Parsing $LL(1)$

$LL(1)$ Grammars

- We have seen that
  - a grammar is $LL(1)$ if and only if for all rules $A \to \beta \mid \gamma$ (with $\beta \neq \gamma$) we have $la(A \to \beta) \cap la(A \to \gamma) = \emptyset$
  - we can compute $la(\cdot)$
  - some non-$LL(1)$ CFGs can be turned into $LL(1)$ grammars

- Now, we we will look at how to implement efficient parsers for $LL(1)$ grammars

Parsing with Deterministic Top-Down Automaton

- The first way to implement an $LL(1)$ parser is based on a deterministic top-down automaton that uses the lookahead sets
- For $LL(1)$ grammar $< \Sigma, N, P, S >$ the automaton is defined by
  - Input alphabet Σ, pushdown alphabet $X = N \cup \Sigma$, output alphabet U = the rule numbers 1,2,3,...
  - States $\Sigma^* \times X^* \times U^*$ with initial state (w, S, ε) for $w \in \Sigma^*$ and final state final state $(\varepsilon, \varepsilon, u)$ where $u \in U^*$
  - Actions:
    - **Expanding** rule $A \rightarrow \beta$ with number i:
      $(aw, A\alpha, z) \rightarrow (w, \beta\alpha, z\ i)$ if $a \in la(A \rightarrow \beta)$
    - **Matching** terminal symbol $a \in \Sigma$:
      $(aw, a\alpha, z) \rightarrow (w, \alpha, z)$
    - **Accepting the entire input** if state $(\varepsilon, \varepsilon, \cdot)$ is reached
    - **Reporting an error** in any other case

Example

- Grammar

  $E \to TE'$ (1)

  $E' \to +TE' \mid \varepsilon$ (2,3)

  $T \to FT'$ (4)

  $T' \to * FT' \mid \varepsilon$ (5,6)

  $F \to (E) \mid a \mid b$ (7,8,9)

Compute $la(\cdot)$ for all rules:

$la(E \to TE') = \{\, (\,, a, b\}$

$la(E' \to +TE') = \{+\}$

$la(E' \to \varepsilon) = \{\varepsilon,)\,\}$

$la(T \to FT') = \{\, (\,, a, b\}$

$la(T' \to * FT') = \{\, * \,\}$

...

- Leftmost analysis of $(a) * b$
  - Initial state: $((a) * b, E, \varepsilon)$
  - Expand E to TE' because the next input symbol "(" is in $la(E \to TE')$: $((a) * b, TE', 1)$
  - ...
  - Final state reached: $(\varepsilon, \varepsilon, 147148635963)$

Action table of the automaton

- Because $la(\cdot)$ does not depend on the input, we can write all possible actions of the automaton in the form of a table:

act	E	E'	T	T'	F	a	b	(	)	$*$	$+$	ε
a	$(TE', 1)$		$(FT', 4)$		$(\mathrm{a}, 8)$	pop						
b	$(TE', 1)$		$(FT', 4)$		$(\mathrm{b}, 9)$		pop					
(	$(TE', 1)$		$\boxed{(FT', 4)}$		$((E), 7)$			pop				
)		$(\varepsilon, 3)$		$(\varepsilon, 6)$					pop			
$*$				$(*FT', 5)$						pop		
$+$		$(+TE', 2)$		$(\varepsilon, 6)$							pop	
ε		$(\varepsilon, 3)$		$(\varepsilon, 6)$								accept

Source: RWTH Aachen

- For the state $((a) * b, TE', 1)$ the next input symbol "(" tells us in which row of the table to look and the next grammar symbol T tells us in which column of the table to look
  - (X, n) = do an expand action to X using rule n
  - pop = do a match action
  - accept = do an accept action
  - empty = report an error

Complexity of $LL(1)$ parsing

- Space and time complexity $O(length\ of\ w)$ for input w
- Idea of the proof (if we ignore rules of the form $A \to \varepsilon$):
  - Parsing w requires $length(w)$ matching steps
  - Every matching step is preceded by at most $|N|$ expansion steps

Recursive-Descent Parsing

- Idea: Do what the automaton does by using the call stack of programming languages
- We start with a very simple example. We want to parse the language generated by this grammar

$$S \rightarrow a\ b$$

- Let's assumer the lexer looks like this:

The lexer could put additional information in the symbol object, for example the line number (useful for error messages)

```
enum Token { a, b, END };
class Symbol {
    Token token;
    Object attribute;   // not needed for this simple example
}
class Lexer {
    Symbol nextSymbol();  // returns END if end of input reached.
                          // whitespaces are not returned.
}
```

Recursive-Descent Parsing: Simple Example

$$S \rightarrow a\ b$$

```
Symbol lookahead;

void main()  throws ParserException {
   lookahead = lexer.nextSymbol();
   match(Token.a);
   match(Token.b);
   match(Token.END);
}

Symbol match(Token token)  throw ParserException {
    if(lookahead.token!=token) {
      throw new ParserException("No match");
    } else {
      Symbol matchingSymbol = lookahead;
      lookahead = lexer.nextSymbol();
      return matchingSymbol;
    }
}
```

Recursive-Descent Parsing: Complex Example

- Let's parse Java-style method definitions like

```
int **rectangleArea**(int a, int b) {
  return a*b;
}
```

- An incomplete grammar (terminal symbols written in lower-case)

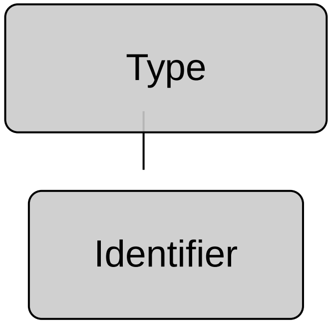
$$Method \rightarrow Type\ identifier\ (\ Params\ )\ Block$$
$$Type \rightarrow identifier \mid \ldots$$
$$Params \rightarrow \varepsilon \mid Param\ MoreParams$$
$$MoreParams \rightarrow \varepsilon \mid\ ,\ Param\ \ MoreParams$$
$$Param \rightarrow Type\ identifier$$
$$Block \rightarrow \{\ Stmts\ \}$$
$$Stmts \rightarrow Stmt\ Stmts \mid \varepsilon$$
$$Stmt \rightarrow return\ Expression; \mid \ldots$$
$$Expression \rightarrow \ldots$$

Parsing a type (simplified version)

$$Type \rightarrow identifier$$

Examples: int Object Point2D

```
class Type {
  String identifier;
}

Type parseType() throw ParserException {
  Symbol identifier = match(Token.identifier);
  return new Type((String)identifier.attribute);
}
```

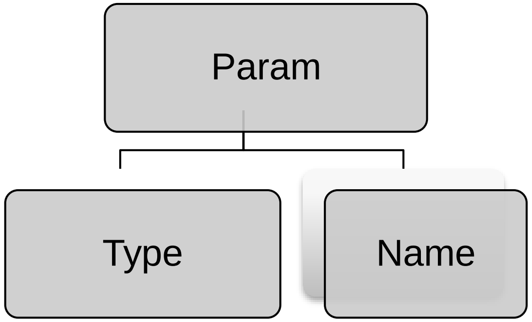


Parsing a parameter

$$Param \rightarrow Type\ identifier$$

Example: int x

```
class Param {
  Type type;
  String name;
}

Param parseParam() throw ParserException {
  Type type = parseType();
  Symbol identifier = match(Token.identifier);
  return new Param(type,(String)identifier.attribute);
}
```

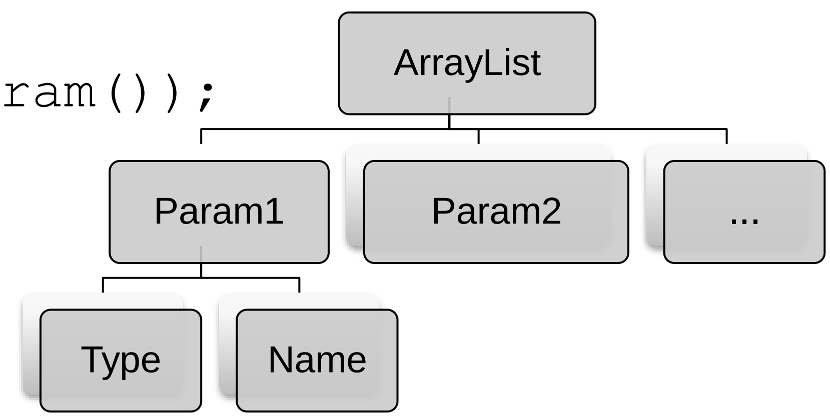


Parsing a parameter list

$$Params \rightarrow Param\ Params \mid \varepsilon$$
$$MoreParams \rightarrow \varepsilon \mid ,Param\ MoreParams$$

Example: int a, int b

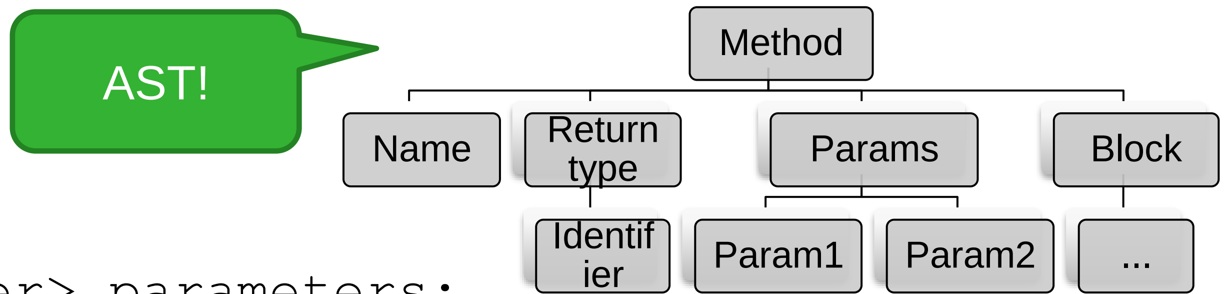
```
ArrayList<Param> parseParams() throw ParserException {
  ArrayList<Param> parameters = new ArrayList<>();

  // check if next symbol ∉ la(Params → ε)
  if(lookahead.token!=Token.ClosingParenthesis) {
    parameters.add(parseParam());
    while(lookahead.token==Token.Comma) {
        match(Token.Comma);
        parameters.add(parseParam());
    }
  }
  return parameters;
}
```



Parsing a method

$$Method \;\rightarrow Type\ identifier\ (\ Params\ )\ Block$$

Example: int rectangleArea(int a, int b) { … }




```
class Method {
  Identifier name;
  Type returnType;
  ArrayList<Parameter> parameters;
  Block body;
}
Method parseMethod() throw ParserException {
  Type returnType = parseType();
  String name = (String)match(Token.identifier).attribute;
  match(Token.OpenParenthesis);
  ArrayList<Param> params = parseParams();
  match(Token.CloseParenthesis);
  Block body = parseBlock();
  return new Method(name, returnType, params, body);
}
```

Recursive-Descent Parsing: Pros and Cons

- Pros:
  - When writing it by hand, you learn how parsing works ☺
  - You can directly create the AST
  - You can do very specific error handling
  - Very flexible
    - For simple languages, you can already do semantic analysis during parsing
    - If parts of the grammar are not $LL(1)$ you can write special parser code for them
- Cons:
  - Easy to make mistakes. Are you really implementing the right grammar?
  - Lot of code to write. And if you make major changes to the grammar, you have to rewrite most of the code

    → Advantage of parser generator tools

Writing a parser by hand using Parser Combinators

- When you write a recursive-descent parser you will notice that you are repeating work
- For example, the parser functions for A and B with rules

$$A \rightarrow aA \mid \varepsilon$$
$$B \rightarrow bB \mid \varepsilon$$

  will look very similar. The only difference is the a vs b
- *Parser Combinator* tools provide useful functions in a library (or even their own definition language) to build complex parsers by combining smaller parsers https://github.com/norswap/autumn
- Example: $C \rightarrow A\ B$ is written with such tools as

```
Parser parserA = zeroOrMore(()->match(Token.a));
Parser parserB = zeroOrMore(()->match(Token.b));
Parser parserC = combineSequential(parserA,parserB);
parserC.parse("aaaabbbb");
```

- Parsers using parser combinators are often slower than recursive-descent parsers
- Debugging is harder: if `zeroOrMore` shows an error what happened?

Parser Generator Tools

- Parser Generator Tools like ANTLR https://www.antlr.org/ take a grammar definition file and generate parser code for it in Java or C
- Example from their website:

```
grammar Expr;
prog:   (expr NEWLINE)* ;
expr:   expr ('*'|'/') expr
    |   expr ('+'|'-') expr
    |   INT
    |   '(' expr ')'
    ;
NEWLINE : [\r\n]+ ;
INT     : [0-9]+ ;
```

ANTLR will automatically eliminate the left recursion

No real difference between lexer and parser rules