An introduction to parsing

Victor Eijkhout

August 2004

1 Levels of parsing

A compiler, or other translation software, has two main tasks: checking the input for validity, and if it is valid, understanding its meaning and transforming it into an executable that realizes this meaning. We will not go into the generation of the executable code here, but focus on the validity check and the analysis of the meaning, both of which are parsing tasks.

A parser needs to look at the input on all sorts of levels:

- Are all characters valid – no 8-bit ascii?
- Are names, or identifiers, well-formed? In most programming languages `a1` is a valid name, but `1a` is not. By contrast, in TEX a name can only have letters, while in certain Lisp dialects `!!important_name!!` is allowed.
- Are expressions well-formed? An arithmetic expression like `5/*6-` does not make sense, nor does `CALL )FOO(` in Fortran.
- If the input is well-formed, are constraints satisfied such as that every name that is used is defined first?

These different levels are best handled by several different software components. In this chapter we will look at the two initial stages of most translators[1].

1. First of all there is the lexical analysis. Here a file of characters is turned into a stream of tokens. The software that performs this task is called a tokenizer, and it can be formalized. The theoretical construct on which the tokenizer is based is called a 'Finite State Automaton'.
2. Next, we need to check if the tokens produced by the tokenizer come in a legal sequence. For instance, opening and closing parentheses need to come in matched pairs. This stage is called the syntactical analysis, and the software doing this is called a parser.

1. I will use the terms 'translating' and 'translater' as informal concepts that cover both compilers and interpreters and all sorts of mixed forms. This is not the place to get philosophical about the differences.

2 Very short introduction

A language is a set of words (strings) over an alphabet, that satisfies certain properties. It is also possible to define a language as the output of a certain type of grammar, or as the strings accepted by a certain type of automaton. We then need to prove the equivalences of the various formulations. In this section we briefly introduce the relevant concepts.

2.1 Languages

A language is a set of words that are constructed from an alphabet. The alphabet is finite in size, and words are finite in length, but languages can have an infinite number of words. The alphabet is often not specified explicitly.

Languages are often described with set notation and regular expressions, for example '$L = \{a^n b^* c^n | n > 0\}$', which says that the language is all strings of equal number of as and cs with an arbitrary number of bs in between.

Regular expressions are built up from the following ingredients:

$\alpha|\beta$ either the expression α or β
$\alpha\beta$ the expression α followed by the expression β
α^* zero or more occurrences of α
$\alpha+$ one or more occurrences of α
$\alpha?$ zero or one occurrences of α

We will see more complicated expressions in the *lex* utility.

2.2 Automata

A description of a language is not very constructive. To know how to generate a language we need a grammar. A grammar is a set of rules or productions $\alpha \rightarrow \beta$ that state that, in deriving a word in the language, the intermediate string α can be replaced by β. These strings can be a combination of

- A start symbol S,
- 'Terminal' symbols, which are letters from the alphabet; these are traditionally rendered with lowercase letters.
- 'Non-terminal' symbols, which are not in the alphabet, and which have to be replaced at some point in the derivation; these are traditionally rendered with uppercase letters.
- The empty symbol ϵ.

Languages can be categorized according to the types of rules in their grammar:

type 0 These are called 'recursive languages', and their grammar rules can be of any form: both the left and right side can have any combination of terminals, non-terminals, and ϵ.

type 1 'Context-sensitive languages' are limited in that ϵ can not appear in the left side of a production. A typical type 1 rule would look like

$$\alpha A \beta \rightarrow \gamma$$

which states that A, in the context of $\alpha A \beta$, is replaced by γ. Hence the name of this class of languages.

type 2 'Context-free languages' are limited in that the left side of a production can only consist of single non-terminal, as in $A \rightarrow \gamma$. This means that replacement of the non-terminal is done regardless of context; hence the name.

type 3 'Regular languages' can additionally have only a single non-terminal in each right-hand side.

In the context of grammars, we use the notation $\alpha \Rightarrow \beta$ to indicate that the string β as derived from α by a single application of a grammar rule; $\alpha \Rightarrow^* \beta$ indicates multiple rules. For example, $\alpha A \beta \Rightarrow \alpha B \gamma$ indicates that the rhs string was derived from the lhs by replacing $A\beta$ with $B\gamma$.

2.3 Automata

Corresponding to these four types of formal languages, there are four types of 'automata': formal machines that can recognize these languages. All these machines have a starting state, they go from one state to another depending on the input symbols they encounter, and if they reach the end state, the string is accepted as being in the language. The difference between the different types of automata lies in the amount of memory they have to store information. Very briefly the classes of automaton are:

for type 3 Finite State Automata. These machines have no memory. They can only make transitions.

for type 2 Pushdown Automata. These machines have a stack where they can store information; only the top of the stack can be inspected.

for type 1 Linear Bounded Automata. These have random-access memory, the size of which is equal to (a linear function of) the size of the input.

for type 0 Turing machines. These have an unbounded tape for storing intermediate calculations.

Lexical analysis.

The lexical analysis phase of program translation takes in a stream of characters and outputs a stream of tokens.

A token is a way of recognizing that certain characters belong together, and form an object that we can classify somehow. In some cases all that is necessary is knowing the class, for instance if the class has only one member. However, in general a token is a pair consisting of *its type and its value*. For instance, in `1/234` the lexical analysis recognizes that `234` is a number, with the value 234. In an assignment `abc = 456`, the characters `abc` are recognized as a variable. In this case the value is not the numeric value, but rather something like the index of where this variable is stored in an internal table.

Lexical analysis is relatively simple; it is performed by software that uses the theory of Finite State Automata and Regular Languages; see section 3.

Remark. It might be tempting to consider the input stream to consist of lines, each of which consist of characters, but this does not always make sense. Programming languages

such as Fortran do look at the source, one line at a time; C does not. TEX is even more complicated: the interpretation of the line end is programmable.[2]

3 Finite state automata and regular languages

Regular languages are the strings accepted by a particularly simple kind of automaton. However, we initially define these languages – non-constructively – by so-called 'regular expressions'.

3.1 Definition of regular languages

A regular language over some alphabet can be described by a 'regular expression'.

- ϵ denotes the empty language: the language with no words in it.
- If `a` is a letter in the alphabet, then `a` denotes the language $\{\texttt{a}\}$.
- If α and β are expressions denoting regular languages A and B, then
  - $\alpha\beta$ or $\alpha \cdot \beta$ denotes the language $\{xy | x \in A, y \in B\}$.
  - $\alpha|\beta$ denotes the language $A \cup B$.
  - α^* denotes the language $\cup_{n \geq 0} A^n$.
- Parentheses can be used to indicate grouping: (α) simply denotes the language A.

Any regular expression built up this way describes a regular language.

3.2 Non-deterministic automata

A Finite State Automaton is an abstract machine that recognizes ('accepts') words from a language:

- The automaton is initially in a beginning state;
- every letter or 'symbol' from the input word causes unambiguously a transition to the same or to a next state; if no transition is defined for a given combination of current state and input symbol, then the word is not in the language;
- a word is accepted if the last symbol causes a transition to a state that is marked as an accepting state.

Formally, we can define a FSA as the combination of

- A set S of states, with a starting state S_0 and a set of final states.
- A finite input alphabet I.
- A transition diagram $I \times S \rightarrow S$ that specifies how the combination of a state and an input symbol effects a transition to a new state.

This kind of automaton is deterministic in the sense that every transition from one state to the next is deterministically made by accepting an input symbol. However, in the context of lexical analysis, the so-called 'non-deterministic FSA' is more convenient. A non-deterministic FSA (also NFA) differs in two ways from the deterministic type:

---

2. Ok, if we want to be precise, TEX does look at the input source on a line-by-line basis. There is something of a preprocessor *before* the lexical analysis which throws away the machine-dependent line end, and replaces it with the TEX-defined one.

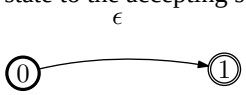
- An NFA can make spontaneous transitions from one state to another. If an automaton has such a transition, we can say that this is caused by the symbol ϵ, and this is called an ϵ-transition.
- An NFA can be ambiguous in that there can be more than one possible transition for a given state and input symbol.

  Exercise 1. Show that the second condition in the definition of an NFA can be reduced to the first. Is a reduction the other way possible?

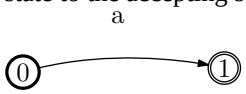
3.3 The NFA of a given language

We now construct a nondeterministic automaton that accepts a regular language.

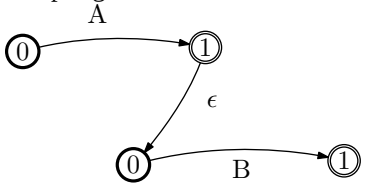
- The automaton that accepts the expression ϵ has a single transition from the starting state to the accepting state.



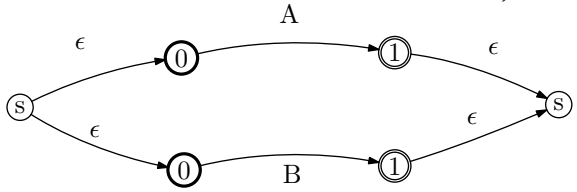
- The automaton that accepts the expression a has a single transition from the starting state to the accepting state.



- If **A** and **B** are automata accepting the languages A and B with expressions α and β, then
  - the language AB is accepted by the automaton that has the states and transition of both automata combined, with the initial state of **A** as the new initial state, the accepting state of **B** as the new accepting state, and an ϵ-transition from the accepting state of **A** to the initial state of **B**;



  - the language $A \cup B$ is accepted by an automaton with a new starting state that has ϵ-transitions to the initial states of **A** and **B**;



  - the expression α^* is accepted by **A** modified such that the initial state is also the accepting state, or equivalently by adding an ϵ-transition from the starting to the accepting state, and one the other way around.

3.4 Examples and characterization

Any language that can be described by the above constructs of repetition, grouping, concatenation, and choice, is a regular language. It is only slightly harder to take a transition diagram and write up the regular expression for the language that it accepts.

An informal way of characterizing regular languages is to say that FSAs 'do not have memory'. That means that any language where parts of words are related, such as $\{a^n b^m | m \geq n\}$, can not be recognized by a FSA. Proof: suppose there is a recognizing FSA. When it first accepts a b, it can come from only a fixed number of states, so that limits the information it can carry with it.

We can give a slightly more rigorous proof if we first characterize regular languages:

Theorem 1 *Let L be a regular language, then there is an n so that all strings α in L longer than n can be written as $\alpha = uvw$, such that for any k $uv^k w$ is also in the language.*

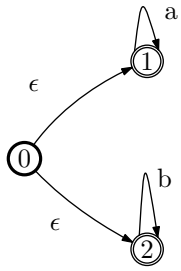
Using this theorem it is easy to see that the above language can not be regular.

This theorem is proved by observing that in order to accept a sufficiently long string the same state must have been encountered twice. The symbols accepted in between these encounters can then be repeated arbitrarily many times.

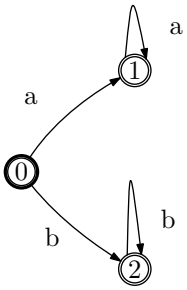
3.5 Deterministic automata

Non-deterministic automata, as defined above, are easy to define. However, from a practical point of view they do not look very constructive: a string in the language is accepted by the automaton if there is *any* sequence of transitions that accepts it. Fortunately, for every NFSA, there is a DFSA that accepts the same language.

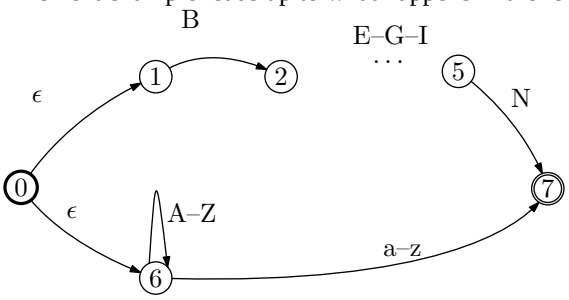
Sometimes it is easy to derive the DFSA. Consider the language $a^*|b^*$ and the automaton



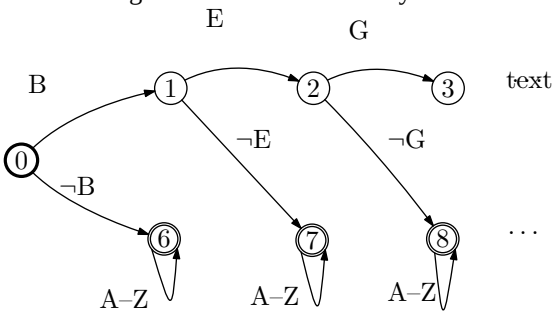
The following automaton is derived by splitting off one a and one b:



This next example leads up to what happens in the lexical analysis of a compiler:



The resulting DFA is a bit more messy:



(and we can collapse states $6 \ldots$ to one.)

Sketch of the proof: the states of the DFSA are sets of states of the NFSA. The states we are actually interested in are defined inductively, and they satisfy the property that they are closed under ϵ-transitions of the original NFSA. The starting state contains the original starting state plus everything reachable with ϵ-transitions from it. Given a state of the DFSA, we then define more states by considering all transitions from the states contained in this state: if there is a transition based on a symbol x, the next state has all states reachable from this state by accepting x, plus any subsequent ϵ-transitions.

Since the number of subsets of a finite set of states is finite, this will define a finite number of states for the DFSA, and it is not hard to see that an accepting sequence in the one automaton corresponds to an accepting sequence in the other.

3.6 Equivalences

Above, we saw how the NFA of a regular language is constructed. Does every NFA correspond to a regular language, and if so, how can that be derived? We make a detour by first talking about the equivalence of automata and grammars.

Let X be a string in the language L of a DFA, and suppose that after t transitions state i is reached. That means we can split $X = X_i(t)Y_i$. This is merely one of the strings that is in state i at time t; let us call the set of all these strings $L_i(t)$. Let us call the set of all strings that, given a state i, bring the automaton to an accepting state R_i. This set is clearly not dependent on t. Defining $L_i = \cup_{t=0}^{\infty} L_i(t)$, we have that $L = \cup_{i=1}^{m} L_i R_i$ where m is the number of states.

This inspires us to tackle the derivation of a grammar by describing the production of the remainder strings R_i. Suppose the automaton is in state i; we will derive the productions $N_i \rightarrow \ldots$. If state i is an accepting state, there will be a production $N_i \rightarrow \epsilon$; for all other transitions by a symbol x to a state $N_{i'}$ we add a production $N_i \rightarrow xN_{i'}$. It is easy to see the equivalence of strings accepted by the DFA and derivations of the grammar thus constructed.

Going the other way, constructing an automaton from a grammar runs into a snag. If there are productions $N_i \rightarrow aN_{i'}$ and $N_i \rightarrow aN_{i''}$, we can of necessity only construct an NFA. However, we know that these are equivalent to DFAs.

We note that the grammars used and constructed in this – informal – proof are right-recursive, so they generate precisely the regular languages.

> **Exercise 2.** Show how this proof can be modified to use left-recursive grammars, that is, grammars that have productions of the form $N_i \rightarrow N_{i'}a$.

4 Lexical analysis with FSAs

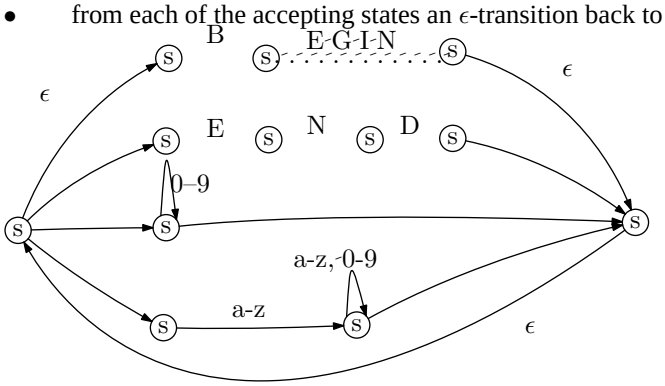
A FSA will recognize a sequence of language elements. However, it's not enough to simply say 'yes, this was a legal sequence of elements': we need to pass information on to the next stage of the translation. This can be done by having some executable code attached to the accepting state; if that state is reached, the code snippet tells the next stage what kind of element has been recognized, and its value. This value can be a numerical value for numbers recognized, but more generally it will be an index into some table or other.

Formally, we can extend the definition of a FSA (section 3.2) by the addition of an output alphabet O and an output table $I \times S \rightarrow O$. This models the output of a symbol, possibly ϵ, at each transition.

> **Exercise 3.** One could also define the output with a mapping $S \rightarrow O$. Show that the definitions are equivalent.

An FSA is not enough to recognize a whole language, but it can recognize elements from a language. For instance, we can build multiple FSAs for each of the keywords of a language ('begin' or 'void'), or for things like numbers and identifiers. We can then make one big FSA for all the language elements by combining the multiple small FSAs into one that has

- a starting state with ϵ-transitions to the start states of the element automata, and
- from each of the accepting states an ϵ-transition back to the start state.



Exercise 4. Write a DFA that can parse Fortran arithmetic expressions. In Fortran, exponentiation is written like `2**n`. It is also not allowed to have two operators in a row, so 2×-3 is notated `2*(-3)`.

There is a problem with the ϵ-transition from the final state to the initial state in the above NFA. This transition should only be taken if no other transitions can be taken, in other words, if the maximal string is recognized. For instance, most programming languages allow quote characters inside a quoted string by doubling them: ‘`"And then he said ""Boo!"""`’. The final state is reached three times in the course of this string; only the last time should the jump back be taken.

However, sometimes finding the maximum matched string is not the right strategy. For instance, in most languages, `4.E3` is a floating point number, but matching the `E` after the decimal point is not necessarily right. In Fortran, the statement `IF (4.EQ.VAR) ...` would then be misinterpreted. What is needed here is one token ‘look-ahead’: the parser needs to see what character follows the `E`.

At this point it would be a good idea to learn the Unix tool *lex*.

Syntax parsing.

Programming languages have for decades been described using formal grammars. One popular way of notating those grammars is Backus Naur Form, but most formalisms are pretty much interchangable. The essential point is that the grammars are almost invariably of the context-free type. That is, they have rules like

⟨`function call`⟩ ⟶ ⟨`function name`⟩ `(` ⟨`optargs`⟩ `)`
⟨`optargs`⟩ ⟶ empty | ⟨`args`⟩
⟨`args`⟩ ⟶ word | word `,` ⟨`args`⟩

The second and third rule in this example can be generated by a regular grammar, but the first rule is different: when the opening parenthesis is matched, the parser has to wait an unlimited time for the closing parenthesis. This rule is of context-free type.

It is important to keep some distinctions straight:

- A grammar has a set of rules, each indicating possible replacements during a derivation of a string in the language. Each rule looks like $A \to \alpha$.
- A derivation is a specific sequence of applications of rules; we denote each step in a derivation as $\alpha \Rightarrow \beta$, where β can be derived from α by application of some rule. The derivation of some string α is a sequence of step such that $S \Rightarrow \cdots \Rightarrow \alpha$; we abbreviate this as $S \Rightarrow^* \alpha$.
- Ultimately, we are interested in the reverse of a derivation: we have a string that we suspect is in the language, and we want to reconstruct whether and how it could be derived. This reconstruction process is called 'parsing', and the result often takes the form of a 'parse tree'.

We will first give some properties of context-free languages, then in section 6 we will discuss the practical parsing of context-free languages.

5 Context-free languages

Context-free languages can be defined as the output of a particular kind of grammar (the left side can only consist of a single nonterminal), or as the set of string accepted by a certain kind of automaton. For languages of this type, we use a Pushdown Automaton (PDA) to recognize them. A PDA is a finite-state automaton, with some scratch memory that takes the form of a stack: one can only push items on it, and inspect or remove the top item. Here we will not give an equivalence proof.

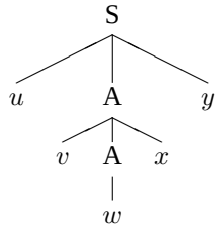
An example of a language that is context-free but not regular is $\{a^n b^n\}$. To parse this, the automaton pushes as on the stack, then pops them when it finds a b in the input, and the string is accepted if the stack is empty when the input string is fully read.

5.1 Pumping lemma

As with regular languages (section 3.4), there is a way to characterize the strings of a context-free language.

Theorem 2 *Let L be a context-free language, then there is an n so that all strings α in L longer than n can be written as $\alpha = uvwxy$, such that for any k the string $uv^k wx^k y$ is also in the language.*

The proof is as before: the derivation of a sufficiently long string must have used the same production twice.



5.2 Deterministic and non-deterministic PDAs

As with Finite State Automata, there are deterministic and non-deterministic pushdown automata. However, in this case they are not equivalent. As before, any DPA is also a NPA, so any language accepted by a DPA is also accepted by a NPA. The question is whether there are languages that are accepted by a NPA, and that are not accepted by a DPA.

A similar example to the language $\{a^n b^n\}$ above is the language over an alphabet of at least two symbols $L = \{\alpha\alpha^R\}$, where α^R stands for the reverse of α. To recognize this language, the automaton pushes the string α on the stack, and pops it to match the reverse part. However, the problem is knowing when to start popping the stack.

Let the alphabet have at least three letters, then the language $L_c = \{\alpha c\alpha^R | c \notin \alpha\}$ can deterministically be recognized. However, in absence of the middle symbol, the automaton needs an ϵ-transition to know when to start popping the stack.

5.3 Normal form

Context-free grammars have rules of the form $A \rightarrow \alpha$ with A a single nonterminal and α any combination of terminals and nonterminals. However, for purposes of parsing it is convenient to have the rules in a 'normal form'. For context-free grammars that is the form $A \rightarrow a\alpha$ where a is a terminal symbol.

One proof that grammars can always be rewritten this way uses 'expression equations'. If $\mathbf{x}$ and $\mathbf{y}$ stand for sets of expressions, then $\mathbf{x} + \mathbf{y}$, $\mathbf{xy}$, and $\mathbf{x}^*$ stand for union, concatenation, and repetition respectively.

Consider an example of expression equations. The scalar equation $\mathbf{x} = \mathbf{a} + \mathbf{xb}$ states that $\mathbf{x}$ contains the expressions in $\mathbf{a}$. But then it also contains $\mathbf{ab}$, $\mathbf{abb}$, et cetera. One can verify that $\mathbf{x} = \mathbf{ab}^*$.

The equation in this example had a regular language as solution; the expression $\mathbf{x} = \mathbf{a} + \mathbf{bxc}$ does not have a regular solution.

Now let $\mathbf{x}$ be a vector of all non-terminals in the grammar of a context-free language, and let $\mathbf{f}$ be the vector of righthandsides of rules in the grammar that are of normal form. We can then write the grammar as

$$\mathbf{x}^t = \mathbf{x}^t\mathbf{A} + \mathbf{f}^t$$

where the multiplication with $\mathbf{A}$ describes all rules not of normal form.

Example:

$$\begin{array}{lcl} S & \rightarrow & aSb|XY|c \\ X & \rightarrow & YXc|b \\ Y & \rightarrow & XS \end{array} \quad \Rightarrow \quad [S,X,Y] = [S,X,Y] \begin{bmatrix} \phi & \phi & \phi \\ Y & \phi & S \\ \phi & Xc & \phi \end{bmatrix} + [aSb + c, b, \phi]$$

The solution to this equation is

$$\mathbf{x}^t = \mathbf{f}^t\mathbf{A}^*$$

which describes rules on normal form. However, we need to find a more explicit expression for $\mathbf{A}^*$.

Noting that $\mathbf{A}^* = \lambda + \mathbf{A}\mathbf{A}^*$ we get

$$\mathbf{x}^t = \mathbf{f}^t + \mathbf{f}^t\mathbf{A}\mathbf{A}^* = \mathbf{f}^t + \mathbf{f}^t\mathbf{B} \tag{1}$$

where $\mathbf{B} = \mathbf{A}\mathbf{A}^*$. This is a grammar on normal form. It remains to work out the rules for $\mathbf{B}$. We have

$$\mathbf{B} = \mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{A}\mathbf{A}^* = \mathbf{A} + \mathbf{A}\mathbf{B}$$

These rules need not be of normal form. However, any elements of $\mathbf{A}$ that start with a nonterminal, can only start with nonterminals in $\mathbf{x}$. Hence we can substitute a rule from equation (1).

6 Parsing context-free languages

The problem of parsing is this:

> Given a grammar G and a string α, determine whether the string is in the language of G, and through what sequence of rule applications it was derived.

We will discuss the LL and LR type parser, which correspond to a top-down and bottom-up way of parsing respectively, then go into the problem of ambiguity

6.1 Top-down parsing: LL

One easy parsing strategy starts from the fact that the expression has to come from the start symbol. Consider the expression `2*5+3`, which is produced by the grammar

Expr $\longrightarrow$ number Tail
Tail $\longrightarrow$ ϵ | + number Tail | * number Tail

In the following analysis the stack has its bottom at the right

initial queue:	$2 * 5 + 3$	
start symbol on stack:		Expr
replace		number Tail
match	$* 5 + 3$	Tail
replace		* number Tail
match	$5 + 3$	number Tail
match	$+ 3$	Tail
replace		+ number Tail
match	3	number Tail
match	ϵ	Tail
match		

The derivation that we constructed here is

$$E \Rightarrow nT \Rightarrow n * nT \Rightarrow n * n + nT \Rightarrow n * n + n$$

that is, we are replacing symbols from the left. Therefore this kind of parsing is called LL parsing: read from left to right, replace from left to right. Because we only need to look at the first symbol in the queue to do the replacement, without need for further 'look ahead' tokens, this is $LL(1)$ parsing.

But this grammar was a bit strange. Normally we would write

Expr $\longrightarrow$ number | number + Expr | number * Expr

If our parser can now see the first *two* symbols in the queue, it can form

initial queue:	$2 * 5 + 3$	
start symbol on stack:		Expr
replace		number * Expr
match	$5 + 3$	Tail
replace		number + Expr
match	3	Expr
replace	3	number
match	ϵ	

This is called $LL(2)$ parsing: we need one token look ahead.

6.1.1 Problems with LL parsing

If our grammar had been written

Expr $\longrightarrow$ number | Expr + number | Expr * number

an $LL(k)$ parser, no matter the value of k, would have gone into an infinite loop.

In another way too, there are many constructs that can not be parsed with an $LL(k)$ parser for any k. For instance if both `A<B` and `A<B>` are legal expressions, where `B` can be of arbitrary length, then no finite amount of look-ahead will allow this to be parsed.

6.1.2 LL and recursive descent

The advantages of $LL(k)$ parsers are their simplicity. To see which rule applies at a given point is a recursive-descent search, which is easily implemented. The code for finding which rule to apply can broadly be sketched as follows:

```
define FindIn(Sym,NonTerm)
  for all expansions of NonTerm:
    if leftmost symbol == Sym
      then found
    else if leftmost symbol is nonterminal
      then FindIn(Sym,that leftmost symbol)
```

This implies that a grammar is LL-parsable if distinct rules for some non-terminal can not lead to different terminals. In other words, by looking at a terminal, it should be clear what production was used.

The LR parsers we will study next are more powerful, but much more complicated to program. The above problems with $LL(k)$ are largely non-existent in languages where statements start with unique keywords.

6.2 Bottom-up parsing: shift-reduce

In this section we will look at the 'bottom-up' parsing strategy, where terminal symbols are gradually replaced by non-terminals.

One easily implemented bottom-up parsing strategy is called 'shift-reduce parsing'. The basic idea here is to move symbols from the input queue to a stack, and every time the symbols on top of the stack form a right hand size of a production, reduce them to the left hand side.

For example, consider the grammar

E $\longrightarrow$ number | E + E | E * E

and the expression $2*5+3$. We proceed by moving symbols from the left side of the queue to the top of the stack, which is now to the right.

	stack	queue
initial state:		$2*5+3$
shift	2	*5+3
reduce	E	*5+3
shift	E*	5+3
shift	E*5	+3
reduce	E*E	+3
reduce	E	+3
shift, shift, reduce	E+E	
reduce	E	

(Can you tell that we have ignored something important here?)

The derivation we have reconstructed here is

$$E \Rightarrow E+E \Rightarrow E+3 \Rightarrow E*E+3 \Rightarrow E*5+3 \Rightarrow 2*5+3$$

which proceeds by each time replacing the right-most nonterminal. This is therefore called a 'rightmost derivation'. Analogously we can define a 'leftmost derivation' as one that proceeds by replacing the leftmost nonterminal.

For a formal definition of shift-reduce parsing, we should also define an 'accept' and 'error' action.

6.3 Handles

Finding the derivation of a legal string is not trivial. Sometimes we have a choice between shifting and reducing, and reducing 'as soon as possible' may not be the right solution. Consider the grammar

S $\longrightarrow$ aAcBe
A $\longrightarrow$ bA | b
B $\longrightarrow$ d

and the string `abbcde`. This string can be derived (writing the derivation backwards for a change) as

`abbcde` $\Leftarrow$ `abAcde` $\Leftarrow$ `aAcde` $\Leftarrow$ `aAcBe` $\Leftarrow$ `S`.

However, if we had started

`abbcde` $\Leftarrow$ `aAbcde` $\Leftarrow aAAcde \Leftarrow ?$

we would be stuck because no further reductions would be applicable.

The problem then is to know where to start replacing terminal symbols and, later in the derivation, non-terminals. The shift-reduce strategy of the previous section is here seen to lead to problems, so some extra power is needed. We introduce the concept of 'handle' as a formal definition of 'the right production and place to start reducing'. The following definition is totally unhelpful:

> If $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta w$ is a right-most derivation, then $A \rightarrow \beta$ at the position after α is a handle of $\alpha A w$.

Clearly, if we can identify handles, we can derive a parse tree for a given string. However, the story so far has not been constructive. Next we will look at ways of actually finding handles.

6.4 Operator-precedence grammars

It is easy to find handles if a grammar is of an 'operator grammar' form. Loosely, by this we mean that expressions in the language look like expression-operator-expression. More strictly, we look at grammars where there are never two adjacent nonterminals, and where no right hand side is ϵ. We also assume that precedence relations between operators and terminals are known.

Let us look again at arithmetic expressions; we will introduce relations $\text{op}_1 \lessdot \text{op}_2$ if the first operator has lower precedence, and $\text{op}_1 \gtrdot \text{op}_2$ if it has higher precedence. If the two operators are the same, we use predence to force associativity rules. For instance, right associativity corresponds to definitions such as $+ \gtrdot +$.

For the $+$ and $*$ operators we then have the following table:

	number	$+$	$\times$
number		$\gtrdot$	$\gtrdot$
$+$	$\lessdot$	$\gtrdot$	$\lessdot$
$\times$	$\lessdot$	$\gtrdot$	$\gtrdot$

Now we can find a handle by scanning left-to-right for the first $\gtrdot$ character, then scanning back for the matching $\lessdot$. After reducing handles thus found, we have a string of operators and nonterminals. Ignoring the nonterminals, we insert again the comparisons; this allows us to find handles again.

For example, $5 + 2 * 3$ becomes $\lessdot 5 \gtrdot + \lessdot 2 \gtrdot * \lessdot 3 \gtrdot$; replacing handles this becomes $E + E * E$. Without the nonterminals, the precedence structure is $\lessdot + \lessdot * \gtrdot$, in which we find $\lessdot E * E \gtrdot$ as the handle. Reducing this leaves us with $E + E$, and we find that we have parsed the string correctly.

This description sounds as if the whole expression is repeatedly scanned to insert precedence relations and find/reduce handle. This is not true, since we only need to scan as far as the right edge of the first handle. Thus, a shift/reduce strategy will still work for operator grammars.

6.5 LR parsers

We will now consider LR parsers in more detail. These are the parsers that scan the input from the left, and construct a rightmost derivation, as in the examples we have seen in section 6.2. Most constructs in programming languages can be parsed in an LR fashion.

An LR parser has the following components

- A stack and an input queue as in the shift-reduce examples you have already seen in section 6.2. The difference is that we now also push state symbols on the stack.
- Actions 'shift', 'reduce', 'accept', 'error', again as before.
- An `Action` and `Goto` function that work as follows:
  - Suppose the current input symbol is a and the state on top of the stack is s.
  - If `Action`(a, s) is 'shift', then a and a new state $s' =$ `Goto`(a, s) are pushed on the stack.
  - If `Action`(a, s) is 'reduce $A \rightarrow \beta$' where $|\beta| = r$, then $2r$ symbols are popped from the stack, a new state $s' =$ `Goto`(a, s'') is computed based on the newly exposed state on the top of the stack, and A and s' are pushed. The input symbol a stays in the queue.

An LR parser that looks at the first k tokens in the queue is called an LR(k) parser. We will not discuss this issue of look-ahead any further.

It is clear that LR parser are more powerful than a simple shift-reduce parser. The latter has to reduce when the top of the stack is the right hand side of a production; an LR parser additionally has states that indicate whether and when the top of the stack is a handle.

6.5.1 A simple example of LR parsing

It is instructive to see how LR parsers can deal with cases for which simple shift/reduce parsing is insufficient. Consider again the grammar

E $\longrightarrow$ E + E | E * E

and the input string $1 + 2 * 3 + 4$. Give the `+` operator precedence 1, and the `*` operator precedence 2. In addition to moving tokens onto the stack, we also push the highest precedence seen so far. In the beginning we declare precedence 0, and pushing a non-operator does not change the precedence.

Shift/reduce conflicts are now resolved with this rule: if we encounter at the front of the queue a lower precedence than the value on top of the stack, we reduce the elements on top of the stack.

	$1 + 2 * 3 + 4$	push symbol; highest precedence is 0
1 S_0	$+2 * 3 + 4$	highest precedence now becomes 1
1 S_0 + S_1	$2 * 3 + 4$	
1 S_0 + S_1 2 S_1	$*3 + 4$	highest precedence becoming 2
1 S_0 + S_1 2 S_1 * S_2	$3 + 4$	
1 S_0 + S_1 2 S_1 * S_2 3 S_2	$+4$	reduce because `P(+)` < 2
1 S_0 + S_1 6 S_1	$+4$	the highest exposed precedence is 1
1 S_0 + S_1 6 S_1 + S_1	4	
1 S_0 + S_1 6 S_1 + S_1 4 S_1		at the end of the queue we reduce

1 S_0 + S_1 10 S_1
11

Even though this example is phrased very informally, we see the key points:

- only the top of the stack and the front of the queue are inspected;
- we have a finite set of rules governing shift/reduce behaviour.

As we shall see, this mechanism can also identify handles.

6.5.2 States of an LR parser

An LR parser is constructed automatically from the grammar. Its states are somewhat complicated, and to explain them we need a couple of auxiliary constructs.

item An 'item' is a grammar rule with a location indicated. From the rule `A` $\rightarrow$ `B C` we get the items `A` $\rightarrow$ $\bullet$`B C`, `A` $\rightarrow$ `B` $\bullet$`C`, `A` $\rightarrow$ `B C`$\bullet$. The interpretation of an item will be that the symbols left of the dot are on the stack, while the right ones are still in the queue. This way, an item describes a stage of the parsing process.

closure The closure of an item is defined as the smallest set that

- Contains that item;
- If the closure contains an item `A` $\rightarrow \alpha$ $\bullet$`B` β with `B` a nonterminal symbol, then it contains all items `B` $\rightarrow \bullet\gamma$. This is a recursive notion: if γ starts with a non-terminal, the closure would also contain the items from the rules of γ.

The states of our LR parser will now be closures of items of the grammar. We motivate this by an example.

Consider now an item `A` $\rightarrow \beta_1 \bullet \beta_2$ in the case that we have recognized $\alpha\beta_1$ so far. The item is called *valid* for that string, if a rightmost derivation $S \Rightarrow^* \alpha A w \Rightarrow \alpha\beta_1\beta_2 w$ exists. If $\beta_2 = \epsilon$, then $A \rightarrow \beta_1$ is a handle and we can reduce. On the other hand, if $\beta_2 \neq \epsilon$, we have not encountered the full handle yet, so we shift β_2.

As an example, take the grammar

E $\longrightarrow$ E+T | T
T $\longrightarrow$ T*F | F
F $\longrightarrow$ (E) | id

and consider the partially parsed string `E+T*`. The (rightmost) derivation

$$E \Rightarrow E + T \Rightarrow E + T * F$$

shows that `T` $\rightarrow$ `T*`$\bullet$`F` is a valid item,

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * (E)$$

gives `F` $\rightarrow \bullet$`(E)` as a valid item, and

$$E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * \mathrm{id}$$

gives `F` $\rightarrow \bullet$`id` as a valid item.

6.5.3 States and transitions

We now construct the actual states of our parser.

- We add a new start symbol S′, and a production $\mathtt{S}' \rightarrow \mathtt{S}$.
- The starting state is the closure of $\mathtt{S}' \rightarrow \bullet\mathtt{S}$.
- The transition function $d(s, \mathtt{X})$ of a state s and a symbol X is defined as the closure of
  $$\{\mathtt{A} \rightarrow \alpha\ \mathtt{X}\bullet\ \beta | \mathtt{A} \rightarrow \alpha\ \bullet\mathtt{X}\ \beta \text{ is in } s\}$$
- The 'follow' of a symbol A is the set of all terminal symbols that can follow its possible expansions. This set is easy to derive from a grammar.

Here is an example

We construct the states and transition for the grammar

S $\longrightarrow$ (S)S | ϵ

which consists of all strings of properly matched left and right parentheses.

Solution: we add the production $\mathtt{S}' \rightarrow \bullet\mathtt{S}$. We now find the states

1. $\{\mathtt{S}' \rightarrow \bullet\mathtt{S}, \mathtt{S} \rightarrow \bullet\mathtt{(S)S}, \mathtt{S} \rightarrow \bullet\}$
2. $\{\mathtt{S}' \rightarrow \mathtt{S}\bullet\}$
3. $\{\mathtt{S} \rightarrow \mathtt{(}\bullet\mathtt{S)S}, \mathtt{S} \rightarrow \bullet\mathtt{(S)S}, \mathtt{S} \rightarrow \bullet\}$
4. $\{\mathtt{S} \rightarrow \mathtt{(S}\bullet\mathtt{)S}\}$
5. $\{\mathtt{S} \rightarrow \mathtt{(S)}\bullet\mathtt{S}, \mathtt{S} \rightarrow \bullet\mathtt{(S)S}, \mathtt{S} \rightarrow \bullet\}$
6. $\{\mathtt{S} \rightarrow \mathtt{(S)S}\bullet\}$

with transitions

$$\begin{aligned} d(0, S) &= 1 \\ d(0,'(') &= 2 \\ d(2, S) &= 3 \\ d(2,'(') &= 2 \\ d(3,')') &= 4 \\ d(4, S) &= 5 \\ d(4,'(') &= 2 \end{aligned}$$

The only thing missing in our parser is the function that describes the stack handling. The parsing stack consists of states and grammar symbols (alternating). Initially, push the start state onto the stack. The current state is always the state on the top of the stack. Also, add a special endmarker symbol to the end of the input string.

Loop:

(1) **if** the current state contains $\mathtt{S}' \rightarrow \mathtt{S}\bullet$
  accept the string

(2) **else if** the current state contains any other final item $\mathtt{A} \rightarrow \alpha\bullet$
  pop all the tokens in α from the stack, along with the corresponding states;
  let s be the state left on top of the stack: push A, push d(s,A)

(3) **else if** the current state contains any item $\mathtt{A} \rightarrow \alpha\ \bullet\mathtt{x}\ \beta$,
  where x is the next input token
  let s be the state on top of the stack: push x, push d(s,x)

else report failure

Explanation:

1. If we have recognized the initial production, the bottom-up parse process was successful.
2. If we have a string of terminals on the stack, that is the right hand side of a production, replace by the left hand side non-terminal.
3. If we have a string of terminals on the stack that is the *start* of a right hand side, we push the current input symbol.

Exercise 5. Give the states and transitions for the grammar

S $\longrightarrow$ x
S $\longrightarrow$ (L)
L $\longrightarrow$ S
L $\longrightarrow$ L S

Apply the above parsing algorithm to the string `(x,x,(x))`.

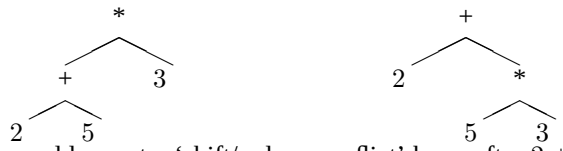
The parsers derived by the above algorithm can only handle cases where there is no ambiguity in condition (3). The class of grammars recognized by this type of parser is called $LR(0)$ and it is not very interesting. We get the more interesting class of $SLR(1)$ by adding to condition (2) the clause that the following symbol is in the follow of A. Another similar class, which is the one recognized by *yacc*, is $LALR(1)$.

6.6 Ambiguity and conflicts

The problem of finding out *how* a string was derived is often important. For instance, with a grammar

$\langle$`expr`$\rangle \longrightarrow \langle$`number`$\rangle \mid \langle$`expr`$\rangle$ + $\langle$`expr`$\rangle \mid \langle$`expr`$\rangle \times \langle$`expr`$\rangle$

the expression $2 + 5 * 3$ is ambiguous: it can mean either $(2 + 5) * 3$ or $2 + (5 * 3)$.



An LR parser would report a 'shift/reduce conflict' here: after $2 + 5$ has been reduced to `<expr> + <expr>`, do we reduce that further to `<expr>`, or do we shift the minus, since `<expr> -` is the start of a legitimate reducible sequence?

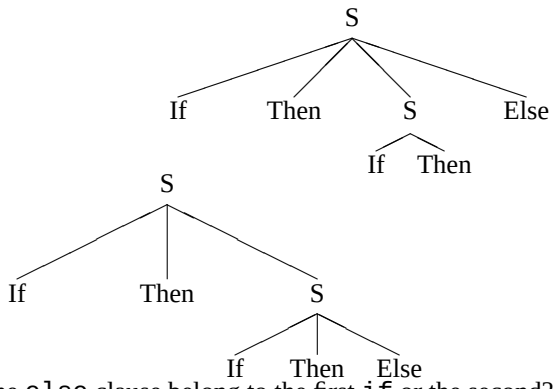
Another example of ambiguity is the 'dangling else' problem. Consider the grammar

$\langle$`statement`$\rangle \longrightarrow$ if $\langle$`clause`$\rangle$ then $\langle$`statement`$\rangle$
$\mid$ if $\langle$`clause`$\rangle$ then $\langle$`statement`$\rangle$ else $\langle$`statement`$\rangle$

and the string

`if` c_1 `then if` c_2 `then` s_1 `else` s_2

This can be parsed two ways:



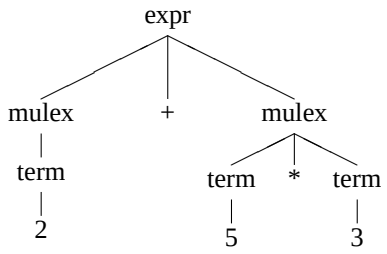
Does the `else` clause belong to the first `if` or the second?

Let us investigate the first example. We can solve the ambiguity problem in two ways:

- Reformulate the grammar as

  $\langle$`expr`$\rangle \longrightarrow \langle$`mulex`$\rangle \mid \langle$`mulex`$\rangle + \langle$`mulex`$\rangle$
  $\langle$`mulex`$\rangle \longrightarrow \langle$`term`$\rangle \mid \langle$`term`$\rangle \times \langle$`term`$\rangle$
  $\langle$`term`$\rangle \longrightarrow$ number

  so that the parser can unambiguously reconstruct the derivation,



  or

- Teach the parser about precedence of operators. This second option may be easier to implement if the number of operators is large: the first option would require a large number of rules, with probably a slower parser.

  Exercise 6. Rewrite the grammar of the second example to eliminate the dangling else problem.

Since we are not used to thinking of keywords such as `then` in terms of precedence, it is a better solution to eliminate the dangling else problem by introducing a `fi` keyword to close the conditional. Often, however, ambiguity is not so easy to eliminate.

Exercise 7. In case of a shift-reduce conflict, yacc shifts. Write an example that proves this. Show what this strategy implies for the dangling else problem.

Another type of conflict is the ‘reduce/reduce conflict’. Consider this grammar:

A $\longrightarrow$ B c d | E c f
B $\longrightarrow$ x y
E $\longrightarrow$ x y

and the input string that starts `x y c`.

- An $LR(1)$ parser will shift `x y`, but can not decide whether to reduce that to `B` or `E` on the basis of the look-ahead token `c`.
- An $LR(2)$ parser can see the subsequent `d` or `f` and make the right decision.
- An LL parser would also be confused, but already at the `x`. Up to three tokens (`x y c`) is unsufficient, but an $LL(4)$ parser can again see the subsequent `d` or `f`.

The following grammar would confuse any $LR(n)$ or $LL(n)$ parser with a fixed amount of look-ahead:

A $\longrightarrow$ B C d | E C f
B $\longrightarrow$ x y
E $\longrightarrow$ x y
C $\longrightarrow$ c | C c

which generates $x\,y\,c^n\,\{d|f\}$.

As usual, the simplest solution is to rewrite the grammar to remove the confusion e.g.:

A $\longrightarrow$ BorE c d | BorE c f
BorE $\longrightarrow$ x y

or assuming we left-factorise the grammar for an $LL(n)$ parser:

A $\longrightarrow$ BorE c tail
tail $\longrightarrow$ d | f
BorE $\longrightarrow$ x y

Another example of a construct that is not LR parsable, consider languages such as Fortran, where function calls and array indexing both look like `A(B,C)`:

⟨`expression`⟩ $\longrightarrow$ ⟨`function call`⟩ | ⟨`array element`⟩
⟨`function call`⟩ $\longrightarrow$ name `(` ⟨`parameter list`⟩ `)`
⟨`array element`⟩ $\longrightarrow$ name `(` ⟨`expression list`⟩ `)`
⟨`parameter list`⟩ $\longrightarrow$ name | name `,` ⟨`parameter list`⟩
⟨`expression list`⟩ $\longrightarrow$ name | name, ⟨`expression list`⟩

After we push `B` on the stack, it is not clear whether to reduce it to the head of a parameter list or of an expression list, and no amount of lookahead will help. This problem can be solved by letting the lexical analyzer have access to the symbol table, so that it can distinguish between function names and array names.

Contents