Part III

Parsing

Parsing

- The goal of the parser is twofold:
  1. Check whether a sentence belongs to the language.
  2. If so: construct the abstract syntax tree.
- Let us try to solve the first problem now, and then we will deal with abstract syntax trees in Part IV.

Reg-exp is not enough

- Regular expressions are not powerful enough to express all the types of sentences we would like to have in a programming language.
- For example, the following grammar can not be expressed by a regular expression:

```
Expr → Expr "+" Expr | "(" Expr ")" | Digit
Digit → [0-9]+
```


Context-free Grammars

- A *context-free grammar* describes a language.
- A grammar has a set of *productions* (or *rules*) of the form:
  `symbol → symbol symbol … symbol`
  with zero or more symbols on the rhs. Each symbol is either a *terminal* (a token) or a *non-terminal* (appears on the lhs of some rule).
- No token can appear on the lhs.
- One non-terminal is the *start symbol*.

Example of a Grammar

```
Swedish → Statement | Question
Statement → Subject Verb
Question → Verb Subject "?"
Subject → "Happi" | "Nick"
Verb → "Springer" | "Sitter" | "Stannar"
```

Is "Springer Happi?" a valid Swedish sentence?

```
Swedish
Question
Verb Subject "?"
"Springer" Subject "?"
"Springer" "Happi" "?"
```

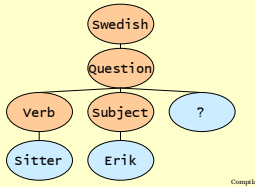

Parse trees

- Nodes are non-terminals.
- Leaves are terminals.
- Branching corresponds to rules of the grammar.
- The leaves gives a sentence of the input language.
- For every sentence in the language there is at least one parse tree.

Example of a Parse Tree

```
Swedish → Statement | Question
Statement → Subject Verb
Question → Verb Subject "?"
Subject → "Erik" | "Nick"
Verb → "Springer" | "Sitter" | "Stannar"

Sitter Erik ?
```



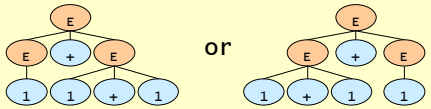
Ambiguous grammars

- Grammars which allow more than one parse tree for some sentences are called ambiguous.
- Ambiguous grammars are not good for programming languages.
- The programmer does not know what a sentence means.
- The compiler can not decide which tree to choose.

Example of Ambiguous Grammar

`E → E "+" E | "1"`

`Is 1 + 1 + 1`

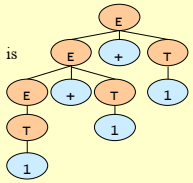


(1 + 1) + 1 1 + (1 + 1)

An ambiguous grammar can be rewritten as a non ambiguous grammar

```
E → E + T | T
T → "1"
```

Now `1 + 1 + 1` is



(1 + 1) + 1

Predictive Parsing

- Some grammars can be parsed using an algorithm called *recursive decent*.
- In a recursive decent parser the rules for each non-terminal are turned into a function, like we did for tokens, but the function can now be recursive. (i.e., can call itself, directly or indirectly.)

A recursive decent parser

```
Example → ID Example NUM
Example → NUM
```

```
void Example() {
  if (token == ID) {
    next(); Example(); eat(NUM);
 } else
   eat(NUM);
}

void next() {token = nextToken();}
void eat(Token T){if(token == T) next();else error();}
```

This does not always work

```
Example → Example "+" Term
Example → Term
Term → Term "*" Factor
Term → Factor
Factor -> NUM | Example
void Example() {
 if(?){Example(); eat(PLUS); Term();}
 else {Term();}
}
```

1 + 2 , 1 + 2* 3

When does recursive decent work?

- It only works for grammars where it is clear which alternative to choose by looking at just one token.
- Such a grammar is called LL(1), for *Left-to-right parse, Leftmost-derivation, 1-symbol lookahead.*
- We can find out automatically if a grammar is LL(1). To do this we need the concepts of *First, Follow,* and *nullable.*

First(S), Follow(S), and nullable.

- First(S) are the terminals that S can start with.
- Follow(S) are the terminals which can follow S.
- A non-terminal is nullable if it can derive the empty string (ε).

```
A → "b" "c" | B "d"
B → "a" | ε
First(A) = { b,a,d }, Follow(A) = {}.
First(B) = { a }, Follow(B) = {d}.
B is nullable, A is not nullable.
```


Formal definition

First(S), Follow(S), and nullable are the smallest sets with the following properties:

For each production $S \to Q_1 \dots Q_k$:

$\forall i\, i \le k$:

$\forall j\, j > i, j \le k$:

$\{Q_1 \dots Q_k\} \subseteq nullable \Rightarrow S \in nullable$

$\{Q_1 \dots Q_{i-1}\} \subseteq nullable \Rightarrow First(S) = First(S) \cup First(Q_i)$

$\{Q_{i+1} \dots Q_k\} \subseteq nullable \Rightarrow Follow(Q_i) = Follow(Q_i) \cup Follow(S)$

$\{Q_{i+1} \dots Q_{j-1}\} \subseteq nullable \Rightarrow Follow(Q_i) = Follow(Q_i) \cup First(Q_j)$

Algorithm

Use a fixpoint iteration:

Start with some initial minimal sets:

```
nullable = ∅
for each terminal t: first(t) = t,
                     follow(t) = ∅
```

Then iterate over the rules for the sets, adding new members as appropriate until the sets does not change.

LL(1)

- When we have calculated first, follow, and nullable for a grammar we can decide whether it is LL(1).
- A grammar is LL(1) if for every production: $S \to u_1 \mid u_2 \mid \dots \mid u_n$
  - $First(u_i) \cap First(u_j) = \emptyset$ if $i \neq j$
  - $Nullable(u_i) = \emptyset$ for at most one i
  - $First(u_i) \cap Follows(S) = \emptyset$ if $nullable(u_j)$ and $i \neq j$

LL(k)

- A LL(1) grammar can be parsed by just looking at one token.
- We can make recursive decent work for some more grammars by looking at more than one token.
- A grammar that can be parsed by looking at two tokens is called LL(2) and a grammar that can be parsed by looking at k tokens is called LL(k).
- Every LL(i) grammar is also LL(i+1).

Eliminating Left Recursion

- Some grammars that are not LL(1) can easily be turned in to LL(1) grammars by eliminating left recursion.

`Example` → `Example` "+" `Term`

`Example` → `Term`

Can be rewritten using right recursion:

`Example` → `Term Example'`

`Example'` → "+" `Term Example'`

`Example'` → ε

Left Factoring

- Grammars where two productions for the same non-terminal starts with the same symbols are not LL(1).

`Factor` → `ID`
`Factor` → `ID` "[" `Expr` "]"
`Factor` → `NUM`

This grammar can be made LL(1) by left factoring:

`Factor` → `ID Array`
`Factor` → `NUM`
`Array` → "[" `Expr` "]"
`Array` → ε

Summary of predictive parsing

- Some context-free grammars, namely those that are LL(k), can be converted directly into a recursive decent to parser program.
- A recursive decent parser finds a parse tree *top down*, from the start symbol to the terminal symbols, by left derivation.
- The weakness is that the parser must decide which rule to use based on the k first input tokens.

Bottom Up Parsing

- A bottom up parser builds a derivation from the terminal symbols, working toward the stat symbol.
- It consists of a *stack* and an *input*.
- It has four *actions*.
- A DFA applied to the stack decides which action to take.

Parsing actions

1. *Shift*, which pushes the next token onto the stack.
2. *Reduce*, removes $\mathcal{Q}_1 \ldots \mathcal{Q}_k$ from the stack and replaced them with S if there is a production $S \rightarrow \mathcal{Q}_1 \ldots \mathcal{Q}_k$.
3. *Accept*, ends the parser with success.
4. *Error*, ends the parser with an error message.

Bottom up parsers

- There are several different ways to construct different parser, and they can handle grammars of different complexity.
- There are:
  - LR(k) (*left-to-right parse, rightmost-derivation, k-token lookahead*),
  - SLR (*simple* LR),
  - LALR(1) (*lookahead* LR(1))
- We will not cover the details of these methods in this course.

LALR(1)

- LALR(1) is probably the most practical parse method for programming languages.
- All "reasonable" language constructs can be handled by al LALR(1) parser.
- The parser can be quite compact and efficient.
- This is the method used by parser generators such as Yacc, Bison, MLYacc, Yecc, SableCC, and most importantly for you by JavaCUP.

JavaCUP

- JavaCUP, just like JLex, takes an input file describing a grammar and produces a Java program that recognizes strings in the grammar.
- JavaCUP generates two files Parser.java and Tokens.java.
- If the grammar in the input file is not LALR(1) an error message is produced.

A JavaCUP parser

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

terminal PLUS, MINUS, TIMES;
terminal DIV, LPAR, RPAR, INT;
non terminal Expr, Term, Factor;

start with Expr;
```


A JavaCUP parser

```
Expr ::=   Expr PLUS Term
     |     Expr MINUS Term
     |     Term
     ;
Term ::=   Term TIMES Factor
     |     Expr DIV Factor
     |     Factor
     ;
Factor ::= INT
       |   LPAR Expr RPAR
       ;
```


Using Precedence

```
package expression;
import java_cup.runtime*;
terminal PLUS, MINUS, TIMES;
terminal DIV, LPAR, RPAR, INT;
non terminal Expr;
precedence left PLUS, MINUS;
precedence left TIMES, DIV;
start with Expr;
```

Using Precedence

```
Expr ::= Expr PLUS Expr
       |   Expr MINUS Expr
       |   Expr TIMES Expr
       |   Expr DIV Expr
       |   INT
       |   LPAR Expr RPAR
       ;
```


Precedence

- Left means left associative, that is, 1-2-3 is parsed as (1-2)-3. (i.e., –4 and not 2)
- Lowest precedence comes on the first row, each following row has higher precedence. So 1+2*3 is parsed as 1+(2*3). (i.e., 7 and not 9)

Shift-reduce Conflicts

- With an ambiguous grammar the parser generator can not decide whether to shift (push a token on the stack) or reduce the contents on the stack to a non terminal.
- E.g., `Expr ::= Expr PLUS Expr`
- Three solutions:
  1. Use precedence directives.
  2. Rewrite the grammar.
  3. Leave it as it is: shift will be chosen.

Not the right solution for the example!

Reduce-reduce Conflicts

- Reduce-reduce conflicts are not so common and often indicate a problem with the language rather than the grammar.
- They generally has to be solved by rewriting the grammar.
- (They can come from trying to solve semantic issues by syntactic rules.)

Syntax vs. Semantics

- The syntax describes the structure of a sentence, and the semantics describes its meaning.
- Example:

  We do not want to allow integer arithmetic on boolean values:

  1 + true.

Wrong solution

```
Expr ::= IExpr | BExpr
IExpr ::= IExpr PLUS IExpr
        |   INT
BExpr ::= BExpr OR BExpr
        |   BOOL
```

Now we have solved the problem

`1 + true`

Since `+` takes two `IExpr` and `true` is a `BExpr`.

Wrong solution (cont)

- But now we also want to allow variables: "`1 + x`", and "`true or x`"
- We add ID to the grammar.

```
Expr ::= IExpr | BExpr
IExpr ::= IExpr PLUS IExpr | INT | ID
BExpr ::= BExpr OR BExpr | BOOL | ID
```

- But now ID can be either IExpr or BExpr, and the parser has now way of knowing which interpretation is correct.
- Right Solution™: Defer this distinction until the semantic analysis. (Part IV)

Error Recovery

- After detecting an error the parser should be able to continue processing the input.
- This processing is only for finding more errors, not for generating code.

Error Recovery in JavaCUP

- Introduce a special symbol `error`.
- The author of a parser can use error in productions. For instance:

```
Statements ::= ε
             | Statement "." Statements
             | error "." Statements
             ;
```


Error recovery (cont)

- If the parser encounters an error, it will pop the stack until it gets into a state, where `error` is legal.
- At this point it shifts `error` onto the stack.
- Then, the input tokens are skipped, until the next input token is one that can legally follow the new state.

Error recovery (cont)

- This scheme is very dependent on a good choice of error productions.
- Assume a production
  `Statements = error "."`
- If the parser encounters an error inside a statement, it will pop the stack until it expects a `statement`.
- At this point it shifts `error` onto the stack.
- Then, the input tokens are skipped, until "`.`" is found.

Where to put `error`

- It is a good idea to have a terminal after error to ensure termination.
- Examples:

```
CondStatement ::= …
  IF Expr LBRACE error RBRACE
…
Expr ::= …
       | LPAR error RPAR
```