

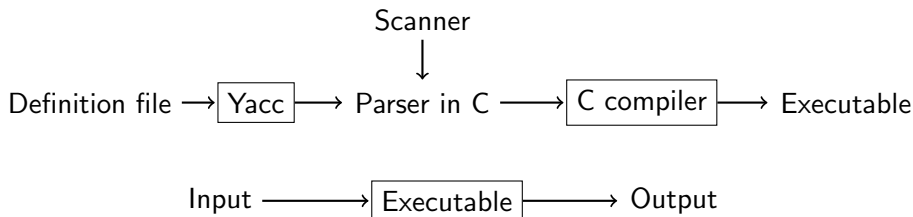
# Yacc

Jiří Techet    Tomáš Masopust    Alexander Meduna

Department of Information Systems  
Faculty of Information Technology  
Brno University of Technology  
Božetěchova 2, Brno 61266, Czech Republic

Modern Formal Language Theory, 2007

- tool for generating parsers
- parser described by context-free productions in a **definition file**
- scanner has to be provided (written manually or generated by Lex)
- Yacc processes the definition file and outputs a **parser** written in C
- this parser can be compiled by a **C compiler** to produce an **executable**
- the executable performs (LALR) **bottom up** parsing of its **input** and performs associated **actions** to produce its **output**



# Structure of Definition File I

## Structure of Definition File

```
%{  
    Prologue  
%}
```

Yacc declarations

```
%%  
Grammar rules  
%%
```

Epilogue

# Structure of Definition File II

- Yacc definition file divided into 3 parts which are separated by **%%**

## Parts of Definition File

### 1 prologue and declarations

- prologue
  - enclosed within **%{ %}**
  - contains any C code needed in actions (macros, function prototypes)
  - several prologues can be mixed with Yacc declarations
- declarations
  - specification of nonterminals, tokens, operator precedence, value types and others

### 2 grammar rules

- specification of grammar rules and associated **actions** performed when a rule is used in a reduction

### 3 epilogue

- any other code (typically definitions of `main()`, `yylex()`, `yyerror()`)

# Token Types

- defined by `%token`, `%left`, `%right`, or `%nonassoc` in the declarations part
- by convention, token name should be upper case
- `%token NUM`
- internally represented as C macros which assign a numerical code to every token type
- **literal character tokens** (`'+'`) and **literal string tokens** (`"<="`) do not have to be declared
- **associativity** defined by `%left`, `%right` and `%nonassoc`
- **precedence** defined by the order of their definition, lowest first

```
%left '-' '+'    /* lowest precedence */
```

```
%left '*' '/'
```

```
%left NEG
```

```
%right '^'      /* highest precedence */
```

## Attribute Types

- 1 if all tokens (and all semantic values) have the same type of their attributes, **YYSTYPE** macro can be used

```
%{  
    #define YYSTYPE double  
%}
```

```
%token NUM
```

- 2 if there are more types, all possible types defined by **%union**

```
%union {  
    double val;  
    symrec *tptr;  
}
```

# Attribute Type Assignment

## Terminal Type Assignment

- each token is assigned its attribute type by putting `<type>` in its definition

```
%union {  
    double val;  
    symrec *tptr;  
}
```

```
%token <val> NUM
```

## Nonterminal Type Assignment

- if `%union` is used, each nonterminal has to be assigned the type of its semantic value

```
%type <val> expr1 expr2
```

# Other Declarations I

## `%initial-action`

- allows to perform some initial actions before `yyparse` is called
- `$$`, `@$` and arguments of `%parse-param` can be used

## Example

```
%parse-param { char const *file_name };

%initial-action
{
    @$.initialize (file_name);
};
```



# Other Declarations II

## `%destructor`

- called when symbols are discarded to properly deallocate the memory (during error recovery, when the parser succeeds)

```
%destructor { code } symbols
```

- `$$` designates the semantic value associated with the discarded symbol
- invoked when user actions cannot change the memory

- 1 stacked symbols popped during the first phase of error recovery
- 2 incoming terminals during the second phase of error recovery
- 3 the current look-ahead and the entire stack when the parser returns immediately
- 4 the start symbol, when the parser succeeds

```
%union { char *string; }
```

```
%type <string> STRING
```

```
%destructor { free($$); } STRING
```

# Other Declarations III

## `%defines`

- write a header file containing macro definitions for token type names defined in the grammar
- used by `yyllex` if it is in another file
- if parser output file is `name.c` then the header file is `name.h`

## `%start`

- possible to specify the start symbol  
`%start S`
- by default, the first rule's left-hand side is the start symbol

# Grammar Rules

- consider the following context-free rules:

$$\text{exp} \rightarrow \varepsilon$$
$$\text{exp} \rightarrow \text{exp} + \text{exp}$$
$$\text{exp} \rightarrow \text{exp} - \text{exp}$$
$$\text{exp} \rightarrow \text{exp} * \text{exp}$$
$$\text{exp} \rightarrow \text{exp} / \text{exp}$$

- in definition file, these rules are represented as follows:

```
exp:                                /* empty line = empty string */
    | exp '+' exp /* | means alternative rhs */
    | exp '-' exp /* for the same lhs */
    | exp '*' exp
    | exp '/' exp
    ;                               /* end of rule */
```

- **actions** can be scattered among the symbols of the right-hand side
- rules in the grammar should be **left recursive**

# Context-Dependent Precedence

## %prec Modifier

- used to set priority when one operator is used for several functions (e.g. unary minus  $\times$  binary minus)

```
%left '+' '-'
```

```
%left '*'
```

```
%left UMIN
```

```
/* dummy operator with the highest priority */
```

```
exp:    exp '+' exp          { }
```

```
      | exp '-' exp          { }
```

```
      | exp '*' exp          { }
```

```
      | '-' exp %prec UMIN    { }
```

```
/* in this context '-' has the same priority as UMIN */
```

```
;
```

# Actions

- actions appear between `{ }` anywhere on the right-hand side of a rule
- usually at the end of a rule

## Semantic Values of Rule Components

`$$` semantic value of the nonterminal on the left-hand side

`$n` semantic value of the *n*th symbol on the right-hand side

- **default action** is `$$ = $1`
- if there are different types of semantic values (specified by `%union`), `$<type>$` and `$<type>n` have to be used

## Example

```
exp:    NUM          /* default action: $$ = $1 */
      | exp '+' exp  { $$ = $1 + $3; }      ;
```

# Locations I

- used to track locations of currently processed tokens in the input file
- useful for generating error messages

## YYLTYPE structure

- for each token, the scanner has to save its position to the variable `yylloc` which is of the type `YYLTYPE`

```
typedef struct YYLTYPE
{
    int first_line;
    int first_column;
    int last_line;
    int last_column;
} YYLTYPE;
```

## Location Values of Rule Components

- in parser, access similar to semantic values:
  - @\$ location of the nonterminal on the left-hand side
  - @n location of the *n*th symbol on the right-hand side

## Default Action for Locations

- executed each time a rule is matched
- by default, it sets the beginning of @\$ to the beginning of the first symbol, and the end of @\$ to the end of the last symbol on the rule's right-hand side – sufficient for most parsers
- can be redefined by **YYLLOC\_DEFAULT** macro

# Generated Parser

```
int yyparse()
```

- parses the input file
- returns 0 if parsing was successful, 1 if there was a syntax error, 2 if memory was exhausted
- in actions, **YYACCEPT** can be used to return 0 and **YYABORT** to return 1

```
int yylex()
```

- has to be provided by the user (written manually or by using Lex)
  - **returns token type**
  - **attribute** is stored in the global variable **yylval**
    - when using multiple attribute types (specified by %union), the corresponding member has to be used
- ```
yylval.intval = value; /* put value onto Yacc stack */  
return INT;           /* return the type of the token */
```



# Error Reporting and Recovery

```
void yyerror(char const *s)
```

- has to be provided by the user
- usually of the following form:

```
void yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}
```

## Error Recovery

- special token **error** which is generated when no rule can be used
- if there is a rule with the error token, parsing can recover
- can be explicitly invoked by **YYERROR** macro

## Example

```
stmts:    /* empty string */
         | stmts '\n'
         | stmts exp '\n'
         | stmts error '\n'    { yyerrok; }
         ;
```

- if there is an error in `exp`, recovery is performed as follows:
  - 1 tokens from `exp` which are already on the stack are discarded
  - 2 error is shifted
  - 3 input symbols are discarded until `'\n'` is the current input token
- by default, error messages are suppressed until 3 tokens successfully shifted – to avoid this **yyerrok** can be used

# Command Line Options

```
bison [OPTION]... FILE
```

## Selected Parameters

- o **outf** output file name
- p **pref** specifies other prefix than yy for Yacc functions
  - d same as %defines

## Options Within Definition File

- many options can be specified within the declarations part of the definition file
  - %defines

# Example I

## Example

```
/* Reverse polish notation calculator. */

%{
    #define YYSTYPE double
    #include <math.h>
    #include <ctype.h>
    #include <stdio.h>
    int yylex (void);
    void yyerror (char const *);
}%

%token NUM

%% /* Grammar rules and actions follow. */
```

## Example II

### Example

```
input:      /* empty */
           | input line
;
line:       '\n'
           | exp '\n'      { printf ("\t%.10g\n", $1); }
;
exp:        NUM            { $$ = $1; }
           | exp exp '+'   { $$ = $1 + $2; }
           | exp exp '-'   { $$ = $1 - $2; }
           | exp exp '*'   { $$ = $1 * $2; }
           | exp exp '/'   { $$ = $1 / $2; }
           | exp exp '^'   { $$ = pow ($1, $2); }
           | exp 'n'       { $$ = -$1; } /* Unary minus */
;
```

# Example III

## Example

```
%% /* Epilogue follows. */

/* The lexical analyzer returns a double floating point
   number on the stack and the token NUM, or the numeric
   code of the character read if not a number. It skips
   all blanks and tabs, and returns 0 for end-of-input. */

int yylex (void)
{
    int c;

    /* Skip white space. */
    while ((c = getchar ()) == ' ' || c == '\t')
        ;
}
```

## Example IV

### Example

```
/* Process numbers. */
if (c == '.' || isdigit (c))
{
    ungetc (c, stdin);
    scanf ("%lf", &yylval);
    return NUM;
}
/* Return end-of-input. */
if (c == EOF)
    return 0;
/* Return a single char. */
return c;
}
```

# Example V

## Example

```
/* Called by yyparse on error.  */
void yyerror (char const *s)
{
    fprintf (stderr, "%s\n", s);
}

int main (void)
{
    return yyparse ();
}
```





Bison documentation.

<http://www.gnu.org/software/bison/manual/index.html>.