NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 9: Practical parsing issues and yacc intro

2020-02-04

Michael Engel

Overview

- Practical parsing issues
  - Error recovery
  - Unary operators
  - Handling context-sensitive ambiguity
  - Left versus right recursion

- A quick yacc intro
  - Syntax of yacc grammar descriptions
  - yacc-lex interaction
  - Example

Error recovery

Syntax analysis

- Syntax errors are common in program development
- Our previous parsers have stopped parsing at the first error
  - Is this what a programmer would want? [2]
- Prefer to find as many syntax errors as possible in each compilation

- A mechanism for ***error recovery*** helps the parser to move on to a state where it can continue parsing when it encounters an error
  - Select one or more words that the parser can use to synchronize the input with its internal state
  - When the parser encounters an error, it discards input symbols until it finds a synchronizing word and then resets its internal state to one consistent with the synchronizing word

Error recovery

- Consider a language using semicolons as statement separators
  - The semicolon can be used as synchronizing element: when an error occurs, the parser calls the scanner repeatedly until it finds a semicolon

```
foo = func)42 ;
return foo ;
```

- Here, a recursive-descent parser can simply discard words until it finds a semicolon and return (*fake*) success [1]
- This resynchronization is more complex in an LR(1) parser:
  - it discards input until it finds a semicolon…
  - scans back down the stack to find state with valid `Goto[s, Stmt]` entry
  - the first such state on represents the statement that contains the error
  - discards entries on the stack above that state, pushes the state `Goto[s, Stmt]` onto the stack and resumes normal parsing

Unary operators

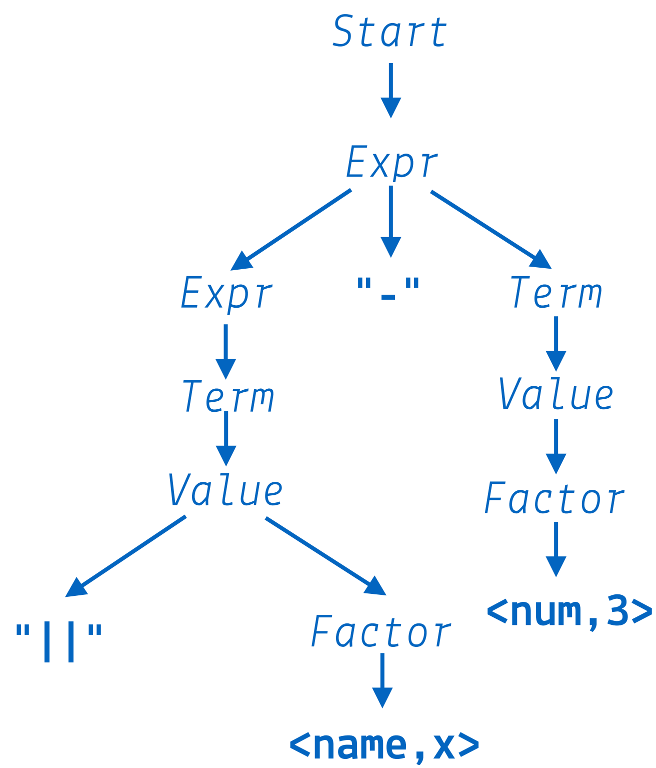
- Classic expression grammar includes binary operators only
- Algebraic notation includes unary operators
  - e.g., unary minus and absolute value
- Other unary operators:
  - autoincrement (`i++`)
  - autodecrement (`i--`)
  - address-of (`&`)
  - dereference (`*`)
  - boolean complement (`!`)
  - typecasts (`(int)x`)
- Adding these to the expression grammar requires some care

Unary operators

Syntax analysis

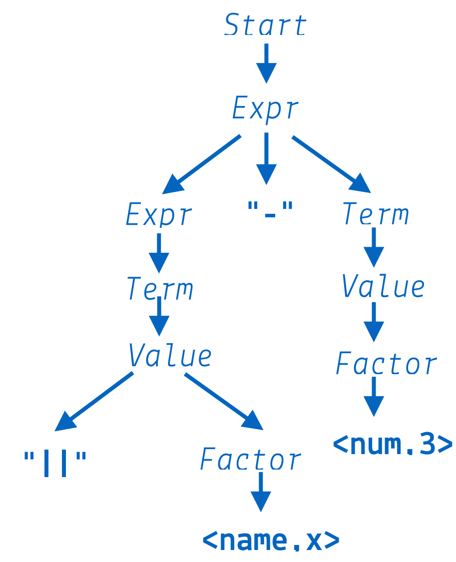
Example: expression grammar with an absolute value operator **||x**

```
Start  → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term × Value
       | Term ÷ Value
       | Value
Value  → "||" Factor
       | Factor
Factor → "(" Expr ")"
       | num
       | name
```



Parse tree for `|| x - 3`

Unary operators



Example: absolute value operator `||x`

- Absolute value should have higher precedence than either × or ÷
- However, it needs lower precedence than *Factor*
  - this enforces evaluation of parenthetic expressions before application of `||`
- The example grammar is still LR(1)
  - but it does not allow to write `|| || x`
- Writing this doesn't make much sense
  - but it's a legal mathematical operation, so why not?
  - This would work: `||(|| x)`
- Problem for other operators like (dereferencing) `*`
  - `**p` is a common operation in C

```
Start  → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term × Value
       | Term ÷ Value
       | Value
Value  → "||" Factor
       | Factor
Factor → "(" Expr ")"
       | num
       | name
```

Unary operators

Problem for other operators like `*`

- `**p` is a common operation in C

- Solution:
  - add a dereference production for *Value* as well: *Value* → `"*"` *Value*
- The resulting grammar is still LR(1)
  - even if we replace the `"×"` operator in *Term* → *Term* × *Value* with `"*"`, overloading the operator `"*"` in the way that C does

- The same approach works for unary minus

```
Start  → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term "*" Value
       | Term ÷ Value
       | Value
Value  → "*" Value
       | "||" Factor
       | Factor
Factor → "(" Expr ")"
       | num
       | name
```

Handling context-sensitive ambiguity

- Using one word to represent two different meanings can create a syntactic ambiguity
  - Common in early programming languages (FORTRAN, PL/I, Ada)
- Parentheses used to enclose both the subscript expressions of an array reference and the argument list of a subroutine or function
  - For the input `fee(i,j)`, the compiler cannot tell if `fee` is a two-dimensional array or a procedure that must be invoked
  - Differentiating between these two cases requires knowledge of `fee`'s declared type
- This information is not syntactically obvious
  - The scanner would classify `fee` as a name in either case

Handling context-sensitive ambiguity

Syntax analysis

- We can add productions that derive both subscript expressions and argument lists from *Factor*

  - Handling this in a classical expression grammar might look like this:

- Since the last two productions have identical right-hand sides, this grammar is ambiguous, which creates a ***reduce-reduce conflict*** in an LR(1) table builder

```
Factor → FunctionReference
       | ArrayReference
       | "(" Expr ")"
       | num
       | name
FunctionReference
       → name "(" ArgList ")"
ArrayReference
       → name "(" ArgList ")"
```

Handling context-sensitive ambiguity

Our grammar results in an LR(1) ***reduce-reduce conflict***

- Resolving this ambiguity requires ***extra-syntactic knowledge***
  - "Is `name` a function or an array?"
- In a recursive-descent parser, the compiler writer can combine the code for *FunctionReference* and *ArrayReference*
  - add the extra code required to check the name's declared type
- In a table-driven parser built with a parser generator, the solution must work within the framework provided by the tools

```
Factor → FunctionReference
       | ArrayReference
       | "(" Expr ")"
       | num
       | name
FunctionReference
       → name "(" ArgList ")"
ArrayReference
       → name "(" ArgList ")"
```

Handling context-sensitive ambiguity


```
Factor → FunctionOrArrayReference
       | "(" Expr ")"
       | num
       | name
FunctionOrArrayReference
       → name "(" ArgList ")"
```

Two different approaches to solve this:

- ***Rewrite*** grammar to combine function invocation and array reference into a single production
  - issue is deferred until a later step in translation
  - there, it can be resolved with information from the declarations
- Scanner can ***classify identifiers*** based on their declared types
  - requires handshaking between scanner and parser
  - works as long as the language has a ***define-before-use*** rule
- Rewritten in this way, the grammar is unambiguous
  - Since the scanner returns a distinct syntactic category in each case, the parser can distinguish the two cases

```
FunctionReference
       → function_name "(" ArgList ")"
FunctionOrArrayReference
       → array_name "(" ArgList ")"
```

Left versus right recursion

Syntax analysis

- Top-down parsers need right-recursive grammars
- Bottom-up parsers can accommodate either left or right recursion
- Compiler writers must choose between left and right recursion in writing the grammar for a bottom-up parser – how?

Stack depth criterion

- Left recursion can lead to smaller stack depths
  - Accordingly, lower memory use, less recursions

```
List → List elt
     | elt
```

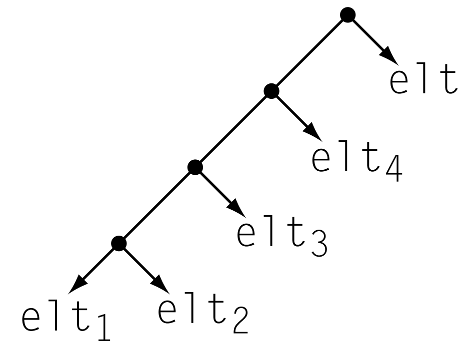
Left recursive grammar

```
List → elt List
     | elt
```

Right recursive grammar

Left versus right recursion: stack depth

- The ***left-recursive grammar*** shifts **`elt1`** onto its stack and immediately reduces it to *`List`*
- Next, it shifts **`elt2`** onto the stack and reduces it to *`List`* and so on…
- It proceeds until it has shifted each of the five **`elt`**'s onto the stack and reduced them to *`List`*
- Thus, the stack reaches
    - a maximum depth of two
    - and an average depth of $\frac{10}{6} = 1\frac{2}{3}$
- The stack depth of a left-recursive grammar depends on the grammar, not the input stream



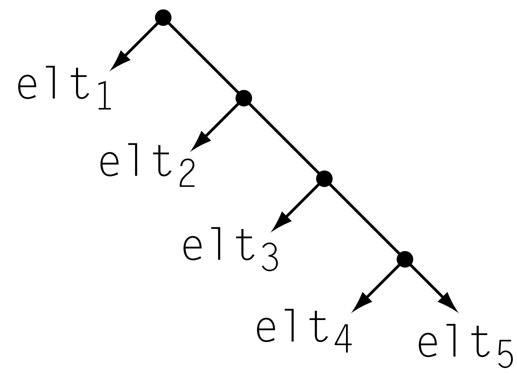

```
List → List elt
     | elt
```

```
List
List elt5
List elt4 elt5
List elt3 elt4 elt5
List elt2 elt3 elt4 elt5
List elt1 elt2 elt3 elt4 elt5
```

Left recursion

Left versus right recursion: stack depth

- The ***right-recursive grammar*** first shifts all five `elt`'s onto its stack
- Next, it reduces `elt5` to *`List`* using rule two and the remaining `elt`'s using rule one
- Thus, its maxium stack depth will be 5 and the average will be $\frac{20}{6} = 3\frac{1}{3}$
- Its maximum stack depth is bounded only by the length of the list
  - With thousands of elements in a list, this can become problematic




```
List → elt List
     | elt
```

```
List
elt1 List
elt1 elt2 List
elt1 elt2 elt3 List
elt1 elt2 elt3 elt4 List
elt1 elt2 elt3 elt4 elt5 List
```

Right recursion

Left versus right recursion: associativity

- Left recursion naturally produces left associativity, and right recursion naturally produces right associativity
- In some cases, the order of evaluation makes a difference
- Consider the string `x1 + x2 + x3 + x4 + x5`
  - the left-recursive grammar implies a left- to-right evaluation order
  - the right-recursive grammar implies a right- to-left evaluation order
- With some number systems, such as floating-point arithmetic, these two evaluation orders can produce ***different results*** [1]

```
Expr → Expr + Operand
     | Expr - Operand
     | Operand
```

```
Expr → Operand + Expr
     | Operand - Expr
     | Operand
```

The problem with floating point

Syntax analysis

- Consider the expression `x1 + x2 + x3` with `x1=1.0, x2=1.0e10, x3=-1.0e10`
  - the left-recursive grammar implies a left-to-right evaluation order:
    `(x1 + x2) + x3`
    `= (1.0 + 1.0e10) + (-1.0e10) = (1.0e10) + (-1.0e10) = 0.0`

    This addition is problematic since 1.0 <<< 1.0e10 (LSBs get shifted out)

  - the right-recursive grammar implies a right-to-left evaluation order:
    `x1 + (x2 + x3)`
    `= 1.0 + (1.0e10 + (-1.0e10)) = 1.0 + 0.0 = 1.0`

- Obviously, these results should not differ. More details can be found in [3]

A parser with yacc: scanner

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- We've seen lex scanners already – each token is assigned a number (starting at 0 if nothing is specified):

Our scanner needs to print some output, so include the header here

example1.l

```
%{
#include <stdio.h>
enum { IF, THEN, ENDIF, INT, END };
%}
%%
[\n\t\v\ ]    { /* Do nothing, this is whitespace */ }
if            { return IF; }
then          { return THEN; }
endif         { return ENDIF; }
end           { return END; }
[0-9]+        { return INT; }
%%
```

In the declarations section you can include C code between %{ and }%. We used enums instead of #defines to automatically enumerate token numbers – yacc will do this for us automaticall

Code supplied for lex

```
<declarations>
%%
<translation rules>
%%
<functions>
```

- We needed a main function that repeatedly calls the generated scanner function yylex():

example1.l

```
<previous declarations>
%%
<previous regexps and actions>
%%
int main (void) {
  int token = 0;
  while (token != END) {
    token = yylex();
    switch (token) {
      case IF: printf ("Found if\n"); break;
      case THEN: printf ("Found then\n"); break;
      case ENDIF: printf ("Found endif\n"); break;
      case INT: printf ("Found integer %s\n", yytext); break;
      case END: printf ("Hanging up... bye\n"); break;
}}}
```

In a yacc/lex parser and scanner, yacc calls yylex() automatically for each token

We call yylex() for each token

The global variable yytext contains the character string of the scanned token

yacc is quite similar

- Description files also have three parts (definitions, rules and auxiliary C functions) separated by "%%":

```
<definitions>
%%
<rules>
%%
<auxiliary routines>
```

example1.y

```
/* definitions */
 ....

%%
/* rules */
....
%%

/* auxiliary routines */
....
```

yacc definitions

- Contain information about the tokens used in the syntax definition

```
<definitions>
%%
<rules>
%%
<auxiliary routines>
```

example1.y

```
%token NUMBER
%token ID
%token WORD 4711
%start nonterminal
%{
…
%}

%%
/* rules */
%%

/* auxiliary routines */
```

yacc will automatically assign token IDs, but you can override these

You can tell yacc which nonterminal symbol is the start symbol (default: the first)

Like in lex, you can include C code (headers, global vars,…) between %{ and %} here

yacc rules

- This defines the grammar in a BNF-like notations and related C actions

```
<definitions>
%%
<rules>
%%
<auxiliary routines>
```

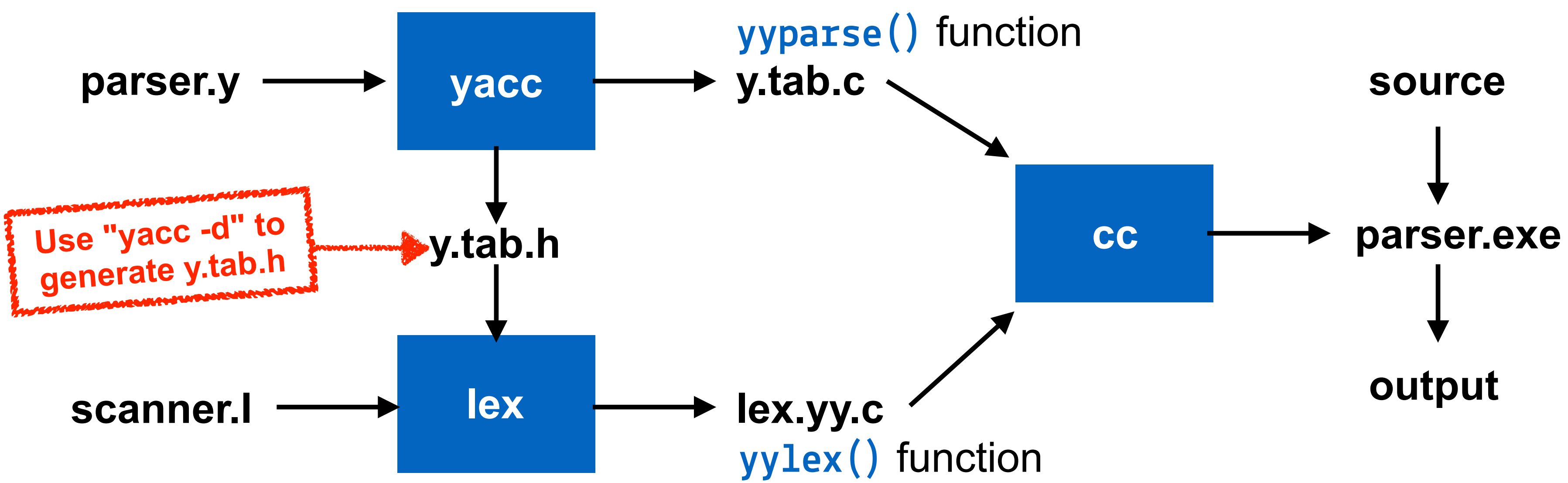
example1.y

```
…

%%
/* rules */

/* here comes your grammar */

%%

/* auxiliary routines */
int main(…)( {
   /* the main function is not automatically generated */
}
```

The grammar definition is similar to our notation and BNF

yacc-lex interaction

- yacc parsers assume the existence of function `yylex()` that implements the scanner (lex generated or handwritten)
- Scanner `yylex()` return value indicates the type of token found
  - Other values passed in variables `yytext` and `yylval`
- yacc determines integer representations (IDs) for tokens
  - Communicated to scanner in file **y.tab.h**



yacc example: parser

```
<definitions>
%%
<rules>
%%
<auxiliary routines>
```

A yacc parser to convert binary numbers to decimal

bindec.y

```
%{
#define YYDEBUG 1
#include <stdio.h>
#include <stdlib.h>

void yyerror(char *s);
int yylex(void);
extern char *yytext;

%}

%token ZERO ONE
%start N
```

Grammar, will be implemented in function yyparse()

Token IDs (→ y.tab.h)

y.tab.h

```
enum yytokentype
{
    ZERO = 258,
    ONE  = 259
};
```

```
%%
N : L     { printf("\n%d", $$); }
L : L B   { $$=$1*2+$2; }
  | B     { $$=$1; }
B : ZERO  { $$=$1; }
  | ONE   { $$=$1; }
%%
void yyerror(char *s)
{
  printf(\n%s: %s\n", s, yytext);
}

int main()
{
  while(yyparse());
}
```

Start parsing!

yacc example: scanner

```
<definitions>
%%
<rules>
%%
<auxiliary routines>
```

The lex scanner for our parser

bindec.l

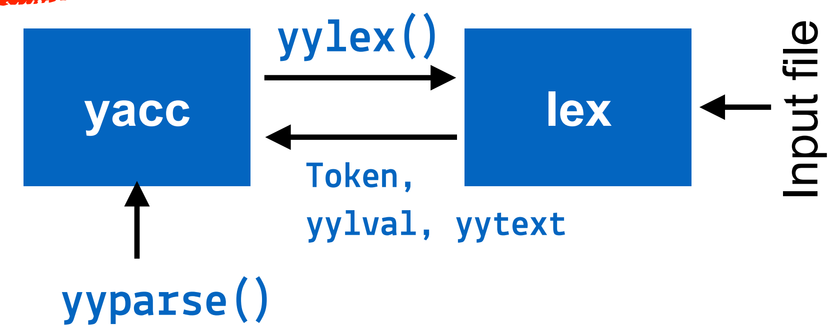
```
%{
  #include <stdio.h>
  #include <stdlib.h>
  #include "y.tab.h"
  extern int yylval;
%}
%%

0 { yylval=0; return ZERO; }
1 { yylval=1; return ONE; }

[ \t] {;}
\n return 0;
. return yytext[0];

%%
int yywrap()
{
  return 1;
}
```

Additional information about parsed token in yylval

Scanner description, implemented in yylex()

Token IDs ZERO/ONE returned to yyparse()

Numeric value for token passed in yylval



yyparse() and yylex()

- `yyparse()` called once (or repeatedly until EOF) from main (user-supplied)
- It repeatedly calls `yylex()` until done
  - On syntax error, calls `yyerror()` (user-supplied)
  - Returns 0 if all input was processed
  - Returns 1 if aborting due to syntax error

- `yylex()` called automatically (repeatedly) from `yyparse()`
  - Every time a new token is required by the parser
  - Its return value is the recognized token
    - Defined in `y.tab.h`, generated from `%token` declarations by yacc (option -d)
  - Token encoding: EOF = 0, character literals get their ASCII value, other tokens are assigned numbers > 127
  - Additional information passed back in variables `yylval` and `yytext`

yacc grammar actions

Like in lex, actions can be specified as C code after each production

- They are executed after the production RHS has been derived
- Special identifiers $$, $1, $2... refer to items on the parser's stack

```
%%

N : L     { printf("\n%d", $$); }
L : L B   { $$=$1*2+$2; }
  | B     { $$=$1; }
B : ZERO  { $$=$1; }
  | ONE   { $$=$1; }

%%
```

$1 is the *semantic value* of the first symbol on the right-hand side. For terminal symbols like ZERO and ONE, it stands for the value of `yylval` returned by the scanner.

$$ $1 $2

```
L : L B  { $$=$1*2+$2; }
```

$$ is the value returned by the production

yacc generates this line of C code:

```
{ yyval=yyvsp[-1]*2+yyvsp[0]; }
```

What’s next?

Syntax analysis

- Data types
- Semantic analysis

References

[1] Spenke, M., Mühlenbein, H., Mevenkamp, M., Mattern, F., & Beilken, C. (1984).
A Language Independent Error Recovery Method for LL(1) Parsers.
Softw., Pract. Exper., 14, 1095-1107

[2] Brett A. Becker et al. 2019.
Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research.
In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR ’19). ACM, New York, NY, USA, 177–210.
DOI:https://doi.org/10.1145/3344429.3372508

[3] David Goldberg. 1991.
What every computer scientist should know about floating-point arithmetic.
ACM Comput. Surv. 23, 1 (March 1991), 5–48. DOI:https://doi.org/10.1145/103162.103163