Compilerconstructie

najaar 2012

`http://www.liacs.nl/home/rvvliet/coco/`

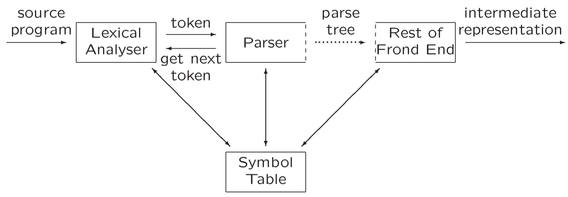
Rudy van Vliet
kamer 124 Snellius, tel. 071-527 5777
rvvliet(at)liacs.nl

college 4, dinsdag 25 september 2012

Syntax Analysis (2)

4.1 Parser's Position in a Compiler

(from college 3)



- Obtain string of tokens
- Verify that string can be generated by the grammar
- Report and recover from syntax errors

Parsing

(from college 3)

Finding parse tree for given string

- Universal (any CFG)
  - Cocke-Younger-Kasami
  - Earley
- Top-down (CFG with restrictions)
  - Predictive parsing
  - LL (Left-to-right, Leftmost derivation) methods
  - LL(1): LL parser, needs only one token to look ahead
- Bottom-up (CFG with restrictions)

Last week: top-down parsing
Today: bottom-up parsing

4.5 Bottom-Up Parsing

LR methods
Left-to-right scanning of input, Rightmost derivation (in reverse)

- Shift-reduce parsing
- Reduce string w to start symbol
- – Simple LR = SLR(1) = SLR
  – Canonical LR = canonical LR(1) = LR
  – Look-ahead LR = LALR

Bottom-Up Parsing (Example)

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

Construct parse tree for $\textbf{id} * \textbf{id}$ bottom-up. . .

Bottom-Up Parsing (Example)

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \textbf{id} \end{aligned}$$

Reducing a sentence

$\underline{\textbf{id}} * \textbf{id}$
$\underline{F} * \textbf{id}$
$T * \underline{\textbf{id}}$
$\underline{T * F}$
$\underline{T}$
E

Bottom-up parsing corresponds to rightmost derivation

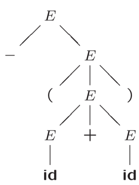
$$\begin{aligned} E &\underset{rm}{\Rightarrow} T \\ &\underset{rm}{\Rightarrow} T * F \\ &\underset{rm}{\Rightarrow} T * \textbf{id} \\ &\underset{rm}{\Rightarrow} F * \textbf{id} \\ &\underset{rm}{\Rightarrow} \textbf{id} * \textbf{id} \end{aligned}$$

Parse Trees and Derivations

(from college 3)

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \textbf{id}$$

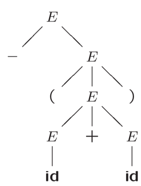
$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$



Many-to-one relationship between derivations and parse trees. . .

Parse Trees and Derivations

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E) \underset{lm}{\Rightarrow} -(E+E) \underset{lm}{\Rightarrow} -(\textbf{id}+E) \underset{lm}{\Rightarrow} -(\textbf{id}+\textbf{id})$$



Leftmost derivation	≈	WLR construction tree
	≈	top-down parsing
Rightmost derivation	≈	WRL construction tree
Bottom-up parsing	≈	LRW construction tree
	≈	rightmost derivation in reverse

Handles

Handle: substring that matches body of production, whose reduction represents one step along reverse of rightmost derivation

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

Reducing a sentence

$\underline{\textbf{id}} * \textbf{id}$
$\underline{F} * \textbf{id}$
$T * \underline{\textbf{id}}$
$\underline{T * F}$
$\underline{T}$
E

Bottom-up parsing corresponds to rightmost derivation

$$E \underset{rm}{\Rightarrow} T \underset{rm}{\Rightarrow} T * F \underset{rm}{\Rightarrow} T * \textbf{id} \underset{rm}{\Rightarrow} F * \textbf{id} \underset{rm}{\Rightarrow} \textbf{id} * \textbf{id}$$

Handles / not a handle...

Handle Pruning

- Formally, if $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta w$, then $A \rightarrow \beta$ is handle of $\alpha\beta w$
- Handle pruning to obtain rightmost derivation in reverse
  - w is string of terminals
  - $S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$
  - Locate handle β_n in γ_n and replace β_n ($A \rightarrow \beta_n$) to obtain right-sentential form γ_{n-1}
  - Repeat until we produce right-sentential form consisting of only S
- Problems
  - How to locate substring to be reduced?
  - How to determine what production to choose?

Shift-Reduce Parsing

Cf. bottom-up PDA from FI2

Use stack to hold symbols corresponding to part of input already read

- Initially,

Stack	Input
$	$w$$

- Repeat
  - Shift zero or more input symbols onto stack
  - Reduce a detected handle on top of stack

until error or

Stack	Input
$$S$	$

Shift-Reduce Parsing

Cf. bottom-up PDA from FI2

Use stack to hold symbols corresponding to part of input already read

Possbile actions shift-reduce parser:

- Shift — shift next symbol onto stack
- Reduce — replace handle on top of stack by nonterminal
- Accept — announce successful completion of parsing
- Error — detect syntax error and call error recovery routine

Shift-Reduce Parsing (Example)

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{id}$$

Stack	Input	Action
$	$\textbf{id}_1 * \textbf{id}_2$$	shift
$$\textbf{id}_1$	$*\textbf{id}_2$$	reduce by $F \rightarrow \textbf{id}$
$$F$	$*\textbf{id}_2$$	reduce by $T \rightarrow F$
$$T$	$*\textbf{id}_2$$	shift
$$T*$	$\textbf{id}_2$$	shift
$$T * \textbf{id}_2$	$	reduce by $F \rightarrow \textbf{id}$
$$T * F$	$	reduce by $T \rightarrow T * F$
$$T$	$	reduce by $E \rightarrow T$
$$E$	$	accept

Problems remain

- How to determine when to reduce
- How to determine what production to choose?

Conflicts

Sometimes stack contents and next input symbol are not sufficient to determine shift / (which) reduce

- Shift/reduce conflicts and reduce/reduce conflicts
- Caused by
  - Ambiguity of grammar
  - Limitation of the LR parsing method used (even when grammar is unambiguous)

Shift/Reduce Conflict (Example)

"Dangling-else"-grammar

stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other**

Stack	Input	Action
$...	...$	...
$...**if** *expr* **then if** *expr* **then** *stmt*	**else**...$	shift or reduce?

Resolve in favour of shift,
so **else** matches closest unmatched **then**

Reduce/Reduce Conflict (Example)

stmt → **id** (*parameter_list*) | *expr* := *expr*
parameter_list → *parameter_list*, *parameter* | *parameter*
parameter → **id**
expr → **id** (*expr_list*) | **id**
expr_list → *expr_list*, *expr* | *expr*

Statement beginning with `p(i,j)` would appear as token stream **id** (**id**, **id**)

Stack	Input	Action
$...	...$	...
$...**id** (**id**	,**id**)...$	reduce by *parameter* → **id** or by *expr* → **id** ?

Reduce/Reduce Conflict (Example)

Possible solution

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textbf{procid}\ (\textit{parameter\_list}\ )\ \mid\ \textit{expr} := \textit{expr} \\
\textit{parameter\_list} & \rightarrow & \textit{parameter\_list}, \textit{parameter}\ \mid\ \textit{parameter} \\
\textit{parameter} & \rightarrow & \textbf{id} \\
\textit{expr} & \rightarrow & \textbf{id}\ (\textit{expr\_list}\ )\ \mid\ \textbf{id} \\
\textit{expr\_list} & \rightarrow & \textit{expr\_list}, \textit{expr}\ \mid\ \textit{expr}
\end{array}
$$

Requires more sophisticated lexical analyser

Stack	Input	Action
$...	...$	...
$... **procid** (**id**	, **id**)...$	reduce by *parameter* → **id**

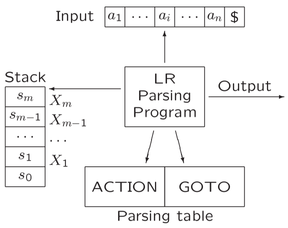
or

Stack	Input	Action
$...	...$	...
$... **id** (**id**	, **id**)...$	reduce by *expr* → **id**

4.6 LR Parsing

- Bottom-up parsing for large class of CFGs
- LR(k)
  - Left-to-right scanning of input
  - Rightmost derivation in reverse
  - k symbols of look-ahead
- LR parser pros:
  - Covers all programming language constructs
  - Most general non-backtracking shift-reduce parsing
  - Allows efficient implementation
  - Detects syntactic errors as soon as possible (in left-to-right scanning)
  - Can parse more grammars than LL(k) parsers
- LR parser con: too much work to be constructed by hand, but: LR parser generators available

LR Parsing



Simple LR Parsing

States are sets of LR(0) items

Production $A \rightarrow XYZ$ yields four items:

$$
\begin{array}{rcl}
A & \rightarrow & \cdot XYZ \\
A & \rightarrow & X \cdot YZ \\
A & \rightarrow & XY \cdot Z \\
A & \rightarrow & XYZ \cdot
\end{array}
$$

Item indicates how much of production we have seen in input

LR(0) items are combined in sets

Canonical LR(0) collection is specific collection of item sets
These item sets are the states in LR(0) automaton,
a DFA that is used for making parsing decisions

Closure of Item Sets

- – Consider $A \rightarrow \alpha \cdot B\beta$
  - We expect to see substring derivable from $B\beta$, with prefix derivable from B, by applying B-production
  - Hence, add $B \rightarrow \cdot\gamma$ for all $B \rightarrow \gamma$
- Let I be item set
  1. Add every item in I to CLOSURE(I)
  2. Repeat

     If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE(I) and $B \rightarrow \gamma$ is production, then add $B \rightarrow \cdot\gamma$ to CLOSURE(I)

     until no more new items are added

Closure of Item Sets (Example)

Augmented grammar

$$
\begin{array}{rcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

If $I = \{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) = ...

Closure of Item Sets (Example)

Augmented grammar

$$
\begin{array}{rcl}
E' & \rightarrow & E \\
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{id}
\end{array}
$$

If $I = \{[E' \rightarrow \cdot E]\}$, then CLOSURE($I$) = I_0:

I_0
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot$**id**

Function GOTO

- Let I be set of items, and X be grammar symbol
- GOTO(I, X): items you can get by moving · over X in items from I (and then taking closure)

Example:

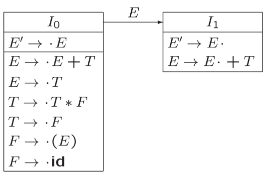
I_0
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + T$
$E \rightarrow \cdot T$
$T \rightarrow \cdot T * F$
$T \rightarrow \cdot F$
$F \rightarrow \cdot (E)$
$F \rightarrow \cdot$**id**

GOTO(I_0, E) = ...

Function GOTO

- Let I be set of items, and X be grammar symbol
- GOTO(I, X): items you can get by moving $\cdot$ over X in items from I (and then taking closure)

Example:

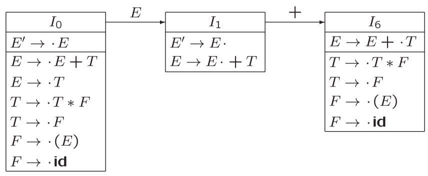


GOTO$(I_0, E) = I_1$ GOTO$(I_1, +) = \ldots$

Function GOTO

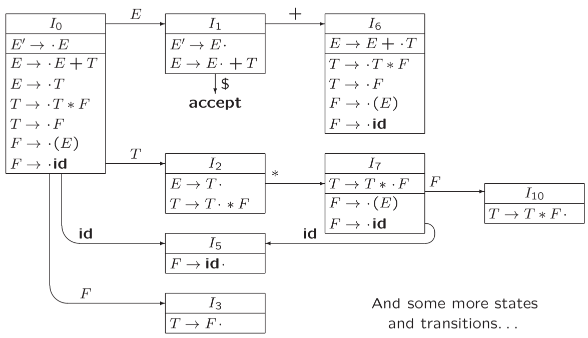
- Let I be set of items, and X be grammar symbol
- GOTO(I, X): items you can get by moving $\cdot$ over X in items from I (and then taking closure)

Example:



GOTO$(I_0, E) = I_1$ GOTO$(I_1, +) = I_6$

LR(0) Automaton (Example)



And some more states and transitions. . .

Use of LR(0) Automaton

- Repeat
  - If possible, then shift on next input symbol
  - Otherwise, reduce

  until error or accept

- Example: parsing **id** $*$ **id**

Line	Stack	Symbols	Input	Action
(1)	0	\$	**id** $*$ **id**\$	. . .

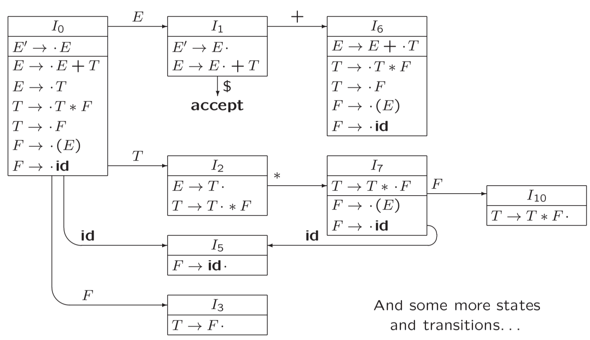
Use of LR(0) Automaton

- Repeat
  - If possible, then shift on next input symbol
  - Otherwise, reduce

  until error or accept

- It is not as simple as this: there may be
  - shift/reduce conflicts
  - reduce/reduce conflicts

LR(0) Automaton (Example)



And some more states and transitions. . .

Possible Actions in SLR Parsing

For state i and input symbol a,

- if $[A \to \alpha \cdot a \beta]$ is in I_i and GOTO$(I_i, a) = I_j$ then shift j is possible (a must be terminal, not \$)
- if $[A \to \alpha \cdot]$ is in I_i and $a \in$ FOLLOW(A), then reduce $A \to \alpha$ is possible (A may not be S')
- if $[S' \to S \cdot]$ is in I_i and $a =$ \$, then accept is possible

If conflicting actions result from this, then grammar is not SLR(1)

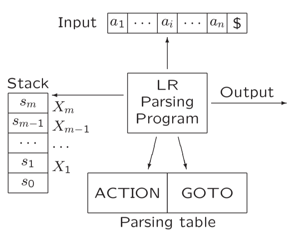
LR Parser

SLR, LR, LALR

For state i and terminal a, ACTION$[i, a]$, can have four possible values:

1. shift (state) j
2. reduce $A \to \beta$
3. accept
4. error

For state i and nonterminal A, GOTO$[i, A]$ is state j



Behaviour of LR Parser

LR parser configuration is pair (stack contents, remaining input):

$$(s_0 s_1 s_2 \ldots s_m, a_i a_{i+1} \ldots a_n\$)$$

which represents right-sentential form

$$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$$

1. If ACTION$[s_m, a_i]$ = shift s, then push s and advance input:
$$(s_0 s_1 s_2 \ldots s_m s, a_{i+1} \ldots a_n\$)$$
2. If ACTION$[s_m, a_i]$ = reduce $A \to \beta$, where $|\beta| = r$, then pop r symbols. If GOTO$[s_{m-r}, A] = s$, then push s:
$$(s_0 s_1 s_2 \ldots s_{m-r} s, a_i a_{i+1} \ldots a_n\$)$$
3. If ACTION$[s_m, a_i]$ = accept, then stop
4. If ACTION$[s_m, a_i]$ = error, then call error recovery routine

SLR Parsing Table (Example)

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to$ **id**

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Blank means error

Line	Stack	Symbols	Input	Action
(1)	0	$	**id** $*$ **id**$	shift to 5
(2)	05	$**id**	$*$**id**$	reduce by $F \to$ **id**
(3)	03	$$F$	$*$**id**$	...

Different LR Parsing Methods

- Simple LR = SLR
  - Easiest to implement, least powerful
- Canonical LR
  - Augment SLR with lookahead information LR(1) items: $[A \to \alpha \cdot \beta, a]$
  - Most expensive to implement, most powerful
- Look-ahead LR = LALR
  - Merge sets of LR(1)-items, so fewer states
  - Often used in practice
- All parsers have same behaviour They differ in how parsing table is built

Compaction of LR Parsing Tables

- Typical grammar: 100 terminals and productions
  - Several hundreds of states, 20,000 action entries
- Two-dimensional array is not efficient
- Compacting action field of parsing table
  - Many rows are identical, so create pointer for each state into one-dimensional array
  - Make list for actions of each state, consisting of pairs (terminal-symbol, action)

Compaction of Parsing Table (Example)

State	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

List for states 0, 4, 6, 7:

Symbol	Action
id	s5
(	s4
any	error

List for state 1:

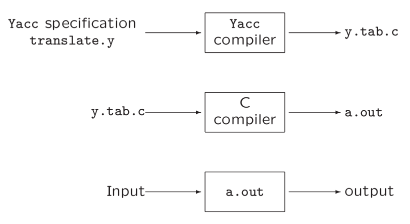
Symbol	Action
+	s6
$	acc
any	error

4.9 Parser Generators

`Yacc`: Yet Another Compiler Compiler

- Is an LALR(1) parser generator
- Automatically produces parser for CFG
- Deals with ambiguity and difficult-to-parse constructs
  - Reports on conflicts
- Available as command on Unix

Yacc: Parser Generator




```
yacc translate.y
gcc y.tab.c -ly
./a.out
```


Yacc Specification

- A `Yacc` program consists of three parts:

  declarations
  %%
  translation rules
  %%
  auxiliary functions

- Translation rules are of the form

  production { semantic actions }

$$\begin{array}{lll} \langle\text{head}\rangle & : & \langle\text{body}\rangle_1 \ \{\langle\text{semantic action}\rangle_1\} \\ & | & \langle\text{body}\rangle_2 \ \{\langle\text{semantic action}\rangle_2\} \\ & & \ldots \\ & | & \langle\text{body}\rangle_n \ \{\langle\text{semantic action}\rangle_n\} \\ & ; & \end{array}$$

Yacc Specification (Example)

Example: Desktop calculator with following grammar

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \textbf{digit}$$

```
/* declarations section */
%{
#include <ctype.h>
}%

%token DIGIT

%%
/* translation rules section */

line   : expr '\n'        { printff("%d\n", $1); }
       ;
```


Yacc Specification (Example)

```
expr   : expr '+' term     { $$ = $1 + $3; }
       | term
       ;
term   : term '+' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;

%%
/* auxiliary functions section */
yylex()
{   int c;
    c = getchar();
    if (isdigit(c))
    {   yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```


Yacc and Ambiguous Grammars

- Ambiguous grammar for our calculator:

  $E \rightarrow E{+}E \;\mid\; E{-}E \;\mid\; E{*}E \;\mid\; E/E \;\mid\; (E) \;\mid\; -E \;\mid\;$ **number**

- Allow sequence of expressions and blank lines:

```
lines : lines expr '\n'   { printf("%f\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```

- LALR algorithm will generate parsing action conflicts
  - invoke Yacc with -v option

Yacc Specification (Example)

```
/* declarations section */
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double  /* double type for Yacc stack */
*}

%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS

%%
/* translation rules section */

lines : lines expr '\n'   { printf("%f\n", $2); }
      | lines '\n'
      | /* empty */
      ;
```


Yacc Specification (Example)

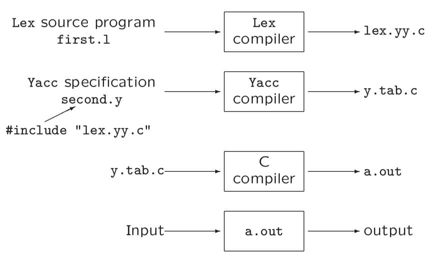
```
expr  : expr '+' expr     { $$ = $1 + $3; }
      | expr '-' expr     { $$ = $1 - $3; }
      | expr '*' expr     { $$ = $1 * $3; }
      | expr '/' expr     { $$ = $1 / $3; }
      | '(' expr ')'      { $$ = $2; }
      | '-' expr  %prec UMINUS { $$ = - $2; }
      | NUMBER
      ;

%%
/* auxiliary functions section */
yylex()
{   int c;
    while ( ( c = getchar() ) == ' ' );
    if ( (c== '.') || (isdigit(c)) )
    { ungetc(c, stdin);
      scanf("%lf", &yylval);
      return NUMBER;
    }
    return c;
}
```


Precedence and Associativity

- Same precedence and left associative:
  `%left '+' '-'`
- Right associative:
  `%right '^'`
- Increasing precedence:
  `%left '+' '-'`
  `%left '*' '/'`
  `%right UMINUS`
- Non-associative binary operator:
  `%nonassoc '<'`
- Precedence and associativity to each production
  - Default: rightmost operator
  - Otherwise: `%prec` ⟨terminal⟩

  `expr : '-' expr  %prec UMINUS { $$ = - $2; }`

Combining Yacc with Lex




```
lex first.l
yacc second.y
gcc y.tab.c -ly -ll
./a.out
```


Volgende week

- Practicum over opdracht 1
- Eerst naar 403, daarna naar 306/308
- Staat al online
- Inleveren 8 oktober

Compiler constructie

college 4
Syntax Analysis (2)

Chapters for reading: 4.5–4.7, 4.9