NTNU | Norwegian University of Science and Technology

Compiler Construction

Theoretical Exercise 3: Parsing

Michael Engel

3.1 Top-down parsing: LL(1) form

The following grammar is given:

$S \rightarrow aBT \mid aBTwK$
$B \rightarrow b$
$T \rightarrow t \mid \epsilon$
$K \rightarrow Ks \mid s$

a. Modify the grammar so that it becomes suitable for LL(1) parsing by left factoring and eliminating left recursion.

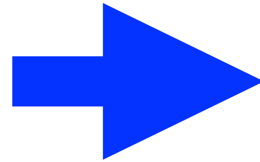
- Provide *unique prefixes* and rewrite *left recursion:*

$S \rightarrow aBTS'$
$S' \rightarrow wK \mid \epsilon$
$B \rightarrow b$
$T \rightarrow t \mid \epsilon$
$K \rightarrow sK'$
$K' \rightarrow sK' \mid \epsilon$

3.1 Top-down parsing: LL(1) form

The following grammar is given:

```
S → aBT | aBTwK
B → b
T → t | ϵ
K → Ks | s
```



```
S  → aBTS'
S' → wK | ϵ
B  → b
T  → t | ϵ
K  → sK'
K' → sK' | ϵ
```

b. Tabulate the FIRST and FOLLOW sets for all nonterminals of the modified grammar, including which nonterminals are nullable (i.e. they can derive the empty string).

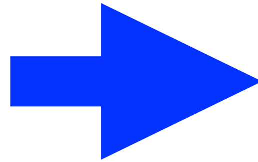
	S	S’	T	B	K	K’
FIRST	a	w	t	b	s	s
FOLLOW	$	$	$,w	$,t,w	$	$
nullable	no	yes	yes	no	no	yes

Online grammar analyzer: https://smlweb.cpsc.ucalgary.ca/vital-stats.php

3.1 Top-down parsing: LL(1) form

The following grammar is given:

```
S → aBT | aBTwK
B → b
T → t | ϵ
K → Ks | s
```



```
S  → aBTS'
S' → wK | ϵ
B  → b
T  → t | ϵ
K  → sK'
K' → sK' | ϵ
```

c. Construct the LL(1) parsing table of the modified grammar.

	a	w	t	b	s	$
S	S → aBTS'					
S'		S' → wK				S' → ε
T		T → ε	T → t			T → ε
B				B → b		
K					K → sK'	
K'					K' → sK'	K' → ε

3.2 LR(1) parsing

```
Proto  → Type “Id” “(” Params “)” “;”
Type   → “int” | “double” | “char”
Params → Params “,” Param | Param
Param  → Type “Id”
```

Trace through the sequence of shift and reduce actions by an LR parser on the input: `double blarf(char foo, int bar);`

```
         ↑ double blarf(char foo, int bar);
shift:   double ↑ blarf(char foo, int bar);      // Type → “double”
reduce:  Type ↑ blarf(char foo, int bar);
shift:   Type blarf ↑ (char foo, int bar);
shift:   Type blarf ( ↑ char foo, int bar);
shift:   Type blarf ( char ↑ foo, int bar);      // Type → “char”
reduce:  Type blarf ( Type ↑ foo, int bar);
shift:   Type blarf ( Type foo ↑, int bar);      // Param → Type “id”
reduce:  Type blarf ( Param ↑, int bar);         // Params → Param
reduce:  Type blarf ( Params ↑, int bar);
         .......
```

3.2 LR(1) parsing

```
Proto  → Type "Id" "(" Params ")" ";"
Type   → "int" | "double" | "char"
Params → Params "," Param | Param
Param  → Type "Id"
```

a. Trace through the sequence of shift and reduce actions by an LR parser on the input: `double blarf(char foo, int bar);`

- You can do this in yacc/lex!

```
// lex file: grammar.l
%{
#include <stdio.h>
#include "y.tab.h"
%}

%%

"int"            { printf("shift int\n"); return Tint; }
"double"         { printf("shift double\n"); return Tdouble; }
"char"           { printf("shift char\n"); return Tchar; }
[A-Za-z0-9]+     { printf("shift id\n"); return Tid; }
[ \t\n]          { ; }
.                { printf("shift %c\n", yytext[0]);
                   return yytext[0]; }
%%

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

3.2 LR(1) parsing

```
Proto  → Type "Id" "(" Params ")" ";"
Type   → "int" | "double" | "char"
Params → Params "," Param | Param
Param  → Type "Id"
```

a. Trace through the sequence of shift and reduce actions by an LR parser on the input: `double blarf(char foo, int bar);`

- You can trace the parsing using yacc itself!

```
// yacc file: grammar.y
%{
#include <stdio.h>
int yylex (void);
void yyerror (char *s);
#define TRACE printf("reduce at line %d\n", __LINE__);
%}
%token Tint Tdouble Tchar Tid
%%

PROTO  : TYPE Tid '(' PARAMS ')' ';' { TRACE $$ = $1; }
TYPE   : Tint { TRACE }
       | Tdouble { TRACE }
       | Tchar  { TRACE }
       ;
PARAMS : PARAMS ',' PARAM { TRACE }
       | PARAM  { TRACE }
       ;
PARAM  : TYPE Tid  { TRACE }
       ;
%%
void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
int main (void) {
   return yyparse ();
}
```

3.2 LR(1) parsing

Output of example run:

```
double blarf(char foo, int bar);
shift double
reduce at line 21
shift id
shift (
shift char
reduce at line 22
shift id
reduce at line 29
reduce at line 26
shift ,
shift int
reduce at line 20
shift id
reduce at line 29
reduce at line 25
shift )
shift ;
reduce at line 18
```

```
   // yacc file: grammar.y
   %{
   #include <stdio.h>
   int yylex (void);
   void yyerror (char *s);
   #define TRACE printf("reduce at line %d\n", __LINE__);
   %}
   %token Tint Tdouble Tchar Tid
   %%

18 PROTO : TYPE Tid '(' PARAMS ')' ';' { TRACE $$ = $1; }

20 TYPE  : Tint { TRACE }
21       | Tdouble { TRACE }
22       | Tchar   { TRACE }
         ;

25 PARAMS : PARAMS ',' PARAM { TRACE }
26        | PARAM   { TRACE }
          ;

29 PARAM : TYPE Tid  { TRACE }
         ;
   %%
   void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
   int main (void) {
      return yyparse ();
   }
```

3.2 LR(1)

```
Starting parse
Entering state 0
Stack now 0
Reading a token
double id(char id, int id);
shift double
Next token is token Tdouble ()
Shifting token Tdouble ()
Entering state 2
Stack now 0 2
Reducing stack by rule 3 (line 20):
   $1 = token Tdouble ()
-> $$ = nterm TYPE ()
Entering state 5
Stack now 0 5
Reading a token
shift id
Next token is token Tid ()
Shifting token Tid ()
Entering state 7
Stack now 0 5 7
Reading a token
shift (
Next token is token '(' ()
Shifting token '(' ()
Entering state 8
Stack now 0 5 7 8
Reading a token
… 114 lines of output …
```

Alternative: YYDEBUG

```
   // yacc file: grammar.y
   %{
   #include <stdio.h>
   int yylex (void);
   void yyerror (char *s);
   #define YYDEBUG 1
   %}
   %token Tint Tdouble Tchar Tid
   %%

17 PROTO : TYPE Tid '(' PARAMS ')' ';' { TRACE $$ = $1; }

19 TYPE   : Tint { TRACE }
20        | Tdouble { TRACE }
21        | Tchar   { TRACE }
          ;

24 PARAMS : PARAMS ',' PARAM { TRACE }
25         | PARAM   { TRACE }
           ;

28 PARAM : TYPE Tid  { TRACE }
         ;
   %%
   void yyerror (char *s) {fprintf (stderr, "%s\n", s);}
   int main (void) {
   #if YYDEBUG
      yydebug = 1;
   #endif
      return yyparse ();
   }
```

3.2 LR(1) parsing

Consider the grammar (terminal symbols are enclosed in double quotes, for the terminal symbol "Id" the value of the identifier token is a sequence of ASCII characters):

```
Proto  → Type "Id" "(" Params ")" ";"
Type   → "int" | "double" | "char"
Params → Params "," Param | Param
Param  → Type "Id"
```

b. Why does the parser not attempt a reduction to Param after pushing the first sequence of type and identifier onto the parse stack? Doesn't the sequence on top of the stack match the right side? What other requirements must be met before a reduction is performed?

- It turns out that yacc actually does this according to its output… oops

```
Stack now 0 5 7 8 9 12
Reducing stack by rule 7 (line 28):
   $1 = nterm TYPE ()
   $2 = token Tid ()
-> $$ = nterm PARAM ()
```

```
28 PARAM : TYPE Tid  { TRACE }
```

3.2 LR(1) parsing

```
Proto  → Type “Id” “(” Params “)” “;”
Type   → “int” | “double” | “char”
Params → Params “,” Param | Param
Param  → Type “Id”
```

c. How many tokens of lookahead does this parser require? …

…so, what is "lookahead"?

- LR(k) parsers use the curent parser state (of the DFA) and k lookahead symbols to decide *whether to reduce* and if so, by which production.
- LR(k) parsers also use a shift transition table to decide which parsing state it should move to after shifting the next input token.
- The shift transition table is keyed by the current state *and the (single) token being shifted*, regardless of the value of k.
- So, *shifting always requires lookahead*, whereas *reducing* in LR(0) doesn’t!

3.2 LR(1) parsing

```
Proto  → Type “Id” “(” Params “)” “;”
Type   → “int” | “double” | “char”
Params → Params “,” Param | Param
Param  → Type “Id”
```

c. How many tokens of lookahead does this parser require? …

This means that the lookahead is used to predict which, if any, reduction should be applied.

- In an LR(0) parser, the decision to shift (more accurately, to attempt to shift) must be made before reading the next input token
- But the computation of the state to transition to do is made after reading the token, at which point it will signal an error if no shift is possible.

- This grammar doesn’t need a lookahead to decide to shift (that’s why we started with the intuitive analysis), so it is LR(0)

3.2 LR(1) parsing

```
Proto  → Type "Id" "(" Params ")" ";"
Type   → "int" | "double" | "char"
Params → Params "," Param | Param
Param  → Type "Id"
```

c. …Is the given grammar LR(0) (i.e., it can be parsed by the weakest of the LR parsers) or does it require a larger lookahead?

The formal way to do the analysis:

- To prove that a grammar is LR(0), build the DFA of viable prefixes.
  - The first prefix should always S -> ● S' (extend the grammar by this rule to make the analysis easier).
- Then follow the rules to build the LR(0) parsing table.
- A grammar is LR(0) if each position in the parsing table LR(0) contains at most one element.

3.2 LR(1) parsing

Difference between LL(k) and LR(k) parsers here:

In both cases, lookahead has the same meaning: it consists of looking at input tokens without moving the input cursor.

If k is 0, then:

- An LR(0) parser must decide whether or not to reduce without examining input
  - No state can have either two different reduce actions or a reduce and a shift action.
- An LL(0) parser must decide which production of a given non-terminal is appicable without examining input.
  - Thus each non-terminal can have only one production, so the language must be finite.

3.3 LR(1) parse tables

The following grammar is given:

S → XX
X → xX | y

Construct the parsing table for this grammar using LR(1).

Doing this by hand is tedious!

The table is already quite large for this very simple grammar…

state	$	y	x	S	X
0		s4	s3	g2	g1
1		s8	s7		g6
2	accept				
3		s4	s3		g5
4		r(X → y)	r(X → y)		
5		r(X → xX)	r(X → xX)		
6	r(S → XX)				
7		s8	s7		g9
8	r(X → y)				
9	r(X → xX)				