City University London

IN2009

Language Processors

Week 3

Parsing I (syntax analysis)

Igor Siveroni

Before we start...

- Coursework: Due Friday 27 February Disregard last slides from previous session, I have changed the question slightly.
- Extra sheet (Cityspace) with regular expression exercises:
  - UK phone numbers
  - UK postcodes
  - Email addresses

Lab 1

- Objective:
  - Practice Java, OO and tree traversal.
  - A complete example of a programming language interpreter (language processing).
  - We may go back to this program - understand it!
  - Solution in CitySpace.

Session Plan

Session 3: Parsing (syntax analysis)

- syntax definition
  - context free grammars (BNF)
- parsing
- ambiguous grammars
- removal of left recursion
- top down recursive descent parsing
- extended BNF (EBNF)
- parsing using JavaCC

Syntax definition

- We need to recognise structures like expressions with parentheses, or nested statements:
  - (109+23) (1+(250+3))
  - **if (…) then if (…)** stms… **else** … **else** …
- How do we do this?

Syntax definition

- It is tempting to attempt to use regular expressions with abbreviations
  - digits = [0-9]+
  - sum = expr "+" expr
  - expr = "(" sum ")" | digits

Syntax definition

- But remember that regular expression abbreviations like digits are only abbreviations and are *substituted* directly (they are macros), so we would get
  - expr = "(" sum ")" | digits
  - expr = "(" *(expr "+" expr)* ")" | digits *(substite **sum**)*
  - expr = "(" (*("(" (expr "+" expr) ")" | digits)* "+" expr) ")" | digits *(substite **expr**, then what?)*
  - ...

Syntax definition

- An automaton cannot be created from such definitions.
- What we need is a notation where recursion does not mean abbreviation and substitution, but instead means ***definition***...
- Recursion gives additional expressive power.

Syntax definition

- Then, (1+(250+3)) can be recognised by our recursive definitions

```
expr  =>  "(" sum ")"
      =>  "(" expr "+" expr ")"       (using the sum definition)
      =>  "(" digits "+" expr ")"     (using the expr definition)
      =>  "(" 1 "+" expr ")"
      =>  "(" 1 "+" "(" sum ")" ")"
      =>  "(" 1 "+" "(" expr "+" expr ")" ")"
      =>  "(" 1 "+" "(" digits "+" expr ")" ")"
      =>  "(" 1 "+" "(" digits "+" digits ")" ")"
      =>  "(" 1 "+" "(" 250 "+" digits ")" ")"
      =>  "(" 1 "+" "(" 250 "+" 3 ")" ")"
```


Syntax definition

- Alternation within definitions is then not needed, since
  - **r = ab(c|d)e** is the same as:
  - **n = (c|d)** and **r = abne**
  - or even **n = c** with **n = d** with **r = abne**, so alternation not needed at all!
  - we will however retain alternation at the top level of definition.

Syntax definition

- repetition via Kleene closure * is not needed, since
  e= (abc)* is the same as **e=(abc)e** with **e=ε**
- this recursive notation is called *context-free grammars* or BNF (see Session 1)
  - recognised by pushdown automata (PDA); recognition is implemented in many ways
  - involves (implicitly or explicitly) building the concrete syntax (parse) tree, matching against the tokens produced by the lexical analyser
  - building the tree can be top-down or bottom-up
  - once again, a tool can produce a parser for us

Context-free grammars

- A *language* is a set of *strings*
- Each string is a finite sequence of *symbols* taken from a finite *alphabet*
- For parsing: symbols = lexical tokens, alphabet = set of token types returned by the lexical analyser
- A grammar describes a language
- A grammar has a set of *productions* of the form
  symbol → symbol symbol … symbol
- Zero or more symbols on RHS
- Each symbol either a *terminal* from the alphabet or a *non-terminal* (appears on LHS of some productions)
- No token ever on LHS of production
- One non-terminal distinguished as *start symbol* of the grammar

Syntax for straight-line programs

1	S → S ; S
2	S → id := E
3	S → print (L)
4	E → id
5	E → num
6	E → E + E
7	E → (S , E)
8	L → E
9	L → L , E

- a *context-free grammar*
- terminal symbols (tokens):
  id print num , ( ) := ; +
- non-terminal symbols:
  S E L
- start symbol S

Derivations

```
a := 7 ;
b := c + (d:= 5+6, d)
```

Repeatedly replace any non-terminal by one of its right-hand sides.

Leftmost derivation: Always replace the leftmost non-terminal.

Rightmost derivation: Always replace the rightmost non-terminal.

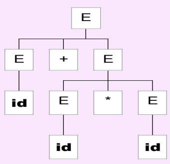
S
S ; S
S ; id := E
id := E ; id := E
id := num ; id := E
id := num ; id := E + E
id := num ; id := E + (S , E)
id := num ; id := id + (S , E)
id := num ; id := (id := E , E)
id := num ; id := (id := E + E , E)
id := num ; id := (id := E+E, id)
id := num ; id := (id := num+E, id)
id := num ; id := (id := num + num, id)

Concrete syntax derivations and parse trees

E → E * E | E / E | E + E | E - E | (E) | **id** | **num**

Leftmost derivation of **id + id * id** :

Concrete syntax tree (parse tree):

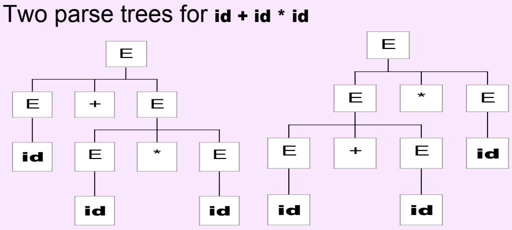


E
⇒ E * E
⇒ E + E * E
⇒ **id +** E * E
⇒ **id + id *** E
⇒ **id + id * Id**

Ambiguous grammars

E → E * E | E / E | E + E | E - E | (E) | **id** | **num**

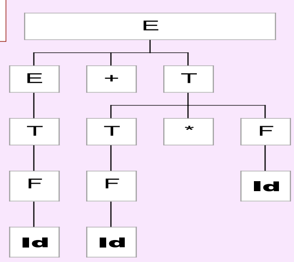
Two parse trees for **id + id * id**



Disambiguating the grammar

E → E + T
E → E - T
E → T
T → T * F
T → T / F
T → F

F → id
F → num
F → (E)



Only one tree now possible from this BNF for input string

id + id * id

Recursive descent parsing

- AKA "Top down" / Predictive
  - One recursive function per non-terminal
  - Each grammar production turns into one clause of a recursive function
  - Only works on grammars where the **first** terminal symbol of each grammatical construct provides enough information to choose the production

Recursive descent parsing

S → if E then S else S
S → begin S L
S → print E
L → end
L → ; S L
E → num = num

```
void eat(int t) {
 if (tok==t) advance();
 else  error();
}

void advance() {
  tok = getToken(); }
```

```
void S() {
   switch (tok) {
   case IF:
        eat(IF); E(); eat(THEN);
        S(); eat(ELSE); S(); break;
   case BEGIN: eat(BEGIN); S(); L();
   break;
   case PRINT: eat(PRINT); E();  break;
}
void E() { eat(NUM); eat(EQ); eat(NUM);
   }

void eat(int t) {
 if (tok==t) advance(); else  error();
}

void advance() { tok = getToken(); }
```

Appel 2002, (p46, Gram 3.11)

But... (p46, Gram 3.10)

- if we try to implement a recursive descent parser for the disambiguated expression grammar…

E → E + T	T → T * F	F → id
E → E - T	T → T / F	F → num
E → T	T → F	F → (E)

```
void E() {
        switch (tok) {
                case ???: E(); eat(PLUS); T(); break;
                case ???: E(); eat(MINUS); T(): break;
                case ???: T(); break;
        }}
```

- Problems:
  - **no initial terminal symbol** to tell us which production to choose
  - Even if we compute the FIRST sets, more then one production to choose from (due to left-recursion)

Eliminating left recursion

$X \rightarrow X\gamma$
$X \rightarrow \alpha$

can always be rewritten

$X \rightarrow \alpha X'$
$X' \rightarrow \gamma X'$
$X' \rightarrow$

S → E $	T → F T'	F → id
E → T E'	T' → * F T'	F → num
E' → + T E'	T' → / F T'	F → (E)
E' → - T E'	T' →	
E' →		

Sketch of resulting recursive descent parser

S → E $	T → F T'	F → id
E → T E'	T' → * F T'	F → num
E' → + T E'	T' → / F T'	F → (E)
E' → - T E'	T' →	
E' →		

```
void E() { T(); E'(); }

void E'() {
        switch (tok) {
                case PLUS: eat(PLUS); T(); E'(); break;
                case MINUS: eat(MINUS); T(); E'(); break;
                default: /* empty - that's ok */ break;
        }
}
```


Extended BNF (EBNF)

- A few additional operators to shorten definitions:
  - e_1 | e_2 | e_3 | ... : choice of e_1, e_2, e_3, etc
  - (…) bracketting allowed
  - […] : the expression in […] may be omitted
    - (may also be written as (...)?).
  - (e)+ : One or more occurrences of e
  - (e)* : Zero or more occurrences of e

Extended BNF (EBNF)

- Note that these may be nested within each other, so we can have
  - ((e_1 | e_2)* [e_3]) | e_4
- examples:

IfStatement → **if (** Expression **)** StatementBlock
[**else** StatementBlock]
StatementBlock → **{** (Statement)+ **}**

Expression grammar in EBNF

$E \rightarrow E + T$ $E \rightarrow E - T$ $E \rightarrow T$	$T \rightarrow T * F$ $T \rightarrow T / F$ $T \rightarrow F$	$F \rightarrow$ id $F \rightarrow$ num $F \rightarrow$ (E)	*Original*
$E \rightarrow T E'$ $E' \rightarrow + T E'$ $E' \rightarrow - T E'$ $E' \rightarrow$	$T \rightarrow F T'$ $T' \rightarrow * F T'$ $T' \rightarrow / F T'$ $T' \rightarrow$	$F \rightarrow$ id $F \rightarrow$ num $F \rightarrow$ (E)	*Left-recursion eliminated*
$E \rightarrow T (+ T \mid - T)^*$	$T \rightarrow F (* T \mid / T)^*$	$F \rightarrow$ id $F \rightarrow$ num $F \rightarrow$ (E)	*EBNF*

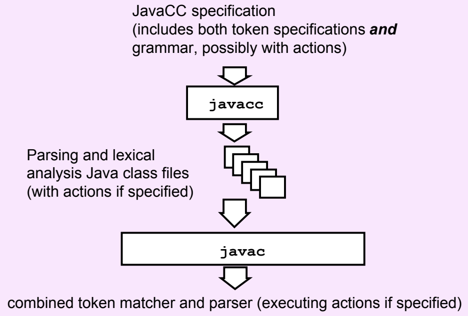
JavaCC:parser & lexical analysis

- Fortunately, we don't have to hand-code parsers…
- Given an (E)BNF grammar, software tools like JavaCC will produce a parser for us.
- Reminder – lexical analysis:
  - tokens defined by regular expressions are recognised by finite state automata (FSA) (see previous session)

JavaCC:parser & lexical analysis

- Fortunately, we don't have to draw out a FSA and implement it to recognise tokens, because, given regular expressions, tools can produce a token matcher program for us
- In our case, given token definitions, our tool JavaCC will produce a lexical analysis method which simulates a FSA and matches tokens and sends them to the parser…

JavaCC



JavaCC

- JavaCC is a *parser generator*. Given as input a set of token definitions, a programming language syntax grammar, and a set of actions written in Java, it produces a Java program which will perform lexical analysis to find tokens and then parse the tokens according to the grammar and execute the actions as appropriate.

JavaCC

- it works on LL(1) grammars (no need to understand this definition), which are similar to those that recursive descent works for.
- it requires a non-ambiguous grammar with left-recursion removed, so we use the techniques from earlier this session.

For the record: LL(1) grammar
Left-to-right parse, *leftmost* derivation, *1* symbol lookahead

JavaCC BNF example

E → T E'

E' → + T E'
E' → - T E'
E' →

T → F T'

T' → * F T'
T' → / F T'
T' →

F → num
F → (E)

```
void E() :
{}
{
  T() Eprime()
}

void Eprime() :
{}
{
    ( "+" T() Eprime() )
  | ( "-" T() Eprime() )
  | {} /* empty */
}
```

```
void T() :
{}
{
  F() Tprime()
}

void Tprime() :
{}
{
    ( "*" F() Tprime() )
  | ( "/" F() Tprime() )
  | {} /* empty */
}
```

```
void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

```
TOKEN :
{
  < NUM: (["0"-"9"])+ >
}
```


JavaCC EBNF example

E → T (+ T | - T)*

T → F (* T | / T)*

F → num
F → (E)

```
void E() :
{}
{
  T() ( "+" T() | "-" T() )*
}

void T() :
{}
{
  F() ( "*" F() | "/" F() )*
}
```

```
void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

```
TOKEN :
{
  < NUM: (["0"-"9"])+ >
}
```


JavaCC input file format (.jj)

PARSER_BEGIN(Parser-name)

Parser-name must be the same in all three places

class Parser-name { }

PARSER_END(Parser-name)

/* Lexical items (ie token definitions) – see previous examples */

Token-definitions

/* Grammar rules – in a stylised form of EBNF (see next slide). */

Syntax-definitions

JavaCC Syntax-definitions

A BNF production: non-terminal-name -> right-hand-side is written:

java_return_type non-terminal-name (java_parameter_list) : (1)
java_block (2)
{ expansion_choices } (3)

(1) gives the name of the non-terminal being defined

The rest of (1) looks like a Java method declaration. Using this feature we can cause values to be passed up and down the parse tree while the parse takes place (up via return values and down via parameters).

(2) (java_block) introduces some Java code which is usually used to declare variables for use in the production

(3) is the EBNF definition and actions…see next slide

JavaCC EBNF expansion_choices

expansion | expansion | ... where the `|' separates alternatives.

expansion expansion ...	matches first expansion then second and so on
(expansion_choices)*	matches zero or more expansion_choices
(expansion_choices)+	matches one or more expansion_choices
(expansion_choices)?	matches expansion_choices or empty string
[expansion_choices]	ditto (ie same as ?)
regexp	matches the token matched by the regexp
java_id = regexp	ditto, assigning token to java_id
non-terminal-name (…)	matches the non-terminal
java_id = non-terminal-name (…)	ditto, assigning returned value to java_id

The java_id will usually be declared in the java_block.

Any of these expansions may be followed by some Java code written in {...} and this code (often called an action) will be **executed** when the generated parser matches the expansion.

JavaCC example: Exp.jj file

```
PARSER_BEGIN(Exp)

public class Exp {
}

PARSER_END(Exp)

SKIP :
{
  " " | "\t" | "\r"
}

TOKEN :
{
  < NUM: (["0"-"9"])+ > | < EOL: "\n" >
}

void S() :
{}
{
    E() <EOL>
  | <EOL>
  | <EOF>
}
```

```
void E() :
{}
{
  T() ( "+" T() | "-" T() )*
}

void T() :
{}
{
  F() ( "*" F() | "/" F() )*
}

void F() :
{}
{
    <NUM>
  | "(" E() ")"
}
```

JavaCC example: `Main.java` file

```
public class Main {
  public static void main(String args[]) throws ParseException {
    Exp parser = new Exp(System.in);
    try {
      System.out.println("Type in an expression on a single line.");
      parser.S();
      System.out.println("Expression parser - parse successful");
    } catch (ParseException e) {
      System.out.println("Expression parser - error in parse");
    }
  }
}
```


What you should do now…

- Read, digest and understand chapter 3
  - don't worry about parsing tables & table generation
- Understand the JavaCC document (CitySpace) and how to write token regular expressions and EBNF definitions in JavaCC
- Now take a first look at the MiniJava language.
  - we'll be using this through the rest of the module

Next Lecture

- This session continued and...
- ***Parsing II (abstract analysis)***
- Monday 23 February, 2008
  - 11:00 - 12:50
  - C.350