

# Compiler Construction

SMD163

## Lecture 6: Parser generators & abstract syntax

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.  
Contains material generously provided by Mark P. Jones



COMPUTER SCIENCE  
AND ELECTRICAL ENGINEERING  
LULEÅ UNIVERSITY OF TECHNOLOGY

1

## The Quick Calculator, revisited:

- ◆ In the first lecture, we looked at the quick calculator program, comprising:
  - A lexical analyzer;
  - A recursive descent parser;
  - A representation for the abstract syntax of expressions, including code for evaluation and code generation;
  - A top-level driver.
- ◆ A few weeks later, what would we change?

2

## Parsing in the Quick Calculator:

- ◆ The code for the quick calculator's parser is:
  - Described at a high-level by a grammar;
  - Large – about a third of the whole program – and filled with little details;
  - Easy enough to write, given the grammar, but a slow and error prone task for large grammars;
  - Hard to read – how can you be sure that it recognizes the correct grammar?
  - Difficult to modify.
- ◆ These are exactly the reasons that we use compilers!

3

## Introducing jacc:

- ◆ jacc is a parser generator:
  - It takes a grammar as its input;
  - It produces a parser (in Java) as its output.
- ◆ jacc is modeled on the classic Unix utility `yacc` – “Yet another Compiler Compiler” – which was written by Johnson in the 1970s, and generates output in C.
- ◆ There are plenty of parser generators, each designed for use with particular languages, classes of grammar, and parsing strategies.

4

## Pros and Cons:

### ◆ Advantages:

- Easier to ensure that the grammar is correct;
- Grammar is usually much smaller than the corresponding parser;
- If the grammar changes, it's easy to build a new parser automatically.

### ◆ Disadvantages:

- We cannot always use the original grammar ... it might result in conflicts.

5

## Technical Details:

- ◆ yacc is the canonical example of a LALR(1) parser generator, producing shift reduce parsers that correctly recognize any language specified by an LALR(1) grammar.
- ◆ In practice, disambiguating rules make it possible to use yacc quite successfully with a larger class of grammars.

6

## The Input to jacc:

### ◆ Input files take the form:

```
%{  
... preliminary Java declarations ...  
%}  
  
%token declarations  
  
%%  
  
... grammar rules ...  
  
%%  
  
... additional program code ...
```

- ◆ jacc grammars are stored in files with a .jacc suffix.

7

## The Output From jacc:

- ◆ To construct a parser from the input file Lang.jacc, we use the command:  
    > jacc Lang.jacc
- ◆ This results in two output files:
  - LangTokens.java: a file containing the definition of a code for each token in the input grammar.
  - LangParser.java: a file containing a parser for the input grammar.

8

## Inside LangParser.java:

- ◆ The LangParser.java file contains:
  - the definition of a parse() function, coded directly in Java as a state machine;
  - any additional code that was included in the input file.
- ◆ (In yacc:
  - input files end with .y;
  - the output files are called y.tab.c and y.tab.h;
  - the parser function is called yyparse(), and is table driven.)

9

## A Simple Example:

- ◆ Our original grammar for the quick calculator was:

```
program ::= expr
         | program ; expr
expr     ::= expr + expr
         | expr - expr
         | expr * expr
         | expr / expr
         | ( expr )
         | integer
```

10

## A Simple Example:

- ◆ This translates directly into jacc/yacc notation as follows:

```
%token INTEGER
%%
program : expr
        | program ';' expr
        ;
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '(' expr ')'
        | INTEGER
        ;
```

token symbols must be defined; not necessary for single character literals.

- ◆ This is a complete, valid input.

11

## Running jacc:

- ◆ To generate the parser:

```
> jacc example.jacc
WARNING: conflicts: 16 shift/reduce, 0 reduce/reduce

> ls example*
example.jacc  exampleParser.java  exampleTokens.java
>
```

- ◆ jacc has still produced a parser for us, but the appearance of conflicts signals a potential problem: how were the conflicts resolved?

12

## Debugging jacc parsers:

- ◆ To generate debugging information:

```
> jacc -v example.jacc
WARNING: conflicts: 16 shift/reduce, 0 reduce/reduce
```

```
> ls example*
example.jacc      exampleParser.java
example.output  exampleTokens.java
>
```

- ◆ The `-v` flag causes jacc to output a debugging report in the file `example.output`. (Use the `-h` flag for HTML output.)
- ◆ (Users of yacc will find their output in `y.output`)

13

## Inside example.output:

- ◆ The output file contains a description of the generated machine, annotated with conflict information.
- ◆ The output for the start state looks like this:

```
state 0 (entry on program)
  $accept : _program $end
  INTEGER shift 3
  '(' shift 4
  . error
  program goto 1
  expr goto 2
```

Annotations:

- state summary
- items for this state
- actions for this state
- default for this state
- gotos for this state

14

## Reporting Conflicts:

- ◆ But state 12 looks like this:

```
12: shift/reduce conflict (shift 6 and red'n 5) on '*'
12: shift/reduce conflict (shift 7 and red'n 5) on '+'
12: shift/reduce conflict (shift 8 and red'n 5) on '-'
12: shift/reduce conflict (shift 9 and red'n 5) on '/'
state 12 (entry on expr)
  expr : expr '+' expr (3)
  expr : expr '-' expr (4)
  expr : expr '*' expr (5)
  expr : expr '/' expr (6)
  '*' shift 6
  '+' shift 7
  '-' shift 8
  '/' shift 9
  . reduce 5
```

Annotations:

- conflict reports
- items for this state
- actions for this state

15

## Notes:

- ◆ The conflict arose as the result of a choice between:
  - Shift, and do the `*` later;
  - Reduce, and do the `*` now.
- ◆ jacc has made a choice: shift
- ◆ So we have a working parser ... but it may not work as intended.
- ◆ There are another 4 conflicts for each of the remaining 3 arithmetic operators. So the total number of conflicts is 16, as reported.

16

## Fixing the Simple Example:

- ◆ We can add precedence annotations:

```

%left '+' '-'
%left '*' '/'
%token INTEGER
%%
program : expr
        | program ';' expr
        ;
expr    : expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '/' expr
        | '(' expr ')'
        | INTEGER
        ;
    
```

- ◆ yacc accepts this without reporting conflicts.

17

## What parse() returns:

- ◆ The parse() function returns:
  - true if the input matched the grammar and an ENDINPUT token was read;
  - false if a syntax error was encountered.
- ◆ But parse() does not return any representation of the parsed expression – no parse tree is actually built!
- ◆ The structure of the parse tree is implicit in the structure of the computation that parse() goes through ... but how does the programmer get access to this?

18

## Getting Some Action:

- ◆ We can annotate each production in the grammar with an action, containing code to execute when the production is reduced:

```

Prog : Prog ';' Expr      { process($3); }
    | Expr                { process($1); }
    ;
Expr : Expr '+' Expr      { $$ = new BinExpr('+', $1, $3); }
    | Expr '-' Expr      { $$ = new BinExpr('-', $1, $3); }
    | Expr '*' Expr      { $$ = new BinExpr('*', $1, $3); }
    | Expr '/' Expr      { $$ = new BinExpr('/', $1, $3); }
    | INTEGER            { $$ = new IntExpr($1); }
    | '(' Expr ')'       { $$ = $2; }
    ;
    
```

Some actions do things ...

Some actions construct things ...

Some actions pass things on ...

19

## Semantic Values:

- ◆ Each symbol in a grammar has an associated semantic value.
  - We use \$n to refer to the semantic value of the n<sup>th</sup> symbol on the right hand side of a production.
  - We use \$\$ to refer to the semantic value that is constructed for the symbol on the left hand side of the production.
- ◆ The type of semantic values is set by adding an annotation (default is object)
  - %semantic Expr
- ◆ More specific types can be given to certain terminals and nonterminals (using %token and %type)

20

## Abstract syntax

- ◆ A particular semantic value that can be constructed is a representation of the parsed input (i.e., the parse tree)
- ◆ This typically a representation that is much simpler than concrete syntax:
  - No parenthesized expressions
  - No disambiguating tricks like expr/term/factor/atom
  - No redundant keywords or separators, just simple node tags
  - Explicit lists instead of recursion
  - etc...
- ◆ The structure of these parse trees can also be described by a grammar – the abstract program syntax

21

## Abstract syntax in Java

- ◆ For QuickCalc we could have defined:

```
Expr ::= BinExpr Expr op Expr
      | IntExpr int
```

- ◆ Note: no need to remember parentheses anymore!
- ◆ Implemented in Java as:
  - An abstract class for each nonterminal (Expr)
  - Methods that reflect what we want to do with the abstract representation (eval, codegen (, print, typecheck, ...))
  - A concrete subclass for each production for an abstract syntax nonterminal (BinExpr, IntExpr)
  - For each concrete class: fields for each right-hand side component (and corresponding constructor parameters)

22

## Abstract syntax in Java

```
abstract static class Expr {
    abstract void compile();
    abstract int eval();
}
```

Limitation: what we want to do must be defined in advance!

Solution: Visitors!  
(more next lecture)

```
static class BinExpr extends Expr {
    private char op;
    private Expr left;
    private Expr right;
    BinExpr(char op, Expr left, Expr right);
    ...
}
```

```
static class IntExpr extends Expr {
    private int value;
    IntExpr(int value);
    ...
}
```

23

## Short-circuiting the Parse Tree:

- ◆ For a really simple calculator, all we really want is an integer result, and can substitute simpler actions that “execute eval()” directly:

```
%semantic int
%%
Prog : Prog ';' Expr { process
      | Expr           { process
      ;
Expr : Expr '+' Expr { $$ = $1 + $$; }
      | Expr '-' Expr { $$ = $1 - $3; }
      | Expr '*' Expr { $$ = $1 * $3; }
      | Expr '/' Expr { $$ = $1 / $3; }
      | '(' Expr ')' { $$ = $2; }
      | INTEGER      { $$ = $1; }
      ;
```

Sets the type of semantic values. Default is Object.

24

## How to Use Generated Parsers:

- ◆ What changes do we need to make to the rest of the program to use a jacc generated parser?
  - Parser – to fit with the rest of our framework:

```
%{
import QuickCalc;
%}
```
  - Error handling – must provide a function called yyerror() to handle errors:

```
private void yyerror(String msg) {
    QuickCalc.error(msg);
}
```
  - Lexical analysis – the parser expects the lexer to be an object called lexer, that exports methods nextToken(), getToken(), and getSemantic() (for advancing, reading, and accessing the semantic value of the current token, respectively)

25

## Token Definitions:

- ◆ jacc generates a Java interface containing codes to represent each type of token:

```
interface QuickCalcTokens {
    int ENDINPUT = 0;
    int INTEGER = 1;
    int error = 2;
    // '(' (code=40)
    ...
}
```

- ◆ The parser and the lexer import (implement) this interface so that they agree what the codes mean!

```
public class QuickCalcLexer implements
    QuickCalcTokens { ...
```

26

## The New Quick Calculator:

```
public static void main(String[] args) {
    System.out.println("welcome!");
    new QuickCalcParser(new
    QuickCalcLexer(System.in)).parse();
}
```

27

## Parser Generators, in practice:

- ◆ We've seen how parser generators work;
- ◆ We've seen how the generated parsers work;
- ◆ We've seen how the generated parsers can be used ...
  
- ◆ But, in practice, there's a little bit more to it...

28

## Conflicts Happen:

- ◆ Not all context-free grammars are LALR(1).
- ◆ Some unambiguous grammars will result in conflicts.
- ◆ All ambiguous grammars will result in conflicts.
- ◆ We could just ban any grammar that isn't LALR(1) (and hence ban all ambiguous grammars) ... but is that going too far?

29

## A More Relaxed Approach:

- ◆ yacc/jacc will accept any grammar as input, and will always produce a parser as its output.
- ◆ If there are no conflicts, then the parser will return:
  - true, only if the input matches the grammar;
  - false, only if the input does not match the grammar.

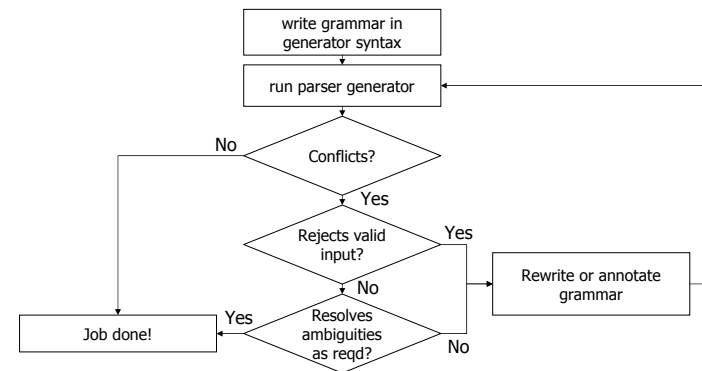
30

## continued ...

- ◆ But if there are conflicts, then the generated parser may:
  - ... sometimes return false, even when the input does actually match the grammar; or
  - ... choose one of several alternatives when there is a choice.
- ◆ The generated parser will never return true on an input that does not match the grammar.
- ◆ Conflicts are not (necessarily) errors.

31

## Using a Parser Generator:



32

## Understand ...

- ◆ In the case of jacc (yacc), run the parser generator with the `-h (-v)` flag.
- ◆ Inspect the output file for descriptions of each conflict.
- ◆ Understand what is causing the conflict, (e.g., try to find a sequence of input tokens that illustrates why the grammar is hard to parse.)
- ◆ Are both resolutions of the conflict reasonable? If so, we must change the grammar.
- ◆ If only one resolution is reasonable, is it the one that the parser generator chose?

33

## ... then Fix:

- ◆ In jacc/yacc, there are only two real ways to work around conflicts:
  - Use a precedence declaration;
  - Rewrite the grammar.
- ◆ Other parser generators may provide different mechanisms; check the documentation if you are using one of them ...

34

## Resolving Conflicts in jacc:

If all else fails, jacc/yacc:

- Resolves shift/reduce conflicts by shifting.
- Resolves reduce/reduce conflicts by reducing the earliest production. (Productions are numbered in the order they appear in the input, and these numbers are included in the .output file.)

35

## Precedence and Fixities:

- ◆ If operator  $\otimes$  has higher precedence than an operator  $\oplus$ , then  $x\otimes y\oplus z$  should be parsed as  $(x\otimes y)\oplus z$ .
- ◆ If operator  $\oplus$  groups to the left, then  $x\oplus y\oplus z$  should be parsed as  $(x\oplus y)\oplus z$ .
- ◆ If operator  $\oplus$  groups to the right, then  $x\oplus y\oplus z$  should be parsed as  $x\oplus (y\oplus z)$ .
- ◆ If operator  $\oplus$  is non-associative, then  $x\oplus y\oplus z$  should be treated as a syntax error.

36

## Precedence Declarations:

- ◆ The precedence and fixity of symbols in yacc can be declared at the beginning of the grammar (before the first %%):
  - %left
  - %right
  - %nonassoc
- ◆ Operators are listed in increasing order of precedence.
- ◆ All operators on the same line have the same precedence.
- ◆ You can't have right and left associative operators with the same precedence.

37

## How Precedence is Used:

Faced with a reduction:

$$A \rightarrow S_1 S_2 \dots S_n \_$$

and a lookahead symbol  $t$ , find the rightmost symbol in the production that has fixity,  $fp$ , and look up the fixity,  $ft$ , of  $t$  (if it has one).

If  $prec(fp) > prec(ft)$ , then reduce;

If  $prec(fp) < prec(ft)$ , then shift;

If  $prec(fp) = prec(ft)$  and both are %left, then reduce;

If  $prec(fp) = prec(ft)$  and both are %right, then shift;

If none of these apply, we can't use precedence to resolve the conflict.

38

## For Example:

- ◆ Consider the following grammar for expressions:

```
%left '+' '-'
%left '*' '/'
%%
Expr : Expr '+' Expr
      | Expr '-' Expr
      | Expr '*' Expr
      | Expr '/' Expr
      | '(' Expr ')'
      | INTEGER
      | '-' Expr
      ;
```

Unary minus: this production gets the same fixity as the '-' symbol; left associative, low precedence. Thus  $-3*2$  will parse as  $-(3*2)$  ...

39

## Changing Precedences:

- ◆ Suppose that we wanted unary minus to have the highest precedence:

```
%left '+' '-'
%left '*' '/'
%left UMINUS
%%
Expr : Expr '+' Expr
      | Expr '-' Expr
      | Expr '*' Expr
      | Expr '/' Expr
      | '(' Expr ')'
      | INTEGER
      | '-' Expr %prec UMINUS
      ;
```

A dummy token, with higher precedence than +, -, \*, /

A %prec annotation gives this production the same fixity as the symbol UMINUS.

Now  $-3*2$  will parse as  $(-3)*2$

40

## The Dangling Else:

- ◆ A famous problem in the design of programming language parsers:

```
%token IF THEN ELSE
%%
stmt : IF expr THEN stmt ELSE stmt
      | IF expr THEN stmt
      | ...
      ;
```

- ◆ This grammar is ambiguous!

41

## ... continued:

- ◆ Consider the statement:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

- ◆ This can be read in two different ways:

```
IF e1 THEN          IF e1 THEN
  IF e2 THEN          ELSE
  THEN                s1
                      ELSE s2
s1                    ELSE
s2                    ELSE
```

This is the 'dangling' else!

42

## Dangling Else Conflicts:

- ◆ jacc reports a single conflict:

```
> jacc -ptv dang.jacc
WARNING: conflicts: 1 shift/reduce,
                0 reduce/reduce
```

- ◆ In the .output file:

```
6: shift/reduce conflict (shift 7 and red'n 2)
   on ELSE
state 6 (entry on stmt)
  stmt : IF expr THEN stmt_ELSE stmt   (1)
  stmt : IF expr THEN stmt_           (2)

ELSE shift 7
      . reduce 2
```

43

## ... continued:

- ◆ Here's where the conflict might hit:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

We'd get this statement if we reduce now ...

- ◆ jacc chooses to shift, and will end up with a stack:

```
IF e1 THEN IF e2 THEN s1 ELSE s2
```

We'll get this statement if we reduce now ...

- ◆ So a jacc generated parser will treat this as:

```
IF e1
  THEN IF e2
        THEN s1
        ELSE s2
```

44

## Your Choice of Fixes:

- ◆ Leave the grammar as is, with documentation that 1 shift/reduce conflict is expected. (Bison even provides a `%expect` declaration for this).
- ◆ Add precedence declarations!  
`%left THEN`  
`%right ELSE`
- ◆ Silences yacc ... but it's a bit of a hack!
- ◆ Change the grammar ...

45

## Eliminating the Dangling Else:

- ◆ Rewrite the grammar to eliminate the ambiguity, without changing the language:

```
%token IF THEN ELSE
%%
stmt          : matched
              | unmatched
              ;
matched       : IF expr THEN matched ELSE matched
              | other
              ;
unmatched    : IF expr THEN stmt
              | IF expr THEN matched ELSE
              ;
```

46

## Grammar Rewriting Woes:

- ◆ The declarative ideal:
  - You know your parser is correct, because it was generated directly from your grammar.
- ◆ The practical reality:
  - Your grammar might not be good enough for the parser generator.
- ◆ Changing the grammar:
  - May obfuscate and complicate it;
  - May introduce errors;
  - May increase the size of the parser;
  - May require duplicated actions.

47

## An example from *eqn*:

- ◆ The *eqn* package for typesetting mathematics uses the following notation for subscripts and superscripts:
  - `foo sub bar`  $\Rightarrow$   $foo_{bar}$
  - `foo sup bar`  $\Rightarrow$   $foo^{bar}$
- ◆ The corresponding grammar has conflicts:  
`exp : exp SUB exp`  
`| exp SUP exp`  
...
- ◆ These are easily resolved using `%right`.

48

## A reduce/reduce conflict:

- ◆ But now suppose that you want both a subscript and a superscript ...
  - foo sub bar sup baz  $\Rightarrow$   $foo_{bar}^{baz}$
  - foo sup baz sub bar  $\Rightarrow$   $foo^{baz}_{bar}$
- ◆ We can add a special case to the grammar:

```
exp : exp SUB exp SUP exp
    | exp SUB exp
    | exp SUP exp
    ...
```
- ◆ But now we get our first reduce/reduce conflict ...

But we want:  
 $foo_{bar}^{baz}$

49

## How did yacc Handle it?

The conflict report looks like this:

8: reduce/reduce conflict (red'ns 1 and 3) on \$end  
state 8 (entry on exp)

```
exp : exp_SUB exp SUP exp (1)
exp : exp SUB exp SUP exp_ (1)
exp : exp_SUB exp (2)
exp : exp_SUP exp (3)
exp : exp SUP exp_ (3)
```

```
SUB shift 3
SUP shift 4
. reduce 1
```

exp SUB exp SUP exp

Use the earliest reduction; this is why we  
put our special case rule first

50

## Reflections on reduce/reduce:

- ◆ Shift/reduce conflicts occur when there is not enough information for the parser to decide what to do next. There may or may not be an ambiguity.
- ◆ Reduce/reduce conflicts are an indication of certain ambiguity, and are usually caused by serious problems in the grammar.
- ◆ Normally, you will have to rewrite the grammar.

51

## Modifiers in Java:

- ◆ In Java, field and method declarations can be annotated with *modifiers*.
- ◆ Different sets of modifiers are allowed for field and method declarations, but some are the same.

```
FieldModifier : public | private | ... | volatile ;
MethModifier  : public | private | ... | abstract ;
```
- ◆ Suppose the parser sees a partial input:

```
class Example { | public static ...
```
- ◆ Should it reduce "public" as a FieldModifier or as a MethModifier?

52

## Approximation:

- ◆ The usual solution to problems like this is to make the grammar weaker so that it accepts more programs than are strictly legal.
- ◆ To parse Java, for example, we might use a single nonterminal for modifiers:  
Modifier : public | private | ... | volatile | abstract ;
- ◆ At a later phase, the compiler must check that the set of modifiers given was appropriate.

53

## More Semantic Tests:

- ◆ In some versions of BASIC, the expression A(42) might be either a function call or an array reference.

```
exp : IDENT '('exp ')' { call action }  
    | IDENT '('exp ')' { array action }  
    ...
```

- ◆ To avoid a conflict, we can rewrite the grammar to use a semantic test:

```
exp : IDENT '('exp ')' { if (test)  
                                                                call action;  
                                                                else  
                                                                array  
                                                                }  
action; }  
...
```

54

## Left / Right Recursion:

- ◆ A production of the form  $A \rightarrow A w$  is said to be left recursive.
- ◆ A production of the form  $A \rightarrow w A$  is said to be right recursive.
- ◆ Many languages can be expressed using either left or right recursion:

```
prog : prog ';' expr  
      prog  
      | expr  
      ;  
prog : expr ';' ;
```

55

## Comparing left and right:

- ◆ Recall that shift/reduce parsers produce a rightmost derivation (in reverse).

- ◆ With a left recursive grammar:

```
e1;e2;e3 → p1;e2;e3 → p1;e2;e3 → p1;e2 → p1
```

- ◆ With a right recursive grammar:

```
e1;e2;e3 → e1;e2;p3 → e1;p2 → e1;p2 → p1
```

- ◆ A right recursive production delays reduction by pushing everything on the stack.

56

## Bottom-up favors left recursion:

- ◆ Both are acceptable grammars;
- ◆ Both result in parsers that take a list of expressions and display the corresponding results.

BUT:

- ◆ The right recursive version uses more stack space;
- ◆ Doesn't give a result until all expressions have been entered;
- ◆ Displays the results in reverse order!

57

## Summary:

- ◆ A conflict isn't necessarily an error ... but you should check conflict reports very carefully.
- ◆ Rewriting grammars can help to eliminate conflicts, but can also introduce new problems.
- ◆ Tools like jacc can be useful in the early stages of language development as a tool for grammar design/debugging, not just parser construction.
- ◆ Next time: Introducing semantic analysis ...

58