1 Part III: Parsing

- Check, whether a sentence belongs to the language.
- Construct the abstract syntax tree (later).

- Top-Down Parsing
- Parsing with JavaCUP
- Bottom-Up Parsing

2 Parse Trees

- Nodes are non-terminals.
- Leaves are terminals.
- Branching corresponds to rules of the grammar.
- The leaves give a sentence of the input language.
- For every sentence of the language there is at least one parse tree.
- Sometimes we have more then one parse tree for a sentence.
- Grammars which allow more than one parse tree for some sentences are called ambiguous and are usually not good for compilation.

3 Examples

Ambigous grammar:

E ::= E "*" E | E "+" E | "1" | "(" E ")"

Unambigous grammar

E ::= E "+" T | T
T ::= T "*" F | F
F ::= "1" | "(" E ")"

4 Top-Down Parsing

- Recursive descent parsing.
- Predictive parsing.
- Regular languages are limited in that they cannot express nesting.
- Therefore, finite state machines in general cannot recognize context-free grammars.
- Let's try the hand-writing method anyway!

Example ::= IDENT Example NUMLIT | NUMLIT.

leads after simplification to the following parser:

```
void Example() {
    if (token == IDENT) {
        token = nextToken();
        Example();
        if (token == NUMLIT) {
            token = nextToken();
        } else {
            error();
        }
    } else if (token == NUMLIT) {
        token = nextToken();
    } else {
        error();
    }
}
```

```
void Example() {
    switch(token) {
    case IDENT:
        token = nextToken();
        Example();
        switch(token) {
        case NUMLIT:
            token = nextToken(); break;
        default:
            error(); break;
        }
        break;
    case NUMLIT:
        token = nextToken(); break;
    default:
        error(); break;
    }
}
```

5 Deriving a Parser from EBNF

To derive a parser from a context-free grammar written in EBNF style:

- Introduce one function **void** A() for each non-terminal A
- The task of A() is to recognize sub-sentences derived from A, or issue an error if no A was found.
- Translate all regular expressions on the right-hand-side of productions as before, but
  - every occurrence of a non-terminal B maps to B()
  - Recursion in the grammar translates naturally to recursion in the parser.
- This technique of writing parsers is called parsing by *recursive descent* or *predictive parsing*.

6 A Parser for Expressions

Expression ::= Expression "−" Term | Term.
Term ::= Term "/" Factor | Factor.
Factor ::= NUMLIT | "(" Expression ")".

```
void Expression() {
    if (token == NUMLIT || token == LPAREN) {
        Expression();
        if (token == MINUS) {
            token = nextToken();
        } else { error(); }
        Term();
    } else {
        Term();
    }
```

7 first(X), follow(X) and nullable

- first(X) are the terminals X can start with.
  - A terminal t is in first(X) if there is a parse tree, such that t is the leftmost leaf under X.
  - ϵ leaves do not count.
  - Example:
    A ::= "b" "c" | B "d".
    B ::= "a" | ϵ.
    first(A) = { b, a, d }
- follow(X) are terminals which can follow X.
  - A terminal t is in follow(X) if there is a parse tree such that t is the leftmost leaf after the leaves under X
  - Again, ϵ leaves do not count.
  - Example: follow(B) = { d }
- A non-terminal is nullable if it can derive the empty string (it may have only ϵ-leaves (Example: B is nullable)

8 How to compute first(X) and follow(X)?

A ::= B "x" C.

- first(B) $\subseteq$ first(A).
- If B is nullable then x $\in$ first(A).
- Naive method: compute first, follow and nullable for right-hand side and from that for A.
- Does not work for recursion!

  E ::= E "+" T | T.
- Idea: Start with empty sets and add elements until all conditions are satisfied.
- This is called a fixpoint algorithm (It runs until there are no more changes, until the solution is fix).

9 Exercise

S ::= E "$".
E ::= T "+" E | T.
T ::= "x".

Find the first and follow sets for T and E. Are there any nullable non-terminals?

10 Formal Definition: first(X), follow(X), nullable

first(X), follow(X) and nullable are the smallest sets with the following properties:

For each production X ::= Y1 .. Yk, $1 \leq i,\ \ j \leq k$:

> **if** { Y1, ..., Yk } ⊆ nullable
>
> > X ∈ nullable
>
> **if** { Y1, ..., Yi–1 } ⊆ nullable
>
> > first(X) = first(X) ∪ first(Yi)
>
> **if** { Yi+1, ..., Yk } ⊆ nullable
>
> > follow(Yi) = follow(Yi) ∪ follow(X)
>
> **if** { Yi+1, ..., Yj–1 } ⊆ nullable
>
> > follow(Yi) = follow(Yi) ∪ first(Yj)

11 Algorithm for first(X), follow(X) and nullable

```
nullable = ∅;
for each terminal t { first(t) = t; follow(t) = ∅; }
for each nonterminal Y { first(Y) = ∅; follow(Y) = ∅; }
repeat {
    nullable' = nullable; first' = first; follow' = follow;
    for each production X ::= Y1 .. Yk, 1 ≤ i, j ≤ k {
        if { Y1, ..., Yk } ⊆ nullable
            nullable =  nullable ∪ X;
        if { Y1, ..., Yi–1 } ⊆ nullable
            first(X) = first(X) ∪ first(Yi);
        if { Yi+1, ..., Yk } ⊆ nullable
            follow(Yi) = follow(Yi) ∪ follow(X);
        if { Yi+1, ..., Yj–1 } ⊆ nullable
            follow(Yi) = follow(Yi) ∪ follow(Yj);
    }
until (nullable = nullable', first = first', follow = follow');
```

12 Extending first und nullable to righthand sides

- $\mathsf{nullable}(\epsilon) = \mathsf{true}$
- $\mathsf{nullable}(tu) = \mathsf{false}$
  if t is a terminal
- $\mathsf{nullable}(Xu) = \mathsf{nullable}(X) \wedge \mathsf{nullable}(u)$
  if X is a non-terminal

- $\mathsf{first}(\epsilon) = \emptyset$
- $\mathsf{first}(tu) = \{t\}$
  if t is a terminal
- $\mathsf{first}(Xu) = \begin{cases} \mathsf{first}(X), & \neg\mathsf{nullable}(X) \\ \mathsf{first}(X) \cup \mathsf{first}(u), & \mathsf{nullable}(X) \end{cases}$
  if X is a non-terminal

13 LL(1)

A grammar is called LL(1), if for every production

$\mathsf{A} ::= u_1 \mid u_2 \mid ... \mid u_n$

- $\mathsf{first}(u_i) \cap \mathsf{first}(u_j) = \emptyset$ if $i \neq j$
- $\mathsf{nullable}(u_i) = \emptyset$ for at most one i
- $\mathsf{first}(u_i) \cap \mathsf{follows}(A) = \emptyset$ if $\mathsf{nullable}(u_j)$ and $i \neq j$

Basically, it has to be clear, which alternative to choose, by looking at 1 token.

For LL(1) grammars recursive descent parsing works!

14 Eliminating Left Recursion

```
Expression ::= Term { "–" Term }.
Term       ::= Factor { "/" Factor }.
Factor     ::= NUMLIT | "(" Expression ")".
```

```
void Expression() {
        Term();
        while (token == MINUS) {
                token = nextToken();
                Term();
        }
}
```

- Here we always need to know, whether to stay in the loop or to leave it.

15 Another Problem

Factor ::= IDENT | IDENT "[" Expression "]" | NUMLIT.

```
void Factor() {
        if (token == IDENT) {
                ??
        } else {
                if (token == NUMLIT) {
                        token = nextToken();
                } else {
                        error();
                }
        }
}
```

16 Left Factoring

Factor ::= IDENT ["[" Expression "]"] | NUMLIT.

```
void Factor() {
    if (token == IDENT) {
        if (token == LBRACKET) {
            token = nextToken();
            Expression();
            if (token == RBRACKET) {
                token = nextToken();
            } else { error(); }
        }
    } else {
        if (token == NUMLIT) {
            nextToken();
        } else { error(); }
    }
}
```

17 From EBNF to BNF

For building parsers (especially bottom-up) a BNF grammar is often better, than EBNF. But it's easy to convert an EBNF Grammar to BNF:

- Convert every repetition { E } to a fresh non-terminal X and add X ::= ϵ | E X.
- Convert every option [E] to a fresh non-terminal X and add X ::= ϵ | E.
- Convert every group (E) to a fresh non-terminal X and add X ::= E.
- We can even do away with alternatives by having several productions with the same non-terminal.
  X ::= E | E'. becomes X ::= E. X ::= E'.

18 Error Recovery for Top-Down

- We choose a set of stop-symbols, e.g. } ;)
- If we encounter an error, we call skip(), give an error message and continue normally.
  - skip() skips the input to the next stop symbol.
  - It also skips subblocks { ... } completely.
- We do not print two error messages for the same position.

```
{
    a = 5 * (3 4);
}
```

19 Summary Top-Down Parsing

- A context-free grammar can be converted directly into a program scheme for a recursive descent parser.
- A recursive-descent parser finds a parse tree top down, from the start symbol towards the terminal symbols.
- Weakness: Must decide what to do based on first input token.

20 The Parser Generator JavaCUP

http://www.cs.princeton.edu/ appel/modern/java/CUP/.

- Generates a class Parser.java, which contains the parser.
- Generates a class Tokens.java, which is suitable to be used by a JLex scanner.
- Recognizes LALR(1) grammars (even more than LL(1)).
  - allows left recursion
  - allows common start, (if it is not too hidden)
  - only BNF
- If a grammar is not LALR(1) it produces an error message.

21 An Expression Parser in JavaCUP

```
package expression;
import java_cup.runtime.*;

parser code {:
    public Parser(Scanner lexer) {
        super(lexer);
    }
    public void report_error(String msg, Object o) {
        if (o instanceof java_cup.runtime.Symbol) {
            java_cup.runtime.Symbol sym =
                    (java_cup.runtime.Symbol) o;
            Report.error(sym.left, msg);
        } else {
            Report.error(Position.UNDEFINED, msg);
        }
    }
:};
```

22 An Expression Parser in JavaCUP (2)

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal NUMLIT;
non terminal Expression, Term, Factor;
start with Expression;

Expression ::= Expression PLUS Term
           |   Expression MINUS Term
           |   Term
           ;
Term       ::= Term TIMES Factor
           |   Term DIV Factor
           |   Factor
           ;
Factor     ::= NUMLIT
           |   LPAREN Expression RPAREN
           ;
```

23 A shift-reduce Conflict

If we enter the grammar

```
Expression ::= Expression PLUS Expression
           ;
```

without precedence JavaCUP will tell us:

```
*** Shift/Reduce conflict found in state #4
    between Expression ::= Expression PLUS Expression (*)
    and     Expression ::= Expression (*) PLUS Expression
    under symbol PLUS
    Resolved in favor of shifting.
```

The grammar is ambiguous!

Still, telling JavaCUP that PLUS is left associative helps!

24 Using Precedence

```
terminal PLUS, MINUS, TIMES, DIV, LPAREN, RPAREN;
terminal NUMLIT;
non terminal Expression, Term, Factor;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
start with Expression;

Expression ::= Expression PLUS Expression
            |  Expression MINUS Expression
            |  Expression TIMES Expression
            |  Expression DIV Expression
            |  NUMLIT
            |  LPAREN Expression RPAREN
            ;
```

25 Precedence

- left means, that a + b + c is parsed as (a + b) + c
- lowest precedence comes first, so a + b ∗ c is parsed as a + (b ∗ c)

26 A reduce-reduce Conflict

These conflicts are less common and often indicate a problem of the language rather than the grammar.

```
Expression  ::= MExpression
              |   DExpression
              ;
MExpression ::= IDENT TIMES IDENT
              |   IDENT
              ;
DExpression ::= IDENT DIV IDENT
              |   IDENT
              ;
*** Reduce/Reduce conflict found in state #4
   between MExpression ::= IDENT (*)
   and     DExpression ::= IDENT (*)
   under symbols: {EOF}
   Resolved in favor of the first production.
```