CONTEXT-FREE GRAMMARS AND SYNTACTIC ANALYSIS

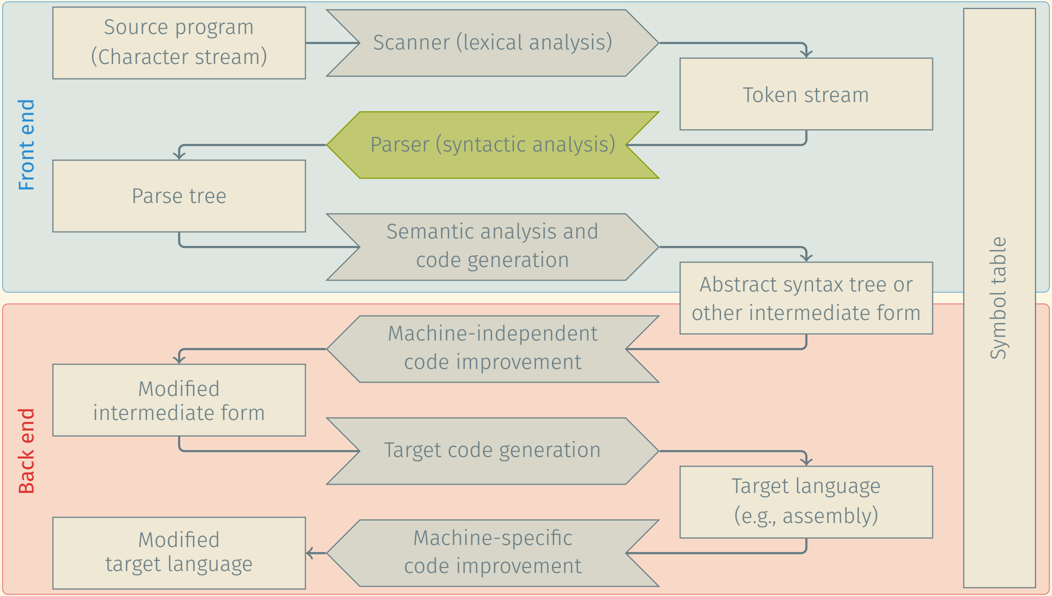
PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

PROGRAM TRANSLATION FLOW CHART



Goal

Convert the token stream produced by the scanner into a parse tree representing the syntactic structure of the program.

Goal

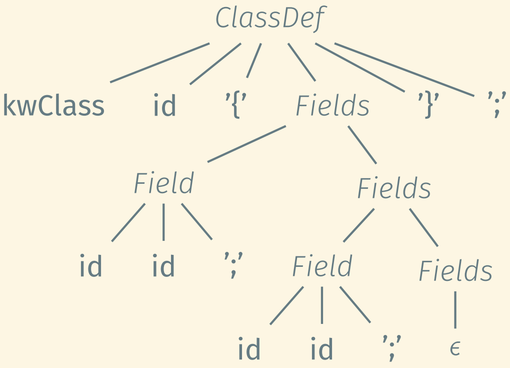
Convert the token stream produced by the scanner into a parse tree representing the syntactic structure of the program.

```
kwClass identifier '{'
    identifier identifier ';'
    identifier identifier ';'
'}' ';'
```

Goal

Convert the token stream produced by the scanner into a parse tree representing the syntactic structure of the program.

kwClass identifier '{'
  identifier identifier ';'
  identifier identifier ';'
'}' ';'



SYNTACTIC ANALYSIS

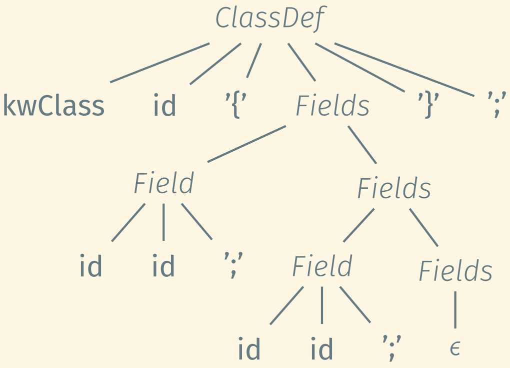
Goal

Convert the token stream produced by the scanner into a parse tree representing the syntactic structure of the program.

Tools

- Context-free grammars, LL(1)/LR(1) grammars
- (Deterministic) push-down automata, recursive-descent parsers

kwClass identifier '{'
  identifier identifier ';'
  identifier identifier ';'
'}' ';'



ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

A grammar for a subset of natural language

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{Phrase Verb Phrase}\ \textbf{.} \\
\textit{Phrase} &\rightarrow \textit{Noun} \\
\textit{Phrase} &\rightarrow \textit{Adjective Noun} \\
\textit{Adjective} &\rightarrow \text{'big'} \mid \text{'green'} \\
\textit{Noun} &\rightarrow \text{'cheese'} \mid \text{'Jim'} \\
\textit{Verb} &\rightarrow \text{'ate'}
\end{aligned}
$$

A grammar for a subset of natural language

$$\mathit{Sentence} \rightarrow \mathit{Phrase}\ \mathit{Verb}\ \mathit{Phrase}\ \textbf{.}$$
$$\mathit{Phrase} \rightarrow \mathit{Noun}$$
$$\mathit{Phrase} \rightarrow \mathit{Adjective}\ \mathit{Noun}$$
$$\mathit{Adjective} \rightarrow \textbf{'big'} \mid \textbf{'green'}$$
$$\mathit{Noun} \rightarrow \textbf{'cheese'} \mid \textbf{'Jim'}$$
$$\mathit{Verb} \rightarrow \textbf{'ate'}$$

Sentences in the language described by this grammar

- big Jim ate green cheese.
- green Jim ate green cheese.
- Jim ate cheese.
- cheese ate Jim.

A grammar for simple arithmetic expressions

$$Expr \rightarrow \texttt{(}\ Expr\ \texttt{)}$$

$$Expr \rightarrow Expr\ \texttt{+}\ Expr$$

$$Expr \rightarrow \texttt{++}\ Expr$$

$$Expr \rightarrow number$$

$$Expr \rightarrow identifier$$

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

CONTEXT-FREE GRAMMARS

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

Example:

$$Expr \rightarrow \text{'('}\ Expr\ \text{')'}$$
$$Expr \rightarrow Expr\ \text{'+'}\ Expr$$
$$Expr \rightarrow \text{'++'}\ Expr$$
$$Expr \rightarrow \textbf{number}$$
$$Expr \rightarrow \textbf{identifier}$$

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

Example:

$$Expr \rightarrow \text{'('}\ Expr\ \text{')'}$$
$$Expr \rightarrow Expr\ \text{'+'}\ Expr$$
$$Expr \rightarrow \text{'++'}\ Expr$$
$$Expr \rightarrow \textbf{number}$$
$$Expr \rightarrow \textbf{identifier}$$

Non-terminals: $Expr$

CONTEXT-FREE GRAMMARS

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

Example:

$$Expr \rightarrow \text{'('}\ Expr\ \text{')'}$$
$$Expr \rightarrow Expr\ \text{'+'}\ Expr$$
$$Expr \rightarrow \text{'++'}\ Expr$$
$$Expr \rightarrow \textbf{number}$$
$$Expr \rightarrow \textbf{identifier}$$

Non-terminals: $Expr$

Terminals: '(', ')', '+', '++', **number**, **identifier**

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

Example:

$$Expr \rightarrow \text{'('}\ Expr\ \text{')'}$$
$$Expr \rightarrow Expr\ \text{'+'}\ Expr$$
$$Expr \rightarrow \text{'++'}\ Expr$$
$$Expr \rightarrow \textbf{number}$$
$$Expr \rightarrow \textbf{identifier}$$

Non-terminals: *Expr*
Terminals: '(', ')', '+', '++', **number**, **identifier**
Rules: the rules above

Definition: Context-free grammar

A quadruple $G = (V, \Sigma, P, S)$ with

- A set of non-terminals or variables V,
- A set of terminals Σ,
- A set of rules or productions in the form

$$V \rightarrow (V \cup \Sigma)^*$$

- A start symbol $S \in V$.

Example:

$$Expr \rightarrow \text{'('}\ Expr\ \text{')'}$$
$$Expr \rightarrow Expr\ \text{'+'}\ Expr$$
$$Expr \rightarrow \text{'++'}\ Expr$$
$$Expr \rightarrow \textbf{number}$$
$$Expr \rightarrow \textbf{identifier}$$

Non-terminals:	*Expr*
Terminals:	'(', ')', '+', '++', **number**, **identifier**
Rules:	the rules above
Start symbol:	*Expr*

NOTATIONAL VARIATIONS IN PRODUCTIONS

Merging alternatives using '|':

$Phrase \rightarrow Noun$

$Phrase \rightarrow Adjective\ Noun$

$Phrase \rightarrow Noun \mid Adjective\ Noun$

NOTATIONAL VARIATIONS IN PRODUCTIONS

Merging alternatives using '|':

$Phrase \rightarrow Noun$

$Phrase \rightarrow Adjective\ Noun$

$Phrase \rightarrow Noun \mid Adjective\ Noun$

Backus-Naur form:

$Phrase \rightarrow Adjective\ Noun$

$Phrase ::= Adjective\ Noun$

Merging alternatives using '|':

$Phrase \rightarrow Noun$

$Phrase \rightarrow Adjective\ Noun$

$Phrase \rightarrow Noun \mid Adjective\ Noun$

Backus-Naur form:

$Phrase \rightarrow Adjective\ Noun$

$Phrase ::= Adjective\ Noun$

Optional components:

$FunctionDef \rightarrow DeclSpecs_{\mathrm{opt}}\ Decl\ DeclList_{\mathrm{opt}}\ CompoundStmt$

Merging alternatives using '|':

$$Phrase \rightarrow Noun$$
$$Phrase \rightarrow Adjective\ Noun$$

$$Phrase \rightarrow Noun \mid Adjective\ Noun$$

Backus-Naur form:

$$Phrase \rightarrow Adjective\ Noun$$

$$Phrase ::= Adjective\ Noun$$

Optional components:

$$FunctionDef \rightarrow DeclSpecs_{\text{opt}}\ Decl\ DeclList_{\text{opt}}\ CompoundStmt$$

Regular expression on RHS:

$$Expr \rightarrow Term\ (\text{op}\ Term)^*$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \rightarrow \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha\beta\gamma$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \rightarrow \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha \beta \gamma$$

Example:

$$\mathit{Sentence} \Rightarrow_G \mathit{Phrase}\ \mathit{Verb}\ \mathit{Phrase}$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \rightarrow \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha \beta \gamma$$

Example:

$$\textit{Sentence} \Rightarrow_G \textit{Phrase Verb Phrase}$$
$$\textit{Phrase Verb Phrase} \Rightarrow_G \textit{Noun Verb Phrase}$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \to \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha \beta \gamma$$

Example:

$$\begin{aligned} \mathit{Sentence} &\Rightarrow_G \mathit{Phrase\ Verb\ Phrase} \\ \mathit{Phrase\ Verb\ Phrase} &\Rightarrow_G \mathit{Noun\ Verb\ Phrase} \\ \mathit{Noun\ Verb\ Phrase} &\Rightarrow_G \mathit{Noun\ Verb\ Noun} \end{aligned}$$

DERIVATIONS

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \to \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha\beta\gamma$$

Example:

$$\begin{aligned} \textit{Sentence} &\Rightarrow_G \textit{Phrase Verb Phrase} \\ \textit{Phrase Verb Phrase} &\Rightarrow_G \textit{Noun Verb Phrase} \\ \textit{Noun Verb Phrase} &\Rightarrow_G \textit{Noun Verb Noun} \\ \textit{Noun Verb Noun} &\Rightarrow_G \text{Jim } \textit{Verb Noun} \end{aligned}$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \rightarrow \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha\beta\gamma$$

Example:

$$\begin{aligned} \textit{Sentence} &\Rightarrow_G \textit{Phrase Verb Phrase} \\ \textit{Phrase Verb Phrase} &\Rightarrow_G \textit{Noun Verb Phrase} \\ \textit{Noun Verb Phrase} &\Rightarrow_G \textit{Noun Verb Noun} \\ \textit{Noun Verb Noun} &\Rightarrow_G \text{Jim } \textit{Verb Noun} \\ \text{Jim } \textit{Verb Noun} &\Rightarrow_G \text{Jim ate } \textit{Noun} \end{aligned}$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \to \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha\beta\gamma$$

Example:

$$
\begin{aligned}
\mathit{Sentence} &\Rightarrow_G \mathit{Phrase\ Verb\ Phrase}\\
\mathit{Phrase\ Verb\ Phrase} &\Rightarrow_G \mathit{Noun\ Verb\ Phrase}\\
\mathit{Noun\ Verb\ Phrase} &\Rightarrow_G \mathit{Noun\ Verb\ Noun}\\
\mathit{Noun\ Verb\ Noun} &\Rightarrow_G \textbf{Jim}\ \mathit{Verb\ Noun}\\
\textbf{Jim}\ \mathit{Verb\ Noun} &\Rightarrow_G \textbf{Jim ate}\ \mathit{Noun}\\
\textbf{Jim ate}\ \mathit{Noun} &\Rightarrow_G \textbf{Jim ate cheese}
\end{aligned}
$$

Definition: Derivation

A sequence of rewriting operations that starts with the string $\sigma = S$ and repeats the following until σ contains only terminals:

- Choose a non-terminal X in σ and a production $X \rightarrow \beta$ in the grammar G.
- Replace X with β in σ.

As a picture:

$$\sigma = \alpha X \gamma \Rightarrow_G \sigma' = \alpha\beta\gamma$$

Example:

$$\begin{aligned}
\textit{Sentence} &\Rightarrow_G \textit{Phrase Verb Phrase}\\
\textit{Phrase Verb Phrase} &\Rightarrow_G \textit{Noun Verb Phrase}\\
\textit{Noun Verb Phrase} &\Rightarrow_G \textit{Noun Verb Noun}\\
\textit{Noun Verb Noun} &\Rightarrow_G \textbf{Jim}\ \textit{Verb Noun}\\
\textbf{Jim}\ \textit{Verb Noun} &\Rightarrow_G \textbf{Jim ate}\ \textit{Noun}\\
\textbf{Jim ate}\ \textit{Noun} &\Rightarrow_G \textbf{Jim ate cheese}
\end{aligned}$$

The intermediate strings are called sentential forms.

LANGUAGE DEFINED BY A CONTEXT-FREE GRAMMAR

We write $S \Rightarrow_G^* \sigma$ if there exists a sequence $S \Rightarrow_G \sigma_1 \Rightarrow_G \sigma_2 \Rightarrow_G \cdots \Rightarrow_G \sigma$.

LANGUAGE DEFINED BY A CONTEXT-FREE GRAMMAR

We write $S \Rightarrow_G^* \sigma$ if there exists a sequence $S \Rightarrow_G \sigma_1 \Rightarrow_G \sigma_2 \Rightarrow_G \cdots \Rightarrow_G \sigma$.

Definition: Language defined by a grammar

The language defined by a grammar $G = (V, \Sigma, P, S)$ is

$$\mathcal{L}(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow_G^* \sigma\}.$$

LANGUAGE DEFINED BY A CONTEXT-FREE GRAMMAR

We write $S \Rightarrow_G^* \sigma$ if there exists a sequence $S \Rightarrow_G \sigma_1 \Rightarrow_G \sigma_2 \Rightarrow_G \cdots \Rightarrow_G \sigma$.

Definition: Language defined by a grammar

The language defined by a grammar $G = (V, \Sigma, P, S)$ is

$$\mathcal{L}(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow_G^* \sigma\}.$$

If G is a context-free grammar, then $\mathcal{L}(G)$ is a context-free language.

We write $S \Rightarrow_G^* \sigma$ if there exists a sequence $S \Rightarrow_G \sigma_1 \Rightarrow_G \sigma_2 \Rightarrow_G \cdots \Rightarrow_G \sigma$.

Definition: Language defined by a grammar

The language defined by a grammar $G = (V, \Sigma, P, S)$ is

$$\mathcal{L}(G) = \{\sigma \in \Sigma^* \mid S \Rightarrow_G^* \sigma\}.$$

If G is a context-free grammar, then $\mathcal{L}(G)$ is a context-free language.

Example:

The "Jim-ate-cheese" grammar defines the language

$$\mathcal{L}(G) = \{\textbf{Jim ate cheese}, \textbf{Jim ate Jim}, \textbf{big Jim ate cheese}, \ldots\}.$$

CONTEXT-FREE LANGUAGES: EXAMPLES

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 0$$

$$S \to 1\ S\ 1$$

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 0$$

$$S \to 1\ S\ 1$$

$$\mathcal{L}(G) = \{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\} \quad (\overleftarrow{\sigma} \text{ is } \sigma \text{ written backwards.})$$

What is the language defined by the following grammar?

$$S \to \epsilon$$
$$S \to 0\ S\ 0$$
$$S \to 1\ S\ 1$$

$$\mathcal{L}(G) = \{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\} \quad (\overleftarrow{\sigma} \text{ is } \sigma \text{ written backwards.})$$

What is the language defined by the following grammar?

$$S \to \epsilon$$
$$S \to 0\ S\ 1$$

CONTEXT-FREE LANGUAGES: EXAMPLES

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 0$$

$$S \to 1\ S\ 1$$

$$\mathcal{L}(G) = \{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\} \quad (\overleftarrow{\sigma} \text{ is } \sigma \text{ written backwards.})$$

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 1$$

$$\mathcal{L}(G) = \{0^n1^n \mid n \geq 0\}$$

CONTEXT-FREE LANGUAGES: EXAMPLES

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 0$$

$$S \to 1\ S\ 1$$

$$\mathcal{L}(G) = \{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\} \quad (\overleftarrow{\sigma} \text{ is } \sigma \text{ written backwards.})$$

What is the language defined by the following grammar?

$$S \to \epsilon$$

$$S \to 0\ S\ 1$$

$$\mathcal{L}(G) = \{0^n 1^n \mid n \geq 0\}$$

Neither of these two languages is regular!

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

Sentence

Sentence

PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

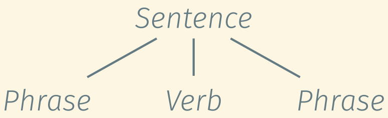
$Sentence \Rightarrow Phrase\ Verb\ Phrase$ $\qquad$ $Sentence$

PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$\textit{Sentence} \Rightarrow \textit{Phrase Verb Phrase}$$

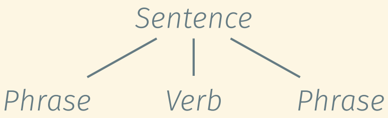


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

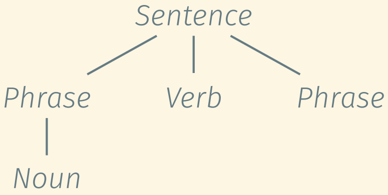
$$\begin{aligned} Sentence &\Rightarrow Phrase\ Verb\ Phrase \\ &\Rightarrow Noun\ Verb\ Phrase \end{aligned}$$



Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$
\begin{aligned}
\textit{Sentence} &\Rightarrow \textit{Phrase Verb Phrase} \\
&\Rightarrow \textit{Noun Verb Phrase}
\end{aligned}
$$

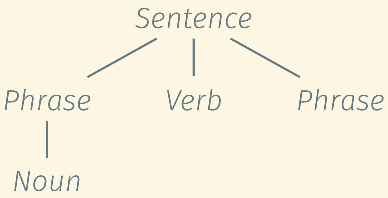


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

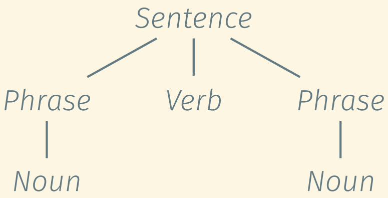
$$
\begin{aligned}
\mathit{Sentence} &\Rightarrow \mathit{Phrase\ Verb\ Phrase} \\
&\Rightarrow \mathit{Noun\ Verb\ Phrase} \\
&\Rightarrow \mathit{Noun\ Verb\ Noun}
\end{aligned}
$$



Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$\begin{aligned} \textit{Sentence} &\Rightarrow \textit{Phrase Verb Phrase} \\ &\Rightarrow \textit{Noun Verb Phrase} \\ &\Rightarrow \textit{Noun Verb Noun} \end{aligned}$$

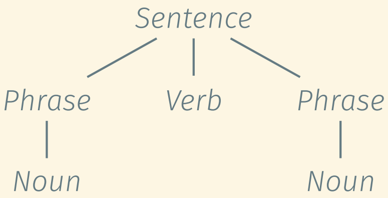


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$
\begin{aligned}
\mathit{Sentence} &\Rightarrow \mathit{Phrase}\ \mathit{Verb}\ \mathit{Phrase}\\
&\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Phrase}\\
&\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Noun}\\
&\Rightarrow \text{Jim}\ \mathit{Verb}\ \mathit{Noun}
\end{aligned}
$$

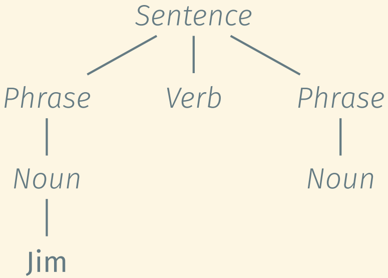


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$\begin{aligned} \mathit{Sentence} &\Rightarrow \mathit{Phrase\ Verb\ Phrase} \\ &\Rightarrow \mathit{Noun\ Verb\ Phrase} \\ &\Rightarrow \mathit{Noun\ Verb\ Noun} \\ &\Rightarrow \text{Jim}\ \mathit{Verb\ Noun} \end{aligned}$$

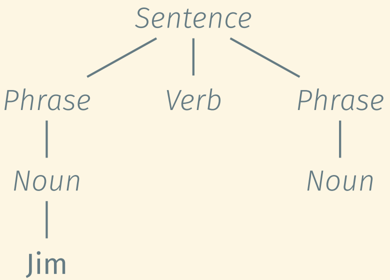


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$\begin{aligned} \mathit{Sentence} &\Rightarrow \mathit{Phrase}\ \mathit{Verb}\ \mathit{Phrase} \\ &\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Phrase} \\ &\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Noun} \\ &\Rightarrow \text{Jim}\ \mathit{Verb}\ \mathit{Noun} \\ &\Rightarrow \text{Jim ate}\ \mathit{Noun} \end{aligned}$$

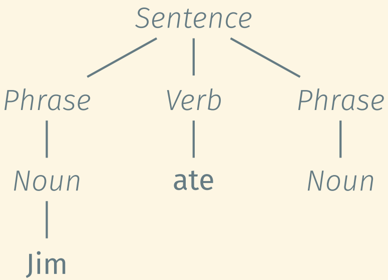


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

$$
\begin{aligned}
\mathit{Sentence} &\Rightarrow \mathit{Phrase}\ \mathit{Verb}\ \mathit{Phrase} \\
&\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Phrase} \\
&\Rightarrow \mathit{Noun}\ \mathit{Verb}\ \mathit{Noun} \\
&\Rightarrow \textbf{Jim}\ \mathit{Verb}\ \mathit{Noun} \\
&\Rightarrow \textbf{Jim ate}\ \mathit{Noun}
\end{aligned}
$$

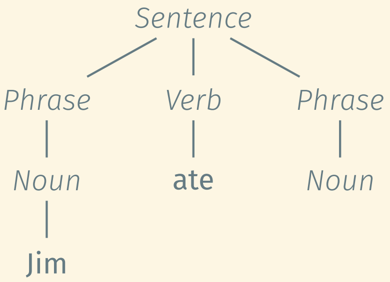


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is *S*.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

Sentence ⇒ *Phrase Verb Phrase*
⇒ *Noun Verb Phrase*
⇒ *Noun Verb Noun*
⇒ **Jim** *Verb Noun*
⇒ **Jim ate** *Noun*
⇒ **Jim ate cheese**

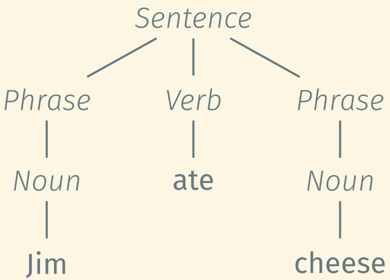


PARSE TREES

Every derivation can be represented by a parse tree:

- The root is S.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

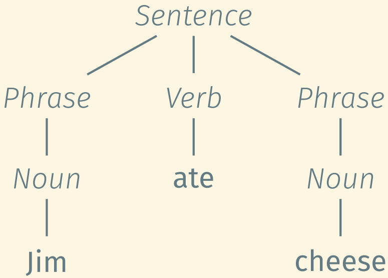
$$\begin{aligned} Sentence &\Rightarrow Phrase\ Verb\ Phrase \\ &\Rightarrow Noun\ Verb\ Phrase \\ &\Rightarrow Noun\ Verb\ Noun \\ &\Rightarrow \textbf{Jim}\ Verb\ Noun \\ &\Rightarrow \textbf{Jim ate}\ Noun \\ &\Rightarrow \textbf{Jim ate cheese} \end{aligned}$$



Every derivation can be represented by a parse tree:

- The root is *S*.
- The leaves, called the yield of the parse tree, are terminals.
- Every internal node is a non-terminal.
- The children of each non-terminal are the symbols it is replaced with.

Sentence ⇒ *Phrase Verb Phrase*
⇒ *Noun Verb Phrase*
⇒ *Noun Verb Noun*
⇒ Jim *Verb Noun*
⇒ Jim ate *Noun*
⇒ Jim ate cheese



Note: In general, there are multiple derivations with the same parse tree.

There are infinitely many context-free grammars generating a given context-free language:

There are infinitely many context-free grammars generating a given context-free language:

> *Just add arbitrary non-terminals to the right-hand sides of productions and then add ϵ-productions for these non-terminals.*

There are infinitely many context-free grammars generating a given context-free language:

> *Just add arbitrary non-terminals to the right-hand sides of productions and then add ϵ-productions for these non-terminals.*

There may be more than one parse tree for the same sentence generated by a grammar G. If this is the case, we call G ambiguous.

There are infinitely many context-free grammars generating a given context-free language:

> *Just add arbitrary non-terminals to the right-hand sides of productions and then add ϵ-productions for these non-terminals.*

There may be more than one parse tree for the same sentence generated by a grammar G. If this is the case, we call G ambiguous.

To allow parsing a programming language unambiguously, its grammar has to be unambiguous. (A bit of a tautology.)

There are infinitely many context-free grammars generating a given context-free language:

> *Just add arbitrary non-terminals to the right-hand sides of productions and then add ϵ-productions for these non-terminals.*

There may be more than one parse tree for the same sentence generated by a grammar G. If this is the case, we call G ambiguous.

To allow parsing a programming language unambiguously, its grammar has to be unambiguous. (A bit of a tautology.)

Some context-free languages are inherently ambiguous, that is, do not have unambiguous grammars. Usually, this is not the case for programming languages.

AMBIGUITY: EXAMPLE (1)

$$2 + 3 * 4$$

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

$E \rightarrow E \text{ '+' } E$

$E \rightarrow E \text{ '-' } E$

$E \rightarrow E \text{ '*' } E$

$E \rightarrow E \text{ '/' } E$

$E \rightarrow \text{'('} \; E \; \text{')'}$

AMBIGUITY: EXAMPLE (1)

$$2 + 3 * 4$$

$E \rightarrow$ num

$E \rightarrow$ id

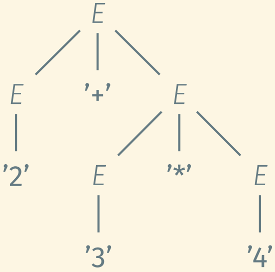
$E \rightarrow E$ '+' E

$E \rightarrow E$ '-' E

$E \rightarrow E$ '*' E

$E \rightarrow E$ '/' E

$E \rightarrow$ '(' E ')'



AMBIGUITY: EXAMPLE (1)

$$2 + 3 * 4$$

$E \rightarrow \text{num}$

$E \rightarrow \text{id}$

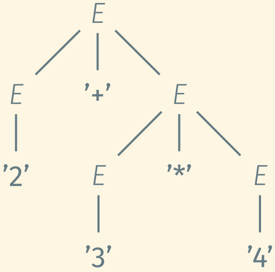
$E \rightarrow E \text{ '+' } E$

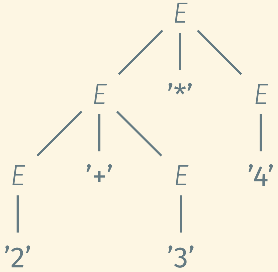
$E \rightarrow E \text{ '-' } E$

$E \rightarrow E \text{ '*' } E$

$E \rightarrow E \text{ '/' } E$

$E \rightarrow \text{'(' } E \text{ ')'}$





$$2 + 3 * 4$$

$E \rightarrow$ num

$E \rightarrow$ id

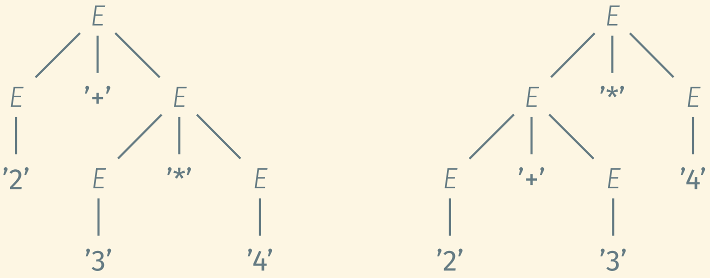
$E \rightarrow E$ '+' E

$E \rightarrow E$ '-' E

$E \rightarrow E$ '*' E

$E \rightarrow E$ '/' E

$E \rightarrow$ '(' E ')'



This grammar is ambiguous!

$$2 + 3 * 4$$

$E \rightarrow \textsf{num}$

$E \rightarrow \textsf{id}$

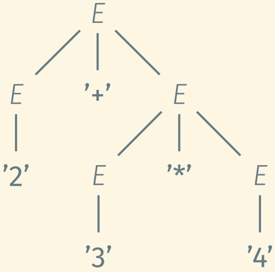
$E \rightarrow E$ '+' E

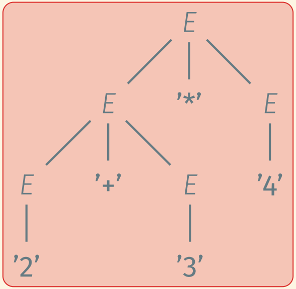
$E \rightarrow E$ '-' E

$E \rightarrow E$ '*' E

$E \rightarrow E$ '/' E

$E \rightarrow$ '(' E ')'





Violates precedence rules!

This grammar is ambiguous!

AMBIGUITY: EXAMPLE (2)

An unambiguous grammar for the same language:

$E \rightarrow T$

$E \rightarrow E$ '+' T

$E \rightarrow E$ '-' T

$T \rightarrow F$

$T \rightarrow T$ '*' F

$T \rightarrow T$ '/' F

$F \rightarrow$ num

$F \rightarrow$ id

$F \rightarrow$ '(' E ')'

AMBIGUITY: EXAMPLE (2)

An unambiguous grammar for the same language:

$E \rightarrow T$

$E \rightarrow E$ '+' T

$E \rightarrow E$ '-' T

$T \rightarrow F$

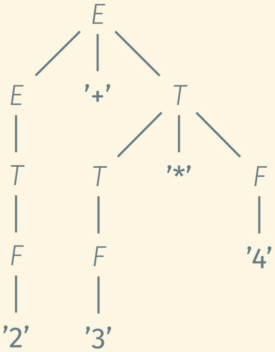
$T \rightarrow T$ '*' F

$T \rightarrow T$ '/' F

$F \rightarrow$ num

$F \rightarrow$ id

$F \rightarrow$ '(' E ')'



An unambiguous grammar for the same language:

$E \rightarrow T$

$E \rightarrow E$ '+' T

$E \rightarrow E$ '-' T

$T \rightarrow F$

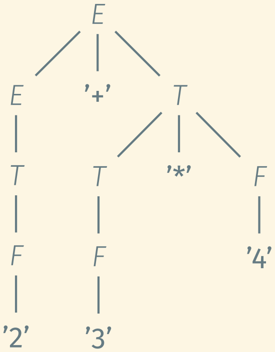
$T \rightarrow T$ '*' F

$T \rightarrow T$ '/' F

$F \rightarrow$ num

$F \rightarrow$ id

$F \rightarrow$ '(' E ')'



This grammar respects precedence rules.

AMBIGUITY: EXAMPLE (2)

An unambiguous grammar for the same language:

$E \rightarrow T$

$E \rightarrow E$ '+' T

$E \rightarrow E$ '-' T

$T \rightarrow F$

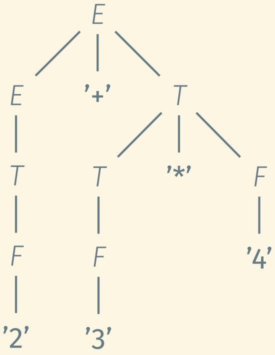
$T \rightarrow T$ '*' F

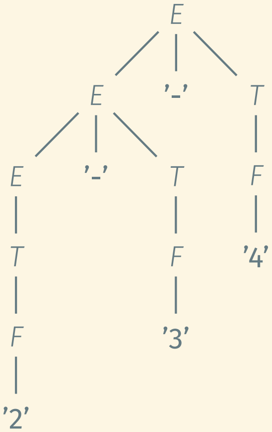
$T \rightarrow T$ '/' F

$F \rightarrow$ **num**

$F \rightarrow$ **id**

$F \rightarrow$ '(' E ')'





This grammar respects precedence rules.

It also respects left-associativity.

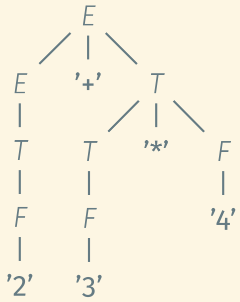
LEFTMOST AND RIGHTMOST DERIVATIONS

For every language defined by an unambiguous grammar, there is only one parse tree that generates each sentence in the language.

LEFTMOST AND RIGHTMOST DERIVATIONS

For every language defined by an unambiguous grammar, there is only one parse tree that generates each sentence in the language.

There are many different derivations corresponding to this parse tree.

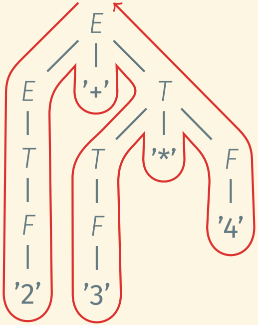


For every language defined by an unambiguous grammar, there is only one parse tree that generates each sentence in the language.

There are many different derivations corresponding to this parse tree.

Leftmost derivation: Replaces the leftmost non-terminal in each step.

$$
\begin{aligned}
E &\Rightarrow E \text{ '+' } T \\
&\Rightarrow T \text{ '+' } T \\
&\Rightarrow F \text{ '+' } T \\
&\Rightarrow \text{'2' '+' } T \\
&\Rightarrow \text{'2' '+' } T \text{ '*' } F \\
&\Rightarrow \text{'2' '+' } F \text{ '*' } F \\
&\Rightarrow \text{'2' '+' '3' '*' } F \\
&\Rightarrow \text{'2' '+' '3' '*' '4'}
\end{aligned}
$$



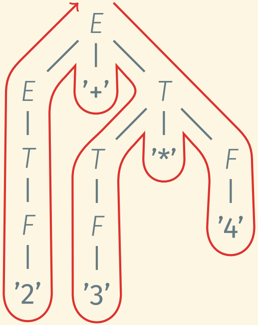
For every language defined by an unambiguous grammar, there is only one parse tree that generates each sentence in the language.

There are many different derivations corresponding to this parse tree.

Leftmost derivation: Replaces the leftmost non-terminal in each step.

Rightmost derivation Replaces the rightmost non-terminal in each step.

$$\begin{aligned} E &\Rightarrow E\ \text{'+'}\ T \\ &\Rightarrow T\ \text{'+'}\ T \\ &\Rightarrow F\ \text{'+'}\ T \\ &\Rightarrow \text{'2'}\ \text{'+'}\ T \\ &\Rightarrow \text{'2'}\ \text{'+'}\ T\ \text{'*'}\ F \\ &\Rightarrow \text{'2'}\ \text{'+'}\ F\ \text{'*'}\ F \\ &\Rightarrow \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{'*'}\ F \\ &\Rightarrow \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{'*'}\ \text{'4'} \end{aligned}$$



$$\begin{aligned} E &\Rightarrow E\ \text{'+'}\ T \\ &\Rightarrow E\ \text{'+'}\ T\ \text{'*'}\ F \\ &\Rightarrow E\ \text{'+'}\ T\ \text{'*'}\ \text{'4'} \\ &\Rightarrow E\ \text{'+'}\ F\ \text{'*'}\ \text{'4'} \\ &\Rightarrow E\ \text{'+'}\ \text{'3'}\ \text{'*'}\ \text{'4'} \\ &\Rightarrow T\ \text{'+'}\ \text{'3'}\ \text{'*'}\ \text{'4'} \\ &\Rightarrow F\ \text{'+'}\ \text{'3'}\ \text{'*'}\ \text{'4'} \\ &\Rightarrow \text{'2'}\ \text{'+'}\ \text{'3'}\ \text{'*'}\ \text{'4'} \end{aligned}$$

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

EFFICIENT PARSERS

Programming language parsers should be efficient:

- Any context-free grammar can be used to derive a parser that runs in $O(n^3)$ time.
- We want linear time.

Programming language parsers should be efficient:

- Any context-free grammar can be used to derive a parser that runs in $O(n^3)$ time.
- We want linear time.

Question: What types of grammars admit linear-time parsers?

Programming language parsers should be efficient:

- Any context-free grammar can be used to derive a parser that runs in $O(n^3)$ time.
- We want linear time.

Question: What types of grammars admit linear-time parsers?

Most common parser types:

- Recursive-descent parser
- Shift-reduce parser

RECURSIVE-DESCENT PARSING

- Start with the start symbol ($\sigma = S$)

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
  - If the first symbol in σ is a non-terminal X,

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
  - If the first symbol in σ is a non-terminal X,

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
    - If x is not the first symbol in τ, then the parse fails.
  - If the first symbol in σ is a non-terminal X,

RECURSIVE-DESCENT PARSING

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
    - If x is not the first symbol in τ, then the parse fails.
  - If the first symbol in σ is a non-terminal X, then choose a production $X \to \rho$ and replace X with ρ in σ.

RECURSIVE-DESCENT PARSING

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
    - If x is not the first symbol in τ, then the parse fails.
  - If the first symbol in σ is a non-terminal X, then choose a production $X \rightarrow \rho$ and replace X with ρ in σ.
- If $\sigma = \epsilon$ and $\tau = \epsilon$, the parse succeeds; otherwise, it fails.

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
    - If x is not the first symbol in τ, then the parse fails.
  - If the first symbol in σ is a non-terminal X, then choose a production $X \rightarrow \rho$ and replace X with ρ in σ.
- If $\sigma = \epsilon$ and $\tau = \epsilon$, the parse succeeds; otherwise, it fails.

This is also called LL-parsing because it consumes the input Left-to-right and produces a Leftmost derivation.

- Start with the start symbol ($\sigma = S$)
- While the current sentential form σ and the input text τ are not empty:
  - If the first symbol in σ is a terminal x:
    - If x is also the first symbol in τ, then "consume" x, that is, remove it from both σ and τ.
    - If x is not the first symbol in τ, then the parse fails.
  - If the first symbol in σ is a non-terminal X, then choose a production $X \rightarrow \rho$ and replace X with ρ in σ.
- If $\sigma = \epsilon$ and $\tau = \epsilon$, the parse succeeds; otherwise, it fails.

This is also called LL-parsing because it consumes the input Left-to-right and produces a Leftmost derivation.

It is one form of top-down parsing because we start with the start symbol S and construct the parse tree top-down.

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree

S_1

Input string	Stack
+ * + 2 3 4 5	S_1

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree

S_1

Input string	Stack
+ * + 2 3 4 5	S_1

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree

S_1

Input string	Stack
+ * + 2 3 4 5	+ S_2 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

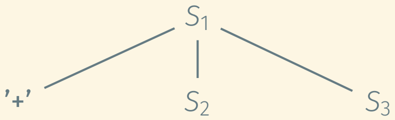
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ * + 2 3 4 5	+ S_2 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

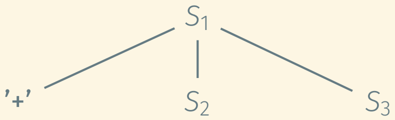
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ * + 2 3 4 5	+ S_2 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

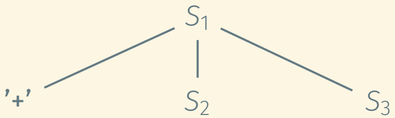
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
* + 2 3 4 5	$S_2\ S_3$

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

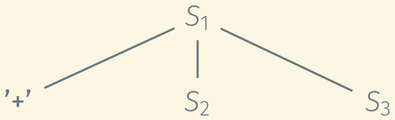
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
* + 2 3 4 5	$S_2\ S_3$

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

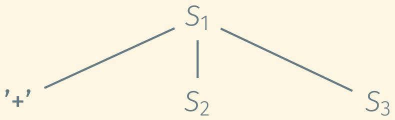
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
* + 2 3 4 5	* S_4 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

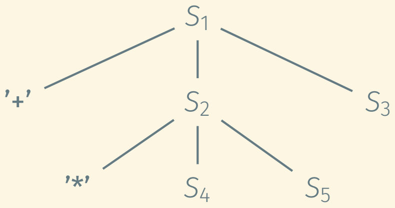
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
* + 2 3 4 5	* S_4 S_5 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ **'+'** $S\ S$

$S \rightarrow$ **'-'** $S\ S$

$S \rightarrow$ **'*'** $S\ S$

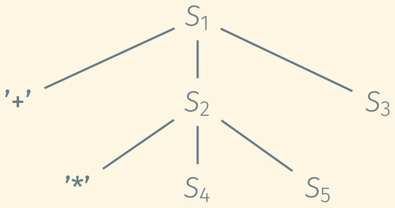
$S \rightarrow$ **'/'** $S\ S$

$S \rightarrow$ **'neg'** S

$S \rightarrow$ **int**

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
* + 2 3 4 5	* S_4 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

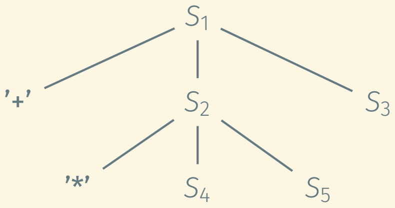
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ 2 3 4 5	$S_4\ S_5\ S_3$

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

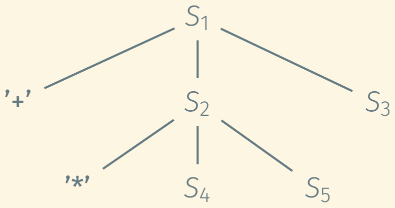
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ 2 3 4 5	$S_4\ S_5\ S_3$

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

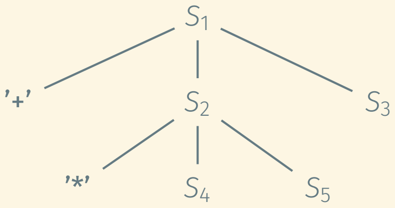
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ 2 3 4 5	+ S_6 S_7 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

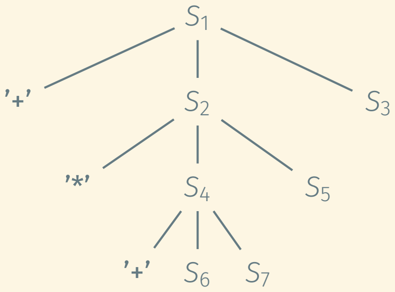
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
+ 2 3 4 5	+ S_6 S_7 S_5 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$$S \rightarrow \text{'+'}\ S\ S$$

$$S \rightarrow \text{'-'}\ S\ S$$

$$S \rightarrow \text{'*'}\ S\ S$$

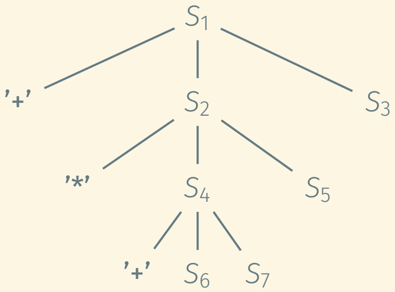
$$S \rightarrow \text{'/'}\ S\ S$$

$$S \rightarrow \text{'neg'}\ S$$

$$S \rightarrow \text{int}$$

$(2 + 3) * 4 + 5 \Rightarrow + * + 2\ 3\ 4\ 5$

Parse tree



Input string	Stack
+ 2 3 4 5	+ S_6 S_7 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

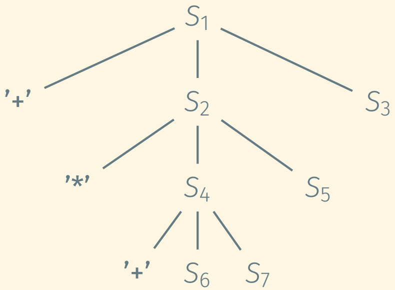
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
2 3 4 5	S_6 S_7 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ **'+'** $S\ S$

$S \rightarrow$ **'-'** $S\ S$

$S \rightarrow$ **'*'** $S\ S$

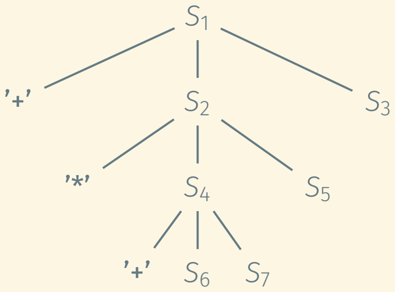
$S \rightarrow$ **'/'** $S\ S$

$S \rightarrow$ **'neg'** S

$S \rightarrow$ **int**

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
2 3 4 5	$S_6\ S_7\ S_5\ S_3$

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

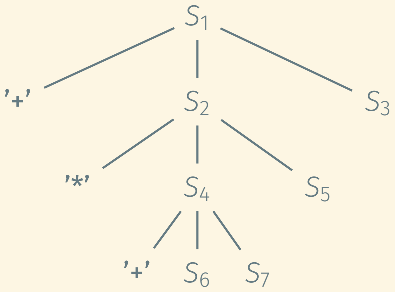
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
2 3 4 5	int S_7 S_5 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

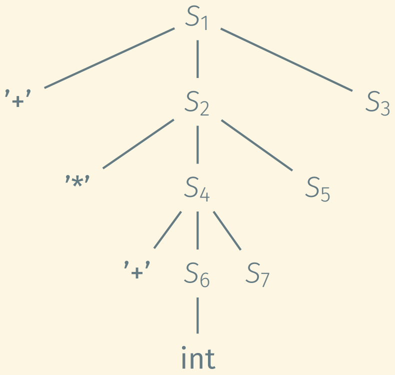
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
2 3 4 5	int S_7 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

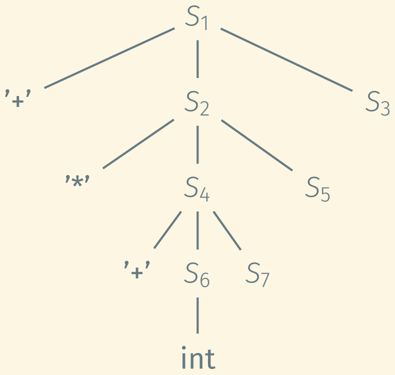
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
2 3 4 5	int S_7 S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

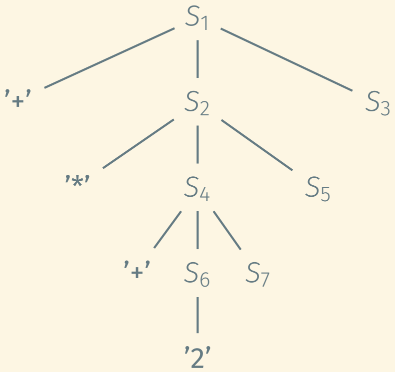
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
3 4 5	S_7 S_5 S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ **'+'** $S\ S$

$S \rightarrow$ **'-'** $S\ S$

$S \rightarrow$ **'*'** $S\ S$

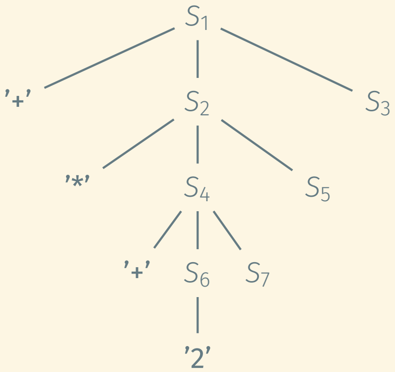
$S \rightarrow$ **'/'** $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
3 4 5	$S_7\ S_5\ S_3$

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

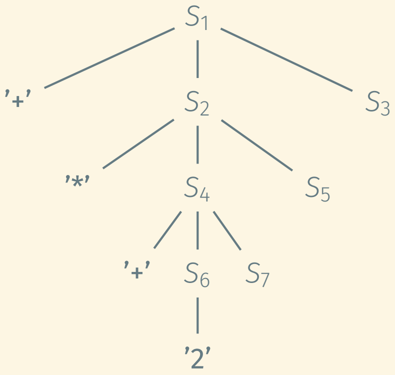
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
3 4 5	int S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ **'+'** $S\ S$

$S \rightarrow$ **'-'** $S\ S$

$S \rightarrow$ **'*'** $S\ S$

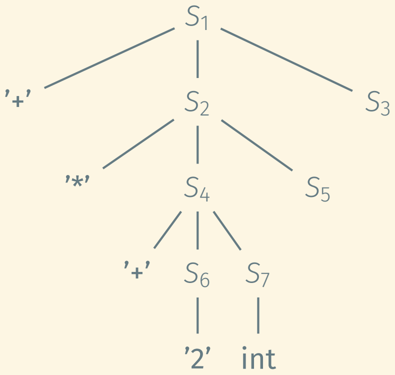
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 ⇒ + * + 2 3 4 5

Parse tree



Input string	Stack
3 4 5	int S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

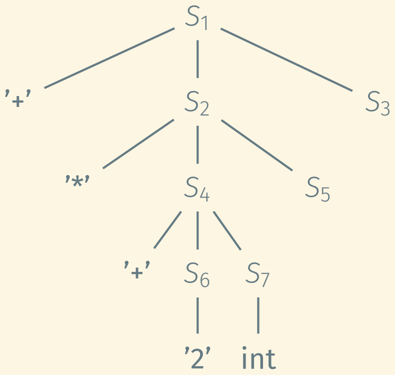
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
3 4 5	int S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \to$ '+' S S

$S \to$ '-' S S

$S \to$ '*' S S

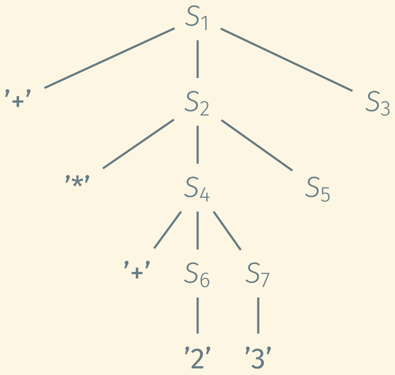
$S \to$ '/' S S

$S \to$ 'neg' S

$S \to$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
4 5	S_5 S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

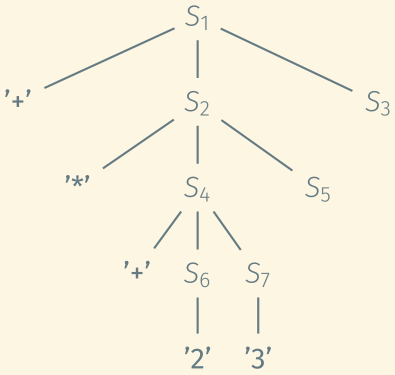
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
4 5	$S_5\ S_3$

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

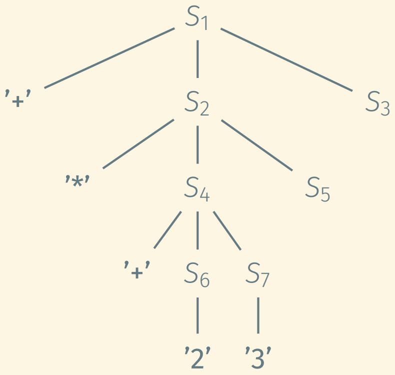
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
4 5	int S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

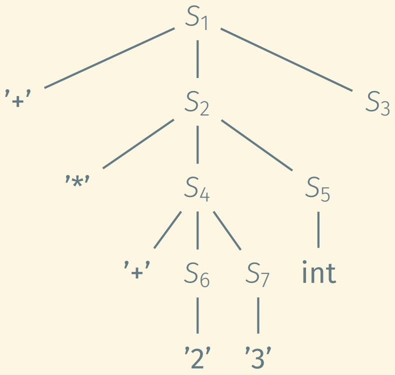
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
4 5	int S_3

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \to$ '+' S S

$S \to$ '-' S S

$S \to$ '*' S S

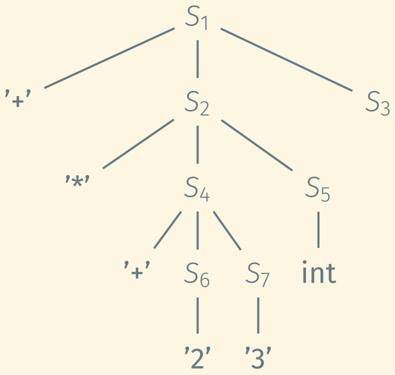
$S \to$ '/' S S

$S \to$ 'neg' S

$S \to$ int

$(2 + 3) * 4 + 5 \Rightarrow + * + 2\ 3\ 4\ 5$

Parse tree



Input string	Stack
4 5	int S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

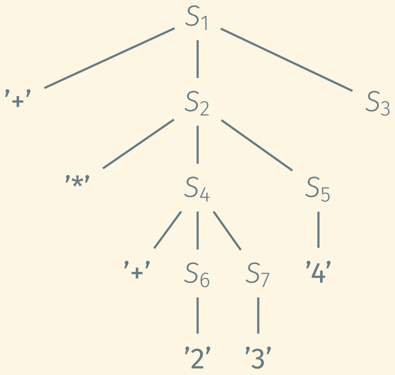
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
5	S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

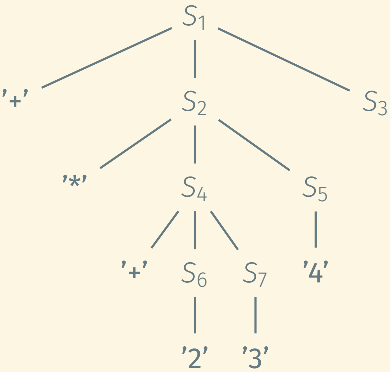
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
5	S_3

An S-grammar for arithmetic expressions in Polish notation:

$S \to$ '+' $S\ S$

$S \to$ '-' $S\ S$

$S \to$ '*' $S\ S$

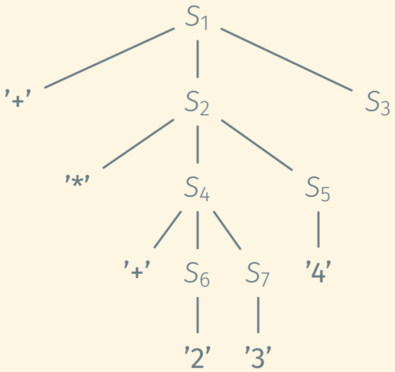
$S \to$ '/' $S\ S$

$S \to$ 'neg' S

$S \to$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
5	int

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' S S

$S \rightarrow$ '-' S S

$S \rightarrow$ '*' S S

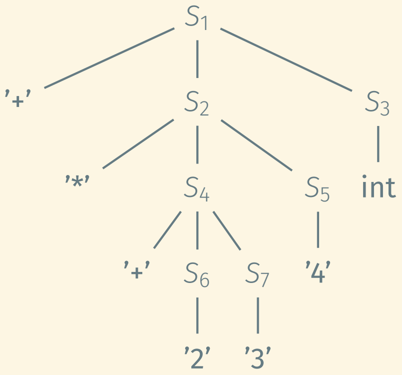
$S \rightarrow$ '/' S S

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
5	int

RECURSIVE-DESCENT PARSING: EXAMPLE

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

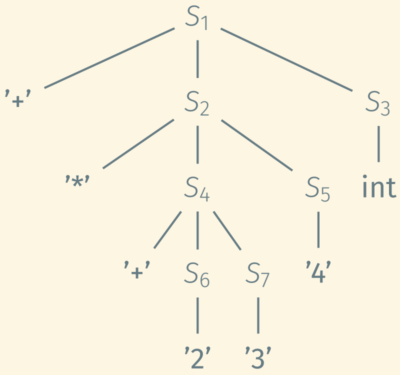
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string	Stack
5	int

An S-grammar for arithmetic expressions in Polish notation:

$S \rightarrow$ '+' $S\ S$

$S \rightarrow$ '-' $S\ S$

$S \rightarrow$ '*' $S\ S$

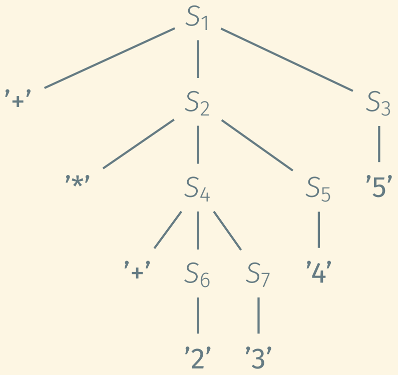
$S \rightarrow$ '/' $S\ S$

$S \rightarrow$ 'neg' S

$S \rightarrow$ int

(2 + 3) * 4 + 5 $\Rightarrow$ + * + 2 3 4 5

Parse tree



Input string Stack

- Begin with an empty stack R

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either

SHIFT-REDUCE PARSING

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either
  - Shift the next input symbol from τ to R (remove it from τ and push it onto R).

SHIFT-REDUCE PARSING

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either
  - Shift the next input symbol from τ to R (remove it from τ and push it onto R).
  - Reduce R by choosing a production $X \to \rho$ such that ρ is the sequence of symbols on the top of R and replace ρ with X in R.

SHIFT-REDUCE PARSING

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either
  - Shift the next input symbol from τ to R (remove it from τ and push it onto R).
  - Reduce R by choosing a production $X \to \rho$ such that ρ is the sequence of symbols on the top of R and replace ρ with X in R.
- The parse succeeds if and only if S is the only symbol on R and $\tau = \epsilon$ at the end.

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either
  - Shift the next input symbol from τ to R (remove it from τ and push it onto R).
  - Reduce R by choosing a production $X \to \rho$ such that ρ is the sequence of symbols on the top of R and replace ρ with X in R.
- The parse succeeds if and only if S is the only symbol on R and $\tau = \epsilon$ at the end.

This is also called LR-parsing because it consumes the input Left-to-right and produce a Rightmost derivation in reverse.

SHIFT-REDUCE PARSING

- Begin with an empty stack R
- While R does not contain the start symbol S and the input $\tau \neq \epsilon$, either
  - Shift the next input symbol from τ to R (remove it from τ and push it onto R).
  - Reduce R by choosing a production $X \to \rho$ such that ρ is the sequence of symbols on the top of R and replace ρ with X in R.
- The parse succeeds if and only if S is the only symbol on R and $\tau = \epsilon$ at the end.

This is also called LR-parsing because it consumes the input Left-to-right and produce a Rightmost derivation in reverse.

It is a form of bottom-up parsing because it produces the parse tree from the leaves up.

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **’+’**

$S \rightarrow S\ S$ **’-’**

$S \rightarrow S\ S$ **’*’**

$S \rightarrow S\ S$ ’/’

$S \rightarrow S$ ’neg’

$S \rightarrow$ int

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$$S \rightarrow S\ S\ \text{'+'}$$
$$S \rightarrow S\ S\ \text{'-'}$$
$$S \rightarrow S\ S\ \text{'*'}$$
$$S \rightarrow S\ S\ \text{'/'}$$
$$S \rightarrow S\ \text{'neg'}$$
$$S \rightarrow \text{int}$$

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 ⇒ 2 3 4 + * 5 +

Parse tree

Input string	Stack
2 3 4 + * 5 +	

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

Input string	Stack
3 4 + * 5 +	2

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

'2'

Input string	Stack
3 4 + * 5 +	2

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

'2'

Input string	Stack
3 4 + * 5 +	2

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

S_1

|

'2'

Input string	Stack
3 4 + * 5 +	2

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

Input string	Stack
3 4 + * 5 +	S_1

A grammar for arithmetic expressions in reverse Polish notation:

$S \to S\ S$ '+'
$S \to S\ S$ '-'
$S \to S\ S$ '*'
$S \to S\ S$ '/'
$S \to S$ 'neg'
$S \to$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

S_1
|
'2'

Input string	Stack
3 4 + * 5 +	S_1

A grammar for arithmetic expressions in reverse Polish notation:

$$S \rightarrow S\ S\ \text{'+'}$$
$$S \rightarrow S\ S\ \text{'-'}$$
$$S \rightarrow S\ S\ \text{'*'}$$
$$S \rightarrow S\ S\ \text{'/'}$$
$$S \rightarrow S\ \text{'neg'}$$
$$S \rightarrow \text{int}$$

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

Input string	Stack
4 + * 5 +	S_1 3

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

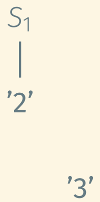
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 ⇒ 2 3 4 + * 5 +

Parse tree



Input string	Stack
4 + * 5 +	S_1 3

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree

S_1
|
'2'

'3'

Input string	Stack
4 + * 5 +	S_1 3

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

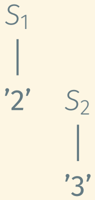
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
4 + * 5 +	S_1 3

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

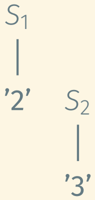
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
4 + * 5 +	$S_1\ S_2$

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

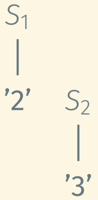
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
4 + * 5 +	$S_1\ S_2$

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

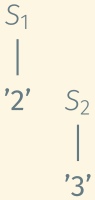
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	S_1 S_2 4

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

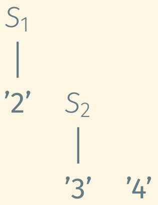
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	$S_1\ S_2$ 4

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ ’+’

$S \rightarrow S\ S$ ’-’

$S \rightarrow S\ S$ ’*’

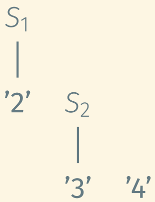
$S \rightarrow S\ S$ ’/’

$S \rightarrow S$ ’neg’

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	$S_1\ S_2$ 4

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

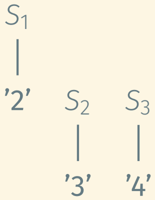
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	S_1 S_2 4

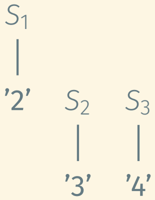
SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'
$S \rightarrow S\ S$ '-'
$S \rightarrow S\ S$ '*'
$S \rightarrow S\ S$ '/'
$S \rightarrow S$ 'neg'
$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	$S_1\ S_2\ S_3$

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

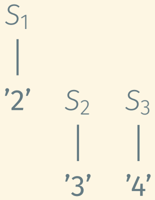
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+ * 5 +	$S_1\ S_2\ S_3$

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

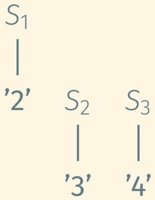
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
* 5 +	$S_1\ S_2\ S_3$ +

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

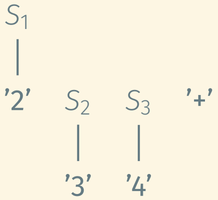
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



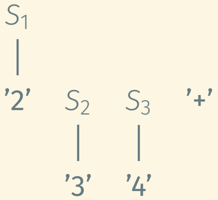
Input string	Stack
* 5 +	S_1 S_2 S_3 +

A grammar for arithmetic expressions in reverse Polish notation:

- $S \rightarrow S\ S$ **'+'**
- $S \rightarrow S\ S$ **'-'**
- $S \rightarrow S\ S$ **'*'**
- $S \rightarrow S\ S$ '/'
- $S \rightarrow S$ 'neg'
- $S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
* 5 +	S_1 S_2 S_3 +

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

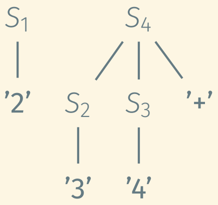
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
* 5 +	$S_1\ S_2\ S_3$ +

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

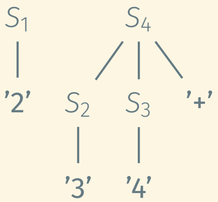
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
* 5 +	$S_1\ S_4$

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

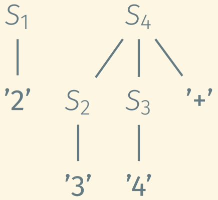
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
* 5 +	$S_1\ S_4$

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

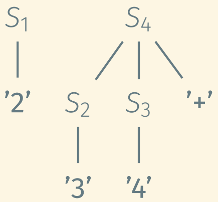
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
5 +	$S_1\ S_4$ *

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

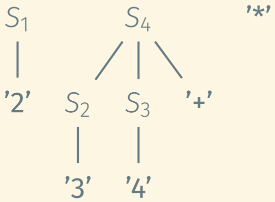
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
5 +	S_1 S_4 *

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

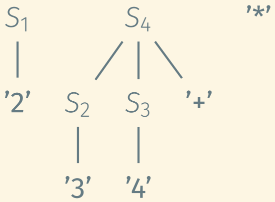
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

$2 * (3 + 4) + 5 \Rightarrow 2\ 3\ 4 + * \ 5 +$

Parse tree



Input string	Stack
5 +	$S_1\ S_4$ *

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

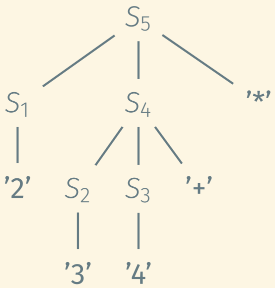
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 ⇒ 2 3 4 + * 5 +

Parse tree



Input string	Stack
5 +	$S_1\ S_4$ *

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

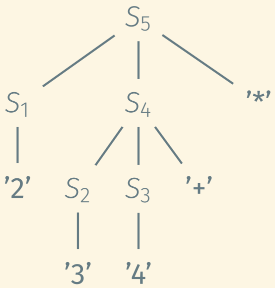
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
5 +	S_5

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \to S\ S$ **'+'**

$S \to S\ S$ **'-'**

$S \to S\ S$ **'*'**

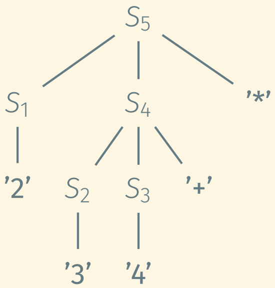
$S \to S\ S$ **'/'**

$S \to S$ **'neg'**

$S \to$ **int**

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
5 +	S_5

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

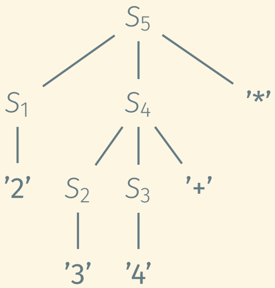
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

$2 * (3 + 4) + 5 \Rightarrow 2\ 3\ 4 + * \ 5\ +$

Parse tree



Input string	Stack
+	S_5 **5**

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

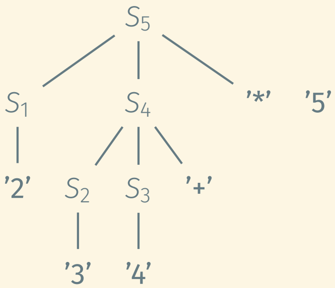
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+	S_5 **5**

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

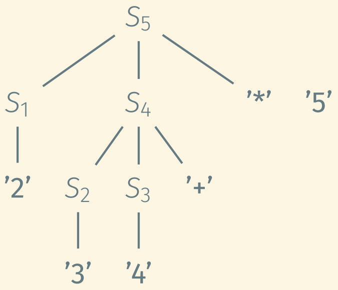
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 ⇒ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+	S_5 5

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

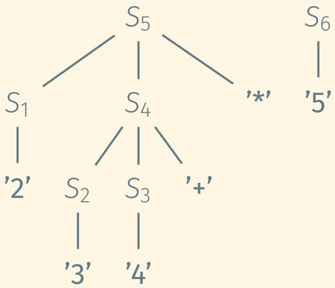
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+	S_5 5

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

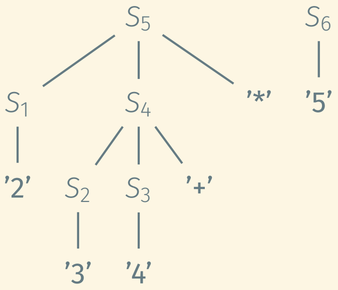
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ **'neg'**

$S \rightarrow$ **int**

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
+	$S_5\ S_6$

SHIFT-REDUCE PARSING: EXAMPLE

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

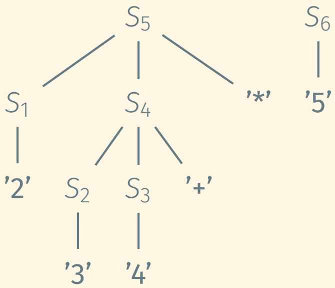
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



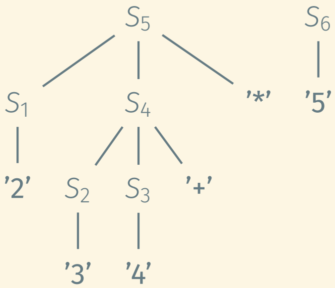
Input string	Stack
+	$S_5\ S_6$

A grammar for arithmetic expressions in reverse Polish notation:

$$S \rightarrow S\ S\ \text{'+'}$$
$$S \rightarrow S\ S\ \text{'-'}$$
$$S \rightarrow S\ S\ \text{'*'}$$
$$S \rightarrow S\ S\ \text{'/'}$$
$$S \rightarrow S\ \text{'neg'}$$
$$S \rightarrow \text{int}$$

$2 * (3 + 4) + 5 \Rightarrow 2\ 3\ 4 + * 5 +$

Parse tree



Input string	Stack
	$S_5\ S_6$ +

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

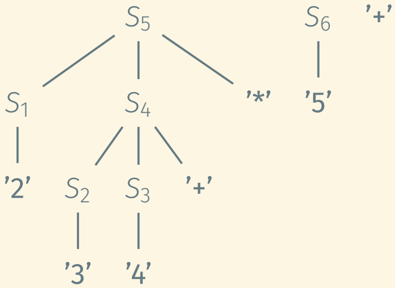
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string

Stack

$S_5\ S_6$ +

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

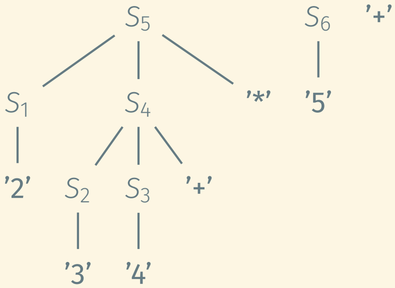
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
	$S_5\ S_6$ +

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ '+'

$S \rightarrow S\ S$ '-'

$S \rightarrow S\ S$ '*'

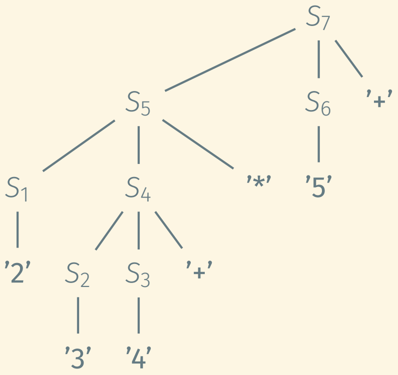
$S \rightarrow S\ S$ '/'

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



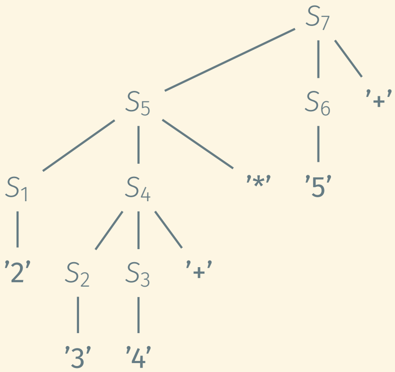
Input string	Stack
	$S_5\ S_6$ +

A grammar for arithmetic expressions in reverse Polish notation:

- $S \rightarrow S\ S$ '+'
- $S \rightarrow S\ S$ '-'
- $S \rightarrow S\ S$ '*'
- $S \rightarrow S\ S$ '/'
- $S \rightarrow S$ 'neg'
- $S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
	S_7

A grammar for arithmetic expressions in reverse Polish notation:

$S \rightarrow S\ S$ **'+'**

$S \rightarrow S\ S$ **'-'**

$S \rightarrow S\ S$ **'*'**

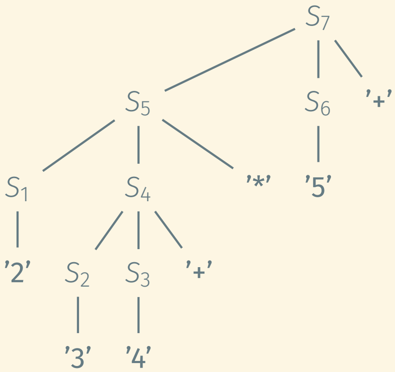
$S \rightarrow S\ S$ **'/'**

$S \rightarrow S$ 'neg'

$S \rightarrow$ int

2 * (3 + 4) + 5 $\Rightarrow$ 2 3 4 + * 5 +

Parse tree



Input string	Stack
	S_7

Key question for top-down parsers: Which production do I use to expand the next non-terminal?

Key question for top-down parsers: Which production do I use to expand the next non-terminal?

Key questions for bottom-up parsers:

- When do I shift, when do I reduce?
- Which production do I use to reduce the top of the stack?

Key question for top-down parsers: Which production do I use to expand the next non-terminal?

Key questions for bottom-up parsers:

- When do I shift, when do I reduce?
- Which production do I use to reduce the top of the stack?

A parser is deterministic if it does not have to guess the answers to these questions and always makes the right choice (assuming the input can be produced using the given grammar).

Key question for top-down parsers: Which production do I use to expand the next non-terminal?

Key questions for bottom-up parsers:

- When do I shift, when do I reduce?
- Which production do I use to reduce the top of the stack?

A parser is deterministic if it does not have to guess the answers to these questions and always makes the right choice (assuming the input can be produced using the given grammar).

Most efficient deterministic parsers use look-ahead to answer these questions: In each step, inspect the next k symbols in the input text and decide what to do based on these symbols.

A grammar is LL(k) if it can be parsed by a recursive-descent parser and a look-ahead of k symbols suffices to decide which production to choose when expanding a non-terminal.

A grammar is LL(k) if it can be parsed by a recursive-descent parser and a look-ahead of k symbols suffices to decide which production to choose when expanding a non-terminal.

A grammar is LR(k) if it can be parsed by a shift-reduce parser and a look-ahead of k symbols suffices to let the parser choose between shift and reduce steps and, for reduce steps, which production to use.

A grammar is LL(k) if it can be parsed by a recursive-descent parser and a look-ahead of k symbols suffices to decide which production to choose when expanding a non-terminal.

A grammar is LR(k) if it can be parsed by a shift-reduce parser and a look-ahead of k symbols suffices to let the parser choose between shift and reduce steps and, for reduce steps, which production to use.

Almost every programming language can be described by an LL(1) or LR(1) grammar.

Advantages of LL parsers:

- Simpler
- Easier to understand
- More space-efficient

CHOOSING BETWEEN LL AND LR PARSERS

Advantages of LL parsers:

- Simpler
- Easier to understand
- More space-efficient

Advantages of LR parsers:

- More general (some languages need less look-ahead than using an LL parser)
- Traditionally faster (modern LL parsers are competitive to LR parsers)

Advantages of LL parsers:

- Simpler
- Easier to understand
- More space-efficient

Advantages of LR parsers:

- More general (some languages need less look-ahead than using an LL parser)
- Traditionally faster (modern LL parsers are competitive to LR parsers)

Variants of LR parsers:

- SLR(1) and LALR(1)
- Less powerful than LR(1) parsers but LALR powerful enough for most programming languages
- Easier to construct than general LR parsers
- More space-efficient than general LR parsers

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

A grammar is right-linear if all productions are of the form $A \to \sigma$ or $A \to \sigma B$, where σ is a (possibly empty) string of terminals.

A grammar is right-linear if all productions are of the form $A \to \sigma$ or $A \to \sigma B$, where σ is a (possibly empty) string of terminals.

A grammar is left-linear if all productions are of the form $A \to \sigma$ or $A \to B\sigma$, where σ is a (possibly empty) string of terminals.

REGULAR GRAMMARS

A grammar is right-linear if all productions are of the form $A \to \sigma$ or $A \to \sigma B$, where σ is a (possibly empty) string of terminals.

A grammar is left-linear if all productions are of the form $A \to \sigma$ or $A \to B\sigma$, where σ is a (possibly empty) string of terminals.

A grammar is regular if it is right-linear or left-linear.

A grammar is right-linear if all productions are of the form $A \to \sigma$ or $A \to \sigma B$, where σ is a (possibly empty) string of terminals.

A grammar is left-linear if all productions are of the form $A \to \sigma$ or $A \to B\sigma$, where σ is a (possibly empty) string of terminals.

A grammar is regular if it is right-linear or left-linear.

The set of languages expressed by regular grammars is exactly the set of regular languages!

REGULAR GRAMMARS

A grammar is right-linear if all productions are of the form $A \to \sigma$ or $A \to \sigma B$, where σ is a (possibly empty) string of terminals.

A grammar is left-linear if all productions are of the form $A \to \sigma$ or $A \to B\sigma$, where σ is a (possibly empty) string of terminals.

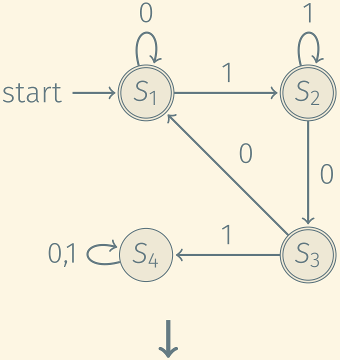
A grammar is regular if it is right-linear or left-linear.

The set of languages expressed by regular grammars is exactly the set of regular languages!

Regular grammars are too weak to express programming languages!

From DFA to right-linear grammar:

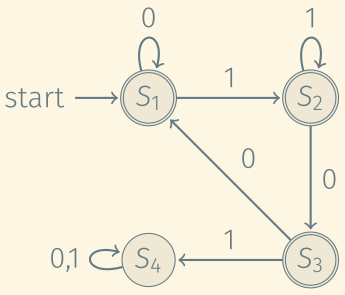
$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$



From DFA to right-linear grammar:

$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$

$$P = P_\delta \cup P_F$$



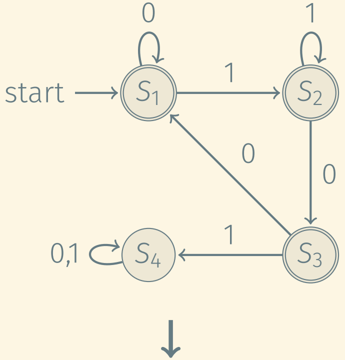
↓

From DFA to right-linear grammar:

$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$

$$P = P_\delta \cup P_F$$

$$P_\delta = \{s \rightarrow x\delta(s, x) \mid s \in S, x \in \Sigma\}$$

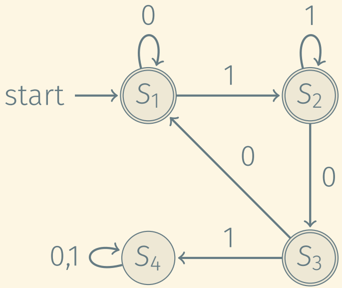


From DFA to right-linear grammar:

$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$

$$P = P_\delta \cup P_F$$

$$P_\delta = \{s \rightarrow x\delta(s, x) \mid s \in S, x \in \Sigma\}$$



↓

$S_1 \rightarrow 0S_1$	$S_1 \rightarrow 1S_2$
$S_2 \rightarrow 0S_3$	$S_2 \rightarrow 1S_2$
$S_3 \rightarrow 0S_1$	$S_3 \rightarrow 1S_4$
$S_4 \rightarrow 0S_4$	$S_4 \rightarrow 1S_4$

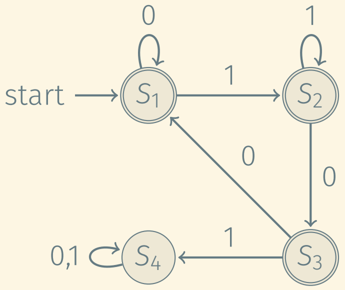
From DFA to right-linear grammar:

$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$

$$P = P_\delta \cup P_F$$

$$P_\delta = \{s \rightarrow x\delta(s, x) \mid s \in S, x \in \Sigma\}$$

$$P_F = \{s \rightarrow \epsilon \mid s \in F\}$$



↓

$S_1 \rightarrow 0S_1$	$S_1 \rightarrow 1S_2$
$S_2 \rightarrow 0S_3$	$S_2 \rightarrow 1S_2$
$S_3 \rightarrow 0S_1$	$S_3 \rightarrow 1S_4$
$S_4 \rightarrow 0S_4$	$S_4 \rightarrow 1S_4$

RIGHT-LINEAR GRAMMARS MODEL REGULAR LANGUAGES (1)

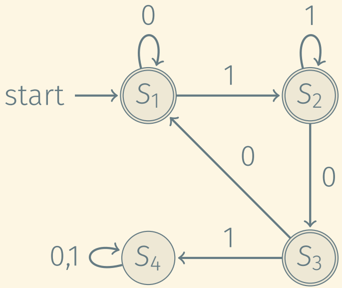
From DFA to right-linear grammar:

$$D = (S, \Sigma, \delta, s_0, F) \rightarrow G = (S, \Sigma, P, s_0)$$

$$P = P_\delta \cup P_F$$

$$P_\delta = \{s \rightarrow x\delta(s, x) \mid s \in S, x \in \Sigma\}$$

$$P_F = \{s \rightarrow \epsilon \mid s \in F\}$$



$\downarrow$

$S_1 \rightarrow 0S_1$	$S_1 \rightarrow 1S_2$	$S_1 \rightarrow \epsilon$
$S_2 \rightarrow 0S_3$	$S_2 \rightarrow 1S_2$	$S_2 \rightarrow \epsilon$
$S_3 \rightarrow 0S_1$	$S_3 \rightarrow 1S_4$	$S_3 \rightarrow \epsilon$
$S_4 \rightarrow 0S_4$	$S_4 \rightarrow 1S_4$	

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \rightarrow G' = (V', \Sigma, P', S)$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$
$$M \rightarrow 101R$$
$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \rightarrow G' = (V', \Sigma, P', S)$$

$$V' = V \cup \bigcup_{p \in P} V_p, \qquad P' = \bigcup_{p \in P} P_p$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$

$$M \rightarrow 101R$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

RIGHT-LINEAR GRAMMARS MODEL REGULAR LANGUAGES (2)

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \to G' = (V', \Sigma, P', S)$$

$$V' = V \cup \bigcup_{p \in P} V_p, \qquad P' = \bigcup_{p \in P} P_p$$

For $p = (X \to [x][Y]) \in P$,

$$V_p = \{X\}, \qquad P_p = p.$$

$$L \to 0L \qquad L \to 1L \qquad L \to M$$

$$M \to 101R$$

$$R \to 0R \qquad R \to 1R \qquad R \to \epsilon$$

$$\downarrow$$

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \rightarrow G' = (V', \Sigma, P', S)$$

$$V' = V \cup \bigcup_{p \in P} V_p, \qquad P' = \bigcup_{p \in P} P_p$$

For $p = (X \rightarrow [x][Y]) \in P$,

$$V_p = \{X\}, \qquad P_p = p.$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$

$$M \rightarrow 101R$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \rightarrow G' = (V', \Sigma, P', S)$$

$$V' = V \cup \bigcup_{p \in P} V_p, \qquad P' = \bigcup_{p \in P} P_p$$

For $p = (X \rightarrow [x][Y]) \in P$,

$$V_p = \{X\}, \qquad P_p = p.$$

For $p = (X \rightarrow x_1 x_2 \ldots x_k[Y]) \in P$, $k \geq 2$,

$$V_p = \{X = X_{p,1}, X_{p,2}, x_{p,3}, \ldots, X_{p,k}\}$$

$$P_p = \{X_{p,i} \rightarrow x_i X_{p,i+1} \mid 1 \leq i \leq k\} \cup \{X_{p,k} \rightarrow [Y]\}$$

$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$

$M \rightarrow 101R$

$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$

↓

$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$

$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$

From right-linear grammar to simplified right-linear grammar:

$$G = (V, \Sigma, P, S) \rightarrow G' = (V', \Sigma, P', S)$$

$$V' = V \cup \bigcup_{p \in P} V_p, \qquad P' = \bigcup_{p \in P} P_p$$

For $p = (X \rightarrow [x][Y]) \in P$,

$$V_p = \{X\}, \qquad P_p = p.$$

For $p = (X \rightarrow x_1 x_2 \ldots x_k[Y]) \in P$, $k \geq 2$,

$$V_p = \{X = X_{p,1}, X_{p,2}, x_{p,3}, \ldots, X_{p,k}\}$$

$$P_p = \{X_{p,i} \rightarrow x_i X_{p,i+1} \mid 1 \leq i \leq k\} \cup \{X_{p,k} \rightarrow [Y]\}$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$
$$M \rightarrow 101R$$
$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$
$$M \rightarrow 1M_2 \qquad M_2 \rightarrow 0M_3 \qquad M_3 \rightarrow 1R$$
$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

From simplified right-linear grammar to NFA:

$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

$$\begin{array}{ccc} L \rightarrow 0L & L \rightarrow 1L & L \rightarrow M_1 \\ M_1 \rightarrow 1M_2 & M_2 \rightarrow 0M_3 & M_3 \rightarrow 1R \\ R \rightarrow 0R & R \rightarrow 1R & R \rightarrow \epsilon \end{array}$$

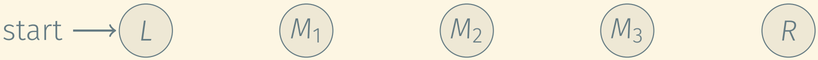
$$\downarrow$$

From simplified right-linear grammar to NFA:

$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

$$\begin{array}{ccc} L \rightarrow 0L & L \rightarrow 1L & L \rightarrow M_1 \\ M_1 \rightarrow 1M_2 & M_2 \rightarrow 0M_3 & M_3 \rightarrow 1R \\ R \rightarrow 0R & R \rightarrow 1R & R \rightarrow \epsilon \end{array}$$

$$\downarrow$$



RIGHT-LINEAR GRAMMARS MODEL REGULAR LANGUAGES (3)

From simplified right-linear grammar to NFA:

$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

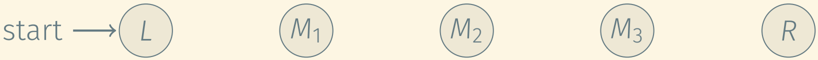
$$F = \{s \in V \mid (s \rightarrow \epsilon) \in P\}$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M_1$$

$$M_1 \rightarrow 1M_2 \qquad M_2 \rightarrow 0M_3 \qquad M_3 \rightarrow 1R$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$



RIGHT-LINEAR GRAMMARS MODEL REGULAR LANGUAGES (3)

From simplified right-linear grammar to NFA:

$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

$$F = \{s \in V \mid (s \rightarrow \epsilon) \in P\}$$

$$\begin{array}{ccc} L \rightarrow 0L & L \rightarrow 1L & L \rightarrow M_1 \\ M_1 \rightarrow 1M_2 & M_2 \rightarrow 0M_3 & M_3 \rightarrow 1R \\ R \rightarrow 0R & R \rightarrow 1R & R \rightarrow \epsilon \end{array}$$

$$\downarrow$$

RIGHT-LINEAR GRAMMARS MODEL REGULAR LANGUAGES (3)

From simplified right-linear grammar to NFA:

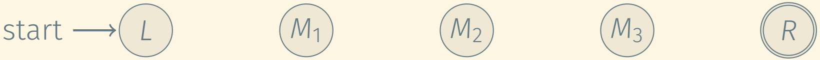
$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

$$F = \{s \in V \mid (s \rightarrow \epsilon) \in P\}$$

$$\delta(s, x) = \{s' \mid (s \rightarrow xs') \in P\}$$

$$\begin{array}{ccc} L \rightarrow 0L & L \rightarrow 1L & L \rightarrow M_1 \\ M_1 \rightarrow 1M_2 & M_2 \rightarrow 0M_3 & M_3 \rightarrow 1R \\ R \rightarrow 0R & R \rightarrow 1R & R \rightarrow \epsilon \end{array}$$

$$\downarrow$$



From simplified right-linear grammar to NFA:

$$G = (V, \Sigma, P, S) \rightarrow N = (V, \Sigma, \delta, S, F)$$

$$F = \{s \in V \mid (s \rightarrow \epsilon) \in P\}$$

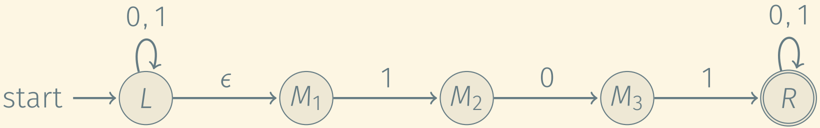
$$\delta(s, x) = \{s' \mid (s \rightarrow xs') \in P\}$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M_1$$

$$M_1 \rightarrow 1M_2 \qquad M_2 \rightarrow 0M_3 \qquad M_3 \rightarrow 1R$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$



Lemma

$\mathcal{L}$ is regular if and only if $\overleftarrow{\mathcal{L}}$ is regular.

$$L \to 0L \qquad L \to 1L \qquad L \to M$$
$$M \to 101R$$
$$R \to 0R \qquad R \to 1R \qquad R \to \epsilon$$

$$\downarrow$$

Lemma

$\mathcal{L}$ is regular if and only if $\overleftarrow{\mathcal{L}}$ is regular.

$$G = (V, \Sigma, P, S) \rightarrow \overleftarrow{G} = (V, \Sigma, \overleftarrow{P}, S)$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$
$$M \rightarrow 101R$$
$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

Lemma

$\mathcal{L}$ is regular if and only if $\overleftarrow{\mathcal{L}}$ is regular.

$$G = (V, \Sigma, P, S) \rightarrow \overleftarrow{G} = (V, \Sigma, \overleftarrow{P}, S)$$

$$\overleftarrow{P} = \{X \rightarrow \overleftarrow{\sigma} \mid (X \rightarrow \sigma) \in P\}$$

$$L \rightarrow 0L \qquad L \rightarrow 1L \qquad L \rightarrow M$$

$$M \rightarrow 101R$$

$$R \rightarrow 0R \qquad R \rightarrow 1R \qquad R \rightarrow \epsilon$$

$$\downarrow$$

Lemma

$\mathcal{L}$ is regular if and only if $\overleftarrow{\mathcal{L}}$ is regular.

$$G = (V, \Sigma, P, S) \to \overleftarrow{G} = (V, \Sigma, \overleftarrow{P}, S)$$

$$\overleftarrow{P} = \{X \to \overleftarrow{\sigma} \mid (X \to \sigma) \in P\}$$

$$L \to 0L \qquad L \to 1L \qquad L \to M$$
$$M \to 101R$$
$$R \to 0R \qquad R \to 1R \qquad R \to \epsilon$$

$$\downarrow$$

$$L \to L0 \qquad L \to L1 \qquad L \to M$$
$$M \to R101$$
$$R \to R0 \qquad R \to R1 \qquad R \to \epsilon$$

S-GRAMMARS

An S-grammar or simple grammar is a special case of an LL(1) grammar:

- Every production starts with a terminal
- The productions for each non-terminal start with different terminals

An S-grammar or simple grammar is a special case of an LL(1) grammar:

- Every production starts with a terminal
- The productions for each non-terminal start with different terminals

A right-linear grammar is easy to transform into an S-grammar.

An S-grammar or simple grammar is a special case of an LL(1) grammar:

- Every production starts with a terminal
- The productions for each non-terminal start with different terminals

A right-linear grammar is easy to transform into an S-grammar.

A non-regular S-grammar:

$$A \rightarrow aAA$$
$$A \rightarrow b$$

An S-grammar or simple grammar is a special case of an LL(1) grammar:

- Every production starts with a terminal
- The productions for each non-terminal start with different terminals

A right-linear grammar is easy to transform into an S-grammar.

A non-regular S-grammar:

$$A \rightarrow aAA$$
$$A \rightarrow b$$

A non-simple context-free grammar:

$$A \rightarrow aAA$$
$$A \rightarrow a$$

EXAMPLE OF AN S-GRAMMAR WE HAVE ALREADY SEEN

An S-grammar for arithmetic expressions in Polish notation:

$$S \rightarrow \text{'+'}\ S\ S$$

$$S \rightarrow \text{'-'}\ S\ S$$

$$S \rightarrow \text{'*'}\ S\ S$$

$$S \rightarrow \text{'/'}\ S\ S$$

$$S \rightarrow \text{'neg'}\ S$$

$$S \rightarrow \text{int}$$

EXAMPLE OF AN S-GRAMMAR WE HAVE ALREADY SEEN

An S-grammar for arithmetic expressions in Polish notation:

$$S \rightarrow \text{'+'}\ S\ S$$
$$S \rightarrow \text{'-'}\ S\ S$$
$$S \rightarrow \text{'*'}\ S\ S$$
$$S \rightarrow \text{'/'}\ S\ S$$
$$S \rightarrow \text{'neg'}\ S$$
$$S \rightarrow \text{int}$$

Recursive-descent parser for S-grammars:

When expanding a leading non-terminal in the current sentential form, use the rule that starts with the next terminal in the input.

PARSING USING LL(1) PARSERS

An LL(1) parser needs to decide which production to apply when the next symbol in the current sentential form is a non-terminal:

Input: $a \ldots$ **Sentential form:** $A \ldots$ **Production:** $A \rightarrow \alpha$?

An LL(1) parser needs to decide which production to apply when the next symbol in the current sentential form is a non-terminal:

Input: $a \ldots$ **Sentential form:** $A \ldots$ **Production:** $A \rightarrow \alpha$?

Two cases:

- $A \Rightarrow \alpha \Rightarrow a\beta$.
- $A \Rightarrow \alpha \Rightarrow \epsilon$ and a derivation of A can be succeeded by a.

An LL(1) parser needs to decide which production to apply when the next symbol in the current sentential form is a non-terminal:

Input: $a \ldots$ **Sentential form:** $A \ldots$ **Production:** $A \rightarrow \alpha$?

Two cases:

- $A \Rightarrow \alpha \Rightarrow a\beta$.
- $A \Rightarrow \alpha \Rightarrow \epsilon$ and a derivation of A can be succeeded by a.

Intuitively (but formally not quite correctly), a terminal a is in the predictor set of the production $A \rightarrow \alpha$ if $A\beta \Rightarrow \alpha\beta \Rightarrow^* a\gamma$ for some $\beta, \gamma \in \Sigma^*$.

An LL(1) parser needs to decide which production to apply when the next symbol in the current sentential form is a non-terminal:

Input: $a\ldots$ **Sentential form:** $A\ldots$ **Production:** $A \to \alpha$?

Two cases:

- $A \Rightarrow \alpha \Rightarrow a\beta$.
- $A \Rightarrow \alpha \Rightarrow \epsilon$ and a derivation of A can be succeeded by a.

Intuitively (but formally not quite correctly), a terminal a is in the predictor set of the production $A \to \alpha$ if $A\beta \Rightarrow \alpha\beta \Rightarrow^* a\gamma$ for some $\beta, \gamma \in \Sigma^*$.

A grammar is LL(1) if the predictor sets of all productions for the same non-terminal are disjoint.

An LL(1) parser needs to decide which production to apply when the next symbol in the current sentential form is a non-terminal:

Input: $a \ldots$ **Sentential form:** $A \ldots$ **Production:** $A \rightarrow \alpha$?

Two cases:

- $A \Rightarrow \alpha \Rightarrow a\beta$.
- $A \Rightarrow \alpha \Rightarrow \epsilon$ and a derivation of A can be succeeded by a.

Intuitively (but formally not quite correctly), a terminal a is in the predictor set of the production $A \rightarrow \alpha$ if $A\beta \Rightarrow \alpha\beta \Rightarrow^* a\gamma$ for some $\beta, \gamma \in \Sigma^*$.

A grammar is LL(1) if the predictor sets of all productions for the same non-terminal are disjoint.

Rule	Predictor set
$S \rightarrow$ + S S	{+}
$S \rightarrow$ - S S	{-}
$S \rightarrow$ * S S	{*}
$S \rightarrow$ / S S	{/}
$S \rightarrow$ neg S	{neg}
$S \rightarrow$ int	{int}

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

Compute three kinds of sets:

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.

- FOLLOW(X) $\subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X$.

COMPUTING PREDICTOR SETS

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.

- FOLLOW(X) $\subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X$.

- PREDICT($A \rightarrow \alpha$) $\subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in$ PREDICT($A \rightarrow \alpha$) if
  - $a \in$ FIRST(α) $\setminus \{\epsilon\}$ or
  - $\epsilon \in$ FIRST(α) and $a \in$ FOLLOW(A).

Compute three kinds of sets:

- $\text{FIRST}(\sigma) \subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* \epsilon$.

- $\text{FOLLOW}(X) \subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X$.

- $\text{PREDICT}(A \rightarrow \alpha) \subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in \text{PREDICT}(A \rightarrow \alpha)$ if

  - $a \in \text{FIRST}(\alpha) \setminus \{\epsilon\}$ or
  - $\epsilon \in \text{FIRST}(\alpha)$ and $a \in \text{FOLLOW}(A)$.

We compute $\text{FIRST}(X)$ only for $X \in V \cup \Sigma$ and generate $\text{FIRST}(\sigma)$ on the fly for some strings $\sigma \in (V \cup \Sigma)^*$ as needed.

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.

We compute FIRST(X) only for $X \in V \cup \Sigma$ and generate FIRST(σ) on the fly for some strings $\sigma \in (V \cup \Sigma)^*$ as needed.

- FOLLOW(X) $\subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X$.

- PREDICT($A \rightarrow \alpha$) $\subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in$ PREDICT($A \rightarrow \alpha$) if
  - $a \in$ FIRST(α) $\setminus \{\epsilon\}$ or
  - $\epsilon \in$ FIRST(α) and $a \in$ FOLLOW(A).

COMPUTING FIRST

Computing FIRST(X), for $X \in V \cup \Sigma$

- For $a \in \Sigma$, FIRST(a) $= \{a\}$.
- For $X \in V$, FIRST(X) $= \emptyset$.
- Repeat until no set FIRST(X) changes for any $X \in V$:
    - FIRST(X) $=$ FIRST(X) $\cup$ FIRST($Y_1Y_2\ldots Y_k$) for each production $X \to Y_1Y_2\ldots Y_k$.

COMPUTING FIRST

Computing FIRST(X), for $X \in V \cup \boldsymbol{\Sigma}$

- For $a \in \Sigma$, $\text{FIRST}(a) = \{a\}$.
- For $X \in V$, $\text{FIRST}(X) = \emptyset$.
- Repeat until no set FIRST(X) changes for any $X \in V$:
  - $\text{FIRST}(X) = \text{FIRST}(X) \cup \text{FIRST}(Y_1 Y_2 \ldots Y_k)$ for each production $X \to Y_1 Y_2 \ldots Y_k$.

Computing FIRST($Y_1 Y_2 \ldots Y_k$) on the fly

- $\text{FIRST}(Y_1 Y_2 \ldots Y_K) = \emptyset$.
- For $i = 1, 2, \ldots, k$:
  - $\text{FIRST}(Y_1 Y_2 \ldots Y_k) = \text{FIRST}(Y_1 Y_2 \ldots Y_k) \cup (\text{FIRST}(Y_i) \setminus \{\epsilon\})$
  - IF $\epsilon \notin \text{FIRST}(Y_i)$, then return.
- $\text{FIRST}(Y_1 Y_2 \ldots Y_k) = \text{FIRST}(Y_1 Y_2 \ldots Y_k) \cup \{\epsilon\}$.

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p				
q				
b				
e				
c				
f				
T				
A				
P				
Q				
B				
C				

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \to A\ B$

$A \to P\ Q$

$A \to B\ C$

$P \to p\ P$

$P \to \epsilon$

$Q \to q\ Q$

$Q \to \epsilon$

$B \to b\ B$

$B \to e$

$C \to c\ C$

$C \to f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			
q	$\{q\}$			
b	$\{b\}$			
e	$\{e\}$			
c	$\{c\}$			
f	$\{f\}$			
T	$\emptyset$			
A	$\emptyset$			
P	$\emptyset$			
Q	$\emptyset$			
B	$\emptyset$			
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$			
A	$\emptyset$			
P	$\emptyset$			
Q	$\emptyset$			
B	$\emptyset$			
C	$\emptyset$			

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$			
P	$\emptyset$			
Q	$\emptyset$			
B	$\emptyset$			
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$	$\emptyset$		
P	$\emptyset$			
Q	$\emptyset$			
B	$\emptyset$			
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$	$\emptyset$		
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$			
B	$\emptyset$			
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$	$\emptyset$		
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$	$\{q, \epsilon\}$		
B	$\emptyset$			
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$	$\emptyset$		
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$	$\{q, \epsilon\}$		
B	$\emptyset$	$\{b, e\}$		
C	$\emptyset$			

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$		
A	$\emptyset$	$\emptyset$		
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$	$\{q, \epsilon\}$		
B	$\emptyset$	$\{b, e\}$		
C	$\emptyset$	$\{c, f\}$		

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$	$\emptyset$	
A	$\emptyset$	$\emptyset$		
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$	$\{q, \epsilon\}$		
B	$\emptyset$	$\{b, e\}$		
C	$\emptyset$	$\{c, f\}$		

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$	$\emptyset$	
A	$\emptyset$	$\emptyset$	$\{p, q, \epsilon, b, e\}$	
P	$\emptyset$	$\{p, \epsilon\}$		
Q	$\emptyset$	$\{q, \epsilon\}$		
B	$\emptyset$	$\{b, e\}$		
C	$\emptyset$	$\{c, f\}$		

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$	$\emptyset$	
A	$\emptyset$	$\emptyset$	$\{p, q, \epsilon, b, e\}$	
P	$\emptyset$	$\{p, \epsilon\}$	$\{p, \epsilon\}$	
Q	$\emptyset$	$\{q, \epsilon\}$	$\{q, \epsilon\}$	
B	$\emptyset$	$\{b, e\}$	$\{b, e\}$	
C	$\emptyset$	$\{c, f\}$	$\{c, f\}$	

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$	$\emptyset$	$\{p, q, b, e\}$
A	$\emptyset$	$\emptyset$	$\{p, q, \epsilon, b, e\}$	
P	$\emptyset$	$\{p, \epsilon\}$	$\{p, \epsilon\}$	
Q	$\emptyset$	$\{q, \epsilon\}$	$\{q, \epsilon\}$	
B	$\emptyset$	$\{b, e\}$	$\{b, e\}$	
C	$\emptyset$	$\{c, f\}$	$\{c, f\}$	

COMPUTING FIRST: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)			
	Iteration 1	Iteration 2	Iteration 3	Iteration 4
p	$\{p\}$			$\{p\}$
q	$\{q\}$			$\{q\}$
b	$\{b\}$			$\{b\}$
e	$\{e\}$			$\{e\}$
c	$\{c\}$			$\{c\}$
f	$\{f\}$			$\{f\}$
T	$\emptyset$	$\emptyset$	$\emptyset$	$\{p, q, b, e\}$
A	$\emptyset$	$\emptyset$	$\{p, q, \epsilon, b, e\}$	$\{p, q, \epsilon, b, e\}$
P	$\emptyset$	$\{p, \epsilon\}$	$\{p, \epsilon\}$	$\{p, \epsilon\}$
Q	$\emptyset$	$\{q, \epsilon\}$	$\{q, \epsilon\}$	$\{q, \epsilon\}$
B	$\emptyset$	$\{b, e\}$	$\{b, e\}$	$\{b, e\}$
C	$\emptyset$	$\{c, f\}$	$\{c, f\}$	$\{c, f\}$

COMPUTING PREDICTOR SETS

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.
- FOLLOW(X) $\subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X$.
- PREDICT($A \rightarrow \alpha$) $\subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in$ PREDICT($A \rightarrow \alpha$) if
  - $a \in$ FIRST(α) $\setminus \{\epsilon\}$ or
  - $\epsilon \in$ FIRST(α) and $a \in$ FOLLOW(A).

Compute three kinds of sets:

- FIRST(σ) $\subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in$ FIRST(σ) if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in$ FIRST(σ) if $\sigma \Rightarrow^* \epsilon$.
- FOLLOW(X) $\subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in$ FOLLOW(X) if $S \Rightarrow^* \alpha X$.
- PREDICT($A \rightarrow \alpha$) $\subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in$ PREDICT($A \rightarrow \alpha$) if
  - $a \in$ FIRST(α) $\setminus \{\epsilon\}$ or
  - $\epsilon \in$ FIRST(α) and $a \in$ FOLLOW(A).

COMPUTING FOLLOW

- $\text{FOLLOW}(S) = \{\epsilon\}$.
- $\text{FOLLOW}(X) = \emptyset$ for all $X \in V \setminus \{S\}$.
- Repeat until no set $\text{FOLLOW}(X)$ changes:
  - For each production $A \rightarrow \alpha B \beta$:
    - $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) \setminus \{\epsilon\})$.
    - If $\epsilon \in \text{FIRST}(\beta)$, then $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$.

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X) Iter 1	FOLLOW(X) Iter 2	FOLLOW(X) Iter 3
T			
A			
P			
Q			
B			
C			

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$		
A	$\emptyset$		
P	$\emptyset$		
Q	$\emptyset$		
B	$\emptyset$		
C	$\emptyset$		

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X) Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$		
P	$\emptyset$		
Q	$\emptyset$		
B	$\emptyset$		
C	$\emptyset$		

Is the following grammar LL(1)?

$T \to A\ B$

$A \to P\ Q$

$A \to B\ C$

$P \to p\ P$

$P \to \epsilon$

$Q \to q\ Q$

$Q \to \epsilon$

$B \to b\ B$

$B \to e$

$C \to c\ C$

$C \to f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	
P	$\emptyset$		
Q	$\emptyset$		
B	$\emptyset$		
C	$\emptyset$		

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$		
Q	$\emptyset$		
B	$\emptyset$		
C	$\emptyset$		

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	
Q	$\emptyset$		
B	$\emptyset$		
C	$\emptyset$		

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	
Q	$\emptyset$	$\emptyset$	
B	$\emptyset$		
C	$\emptyset$		

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	
Q	$\emptyset$	$\emptyset$	
B	$\emptyset$	$\{\epsilon, c, f\}$	
C	$\emptyset$		

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \to A\ B$

$A \to P\ Q$

$A \to B\ C$

$P \to p\ P$

$P \to \epsilon$

$Q \to q\ Q$

$Q \to \epsilon$

$B \to b\ B$

$B \to e$

$C \to c\ C$

$C \to f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	
Q	$\emptyset$	$\emptyset$	
B	$\emptyset$	$\{\epsilon, c, f\}$	
C	$\emptyset$	$\emptyset$	

Is the following grammar LL(1)?

$T \to A\ B$

$A \to P\ Q$

$A \to B\ C$

$P \to p\ P$

$P \to \epsilon$

$Q \to q\ Q$

$Q \to \epsilon$

$B \to b\ B$

$B \to e$

$C \to c\ C$

$C \to f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	$\{q, b, e\}$
Q	$\emptyset$	$\emptyset$	
B	$\emptyset$	$\{\epsilon, c, f\}$	
C	$\emptyset$	$\emptyset$	

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	$\{q, b, e\}$
Q	$\emptyset$	$\emptyset$	$\{b, e\}$
B	$\emptyset$	$\{\epsilon, c, f\}$	
C	$\emptyset$	$\emptyset$	

Is the following grammar LL(1)?

$T \rightarrow A\ B$

$A \rightarrow P\ Q$

$A \rightarrow B\ C$

$P \rightarrow p\ P$

$P \rightarrow \epsilon$

$Q \rightarrow q\ Q$

$Q \rightarrow \epsilon$

$B \rightarrow b\ B$

$B \rightarrow e$

$C \rightarrow c\ C$

$C \rightarrow f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	$\{q, b, e\}$
Q	$\emptyset$	$\emptyset$	$\{b, e\}$
B	$\emptyset$	$\{\epsilon, c, f\}$	$\{\epsilon, c, f\}$
C	$\emptyset$	$\emptyset$	

COMPUTING FOLLOW: EXAMPLE

Is the following grammar LL(1)?

$T \to A\ B$

$A \to P\ Q$

$A \to B\ C$

$P \to p\ P$

$P \to \epsilon$

$Q \to q\ Q$

$Q \to \epsilon$

$B \to b\ B$

$B \to e$

$C \to c\ C$

$C \to f$

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)		
	Iter 1	Iter 2	Iter 3
T	$\{\epsilon\}$	$\{\epsilon\}$	$\{\epsilon\}$
A	$\emptyset$	$\{b, e\}$	$\{b, e\}$
P	$\emptyset$	$\{q\}$	$\{q, b, e\}$
Q	$\emptyset$	$\emptyset$	$\{b, e\}$
B	$\emptyset$	$\{\epsilon, c, f\}$	$\{\epsilon, c, f\}$
C	$\emptyset$	$\emptyset$	$\{b, e\}$

Compute three kinds of sets:

- $\text{FIRST}(\sigma) \subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* \epsilon$.

- $\text{FOLLOW}(X) \subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X$.

- $\text{PREDICT}(A \rightarrow \alpha) \subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in \text{PREDICT}(A \rightarrow \alpha)$ if

  - $a \in \text{FIRST}(\alpha) \setminus \{\epsilon\}$ or
  - $\epsilon \in \text{FIRST}(\alpha)$ and $a \in \text{FOLLOW}(A)$.

COMPUTING PREDICTOR SETS

Compute three kinds of sets:

- $\text{FIRST}(\sigma) \subseteq \Sigma \cup \{\epsilon\}$, for all $\sigma \in (V \cup \Sigma)^*$:
  - For $a \in \Sigma$, $a \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* a\beta$.
  - $\epsilon \in \text{FIRST}(\sigma)$ if $\sigma \Rightarrow^* \epsilon$.

- $\text{FOLLOW}(X) \subseteq \Sigma \cup \{\epsilon\}$, for all $X \in V$:
  - For $a \in \Sigma$, $a \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X a \beta$.
  - $\epsilon \in \text{FOLLOW}(X)$ if $S \Rightarrow^* \alpha X$.

- $\text{PREDICT}(A \rightarrow \alpha) \subseteq \Sigma \cup \{\epsilon\}$, for all $(A \rightarrow \alpha) \in P$:

  $a \in \text{PREDICT}(A \rightarrow \alpha)$ if

  - $a \in \text{FIRST}(\alpha) \setminus \{\epsilon\}$ or
  - $\epsilon \in \text{FIRST}(\alpha)$ and $a \in \text{FOLLOW}(A)$.

- For every production $A \rightarrow \alpha$:
  - $\text{PREDICT}(A \rightarrow \alpha) = \text{FIRST}(\alpha) \setminus \{\epsilon\}$.
  - If $\epsilon \in \text{FIRST}(\alpha)$, then $\text{PREDICT}(A \rightarrow \alpha) = \text{PREDICT}(A \rightarrow \alpha) \cup \text{FOLLOW}(A)$.

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	
$A \rightarrow P\ Q$	
$A \rightarrow B\ C$	
$P \rightarrow p\ P$	
$P \rightarrow \epsilon$	
$Q \rightarrow q\ Q$	
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	
$A \rightarrow B\ C$	
$P \rightarrow p\ P$	
$P \rightarrow \epsilon$	
$Q \rightarrow q\ Q$	
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	
$P \rightarrow p\ P$	
$P \rightarrow \epsilon$	
$Q \rightarrow q\ Q$	
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \to A\ B$	$\{p, q, b, e\}$
$A \to P\ Q$	$\{p, q, b, e\}$
$A \to B\ C$	$\{b, e\}$
$P \to p\ P$	
$P \to \epsilon$	
$Q \to q\ Q$	
$Q \to \epsilon$	
$B \to b\ B$	
$B \to e$	
$C \to c\ C$	
$C \to f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	
$Q \rightarrow q\ Q$	
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	$\{q, b, e\}$
$Q \rightarrow q\ Q$	
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	$\{q, b, e\}$
$Q \rightarrow q\ Q$	$\{q\}$
$Q \rightarrow \epsilon$	
$B \rightarrow b\ B$	
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \to A\ B$	$\{p, q, b, e\}$
$A \to P\ Q$	$\{p, q, b, e\}$
$A \to B\ C$	$\{b, e\}$
$P \to p\ P$	$\{p\}$
$P \to \epsilon$	$\{q, b, e\}$
$Q \to q\ Q$	$\{q\}$
$Q \to \epsilon$	$\{b, e\}$
$B \to b\ B$	
$B \to e$	
$C \to c\ C$	
$C \to f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	$\{q, b, e\}$
$Q \rightarrow q\ Q$	$\{q\}$
$Q \rightarrow \epsilon$	$\{b, e\}$
$B \rightarrow b\ B$	$\{b\}$
$B \rightarrow e$	
$C \rightarrow c\ C$	
$C \rightarrow f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \to A\ B$	$\{p, q, b, e\}$
$A \to P\ Q$	$\{p, q, b, e\}$
$A \to B\ C$	$\{b, e\}$
$P \to p\ P$	$\{p\}$
$P \to \epsilon$	$\{q, b, e\}$
$Q \to q\ Q$	$\{q\}$
$Q \to \epsilon$	$\{b, e\}$
$B \to b\ B$	$\{b\}$
$B \to e$	$\{e\}$
$C \to c\ C$	
$C \to f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \to A\ B$	$\{p, q, b, e\}$
$A \to P\ Q$	$\{p, q, b, e\}$
$A \to B\ C$	$\{b, e\}$
$P \to p\ P$	$\{p\}$
$P \to \epsilon$	$\{q, b, e\}$
$Q \to q\ Q$	$\{q\}$
$Q \to \epsilon$	$\{b, e\}$
$B \to b\ B$	$\{b\}$
$B \to e$	$\{e\}$
$C \to c\ C$	$\{c\}$
$C \to f$	

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	$\{q, b, e\}$
$Q \rightarrow q\ Q$	$\{q\}$
$Q \rightarrow \epsilon$	$\{b, e\}$
$B \rightarrow b\ B$	$\{b\}$
$B \rightarrow e$	$\{e\}$
$C \rightarrow c\ C$	$\{c\}$
$C \rightarrow f$	$\{f\}$

COMPUTING PREDICT: EXAMPLE

X	FIRST(X)
p	$\{p\}$
q	$\{q\}$
b	$\{b\}$
e	$\{e\}$
c	$\{c\}$
f	$\{f\}$
T	$\{p, q, b, e\}$
A	$\{p, q, \epsilon, b, e\}$
P	$\{p, \epsilon\}$
Q	$\{q, \epsilon\}$
B	$\{b, e\}$
C	$\{c, f\}$

X	FOLLOW(X)
T	$\{\epsilon\}$
A	$\{b, e\}$
P	$\{q, b, e\}$
Q	$\{b, e\}$
B	$\{\epsilon, c, f\}$
C	$\{b, e\}$

Rule R	PREDICT(R)
$T \rightarrow A\ B$	$\{p, q, b, e\}$
$A \rightarrow P\ Q$	$\{p, q, b, e\}$
$A \rightarrow B\ C$	$\{b, e\}$
$P \rightarrow p\ P$	$\{p\}$
$P \rightarrow \epsilon$	$\{q, b, e\}$
$Q \rightarrow q\ Q$	$\{q\}$
$Q \rightarrow \epsilon$	$\{b, e\}$
$B \rightarrow b\ B$	$\{b\}$
$B \rightarrow e$	$\{e\}$
$C \rightarrow c\ C$	$\{c\}$
$C \rightarrow f$	$\{f\}$

This grammar is not LL(1)!

There exist context-free languages that do not have LL(1) grammars.

SOME FACTS ABOUT LL(1) LANGUAGES

There exist context-free languages that do not have LL(1) grammars.

There is no known algorithm to determine whether a language is LL(1) (but there is an algorithm to decide whether a grammar is LL(1)).

SOME FACTS ABOUT LL(1) LANGUAGES

There exist context-free languages that do not have LL(1) grammars.

There is no known algorithm to determine whether a language is LL(1) (but there is an algorithm to decide whether a grammar is LL(1)).

The “obvious” grammar for most programming languages is usually not LL(1).

SOME FACTS ABOUT LL(1) LANGUAGES

There exist context-free languages that do not have LL(1) grammars.

There is no known algorithm to determine whether a language is LL(1) (but there is an algorithm to decide whether a grammar is LL(1)).

The "obvious" grammar for most programming languages is usually not LL(1).

In many situations, a non-LL(1) grammar can be transformed into an LL(1) grammar for the same language.

CONVERTING A GRAMMAR TO LL(1)

Two common reasons why a grammar is not LL(1) are "left-recursion" and "common prefixes", both of which can be eliminated by modifying the grammar.

Two common reasons why a grammar is not LL(1) are “left-recursion” and “common prefixes”, both of which can be eliminated by modifying the grammar.

Left recursion:

$$A \rightarrow \alpha | A\beta$$

CONVERTING A GRAMMAR TO LL(1)

Two common reasons why a grammar is not LL(1) are “left-recursion” and “common prefixes”, both of which can be eliminated by modifying the grammar.

Left recursion:

$$A \rightarrow \alpha | A\beta$$

Common prefix:

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

Two common reasons why a grammar is not LL(1) are "left-recursion" and "common prefixes", both of which can be eliminated by modifying the grammar.

Left recursion:

$$A \to \alpha | A\beta$$

Common prefix:

$$A \to \alpha\beta | \alpha\gamma$$

Example of a common prefix:

$$Expr \to Term$$

$$Expr \to Term + Expr$$

REMOVING LEFT-RECURSION

Left-recursion can be replaced with right-recursion:

$$A \rightarrow \alpha | A\beta$$

$$\Downarrow$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \epsilon | \beta A'$$

Caveat:

- Left-recursion is often used intentionally to capture the structure of the language (e.g., associativity of operators in arithmetic expressions).
- The above conversion discards this information.

REMOVING COMMON PREFIXES

Common prefixes can be removed using left-factoring:

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

$$\Downarrow$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta|\gamma$$

CONVERTING A GRAMMAR TO LL(1): EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\$$	$\{n, (\}$
$E \rightarrow E\ A\ T$	$\{n, (\}$
$E \rightarrow T$	$\{n, (\}$
$T \rightarrow T\ M\ F$	$\{n, (\}$
$T \rightarrow F$	$\{n, (\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (\ E\ )$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

CONVERTING A GRAMMAR TO LL(1): EXAMPLE

Rule R	PREDICT(R)
$S \to E\$$	$\{n, (\}$
$E \to E\ A\ T$	$\{n, (\}$
$E \to T$	$\{n, (\}$
$T \to T\ M\ F$	$\{n, (\}$
$T \to F$	$\{n, (\}$
$F \to n$	$\{n\}$
$F \to (\ E\ )$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

Rule R	PREDICT(R)
$S \to E\$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

LL(1) languages can be parsed using efficient, easy-to-implement parsers.

PARSING LL(1) LANGUAGES

LL(1) languages can be parsed using efficient, easy-to-implement parsers.

Two approaches:

- Recursive-descent parser
- Deterministic push-down automaton

LL(1) languages can be parsed using efficient, easy-to-implement parsers.

Two approaches:

- Recursive-descent parser
- Deterministic push-down automaton

Recursive-descent parser:

For each non-terminal X, write a produce parse$X$:

- Choose production $X \rightarrow Y_1 Y_2 \ldots Y_k$ whose predictor set contains next token.
- For $i = 1, 2, \ldots, k$:
  - If Y_i is a terminal, match Y_1 with next input token.
  - If Y_i is a non-terminal, call parseY_i.

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\,T\,E'$	$\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\,F\,T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseT():
  if next token is n or (:
    parseF()
    parseT'()


def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseT():
  if next token is n or (:
    parseF()
    parseT'()


def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

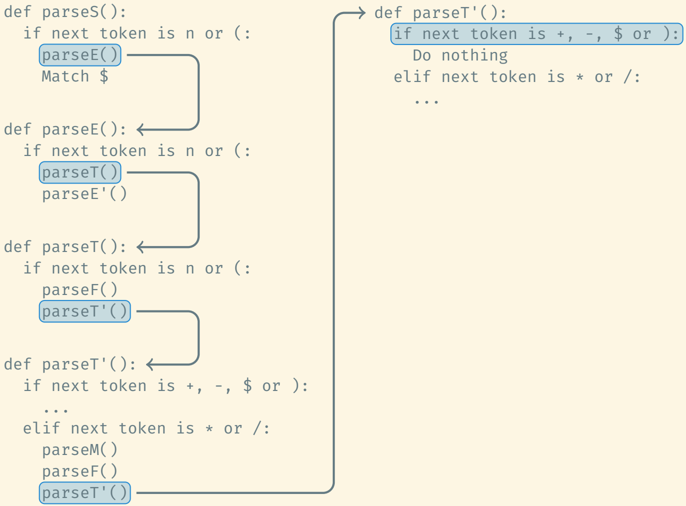
```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    Do nothing
  elif next token is * or /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $



RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    Do nothing
  elif next token is * or /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

```
def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseA():
  if next token is +:
    ...
  elif next token is -:
    Match -
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseA():
  if next token is +:
    ...
  elif next token is -:
    Match -
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseA():
  if next token is +:
    ...
  elif next token is -:
    Match -
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\,T\,E'$	$\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\,F\,T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseA():
  if next token is +:
    ...
  elif next token is -:
    Match -
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseA():
  if next token is +:
    ...
  elif next token is -:
    Match -
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()


def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\,T\,E'$	$\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\,F\,T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

2 * 40 - 18 * 3 $

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 [*] 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseM():
  if next token is *:
    Match *
  elif next token is /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseF():
  if next token is n:
    Match n
  elif next token is (:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

```
def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

```
def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

```
def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

```
def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

```
def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

```
def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    Do nothing
  elif next token is * or /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    Do nothing
  elif next token is * or /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()

def parseT'():
  if next token is +, -, $ or ):
    Do nothing
  elif next token is * or /:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\,T\,E'$	$\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\,F\,T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

```
def parseT'():
  if next token is +, -, $ or ):
    ...
  elif next token is * or /:
    parseM()
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseT():
  if next token is n or (:
    parseF()
    parseT'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    Do nothing
  elif next token is + or -:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    Do nothing
  elif next token is + or -:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()

def parseE'():
  if next token is $ or ):
    Do nothing
  elif next token is + or -:
    ...
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $


def parseE():
  if next token is n or (:
    parseT()
    parseE'()


def parseE'():
  if next token is $ or ):
    ...
  elif next token is + or -:
    parseA()
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $

def parseE():
  if next token is n or (:
    parseT()
    parseE'()
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

`2 * 40 - 18 * 3 $`

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

```
def parseS():
  if next token is n or (:
    parseE()
    Match $
```

RECURSIVE DESCENT PARSING: EXAMPLE

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

```
2 * 40 - 18 * 3 $
```

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

ROAD MAP

- **Parsing:** Transform (tokenized) program text into parse tree
- **Modelling programming languages:** Context-free grammars and languages
- **Capturing the syntactic structure of a program:** Parse trees

- Types of parsers and types of grammars they can parse
- Grammars that describe programming languages and can be parsed efficiently

- Construction of an LL(1) grammar
- Parsing LL(1) languages
- Push-down automata

We proved that a language can be parsed by a finite automaton if and only if it is regular.

PUSH-DOWN AUTOMATON: MOTIVATION

We proved that a language can be parsed by a finite automaton if and only if it is regular.

We also proved that some context-free languages, including most programming languages, are not regular.

We proved that a language can be parsed by a finite automaton if and only if it is regular.

We also proved that some context-free languages, including most programming languages, are not regular.

Thus, finite automata are not expressive enough to parse context-free languages.

PUSH-DOWN AUTOMATON: MOTIVATION

We proved that a language can be parsed by a finite automaton if and only if it is regular.

We also proved that some context-free languages, including most programming languages, are not regular.

Thus, finite automata are not expressive enough to parse context-free languages.

A push-down automaton (PDA) is an NFA with a stack.

PUSH-DOWN AUTOMATON: MOTIVATION

We proved that a language can be parsed by a finite automaton if and only if it is regular.

We also proved that some context-free languages, including most programming languages, are not regular.

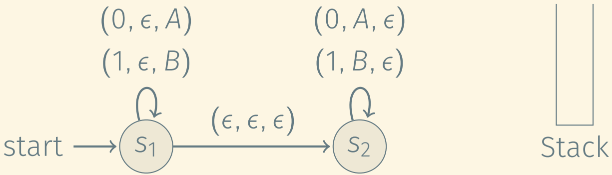
Thus, finite automata are not expressive enough to parse context-free languages.

A push-down automaton (PDA) is an NFA with a stack.

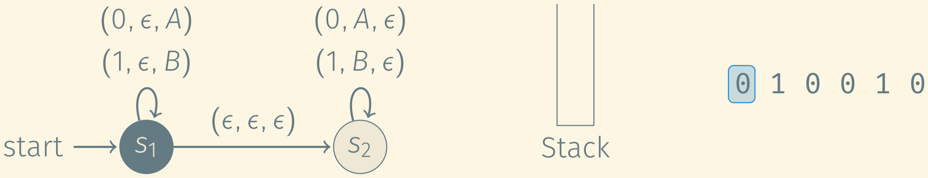
Any context-free language can be parsed by a PDA.

PUSH-DOWN AUTOMATON: EXAMPLE

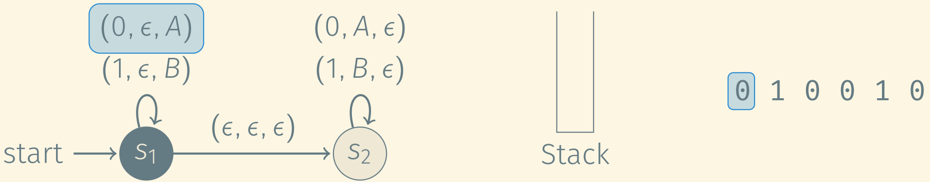
A PDA for the language $\{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

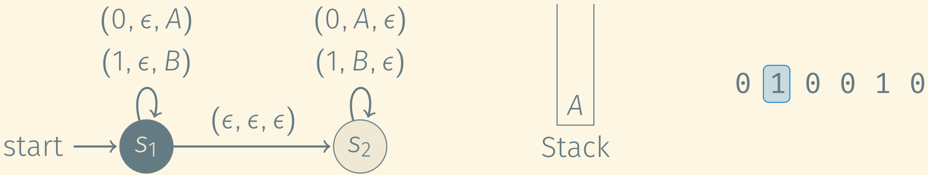


A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

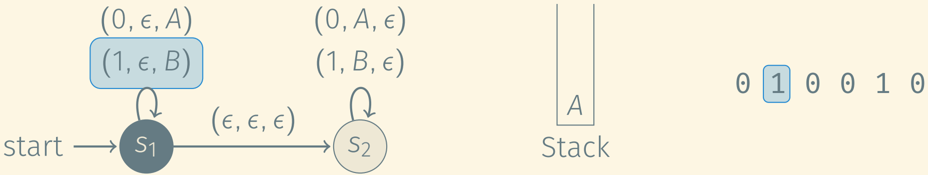


PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

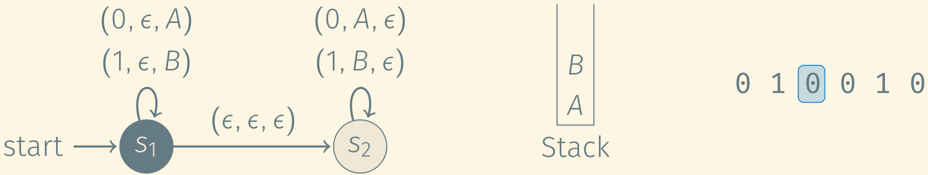


A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



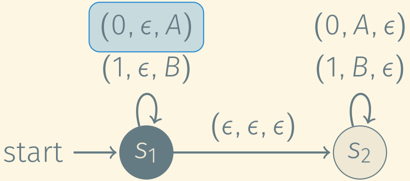
PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



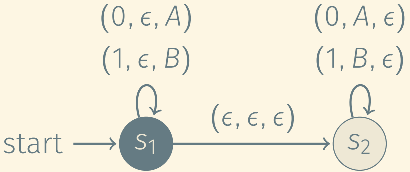
PUSH-DOWN AUTOMATON: EXAMPLE

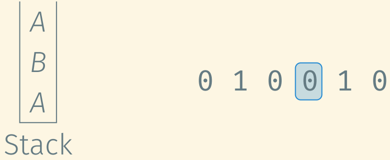
A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



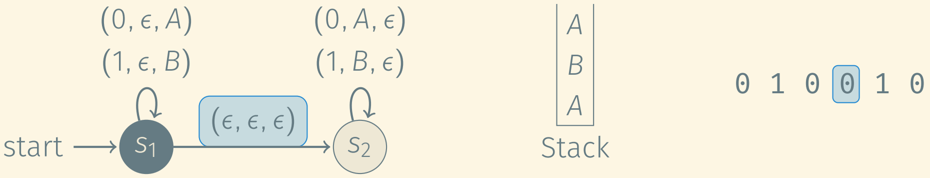
0 1 0 0 1 0

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



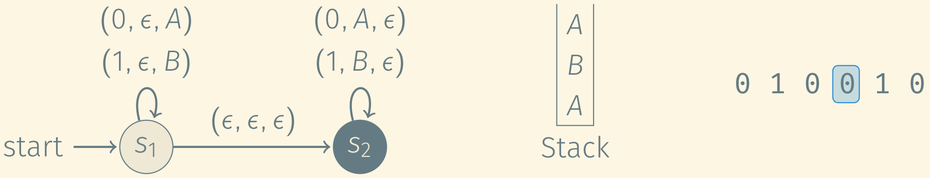


A PDA for the language $\{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



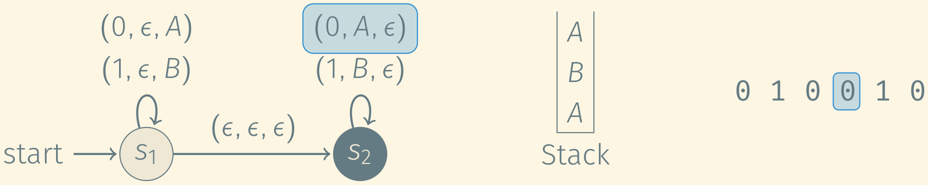
PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

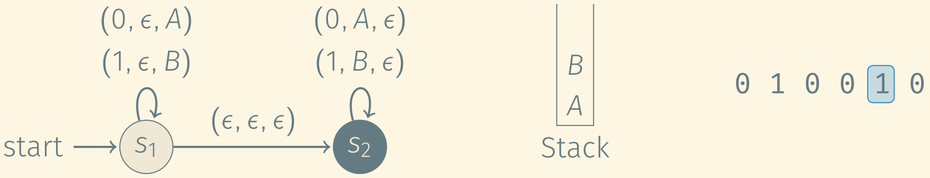


PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

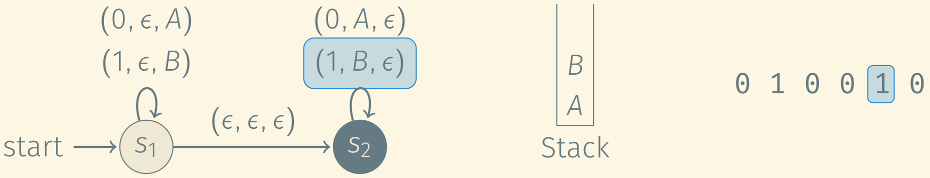


A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



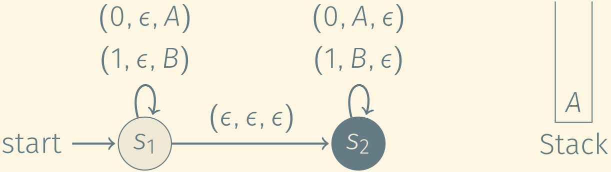
PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma \overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



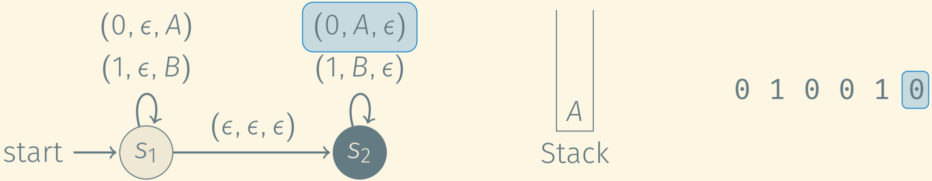
PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

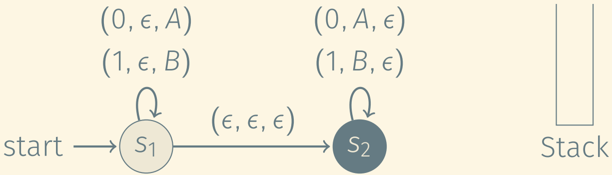


`0 1 0 0 1 0`

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

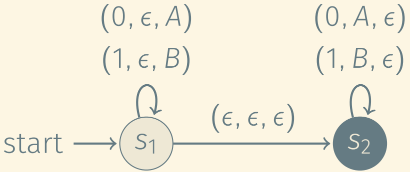


A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



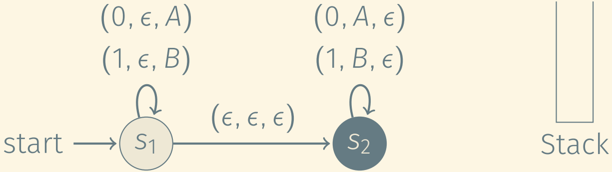
0 1 0 0 1 0

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



Accept by empty stack:
The PDA has consumed all input and the stack is empty.

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

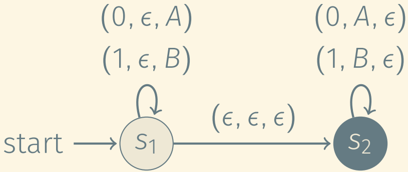


Accept by empty stack:
The PDA has consumed all input and the stack is empty.

Problem: We need to guess where the left half ends and the right half starts.

PUSH-DOWN AUTOMATON: EXAMPLE

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:

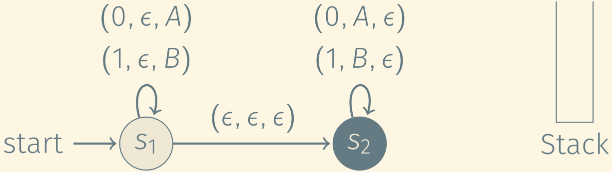


Accept by empty stack: The PDA has consumed all input and the stack is empty.

Problem: We need to guess where the left half ends and the right half starts.

This language cannot be parsed deterministically in a single pass!

A PDA for the language $\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$:



Accept by empty stack: The PDA has consumed all input and the stack is empty.

Problem: We need to guess where the left half ends and the right half starts.

This language cannot be parsed deterministically in a single pass!

In particular, it is not LL(k) or LR(k) for any k!

Definition: Push-down automaton (PDA)

A tuple $(S, \Sigma, \Gamma, \delta, s_0, \gamma, F)$:

- S is a finite set of states.
- Σ is the input alphabet.
- Γ is the stack alphabet.
- $\delta : S \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to 2^{S \times \Gamma^*}$ is the transition function.
- s_0 is the start state.
- $\gamma \in \Gamma$ is the start symbol of the stack.
- $F \subseteq S$ is the set of accepting states.

PUSH-DOWN AUTOMATON: FORMAL DEFINITION

Definition: Push-down automaton (PDA)

A tuple $(S, \Sigma, \Gamma, \delta, s_0, \gamma, F)$:

- S is a finite set of states.
- Σ is the input alphabet.
- Γ is the stack alphabet.
- $\delta : S \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow 2^{S \times \Gamma^*}$ is the transition function.
- s_0 is the start state.
- $\gamma \in \Gamma$ is the start symbol of the stack.
- $F \subseteq S$ is the set of accepting states.

Acceptance by empty stack

Accept σ if and only if it is possible to reach a configuration where the input has been consumed completely and the stack is empty.

PUSH-DOWN AUTOMATON: FORMAL DEFINITION

Definition: Push-down automaton (PDA)

A tuple $(S, \Sigma, \Gamma, \delta, s_0, \gamma, F)$:

- S is a finite set of states.
- Σ is the input alphabet.
- Γ is the stack alphabet.
- $\delta : S \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \rightarrow 2^{S \times \Gamma^*}$ is the transition function.
- s_0 is the start state.
- $\gamma \in \Gamma$ is the start symbol of the stack.
- $F \subseteq S$ is the set of accepting states.

Acceptance by empty stack

Accept σ if and only if it is possible to reach a configuration where the input has been consumed completely and the stack is empty.

Acceptance by final state

Accept σ if and only if it is possible to reach a configuration where the input has been consumed completely and the current state is an accepting state.

SOME FACTS ABOUT PDA

Lemma

The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

SOME FACTS ABOUT PDA

Lemma

The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

Lemma

A language is context-free if and only if it can be recognized by a PDA.

Lemma

The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

Lemma

A language is context-free if and only if it can be recognized by a PDA.

"Proof" by example:

$S \to \epsilon$

$S \to (S)S$

SOME FACTS ABOUT PDA

Lemma

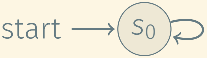
The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

Lemma

A language is context-free if and only if it can be recognized by a PDA.

“Proof” by example:

$$S \to \epsilon$$
$$S \to (S)S$$



SOME FACTS ABOUT PDA

Lemma

The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

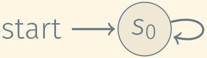
Lemma

A language is context-free if and only if it can be recognized by a PDA.

"Proof" by example:

$$S \rightarrow \epsilon$$

$$S \rightarrow (S)S$$



Start symbol: S

Lemma

The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

Lemma

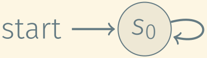
A language is context-free if and only if it can be recognized by a PDA.

"Proof" by example:

Terminals:

$S \rightarrow \epsilon$

$S \rightarrow (S)S$



$\delta(s_0, (, () = (s_0, \epsilon)$

$\delta(s_0,),)) = (s_0, \epsilon)$

Start symbol: S

Lemma

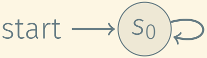
The two modes of acceptance are equivalent: The exists a PDA deciding a language $\mathcal{L}$ by empty stack if and only if there exists a PDA deciding $\mathcal{L}$ by final state.

Lemma

A language is context-free if and only if it can be recognized by a PDA.

"Proof" by example:

$$S \to \epsilon$$
$$S \to (S)S$$



Terminals:

$$\delta(s_0, (, () = (s_0, \epsilon)$$
$$\delta(s_0,),)) = (s_0, \epsilon)$$

Rules:

$$\delta(s_0, \epsilon, S) = \{(s_0, \epsilon), (s_0, (S)S)\}$$

Start symbol: S

DETERMINISTIC PUSH-DOWN AUTOMATA

By default, a PDA is non-deterministic: Multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

DETERMINISTIC PUSH-DOWN AUTOMATA

By default, a PDA is non-deterministic: Multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a deterministic PDA (DPDA).

By default, a PDA is non-deterministic: Multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a deterministic PDA (DPDA).

Lemma

A language can be decided by a DPDA if and only if it is LL(k) or LR(k) for some k.

By default, a PDA is non-deterministic: Multiple transitions are possible for a given combination of state, input symbol, and symbol on the top of the stack.

If there is only one possible transition, for any combination of state, input symbol, and stack symbol, we call the PDA a deterministic PDA (DPDA).

Lemma

A language can be decided by a DPDA if and only if it is LL(k) or LR(k) for some k.

In particular, there exist context-free languages that cannot be decided by a DPDA:

$$\{\sigma\overleftarrow{\sigma} \mid \sigma \in \{0,1\}^*\}$$

PARSING LL(1) LANGUAGES USING DPDA (1)

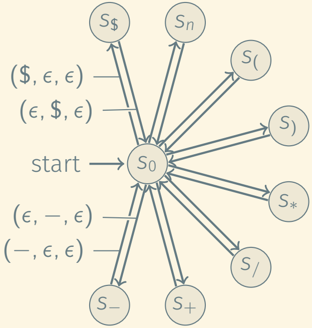
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

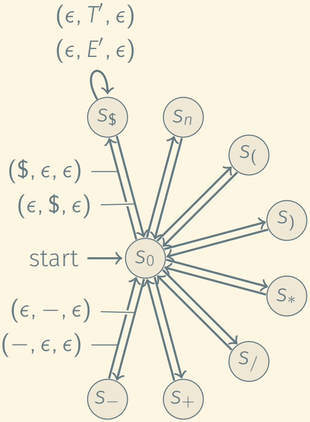
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



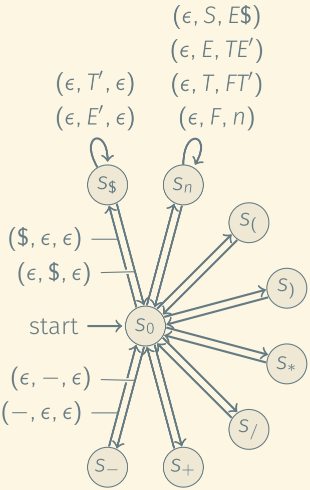
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



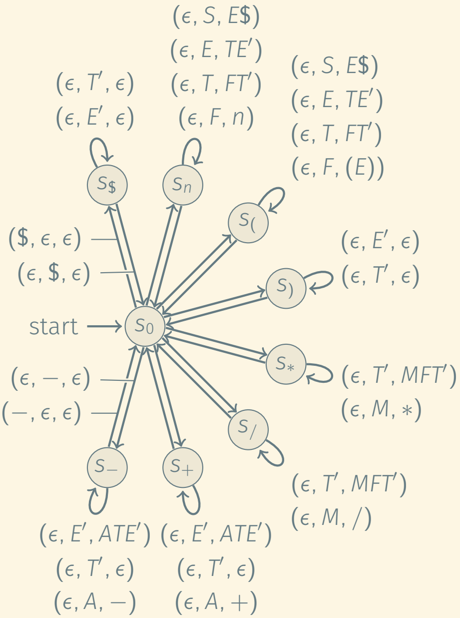
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



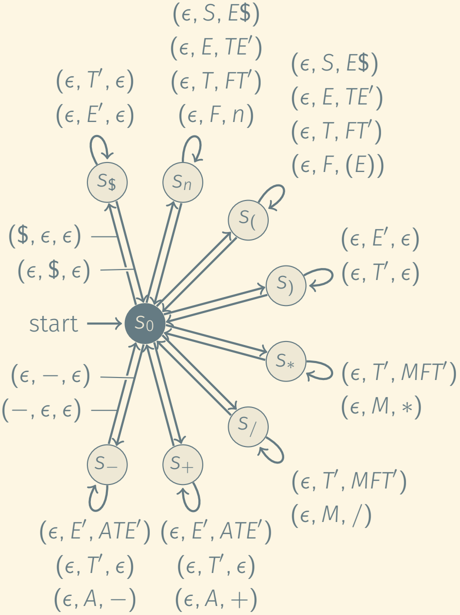
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$



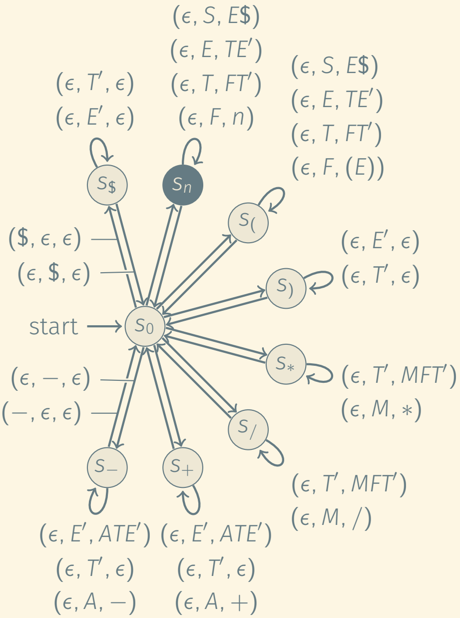
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



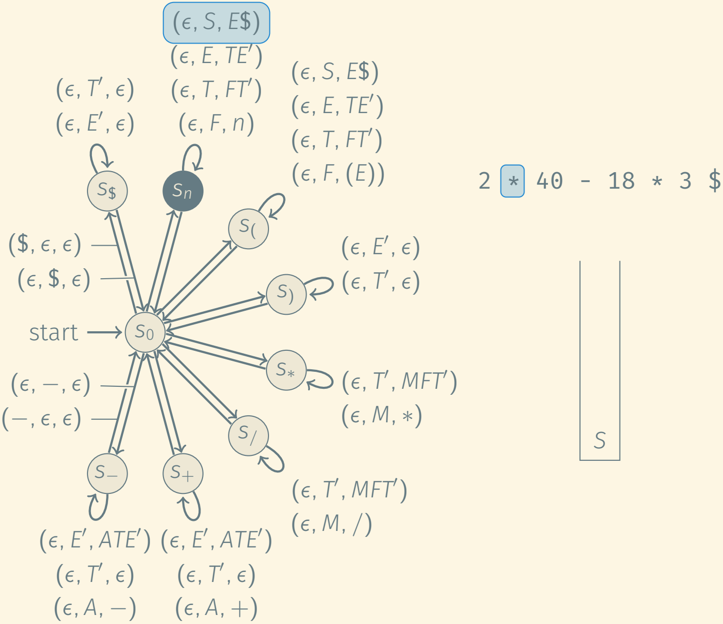
2 * 40 - 18 * 3 $

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

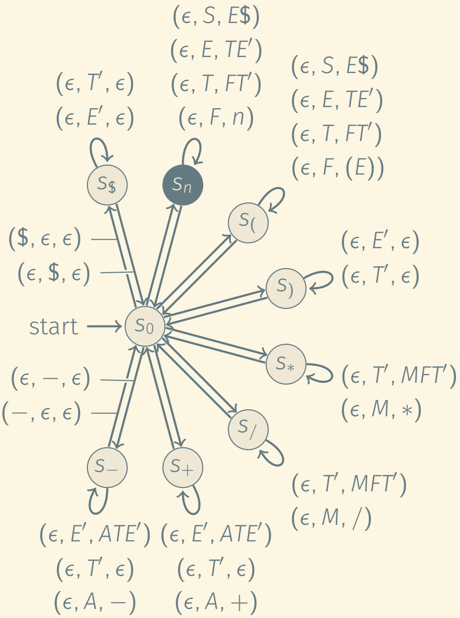


2 * 40 - 18 * 3 $

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



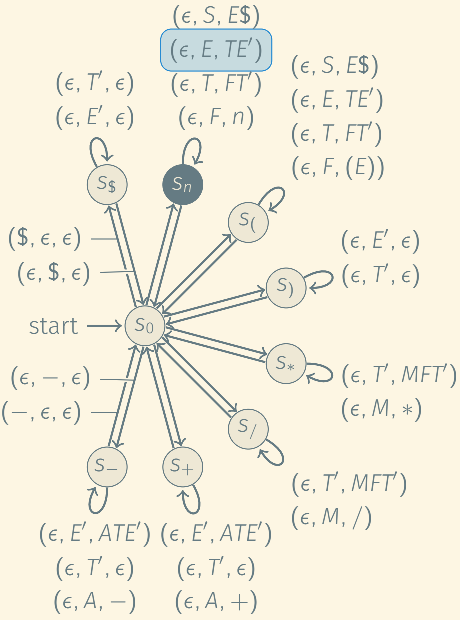
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



2 * 40 - 18 * 3 $

PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



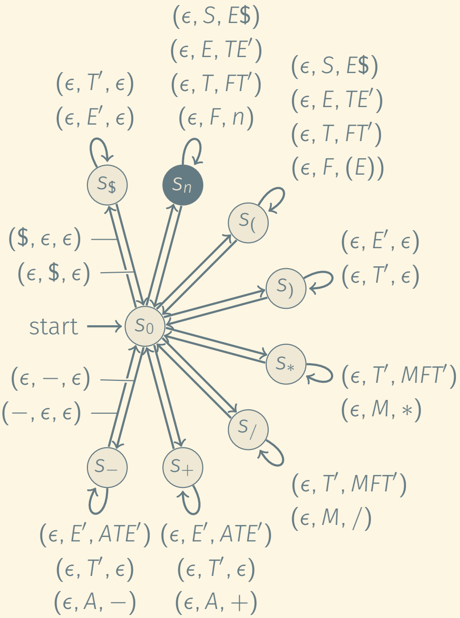
`2 * 40 - 18 * 3 $`

Stack (top to bottom):

E
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

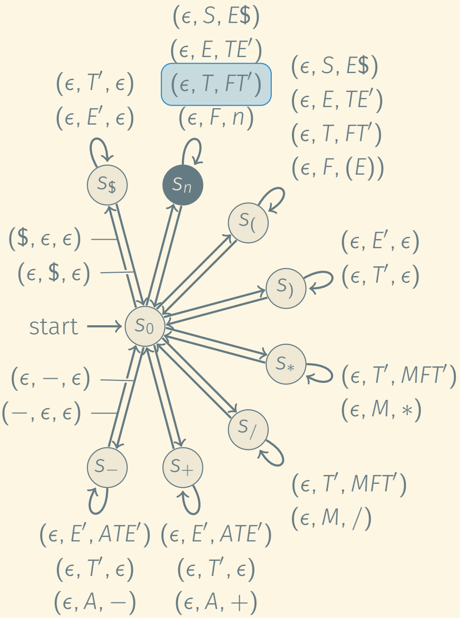
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$ $E' \to A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$ $T' \to M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \to n$ $F \to (E)$	$\{n\}$ $\{(\}$
$A \to +$ $A \to -$	$\{+\}$ $\{-\}$
$M \to *$ $M \to /$	$\{*\}$ $\{/\}$



`2 * 40 - 18 * 3 $`

T
E'
$\$$

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

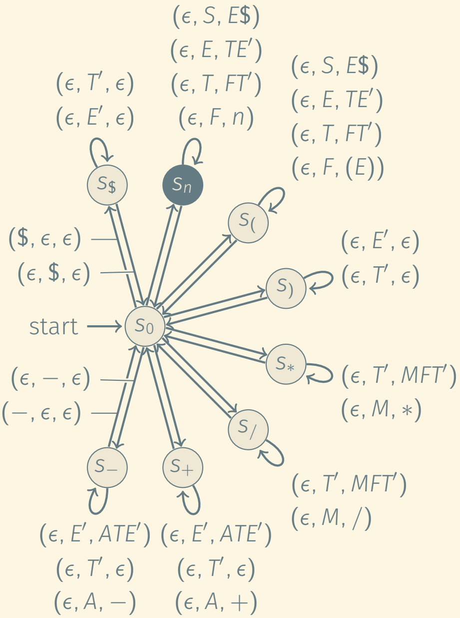


`2 * 40 - 18 * 3 $`

T
E'
$

PARSING LL(1) LANGUAGES USING DPDA (1)

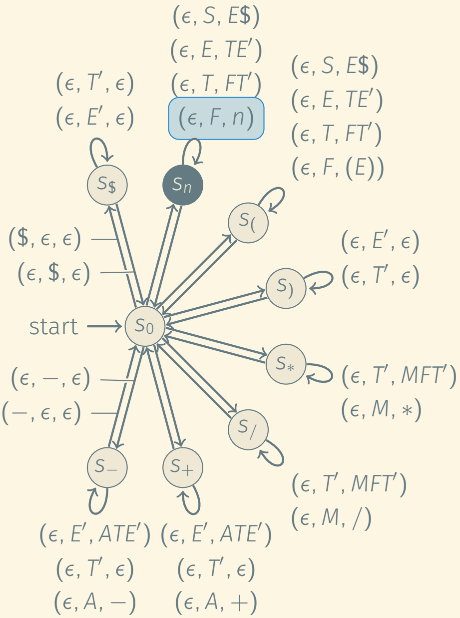
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



2 * 40 - 18 * 3 $

F
T'
E'
$\$$

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



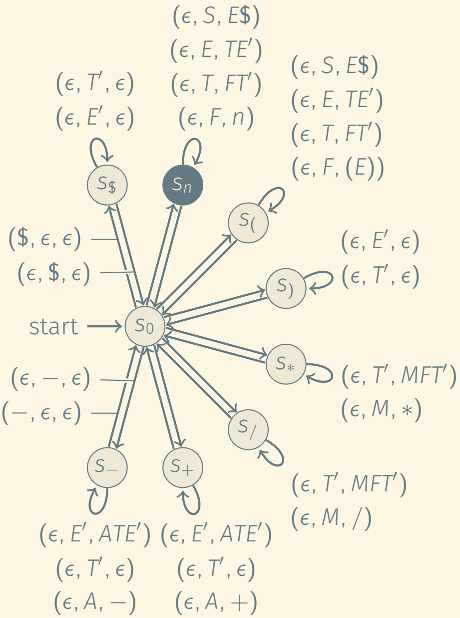
`2 * 40 - 18 * 3 $`

Stack:

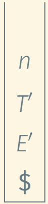
F
T'
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

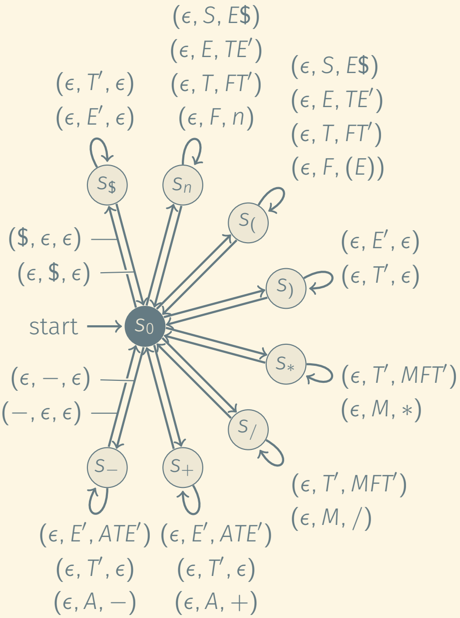



```
2 * 40 - 18 * 3 $
```

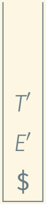


PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$ $E' \to A\,T\,E'$	$\{\$,)\}$ $\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$ $T' \to M\,F\,T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \to n$ $F \to (E)$	$\{n\}$ $\{(\}$
$A \to +$ $A \to -$	$\{+\}$ $\{-\}$
$M \to *$ $M \to /$	$\{*\}$ $\{/\}$

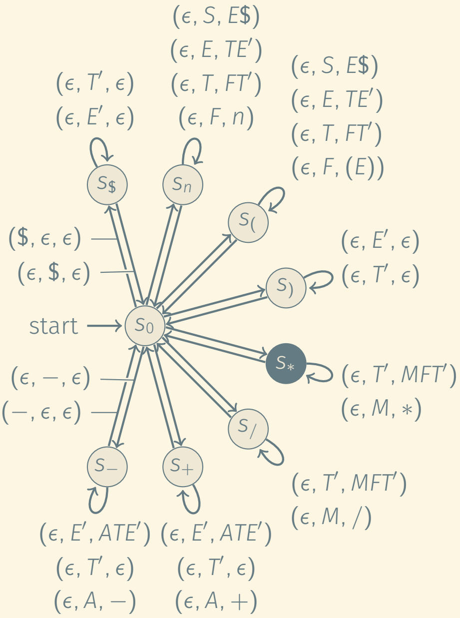


`2 * 40 - 18 * 3 $`

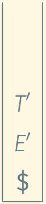


PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$

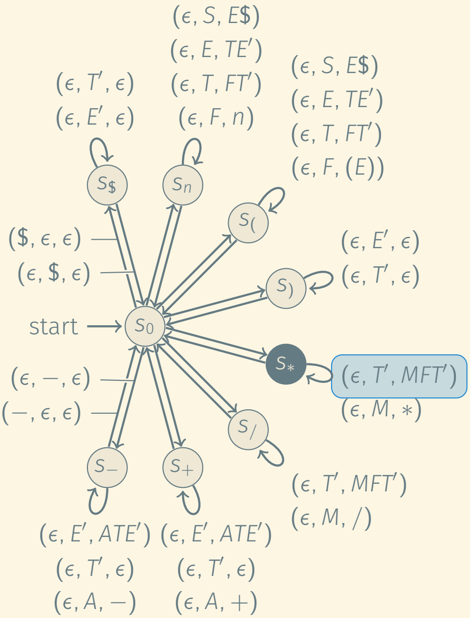


`2 * 40 - 18 * 3 $`



PARSING LL(1) LANGUAGES USING DPDA (1)

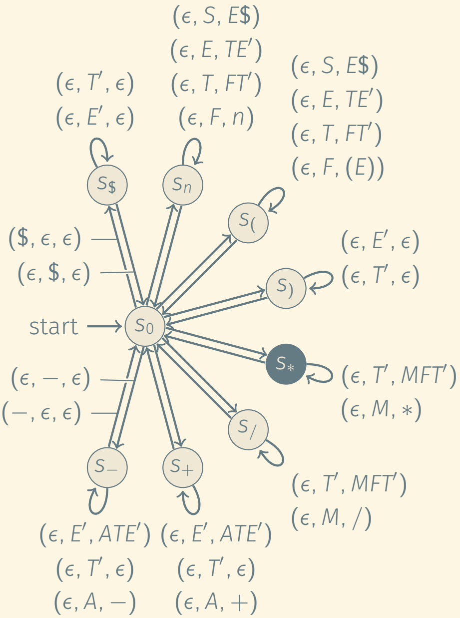
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$ $E' \to A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$ $T' \to M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \to n$ $F \to (E)$	$\{n\}$ $\{(\}$
$A \to +$ $A \to -$	$\{+\}$ $\{-\}$
$M \to *$ $M \to /$	$\{*\}$ $\{/\}$



`2 * 40 - 18 * 3 $`

Stack
T'
E'
$\$$

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

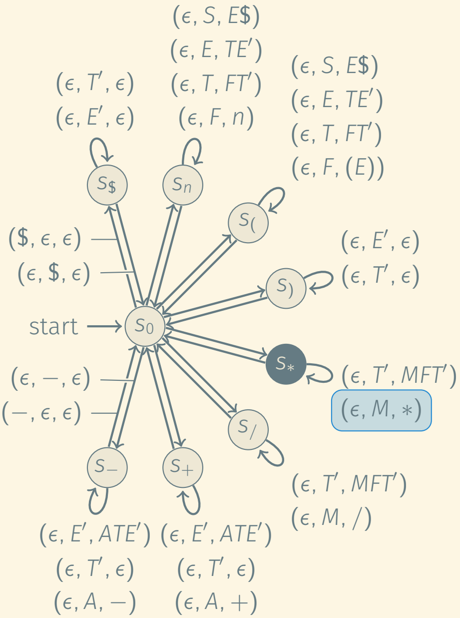


`2 * 40 - 18 * 3 $`

M
F
T'
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

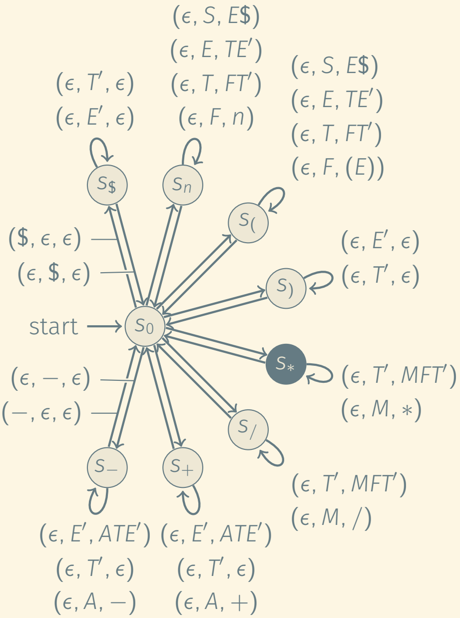
Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

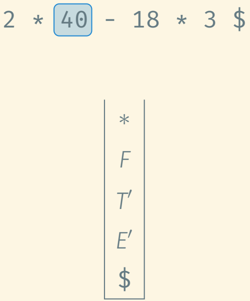


2 * 40 - 18 * 3 $

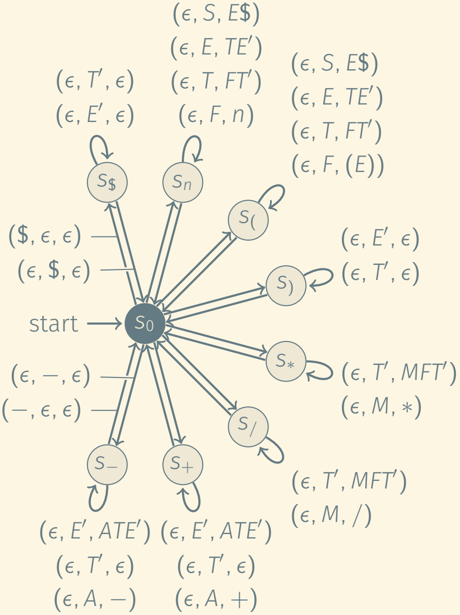
M
F
T'
E'
$\$$

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

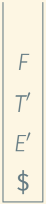




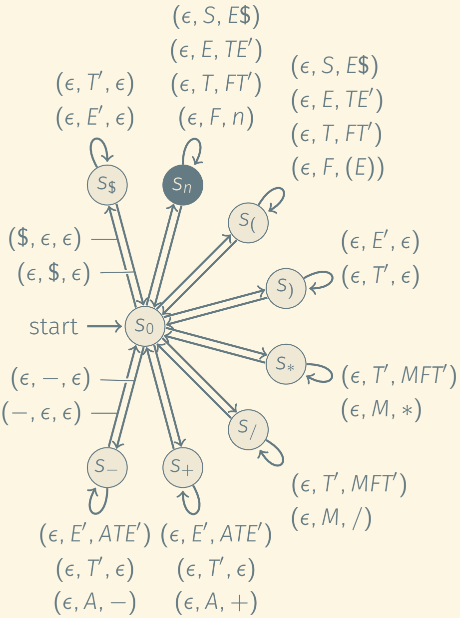
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



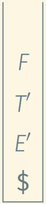
`2 * 40 - 18 * 3 $`



Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

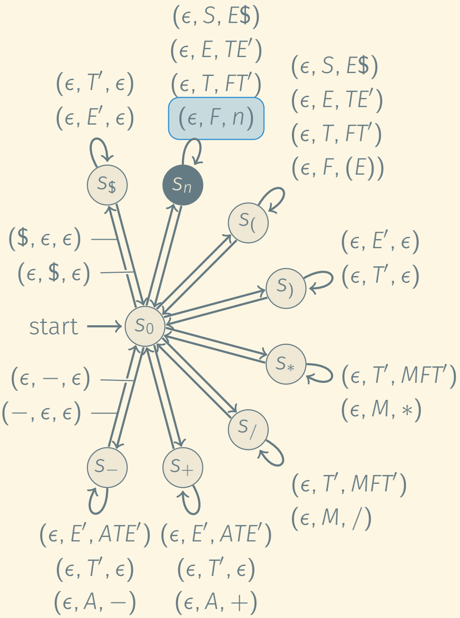


`2 * 40 - 18 * 3 $`

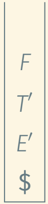


PARSING LL(1) LANGUAGES USING DPDA (1)

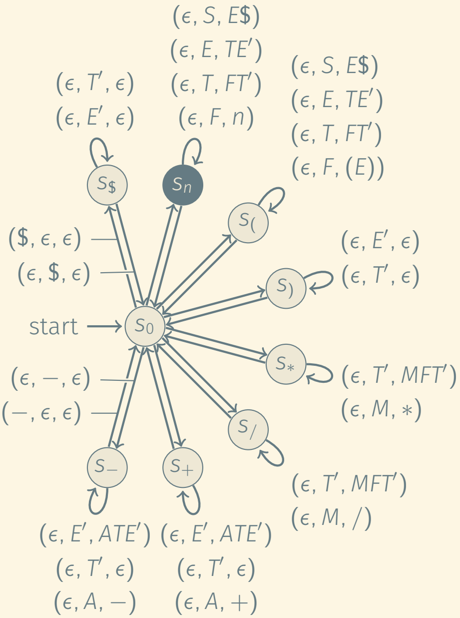
Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\,T\,E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\,F\,T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$



2 * 40 - 18 * 3 $



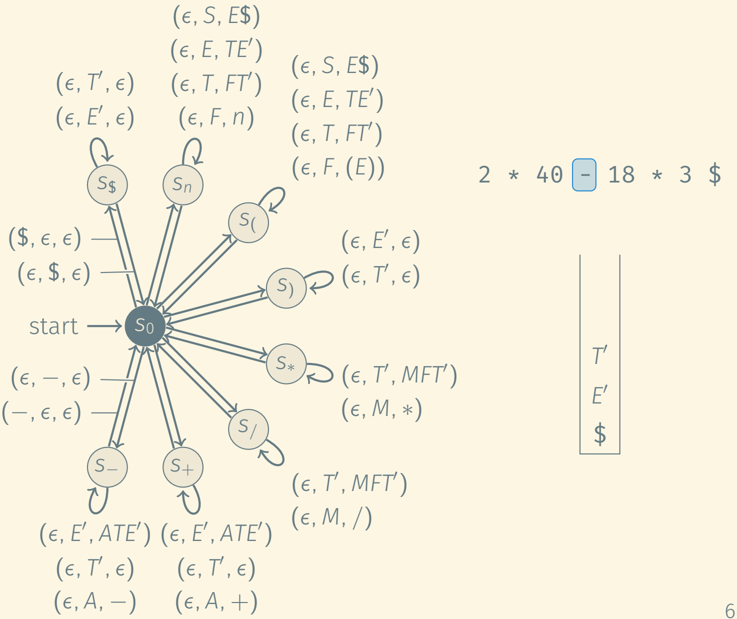
Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



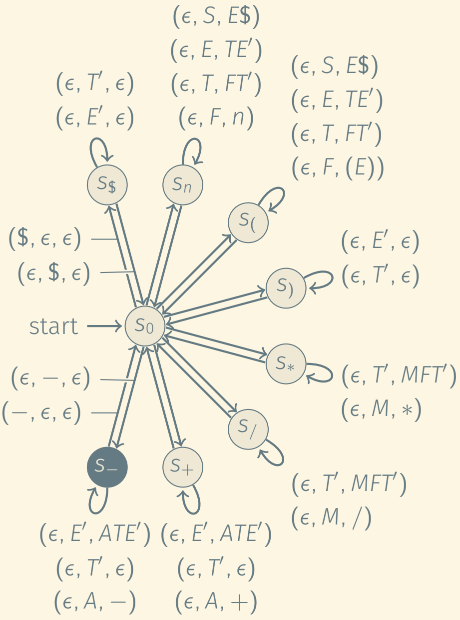
`2 * 40 - 18 * 3 $`

Stack
n
T'
E'
$\$$

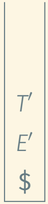
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

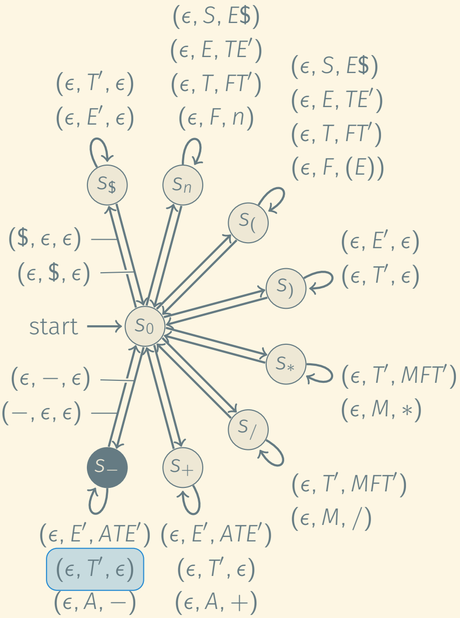


`2 * 40 - 18 * 3 $`



PARSING LL(1) LANGUAGES USING DPDA (1)

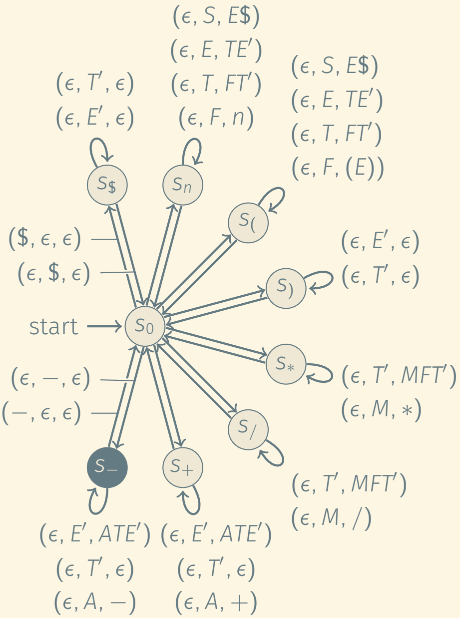
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



`2 * 40 - 18 * 3 $`

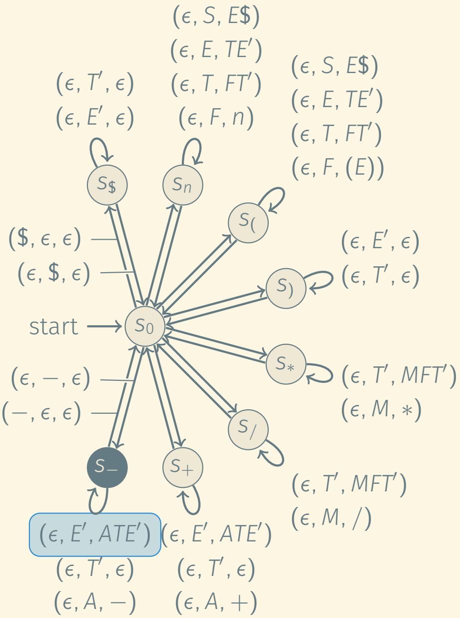
T'
E'
$\$$

Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



`2 * 40 - 18 * 3 $`

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

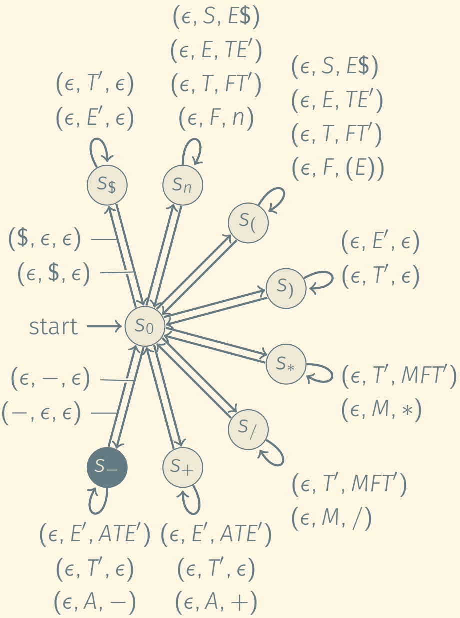


2 * 40 - 18 * 3 $

Stack: E', $\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

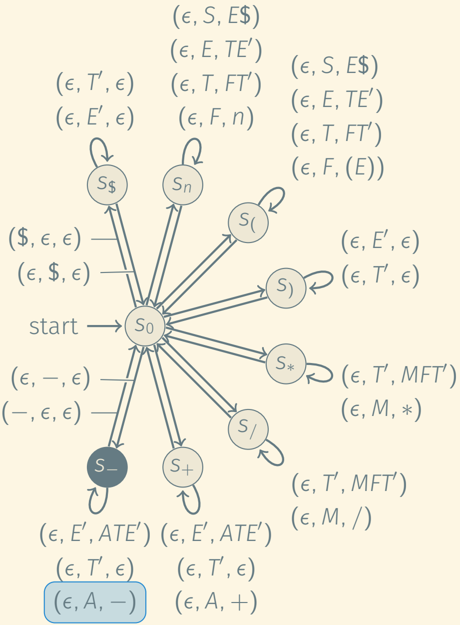


`2 * 40 - 18 * 3 $`

Stack
A
T
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$

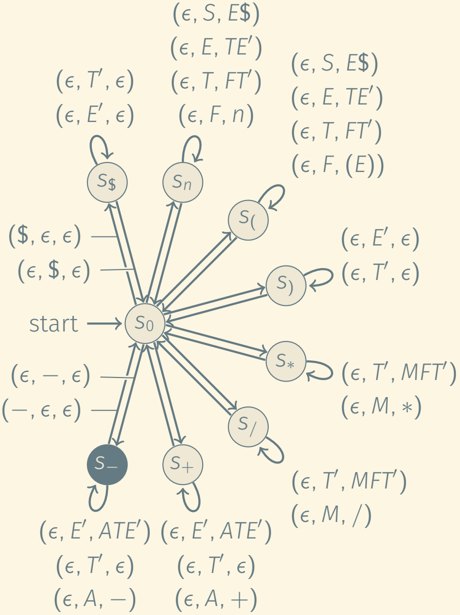


`2 * 40 - 18 * 3 $`

A
T
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

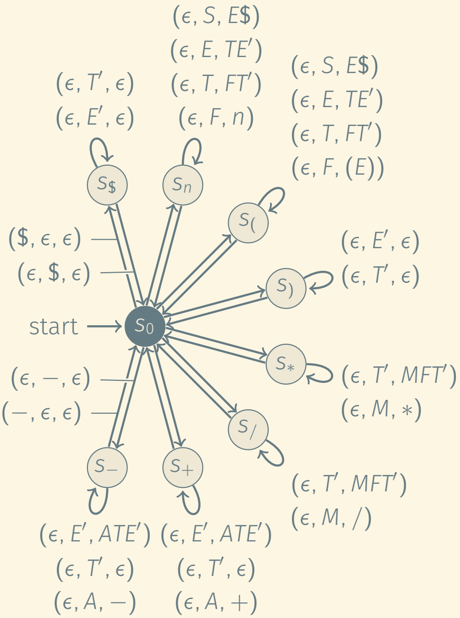
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

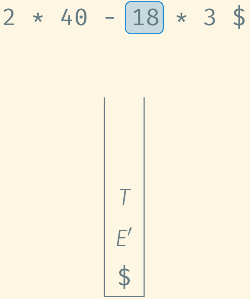


`2 * 40 - 18 * 3 $`

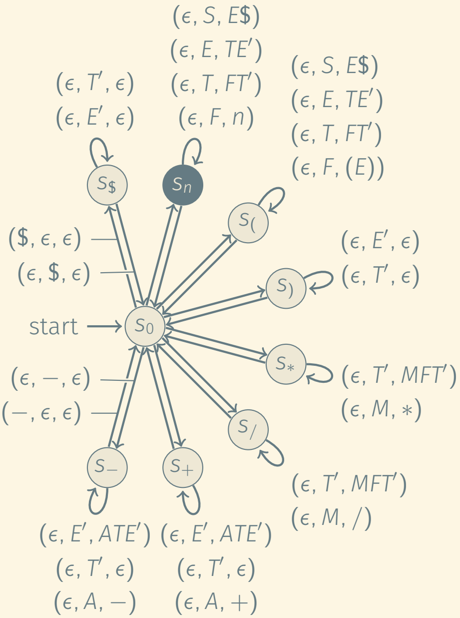
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$





Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



`2 * 40 - 18 * 3 $`

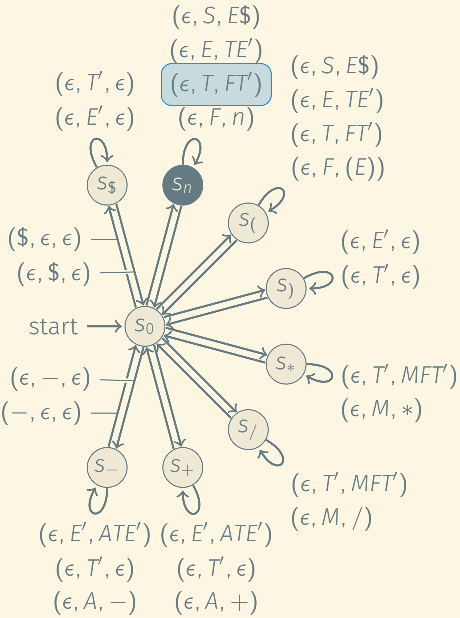
T

E'

$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

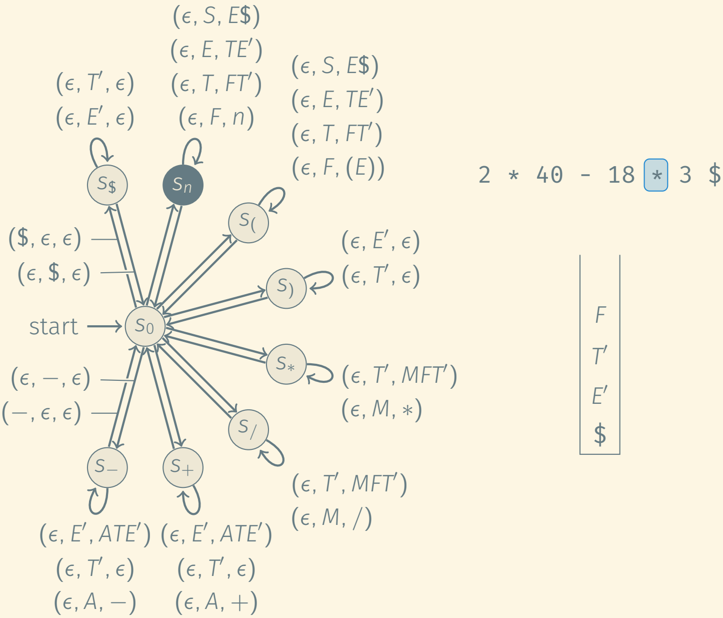
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$ $E' \to A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$ $T' \to M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \to n$ $F \to (E)$	$\{n\}$ $\{(\}$
$A \to +$ $A \to -$	$\{+\}$ $\{-\}$
$M \to *$ $M \to /$	$\{*\}$ $\{/\}$



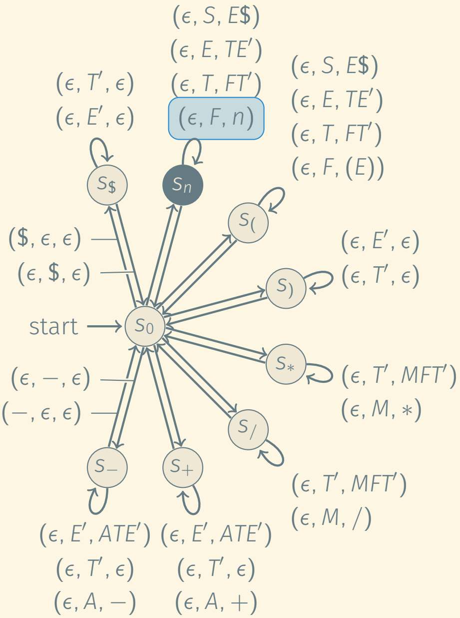
`2 * 40 - 18 * 3 $`

Stack
T
E'
$\$$

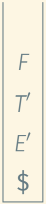
Rule R	PREDICT(R)
$S \to E\,\$$	$\{n, (\}$
$E \to T\,E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\,T\,E'$	$\{+, -\}$
$T \to F\,T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\,F\,T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

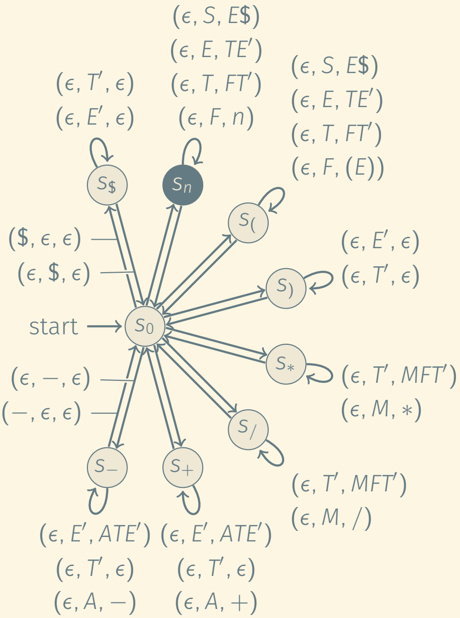


`2 * 40 - 18 * 3 $`



PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

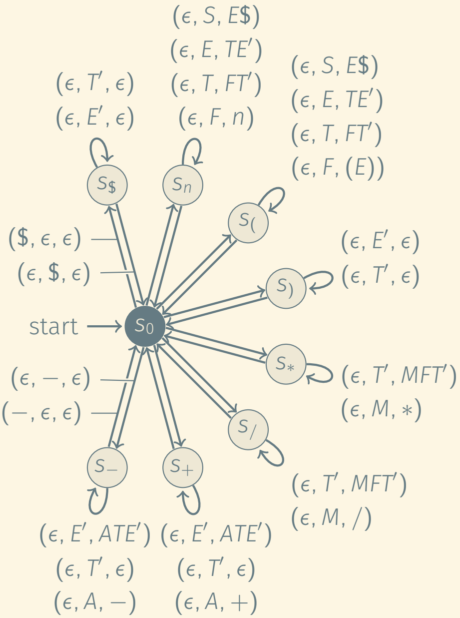


`2 * 40 - 18 * 3 $`

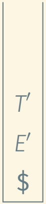
n
T'
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

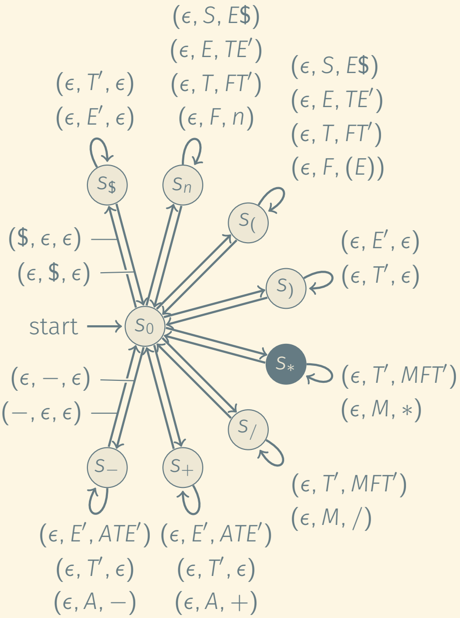
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



`2 * 40 - 18 * 3 $`



Rule R	PREDICT(R)
$S \rightarrow E\,\$$	$\{n, (\}$
$E \rightarrow T\,E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\,T\,E'$	$\{+, -\}$
$T \rightarrow F\,T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\,F\,T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

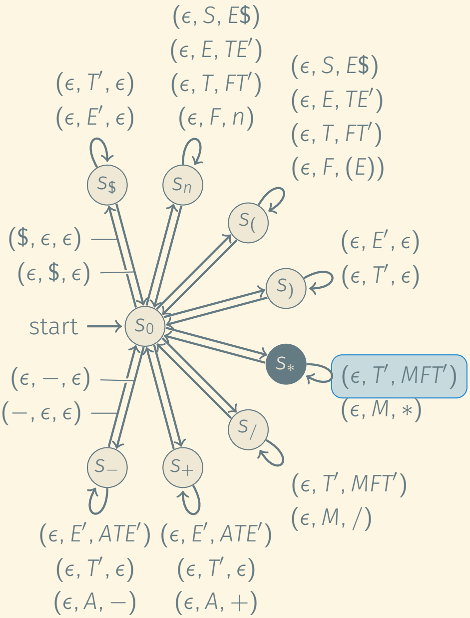


`2 * 40 - 18 * 3 $`

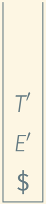
Stack:

T'
E'
$\$$

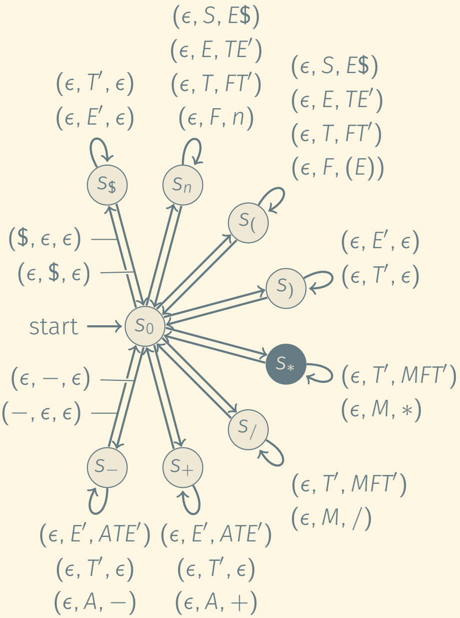
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



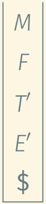
`2 * 40 - 18 * 3 $`



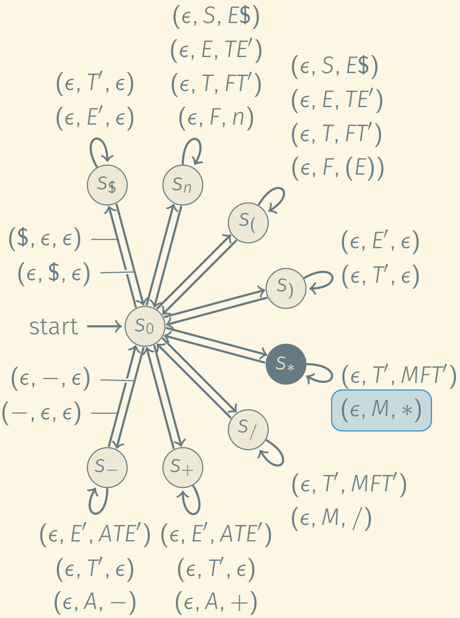
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



`2 * 40 - 18 * 3 $`



Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

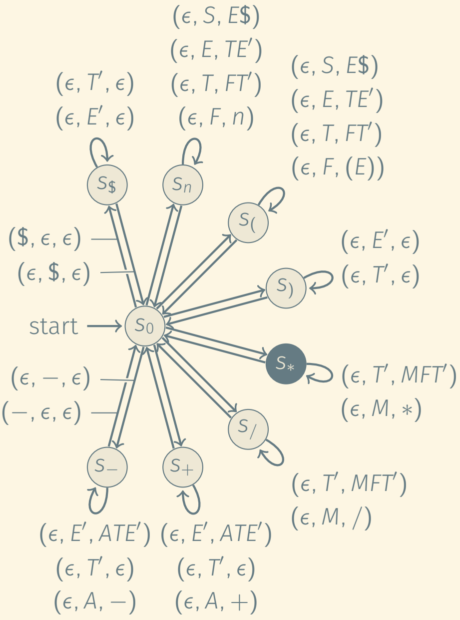


`2 * 40 - 18 * 3 $`

Stack
M
F
T'
E'
$\$$

PARSING LL(1) LANGUAGES USING DPDA (1)

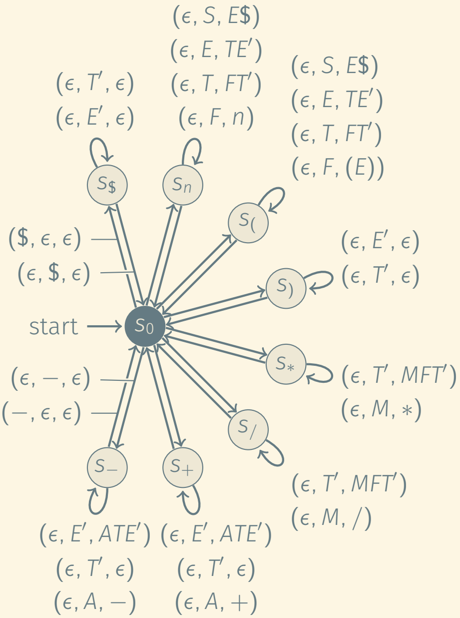
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



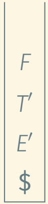
`2 * 40 - 18 * 3 $`

Stack
$*$
F
T'
E'
$\$$

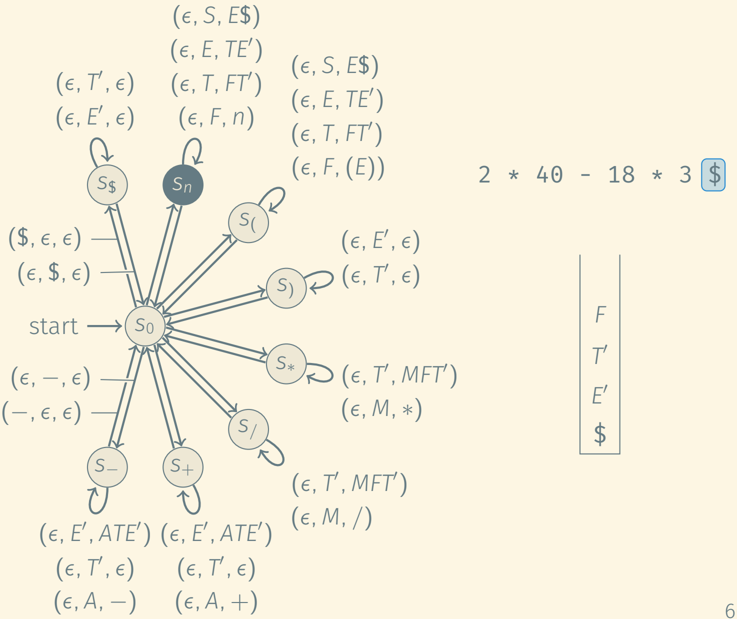
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



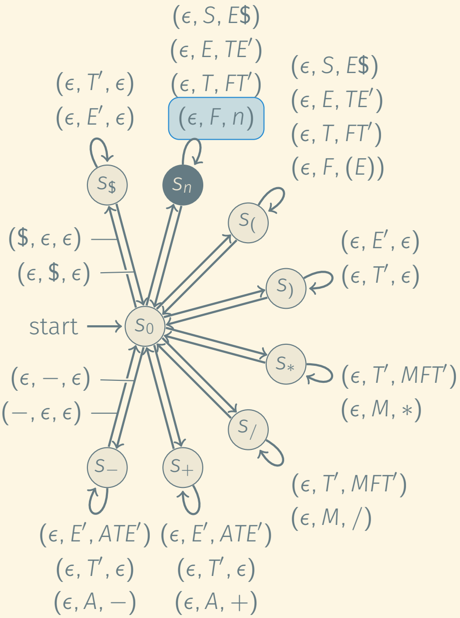
`2 * 40 - 18 * 3 $`



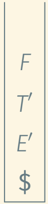
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$



Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

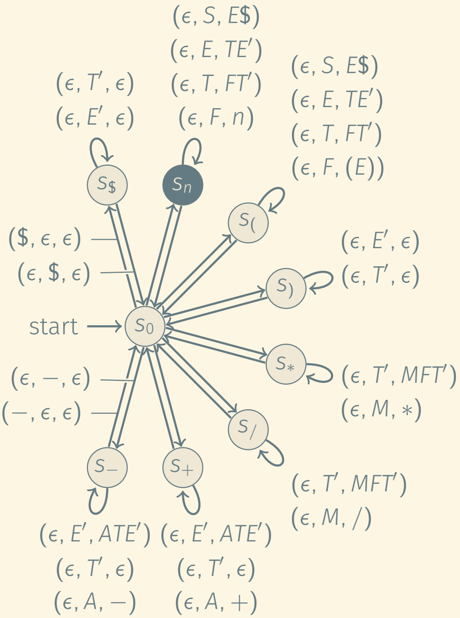


`2 * 40 - 18 * 3 $`



PARSING LL(1) LANGUAGES USING DPDA (1)

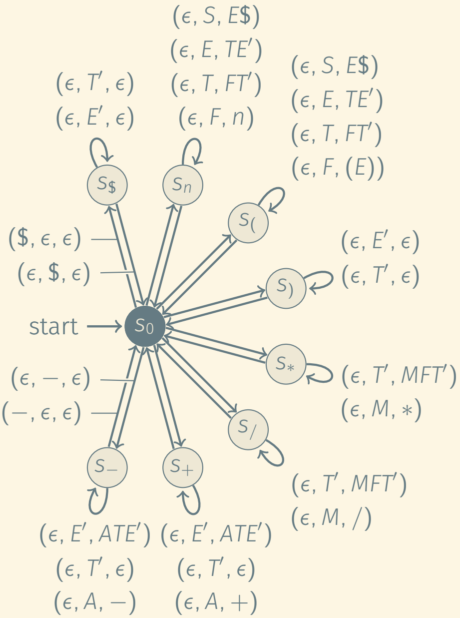
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



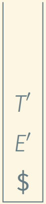
`2 * 40 - 18 * 3 $`

n
T'
E'
$\$$

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$

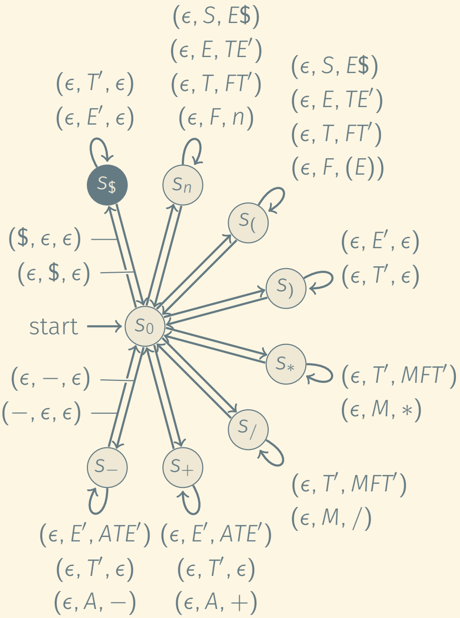


2 * 40 - 18 * 3 $

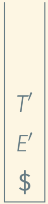


PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$

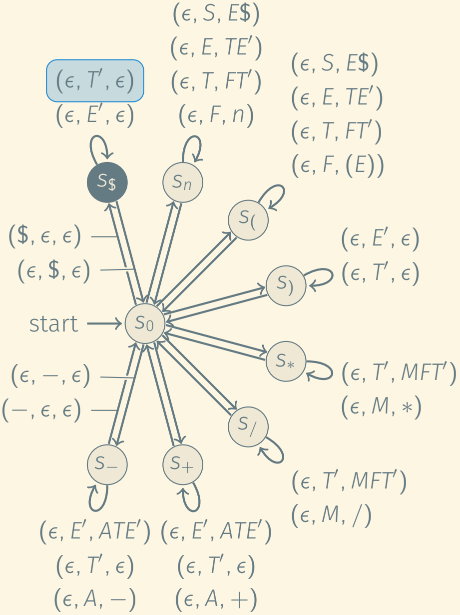


`2 * 40 - 18 * 3 $`



PARSING LL(1) LANGUAGES USING DPDA (1)

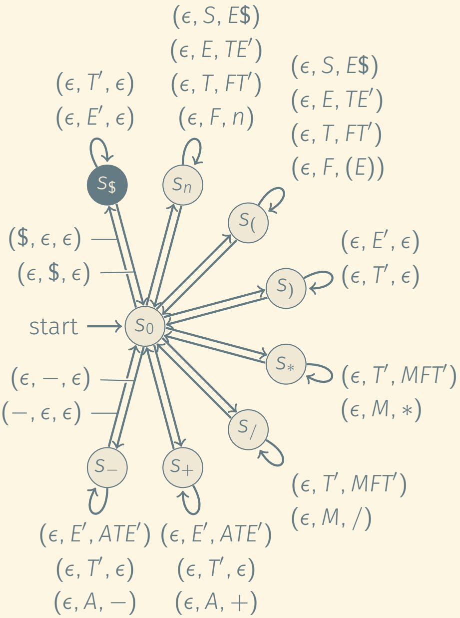
Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$ $E' \to A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$ $T' \to M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \to n$ $F \to (E)$	$\{n\}$ $\{(\}$
$A \to +$ $A \to -$	$\{+\}$ $\{-\}$
$M \to *$ $M \to /$	$\{*\}$ $\{/\}$



`2 * 40 - 18 * 3 $`

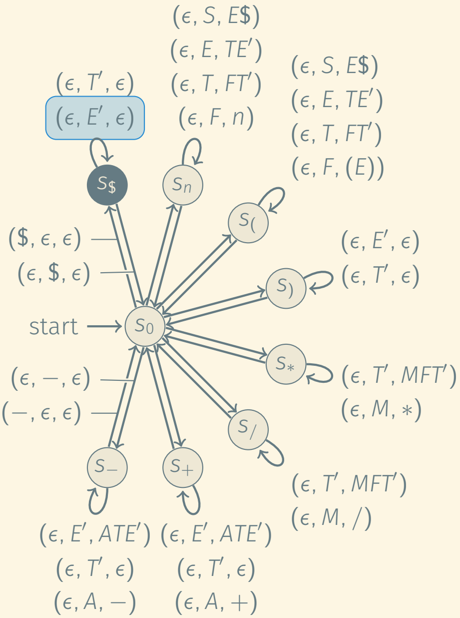
Stack
T'
E'
$\$$

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$ $E' \rightarrow A\ T\ E'$	$\{\$,)\}$ $\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$ $T' \rightarrow M\ F\ T'$	$\{+, -, \$,)\}$ $\{*, /\}$
$F \rightarrow n$ $F \rightarrow (E)$	$\{n\}$ $\{(\}$
$A \rightarrow +$ $A \rightarrow -$	$\{+\}$ $\{-\}$
$M \rightarrow *$ $M \rightarrow /$	$\{*\}$ $\{/\}$



`2 * 40 - 18 * 3 $`

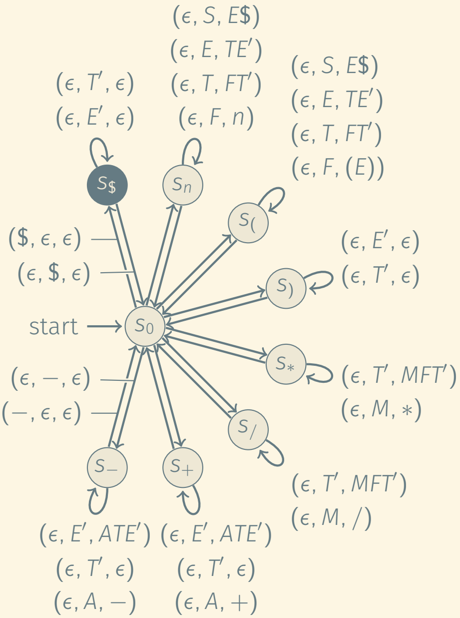
Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



2 * 40 - 18 * 3 $

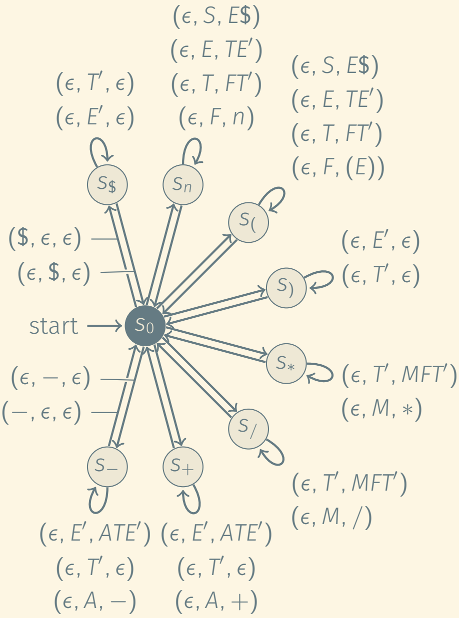
PARSING LL(1) LANGUAGES USING DPDA (1)

Rule R	PREDICT(R)
$S \to E\ \$$	$\{n, (\}$
$E \to T\ E'$	$\{n, (\}$
$E' \to \epsilon$	$\{\$,)\}$
$E' \to A\ T\ E'$	$\{+, -\}$
$T \to F\ T'$	$\{n, (\}$
$T' \to \epsilon$	$\{+, -, \$,)\}$
$T' \to M\ F\ T'$	$\{*, /\}$
$F \to n$	$\{n\}$
$F \to (E)$	$\{(\}$
$A \to +$	$\{+\}$
$A \to -$	$\{-\}$
$M \to *$	$\{*\}$
$M \to /$	$\{/\}$



2 * 40 - 18 * 3 $

Rule R	PREDICT(R)
$S \rightarrow E\ \$$	$\{n, (\}$
$E \rightarrow T\ E'$	$\{n, (\}$
$E' \rightarrow \epsilon$	$\{\$,)\}$
$E' \rightarrow A\ T\ E'$	$\{+, -\}$
$T \rightarrow F\ T'$	$\{n, (\}$
$T' \rightarrow \epsilon$	$\{+, -, \$,)\}$
$T' \rightarrow M\ F\ T'$	$\{*, /\}$
$F \rightarrow n$	$\{n\}$
$F \rightarrow (E)$	$\{(\}$
$A \rightarrow +$	$\{+\}$
$A \rightarrow -$	$\{-\}$
$M \rightarrow *$	$\{*\}$
$M \rightarrow /$	$\{/\}$



2 * 40 - 18 * 3 $

PARSING LL(1) LANGUAGES USING DPDA (2)

Implementation:

PARSING LL(1) LANGUAGES USING DPDA (2)

Implementation:

- Using nested case statements:
  - Level 1: Branch on current state
  - Level 2: Branch on current input symbol
  - Level 3: Branch on current stack symbol

PARSING LL(1) LANGUAGES USING DPDA (2)

Implementation:

- Using nested case statements:
  - Level 1: Branch on current state
  - Level 2: Branch on current input symbol
  - Level 3: Branch on current stack symbol

  Some similarity to recursive-descent parsing.

  Instead of recursion, maintain the stack explicitly.

PARSING LL(1) LANGUAGES USING DPDA (2)

Implementation:

- Using nested case statements:
  - Level 1: Branch on current state
  - Level 2: Branch on current input symbol
  - Level 3: Branch on current stack symbol

  Some similarity to recursive-descent parsing.

  Instead of recursion, maintain the stack explicitly.

- Table-driven:
  - 3-d table mapping (state, input symbol, stack symbol) triples to strings to be pushed on the stack.

PARSING LL(1) LANGUAGES USING DPDA (2)

Implementation:

- Using nested case statements:
  - Level 1: Branch on current state
  - Level 2: Branch on current input symbol
  - Level 3: Branch on current stack symbol

  Some similarity to recursive-descent parsing.

  Instead of recursion, maintain the stack explicitly.

- Table-driven:
  - 3-d table mapping (state, input symbol, stack symbol) triples to strings to be pushed on the stack.

Generating the parser:

- Hand-coded
- Automatic generation from grammar

SUMMARY

- Context-free grammars can be used to describe the structure of programming languages.
- Every context-free grammar can be parsed by PDA.
- Every context-free grammar can be parsed deterministically in $O(n^3)$ time.
- Linear-time parsing is possible for restricted grammars (S-grammar, LL(k), LR(k), ...).
- Tools: Recursive descent parser, shift-reduce parser, DPDA.