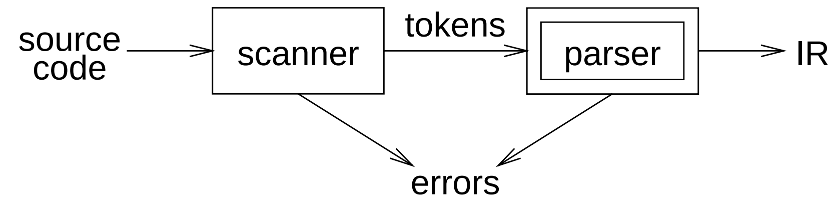
CS502: Parsing (Chapter 3)

The role of the parser



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

For the next few weeks, we will look at parser construction

Syntax analysis

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 is a set of syntactic variables (the *nonterminals*) that denote sets of (sub)strings occurring in the language. These are used to impose a structure on the grammar.

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* specifying how terminals and non-terminals can be combined to form strings in the language. Each production must have a single non-terminal on its LHS.

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 $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha\gamma\beta$ is a *1-step derivation* using $A \rightarrow \gamma$

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

Backus-Naur Form (BNF) for grammars

Example: simple expressions over numbers and identifiers

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

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

Scanning vs. parsing

Where do we draw the line?

$$
\begin{array}{lcl}
\mathit{term} & ::= & [\mathtt{a}-\mathtt{zA}-\mathtt{z}]([\mathtt{a}-\mathtt{zA}-\mathtt{z}] \mid [0-9])^* \\
& \mid & 0 \mid [1-9][0-9]^* \\
\mathit{op} & ::= & + \mid - \mid * \mid / \\
\mathit{expr} & ::= & (\mathit{term}\ \mathit{op})^*\mathit{term}
\end{array}
$$

Scanning vs. parsing

Regular expressions are used to classify:

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

Context-free grammars are used to count:

- brackets: `()`, `begin`...`end`, `if`...`then`...`else`
- imparting structure: expressions

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

Derivations

We can view the productions of a CFG as rewriting rules.
Using our example CFG:

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

We have derived the sentence $\mathtt{x} + 2 * \mathtt{y}$.
We denote this derivation $\langle\text{goal}\rangle \Rightarrow^* \mathtt{id} + \mathtt{num} * \mathtt{id}$.
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.

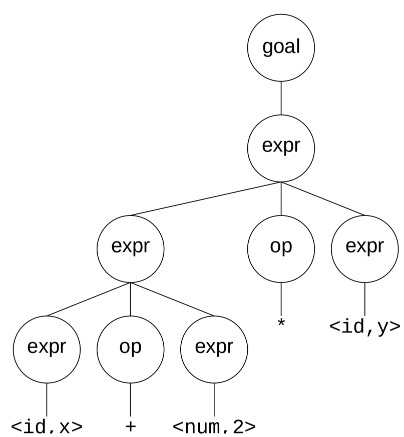
Rightmost derivation

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

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

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

Precedence



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

Precedence

These two derivations point out a problem with the grammar: it has no notion of precedence, or implied order of evaluation.
To add precedence takes additional machinery:

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

Terms *must* derive from expressions, forcing “correct” tree

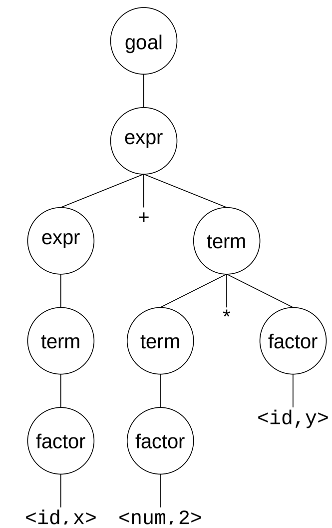
Precedence

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

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

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

Precedence



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

Ambiguity

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

Example:

$$\begin{array}{lll}
\langle\text{stmt}\rangle & ::= & \mathtt{if}\ \langle\text{expr}\rangle\mathtt{then}\ \langle\text{stmt}\rangle \\
 & \mid & \mathtt{if}\ \langle\text{expr}\rangle\mathtt{then}\ \langle\text{stmt}\rangle\mathtt{else}\ \langle\text{stmt}\rangle \\
 & \mid & \mathtt{other\ stmts}
\end{array}$$

Consider deriving the sentential form:

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

It has two derivations.

This ambiguity is purely grammatical.

It is a *context-free* ambiguity.

Ambiguity

May be able to eliminate ambiguities by rearranging the grammar:

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

This generates the same language as the ambiguous grammar, but applies the common sense rule:

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

This is most likely the language designer's intent.

Ambiguity

Ambiguity is often due to confusion in the context-free specification.

Context-sensitive confusions can arise from *overloading*:

```
a = f(17)
```

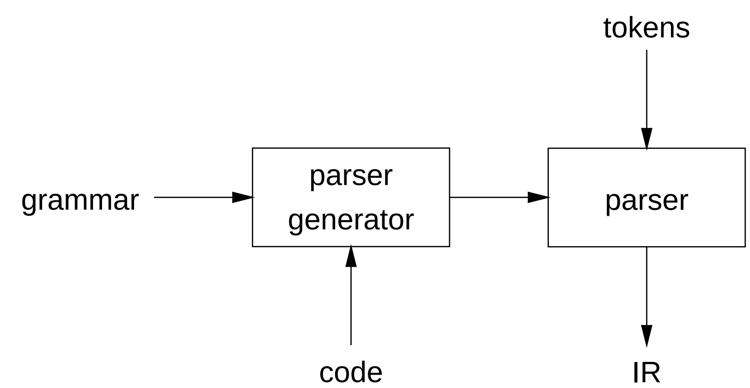
In many Algol-like languages, `f` could be a function or subscripted variable.

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

Parsing: the big picture



Our goal is a flexible parser generator system

Top-down versus 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

$\Rightarrow$ should be guided by input string

Simple expression grammar

Recall our grammar for simple expressions:

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{expr}\rangle + \langle\text{term}\rangle$
3		$\mid$	$\langle\text{expr}\rangle - \langle\text{term}\rangle$
4		$\mid$	$\langle\text{term}\rangle$
5	$\langle\text{term}\rangle$	$::=$	$\langle\text{term}\rangle * \langle\text{factor}\rangle$
6		$\mid$	$\langle\text{term}\rangle / \langle\text{factor}\rangle$
7		$\mid$	$\langle\text{factor}\rangle$
8	$\langle\text{factor}\rangle$	$::=$	$\mathtt{num}$
9		$\mid$	$\mathtt{id}$

Consider the input string $\mathtt{x} - 2 * \mathtt{y}$

Example

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
4	$\langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
7	$\langle\text{factor}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
9	$\mathtt{id} + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} + \langle\text{term}\rangle$	$\mathtt{x}\ \uparrow -\ 2\ *\ \mathtt{y}$
–	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
3	$\langle\text{expr}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
4	$\langle\text{term}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
7	$\langle\text{factor}\rangle - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
9	$\mathtt{id} - \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ \uparrow -\ 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
7	$\mathtt{id} - \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
8	$\mathtt{id} - \mathtt{num}$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num}$	$\mathtt{x}\ -\ 2\ \uparrow *\ \mathtt{y}$
–	$\mathtt{id} - \langle\text{term}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
5	$\mathtt{id} - \langle\text{term}\rangle * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
7	$\mathtt{id} - \langle\text{factor}\rangle * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
8	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ \uparrow 2\ *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ 2\ \uparrow *\ \mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \langle\text{factor}\rangle$	$\mathtt{x}\ -\ 2\ *\ \uparrow\mathtt{y}$
9	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	$\mathtt{x}\ -\ 2\ *\ \uparrow\mathtt{y}$
–	$\mathtt{id} - \mathtt{num} * \mathtt{id}$	$\mathtt{x}\ -\ 2\ *\ \mathtt{y}\ \uparrow$

Example

Another possible parse for $\mathtt{x} - 2 * \mathtt{y}$

Prod'n	Sentential form	Input
–	$\langle\text{goal}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
1	$\langle\text{expr}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \langle\text{term}\rangle$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\langle\text{expr}\rangle + \langle\text{term}\rangle + \cdots$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$
2	$\cdots$	$\uparrow\mathtt{x}\ -\ 2\ *\ \mathtt{y}$

If parser makes wrong choices, expansion doesn't terminate.

This isn't a good property for a parser to have.

(Parsers should terminate!)

Left-recursion

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

Formally, a grammar is *left-recursive* if

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

Our simple expression grammar is left-recursive

Eliminating left-recursion

To remove left-recursion, we can transform the grammar

Consider the grammar fragment:

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

where α and β do not start with $\langle\text{foo}\rangle$

We can rewrite this as:

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

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

This fragment contains no left-recursion

Example

Our expression grammar contains two cases of left-recursion:

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

Applying the transformation yields:

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

With this new grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

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

It is

- right-recursive
- free of ε-productions

Unfortunately, it generates different associativity
Same syntax, different meaning

Example

Our long-suffering expression grammar:

1	$\langle\text{goal}\rangle$	$::=$	$\langle\text{expr}\rangle$
2	$\langle\text{expr}\rangle$	$::=$	$\langle\text{term}\rangle\langle\text{expr}'\rangle$
3	$\langle\text{expr}'\rangle$	$::=$	$+\langle\text{term}\rangle\langle\text{expr}'\rangle$
4		$\mid$	$-\langle\text{term}\rangle\langle\text{expr}'\rangle$
5		$\mid$	ε
6	$\langle\text{term}\rangle$	$::=$	$\langle\text{factor}\rangle\langle\text{term}'\rangle$
7	$\langle\text{term}'\rangle$	$::=$	$*\langle\text{factor}\rangle\langle\text{term}'\rangle$
8		$\mid$	$/\langle\text{factor}\rangle\langle\text{term}'\rangle$
9		$\mid$	ε
10	$\langle\text{factor}\rangle$	$::=$	`num`
11		$\mid$	`id`

Recall, we factored out left-recursion

How much lookahead is needed?

We saw that top-down parsers may need to backtrack when they select the wrong production

Do we need arbitrary lookahead to parse CFGs?

- in general, yes
- use the Earley or Cocke-Younger, Kasami 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, **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 $\text{FIRST}(\alpha)$ as the set of tokens that appear first in some string derived from α.
That is, for some $w \in V_t^*$, $w \in \text{FIRST}(\alpha)$ iff. $\alpha \Rightarrow^* w\gamma$.

Key property:
Whenever two productions $A \to \alpha$ and $A \to \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \phi$$

This would allow the parser to make a correct choice with a lookahead of only one symbol!

The example grammar has this property!

Left factoring

What if a grammar does not have this property?

Sometimes, we can transform a grammar to have this property.

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

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

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

where A' is a new non-terminal.

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

Example

Consider a *right-recursive* (and *right-associative*) version of the expression grammar:

1	$\langle goal\rangle$	::=	$\langle expr\rangle$
2	$\langle expr\rangle$	::=	$\langle term\rangle + \langle expr\rangle$
3		|	$\langle term\rangle - \langle expr\rangle$
4		|	$\langle term\rangle$
5	$\langle term\rangle$	::=	$\langle factor\rangle * \langle term\rangle$
6		|	$\langle factor\rangle / \langle term\rangle$
7		|	$\langle factor\rangle$
8	$\langle factor\rangle$	::=	`num`
9		|	`id`

To choose between productions 2, 3, & 4, the parser must see past the `num` or `id` and look at the $+$, $-$, $*$, or $/$.

$$\text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) \neq \phi$$

This grammar *fails* the test.

Example

There are two non-terminals to left factor:

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

Applying the transformation:

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

Example

Substituting back into the grammar yields

1	$\langle goal\rangle$	::=	$\langle expr\rangle$
2	$\langle expr\rangle$	::=	$\langle term\rangle\langle expr'\rangle$
3	$\langle expr'\rangle$	::=	$+\langle expr\rangle$
4		|	$-\langle expr\rangle$
5		|	ε
6	$\langle term\rangle$	::=	$\langle factor\rangle\langle term'\rangle$
7	$\langle term'\rangle$	::=	$*\langle term\rangle$
8		|	$/\langle term\rangle$
9		|	ε
10	$\langle factor\rangle$	::=	`num`
11		|	`id`

Now, selection requires only a single token lookahead.

Note: *This grammar is still right-associative.*

Example

	Sentential form	Input
–	$\langle goal\rangle$	↑x – 2 * y
1	$\langle expr\rangle$	↑x – 2 * y
2	$\langle term\rangle\langle expr'\rangle$	↑x – 2 * y
6	$\langle factor\rangle\langle term'\rangle\langle expr'\rangle$	↑x – 2 * y
11	id$\langle term'\rangle\langle expr'\rangle$	↑x – 2 * y
–	id$\langle term'\rangle\langle expr'\rangle$	x ↑- 2 * y
9	idε $\langle expr'\rangle$	x ↑- 2
4	id– $\langle expr\rangle$	x ↑- 2 * y
–	id– $\langle expr\rangle$	x – ↑2 * y
2	id– $\langle term\rangle\langle expr'\rangle$	x – ↑2 * y
6	id– $\langle factor\rangle\langle term'\rangle\langle expr'\rangle$	x – ↑2 * y
10	id– num$\langle term'\rangle\langle expr'\rangle$	x – ↑2 * y
–	id– num$\langle term'\rangle\langle expr'\rangle$	x – 2 ↑* y
7	id– num* $\langle term\rangle\langle expr'\rangle$	x – 2 ↑* y
–	id– num* $\langle term\rangle\langle expr'\rangle$	x – 2 * ↑y
6	id– num* $\langle factor\rangle\langle term'\rangle\langle expr'\rangle$	x – 2 * ↑y
11	id– num* id$\langle term'\rangle\langle expr'\rangle$	x – 2 * ↑y
–	id– num* id$\langle term'\rangle\langle expr'\rangle$	x – 2 * y↑
9	id– num* id$\langle expr'\rangle$	x – 2 * y↑
5	id– num* id	x – 2 * y↑

The next symbol determined each choice correctly.

Back to left-recursion elimination

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

if $\exists\, A \to A\alpha$ then replace all of the A productions

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

with

$$A \to NA'$$
$$N \to \beta \mid \ldots \mid \gamma$$
$$A' \to \alpha A' \mid \varepsilon$$

where N and A' are new productions.

Repeat until there are no left-recursive productions.

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:

Given a context-free grammar that doesn't meet our conditions, it is undecidable whether an equivalent grammar exists that does meet our conditions.

Many *context-free languages* do not have such a grammar:

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

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

Recursive descent parsing

Now, we can produce a simple recursive descent parser from the (right-associative) grammar:

```
goal:
   token ← next_token();
   if (expr() = ERROR | token ≠ EOF) then
      return ERROR;
expr:
   if (term() = ERROR) then
      return ERROR;
   else return expr_prime();
expr_prime:
   if (token = PLUS) then
      token ← next_token();
      return expr();
   else if (token = MINUS) then
      token ← next_token();
      return expr();
   else return OK;
```


Recursive descent parsing

```
term:
   if (factor() = ERROR) then
      return ERROR;
   else return term_prime();
term_prime:
   if (token = MULT) then
      token ← next_token();
      return term();
   else if (token = DIV) then
      token ← next_token();
      return term();
   else return OK;
factor:
   if (token = NUM) then
      token ← next_token();
      return OK;
   else if (token = ID) then
      token ← next_token();
      return OK;
   else return ERROR;
```

Building the tree

One of the key jobs of the parser is to build an intermediate representation of the source code.

To build an abstract syntax tree, we can simply insert code at the appropriate points:

- `factor()` can stack nodes `id`, `num`
- `term_prime()` can stack nodes $*$, $/$
- `term()` can pop 3, build and push subtree
- `expr_prime()` can stack nodes $+$, $-$
- `expr()` can pop 3, build and push subtree
- `goal()` can pop and return tree

Non-recursive predictive parsing

Observation:

> *Our recursive descent parser encodes state information in its run-time stack, or call stack.*

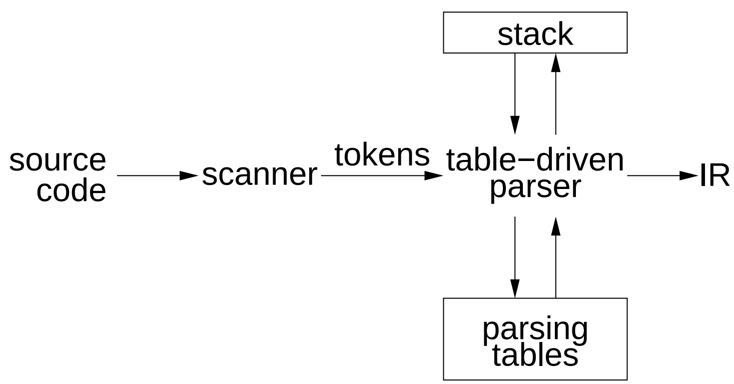
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

This suggests other implementation methods:

- explicit stack, hand-coded parser
- stack-based, table-driven parser

Non-recursive predictive parsing

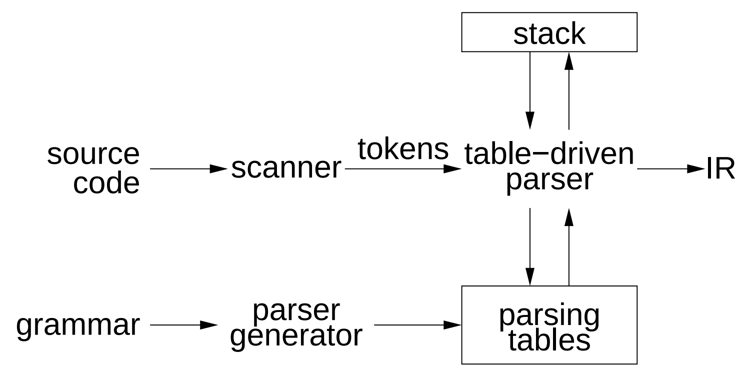
Now, a predictive parser looks like:



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

Table-driven parsers

A parser generator system often looks like:



This is true for both top-down (LL) and bottom-up (LR) parsers

Non-recursive predictive parsing

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

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         push Yk,Yk−1,···,Y1
      else error()
until X = EOF
```


Non-recursive predictive parsing

What we need now is a parsing table M.

Our expression grammar:

1	$\langle$goal$\rangle$	::=	$\langle$expr$\rangle$
2	$\langle$expr$\rangle$	::=	$\langle$term$\rangle\langle$expr$'\rangle$
3	$\langle$expr$'\rangle$	::=	$+\langle$expr$\rangle$
4		|	$-\langle$expr$\rangle$
5		|	ε
6	$\langle$term$\rangle$	::=	$\langle$factor$\rangle\langle$term$'\rangle$
7	$\langle$term$'\rangle$	::=	$*\langle$term$\rangle$
8		|	$/\langle$term$\rangle$
9		|	ε
10	$\langle$factor$\rangle$	::=	`num`
11		|	`id`

Its parse table:

	`id`	`num`	+	−	*	/	\$†
$\langle$goal$\rangle$	1	1	–	–	–	–	–
$\langle$expr$\rangle$	2	2	–	–	–	–	–
$\langle$expr$'\rangle$	–	–	3	4	–	–	5
$\langle$term$\rangle$	6	6	–	–	–	–	–
$\langle$term$'\rangle$	–	–	9	9	7	8	9
$\langle$factor$\rangle$	11	10	–	–	–	–	–

† we use \$ to represent `EOF`

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)

FIRST(α) contains the tokens valid in the initial position in 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 $X \to Y_1Y_2\cdots Y_k$:
   (a) Put FIRST$(Y_1) - \{\varepsilon\}$ in FIRST(X)
   (b) $\forall i : 1 < i \leq k$, 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)
   (c) 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$(\langle$goal$\rangle)$
2. If $A \to \alpha B \beta$:
   (a) Put FIRST$(\beta) - \{\varepsilon\}$ in FOLLOW(B)
   (b) If $\beta = \varepsilon$ (i.e., $A \to \alpha B$) 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 $A \to \beta$ and $A \to \gamma$ satisfy the condition $\text{FIRST}(\beta) \bigcap \text{FIRST}(\gamma) = \phi$.

What if $A \Rightarrow^* \varepsilon$?

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)
2. No ambiguous grammar is LL(1)
3. Some languages have no LL(1) grammar
4. A ε–free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example

- $S \to aS \mid a$ is not LL(1) because $\text{FIRST}(aS) = \text{FIRST}(a) = \{a\}$
- $S \to aS'$
  $S' \to aS' \mid \varepsilon$
  accepts the same language and is LL(1)

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

1. $\forall$ productions $A \to \alpha$:
   (a) $\forall a \in \text{FIRST}(\alpha)$, add $A \to \alpha$ to $M[A, a]$
   (b) If $\varepsilon \in \text{FIRST}(\alpha)$:
      i. $\forall b \in \text{FOLLOW}(A)$, add $A \to \alpha$ to $M[A, b]$
      ii. If $\$ \in \text{FOLLOW}(A)$ 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).

Note: recall $a, b \in V_t$, so $a, b \neq \varepsilon$

Example

Our long-suffering expression grammar:

$$\begin{array}{l|l|l} S \to E_1 & E' \to +E_3 \mid -E_4 \mid \varepsilon_5 & T' \to *T_7 \mid /T_8 \mid \varepsilon_9 \\ E \to TE'_2 & T \to FT'_6 & F \to \mathtt{num}_{10} \mid \mathtt{id}_{11} \end{array}$$

	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`	–							
$*$	$*$	–							
$/$	$/$	–							
$+$	$+$	–							
$-$	$-$	–							

Building the tree

Again, we insert code at the right points:

```
tos ← 0
Stack[tos] ← EOF
Stack[++tos] ← root node
Stack[++tos] ← Start Symbol
token ← next_token()
repeat
   X ← Stack[tos]
   if X is a terminal or EOF then
      if X = token then
         pop X
         token ← next_token()
         pop and fill in node
      else error()
   else /* X is a non-terminal */
      if M[X,token] = X → Y1Y2···Yk then
         pop X
         pop node for X
         build node for each child and
         make it a child of node for X
         push nk, Yk, nk−1, Yk−1, ···, n1, Y1
      else error()
until X = EOF
```


A grammar that is not LL(1)

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

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

Now, $\text{FIRST}(\langle\text{stmt}'\rangle) = \{\varepsilon, \texttt{else}\}$

Also, $\text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}, \$\}$

But, $\text{FIRST}(\langle\text{stmt}'\rangle) \bigcap \text{FOLLOW}(\langle\text{stmt}'\rangle) = \{\texttt{else}\} \neq \phi$

On seeing `else`, there is a conflict between choosing

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

$\Rightarrow$ grammar is not LL(1)!

The fix:

Put priority on $\langle\text{stmt}'\rangle ::= \texttt{else}\ \langle\text{stmt}\rangle$ to associate `else` with closest previous `then`.

Error recovery

Key notion:

- For each non-terminal, construct a set of terminals on which the parser can synchronize
- When an error occurs looking for A, scan until an element of $\text{SYNCH}(A)$ is found

Building SYNCH:

1. $a \in \text{FOLLOW}(A) \Rightarrow a \in \text{SYNCH}(A)$
2. place keywords that start statements in $\text{SYNCH}(A)$
3. add symbols in $\text{FIRST}(A)$ to $\text{SYNCH}(A)$

If we can't match a terminal on top of stack:

1. pop the terminal
2. print a message saying the terminal was inserted
3. continue the parse

(i.e., $\text{SYNCH}(a) = V_t - \{a\}$)

LL(k) parsing

Definition:

G is *strong* LL(k) iff. for $A \to \beta$ and $A \to \gamma$ ($\beta \neq \gamma$),
$\text{FIRST}_k(\beta\text{FOLLOW}_k(A)) \bigcap \text{FIRST}_k(\gamma\text{FOLLOW}_k(A)) = \phi$

LL(1) = Strong LL(1), but Strong LL$(k) \subset$ LL(k) for $k > 1$.

Definition:

G is LL(k) iff.:

1. $S \Rightarrow^*_{\text{lm}} wA\alpha \Rightarrow_{\text{lm}} w\beta\alpha \Rightarrow^* wx$
2. $S \Rightarrow^*_{\text{lm}} wA\alpha \Rightarrow_{\text{lm}} w\gamma\alpha \Rightarrow^* wy$
3. $\text{FIRST}_k(x) = \text{FIRST}_k(y)$

$\Rightarrow \beta = \gamma$

i.e., knowing w, A, and next k input tokens, $\text{FIRST}_k(x) = \text{FIRST}_k(y)$ accurately predicts production to use: $A \to \beta (= \gamma)$

- LL$(k) \subset$ LL$(k+1)$
- Strong LL$(k) \subset$ Strong LL$(k+1)$

Strong LL(k): example

Consider the grammar:

$$\begin{aligned} &G \to S\$ \\ &S \to aAa \\ &S \to bAba \\ &A \to b \\ &A \to \varepsilon \end{aligned}$$

It is:

- NOT LL(1)
- LL(2)
- NOT Strong LL(2)

Some definitions

Recall

For a grammar G, with start symbol S, any string α such that $S \Rightarrow^* \alpha$ is called a *sentential form*

- If $\alpha \in V_t^*$, then α is called a *sentence* in $L(G)$
- Otherwise it is just a sentential form (not a sentence in $L(G)$)

A *left-sentential form* is a sentential form that occurs in the leftmost derivation of some sentence.

A *right-sentential form* is a sentential form that occurs in the rightmost derivation of some sentence.

Bottom-up parsing

Goal:

Given an input string w and a grammar G, construct a parse tree by starting at the leaves and working to the root.

The parser repeatedly matches a *right-sentential* form from the language against the tree's upper frontier.

At each match, it applies a *reduction* to build on the frontier:

- each reduction matches an upper frontier of the partially built tree to the RHS of some production
- each reduction adds a node on top of the frontier

The final result is a rightmost derivation, in reverse.

Example

Consider the grammar

$$\begin{array}{r|lcl} 1 & S & \to & \mathtt{a}AB\mathtt{e} \\ 2 & A & \to & A\mathtt{bc} \\ 3 & & | & \mathtt{b} \\ 4 & B & \to & \mathtt{d} \end{array}$$

and the input string `abbcde`

Prod'n.	Sentential Form
3	$\mathtt{a}\boxed{\mathtt{b}}\mathtt{bcde}$
2	$\mathtt{a}\boxed{A\mathtt{bc}}\mathtt{de}$
4	$\mathtt{a}A\boxed{\mathtt{d}}\mathtt{e}$
1	$\boxed{\mathtt{a}AB\mathtt{e}}$
–	S

The trick appears to be scanning the input and finding valid sentential forms.

Handles

What are we trying to find?

A substring α of the tree's upper frontier that

matches some production $A \to \alpha$ where reducing α to A is one step in the reverse of a rightmost derivation

We call such a string a *handle*.

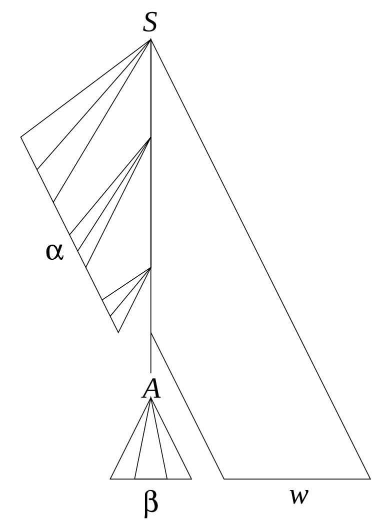
Formally:

a *handle* of a right-sentential form γ is a production $A \to \beta$ and a position in γ where β may be found and replaced by A to produce the previous right-sentential form in a rightmost derivation of γ.

i.e., if $S \Rightarrow^{*}_{\text{rm}} \alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$ then $A \to \beta$ in the position following α is a handle of $\alpha\beta w$

Because γ is a right-sentential form, the substring to the right of a handle contains only terminal symbols.

Handles



The handle $A \to \beta$ in the parse tree for $\alpha\beta w$

Handles

Theorem:

If G is unambiguous then every right-sentential form has a unique handle.

Proof: (by definition)

1. G is unambiguous $\Rightarrow$ rightmost derivation is unique
2. $\Rightarrow$ a unique production $A \to \beta$ applied to take γ_{i-1} to γ_i
3. $\Rightarrow$ a unique position k at which $A \to \beta$ is applied
4. $\Rightarrow$ a unique handle $A \to \beta$

Example

The left-recursive expression grammar (*original form*)

1	⟨goal⟩	::=	⟨expr⟩
2	⟨expr⟩	::=	⟨expr⟩ + ⟨term⟩
3		|	⟨expr⟩ − ⟨term⟩
4		|	⟨term⟩
5	⟨term⟩	::=	⟨term⟩ ∗ ⟨factor⟩
6		|	⟨term⟩ / ⟨factor⟩
7		|	⟨factor⟩
8	⟨factor⟩	::=	`num`
9		|	`id`

Prod'n.	Sentential Form
–	$\underline{\langle\text{goal}\rangle}$
1	$\underline{\langle\text{expr}\rangle}$
3	$\underline{\langle\text{expr}\rangle - \langle\text{term}\rangle}$
5	$\langle\text{expr}\rangle - \underline{\langle\text{term}\rangle * \langle\text{factor}\rangle}$
9	$\langle\text{expr}\rangle - \langle\text{term}\rangle * \underline{\texttt{id}}$
7	$\langle\text{expr}\rangle - \underline{\langle\text{factor}\rangle} * \texttt{id}$
8	$\langle\text{expr}\rangle - \underline{\texttt{num}} * \texttt{id}$
4	$\underline{\langle\text{term}\rangle} - \texttt{num} * \texttt{id}$
7	$\underline{\langle\text{factor}\rangle} - \texttt{num} * \texttt{id}$
9	$\underline{\texttt{id}} - \texttt{num} * \texttt{id}$

Handle-pruning

The process to construct a bottom-up parse is called *handle-pruning*.

To construct a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n = w$$

we set i to n and apply the following simple algorithm

for i = n downto 1

1. find the handle $A_i \rightarrow \beta_i$ in γ_i
2. replace β_i with A_i to generate γ_{i-1}

This takes $2n$ steps, where n is the length of the derivation

Stack implementation

One scheme to implement a handle-pruning, bottom-up parser is called a *shift-reduce* parser.

Shift-reduce parsers use a *stack* and an *input buffer*

1. initialize stack with $
2. Repeat until the top of the stack is the goal symbol and the input token is $
   a) *find the handle*
   if we don't have a handle on top of the stack, *shift* an input symbol onto the stack
   b) *prune the handle*
   if we have a handle $A \rightarrow \beta$ on the stack, *reduce*
      i) pop $|\beta|$ symbols off the stack
      ii) push A onto the stack

Example: back to x − 2 ∗ y

1	$S \rightarrow E$
2	$E \rightarrow E + T$
3	$\mid E - T$
4	$\mid T$
5	$T \rightarrow T * F$
6	$\mid T / F$
7	$\mid F$
8	$F \rightarrow$ num
9	$\mid$ id

Stack	Input	Action
$	id − num ∗ id	S
$<u>id</u>	− num ∗ id	R9
$<u>⟨factor⟩</u>	− num ∗ id	R7
$<u>⟨term⟩</u>	− num ∗ id	R4
$⟨expr⟩	− num ∗ id	S
$⟨expr⟩ −	num ∗ id	S
$⟨expr⟩ − <u>num</u>	∗ id	R8
$⟨expr⟩ − <u>⟨factor⟩</u>	∗ id	R7
$⟨expr⟩ − ⟨term⟩	∗ id	S
$⟨expr⟩ − ⟨term⟩ ∗	id	S
$⟨expr⟩ − ⟨term⟩ ∗ <u>id</u>		R9
$⟨expr⟩ − <u>⟨term⟩ ∗ ⟨factor⟩</u>		R5
$<u>⟨expr⟩ − ⟨term⟩</u>		R3
$<u>⟨expr⟩</u>		R1
$⟨goal⟩		A

Shift-reduce parsing

Shift-reduce parsers are simple to understand

A shift-reduce parser has just four canonical actions:

1. *shift* — next input symbol is shifted onto the top of the stack
2. *reduce* — right end of handle is on top of stack;
   locate left end of handle within the stack;
   pop handle off stack and push appropriate non-terminal LHS
3. *accept* — terminate parsing and signal success
4. *error* — call an error recovery routine

Key insight: recognize handles with a DFA:

- DFA transitions shift states instead of symbols
- accepting states trigger reductions

LR parsing

The skeleton parser:

```
push s_0
token ← next_token()
repeat forever
  s ← top of stack
  if action[s,token] = "shift s_i" then
    push s_i
    token ← next_token()
  else if action[s,token] = "reduce A → β"
    then
    pop |β| states
    s' ← top of stack
    push goto[s',A]
  else if action[s, token] = "accept" then
    return
  else error()
```

k shifts, l reduces, and 1 accept, where k is length of input string and l is length of reverse rightmost derivation

Example tables

state	ACTION				GOTO		
	id	+	*	$	E	T	F
0	s4	–	–	–	1	2	3
1	–	–	–	acc	–	–	–
2	–	s5	–	r3	–	–	–
3	–	r5	s6	r5	–	–	–
4	–	r6	r6	r6	–	–	–
5	s4	–	–	–	7	2	3
6	s4	–	–	–	–	8	3
7	–	–	–	r2	–	–	–
8	–	r4	–	r4	–	–	–

The Grammar

1	$S \to E$
2	$E \to T + E$
3	$\quad \mid T$
4	$T \to F * T$
5	$\quad \mid F$
6	$F \to \mathtt{id}$

Note: This is a simple little right-recursive grammar. It is *not* the same grammar as in previous lectures.

Example using the tables

Stack	Input	Action
$ 0	id∗ id+ id$	s4
$ 0 4	∗ id+ id$	r6
$ 0 3	∗ id+ id$	s6
$ 0 3 6	id+ id$	s4
$ 0 3 6 4	+ id$	r6
$ 0 3 6 3	+ id$	r5
$ 0 3 6 8	+ id$	r4
$ 0 2	+ id$	s5
$ 0 2 5	id$	s4
$ 0 2 5 4	$	r6
$ 0 2 5 3	$	r5
$ 0 2 5 2	$	r3
$ 0 2 5 7	$	r2
$ 0 1	$	acc

LR(k) grammars

Informally, we say that a grammar G is LR(k) if, given a rightmost derivation

$$S = \gamma_0 \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \cdots \Rightarrow \gamma_n = w,$$

we can, for each right-sentential form in the derivation:

1. *isolate the handle of each right-sentential form*, and
2. *determine the production by which to reduce*

by scanning γ_i from left to right, going at most k symbols beyond the right end of the handle of γ_i.

LR(k) grammars

Formally, a grammar G is LR(k) iff.:

1. $S \Rightarrow^*_{\text{rm}} \alpha A w \Rightarrow_{\text{rm}} \alpha\beta w$, and
2. $S \Rightarrow^*_{\text{rm}} \gamma B x \Rightarrow_{\text{rm}} \alpha\beta y$, and
3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

$\Rightarrow \alpha A y = \gamma B x$

i.e., Assume sentential forms $\alpha\beta w$ and $\alpha\beta y$, with common prefix $\alpha\beta$ and common k-symbol lookahead $\text{FIRST}_k(y) = \text{FIRST}_k(w)$, such that $\alpha\beta w$ reduces to $\alpha A w$ and $\alpha\beta y$ reduces to $\gamma B x$.

But, the common prefix means $\alpha\beta y$ also reduces to $\alpha A y$, for the same result.

Thus $\alpha A y = \gamma B x$.

Why study LR grammars?

LR(1) grammars are often used to construct parsers.

We call these parsers LR(1) parsers.

- virtually all context-free programming language constructs can be expressed in an LR(1) form
- LR grammars are the most general grammars parsable by a deterministic, bottom-up parser
- efficient parsers can be implemented for LR(1) grammars
- LR parsers detect an error as soon as possible in a left-to-right scan of the input
- LR grammars describe a proper superset of the languages recognized by predictive (i.e., LL) parsers

  LL(k)**:** recognize use of a production $A \to \beta$ seeing first k symbols derived from β

  LR(k)**:** recognize the handle β after seeing everything derived from β plus k lookahead symbols

LR parsing

Three common algorithms to build tables for an “LR” parser:

1. SLR(1)
   - smallest class of grammars
   - smallest tables (number of states)
   - simple, fast construction
2. LR(1)
   - full set of LR(1) grammars
   - largest tables (number of states)
   - slow, large construction
3. LALR(1)
   - intermediate sized set of grammars
   - same number of states as SLR(1)
   - canonical construction is slow and large
   - better construction techniques exist

SLR vs. LR/LALR

An LR(1) parser for either Algol or Pascal has several thousand states, while an SLR(1) or LALR(1) parser for the same language may have several hundred states.

LR(k) items

The table construction algorithms use sets of LR(k) *items* or *configurations* to represent the possible states in a parse.

An LR(k) item is a pair $[\alpha, \beta]$, where

- α is a production from G with a $\bullet$ at some position in the RHS, marking how much of the RHS of a production has already been seen
- β is a lookahead string containing k symbols (terminals or `$`)

Two cases of interest are $k = 0$ and $k = 1$:

LR(0) items play a key role in the SLR(1) table construction algorithm.

LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms.

Example

The $\bullet$ indicates how much of an item we have seen at a given state in the parse:

$[A \rightarrow \bullet XYZ]$ indicates that the parser is looking for a string that can be derived from XYZ

$[A \rightarrow XY \bullet Z]$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

LR(0) items: (*no lookahead*)

$A \rightarrow XYZ$ generates 4 LR(0) items:

1. $[A \rightarrow \bullet XYZ]$
2. $[A \rightarrow X \bullet YZ]$
3. $[A \rightarrow XY \bullet Z]$
4. $[A \rightarrow XYZ \bullet]$

The characteristic finite state machine (CFSM)

The CFSM for a grammar is a DFA which recognizes *viable prefixes* of right-sentential forms:

> A *viable prefix* is any prefix that does not extend beyond the handle.

It accepts when a handle has been discovered and needs to be reduced.

To construct the CFSM we need two functions:

- `closure0`(I) to build its states
- `goto0`(I, X) to determine its transitions

closure0

Given an item $[A \rightarrow \alpha \bullet B\beta]$, its closure contains the item and any other items that can generate legal substrings to follow α.

Thus, if the parser has viable prefix α on its stack, the input should reduce to $B\beta$ (or γ for some other item $[B \rightarrow \bullet\gamma]$ in the closure).

```
function closure0(I)
repeat
  if [A → α•Bβ] ∈ I
    add [B → •γ] to I
until no more items can be added to I
return I
```

goto0

Let I be a set of LR(0) items and X be a grammar symbol.

Then, GOTO(I,X) is the closure of the set of all items

$[A \to \alpha X \bullet \beta]$ such that $[A \to \alpha \bullet X\beta] \in I$

If I is the set of valid items for some viable prefix γ, then GOTO(I,X) is the set of valid items for the viable prefix γX.

GOTO(I,X) represents state after recognizing X in state I.

```
function goto0(I,X)
  let J be the set of items [A → αX•β]
    such that [A → α•Xβ] ∈ I
  return closure0(J)
```


Building the LR(0) item sets

We start the construction with the item $[S' \to \bullet S\$]$, where

S' is the start symbol of the augmented grammar G'
S is the start symbol of G
\$ represents EOF

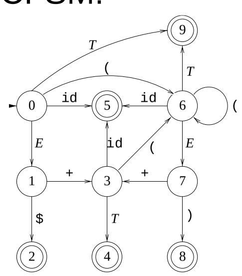
To compute the collection of sets of LR(0) items

```
function items(G')
  s0 ← closure0({[S' → •S$]})
  𝒮 ← {s0}
  repeat
    for each set of items s ∈ 𝒮
      for each grammar symbol X
        if goto0(s,X) ≠ φ and goto0(s,X) ∉ 𝒮
          add goto0(s,X) to 𝒮
  until no more item sets can be added to 𝒮
  return 𝒮
```


LR(0) example

1	S	$\to$	$E\$$
2	E	$\to$	$E+T$
3		$\mid$	T
4	T	$\to$	id
5		$\mid$	(E)

The corresponding CFSM:



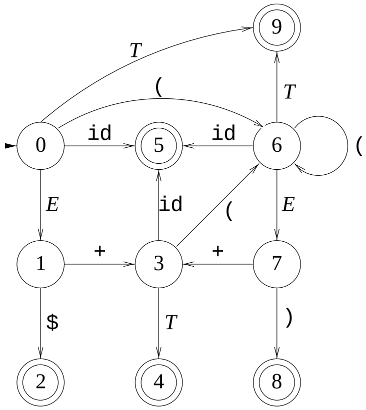
$I_0 : S \to \bullet E\$$
$E \to \bullet E+T$
$E \to \bullet T$
$T \to \bullet \texttt{id}$
$T \to \bullet(E)$
$I_1 : S \to E \bullet \$$
$E \to E \bullet +T$
$I_2 : S \to E\$\bullet$
$I_3 : E \to E + \bullet T$
$T \to \bullet \texttt{id}$
$T \to \bullet(E)$

$I_4 : E \to E+T\bullet$
$I_5 : T \to \texttt{id}\bullet$
$I_6 : T \to (\bullet E)$
$E \to \bullet E+T$
$E \to \bullet T$
$T \to \bullet \texttt{id}$
$T \to \bullet(E)$
$I_7 : T \to (E\bullet)$
$E \to E \bullet +T$
$I_8 : T \to (E)\bullet$
$I_9 : E \to T\bullet$

Constructing the LR(0) parsing table

1. construct the collection of sets of LR(0) items for G'
2. state i of the CFSM is constructed from I_i
   (a) $[A \to \alpha \bullet a\beta] \in I_i$ and $\texttt{goto0}(I_i,a) = I_j$
   $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*shift j*"
   (b) $[A \to \alpha\bullet] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*reduce* $A \to \alpha$", $\forall a$
   (c) $[S' \to S\$\bullet] \in I_i$
   $\Rightarrow$ ACTION$[i,a] \leftarrow$ "*accept*", $\forall a$
3. $\texttt{goto0}(I_i,A) = I_j$
   $\Rightarrow$ GOTO$[i,A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{closure0}([S' \to \bullet S\$])$

LR(0) example



state	ACTION					GOTO		
	id	(	)	+	$	S	E	T
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	s2	–	–	–
2	acc	acc	acc	acc	acc	–	–	–
3	s5	s6	–	–	–	–	–	4
4	r2	r2	r2	r2	r2	–	–	–
5	r4	r4	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	r5	r5	r5	r5	r5	–	–	–
9	r3	r3	r3	r3	r3	–	–	–

Conflicts in the ACTION table

If the LR(0) parsing table contains any multiply-defined ACTION entries then G is not LR(0)

Two conflicts arise:

shift-reduce: both shift and reduce possible in same item set

reduce-reduce: more than one distinct reduce action possible in same item set

Conflicts can be resolved through *lookahead* in ACTION.

Consider:

- $A \to \varepsilon \mid a\alpha$
  $\Rightarrow$ shift-reduce conflict
- `a:=b+c*d`
  requires lookahead to avoid shift-reduce conflict after shifting `c`
  (need to see `*` to give precedence over `+`)

SLR(1): simple lookahead LR

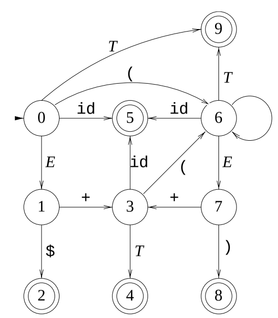
Add lookaheads after building LR(0) item sets

Constructing the SLR(1) parsing table:

1. construct the collection of sets of LR(0) items for G'
2. state i of the CFSM is constructed from I_i
   (a) $[A \to \alpha \bullet a\beta] \in I_i$ and $\mathtt{goto0}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift* j", $\underline{\forall a \neq \$}$
   (b) $[A \to \alpha\bullet] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*reduce* $A \to \alpha$", $\underline{\forall a \in \text{FOLLOW}(A)}$
   (c) $\underline{[S' \to S \bullet \$]} \in I_i$
   $\underline{\Rightarrow \text{ACTION}[i, \$]} \leftarrow$ "*accept*"
3. $\mathtt{goto0}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\mathtt{closure0}([S' \to \bullet S\$])$

From previous example

1	S	$\to$	$E\$$
2	E	$\to$	$E + T$
3		$\mid$	T
4	T	$\to$	id
5		$\mid$	(E)



FOLLOW(E) = FOLLOW$(T) = \{\$, +,)\}$

state	ACTION					GOTO		
	id	(	)	+	$	S	E	T
0	s5	s6	–	–	–	–	1	9
1	–	–	–	s3	acc	–	–	–
2	–	–	–	–	–	–	–	–
3	s5	s6	–	–	–	–	–	4
4	–	–	r2	r2	r2	–	–	–
5	–	–	r4	r4	r4	–	–	–
6	s5	s6	–	–	–	–	7	9
7	–	–	s8	s3	–	–	–	–
8	–	–	r5	r5	r5	–	–	–
9	–	–	r3	r3	r3	–	–	–

Example: A grammar that is not LR(0)

$$
\begin{array}{l|lcl}
1 & S & \rightarrow & E\$ \\
2 & E & \rightarrow & E+T \\
3 & & | & T \\
4 & T & \rightarrow & T*F \\
5 & & | & F \\
6 & F & \rightarrow & \mathtt{id} \\
7 & & | & (E)
\end{array}
$$

	FOLLOW
E	$\{+,),\$\}$
T	$\{+,*,),\$\}$
F	$\{+,*,),\$\}$

$$
\begin{array}{ll}
I_0: & S \rightarrow \bullet E\$ \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \mathtt{id} \\
& F \rightarrow \bullet (E) \\
I_1: & S \rightarrow E \bullet \$ \\
& E \rightarrow E \bullet +T \\
I_2: & S \rightarrow E\$\bullet \\
I_3: & E \rightarrow E+\bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \mathtt{id} \\
& F \rightarrow \bullet (E) \\
I_4: & T \rightarrow F\bullet \\
I_5: & F \rightarrow \mathtt{id}\bullet
\end{array}
\qquad
\begin{array}{ll}
I_6: & F \rightarrow (\bullet E) \\
& E \rightarrow \bullet E+T \\
& E \rightarrow \bullet T \\
& T \rightarrow \bullet T*F \\
& T \rightarrow \bullet F \\
& F \rightarrow \bullet \mathtt{id} \\
& F \rightarrow \bullet (E) \\
I_7: & E \rightarrow T\bullet \\
& T \rightarrow T \bullet *F \\
I_8: & T \rightarrow T*\bullet F \\
& F \rightarrow \bullet \mathtt{id} \\
& F \rightarrow \bullet (E) \\
I_9: & T \rightarrow T*F\bullet \\
I_{10}: & F \rightarrow (E)\bullet \\
I_{11}: & E \rightarrow E+T\bullet \\
& T \rightarrow T \bullet *F \\
I_{12}: & F \rightarrow (E\bullet) \\
& E \rightarrow E \bullet +T
\end{array}
$$

Example: But it is SLR(1)

state	ACTION						GOTO			
	$+$	$*$	id	(	)	\$	S	E	T	F
0	–	–	s5	s6	–	–	–	1	7	4
1	s3	–	–	–	–	acc	–	–	–	–
2	–	–	–	–	–	–	–	–	–	–
3	–	–	s5	s6	–	–	–	–	11	4
4	r5	r5	–	–	r5	r5	–	–	–	–
5	r6	r6	–	–	r6	r6	–	–	–	–
6	–	–	s5	s6	–	–	–	12	7	4
7	r3	s8	–	–	r3	r3	–	–	–	–
8	–	–	s5	s6	–	–	–	–	–	9
9	r4	r4	–	–	r4	r4	–	–	–	–
10	r7	r7	–	–	r7	r7	–	–	–	–
11	r2	s8	–	–	r2	r2	–	–	–	–
12	s3	–	–	–	s10	–	–	–	–	–

Example: A grammar that is not SLR(1)

Consider:

$$
\begin{array}{lcl}
S & \rightarrow & L=R \\
& | & R \\
L & \rightarrow & *R \\
& | & \mathtt{id} \\
R & \rightarrow & L
\end{array}
$$

Its LR(0) item sets:

$$
\begin{array}{ll}
I_0: & S' \rightarrow \bullet S\$ \\
& S \rightarrow \bullet L=R \\
& S \rightarrow \bullet R \\
& L \rightarrow \bullet *R \\
& L \rightarrow \bullet \mathtt{id} \\
& R \rightarrow \bullet L \\
I_1: & S' \rightarrow S \bullet \$ \\
I_2: & S \rightarrow L\bullet = R \\
& R \rightarrow L\bullet \\
I_3: & S \rightarrow R\bullet \\
I_4: & L \rightarrow \mathtt{id}\bullet
\end{array}
\qquad
\begin{array}{ll}
I_5: & L \rightarrow * \bullet R \\
& R \rightarrow \bullet L \\
& L \rightarrow \bullet *R \\
& L \rightarrow \bullet \mathtt{id} \\
I_6: & S \rightarrow L = \bullet R \\
& R \rightarrow \bullet L \\
& L \rightarrow \bullet *R \\
& L \rightarrow \bullet \mathtt{id} \\
I_7: & L \rightarrow *R\bullet \\
I_8: & R \rightarrow L\bullet \\
I_9: & S \rightarrow L=R\bullet
\end{array}
$$

Now consider I_2: $=\ \in$ FOLLOW(R) $(S \Rightarrow L=R \Rightarrow *R=R)$

LR(1) items

Recall: An LR(k) item is a pair $[\alpha,\beta]$, where

- α is a production from G with a $\bullet$ at some position in the RHS, marking how much of the RHS of a production has been seen
- β is a lookahead string containing k symbols (terminals or \$)

What about LR(1) items?

- All the lookahead strings are constrained to have length 1
- Look something like $[A \rightarrow X \bullet YZ, a]$

LR(1) items

What's the point of the lookahead symbols?

- carry along to choose correct reduction when there is a choice
- lookaheads are bookkeeping, unless item has $\bullet$ at right end:
  - in $[A \to X \bullet YZ, a]$, a has no direct use
  - in $[A \to XYZ \bullet, a]$, a is useful
- allows use of grammars that are not *uniquely invertible*†

The point: For $[A \to \alpha \bullet, a]$ and $[B \to \alpha \bullet, b]$, we can decide between reducing to A or B by looking at limited right context

†No two productions have the same RHS

closure1(I)

Given an item $[A \to \alpha \bullet B\beta, a]$, its closure contains the item and any other items that can generate legal substrings to follow α.

Thus, if the parser has viable prefix α on its stack, the input should reduce to $B\beta$ (or γ for some other item $[B \to \bullet\gamma, b]$ in the closure).

```
function closure1(I)
repeat
  if [A → α • Bβ, a] ∈ I
    add [B → •γ, b] to I, where b ∈ first(βa)
until no more items can be added to I
return I
```


goto1(I)

Let I be a set of LR(1) items and X be a grammar symbol.

Then, GOTO(I, X) is the closure of the set of all items

$$[A \to \alpha X \bullet \beta, a] \text{ such that } [A \to \alpha \bullet X\beta, a] \in I$$

If I is the set of valid items for some viable prefix γ, then GOTO(I, X) is the set of valid items for the viable prefix γX.

goto(I, X) represents state after recognizing X in state I.

```
function goto1(I,X)
  let J be the set of items [A → αX • β, a]
    such that [A → α • Xβ, a] ∈ I
  return closure1(J)
```


Building the LR(1) item sets for grammar G

We start the construction with the item $[S' \to \bullet S, \$]$, where

S' is the start symbol of the augmented grammar G'
S is the start symbol of G
\$ represents EOF

To compute the collection of sets of LR(1) items

```
function items(G')
  s0 ← closure1({[S' → •S, $]})
  𝒮 ← {s0}
  repeat
    for each set of items s ∈ 𝒮
      for each grammar symbol X
        if goto1(s,X) ≠ φ and goto1(s,X) ∉ 𝒮
          add goto1(s,X) to 𝒮
  until no more item sets can be added to 𝒮
  return 𝒮
```

Constructing the LR(1) parsing table

Build lookahead into the DFA to begin with

1. construct the collection of sets of LR(1) items for G'
2. state i of the LR(1) machine is constructed from I_i
   (a) $[A \to \alpha \bullet a\beta, b] \in I_i$ and $\texttt{goto1}(I_i, a) = I_j$
   $\Rightarrow$ ACTION$[i, a] \leftarrow$ "*shift j*"
   (b) $[A \to \alpha\bullet, \underline{a}] \in I_i, A \neq S'$
   $\Rightarrow$ ACTION$[i, \underline{a}] \leftarrow$ "*reduce* $A \to \alpha$"
   (c) $[S' \to S\bullet, \$] \in I_i$
   $\Rightarrow$ ACTION$[i, \$] \leftarrow$ "*accept*"
3. $\texttt{goto1}(I_i, A) = I_j$
   $\Rightarrow$ GOTO$[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "*error*"
5. initial state of parser s_0 is $\texttt{closure1}([S' \to \bullet S, \$])$

Back to previous example ($\notin$ SLR(1))

$$\begin{array}{lcl} S & \to & L = R \\ & | & R \\ L & \to & *R \\ & | & \texttt{id} \\ R & \to & L \end{array}$$

$$\begin{array}{lll}
I_0: & S' \to \bullet S, & \$ \\
& S \to \bullet L = R, & \$ \\
& S \to \bullet R, & \$ \\
& L \to \bullet * R, & = \\
& L \to \bullet \texttt{id}, & = \\
& R \to \bullet L, & \$ \\
& L \to \bullet * R, & \$ \\
& L \to \bullet \texttt{id}, & \$ \\
I_1: & S' \to S\bullet, & \$ \\
I_2: & S \to L\bullet = R, & \$ \\
& R \to L\bullet, & \$ \\
I_3: & S \to R\bullet, & \$ \\
I_4: & L \to * \bullet R, & = \$ \\
& R \to \bullet L, & = \$ \\
& L \to \bullet * R, & = \$ \\
& L \to \bullet \texttt{id}, & = \$ \\
I_5: & L \to \texttt{id}\bullet, & = \$ \\
I_6: & S \to L = \bullet R, & \$ \\
& R \to \bullet L, & \$ \\
& L \to \bullet * R, & \$ \\
& L \to \bullet \texttt{id}, & \$ \\
I_7: & L \to * R\bullet, & = \$ \\
I_8: & R \to L\bullet, & = \$ \\
I_9: & S \to L = R\bullet, & \$ \\
I_{10}: & R \to L\bullet, & \$ \\
I_{11}: & L \to * \bullet R, & \$ \\
& R \to \bullet L, & \$ \\
& L \to \bullet * R, & \$ \\
& L \to \bullet \texttt{id}, & \$ \\
I_{12}: & L \to \texttt{id}\bullet, & \$ \\
I_{13}: & L \to * R\bullet, & \$
\end{array}$$

I_2 no longer has shift-reduce conflict: reduce on \$, shift on $=$

Example: back to SLR(1) expression grammar

In general, LR(1) has many more states than LR(0)/SLR(1):

$$\begin{array}{r|lcl} 1 & S & \to & E \\ 2 & E & \to & E + T \\ 3 & & | & T \end{array} \qquad \begin{array}{r|lcl} 4 & T & \to & T * F \\ 5 & & | & F \\ 6 & F & \to & \texttt{id} \\ 7 & & | & (E) \end{array}$$

LR(1) item sets:

$$\begin{array}{lll}
I_0: & & \\
& S \to \bullet E, & \$ \\
& E \to \bullet E + T, & +\$ \\
& E \to \bullet T, & +\$ \\
& T \to \bullet T * F, & *+\$ \\
& T \to \bullet F, & *+\$ \\
& F \to \bullet \texttt{id}, & *+\$ \\
& F \to \bullet (E), & *+\$
\end{array}
\quad
\begin{array}{lll}
I'_0: & \text{shifting (} & \\
& F \to (\bullet E), & *+\$ \\
& E \to \bullet E + T, & +) \\
& E \to \bullet T, & +) \\
& T \to \bullet T * F, & *+) \\
& T \to \bullet F, & *+) \\
& F \to \bullet \texttt{id}, & *+) \\
& F \to \bullet (E), & *+)
\end{array}
\quad
\begin{array}{lll}
I''_0: & \text{shifting (} & \\
& F \to (\bullet E), & *+) \\
& E \to \bullet E + T, & +) \\
& E \to \bullet T, & +) \\
& T \to \bullet T * F, & *+) \\
& T \to \bullet F, & *+) \\
& F \to \bullet \texttt{id}, & *+) \\
& F \to \bullet (E), & *+)
\end{array}$$

Another example

Consider:

$$\begin{array}{r|lcl} 0 & S' & \to & S \\ 1 & S & \to & CC \\ 2 & C & \to & cC \\ 3 & & | & d \end{array}$$

state	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4	–	1	2
1	–	–	acc	–	–
2	s6	s7	–	–	5
3	s3	s4	–	–	8
4	r3	r3	–	–	–
5	–	–	r1	–	–
6	s6	s7	–	–	9
7	–	–	r3	–	–
8	r2	r2	–	–	–
9	–	–	r2	–	–

LR(1) item sets:

$$\begin{array}{lll}
I_0: & S' \to \bullet S, & \$ \\
& S \to \bullet CC, & \$ \\
& C \to \bullet cC, & cd \\
& C \to \bullet d, & cd \\
I_1: & S' \to S\bullet, & \$ \\
I_2: & S \to C \bullet C, & \$ \\
& C \to \bullet cC, & \$ \\
& C \to \bullet d, & \$ \\
I_3: & C \to c \bullet C, & cd \\
& C \to \bullet cC, & cd \\
& C \to \bullet d, & cd
\end{array}
\qquad
\begin{array}{lll}
I_4: & C \to d\bullet, & cd \\
I_5: & S \to CC\bullet, & \$ \\
I_6: & C \to c \bullet C, & \$ \\
& C \to \bullet cC, & \$ \\
& C \to \bullet d, & \$ \\
I_7: & C \to d\bullet, & \$ \\
I_8: & C \to cC\bullet, & cd \\
I_9: & C \to cC\bullet, & \$
\end{array}$$

LALR(1) parsing

Define the *core* of a set of LR(1) items to be the set of LR(0) items derived by ignoring the lookahead symbols.

Thus, the two sets

- $\{[A \to \alpha \bullet \beta, \mathtt{a}], [A \to \alpha \bullet \beta, \mathtt{b}]\}$, and
- $\{[A \to \alpha \bullet \beta, \mathtt{c}], [A \to \alpha \bullet \beta, \mathtt{d}]\}$

have the same core.

Key idea:

If two sets of LR(1) items, I_i and I_j, have the same core, we can merge the states that represent them in the ACTION and GOTO tables.

LALR(1) table construction

To construct LALR(1) parsing tables, we can insert a single step into the LR(1) algorithm

(1.5) For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union.

The goto function must be updated to reflect the replacement sets.

The resulting algorithm has large space requirements.

LALR(1) table construction

The revised (*and renumbered*) algorithm

1. construct the collection of sets of LR(1) items for G'
2. for each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union (update the `goto` function incrementally)
3. state i of the LALR(1) machine is constructed from I_i.
   (a) $[A \to \alpha \bullet a\beta, b] \in I_i$ and $\mathtt{goto1}(I_i, a) = I_j$
   $\Rightarrow \text{ACTION}[i,a] \leftarrow$ "*shift j*"
   (b) $[A \to \alpha\bullet, a] \in I_i, A \neq S'$
   $\Rightarrow \text{ACTION}[i,a] \leftarrow$ "*reduce* $A \to \alpha$"
   (c) $[S' \to S\bullet, \$] \in I_i \Rightarrow \text{ACTION}[i,\$] \leftarrow$ "*accept*"
4. $\mathtt{goto1}(I_i, A) = I_j \Rightarrow \text{GOTO}[i,A] \leftarrow j$
5. set undefined entries in ACTION and GOTO to "*error*"
6. initial state of parser s_0 is $\mathtt{closure1}([S' \to \bullet S, \$])$

Example

Reconsider:

$$\begin{array}{c|lcl} 0 & S' & \to & S \\ 1 & S & \to & CC \\ 2 & C & \to & cC \\ 3 & & \mid & d \end{array}$$

$$\begin{array}{ll} I_0: S' \to \bullet S, & \$ \\ \quad S \to \bullet CC, & \$ \\ \quad C \to \bullet cC, & cd \\ \quad C \to \bullet d, & cd \\ I_1: S' \to S\bullet, & \$ \\ I_2: S \to C \bullet C, & \$ \\ \quad C \to \bullet cC, & \$ \\ \quad C \to \bullet d, & \$ \end{array} \qquad \begin{array}{ll} I_3: C \to c \bullet C, & cd \\ \quad C \to \bullet cC, & cd \\ \quad C \to \bullet d, & cd \\ I_4: C \to d\bullet, & cd \\ I_5: S \to CC\bullet, & \$ \end{array} \qquad \begin{array}{ll} I_6: C \to c \bullet C, & \$ \\ \quad C \to \bullet cC, & \$ \\ \quad C \to \bullet d, & \$ \\ I_7: C \to d\bullet, & \$ \\ I_8: C \to cC\bullet, & cd \\ I_9: C \to cC\bullet, & \$ \end{array}$$

Merged states:

$$\begin{array}{ll} I_{36}: C \to c \bullet C, & cd\$ \\ \quad C \to \bullet cC, & cd\$ \\ \quad C \to \bullet d, & cd\$ \\ I_{47}: C \to d\bullet, & cd\$ \\ I_{89}: C \to cC\bullet, & cd\$ \end{array}$$

state	ACTION			GOTO	
	c	d	\$	S	C
0	s36	s47	–	1	2
1	–	–	acc	–	–
2	s36	s47	–	–	5
36	s36	s47	–	–	8
47	r3	r3	r3	–	–
5	–	–	r1	–	–
89	r2	r2	r2	–	–

More efficient LALR(1) construction

Observe that we can:

- represent I_i by its *basis* or *kernel*:
  items that are either $[S' \to \bullet S, \$]$
  or do not have $\bullet$ at the left of the RHS

- compute *shift*, *reduce* and *goto* actions for state derived from I_i directly from its kernel

This leads to a method that avoids building the complete canonical collection of sets of LR(1) items

The role of precedence

Precedence and associativity can be used to resolve shift/reduce conflicts in ambiguous grammars.

- lookahead with higher precedence $\Rightarrow$ *shift*
- same precedence, left associative $\Rightarrow$ *reduce*

Advantages:

- more concise, albeit ambiguous, grammars
- shallower parse trees $\Rightarrow$ fewer reductions

Classic application: expression grammars

The role of precedence

With precedence and associativity, we can use:

$$
\begin{array}{lcl}
E & \to & E * E \\
 & | & E / E \\
 & | & E + E \\
 & | & E - E \\
 & | & (E) \\
 & | & \texttt{-}E \\
 & | & \texttt{id} \\
 & | & \texttt{num}
\end{array}
$$

This eliminates useless reductions (*single productions*)

Error recovery in shift-reduce parsers

The problem

- encounter an invalid token
- bad pieces of tree hanging from stack
- incorrect entries in symbol table

We want to *parse* the rest of the file

Restarting the parser

- find a restartable state on the stack
- move to a consistent place in the input
- print an informative message to `stderr` (*line number*)

Error recovery in yacc/bison/Java CUP

The error mechanism

- designated token `error`
- valid in any production
- `error` shows syncronization points

When an error is discovered

- pops the stack until `error` is legal
- skips input tokens until it successfully shifts 3
- `error` productions can have actions

This mechanism is fairly general

See §Error Recovery of the on-line CUP manual

Example

Using `error`

```
stmt_list  :  stmt
           |  stmt_list ; stmt
```

can be augmented with `error`

```
stmt_list  :  stmt
           |  error
           |  stmt_list ; stmt
```

This should

- throw out the erroneous statement
- synchronize at "`;`" or "`end`"
- invoke `yyerror("syntax error")`

Other "natural" places for errors

- all the "lists": `FieldList`, `CaseList`
- missing parentheses or brackets (`yychar`)
- extra operator or missing operator

Left versus right recursion

Right Recursion:

- needed for termination in predictive parsers
- requires more stack space
- right associative operators

Left Recursion:

- works fine in bottom-up parsers
- limits required stack space
- left associative operators

Rule of thumb:

- right recursion for top-down parsers
- left recursion for bottom-up parsers

Parsing review

Recursive descent

A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

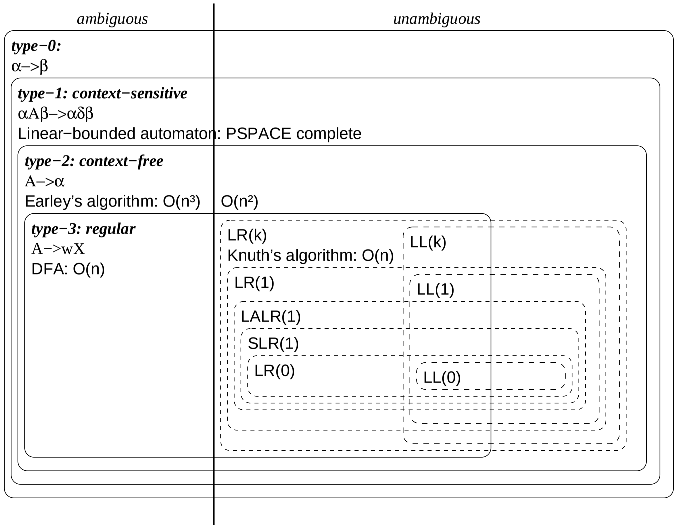
LL(k)

An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

LR(k)

An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Complexity of parsing: grammar hierarchy



Note: this is a hierarchy of grammars *not* languages

Language vs. grammar

For example, every regular *language* has a grammar that is LL(1), but not all regular grammars are LL(1). Consider:

$$S \rightarrow ab$$
$$S \rightarrow ac$$

Without left-factoring, this grammar is not LL(1).