Parsing

Lembit Jürimägi

Parser

Several types of parsers

- Bottom-up
  - Shift-reduce
  - LR - left to right scan, rightmost derivation
  - LALR - look-ahead LR
    - LALR(1) 1 token look-ahead
    - LALR(k) k token look-ahead
- Top-down
  - LL - left to right scan/generation, leftmost derivation
    - LL(k) k token look-ahead

Parse Tree

- Tree structure to represent syntax of a given sentence
  - Starting symbol as root element
  - Intermediary non-terminals as branches
  - Terminal elements as leaves
- Every required syntax element is present as a node in the tree
- Parser may omit generating a parse tree

Bison

- Open source version of classic AT&T tool yacc (yet another compiler compiler)
- Generates a LALR(1) parser based on rules section
- Uses Backus-Naur form for pattern matching
- Allows to be interfaced with flex or can be used on its own (but then you need to write your own scanner)
- Provides C functions and global variables for operating the parser
- Manual at: https://www.gnu.org/software/bison/manual/bison.html

Bison Source File Structure

- 4 sections

```
        top
        definitions / priority rules
        %%
        rules / actions
        %%
        code
```

- Some sections may be empty

Bison source file example (calc.y)

```
%{
#include <stdio.h>
extern int yylex(void);
void yyerror(const char *s);
%}
%error-verbose
%token NUM ADD
%%
root    : expr '\n'                    { printf("%d\n", $1); }
        ;
expr    : NUM                          { $$ = $1; }
        | ADD expr expr                { $$ = $2 + $3; }
        ;
%%
void yyerror(const char *s) { printf("ERROR: %s\n", s); }
int main(void) { return yyparse(); }
```

Flex source file example (calc.l)

```
%top{
#include "calc.tab.h"
}
%option noyywrap
%option warn
%%
ADD         { return ADD; }
[0-9]+      { yylval = atoi(yytext); return NUM; }
[\n]        { return *yytext; }
.           ; // filter everything else
%%
```

Bison: rules section

```
root :
     | root line
     ;
```

- `root` – starting nonterminal symbol
- `:` – Bison's version of ->, separates left and right side of rule
- `|` – same as above but without repeating the left side again
- `;` – ends the rules for current nonterminal (in this case `root`)
- `line` – another nonterminal, must have its own rules later on
- `\n` – no rule after "`:`", this declares that empty sentence is valid input

Bison: rules section, part 2

```
line : NUM ';'              { $$ = $1; }
     | NUM '+' NUM ';' { $$ = $1 + $2; }
     | error ';'            { yyerrok; }
     ;
```

- `NUM` – terminal symbol, because we defined it as token on a previous slide
- `';'` and `'+'` – also terminal symbols that weren't explicitly declared
- `{}` – action, a code section
- `$$` – refers to left side of rule at current line and its associated value
- `$1` – refers to first element (`NUM`) at the right side of rule and its value
- `error` – a grammar error in the input
- `yyerrok` – a macro to recover from the error without exiting

Bison: definitions section

- `%error-verbose` – provides detailed error messages when parser fails
- `token NUM` – lists terminal NUM
- `%left` – specifies that in a recursive rule leftmost terminal is solved (reduced) first, for example "`%left '+'`" means that in "a + b + c", "a + b" is solved first
- `%right` – specifies that in a recursive rule rightmost terminal is solved (reduced) first, for example "`%right '+'`" means that in "a + b + c", "b + c" is solved first

Bison: some command-line options

- -h – help
- -d – generates a *xxx*.tab.h header file (if source is *xxx*.y) to interface with flex
- -v – creates a *xxx*.output file with information about generated parser
- -o – lets you specify filename (default is *xxx*.tab.c)
- -p prefix – renames yy* variables and functions to prefix*
- -g – generates a *xxx*.dot graph description file (can be viewed at http://webgraphviz.com for example)

Example:

```
bison -d -v calc.y
```