*Compilation 2010*

# SableCC

Jan Midtgaard
Michael I. Schwartzbach
Aarhus University

# The SableCC Tool

- SableCC is a (lexer and) parser generator
- The input is:
  - a sequence of token definitions
  - a context-free grammar
- The output is:
  - a LALR(1) parser for the defined language
  - available as a Java class

# Our Favorite Grammar in SableCC

```
Helpers
  tab   = 9;
  cr    = 13;
  lf    = 10;

Tokens
  eol   = cr | lf | cr lf;
  blank = ' ' | tab;
  star  = '*';
  slash = '/';
  plus  = '+';
  minus = '-';
  lpar  = '(';
  rpar  = ')';
  id = 'x' | 'y' | 'z';

Ignored Tokens
  blank,eol;
```

```
Productions
  start  = {plus}   start plus term |
           {minus}  start minus term |
           {term}   term;
  term   = {mult}   term star factor |
           {div}    term slash factor |
           {factor} factor;
  factor = {id}     id |
           {paren}  lpar start rpar;
```

# Generated Classes

```
drwxr-xr-x  2 mis users 4096 Sep  7 09:28 analysis/
drwxr-xr-x  2 mis users 4096 Sep  7 09:28 lexer/
drwxr-xr-x  2 mis users 4096 Sep  7 09:32 node/
drwxr-xr-x  2 mis users 4096 Sep  7 09:28 parser/
-rw-r--r--  1 mis users  536 Sep  7 09:32 xyz.sablecc
```

We never need to look at this output

# The Main Application

```
import parser.*;
import lexer.*;
import node.*;
import java.io.*;

class Main {
  public static void main(String args[]) {
    try {
      Parser p =
         new Parser (
           new Lexer (
              new PushbackReader(new InputStreamReader(System.in))));
      Start tree = p.parse();    /* parse the input */
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

# An Ambiguous Grammar

$$X \rightarrow \Lambda \mid \mathtt{a}\, X \mid \mathtt{a}\ \mathtt{a}\, X$$

Any string in this language has exponentially many different parse trees

$$\underbrace{\mathtt{a}\ \mathtt{a}\ \ldots\ \mathtt{a}}_{n}$$ has exactly *Fib(n)* parse trees

# The SableCC Version

```
Tokens
  a = 'a';

Productions
  x = {empty} |
      {one} a x |
      {two} [first]:a [second]:a x;
```

- Note that all symbols must have unique names
- The default name for `foo` is `[foo]:`

# SableCC is Unhappy

```
reduce/reduce conflict in state [stack: TA TA PX *] on EOF in {
        [ PX = TA PX * ] followed by EOF (reduce),
        [ PX = TA TA PX * ] followed by EOF (reduce)
}
```

The LALR(1) table contains conflicting actions

# Solution: Less Stupid Grammar

```
Tokens
  a = 'a';

Productions
  x = {empty} |
      {one} a x ;
```

# A Grammar for If-Statements

```
Tokens
  eol   = cr | lf | cr lf;
  blank = ' ' | tab;
  exp = 'exp';
  if = 'if';
  then = 'then';
  else = 'else';
  assign = 'assign';

Ignored Tokens
  blank,eol;

Productions
  stm = {one}  if exp then stm |
        {both} if exp then [thenbranch]:stm else [elsebranch]:stm |
        {assign} assign;
```

# SableCC is Unhappy

```
shift/reduce conflict in state [stack: TIf TExp TThen PStm *]
on TElse in {
        [ PStm = TIf TExp TThen PStm * TElse PStm ] (shift),
        [ PStm = TIf TExp TThen PStm * ] followed by TElse (reduce)
}
```

But the grammar does not appear to be stupid...

# Solution: Less Natural Grammar

```
Productions
  stm  = {one}  if exp then stm |
         {both} if exp then [thenbranch]:stm2 else [elsebranch]:stm |
         {assign} assign;
  stm2 = {both} if exp then [thenbranch]:stm2 else [elsebranch]:stm2 |
         {assign} assign;
```

# Dangling Else Problem

- An example statement:

```
if exp then if exp then assign else assign
```

- To which `if` does the `else` belong?
- The first grammar is ambiguous
- Our modified grammar parses the string as:

```
if exp then (if exp then assign else assign)
```

# The Palindrome Grammar

```
Tokens
  zero = '0';
  one  = '1';

Productions
  pal = {empty} |
        {one} one |
        {zero} zero |
        {oneone} [first]:one pal [second]:one |
        {zerozero} [first]:zero pal [second]:zero;
```

# SableCC is Unhappy

```
shift/reduce conflict in state [stack: TZero *] on TZero in {
        [ PPal = * TZero PPal TZero ] (shift),
        [ PPal = * TZero ] (shift),
        [ PPal = * ] followed by TZero (reduce),
        [ PPal = TZero * ] followed by TZero (reduce)
}

shift/reduce conflict in state [stack: TZero *] on TZero in {
        [ PPal = * TZero PPal TZero ] (shift),
        [ PPal = * TZero ] (shift),
        [ PPal = * ] followed by TZero (reduce),
        [ PPal = TZero * ] followed by TZero (reduce)
}

shift/reduce conflict in state [stack: TZero *] on TOne in {
        [ PPal = * TOne PPal TOne ] (shift),
        [ PPal = * TOne ] (shift),
        [ PPal = TZero * ] followed by TOne (reduce)
}
```

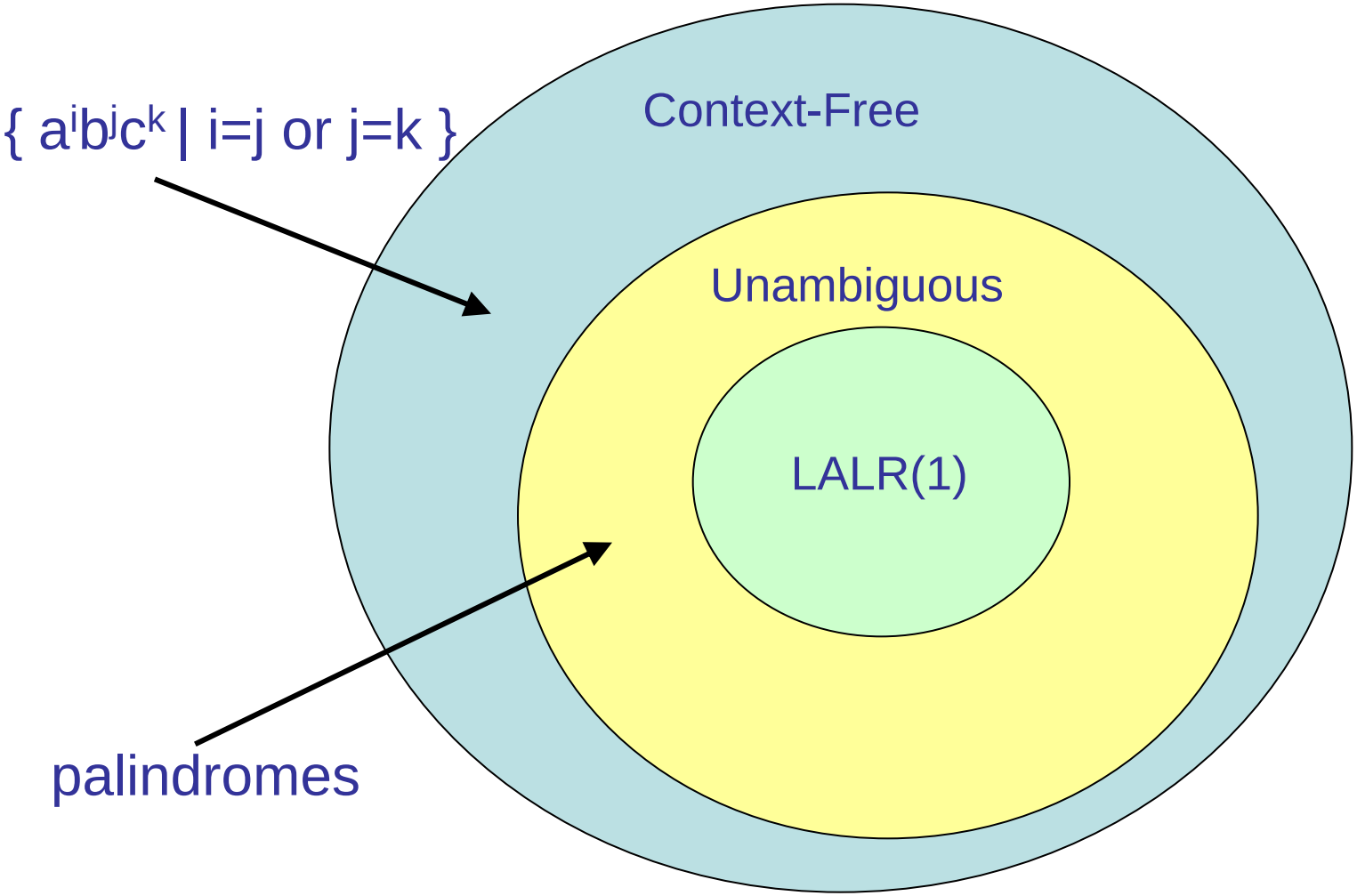
# No Solution!

- There is no LALR(1) grammar for this language

- Some **grammars** are not LALR(1)
- And some **languages** are not LALR(1)

- Some **grammars** are ambiguous
- And some **languages** are ambiguous

# Language Containments



# EBNF Features

SableCC allows right-hand side abbreviations:

- Optional: `x = y?`
- List: `x = y*`
- Non-empty list: `x = y+`

This has many benefits:

- shorter
- less error-prone
- fewer names must be invented

# EBNF Example

```
block = lbrace decl* stm+ rbrace ;
decl = type id init? semicolon ;
init = equals exp;
```

# EBNF Expansion

- `x = y?`

  ↓

  `x = {some} y | {none} ;`

- `x = y*`

  ↓

  `x = {zero} | {more} y x ;`

- `x = y+`

  ↓

  `x = {one} y | {more} y x ;`