Compilers

Chapter 1

Lexical & Syntax Analysis

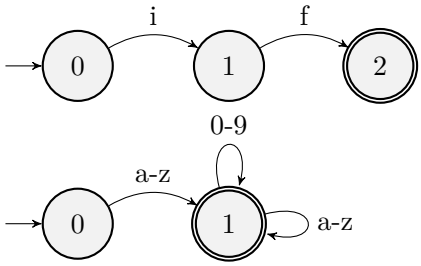
1.1 Finite Automata

Finite Automata are formal models for describing algorithms.

Technically each non-accepting state should have a transition for every symbol, which represents errors. But these are omitted from diagrams.

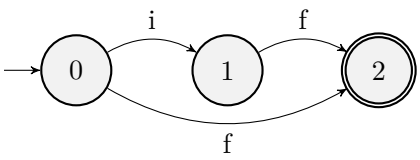
1.1.1 Deterministic Finite Automata

No two transitions leaving a state have the same symbol. (Informally, no backtrace needed)



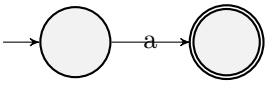
1.1.2 Non-Deterministic Finite Automata

Allow a choice of transitions out of a state. (Informally, backtrace allowed)

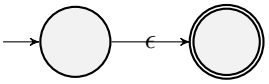


1.1.3 Transforming From RegExp to NFA: Thompson's Construction

Symbol

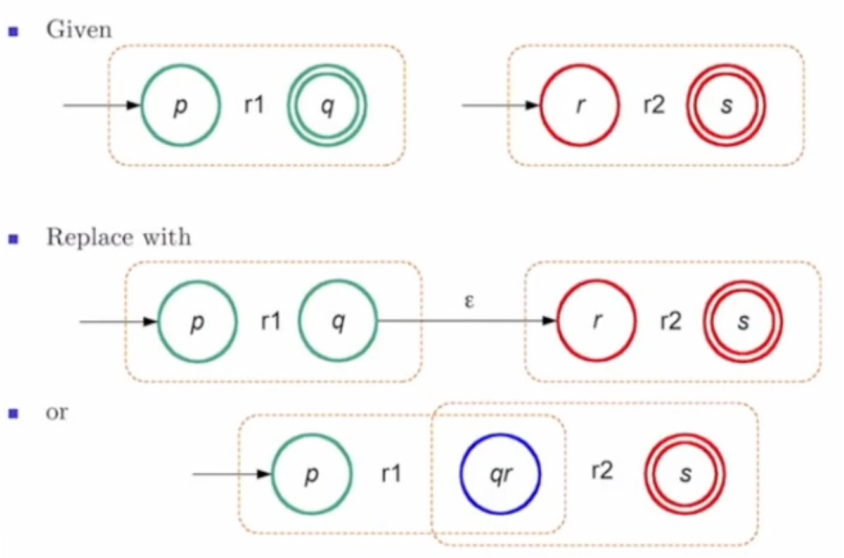


Epsilon

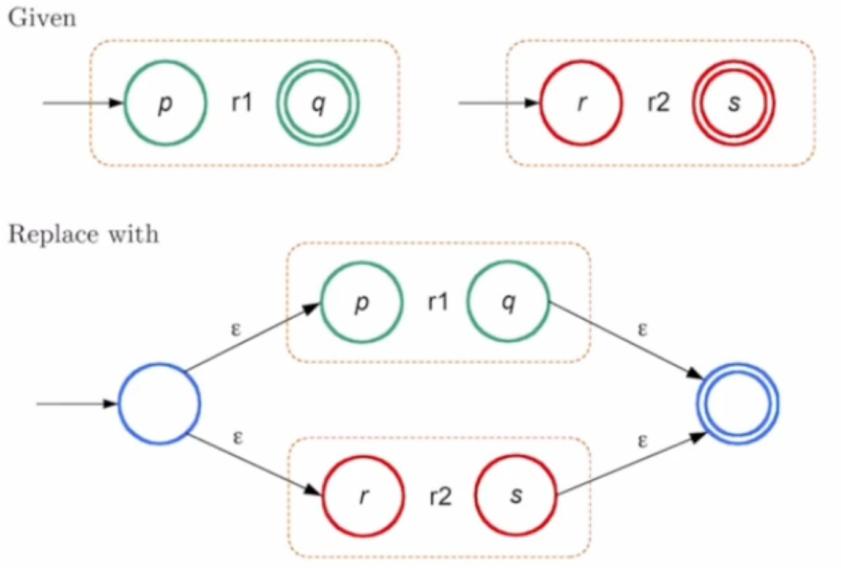


Grouping

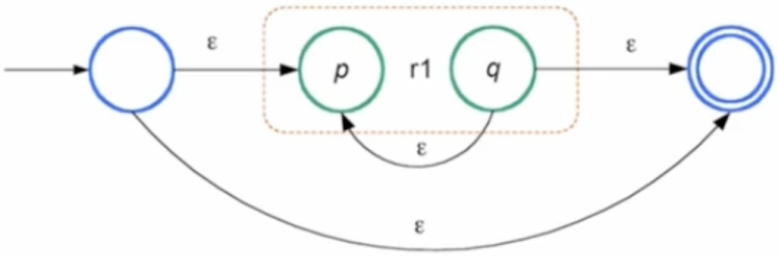
Concatenation: $r_1\ r_2$



Alternation: $r_1 \mid r_2$



Repetition: r^*



1.1.4 Transforming From NFA to DFA: Subset Construction

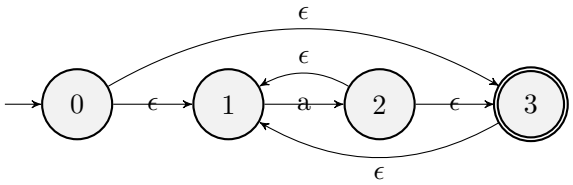
Start with initial node, in each turn, assume the node we are processing is A:

- Add ϵ-closure targets to form a combined node group, in which there is a ϵ-transformation from the node to other nodes. For ϵ-transformations $A \to B; A \to C; B \to D$: create a new node group (A, B, C, D) if it does not exist, and work on that combined group for the following steps.
- For each node in the node group:
  - find and **combine one-step** targets of **each** its outgoing path. For $A \xrightarrow{g} B; A \xrightarrow{g} C; A \xrightarrow{h} D; D \xrightarrow{\epsilon} E$:
    - A can go along g to B or C, and B, C has no outgoing ϵ paths, so add $A \xrightarrow{g} (B, C)$. (B, C) must be treated as a whole, so you cannot add $A \xrightarrow{g} (B, C, E)$, nor $A \xrightarrow{g} (B)$.
    - A can go along h to D, and D goes ϵ to E, so add exactly $A \xrightarrow{h} (D, E)$.

* If (B, C) or (D, E) already exists, just add a path. If not, create a new such node then add a path.
* If you calculated a $(A, C) \xrightarrow{g} (A, C)$, then this is the only case you can add a circle from (A, C) to itself (by g). Neither $(A) \xrightarrow{g} (A, C)$ nor $(A, C) \xrightarrow{g} A$ form a circle, since **combined nodes must be treated as a whole**.

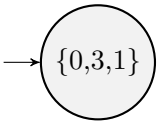
– You should repeat this process for all the outgoing paths for all the nodes in the node group.

The example below shows how to transform the given NFA to DFA. Note this example is from the Lecture and is too simple. Try questions from tutorial sheet for reference. (sample answers)



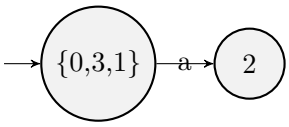
Start with initial node, we aim to create a combined node from its ϵ-transformations.

Node 0 has ϵ transformation to 3, and 3 has ϵ transformation to 1. So merge three nodes:



Iterate path from nodes in the group. The node group $(0, 3, 1)$ contains 3 nodes $0, 3, 1$, so for first node 0:

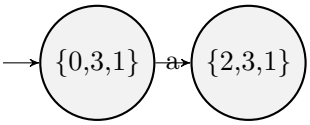
By a, 1 goes to exactly node (2). So $0, 3, 1$ should go to exactly (2) by a:



Node 3 and 0 does not have non-ϵ outgoing paths. So we finished this step.

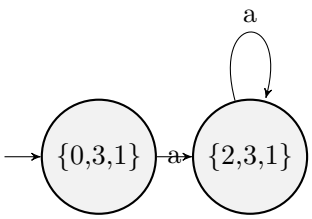
Now we move to the node we just added, the (2). First combine ϵ-clousure:

Node 2 has ϵ transformation to 3, and 3 has ϵ transformation to 1. So add three nodes to create a node group (we are replacing (2) to $(2, 3, 1)$, and this is the only step we can replace a node to node group):

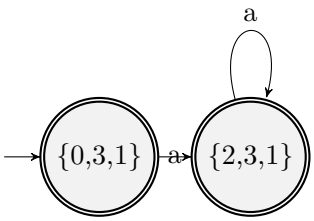


Since 1 goes to exactly node 2 by a.

We want to add path $(2, 3, 1) \xrightarrow{a} (2)$, but $(2) \xrightarrow{\epsilon} (3)$ and $(3) \xrightarrow{\epsilon} (1)$, so we actually should add $(2, 3, 1) \xrightarrow{a} (2, 3, 1)$, which is a circle.



Now consider accepting states. Originally 3 is an accepting state, so all new node groups that contains 3 should also be accepting state:



1.2 LR Parsing

LR (bottom-up) parsing is also known as shift-reduce parsing.

- *bottom-up* reflects the direction of the AST is constructed.
- *shift-reduce* reflects the two main actions performed by an LR parser.

1.2.1 LR(0) Parsers

LR(0) parsers are the simplest LR parsers.
In grammar, a special symbol $ is used to indicate end-of-input.

LR(0) items

An LR(0) item indicates how much of a rule has been seen, e.g. the item $E \to E+\cdot \text{int}$ indicates we have seen an expression and a plus sign, and hope to encounter an integer next.
LR(0) Items are used as states of a finite automation that maintain information about the progress of a shift-reduce parse.
We can build NFA from LR(0) items, then build DFA using subset construction Transforming From NFA to DFA: Subset Construction.

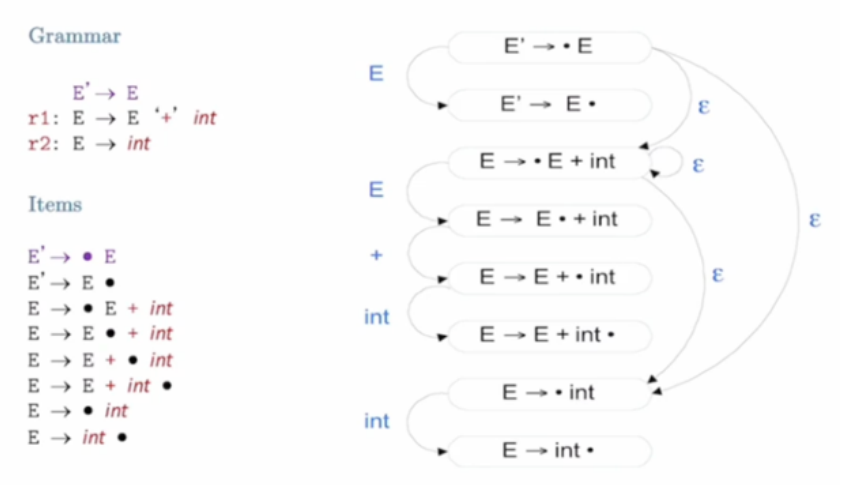


Figure 1.2: Building NFA from LR(0) Items

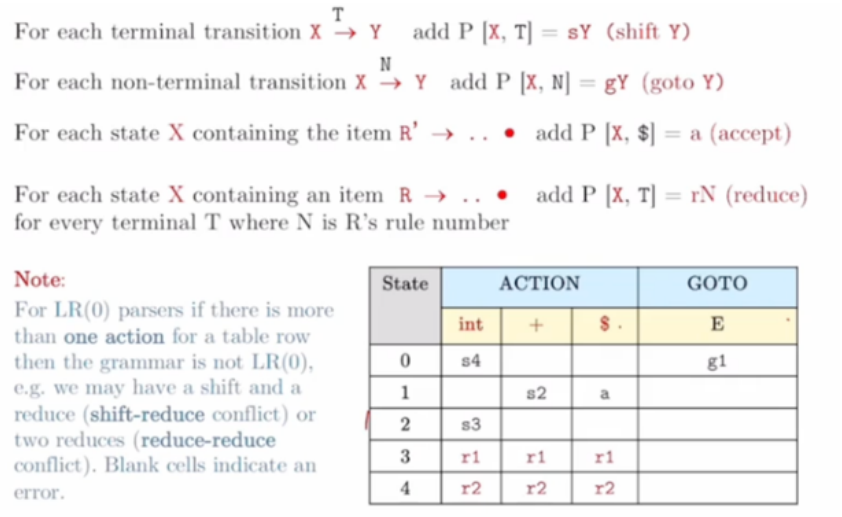


Figure 1.3: DFA to LR(0) Parsing Table

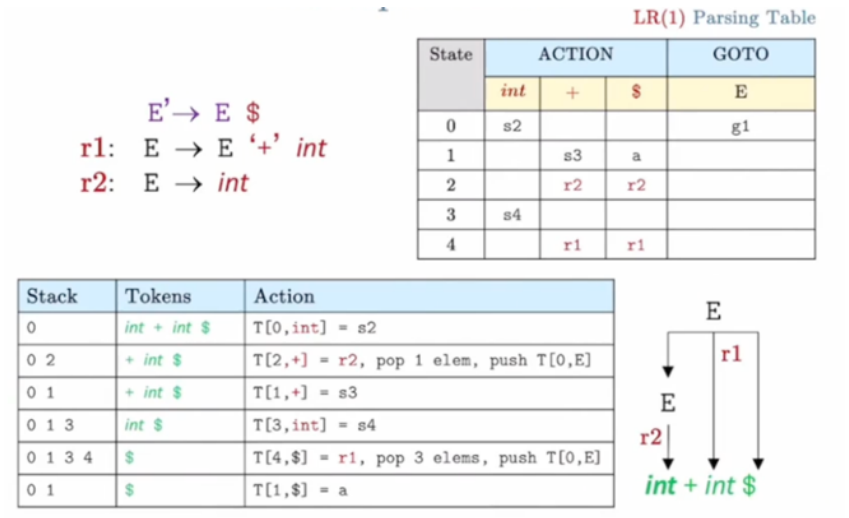


Figure 1.4: LR Parsing Example

1.2.2 DFA to LR(0) Parsing Table

1.2.3 Model of an LR Parser

For a stream of input tokens, LR parser maintains a stack of states.

Initially, push state 0 to the stack. Then repeatedly follow the switch statement by looking up the Parsing Table.

shift Sn Push state n onto stack. Advance current token.

reduce Rn Remove L elements from the stack where L = length of RHS of rule n. Push a goto n action. *reduce* also generates an AST node for the rule.

accept a Accept input stream (parse was successful)

error Report error

goto Gn Push ParsingTable[Stack[Top], LHS of [R]]

See https://imperial.cloud.panopto.eu/Panopto/Pages/Viewer.aspx?id=b12a0301-7f1d-466 video for detailed explanations.

1.2.4 Different LR Parsers

LR(0) A reduce item $X \to A\cdot$ **always** causes a reduction.

SLR(1) A reduce item $X \to A\cdot$ causes a reduction only if the current token is in FOLLOW(X), i.e. can follow A somewhere in the grammar.

LR(1) A reduce item $X \to A\cdot, t$ causes a reduction only if the current token is equal to the look-ahead token t.

LALR(1) Like LR(1) but combines LR(1) states that differ in look-ahead token of the item only.

1.2.5 FIRST and FOLLOW sets

FIRST set for a sequence of rules (non-terminals) and tokens (terminals) α, is the set of all tokens that could **start a derivation** of α, plus ϵ if α could derive ϵ.

FOLLOW set for a rule A is the set of all tokens that could **immediately follow** A, plus \$ if A can end the input.

FIRST set for token T is T, for ϵ is ϵ. FOLLOW sets does not contain ϵ (not sensible).

Constructing FIRST sets

FIRST(A) can be computed by iterating rules of A:
For $A \to B|C|D|\epsilon$:

- For each alternatives:
  - If B is a token, add B.
  - If B is a rule, add $FIRST(B)$.
- A can be ϵ, so add ϵ in FIRST set.

Constructing FOLLOW sets

FOLLOW(A) can be computed by iterating rules that has A in its RHS:

- For **each** $B \to C\ A$, include FOLLOW(B).
- For **each** $B \to C\ A\ D$:
  - Since D follows A, include tokens that can start derivation of D, i.e.FIRST(D).
  - If D can be ϵ, which means the rule can be $B \to C\ A$, include FOLLOW(B).
- Also add \$ if B can end input.

Alternative Video Tutorial

Watch lecture video at 2:42 for how to derive FIRST set. Watch at 6:06 for how to derive FOLLOW set.

1.2.6 LR(1) Parsers

LR(1) Items

An LR(1) item is a pair $[LR(0)item, t]$.

1.3 LL Parsing

A grammar can be $LL(1)$ iff:

- For **each** distinct pairs of alternatives (α, β) of a rule $A \to \alpha \mid \beta$, FIRST(α) and FIRST(β) are disjoint, i.e. the tokens that could start α are distinct from the token that could start β.
- **and,**
  - If FIRST(α) contains ϵ then FIRST(β) and FOLLOW(A) are disjoint.
  - and, if FIRST(β) contains ϵ then FIRST(α) and FOLLOW(A) are disjoint.

More generally speaking, $LL(1)$ **rules must be unambiguous, and not left recursive**.

1.3.1 Backus-Naur Form

The rules for a context free grammar take the following **canonical form**:

$$A \to \alpha$$

where A is a rule and α is possibly empty sequence of rules and tokens.
BNF extends the canonical form, such that alternations $A \to \alpha|\beta$ can be used to replace $A \to \alpha$ and $A \to \beta$.

1.3.2 Extended BNF

EBNF extends BNF with rules:

- α: 0 or more occurrences of α (Repetition)
- $[\alpha]$: 0 or 1 occurrences of α (Optional)
- (α): α. Useful for grouping. (Grouping)

1.3.3 Grammar to LL Parse Functions

Convert body of a rule according to the mappings:

- Rule A to $A()$
- Terminal T to $match(T)$

Body of Rule	Mapped Code
$A\ B$	$A();\ B();$

$A \mid B$	**when** (token) {
	in FIRST(A) -> A()
	in FIRST(B) -> B()
	// So FIRST(A) and FIRST(B) must be disjoint
	else -> error()
	}

A	**while** (token **in** FIRST(A)) A() // So FIRST(A) must be disjoint with what follows {A}
$[A]$	**if** (token **in** FIRST(A)) A() // So FIRST(A) must be disjoint with what follows [A]

1.3.4 Context Free Grammar to LL Parse Functions

Left Factorisation

Common prefix in alternatives can be **left-factorised**:

- EBNF:
  - $A \to B\ C \mid B\ D$ can be $A \to B\ (C \mid D)$
  - $A \to B\ C \mid B$ can be $A \to B\ [C]$
- BNF:
  - $A \to B\ C \mid B\ D$ can be $A \to B\ X$, $X \to C \mid D$
  - $A \to B\ C \mid B$ can be $A \to B\ X$, $X \to C \mid \epsilon$

Substitution

Substitution is replacing a rule A with its alternatives, which possibly enabled

Example:

```
Statement  -> Assignment | ProcCall | pass
Assignment -> id '=' Expr
ProcCall   -> id '(' Expression ')'
```

Substituting Assignment and ProcCall:

```
Statement  -> id '=' Expr | id '(' Expression ')' | pass
```

Then left-factor **id**:

```
Statement  -> id ('=' Expr | '(' Expression ')') | pass
```

So we are able to left-factor some rules by using substitution. Note that we have changed the structure of our grammar. A grammar generator should still generate proper AST for Assignment and ProcCall using the left-factorised $LL(1)$ grammar.

Left-Recursion Removal

The example shows how to perform **direct left-recursion removal**:

```
A -> X | A Y
```

By expanding the rule second alternative of rule A:

```
AY => AYY => AYYY => ...
```

So we can remove the direct left-recursion:

```
A -> X | A Y  =>  A -> X { Y }
```

Note that this **may** change associativity. We need to ensure correct associativity when constructing AST.

Error Recovery

If error recovery is attempted, skip as little as possible in order to parse as much of the remaining code as possible. We should also generate helpful error messages.
If error correction is attempted, ensure the corrected program has the same syntax tree and semantics.

Panic-mode Error Recovery

In **Panic-mode Error Recovery** we provide each parse function an additional parameter **a set of synchronizing tokens**, i.e. **syncset**.
As parsing proceeds, additional tokens added to the syncset when calling other parse functions.
When error occurs we skip ahead discarding tokens until one of the synchronising token is seen.

A common heuristic is to add all tokens in FOLLOW(A) to the *syncset* for a rule A. If A occurs in an 'outer' construct then we **also** add **First** set of the outer construct.
Example:

```
ifStatement -> if expr1 then expr2 else expr3 fi
expr -> ifStatement | beginStatement | printStatement
beginStatement -> begin
printStatement -> print
```

Suppose we see an unwanted token after seeing **then** (i.e. we are pasing expr2), we would discard tokens ahead until we see **else**. So syncset for parsing expr2 should be **else**.
When parsing expr2 (an expr), we would expect to see **if**, **begin**, or **print** (**expectset**).
So we check if current token is either in **expectset** —in which case we proceed parsing an expr, or we skip until **syncset** —to recover error.