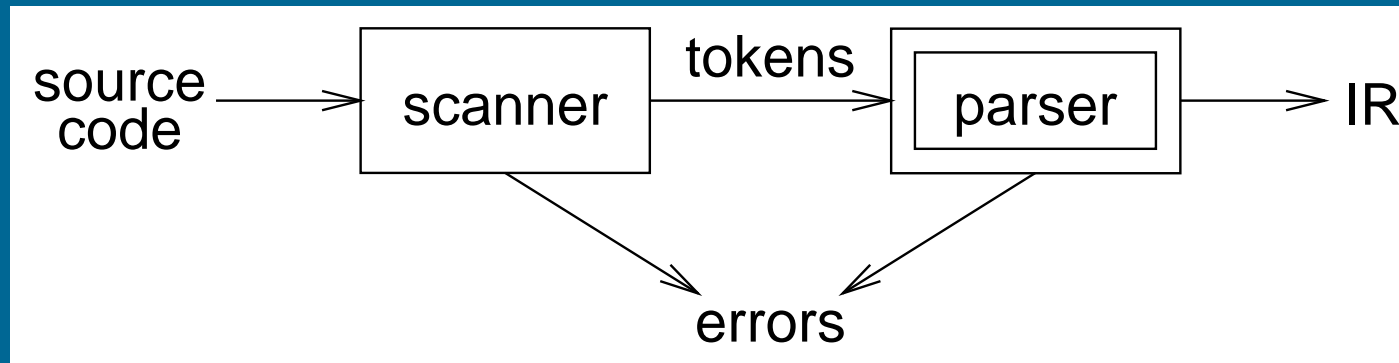


The role of the parser



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

Copyright ©2001 by Antony L. Hosking. *Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or fee. Request permission to publish from hosking@cs.purdue.edu.*

Syntax analysis

Context-free syntax is specified with a *context-free grammar*.

Formally, a CFG G is a 4-tuple (V_n, V_t, P, S) , where:

V_n , the *nonterminals*, is a set of syntactic variables that denote sets of (sub)strings occurring in the language.

These are used to impose a structure on the grammar.

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.

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 left hand side.

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*.

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

Notation and terminology

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

If $A \rightarrow \gamma$ then $\alpha A \beta \Rightarrow \alpha \gamma \beta$ is a *single-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, $L(G) = \{\beta \in V^* \mid S \Rightarrow^* \beta\} \cap V_t^*$

Syntax analysis

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

Example:

```
1 | <goal> ::= <expr>
2 | <expr> ::= <expr><op><expr>
3 |           | num
4 |           | id
5 | <op> ::= +
6 |           | -
7 |           | *
8 |           | /
```

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 underline
3. productions as in the example

Scanning vs. parsing

Where do we draw the line?

$$\begin{aligned} \textit{term} & ::= [\text{a} - \text{zA} - \text{z}]([\text{a} - \text{zA} - \text{z}] \mid [0 - 9])^* \\ & \quad \mid 0 \mid [1 - 9][0 - 9]^* \\ \textit{op} & ::= + \mid - \mid * \mid / \\ \textit{expr} & ::= (\textit{term} \textit{op})^* \textit{term} \end{aligned}$$

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

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:

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

We have derived the sentence $x + 2 * y$.

We denote this $\langle \text{goal} \rangle \Rightarrow^* \text{id} + \text{num} * \text{id}$.

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.

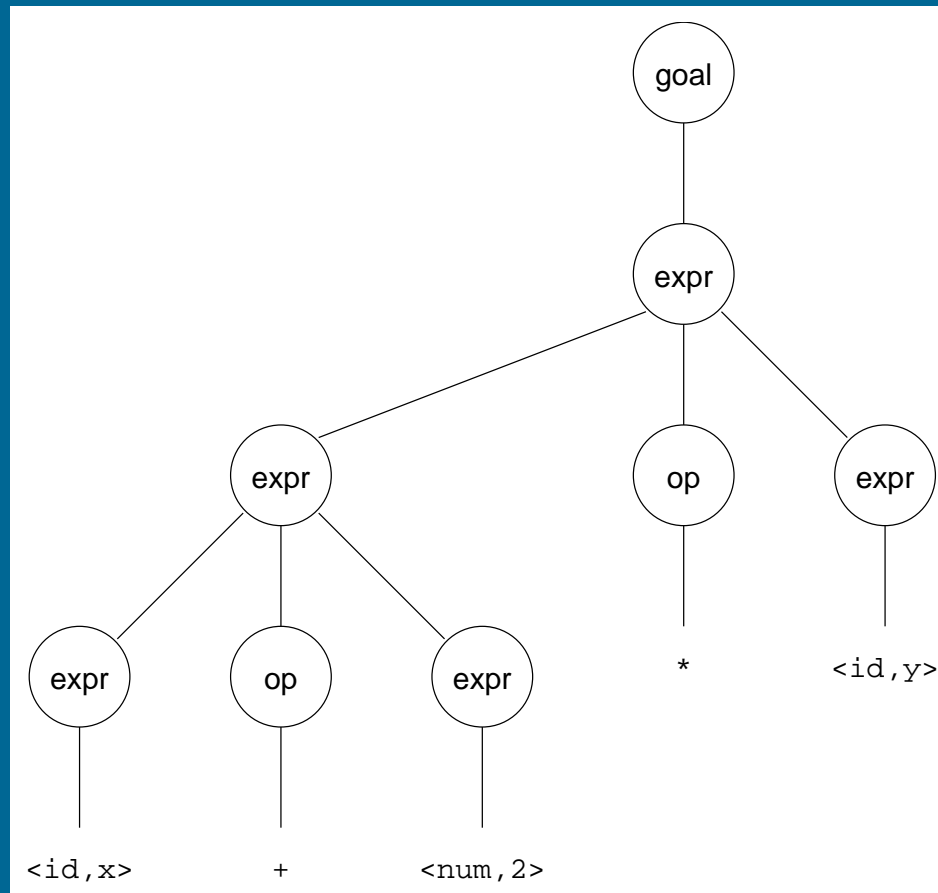
Rightmost derivation

For the string $x + 2 * y$:

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

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

Precedence



*Treewalk evaluation computes $(x + 2) * y$*
— the “wrong” answer!

Should be $x + (2 * y)$

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:

| | | | | |
|---|--|---------------------------------|-----|---|
| 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 | | $\langle \text{factor} \rangle$ | ::= | num |
| 9 | | | | id |

This grammar enforces a precedence on the derivation:

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

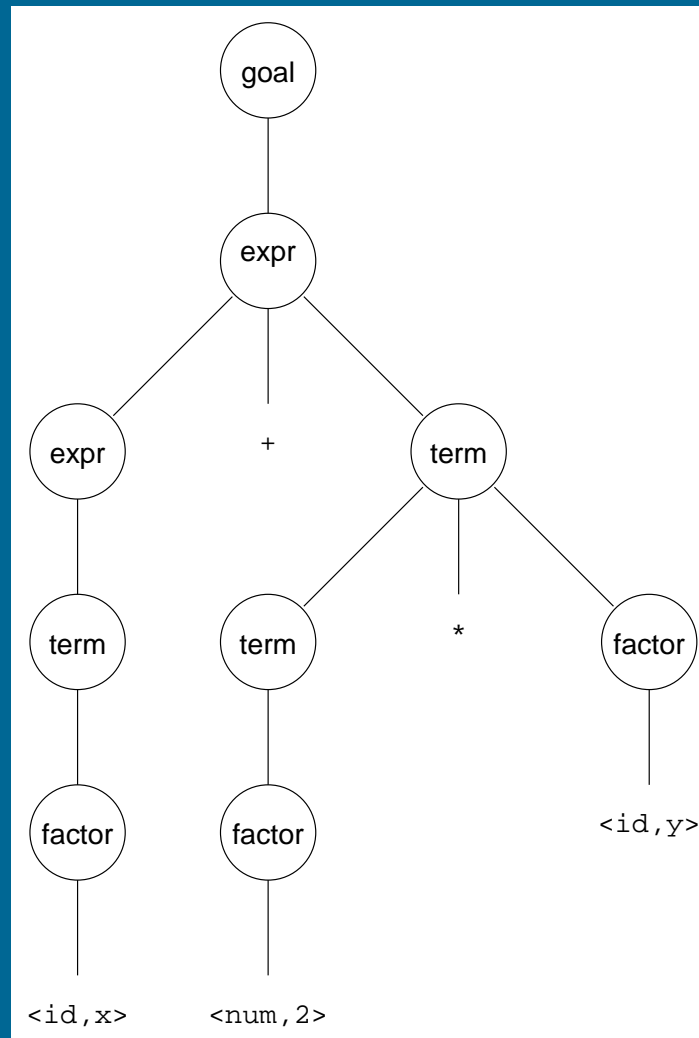
Precedence

Now, for the string $x + 2 * y$:

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

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

Precedence



*Treewalk evaluation computes $x + (2 * y)$*

Ambiguity

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

Example:

```
<stmt> ::= if <expr> then <stmt>
        | if <expr> then <stmt> else <stmt>
        | other stmts
```

Consider deriving the sentential form:

if E_1 then if E_2 then S_1 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:

```
⟨stmt⟩      ::=  ⟨matched⟩
              |  ⟨unmatched⟩
⟨matched⟩   ::=  if ⟨expr⟩ then ⟨matched⟩ else ⟨matched⟩
              |  other stmts
⟨unmatched⟩ ::=  if ⟨expr⟩ then ⟨stmt⟩
              |  if ⟨expr⟩ then ⟨matched⟩ else ⟨unmatched⟩
```

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*.

Example:

$$a = f(17)$$

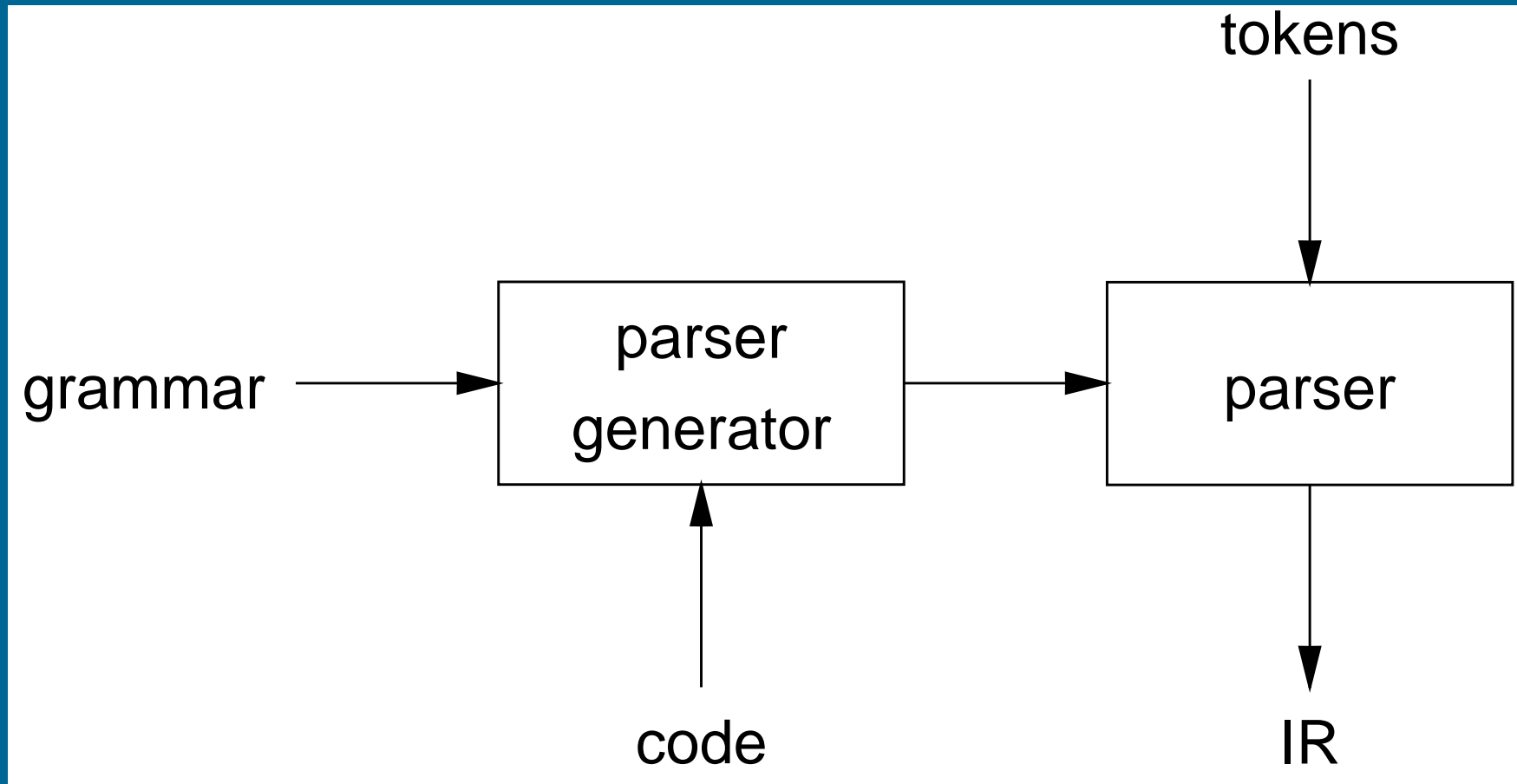
In many Algol-like languages, f could be a function or subscripted variable.

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 the next node to be expanded (must have a label in V_n)

The key is selecting the right production in step 1

⇒ 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 | | | | $\langle \text{expr} \rangle - \langle \text{term} \rangle$ |
| 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 | | $\langle \text{factor} \rangle$ | $::=$ | num |
| 9 | | | | id |

Consider the input string $x - 2 * y$

Example

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

Example

Another possible parse for $x - 2 * y$

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

If the parser makes the wrong choices, expansion doesn't terminate.
This isn't a good property for a parser to have.

(Parsers should terminate!)

Top-down parsing with pushdown automaton

A top-down parser for grammar $G = (V_n, V_t, P, S)$ is a pushdown automaton $A = (Q, V_t, V_k, \delta, q_0, k_0)$ that accepts input with empty pushdown where

- $Q = \{q_0\}$ is the set of states
- $V_k = V_n \cup V_t$ is the alphabet of pushdown symbols
- $\delta : Q \times V_t \cup \{\epsilon\} \times V_k \rightarrow Q \times V_k^*$
- q_0 is the initial state
- $k_0 = S$ is the initial pushdown symbol

where

- $\delta(q_0, \epsilon, A) = (q_0, \alpha)$ for each production $A \rightarrow \alpha \in P$
- $\delta(q_0, x, x) = (q_0, \epsilon)$

Pushdown automaton example

| Pushdown (rev) | Input | Prod'n |
|---|-------|----------|
| $\langle \text{goal} \rangle$ | x-2*y | 1 |
| $\langle \text{expr} \rangle$ | x-2*y | 3 |
| $\langle \text{term} \rangle - \langle \text{expr} \rangle$ | x-2*y | 4 |
| $\langle \text{term} \rangle - \langle \text{term} \rangle$ | x-2*y | 7 |
| $\langle \text{term} \rangle - \langle \text{factor} \rangle$ | x-2*y | 9 |
| $\langle \text{term} \rangle - \text{id}$ | x-2*y | shift |
| $\langle \text{term} \rangle -$ | -2*y | shift |
| $\langle \text{term} \rangle$ | 2*y | 5 |
| $\langle \text{factor} \rangle * \langle \text{term} \rangle$ | 2*y | 7 |
| $\langle \text{factor} \rangle * \langle \text{factor} \rangle$ | 2*y | 8 |
| $\langle \text{factor} \rangle * \text{num}$ | 2*y | shift |
| $\langle \text{factor} \rangle *$ | *y | shift |
| $\langle \text{factor} \rangle$ | y | 9 |
| id | y | shift |
| | | accepted |

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:

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

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

We can rewrite this as:

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

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

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

Applying the transformation gives

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

With this grammar, a top-down parser will

- terminate
- backtrack on some inputs

Example

This cleaner grammar defines the same language

| | | | | |
|---|--|---------------------------------|-------|---|
| 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 | | $\langle \text{factor} \rangle$ | $::=$ | num |
| 9 | | | | id |

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 | | | | $-\langle \text{term} \rangle \langle \text{expr}' \rangle$ |
| 5 | | | | 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 | | | | $/\langle \text{factor} \rangle \langle \text{term}' \rangle$ |
| 9 | | | | ϵ |
| 10 | | $\langle \text{factor} \rangle$ | ::= | num |
| 11 | | | | 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): left to right scan, left-most derivation, 1-token lookahead; and

LR(1): left to right scan, right-most derivation, 1-token lookahead

Predictive parsing

Basic idea:

For any two productions $A \rightarrow \alpha \mid \beta$, we would like a distinct way of choosing the correct production to expand.

For $\alpha \in V^*$ and $k \in \mathbf{N}$, define $\text{FIRST}_k(\alpha)$ as the set of terminal strings of length less than or equal to k that appear first in a string derived from α . That is, if $\alpha \Rightarrow^* w \in V_t^*$, then $w|_k \in \text{FIRST}_k(\alpha)$.

Key property:

Whenever two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

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

for some k . If $k = 1$, then the parser could 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

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

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

Consider a *right-recursive* version of the 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 | | | | $\langle \text{term} \rangle - \langle \text{expr} \rangle$ |
| 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 | | $\langle \text{factor} \rangle$ | ::= | num |
| 9 | | | | id |

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

$$\text{FIRST}_1(2) \cap \text{FIRST}_1(3) \cap \text{FIRST}_1(4) \neq \emptyset$$

This grammar *fails* the test.

Note: *This grammar is right-associative.*

Example

There are two nonterminals that must be left-factored:

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

Applying the transformation gives us:

$$\begin{aligned}\langle \text{expr} \rangle & ::= \langle \text{term} \rangle \langle \text{expr}' \rangle \\ \langle \text{expr}' \rangle & ::= + \langle \text{expr} \rangle \\ & \quad | - \langle \text{expr} \rangle \\ & \quad | \epsilon \\ \langle \text{term} \rangle & ::= \langle \text{factor} \rangle \langle \text{term}' \rangle \\ \langle \text{term}' \rangle & ::= * \langle \text{term} \rangle \\ & \quad | / \langle \text{term} \rangle \\ & \quad | \epsilon\end{aligned}$$

Example

Substituting back into the grammar yields

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

Now, selection requires only a single token lookahead.

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

Example

| | Sentential form | Input |
|----|--|----------------------|
| - | $\langle \text{goal} \rangle$ | $\uparrow x - 2 * y$ |
| 1 | $\langle \text{expr} \rangle$ | $\uparrow x - 2 * y$ |
| 2 | $\langle \text{term} \rangle \langle \text{expr}' \rangle$ | $\uparrow x - 2 * y$ |
| 6 | $\langle \text{factor} \rangle \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $\uparrow x - 2 * y$ |
| 11 | $\text{id} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $\uparrow x - 2 * y$ |
| - | $\text{id} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x \uparrow - 2 * y$ |
| 9 | $\text{id} \epsilon \langle \text{expr}' \rangle$ | $x \uparrow - 2$ |
| 4 | $\text{id} - \langle \text{expr} \rangle$ | $x \uparrow - 2 * y$ |
| - | $\text{id} - \langle \text{expr} \rangle$ | $x - \uparrow 2 * y$ |
| 2 | $\text{id} - \langle \text{term} \rangle \langle \text{expr}' \rangle$ | $x - \uparrow 2 * y$ |
| 6 | $\text{id} - \langle \text{factor} \rangle \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - \uparrow 2 * y$ |
| 10 | $\text{id} - \text{num} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - \uparrow 2 * y$ |
| - | $\text{id} - \text{num} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - 2 \uparrow * y$ |
| 7 | $\text{id} - \text{num} * \langle \text{term} \rangle \langle \text{expr}' \rangle$ | $x - 2 \uparrow * y$ |
| - | $\text{id} - \text{num} * \langle \text{term} \rangle \langle \text{expr}' \rangle$ | $x - 2 * \uparrow y$ |
| 6 | $\text{id} - \text{num} * \langle \text{factor} \rangle \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - 2 * \uparrow y$ |
| 11 | $\text{id} - \text{num} * \text{id} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - 2 * \uparrow y$ |
| - | $\text{id} - \text{num} * \text{id} \langle \text{term}' \rangle \langle \text{expr}' \rangle$ | $x - 2 * y \uparrow$ |
| 9 | $\text{id} - \text{num} * \text{id} \langle \text{expr}' \rangle$ | $x - 2 * y \uparrow$ |
| 5 | $\text{id} - \text{num} * \text{id}$ | $x - 2 * y \uparrow$ |

The next symbol determined each choice correctly.

Back to left-recursion elimination

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

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

$$A \rightarrow A\alpha \mid \beta \mid \dots \mid \gamma$$

with

$$A \rightarrow NA'$$

$$N \rightarrow \beta \mid \dots \mid \gamma$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

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:

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

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

Recursive descent parsing

General idea: Turn the grammar into a set of mutually recursive functions!

- Each non-terminal maps to a function
- The body of the function for $A \in V_n$ is determined by the productions $A \rightarrow \alpha_1 \mid \dots \mid \alpha_k$
 - on function entry, use lookahead to determine the correct RHS $\alpha = \alpha_j$, say
 - in the body, generate code for each symbol of α in sequence
 - for a terminal symbol, the code consumes a matching input token
 - for a non-terminal symbol, the code invokes the non-terminal's function

Recursive descent parsing

In that manner, 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 have each function return the AST for the word parsed by it. The function for a production gobbles up the ASTs for the non-terminal's on the RHS and applies the appropriate AST constructor.

Alternatively, the functions use an auxiliary stack for AST fragments.

Non-recursive predictive parsing

Observation:

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

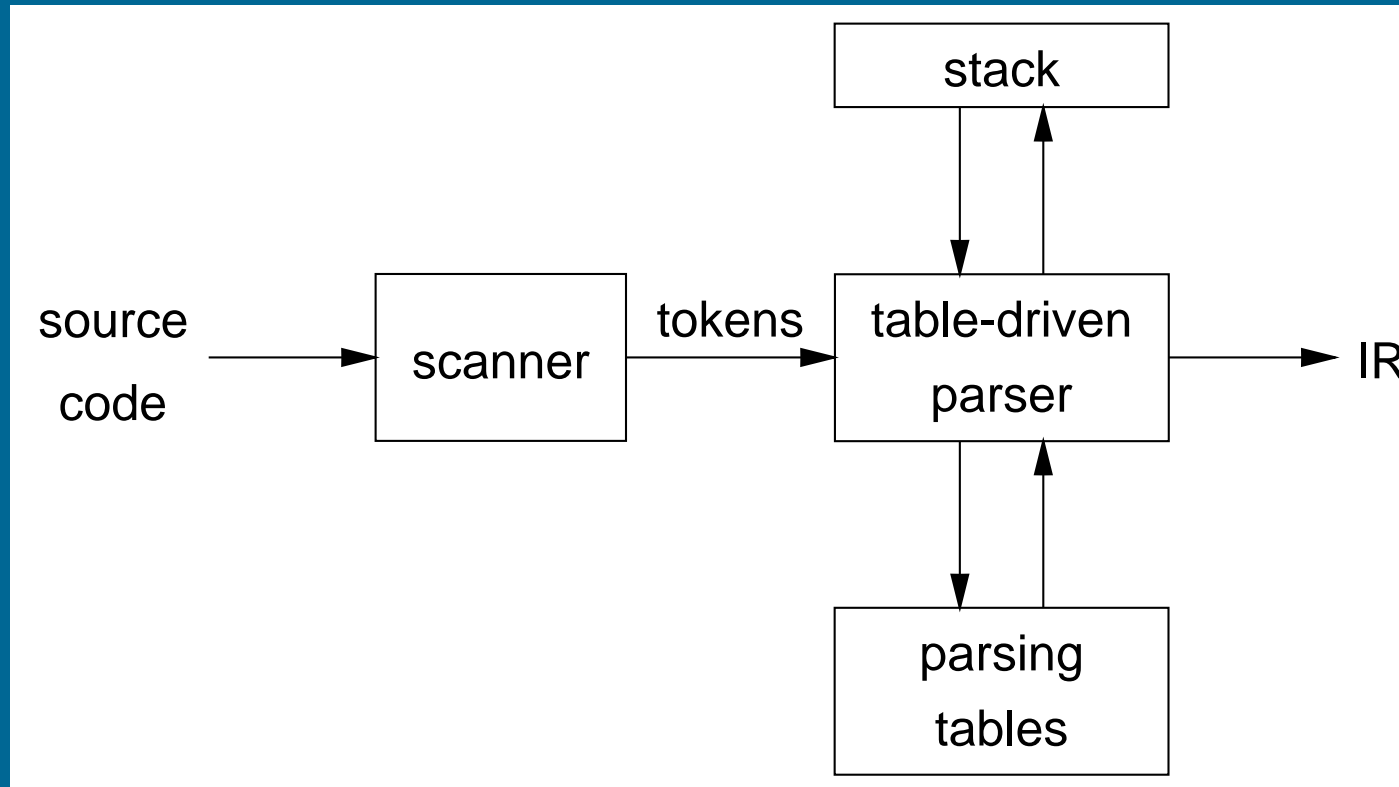
Using recursive procedure calls to implement a stack abstraction may not be particularly efficient.

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:

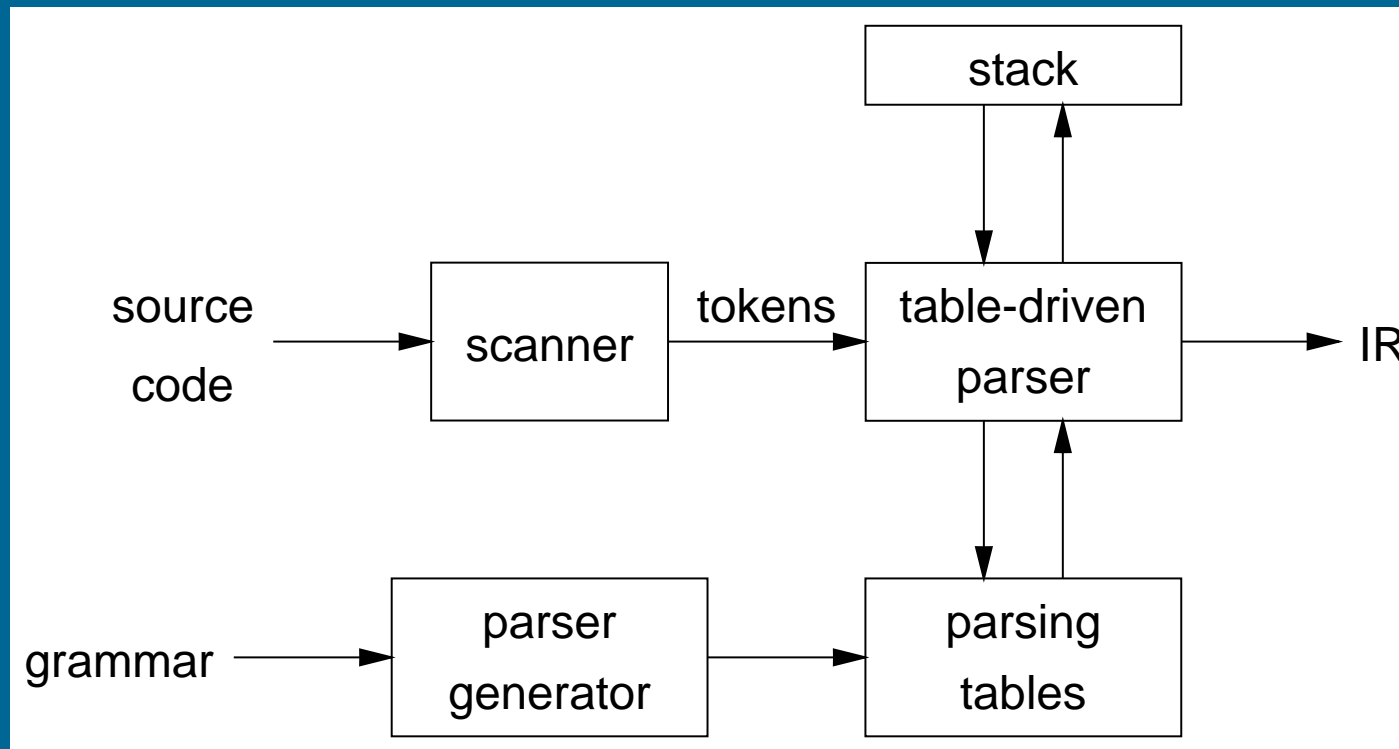


Rather than writing code, we build tables.

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  $\leftarrow$  0
Stack[tos]  $\leftarrow$  EOF
Stack[++tos]  $\leftarrow$  Start Symbol
token  $\leftarrow$  next_token()
repeat
  X  $\leftarrow$  Stack[tos]
  if X is a terminal or EOF then
    if X = token then
      pop X
      token  $\leftarrow$  next_token()
    else error()
  else /* X is a non-terminal */
    if  $M[X, \text{token}] = X \rightarrow Y_1 Y_2 \cdots Y_k$  then
      pop X
      push  $Y_k, Y_{k-1}, \dots, Y_1$ 
    else error()
until X = EOF
```

Non-recursive predictive parsing

What we need now is a parsing table M .

Our 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 | $\langle \text{expr}' \rangle$ | $::=$ | $+\langle \text{expr} \rangle$ |
| 4 | | | $-\langle \text{expr} \rangle$ |
| 5 | | | ϵ |
| 6 | $\langle \text{term} \rangle$ | $::=$ | $\langle \text{factor} \rangle \langle \text{term}' \rangle$ |
| 7 | $\langle \text{term}' \rangle$ | $::=$ | $*\langle \text{term} \rangle$ |
| 8 | | | $/\langle \text{term} \rangle$ |
| 9 | | | ϵ |
| 10 | $\langle \text{factor} \rangle$ | $::=$ | num |
| 11 | | | id |

Its parse table:

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

\dagger we use $\$$ to represent EOF

Computing $\text{FIRST} = \text{FIRST}_1$

For a string of grammar symbols α , define $\text{FIRST}(\alpha)$ as:

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

$\text{FIRST}(\alpha)$ contains the set of tokens valid in the initial position in α

To compute $\text{FIRST}(\alpha)$ it is sufficient to know $\text{FIRST}(X)$, for all $X \in V$:

$$\text{FIRST}(Y_1Y_2 \dots Y_k) = \text{FIRST}(Y_1) \oplus \text{FIRST}(Y_2) \oplus \dots \oplus \text{FIRST}(Y_k)$$

where

$$M \oplus N = \begin{cases} M & \varepsilon \notin M \\ (M \setminus \{\varepsilon\}) \cup N & \varepsilon \in M \end{cases}$$

Clearly, $\text{FIRST}(a) = \{a\}$ for $a \in V_t$.

Computing FIRST

- Initialize $\text{FIRST}(A) = \emptyset$, for all $A \in V_n$
- Repeat the following steps for all productions until no further additions can be made:
 1. If $A \rightarrow \varepsilon$ then:
$$\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \{\varepsilon\}$$
 2. If $A \rightarrow Y_1 Y_2 \cdots Y_k$:
$$\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup (\text{FIRST}(Y_1) \oplus \text{FIRST}(Y_2) \oplus \dots \oplus \text{FIRST}(Y_k))$$

FOLLOW

For a non-terminal A , define $\text{FOLLOW}(A)$ as

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

That is, $\text{FOLLOW}(A) = \{a \mid S\$ \Rightarrow^* \alpha A a \beta\}$

Thus, a non-terminal's FOLLOW set specifies the tokens that can legally appear after it, with $\$$ acting as end of input marker.

A terminal symbol has no FOLLOW set.

To build $\text{FOLLOW}(A)$:

1. Initialize $\text{FOLLOW}(A) = \emptyset$, for $A \in V_n$, $A \neq S$, and $\text{FOLLOW}(S) = \{\$\}$
2. Repeat the following steps for all productions $A \rightarrow \alpha B \beta$ until no further additions can be made:
 - (a) $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) - \{\epsilon\})$
 - (b) If $\epsilon \in \text{FIRST}(\beta)$, then
 $\text{FOLLOW}(B) \leftarrow \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

LL(1) grammars

Previous definition

A grammar G has a deterministic unambiguous predictive parser if for all non-terminals A , each distinct pair of productions $A \rightarrow \beta$ and $A \rightarrow \gamma$ satisfy the condition $\text{FIRST}(\beta) \cap \text{FIRST}(\gamma) = \phi$.

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

Revised definition

A grammar G is LL(1) iff. for each set of productions

$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$:

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

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. An ϵ -free grammar where each alternative expansion for A begins with a distinct terminal is a *simple* LL(1) grammar.

Example

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

LL(1) parse table construction

Input: Grammar G

Output: Parsing table M

Method:

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

Example

Our long-suffering expression grammar:

$$\begin{array}{l|l}
 S \rightarrow E & T \rightarrow FT' \\
 E \rightarrow TE' & T' \rightarrow *T \mid /T \mid \varepsilon \\
 E' \rightarrow +E \mid -E \mid \varepsilon & F \rightarrow \text{id} \mid \text{num}
 \end{array}$$

| | FIRST | FOLLOW |
|------|-------------------------|---------------------|
| S | {num, id} | { $\$$ } |
| E | {num, id} | { $\$$ } |
| E' | { ε , +, -} | { $\$$ } |
| T | {num, id} | {+, -, $\$$ } |
| T' | { ε , *, /} | {+, -, $\$$ } |
| F | {num, id} | {+, -, *, /, $\$$ } |
| id | {id} | - |
| num | {num} | - |
| * | {*} | - |
| / | {/} | - |
| + | {+} | - |
| - | {-} | - |

| | id | num | + | - | * | / | $\$$ |
|------|---------------------------|----------------------------|------------------------------|------------------------------|---------------------|---------------------|------------------------------|
| S | $S \rightarrow E$ | $S \rightarrow E$ | - | - | - | - | - |
| E | $E \rightarrow TE'$ | $E \rightarrow TE'$ | - | - | - | - | - |
| E' | - | - | $E' \rightarrow +E$ | $E' \rightarrow -E$ | - | - | $E' \rightarrow \varepsilon$ |
| T | $T \rightarrow FT'$ | $T \rightarrow FT'$ | - | - | - | - | - |
| T' | - | - | $T' \rightarrow \varepsilon$ | $T' \rightarrow \varepsilon$ | $T' \rightarrow *T$ | $T' \rightarrow /T$ | $T' \rightarrow \varepsilon$ |
| F | $F \rightarrow \text{id}$ | $F \rightarrow \text{num}$ | - | - | - | - | - |

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  $M[X,token] = X \rightarrow Y_1Y_2 \cdots Y_k$  then
            pop X
            pop node for X
            build node for each child and
            make it a child of node for X
            push  $n_k, Y_k, n_{k-1}, Y_{k-1}, \cdots, n_1, Y_1$ 
        else error()
until X = EOF
```

A grammar that is not LL(1)

$$\begin{aligned}\langle \text{stmt} \rangle & ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \\ & \quad | \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \text{ else } \langle \text{stmt} \rangle \\ & \quad | \dots\end{aligned}$$

Left-factored:

$$\begin{aligned}\langle \text{stmt} \rangle & ::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{stmt} \rangle \langle \text{stmt}' \rangle | \dots \\ \langle \text{stmt}' \rangle & ::= \text{else } \langle \text{stmt} \rangle | \varepsilon\end{aligned}$$

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

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

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

On seeing `else`, conflict between choosing

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

\Rightarrow grammar is not LL(1)!

The fix:

Put priority on $\langle \text{stmt}' \rangle ::= \text{else } \langle \text{stmt} \rangle$ 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., $\text{SYNCH}(a) = V_t - \{a\}$)

Some definitions

Recall

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

- If $\alpha \in V_t^*$, then α is 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

| | | | | |
|---|--|-----|---------------|--------|
| 1 | | S | \rightarrow | $aABe$ |
| 2 | | A | \rightarrow | Abc |
| 3 | | | | b |
| 4 | | B | \rightarrow | d |

and the input string $abbcd$

| Prod'n. | Sentential Form |
|---------|--|
| 3 | a b bcde |
| 2 | a Abc de |
| 4 | aA d e |
| 1 | aABe |
| — | S |

Scan the input and find valid sentential forms!

Handles

What are we trying to find?

A substring α of the tree's upper frontier that

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

We call such a string a *handle*.

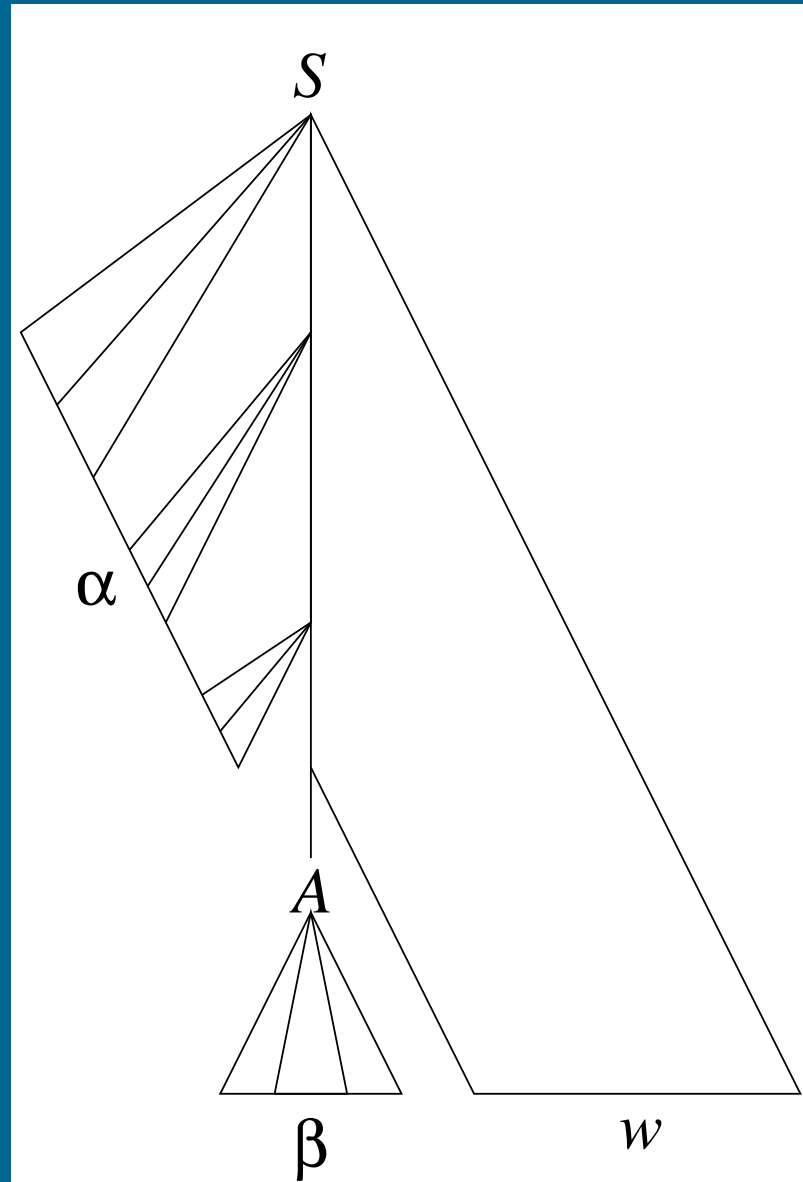
Formally:

In a right-sentential form $\alpha\beta w$, the string β is a handle for production $A \rightarrow \beta$

i.e., if $S \Rightarrow_{\text{rm}}^* \alpha A w \Rightarrow_{\text{rm}} \alpha \beta w$ then β is a handle for $A \rightarrow \beta$ in $\alpha \beta w$

All right-sentential forms have a suffix containing only terminal symbols.

Handles



The handle $A \rightarrow \beta$ 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 $A \rightarrow \beta$ applied to take γ_{i-1} to γ_i
3. \Rightarrow a unique position k at which $A \rightarrow \beta$ is applied
4. \Rightarrow a unique handle $A \rightarrow \beta$

Example

The left-recursive expression grammar

(original form)

| | |
|---|---|
| 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 | $\langle \text{factor} \rangle ::= \text{num}$ |
| 9 | id |

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

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 for $A \rightarrow \beta$ on the stack, *reduce*:
 - i) pop $|\beta|$ symbols off the stack
 - ii) push A onto the stack

Example: back to $x - 2 * y$

| | Stack | Input | Action |
|---|--|---------------|----------|
| | \$ | id - num * id | shift |
| 1 $\langle \text{goal} \rangle ::= \langle \text{expr} \rangle$ | \$ <u>id</u> | - num * id | reduce 9 |
| 2 $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$ | \$ <u>$\langle \text{factor} \rangle$</u> | - num * id | reduce 7 |
| 3 $\langle \text{expr} \rangle - \langle \text{term} \rangle$ | \$ <u>$\langle \text{term} \rangle$</u> | - num * id | reduce 4 |
| 4 $\langle \text{term} \rangle$ | \$ <u>$\langle \text{expr} \rangle$</u> | - num * id | shift |
| 5 $\langle \text{term} \rangle ::= \langle \text{term} \rangle * \langle \text{factor} \rangle$ | \$ $\langle \text{expr} \rangle -$ <u>num</u> | num * id | shift |
| 6 $\langle \text{term} \rangle / \langle \text{factor} \rangle$ | \$ $\langle \text{expr} \rangle -$ <u>$\langle \text{factor} \rangle$</u> | * id | reduce 8 |
| 7 $\langle \text{factor} \rangle$ | \$ $\langle \text{expr} \rangle -$ <u>$\langle \text{term} \rangle$</u> | * id | reduce 7 |
| 8 $\langle \text{factor} \rangle ::= \text{num}$ | \$ $\langle \text{expr} \rangle -$ <u>$\langle \text{term} \rangle * \text{id}$</u> | * id | shift |
| 9 id | \$ $\langle \text{expr} \rangle -$ <u>$\langle \text{term} \rangle * \langle \text{factor} \rangle$</u> | id | shift |
| | \$ $\langle \text{expr} \rangle -$ <u>$\langle \text{term} \rangle$</u> | | reduce 9 |
| | \$ <u>$\langle \text{expr} \rangle$</u> | | reduce 5 |
| | \$ <u>$\langle \text{goal} \rangle$</u> | | reduce 3 |
| | | | reduce 1 |
| | | | accept |

1. Shift until top of stack is the right end of a handle
2. Find the left end of the handle and reduce

5 shifts + 9 reduces + 1 accept

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

But how do we know

- that there is a complete handle on the stack?
- which handle to use?

LR parsing: key insight

Recognize handles with a DFA [Knuth1965]

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

LR parsing

The skeleton parser:

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

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

Example tables

| state | ACTION | | | | GOTO | | |
|-------|--------|----|----|-----|-------------------------------|-------------------------------|---------------------------------|
| | id | + | * | \$ | $\langle \text{expr} \rangle$ | $\langle \text{term} \rangle$ | $\langle \text{factor} \rangle$ |
| 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 | $\langle \text{goal} \rangle$ | $::=$ | $\langle \text{expr} \rangle$ |
| 2 | $\langle \text{expr} \rangle$ | $::=$ | $\langle \text{term} \rangle + \langle \text{expr} \rangle$ |
| 3 | | $ $ | $\langle \text{term} \rangle$ |
| 4 | $\langle \text{term} \rangle$ | $::=$ | $\langle \text{factor} \rangle * \langle \text{term} \rangle$ |
| 5 | | $ $ | $\langle \text{factor} \rangle$ |
| 6 | $\langle \text{factor} \rangle$ | $::=$ | id |

Note: This is a simple little right-recursive grammar; *not* the same 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. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$

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

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 $A \rightarrow \beta$ 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

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:

- $\text{closure}(I)$ to build its states
- $\text{goto}(I, X)$ 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  $[A \rightarrow \alpha \bullet B\beta] \in I$ 
    add  $[B \rightarrow \bullet\gamma]$  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, $\text{GOTO}(I, X)$ is the closure of the set of all items

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

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

$\text{GOTO}(I, X)$ 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 $[S' \rightarrow \bullet S\$]$, where

S' is the start symbol of the augmented grammar G'

S is the start symbol of G

$\$$ represents EOF

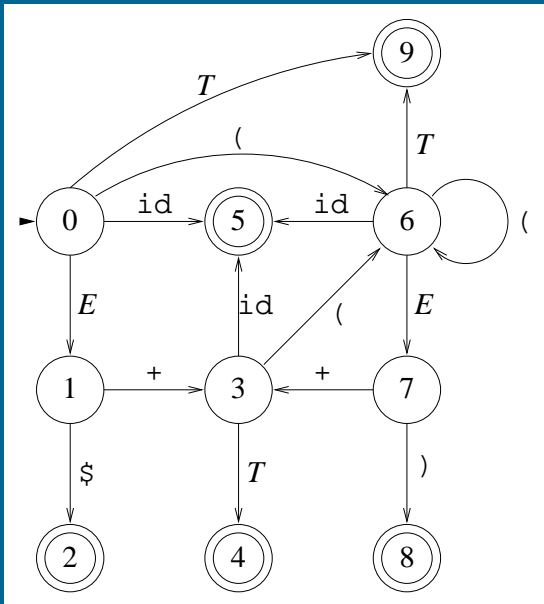
To compute the collection of sets of LR(0) items

```
function items( $G'$ )
   $s_0 \leftarrow \text{closure}_0(\{[S' \rightarrow \bullet S\$]\})$ 
   $\mathcal{S} \leftarrow \{s_0\}$ 
  repeat
    for each set of items  $s \in \mathcal{S}$ 
      for each grammar symbol  $X$ 
        if  $\text{goto}_0(s, X) \neq \emptyset$  and  $\text{goto}_0(s, X) \notin \mathcal{S}$ 
          add  $\text{goto}_0(s, X)$  to  $\mathcal{S}$ 
  until no more item sets can be added to  $\mathcal{S}$ 
  return  $\mathcal{S}$ 
```

LR(0) example

| | |
|---|---------------------------|
| 1 | $S \rightarrow E\$$ |
| 2 | $E \rightarrow E + T$ |
| 3 | $\quad \quad \quad T$ |
| 4 | $T \rightarrow \text{id}$ |
| 5 | $\quad \quad \quad (E)$ |

The corresponding CFMSM:

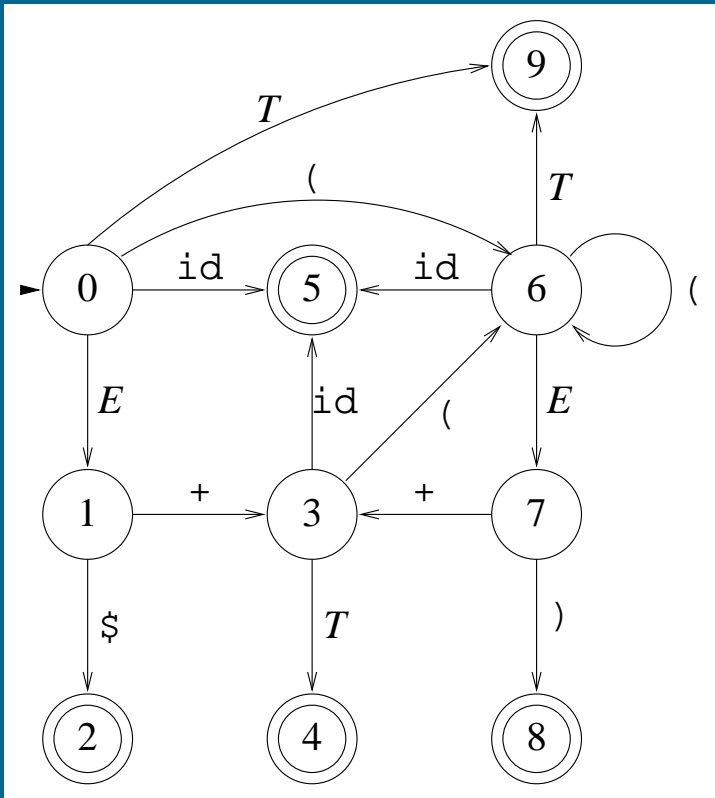


| | |
|-------------------------------------|---|
| $I_0 : S \rightarrow \bullet E \$$ | $I_4 : E \rightarrow E + T \bullet$ |
| $E \rightarrow \bullet E + T$ | $I_5 : T \rightarrow \text{id} \bullet$ |
| $E \rightarrow \bullet T$ | $I_6 : T \rightarrow (\bullet E)$ |
| $T \rightarrow \bullet \text{id}$ | $E \rightarrow \bullet E + T$ |
| $T \rightarrow \bullet (E)$ | $E \rightarrow \bullet T$ |
| $I_1 : S \rightarrow E \bullet \$$ | $T \rightarrow \bullet \text{id}$ |
| $E \rightarrow E \bullet + T$ | $T \rightarrow \bullet (E)$ |
| $I_2 : S \rightarrow E \$ \bullet$ | $I_7 : T \rightarrow (E \bullet)$ |
| $I_3 : E \rightarrow E + \bullet T$ | $E \rightarrow E \bullet + T$ |
| $T \rightarrow \bullet \text{id}$ | $I_8 : T \rightarrow (E) \bullet$ |
| $T \rightarrow \bullet (E)$ | $I_9 : E \rightarrow T \bullet$ |

Constructing the LR(0) parsing table

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

LR(0) example



| state | ACTION | | | | | GOTO | |
|-------|--------|-----|-----|-----|-----|----------|-------------------|
| | id | (|) | + | \$ | <i>S</i> | <i>E</i> <i>T</i> |
| 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:

- $A \rightarrow \varepsilon \mid a\alpha$
 \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:

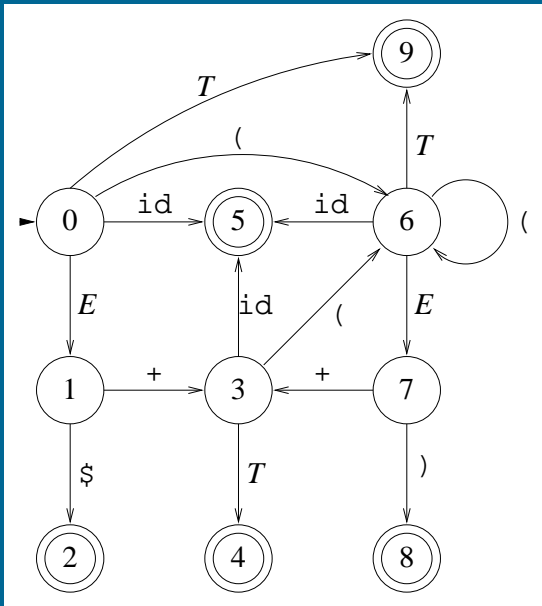
1. construct the collection of sets of LR(0) items for G'
2. state i of the CFM is constructed from I_i
 - (a) $[A \rightarrow \alpha \bullet a \beta] \in I_i$ and $\text{goto}_0(I_i, a) = I_j$
 $\Rightarrow \text{ACTION}[i, a] \leftarrow \text{"shift } j\text{"}, \underline{\forall a \neq \$}$
 - (b) $[A \rightarrow \alpha \bullet] \in I_i, A \neq S'$
 $\Rightarrow \text{ACTION}[i, a] \leftarrow \text{"reduce } A \rightarrow \alpha\text{"}, \underline{\forall a \in \text{FOLLOW}(A)}$
 - (c) $\underline{[S' \rightarrow S \bullet \$]} \in I_i$
 $\Rightarrow \text{ACTION}[i, \underline{\$}] \leftarrow \text{"accept"}$
3. $\text{goto}_0(I_i, A) = I_j$
 $\Rightarrow \text{GOTO}[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser s_0 is $\text{closure}_0([S' \rightarrow \bullet S \$])$

From previous example

| | |
|---|---------------------------------|
| 1 | $S \rightarrow E\$$ |
| 2 | $E \rightarrow E + T$ |
| 3 | $\quad \quad \quad \quad T$ |
| 4 | $T \rightarrow id$ |
| 5 | $\quad \quad \quad \quad (E)$ |

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

| state | ACTION | | | | | GOTO | | |
|-------|--------|----|----|----|-----|----------|----------|----------|
| | id | (| + | \$ | | <i>S</i> | <i>E</i> | <i>T</i> |
| 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)

| | | | |
|---|-----|---------------|-------------|
| 1 | S | \rightarrow | $E\$$ |
| 2 | E | \rightarrow | $E + T$ |
| 3 | | $ $ | T |
| 4 | T | \rightarrow | $T * F$ |
| 5 | | $ $ | F |
| 6 | F | \rightarrow | id |
| 7 | | $ $ | (E) |

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

$$I_0 : S \rightarrow \bullet E \$$$

$$E \rightarrow \bullet E + T$$

$$E \rightarrow \bullet T$$

$$T \rightarrow \bullet T * F$$

$$T \rightarrow \bullet F$$

$$F \rightarrow \bullet \text{id}$$

$$F \rightarrow \bullet (E)$$

$$I_1 : S \rightarrow E \bullet \$$$

$$E \rightarrow E \bullet + T$$

$$I_2 : S \rightarrow E \$ \bullet$$

$$I_3 : E \rightarrow E + \bullet T$$

$$T \rightarrow \bullet T * F$$

$$T \rightarrow \bullet F$$

$$F \rightarrow \bullet \text{id}$$

$$F \rightarrow \bullet (E)$$

$$I_4 : T \rightarrow F \bullet$$

$$I_5 : F \rightarrow \text{id} \bullet$$

$$I_6 : F \rightarrow (\bullet E)$$

$$E \rightarrow \bullet E + T$$

$$E \rightarrow \bullet T$$

$$T \rightarrow \bullet T * F$$

$$T \rightarrow \bullet F$$

$$F \rightarrow \bullet \text{id}$$

$$F \rightarrow \bullet (E)$$

$$I_7 : E \rightarrow T \bullet$$

$$T \rightarrow T \bullet * F$$

$$I_8 : T \rightarrow T * \bullet F$$

$$F \rightarrow \bullet \text{id}$$

$$F \rightarrow \bullet (E)$$

$$I_9 : T \rightarrow T * F \bullet$$

$$I_{10} : F \rightarrow (E) \bullet$$

$$I_{11} : E \rightarrow E + T \bullet$$

$$T \rightarrow T \bullet * F$$

$$I_{12} : F \rightarrow (E \bullet)$$

$$E \rightarrow E \bullet + T$$

Example: But it is SLR(1)

| state | ACTION | | | | | | GOTO | | | |
|-------|--------|----|----|----|-----|-----|----------|----------|----------|----------|
| | + | * | id | (|) | \$ | <i>S</i> | <i>E</i> | <i>T</i> | <i>F</i> |
| 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:

$$\begin{array}{l} S \rightarrow L = R \\ \quad | R \\ L \rightarrow *R \\ \quad | id \\ R \rightarrow L \end{array}$$

Its LR(0) item sets:

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

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

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 $[A \rightarrow X \bullet YZ, a]$, a has no direct use
 - in $[A \rightarrow XYZ \bullet, a]$, a is useful
- allows use of grammars that are not *uniquely invertible*[†]

The point: For $[A \rightarrow \alpha \bullet, a]$ and $[B \rightarrow \alpha \bullet, b]$, 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 $[A \rightarrow \alpha \bullet B\beta, a]$, 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 $[B \rightarrow \bullet \gamma, b]$ in the closure).

```
function closure1( $I$ )
repeat
  if  $[A \rightarrow \alpha \bullet B\beta, a] \in I$ 
    add  $[B \rightarrow \bullet \gamma, b]$  to  $I$ , where  $b \in \text{first}(\beta a)$ 
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, $\text{GOTO}(I, X)$ is the closure of the set of all items

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

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

$\text{goto}(I, X)$ represents state after recognizing X in state I .

```
function goto1( $I, X$ )
  let  $J$  be the set of items  $[A \rightarrow \alpha X \bullet \beta, a]$ 
    such that  $[A \rightarrow \alpha \bullet X \beta, a] \in I$ 
  return closure1( $J$ )
```

Building the LR(1) item sets for grammar G

We start the construction with the item $[S' \rightarrow \bullet S, \$]$, 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( $G'$ )
   $s_0 \leftarrow \text{closure1}(\{[S' \rightarrow \bullet S, \$]\})$ 
   $\mathcal{S} \leftarrow \{s_0\}$ 
  repeat
    for each set of items  $s \in \mathcal{S}$ 
      for each grammar symbol  $X$ 
        if  $\text{goto1}(s, X) \neq \emptyset$  and  $\text{goto1}(s, X) \notin \mathcal{S}$ 
          add  $\text{goto1}(s, X)$  to  $\mathcal{S}$ 
  until no more item sets can be added to  $\mathcal{S}$ 
  return  $\mathcal{S}$ 
```

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) $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $\text{goto}_1(I_i, a) = I_j$
 $\Rightarrow \text{ACTION}[i, a] \leftarrow \text{"shift } j\text{"}$
 - (b) $[A \rightarrow \alpha \bullet, \underline{a}] \in I_i, A \neq S'$
 $\Rightarrow \text{ACTION}[i, \underline{a}] \leftarrow \text{"reduce } A \rightarrow \alpha\text{"}$
 - (c) $[S' \rightarrow S \bullet, \$] \in I_i$
 $\Rightarrow \text{ACTION}[i, \$] \leftarrow \text{"accept"}$
3. $\text{goto}_1(I_i, A) = I_j$
 $\Rightarrow \text{GOTO}[i, A] \leftarrow j$
4. set undefined entries in ACTION and GOTO to "error"
5. initial state of parser s_0 is $\text{closure}_1([S' \rightarrow \bullet S, \$])$

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

$$\begin{array}{l}
 S \rightarrow L = R \\
 \quad | \quad R \\
 L \rightarrow *R \\
 \quad | \quad \text{id} \\
 R \rightarrow L
 \end{array}$$

| | |
|---|--|
| $I_0 : S' \rightarrow \bullet S, \quad \$$ $S \rightarrow \bullet L = R, \quad \$$ $S \rightarrow \bullet R, \quad \$$ $L \rightarrow \bullet *R, \quad =$ $L \rightarrow \bullet \text{id}, \quad =$ $R \rightarrow \bullet L, \quad \$$ $L \rightarrow \bullet *R, \quad \$$ $L \rightarrow \bullet \text{id}, \quad \$$ | $I_5 : L \rightarrow \text{id}\bullet, \quad = \$$ $I_6 : S \rightarrow L = \bullet R, \quad \$$ $R \rightarrow \bullet L, \quad \$$ $L \rightarrow \bullet *R, \quad \$$ $L \rightarrow \bullet \text{id}, \quad \$$ |
| $I_1 : S' \rightarrow S\bullet, \quad \$$ $I_2 : S \rightarrow L\bullet = R, \quad \$$ $R \rightarrow L\bullet, \quad \$$ $I_3 : S \rightarrow R\bullet, \quad \$$ $I_4 : L \rightarrow *\bullet R, \quad = \$$ $R \rightarrow \bullet L, \quad = \$$ $L \rightarrow \bullet *R, \quad = \$$ $L \rightarrow \bullet \text{id}, \quad = \$$ | $I_7 : L \rightarrow *R\bullet, \quad = \$$ $I_8 : R \rightarrow L\bullet, \quad = \$$ $I_9 : S \rightarrow L = R\bullet, \quad \$$ $I_{10} : R \rightarrow L\bullet, \quad \$$ $I_{11} : L \rightarrow *\bullet R, \quad \$$ $R \rightarrow \bullet L, \quad \$$ $L \rightarrow \bullet *R, \quad \$$ $L \rightarrow \bullet \text{id}, \quad \$$ $I_{12} : L \rightarrow \text{id}\bullet, \quad \$$ $I_{13} : L \rightarrow *R\bullet, \quad \$$ |

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):

| | | | |
|---|-------------------------------|---|---------------------------------|
| 1 | $S \rightarrow E$ | 4 | $T \rightarrow T * F$ |
| 2 | $E \rightarrow E + T$ | 5 | $\quad \quad \quad \quad F$ |
| 3 | $\quad \quad \quad \quad T$ | 6 | $F \rightarrow \text{id}$ |
| | | 7 | $\quad \quad \quad \quad (E)$ |

LR(1) item sets:

I_0 :

$S \rightarrow \bullet E, \quad \$$
 $E \rightarrow \bullet E + T, +\$$
 $E \rightarrow \bullet T, \quad +\$$
 $T \rightarrow \bullet T * F, * + \$$
 $T \rightarrow \bullet F, \quad * + \$$
 $F \rightarrow \bullet \text{id}, \quad * + \$$
 $F \rightarrow \bullet (E), \quad * + \$$

I'_0 : shifting (

$F \rightarrow (\bullet E), \quad * + \$$
 $E \rightarrow \bullet E + T, +)$
 $E \rightarrow \bullet T, \quad +)$
 $T \rightarrow \bullet T * F, * +)$
 $T \rightarrow \bullet F, \quad * +)$
 $F \rightarrow \bullet \text{id}, \quad * +)$
 $F \rightarrow \bullet (E), \quad * +)$

I''_0 : shifting (

$F \rightarrow (\bullet E), \quad * +)$
 $E \rightarrow \bullet E + T, +)$
 $E \rightarrow \bullet T, \quad +)$
 $T \rightarrow \bullet T * F, * +)$
 $T \rightarrow \bullet F, \quad * +)$
 $F \rightarrow \bullet \text{id}, \quad * +)$
 $F \rightarrow \bullet (E), \quad * +)$

Another example

Consider:

| | |
|---|--------------------|
| 0 | $S' \rightarrow S$ |
| 1 | $S \rightarrow CC$ |
| 2 | $C \rightarrow cC$ |
| 3 | d |

| state | ACTION | | | GOTO | |
|-------|----------|----------|-----|----------|----------|
| | <i>c</i> | <i>d</i> | \$ | <i>S</i> | <i>C</i> |
| 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:

| | |
|--------------------------------------|--------------------------------------|
| $I_0 : S' \rightarrow \bullet S, \$$ | $I_4 : C \rightarrow d\bullet, cd$ |
| $S \rightarrow \bullet CC, \$$ | $I_5 : S \rightarrow CC\bullet, \$$ |
| $C \rightarrow \bullet cC, cd$ | $I_6 : C \rightarrow c\bullet C, \$$ |
| $C \rightarrow \bullet d, cd$ | $C \rightarrow \bullet cC, \$$ |
| $I_1 : S' \rightarrow S\bullet, \$$ | $C \rightarrow \bullet d, \$$ |
| $I_2 : S \rightarrow C\bullet C, \$$ | $I_7 : C \rightarrow d\bullet, \$$ |
| $C \rightarrow \bullet cC, \$$ | $I_8 : C \rightarrow cC\bullet, cd$ |
| $C \rightarrow \bullet d, \$$ | $I_9 : C \rightarrow cC\bullet, \$$ |
| $I_3 : C \rightarrow c\bullet C, cd$ | |
| $C \rightarrow \bullet cC, cd$ | |
| $C \rightarrow \bullet d, cd$ | |

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

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

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) $[A \rightarrow \alpha \bullet a \beta, b] \in I_i$ and $\text{goto}_{01}(I_i, a) = I_j$
 $\Rightarrow \text{ACTION}[i, a] \leftarrow \text{"shift } j\text{"}$
 - (b) $[A \rightarrow \alpha \bullet, a] \in I_i, A \neq S'$
 $\Rightarrow \text{ACTION}[i, a] \leftarrow \text{"reduce } A \rightarrow \alpha\text{"}$
 - (c) $[S' \rightarrow S \bullet, \$] \in I_i \Rightarrow \text{ACTION}[i, \$] \leftarrow \text{"accept"}$
4. $\text{goto}_{01}(I_i, A) = I_j \Rightarrow \text{GOTO}[i, A] \leftarrow j$
5. set undefined entries in ACTION and GOTO to "error"
6. initial state of parser s_0 is $\text{closure}_1([S' \rightarrow \bullet S, \$])$

Example

Reconsider:

| | | | |
|---|------|---------------|------|
| 0 | S' | \rightarrow | S |
| 1 | S | \rightarrow | CC |
| 2 | C | \rightarrow | cC |
| 3 | | | d |

| | | |
|---------------------------------------|---------------------------------------|---------------------------------------|
| $I_0 : S' \rightarrow \bullet S, \$$ | $I_3 : C \rightarrow c \bullet C, cd$ | $I_6 : C \rightarrow c \bullet C, \$$ |
| $S \rightarrow \bullet CC, \$$ | $C \rightarrow \bullet cC, cd$ | $C \rightarrow \bullet cC, \$$ |
| $C \rightarrow \bullet cC, cd$ | $C \rightarrow \bullet d, cd$ | $C \rightarrow \bullet d, \$$ |
| $C \rightarrow \bullet d, cd$ | $I_4 : C \rightarrow d \bullet, cd$ | $I_7 : C \rightarrow d \bullet, \$$ |
| $I_1 : S' \rightarrow S \bullet, \$$ | $I_5 : S \rightarrow CC \bullet, \$$ | $I_8 : C \rightarrow cC \bullet, cd$ |
| $I_2 : S \rightarrow C \bullet C, \$$ | | $I_9 : C \rightarrow cC \bullet, \$$ |
| $C \rightarrow \bullet cC, \$$ | | |
| $C \rightarrow \bullet d, \$$ | | |

Merged states:

| |
|--|
| $I_{36} : C \rightarrow c \bullet C, cd\$$ |
| $C \rightarrow \bullet cC, cd\$$ |
| $C \rightarrow \bullet d, cd\$$ |
| $I_{47} : C \rightarrow d \bullet, cd\$$ |
| $I_{89} : C \rightarrow cC \bullet, cd\$$ |

| 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 $[S' \rightarrow \bullet S, \$]$
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:

$$\begin{array}{l} E \rightarrow E * E \\ | \\ E / E \\ | \\ E + E \\ | \\ E - E \\ | \\ (E) \\ | \\ -E \\ | \\ id \\ | \\ num \end{array}$$

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 synchronization 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
- extra operator or missing operator

(`yychar`)

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

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).

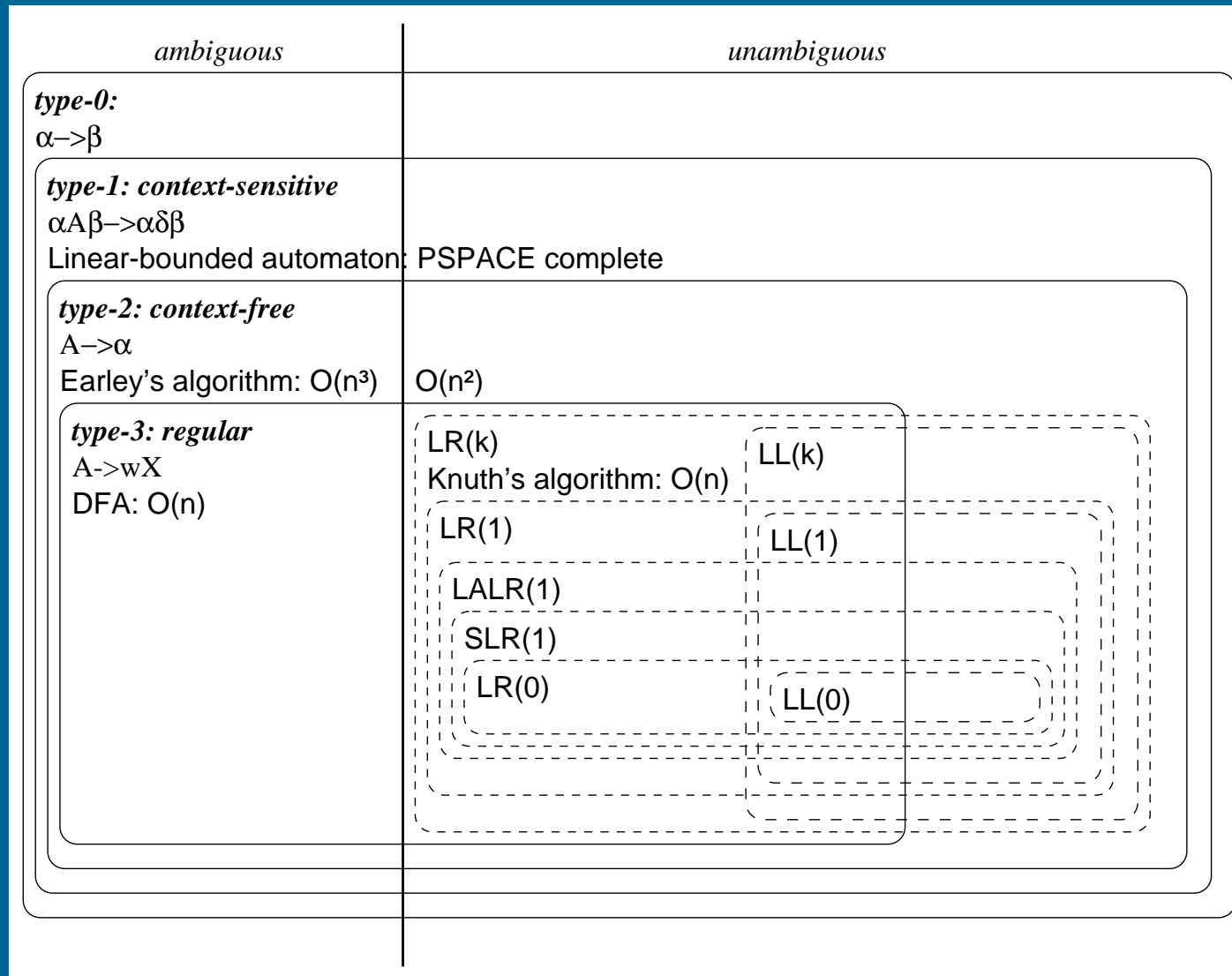
LL(k)

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).