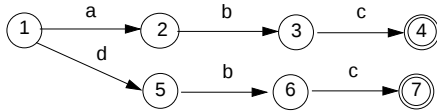


transition to follow for some character, the merged state is *split* into two or more smaller states that do agree.

As an example, assume we start with the following automaton:



Initially we have a merged non-accepting state $\{1,2,3,5,6\}$ and a merged accepting state $\{4,7\}$.

A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur.

We will add an error state s_E to the original DFA that is the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state; rather it allows us to assume every state has a successor under every character. s_E is never merged with any real state.

Algorithm **split**, shown below, splits merged states whose constituents do not agree on a common successor state for all characters. When **split** terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

```

Split(FASet StateSet) {
  repeat
    for(each merged state S in StateSet) {
      Let S correspond to  $\{s_1, \dots, s_n\}$ 
      for(each char c in Alphabet){
        Let  $t_1, \dots, t_n$  be the successor
        states to  $s_1, \dots, s_n$  under c
        if( $t_1, \dots, t_n$  do not all belong to
        the same merged state){
          Split S into two or more new
          states such that  $s_i$  and  $s_j$ 
          remain in the same merged
          state if and only if  $t_i$  and  $t_j$ 
          are in the same merged state}
        }
      until no more splits are possible
    }
}
  
```

Returning to our example, we initially have states $\{1,2,3,5,6\}$ and $\{4,7\}$. Invoking **split**, we first observe that states 3 and 6 have a common successor under c, and states 1, 2, and 5 have no successor under c (equivalently, have the error state s_E as a successor).

This forces a split, yielding $\{1,2,5\}$, $\{3,6\}$ and $\{4,7\}$.

Now, for character b, states 2 and 5 would go to the merged state $\{3,6\}$, but state 1 would not, so another split occurs.

We now have: $\{1\}$, $\{2,5\}$, $\{3,6\}$ and $\{4,7\}$.

At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

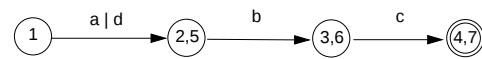
Once **split** is executed, we are essentially done.

Transitions between merged states are the same as the transitions between states in the original DFA.

Thus, if there was a transition between state s_i and s_j under character c , there is now a transition under c from the merged state containing s_i to the merged state containing s_j . The start state is that merged state containing the original start state.

Accepting states are those merged states containing accepting states (recall that accepting and non-accepting states are never merged).

Returning to our example, the minimum state automaton we obtain is



PROPERTIES OF REGULAR EXPRESSIONS AND FINITE AUTOMATA

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form $[[[\dots]]]$. This set is defined formally as $\{ [^m]^m \mid m \geq 1 \}$.

This set is *not* regular.

No finite automaton that recognizes *exactly* this set can exist.

Why? Consider the inputs $[$, $[[$, $[[[$, ... For two different counts (call them i and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)

Once that happens, we know that if $[^i$ is accepted (as it should be), the $[^j$ will also be accepted (and that should not happen).

- $\bar{R} = V^* - R$ is regular if R is. Why? Build a finite automaton for R . Be careful to include transitions to an "error state" s_E for illegal characters. Now invert final and non-final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, $\bar{R}$ is accepted by the modified automaton.

- Not all subsets of a regular set are themselves regular. The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R , in reversed (backward) character order.
Thus if $R = \{abc, def\}$
then $R^{\text{rev}} = \{cba, fed\}$.
If R is regular, then R^{rev} is too.
Why? Build a finite automaton for R . Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
 1. Build two finite automata, one for R_1 and one for R_2 . Pair together states of the two automata to match R_1 and R_2 simultaneously. The paired-state automaton accepts only if both R_1 and R_2 would, so $R_1 \cap R_2$ is matched.
 2. We can use the fact that $R_1 \cap R_2$ is $= \overline{\overline{R_1} \cup \overline{R_2}}$. We already know union and complementation are regular.

Reading Assignment

- Read Chapter 4 of **Crafting a Compiler, Second Edition**.

CONTEXT FREE GRAMMARS

A context-free grammar (CFG) is defined as:

- A finite terminal set V_t ; these are the tokens produced by the scanner.
- A set of intermediate symbols, called non-terminals, V_n .
- A start symbol, a designated non-terminal, that starts all derivations.
- A set of productions (sometimes called rewriting rules) of the form $A \rightarrow X_1 \dots X_m$
 X_1 to X_m may be any combination of terminals and non-terminals.
If $m = 0$ we have $A \rightarrow \lambda$ which is a valid production.

Example

Prog $\rightarrow \{ \text{Stmts} \}$
Stmts $\rightarrow \text{Stmts} ; \text{Stmt}$
Stmts $\rightarrow \text{Stmt}$
Stmt $\rightarrow \text{id} = \text{Expr}$
Expr $\rightarrow \text{id}$
Expr $\rightarrow \text{Expr} + \text{id}$

Often more than one production shares the same left-hand side. Rather than repeat the left hand side, an “or notation” is used:

Prog $\rightarrow \{ \text{Stmts} \}$
Stmts $\rightarrow \text{Stmts} ; \text{Stmt}$
 | **Stmt**
Stmt $\rightarrow \text{id} = \text{Expr}$
Expr $\rightarrow \text{id}$
 | **Expr + id**

DERIVATIONS

Starting with the start symbol, non-terminals are rewritten using productions until only terminals remain.

Any terminal sequence that can be generated in this manner is syntactically valid.

If a terminal sequence can't be generated using the productions of the grammar it is invalid (has syntax errors).

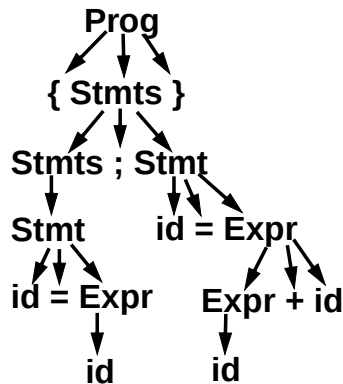
The set of strings derivable from the start symbol is the *language* of the grammar (sometimes denoted $L(G)$).

For example, starting at **Prog** we generate a terminal sequence, by repeatedly applying productions:

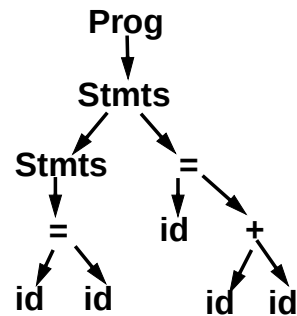
Prog
{ Stmts }
{ Stmts ; Stmt }
{ Stmt ; Stmt }
{ id = Expr ; Stmt }
{ id = id ; Stmt }
{ id = id ; id = Expr }
{ id = id ; id = Expr + id }
{ id = id ; id = id + id }

PARSE TREES

To illustrate a derivation, we can draw a *derivation tree* (also called a *parse tree*):



An *abstract syntax tree* (AST) shows essential structure but eliminates unnecessary delimiters and intermediate symbols:



If $A \rightarrow \gamma$ is a production then
 $\alpha A \beta \Rightarrow \alpha \gamma \beta$
 where $\Rightarrow$ denotes a one step
 derivation (using production
 $A \rightarrow \gamma$).

We extend $\Rightarrow$ to $\Rightarrow^+$ (derives in
 one or more steps), and $\Rightarrow^*$
 (derives in zero or more steps).

We can show our earlier
 derivation as

$\text{Prog} \Rightarrow$
 $\{ \text{Stmts} \} \Rightarrow$
 $\{ \text{Stmts} ; \text{Stmt} \} \Rightarrow$
 $\{ \text{Stmt} ; \text{Stmt} \} \Rightarrow$
 $\{ \text{id} = \text{Expr} ; \text{Stmt} \} \Rightarrow$
 $\{ \text{id} = \text{id} ; \text{Stmt} \} \Rightarrow$
 $\{ \text{id} = \text{id} ; \text{id} = \text{Expr} \} \Rightarrow$
 $\{ \text{id} = \text{id} ; \text{id} = \text{Expr} + \text{id} \} \Rightarrow$
 $\{ \text{id} = \text{id} ; \text{id} = \text{id} + \text{id} \}$
 $\text{Prog} \Rightarrow^+ \{ \text{id} = \text{id} ; \text{id} = \text{id} + \text{id} \}$

When deriving a token sequence,
 if more than one non-terminal is
 present, we have a choice of
 which to expand next.

We must specify, at each step,
 which non-terminal is expanded,
 and what production is applied.

For simplicity we adopt a
 convention on what non-terminal
 is expanded at each step.

We can choose the leftmost
 possible non-terminal at each
 step.

A derivation that follows this rule
 is a *leftmost derivation*.

If we know a derivation is
 leftmost, we need only specify
 what productions are used; the
 choice of non-terminal is always
 fixed.

To denote derivations that are leftmost,

we use $\Rightarrow_L$, $\Rightarrow_L^+$, and $\Rightarrow_L^*$

The production sequence discovered by a large class of parsers (the top-down parsers) is a leftmost derivation, hence these parsers produce a *leftmost parse*.

Prog $\Rightarrow_L$

{ Stmt_s } $\Rightarrow_L$

{ Stmt_s ; Stmt } $\Rightarrow_L$

{ Stmt ; Stmt } $\Rightarrow_L$

{ id = Expr ; Stmt } $\Rightarrow_L$

{ id = id ; Stmt } $\Rightarrow_L$

{ id = id ; id = Expr } $\Rightarrow_L$

{ id = id ; id = Expr + id } $\Rightarrow_L$

{ id = id ; id = id + id }

Prog $\Rightarrow_L^+ \{ id = id ; id = id + id \}$

Rightmost Derivations

A rightmost derivation is an alternative to a leftmost derivation. Now the rightmost non-terminal is always expanded.

This derivation sequence may seem less intuitive given our normal left-to-right bias, but it corresponds to an important class of parsers (the bottom-up parsers, including CUP).

As a bottom-up parser discovers the productions used to derive a token sequence, it discovers a rightmost derivation, but in *reverse order*.

The last production applied in a rightmost derivation is the first that is discovered. The first production used, involving the start symbol, is discovered last.

The sequence of productions recognized by a bottom-up parser is a rightmost parse.

It is the exact reverse of the production sequence that represents a rightmost derivation.

For rightmost derivations, we use the notation $\Rightarrow_R$, $\Rightarrow_R^+$, and $\Rightarrow_R^*$

Prog $\Rightarrow_R$

{ Stmt_s } $\Rightarrow_R$

{ Stmt_s ; Stmt } $\Rightarrow_R$

{ Stmt_s ; id = Expr } $\Rightarrow_R$

{ Stmt_s ; id = Expr + id } $\Rightarrow_R$

{ Stmt_s ; id = id + id } $\Rightarrow_R$

{ Stmt ; id = id + id } $\Rightarrow_R$

{ id = Expr ; id = id + id } $\Rightarrow_R$

{ id = id ; id = id + id }

Prog $\Rightarrow_R^+ \{ id = id ; id = id + id \}$

You can derive the same set of tokens using leftmost and rightmost derivations; the only difference is the order in which productions are used.

Ambiguous GRAMMARS

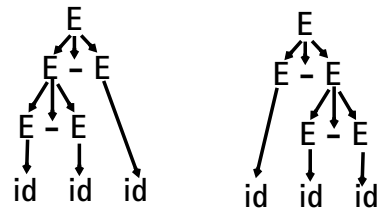
Some grammars allow more than one parse tree for the same token sequence. Such grammars are *ambiguous*. Because compilers use syntactic structure to drive translation, ambiguity is undesirable—it may lead to an unexpected translation.

Consider

$E \rightarrow E - E$

| id

When parsing the input a-b-c (where a, b and c are scanned as identifiers) we can build the following two parse trees:



The effect is to parse a-b-c as either (a-b)-c or a-(b-c). These two groupings are certainly not equivalent.

Ambiguous grammars are usually voided in building compilers; the tools we use, like Yacc and CUP, strongly prefer unambiguous grammars.

To correct this ambiguity, we use

$E \rightarrow E - id$

| id

Now a-b-c can only be parsed as:

