

Compiler Design

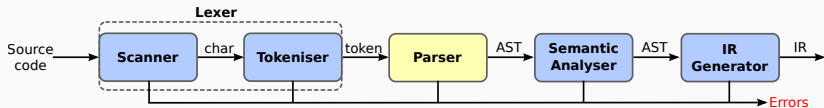
Lecture 5: Top-Down Parsing

Christophe Dubach

Winter 2022

Timestamp: 2022/01/11 14:38:11

The Parser



- Checks grammatical correctness of the stream of words/tokens produced by the lexer
- Outputs the AST (Abstract Syntax Tree) which represents the input program

Table of contents

Context-Free Grammar (CFG)

- Definition

- RE to CFG

Recursive-Descent Parsing

- Writing a Parser

LL(K) grammars

- Need for lookahead

- LL(1) property

- LL(K)

Problems with LL(k) parsers

Context-Free Grammar (CFG)

Context-Free Grammar (CFG)

Definition

Specifying syntax with a grammar

A Context-Free Grammar (CFG) is used to specify the syntax

Definition

A Context-Free Grammar G is a quadruple (S, N, T, P) where:

- S is a start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of production or rewrite rules where **only a single non-terminal appears on the left-hand side** $P : N \rightarrow (N \cup T)^*$

Context-Free Grammar (CFG)

RE to CFG

From Regular Expression to Context-Free Grammar

- **Kleene closure A^* :**

replace A^* to A_{rep} in all production rules and add

$$A_{rep} = A A_{rep} \mid \epsilon$$

as a new production rule.

- **Positive closure A^+ :**

replace A^+ to A_{rep} in all production rules and add

$$A_{rep} = A A_{rep} \mid A$$

as a new production rule.

- **Option $[A]$:**

replace $[A]$ to A_{opt} in all production rules and add

$$A_{opt} = A \mid \epsilon$$

as a new production rule.

Example: function call

```
funcall ::= IDENT "(" [ IDENT ("," IDENT)* ] ")"
```

after removing the option:

```
funcall ::= IDENT "(" arglist ")"  
arglist ::= IDENT ("," IDENT)*  
          |  $\epsilon$ 
```

after removing the closure:

```
funcall ::= IDENT "(" arglist ")"  
arglist ::= IDENT argrep  
          |  $\epsilon$   
argrep  ::= "," IDENT argrep  
          |  $\epsilon$ 
```

Recursive-Descent Parsing

Main idea

Steps to derive a **syntactic analyser** (*i.e.* half a parser) for a context free grammar expressed in an EBNF style:

- Convert all the regular expressions as seen;
- Implement a function for each non-terminal symbol A .
This function recognises sentences derived from A ;
- Recursion in the grammar corresponds to recursive calls of the created functions.

This technique is called recursive-descent parsing or predictive parsing.

Recursive-Descent Parsing

Writing a Parser

Parser class (pseudo-code)

```
Token currentToken;  
  
void error(TokenClass... expected) { /* ... */}  
  
boolean accept(TokenClass... expected) {  
    return (currentToken ∈ expected);  
}  
  
Token expect(TokenClass... expected) {  
    Token token = currentToken;  
    if (accept(expected)) {  
        nextToken(); // modifies currentToken  
        return token;  
    }  
    else  
        error(expected);  
}
```

CFG for function call

```
funcall ::= IDENT "(" arglist ")"
arglist ::= IDENT argrep
           |  $\epsilon$ 
argrep  ::= "," IDENT argrep
           |  $\epsilon$ 
```

Recursive-Descent Parser

```
void parseFunCall() {
    expect(IDENT);
    expect(LPAR);
    parseArgList();
    expect(RPAR);
}

void parseArgList() {
    if (accept(IDENT)) {
        nextToken();
        parseArgRep();
    }
    // else nothing to do
}

void parseArgRep() {
    if (accept(COMMA)) {
        nextToken();
        expect(IDENT);
        parseArgRep();
    }
    // else nothing to do
}
```

Recursive vs Iterative approaches

Project hint: sometime you may want to keep the EBNF syntax and use an iterative approach rather than a recursive one as this might simplify your code.

Example: function call

```
funcall ::= IDENT "(" [ IDENT ("," IDENT)* ] ")"
```

Recursive-Descent Parser

```
void parseFuncall() {  
    expect(IDENT);  
    expect(LPAR);  
    if (accept(IDENT)) {  
        nextToken();  
        while (accept(COMMA)) {  
            nextToken();  
            expect(IDENT);  
        }  
        expect(RPAR);  
    }  
}
```

LL(K) grammars

LL(K) grammars

Need for lookahead

Consider the following bit of grammar

```
stmt      ::= assign ";"
           | funccall ";"
funccall  ::= IDENT "(" arglist ")"
assign    ::= IDENT "=" exp
```

```
void parseAssign() {
    expect(IDENT);
    expect(EQ);
    parseExp();
}

void parseStmt() {
    ???
}

void parseFunCall() {
    expect(IDENT);
    expect(LPAR);
    parseArgList();
    expect(RPAR);
}
```

If the parser picks the wrong production, it may have to backtrack.
Alternative is to look ahead to pick the correct production.

LL(K) grammars

LL(1) property

How much lookahead is needed?

- In general, an arbitrarily large amount

Fortunately:

- Large subclasses of CFGs can be parsed with limited lookahead
- Most programming language constructs fall in those subclasses

Among the interesting subclasses are LL(1) grammars.

LL(1)

Left-to-Right parsing;

Leftmost derivation; (i.e. apply production for leftmost non-terminal first)

only 1 current symbol required for making a decision.

Basic idea: given $A \rightarrow \alpha|\beta$, the parser should be able to choose between α and β .

First sets

For some symbol $\alpha \in N \cup T$, define $\text{First}(\alpha)$ as the set of symbols that appear first in some string that derives from α :

$$x \in \text{First}(\alpha) \text{ iif } \alpha \rightarrow \cdots \rightarrow x\gamma, \text{ for some } \gamma$$

The $LL(1)$ property: if $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like:

$$\text{First}(\alpha) \cap \text{First}(\beta) = \emptyset$$

This would allow the parser to make the correct choice with a lookahead of exactly one symbol! (almost, see next slide!)

What about ϵ -productions (the ones that consume no symbols)?

G ::= C b

C ::= A

| B

A ::= a

| ϵ

B ::= b

input1: ab

input2: b

Both inputs are correct.

However, when seeing the b in the second example, the parser does not know whether to go down the A derivation or B derivation:

- In the case of A, we could choose the ϵ and consume nothing, and the b will be consumed in G (which is the only valid derivation);
- In the case of B, we could directly consume the b, but then we will have a problem later on and would need to **backtrack**.

Therefore, **the parser may have to backtrack** since it needs to try out different paths.

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\epsilon \in \text{First}(\alpha)$, then we need to ensure that $\text{First}(\beta)$ is disjoint from $\text{Follow}(\alpha)$.

$\text{Follow}(\alpha)$ is the set of all terminal symbols in the grammar that can legally appear immediately after α .

(See EaC§3.3 for details on how to build the First and Follow sets.)

Let's define $\text{First}^+(\alpha)$ as:

- $\text{First}(\alpha) \cup \text{Follow}(\alpha)$, if $\epsilon \in \text{First}(\alpha)$
- $\text{First}(\alpha)$ otherwise

LL(1) grammar

A grammar is $LL(1)$ iff $A \rightarrow \alpha$ and $B \rightarrow \beta$ implies:

$$\text{First}^+(\alpha) \cap \text{First}^+(\beta) = \emptyset$$

Given a grammar that has the $LL(1)$ property:

- each non-terminal symbols appearing on the left hand side is recognised by a simple routine;
- the code is both simple and fast.

Predictive Parsing

Grammar with the $LL(1)$ property are called *predictive grammars* because the parser can “predict” the correct expansion at each point. Parsers that capitalise on the $LL(1)$ property are called *predictive parsers*. One kind of predictive parser is the *recursive descent* parser.

LL(K) grammars

LL(K)

Sometimes, we might need to lookahead one or more tokens.

LL(2) Grammar Example

```
stmt      ::= assign ";"  
           | funccall ";"  
funccall ::= IDENT "(" arglist ")"  
assign   ::= IDENT "=" exp
```

```
void parseStmt() {  
    if (accept(IDENT)) {  
        if (lookAhead(1) == LPAR)  
            parseFunCall();  
        else if (lookAhead(1) == EQ)  
            parseAssign();  
        else  
            error();  
    }  
    else  
        error();  
}
```

Problems with LL(k) parsers

Non-distinct first set in the grammar

Example

```
Stmt ::= Assign
      | Exp ";"
Assign ::= Exp "=" Exp ";"
```

How do you choose between assignment or expression?

```
void parseStmt() {
    if (accept(first(Exp) ??))
        parseAssign();

    else if (accept(first(Exp) ??))
        parseExp();
}
```

What about using a lookahead?

⇒ not possible since Exp can be of any length.

Left factorization

Rewrite : $A \rightarrow \alpha\beta|\alpha\gamma|\dots$

Into:

$A \rightarrow \alpha A'$

$A' \rightarrow (\beta|\gamma)$

May need to apply this indirectly.

```
Stmt ::= Assign
      | Exp ";"
Assign ::= Exp "=" Exp ";"
```

becomes:

```
Stmt ::= Exp Stmt'
Stmt' ::= Assign | ";"
Assign ::= "=" Exp ";"
```

```
void parseStmt() {
    parseExp();
    parseStmtPrime();
}

void parseStmtPrime() {
    if (accept(EQUAL))
        parseAssign();
    else
        expect(SC);
}

void parseAssign() {
    expect(EQUAL);
    parseExp();
    expect(SC);
}
```

Beware of left recursion!

Left Recursion

```
Expr ::= Expr Op Expr
      | "(" Expr ")"
      | Number
Op    ::= '+' | '*'
```

```
void parseExpr() {
    if (accept(LPAR)) {
        expect(LPAR);
        parseExpr();
        expect(RPAR);
    }
    else if (accept(DIGIT))
        parseNumber();
    else if (accept(LPAR, DIGIT))
        parseExpr();
}
```

Example inputs: 1+1

Not recognized!

What about first checking for Expr?

Infinite recursion!

Luckily, we can transform this grammar...

Removing Left Recursion

You can use the following rule to remove direct left recursion:

$$A \rightarrow A\alpha_1|A\alpha_2|\dots|A\alpha_m|\beta_1|\beta_2|\dots|\beta_n$$

where β_i does not start with an A and $\alpha_i \neq \varepsilon$

can be rewritten into:

$$A \rightarrow \beta_1A'|\beta_2A'|\dots|\beta_nA'$$

$$A' \rightarrow \alpha_1A'|\alpha_2A'|\dots|\alpha_mA'|\varepsilon$$

Hint

Use this to deal with binary operators, arrayaccess and fieldaccess in the project

Left recursive grammar

```
Expr ::= Expr Op Expr
      | "(" Expr ")"
      | Number
Op    ::= '+' | '*'
```

Equivalent non-left recursive grammar

```
Expr  ::= "(" Expr ")" Expr'
      | Number Expr'
Expr' ::= Op Expr Expr'
      | ε
Op    ::= "+" | "*"
```

```
void parseExpr() {
    if (accept(LPAR)) {
        expect(LPAR);
        parseExpr();
        expect(RPAR);
        parseExprPrime();
    }
    else if (accept(DIGIT)) {
        parseNumber();
        parseExprPrime();
    }
    else
        expect(LPAR, DIGIT); //error
}

void parseExprPrime() {
    if (accept(PLUS, MINUS)) {
        parseOp();
        parseExpr();
        parseExprPrime();
    }
}
```


To write a recursive descent parser, follow these steps:

1. Express the language syntax as an LL(k) CFG;
2. Left factorize the grammar if necessary;
3. Remove left recursion from the grammar if present;
4. Write the recursive parser using at most k lookaheads.

Your parser will never have to backtrack!

⇒ $O(N)$ time complexity

- Bottom-up parsing