Compiladores

@fudef

L.EIC
#3.2

Cheatsheet for MT1

Context-free grammar

1. Ambiguity

For a single sentential form, has two or more leftmost derivations, or two or more rightmost derivations

- **How to remove it?:** Change the grammar

Ex// IF-then-else problem

S → if E then S
| if E then S else S
| ε // other stmts

S → withElse | NoElse
WithElse → if E then WithElse else WithElse
| ε // other stmts
NoElse → if E then S
| if E then WithElse else NoElse

Left-recursivity

Incompatible with top-down parsers

- **Convert to right recursion:** Ex// $F \rightarrow F\alpha \mid \beta$

$F \rightarrow \beta T$
$T \rightarrow \alpha T$
$\mid \varepsilon$

Top-Down / LL Parsing

2. FIRST()

Set of first symbols in some string that derives from α

$x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x\gamma$ for some γ

FOLLOW()

Set of all symbols that can appear immediately after α

- $\$ \in \text{FOLLOW}(\langle\text{start symbol}\rangle)$
- IF $A \rightarrow BCD$, then $(\text{FIRST}(D) / \varepsilon) \in \text{FOLLOW}(C)$
- IF $A \rightarrow BC$ or $A \rightarrow BCD$ and $\varepsilon \in \text{FIRST}(D)$, then $\text{FOLLOW}(A) \in \text{FOLLOW}(C)$

LL(1) Property

The parser can make the correct choice of production with a lookahead of exactly one symbol

- **Check:**

Ex// $A \rightarrow \alpha \mid \beta$

$\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset$

$$\text{FIRST}^+(\alpha) = \begin{cases} \text{FIRST}(\alpha) \cup \text{FOLLOW}(A), & \varepsilon \in \text{FIRST}(\alpha) \\ \text{FIRST}(\alpha), & \text{otherwise} \end{cases}$$

- **Left Factoring**

Ex// $A \rightarrow \alpha B$
$\mid \alpha C$

$A \rightarrow \alpha Z$
$Z \rightarrow B$
$\mid C$

LL(1) Table

| Used later to select the correct production in the top-down parser.
| If any entry is defined multiple times, the grammar is not LL(1).

- **Filling entry** $M[X,y]$
  - $X \to \beta$ if $y \in \text{FIRST}(\beta)$
  - $X \to \epsilon$ if $y \in \text{Follow}(X)$ and $X \to \epsilon \in$ grammar
  - error otherwise

Ex//

$S \to aT$
$\quad | \; e$
$T \to bS$
$\quad | \; S$

	a	b	e
S	$S \to aT$		$S \to e$
T	$T \to S$	$T \to bS$	$T \to S$

FIRST (S) = {a,e} Follow (S) = {$} NULLABLE = {}
FIRST (T) = {a,b,e} Follow (T) = {$}

3. Bottom-up / LR Parsing

Actions

- **Shift**: Add symbol / element to the stack
- **Reduce**: Pop elements, apply a production and push LHS
- **Accept**: Report success (EoF reached and stack only has start symbol)
- **Reject**: Report failure (EoF reached but stack has more elements)

Ex//

$S \to X\$$ (1)
$X \to (X)$ (2)
$\quad | \; ()$ (3)

Input:
(())$

State	(	)	$	X
0	shift 2	×	×	go to 1
1	×	×	accept	
2	shift 2	shift 5	×	go to 3
3	×	shift 4	×	
4	reduce (2)	reduce (2)	reduce (2)	
5	reduce (3)	reduce (3)	reduce (3)	

0			
0	2		
0	2	2	
0	2	2	5
0	2	3	
0	2	3	4
0	1		

$				shift 2
$	(			shift 2
$	(	(		shift 5
$	(	(	)	reduce (3)
$	(	X		shift 4
$	(	X	)	reduce (2)
$	X			accept ✓

4. Syntax-Directed Translation

Attributes

- **Synthesized**
  - Only uses values from children, self and constants;
  - S-attributed grammars;
  - Good for LR parsing.
- **Inherited**
  - Also uses values from parents and siblings;
  - More natural, but harder to parse.

Ex//

```
S → El * Er
{
    S.type = match (El.type, Er.type) {
        (a, b) if a == b => a
        (int, float) | (float, int) => float   // *
        (_, _) => undefined   // Error!
    }
}
```

- The "type" attribute is synthesized.

* "int" type can be overloaded into a "float" type

Symbol Table of a scope

Includes (usually):

- scope
- kind
- symbol
- type
- size
- parent ref.

Ex//

```
int a;

int fn (int a) {
    int b = 3.2;
    return a * 3;
}
```

scope : file		parent : •	
kind	symbol	type	size
local /var	a	int	4

scope : fn		parent : •	
kind	symbol	type	size
param	a	int	4
local /var	b	double	8

Three - Address code generation

```
x = y [i]        ⟶     tmp = i * 8      // 8 = sizeof (<type>)
                        x = y [tmp]

x = fn (y, z)    ⟶     tmp1 = y
                        tmp2 = z
                        putparam tmp1
                        putparam tmp2
                        x = call fn , 2    // 2 = # arguments

int fn (y, z) {  ⟶     fn:  getparam z      // Reverse order!
    return y + z;            getparam y
}                            tmp = y + z
                             return tmp

while (a < b) {  ⟶            if (a ≥ b) goto next
    if (a > 10) {break;}  loop: if (a > 10) goto next
    a = a + 1                   a = a + 1
}                               if (a < b) goto loop
                          next: ...
```

Compiladores

@fudef

L.EIC

#3.2

Cheatsheet for MT2

1. Intermediate Code Generation

Arithmetic

```
i = (10 + 20) * 30;
        ↓
t1 = 10 + 20
i = t1 * 30
```

Structs

```
typedef struct { offset
    int i;        +0
    int j;        +4
} coord;
coord *e;
x = e -> j;
e -> i = y;
        ↓
t1 = e + 4;
x = *t1;
*e = y;
```

Boolean

```
b = 10 < 20;
        ↓
    if (10 < 20) goto L1
L0: b = 0
    goto END
L1: b = 1
END: ...
```

Function Decl.

```
int fn (int x, int y) {
    return x + y
}
        ↓
getparam y  } ⚠ In reverse!
getparam x  }
t1 = x + y
return t1
```

Short-circuit

```
b = w < x && y < z;
        ↓
    if w < x goto L2
    goto L0
L2: if y < z goto L1
    goto L0
L1: b = 1
    goto END
L0: b = 0
END: ...
```

Array Index

```
i = a[idx];
        ↓   ⚠ 32 bit reference!
t1 = idx x 4
t1 = a + t1
i = *t1
```

while

```
i = 0;
while (i < 10) {i++;}
        ↓
    i = 0
WL: if (i < 10) goto L1
    goto END
L1: i = i + 1
    goto WL
END: ...
```

Function Call

```
i = fn (x, y);
        ↓
t1 = x
t2 = y
putparam t1  } 2
putparam t2  } args
i = call fn, 2
```

2D Array Access

$A[i_1, i_2]$

- Row-major : $baseA + [(i_1 - low_1) * (high_2 - low_2 + 1) + i_2 - low_2] * sizeof(Abasetype)$
- Column-major : $baseA + [(i_2 - low_2) * (high_1 - low_1 + 1) + i_1 - low_1] * sizeof(Abasetype)$
- Indirection vectors : $*(A[i_1])[i_2]$

2. Runtime Environments

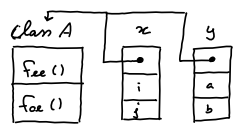
parameters
register save area
return value
return address
access link
Caller's ARP ←
local variables (variable size)

Allocation

- Static : Procedure makes no calls
- Heap : Procedure can outlive its caller or can return an object that can reference its execution state
- Stack : AR and invocation lifetimes match and procedure executes a return.

Class Structure

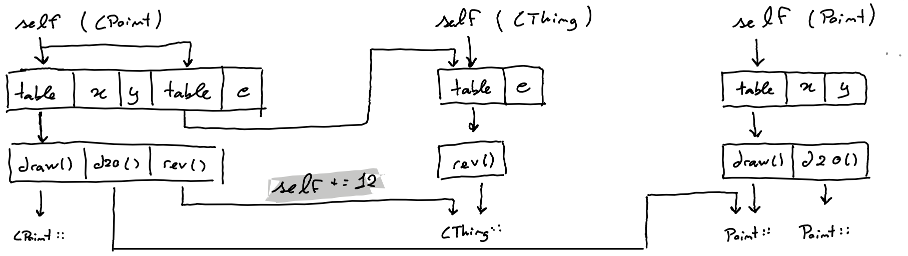
Access to object data is analogous to struct fields



3. Register Allocation

Inheritance

Ex// class Point { int x, y; void draw(); void d2o(); }
class CThing { Color c; void rev(); }
class CPoint extends Point, CThing { void draw(); }



Importance

- Use registers as much as possible / avoid loads and stores
  - ↳ Faster accesses compared to memory
  - ↳ Less number of instructions

Top-Down

Estimate the overall benefit of each variable and assign the highest-payoff ones to registers.

$$\text{TotBenefit}(V) = \sum_{B} \text{Benefit}(V, B) \times \text{freq}(B)$$

- $V \rightarrow$ variable
- $B \rightarrow$ basic block

↳ $\text{Benefit}(V, B)$ = Number of uses and defs of V in B

↳ $\text{freq}(B) = 10^{\#\text{loops or depth}}$

Bottom-Up

Allocate variables to unused registers and release register whose value is to be used the farthest into the future when out of registers.

Ex// Size = 3

$t_3 \leftarrow t_1 \text{ op } t_2$

$t_5 \leftarrow t_4 \text{ op } t_1$

$t_6 \leftarrow t_3 \text{ op } t_5$

mem → r_0
mem → r_1
$r_2 \leftarrow r_0 \text{ op } r_1$

	r_0	r_1	r_2
Name	t_1	t_2	t_3
Next	1	∞	2
Free	F	F	F

mem ← r_1
mem ← r_2

	r_0	r_1	r_2
Name	t_1	-	-
Next	1	-	-
Free	F	T	T

mem → r_1
$r_2 \leftarrow r_0 \text{ op } r_1$

	r_0	r_1	r_2
Name	t_1	t_4	t_5
Next	∞	∞	1
Free	F	F	F

Web

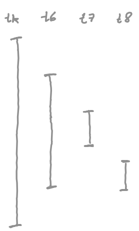
DU chains:
- All defs that reach a use are in the same web
- All uses of a def are in the same web

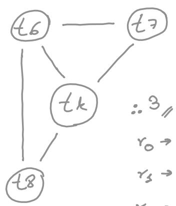
- **Interference** if two web ranges overlap in time

Global RA

1. Draw an interference graph (webs are nodes and interferences are edges)
2. Find min. #registers with **graph coloring**

Ex//
```
tk = $sp + off_k
t6 = $sp + off_6
t7 = * t6
t8 = 1 + t7
*t6 = t8
tk = tk + 1
```





∴ 3//
$r_0 \rightarrow tk$
$r_1 \rightarrow t6$
$r_2 \rightarrow t7, t8$

4. Control - Flow Analysis

Constant Propagation

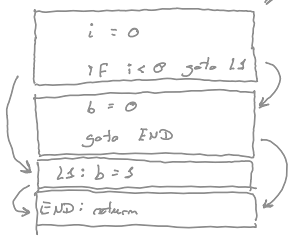
- Find an RHS exp. that is a constant.
- Replace the variable with the constant until redefinition.

Ex// $y = 0;\ x = 3 + y; \longrightarrow y = 0;\ x = 3 + 0$

Basic Block

A maximal sequence of instructions that start with the "leader" instruction

Ex//



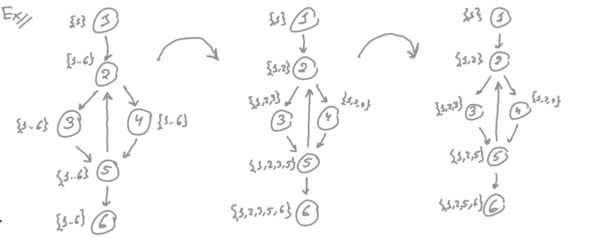
Leader detection:

- The first statement;
- Any statement that is the target of a goto;
- Any statement that immediately follows a goto.

Dominators

x dominates y if every execution path from entry to y includes x.

Ex//



Iterative algorithm:

- Start with the entry node dominating itself and every node dominating the remaining;
- Visit nodes in any order
  - ↳ Dominator set of the node is the intersection of all predecessors + current node
- Repeat until no change

5. Data - Flow Analysis

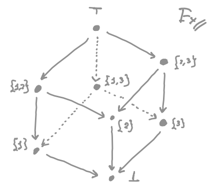
Definition

A collection of techniques for compile-time reasoning about the runtime flow of values in a program.

Lattice

- V | Set of values $= \{1,2,3\}$
- $\top$ | Top value $= \{1,2,3\}$
- $\bot$ | Bottom value $= \{\ \}$
- $\wedge$ | Meet operation (greatest lower bound) $\{1,2\} \wedge \{2,3\} = \{2\}$
- $\vee$ | Join operation (lowest upper bound) $\{1,2\} \vee \{3\} = \{1,2,3\}$

Ex//



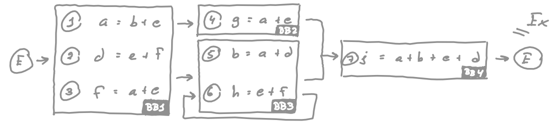
Iterative Algorithm

(*) Transfer function properties:
- Commutativity
- Associativity

- CFG with "Entry" and "Exit" nodes
- Direction of the data-flow: forward or backward
- Set of values V • Meet operator $\wedge$
- Transfer functions* for each block
- Constant boundary value: v_{entry} or v_{exit}

⟶ IN and OUT sets for each block with values of V

Available Expressions



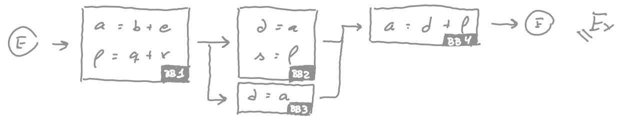
- V = Set of expressions = {1..7}
- Data flows forward (In → out)
- Functions:
  - ↳ Out = gen ∪ (In − kill)
  - ↳ gen = expressions calculated in the BB
  - ↳ kill = expressions that use the var.
- ∩ = In = ∩ out
- In_{init} = { }

	BB1	BB2	BB3	BB4
Gen	1,3	4	5,6	7
kill	2,3,4,5,6,7	∅	1,7	∅
In	∅	1,3	3	3
Out	1,3	1,3,4	3,5,6	3,7

∴ $\boxed{ae}$ {} ① {1} ② {1,2} ③ → {1,3} ④ {1,3,4} → {3} ⑦ {3,7}

↳ {3} ⑤ {3,5} ⑥ {3,5,6}

Copy Propagation



- V = set of tuples $\langle v,u \rangle$ if "v = u" stmt. exists
- Data flows forwards

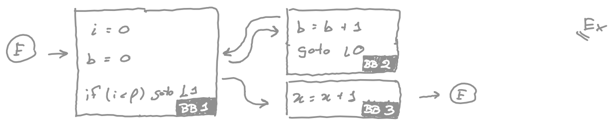
$V = \{\langle d,a \rangle, \langle s,p \rangle\}$

- Functions:
  - ↳ Out = gen ∪ (In − kill)
  - ↳ Gen = {$\langle v,u \rangle$ | "v = u" in a stmt.}
  - ↳ Kill = {$\langle v,u \rangle$ | LHS assign. stmt. is either v or u}
- ∩ = In = ∩ out
- V_{init} = { }

	BB1	BB2	BB3	BB4
Gen	∅	$\langle d,a \rangle, \langle s,p \rangle$	$\langle d,a \rangle$	∅
kill	$\langle d,a \rangle, \langle s,p \rangle$	∅	∅	$\langle d,a \rangle$
In	∅	∅	∅	$\langle d,a \rangle$
Out	∅	$\langle d,a \rangle, \langle s,p \rangle$	$\langle d,a \rangle$	∅

∴ a = d + p → a = a + p

Live - Variable Analysis



- V = set of variables
- Data flows backwards (In → out)
- Functions:
  - ↳ In = gen ∪ (Out − kill)
  - ↳ gen = variables <u>used</u> in a BB
  - ↳ kill = variables <u>defined</u> in a BB
- ∩ = Out = ∪ In(successors)
- v_{init} = { }

		BB1	BB2	BB3
(use)	Gen	i, p	b	x
(def)	kill	i, b	b	x
	In	i, p, x	b, i, p, x	x
∴	Out	b, i, p, x	i, p, x	∅

Loop Invariant Code Motion

for i = 1 to N
  a += i * 2 * N

↪ tmp = 2 * N
for i = 1 to N {a += i * tmp}

Induction Variable Recognition

for i = 1 to N
  a[i] = 8

↪ t1 = @a(i,1)
for i = 1 to N {*t1 = 8; t1 += 8}