Beyond Regular Languages

Some languages are not regular. Here are three examples. I hope you can extrapolate the intended language from the example strings

- $\epsilon, 01, 0011, 000111, 00001111, \cdots$
- $\epsilon, (), [], (()), ([]), [[]], ((())), \cdots$
- $123, 12 + 3, 42 \times 2, (1 + 2) \times x, 2 \times y + 3 \times z, \cdots$

Let's prove that the first one is not regular. The proof is by contradiction: we'll assume that the language is regular and derive a contradiction.

Proof that $L = \{\epsilon, 01, 0011, 000111, 00001111, \cdots\}$ is not regular

(0) Assume that L is regular.

(1) From (0), there is a DFR that describes the language.

(2) Let A be any such DFR. $L(A) = L$

(3) For each $i \in \mathbb{N}$, let $f(i)$ be the state that A is in after processing the string "0"^{i}.

I.e., $f(0) = q_0$ and, for $i > 0$, $\{f(i)\} = \delta(f(i-1),$'0'$)$.

(4) Let n be the number of states in A.

(5) Consider the sequence $f(0), f(1), ..., f(n)$ of length $n + 1$.

(6) Since there are only n states in A, at least one state in this sequence must be repeated.

(7) So we can let i and k be such that $i \neq k$ and $f(i) = f(k)$.

(8) Since "0"^{i}^"1"^{i} is in L, "1"$^i \in L(f(i))$.

(9) "1"$^i \in L(f(k))$ (By (7) and (8))

(10) Consider the string $s =$ "0"^{k}^"1"^{i}.

(11) After processing the k 0s, A will be in state $f(k)$.

(12) From (11) and (9), A will accept "0"^{k}^"1"^{i}.

(13) But since $k \neq i$, s is not in L.

(14) By (12) and (13), the language of A is not L.

(15) We have a contradiction between (2) and (14)

Conclusion. Assumption 0 must be untrue.

End of Proof.

Grammars and Parsing

A formal language is simply a set of sequences.

Usually we restrict ourselves to *possibly infinite* sets of *finite sequences* over a *finite set* S.

Formal language theory considers *finite* descriptions of languages.

We are particularly interested in description methods that are

- easy to understand and use
- lead to algorithms for analyzing sequences
- suitable for automated processing

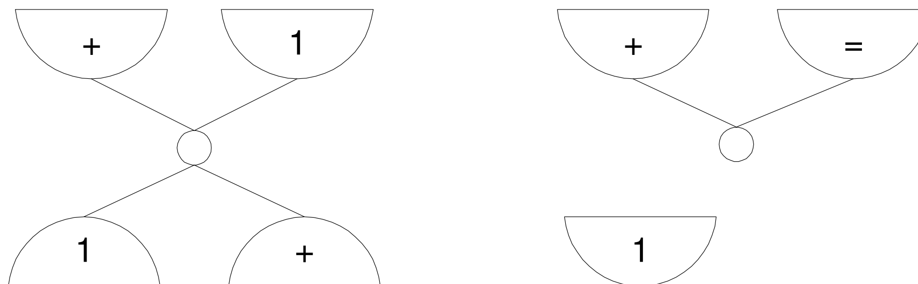
Finite recognizers meet these criteria, but there are many important languages that can not be described by finite recognizers because they (the languages) require more than a fixed amount of memory.

Grammars

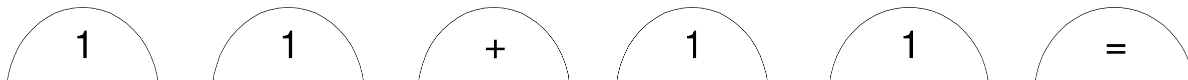
Unrestricted Grammars

Game 1

We have an unlimited supply of puzzle pieces of each of 3 shapes



The puzzle starts with a sequence of pieces:



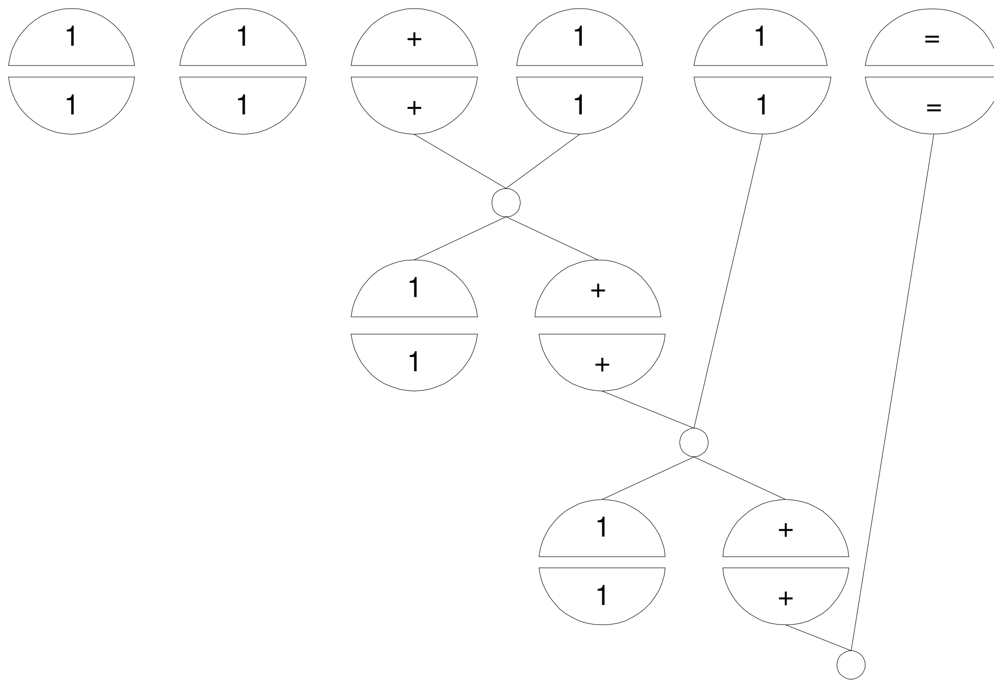
The idea is to close all the circles.

In each step we can add one pieces from our infinite supply.

We can stretch the lines vertically, but can not alter the sequence of symbols along their top or bottom and can not rotate the pieces.

We also can not cross lines.

In this case we get



Now looking at the sequence of closed circles (with no lines at the bottom), we get:

$$1111$$

Thus our puzzle pieces describe an algorithm for computing addition in tally notation.

In general, a set of puzzle pieces defines a function from finite sequences to finite sequences.

Proposition 1 *Any function from strings to strings that can be computed by an algorithm can be turned into a game like this.*

We call such a function a “computable function”.

Challenge: find a game that will add numbers presented in binary notation. E.g. with input

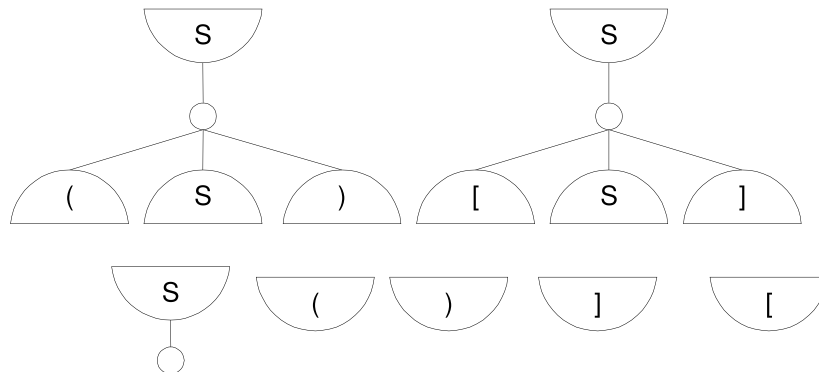
$$1100 + 111 =$$

the output will be 10011.

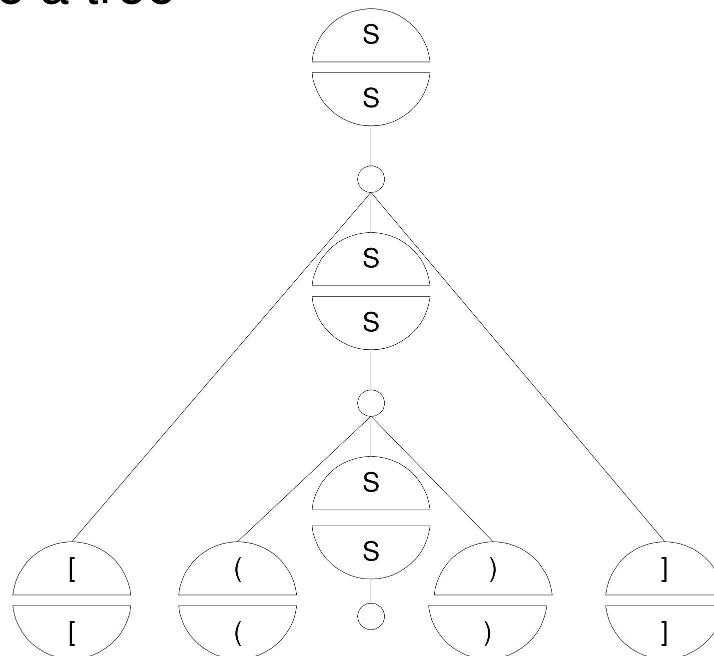
Challenge: How can you multiply tally notation? An input of $111 \times 11 =$ should result in an output of 111111.

Game 2

For this game we start always start with an symbol S and the 7 piece kinds are



We can make a tree



The final sequence is $[()]$ other sequences we can reach are $[[[(([]))]]]$ and the empty sequence.

This game defines an infinite set of finite sequences over the *alphabet* $\{$‘(’, ‘)’, ‘[’, ‘]’$\}$.

Proposition 2 *Any language that can be generated by an algorithm can be defined by a game like this. (I.e., the algorithm produces a possibly infinite list containing each member of the language.)*

We call set a language a “recursively enumerable language”.

Handier Notation

To save space, we will use a more compact notation.

Game 1 again

We define a finite set of "terminal symbols" $S = \{1\}$.

We define a finite set of "nonterminal symbols" (a.k.a. "variables") $V = \{\text{‘}+\text{’}, \text{‘}=\text{’}\}$.

We define a finite set of "production rules"

$$P = \{ \text{‘}+\text{’} = \longrightarrow \epsilon, \\ \text{‘}+\text{’}\,\text{‘}1\text{’} \longrightarrow \text{‘}1\text{’}\,\text{‘}+\text{’} \}$$

The above comprise a "grammar".

We play the game by starting with a sequence, say 11+111=, and replacing any occurrence of the right hand side of a production with its left hand side, stopping when only terminals remain

$$\begin{aligned} & \text{11+111=} \\ \Longrightarrow\ & \text{111+11=} \\ \Longrightarrow\ & \text{1111+1=} \\ \Longrightarrow\ & \text{11111+=} \\ \Longrightarrow\ & \text{11111} \end{aligned}$$

Game 2 again

We define a finite set of “terminal symbols” or “alphabet symbols” $S = \{$‘(’,‘)’,‘[’,‘]’$\}$.

We define a finite set of “nonterminal symbols” $V = \{\text{Start}\}$.

We define a finite set of “productions”

$$P = \{\ \text{Start} \longrightarrow \epsilon,\quad \text{Start} \longrightarrow \text{‘(’}\ \text{Start}\ \text{‘)’},\quad \text{Start} \longrightarrow \text{‘[’}\ \text{Start}\ \text{‘]’}\ \}$$

We define a starting nonterminal Start.

We play the game by starting with the starting nonterminal and replacing left hand sides with right hand sides until we only have terminals

$$\begin{aligned} & \text{Start} \\ \Longrightarrow\ & (\text{Start}) \\ \Longrightarrow\ & ((\text{Start})) \\ \Longrightarrow\ & (([\text{Start}])) \\ \Longrightarrow\ & (([])) \end{aligned}$$

Unlike game 1, we are faced with some choices.

Formalizing a bit

A bit of notation

Letter Conventions

- α, β, γ, δ, η, and κ are strings of symbols (terminal or nonterminal).
- s, t, u, v, w, are strings of terminals.
- a, b, c, d, and e are terminals.
- A, B, C, D, E are nonterminals
- X, Y, Z are symbols (terminal or nonterminal)

Abbreviations

- Strings of length 1 may be written as symbols. E.g. a instead of $[a]$.
- Carets (catenations) may be left out. E.g. st instead of $s\hat{}t$.
- These abbreviations may be combined: for example αBc means $\alpha\hat{}[B]\hat{}[c]$

Grammars

Definition: A "grammar" is a tuple $G = (V, S, P, A_{\mathrm{start}})$ or $G = (V, S, P)$ where

- V is a finite set of nonterminal symbols
- S is a finite set of terminal symbols (disjoint from V)
- P is a finite set of production rules of the form $\alpha \longrightarrow \beta$ with at least one nonterminal in α.
- A_{start} is a member of V called the "start symbol"

Production

If we have a production rule $\alpha \longrightarrow \beta$, we say a string $\gamma\alpha\delta$ "can produce" a string $\gamma\beta\delta$.

Definition: More formally, given a grammar $G = (V, S, P, A_{\mathrm{start}})$ we say that η "can produce" κ exactly if there exist

- a production rule $(\alpha \longrightarrow \beta) \in P$
- and strings γ and δ such that $\eta = \gamma\alpha\delta$ and $\kappa = \gamma\beta\delta$.

We write $\eta \Longrightarrow \kappa$ to mean η "can produce" κ

Derivation

Definition: If there exists a finite sequence of strings α_0, α_1, ... , α_n such that

$$\alpha = \alpha_0 \Longrightarrow \alpha_1 \Longrightarrow ... \Longrightarrow \alpha_n = \beta$$

then we say that α “derives” β. In notation:

$$\alpha \stackrel{*}{\Longrightarrow} \beta$$

And we say that α, α_1, ... , β is a “derivation”.

The function defined by a grammar

Each grammar defines a relation $r_G \in (V \cup S)^* \leftrightarrow S^*$ so that if $\alpha \stackrel{*}{\Longrightarrow} \beta \in S^*$ then

$$(\alpha, \beta) \in \text{graph}(r_G)$$

For some grammars, this relation is a function.

Every computable function is expressible as a grammar.

The language generated by a grammar

Definition: If a grammar G has a start symbol A_{start}, then **the language generated by** the grammar is

$$L(G) = \{w \mid A_{\text{start}} \stackrel{*}{\Longrightarrow} w\}$$

Note that $L(G) \subseteq S^*$. For example for G from game 1 we have

$$L(G) = \{\epsilon, (), [], (()), ([]), [()], [[]], \ldots\}$$

Every language that can be recognized by some sort of digital computer can be expressed by a grammar.

Context Free Grammars

Game 1 and Game 2 have a significant difference.

Game 2 only produces trees. This is because each puzzle piece has only one semicircle in its top row.

We call such a grammar "context free".

Definitions

Definition: A "context free grammar" is a grammar where each production rule is of the form

$$A \longrightarrow \beta$$

for some $A \in V$.

Definition: A "context free language" is a language generated by some context free grammar.

Significance

Con: There are recursively enumerable languages that are not context free.

Pro: context free grammars

- Are easy to use and understand
- Given a grammar, there is always an algorithm to determine whether or not a string is in the language generated by the grammar: $\Theta(N^3)$
  - For common special cases there are fast algorithms: $\Theta(N)$.
- Many useful and important languages are context free.
  - Example: The language of syntactically correct Java classes

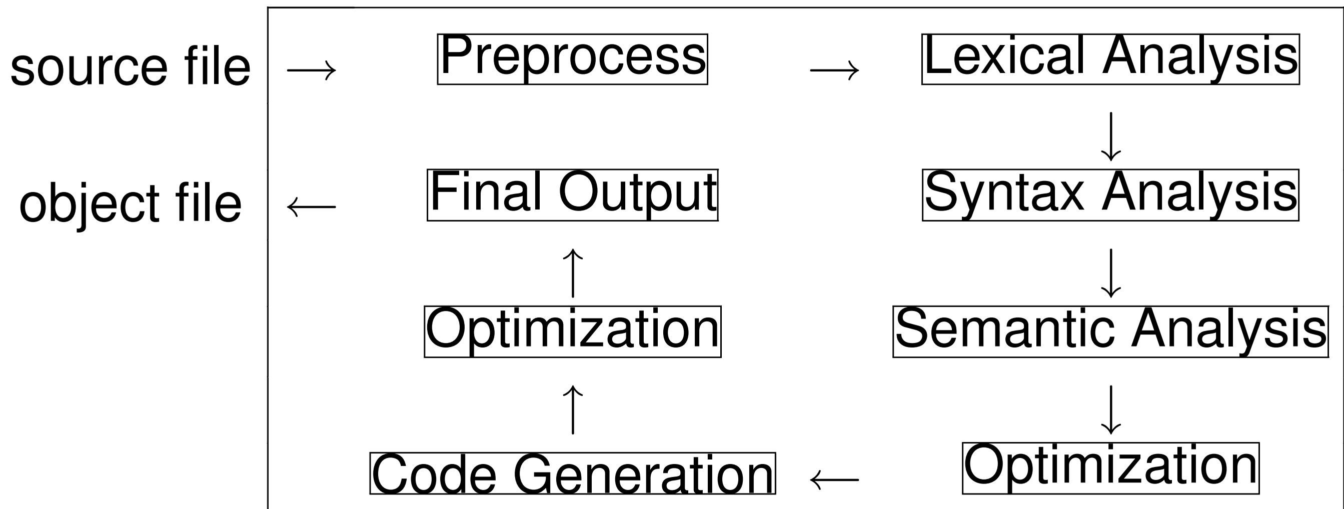
  * Example: Well-formed XML documents.
  * Example: Valid XML documents.
  * Example: Correct usages of many communication protocols.

- For languages that are not context free, we can often start by defining a context free language and then restricting that language
  * Example: The language of compile-time error free Java classes.

Examples Of Context Free Grammars

Programming language examples

Typical Compiler phase structure



Phase goals

- Preprocessing: character sequence to character sequence.
- Lexical analysis: character sequence to sequence of tokens
- Syntax analysis (aka parsing): token sequence to "abstract syntax tree"
- Semantic analysis: build symbol table and find errors
- Code Generation: Select instruction sequences
- Optimization: various time and space improvements
- Final output: output machine code (or assembly code).

Role of grammars: Grammars are used in

- **Preprocessing** to parse macro definitions & uses, includes, conditional compilation etc.

- **Lexical analysis** uses a grammar to describe how to break a sequence of characters into a sequence of "tokens"
  - * spaces, newlines, and comments not output
  - * Example input to lexical analysis:

```
// Read two numbers
    var i :  float    read i
    var j :  float    read j
// find the average, and print it
    var k    k := (i+j)/2    print k
```

  - * Example Output:
    var, (**id**, i), **:**, **float**, ..., /, (**num**, 2), **print** (**id** k)
- **Syntax Analysis (parsing)** determines if the sequence of tokens is syntactically in the language and (typically) builds a tree representation.
- **Code generation** Grammars are sometimes used to describe sequences of operations that correspond to machine instructions.

A handy abbreviation:

We abbreviate multiple productions with the same left-hand side by writing

$$A \longrightarrow \alpha \mid \beta$$

to mean that both $A \longrightarrow \alpha$ and $A \longrightarrow \beta$ are productions.

Floating point numbers in C/C++ (lexical phase).

Terminals are characters written in `typewriter` font.

$$\begin{array}{rcl}
\textit{floatNum} & \longrightarrow & \textit{fract optExp optFloatSufix} \\
 & \mid & \textit{digits exp optFloatSufix} \\
\textit{fract} & \longrightarrow & \textit{optDigits}\ \texttt{.}\ \textit{digits} \mid \textit{digits}\ \texttt{.} \\
\textit{digits} & \longrightarrow & \textit{digit} \mid \textit{digit digits} \\
\textit{exp} & \longrightarrow & \texttt{E}\ \textit{sign digits} \\
\textit{exp} & \longrightarrow & \texttt{e}\ \textit{sign digits} \\
\textit{optExp} & \longrightarrow & \epsilon \mid \textit{exp} \\
\textit{optDigits} & \longrightarrow & \epsilon \mid \textit{digits} \\
\textit{optFloatSufix} & \longrightarrow & \epsilon \mid \texttt{f} \mid \texttt{l} \mid \texttt{F} \mid \texttt{L} \\
\textit{digit} & \longrightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \\
 & \mid & \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{sign} & \longrightarrow & \epsilon \mid + \mid -
\end{array}$$

Example strings in the language:

`123.456`   `.456E+789`   `123E798`

Not in the language:

```
123$456
.456.789
123
123E0.1
```

An example derivation:

$$
\begin{aligned}
\mathit{floatNum} &\Longrightarrow \mathit{fract}\ \mathit{optExp}\ \underline{\mathit{optFloatSufix}} \\
&\Longrightarrow \mathit{fract}\ \underline{\mathit{optExp}} \\
&\Longrightarrow \underline{\mathit{fract}} \\
&\Longrightarrow \underline{\mathit{optDigits}}\ .\ \mathit{digits} \\
&\Longrightarrow \underline{\mathit{digits}}\ .\ \mathit{digits} \\
&\Longrightarrow \mathit{digit}\ .\ \underline{\mathit{digits}} \\
&\Longrightarrow \mathit{digit}\ .\ \mathit{digit}\ \underline{\mathit{digits}} \\
&\Longrightarrow \underline{\mathit{digit}}\ .\ \mathit{digit}\ \mathit{digit} \\
&\Longrightarrow 1\ .\ \underline{\mathit{digit}}\ \mathit{digit} \\
&\Longrightarrow 1\ .\ 2\ \underline{\mathit{digit}} \\
&\Longrightarrow 1\ .\ 2\ 3
\end{aligned}
$$

A simple programming language (lexical level, partial grammar)

Terminals are all ASCII characters

$$
\begin{aligned}
\mathit{token} &\longrightarrow \mathit{spaces}\ \mathit{tk} \\
\mathit{tk} &\longrightarrow \mathit{keyword} \mid \mathit{id} \mid \mathit{num} \mid \mathit{punc} \mid \mathit{op} \mid \mathit{EOF} \\
\mathit{spaces} &\longrightarrow \epsilon \mid \mathit{space}\ \mathit{spaces} \\
\mathit{space} &\longrightarrow \mathit{spacechar} \mid \mathit{newlinechar} \\
&\mid\quad \mathit{tabchar} \mid \mathit{comment} \\
\mathit{comment} &\longrightarrow /\ /\ \mathit{nonnewlines}\ \mathit{newlinechar} \\
\mathit{nonnewlines} &\longrightarrow \epsilon \mid \mathit{nonnewline}\ \mathit{nonnewlines} \\
\mathit{nonnewline} &\longrightarrow \mathit{alpha} \mid \mathit{digit} \mid \texttt{(} \mid \texttt{)} \mid + \mid - \mid / \mid * \mid \cdots \\
\mathit{keyword} &\longrightarrow \texttt{p r i n t} \\
&\mid\quad \texttt{r e a d} \\
&\mid\quad \ldots \\
\mathit{id} &\longrightarrow \mathit{alpha}\ \mathit{alphasOrDigits} \\
\mathit{num} &\longrightarrow \mathit{digit} \mid \mathit{digit}\ \mathit{num} \\
\mathit{alphasOrDigits} &\longrightarrow \epsilon \mid \mathit{alphaOrDigit}\ \mathit{alphasOrDigits} \\
\mathit{alphaOrDigit} &\longrightarrow \mathit{alpha} \mid \mathit{digit} \\
\mathit{alpha} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \cdots \mid \texttt{Z} \\
\mathit{digit} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \cdots \mid \texttt{9} \\
\mathit{punc} &\longrightarrow (\mid) \\
\mathit{op} &\longrightarrow + \mid - \mid / \mid * \mid = \mid !=
\end{aligned}
$$

Expressions for a programming language (syntactic level)

Terminals are **(**, **)**, **num**, **id**, **+**, -, /, ***, =,** and **!=**. Starting nonterminal is *exp*

$$
\begin{aligned}
\textit{exp} &\longrightarrow \mathbf{num} \\
&\quad\mid\ \mathbf{id} \\
&\quad\mid\ \textit{exp bop exp} \\
&\quad\mid\ \textit{uop exp} \\
&\quad\mid\ \mathbf{(}\ \textit{exp}\ \mathbf{)} \\
\textit{bop} &\longrightarrow \mathbf{+} \mid \mathbf{-} \mid \mathbf{*} \mid / \mid \mathbf{=} \mid \mathbf{!=} \\
\textit{uop} &\longrightarrow \mathbf{+} \mid \mathbf{-}
\end{aligned}
$$

Example string in the language:

(num + num) / - id = id / num / id

Not in the language:

(num + (id * id)))

- num (num * * id)

A simple programming language (syntactic level)

Terminals are as in the previous example plus **print**, **read**, **var**, **if**, **else**, **end**, **while**, **int**, **float**, **bool**, **eof**

$$
\begin{array}{rcl}
\textit{prog} & \longrightarrow & \textit{stat}\ \mathbf{eof} \\
\textit{stat} & \longrightarrow & \mathbf{print}\ \textit{exp} \\
 & \mid & \mathbf{read}\ v \\
 & \mid & \mathbf{var}\ v : \textit{type} \\
 & \mid & v := \textit{exp} \\
 & \mid & \mathbf{if}\ \textit{exp}\ \textit{stat}\ \mathbf{else}\ \textit{stat}\ \mathbf{end} \\
 & \mid & \mathbf{while}\ \textit{exp}\ \textit{stat}\ \mathbf{end} \\
 & \mid & \textit{stat}\ \textit{stat} \\
v & \longrightarrow & \mathbf{id} \\
\textit{type} & \longrightarrow & \mathbf{int} \mid \mathbf{float} \mid \mathbf{bool} \\
\textit{exp} & \longrightarrow & \textit{as in previous example}
\end{array}
$$

Strings in the language:

var id : int read id print num + id eof

if num id := num else id := id end eof

Note that there may still be “semantic” errors.

Strings not in the language:

var id read num print num + eof

if num id := num else id := id eof

Internet applications

HTML tags (as Netscape and IE recognize them)

$$\begin{aligned}
\textit{startTag} &\longrightarrow < \textit{elementName attributes} > \\
\textit{elementName} &\longrightarrow \textit{letter moreElementName} \\
\textit{letter} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\textit{moreElementName} &\longrightarrow \textit{nonSpace moreElementName} \mid \epsilon \\
\textit{nonSpace} &\longrightarrow \textit{letter} \mid \ldots \\
\textit{attributes} &\longrightarrow \textit{etc.}
\end{aligned}$$

Attributes is a bit complex, so let's leave it for now

The http URI

$$\begin{aligned}
\textit{http_URL} &\longrightarrow \texttt{h t t p : / /}\ \textit{host optPort optAbsPath} \\
\textit{optPort} &\longrightarrow \epsilon \mid \texttt{:}\ \textit{port} \\
\textit{optAbsPath} &\longrightarrow \epsilon \mid \textit{absPath} \mid \textit{absPath}\ ?\ \textit{query}
\end{aligned}$$

Hosts and ports are defined by

$$
\begin{aligned}
\mathit{host} &\longrightarrow \mathit{hostName} \mid \mathit{iPv4address} \\
\mathit{hostName} &\longrightarrow \mathit{labels}\ \mathit{optDot} \\
\mathit{labels} &\longrightarrow \mathit{dlabel}\ .\ \mathit{labels} \mid \mathit{tlabel} \\
\mathit{dlabel} &\longrightarrow \mathit{alphaNum} \mid \mathit{alphaNum}\ \mathit{labelChars}\ \mathit{alphaNum} \\
\mathit{tlabel} &\longrightarrow \mathit{alpha} \mid \mathit{alpha}\ \mathit{labelChars}\ \mathit{alphaNum} \\
\mathit{labelChars} &\longrightarrow \mathit{alphaNum} \mid - \\
\mathit{optDot} &\longrightarrow \epsilon \mid \texttt{.} \\
\mathit{iPv4address} &\longrightarrow \mathit{digits}\ \texttt{.}\ \mathit{digits}\ \texttt{.}\ \mathit{digits}\ \texttt{.}\ \mathit{digits} \\
\mathit{digits} &\longrightarrow \mathit{num} \mid \mathit{num}\ \mathit{digits} \\
\mathit{alphaNum} &\longrightarrow \mathit{alpha} \mid \mathit{num} \\
\mathit{alpha} &\longrightarrow \texttt{a} \mid \texttt{b} \mid \ldots \mid \texttt{z} \mid \texttt{A} \mid \texttt{B} \mid \ldots \mid \texttt{Z} \\
\mathit{num} &\longrightarrow \texttt{0} \mid \texttt{1} \mid \ldots \mid \texttt{9} \\
\mathit{port} &\longrightarrow \epsilon \mid \mathit{num}\ \mathit{port}
\end{aligned}
$$

Paths

$$
\begin{aligned}
\mathit{absPath} &\longrightarrow /\ \mathit{segments} \\
\mathit{segments} &\longrightarrow \mathit{segment} \mid \mathit{segment}\ /\ \mathit{segments} \\
\mathit{segment} &\longrightarrow \text{etc} \\
\mathit{query} &\longrightarrow \text{etc}
\end{aligned}
$$

I won't go into all the details, but just comment that a segment is a sequence of almost any characters, as is a query.

Protocols

In this case the tokens are requests and replies. Requests go from client to server and replies from server to client.

This is a greatly simplified FTP (File Transfer Protocol).

session $\longrightarrow$ **greetingRequest greetingReply** $moreSesn$
$moreSesn$ $\longrightarrow$ **quitRequest quitReply**
$\mid$ **sendFileRequest sendFileReply** $moreSesn$
$\mid$ **sendFileRequest errorReply** $moreSesn$
$\mid$ **getFileRequest getFileReply** $moreSesn$
$\mid$ **getFileRequest errorReply** $moreSesn$

The syntax of the various requests and replies can also be specified in terms of sequences of bytes, just as tokens are specified in compilers.

Aside: Relation to NDFRs.

Define an NDR just as we defined an NDFR, except without the restriction that the number of states be finite.

Given a CFG $G = (V, S, P, A_{\text{start}})$ we can define an NDR as follows.

- Alphabet is S
- States are all strings α where and $\alpha \in (V \cup S)^*$.
- Initial state is A_{start}
- The only final state is ϵ.
- There are two kinds of transitions
  - $*$ All $(a\beta, a, \beta)$ where $s \in A$, $\beta \in (V \cup S)^*$ and
  - $*$ all $(\alpha A\beta, \epsilon, \alpha\gamma\beta)$ where $A \in V$, $\alpha, \beta, \gamma \in (V \cup S)^*$, and $A \to \gamma \in P$.

Now the language described by the grammar and the recognizer are the same.

Note that we can not easily adapt our recognition algorithm for NDFRs because the ϵ-closure of a state may be infinite in size.

Recognition and Parsing

Given a grammar G, the recognition problem is this

Input: A string w of terminal symbols.

Output: Whether or not w is in $L(G)$

Parsing problems are similar, but the output also includes a useful data structure when $w \in L(G)$.

For example:

- In a compiler: we might output an abstract syntax tree.
- In a calculator: we might output the numerical value of an expression.
- We might output machine code or a reverse polish notation (RPN) representation of the input.

Derivation Trees and Left-most Derivations

Definition: A left-most derivation is one where, at each step, the left-most nonterminal is replaced. We write $\alpha \Longrightarrow_{\mathrm{lm}} \beta$

More formally, have $\alpha \Longrightarrow_{\mathrm{lm}} \beta$ iff there are A, s, γ, and δ such that $\alpha = sA\gamma$ and $A \longrightarrow \delta$ and $\beta = s\delta\gamma$. (Recall that $s \in S^*$).

We write $\alpha \overset{*}{\Longrightarrow}_{\mathrm{lm}} \beta$ to indicate that there is a derivation of β from α in 0 or more left-most production steps.

Given a grammar G and a string s in $L(G)$ we can consider a tree that illustrates the proof that the tree is in the language. Any derivation corresponds to a "derivation tree".

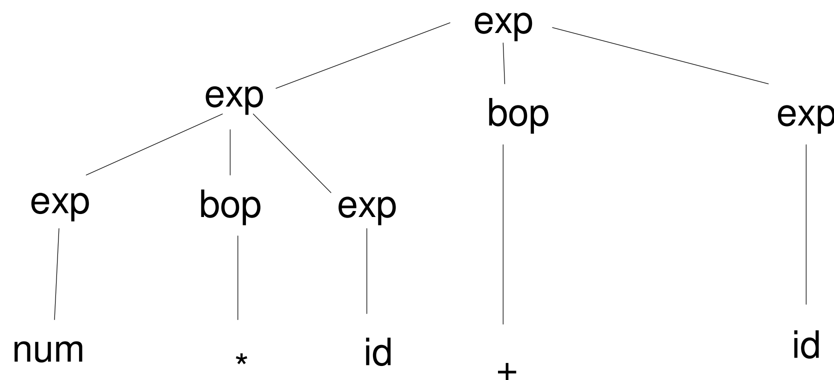
Example

exp $\longrightarrow$ **num** | **id** | *exp bop exp* | *uop exp* | **(** *exp* **)**
bop $\longrightarrow$ **+** | **-** | **/** | ***** | **=** | **!=**
uop $\longrightarrow$ **+** | **-**

Consider the input 2*i+j, which as a string of terminals is: **num * id + id**. One derivation is

$$\begin{aligned} exp &\Longrightarrow exp\ bop\ \underline{exp} \\ &\Longrightarrow \underline{exp}\ bop\ \mathbf{id} \\ &\Longrightarrow exp\ bop\ \underline{exp}\ bop\ \mathbf{id} \\ &\Longrightarrow exp\ \underline{bop}\ \mathbf{id}\ bop\ \mathbf{id} \\ &\Longrightarrow \underline{exp} * \mathbf{id}\ bop\ \mathbf{id} \\ &\Longrightarrow \mathbf{num} * \mathbf{id}\ \underline{bop}\ \mathbf{id} \\ &\Longrightarrow \mathbf{num} * \mathbf{id} + \mathbf{id} \end{aligned}$$

from which we can build the following “derivation tree”.

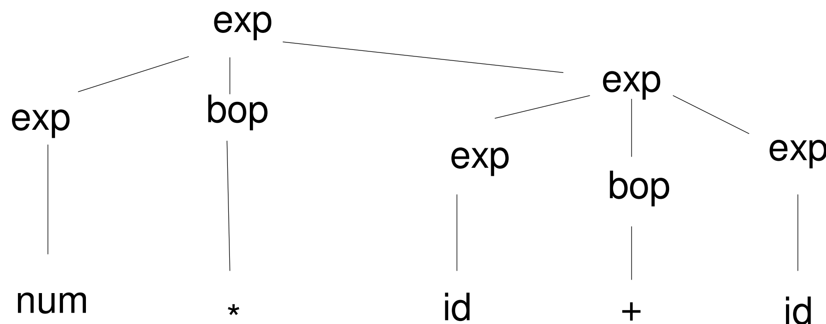


We can build a “left-most derivation” by traversing the tree depth-first and left to right, expanding the nonterminal

that we encounter

$$
\begin{aligned}
\underline{exp} &\Longrightarrow_{\mathrm{lm}} \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \underline{exp}\ bop\ exp\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \underline{bop}\ exp\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \underline{bop}\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \texttt{+}\ \underline{exp} \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \texttt{+}\ \underline{exp}
\end{aligned}
$$

But with the same grammar and the same string, we can build a *different* tree.



The corresponding left-most derivation

$$
\begin{aligned}
\underline{exp} &\Longrightarrow_{\mathrm{lm}} \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \underline{bop}\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \underline{exp} \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \underline{exp}\ bop\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \underline{bop}\ exp \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \texttt{+}\ \underline{exp} \\
&\Longrightarrow_{\mathrm{lm}} \mathbf{num}\ \texttt{*}\ \mathbf{id}\ \texttt{+}\ \mathbf{id}
\end{aligned}
$$

Ambiguity

Definition: We say that a grammar is *ambiguous* iff for some string there exist two or more derivation trees.

Equivalently: a grammar is ambiguous iff some string has two or more left-most derivations.

For many applications, we should avoid ambiguous grammars since

- other matters (semantics) are generally described in terms of the grammar and we don't want ambiguous semantics.
- it is hard to build an efficient parser for ambiguous grammars.

For other applications: e.g. natural language understanding, ambiguity is useful.

Consider

- "I saw a bird with a telescope"
- "I saw a man with a hat"
- "I saw a man with a telescope"

All 3 sentences fit the pattern

pronoun verb det noun prep det noun

but, in the first, the prepositional phrase attaches to the verb, whereas in the second the prepositional phrase attaches to the object. This means we want multiple derivation trees for the same sequence of word forms.

Ambiguity and expression grammars

Here is a grammar for expressions Exp0

$$E \longrightarrow \textbf{n} \mid \textbf{(}\ E\ \textbf{)} \mid E\ \textbf{+}\ E \mid E\ \textbf{-}\ E \mid E\ \textbf{*}\ E \mid E\ \textbf{/}\ E$$

This grammar is highly ambiguous.

How many derivation trees are there for **n - n / n / n - n** ?

14? In a sense only 1 reflects the correct precedence and associativity of the operators.

To ‘enforce’ ‘correct’ parsing of expressions we can write a new grammar, Exp1

$$E \longrightarrow T \mid E\ \textbf{+}\ T \mid E\ \textbf{-}\ T$$
$$T \longrightarrow F \mid T\ \textbf{*}\ F \mid T\ \textbf{/}\ F$$
$$F \longrightarrow \textbf{n} \mid \textbf{(}\ E\ \textbf{)}$$

This grammar is unambiguous.

Left-recursion

A nonterminal A is said to be *left-recursive* if there is a derivation

$$A \stackrel{*}{\Longrightarrow} A\beta$$

with at least one step (for some β).

A grammar is *left-recursive* iff it has at least one left-recursive nonterminal.

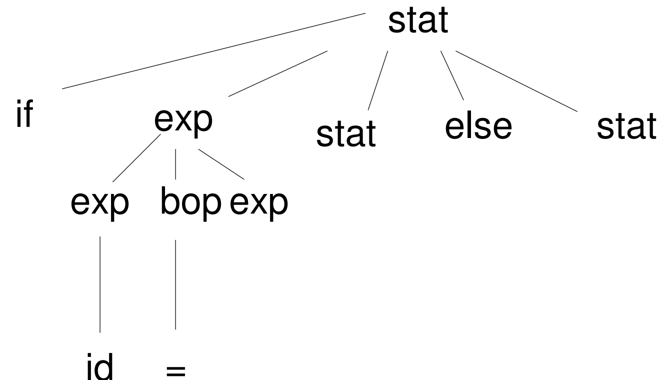
Clearly both Exp0 and Exp1 are left recursive. Here is an unambiguous grammar for the same language as Exp0 and Exp1 that is not left-recursive. Exp2:

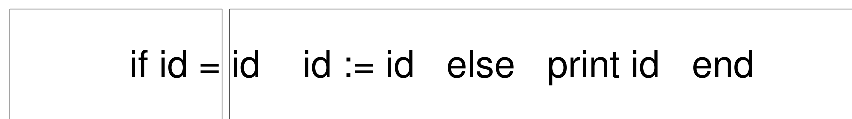
$$\begin{aligned} E &\longrightarrow T\ E1 \\ E1 &\longrightarrow \mathbf{+}\ T\ E1 \mid \mathbf{-}\ T\ E1 \mid \epsilon \\ T &\longrightarrow F\ T1 \\ T1 &\longrightarrow \mathbf{*}\ F\ T1 \mid \mathbf{/}\ F\ T1 \mid \epsilon \\ F &\longrightarrow \mathbf{n} \mid \mathbf{(}\ E\ \mathbf{)} \end{aligned}$$

Top-Down predictive parsing and recognition.

A top-down predictive parser works by trying to build the derivation tree from the top down.

For example here is a derivation tree that is partially built.





Consumed input

Input yet to be consumed

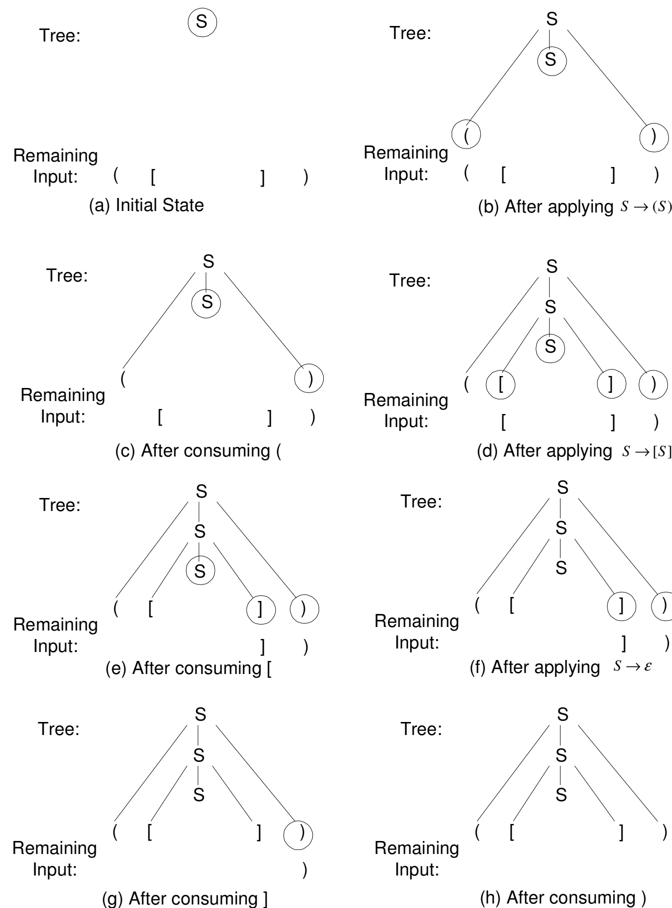
The leaves of the tree are **if id =** *exp stat* **else** *stat*

This tree is a proof that

$$\mathit{stat} \stackrel{*}{\Longrightarrow} \mathbf{if}\ \mathbf{id}\ = \mathit{exp}\ \mathit{stat}\ \mathrm{else}\ \mathit{stat}$$

Conceptual view

The idea is to walk the derivation tree in a depth-first manner, while (conceptually) building the tree and consuming the input. Circled nodes are not yet visited. The left to right sequence of circled nodes is called *the prediction*.



(a) Initial State

(b) After applying $S \to (S)$

(c) After consuming (

(d) After applying $S \to [S]$

(e) After consuming [

(f) After applying $S \to \varepsilon$

(g) After consuming]

(h) After consuming)

Augmenting the grammar

It will simplify things later if we augment the grammar with

- a new terminal, $\$$, called a sentinel.
- a new starting nonterminal A'_{start} and
- a new production rule $A'_{\mathrm{start}} \longrightarrow A_{\mathrm{start}}\ \$$, where A_{start} was the original starting nonterminal.

We will also add a $ to the end of each input string.

(Typically $ represents the end of the file.)

States, steps, and stops

We will process one terminal at a time.

States

A top-down predictive parser's state consists of

- the input processed so far s
- the current prediction α and
- the remaining input sequence, t

Let's write this as

$$s_{\blacktriangle}\alpha,\ t$$

(Note: we don't represent the partially built tree, but only the sequence of unvisited nodes α).

As a loop invariant, we'll have that

$$A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} s\alpha$$

and that $st = w\$$ where w is the original input.

Inv: $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} s\alpha$ and $st = w\$$ for state $s_\blacktriangle\alpha,\ t$.

As an initial state, we'll start with

$$\epsilon_\blacktriangle A'_{\mathrm{start}},\ w\$$$

where w is the input string.

Steps

We use two rules to step from one state to another

- Shift: (Read one terminal from the input)

$$s_\blacktriangle a\beta,\ au \quad \vdash \quad sa_\blacktriangle\beta,\ u$$

- Produce: (Expand the leftmost nonterminal)

$$s_\blacktriangle A\beta,\ t \quad \vdash \quad s_\blacktriangle\gamma\beta,\ t$$

  where $A \longrightarrow \gamma$ is a production rule.

Note that the initial state implies the invariant and that each step preserves the invariant.

Theorem: $A_{\mathrm{start}} \overset{*}{\Longrightarrow} w$ iff there is a finite sequence of steps that goes

$$\epsilon_\blacktriangle A'_{\mathrm{start}},\ w\$ \quad \vdash \quad \ldots \quad \vdash \quad w\$_\blacktriangle\epsilon,\ \epsilon$$

Proof outline: Suppose $A_{\mathrm{start}} \overset{*}{\Longrightarrow} w$, then we can use a derivation sequence to construct a sequence of steps. Details left as an exercise.

Suppose there is a sequence of steps

$$\epsilon_\blacktriangle A'_{\mathrm{start}},\ w\$ \quad \vdash \quad \ldots \quad \vdash \quad w\$_\blacktriangle\epsilon,\ \epsilon$$

Since the invariant is true of the first state and each step preserves the invariant, the invariant will be true of the final state, meaning $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} w\$$ and so $A_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} w$.

Stopping

Inv: $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} s\alpha$ and $st = w\$$ for state $s_\blacktriangle \alpha,\ t$.

Since $st = w\$$ and w contains no $\$$, $(t = \epsilon) = (s$ contains a $\$)$

Since $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} s\alpha$ and any string derived from A'_{start} has one $\$$ at its end, $(\alpha = \epsilon) = (s$ contains a $\$)$.

Putting these together $(\alpha = \epsilon) = (t = \epsilon)$. So we don't need to worry about states of the form $s_\blacktriangle \epsilon,\ t$, where $t \neq \epsilon$, nor about states of the form $s_\blacktriangle \alpha,\ \epsilon$. where $\alpha \neq \epsilon$.

There are three ways to stop recognition:

- Successful stop: We stop when the state is $s_\blacktriangle \epsilon,\ \epsilon$. In this case (from the invariant) $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} s$ and that $s = w\$$. Thus $A'_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} w\$$. Thus $A_{\mathrm{start}} \overset{*}{\Longrightarrow}_{\mathrm{lm}} w$.
- Error stop 0: We also stop when the predicted input does not match the actual input, i.e. the state is $s_\blacktriangle a\beta,\ bu$ where $a \neq b$.
- Error stop 1: We also stop when the state is $s_\blacktriangle A\beta,\ t$ and there is no appropriate production rule.

If we come to a "Successful stop" then

- the input was in the language: $A_{\mathrm{start}} \overset{*}{\Longrightarrow} w$

If we come to an "Error stop" then either

- the input was not in the language,
- or we made a bad choice in a 'produce' step.

Later we'll see how to avoid bad choices.

Example

Here is a trace of the algorithm for grammar Exp2 (augmented) and an input of **n * n**

	$s_\blacktriangle \alpha,$	t	**Action**
	$\epsilon_\blacktriangle E',$	**n * n $**	Produce $E' \longrightarrow E\$$
$\vdash$	$\epsilon_\blacktriangle E\ \$,$	**n * n $**	Produce $E \longrightarrow T\ E1$
$\vdash$	$\epsilon_\blacktriangle T\ E1\ \$,$	**n * n $**	Produce $T \longrightarrow F\ T1$
$\vdash$	$\epsilon_\blacktriangle F\ T1\ E1\ \$,$	**n * n $**	Produce $F \longrightarrow$ **n**
$\vdash$	$\epsilon_\blacktriangle \mathbf{n}\ T1\ E1\ \$,$	**n * n $**	Shift
$\vdash$	$\mathbf{n}_\blacktriangle T1\ E1\ \$,$	*** n $**	Produce $T1 \longrightarrow$ * $F\ T1$
$\vdash$	$\mathbf{n}_\blacktriangle$* $F\ T1\ E1\ \$,$	*** n $**	Shift
$\vdash$	**n *** $_\blacktriangle\ F\ T1\ E1\ \$,$	**n $**	Produce $F \longrightarrow$ **n**
$\vdash$	**n *** $_\blacktriangle \mathbf{n}\ T1\ E1\ \$,$	**n $**	Shift
$\vdash$	**n * n** $_\blacktriangle T1\ E1\ \$,$	**$**	Produce $T1 \longrightarrow \epsilon$
$\vdash$	**n * n** $_\blacktriangle E1\ \$,$	**$**	Produce $E1 \longrightarrow \epsilon$
$\vdash$	**n * n** $_\blacktriangle \$,$	**$**	Shift
$\vdash$	**n * n $**$_\blacktriangle \epsilon,$	ϵ	Success

So the string is in the language.

Note how the sequence of produce steps corresponds to the left-most derivation.

In a more algorithmic form

Algorithm: Top down predictive parsing

Input: A string w not containing a $\$$.

Output: 'success' or 'error'. If the output is 'success' then $w \in L(G)$. If the output is 'error', then either $w \notin L(G)$ or a bad choice was made.

var $t := w\$$ *// the remaining input.*
var $s := \epsilon$ *// the consumed input, not really needed.*
var $\alpha := A'_{\mathrm{start}}$ *// the predicted input.*
invariant $A'_{\mathrm{start}} \stackrel{*}{\Longrightarrow}_{\mathrm{lm}} s\alpha$ and $st = w\$$
while $\alpha \neq \epsilon$ **do**
  if $\alpha(0) \in S$ **then**
    if $\alpha(0) = t(0)$ **then** $($
      // Shift step
      $s := s\,t(0) \quad t := \mathrm{tail}(t) \quad \alpha := \mathrm{tail}(\alpha)\ )$
    else /* $\alpha(0) \neq t(0)$ */ error
  else /* $\alpha(0) \in V$ */ $($
    try to pick a suitable production rule $\alpha(0) \to \gamma$
    if a suitable production rule exists
      // Produce step
      $\alpha := \gamma\ \mathrm{tail}(\alpha)$
    else */* no suitable production rule exists */*
      error $)\ )$
/* Since $\alpha = \epsilon$, $t = \epsilon$ and $A_{\mathrm{start}} \stackrel{*}{\Longrightarrow}_{\mathrm{lm}} w$.*/ success

($\mathrm{tail}(t)$ is the string $[t(1), t(2), ..., t(\|t\| - 1)]$. error means stop with output 'error'. success means stop with output 'success'.)

Assuming the grammar is not left-recursive, the top-down predictive parsing algorithm must terminate and is $\Theta(N)$ time, where N is the length of the input.

To be done: We still haven't said how to pick a suitable production rule when a nonterminal comes to the top of the α stack.

LL(1) Grammars

Definition: An augmented grammar is called an 'LL(1) grammar' when the suitable production rule can always be chosen on the basis of the next input item $t(0)$ and the left-most nonterminal $\alpha(0)$.

Left-recursive grammars can never be LL(1). (Why?)

For each production rule $A \longrightarrow \gamma$ we compute the set of terminals which $t(0)$ might equal when $A \longrightarrow \gamma$ is chosen as the production rule in a successful run of the TDPP algorithm.

This set is called the *selector set* of the production.

So let's consider Exp2 augmented

$E' \longrightarrow E\$$	$\{$**n**$,$ **(**$\}$	$T1 \longrightarrow$ ***** $F\ T1$	$\{$*****$\}$
$E \longrightarrow T\ E1$	$\{$**n**$,$ **(**$\}$	$T1 \longrightarrow$ **/** $F\ T1$	$\{$**/**$\}$
$E1 \longrightarrow$ **+** $T\ E1$	$\{$**+**$\}$	$T1 \longrightarrow \epsilon$	$\{\$,$ **)**$,$ **+**$,$ **-**$\}$
$E1 \longrightarrow$ **-** $T\ E1$	$\{$**-**$\}$	$F \longrightarrow$ **n**	$\{$**n**$\}$
$E1 \longrightarrow \epsilon$	$\{\$,$ **)**$\}$	$F \longrightarrow$ **(** E **)**	$\{$**(**$\}$
$T \longrightarrow F\ T1$	$\{$**n**$,$ **(**$\}$		

On the right we have a selector set for each production rule.

We implement the picking part of the algorithm

/*do pick a suitable production rule $\alpha(0) \rightarrow \gamma$ by*/
  if there is a production rule $\alpha(0) \rightarrow \gamma$
  with $t(0)$ in its selector set **then**
    pick that production rule
  else
    there is no suitable production rule

A grammar is LL(1) iff for each nonterminal A and for each pair of productions for A, the selectors sets of the two productions are disjoint.

I.e. a grammar is LL(1) iff, for all A, α, β, such that $A \longrightarrow \alpha$ and $A \longrightarrow \beta$ are distinct productions,

$$\left(\begin{array}{rl} & \mathrm{SelectorSet}\,(A \longrightarrow \alpha) \\ \cap & \mathrm{SelectorSet}\,(A \longrightarrow \beta) \end{array}\right) = \emptyset$$

Computing the selector sets.

Consider the selector set for a production rule $A \longrightarrow \alpha$

Look at $E1 \longrightarrow +\ E$. It is clear that this production rule should only be picked if the next terminal is a + sign.

In general if $\alpha \stackrel{*}{\Longrightarrow} b\beta$ then b should be in the selector set of $A \longrightarrow \alpha$.

But there is more to it than that when $\alpha \stackrel{*}{\Longrightarrow} \epsilon$.

Consider $E1 \rightarrow \epsilon$ the next item in the input should be one that could legitimately follow an $E1$ in a successful derivation. Only items that could follow E qualify and these are $\$$ and **)**.

Suppose $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A b \gamma \Longrightarrow \beta \alpha b \gamma \stackrel{*}{\Longrightarrow} \beta b \gamma$ then b should be in the selector set of $A \longrightarrow \alpha$.

(Note that $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A \Longrightarrow \beta\alpha \stackrel{*}{\Longrightarrow} \beta$ is not possible!)

Define functions First and Follow for an augmented grammar by

- $b \in \text{First}(\alpha)$ iff $\alpha \stackrel{*}{\Longrightarrow} b\beta$, for some β and
- $b \in \text{Follow}(A)$ iff $A'_{\text{start}} \stackrel{*}{\Longrightarrow} \beta A b \gamma$, for some β and γ.

The selector set for $A \longrightarrow \alpha$ is

$$\text{First}(\alpha)\text{, when } \alpha \stackrel{*}{\not\Longrightarrow} \epsilon$$

and is

$$\text{First}(\alpha) \cup \text{Follow}(A)\text{, when } \alpha \stackrel{*}{\Longrightarrow} \epsilon$$

Recursive Descent Parsing

Recursive descent parsing works on the same principle as TDPP, but uses the call-return stack rather than an explicit stack.

Let w be the input and f be a boolean variable to indicate recognition. Our specification is

$$\left\langle f' = \left(A_{start} \stackrel{*}{\Longrightarrow} w\right)\right\rangle$$

As with TDPP we will use a variable t that is initialized to $w\hat{}[\$]$ where $\$$ is a sentinel symbol that does not occur in w.

We create a subroutine for each nonterminal

Roughly speaking the job of subroutine A is to either

- indicate an error by setting f to false

$$\langle \neg f' \rangle$$

  or

- remove from t a prefix u derived from A

$$\left\langle f' \wedge \exists u \cdot t = u\hat{}t' \wedge \left(A \stackrel{*}{\Longrightarrow} u\right)\right\rangle$$

  Later we'll look at how to choose between these actions and how to choose the string u.

Recursive Descent parsing of LL(1) grammars

If the grammar is LL(1) then creating an R.D. parser for it is a mechanical process.

Example

We can write a recursive descent recognizer for Exp2 as follows

global var $t : (S \cup \{\$\})^* \cdot$
global var $f : \mathbb{B} \cdot$

procedure main($w : S^*$) **is**
    $f := \mathfrak{true}$;
    $t := w\hat{}[\$]$; // Where w is the input
    E;
    expect($\$$) ;
    return f

procedure consume **is**
    $t := \text{tail}(t)$

procedure expect(a) **is**
    if $t(0) = a$ **then** $t := \text{tail}(t)$
    else error

procedure error **is**
    $f := \mathfrak{false}$;

procedure E **is**
    if $t(0) \in \{$'**n**','**(**'$\}$ **then** (
        T ;
        E1)
    else
        error

procedure E1 **is**
    if $t(0) =$ '**+**' **then** (
        consume ;
        T ;
        E1)
    else if $t(0) =$ '**-**' **then** (
        consume ;
        T ;
        E1)
    else if $t(0) \notin \{\$,$'**)**'$\}$ **then**
        error

... T and T1 are similar to E and E1 ...

procedure F **is**
    if $t(0) =$ '**n**' **then**
        consume
    else if $t(0) =$ '**(**' **then** (
        consume ;
        E ;
        expect('**)**'))
    else error

Getting results

Often we not only want to recognize the input but also process the input to create an output in the case where the input is recognized.

We can do this by augmenting the recursive descent parser with extra code.

As an example, we will produce numbers.

global var $t \in (S \cup \{\$\})^* \cdot$
global var $f : \mathbb{B} \cdot$

procedure main **is**
  $f := \mathfrak{true}$;
  $t := w\hat{}[\$]$; // Where w is the input
  var $p :=$ E
  expect($\$$)
  return (f, p)

procedure error **is**
  $f := \mathfrak{false}$;
  return 0

procedure consume **is**
  $t := \text{tail}(t)$

procedure expect(a) **is**
  if $t(0) = a$ **then** $t := \text{tail}(t)$
  else $f := \mathfrak{false}$

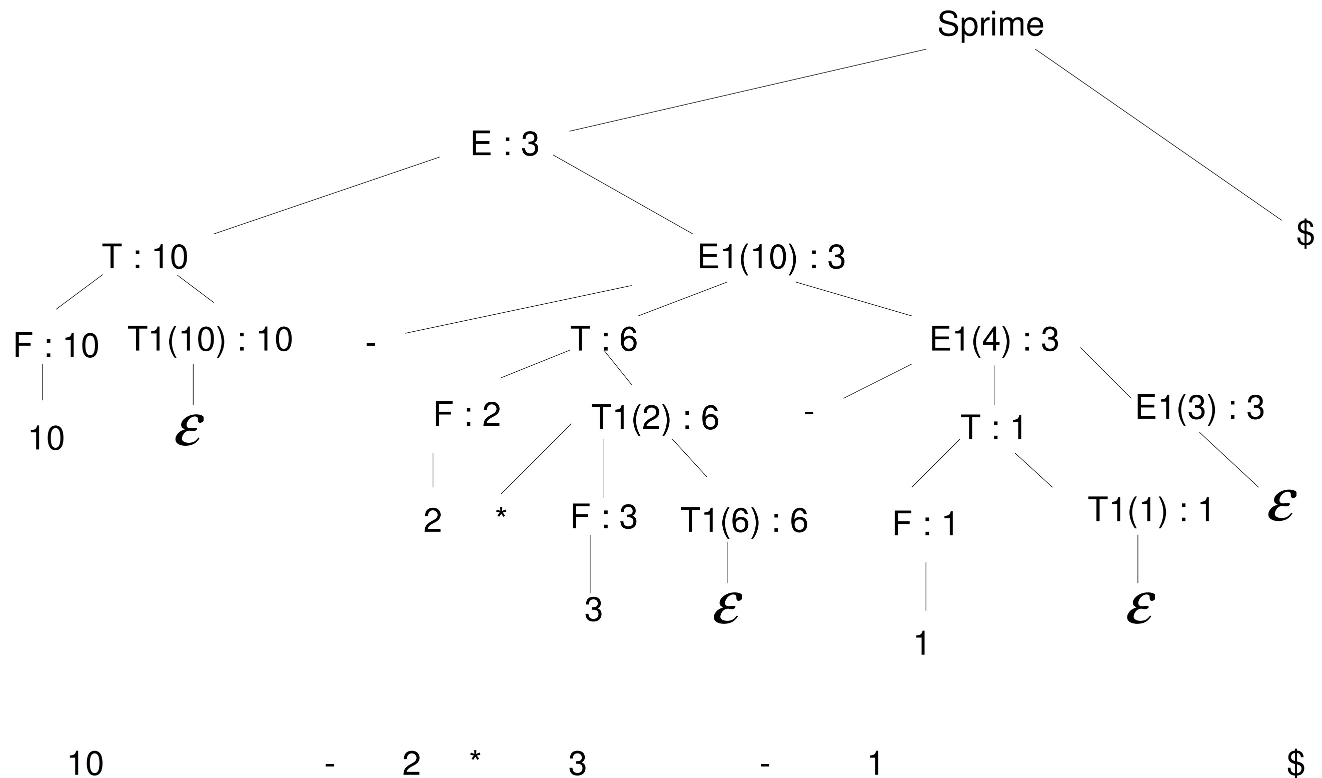
```
procedure E is
    if t(0) ∈ {'n','('} then (
        var p := T ·
        return E1(p) )
    else
        return error

procedure E1( p ) is
    if t(0) = '+' then (
        t := tail(t) ;
        var q := T ·
        return E1( p + q ) )
    else if t(0) = '-' then (
        t := tail(t) ;
        var q := T ·
        return E1( p − q ) )
    else if t(0) ∉ {$,')'} then
        return error
    else
        return p
```

... T and T1 are similar to E and E1 ...

```
procedure F is
    if t(0) = 'n' then (
        var p:= the value associated with t(0)·
        t := tail(t) ;
        return p )
    else if t(0) = '(' then (
        t := tail(t)
        var p:= E then (
        expect( ')' )
        return p )
    else
        return error
```

Consider parsing the sequence: **10-2*3-1** As a string of terminals we have **n-n*n-n**.

We have the following call tree which reflects the derivation tree. Return values are shown after the colon.



Parsing 10-2*3-1 with the parameter values and the return values shown.

Questions to consider:

- Modify the recursive descent parser above to produce an abstract syntax tree for an expression.
- Modify the recursive descent parser above to produce RPN.
- How could you alter the top-down predictive recognizer to compute results rather than to just recognize? *Hint: consider adding "commands" to the productions and executing the commands when they come to the top of the stack. See the 'Command' pattern in Gamma et al.*
- Modify the grammar Exp2 with added commands so that it computes the right result (hint use an extra stack to hold intermediate results).
- Add unary operators to the grammar Exp2 so that you get a LL(1) grammar. Can you parse $12/-2$?

Formal specification of recursive descent

Consider a procedure A for a nonterminal A.

Some observations

- As a precondition, each procedure can assume that t is properly terminated with a \$. Define
$$\text{terminated}(t) = (\|t\| > 0 \wedge t(\|t\| - 1) = \$)$$
- $\text{terminated}(t')$ will be a postcondition
- If f is false, f' should also be false.
- Either f' is false or the procedure should remove a prefix from t that derives from A.
  - ∗ Consequently, if there is no prefix of t that derives from A, then f' must be false.
  - ∗ Note that, if f' is an error is detected, the procedure can make any change to t (except removing the final \$). Consider a call to E1 when the input is $[+,), \$]$; procedure E1 will remove the + and set f to false.
- If f is true and there is a prefix of t that could lead to an overall success then such a u should be picked and f' must not be false.
  - ∗ Example. Consider a call to E1 when the remaining input is $[+, \mathbf{n}, \$]$. Although $E1 \overset{*}{\Longrightarrow} \epsilon$, it is important that the E1 procedure does not simply return. The production $E1 \longrightarrow \epsilon$ can not lead to success when the next item of input is a +. But the production $E1 \longrightarrow +\, T\, E1$ can.
  - ∗ A prefix u can lead to overall success if for some

strings of terminals s and v

$$A_{\mathsf{start}} \overset{*}{\Longrightarrow} sAv \overset{*}{\Longrightarrow} suv \text{ and } t = uv\$$$

- In every other case (f is true, there is a prefix that derives from A, but no choice of prefix can lead to success), it does not matter whether an error is reported or not.
  - ∗ This point means that our procedure is allowed to not fail even when failure is inevitable. Consider a call to F when the remaining input is $[\mathbf{n}, (, \$]$. Although failure of the main procedure is inevitable, it is not the job of the F routine to detect this error.
  - ∗ A subtler example is a call to E1 when the remaining input is $[\mathbf{n}, \$]$. The implementation above detects an error. But it would also be acceptable for E1 to do nothing. The error will eventually be reported regardless.

So the specification of the procedure for A is

$$\langle \text{terminated}(t) \rangle \Rightarrow g_0 \wedge g_1 \wedge g_2 \wedge g_3$$

where

$$g_0 = \langle \text{terminated}(t') \rangle$$

$$g_1 = \langle \neg f \Rightarrow \neg f' \rangle = \langle f' \Rightarrow f \rangle$$

$$g_2 = \left\langle \neg f' \vee \left(\exists u \cdot A \overset{*}{\Longrightarrow} u \wedge t = ut' \right) \right\rangle$$

$$g_3 = \left\langle \begin{array}{l} f \wedge \left(\exists u, s, v \cdot A_{\mathsf{start}} \overset{*}{\Longrightarrow} sAv \overset{*}{\Longrightarrow} suv \wedge t = uv\$ \right) \\ \Rightarrow f \wedge \left(\exists u, s, v \cdot \begin{array}{l} A_{\mathsf{start}} \overset{*}{\Longrightarrow} sAv \overset{*}{\Longrightarrow} suv \\ \wedge\ t = uv\$ = ut' \end{array} \right) \end{array} \right\rangle$$

Dealing with non LL(1) grammars

Converting to LL(1)

In many cases we can convert an non-LL(1) grammar to an LL(1) grammar.

Factoring

When two productions for a nonterminal start the same way, the nonterminal will not be LL(1).

Example

$$\begin{array}{rcl} \mathit{Stat} & \longrightarrow & \mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Stat}\ \mathbf{end} \\ \mathit{Stat} & \longrightarrow & \mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Stat}\ \mathbf{else}\ \mathit{Stat}\ \mathbf{end} \\ \mathit{Stat} & \longrightarrow & \mathbf{other} \end{array}$$

The terminal **if** will be in the selector sets of the first two productions.

We can factor out the common left parts to get:

$$\begin{array}{rcl} \mathit{Stat} & \longrightarrow & \mathbf{if}\ \mathit{Exp}\ \mathbf{then}\ \mathit{Stat}\ \mathit{MoreIf} \\ \mathit{Stat} & \longrightarrow & \mathbf{other} \\ \mathit{MoreIf} & \longrightarrow & \mathbf{else}\ \mathit{Stat}\ \mathbf{end} \\ \mathit{MoreIf} & \longrightarrow & \mathbf{end} \end{array}$$

Eliminating left recursion

Consider a sequence of one or more items separated by commas

$$\begin{array}{lll} List & \longrightarrow & List \;\; , \; \mathbf{item} \\ List & \longrightarrow & \mathbf{item} \end{array}$$

We can replace these rules with equivalent productions

$$\begin{array}{lll} List & \longrightarrow & \mathbf{item}\ List' \\ List' & \longrightarrow & ,\ \mathbf{item}\ List' \mid \epsilon \end{array}$$

In general you can eliminate direct left-recursion by replacing rules

$$A \longrightarrow A\alpha_0 \mid A\alpha_1 \mid \beta_0 \mid \beta_1$$

with rules

$$\begin{array}{lll} A & \longrightarrow & \beta_0 A' \mid \beta_1 A' \\ A' & \longrightarrow & \alpha_0 A' \mid \alpha_1 A' \mid \epsilon \end{array}$$

There exist methods for eliminating indirect left-recursion, e.g.:

$$A \longrightarrow B\alpha \mid \beta \qquad B \longrightarrow A\gamma \mid \delta$$

Recursive Descent or TDPP parsing with non-LL(1) grammars

Ambiguous else

The above methods will convert many grammars to LL(1) form.

But not all.

In fact there exist languages for which there exists no LL(1) grammar.

Consider if statements in C/C++/Java/Pascal

$$\begin{aligned} \textit{Stat} &\longrightarrow \mathbf{if}\ (\ E\ )\ \textit{Stat MoreIf} \mid \ldots \\ \textit{MoreIf} &\longrightarrow \mathbf{else}\ \textit{Stat} \mid \epsilon \\ E &\longrightarrow \ldots \end{aligned}$$

(This is not LL(1) since **else** is in $\mathrm{Follow}(\textit{Stat})$ and hence in $\mathrm{Follow}(\textit{MoreIf})$ and hence in the selector sets for both productions for *MoreIf*)

You can still use recursive descent or top-down predictive parsing, in this case, but you have to use a means other than the selector set to pick the production rule.

For example in the “ambiguous else” example, the parser should (according to the rules of languages like C) pick the first production rule for *MoreIf* when the next terminal is **else**.

Syntactic lookahead

Another example comes from the Turing Programming Language. In Turing, both of these are statements

```
a(1,2,3) := 10
```

and

```
a(1,2,3)
```

The first is an assignment to an array, while the second is a call to a procedure. The relevant part of the grammar

looks like this

$$\begin{aligned} Stat &\longrightarrow LHS := E \\ Stat &\longrightarrow Call \\ Call &\longrightarrow \textbf{id}\ OptParamList \\ LSH &\to \textbf{id}\ OptIndexList \end{aligned}$$

We might be able to solve this using left-factoring, but, for parsing, we likely want a different sort of result from a LHS versus a $Call$, so left-factoring is not a good alternative.

A recursive descent parser could look ahead in the input stream to see if there is a := after what looks like a LHS.

procedure Stat **is**
    if $t(0) =$ id **then**
        if LookAhead **then**
            LHS ; expect(“**:=**”) ; E
        else
            Call
        end if
    else ...

procedure LookAhead **is**
    // return true if a prefix of t derives from $LHS :=$
    // otherwise return false

Semantic lookahead

Semantic lookahead uses information beyond the sequence of tokens.

E.g., in the C++ language a statement "foo (a) ;" could be either

- an expression statement —the expression is a call to function foo— or
- the declaration of a variable named a of type foo.

Which is the correct parse depends on whether foo has been declared to be a type or not using a typedef, enum, struct, or class declaration. For example, here it is a declaration.

{ **typedef int** foo ; . . . foo (a) ; . . . }

The relevant part of the grammar is

$$Statement \longrightarrow Declaration$$
$$Statement \longrightarrow Expression\ ;$$

The terminal $\mathbf{id}$ is in the selector sets of both productions.

When parsing C++, we keep a table of all declared types and consult the table when a choice needs to be made

procedure Statement() **is**
...
else if $t(0) \in \{\mathbf{int}, \mathbf{float}, \ldots\}$
$\vee\ t(0) = id \wedge$ the text of $t(0)$ currently represents a type **then**
Declaration
else if $t(0) \in \{\mathbf{id}, +, -, \ldots\}$ **then**
Expression ; expect(';')
else ...

Bottom-up, Shift-Reduce Parsing

We don't need (or use) augmented grammars for this.

State

In this parsing method the state is $\alpha_\wedge t$ where

- α is a stack (top is at *right*) representing consumed input and
- t is the remaining input

We initialize the state to $\epsilon_\wedge w\$$, where w is the original input, where w is the original input and **$** is a symbol not in S

Invariant: $\alpha t \overset{*}{\Longrightarrow} w\$$ I.e. there is a derivation from αt to $w\$$.

Steps

There are two kinds of steps

- Shift steps: $\alpha_\wedge au \quad \vdash_{\text{bu}} \quad \alpha a_\wedge u$
- Reduce steps: $\beta\gamma_\wedge t \quad \vdash_{\text{bu}} \quad \beta A_\wedge t$
  where $A \longrightarrow \gamma$ is a production rule

Stops

- State $A_{\text{start}\wedge}\$$ means success
- If neither a shift nor a reduce can lead to a successful parse, then an error is declared.

Example using grammar Exp1

$$E \longrightarrow T \mid E \,\textbf{+}\, T \mid E \,\textbf{-}\, T$$
$$T \longrightarrow F \mid T \,\textbf{*}\, F \mid T \,\textbf{/}\, F$$
$$F \longrightarrow \textbf{n} \mid \textbf{(}\, E \,\textbf{)}$$

	$\alpha_\wedge t\$$	**Action**
	$\epsilon_\wedge$ **n - n * n - n $**	Shift
$\vdash_{\rm bu}$	**n** $_\wedge$ **- n * n - n $**	Reduce $F \longrightarrow \mathbf{n}$
$\vdash_{\rm bu}$	F $_\wedge$ **- n * n - n $**	Reduce $T \longrightarrow F$
$\vdash_{\rm bu}$	T $_\wedge$ **- n * n - n $**	Reduce $E \longrightarrow T$
$\vdash_{\rm bu}$	E $_\wedge$ **- n * n - n $**	Shift
$\vdash_{\rm bu}$	E **-** $_\wedge$ **n * n - n $**	Shift
$\vdash_{\rm bu}$	E **- n** $_\wedge$ *** n - n $**	Reduce $F \longrightarrow \mathbf{n}$
$\vdash_{\rm bu}$	E **-** F $_\wedge$ *** n - n $**	Reduce $T \longrightarrow F$
$\vdash_{\rm bu}$	E **-** T $_\wedge$ *** n - n $**	Shift
$\vdash_{\rm bu}$	E **-** T ***** $_\wedge$ **n - n $**	Shift
$\vdash_{\rm bu}$	E **-** T *** n** $_\wedge$ **- n $**	Reduce $F \longrightarrow \mathbf{n}$
$\vdash_{\rm bu}$	E **-** T ***** F $_\wedge$ **- n $**	Reduce $T \longrightarrow T$ ***** F
$\vdash_{\rm bu}$	E **-** T $_\wedge$ **- n $**	Reduce $E \longrightarrow E$ **-** T
$\vdash_{\rm bu}$	E $_\wedge$ **- n $**	Shift
$\vdash_{\rm bu}$	E **-** $_\wedge$ **n $**	Shift
$\vdash_{\rm bu}$	E **- n** $_\wedge$**$**	Reduce $F \longrightarrow \mathbf{n}$
$\vdash_{\rm bu}$	E **-** F $_\wedge$**$**	Reduce $T \longrightarrow F$
$\vdash_{\rm bu}$	E **-** T $_\wedge$**$**	Reduce $E \longrightarrow E$ **-** T
$\vdash_{\rm bu}$	$E_\wedge$**$**	

Notice how this traces out a (right-most) derivation in reverse.

LR(1) grammars and Parser Generators

Definition: If you can always pick the correct step on the basis of

- the current stack, and
- the first terminal in the remaining input

then the grammar is said to be LR(1)

Theorem (Knuth): This decision can be made by running a deterministic finite state machine on the stack and then basing the decision on the final state of that machine and the next terminal.

Theorem (Knuth): LR(1) grammars can be parsed in O(N) time

Proof idea: Represent the stack of symbols with a stack of finite machine states (this is a data refinement). Then only the top state on this stack and the next input symbol need to be consulted, not the whole stack.

Theorem: All LL(1) grammars are LR(1).

Why: LL(1) parsers must decide the production rule for A on the basis of the first symbol after the *start* of A. An LR(1) parser must decide the production rule for A on the basis of the first symbol after the *end* of A. Thus an

LR(1) parser has at least as much information on which to base a decision.

Implementing an LR(1) parser without tool support is not easy for nontrivial grammars.

The “*yacc*” and “*bison*” parser generators use shift-reduce parsing and can handle almost all LR(1) grammars.

yacc and bison produce parsers written in C.

Deterministic shift-reduce parsing

Algorithm: Deterministic shift-reduce parsing

Input: a string w

Output: 'error' or 'success'

var $t := w\$\cdot$ *// where w is the input string*
var $\alpha := \epsilon\cdot$ *// Note: α behaves as a stack.*
while $\alpha \neq A_{\text{start}} \vee t \neq [\$]$ **do** (
    var $q :=$ decide what to do $\cdot$
    if $q = \text{shift}$ **then** (
        $\alpha := (\alpha\,\hat{}\,t(0))$;
        $t := \text{tail}(t)$)
    else if $q = \text{reduce}(A \to \gamma)$ **then**
        let β be such that $\alpha = \beta\gamma\cdot$
        $\alpha := \beta A$
    else /* $q = \text{error}$ */
        error)
success

The tricky bit is deciding what to do next: There are three possibilities

- shift
- reduce($A \to \gamma$) where $A \to \gamma$ is a production and γ is the top of the stack.
- error

Extended CFGs (or EBNF)

Terminological aside:

- Context-Free Grammars were invented by *Noam Chomsky* in 1957 in the study of natural languages.
- *John Backus* invented an equivalent formalism for describing programming languages.
- *Peter Naur* used Backus's notation in the description of Algol-60.
- Thus CFG notation is often called BNF for "Backus-Naur Form".
- Extended CFGs are often called Extended BNF (EBNF).
- Augmented BNF (ABNF) is a standardized form of EBNF used a lot in internet standards.

Back to Extended CFGs (ECFGs, EBNFs, ABNFs)

We extend CFG notation with convenience notations.

These do not extend range of languages we can describe.

Each production rule now has the form

$$\textit{NonTerminal} \longrightarrow \textit{RegularExpression}$$

where a regular expression is one of the following (where x and y are smaller regular expressions).

An epsilon: ϵ

A terminal: a

A nonterminal: A

A parenthesized regular expression: (x)

A repetition regular expression: x^*

A sequence regular expression: $x; y$

A choice regular expression: $x \mid y$

These have the following interpretations:

ϵ describes the string ϵ

a describes the string a

A describes all strings s such that $A \overset{*}{\Longrightarrow} s$

(x) describes the same language as x

x^* describes any string of the form $s_0\hat{}s_1\hat{}...\hat{}s_{n-1}$ where x describes each s_i and $n \geq 0$

$x; y$ describes any language of the form $s\hat{}t$ where x describes s and y describes t

$x \mid y$ describes any language described by either x or y

Note that in x^*, 0 repetitions are allowed.

For example we can write Exp3 (equivalent to Exp0, Exp1, Exp2) as

$$E \longrightarrow E; ((\text{‘+’} \mid \text{‘-’} \mid \text{‘*’} \mid \text{‘/’});\ E)^* \mid \text{‘(’}\ E\ \text{‘)’} \mid \text{‘n’}$$

Precedence clarification: Alternation has lower precedence than juxtaposition, which has lower precedence than repetition.

Some common abbreviations:

$x\,y$ abbreviates $x; y$.

$x^?$ abbreviates $(x \mid \epsilon)$ I.e. an optional x

x^+ abbreviates $x^*; x$ I.e. one or more repetitions of x

$[a - b]$, where s and t are members of a sequence, abbreviates a choice of any of the terminals in the sequence. For example $[\mathbf{0} - \mathbf{9}]$ is a digit and $[\mathbf{a} - \mathbf{z}]$ is any lower-case letter.

Revisiting the C++ Floating Number grammar. We can now be more concise.

$$\begin{aligned} \textit{floatNum} &\longrightarrow \textit{fract exp}^? \,(\text{‘}\mathbf{f}\text{’} \mid \text{‘}\mathbf{l}\text{’} \mid \text{‘}\mathbf{F}\text{’} \mid \text{‘}\mathbf{L}\text{’})^? \\ &\quad\mid \quad \textit{int exp}\,(\text{‘}\mathbf{f}\text{’} \mid \text{‘}\mathbf{l}\text{’} \mid \text{‘}\mathbf{F}\text{’} \mid \text{‘}\mathbf{L}\text{’})^? \\ \textit{int} &\longrightarrow [\text{‘}\mathbf{0}\text{’} - \text{‘}\mathbf{9}\text{’}]^+ \\ \textit{fract} &\longrightarrow \textit{int}\,\text{‘.’}\,\textit{int} \mid \textit{int}\,\text{‘.’} \mid \text{‘.’}\,\textit{int} \\ \textit{exp} &\longrightarrow (\text{‘}\mathbf{e}\text{’} \mid \text{‘}\mathbf{E}\text{’})\,(\text{‘}+\text{’} \mid \text{‘}-\text{’} \mid \epsilon)\,\textit{int} \end{aligned}$$

Extended CFG is no more powerful than CFG

Given an ECFG grammar, we can rewrite its productions to obtain a CFG for the same language:

Algorithm: Apply the following replacements until the grammar is a CFG..

$A \rightarrow (x)$	replace with	$A \rightarrow x$
$A \rightarrow x^*$	replace with	$A \rightarrow A1$
		$A1 \rightarrow x; A1$
		$A1 \rightarrow \epsilon$
$A \rightarrow x; y$	replace with	$A \rightarrow A1\ A2$
		$A1 \rightarrow x$
		$A2 \rightarrow y$
$A \rightarrow x \mid y$	replace with	$A \rightarrow x$
		$A \rightarrow y$

where $A1$ and $A2$ are new nonterminals.

Recursive Descent Parsing with ECFGs

We can implement repetition with a while command and choice with an if command.

Consider this augmented grammar for expressions Exp4

$$\begin{aligned} E' &\longrightarrow E\,\$ \\ E &\longrightarrow T\ (\text{‘}\mathbf{+}\text{’}\ T \mid \text{‘}\mathbf{-}\text{’}\ T)^* \\ T &\longrightarrow F\ (\text{‘}\mathbf{*}\text{’}\ F \mid \text{‘}\mathbf{/}\text{’}\ F)^* \\ F &\longrightarrow \text{‘}\mathbf{n}\text{’} \mid \text{‘}\mathbf{(}\text{’}\ E\ \text{‘}\mathbf{)}\text{’} \end{aligned}$$

The choices implicit in the repetitions can be made on the basis of the next token since:

- neither + nor - are in the Follow set of E
- neither * nor / are in the Follow set of T

We can implement E with a subroutine

procedure E is
  var p:= T ·
  while $t(0) \in \{$‘**+**’,‘**-**’$\}$ **do**
    if $t(0) =$ ‘**+**’ **then** (
      t:= $\text{tail}(t)$;
      var q := T ·
      p:= $p + q$)
    else (
      t := $\text{tail}(t)$;
      var q:= T ·
      p := $p - q$)
  return p

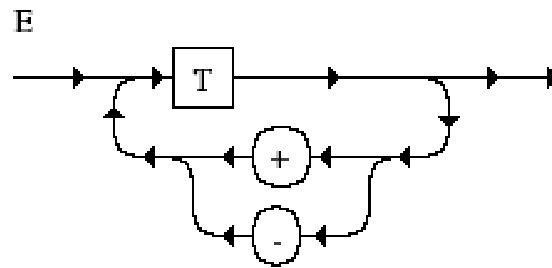
Note that I didn’t bother to check that $t(0)$ is ‘**)**’ or **$** prior to returning, since the caller of E will presumably make that check and can provide a better error message.

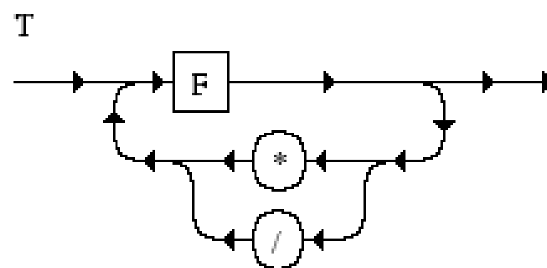
Parser Generation: The *JavaCC* parser generator accepts ECFG grammars and produces parsers written in Java. (https://javacc.dev.java.net/)

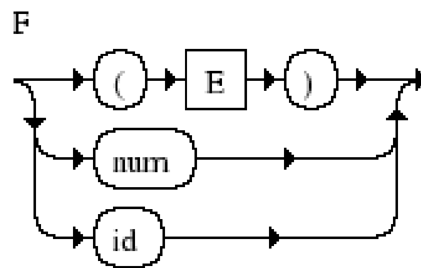
The ANTLR parser generator is similar.

Syntax diagrams (or railroad diagrams)

Syntax diagrams are similar to ECFGs except, instead of using regular expressions, we use NDFRs. These NDFRs are conventionally drawn with networks of lines representing the states and boxes representing the transitions. For example:







Regular languages

Parsing with LL(1) and LR(1) grammars takes

- $O(N)$ time and (worst case)
- $O(N)$ space. (N is the length of the input)

Regular languages are those that take $O(1)$ space.

We can define regular languages in terms of grammars one of several equivalent ways

First way

A regular language is one that can be described by an ECFG grammar with no recursion (direct or indirect) between the nonterminals

For example the syntax of floating point numbers in C++.

Example:

$$M \longrightarrow \mathbf{\$}DD^?D^?(\mathbf{,}DDD)^*\mathbf{.}DD$$
$$D \longrightarrow \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Counter-example:

$$E \longrightarrow (\mathbf{n}\ ((\mathbf{+} \mid \mathbf{-} \mid \mathbf{*} \mid \mathbf{/})\ \mathbf{n})^*) \mid \text{‘}\mathbf{(}\text{’}\ E\ \text{‘}\mathbf{)}\text{’}$$

has recursion.

Second way

A production rule is right linear if it is of the form

$$A \longrightarrow sB$$

or of the form

$$A \longrightarrow s$$

(recall that s contains no nonterminal).

A regular language is one that can described by a CFG containing only right-linear productions.

Example

$$\begin{array}{rcl} A & \longrightarrow & \mathbf{\$}\,B \\ B & \longrightarrow & \mathbf{d}\,C \\ C & \longrightarrow & D \mid \mathbf{d}\,D \\ D & \longrightarrow & E \mid \mathbf{d}\,E \\ E & \longrightarrow & ,\mathbf{d}\,\mathbf{d}\,\mathbf{d}\,E | \mathbf{.}\,\mathbf{d}\,\mathbf{d} \end{array}$$

Counter-example:

$$E \longrightarrow \mathbf{n} \mid \text{‘}\mathbf{(}\text{’}\ E\ \text{‘}\mathbf{)}\text{’}$$

Recursion of E is not at the very right of the production rule

Attribute grammars

Attribute grammars augment terminals and nonterminals with attributes.

Each production has an boolean expression that must be satisfied at each node in the derivation tree.

Recognition by "dynamic programming"

Input: A string w and a grammar $G = (V, S, P, A_{\text{start}})$ such that every production is in one of the following forms

$$A \longrightarrow a$$
$$A \longrightarrow B\,C$$

(Exercise. Show that any grammar such that $\epsilon \notin L(G)$ can be transformed to an equivalent grammar that satisfies this constraint.)

Let n be the length of w.

var $m : \mathbf{array}\ \{0, .., n\} \times \{0, ..n\}\ \mathbf{of}\ \mathcal{P}(V)$
// Each element of m is a set of nonterminals
for $(i, j) \in \{0, .., n\} \times \{0, ..n\}$ **do** $m(i, j) := \emptyset$

The idea is to put into each element $m(i, j)$ all nonterminals that describe the substring $w[i, .., j]$ where $i < j$.

for $i \in \{0, ..n\}$ **do** $m(i, i + 1) := \{A \mid A \longrightarrow w(i) \in P\}$

So far we have succeeded for substrings of length 1.

We can 'multiply' two sets of nonterminals U and V as follows: if $B \in U$ and $C \in V$ and $A \longrightarrow B\,C$ is a production then $A \in U \otimes V$. I.e.

$U \otimes V = \{A, B, C \mid (A \longrightarrow B\,C) \in P \wedge B \in U \wedge C \in V \cdot A\}$

If s and t are strings and U contains all nonterminals B such that $B \stackrel{*}{\Longrightarrow} s$ and V contains all nonterminals C such that $C \stackrel{*}{\Longrightarrow} t$, then $U \otimes V$ contains all nonterminals A such that $A \stackrel{*}{\Longrightarrow} s \hat{\ } t$.

The rest of the algorithm fills in the table for segments of increasing length.

> **for** k **from** 2 **to** n **do**
>
>   **for** $i \in \{0, .., n - k\}$ **do**
>
>   **let** $j := i + k \cdot$
>
>   **for** $\ell \in \{i + 1, .., j - 1\}$ **do**
>
> $m(i, j) := m(i, j) \cup (m(i, \ell) \otimes m(\ell, j))$

Upon completion, the string is in the language iff $A_{\mathrm{start}} \in m(0, n)$.