COMP50006 Compilers
Imperial College London

Boxuan Tang

Spring 2023

Contents

1 Lexer **2**
- 1.1 Regex $\rightarrow$ NFA (Thompson's Construction) . . . 2
- 1.2 NFA $\rightarrow$ DFA (Subset Construction) . . . 2

2 LR Bottom-Up/Shift-Reduce Parser **2**
- 2.1 LR Model . . . 2
- 2.2 LR(0) . . . 2
  - 2.2.1 LR(0) $\rightarrow$ NFA . . . 2
  - 2.2.2 DFA $\rightarrow$ LR(0) Parsing Table . . . 3
- 2.3 LR(1) . . . 3
  - 2.3.1 FIRST and FOLLOW . . . 3
  - 2.3.2 LR(1) $\rightarrow$ NFA . . . 3
  - 2.3.3 NFA $\rightarrow$ LR(1) Parsing Table . . . 3
- 2.4 LALR(1) . . . 3
- 2.5 Conflicts . . . 3
  - 2.5.1 Shift-Reduce Conflict . . . 3
  - 2.5.2 Reduce-Reduce Conflict . . . 3

3 LL Top-Down Parsing **3**
- 3.1 LL(1) . . . 3
  - 3.1.1 CFG $\rightarrow$ LL(1) . . . 4
  - 3.1.2 LL(1) to Parse Function . . . 4

4 Garbage Collection **4**

5 Register Allocation **4**
- 5.1 Register Machine Strategy . . . 4
- 5.2 Accumulator Strategy . . . 4
- 5.3 Sethi-Ullman Weights . . . 4
- 5.4 Graph Colouring . . . 4

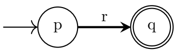
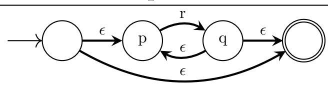
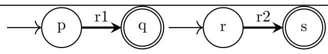
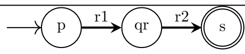
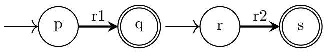
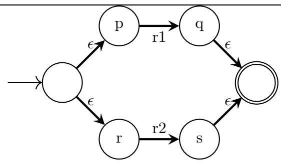
6 Optimisation **5**
- 6.1 Live Variables . . . 5
- 6.2 Reaching Definitions . . . 5
- 6.3 Loops . . . 5

1 Lexer

Definition 1.1 (Context Free Grammar) *Contains a non-terminal start symbol, a set of productions, a set of terminals (tokens) and a set of non-terminals (S,P,t,nt).*

- *A **sentence** is a derived string comprising only terminals.*
- *The **language** is the set of all sentences derived from the start symbol.*
- *It is **ambiguous** if the language contains strings that can be generated from two different ways*

1.1 Regex → NFA (Thompson's Construction)

Regex	Given	Replace
(1) r*		
(2) r1 r2		
(3) r1 | r2		

1.2 NFA → DFA (Subset Construction)

DFA start state = ϵ-Closure(NFA start state)
foreach new subset state S of the DFA:
    foreach unique symbol r leading out from any state of S:
        add a transition r from S to S' where S' = ϵ-Closure(states reached by r in 1 step)
Mark subset states accepting if any member state accepting in NFA

2 LR Bottom-Up/Shift-Reduce Parser

For LR, we add on an auxiliary rule with end-of-input symbol \$. For example, E' → E \$

2.1 LR Model

Parsing table contains all terminals under ACTIONS and non-terminals under GOTO
Push state 0 (start state) onto stack and then repeatedly perform

- **shift sN**: push state n onto stack, advance current token
- **goto gN**: not selected directly (reduce)
- **accept a**: accept input
- **reduce rN**: remove L elements from stack where L = length rhs of rule N
  push Table[stack.top(), LHS of rule N]

2.2 LR(0)

Definition 2.1 (LR(0) items) *are instances of the grammar rules with a • on the rhs of the rule*

2.2.1 LR(0) → NFA

Given a state with item $X \to A \bullet BC$, add $\boxed{X \to A \bullet BC} \xrightarrow{B} \boxed{X \to AB \bullet C}$

And if B is non-terminal, foreach initial item $B \to \bullet D$, add $\boxed{X \to A \bullet BC} \xrightarrow{\epsilon} \boxed{B \to \bullet D}$

2.2.2 DFA → LR(0) Parsing Table

- state X with terminal transition $X \xrightarrow{t} Y$, add P[X,t] = sY
- state X with non-terminal transition $X \xrightarrow{N} Y$, add P[X,N] = gY
- state containing item $R' \to ..\bullet$, add P[X,$] = a
- state containing item $R \to ..\bullet$, add P[X,t] = rN for all terminals t where N is R's rule number

2.3 LR(1)

Definition 2.2 (LR(1) items) *is a pair [LR(0) item, look-ahead token t]*

2.3.1 FIRST and FOLLOW

Definition 2.3 (FIRST) *FIRST(α) is the set of all terminals that could start the derivation of α*

FIRST(ϵ) = {ϵ}, FIRST(a) = {a}
FIRST(A) = foreach $A \to \beta_1\beta_2...\beta_n$
include FIRST(β_1) - {ϵ} in FIRST(A)
if ϵ in FIRST(β_1): include FIRST(β_2) - {ϵ} in FIRST(A) ...
... if ϵ in FIRST(β_n): include ϵ in FIRST(A)

Definition 2.4 (FOLLOW) *FOLLOW(A) is the set of all terminals that follows non-terminal A*

foreach $B \to CAD$
include FIRST(D) - {ϵ} in FOLLOW(A)
if ϵ in FIRST(D): include FOLLOW(B) in FOLLOW(A)
if A ends the input include $ in FOLLOW(A)

2.3.2 LR(1) → NFA

Add initial item $[R' \to \bullet R, \$]$

Given a state with item $[X \to A \bullet BC, t]$, add $\boxed{[X \to A \bullet BC, t]} \xrightarrow{\text{B}} \boxed{[X \to AB \bullet C, t]}$
And if B is non-terminal, foreach rule $B \to \bullet D$, foreach token u in FIRST(Ct),
add $\boxed{[X \to A \bullet BC, t]} \xrightarrow{\epsilon} \boxed{[B \to \bullet D, u]}$

2.3.3 NFA → LR(1) Parsing Table

Like LR(0) but for states containing $[X \to A\bullet, t]$, only add reduction for column t

2.4 LALR(1)

If any 2 LR(1) states have the same LR(0) items, combines the states

2.5 Conflicts

2.5.1 Shift-Reduce Conflict

S → if E then S | if E then S else S | other *becomes*
S → MS | UMS
MS → if E then MS else MS | other
UMS → if E then S | if E then UMS else UMS

2.5.2 Reduce-Reduce Conflict

Expr → Expr + Expr | Expr * Expr | (Expr) | int *becomes*
Expr → Expr + Term | Term
Term → Term * Factor | Factor
Factor → (Expr) | int

3 LL Top-Down Parsing

3.1 LL(1)

Definition 3.1 (LL(1) Grammar) *For all rules $A \to \alpha \mid \beta$, FIRST(α) and FIRST(β) are disjoint. If FIRST(α) contains ϵ then FIRST(β) and FOLLOW(α) are disjoint. Vice-versa.*

Definition 3.2 (Extended BNF) *contains {α} for 0 or more occurrences and [α] for 0 or 1 occurrences*

3.1.1 CFG → LL(1)

- **Left Factorisation** $\underline{A \to BC \mid BD} \longrightarrow \underline{A \to B(C \mid D)}$ and $\underline{A \to BC \mid B} \longrightarrow \underline{A \to B[C]}$
- **Substitution** replaces a rule with its alternatives to make conflicts direct
- **Left Recursion Removal** $\underline{A \to X_1 \mid ... \mid X_n \mid AY_1 \mid ... \mid AY_n} \longrightarrow \underline{A \to (X_1 \mid ...X_n)\{Y_1 \mid ... \mid Y_n\}}$
  more specifically $\underline{A \to X \mid AY} \longrightarrow \underline{A \to X\{Y\}}$

3.1.2 LL(1) to Parse Function

- **A B**: A(); B();
- **A**|**B**: if token in FIRST(A) then A(); elif token in FIRST(B) then B(); else error();
- **{A}**: while token in FIRST(A): A();
- [**A**]: if token in FIRST(A): A();

4 Garbage Collection

Heap Compaction: Mark live blocks → Relocate live blocks → Update pointers to relocated blocks

- **Reference-Counting**: Requires special techniques for cyclic data structures
- **Mark-Sweep**: **Mark**: Mark all blocks reachable from non-heap references as live
  Sweep: Scan all blocks to reclaim dead blocks, unmark live blocks for next sweep
  Use pointer reversal technique to visit all nodes of a Directed Graph without additional stack space
  Provides the largest possible block available when combined with compaction
- **Two-Space**: Allocate blocks to **From-Space**. When it's exhausted, copy live blocks to **To-Space**
  Automatic compaction, very fast to allocate objects, wastes half of memory, relocate long-lived objects
- **Generational**: Heap divided areas based on block age. Perform GC on younger generations more

5 Register Allocation

```
transExp (Binop Minus e1 e2) r
```

5.1 Register Machine Strategy

```
transExp e1 r ++
transExp e2 (r+1) ++
[Sub r (r+1)]
```

5.2 Accumulator Strategy

```
transExp e2 r ++
[Push r] ++
transExp e1 r ++
[SubStack r]
```

5.3 Sethi-Ullman Weights

```
if weight e2 > weight e1 then
  transExp (nxtreg:dstreg:regs) ++
  transExp (dstreg:regs) ++
  [Sub dstreg nxtreg]
```

```
weight (Binop Minus e1 e2) = min [
  max [weight e1, (weight e2) + 1],
  max [weight e2, (weight e1) + 1]
]
```

5.4 Graph Colouring

1. Generate intermediate three-address code where values are always saved in named locations
2. Construct inference graph where nodes are locations and nodes are linked if their live ranges overlap
3. Try to colour the nodes so no connected nodes have the same colour

6 Optimisation

6.1 Live Variables

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

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

```
foreach n in CFG:
  LiveIn(n) := {}; LiveOut(n) := {}
while true: foreach n in CFG (backwards):
  LiveIn(n) = uses(n)+(LiveOut(n)-defs(n))
  LiveOut(n) = union [LiveIn(s)|s<-succ n]
```

6.2 Reaching Definitions

$$ReachIn(n) = \bigcup_{s \in pred(n)} ReachIn(s)$$

$$ReachOut(n) = Gen(n) \cup (ReachIn(n) - Kill(n))$$

For $n : t = u1 \oplus u2$ (defs(n)={t}, uses(n)={u1,u2})

- Gen(n) is definitions generated by n, {n}
- Kill(n) is all definitions of t except n

6.3 Loops

Definition 6.1 (Loop) *A set of nodes S including a header node h such that*

- *Any node in S has a path leading to h*
- *There is a path from h to any node in S*
- *There is no edge from any node outside S to any node in S other than h*

Definition 6.2 (Dominator) *A node d dominates node n if every path from the start node to n must go through d. Every node dominates itself.*

$$Doms(s) = \{s\}, \text{ for start node s}$$

$$Doms(n) = \{n\} \cup (\bigcap_{p \in preds(n)} Doms(p)), \text{ otherwise}$$

Definition 6.3 (Back Edge) *An edge from node n to node h that dominates n is called a back edge. For every back edge, there is a loop. Two loops can share the same header.*

Definition 6.4 (Loop Invariants) *For hoisting a node d: $t = a \oplus b$,*

- ***Reaching Definitions**: All reaching defs used by d occur outside loop*
- ***Dominators**: d dominates all loop exits*
- ***Count**: There must only be one def of t in the loop*
- ***Live Variables**: t must not be LiveOut from the loop's preheader*

Definition 6.5 (Single State Assignment) *Introduces a new name each time a variable is assigned.*
At control-flow joins, insert a dummy operator $t = \phi(t_1, t_2)$ which magically picks either value depending on what path is taken.
In the generated code, we push the assignments of t backwards into the two predecessor paths.
*To hoist a node after SSA conversion, only need to check **Reaching definitions**.*