()
6 | If Action[state, word] = "reduce A → β" then
7 | | stack.pop(2 × |β| symbols)
8 | | state ← stack.peek()
9 | | stack.push(A)
10 | | stack.push(Goto[state, A])
11 | else if Action[state, word] = "shift si" then
12 | | stack.push(word)
13 | | stack.push(si)
14 | | word ← NextWord()
15 | else if Action[state, word] = "accept" then
16 | | break()
17 | else
18 | | report an error
19 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 common by all objects of that type. Types can be specified by membership (e.g. defining the set of integers 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 behavior 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 called **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 performance perspective, dynamic typing is really bad.

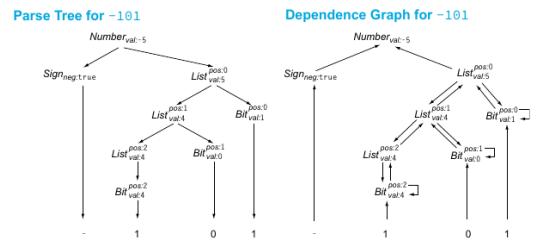
## Components

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 whether 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 the programmer to obey reasonable rules without complete prior knowledge

### 1.3.2 Attribute-Grammar Framework

- Formalism to perform context-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. don't 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 restricted to **noncircular** grammars.

Attribute grammars 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 enforce 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.

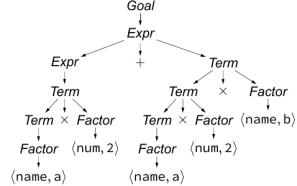
### 2.1 Graph-based IRs

#### 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.

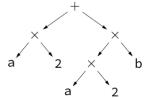
```

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

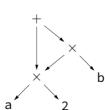
A **directed acyclic graph (DAG)** is a contraction of the AST that avoids this duplication. Identical subtrees are instantiated once, with multiple parents.

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

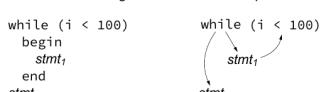
##### Control-Flow Graph

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.



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

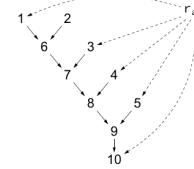
#### 2.1.5 Dependence Graph

- 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

```

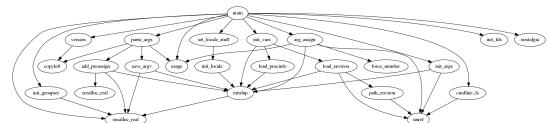
1 loadAI rarp, @a ⇒ ra
2 loadI 2 ⇒ r2
3 loadAI rarp, @b ⇒ rb
4 loadAI rarp, @c ⇒ rc
5 loadAI rarp, @d ⇒ rd
6 mult ra, r2 ⇒ ra
7 mult ra, rb ⇒ ra
8 mult ra, rc ⇒ ra
9 mult ra, rd ⇒ ra
10 storeAI ra ⇒ rarp, @a
  
```



**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.

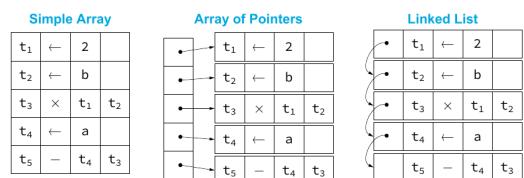
#### 2.1.6 Call Graph

- 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  $a - 2 \times b$**

```

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  $i \leftarrow j \text{ op } k$

- one operator
- two operands ( $j$  and  $k$ )
- one result ( $i$ )

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

**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

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

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

## 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  $\phi$ -functions, is inserted at points where control-flow paths meet.

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

### $\phi$ -function

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

The  $\phi$ -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  $\phi$ -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:**  $\phi$ -functions do not conform 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 callees

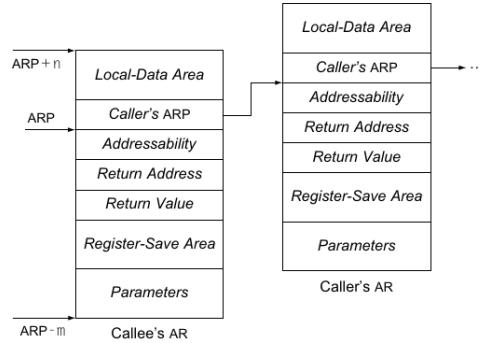
### 2.4.1 Calls

- 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 inaccessible for the outside name space. A **Scope** refers to a Name Space. **Lexical scopes** are nested name spaces that nested as they are encountered during 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 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 crucial.

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 being 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 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.

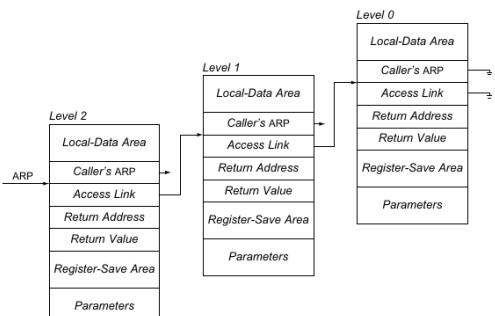
**Addressing in a Stack Frame** The memory region that holds the data for a scope is called **data area**. The address at the beginning of the data area is called **base address**. The base address can be static or dynamic (wrt. compile and run time).

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).

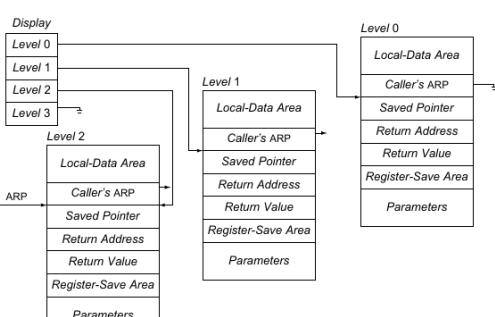
For procedures the compiler constructs an AR with a dynamic base address. Lexical scoping declares rules for which ARs shall be accessible from where.

**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

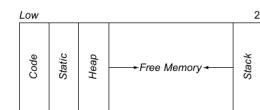
The compiler must assign storage locations to all variables. We can distinguish between **named** and **unnamed or anonymous** values. Named values follow the convention of the source language, while unnamed values must be handled consistently with those, but effectively location and lifetime are managed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

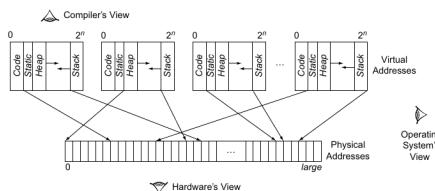
- compiler assumes that it has enough registers to express the computation
- use a distinct **virtual register** for each value that can legally reside in a register
- store a virtual register's value to memory only when absolutely necessary



#### 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).

<sup>1</sup>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 unambiguous values 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(loadA, t1, t2, r)
12  else if n = num then
13    r ← NextRegister()
14    emit(loadI, n, _, r)
15  return r

```

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. save 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**  $\equiv \neg \text{False}$  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 zero  
**CF** result caused carry out of most significant bit  
**SF** result was negative (sign bit was set)  
**OF** result caused (signed) overflow

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

**jcc** jumps to the specified label if cc holds  
**setcc** sets the given (single-byte) register to 1 or 0 depending on whether cc holds or not  
**cmovecc** 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.

$$\begin{aligned} \forall x \text{ false} \wedge x &= \text{false} \\ \forall x \text{ true} \vee x &= \text{true} \end{aligned}$$

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

## Hardware Support for RelOps

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

Source Code	$\begin{array}{l} \text{if } (x < y) \\ \quad \text{then } a \leftarrow c + d \\ \quad \text{else } a \leftarrow e + f \end{array}$	$\begin{array}{l} \text{comp\_LT } r_x, r_y \Rightarrow cc_1 \\ \text{cbr\_LT } cc_1 \rightarrow L_1, L_2 \\ L_1: \text{add } r_c, r_d \Rightarrow r_a \\ \text{jumpI } \rightarrow L_{out} \\ L_2: \text{add } r_e, r_f \Rightarrow r_a \\ \text{jumpI } \rightarrow L_{out} \\ L_{out}: \text{nop} \end{array}$	$\begin{array}{l} \text{comp\_LT } r_x, r_y \Rightarrow r_1 \\ \text{cbr } r_1 \rightarrow L_1, L_2 \\ L_1: \text{add } r_c, r_d \Rightarrow r_a \\ \text{jumpI } \rightarrow L_{out} \\ L_2: \text{add } r_e, r_f \Rightarrow r_a \\ \text{jumpI } \rightarrow L_{out} \\ L_{out}: \text{nop} \end{array}$
ILOC Code	$\begin{array}{l} \text{Straight Condition Codes} \\ \text{Conditional Move} \end{array}$	$\begin{array}{l} \text{Boolean Compare} \\ \text{Predicated Execution} \end{array}$	

Source Code	$x \leftarrow a < b \wedge c < d$	$\begin{array}{l} \text{comp } r_a, r_b \Rightarrow cc_1 \\ \text{121\_LT } cc_1, r_t, rf \Rightarrow r_1 \\ \text{comp } r_c, r_d \Rightarrow cc_2 \\ \text{121\_LT } cc_2, r_t, rf \Rightarrow r_2 \\ \text{and } r_1, r_2 \Rightarrow r_x \end{array}$	$\begin{array}{l} \text{comp\_LT } r_a, r_b \Rightarrow cc_1 \\ \text{121\_LT } cc_1, r_t, rf \Rightarrow r_1 \\ \text{comp } r_c, r_d \Rightarrow cc_2 \\ \text{121\_LT } cc_2, r_t, rf \Rightarrow r_2 \\ \text{and } r_1, r_2 \Rightarrow r_x \end{array}$
ILOC Code	$\begin{array}{l} \text{Straight Condition Codes} \\ \text{Conditional Move} \end{array}$	$\begin{array}{l} \text{Boolean Compare} \\ \text{Predicated Execution} \end{array}$	

#### 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 [lower, upper], the offset is calculated by  $(i - \text{low}) \cdot \text{size}_t$

### 2.8.2 Storage Layout

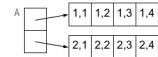
In **row-major order**, the elements of a are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations.

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4
-----	-----	-----	-----	-----	-----	-----	-----

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

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4
-----	-----	-----	-----	-----	-----	-----	-----

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*, and *high*, denote *low* and *high* of the *i*-th dimension, respectively
  - *len<sub>i</sub>* = *high<sub>i</sub>* - *low<sub>i</sub>* + 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 simplified to

$$@Base + (i \cdot len_2 + j) \cdot w$$

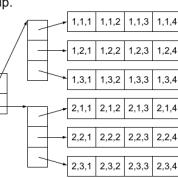
### Indirection Vectors

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

$B[1\dots 2, 1\dots 3, 1\dots 4]$

1. use  $@B_0$ ,  $i$ , 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  $B[i, j, k]$



### 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      => rgb    ...load address of b
loadAI   rgb, -4 => r1     ...load b's length
loadI    @a      => rga    ...load address of a
loadAI   rga, -4 => r2     ...load a's length
cmp_LT   r1, r2  => r3    ...does b fit into a?
cbr      r3      => L0, 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 rgb, r4  => r6    ...load character from b
cstoreAO r6, r2  => rga, r4 ...store character in a
addI    r6, 1    => r7    ...increment counter
cmp_LT   r7, r1  => r8    ...more to copy?
cbr      r8      => L2, L3 ...jump
L3: storeAI r1    => rgb, -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')

```

```

L1: loadI    @b      => rgb    ...load address of b
loadI    @a      => rga    ...load address of a
loadI    NULL    => r1     ...load terminator
L1: cloadI   rgb    => r2    ...load character
cstoreI  r2    => rga    ...store character
addI    rgb, 1    => rgb    ...move pointer
addI    rga, 1    => rga    ...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`, `lod`, `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, %rl          # 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	1 <sup>st</sup> 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

```

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

```

```

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

```

```

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

```

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

one.inode or one.fnnode 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 `statement1` or `statement2`, accordingly
- implements the two statements and at the end jumps to `statement3`

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

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.

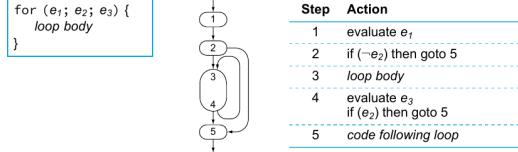
In order to support **different** threads doing **different** things, CUDA has predicated instructions that are executed only if a logical flag is **true**.

```
if (x < 0.0)          p = (x < 0.0);
r = -1.0;             p: r = -1.0;
else                  !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.



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 $\Rightarrow r_1$ ...Step 1 loadI 100 $\Rightarrow r_1$ ...Step 2 cmp_GT r <sub>1</sub> , r <sub>1</sub> $\Rightarrow r_2$ cbr r <sub>2</sub> $\rightarrow L_1, L_2$ L <sub>1</sub> : loop body ...Step 3 addI r <sub>1</sub> , 1 $\Rightarrow r_1$ ...Step 4 cmp_LE r <sub>1</sub> , r <sub>1</sub> $\Rightarrow r_3$ cbr r <sub>3</sub> $\rightarrow L_1, L_2$ L <sub>2</sub> : 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 r <sub>x</sub> , r <sub>y</sub> $\Rightarrow r_1$ ...Step 2 cbr r <sub>1</sub> $\rightarrow L_1, L_2$ L <sub>1</sub> : loop body ...Step 3 cmp_LT r <sub>x</sub> , r <sub>y</sub> $\Rightarrow r_2$ ...Step 4 cbr r <sub>2</sub> $\rightarrow L_1, L_2$ L <sub>2</sub> : 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	L <sub>1</sub> : loop body ...Step 3 cmp_LT r <sub>x</sub> , r <sub>y</sub> $\Rightarrow r_1$ ...Step 4 cbr r <sub>1</sub> $\rightarrow L_1, L_2$ L <sub>2</sub> : 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;
}
```

The compiler should order cases according to estimated execution frequency.

### Binary Search

switch(e) { case 0: block <sub>0</sub> ; break; case 15: block <sub>15</sub> ; break; case 23: block <sub>23</sub> ; break; ... case 99: block <sub>99</sub> ; break; default: block <sub>d</sub> ; break; }	t <sub>1</sub> $\leftarrow e$ down $\leftarrow 0$ ...lower bound up $\leftarrow 10$ ...upper bound + 1 while (down + 1 < up) { middle $\leftarrow (up + down) \div 2$ if (Value[middle] $\leq t_1$ ) then down $\leftarrow$ middle else up $\leftarrow$ middle }	Value   Label 0   LB <sub>0</sub> 15   LB <sub>15</sub> 23   LB <sub>23</sub> 37   LB <sub>37</sub> 41   LB <sub>41</sub> 50   LB <sub>50</sub> 68   LB <sub>68</sub> 72   LB <sub>72</sub> 83   LB <sub>83</sub> 99   LB <sub>99</sub>
---	---	---

### Computing the Address Directly

switch(e) { case 0: block <sub>0</sub> ; break; case 1: block <sub>1</sub> ; break; case 2: block <sub>2</sub> ; break; ... case 9: block <sub>9</sub> ; break; default: block <sub>d</sub> ; break; }	t <sub>1</sub> $\leftarrow e$ if (t <sub>1</sub> < 0 or t <sub>1</sub> > 9) then jump to LB <sub>d</sub> else t <sub>2</sub> $\leftarrow @Table + t_1 \times 4$ t <sub>3</sub> $\leftarrow$ memory(t <sub>2</sub> ) jump to t <sub>3</sub>	Label LB <sub>0</sub> LB <sub>1</sub> LB <sub>2</sub> LB <sub>3</sub> LB <sub>4</sub> LB <sub>5</sub> LB <sub>6</sub> LB <sub>7</sub> LB <sub>8</sub> LB <sub>9</sub>
---	--	---

## 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.

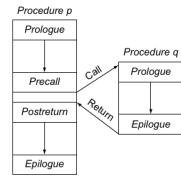
### Evaluating Actual Parameters

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

### 3.1 Instruction Selection

#### 3.1.1 Tree Pattern-Matching

#### 3.1.2 Peephole Optimization

### 3.2 Instruction Scheduling

#### 3.2.1 Local List Scheduling

#### 3.2.2 Regional List Scheduling

### 3.3 Register Allocation

#### 3.3.1 Local Register Allocation

Top-Down

Bottom-Up

#### 3.3.2 Global Register Allocation

Top-Down

Bottom-Up