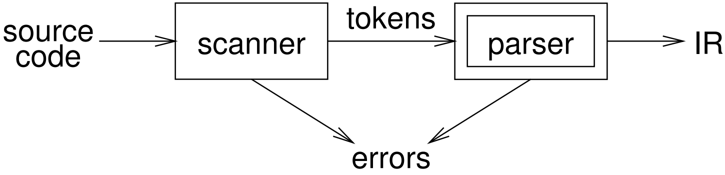
CS3300 - Compiler Design

Parsing

KC Sivaramakrishnan

IIT Madras

The role of the parser



A parser

- performs context-free syntax analysis
- guides context-sensitive analysis
- constructs an intermediate representation
- produces meaningful error messages
- attempts error correction

For the next several classes, we will look at parser construction

Syntax analysis by using a CFG

Context-free syntax is specified with a *context-free grammar*.
Formally, a CFG G is a 4-tuple (V_t, V_n, S, P), where:

- V_t is the set of *terminal* symbols in the grammar.
  For our purposes, V_t is the set of tokens returned by the scanner.
- V_n, the *nonterminals*, is a set of syntactic variables that denote sets of (sub)strings occurring in the language.
- S is a distinguished nonterminal ($S \in V_n$) denoting the entire set of strings in $L(G)$.
  This is sometimes called a *goal symbol*.
- P is a finite set of *productions*
  Each production must have a single non-terminal on its left hand side.

The set $V = V_t \cup V_n$ is called the *vocabulary* of G.

Notation and terminology

- $a, b, c, \ldots \in V_t$
- $A, B, C, \ldots \in V_n$
- $U, V, W, \ldots \in V$
- $\alpha, \beta, \gamma, \ldots \in V^*$
- $u, v, w, \ldots \in V_t^*$

If $A \to \gamma$ then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is a *single-step derivation* using $A \to \gamma$

Similarly, $\Rightarrow^*$ and $\Rightarrow^+$ denote derivations of ≥ 0 and ≥ 1 steps

If $S \Rightarrow^* \beta$ then β is said to be a *sentential form* of G

$L(G) = \{w \in V_t^* \mid S \Rightarrow^+ w\}$, $w \in L(G)$ is called a *sentence* of G

Note, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Syntax analysis

Grammars are often written in Backus-Naur form (BNF).
Example:

$$
\begin{array}{r|ccl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
3 & & | & \texttt{num} \\
4 & & | & \texttt{id} \\
5 & \langle\text{op}\rangle & ::= & + \\
6 & & | & - \\
7 & & | & * \\
8 & & | & /
\end{array}
$$

This describes simple expressions over numbers and identifiers.
In a BNF for a grammar, we represent

1. non-terminals with angle brackets or capital letters
2. terminals with `typewriter` font or <u>underline</u>
3. productions as in the example

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG (for $\texttt{x} + \texttt{2} * \texttt{y}$):

$$
\begin{aligned}
\langle\text{goal}\rangle &\Rightarrow \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle + \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle + \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle + \langle\text{num},\texttt{2}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle + \langle\text{num},\texttt{2}\rangle * \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{id},\texttt{x}\rangle + \langle\text{num},\texttt{2}\rangle * \langle\text{id},\texttt{y}\rangle
\end{aligned}
$$

We have derived the sentence $\texttt{x} + \texttt{2} * \texttt{y}$.
We denote this $\langle\text{goal}\rangle\Rightarrow^* \texttt{id} + \texttt{num} * \texttt{id}$.
Such a sequence of rewrites is a *derivation* or a *parse*.
The process of discovering a derivation is called *parsing*.

Derivations

At each step, we chose a non-terminal to replace.
This choice can lead to different derivations.
Two are of particular interest:

leftmost derivation
the leftmost non-terminal is replaced at each step

rightmost derivation
the rightmost non-terminal is replaced at each step

The previous example was a leftmost derivation.

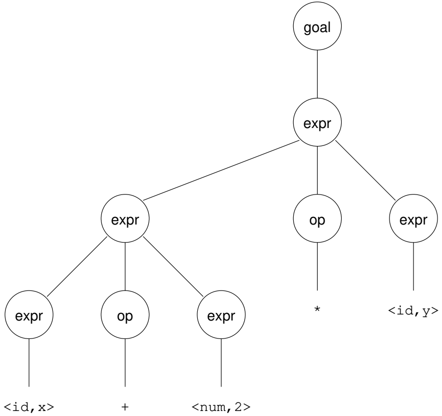
Rightmost derivation

For the string $\texttt{x} + \texttt{2} * \texttt{y}$:

$$
\begin{array}{rcl}
\langle\text{goal}\rangle & \Rightarrow & \langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{id},\texttt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle * \langle\text{id},\texttt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{expr}\rangle * \langle\text{id},\texttt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle\langle\text{op}\rangle\langle\text{num},\texttt{2}\rangle * \langle\text{id},\texttt{y}\rangle \\
& \Rightarrow & \langle\text{expr}\rangle + \langle\text{num},\texttt{2}\rangle * \langle\text{id},\texttt{y}\rangle \\
& \Rightarrow & \langle\text{id},\texttt{x}\rangle + \langle\text{num},\texttt{2}\rangle * \langle\text{id},\texttt{y}\rangle
\end{array}
$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \texttt{id} + \texttt{num} * \texttt{id}$.

Precedence



Treewalk evaluation computes $(\texttt{x} + \texttt{2}) * \texttt{y}$
— the "wrong" answer!
Should be $\texttt{x} + (\texttt{2} * \texttt{y})$

Precedence

These two derivations point out a problem with the grammar.

It has no notion of precedence, or implied order of evaluation.

The grammar is ambiguous, as a string in the language can have multiple parse trees.

Is precedence the only source of ambiguity? Other examples of strings with multiple parse trees?

Ambiguity - Associativity

The expression `a-b-c` may be parsed as:

- `(a-b)-c` or
- `a-(b-c)`

In C, assignment = is right-associative. `a=b=c` may be parsed as:

- `a=(b=c)` or
- `(a=b)=c`

Removing Ambiguity

To remove ambiguity, the grammar needs to be modified:

$$
\begin{array}{r|lcl}
1 & \langle\text{goal}\rangle & ::= & \langle\text{expr}\rangle \\
2 & \langle\text{expr}\rangle & ::= & \langle\text{expr}\rangle + \langle\text{term}\rangle \\
3 & & | & \langle\text{expr}\rangle - \langle\text{term}\rangle \\
4 & & | & \langle\text{term}\rangle \\
5 & \langle\text{term}\rangle & ::= & \langle\text{term}\rangle * \langle\text{factor}\rangle \\
6 & & | & \langle\text{term}\rangle / \langle\text{factor}\rangle \\
7 & & | & \langle\text{factor}\rangle \\
8 & \langle\text{factor}\rangle & ::= & \texttt{num} \\
9 & & | & \texttt{id}
\end{array}
$$

This grammar enforces a *precedence* and *associativity* on the derivation:

- terms *must* be derived from expressions
- forces the “correct” tree

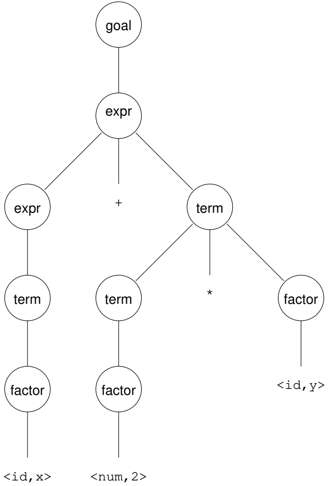
Precedence

Now, for the string $\mathtt{x} + \mathtt{2} * \mathtt{y}$:

$$
\begin{aligned}
\langle\text{goal}\rangle &\Rightarrow \langle\text{expr}\rangle \\
&\Rightarrow \langle\text{expr}\rangle + \langle\text{term}\rangle \\
&\Rightarrow \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{factor}\rangle \\
&\Rightarrow \langle\text{expr}\rangle + \langle\text{term}\rangle * \langle\text{id},\mathtt{y}\rangle \\
&\Rightarrow \langle\text{expr}\rangle + \langle\text{factor}\rangle * \langle\text{id},\mathtt{y}\rangle \\
&\Rightarrow \langle\text{expr}\rangle + \langle\text{num},\mathtt{2}\rangle * \langle\text{id},\mathtt{y}\rangle \\
&\Rightarrow \langle\text{term}\rangle + \langle\text{num},\mathtt{2}\rangle * \langle\text{id},\mathtt{y}\rangle \\
&\Rightarrow \langle\text{factor}\rangle + \langle\text{num},\mathtt{2}\rangle * \langle\text{id},\mathtt{y}\rangle \\
&\Rightarrow \langle\text{id},\mathtt{x}\rangle + \langle\text{num},\mathtt{2}\rangle * \langle\text{id},\mathtt{y}\rangle
\end{aligned}
$$

Again, $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$, but this time, we build the desired tree.

Precedence



Treewalk evaluation computes $\texttt{x} + (\texttt{2} * \texttt{y})$

Role of CFGs in Compilers

CFGs offer significant advantages for language designers, compiler developers, and end-users of the compiler:

- A grammar gives a formal, precise, yet easy-to-understand syntactic specification of the programming languages. Useful for end-users
- For certain classes of grammars, there are procedures to automatically construct efficient parsers from the grammar description. Useful for compiler developers
- A grammar can reveal syntactic ambiguities and trouble spots. Useful for language designers
- A grammar imparts structure to a program, which is directly used for its translation into object code. Useful for compiler developers
- A grammar allows a language to be evolved iteratively by adding new constructs. Useful for language designers and compiler developers

Ambiguity

If a grammar has more than one derivation for a single sentential form, then it is *ambiguous*

Example:

$$\begin{array}{lcl} \langle\text{stmt}\rangle & ::= & \texttt{if}\ \langle\text{expr}\rangle\texttt{then}\ \langle\text{stmt}\rangle \\ & | & \texttt{if}\ \langle\text{expr}\rangle\texttt{then}\ \langle\text{stmt}\rangle\texttt{else}\ \langle\text{stmt}\rangle \\ & | & \texttt{other} \end{array}$$

Consider deriving the sentential form:

$$\texttt{if}\ E_1\ \texttt{then}\ \texttt{if}\ E_2\ \texttt{then}\ S_1\ \texttt{else}\ S_2$$

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

We would like to parse `if-then-else` statements using the following rule:

match each `else` *with the closest unmatched* `then`

Grammar which eliminates the ambiguity by following the above rule:

$$
\begin{array}{lcl}
\langle \text{stmt} \rangle & ::= & \langle \text{matched} \rangle \\
 & \mid & \langle \text{unmatched} \rangle \\
\langle \text{matched} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{matched} \rangle\ \texttt{else}\ \langle \text{matched} \rangle \\
 & \mid & \texttt{other} \\
\langle \text{unmatched} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle \\
 & \mid & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{matched} \rangle\ \texttt{else}\ \langle \text{unmatched} \rangle
\end{array}
$$

Ambiguity

Ambiguity is often due to confusion in the context-free specification. Context-sensitive confusions can arise from *overloading*.
Example:

```
a = b + c
```

In many languages, $+$ can refer to both integer addition and floating point addition. Disambiguating this statement requires context:

- need *values* of declarations
- not *context-free*
- really an issue of *type*

Rather than complicate parsing, we will handle this separately.

Scanning vs. parsing

Where do we draw the line?

$$
\begin{array}{lcl}
\langle id \rangle & ::= & [\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}]([\mathtt{a}-\mathtt{z}\mathtt{A}-\mathtt{z}] \mid [0-9])^* \\
\langle num \rangle & ::= & 0 \mid [\mathtt{1}-\mathtt{9}][\mathtt{0}-\mathtt{9}]^* \\
\langle op \rangle & ::= & + \mid - \mid * \mid / \\
\langle expr \rangle & ::= & \langle expr \rangle \langle op \rangle \langle expr \rangle \mid \langle id \rangle \mid \langle digit \rangle
\end{array}
$$

Regular expressions are used to classify:

- identifiers, numbers, keywords
- REs are more concise and simpler for tokens than a grammar
- more efficient scanners can be built from REs (DFAs) than grammars

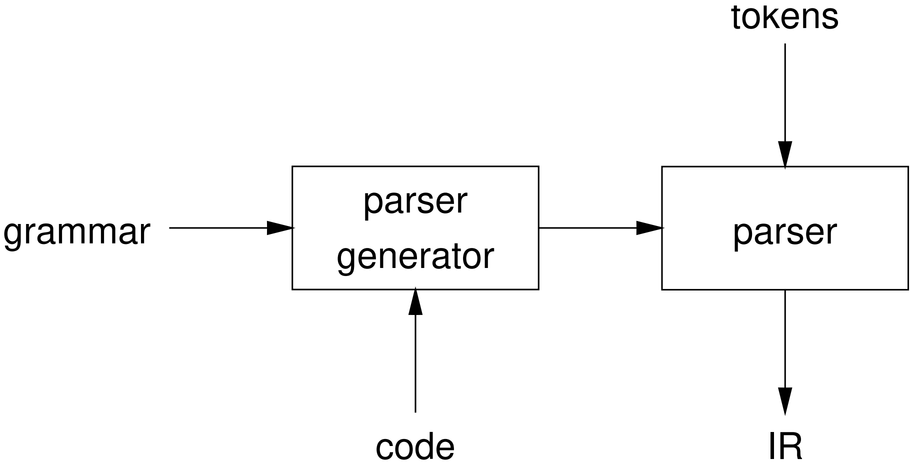
Scanning vs. parsing

Context-free grammars are used to count:

- brackets: `()`, `begin`...`end`, `if`...`then`...`else`
- imparting structure
  - arithmetic expressions can be described by regular expressions
  - but, must deal with precedence and associativity separately ...

Syntactic analysis is complicated enough: grammar for C has around 200 productions. Factoring out lexical analysis as a separate phase makes compiler more manageable.

Parsing: the big picture



Our goal is a flexible parser generator system

Different ways of parsing: Top-down Vs Bottom-up

Top-down parsers

- start at the root of derivation tree and fill in
- picks a production and tries to match the input
- may require backtracking
- some grammars are backtrack-free (*predictive*)

Bottom-up parsers

- start at the leaves and fill in
- start in a state valid for legal first tokens
- as input is consumed, change state to encode possibilities (*recognize valid prefixes*)
- use a stack to store both state and sentential forms

Top-down parsing

A top-down parser starts with the root of the parse tree, labelled with the start or goal symbol of the grammar.

To build a parse, it repeats the following steps until the fringe of the parse tree matches the input string

1. At a node labelled A, select a production $A \rightarrow \alpha$ and construct the appropriate child for each symbol of α
2. When a terminal is added to the fringe that doesn't match the input string, backtrack
3. Find next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1.

Example

$$\begin{array}{lcl}\langle \textit{term} \rangle & ::= & \texttt{id} \mid \texttt{num} \\ \langle \textit{op} \rangle & ::= & + \mid - \\ \langle \textit{expr} \rangle & ::= & \langle \textit{expr} \rangle \langle \textit{op} \rangle \langle \textit{term} \rangle \mid \langle \textit{term} \rangle \end{array}$$

Consider the string `x+5`.

Immediate Left-recursion

Top-down parsers cannot handle left-recursion in a grammar.

Formally, a grammar is *immediate left-recursive* if

$\exists A \in V_n$ *such that* $A \Rightarrow^{+} A\alpha$ *for some string* α

Our simple expression grammar is immediate left-recursive.

Eliminating immediate left-recursion

To remove immediate left-recursion, we can transform the grammar
Consider the grammar fragment:

$$\begin{array}{rcl} \langle\text{foo}\rangle & ::= & \langle\text{foo}\rangle\alpha \\ & | & \beta \end{array}$$

where α and β do not start with $\langle\text{foo}\rangle$
We can rewrite this as:

$$\begin{array}{rcl} \langle\text{foo}\rangle & ::= & \beta\langle\text{bar}\rangle \\ \langle\text{bar}\rangle & ::= & \alpha\langle\text{bar}\rangle \\ & | & \varepsilon \end{array}$$

where $\langle\text{bar}\rangle$ is a new non-terminal

This fragment contains no immediate left-recursion

Eliminating immediate left-recursion

In general, if the grammar contains the following production rules:

$$\langle \mathrm{A} \rangle ::= \langle \mathrm{A} \rangle \alpha_1 \mid \langle \mathrm{A} \rangle \alpha_2 \mid \ldots \mid \langle \mathrm{A} \rangle \alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

they can be replaced by the following:

$$\begin{array}{lcl} \langle \mathrm{A} \rangle & ::= & \beta_1 \langle \mathrm{A'} \rangle \mid \beta_2 \langle \mathrm{A'} \rangle \mid \ldots \mid \beta_n \langle \mathrm{A'} \rangle \\ \langle \mathrm{A'} \rangle & ::= & \alpha_1 \langle \mathrm{A'} \rangle \mid \alpha_2 \langle \mathrm{A'} \rangle \mid \ldots \alpha_m \langle \mathrm{A'} \rangle \mid \varepsilon \end{array}$$

Example

Consider the simplified expression grammar:

$$
\begin{array}{rcl}
E & ::= & E + T \mid T \\
T & ::= & \texttt{id} \mid \texttt{num}
\end{array}
$$

After eliminating left-recursion:

$$
\begin{array}{rcl}
E & ::= & TE' \\
E' & ::= & +TE' \mid \varepsilon \\
T & ::= & \texttt{id} \mid \texttt{num}
\end{array}
$$

How much lookahead is needed?

We saw that top-down parsers need to select a production rule at every step, for which we may have to look ahead in the input string

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or CYK algorithms

Fortunately

- large subclasses of CFGs can be parsed with limited lookahead
- most programming language constructs can be expressed in a grammar that falls in these subclasses

Among the interesting subclasses are:

LL(1): **l**eft to right scan, **l**eft-most derivation, **1**-token lookahead; and

LR(1): **l**eft to right scan, reversed **r**ight-most derivation, **1**-token lookahead

Predictive parsing

Basic idea:

- For any two productions $A \to \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.
- For some RHS $\alpha \in G$, define FIRST(α) as the set of tokens that appear first in some string derived from α.
  - That is, for some $a \in V_t$, $w \in$ FIRST(α) iff. $\alpha \Rightarrow^* a\gamma$.

Key property:

- Whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like:
  - FIRST$(\alpha) \cap$ FIRST$(\beta) = \phi$
- This would allow the parser to make a correct choice with a lookahead of only one symbol!

Recursive descent parsing and Predictive parsing

- If top-down parsing is performed recursively, it is also called *recursive descent parsing*.
  - To prevent infinite recursion, the grammar should not be left-recursive.
  - In general, may require backtracking if the wrong production rule is picked.
- Top-down parsing with lookahead which ensures that the correct production rule is always picked is called *predictive parsing*.

Recursive descent parsing

A set of procedures, one for each non-terminal.

```
int A()
begin
    foreach production of the form A → X_1 X_2 X_3 ··· X_k do
        for i = 1 to k do
            if X_i is a non-terminal then
                if (X_i() = 0) then
                    backtrack; break; // Try the next production
            else if X_i matches the current input symbol a then
                advance the input to the next symbol;
            else
                backtrack; break; // Try the next production
        if i = k + 1 then
            return 1; // Success
    return 0; // Failure
```

Recursive descent parsing

- Backtracks in general – in practise may not do much.
- How to backtrack?
- A left recursive grammar will lead to infinite loop.

For Predictive Parsing

- For a production $A \to \alpha$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear first in some string derived from α.
  - That is, for some $a \in V_t$, $w \in \text{FIRST}(\alpha)$ iff. $\alpha \Rightarrow^* a\gamma$.
- Whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like
  - $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$

- If the grammar has two productions rules of the form $A \to \alpha\beta_1 \mid \alpha\beta_2$, we cannot directly use predictive parsing.

Left factoring

Some grammars can be transformed by left-factoring to enable predictive parsing.

For each non-terminal A find the longest prefix α common to two or more of its production rules.

if $\alpha \neq \varepsilon$ then replace all of the A productions
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n$
with

$A \rightarrow \alpha A'$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

where A' is a new non-terminal.

Repeat until no two alternatives for a single non-terminal have a common prefix.

Example

There are two non-terminals to left factor:

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle + \langle\text{expr}\rangle \\
 & | & \langle\text{term}\rangle - \langle\text{expr}\rangle \\
 & | & \langle\text{term}\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle * \langle\text{term}\rangle \\
 & | & \langle\text{factor}\rangle / \langle\text{term}\rangle \\
 & | & \langle\text{factor}\rangle
\end{array}
$$

Question: What's different here from the previous similar grammar that we've seen?

Applying the transformation:

$$
\begin{array}{lcl}
\langle\text{expr}\rangle & ::= & \langle\text{term}\rangle\langle\text{expr}'\rangle \\
\langle\text{expr}'\rangle & ::= & +\langle\text{expr}\rangle \\
 & | & -\langle\text{expr}\rangle \\
 & | & \varepsilon
\end{array}
$$

$$
\begin{array}{lcl}
\langle\text{term}\rangle & ::= & \langle\text{factor}\rangle\langle\text{term}'\rangle \\
\langle\text{term}'\rangle & ::= & *\langle\text{term}\rangle \\
 & | & /\langle\text{term}\rangle \\
 & | & \varepsilon
\end{array}
$$

Left-recursion Elimination

- Predictive Parsing is a form of recursive-descent parsing, and hence cannot handle grammars with left recursion.

- We have seen how to eliminate immediate left-recursion, i.e. when there is a production rule of the form $A \rightarrow A\alpha$.
- However, left-recursion can also be indirect.
  - Example: $A \rightarrow B\alpha$ and $B \rightarrow A\beta$.

- In the general case, A grammar is left-recursive if $\exists A \in V_n$ such that $A \Rightarrow^{+} A\alpha$ for some string α.

Indirect Left-recursion Elimination

Given a left-factored CFG, to eliminate left-recursion:

Input: Grammar G with no *cycles* (no $A \Rightarrow^* A$) and no ε productions.

Output: Equivalent grammar with no left-recursion.

begin

  Arrange the non terminals in some order $A_1, A_2, \cdots A_n$;

  foreach $i = 1 \cdots n$ **do**

    foreach $j = 1 \cdots i-1$ **do**

      For production p of the form $A_i \rightarrow A_j\gamma$ and $A_j \rightarrow \delta_1|\delta_2|\cdots|\delta_k$;

      Replace the production p by:

      $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots\ \delta_n\gamma$;

    Eliminate immediate left recursion in A_i;

Indirect Left-recursion Elimination Algorithm Analysis

- At the end of ith iteration of the outer loop, the algorithm ensures that in all productions of the form $A_i \to A_j\gamma$, $i < j$.
- The algorithm assumes that the grammar has no cycles, i.e. $A \Rightarrow^* A$ is not possible for any non-terminal A.

- *Questions to ponder:*
  - What happens if there are cycles in the input grammar?
  - What happens if there are ε-productions in the input grammar?
- Does the algorithm work for all context-free languages?
  - Yes, it works for all CFL which do not contain ε. For any such CFL, we can always obtain a CFG which does not contain ε-productions and unit-productions.

Example

Consider the following grammar:

$$\begin{array}{rcl} \langle \mathrm{S} \rangle & ::= & \langle \mathrm{A} \rangle a \mid b \\ \langle \mathrm{A} \rangle & ::= & \langle \mathrm{S} \rangle d \mid c \end{array}$$

It has indirect left recursion: $\langle \mathrm{S} \rangle \Rightarrow^* \langle \mathrm{S} \rangle da$
Grammar after eliminating left recursion:

$$\begin{array}{rcl} \langle \mathrm{S} \rangle & ::= & \langle \mathrm{A} \rangle a \mid b \\ \langle \mathrm{A} \rangle & ::= & bd\langle \mathrm{A'} \rangle \mid c\langle \mathrm{A'} \rangle \\ \langle \mathrm{A'} \rangle & ::= & ad\langle \mathrm{A'} \rangle \mid \varepsilon \end{array}$$

Generality

Question:

By left factoring and eliminating left-recursion, can we transform an arbitrary context-free grammar to a form where it can be predictively parsed with a single token lookahead?

Answer: No. Example:

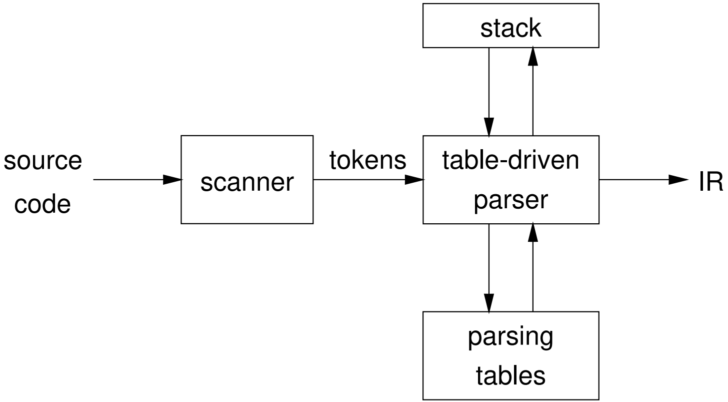
$$\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\}$$

Must look past an arbitrary number of a's to discover the 0 or the 1 and so determine the derivation.

Not all CFG are LL(1).

Non-recursive predictive parsing

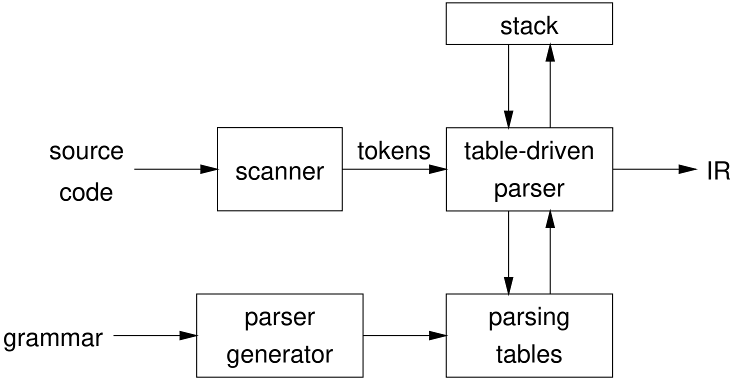
Now, a predictive parser looks like:



Rather than writing recursive code, we build tables.
Building tables can be automated easily.

Table-driven parsers

A parser generator system often looks like:



- We will first look at the information required for generating the parsing table.

FIRST

For a string of grammar symbols α, define FIRST(α) as:

- the set of terminals that begin strings derived from α: $\{a \in V_t \mid \alpha \Rightarrow^* a\beta\}$
- If $\alpha \Rightarrow^* \varepsilon$ then $\varepsilon \in$ FIRST(α)

To build FIRST(X):

1. If $X \in V_t$ then FIRST(X) is $\{X\}$
2. If $X \to \varepsilon$ then add ε to FIRST(X)
3. If $X \to Y_1 Y_2 \cdots Y_k$:
   1. Put FIRST$(Y_1) - \{\varepsilon\}$ in FIRST(X)
   2. $\forall i : 1 < i \leq k$, if $\varepsilon \in$ FIRST$(Y_1) \cap \cdots \cap$ FIRST(Y_{i-1}) (i.e., $Y_1 \cdots Y_{i-1} \Rightarrow^* \varepsilon$) then put FIRST$(Y_i) - \{\varepsilon\}$ in FIRST(X)
   3. If $\varepsilon \in$ FIRST$(Y_1) \cap \cdots \cap$ FIRST(Y_k) then put ε in FIRST(X)

   Repeat until no more additions can be made.

FOLLOW

For a non-terminal A, define FOLLOW(A) as

the set of terminals that can appear immediately to the right of A in some sentential form

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it.
A terminal symbol has no FOLLOW set.
To build FOLLOW(A):

1. Put \$ in FOLLOW($\langle \text{goal} \rangle$)
2. If $A \rightarrow \alpha B \beta$:
    1. Put FIRST(β) $- \{\varepsilon\}$ in FOLLOW(B)
    2. If $\beta = \varepsilon$ (i.e., $A \rightarrow \alpha B$) or $\varepsilon \in$ FIRST(β) (i.e., $\beta \Rightarrow^* \varepsilon$) then put FOLLOW($A$) in FOLLOW($B$)

    Repeat until no more additions can be made

LL(1) grammars

Previous definition

> *A grammar G is LL(1) iff. for all non-terminals A, each distinct pair of productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition* $\text{FIRST}(\beta) \bigcap \text{FIRST}(\gamma) = \phi$.

What if $\varepsilon \in \text{FIRST}(\beta)$?

Consider that the current imput symbol is a. Introduces ambiguity between choosing:

- $A \rightarrow \beta$ when $a \in \text{FOLLOW}(A)$
- $A \rightarrow \gamma$ when $a \in \text{FIRST}(\gamma)$

Ambiguity is bad because we may need to backtrack – not predictive parsing anymore!

LL(1) grammars

Revised definition

A grammar G is LL(1) iff. for each set of productions $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$:

1. $\text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), \ldots, \text{FIRST}(\alpha_n)$ *are all pairwise disjoint*
2. *If $\alpha_i \Rightarrow^* \varepsilon$ then* $\text{FIRST}(\alpha_j) \bigcap \text{FOLLOW}(A) = \phi, \forall 1 \leq j \leq n, i \neq j$.

If G is ε-free, condition 1 is sufficient.

LL(1) grammars

Provable facts about LL(1) grammars:

1. No left-recursive grammar is LL(1)
   - Consider $A \rightarrow A\alpha \mid \beta$. Here, $\text{FIRST}(\beta) \subseteq \text{FIRST}(A)$ (by definition). Also, $\text{FIRST}(A) \subseteq \text{FIRST}(A\alpha)$. We know FIRST sets are never empty. Hence, $\text{FIRST}(\beta) \cap \text{FIRST}(A\alpha) \neq \emptyset$.
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
   - Some CFLs are inherently ambiguous i.e., no unambiguous CFGs exist for that CFL.
4. A grammar which is not LL(1) may be converted into a LL(1) grammar.
   - Consider $S \rightarrow aS \mid a$. Not LL(1) since $\text{FIRST}(aS) = \text{FIRST}(a)$. Use left-factoring to get: $$\begin{aligned} S &\rightarrow aS' \\ S' &\rightarrow aS' \mid \varepsilon \end{aligned}$$ accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G
Output: Parsing table M
Method:

1. $\forall$ productions $A \to \alpha$:
    1. $\forall a \in \textsf{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$
    2. If $\varepsilon \in \textsf{FIRST}(\alpha)$:
        1. $\forall b \in \textsf{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$
        2. If $\$ \in \textsf{FOLLOW}(A)$ then add $A \to \alpha$ to $M[A, \$]$
2. Set each undefined entry of M to `error`

If $\exists M[A, a]$ with multiple entries then grammar is not LL(1).

Example

Our expression grammar:

$$
\begin{array}{llll|llll}
1. & S & \rightarrow E & & 6. & T & \rightarrow FT' \\
2. & E & \rightarrow TE' & & 7. & T' & \rightarrow *T \\
3. & E' & \rightarrow +E & & 8. & & \mid /T \\
4. & & \mid -E & & 9. & & \mid \varepsilon \\
5. & & \mid \varepsilon & & 10. & F & \rightarrow \texttt{num} \\
 & & & & 11. & & \mid \texttt{id}
\end{array}
$$

	FIRST	FOLLOW	id	num	$+$	$-$	$*$	$/$	\$
S									
E									
E'									
T									
T'									
F									
id									
num									
$*$									
$/$									
$+$									
$-$									

Example: Calculating FIRST

$$
\begin{array}{llll|llll}
1. & S & \to E & & 6. & T & \to FT' \\
2. & E & \to TE' & & 7. & T' & \to *T \\
3. & E' & \to +E & & 8. & & \mid /T \\
4. & & \mid -E & & 9. & & \mid \varepsilon \\
5. & & \mid \varepsilon & & 10. & F & \to \texttt{num} \\
 & & & & 11. & & \mid \texttt{id}
\end{array}
$$

$$
\begin{array}{rcl}
\text{FIRST}(E) & \subseteq & \text{FIRST}(S) \\
\text{FIRST}(T) & \subseteq & \text{FIRST}(E) \\
\{+, -, \varepsilon\} & \subseteq & \text{FIRST}(E') \\
\text{FIRST}(F) & \subseteq & \text{FIRST}(T) \\
\{*, /, \varepsilon\} & \subseteq & \text{FIRST}(T') \\
\{\texttt{num}, \texttt{id}\} & \subseteq & \text{FIRST}(F)
\end{array}
$$

Example: Calculating FIRST

1.	S	$\rightarrow E$		6.	T	$\rightarrow FT'$
2.	E	$\rightarrow TE'$		7.	T'	$\rightarrow *T$
3.	E'	$\rightarrow +E$		8.		$\mid /T$
4.		$\mid -E$		9.		$\mid \varepsilon$
5.		$\mid \varepsilon$		10.	F	$\rightarrow \texttt{num}$
				11.		$\mid \texttt{id}$

	FIRST	FOLLOW	id	num	$+$	$-$	$*$	$/$	\$
S	num, id								
E	num, id								
E'	$\varepsilon, +, -$								
T	num, id								
T'	$\varepsilon, *, /$								
F	num, id								
id	id	$-$							
num	num	$-$							
$*$	$*$	$-$							
$/$	$/$	$-$							
$+$	$+$	$-$							
$-$	$-$	$-$							

Example: Calculating FOLLOW

$$
\begin{array}{llll|llll}
1. & S & \rightarrow E & & 6. & T & \rightarrow FT' \\
2. & E & \rightarrow TE' & & 7. & T' & \rightarrow *T \\
3. & E' & \rightarrow +E & & 8. & & \mid /T \\
4. & & \mid -E & & 9. & & \mid \varepsilon \\
5. & & \mid \varepsilon & & 10. & F & \rightarrow \texttt{num} \\
 & & & & 11. & & \mid \texttt{id}
\end{array}
$$

$$
\begin{array}{rcl}
\{\$\} & \subseteq & \mathsf{FOLLOW}(S) \\
\mathsf{FOLLOW}(S) & \subseteq & \mathsf{FOLLOW}(E) \\
\mathsf{FIRST}(E') - \{\varepsilon\} & \subseteq & \mathsf{FOLLOW}(T) \\
\mathsf{FOLLOW}(E) & \subseteq & \mathsf{FOLLOW}(E') \\
\mathsf{FOLLOW}(E) & \subseteq & \mathsf{FOLLOW}(T) \\
\mathsf{FOLLOW}(E') & \subseteq & \mathsf{FOLLOW}(E) \\
\mathsf{FIRST}(T') - \{\varepsilon\} & \subseteq & \mathsf{FOLLOW}(F) \\
\mathsf{FOLLOW}(T) & \subseteq & \mathsf{FOLLOW}(T') \\
\mathsf{FOLLOW}(T) & \subseteq & \mathsf{FOLLOW}(F) \\
\mathsf{FOLLOW}(T') & \subseteq & \mathsf{FOLLOW}(T)
\end{array}
$$

Example: Calculating FOLLOW

1.	S	$\rightarrow E$		6.	T	$\rightarrow FT'$
2.	E	$\rightarrow TE'$		7.	T'	$\rightarrow *T$
3.	E'	$\rightarrow +E$		8.		$\mid /T$
4.		$\mid -E$		9.		$\mid \varepsilon$
5.		$\mid \varepsilon$		10.	F	$\rightarrow$ num
				11.		$\mid$ id

	FIRST	FOLLOW	id	num	$+$	$-$	$*$	$/$	\$
S	num, id	\$							
E	num, id	\$							
E'	$\varepsilon, +, -$	\$							
T	num, id	$+, -,$ \$							
T'	$\varepsilon, *, /$	$+, -,$ \$							
F	num, id	$+, -, *, /,$ \$							
id	id	$-$							
num	num	$-$							
$*$	$*$	$-$							
$/$	$/$	$-$							
$+$	$+$	$-$							
$-$	$-$	$-$							

Example: Calculating the Parsing Table

$$
\begin{array}{llll|llll}
1. & S & \to E & & 6. & T & \to FT' \\
2. & E & \to TE' & & 7. & T' & \to *T \\
3. & E' & \to +E & & 8. & & \mid /T \\
4. & & \mid -E & & 9. & & \mid \varepsilon \\
5. & & \mid \varepsilon & & 10. & F & \to \texttt{num} \\
 & & & & 11. & & \mid \texttt{id}
\end{array}
$$

	FIRST	FOLLOW	id	num	$+$	$-$	$*$	$/$	\$
S	num, id	\$	1	1	–	–	–	–	–
E	num, id	\$	2	2	–	–	–	–	–
E'	$\varepsilon, +, -$	\$	–	–	3	4	–	–	5
T	num, id	$+, -, \$$	6	6	–	–	–	–	–
T'	$\varepsilon, *, /$	$+, -, \$$	–	–	9	9	7	8	9
F	num, id	$+, -, *, /, \$$	11	10	–	–	–	–	–
id	id	–							
num	num	–							
$*$	$*$	–							
$/$	$/$	–							
$+$	$+$	–							
$-$	$-$	–							

Table driven Predictive parsing

Input: A string w and a parsing table M for a grammar G

Output: If w is in $L(G)$, a leftmost derivation of w; otherwise, indicate an error

push \$ onto the stack; push S onto the stack;
let $a =$ first_symbol(w);
X = stack.top();
while $X \neq \$$ **do**
    if $X == a$ **then**
        stack.pop(); **let** $a =$ next_symbol(w);
    else if X *is a terminal* **then**
        error();
    else if $M[X,a]$ *is an error entry* **then**
        error();
    else if $M[X,a] = X \rightarrow Y_1 Y_2 \cdots Y_k$ **then**
        output the production $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        stack.pop();
        push $Y_k, Y_{k-1}, \cdots Y_1$ in that order;
    X = stack.top();

A grammar that is not LL(1)

$$\begin{array}{rcl} \langle \text{stmt} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle \\ & | & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \texttt{else}\ \langle \text{stmt} \rangle \\ & | & \texttt{other} \end{array}$$

Left-factored: $\begin{array}{rcl} \langle \text{stmt} \rangle & ::= & \texttt{if}\ \langle \text{expr} \rangle\ \texttt{then}\ \langle \text{stmt} \rangle\ \langle \text{stmt}' \rangle \mid \texttt{other} \\ \langle \text{stmt}' \rangle & ::= & \texttt{else}\ \langle \text{stmt} \rangle \mid \varepsilon \end{array}$

$$\begin{array}{rcl} \text{FIRST}(\langle \text{stmt}' \rangle) & = & \{\texttt{else}, \varepsilon\} \\ \$ & \in & \text{FOLLOW}(\langle \text{stmt} \rangle) \\ \text{FOLLOW}(\langle \text{stmt} \rangle) & \subseteq & \text{FOLLOW}(\langle \text{stmt}' \rangle) \\ \text{FIRST}(\langle \text{stmt}' \rangle) - \{\varepsilon\} & \subseteq & \text{FOLLOW}(\langle \text{stmt} \rangle) \end{array}$$

Picking the smallest set that can satisfy the constraints gives us:
$\text{FOLLOW}(\langle \text{stmt}' \rangle) \;=\; \{\texttt{else}, \$\}$

Given $\langle \text{stmt}' \rangle \Rightarrow^* \varepsilon$, LL(1) grammar requires
$\text{FIRST}(\texttt{else}\langle \text{stmt} \rangle) \cap \text{FOLLOW}(\langle \text{stmt}' \rangle) = \emptyset$.

A grammar that is not LL(1)

Left-factored: $\langle\text{stmt}\rangle \ ::= \ \texttt{if}\ \langle\text{expr}\rangle\ \texttt{then}\ \langle\text{stmt}\rangle\ \langle\text{stmt}'\rangle \mid \texttt{other}$
$\langle\text{stmt}'\rangle \ ::= \ \texttt{else}\ \langle\text{stmt}\rangle \mid \varepsilon$

Picking the smallest set that can satisfy the constraints gives us:
$\text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}, \$\}$

Given $\langle\text{stmt}'\rangle \Rightarrow^* \varepsilon$, LL(1) grammar requires $\text{FIRST}(\texttt{else}\langle\text{stmt}\rangle) \cap \text{FOLLOW}(\langle\text{stmt}'\rangle) = \emptyset$.

But $\text{FIRST}(\texttt{else}\langle\text{stmt}\rangle) \cap \text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}\}$

The parsing table entry for $M[\langle\text{stmt}'\rangle, \texttt{else}]$ will contain both:

- $\langle\text{stmt}'\rangle ::= \texttt{else}\langle\text{stmt}\rangle$
- $\langle\text{stmt}'\rangle ::= \varepsilon$

Intuitively, prioritise $\langle\text{stmt}'\rangle ::= \texttt{else}\langle\text{stmt}\rangle$ to associate `else` with closest `then`.

Another common example

- Here is a typical example where a programming language fails to be LL(1):

$$\begin{array}{rcl} \langle\text{stmt}\rangle & \rightarrow & \langle\text{assignment}\rangle \mid \langle\text{call}\rangle \mid \langle\text{other}\rangle \\ \langle\text{assignment}\rangle & \rightarrow & \langle\text{id}\rangle = \langle\text{expr}\rangle \\ \langle\text{call}\rangle & \rightarrow & \langle\text{id}\rangle(\langle\text{expr-list}\rangle) \end{array}$$

- This grammar is not in a form that can be left factored. We must first replace assignment and call by the right-hand sides of their defining productions:

$$\langle\text{stmt}\rangle \rightarrow \langle\text{id}\rangle = \langle\text{expr}\rangle \mid \langle\text{id}\rangle(\langle\text{expr-list}\rangle) \mid \langle\text{other}\rangle$$

- We left factor:

$$\begin{array}{rcl} \langle\text{stmt}\rangle & \rightarrow & \langle\text{id}\rangle\langle\text{stmt'}\rangle \mid \langle\text{other}\rangle \\ \langle\text{stmt'}\rangle & \rightarrow & = \langle\text{expr}\rangle \mid (\langle\text{expr-list}\rangle) \end{array}$$

- See how the grammar obscures the language semantics.
  - Most of PL syntax cannot be expressed naturally as LL(1) grammar.

Error recovery in Predictive Parsing

- An error is detected when the terminal on top of the stack does not match the next input symbol or $M[A, a]$ = error.

Panic mode error recovery

- Skip input symbols till a “synchronizing” token appears.

Q: How to identify a synchronizing token?
Some heuristics:

- All symbols in FOLLOW(A) in the synchronizing set for the non-terminal A.
  - For example, while parsing `id *+ id`, after parsing $\star$, T will on the top of the stack. This will lead to error, since $M[T, +]$ is empty. Since $+ \in FOLLOW(T)$, we consider + as a synchronizing token. T will be removed from top of the stack, and parsing can proceed.
- Semicolon after a Stmt production: assignmentStmt; assignmentStmt;
- If a terminal on top of the stack cannot be matched? –
  - pop the terminal.
  - issue a message that the terminal was inserted.