

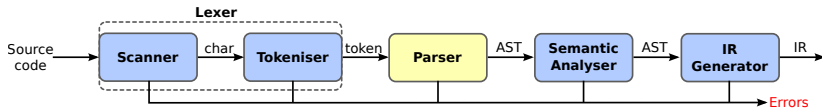
# Compiling Techniques

## Lecture 5: Top-Down Parsing

Christophe Dubach

26 September 2017

# The Parser



- Checks the stream of words/tokens produced by the lexer for grammatical correctness
- Determine if the input is syntactically well formed
- Guides checking at deeper levels than syntax
- Used to build an IR representation of the code

# Table of contents

- 1 Context-Free Grammar (CFG)
  - Definition
  - RE to CFG
- 2 Recursive-Descent Parsing
  - Main idea
  - Writing a Parser
  - Left Recursion
- 3 LL(K) grammars
  - Need for lookahead
  - LL(1) property
  - LL(K)

## Specifying syntax with a grammar

- Use Context-Free Grammar (CFG) to specify syntax

### Context-Free Grammar 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 is allowed on the left-hand side

$$P : N \rightarrow (N \cup T)^*$$

# 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  
          |  $\epsilon$ 
```

Steps to derive a syntactic analyser 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.

## 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();  
    }  
}  
  
void parseArgRep() {  
    if (accept (COMMA)) {  
        nextToken();  
        expect (IDENT);  
        parseArgRep();  
    }  
}
```

# Be aware of infinite recursion!

## Left Recursion

$$E ::= E "+" T \\ \quad | T$$

The parser would recurse indefinitely!

Luckily, we can transform this grammar to:

$$E ::= T ("+" T)^*$$

# Removing Left Recursion

You can use the following rule to remove left recursion:

$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_m | \beta_1 | \beta_2 | \dots | \beta_n$  where  
 $first(\beta_i) \cap first(A) = \emptyset$  and  $\varepsilon \notin first(\alpha_i)$

can be rewritten into:

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

Hint:

Use this to deal with arrayaccess and fieldaccess for the coursework

## Consider the following bit of grammar

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

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

```
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.

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)

1 symbol lookahead.

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

## 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  $\alpha$ :

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

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  $\epsilon$ -productions (the ones that consume no symbols)?

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

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

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

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

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

### LL(1) grammar

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

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

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.



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

## LL(2) Grammar Example

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

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

## Next lecture

- More about LL(1) & LL(k) languages and grammars
- Dealing with ambiguity
- Left-factoring
- Bottom-up parsing