Compilers (CS31003)

Lecture 18-19

Bison Specification

Compiler Phases

- **Lexical Analyser**: We have already discussed how to write a simple lexical analyser using Flex.
- **Syntax Analyser**: We show how to write a parser for a simple expression grammar using Bison.
- **Semantic Analyser**: We extend the parser of expression grammar semantically:
  - To build a Simple Calculator from the expression grammar (computational semantics).
  - To build a programmable calculator from the simple calculator (identifier / storage semantics).

We show how parser / translator generators can be simplified by using Ambiguous Grammar.

Bison Specs – Fundamentals

- Like Flex, has three sections – Definition, Rules, and Auxiliary
- Terminal Symbols
  - Symbolized terminals (like `NUMBER`) are identified by `%token`. Usually, but not necessarily, these are multi-character.
  - Single character tokens (like `'+'`) may be specified in the rules simply with quotes.
- Non-Terminal Symbols
  - Non-Terminal symbols (like `expression`) are identified by `%type`.
  - Any symbol on the left-hand side of a rule is a non-terminal.
- Production Rules
  - Production rules are written with left-hand side non-terminal separated by a colon (`:`) from the right-hand side symbols.
  - Multiple rules are separated by alternate (|).
  - ϵ productions are marked by empty right-hand side.
  - Set of rules from a non-terminal is terminated by semicolon (`;`).
- Start Symbol
  - Non-terminal on the left-hand side of the first production rule is taken as the start symbol by default.
  - Start symbol may be explicitly defined by `%start`: `%start statement`.

Simple Expression Parser

A Simple Expression Grammar

$$
\begin{array}{rlcl}
1: & S & \rightarrow & E \\
2: & E & \rightarrow & E + T \\
3: & E & \rightarrow & E - T \\
4: & E & \rightarrow & T \\
5: & T & \rightarrow & T * F \\
6: & T & \rightarrow & T \;/\; F \\
7: & T & \rightarrow & F \\
8: & F & \rightarrow & (E) \\
9: & F & \rightarrow & - F \\
10: & F & \rightarrow & \textbf{num}
\end{array}
$$

Expressions involve only constants, operators, and parentheses and are terminated by a \$.

Flex Specs (calc.l) for Simple Expressions

```
%{
#include "y.tab.h" // Generated from Bison
#include <math.h>
%}

%%
[1-9]+[0-9]*    {
                     return NUMBER;
                }

[ \t]           ;  /* ignore white space */

"$"             {
                     return 0; /* end of input */
                }

\n|.            return yytext[0];
%%
```

Bison Specs (calc.y) for Simple Expression Parser

1:	S	$\rightarrow$	E
2:	E	$\rightarrow$	$E + T$
3:	E	$\rightarrow$	$E - T$
4:	E	$\rightarrow$	T
5:	T	$\rightarrow$	$T * F$
6:	T	$\rightarrow$	$T \, / \, F$
7:	T	$\rightarrow$	F
8:	F	$\rightarrow$	(E)
9:	F	$\rightarrow$	$- F$
10:	F	$\rightarrow$	**num**

```
%{ /* C Declarations and Definitions */
#include <string.h>
#include <iostream>
extern int yylex(); // Generated by Flex
void yyerror(char *s);
%}

%token NUMBER

%%
statement: expression
         ;
expression: expression '+' term
          | expression '-' term
          | term
          ;
```

```
term: term '*' factor
    | term '/' factor
    | factor
    ;
factor: '(' expression ')'
      | '-' factor
      | NUMBER
      ;
%%

void yyerror(char *s) { // Called on error
    std::cout << s << std::endl;
}

int main() {
    yyparse(); // Generated by Bison
}
```

Note on Bison Specs (calc.y)

- Terminal Symbols
  - Symbolized terminals (like `NUMBER`) are identified by `%token`. Usually, but not necessarily, these are multi-character. These are defined as manifest constants in `y.tab.h`
  - Single character tokens (like `'+'`) may be specified in the rules simply with quotes.
- Non-Terminal Symbols
  - Non-Terminal symbols (like `expression`) are identified by `%type`.
  - Any symbol on the left-hand side of a rule is a non-terminal.
- Production Rules
  - Production rules are written with left-hand side non-terminal separated by a colon (`:`) from the right-hand side symbols.
  - Multiple rules are separated by alternate (|).
  - ϵ productions are marked by empty right-hand side.
  - Set of rules from a non-terminal is terminated by semicolon (`;`).
- Start Symbol
  - Non-terminal on the left-hand side of the first production rule is taken as the start symbol by default.
  - Start symbol may be explicitly defined by `%start`: `%start statement`.

Simple Calculator

A Simple Calculator Grammar

$$
\begin{array}{rlll}
1: & S & \rightarrow & E \\
2: & E & \rightarrow & E + T \\
3: & E & \rightarrow & E - T \\
4: & E & \rightarrow & T \\
5: & T & \rightarrow & T * F \\
6: & T & \rightarrow & T \;/\; F \\
7: & T & \rightarrow & F \\
8: & F & \rightarrow & (E) \\
9: & F & \rightarrow & -F \\
10: & F & \rightarrow & \textbf{num}
\end{array}
$$

- We build a calculator with the simple expression grammar
- Every expression involves only constants, operators, and parentheses and are terminated by a $
  - Need to bind its *value* to a *constant* (terminal symbol)
  - Need to bind its *value* to an *expression* (non-terminal symbol)
- On completion of parsing (and processing) of the expression, the evaluated value of the expression should be printed

Bison Specs (calc.y) for Simple Calculator

```
%{ /* C Declarations and Definitions */
#include <string.h>
#include <iostream>
extern int yylex();
void yyerror(char *s);
%}

%union { // Placeholder for a value
    int intval;
}

%token <intval> NUMBER

%type <intval> expression
%type <intval> term
%type <intval> factor

%%
statement: expression
               { printf("= %d\n", $1); }
         ;
expression: expression '+' term
                { $$ = $1 + $3; }
          | expression '-' term
                { $$ = $1 - $3; }
          | term
          ;
term: term '*' factor
          { $$ = $1 * $3; }
    | term '/' factor
          { if ($3 == 0)
              yyerror("divide by zero");
            else $$ = $1 / $3;
          }
    | factor
    ;
factor: '(' expression ')'
           { $$ = $2; }
      | '-' factor
           { $$ = -$2; }
      | NUMBER
      ;
%%

void yyerror(char *s) {
    std::cout << s << std::endl;
}

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

Note on Bison Specs (calc.y)

- Attributes
  - Every terminal and non-terminal has an (optional) attribute.
  - Multiple types of attributes are possible. They are bundled in a `C` `union` by `%union`.
  - An attribute is associated with a terminal by the `%token`: `%token <intval> NUMBER`
  - An attribute is associated with a non-terminal by the `%type`: `%type <intval> term`
- Actions
  - Every production rule has an action (`C` code snippet) at the end of the rule that fires when a reduction by the rule takes place.
  - In an action the attribute of the left-hand side non-terminal is identified as `$$` and the attributes of the symbols on the right-hand side are identified as `$1`, `$2`, `$3`, ... counting from left to right.
  - Missing actions for productions with single right-hand side symbol (like `factor` $\rightarrow$ `NUMBER`) imply a default action of copying the attribute (should be of compatible types) from the right to left: `{ $$ = $1 }` .

Header (y.tab.h) for Simple Calculator

```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers
      know about them.  */
   enum yytokentype {
     NUMBER = 258
   };
#endif
/* Tokens.  */
#define NUMBER 258

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE
{
/* Line 2068 of yacc.c  */
#line 8 "calc.y"

int intval;

/* Line 2068 of yacc.c  */
#line 62 "y.tab.h"
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Note on Header (y.tab.h)

- `y.tab.h` is generated by Bison from `calc.y` to specify the token constants and attribute type.
- `y.tab.h` is automatically included in `y.tab.c` and must be included in `calc.l` so that it can feature in `lex.yy.c`.
- Symbolized tokens are enumerated beyond 256 to avoid clash with ASCII codes returned for single character tokens.
- `%union` has generated a `C union YYSTYPE` .
- Line directives are used for cross references to source files. These help debug messaging. For example:

  ```
  #line 8 "calc.y"
  ```
- `yylval` is a pre-defined global variable of `YYSTYPE` type.

  ```
  extern YYSTYPE yylval;
  ```

  This is used by `lex.yy.c`.

Flex Specs (calc.l) for Calculator Grammar

```
%{
#include "y.tab.h" // Bison generated file of token symbols and attributes
#include <math.h>
%}

%%
[1-9]+[0-9]*    {
                    yylval.intval = atoi(yytext); // yylval denotes the attribute
                                                  // of the current symbol
                    return NUMBER;
                }

[ \t]           ;  /* ignore white space */

"$"             {
                    return 0; /* end of input */
                }

\n|.            return yytext[0];
%%
```

Note on Flex Specs (calc.l)

- `y.tab.h` is automatically included in `y.tab.c` and must be included in `calc.l` so that it can feature in `lex.yy.c`.
- `yylval` is a pre-defined global variable of `YYSTYPE` type. So attributes of terminal symbols should be populated in it as appropriate. So for `NUMBER` we have:

  ```
  yylval.intval = atoi(yytext);
  ```

  Recall, in `calc.y`, we specified:

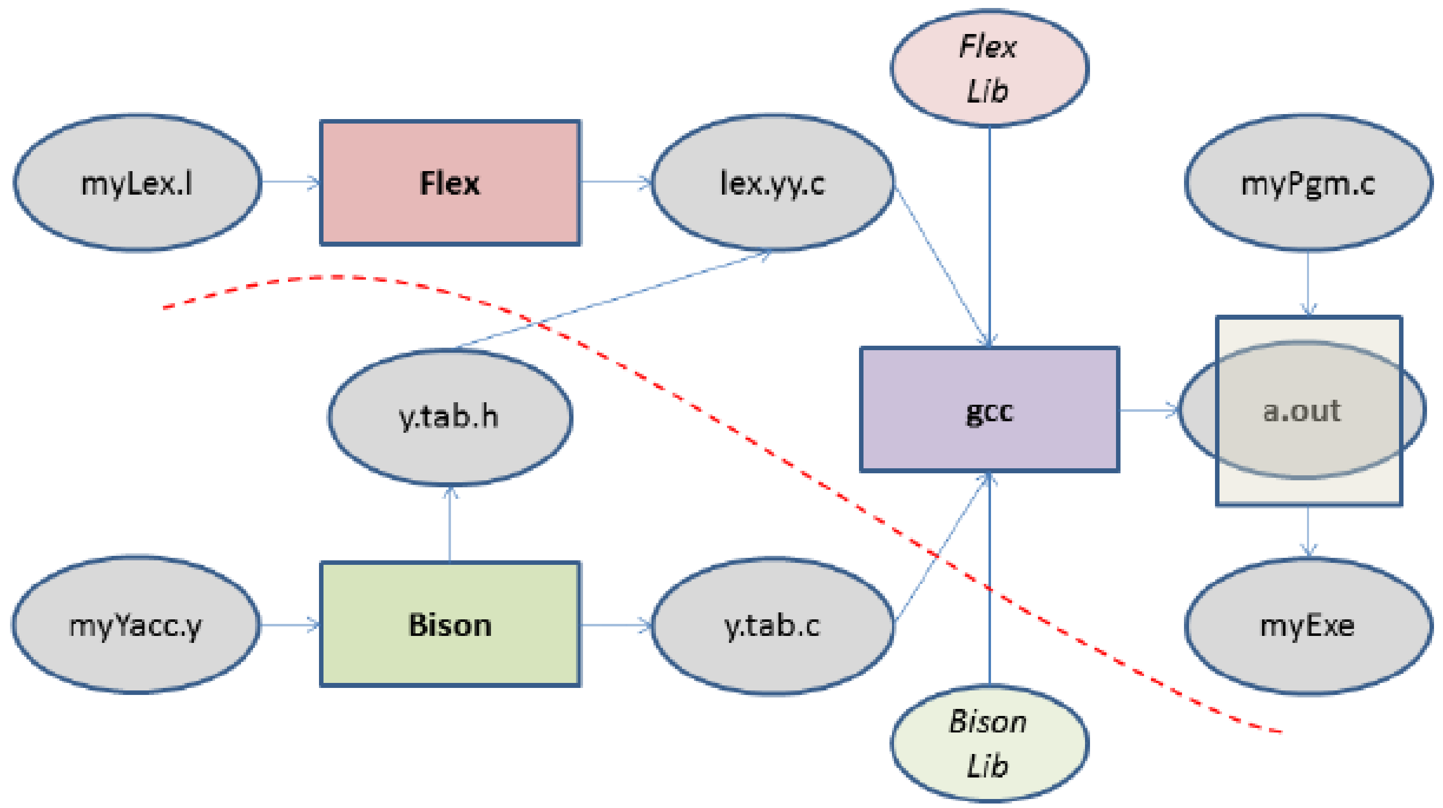
  ```
  %token <intval> NUMBER
  ```

  binding `intval` to `NUMBER`.
- Note how

  ```
  \n|.            return yytext[0];
  ```

  would return single character operators by their ASCII code.
- Newline is not treated as a white space but returned separately so that calc.y can generate error messages on line numbers if needed (not shown in the current example).

Flex-Bison Flow & Build Commands




```
$ flex calc.l
$ yacc -dtv calc.y
$ g++ -c lex.yy.c
$ g++ -c y.tab.c
$ g++ lex.yy.o y.tab.o -lfl
```

Sample Run

```
$ ./a.out
12+8 $
= 20
$ ./a.out
12+2*45/4-23*(7+1) $
= -150
```

Handling of 12+8 $

- In the next slide we show the working of the parser on the input:

  `12 + 8 $`
- We use a pair of stacks – one for the grammar symbols for parsing and the other for keeping the associated attributes.
- We show the snapshot on every reduction (skipping the shifts).

Handling of 12+8 $

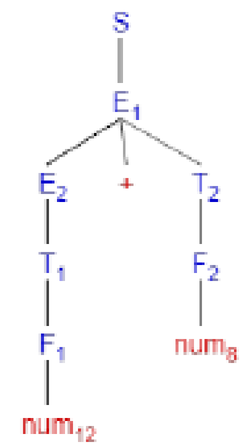
Grammar

1:	S	$\rightarrow$	E	{ printf("= %d\n", $1); }
2:	E	$\rightarrow$	$E + T$	{ $$ = $1 + $3; }
3:	E	$\rightarrow$	$E - T$	{ $$ = $1 – $3; }
4:	E	$\rightarrow$	T	{ $$ = $1; }
5:	T	$\rightarrow$	$T * F$	{ $$ = $1 * $3; }
6:	T	$\rightarrow$	T / F	{ $$ = $1 / $3; }
7:	T	$\rightarrow$	F	{ $$ = $1; }
8:	F	$\rightarrow$	(E)	{ $$ = $2; }
9:	F	$\rightarrow$	$-E$	{ $$ = –$2; }
10:	F	$\rightarrow$	num	{ $$ = $1; }

Reductions

$$
\begin{array}{ll}
 & \underline{\mathbf{num}_{12}} + \mathbf{num}_8\ \$ \\
\Rightarrow & \underline{F} + \mathbf{num}_8\ \$ \\
\Rightarrow & \underline{T} + \mathbf{num}_8\ \$ \\
\Rightarrow & E + \underline{\mathbf{num}_8}\ \$ \\
\Rightarrow & E + \underline{F}\ \$ \\
\Rightarrow & \underline{E + T}\ \$ \\
\Rightarrow & \underline{E}\ \$ \\
\Rightarrow & S\ \$
\end{array}
$$

Parse Tree



Stack

		‖			‖			‖			‖		
		‖			‖			‖			‖	num	8
		‖			‖			‖			‖	+	
num	12	‖	F	12	‖	T	12	‖	E	12	‖	E	12

Stack

		‖			‖			‖			‖
F	8	‖	T	8	‖			‖			‖
+		‖	+		‖			‖			‖
E	12	‖	E	12	‖	E	20	‖	S		‖

Output ‖ ‖ ‖ = 20 ‖

Programmable Calculator

A Programmable Calculator Grammar

1:	L	$\rightarrow$	$L\ S$ \n
2:	L	$\rightarrow$	S \n
3:	S	$\rightarrow$	$\mathbf{id} = E$
4:	S	$\rightarrow$	E
5:	E	$\rightarrow$	$E + T$
6:	E	$\rightarrow$	$E - T$
7:	E	$\rightarrow$	T
8:	T	$\rightarrow$	$T * F$
9:	T	$\rightarrow$	$T\ /\ F$
10:	T	$\rightarrow$	F
11:	F	$\rightarrow$	(E)
12:	F	$\rightarrow$	$-F$
13:	F	$\rightarrow$	**num**
14:	F	$\rightarrow$	**id**

- Rules 4 through 13 are same as before.
- $F \rightarrow$ **id** (Rule 14) supports storable computations (partial). This rule depicts the *use* of a stored value.
- $S \rightarrow$ **id** $= E$ (Rule 3) is added to store a partial computation to a variable. This rule depicts the *definition* of a stored value.
- $L \rightarrow L\ S$ \n (Rule 1) and $L \rightarrow S$ \n (Rule 2) allow for a list of statements, each on a separate source line – expressions ($S \rightarrow E$) or assignments ($S \rightarrow$ **id** $= E$) – to be concatenated. For example,
  a = 8 + 9
  a + 4
- The above exposes us to semantic issues. Like,
  a = 8 + 9
  b + 4
  is syntactically right, but semantically wrong (b is <u>undefined</u>).
- We now need a **Symbol Table** to record the variables defined. Note that there is no declaration for variables – a variable is declared the first time it is defined (assigned a value).

Bison Specs (calc.y) for Programmable Calculator Grammar

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"

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

#define NSYMS 20 /* max # of symbols */
symboltable symtab[NSYMS];
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%type <intval> expression
%type <intval> term
%type <intval> factor

%%
stmt_list: stmt_list statement '\n'
         | statement '\n'
         ;
statement: NAME '=' expression
               { $1->value = $3; }
         | expression
               { printf("= %d\n", $1); }
         ;
expression: expression '+' term
               { $$ = $1 + $3; }
          | expression '-' term
               { $$ = $1 - $3; }
          | term
          ;
term: term '*' factor
          { $$ = $1 * $3; }
    | term '/' factor
          { if ($3 == 0.0)
                yyerror("divide by zero");
            else
                $$ = $1 / $3;
          }
    | factor
    ;
factor: '(' expression ')'
            { $$ = $2; }
      | '-' factor
            { $$ = -$2; }
      | NUMBER
      | NAME
            { $$ = $1->value; }
      ;
%%
```

Bison Specs (calc.y) for Programmable Calculator Grammar

```
struct symtab *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */
```

```
void yyerror(char *s) {
    std::cout << s << std::endl;
}

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

Header (y.tab.h) for Programmable Calculator

```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259
   };
#endif
/* Tokens.  */
#define NAME 258
#define NUMBER 259

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 11 "calc.y" /* Line 2068 of yacc.c  */

    int intval;
    struct symtab *symp;

#line 65 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Header (parser.h) for Programmable Calculator

```
#ifndef __PARSER_H
#define __PARSER_H

typedef struct symtab {
    char *name;
    int value;
} symboltable;

symboltable *symlook(char *);

#endif // __PARSER_H
```

Flex Specs (calc.l) for Programmable Calculator Grammar

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID          [A-Za-z][A-Za-z0-9]*

%%
[0-9]+      {
              yylval.intval = atoi(yytext);
              return NUMBER;
            }

[ \t]       ;  /* ignore white space */


{ID}        { /* return symbol pointer */
              yylval.symp = symlook(yytext);
              return NAME;
            }

"$"         { return 0; /* end of input */ }

\n|.        return yytext[0];
%%
```

Note on Programmable Calculator

- **Symbol Table**
  - We have introduced variables (**id**) in the grammar now to support programmability (to store intermediate results).
  - **id**'s are maintained in the (rudimentary) symbol table as a name-value doublet (refer: `parser.h`).

    ```
    struct symtab { char *name; int value; };
    ```
  - Every **id**, as soon as found in the lexer for the first time, is inserted in the symbol table. On every subsequent occurrence the same **id** is referred from the symbol table. The function `struct symtab *symlook(char *);` achieves this.
- `union` **Wrapper**
  - Tokens `NAME` and `NUMBER` have different attributes `intval` and `symp` respectively.
  - For defining a value-stack in `C`, these are wrapped in a single union:

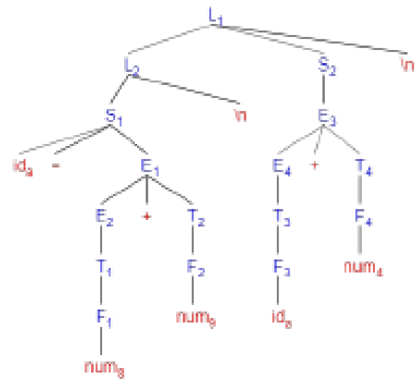
    ```
    typedef union YYSTYPE {
        int intval;
        struct symtab *symp;
    } YYSTYPE;
    ```

Sample Run

Output

```
$ ./a.out
a = 8 + 9
a + 4
= 21
$
```

Parse Tree



Grammar

1:	L	$\longrightarrow$	$L\ S$ \n
2:	L	$\longrightarrow$	S \n
3:	S	$\longrightarrow$	id $=$ E
4:	S	$\longrightarrow$	E
5:	E	$\longrightarrow$	$E + T$
6:	E	$\longrightarrow$	$E - T$
7:	E	$\longrightarrow$	T
8:	T	$\longrightarrow$	$T * F$
9:	T	$\longrightarrow$	$T\ /\ F$
10:	T	$\longrightarrow$	F
11:	F	$\longrightarrow$	(E)
12:	F	$\longrightarrow$	$-E$
13:	F	$\longrightarrow$	num
14:	F	$\longrightarrow$	id

Derivation

L \$	$\Rightarrow$	$\underline{L\ S\ \backslash n}$ \$
	$\Rightarrow$	$L\ \underline{E}\ \backslash n$ \$
	$\Rightarrow$	$L\ \underline{E + T}\ \backslash n$ \$
	$\Rightarrow$	$L\ E + \underline{F}\ \backslash n$ \$
	$\Rightarrow$	$L\ E + \underline{\text{num}_4}\ \backslash n$ \$
	$\Rightarrow$	$L\ \underline{T} + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$L\ \underline{F} + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$L\ \underline{\text{id}_a} + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\underline{S\ \backslash n}\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\underline{\text{id}_a = E}\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = \underline{E + T}\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = E + \underline{F}\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = E + \underline{\text{num}_9}\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = \underline{T} + \text{num}_9\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = \underline{F} + \text{num}_9\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$
	$\Rightarrow$	$\text{id}_a = \underline{\text{num}_8} + \text{num}_9\ \backslash n\ \text{id}_a + \text{num}_4\ \backslash n$ \$

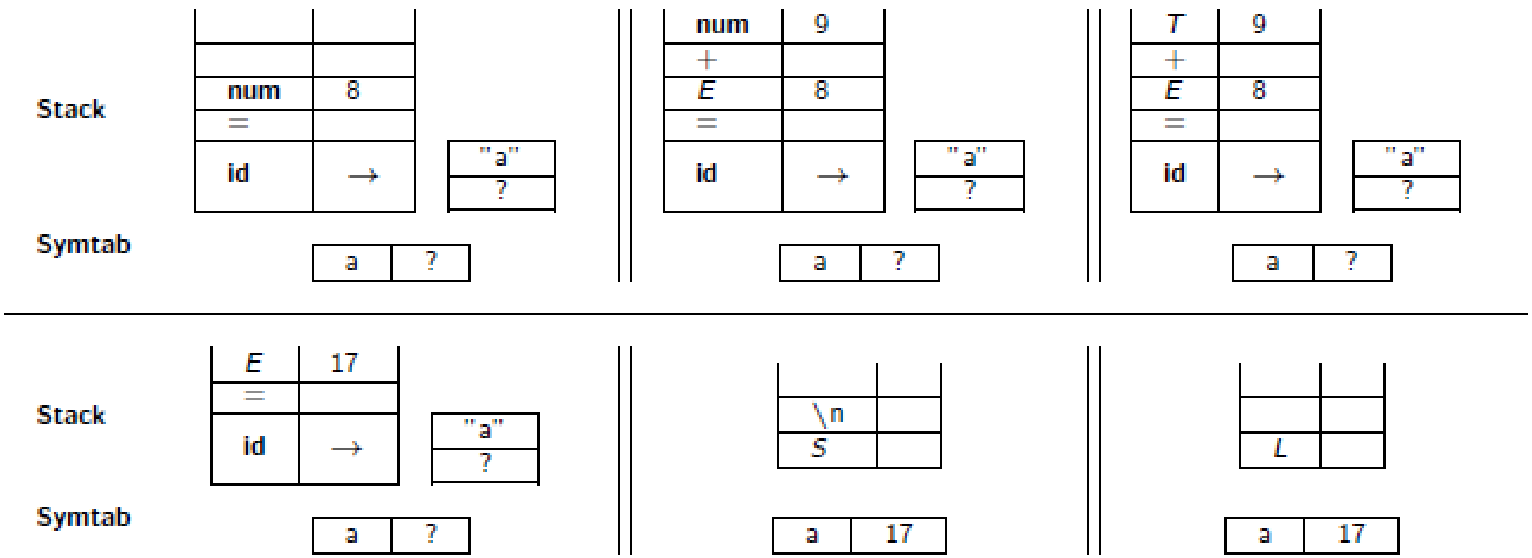
Handling of a = 8 + 9 \n a + 4 \n $

Grammar

L	$\rightarrow$	$L\ S$ \n	T	$\rightarrow$	$T * F$
L	$\rightarrow$	S \n	T	$\rightarrow$	$T\ /\ F$
S	$\rightarrow$	id $= E$	T	$\rightarrow$	F
S	$\rightarrow$	E	F	$\rightarrow$	(E)
E	$\rightarrow$	$E + T$	F	$\rightarrow$	$-\ E$
E	$\rightarrow$	$E - T$	F	$\rightarrow$	num
E	$\rightarrow$	T	F	$\rightarrow$	id

Reductions

$$\mathbf{id}_a = \underline{\mathbf{num}_8} + \mathbf{num}_9 \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \mathbf{id}_a = \underline{F} + \mathbf{num}_9 \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \mathbf{id}_a = \underline{T} + \mathbf{num}_9 \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \mathbf{id}_a = E + \underline{\mathbf{num}_9} \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \mathbf{id}_a = E + \underline{F} \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \mathbf{id}_a = \underline{E + T} \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \underline{\mathbf{id}_a = E} \ \backslash\mathrm{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \underline{S \ \backslash\mathrm{n}}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$
$$\Leftarrow \underline{L}\ \underline{\mathbf{id}_a} + \mathbf{num}_4 \ \backslash\mathrm{n}\ \$$$



Handling of a = 8 + 9 \n a + 4 \n $

Grammar

L	$\rightarrow$	$L\ S$ \n	T	$\rightarrow$	$T * F$
L	$\rightarrow$	S \n	T	$\rightarrow$	$T\ /\ F$
S	$\rightarrow$	id $= E$	T	$\rightarrow$	F
S	$\rightarrow$	E	F	$\rightarrow$	(E)
E	$\rightarrow$	$E + T$	F	$\rightarrow$	$-\ E$
E	$\rightarrow$	$E - T$	F	$\rightarrow$	num
E	$\rightarrow$	T	F	$\rightarrow$	id

Reductions

$\Leftarrow\ L\ \underline{\textbf{id}_a} + \textbf{num}_4$ \n **$**
$\Leftarrow\ L\ \underline{F} + \textbf{num}_4$ \n **$**
$\Leftarrow\ L\ \underline{T} + \textbf{num}_4$ \n **$**
$\Leftarrow\ L\ E + \underline{\textbf{num}_4}$ \n **$**
$\Leftarrow\ L\ E + \underline{F}$ \n **$**
$\Leftarrow\ L\ \underline{E + T}$ \n **$**
$\Leftarrow\ L\ \underline{E}$ \n **$**
$\Leftarrow\ \underline{L\ S\ \text{\\n}}$ **$**
$\Leftarrow\ L$ **$**

Stack

id	$\rightarrow$
L	

"a"
17

Symtab

a	17

Stack

num	4
+	
E	17
L	

Symtab

a	17

Stack

T	4
+	
E	17
L	

Symtab

a	17

Stack

E	21
L	

Symtab

a	17

Output

Stack

\n	
S	
L	

Symtab

a	17

Output = 21

Stack

L	

Symtab

a	17

Output

Ambiguous Grammars

LR Parser with Ambiguous Grammar

Unambiguous Grammar G_1

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

- Unique Parse Tree
- Associativity & Precedence Resolved
- Free of Conflict
- Larger Parse Tree
- Several Single Productions
- Non-intuitive
- Difficult for Semantic Actions

Ambiguous Grammar G_{1A}

1:	E	$\rightarrow$	$E + E$
2:	E	$\rightarrow$	$E * E$
3:	E	$\rightarrow$	(E)
4:	E	$\rightarrow$	id

- Multiple Parse Trees
- Associativity & Precedence Unresolved
- S/R Conflict
- Smaller Parse Tree
- No Single Productions
- Intuitive
- Easy for Semantic Actions

Ambiguous Grammar

Expression Parsing

Expression Grammar

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_3: $E \to \mathbf{id}\cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot(E)$
$E \to \cdot\mathbf{id}$

I_6: $E \to (E\cdot)$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_7: $E \to E + E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_8: $E \to E * E\cdot$
$E \to E \cdot + E$
$E \to E \cdot * E$

I_9: $E \to (E)\cdot$

Ambiguous Grammar G_{1A}

1:	E	$\to$	$E + E$
2:	E	$\to$	$E * E$
3:	E	$\to$	(E)
4:	E	$\to$	id

- In State#7 (State#8), do we have a conflict: shift on $+$ or $*$ / reduce by $E \to E + E$ (by $E \to E * E$)
- SLR(1) construction fails for both states as $\{+, *\} \subset FOLLOW(E)$. That is:

	$+$	$*$
State#7	s4/r1	s5/r1
State#8	s4/r2	s5/r2

- All other LR constructions too will fail
- To resolved, we use left associativity of $+$ & $*$, and higher precedence of $*$ over $+$ (recall operator precedence rules)

	$+$	$*$
State#7	r1	s5
State#8	r2	r2

- We get a more compact parse table

Source: Dragon Book

Expression Grammar

Unambiguous Grammar G_1

$$
\begin{array}{llll}
1: & E & \rightarrow & E + T \\
2: & E & \rightarrow & T \\
3: & T & \rightarrow & T * F \\
4: & T & \rightarrow & F \\
5: & F & \rightarrow & (E) \\
6: & F & \rightarrow & id
\end{array}
$$

Ambiguous Grammar G_{1A}

$$
\begin{array}{llll}
1: & E & \rightarrow & E + E \\
2: & E & \rightarrow & E * E \\
3: & E & \rightarrow & (E) \\
4: & E & \rightarrow & id
\end{array}
$$

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			

STATE	ACTION						GOTO
	id	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1	s5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Source: Dragon Book

Ambiguous Grammar

Programmable Calculator

A Programmable Calculator Grammar (with Ambiguous Grammar)

$$
\begin{array}{rlcl}
1: & L & \rightarrow & L\ S\ \backslash\mathrm{n} \\
2: & L & \rightarrow & S\ \backslash\mathrm{n} \\
3: & S & \rightarrow & \mathbf{id} = E \\
4: & S & \rightarrow & E \\
5: & E & \rightarrow & E + E \\
6: & E & \rightarrow & E - E \\
7: & E & \rightarrow & E * E \\
8: & E & \rightarrow & E\ /\ E \\
9: & E & \rightarrow & (E) \\
10: & E & \rightarrow & -E \\
11: & E & \rightarrow & \mathbf{num} \\
12: & E & \rightarrow & \mathbf{id}
\end{array}
$$

Bison Specs (calc.y) for Programmable Calculator Grammar

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"
extern int yylex();
void yyerror(char *s);
#define NSYMS 20 /* max # of symbols */
symboltable symtab[NSYMS];
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

%type <intval> expression
%%

stmt_list: statement '\n'
         | stmt_list statement '\n'
         ;
```

```
statement: NAME '=' expression
               { $1->value = $3; }
         | expression
               { printf("= %d\n", $1); }
         ;

expression: expression '+' expression
                { $$ = $1 + $3; }
          | expression '-' expression
                { $$ = $1 - $3; }
          | expression '*' expression
                { $$ = $1 * $3; }
          | expression '/' expression
                { if ($3 == 0)
                      yyerror("divide by zero");
                  else
                      $$ = $1 / $3;
                }
          | '(' expression ')'
                { $$ = $2; }
          | '-' expression %prec UMINUS
                { $$ = -$2; }
          | NUMBER
          | NAME
                { $$ = $1->value; }
          ;
%%
```

Bison Specs (calc.y) for Programmable Calculator Grammar

```
struct symtab *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */
```

```
void yyerror(char *s) {
    std::cout << s << std::endl;
}

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

Note on Bison Specs (calc.y) for Ambiguous Grammar

- Ambiguous Grammars
  - Ease specification of languages - particularly the operator expressions.
  - Offer shorter and more compact representation.
  - Lead to less reduction steps during parsing.
  - Introduce shift / reduce conflicts in the LR parser.
  - Conflict are resolved by precedences and associativities of operators.
- Associativity
  - `%left` is used to specify left-associative operators.
  - `%right` is used to specify right-associative operators.
  - `%nonassoc` is used to specify non-associative operators.
- Precedence
  - Precedence is specified by the order of `%left`, `%right`, or `%nonassoc` definitions. Later in the order, higher the precedence. However, all operators in the same definition have the same precedence.
  - All operators having the same precedence must have the same associativity.

Note on Bison Specs (calc.y) for Ambiguous Grammar

- Overloaded Operators
  - Operators like `'-'` are overloaded in unary and binary forms and have different precedences. We use a symbolic name `UMINUS` for (say) the unary operator while the binary one is marked as `'-'`.

    ```
    %left '-'
    %nonassoc UMINUS
    ```
  - The rule with the unary minus is bound to this symbolic name using `%prec` marker.

    ```
    expression: '-' expression %prec UMINUS
              | expression '-' expression
    ```
  - Note that the lexer (calc.l) would continue to return the same `'-'` token for unary as well as binary instances of the operators. However, Bison can use the precedence information to resolve between the two.

Header (y.tab.h) for Programmable Calculator

```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens. */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 11 "calc.y" /* Line 2068 of yacc.c  */

    int intval;
    struct symtab *symp;

#line 67 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif

extern YYSTYPE yylval;
```

Header (parser.h) for Programmable Calculator

```
#ifndef __PARSER_H
#define __PARSER_H

typedef struct symtab {
    char *name;
    int value;
} symboltable;

symboltable *symlook(char *);

#endif // __PARSER_H
```

Flex Specs (calc.l) for Programmable Calculator Grammar

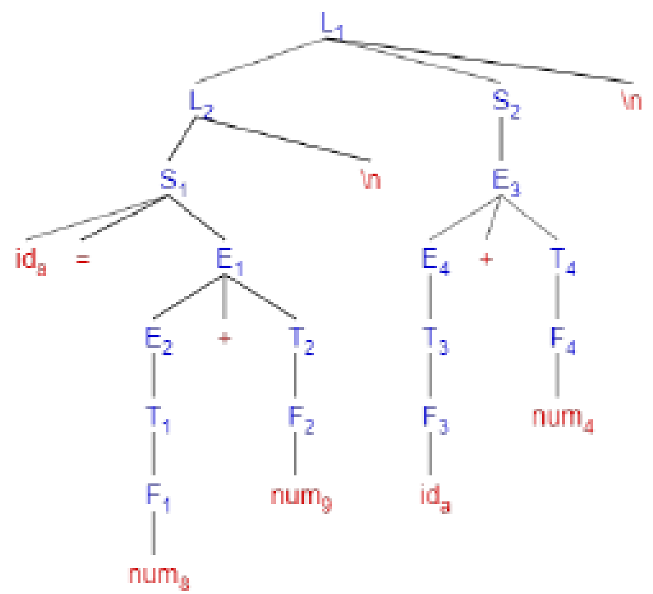
```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID          [A-Za-z][A-Za-z0-9]*

%%
[0-9]+      {
              yylval.intval = atoi(yytext);
              return NUMBER;
            }

[ \t]       ;  /* ignore white space */


{ID}        { /* return symbol pointer */
              yylval.symp = symlook(yytext);
              return NAME;
            }

"$"         { return 0; /* end of input */ }

\n|.        return yytext[0];
%%
```

Sample Run

Output

```
$ ./a.out
a = 8 + 9
a + 4
= 21
$
```

Parse Tree



Grammar

1:	L	$\rightarrow$	$L\ S\ \backslash n$
2:	L	$\rightarrow$	$S\ \backslash n$
3:	S	$\rightarrow$	$\textbf{id} = E$
4:	S	$\rightarrow$	E
5:	E	$\rightarrow$	$E + E$
6:	E	$\rightarrow$	$E - E$
7:	E	$\rightarrow$	$E * E$
8:	E	$\rightarrow$	$E\ /\ E$
9:	E	$\rightarrow$	(E)
10:	E	$\rightarrow$	$-E$
11:	E	$\rightarrow$	$\textbf{num}$
12:	E	$\rightarrow$	$\textbf{id}$

Derivation

$$
\begin{aligned}
L\ \$ &\Rightarrow \underline{L\ S\ \backslash n}\ \$ \\
&\Rightarrow \underline{L}\ \underline{E}\ \backslash n\ \$ \\
&\Rightarrow L\ \underline{E + E}\ \backslash n\ \$ \\
&\Rightarrow L\ E + \underline{E}\ \backslash n\ \$ \\
&\Rightarrow L\ E + \underline{\textbf{num}_4}\ \backslash n\ \$ \\
&\Rightarrow L\ \underline{\textbf{id}_a} + \textbf{num}_4\ \backslash n\ \$ \\
&\Rightarrow \underline{S\ \backslash n}\ \textbf{id}_a + \textbf{num}_4\ \backslash n\ \$ \\
&\Rightarrow \underline{\textbf{id}_a = E}\ \backslash n\ \textbf{id}_a + \textbf{num}_4\ \backslash n\ \$ \\
&\Rightarrow \textbf{id}_a = \underline{E + E}\ \backslash n\ \textbf{id}_a + \textbf{num}_4\ \backslash n\ \$ \\
&\Rightarrow \textbf{id}_a = E + \underline{\textbf{num}_9}\ \backslash n\ \textbf{id}_a + \textbf{num}_4\ \backslash n\ \$ \\
&\Rightarrow \textbf{id}_a = \underline{\textbf{num}_8} + \textbf{num}_9\ \backslash n\ \textbf{id}_a + \textbf{num}_4\ \backslash n\ \$
\end{aligned}
$$

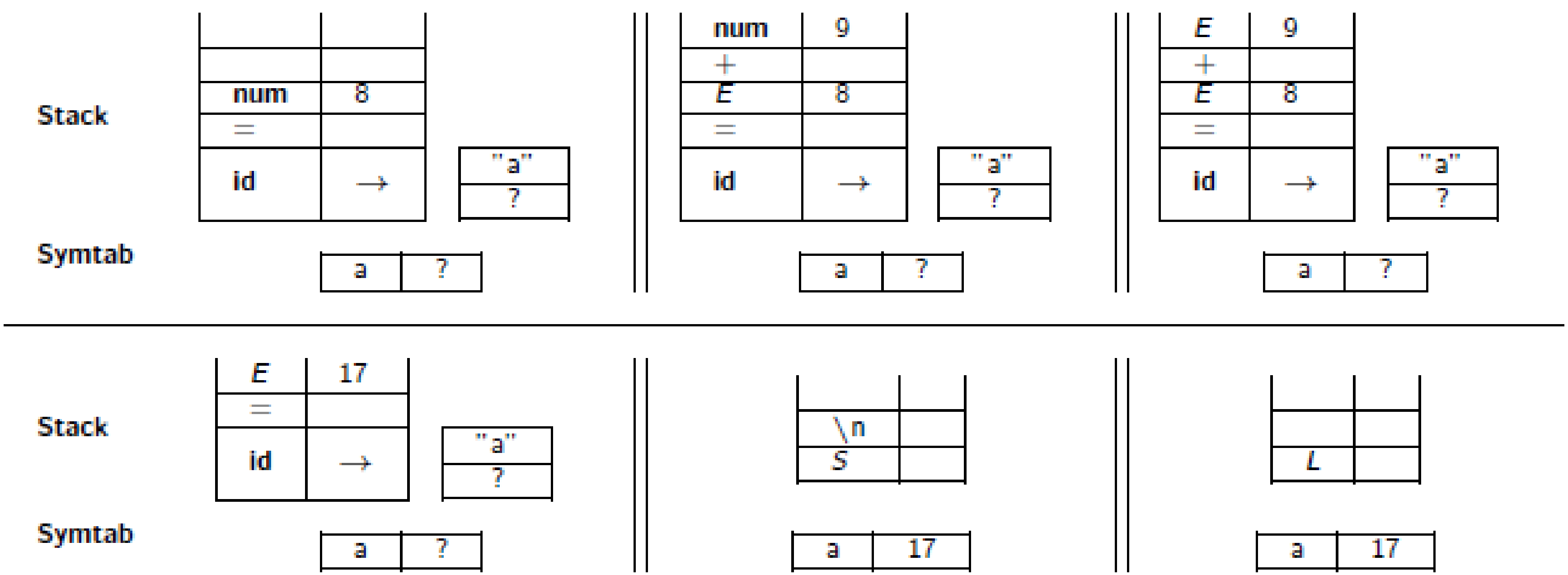
Handling of a = 8 + 9 \n a + 4 \n $

Grammar

L	$\rightarrow$	$L\ S$ \n	E	$\rightarrow$	$E * E$
L	$\rightarrow$	S \n	E	$\rightarrow$	$E\ /\ E$
S	$\rightarrow$	id $= E$	E	$\rightarrow$	(E)
S	$\rightarrow$	E	E	$\rightarrow$	$-E$
E	$\rightarrow$	$E + E$	E	$\rightarrow$	num
E	$\rightarrow$	$E - E$	E	$\rightarrow$	id

Reductions

$$\mathrm{id}_a = \underline{\mathrm{num}_8} + \mathrm{num}_9 \ \backslash n \ \mathrm{id}_a + \mathrm{num}_4 \ \backslash n \ \$$$
$$\Rightarrow \mathrm{id}_a = E + \underline{\mathrm{num}_9} \ \backslash n \ \mathrm{id}_a + \mathrm{num}_4 \ \backslash n \ \$$$
$$\Rightarrow \mathrm{id}_a = \underline{E + E} \ \backslash n \ \mathrm{id}_a + \mathrm{num}_4 \ \backslash n \ \$$$
$$\Rightarrow \underline{\mathrm{id}_a = E} \ \backslash n \ \mathrm{id}_a + \mathrm{num}_4 \ \backslash n \ \$$$
$$\Rightarrow \underline{S \ \backslash n} \ \mathrm{id}_a + \mathrm{num}_4 \ \backslash n \ \$$$
$$\Rightarrow L \ \underline{\mathrm{id}_a} + \mathrm{num}_4 \ \backslash n \ \$$$



Handling of a = 8 + 9 \n a + 4 \n $

Grammar

L	$\rightarrow$	$L\ S$ \n	E	$\rightarrow$	$E * E$
L	$\rightarrow$	S \n	E	$\rightarrow$	$E\ /\ E$
S	$\rightarrow$	id $= E$	E	$\rightarrow$	(E)
S	$\rightarrow$	E	E	$\rightarrow$	$-\ E$
E	$\rightarrow$	$E + E$	E	$\rightarrow$	num
E	$\rightarrow$	$E - E$	E	$\rightarrow$	id

Reductions

$\Rightarrow\ L\ \underline{\mathrm{id}_a} + \mathrm{num}_4$ \n $
$\Rightarrow\ L\ E + \underline{\mathrm{num}_4}$ \n $
$\Rightarrow\ L\ E + \underline{E}$ \n $
$\Rightarrow\ L\ \underline{E + E}$ \n $
$\Rightarrow\ L\ \underline{E}$ \n $
$\Rightarrow\ \underline{L\ S\ \backslash n}$ $
$\Rightarrow\ L$ $

Stack

id	$\rightarrow$
L	

"a"
17

num	4
+	
E	17
L	

E	4
+	
E	17
L	

Symtab

a	17

a	17

a	17

Stack

E	21
L	

\n	
S	
L	

L	

Symtab

a	17

a	17

a	17

Output

= 21

Ambiguous Grammar

Dangling Else Parsing

Dangling Else Ambiguity

Consider:

stmt $\rightarrow$ **if** *expr* **then** *stmt* **else** *stmt* | **if** *expr* **then** *stmt* | **other**

Using **i** for **if** *expr* **then**, **e** for **else**, and **a** for **other**, we get:

$G_{12} = S \rightarrow \mathbf{i}\ S\ \mathbf{e}\ S \mid \mathbf{i}\ S \mid \mathbf{a}$

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1: $S' \rightarrow S \cdot$

I_2: $S \rightarrow i \cdot SeS$
$S \rightarrow i \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3: $S \rightarrow a \cdot$

I_4: $S \rightarrow iS \cdot eS$

I_5: $S \rightarrow iSe \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6: $S \rightarrow iSeS \cdot$

STATE	ACTION				GOTO
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

$FOLLOW(S) = \{e, \$\}$. Hence in State#4, we have shift/reduce conflict on e between $S \rightarrow iS.eS$ and $S \rightarrow iS.$ items. We choose shift binding else with the nearest earlier **then**.

Source: Dragon Book