Programming Languages: Syntax Description and Parsing

Onur Tolga Şehitoğlu

Computer Engineering,METU

27 May 2009

Outline

Introduction

- Syntax: the form and structure of a program.
- Semantics: meaning of a program
- Language definitions are used by:
  - Programmers
  - Implementors of the language processors
  - Language designers

Definitions

- A sentence is a string of characters over some alphabet
- A language is a set of sentences
- A lexeme is the lowest level syntactic unit of the language (i.e. `++`, `int`, `total`)
- A token is a category of lexemes (i.e. *identifier*)

Definitions

- syntax recognition: read input strings of the language and verify the input belonging to the language
- syntax generation: generate sentences of the language (i.e. from a given data structure)
- Compilers and interpreters recognize syntax and convert it into machine understandable form.

Backus-Naur Form and CFGs

- CFG's introduced by Noam Chomsky (mid 1950s)
- Programming languages are usually in context free language class
- BNF introduced by John Bakus and modified by Peter Naur for describing Algol language
- BNF is equivalent to CFGs. It is a meta-language that decribes other languages
- Extended BNF improves readability of BNF

A Grammar Rule

⟨**while_stmt**⟩ → `while ( ` ⟨**logic_expr**⟩ `)` ⟨**stmt**⟩

- LHS is a non-terminal denoting an intermediate phrase
- LHS can be defined (rewritten) as the RHS sequence which can contain terminals (lexems and tokens) of the language and other non-terminals
- Non-terminals are denoted as strings enclosed in angle brackets.
- `::=` may be used in BNF notation instead of the arrow
- | is used to combine multiple rules with same LHS in a single rule

⟨**lgc_cons**⟩ `::= true` ⟨**lgc_cons**⟩ `::= false` ≡ ⟨**lgc_cons**⟩ `::= true | false`

Context Free Grammar

- A grammar G is defined as $G = (V, \Sigma, R, S)$:
  - N, finite set of non terminals
  - Σ, finite set of terminals
  - R is a set of grammar rules. A relation from V to $(V \cup \Sigma)^*$.
  - $S \in N$ the start symbol
- Application of a rule maps one sentential form into the other by replacing a non-terminal element in sentential form with its right handside seuqence in the rule, $u \mapsto v$.
- Language of a grammar $L(G) = \left\{ w \mid w \in \Sigma^*, S \overset{*}{\mapsto} w \right\}$

- Recursive or list like structures can be represented using recursion
  $\langle \textbf{expr\_list} \rangle \rightarrow \langle \textbf{expr} \rangle \ , \ \langle \textbf{expr\_list} \rangle$
  $\langle \textbf{btree} \rangle \rightarrow \langle \textbf{head} \rangle \ (\ \langle \textbf{btree} \rangle \ , \ \langle \textbf{btree} \rangle \ )$
- A derivation starts with a starting non-terminal and rules are applied repeteadly to end with a sentence containing only terminal symbols.
- leftmost derivation: always leftmost non-terminal is chosen for replacement
- rightmost derivation: always rightmost non-terminal is chosen for replacement
- Same sentence can be derived using leftmost, rightmost, or other derivaionts.

Sample Grammar

$\langle \textbf{stmt} \rangle \rightarrow \langle \textbf{id} \rangle$ = $\langle \textbf{expr} \rangle$
$\langle \textbf{expr} \rangle \rightarrow \langle \textbf{expr} \rangle \ \langle \textbf{op} \rangle \ \langle \textbf{expr} \rangle \mid \langle \textbf{id} \rangle$
$\langle \textbf{op} \rangle \rightarrow$ + | *
$\langle \textbf{id} \rangle \rightarrow$ a | b | c

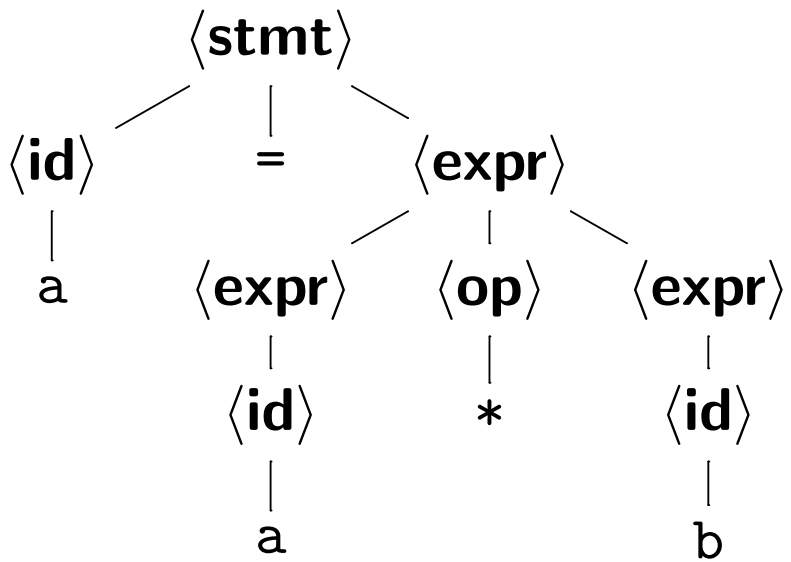
- Leftmost derivation of a = a * b :
  $\langle \textbf{stmt} \rangle \mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle \mapsto$ a = $\langle \textbf{expr} \rangle$
  $\mapsto$ a = $\langle \textbf{id} \rangle \ \langle \textbf{op} \rangle \ \langle \textbf{expr} \rangle \mapsto$ a = b $\langle \textbf{op} \rangle \ \langle \textbf{expr} \rangle$
  $\mapsto$ a = b * $\langle \textbf{expr} \rangle \mapsto$ a = b * $\langle \textbf{id} \rangle \mapsto$ a = b * c
- Rightmost derivation of a = a * b :
  $\langle \textbf{stmt} \rangle \mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle \mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle \ \langle \textbf{op} \rangle \ \langle \textbf{expr} \rangle$
  $\mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle \ \langle \textbf{op} \rangle \ \langle \textbf{id} \rangle \mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle \ \langle \textbf{op} \rangle$ b
  $\mapsto \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle$ * b $\mapsto \langle \textbf{id} \rangle = \langle \textbf{id} \rangle$ * b
  $\mapsto \langle \textbf{id} \rangle$ = a * b $\mapsto$ a = a * b

Parse Tree

- Steps of a derivation gives the structure of the sentence. This structure can be represented as a tree.
- All non-terminals used in derivation are intermediate nodes. Each grammar rule replaces the non-terminal node with is children. Root node is the start symbol.
- Terminal nodes are the leaf nodes.
- preorder traversal of leaf nodes gives the resulting sentence.
- leftmost and rightmos derivations can be retrieved by traversal of the tree.

Parse Tree Example

```
a = a * b
```

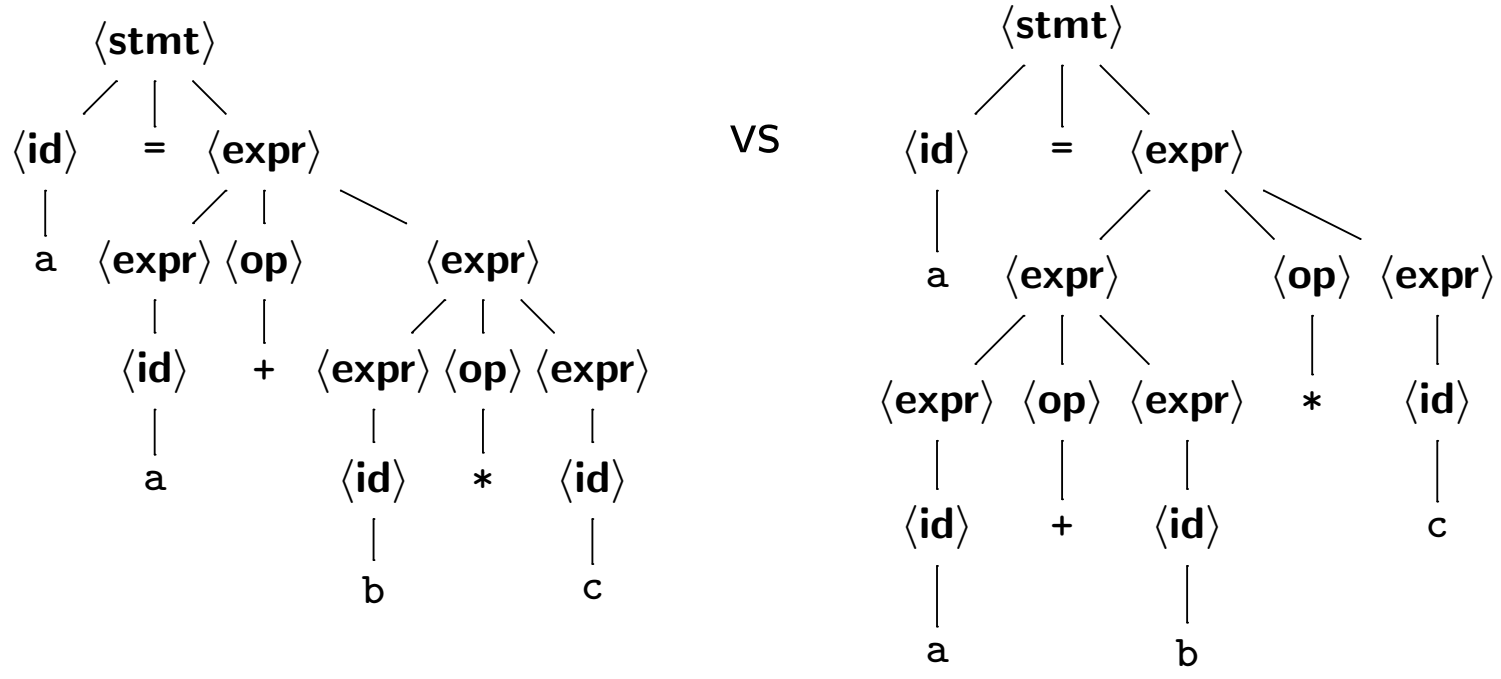


Parse Tree Generation

- A parse tree gives the structure of the program so semantics of the program is related to this structure.
- For example local scopes, evaluation order of expressions etc.
- During compilation, parse trees might be required for code generation, semantic analysis and optimization phases.
- After a parse tree generated, it can be traversed to do various tasks of compilation.
- The processing of parse tree takes too long, so creation of parse trees is usually avoided.
- Approaches like syntax directed translation combines parsing with code generation, semantic analysis etc..

Ambigous Grammars

- Consider `a = a + b * c` in our grammar:



- Both can be derived by the grammar!

- A grammar is called ambigous if same sentence can be derived by following different set of rules, thus resulting in a different parse tree
- If structure changes semantic meaning of the program, ambiguity is a serious problem.
- Even if not, which one is the result?
- i.e. Precedence of operators affects the value of the expression.
- Programming languages enforces precedence rules to resolv ambiguity.
- Solution:
  1. design grammar not to be ambigous, or
  2. during parsing, choose rules to generate the correct parse tree

Precedence and Grammar

- Operators with different precedence levels should be treated differently
- Higher precedence operations should be deep in the parse tree $\rightarrow$ their rules should be applied later.
- Lower precedence operations should be closer to root $\rightarrow$ applied earlier in derivation.
- For each precedence level, define a non-terminal
- One rewritten on the other based on the precedence lower to higher

Rewritten Grammar

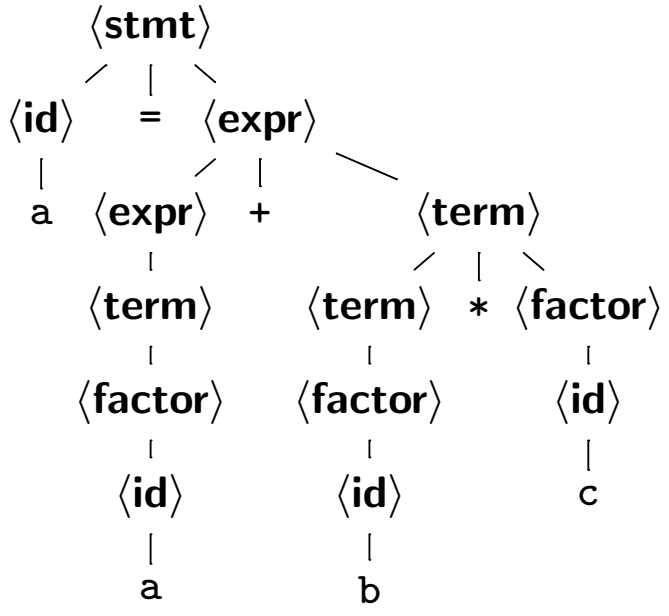
$\langle \textbf{stmt} \rangle \rightarrow \langle \textbf{id} \rangle = \langle \textbf{expr} \rangle$

$\langle \textbf{expr} \rangle \rightarrow \langle \textbf{expr} \rangle + \langle \textbf{term} \rangle \mid \langle \textbf{term} \rangle$

$\langle \textbf{term} \rangle \rightarrow \langle \textbf{term} \rangle * \langle \textbf{factor} \rangle \mid \langle \textbf{factor} \rangle$

$\langle \textbf{factor} \rangle \rightarrow \langle \textbf{id} \rangle \mid (\langle \textbf{expr} \rangle)$

$\langle \textbf{id} \rangle \rightarrow \texttt{a} \mid \texttt{b} \mid \texttt{c}$



- ⟨**term**⟩ and ⟨**expr**⟩ has different precedence.
- Once inside of ⟨**term**⟩, there is no way to derive +
- Only one parse possible

Associativity

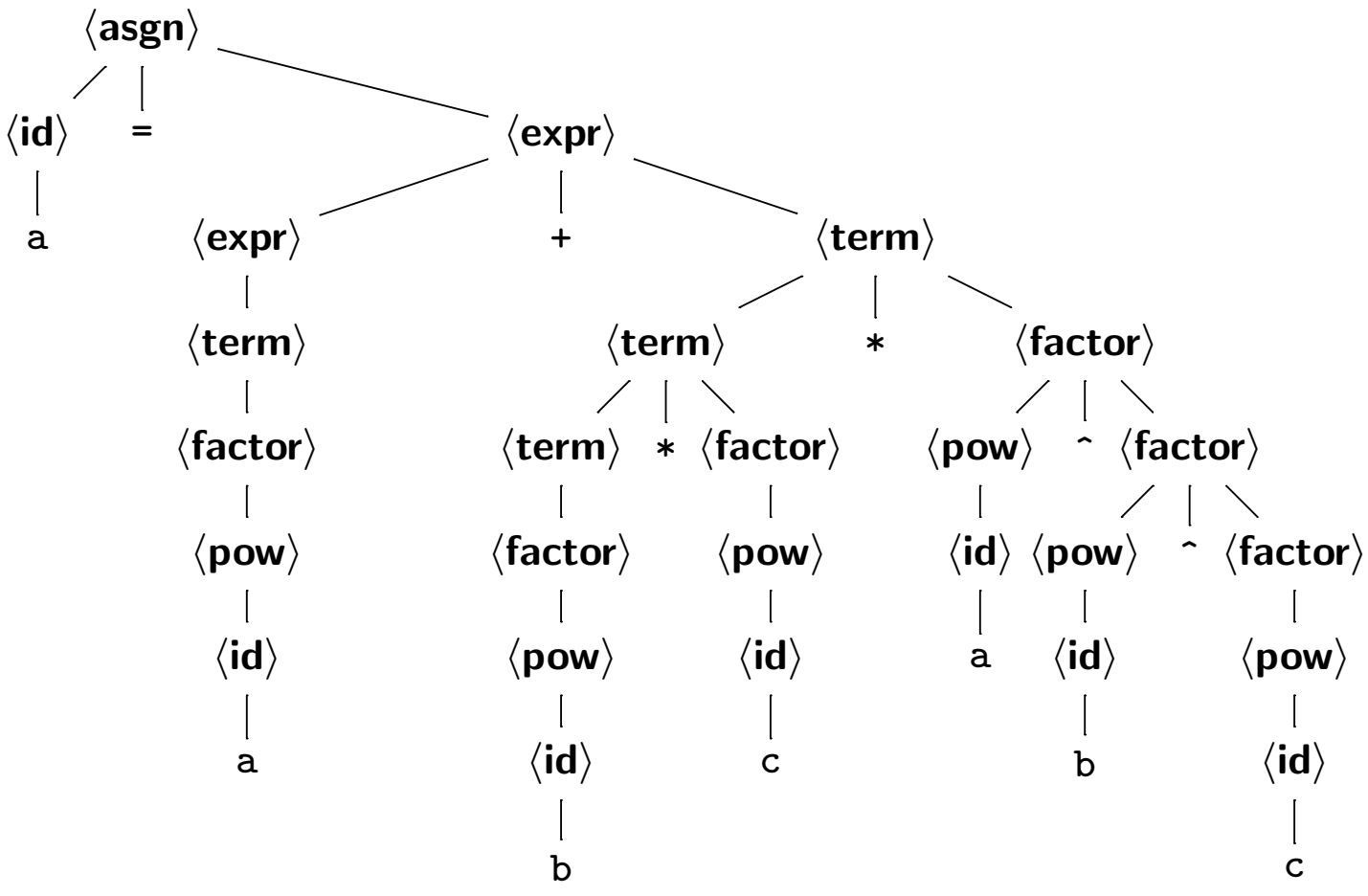
- Associativity of operators is another issue
  `a - b - c` ≡ `( a - b ) - c` or `a - ( b - c)`
- Recursion of grammar defines how tree is constructed for operators in the same level.
- If left recursive, later operators in the sentence will be closer to root, if right recursive earlier operators will be closer to root
- left recursion implies left associativity, right recursion implies right associativity.
- Consider `a + b + c` in these grammars:
  ⟨**expr**⟩ → ⟨**expr**⟩ `+` ⟨**id**⟩ | ⟨**id**⟩
  ⟨**id**⟩ → `a` | `b` | `c`
  vs
  ⟨**expr**⟩ → ⟨**id**⟩ `+` ⟨**expr**⟩ | ⟨**id**⟩
  ⟨**id**⟩ → `a` | `b` | `c`

Sample Grammar

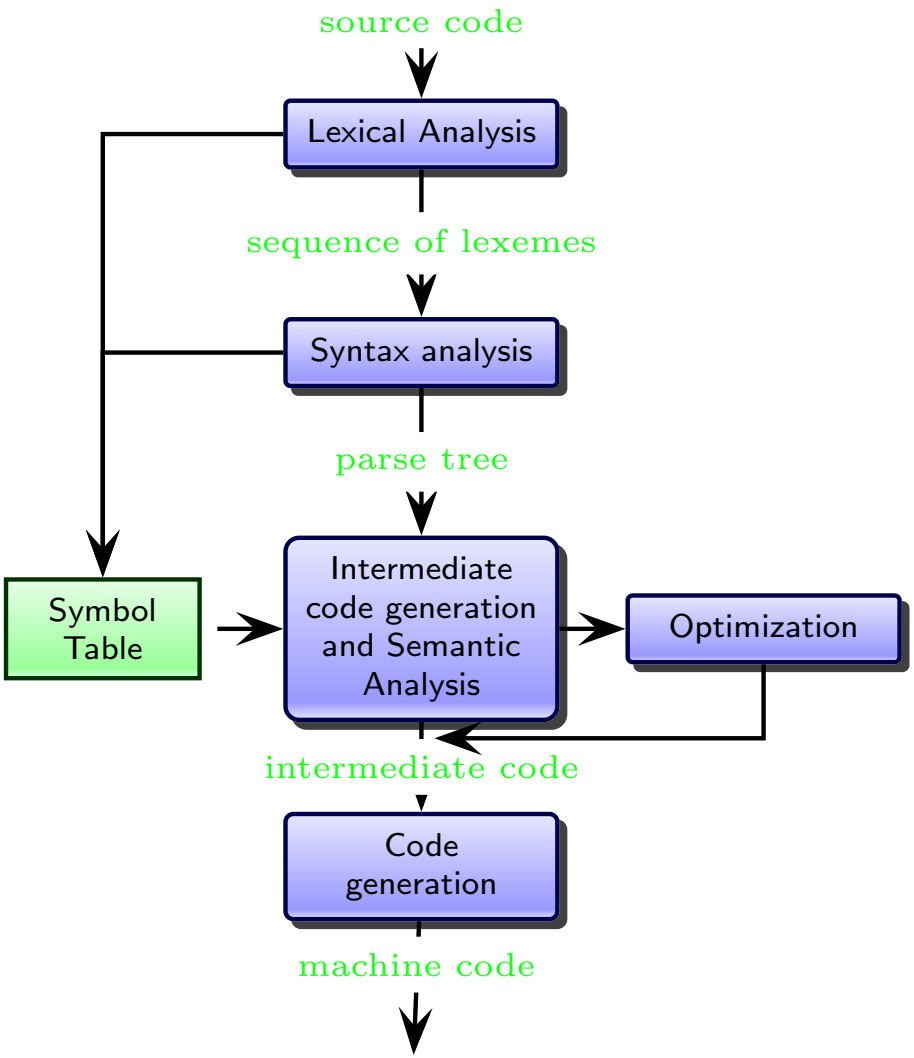
$\langle \textbf{asgn} \rangle \rightarrow \langle \textbf{id} \rangle$ = $\langle \textbf{asgn} \rangle \mid \langle \textbf{id} \rangle$ = $\langle \textbf{expr} \rangle$
$\langle \textbf{expr} \rangle \rightarrow \langle \textbf{expr} \rangle$ + $\langle \textbf{term} \rangle \mid \langle \textbf{term} \rangle$
$\langle \textbf{term} \rangle \rightarrow \langle \textbf{term} \rangle$ * $\langle \textbf{factor} \rangle \mid \langle \textbf{factor} \rangle$
$\langle \textbf{factor} \rangle \rightarrow \langle \textbf{pow} \rangle$ ^ $\langle \textbf{factor} \rangle \mid \langle \textbf{pow} \rangle$
$\langle \textbf{pow} \rangle \rightarrow \langle \textbf{id} \rangle \mid$ ($\langle \textbf{expr} \rangle$)
$\langle \textbf{id} \rangle \rightarrow$ a | b | c

- $\langle \textbf{asgn} \rangle$ is right recursive like right associative C assignments.
- $\langle \textbf{expr} \rangle$ and $\langle \textbf{term} \rangle$ are left recursive, * and + left associative
- $\langle \textbf{factor} \rangle$ is right recursive for power operation ^ to be right associative.
- precedence order is (...) $\prec$ ^ $\prec$ * $\prec$ + $\prec$ =

`a = a + b * c * a ^ b ^ c`

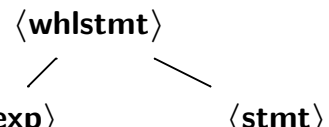

⟨asgn⟩
⟨id⟩ = ⟨expr⟩
a ⟨expr⟩ + ⟨term⟩
⟨term⟩ ⟨term⟩ * ⟨factor⟩
⟨factor⟩ ⟨term⟩ * ⟨factor⟩ ⟨pow⟩ ^ ⟨factor⟩
⟨pow⟩ ⟨factor⟩ ⟨pow⟩ ⟨id⟩ ⟨pow⟩ ^ ⟨factor⟩
⟨id⟩ ⟨pow⟩ ⟨id⟩ a ⟨id⟩ ⟨pow⟩
a ⟨id⟩ c b ⟨id⟩
b c

Compilation




```
while (counter < 12341) {
        f() ;
        counter += 12;
}
```

```
WHL LP ID LT ILIT RP LB
        ID LP RP SC
        ID PLEQ ILIT SC
RB
```



Parsing

- input: sequence of lexemes (output of lexical analysis) or characters.
- output: parse tree, intermediate code, translated code, or sometimes only if document is valid or not.
- Two main classes of parser:
  - Top down parsing
  - Tottom up parsing

Top-down Parsing

- Start from the starting non-terminal, apply grammar rules to reach the input sentence

$$\begin{aligned}&\langle assign\rangle \;\mapsto\; a = \langle expr\rangle \;\mapsto\; a = \langle expr\rangle + \langle term\rangle \;\mapsto\\ &a = \langle term\rangle + \langle term\rangle \;\mapsto\; a = \langle fact\rangle + \langle term\rangle \;\mapsto\\ &a = a + \langle term\rangle \;\mapsto\; a = a + \langle term\rangle * \langle fact\rangle \;\mapsto\\ &a = a + \langle fact\rangle * \langle fact\rangle \;\mapsto\; a = a + b * \langle fact\rangle \;\mapsto\\ &a = a + b * a\end{aligned}$$

- Simplest form gives leftmost derivation of a grammar processing input from left to right.
- Left recursion in grammar is a problem. Elimination of left recursion needed.
- Deterministic parsing: Look at input symbols to choose next rule to apply.
- recursive descent parsers, LL family parsers are top-down parsers

Recursive Descent Parser

```
typedef enum {ident, number, lparen, rparen, times,
        slash, plus, minus} Symbol;
int accept(Symbol s) { if (sym == s) { next(); return 1; }
        return 0;
}
void factor(void) {
    if (accept(ident)) ;
    else if (accept(number)) ;
    else if (accept(lparen)) { expression(); expect(rparen);}
    else { error("factor: syntax error at ",currsym); next(); }
}
void term(void) {
    factor();
    while (accept(times) || accept(slash))
        factor();
}
void expression(void) {
    term();
    while (accept(plus) || accept(minus))
        term();
}
```

- Each non-terminal realized as a parsing function
- Parsing functions calls the right handside functions in sequence
- Rule choices are based on the current input symbol. `accept` checks a terminal and consumes if matches.
- Cannot handle direct or indirect left recursion. A function has to call itself before anything else.
- Hand coded, not flexible.

LL Parsers

- First L is 'left to right input processing', second is 'leftmost derivation'
- Checks next N input symbols to decide on which rule to apply: LL(N) parsing.
- For example LL(1) checks the next input symbol only.
- LL(N) parsing table: A table for $V \times \Sigma^N \mapsto R$
- for expanding a nonterminal $NT \in V$, looking at this table and the next N input symbols, LL(N) parser chooses the grammar rule $r \in R$ to apply in the next step.

- Grammar and lookup table for a LL(1) parser:

$$
\begin{array}{ll}
1 & S \to E \\
2 & S \to -E \\
3 & E \to N{+}E \\
4 & E \to (E) \\
5 & N \to \mathtt{a} \\
6 & N \to \mathtt{b}
\end{array}
$$

	a	b	-	(
S	1	1	2	1
E	3	3		4
N	5	6		

- What if we add $E \to N$ to grammar?
- You need an LL(2) grammar. What if N is recursive?

Bottom-up Parsing

- Start from input sentence and merge parts of sentential form matching RHS of a rule into LHS at each step. Try to reach the starting non-terminal. reach the input sentence

$$a = a + b * a \quad \mapsto \quad a = \langle fact \rangle + b * a \quad \mapsto \quad a = \langle term \rangle + b * a \quad \mapsto$$
$$a = \langle expr \rangle + b * a \quad \mapsto \quad a = \langle expr \rangle + \langle fact \rangle * a \quad \mapsto$$
$$a = \langle expr \rangle + \langle term \rangle * a \quad \mapsto \quad a = \langle expr \rangle + \langle term \rangle * \langle fact \rangle \quad \mapsto$$
$$a = \langle expr \rangle + \langle term \rangle \quad \mapsto \quad a = \langle expr \rangle \quad \mapsto \langle assign \rangle$$

- Simplest form gives rightmost derivation of a grammar (in reverse) processing input from left to right.
- Shift-reduce parsers are bottom-up:
  - shift: take a symbol from input and push to stack.
  - reduce: match and pop a RHS from stack and reduce into LHS.

Shift-Reduce Parser in Prolog

```
% Grammar is E-> E-T|E+T|T  T -> a|b
rule(e,[e,-,t]).
rule(e,[e,+,t]).
rule(e,[t]).
rule(t,[a]).
rule(t,[b]).

parse([],[S]) :- S = e .  % starting symbol alone in the stack
% reduce: find RHS of a rule on stack, reduce it to LHS
parse(Input,Stack) :- match(LHS,Stack,Remainder),
                      parse(Input,[LHS|Remainder]).

% shift: nonterminals are removed from input added on stack
parse([H|Input],Stack) :- member(X,[a,b,-,+]),
                          parse(Input,[H|Stack]).

% check if RSH of a rule is a prefix of Stack (reversed).
match(LHS,List,L) :- rule(LHS,RHS),  reverse(RHS,NRHS),
                     prefix(NRHS,List,L).
```

- Shift reduce parser tries all non-deterministic shift combinations to get all parses.
- Deterministic bottom up parsers: LALR, SLR(1).