Syntax Analysis

Amitabha Sanyal

(www.cse.iitb.ac.in/~as)

Department of Computer Science and Engineering,

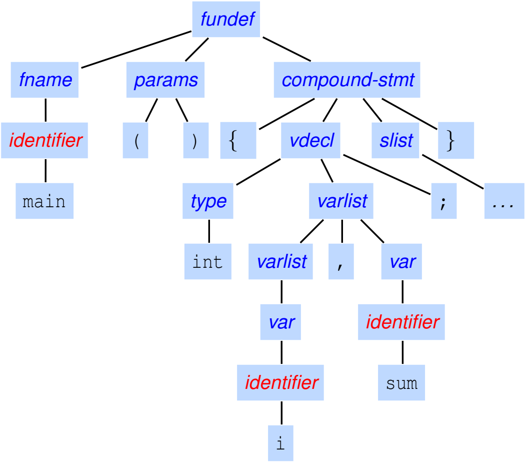
Indian Institute of Technology, Bombay

January 2014

Syntax analysis – example

Syntax analysis discovers the larger structures in a program.

```
main ()
{
  int i,sum;
  sum = 0;
  for (i=1; i<=10; i++)
    sum = sum + i;
  printf("%d\n",sum);
}
```



Parsing

A syntax analyzer or parser

- Ensures that the input program is well-formed by attempting to group tokens according to certain rules. This is syntax checking.

Parsing

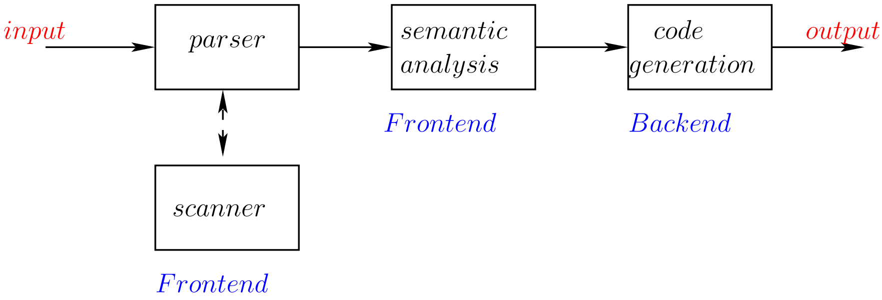
A syntax analyzer or parser

- Ensures that the input program is well-formed by attempting to group tokens according to certain rules. This is syntax checking.
-
    - May also create the hierarchical structure that arises out of such grouping.
    - The tree like representation of the structure is called a *parse tree*.
    - This information is required by subsequent phases.

Place of a parser in a compiler organization

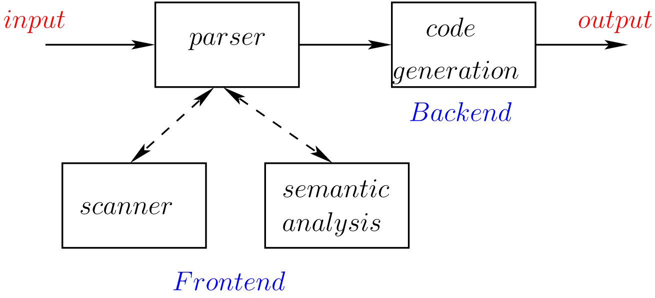
Where is the place of the parser in the overall organization of the compiler?

1. Parser driven syntax tree creation. The parser creates the syntax tree and passes control to the later stages.



Place of a parser in a compiler organization

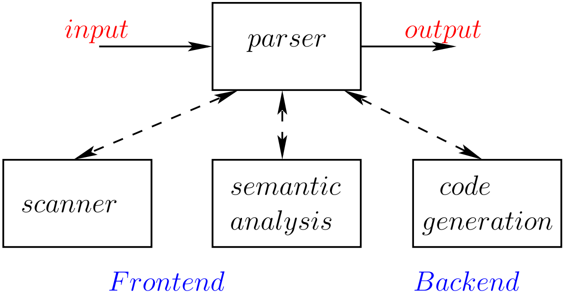
2. Parser driven front-end. The parser also does the semantic analysis along with parsing.



GCC does this. You will also do this.

Place of a parser in a compiler organization

3. Parser driven compilation. The entire compilation is interleaved along with parsing.



Parser Construction

How are parsers constructed ?

- Till early seventies, parsers (in fact the entire compiler) were written manually.
- A better understanding of parsing algorithms has resulted in tools that can automatically generate parsers.
- Examples of parser generating tools:
    - Yacc/Bison: Bottom-up (LALR) parser generator
    - Antlr: Top-down (LL) scanner cum parser generator. (Terence Parr)
    - PCCTS:Precursor of Antlr (Terence Parr)
    - COCO/R: Lexer and Parser Generators in various languages, generates recursive descent parsers (Hanspeter Mossenbock).
    - Java Compiler Compiler (JavaCC)
- GCC and LLVM both use hand-written parsers for better efficiency and error reporting.

Specification of syntax

- To check whether a program is well-formed requires a specification of what is a well-formed program:

  1. The specification should be unambiguous.
  2. The specification should be correct and complete. Must cover all the syntactic details of the language
  3. the specification must be convenient to use by both language designer and the implementer

A *context free grammar* meets these requirements.

Context Free Grammar (CFG)

start symbol

non-terminals

$$\begin{array}{lcl} \textit{declaration} & \rightarrow & \textit{type idlist } \mathbf{;} \\ \textit{idlist} & \rightarrow & \mathbf{id} \mid \textit{idlist}\ \mathbf{,\ id} \\ \boxed{\textit{type} \quad \rightarrow \quad \mathbf{integer} \mid \mathbf{float}} \end{array}$$

terminals

production

A CFG G is a 4-tuple (N, T, S, P), where :

1. N is a finite set of nonterminals.
2. T is a finite set of terminals.
3. S is a special nonterminal (from N) called the *start* symbol.
4. P is a finite set of production rules of the form such as $A \rightarrow \alpha$, where A is from N and α from $(N \cup T)^*$

Derivation

What is the language defined by a grammar? To answer this, we need the notion of a *derivation*.

Derivation

What is the language defined by a grammar? To answer this, we need the notion of a *derivation*.

A derivation is traced out as follows:

$$
\begin{array}{ll}
 & \textit{declaration} \\
\Rightarrow & \textit{type idlist}; \\
\Rightarrow & \textbf{integer } \textit{idlist}; \\
\Rightarrow & \textbf{integer } \textit{idlist}, \textbf{id}; \\
\Rightarrow & \textbf{integer id}, \textbf{id};
\end{array}
$$

Derivation

What is the language defined by a grammar? To answer this, we need the notion of a *derivation*.

A derivation is traced out as follows:

$$
\begin{array}{ll}
 & \textit{declaration} \\
\Rightarrow & \textit{type idlist}; \\
\Rightarrow & \textbf{integer } \textit{idlist}; \\
\Rightarrow & \textbf{integer } \textit{idlist}, \textbf{id}; \\
\Rightarrow & \textbf{integer id}, \textbf{id};
\end{array}
$$

- A *derivation* is the transformation of a string of grammar symbols by replacing a non-terminal by the corresponding right hand side of a production.
- The set of all possible strings of terminal symbols that can be derived from the start symbol of a CFG is the *language generated by the CFG*.

Specification of Syntax by Context Free Grammars

Informal description of variable declarations in C:

- starts with **integer** or **float** as the first token.
- followed by identifier tokens, separated by token **comma**
- followed by token **semicolon**

Question: Can the list of identifier tokens be empty?

$$
\begin{array}{lcl}
\textit{declaration} & \rightarrow & \textit{type idlist} \ \textbf{;} \\
\textit{idlist} & \rightarrow & \textbf{id} \mid \textit{idlist} \ \textbf{,} \ \textbf{id} \\
\textit{type} & \rightarrow & \textbf{integer} \mid \textbf{float}
\end{array}
$$

Illustrates the usefulness of a formal specification.

Question: How does one write a grammar in which the list of identifiers is empty?

Describing prog. language constructs using grammars

- Question: How dose one write a grammar for assignment statements?
- Question: What language does the following grammar represent?

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
E & \rightarrow & T \\
T & \rightarrow & T * F \\
T & \rightarrow & F \\
F & \rightarrow & (E) \\
F & \rightarrow & \mathtt{id}
\end{array}
$$

Why the Term "Context Free" ?

1. The only kind of productions permitted are of the form
   non-terminal $\rightarrow$ *sequence of terminals and non-terminals*
2. In a derivation, the replacement is made regardless of the context (symbols surrounding the non-terminal).

As a contrast, observe this context-sensitive grammar.

sentence	$\rightarrow$	*NP VP*
NP	$\rightarrow$	**the** *SN* | **the** *PN*
SN VP	$\rightarrow$	*SN SV*
PN VP	$\rightarrow$	*PN PV*
SN $\rightarrow$	**child**	
PN $\rightarrow$	**children**	
SV $\rightarrow$	**plays**	
PV $\rightarrow$	**play**	

Notational Conventions

Symbol type	Convention
single terminal	letters `a, b, c, operators` delimiters, keywords
single nonterminal	letters A, B, C and names such as *declaration* , *list* and S is the start symbol
single grammar symbol (symbol from $\{N \cup T\}$)	X, Y, Z
string of terminals	letters `x , y , z`
string of grammar symbols	α, β, γ
null string	ε

Derivation as a relation

Consider the derivation:

$$
\begin{array}{ll}
 & \textit{declaration} \\
\Rightarrow & \textit{type}, \textit{idlist}; \\
\Rightarrow & \textbf{integer}\ \textit{idlist}; \\
\Rightarrow & \textbf{integer}\ \textit{idlist}, \textbf{id}; \\
\Rightarrow & \textbf{integer id}, \textbf{id};
\end{array}
$$

We would like to say:

$$
\begin{array}{l}
\textit{type idlist}; \Rightarrow \textbf{integer}\ \textit{idlist}, \textbf{id}; \\
\textit{type idlist}; \stackrel{*}{\Rightarrow} \textit{type idlist}; \\
\textit{type idlist}; \stackrel{*}{\Rightarrow} \textbf{integer id}, \textbf{id}; \\
\textit{type idlist}; \stackrel{+}{\Rightarrow} \textbf{integer id}, \textbf{id};
\end{array}
$$

Derivation as a relation

- $A \rightarrow \gamma$ – a production rule
- $\alpha\ A\ \beta$ – a string of grammar symbols
-
  - Replacing A in $\alpha A \beta$ yields $\alpha\gamma\beta$.
  - Formally, this is stated as $\alpha\ A\ \beta$ derives $\alpha\ \gamma\ \beta$ in one step.
  - Symbolically $\alpha\ A\ \beta \Rightarrow \alpha\ \gamma\ \beta$.
- $\alpha_1 \Rightarrow \alpha_2$ means α_1 derives α_2 in one step.
- $\alpha_1 \stackrel{*}{\Rightarrow} \alpha_2$ means α_1 derives α_2 in zero or more steps. Clearly $\alpha \stackrel{*}{\Rightarrow} \alpha$ is always true for any α.
- $\alpha_1 \stackrel{+}{\Rightarrow} \alpha_2$ means α_1 derives α_2 in one or more steps.

Sentential forms and sentences

- The *language* $L(G)$ generated by a grammar G is defined as $\{w \mid S \stackrel{+}{\Rightarrow} w, w \in T^*\}$.

  The language generated by the type declaration grammar is the set of strings consisting of:

  - A type name (**integer** or **float**), followed by
  - a **,** separated list of one or more **id**s, followed by
  - a **;**.

  Strings in $L(G)$ are called *sentences* of G.

Sentential forms and sentences

- A string α, $\alpha \in (N \bigcup T)^*$, such that $S \stackrel{*}{\Rightarrow} \alpha$, is called a *sentential form* of G.
  - *type idlist*;,
    integer *idlist*, **id;**, and
    integer id, id; are all sentential forms.

    However, only **integer id, id;** is a sentence.

Equivalent grammars

- Two grammars are *equivalent*,if they generate the same language.

- The grammars:

$$
\begin{array}{lcl}
\textit{declaration} & \rightarrow & \textit{type idlist };\\
\textit{idlist} & \rightarrow & \textbf{id} \mid \textit{idlist}\ ,\ \textbf{id}\\
\textit{type} & \rightarrow & \texttt{integer} \mid \texttt{float}
\end{array}
$$

and

$$
\begin{array}{lcl}
\textit{declaration} & \rightarrow & \textit{type idlist };\\
\textit{idlist} & \rightarrow & \textbf{id}\ \textit{commaidlist}\\
\textit{commaidlist} & \rightarrow & ,\ \textbf{id}\ \textit{commaidlist} \mid \varepsilon\\
\textit{type} & \rightarrow & \texttt{integer} \mid \texttt{float}
\end{array}
$$

are equivalent.

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathtt{id} \end{array}$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$\underline{E} \quad \stackrel{lm}{\Rightarrow} \quad \underline{E} + T$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \texttt{id} \end{array}$$

Leftmost derivation: Expand the leftmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\ & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{T} + T + T \end{array}$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathtt{id} \end{array}$$

Leftmost derivation: Expand the leftmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\ & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\ & \overset{lm}{\Rightarrow} & id + \underline{T} + T \end{array}$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T}
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{F}
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \stackrel{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\underline{E} \overset{rm}{\Rightarrow} E + \underline{T}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \stackrel{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \stackrel{rm}{\Rightarrow} & E + \underline{T} \\
 & \stackrel{rm}{\Rightarrow} & E + \underline{F}
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathtt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \texttt{id} \end{array}$$

Leftmost derivation: Expand the leftmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\ & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\ & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\ & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\ & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\ & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\ & \overset{lm}{\Rightarrow} & id + id + id \end{array}$$

Rightmost derivation: Expand the rightmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\ & \overset{rm}{\Rightarrow} & E + \underline{F} \\ & \overset{rm}{\Rightarrow} & \underline{E} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\ & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\ & \overset{rm}{\Rightarrow} & \underline{T} + id + id \end{array}$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Leftmost derivation: Expand the leftmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation: Expand the rightmost non-terminal.

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{T} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{F} + id + id
\end{array}
$$

Leftmost and rightmost derivations

- During a derivation, there is choice of non-terminals to expand at each sentential form.

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \mathtt{id} \end{array}$$

Leftmost derivation: Expand the leftmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\ & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\ & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\ & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\ & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\ & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\ & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\ & \overset{lm}{\Rightarrow} & id + id + id \end{array}$$

Rightmost derivation: Expand the rightmost non-terminal.

$$\begin{array}{lcl} \underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\ & \overset{rm}{\Rightarrow} & E + \underline{F} \\ & \overset{rm}{\Rightarrow} & \underline{E} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\ & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\ & \overset{rm}{\Rightarrow} & \underline{T} + id + id \\ & \overset{rm}{\Rightarrow} & \underline{F} + id + id \\ & \overset{rm}{\Rightarrow} & id + id + id \end{array}$$

Leftmost and rightmost derivations

- For constructing a derivation, there are choices at each sentential form.
  - choice of the non-terminal to be replaced
  - choice of a rule corresponding to the non-terminal.
- Instead of choosing the non-terminal to be replaced, in an arbitrary fashion, it is possible to make an uniform choice at each step.
  - *leftmost derivation:* replace the *leftmost non-terminal* in a sentential form
  - *leftmost derivation:* replace the *rightmost non-terminal* in a sentential form

Parse Trees

What is common to the leftmost derivation and the rightmost derivation shown before?

Leftmost derivation:

$$
\begin{array}{lll}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \overset{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Rightmost derivation:

$$
\begin{array}{lll}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{T} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{F} + id + id \\
 & \overset{rm}{\Rightarrow} & id + id + id
\end{array}
$$

Parse Trees

What is common to the leftmost derivation and the rightmost derivation shown before?

Leftmost derivation:

$$
\begin{aligned}
\underline{E} &\overset{lm}{\Rightarrow} \underline{E} + T \\
&\overset{lm}{\Rightarrow} \underline{E} + T + T \\
&\overset{lm}{\Rightarrow} \underline{T} + T + T \\
&\overset{lm}{\Rightarrow} \underline{F} + T + T \\
&\overset{lm}{\Rightarrow} id + \underline{T} + T \\
&\overset{lm}{\Rightarrow} id + \underline{F} + T \\
&\overset{lm}{\Rightarrow} id + id + \underline{T} \\
&\overset{lm}{\Rightarrow} id + id + \underline{F} \\
&\overset{lm}{\Rightarrow} id + id + id
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
\underline{E} &\overset{rm}{\Rightarrow} E + \underline{T} \\
&\overset{rm}{\Rightarrow} E + \underline{F} \\
&\overset{rm}{\Rightarrow} \underline{E} + id \\
&\overset{rm}{\Rightarrow} E + \underline{T} + id \\
&\overset{rm}{\Rightarrow} E + \underline{F} + id \\
&\overset{rm}{\Rightarrow} \underline{E} + id + id \\
&\overset{rm}{\Rightarrow} \underline{T} + id + id \\
&\overset{rm}{\Rightarrow} \underline{F} + id + id \\
&\overset{rm}{\Rightarrow} id + id + id
\end{aligned}
$$

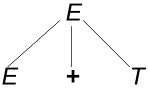
If a non-terminal A is replaced using a production $A \rightarrow \alpha$ in a left-sentential form, then A is also replaced by the same rule in a right-sentential form.

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

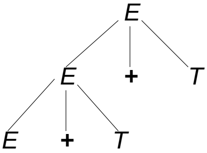


Leftmost derivation:

$$\underline{E} \quad \stackrel{lm}{\Rightarrow} \quad \underline{E} + T$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

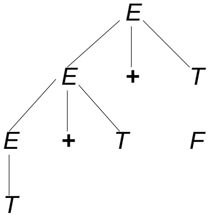


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

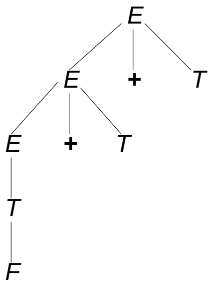


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

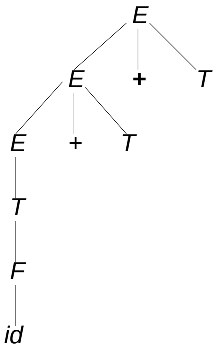


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

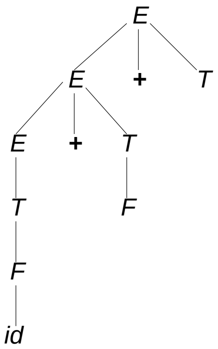


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

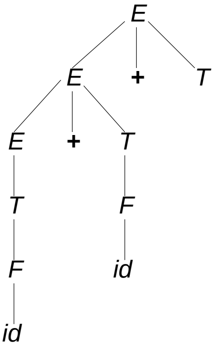


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

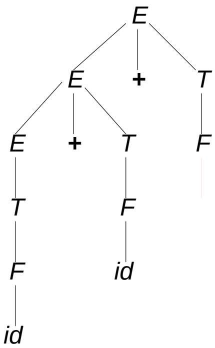


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{T}
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

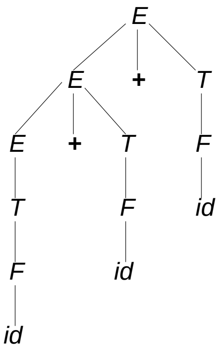


Leftmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{lm}{\Rightarrow} & \underline{E} + T \\
 & \overset{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \overset{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \overset{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \overset{lm}{\Rightarrow} & id + id + \underline{F}
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.



Leftmost derivation:

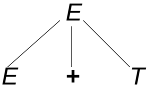
$$
\begin{array}{lcl}
\underline{E} & \stackrel{lm}{\Rightarrow} & \underline{E} + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{E} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{T} + T + T \\
 & \stackrel{lm}{\Rightarrow} & \underline{F} + T + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{T} + T \\
 & \stackrel{lm}{\Rightarrow} & id + \underline{F} + T \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{T} \\
 & \stackrel{lm}{\Rightarrow} & id + id + \underline{F} \\
 & \stackrel{lm}{\Rightarrow} & id + id + id
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

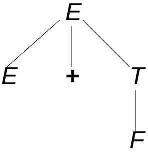


Rightmost derivation:

$$\underline{E} \quad \overset{rm}{\Rightarrow} \quad E + \underline{T}$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

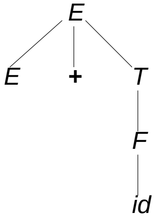


Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F}
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

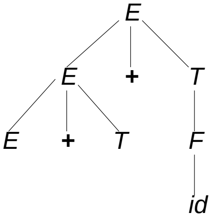


Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

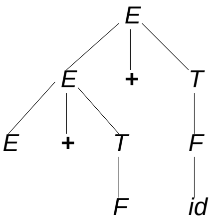


Rightmost derivation:

$$
\begin{aligned}
\underline{E} &\overset{rm}{\Rightarrow} E + \underline{T} \\
&\overset{rm}{\Rightarrow} E + \underline{F} \\
&\overset{rm}{\Rightarrow} \underline{E} + id \\
&\overset{rm}{\Rightarrow} E + \underline{T} + id
\end{aligned}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

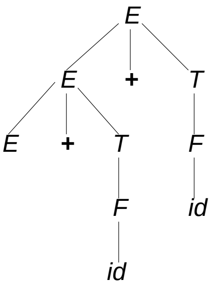


Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

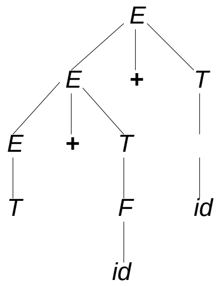


Rightmost derivation:

$$\begin{array}{lcl} \underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\ & \overset{rm}{\Rightarrow} & E + \underline{F} \\ & \overset{rm}{\Rightarrow} & \underline{E} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\ & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\ & \overset{rm}{\Rightarrow} & \underline{E} + id + id \end{array}$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

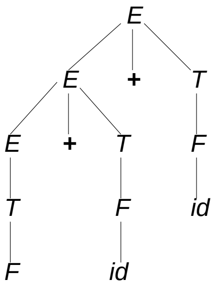


Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{T} + id + id
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.

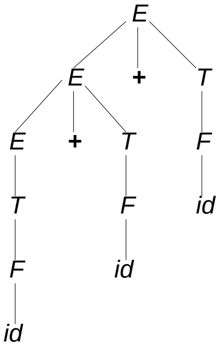


Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E + \underline{T} \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{T} + id \\
 & \overset{rm}{\Rightarrow} & E + \underline{F} + id \\
 & \overset{rm}{\Rightarrow} & \underline{E} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{T} + id + id \\
 & \overset{rm}{\Rightarrow} & \underline{F} + id + id
\end{array}
$$

Parse Trees

The commonality of the two derivations is expressed as a parse tree.



Rightmost derivation:

$$
\begin{array}{lcl}
\underline{E} & \overset{rm}{\Rightarrow} & E+\underline{T} \\
 & \overset{rm}{\Rightarrow} & E+\underline{F} \\
 & \overset{rm}{\Rightarrow} & \underline{E}+id \\
 & \overset{rm}{\Rightarrow} & E+\underline{T}+id \\
 & \overset{rm}{\Rightarrow} & E+\underline{F}+id \\
 & \overset{rm}{\Rightarrow} & \underline{E}+id+id \\
 & \overset{rm}{\Rightarrow} & \underline{T}+id+id \\
 & \overset{rm}{\Rightarrow} & \underline{F}+id+id \\
 & \overset{rm}{\Rightarrow} & id+id+id
\end{array}
$$

Parse Trees

A *parse tree* is a pictorial form of depicting a derivation.

1. root of the tree is labeled with S
2. each leaf node is labeled by a token or by ϵ
3. an internal node of the tree is labeled by a nonterminal
4. if an internal node has A as its label and the children of this node from left to right are labeled with $X_1, X_2, \ldots, X_n$ then there must be a production

$$A \rightarrow X_1 X_2 \ldots X_n$$

where X_i is a grammar symbol.

Derivations and Parse Trees

The following summarize some interesting relations between the two concepts

- Parse tree filters out the choice of replacements made in the sentential forms.

Derivations and Parse Trees

The following summarize some interesting relations between the two concepts

- Parse tree filters out the choice of replacements made in the sentential forms.
- Given a left (right) derivation for a sentence, one can construct a unique parse tree for the sentence.

Derivations and Parse Trees

The following summarize some interesting relations between the two concepts

- Parse tree filters out the choice of replacements made in the sentential forms.
- Given a left (right) derivation for a sentence, one can construct a unique parse tree for the sentence.
- For every parse tree for a sentence there is a unique leftmost and a unique rightmost derivation.

Derivations and Parse Trees

The following summarize some interesting relations between the two concepts

- Parse tree filters out the choice of replacements made in the sentential forms.
- Given a left (right) derivation for a sentence, one can construct a unique parse tree for the sentence.
- For every parse tree for a sentence there is a unique leftmost and a unique rightmost derivation.
- *Can a sentence have more than one distinct parse trees, and therefore more than one left (right) derivations?*

Ambiguous Grammars

Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Ambiguous Grammars

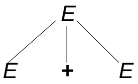
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 1:



Leftmost derivation 1:

$$\underline{E} \overset{lm}{\Rightarrow} \underline{E} + E$$

Ambiguous Grammars

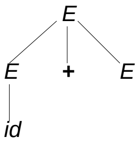
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 1:



Leftmost derivation 1:

$$\begin{aligned} \underline{E} &\stackrel{lm}{\Rightarrow} \underline{E} + E \\ &\stackrel{lm}{\Rightarrow} id + \underline{E} \end{aligned}$$

Ambiguous Grammars

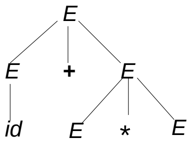
Consider the grammar:

$$E \rightarrow E + E \;\mid\; E * E \;\mid\; id$$

And consider the sentence:

$$id + id * id$$

Parse tree 1:



Leftmost derivation 1:

$$\begin{aligned} \underline{E} &\overset{lm}{\Rightarrow} \underline{E} + E \\ &\overset{lm}{\Rightarrow} id + \underline{E} \\ &\overset{lm}{\Rightarrow} id + \underline{E} * E \end{aligned}$$

Ambiguous Grammars

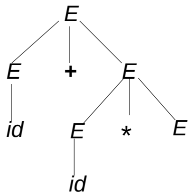
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 1:



Leftmost derivation 1:

$$\begin{aligned}
\underline{E} &\overset{lm}{\Rightarrow} \underline{E} + E \\
&\overset{lm}{\Rightarrow} id + \underline{E} \\
&\overset{lm}{\Rightarrow} id + \underline{E} * E \\
&\overset{lm}{\Rightarrow} id + id * \underline{E}
\end{aligned}$$

Ambiguous Grammars

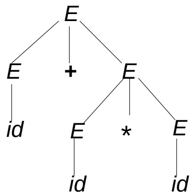
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 1:



Leftmost derivation 1:

$$
\begin{aligned}
\underline{E} &\overset{lm}{\Rightarrow} \underline{E} + E \\
&\overset{lm}{\Rightarrow} id + \underline{E} \\
&\overset{lm}{\Rightarrow} id + \underline{E} * E \\
&\overset{lm}{\Rightarrow} id + id * \underline{E} \\
&\overset{lm}{\Rightarrow} id + id * id
\end{aligned}
$$

Ambiguous Grammars

Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Ambiguous Grammars

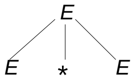
Consider the grammar:

$$E \rightarrow E + E \;\mid\; E * E \;\mid\; id$$

And consider the sentence:

$$id + id * id$$

Parse tree 2:



Leftmost derivation 2:

$$\underline{E} \quad \overset{lm}{\Rightarrow} \quad \underline{E} * E$$

Ambiguous Grammars

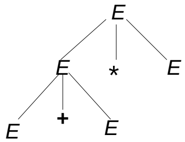
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 2:



Leftmost derivation 2:

$$\begin{aligned} \underline{E} \quad & \overset{lm}{\Rightarrow} \quad \underline{E} * E \\ & \overset{lm}{\Rightarrow} \quad \underline{E} + E * E \end{aligned}$$

Ambiguous Grammars

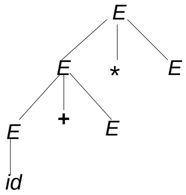
Consider the grammar:

$$E \rightarrow E + E \;\mid\; E * E \;\mid\; id$$

And consider the sentence:

$$id + id * id$$

Parse tree 2:



Leftmost derivation 2:

$$\begin{array}{lcl} \underline{E} & \overset{lm}{\Rightarrow} & \underline{E} * E \\ & \overset{lm}{\Rightarrow} & \underline{E} + E * E \\ & \overset{lm}{\Rightarrow} & id + \underline{E} * E \end{array}$$

Ambiguous Grammars

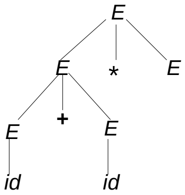
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 2:



Leftmost derivation 2:

$$\begin{aligned}
\underline{E} &\overset{lm}{\Rightarrow} \underline{E} * E \\
&\overset{lm}{\Rightarrow} \underline{E} + E * E \\
&\overset{lm}{\Rightarrow} id + \underline{E} * E \\
&\overset{lm}{\Rightarrow} id + id * \underline{E}
\end{aligned}$$

Ambiguous Grammars

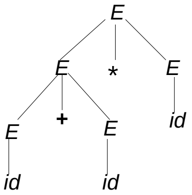
Consider the grammar:

$$E \rightarrow E + E \mid E * E \mid id$$

And consider the sentence:

$$id + id * id$$

Parse tree 2:



Leftmost derivation 2:

$$\begin{aligned}
\underline{E} &\overset{lm}{\Rightarrow} \underline{E} * E \\
&\overset{lm}{\Rightarrow} \underline{E} + E * E \\
&\overset{lm}{\Rightarrow} id + \underline{E} * E \\
&\overset{lm}{\Rightarrow} id + id * \underline{E} \\
&\overset{lm}{\Rightarrow} id + id * id
\end{aligned}$$

There are two parse trees and two leftmost derivations for the sentence.

Ambiguous Grammars

A grammar is *ambiguous*, if there is a sentence for which there are

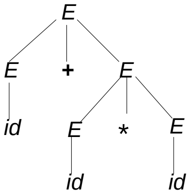
- more than one parse tress, or equivalently
- more than one leftmost derivations, or equivalently
- more than one rightmost derivations.

Ambiguous Grammars

Why is ambiguity an issue?

For the expression grammar, the parse tree represent an implicit parenthesizing of the sentence.

Parse tree 1:



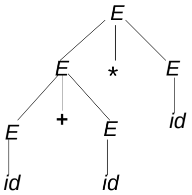
$$\Longrightarrow id + (id * id)$$

Ambiguous Grammars

Why is ambiguity an issue?

For the expression grammar, the parse tree represent an implicit parenthesizing of the sentence.

Parse tree 2:



$\Longrightarrow (id + id) * id$

And the meanings of the expressions $id + (id * id)$ and $(id + id) * id$ are not the same.

Ambiguous Grammars – A second example

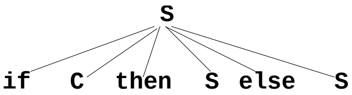
Example:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S \\
S & \rightarrow & \texttt{ass}
\end{array}
$$

Consider the sentence:

$\texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \texttt{ass}\ \texttt{else}\ \texttt{ass}$

First parse tree:



First rightmost derivation:

$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ \underline{S}$

Ambiguous Grammars – A second example

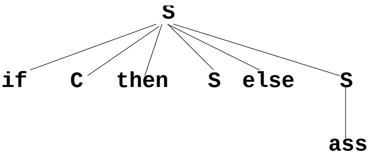
Example:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if } C \texttt{ then } S \texttt{ else } S \\
S & \rightarrow & \texttt{if } C \texttt{ then } S \\
S & \rightarrow & \texttt{ass}
\end{array}
$$

Consider the sentence:

`if` C `then if` C `then ass else ass`

First parse tree:



First rightmost derivation:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if } C \texttt{ then } S \texttt{ else } \underline{S} \\
S & \rightarrow & \texttt{if } C \texttt{ then } \underline{S} \texttt{ else ass}
\end{array}
$$

Ambiguous Grammars – A second example

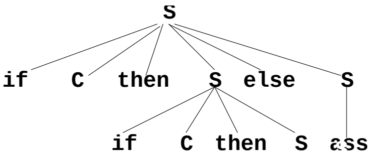
Example:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S \\
S & \rightarrow & \texttt{ass}
\end{array}
$$

Consider the sentence:

$\texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \texttt{ass}\ \texttt{else}\ \texttt{ass}$

First parse tree:



First rightmost derivation:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ \underline{S} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ \texttt{ass} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ \texttt{C}\ \texttt{then}\ \underline{\texttt{S}}\ \texttt{else}\ \texttt{ass}
\end{array}
$$

Ambiguous Grammars – A second example

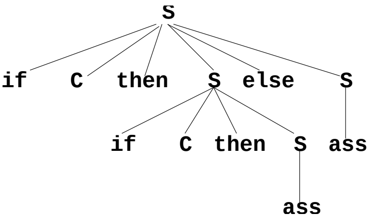
Example:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S \\
S & \rightarrow & \texttt{ass}
\end{array}
$$

Consider the sentence:

$\texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \texttt{ass}\ \texttt{else}\ \texttt{ass}$

First parse tree:



First rightmost derivation:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ \underline{S} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ \texttt{ass} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ \ \texttt{C}\ \ \texttt{then}\ \ \underline{\texttt{S}}\ \texttt{else}\ \texttt{ass} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \texttt{ass}\ \texttt{else}\ \texttt{ass}
\end{array}
$$

Ambiguous Grammars

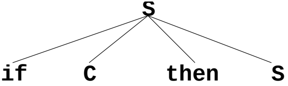
Example:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S$$
$$S \rightarrow \texttt{ass}$$

Consider the sentence:

`if` C `then if C then ass else ass`

The second parse tree:



The second rightmost derivation:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \underline{S}$$

Ambiguous Grammars

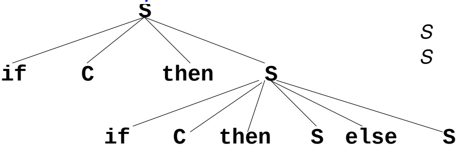
Example:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ S \\
S & \rightarrow & \texttt{ass}
\end{array}
$$

Consider the sentence:

`if` C `then if  C then  ass else  ass`

The second parse tree:



The second rightmost derivation:

$$
\begin{array}{lcl}
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \underline{S} \\
S & \rightarrow & \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ S
\end{array}
$$

Ambiguous Grammars

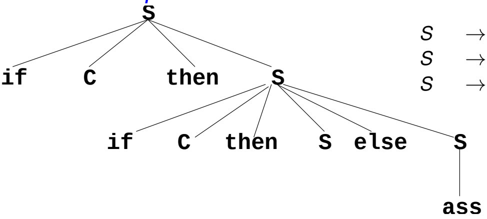
Example:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S$$
$$S \rightarrow \texttt{ass}$$

Consider the sentence:

$\texttt{if}\ C\ \texttt{then}\ \texttt{if C then ass else ass}$

The second parse tree:



The second rightmost derivation:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \underline{S}$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ S$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ \texttt{ass}$$

Ambiguous Grammars

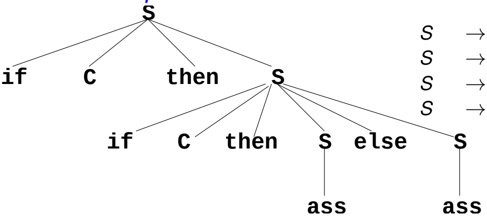
Example:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S\ \texttt{else}\ S$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ S$$
$$S \rightarrow \texttt{ass}$$

Consider the sentence:

$\texttt{if}\ C\ \texttt{then}\ \texttt{if}\ \ \texttt{C}\ \texttt{then}\ \ \texttt{ass}\ \texttt{else}\ \ \texttt{ass}$

The second parse tree:



The second rightmost derivation:

$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \underline{S}$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ S$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \underline{S}\ \texttt{else}\ \texttt{ass}$$
$$S \rightarrow \texttt{if}\ C\ \texttt{then}\ \texttt{if}\ C\ \texttt{then}\ \texttt{ass}\ \texttt{else}\ \texttt{ass}$$

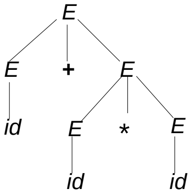
Disambiguation

How does one disambiguate to obtain a single parse tree for a sentence?

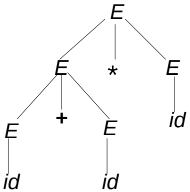
- *Disambiguate during parsing:* Disambiguation rules are incorporated into a parser to choose between possible parse trees.
  - Makes a choice during parse tree construction.
  - Yacc has provisions for such disambiguation.
- *Disambiguate the grammar:* Rewrite the grammar.

Disambiguation by grammar rewriting

- Decide on general rules to choose one of many possible parse trees. As example, choose



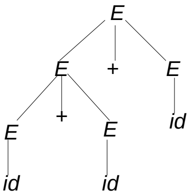
over



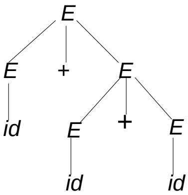
This amounts to giving a higher precedence to $*$ over $+$.

Disambiguation by grammar rewriting

- Similarly, choose:



over



This amounts to saying that $+$ is left associative.

Disambiguation by grammar rewriting

- Consider a sentence $a + b * c * d + d * e$. Denote as T the sub-expressions consisting of products of *id*s or a single *id*.

  Then the expression can be re-written as $T + T + T$

- Because $+$ is left associative, the expression above should be parsed as $(T + T) + T$.

  A grammar which does this is:
  $E \rightarrow E + T \mid T$

Disambiguation by grammar rewriting

- Let F denote either a single *id* or a (E). Then the strings represented by T can be written as $F * F * F$ or a single F.
- A grammar which generates such strings, taking into account the associativity of $*$ is:
  $T \rightarrow T * F \mid F$
- Finally we also have
  $F \rightarrow (E) \mid id$

Disambiguation by grammar rewriting

- Now consider disambiguation of the grammar:

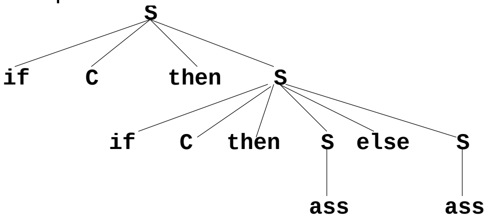
$$S \rightarrow \texttt{if } C \texttt{ then } S \texttt{ else } S$$
$$S \rightarrow \texttt{if } C \texttt{ then } S$$
$$S \rightarrow \texttt{ass}$$

  and the sentence

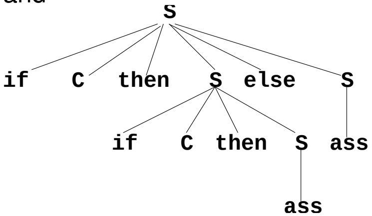
  $\texttt{if } C \texttt{ then if C then ass else ass}$

Disambiguation by grammar rewriting

- The parse trees are:



and



- We choose the first parse tree over the second on the basis of the following rule:

Disambiguation

In other words:

If a then and a else are derived from the same production, then the parse tree between them should have matching then and else.

The following grammar captures this idea:

$$
\begin{aligned}
\mathit{stmt} &\rightarrow \mathit{matched\_stmt} \mid \mathit{unmatched\_stmt} \\
\mathit{matched\_stmt} &\rightarrow \texttt{if}\ C\ \texttt{then}\ \mathit{matched\_stmt}\ \texttt{else}\ \mathit{matched\_stmt} \\
&\mid \texttt{ass} \\
\mathit{unmatched\_stmt} &\rightarrow \texttt{if}\ C\ \texttt{then}\ \mathit{stmt} \\
&\mid \texttt{if}\ C\ \texttt{then}\ \mathit{matched\_stmt}\ \texttt{else}\ \mathit{unmatched\_stmt}
\end{aligned}
$$

Introduction to Parsing

A *parser* for a context free grammar G is a program P that given an input w,

- either verifies that w is a sentence of G and, additionally, may also give the parse tree for w.
- or gives an error message stating that w is not a sentence. May provide some information to locate the error.

Parsing Strategies

Two ways of creating a parse tree:

- *Top-down parsers* – Created from the root down to leaves.
- *Bottom-up parsers* – Created from leaves upwards to the root.

Both the parsing strategies can also be rephrased in terms of derivations.

Example of Bottom Up Parsing

Grammar:

$$
\begin{aligned}
D &\rightarrow \texttt{var}\ \textit{list} : \textit{type}\ ; \\
\textit{type} &\rightarrow \texttt{integer} \mid \texttt{float} \\
\textit{list} &\rightarrow \textit{list}, \texttt{id} \mid \texttt{id}
\end{aligned}
$$

Input string: `var id,id :  integer;`

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ \mathit{list} : \mathit{type}\ ;$$
$$\mathit{type} \rightarrow \texttt{integer} \mid \texttt{float}$$
$$\mathit{list} \rightarrow \mathit{list}, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ list : type\ ;$$
$$type \rightarrow \texttt{integer} \mid \texttt{float}$$
$$list \rightarrow list, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

  `var id, id : integer ;`

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ \mathit{list} : \mathit{type}\ ;$$
$$\mathit{type} \rightarrow \texttt{integer} \mid \texttt{float}$$
$$\mathit{list} \rightarrow \mathit{list}, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

$$\begin{array}{ll} & \texttt{var id, id : integer ;} \\ \Rightarrow & \texttt{var}\ \mathit{list}\ \texttt{, id : integer ;} \end{array}$$

Example of Bottom Up Parsing

Grammar:

$$
\begin{aligned}
D &\rightarrow \texttt{var}\ \mathit{list} : \mathit{type}\ ; \\
\mathit{type} &\rightarrow \texttt{integer} \mid \texttt{float} \\
\mathit{list} &\rightarrow \mathit{list}, \texttt{id} \mid \texttt{id}
\end{aligned}
$$

Input string: `var id,id :  integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

$$
\begin{aligned}
& \texttt{var id, id : integer ;} \\
\Rightarrow\ & \texttt{var}\ \mathit{list}\ \texttt{, id : integer ;} \\
\Rightarrow\ & \texttt{var}\ \mathit{list} : \texttt{integer;}
\end{aligned}
$$

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ \mathit{list} : \mathit{type}\ ;$$
$$\mathit{type} \rightarrow \texttt{integer} \mid \texttt{float}$$
$$\mathit{list} \rightarrow \mathit{list}, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

$$\begin{array}{ll} & \texttt{var id, id : integer ;} \\ \Rightarrow & \texttt{var}\ \mathit{list}\ \texttt{, id : integer ;} \\ \Rightarrow & \texttt{var}\ \mathit{list} : \texttt{integer;} \\ \Rightarrow & \texttt{var}\ \mathit{list} : \mathit{type}\ ; \end{array}$$

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ \mathit{list} : \mathit{type}\ ;$$
$$\mathit{type} \rightarrow \texttt{integer} \mid \texttt{float}$$
$$\mathit{list} \rightarrow \mathit{list}, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

$$\begin{array}{ll} & \texttt{var id, id : integer ;} \\ \Rightarrow & \texttt{var}\ \mathit{list}\ \texttt{, id : integer ;} \\ \Rightarrow & \texttt{var}\ \mathit{list} : \texttt{integer;} \\ \Rightarrow & \texttt{var}\ \mathit{list} : \mathit{type}\ ; \\ \Rightarrow & D \end{array}$$

Example of Bottom Up Parsing

Grammar:

$$D \rightarrow \texttt{var}\ \textit{list} : \textit{type}\ ;$$
$$\textit{type} \rightarrow \texttt{integer} \mid \texttt{float}$$
$$\textit{list} \rightarrow \textit{list}, \texttt{id} \mid \texttt{id}$$

Input string: `var id,id : integer;`

- The bottom-up parse and the sentential forms produced during the parse are:

$$
\begin{aligned}
& \texttt{var}\ \texttt{id},\ \texttt{id} :\ \texttt{integer}\ ; \\
\Rightarrow\ & \texttt{var}\ \textit{list}\ ,\ \texttt{id} :\ \texttt{integer}\ ; \\
\Rightarrow\ & \texttt{var}\ \textit{list} : \texttt{integer}; \\
\Rightarrow\ & \texttt{var}\ \textit{list} : \textit{type}\ ; \\
\Rightarrow\ & D
\end{aligned}
$$

- The sentential forms happen to be a *right most derivation in the reverse order.*

Example of Bottom Up Parsing

Here is bottom up parsing, viewed in terms of parse tree construction:

var id , id : integer

Example of Bottom Up Parsing

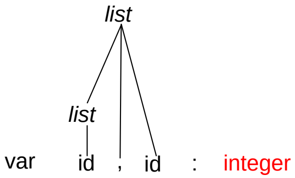
Here is bottom up parsing, viewed in terms of parse tree construction:

list

var id , id : integer

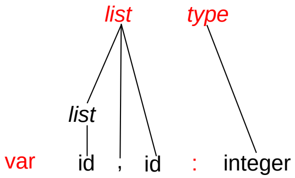
Example of Bottom Up Parsing

Here is bottom up parsing, viewed in terms of parse tree construction:



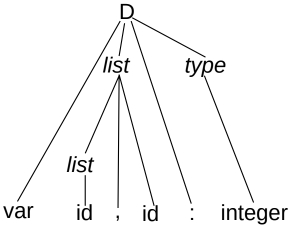
Example of Bottom Up Parsing

Here is bottom up parsing, viewed in terms of parse tree construction:



Example of Bottom Up Parsing

Here is bottom up parsing, viewed in terms of parse tree construction:



Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

$\texttt{var}\ \underline{\texttt{id}}\ \texttt{,}\ \texttt{id}\ \texttt{:}\ \texttt{integer}\ \texttt{;}$

Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

```
   var id , id :  integer ;
⇒  var list , id :  integer ;
```

Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

$$
\begin{array}{ll}
 & \texttt{var}\ \underline{\texttt{id}}\ ,\ \texttt{id}\ :\ \ \texttt{integer}\ ; \\
\Rightarrow & \texttt{var}\ \underline{\textit{list}\ ,\ \texttt{id}}\ :\ \ \texttt{integer}\ ; \\
\Rightarrow & \texttt{var}\ \textit{list}\ :\ \ \underline{\texttt{integer}}\ ;
\end{array}
$$

Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

$$
\begin{array}{ll}
 & \texttt{var}\ \underline{\texttt{id}}\ \texttt{,}\ \texttt{id}\ \texttt{:}\ \ \texttt{integer}\ \texttt{;} \\
\Rightarrow & \texttt{var}\ \underline{\textit{list}\ \texttt{,}\ \texttt{id}}\ \texttt{:}\ \ \texttt{integer}\ \texttt{;} \\
\Rightarrow & \texttt{var}\ \textit{list}\ \texttt{:}\ \ \underline{\texttt{integer}}\ \texttt{;} \\
\Rightarrow & \underline{\texttt{var}\ \textit{list}\ \texttt{:}\ \textit{type}}\ \texttt{;}
\end{array}
$$

Principles of Bottom Up Parsing - Handles

The basic steps of a bottom-up parser are

1. to identify a *substring* within a *rightmost sentential* form which matches the rhs of a rule.
2. when this substring is replaced by the lhs of the matching rule, it must produce the previous rm-sentential form.

Such a substring is called a *handle* .

$$
\begin{array}{ll}
 & \texttt{var}\ \underline{\texttt{id}}\ \texttt{,}\ \texttt{id}\ \texttt{:}\ \ \texttt{integer}\ \texttt{;} \\
\Rightarrow & \texttt{var}\ \underline{\mathit{list}\ \texttt{,}\ \texttt{id}}\ \texttt{:}\ \ \texttt{integer}\ \texttt{;} \\
\Rightarrow & \texttt{var}\ \mathit{list}\ \texttt{:}\ \ \underline{\texttt{integer}}\ \texttt{;} \\
\Rightarrow & \underline{\texttt{var}\ \mathit{list} : \mathit{type}}\ ; \\
\Rightarrow & D
\end{array}
$$

Handle – Definition

A *handle* of a right sentential form γ, is

- a production rule $A \rightarrow \beta$, and
- an occurrence of a sub-string β in γ

such that when the occurrence of β is replaced by A in γ, we get the previous right sentential form in a rightmost derivation of γ.

Handle – Definition

A *handle* of a right sentential form γ, is

- a production rule $A \rightarrow \beta$, and
- an occurrence of a sub-string β in γ

such that when the occurrence of β is replaced by A in γ, we get the previous right sentential form in a rightmost derivation of γ.

Formally, if

$$S \overset{*rm}{\Rightarrow} \alpha\, A\, \mathtt{w} \overset{rm}{\Rightarrow} \alpha\, \beta\, \mathtt{w}$$

then the rule $A \rightarrow \beta$ and the occurrence β is the handle in $\alpha\, \beta\mathtt{w}$.

Handle – Definition

A *handle* of a right sentential form γ, is

- a production rule $A \rightarrow \beta$, and
- an occurrence of a sub-string β in γ

such that when the occurrence of β is replaced by A in γ, we get the previous right sentential form in a rightmost derivation of γ.

Formally, if

$$S \overset{*rm}{\Rightarrow} \alpha\, A\, \mathtt{w} \overset{rm}{\Rightarrow} \alpha\, \beta\, \mathtt{w}$$

then the rule $A \rightarrow \beta$ and the occurrence β is the handle in $\alpha\, \beta \mathtt{w}$.

Only terminal symbols can appear to the right of a handle in a rightmost sentential form. Why?

Handles

- *Bottom up parsing is essentially the process of detecting handles and reducing them.*
- *Different bottom-up parsers differ in the way they detect handles.*

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift -

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`
`$-id`	`** id / id $`	reduce by $B \rightarrow$ `id`

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`
`$-id`	`** id / id $`	reduce by $B \rightarrow$ `id`
`$-`B	`** id / id $`	reduce by $P \rightarrow B$

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`
`$-id`	`** id / id $`	reduce by $B \rightarrow \texttt{id}$
`$-`B	`** id / id $`	reduce by $P \rightarrow B$
`$-`P	`** id / id $`	reduce by $P \rightarrow -P$

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`
`$-id`	`** id / id $`	reduce by $B \rightarrow$ `id`
`$-`B	`** id / id $`	reduce by $P \rightarrow B$
`$-`P	`** id / id $`	reduce by $P \rightarrow -P$
`$`P	`** id / id $`	shift `**`

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E+T \mid E-T \mid T \\
T & \rightarrow & T*F \mid T/F \mid F \\
F & \rightarrow & P**F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

stack	*input*	*parser move*
`$`	`-id ** id / id $`	shift `-`
`$-`	`id ** id / id $`	shift `id`
`$-id`	`** id / id $`	reduce by $B \rightarrow$ `id`
`$-`B	`** id / id $`	reduce by $P \rightarrow B$
`$-`P	`** id / id $`	reduce by $P \rightarrow -P$
`$`P	`** id / id $`	shift `**`
…	…	…
`$`E	`$`	*accept*

Shift-Reduce Parsers

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & P ** F \mid P \\
P & \rightarrow & -P \mid B \\
B & \rightarrow & (E) \mid \texttt{id}
\end{array}
$$

Note that the contents of the stack appended to the input constitutes a right-sentential form

Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.

Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.
2. a data structure for detecting handles (a stack happens to be adequate)

Shift-Reduce Parsers

Shift-reduce parsers require the following data structures.

1. a buffer for holding the input string to be parsed.
2. a data structure for detecting handles (a stack happens to be adequate)
3. a data structure for storing and accessing the lhs and rhs of rules.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

- **Shift:** Moving a single token from the input buffer onto the stack till a handle appears on the stack.
- **Reduce:** When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

- Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.
- Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.
- Accept: When the stack contains only the start symbol and input buffer is empty, the parser halts announcing a *successful* parse.

Shift-Reduce Parsers

Basic actions of the shift-reduce parser are:

- Shift: Moving a single token from the input buffer onto the stack till a handle appears on the stack.
- Reduce: When a handle appears on the stack, it is popped and replaced by the left hand side of the corresponding production.
- Accept: When the stack contains only the start symbol and input buffer is empty, the parser halts announcing a *successful* parse.
- Error: When the parser can neither shift nor reduce nor accept. Halts announcing an error.

Properties of shift-reduce parsers

Is the following situation possible?

- $\alpha\, \beta\, \gamma$ is the stack contents and $A \rightarrow \gamma$ is the handle.
- The stack contents reduces to $\alpha\, \beta\, A$
- Now $B \rightarrow \beta$ is the next handle.

Implication: The handle is buried in the stack. The search for the handle can be expensive.

Properties of shift-reduce parsers

Is the following situation possible?

- $\alpha\,\beta\,\gamma$ is the stack contents and $A \rightarrow \gamma$ is the handle.
- The stack contents reduces to $\alpha\,\beta\,A$
- Now $B \rightarrow \beta$ is the next handle.

Implication: The handle is buried in the stack. The search for the handle can be expensive.

Assume that this is true. Then, by the definition of a handle, there is a sequence of rightmost derivations:

$$S \stackrel{*rm}{\Rightarrow} \alpha BAxyz \stackrel{rm}{\Rightarrow} \alpha\beta Axyz \stackrel{rm}{\Rightarrow} \alpha\beta\gamma xyz$$

But in the right sentential form $\alpha BAxyz$, B is not the rightmost non-terminal, and thus $\stackrel{rm}{\Rightarrow}$ is not a rightmost derivation. Therefore the above scenario is not possible.

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$\alpha\beta\gamma xyz$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\alpha\beta A\ \ xyz$$
$$\Downarrow_{rm}$$
$$\alpha\beta\gamma xyz$$

Production used is $A \rightarrow \gamma$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\alpha C \;\; xyz \;\overset{rm}{\Rightarrow}\; \alpha\beta A \;\; xyz \;\overset{rm}{\Rightarrow}\; \alpha\beta\gamma xyz$$

Production used is $C \rightarrow \beta A$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd} \alpha C\ \ xyz \arrow[dr, Rightarrow, "rm"] & \alpha D\ \ yz \arrow[d, Rightarrow, "rm"] \\ & \alpha\beta A\ \ xyz \arrow[d, Rightarrow, "rm"] \\ & \alpha\beta\gamma xyz \end{tikzcd}$$

Production used is $D \rightarrow \beta A x$

Properties of shift-reduce parsers

So what scenarios are possible after a reduction?

$$\begin{tikzcd} \alpha C\ \ xyz \arrow[dr, Rightarrow, "rm"] & \alpha D\ \ yz \arrow[d, Rightarrow, "rm"] & \alpha\beta AxE\ \ z \arrow[dl, Rightarrow, "rm"'] \\ & \alpha\beta A\ \ xyz \arrow[d, Rightarrow, "rm"] & \\ & \alpha\beta\gamma xyz & \end{tikzcd}$$

Production used is $E \rightarrow y$

Example of Shift-Reduce Parsing

Conflicts in a Shift-Reduce Parser

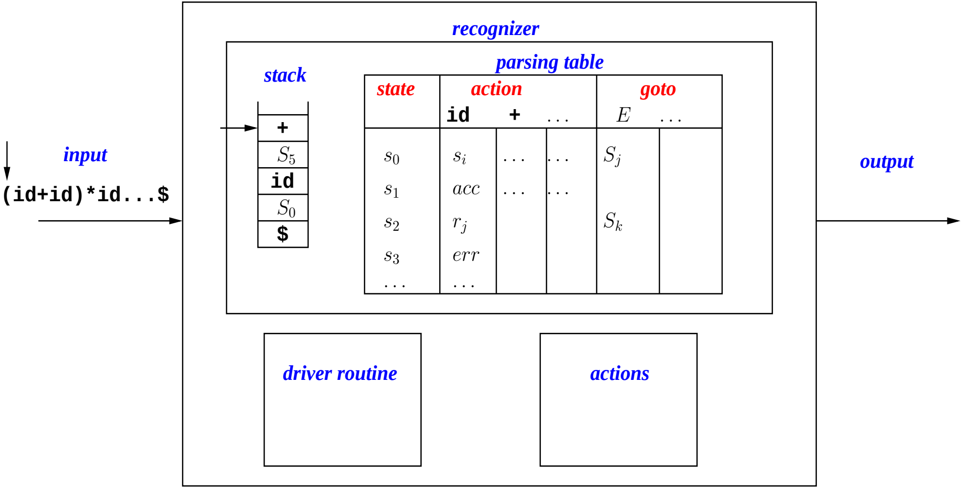
For some grammars, the shift-reduce parser may get into the following conflicting situations.

- *Shift-reduce conflict* A handle β occurs at *tos*; the *nexttoken* $\mathtt{a}$ is such that $\beta \mathtt{a} \gamma$ happens to be another handle. The parser has two options
  - reduce the handle using $A \rightarrow \beta$
  - ignore the handle β; shift $\mathtt{a}$ and continue parsing and eventually reduce using $B \rightarrow \beta \mathtt{a} \gamma$.
- *Reduce-reduce conflict* the stack contents are $\alpha\beta\gamma$ and both $\beta\gamma$ and γ are handles with $A \rightarrow \beta\gamma$ and $B \rightarrow \gamma$ as the corresponding rules. Then the parser has two reduce possibilities.

To handle such conflicts, the nexttoken could be used to prefer one move over the other.

- choose shift (or reduce) in a shift-reduce conflict
- prefer one reduce (over others) in a reduce-reduce conflict.

LR Parser Model



LR Parsers

Consist of

- a stack which contains strings of the form $s_0 X_1 s_1 X_2 \ldots X_m s_m$, where X_i is a grammar symbol and s_i is a special symbol called a *state*.
- a parsing table which comprises two parts, usually named as *Action* and *Goto*.

The entries in the Action part are:

- s_i which means shift to state i
- r_j which stands for reduce by the j^{th} rule,
- *accept*
- *error*

The Goto part contains blank entries or state symbols.

The Driver Routine

- Initializes stack with *start* state. Calls scanner to get next token.
- Consults the parsing table and performs the action specified there.
- Parsing continues till either an error or accept entry is encountered.

top of stack	*nexttoken*	*action*	*parsing action*
state j	a	si	push a; push state i
	a	rj	$rj : A \rightarrow \alpha$; $length(\alpha) = r$; pop $2r$ symbols from stack; top of stack contains state k; goto[k, a] = cl; push A; push state l;
state j	\$	acc	successful parse; halt
state j	a	err	error handling

SLR(1) Parser

1.	E	$\rightarrow$	$E + T$	2.	E	$\rightarrow$	T
3.	T	$\rightarrow$	$T * F$	4.	T	$\rightarrow$	F
5.	F	$\rightarrow$	(E)	6.	F	$\rightarrow$	id

	action						*goto*		
state	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			c1	c2	c3
1		s6	*			acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			c8	c2	c3
5		r6	r6		r6	r6			
6	s5			s4				c9	c3
7	s5			s4					c10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Configuration of a LR Parser

- A *configuration* of a LR parser is defined by a tuple, (stack contents , unexpended part of input).
- Initial configuration is $(s_0, a_1 a_2 \ldots a_n\$)$, where s_0 is a designated *start* state and the second component is the entire sentence to be parsed.
- Let an intermediate configuration be given by $(s_0 X_1 s_1 \ldots X_i s_i$, $a_j a_{j+1} \ldots a_n\$)$, then resulting configuration
  i) after a shift action is given by $(s_0 X_1 s_1 \ldots X_i s_i a_j s_k, a_{j+1} \ldots a_n\$)$ provided $Action[s_i, a_j] = s_k$; both *stack* and *nexttoken* change after a shift.
  ii) after a reduce action is given by $(s_0 X_1 s_1 \ldots X_{i-r} As, a_j \ldots a_n\$)$ where $Action[s_i, a_j] = r_l$; rule p_l : $A \rightarrow \beta$, β has r grammar symbols and $goto(s_{i-r}, A) = s$. Only the *stack* changes here.

\$

LR(0) Items

- *LR(0) item* : An LR(0) item for a grammar G is a production rule of G with the symbol $\bullet$ (read as *dot* or *bullet*) inserted at some position in the rhs of the rule.
- Example of LR(0) items : For the rule given below

$$decls \rightarrow decls\ decl$$

the possible LR(0) items are :

$$I_1 : decls \rightarrow \bullet decls\ decl$$
$$I_2 : decls \rightarrow decls\ \bullet decl$$
$$I_3 : decls \rightarrow decls\ decl\bullet$$

The rule $decls \rightarrow \ \varepsilon$ has only one LR(0) item,

$$I_4 : decls \rightarrow \ \bullet$$

LR(0) Items

- An LR(0) item is *complete* if the $\bullet$ is the last symbol in the rhs.
  Example : I_3 and I_4 are complete items and I_1 and I_2 are incomplete items.
- An LR(0) item is called a *kernel item*, if the dot is not at the left end. However the item $S' \rightarrow \bullet S$ is an exception and is defined to be a kernel item.
  Example : I_1, I_2, I_3 are all kernel items and I_4 is a non-kernel item.

Canonical Collection of LR(0) Items

The construction of the SLR parsing table requires two functions *closure* and *goto*.

closure: Let U be the collection of all LR(0) items of a cfg G. Then $closure : U \to 2^U$.

1. $closure(I) = \{I\},\ for\ I \in U$
2. If $A \to \alpha \bullet B\beta \in closure(I)$, then the item $B \to \bullet\eta$ is added to $closure(I)$.
3. Apply step (ii) above repeatedly till no more new items can be added to $closure(I)$.

Example: Consider the grammar $A \to A\ \mathtt{a} \mid \mathtt{b}$

$closure(A \to \bullet A\ \mathtt{a}) = \{A \to \bullet A\ \mathtt{a},\ A \to \bullet\mathtt{b}\}$

Canonical Collection of LR(0) Items

goto: $goto : U \times X \rightarrow 2^U$, where X is a grammar symbol.

$goto(A \rightarrow \alpha \bullet X\beta, X) = closure(A \rightarrow \alpha X \bullet \beta)$.

Example: $goto(A \rightarrow \bullet A\mathtt{a}, A) = closure(A \rightarrow A \bullet \mathtt{a}) = \{A \rightarrow A \bullet \mathtt{a}\}$
closure and *goto* can be extended to a set S of $LR(0)$ items by appropriate generalizations

- $closure(S) = \bigcup_{\mathtt{I} \in \mathtt{S}} \{closure(I)\}$
- $goto(S, X) = \bigcup_{\mathtt{I} \in \mathtt{S}} \{goto(I, X)\}$

Illustration of the Algorithm

For the grammar:

$$\begin{array}{lcl} E & \rightarrow & E + T \mid T \\ T & \rightarrow & T * F \mid F \\ F & \rightarrow & (E) \mid \texttt{id} \end{array}$$

I_0 is closure($E' \rightarrow \bullet E$).

- The items $E' \rightarrow \bullet E$, $E \rightarrow \bullet E + T$ and $E \rightarrow \bullet T$ are added to I_0.
- The last one in turn causes the addition of $T \rightarrow \bullet T * F$ and $T \rightarrow \bullet F$.
- The item $T \rightarrow \bullet F$ leads to the addition of $F \rightarrow \bullet (E)$ and $F \rightarrow \bullet\ \texttt{id}$.
- No more items can be added and the collection I_0 is complete

Canonical Collection of LR(0) items

$$\begin{array}{rcl} I_0 : E' & \rightarrow & \bullet\, E \\ E & \rightarrow & \bullet E{+}T \mid \bullet\, T \\ T & \rightarrow & \bullet T{*}F \mid \bullet\, F \\ F & \rightarrow & \bullet(E) \mid \bullet\,\texttt{id} \end{array}$$

$$\begin{array}{rcl} I_1 : E' & \rightarrow & E\bullet \\ E & \rightarrow & E\bullet{+}T \end{array}$$

$$\begin{array}{rcl} I_2 : E & \rightarrow & T\bullet \\ T & \rightarrow & T\bullet{*}F \end{array}$$

$$\begin{array}{rcl} I_3 : T & \rightarrow & F\bullet \end{array}$$

$$\begin{array}{rcl} I_4 : F & \rightarrow & (\bullet E) \\ E & \rightarrow & \bullet E{+}T \mid \bullet\, T \\ T & \rightarrow & \bullet T{*}F \mid \bullet\, F \\ F & \rightarrow & \bullet(E) \mid \bullet\,\texttt{id} \end{array}$$

$$\begin{array}{rcl} I_5 : F & \rightarrow & \texttt{id}\bullet \end{array}$$

$$\begin{array}{rcl} I_6 : E & \rightarrow & E{+}\bullet\, T \\ T & \rightarrow & \bullet T{*}F \mid \bullet\, F \\ F & \rightarrow & \bullet(E) \mid \bullet\,\texttt{id} \end{array}$$

$$\begin{array}{rcl} I_7 : T & \rightarrow & T{*}\bullet\, F \\ F & \rightarrow & \bullet(E) \mid \bullet\,\texttt{id} \end{array}$$

$$\begin{array}{rcl} I_8 : F & \rightarrow & (E\bullet) \\ E & \rightarrow & E\bullet{+}T \end{array}$$

$$\begin{array}{rcl} I_9 : E & \rightarrow & E+T\bullet \\ T & \rightarrow & T\bullet{*}F \end{array}$$

$$\begin{array}{rcl} I_{10} : T & \rightarrow & T{*}F\bullet \end{array}$$

$$\begin{array}{rcl} I_{11} : F & \rightarrow & (E)\bullet \end{array}$$

Construction of SLR(1) Parsing Table

The procedure is described below:

1. From the input grammar G, construct an equivalent augmented grammar G'.
2. Use the algorithm for constructing *FOLLOW* sets to compute $FOLLOW(A)$, $\forall\, A \in N'$.
3. Call procedure $items(G', C)$ to get the desired canonical collection $C = \{I_0, I_1, \ldots, I_n\}$.
4. Choose as many state symbols as the cardinality of C. We use numbers 0 through n to represent states.

Rules for constructing *FOLLOW*

The *FOLLOW* set of a nonterminal is the smallest set satisfying the following constraints:

1. For the start symbol S, $\$ \in FOLLOW(S)$.
2. For a production $A \rightarrow \alpha B\beta$, $FIRST(\beta) - \{\varepsilon\} \subseteq FOLLOW(B)$.
3. If $A \rightarrow \alpha B$ is a production, or if $A \rightarrow \alpha B\beta$ is a production and $\varepsilon \in FIRST(\beta)$, $FOLLOW(A) \subseteq FOLLOW(B)$.

Example : *FOLLOW* sets for the grammar of Figure 3.8

- $FOLLOW(\ E') = \{\ \$\ \}$
- $FOLLOW(\ E\ ) = \{\ \$, +,)\ \}$
- $FOLLOW(\ T\ ) = \{\ \$, +,), *\ \}$
- $FOLLOW(\ F\ ) = \{\ \$, +,), *\ \}$

Rules for constructing *FIRST*

The *FIRST* set of a nonterminal is the smallest set satisfying the following constraints:

1. To start with, $FIRST(\beta)$ is defined as follows:
   - If β is a single terminal $\mathtt{a}$, $FIRST(\beta) = \{\ \mathtt{a}\ \}$.
   - If β is ε, then $FIRST(\beta) = \{\ \varepsilon\ \}$.
   - If β is a string of grammar symbols, $Y_1 Y_2 \ldots Y_r$, then $FIRST(\beta) = \bigcup_k FIRST(Y_k)$, $1 \leq k \leq r$, provided $\varepsilon \in FIRST(Y_i), 1 \leq i \leq k-1$.

     $FIRST(Y_k)$ is included, only if all the preceding nonterminals $Y_1, Y_2, Y_3, \ldots Y_{k-1}$ derive ε.
2. If there is a production $A \rightarrow \beta$, $FIRST(B) \subseteq FIRST(A)$.

Construction of SLR(1) Parsing Table

For each $I_i, 0 \leq i \leq n$ do steps given below.

1. if $A \rightarrow \alpha \bullet$ a $\beta \in I_i$ and $goto(I_i$,a) = I_j, then $action[i,a] = shift\ j$.
2. If $A \rightarrow \alpha \bullet \in I_i$, then $action[i,a] = reduce\ A \rightarrow \alpha$ for all $a \in$FOLLOW(A).
3. If I_i contains the item $S' \rightarrow S\bullet$, then $action[i,\$] = accept$
4. All remaining entries of state i in the action table are marked as *error*.
5. For nonterminals A , such that $goto(I_i, A) = I_j$ create $goto[i,A] = j$. The remaining entries in the goto table are marked as *error*.

The initial state of the parser is the state corresponding to the set I_0.

SLR(1) Grammar and Parser

stack		*input*
$	●	((id + id))$
$(	●	(id + id))$
$((	●	id + id))$
$((id	●	+ id))$
$((F	●	+ id))$
$((T	●	+ id))$
$((*E*	●	+ id))$
$((*E* +	●	id))$
$((*E* + id	●	))$
$((*E* + *F*	●	))$
$((*E* + *T*	●	))$
$((*E*	●	))$
$((*E*)	●	)$
$(*F*	●	)$
$(*T*	●	)$
$(*E*	●	)$
$(*E*)	●	$
$*F*	●	$
$*T*	●	$
$*E*	●	$

SLR(1) Grammar and Parser

A grammar for which there is a conflict free SLR(1) parsing table is called a *SLR(1) grammar* and a parser which uses such a table is known as *SLR(1) parser*.

How do conflicts manifest in a SLR(1) parser ?

- A shift- reduce conflict is detected when a state has
  1. a complete item of the form $A \rightarrow \alpha\bullet$ with $\mathtt{a} \in \mathit{FOLLOW}(A)$, and also
  2. an incomplete item of the form $B \rightarrow \beta\bullet\ \mathtt{a}\ \gamma$
- A reduce-reduce conflict is noticed when a state has two or more complete items of the form $A \rightarrow \alpha\bullet$ and $B \rightarrow \beta\bullet$, and $\mathit{FOLLOW}(A) \bigcap \mathit{FOLLOW}(B) \neq \Phi$.

Conceptual Issues

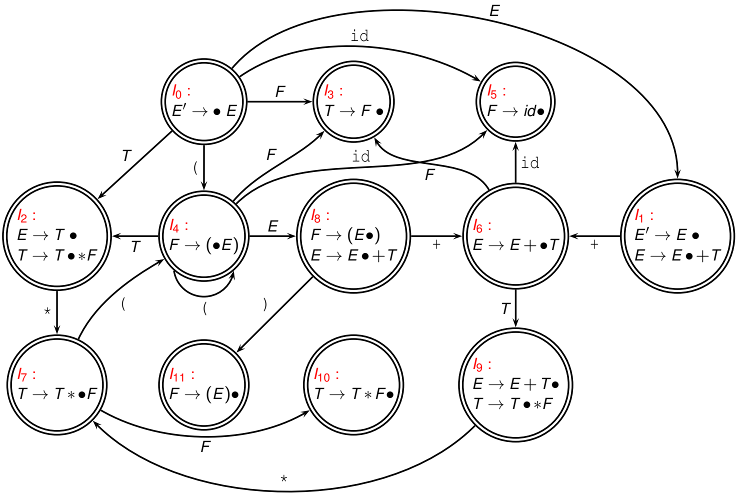
1. What information do the states contain?
2. Where exactly is handle detection taking place in the parser?
3. Why is *FOLLOW* information used to create the reduce entries in the action table ?

To answer these questions, we need to see the canonical collection of LR(0) items as a DFA.

- A node labeled I_i is constructed for each member of C.
- For every nonempty goto(I_i, X) = I_j , a directed edge (I_i, I_j) is added labeled with X.
- The graph is a deterministic finite automaton if the node labeled I_0 is treated as the *start* state and all other nodes are made final states.

What does the automaton recognize?

The DFA of a LR parser



Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

The automation shown before recognizes viable prefixes only.

Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

The automation shown before recognizes viable prefixes only.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.

Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

The automation shown before recognizes viable prefixes only.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.
2. Viable prefixes are precisely the set of symbols that can ever appear on the stack of a LR parser

Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

The automation shown before recognizes viable prefixes only.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.
2. Viable prefixes are precisely the set of symbols that can ever appear on the stack of a LR parser
3. A viable prefix either contains a handle or contains a part of a handle.

Viable Prefix and Valid Items

A *viable prefix* is the prefix of a right-sentential form that does not contain any symbols to the right of a handle.

The automation shown before recognizes viable prefixes only.

1. By adding terminal symbols to viable prefixes, rightmost sentential forms can be constructed.
2. Viable prefixes are precisely the set of symbols that can ever appear on the stack of a LR parser
3. A viable prefix either contains a handle or contains a part of a handle.
4. For a viable prefix, if is useful to identify the portion of the handle that it contains.

Viable Prefix and Valid Items

A LR(0) item $A \rightarrow \beta_1 \bullet \beta_2$ is defined to be *valid* for a viable prefix, $\alpha\beta_1$, provided $S \overset{*}{\Rightarrow}_{rm} \alpha A\, \mathtt{w} \Rightarrow_{rm} \alpha\beta_1\beta_2\, \mathtt{w}$

1. There could be several distinct items which are valid for the same viable prefix γ.
2. It is interesting to note that in above, if $\beta_2 = B\gamma$ and $B \rightarrow \delta$, then $B \rightarrow \bullet\delta$ is also a valid item for this viable prefix.
3. A particular item may be valid for many distinct viable prefixes.

Viable Prefixes and Valid Items

- For the LR-automaton shown earlier, consider the path labeled by the viable prefix $(E+$ ending in I_6. The items valid for $(E+$ are:
  1. $E' \Rightarrow_{rm} E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T)$ shows that $E \rightarrow E + \bullet T$ is a valid item for $(E+$.
  2. $E' \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (E+T) \Rightarrow (E+T*F)$ shows that $T \rightarrow \bullet T * F$ is also a valid item.
  3. $E' \overset{*}{\Rightarrow}_{rm} (E+T) \Rightarrow (E+F)$ shows that $T \rightarrow \bullet F$ is another such item.
  4. $E' \overset{*}{\Rightarrow}_{rm} (E+T) \Rightarrow (E+F) \Rightarrow (E+(E))$ shows that $F \rightarrow \bullet(E)$ is also a valid item for $(E+$.
  5. Finally, $E' \overset{*}{\Rightarrow}_{rm} (E+F) \Rightarrow (E+\texttt{id})$ shows that $F \rightarrow \bullet\ \texttt{id}$ is a valid item for $(E+$.

  It should be noted that are no other valid items for this viable prefix.

Viable Prefixes and Valid Items

Given a LR(0) item, say $T \rightarrow T \bullet * F$, there may be several viable prefixes for which it is valid.

1. $E' \Rightarrow_{rm} E \Rightarrow T \Rightarrow T * F$ shows that this item is valid for the viable prefix T .
2. $E' \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T) \Rightarrow (T * F)$ shows that it is also valid for (T .
3. $E' \Rightarrow E \Rightarrow T \Rightarrow T * F \Rightarrow T * (E) \Rightarrow T * (T) \Rightarrow T * (T * F)$ shows that it is valid also for T * (T .
4. $E' \Rightarrow E \Rightarrow E + T \Rightarrow E + T * F$ shows validity for $E + T$.

There may be several other viable prefixes for which this item is valid.

Theory of LR Parsing

THEOREM : Starting from I_0, if traversing the LR(0) automaton γ results in state j, then set items in I_j are the only valid items for the viable prefix γ.

- The theorem stated without proof above is a key result in LR Parsing. It provides the basis for the correctness of the construction process we learnt earlier.
- An LR parser does not scan the entire stack to determine when and which handle appears on top of stack (compare with shift-reduce parser).
- The state symbol on top of stack provides all the information that is present in the stack.
- In a state which contains a complete item a reduction is called for. However, the lookahead symbols for which the reduction should be applied is not obvious.
- *In SLR(1) parser the FOLLOW information is used to guide reductions.*

Limitations of SLR(1) PARSER

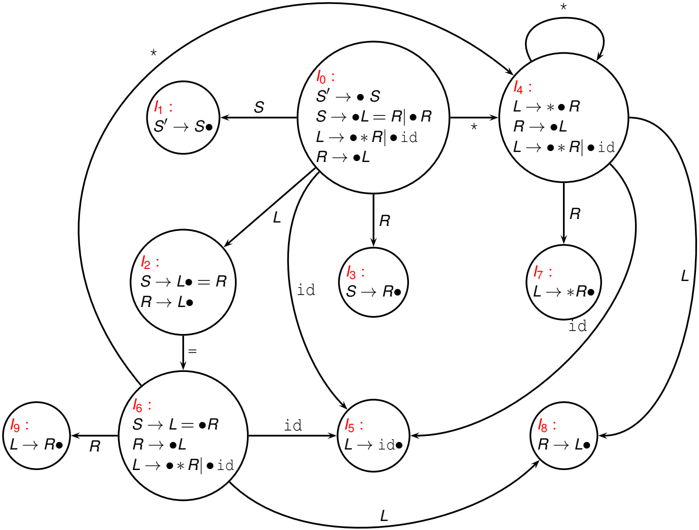
Using *FOLLOW* information for reduction is imprecise. Consider the following grammar.

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

The *FOLLOW* set for this grammar is:

symbol a	$FOLLOW(a)$
S	$\{\$\}$
S'	$\{\$\}$
L	$\{=, \$\}$
R	$\{=, \$\}$

LR Automaton For Expression Grammar



SLR Parsing Table

	action				goto		
state	id	*	=	$	S	L	R
0	s5	s4			c1	c2	c3
1				acc			
2			s6/r5	r5			
3				r2			
4	s5	s4				c8	c7
5			r4	r4			
6	s5	s4				c8	c9
7			r3	r3			
8			r5	r5			
9				r1			

SLR Parsing Table

- Observe that $\{=\} \in$ FOLLOW(R), and $R \rightarrow L\bullet$ is an item in state 2. There is another item, $S \rightarrow L\bullet = R$ in the same state
- Note that state 2 recognizes the viable prefix L and the shift entry is justifiable.
- How about the reduce entry ? After seeing L, if we reduce it to R, with the expectation of = to follow, there must exist a viable prefix $R = \ldots$ and hence a right sentential form of the form R = z. It can be shown that such is not possible. The problem seems to be with our FOLLOW information.

Limitations of SLR(1) Parser

What is wrong with FOLLOW ?

- This information is not correct for state 2.
- The sentential forms that permit $=$ to follow R are of the form $*L\ldots$ and taken care of in the state 8 of the parser.
- The context in state 2 is different (viable prefix is L), and use of FOLLOW constrains the parser.

Given an item and a state the need is to identify the terminal symbols that can actually follow the lhs nonterminal.

LR(1) items

- An $LR(1)$ item is an item of the form

$$A \rightarrow \alpha \bullet \beta,\ LA$$

  - The first component is a $LR(0)$ item.
  - The second component is a set of terminals called the *lookahead* symbols.

  $LR(1)$ indicates that the length of lookahead is 1.
- The lookahead symbol is used as follow:
  - For an item of the form, $A \rightarrow \alpha \bullet \beta,\ LA$, the lookahead has no effect.
  - For an item, $A \rightarrow \alpha \bullet,\ LA$, the reduction is applied when the next input symbol *nexttoken* is in LA.

LR(1) PARSER CONSTRUCTION

VALID LR(1) ITEM

- An LR(1) item $A \rightarrow \beta_1 \bullet \beta_2$, $\mathtt{a}$ is valid for a viable prefix $\alpha\beta$, if there is a derivation $S \overset{*}{\Rightarrow}_{rm} \alpha A \mathtt{w} \Rightarrow_{rm} \alpha\beta_1\beta_2\ \mathtt{w}$, and $\mathtt{a} \in \mathit{FIRST}(w\$)$.
- If $A \rightarrow \alpha \bullet B\beta$, $\mathtt{a}$ is a valid item for a viable prefix γ, then the item $B \rightarrow \bullet\eta$, $\mathtt{b}$ is also valid for the same prefix γ. Here $B \rightarrow \eta$ is a rule and $\mathtt{b} \in \mathit{FIRST}(\beta\ \mathtt{a})$.

LR(1) PARSER CONSTRUCTION

1. Consider the grammar given in Figure 3.22. The first collection, i.e., state I_0, is given by $closure(S' \rightarrow \bullet S, \$)$.
2. This causes the LR(1) items $S \rightarrow \bullet R, \$$ and $S \rightarrow \bullet L = R, \$$ to be added to I_0.
3. Closure of the item $S \rightarrow \bullet R, \$$ causes the addition of the item $R \rightarrow \bullet L, \$$ whose closure in turn adds the items $L \rightarrow \bullet * R, \$$ and $L \rightarrow \bullet \ \mathtt{id}, \$$ to I_0.

LR(1) PARSER CONSTRUCTION

LR(1) PARSING TABLE CONSTRUCTION

4. Closure of the item $S \to \bullet L = R, \$$ leads to the inclusion of items $L \to \bullet * R, =$ and $L \to \bullet\mathtt{id}, =$. The lookaheads for these items are given by $FIRST($ = R \$).
5. Since some LR(1) items added by steps 3 and 4 above have the same core, but different lookaheads, we combine the lookaheads to write them in a compact form, $L \to \bullet * R, \$ =$ and and $L \to \bullet\ \mathtt{id}, \$ =$
6. Collection $I_0 = \{S' \to \bullet S, \$\ ;\ S \to \bullet R, \$\ ;\ S \to \bullet L = R, \$\ ;$ $L \to \bullet * R, \$ =\ ;\ L \to \bullet\ \mathtt{id}, \$ =\ ;\ R \to \bullet L, \$\ \}$
7. The *goto* function on I_0 is defined for the symbols S, L, R, id and *. The resulting states are obtained by taking closures of the kernel items as given below. $I_1 = goto(I_0, S) = closure(S' \to S\bullet, \$)$
   $I_2 = goto(I_0, L) = closure(S \to L\bullet = R, \$; R \to L\bullet, \$\ )$
   $I_3 = goto(I_0, R) = closure(S \to R\bullet, \$\ )$
   $I_4 = goto(I_0, *) = closure(L \to * \bullet R, \$ =\ )$ $I_5 = goto(I_0, \mathtt{id}\ ) =$ $closure(L \to \mathtt{id}\bullet, \$ =\ )$ The rest of the collection can be constructed on

LR(1) PARSER CONSTRUCTION

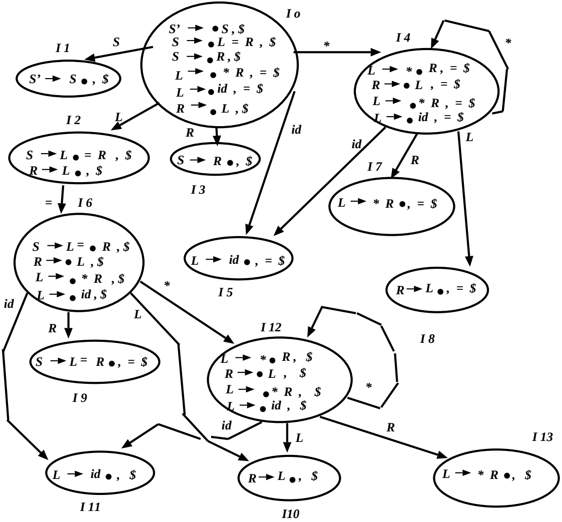
LR(1) PARSING TABLE CONSTRUCTION

The set I_i and its constituent items define the entries for state i of the Action table as follows

1. if $A \rightarrow \alpha \bullet$ a β, b $\in I_i$ and $goto(I_i$, a) = I_j
   Action[i,a] = shift j
2. $A \rightarrow \alpha \bullet$,a $\in I_i$
   Action[i,a] = reduce by $A \rightarrow \alpha$
3. $S' \rightarrow S \bullet, \$ \in I_i$
   Action[i,$] = accept
4. All remaining entries of state i are marked as error.
5. The Goto entries of the state i are
   $goto(I_i, A) = I_j$ leads to Goto[i,A] = j The remaining entries are marked as error.

A grammar for which there is a conflict free LR(1) parsing table is called a LR(1) grammar and a parser which uses such a table is known as LR(1) parser.

LR(1) PARSER CONSTRUCTION

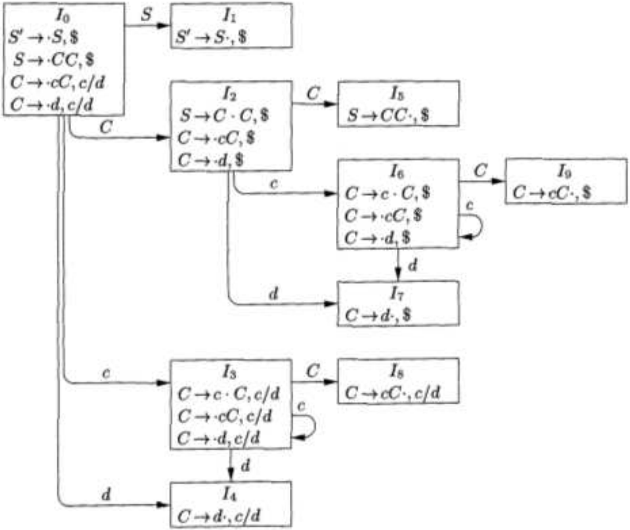


LR(1) PARSER CONSTRUCTION

	Action				Goto		
state	id	*	=	$	*S*	*L*	*R*
0	s5	s4	-	-	1	2	3
1	-	-	-	acc	-	-	-
2	-	-	s6	r5	-	-	-
3	-	-	-	r2	-	-	-
4	s5	s4	-	-	-	8	7
5	-	-	r4	r4	-	-	-
6	s11	s12	-	-	-	10	9
7	-	-	r3	r3	-	-	-
8	-	-	r5	r5	-	-	-
9	-	-	-	r1	-	-	-
10	-	-	-	r5	-	-	-
11	-	-	-	r4	-	-	-
12	s11	s12	-	-	-	10	13
13	-	-		r3	-	-	-

LR(1) PARSER CONSTRUCTION

Yet another example:



LR(1) PARSER CONSTRUCTION

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

LALR(1) PARSER CONSTRUCTION

LALR(1) PARSER

- This parser is of intermediate capability as compared to SLR(1) and LR(1). The parser has the size advantage of SLR and lookahead capability like LR.
- In a LR parser, several states may have the same first components (LR(0) items) but differ in the lookaheads associated and hence are represented as distinct states.
- In terms of the first component only, both SLR and LR define the same collection of LR(0) items.
- LALR automaton is created by merging states of LR automaton that have the same core (set of first components). The merge results in the union of the lookahead symbols of the respective items.

LALR(1) PARSING TABLE CONSTRUCTION

1. Construct the collection $C = \{I_0, I_1, \ldots, , I_n\}$ of LR(1) items.
2. For each core among the set of LR(1) items, find all sets having the same core and replace these sets by their union.