Parsing Theory

Left and right associative operators

To remove the ambiguit and treat "-" as a left associate operator (as usual) we can rewrite the grammar to
$E \to E - T$
$E \to T$
$T \to N$
and to treat "-" as right associative we use
$E \to T - E$
$E \to T$
$T \to N$

Recusive-Descent Parsing

Translating EBNF to BNF

Replace optional construct $[S]$ by a new nonterminal $OptS$:
$OptS \to S|\epsilon$
For example, we can rewrite
$RelCondition \to Exp\ [RelOp\ Exp]$
as
$RelCondition \to Exp\ OptRelExp$
$OptRelExp \to RelOp\ Exp|\epsilon$
Replace grouping construct (S) by a new nonterminal $GrpS$:
$GrpS \to S$
For example, we can rewrite
$RepF \to$
$(TIMES \mid DIVIDE)\ Factor\ RepF\ |\epsilon$
as
$RepF \to TermOp\ Factor\ RepF\ |\epsilon$
$TermOp \to TIMES \mid DIVIDE$

parsing example

First & Follow Sets

First Sets

The first set for a construct α records
- the set of terminal symbols α can start with, and
- if α is nullable, it contains the empty string ϵ to indicate that.

Calculating First Sets

Let
- α be a terminal symbok,
- $\alpha_1, \alpha_2, \ldots, \alpha_n$ be strings of (terminal and nonterminal) symbols, and
- A be a nonterminal symbol defined by a single production

$$A \to \alpha_1|\alpha_2|\ldots|\alpha_n,$$

then

$$\begin{aligned} Fst(\epsilon) &= \{\epsilon\} \\ Fst(\alpha) &= \{\alpha\} \\ Fst(\alpha_1|\alpha_2|\ldots|\alpha_n) &= Fst(\alpha_1) \cup Fst(\alpha_2) \\ &\quad \cup \cdots \cup Fst(\alpha_n) \\ Fst(A) &= Fst(\alpha_1|\alpha_2|\ldots|\alpha_n) \end{aligned}$$

Let $S_1, S_2, \ldots, S_n$ be (terminal or nonterminal) symbols, then

$$\begin{aligned} &Fst(S_1 S_2 \ldots S_n) = \\ &\quad Fst(S_1) - \{\epsilon\} \\ &\quad \cup Fst(S_2) - \{\epsilon\} && \text{if } S_1 \text{ is nullable} \\ &\quad \cup \ldots \\ &\quad \cup Fst(S_i) - \{\epsilon\} && \text{if } S_1\text{-}S_{i-1} \text{ is nullable} \\ &\quad \cup \ldots \\ &\quad \cup Fst(S_n) - \{\epsilon\} && \text{if } S_1\text{-}S_{n-1} \text{ is nullable} \\ &\quad \cup \{\epsilon\} && \text{if } S_1\text{-}S_n \text{ is nullbale} \end{aligned}$$

Calculating Algorithmically

Start with the first sets for all nonterminals being the empty set and note that the first set for every terminal symbol, α, is the singleton set $\{\alpha\}$. We then make a pass over all production in a grammer considering all atlternatives and process as follows.
- If there is a production of the form $N \to \epsilon$, we add ϵ to the first set of N.
- If there is a production of the form $N \to S_1 S_2 \ldots S_n$, then for each $i \in 1..n$, if for all $j \in 1..i-1$, S_j is nullabe, we add the current first set for S_i minus ϵ to the first set for N.
- If every construct $S_1, ..., S_n$ is nullabe, we add ϵ to the first set for N.

We repeat the the passes until no set is modified in the pass, in which case we are finished.

Example

$$\begin{aligned} A &\to B\,x|C \\ B &\to C\,y|D \\ C &\to D\,z|\epsilon \\ D &\to A\,w \end{aligned}$$

A	$\{\}$	$\{\epsilon\}$	$\{\epsilon, y\}$	$\{\epsilon, y, w\}$	$\{\epsilon, y, w\}$
B	$\{\}$	$\{y\}$	$\{y\}$	$\{y, w\}$	$\{y, w\}$
C	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon, w\}$	$\{\epsilon, w, y\}$	$\{\epsilon, w, y\}$
D	$\{\}$	$\{w\}$	$\{w, y\}$	$\{w, y\}$	$\{w, y\}$

Follow Sets

The follow set for a nonterminal, N, is the set of terminal symbols that may follow N in any context within the grammar. End-of-file is represented by the special terminal symbol \$ in which a follows N.
A nonterminal, N, is followed by a terminal symbol, a, if there is a derivation from $S\$$

Calculating Follow Sets

We compute the Follow set for a nonterminal, N, using two rules.
- If there is a production of the form
$$A \to \alpha N \beta$$
then any symbols that can start β can follow N, and hence $Follow(N)$ must include all the terminal symbols in $First(\beta)$. Note that ϵ is not included even if it appears in $First(\beta)$.
$$First(\beta) - \{\epsilon\} \subseteq Follow(N)$$
- If there is a production of the form
$$A \to \alpha N \beta$$
and β is nullable, then any token that can follow A can also follow N. Hence,
$$Follow(A) \subseteq Follow(N)$$
The case where β is nullable includes the case when β is empty and the production is of the form
$$A \to \alpha N$$

Calculating Algorithmically

Start with all nonterminal symbols having an empty follow set, {}, except for the start symbol, S, which has the follow set {\$}.
We make a pass through the grammar examining the right side of every production. For each occurence of a nonterminal within the right side of some production, we augment the Follow set for that nonterminal according to the following process. Assume we are processing an occurence of N and the production is of the form
$$A \to \alpha N \beta$$
we add $First(\beta) - \{\epsilon\}$ to the Follow set computer for N so far, and if β is nullable, we also add the current Follow set for A to the Follow set for N.
After making a complete pass, we repeat the process with the Follow sets computed so far until no Follow sets are modified, in which case we are done.

Example

$$\begin{aligned} S &\to xAB && Fst(S) = \{x\} \\ A &\to y|zB && Fst(A) = \{y, z\} \\ B &\to \epsilon|Ax && Fst(B) = \{\epsilon, y, z\} \end{aligned}$$

S	{\$}	{\$}	{\$}	{\$}
A	$\{\}$	$\{y, z, \$, x\}$	$\{y, z, \$, x\}$	$\{y, z, \$, x\}$
B	$\{\}$	$\{\$, y, z\}$	$\{\$, y, z, x\}$	$\{\$, y, z, x\}$

LL(1) Grammar

A BNF grammer is LL(1) if for each nonterminal, N, wher $N \to \alpha_1|\alpha_2|\ldots|\alpha_n$,
- the First sets for each pair of alternatives for N are disjoint, and
- if N is nullable, $First(N)$ and $Follow(N)$ are disjoin.

Bottom Up Parsing

Shift/Reduce Parsing

Makes use of a parse stack and has three actions:
- **shift** - push the next input symbol onto the stack,
- **reduce** - if a sequence of symbols on the top of the stack, α, matches the right side of some production $N \to \alpha$, then the sequence α on top of the stack is replaced by N.
- **accept** - if the stack contains just the start symbol and there is no input left, the input has been recognised and is accepted.

LR(0) Grammars

An LR(0) parsing item is of the form
$$N \to \alpha \bullet \beta$$
which indicates that, in matching N, α has been matched and β is yet to be matched where
- N is a nonterminal symbol,
- α and β are possibly empty sequences of (terminal and nonterminal) symbols such that $N \to \alpha\ \beta$ is a production of the grammer, and
- $\bullet$ marks the current position in matching the right side.

Parsing Automaton

An LR(0) parsing automaton consists of
- a finite set of states, each of which consists of a set of LR(0) parsing items, and
- transitions between states, each of which is labelled by a transition symbol.

Each state in an LR(0) parsing automaton must have only one associated parsing action.

Automaton States

If a state has an LR(0) item of the form

$$N \to \alpha \bullet M\beta$$

where the nonterminal M has productions

$$M \to \alpha_1$$
$$M \to \alpha_2$$
$$\vdots$$
$$M \to \alpha_m$$

then the state also includes the derived items

$$M \to \bullet\alpha_1$$
$$M \to \bullet\alpha_2$$
$$\vdots$$
$$M \to \bullet\alpha_m$$

Goto States

If a state s_i has an LR(0) item of the form

$$N \to \alpha \bullet x\beta$$

where x is either a terminal symbol or a nonterminal symbol, then there is a goto state, s_j, from the state s_1 on x, and s_j includes a kernel item of the form

$$N \to \alpha x \bullet \beta$$

If there are multiple items in s_i with the same x immediately to the right of the $\bullet$ then the goto state s_j includes all those items but with the $\bullet$ after that occurence of x rather than before it.

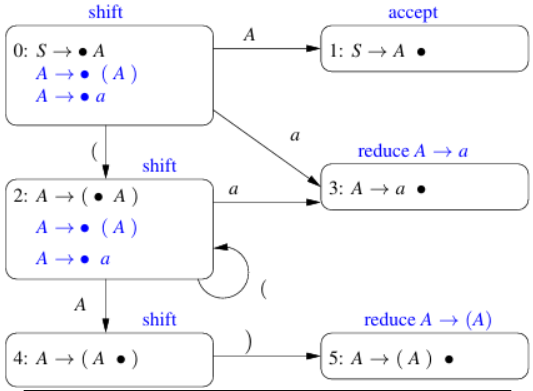
Actions

An LR(0) item of the form
- $N \to \alpha \bullet a\beta$ where a is a terminal symbol, indicates the state containing the item has a shift **parsing** action
- $S' \to S\bullet$ where S' is the (introduced) start symbol for the grammer, inidcates the state containing the item has an **accept** action
- $N \to \alpha\bullet$ where N is not the (introduced) start symbol for the grammar, indicates the state containing the item has a parsing action **reduce** $N \to \alpha$

A shift action at end-of-file is an error, as is an accept action when the input is not at end-of-file.

Example

$$S \to A \qquad A \to (\ A\ ) \qquad A \to a$$



Parsing stack	Input	Parsing action
\$0	((a))\$	shift
\$0(2	(a))\$	shift
\$0(2(2	a))\$	shift
\$0(2(2a3	))\$	reduce $A \to a$
\$0(2(2A4	))\$	shift
\$0(2(2A4)5	)\$	reduce $A \to (A)$
\$0(2A4	)\$	shift
\$0(2A4)5	\$	reduce $A \to (A)$
\$0A1	\$	accept

Parsing Conflicts

If a state in an LR(0) parsing automaton has more than one action, there is a parsing action conflict.
A grammar is LR(0) if none of the states in its LR(0) parsing automaton contains a parsing action conflict.

LR(1) Grammars

An LR(1) parsing item is a pair

$$[N \to \alpha \bullet \beta, T]$$

consisting of
- an LR(0) parsing item $N \to a \bullet \beta$, and
- a set T of terminal symbols called a look-ahead set.

The above item indicates that in matchin N, α has been matched and β is yet to be matched, in a context in which N can be followed by a terminal symbol in the set T.

Parsing Automaton

The kernal item of the initial state is

$$[S' \to \bullet S, \$]$$

where S is the start symbol of the grammar and we introduce a fresh replacement start symbol S' and production $S' \to S$. This fresh production is used to determine when parsing has completed.

Derived Items

If a state has an LR(1) item of the form

$$[N \to \alpha \bullet M\beta, T]$$

where the nonterminal M has productions

$$M \to \alpha_1 | \alpha_2 | \ldots | \alpha_m$$

and $T = \{a_1, a_2, \ldots, a_n\}$, then the state also includes the derived items

$$[N \to \bullet\alpha_1, T] \ldots [N \to \bullet\alpha_m, T]$$

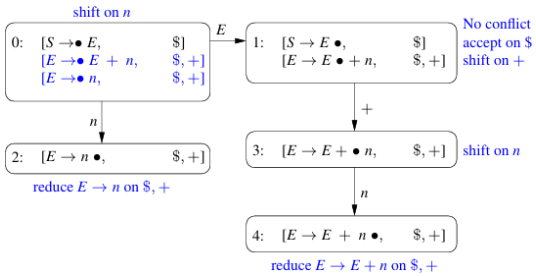
where if β is not nullable $T' = First(\beta)$ and if β is nullable, $T' = First(\beta) - \{\epsilon\} \cup T$.

Parsing Actions

An LR(1) item of the form
- $[N \to \alpha \bullet a\beta, T]$, where a is a terminal symbol, indicates the state containing the item has a **shift** parsion action if the next input x is a
- $[S' \to \S\bullet, \$]$, where S' is the added start symbol for the grammar, indicates the state containing the item has an **accept** action if there is no more input.
- $[N \to \alpha\bullet, T]$, where N is not S', indicates the state containing the item has a parsing action **reduce** $N \to \alpha$ if the next input x is in T.

Example

$$S \to E \qquad E \to E + n \qquad E \to n$$



Parsing Action Conflicts

If a state in an LR(1) parsing automaton has more than one action for a look-ahead terminal symbol, there is a parsing action conflict.
A grammar is LR(1) if none of the states in its LR(1) parsing automaton contains a parsing action conflict.

LALR(1) Parsing

- Look-Ahead merged LR(1) parsing where each state of an LALR(1) parsing automaton consists of a set of LR(1) items.
- An LALR(1) parsing automaton can be formed from an LR(1) parsing automaton by merging states that have identical sets of LR(0) items but possibly different look-ahead sets.
- Each item in the merged state consists of one of the LR(0) items in the states being merged with a look-ahead set that is the union of the look-ahead sets of that item in all the states being merged.
- The parsing actions for LALR(1) parsing are defined in the same way as for LR(1) parsing.
- A grammar is LALR(1) if none of the states in its LALR(1) parsing automaton contains a parsing action conflict.

Left Factoring & Left Recursion

Left Factoring Productions

Not all EBNF grammars are suitable for Recursive-Descent Parsing, however, sometimes we can rewrite them into a form that is suitable.

Left Factor Rewriting Rule

To remove the left factor from

$$A \to \alpha\ \beta \mid \alpha\ \gamma,$$

we can rewrite the production using an aditional nonterminal A' as

$$A \to \alpha\ A'$$
$$A' \to \beta \mid \gamma$$

Left Recursive Productions

A production of the form

$$E \to E\ +\ T \mid T$$

is not suitable for RDP because the left recursion in the grammar leads to an infinite recursion.

Immediate Left Recursion Rewriting Rule (Simple)

To remove the left recursion from

$$A \to A\ \alpha \mid \beta,$$

we can rewrite the production as

$$A \to \beta\ A'$$
$$A' \to \epsilon \mid \alpha\ A'$$

Immediate Left Recursion Rewriting Rule (General)

To remove the left recursion from the general case

$$A \to A\alpha_1|A\alpha_2|\dots|A\alpha_n|\beta_1|\beta_2|\dots|\beta_m,$$

we can use grouping to see the structure in the same form as the simple case

$$A \to A(\alpha_1|\alpha_2|\dots|\alpha_n)|(\beta_1|\beta_2|\dots|\beta_m)$$

which allows us to rewrite the production as follows,

$$A \to (\beta_1|\beta_2|\dots|\beta_m)A'$$
$$A' \to \epsilon|(\alpha_1|\alpha_2|\dots|\alpha_n)A'$$

Indirect Left Recursion Rewriting Rule

The following productions have indirect recursion from $A \to B \to C \to A$.

$$A \to B\ \alpha$$
$$B \to C\ \beta$$
$$C \to A\ \gamma_1 \mid \gamma_2$$

To remove the recursion we can first collapse B into A as follows.

$$A \to C\ \beta\ \alpha$$
$$C \to A\ \gamma_1 \mid \gamma_2$$

Then we can collapse C into A.

$$A \to (A\ \gamma_1 \mid \gamma_2)\ \beta\ \alpha$$
$$\downarrow$$
$$A \to A\ \gamma_1\ \beta\ \alpha \mid \gamma_2\ \beta\ \alpha$$

This now leaves a simple direct left recursion which we can remove as follows.

$$A \to \gamma_2\ \beta\ \alpha\ A'$$
$$A' \to \gamma_1\ \beta\ \alpha\ A'$$

Stack Organisation

Definition

A stack machine consists of the following chunks of memory.

- **stack** - the portion of memory used for both calculating the values of expressions as well as storing activation records for every active procedure call.
- **heap** - the portion of memory for dynamic allocation of Objects.
- **code space** - the portion of memory where the machine instructions are stored.

The stack and heap are stored in one contiguous area of memory. The stack grows from the bottom (address 0) and the heap grows down from the top.

The machine has four special registers:

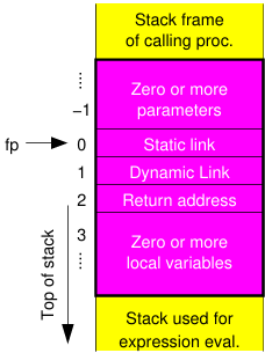
- **stack pointer** - contains the address for the top of the stack + 1
- **stack limit** - contains the address of the upper limit for the stack and the bottom of the heap
- **frame pointer** - contains the address of teh stack frame for the current procedure
- **program counter** - contains the address of the next machine instruction

Procedures

Each time a procedure is entered via a call, the machine must keep track of information such as the return address and the local variables for that call in a procedure frame. This frame is stored on the stack.
A pointer to the current procedures' frame is stored in a globally known location.

Calling a Procedure

- parameters to the procedure are pushed to the stack
- a static link is pushed onto the stack
- the current frame pointer is pushed onto the stack to create the dynamic link
- the frame pointer is set so that it contains the address of the start of the new stack frame
- the current value of the program counter is pushed onto the stack to form the return address
- the program counter is set to the address of the procedure
- space is allocated on the stack for any local variables



Returning From a Procedure

- the program counter is set to the return address in the current activation record
- the frame pointer is set to the dynamic link
- the stack pointer is set so that all the space used by the stack frame (but not parameters) is popped from the stack
- execution continues at the instruction addressed by the (restored) program counter
- after return, the calling procedure handles deallocating any parameters

Local Variables

Local variables are stored within the procedures frame. They are accessed by an offset relative to the frame pointer.

Non-local Variables

To allow access to variables outside of the enclosing procedure, the stack frame for a procedure includes a *static link* which contains the address of the stack frame for the enclosing procedure (the procedure in which the enclosed procedure is defined).
To access the non-local variable n from a procedure, we continuesly access the static link of the enclosing procedures until we are in the procedure in which n is defined. We can then add the offset to the variable n to get the final address of the non-local variable.

Parameters

- **Formal parameters** are the paremeters used in the declaration of the procedure in its header.
- **Actual parameters** are the actual parameters passed to a procedure on a call.

Call-by-value

- parameters are expressions that are evaluated and coerced to the type of the formal paramter.
- the values of the parameters are loaded onto the stack as part of the calling sequence.
- Once the procedure is called and the stack frame has been established, the formal parameters of the procedure can be accessed like local variables
- Accesses to the formal parameter access the location on the stack containing the value of the corresponding actual parameter.

Call-by-const

The formal parameter acts as a read-only local variable that is assigned the value of the actual parameter expression.

Call-by-result

The formal parameter acts as a local variable whose final value is assigned to the actual parameter variable.

Call-by-value-result

A single parameter acts as both a value and a result parameter.

Call-by-reference

The formal parameter is the address of the actual parameter variable. All references to the formal parameter are applied to the actual parameter variable immediately.

Call-by-sharing

The same as call-by-value, but what is passed is a reference to an object (e.g. Java) rather than the values of the object.

Call-by-name

The actual parameter expression is evaluated every time the formal parameter is accessed.

Passing Procedures as Parameters

The address of the procedure as well as the static link for the procedure's environment is passed.

Function Results

- A function can return a result that can be used as part of an expression.
- The result of a function call should be left on top of the stack after the stack frame is removed.
- To ensure the returned value is before the stack frame, free space is allocataed for the result before the parameters are loaded onto the stack and the frame is set up.

Returning Procedures

Return the address of the procedure as well as the static link for the procedures environment. This requires the environment of the returned procedure to be maintained which makes the simple stack-based allocation of frames insufficient.

Variable Aliasing

Parameter Aliasing

In languages with call-by-reference, the same variable can be passed to two (or more) different parameters leading to variable aliasing.

Global Variable Aliasing

If a variable is passed as a reference parameter to a procedure that can access the same variable as a global variabel, then within the procedure there are two aliases for the same variable.

Pointer Aliasing

Parameter Aliasing

In languages with call by sharing, the same reference to an object can be passed to two different parameters leading to one having two aliases for the same reference.

Global Variable Aliasing

If a reference that is passed as a parameter to a procedure is also directly accessible as a global variable from the procedure. this leads to the procedure having two aliases for the one reference.

Objects

Heap Organisation

Dynamic Memory Allocation

Dynamic allocation of objects takes place on the *heap*, an area of memory seperate from the stack.

Dynamic Memory Deallocation

There are two main forms of memory deallocation used in programming languages:

- **Explicit deallocation** in which there is a function to explicitly deallocate and object
- **Garbage collection** in which the space used by unreachable objects is reclaimed by a garbage collector.

Explicit Memory Deallocation

- freed memory is returned to the "free list"
- if the memory freed is adjacent to other blocks of memory in the free list, the adjacent free blocks are merged into a single larger free block to reduce fragmentation
- the freed memory is available for re-allocation

Issues w/ Explicit Deallocation

- **Dangling references** - an object can be freed via one reference but still be accessible via other references
- **Memory leaks** - all references to an object are removed but the object was not freed.
- **Memory fragmentation** - the memory consists of alternating alloctated and free areas, with no large area of free memory.
- **Locality of reference** - how spread out the memory is in the address space.

All of these issues can be solved with a garbage collector.

Garbage Collection

An object is accessible if it can be reached either

- directly via a reference in a global or local variable, or
- indirectly via a reference in an object that is accessible.

Garbage collection determined which objects are not accessible and recovers the space used by them.

Mark-and-sweep

Mark and sweep consists of

- a phase that **marks** all the accessible objects
- a phase that **sweeps** up the objects left unmarked and adds them to the free list

Stop-and-copy

- divides the available memory into two spaces
- memory is allocated sequentially from one space until it runs out
- garbage collection consists of relacating all accessible objects from the first space to the second space
- because the objects are allocated sequentially in the second space, they are compacted
- when the copy is completed, the roles of the spaces are swapped for the next garbage collection

Overcomes memory fragmentation problems, but copying can be expensive in time.

Generational Schemes

Generational schemes use a scheme similar to the stop-and-copy scheme, but make us of more spaces

- the spaces are organised based on the length of time its objects have survived
- the ages of an object is the number of times it has been collected
- older objects are migrated to an old object space
- newer objects go in the new object space
- the objects in the old object space is garbage collected
- the old object space may have to be garbage colelcted at some stage

Issues w/ Garbage Collection

- Garbage collection has a greater time overhead than explicit deallocation
- Response time can vary because
- garbage collection can happen at any time
- it can take a significant amount of time
- Real-time response hard to guarantee
- Pre-allocating all objects necessary
- Use incremental on-the-fly garbage collected
- Use concurrent garbage collector
- It is complex and tricky to get correct

On-the-fly

- Garbage collection process takes place incrementally, interleaed with execution of the program.
- Each time an object is allocated, the garbage collector can do a bit of work.
- Avoids the program stopping for a longish time interval to garbage collect in one go, which can be a problem when real-time response is required.

Concurrent

- The garbage collector runs concurrently with the program on a separate processor.

Code Generation

Regular Expressions

The syntax of regular expressions in BNF is as follows:

$$e ::= a|\epsilon|\theta|e"|"e|e\ e|e"*"|(e)$$

Note that regex uses "—" to list alternatives similar to BNF notation.

Language of a regular expression

Given an alphabet Σ, the set of regular expressions defines a language (i.e., a set opf strings of symbols from the alphabet). Let "a" be a symbol, and e and f regular expressions.

- $\mathcal{L}(a) = \{"a"\}$
- $\mathcal{L}(\epsilon) = \{""\}$
- $\mathcal{L}(\theta) = \{\}$
- $\mathcal{L}(e|f) = \mathcal{L}(e) \cup \mathcal{L}(f)$
- $\mathcal{L}(e\ f) = \mathcal{L}(e) \frown \mathcal{L}(f)$
- $\mathcal{L}(e^*) = \cup_{n\in\mathbb{N}}(\mathcal{L}(e))^n$
- $\mathcal{L}((e)) = \mathcal{L}(e)$

Concatenation of languages

The concatenation of two langauges (sets of string), L_1 and L_2 is defined as the set of all strings formed by taking a string, s_1 from L_1 and a string, s_2 from L_2, and concatenating them. For example,

$$\{a, b\} \frown \{c, d\} = \{ac, ad, bc, bd\}$$

Iteration of a language

The iteration, L_n, of a language, L, to the power n, consists of L concatenated with itself n times.

$$\begin{aligned} L^0 &= \{\} \\ L^1 &= L \\ L^2 &= L \frown L \\ L^3 &= L \frown L \frown L \\ &\dots \end{aligned}$$

$\mathcal{L}(a|b) = a, b$ and

$$\begin{aligned} \{a, b\}^0 &= \{\} \\ \{a, b\}^1 &= \{a, b\} \\ \{a, b\}^2 &= \{aa, ab, ba, bb\} \\ &\dots \end{aligned}$$

Hence $\mathcal{L}((a|b)^*) = \{, a, b, aa, ab, ba, bb \dots\}$ Note that this does not include any infinite strings of symbols because the iteration only generates finite concatenations.

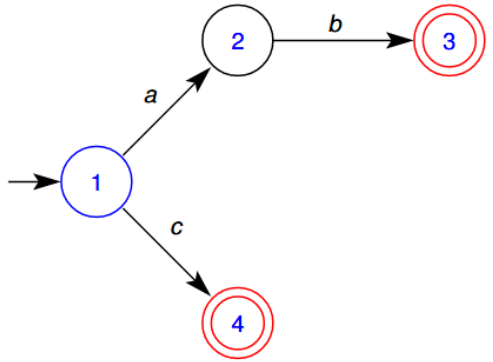
Finite Automata

- A finite automaton is a finite state machine consisting of a fine set of states with labelled transitions between states.
- Each transitioin is labelled with either a symbol ($a \in \Sigma$) or empty (ϵ).
- We distinguish between deterministic and nondeterminisitc finite automata.
- For a deterministic finite automaton (DFA) empty transitions are not allowed, and for each state and symbol the next statte, if there is one, is uniquely determined.
- For a nondeterministic finite automaton (NFA) empty transitions are allowed, and there may be multiple transitions from a state to a different next states on the same input symbol.

Deterministic Finite Automaton (DFA)

A DFA, D, consists of

- A finite alphabet of symbols, Σ
- A finite set of states, S
- A transition function, $T : \Sigma \times S \to S$, which maps an (input) symbol and a (current) state to the (next) state; the function T may not be defined for all pairs of symbols and state.
- A start state, s_0
- A set of final (or accepting) states, F



- $\Sigma = a, b, c$
- $S = \{1, 2, 3, 4\}$
- $T(a, 1) = 2$
- $T(c, 1) = 4$
- $T(b, 2) = 3$
- $s_0 = 1$
- $F = \{3, 4\}$

Language of a DFA

The language $\mathcal{L}(D)$, of a DFA D, is the set of finite strings fo symbols from Σ such that $c \in \mathcal{L}(D)$ if and only if there exists a sequence of states $s_1, s_2, \ldots, s_n$ where n is the legnth of c, such that

$$T(c_1, s_0) = s_1$$
$$T(c_2, s_1) = s_2$$
$$\ldots$$
$$T(c_n, s_{n-1}) = s_n$$

and s_0 is the start state and s_n is in the set of final (or accepting) states F.

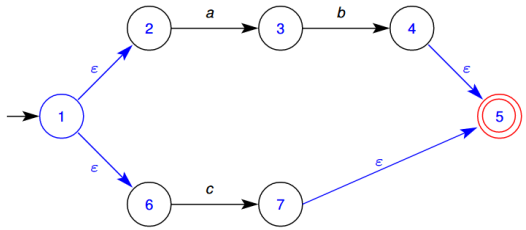
The language of the above DFA is $\mathcal{L}(D) = \{ab, c\}$

Nondeterministic Finite Automaton (DFA)

A NFA, N, is different to a DFA in that its Transition function is $T : (\Sigma \cup \{\epsilon\}) \times S \to \mathbb{P}S$, which maps an (input) symbol and a (current) state to a set of possible (next states); the function T may not be defined for all pairs of symbol and state.

NFA

- $\Sigma = a, b, c$
- $S = 1, 2, 3, 4, 5, 6, 7$
- $T(\epsilon) = \{2, 6\} T(a, 2) = \{3\}$
- $T(b, 3) = \{4\} T(\epsilon, 4) = \{5\}$
- $T(c, 6) = \{7\} T(\epsilon, 7) = \{5\}$
- $s_0 = 1$
- $F = \{5\}$



Language of a NFA

The language $\mathcal{L}(N)$, of a NFA N, is the set of finite strings fo symbols from Σ such that $c \in \mathcal{L}(N)$ if and only if there exists a sequence $c' \in (\Sigma \cup \{\epsilon\})^*$, such that removing all the ϵ elements from c' gives c and there exists a sequence of states $s_1, s_2, \ldots, s_n$ where n is the length of c', such that

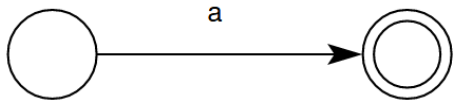
$$T(c'_1, s_0) = s_1$$
$$T(c'_2, s_1) = s_2$$
$$\ldots$$
$$T(c'_n, s_{n-1}) = s_n$$

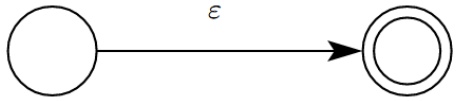
and s_0 is the start state and s_n is in the set of final (or accepting) states F.

The language of the above NFA is $\mathcal{L}(D) = \{ab, c\}$ Empty transitions are used to generate the strings by then omitted from the strings.

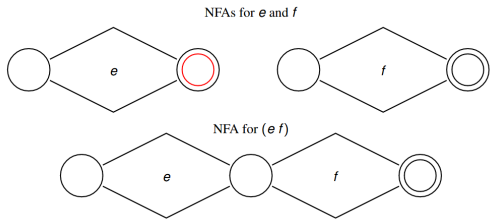
From a Regex to an NFA

- The translation of a regular expression to a nondeterministic finite automaton is based on the structure of the regular expression.
- For each of the syntactic forms of regular expressions given in Definition 1, an NFA is given for that form.
- If the syntactic form contains sub-expressions (e.g. e — f contains sub-expressions e and f) the NFA for that form is constructed from the NFAs for the sub-expressions e and f .



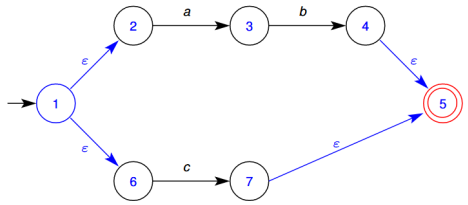


The final (acceptiong states are indicted by double circle.)



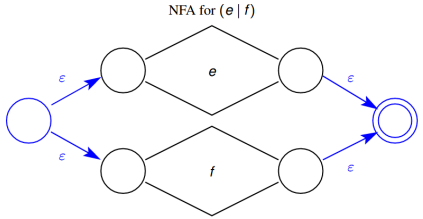
The final state of the NFA for e is no longer a final in the NFA for $e\ f$.

Example a b | c



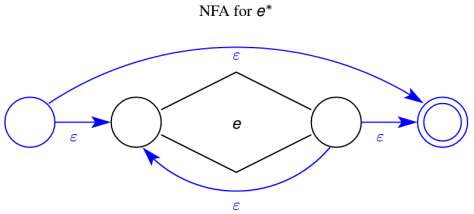
Alternation e | f

The NFAs for e and f are the same as for concatenation.



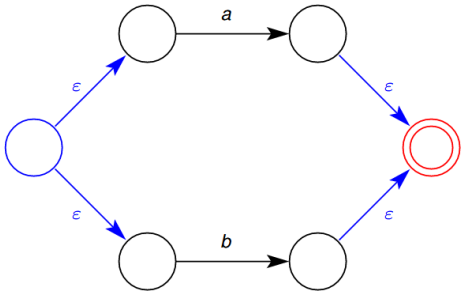
The final states of the NFAs for both e and f are no longer final states in the NFA for e | f.

*Repetition e**

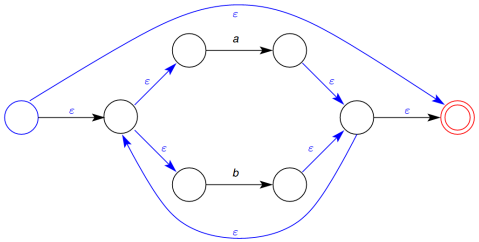


The final state in the NFA for e is no longer an final state in the NFA for e^*.

NFA for a — b



*NFA for (a — b)**



From an Regex to a NFA

In the translation from a regular expression, r, to an NFA, thegenerated NFA has a few properties that do not necessarily hold for an arbitrary NFA (i.e. one not generated from a regular expression).

- The NFA has a single final (accepting) state.
- The initial state of the NFA only has outgoing transitions.
- The final state only has incoming transitions.

The translation rules preserve these properties.

From NFA to DFA

A DFA cannot have

- more than one transition leaving a state on the same symbol
- any empty transitions

An NFA N can be translated to an equivalent DFA D.

- equivalent in the sense that they accept the same languages, i.e., $\mathcal{L}(N) = \mathcal{L}(D)$.

From NFA to DFA

An NFA is transformed to a DFA in which the labels on the states of the DFA are sets

of states from the NFA. The sets of states that label a DFA state are formed by collecting all the states that can be reached from NFA states by empty transitions.

Empty Closure of a state

Empty Closure of a state The empty closure of a state x in an NFA N, ϵ-closure(x, N), is the set of states in N that are reachable from x via any number of empty transitions

Empty Closure of a set of states

The empty closure of a set of states X in an NFA N, ϵ-closure(X, N), is the set of states in N that are reachable from any of the states in X via any number of empty transitions.

$$\epsilon\text{-closure}(X, N) = \bigcup_{x \in X} \epsilon\text{-closure}(x, N)$$

Empty close of $a\ b \mid c$

State x	ϵ-closure$(x$, NFA$)$
1	{1, 2, 6}
2	{2}
3	{3}
4	{4, 5}
5	{5}
6	{6}
7	{7, 5}

Constructing the DFA from the NFA

The label of the start state of the DFA consists of the set of states containing the start state s_0 of the NFA plus all the states in the NFA that are reachable from its start state by one or more empty transitions.

$$S_0 = \epsilon\text{-closure}(s_0, N).$$

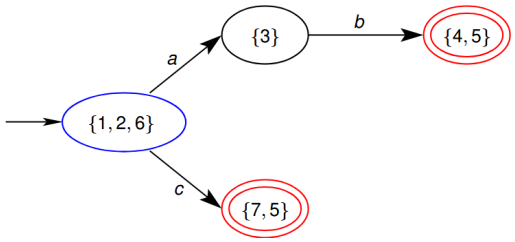
The process for forming a DFA works with a set of unmarked DFA states by selecting an unmarked DFA state and considering all transitions from it on symbols. The initial set of unmarked states contains just S_0.

From NFA to DFA

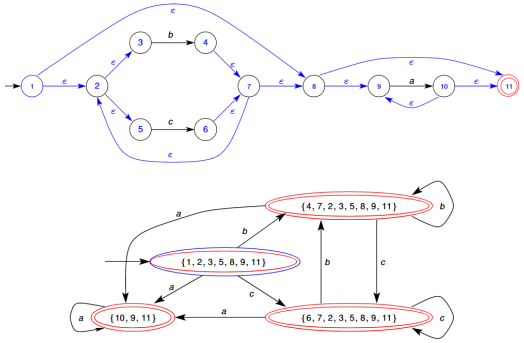
The following process is repeated until there are no unmarked DFA states left:

- An unmarked DFA state S is selected (the first one is S_0).
- For each symbol a,
  - we consider the set of states that can be reached from any state in S by a transition on a; call this set of states X.
  - If X is nonempty, we add a new state to the DFA labeled with $X' = \epsilon$-closure(X, N), unless a state with that label already exists, in which case it is reused.
  - A transition from S to X' on a is added to the DFA.
- The state S is marked as having been processed.

From NFA to DFA for a b — c

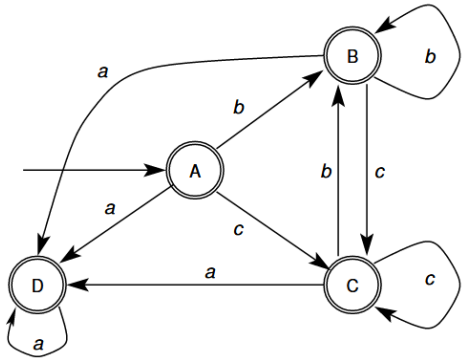


From NFA to DFA for (b — c) a**

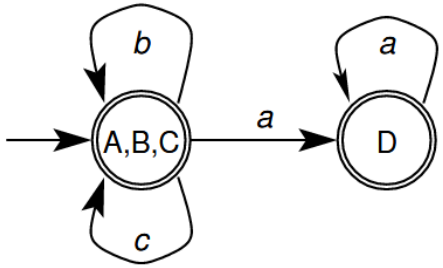


Minimising a DFA

The DFA generated in the above example can be simplified to one with only two states. Below is a simplified version of the DFA with the states relabelled to to A, B, C and D.



To minimise the DFA we merge states that have the same transition to the equivalent states. For the example, A, B and C are equivalent.



Start by partitioning states into a group of final states and a group of non-final states. For the example, all states are all final so just get the one group {A, B, C, D}. For each symbol x, we require all states in a group $G1$ to transition to the same group $G2$. For the example, a transition on b takes states A, B and C back to the same group {A, B, C} but there is no transition on b from state D and hence we must split the group into groups {A, B, C} and {D}. Once that is done for every symbol x, the transitions from all states in each group on x are to the same group, and hence we have finished.

Regular expressions for lexical analysis

- The lexical analyser generator JFlex makes use of translating regular expressions into DFAs in order to build a scanner.
- The input to the lexical analyser generator is a list of regular expressions along with an action to be taken when that token is matched.
- The generated lexical analyser matches
  - the longest prefix of the input from the current position that matches any of the regular expressions in the list and
  - if the prefix of the input matches more than one regular expression, the action associated with the first matching regular expression in the list is executed.
- For PL0 the string "end" followed by a blank matches both
  - the regular expression for the keyword "end" and
  - the regular expression for an identifier.
- The action selected for "end" is that for the keyword "end" because the regular expression for the keyword appears before the regular expression for an identifier.
- The string "ending" followed by a blank matches only the regular expression for an identifier; it will not be split into the keyword "end" followed by the identifier "ing".
- But the string "end ing " will be matched as the keyword "end" followed by the identifier "ing".