Parser

Larissa von Witte

Institut für Softwaretechnik und Programmiersprachen

11. Januar 2016

Contents

Introduction

- analyses the syntax of an input text with a given grammar or regular expression
- returns a parse tree
- important for the further compiling process

Lookahead

Definition: Lookahead

The lookahead k are the following k tokens of the text, that are provided by the scanner.

Context-free Grammar

Definition: Formal Grammar

A formal grammar is a tuple $G = (T, N, S, P)$, with

- T as a finite set of terminal symbols
- N as a finite set of nonterminal symbols and $N \cap T = \emptyset$
- S as a start symbol and $S \in N$
- P as a finite set of production rules of the form $l \rightarrow r$ with $l, r \in (N \cup T)^*$

Definition: Context-free Grammar

A grammar $G = (N, T, S, P)$ is called context-free if every rule $l \rightarrow r$ holds the condition:
l is a single nonterminal symbol, so $l \in N$.

LL(1) Grammar

Definition: First(A)

$$\text{First}(A) = \{t | A \Rightarrow^* t\alpha\} \cup \{\varepsilon | A \Rightarrow^* \varepsilon\}$$

Definition: Follow(A)

$$\text{Follow}(A) = \{t | S \Rightarrow^* \alpha A t \beta\}$$

Definition: *LL(1)* Grammar

A context-free grammar is called *LL(1)* grammar if it holds the following conditions for every rule $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ with $i \neq j$

$$\text{First}(\alpha_i) \cap \text{First}(\alpha_j) = \emptyset$$

$$\varepsilon \in \text{First}(\alpha_i) \rightarrow \text{Follow}(A) \cap \text{First}(\alpha_j) = \emptyset$$

Recursive Descent Parser

- top-down parser
- basic idea: create an own parser $parse_A$ for every nonterminal symbol A
- every parser $parse_A$ is basically a method which consists of a case-by-case analysis
- it compares the lookahead with the expected symbols
- begins with $parse_S$ and determines the next parser based on the lookahead k (usually $k = 1$)
- needs *LL(k)* grammar for a distinct decision
- grammar must not be left recursive because it could lead to a non-terminating parser

Example: Recursive Descent Parser

Example Grammar

expression $\rightarrow$ number | (expression operator expression)
operator $\rightarrow +| - | * |/$

Example: Recursive Descent Parser

```
boolean parseOperator(){
  char op = Text.getLookahead();
  if (op == '+' || op == '-' || op == '*' || op == '/'){
      Text.removeChar(); //removes the operator from the input
      return true;
  }else{
      throwException();
  }

boolean parseExpression(){
  if (Text.getLookahead().isDigit()){
      return parseNumber();
  }else if (Text.getLookahead() == '(' ){
    boolean check = true;
    Text.removeChar();
    check  &=  parseExpression() && parseOperator() && parseExpression();
    if (Text.getLookahead() != ')' ){
      throwException();
    }else{
      return check;
    }
  }else{
    throwException();
  }
}
```

Recursive descent parser

- often used for hand-written parsers
- needs special grammar
- often requires a grammar transformation
- usually lookahead $= 1$

Shift Reduce Parser

- bottom-up parser
- uses a parser table to determine the next operation
- parser table gets the upper state of the stack and the lookahead as input and returns the operation

Shift Reduce Parser

- uses a push-down automaton to analyse the syntax of the input
- notation: $\alpha \bullet au$:
  - α represents the already read and partially processed input (on the stack)
  - au represents the tokens that are not yet analysed
- possible operations:
  - *shift*: read the next token and switch to the state $\alpha a \bullet u$
  - *reduce*:
    1. detect the tail α_2 of α as the right side of the production rule $A \to \alpha_2$
    2. remove α_2 from the top of the stack and put A on the stack

    transforms $\alpha_1 \alpha_2 \bullet au$ into $\alpha_1 A \bullet au$ with the production rule $A \to \alpha_2$

Example: Grammar & items

- grammar:

$$
\begin{aligned}
S' &\to S\ \textbf{eof} && (1)\\
S &\to (S) && (2)\\
&\mid [S] && (3)\\
&\mid \textbf{id} && (4)
\end{aligned}
$$

- items:

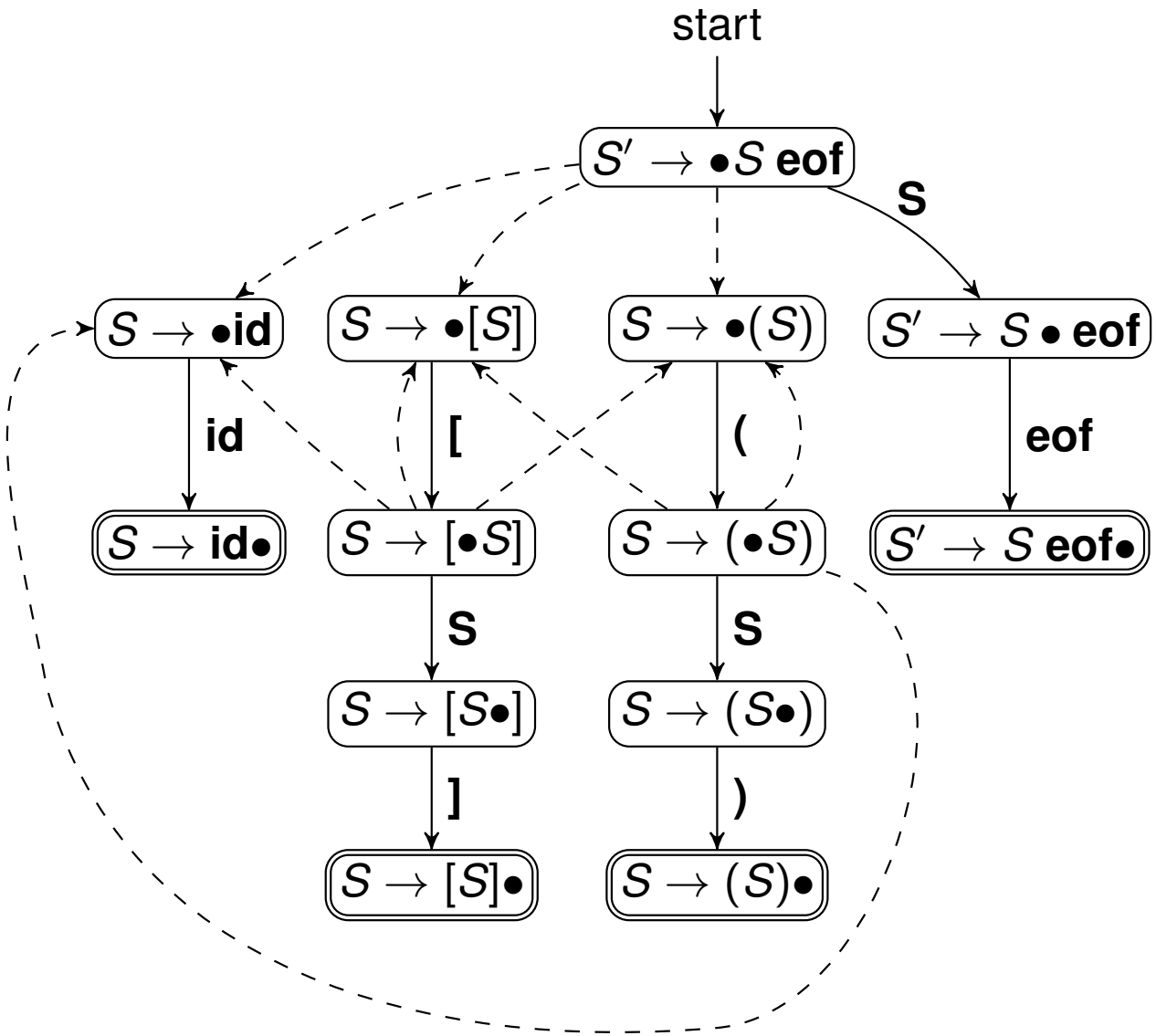
$S' \to \bullet\ S\ \textbf{eof}$ $S \to \bullet\ (S)$ $S \to \bullet\ [S]$ $S \to \bullet\ \textbf{id}$

$S' \to S \bullet\ \textbf{eof}$ $S \to (\bullet\ S)$ $S \to [\bullet\ S]$ $S \to \textbf{id} \bullet$

$S' \to S\ \textbf{eof} \bullet$ $S \to (S \bullet\ )$ $S \to [S \bullet\ ]$

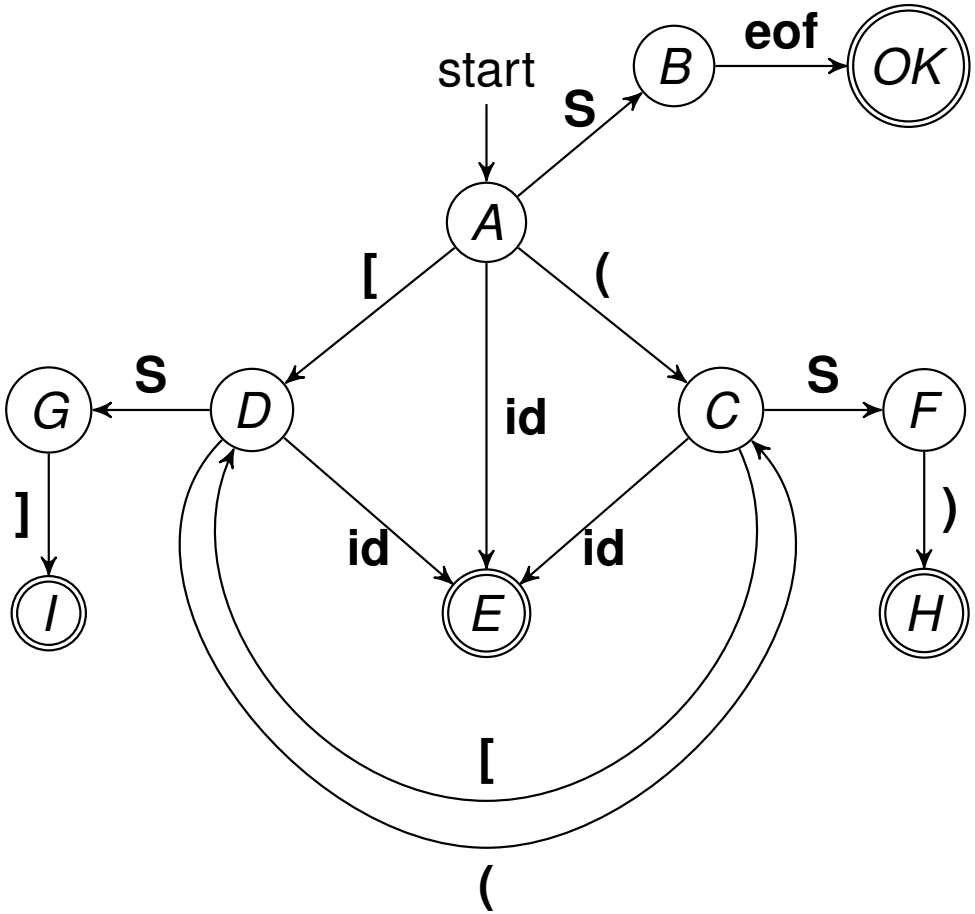
$S \to (S) \bullet$ $S \to [S] \bullet$

Example: Non-deterministic automaton



Example: Deterministic automaton

- every state is a set of the states of the non-deterministic automaton



- *H,I,E* and *OK* contain reduce items

Example: Parser table

- rows: states of the deterministic automaton
- columns: terminal and nonterminal symbols
- the resulting parser table:

	(	)	[	]	**id**	S	**eof**
A	C		D		E	B	
B							OK
C			D		E	F	
D	C				E	G	
E		r(4)		r(4)			r(4)
F		H					
G				I			
H		r(2)		r(2)			r(2)
I		r(3)		r(3)			r(3)

Shift Reduce Parser

- needs *LR(k)* grammar but modern grammars often are in that form
- often created by parser generators because they are complex

Parser Generators

- parser generators automatically generate parsers for a grammar or a regular expression.
- often *LR* or *LALR* parsers
- Yacc ("yet another compiler compiler") and Bison are famous LALR-parser generators
- Bison generates two output files:
  1. executable code
  2. grammar and parser table

Example: Input for Bison

- input file consists of three parts that are seperated with %% :
  1. declarations of the tokens
  2. production rules
  3. C-function that executes the parser (optional)

```
% token ID
%%
   S : '(' S ')'
     | '[' S ']'
     | ID
     ;
%%
```

Example: Output of Bison

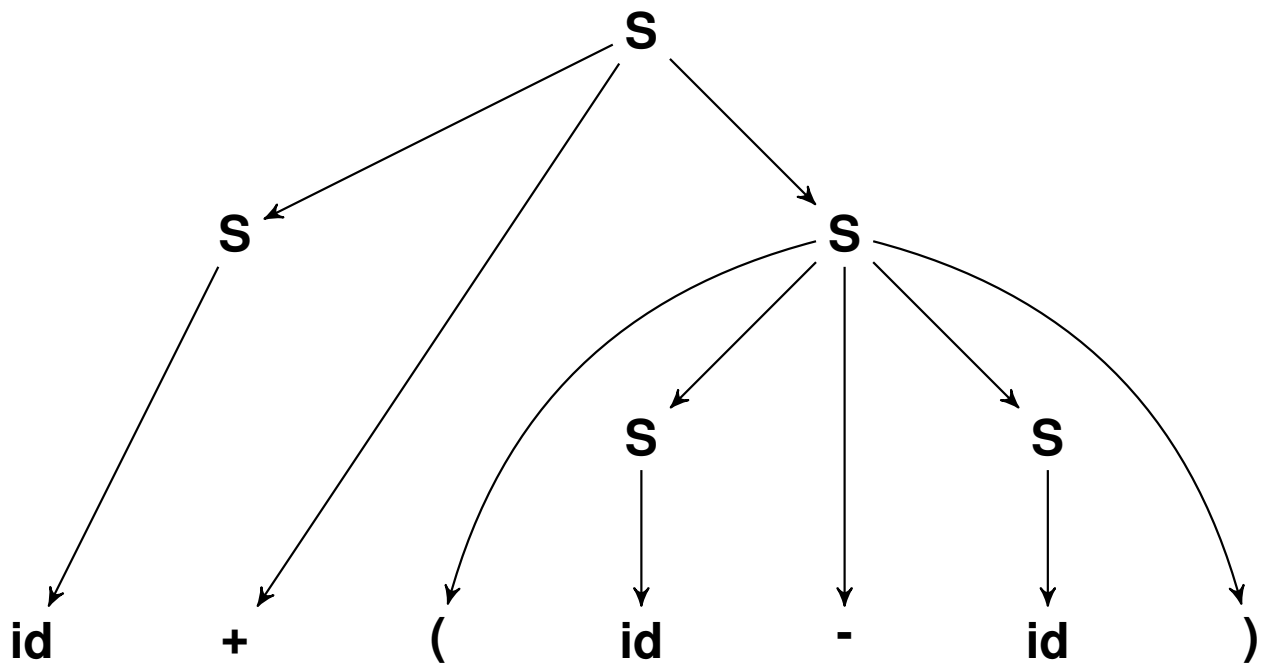
```
Grammar
    0 $accept: S $end
    1 S: '(' S ')'
    2  | '[' S ']'
    3  | ID

[...]

State 0
    0 $accept: . S $end
    ID   shift, and go to state 1
    '('  shift, and go to state 2
    '['  shift, and go to state 3
    S  go to state 4
State 1
    3 S: ID .
    $default  reduce using rule 3 (S)
State 2
    1 S: '(' . S ')'
    ID   shift, and go to state 1
    '('  shift, and go to state 2
    '['  shift, and go to state 3
    S  go to state 5

[...]
```

Parse Tree

- describes the derivation of the expression from the grammar
- important for the compiling process

Example

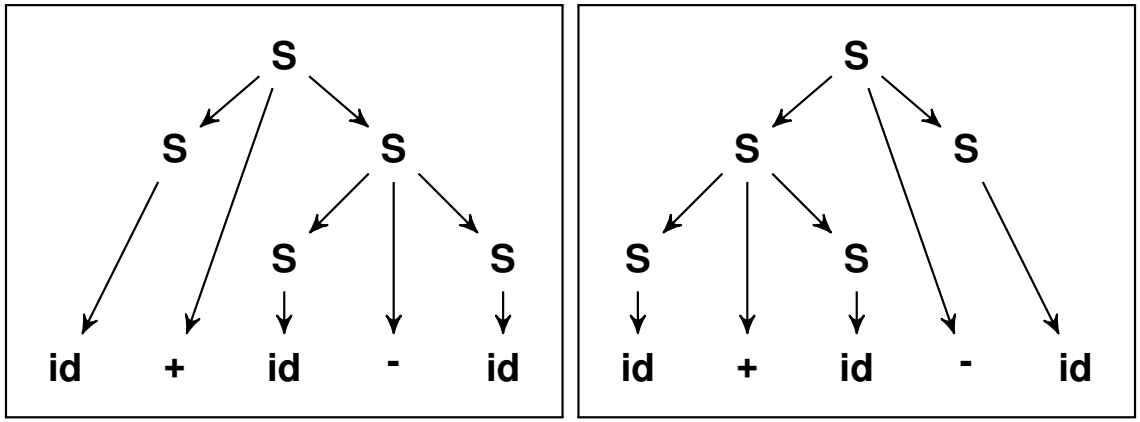
unambigous grammar: $S \rightarrow S + S \mid (S - S) \mid \mathbf{id}$
expression: $\mathbf{id} + (\mathbf{id} - \mathbf{id})$



Parse Tree

Example

ambigous grammar: $S \rightarrow S + S \mid S - S \mid \textbf{id}$
expression: $\textbf{id} + \textbf{id} - \textbf{id}$



Conclusion

- choice of parser type is important because each one has its advantages
- parser development has become much easier with parser generators

Questions?