

Parsing -- Overview

The second phase of front-end analysis is **parsing**. It is the parser which is the driving force of the front-end, requesting and consuming tokens from the lexer, and performing semantic analysis and intermediate code generation via embedded actions, which are triggered as parts of the input are recognized in the framework of the defined language.

While lexical rules are one-dimensional, the rules of syntax analysis form a two-dimensional tree of syntax relationships. The rules which the parser follows in building this tree form a **grammar**. By way of example, here is a simple grammar for arithmetic expressions:

```
/* Grammar 1 */
expr:     IDENT
          |NUM
          | '(' expr ')'
          | expr '+' expr
          | expr '-' expr
          | expr '*' expr
          | expr '/' expr
          ;
```

Grammars are traditionally written in **Backus Naur Form (BNF)**, and many textbooks use arrows, greek letters and other symbols which aren't readily found on the typical keyboard. In these notes, we will adhere to the format used by the parser generator YACC (Yet Another Compiler Compiler) and the open-source, compatible program Bison.

In the example above, `expr` is an example of a **non-terminal** symbol. The name of a non-terminal, at the beginning of the line and followed by a :, indicates the start of rules listing all of the possible ways of forming that non-terminal. The choices (if more than one) are delimited by the | character and terminated by the semicolon. It is also permitted to have multiple rules for the same non-terminal and this is equivalent to having them joined with the | character. Stylistically, we line up the vertical bars and the semicolon as shown above. Furthermore, one (and only one) of the choices can be nothing, i.e. ε the empty string. This is often used for parts of the language which are optional.

It is conventional to use all UPPERCASE for **terminal** symbols (a terminal is the same thing as a token). Because Bison/YACC uses integers to represent terminals internally, and because it starts assigning symbolic terminal names at 257, we can also use single-quoted characters directly as terminals, assuming of course that the lexer follows the same convention.

Each possible expansion of a non-terminal is called a **production**. For our purposes, the terms rule and production are equivalent. We can see that a production must have one non-terminal as the left-hand side, and can have a right-hand side that contains zero or more terminals OR non-terminals.

Grammars vs Regular Expressions

There is a certain similarity between grammar rules and the lexical rules, expressed as Regular Expressions, which we saw in Unit 1. The difference is that the productions of a non-terminal may contain other non-terminals, including direct (same non-terminal is on both sides of the rule) or indirect (right-hand-side symbol is ultimately defined in terms of left-hand-side) recursion. This is what gives the grammar a 2-dimensional feel, as opposed to the linear world of regular expressions.

Obviously, since the entire input stream of the program must pass through lexical analysis first, the regular expressions which comprise the lexical rules can describe any valid input stream. However, they will also *accept* as valid input streams which are not syntatically valid. A classic example is being able to count and balance pairs of nested parentheses. There is no regular expression which can check this and reject unbalanced streams.

Therefore, we say that grammars are more expressive, or more powerful, than regular expressions. One might then ask why it is prudent to bother with RE at all? Why not simply describe the lexical rules (which, after all, are often called the **micro-syntax** of a language) directly in the grammar, for in fact, the C language standard does this? The answer is that doing so in an actual production compiler would make the parser unbearably large, awkward and inefficient. As we shall see, the more powerful the method of expression, the less efficient and more unwieldy is the recognizer for it. We are much better off using a more limited method (regular expressions) to gain maximum performance for that part of the compiler which must react to each and every input character.

Leftmost and Rightmost derivations

Every grammar has a particular **start symbol** which represents the entire language. In the example above, the start symbol is `expr`. We can **derive** any valid sequence of symbols (or "sentence") from the start symbol by progressively replacing any non-terminals by their appropriate productions until only the terminals remain. If the input sequence is `a+(b-c)`, one possible sequence of derivation is:

```
expr
expr + expr
IDENT + expr
IDENT + ( expr )
IDENT + ( expr - expr )
IDENT + ( IDENT - expr )
IDENT + ( IDENT - IDENT )
```

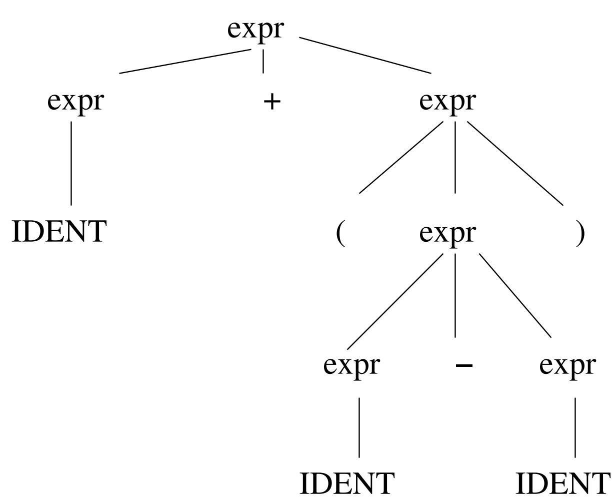
In this derivation, the leftmost non-terminal was expanded at each step. This is known as the **leftmost derivation**. We could have also expanded the rightmost non-terminal:

```
expr
expr + expr
```

```
expr + ( expr )
expr + ( expr - expr)
expr + ( expr - IDENT)
expr + ( IDENT - IDENT)
IDENT + (IDENT - IDENT)
```

We say that each of the above steps is a **sentential form** of the input sentence. Definition: a sequence of symbols (terminals or non-terminals) is a sentential form if it can be generated by some expansion of the start symbol, and if by further expansion it will yield a valid sentence of terminals.

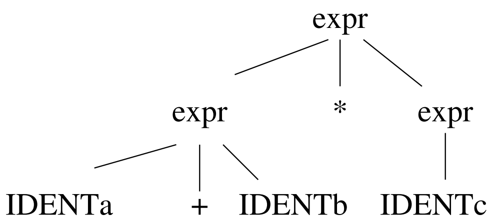
A tree which represents a partial or complete derivation is known as a **Parse Tree**. Each node of the parse tree represents a symbol, and the edges leading down from an interior node represent the right-hand-side symbols of a rule deriving that node. The reader will observe that in reading the tree from left to right across the bottom frontier, a sentential form is always seen. The final parse tree for the example above is:

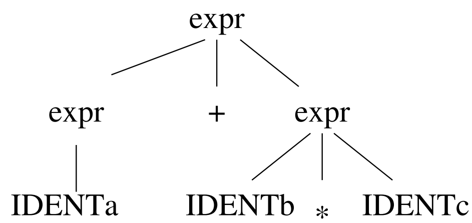


A grammar which is **unambiguous** has the property that regardless of how the derivation was performed (leftmost, rightmost, or any other way) it yields the same parse tree, for any given *valid* sequence of input tokens. Furthermore, there is exactly one leftmost and one rightmost derivation possible. Ambiguous grammars do not have these properties.

Ambiguity

A grammar is ambiguous if there exists a sentence of terminals which could have two or more different final parse trees. In our example grammar, what is the derivation of the sentence `a+b*c`? There are two possible answers



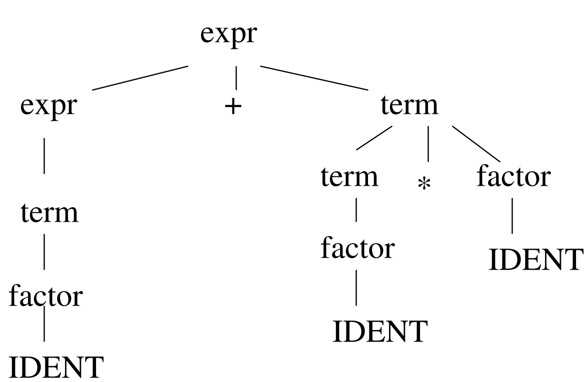


Therefore, although this grammar is sufficient for determining valid vs invalid sentences, it is not useful as part of a compiler because it does not capture the meaning of the sentence.

One way to remove the ambiguity is to define operator precedence and associativity. This method is often used with yacc/bison parser generators, and we shall see it shortly. However, this mechanism is outside of the BNF notation, so let us explore re-writing the grammar to be unambiguous:

```
/*Grammar 2 */
expr:   expr '+' term
     |  expr '-' term
     |  term
     ;

term:   term '*' factor
     |  term '/' factor
     |  factor
     ;

factor: '(' expr ')'
     |  NUM
     |  IDENT
     ;
```

We have redefined `expr` so that the rightmost non-terminal can not be something which ultimately contains a + or -. This removes the ambiguity both about precedence and associativity. Now a+b*c and a-b-c are unambiguous. Note in the parse tree below the introduction of extra levels because of the new non-terminals introduced to resolve the ambiguity:



Context-free grammars

In order to write an efficient parser, the grammar rules must be context-free, i.e. we must know them in advance without looking at the input. The grammar rules be defined only in terms of sequences of tokens, not their semantic values. A language which has dynamic syntax can not be expressed with a context-free grammar. Imagine a Perl-like extension to C which supports iterating over arrays using the `for` keyword:

```
int func()
{
        int ary[10];

                for(ary)
                {
                        /*...*/
                }
}
```

It can not be determined using syntactic analysis alone whether the sequence of tokens `FOR, '(', IDENT, ')'` is valid. In order to parse this context-dependent language, we must write a context-free grammar which is more permissive, allowing incorrect sentences to be accepted, and then catch the error through semantic analysis (type checking).

Another classic example of context dependence is the inability of a context free grammar to accept expressions in which the identifiers must have previously been declared. The text (section 4.3.5) contains a discussion of the impossibility of using context-free grammar to impose this restriction.

For the most part, the C language can be expressed with a context-free grammar. We'll see how to handle the few exceptions by feeding back information to modify the lexer's behavior dynamically.

LL / Top-Down Recursive parsing

For complex projects, parsers are generally created using a tool such as yacc. However, it is very useful to have an intuitive feel for how to create parsers by hand using recursion. For example, one might need to construct a small parser in a language which doesn't work with yacc/bison, or in an embedded environment where memory is very scarce.

In implementing **recursive descent parsing**, we will write a program structure which mirrors the grammar. Let us say our grammar is just:

```
/* Grammar 3*/
expr:   '(' expr ')'
  |     NUM
  |     IDENT
  ;
```

Our parser could be:

```
int curtok;

#define NEXT (curtok=yylex())
expr()
{
        if (curtok=='(')
        {
                NEXT;
                if (!expr()) return 0;
                if (curtok!=')') return 0;
                NEXT;
                return 1;
        }
        if (curtok == IDENT || curtok == NUM) {NEXT;return 1;}
        return 0;
}
```

We return 1 if the input is valid according to the grammar, and 0 otherwise. The recursion must eventually terminate. However, try applying this method to the unambiguous expression grammar #2 above. We will quickly get into an endless recursion because of the way the rules are written. They are **left recursive**. To correct this, we must re-write the grammar.

```
/* Grammar 4 */
expr:   term
    |   term '+' expr
    |   term '-' expr
    ;

term:   factor
    |   factor '*' term
    |   factor '/' term
    ;

factor: '(' expr ')'
     |  NUM
     |  IDENT
     ;
```

A grammar which has no left recursion and which can be parsed by examining the input stream from left to right and taking the leftmost derivation, looking at most one token ahead is called LL(1). From this grammar, we can write a top-down parser:

```
#define NEXT (curtok=yylex())
int curtok;
main()
{
        NEXT;
        if (expr())
                printf("GOOD\n")
        else
                printf("BAD\n");
```

```
}

expr()
{
        if (!term()) return 0;
        {
        if (curtok=='+')
        {
                NEXT;
                if (!expr()) return 0;
                return 1;
        }
        if (curtok=='-')
        {
                NEXT;
                if (!expr()) return 0;
                return 1;
        }
        if (curtok==EOD) return 1;
        return 0;
}

term()
{
        if (!factor()) return 0;
        if (curtok == '*')
        {
                NEXT;
                if (!term()) return 0;
                return 1;
        }
        if (curtok == '/')
        {
                NEXT;
                if (!term()) return 0;
                return 1;
        }
        return 1;
}

factor()
{
        if (curtok=='(')
        {
                NEXT;
                if (!expr(v)) return 0;
                if (curtok!=')') return 0;
                NEXT;
                return 1;
        }
        if (curtok==NUM || curtok==IDENT)
        {
                NEXT;
```

```
                return 1;
        }
        return 0;
}
```

We see that the structure of the above recursive parser is that on entry to any of the functions, we are "looking for" an expansion of a non-terminal symbol with the corresponding name, and `curtok` contains the current token. If the grammar rule contains a non-terminal, we recurse using that name. If the rule contains a terminal, we check to see if it matches. If it does not match, we flag an error. If we do have a match, then we must consume the token with `curtok=yylex()`. Thus we are using a look-ahead of one token. Upon return from any function, `curtok` is expected to contain the *next* token (i.e. not any of the tokens associated with the derivation of that function which just returned).

One problem which comes to light is that because we started with right-recursive grammar #4, our parser has right associativity. Consider an expression such as a-b-c. We will build a parse tree as if a-(b-c) which is incorrect. It didn't matter in the example above, because we are neither compiling nor interpreting the code, just verifying it for syntax.

Now we'll see a technique for rectifying this associativity problem. Let's do this parser again, this time adding support for calculating the value of the expression:

```
#include <stdio.h>
#include "tokval.h"

int curtok,yylval;

main()
{
 int val,rc;
        NEXT;
        rc=expr(&val);
        printf("%d\n",val);
        if (rc) printf("GOOD\n"); else printf("BAD\n");
}


expr(int *v)
{
 int v1,v2,op;
        if (!term(&v1)) return 0;
        while (curtok=='+' || curtok=='-')
        {
                op=curtok;
                NEXT;
                if (!term(&v2)) return 0;
                if (op=='+') v1+=v2; else v1-=v2;
        }
        *v=v1;
```

```
        return curtok==EOF;
}

term(int *v)
{
 int v1,v2,op;
        if (!factor(&v1)) return 0;
        while (curtok=='*' || curtok=='/')
        {
                op=curtok;
                NEXT;
                if (!factor(&v2)) return 0;
                if (op=='*')
                        v1*=v2;
                else
                {
                        if (v2==0)
                        {
                                fprintf(stderr,"/0 error0);
                                return 0;
                        }
                        v1/=v2;
                }
        }
        *v=v1;
        return 1;
}

factor(int *v)
{
        if (curtok=='(')
        {
                NEXT;
                if (!expr(v)) return 0;
                if (curtok!=')') return 0;
                NEXT;
                return 1;
        }
        /* Let's drop support for IDENT for now because we haven't
                developed a symbol table yet */
        if (curtok==NUM)
        {
                *v=yylval;
                NEXT;
                return 1;
        }
        if (curtok==EOF)
        {
                fprintf(stderr,"found EOF in factor");
                return 1;
        }
        fprintf(stderr,"found bad token %d in factor",curtok);
        return 0;
```

```
}
```

This is still a top-down recursive descent parser, but we have replaced tail recursion with a while loop. Doing so makes it easy to express a syntax like:

```
expr: term (('+'|'-') term)*
```

Of course, this regexp-like way of expressing syntax does not conform to the rules which we have already stated, i.e. it is not a form of BNF grammar. Writing a list this way is more intuitive and this notation is sometimes used informally. However, lists in formal grammars are written using either left or right recursion. The text demonstrates that this intuitive notation is no more powerful than BNF, i.e. it does not describe a richer set of languages, it merely describes them in a different form.

LR / Bottom-Up Parsing

Top-down, or LL parsing, is weak because we must make derivation decisions based only on the lookahead token. If there are multiple rules for a given symbol which have different non-terminals on the right side, they must all have distinct tokens which flag them. Here is a contrived example of a grammar which is not LL:

```
start: a
      |b
  ;

a: A a B
  | /* empty */
  ;

b: A b B B
 | /* empty */
 ;
```

Note the stylistic use of / empty */ Yacc/Bison use traditional C-style comment syntax, so this is ignored. But it reminds us that we meant to leave that rule blank, as opposed to an extra vertical bar that we forgot about.*

This grammar produces sequences such as AB, AABB, AAABBB which are rule a or ABB, AABBB, AAABBBBBB which are b. Since the rules are recursive, the valid sentences are potentially infinite. Given the input token sequence, for example, of A B B, we can not determine if the input reduces to the non-terminal a or b, with just one token of look-ahead. Having seen A and with B as the lookahead token, and not knowing that another B is up next, we might start down the path of a->A empty B. This language can not be parsed using top-down LL parsing without an infinite amount of lookahead. It can be parsed however in LR(1)

That being said, many languages, especially older languages, were written with a grammar than can be parsed in LL. LR parsing, we will see, gives us more flexibility.

Whereas top-down parsing starts with the highest-level node and attempts to work down to the terminals, bottom-up parsing consumes terminals and, as it recognizes sequences of terminals which could be the right-hand-side of a production, replaces them with non-terminals. This process continues, replacing sequences of symbols (both terminals and non-terminals) with higher-level non-terminals, until the input is consumed and the highest-level terminal has been recognized. Therefore we can look at more than one token at a time when deciding which rule to apply, which makes LR parsing more powerful than LL.

LR parsing uses a symbol stack as its primary data structure. We can informally describe the bottom-up "shift-reduce" parsing method as:

```
while (there are still input tokens)
        shift a token onto a stack of symbols
        while (the top N symbols represent "best" rhs of a production)
                replace the top N symbols with the lhs symbol

stack must now contain only one symbol, the start or "goal" symbol,
  otherwise report an error
```

Determining whether the stack has the "best" rule to match, or whether more tokens should be shifted, is the "hard part". An error occurs if the result of this process, when all input has been consumed, is not the goal symbol which represents the entirety of the language (`expr` in our example).

```
/* Grammar 1, again */
expr:   IDENT                   /* RULE 1 */
        |NUM                    /* RULE 2 */
        | '(' expr ')'                  /* RULE 3 */
        | expr '+' expr                 /* RULE 4 */
        | expr '-' expr                 /* RULE 5 */
        | expr '*' expr                 /* RULE 6 */
        | expr '/' expr                 /* RULE 7 */
        ;
```

Consider the input sequence a+(b−c) where, as before, a, b, and c are all IDENT. The sequence of moves made during bottom-up parsing is:

```
MOVE                          Symbol stack after move     Tokens Ahead
----------                    ------------------------------------
SHIFT                         IDENT                       + ( IDENT - IDENT )
REDUCE(1)                     expr                        + ( IDENT - IDENT )
SHIFT                         expr +                          ( IDENT - IDENT )
SHIFT                         expr + (                    IDENT - IDENT )
SHIFT                         expr + ( IDENT                 - IDENT )
REDUCE(1)                     expr + ( expr               - IDENT )
SHIFT                         expr + ( expr -             IDENT )
SHIFT                         expr + ( expr - IDENT       )
REDUCE(1)                     expr + ( expr - expr        )
REDUCE(5)                     expr + ( expr               )
SHIFT                         expr + ( expr )
REDUCE(3)                     expr + expr
REDUCE(4)                     expr
```

Note that if one reads the above sequence of operations backwards, it forms a right derivation. Bottom-up parsing is called LR parsing because it consumes the input from left-to-right, and builds the derivation from right-to-left (whereas LL parsing builds left-to-right).

Grammar Power

We say that a grammar is LR(1) if it can be processed by LR parsing, using at most one token of look-ahead. An LR(2) grammar requires two tokens of look-ahead. Here is an example of LR(2) lifted from the O'Reilly book:

```
stmt: COMMAND opt_keyword '(' IDENT ')'                  /* Rule 1*/
         ;

opt_keyword:       /*empty*/                    /* Rule 2*/
         | '(' KEYWORD ')'                      /* Rule 3*/
        ;
```

This grammar produces COMMAND (IDENT) or COMMAND (KEYWORD) (IDENT). Having shifted COMMAND, with the single look-ahead token '(', it is ambiguous whether we have just seen an empty opt_keyword and should reduce by Rule 2, or should shift the '(' in anticipation of Rule 3. Yacc or Bison would call this a shift/reduce conflict. If we could look ahead one more token, and determine if what follows '(' is IDENT or KEYWORD, then we could decide this positively. Therefore the grammar is LR(2).

It can be shown that any LR(n) grammar, where n>1 but bounded, can be re-written to be LR(1). However, the resulting grammar may be more cumbersome, or may not be able to capture the meaning of the syntax as effectively. We can re-write the above to:

```
stmt: COMMAND '(' KEYWORD ')' '(' IDENT )''
```

```
  |    COMMAND '(' IDENT ')'
  ;
```

The above technique known a **In-lining** the rules can also be useful in resolving some mysterious grammar conflict problems, as discussed later in this unit.

LR grammars are more powerful than LL grammars, in that they can describe a larger set of languages. All LL grammars are LR, but not all LR grammars are LL. LR(n), n>1 (n is bounded) grammars are more expressive than LR(1), but they do not describe more languages, they merely describe them more intuitively.

Yacc and Bison generate parsers which can only handle LR(1) grammars, and in fact a subset of LR(1) known as LALR(1), discussed directly below. Bison has support for generating a parser which can use an LR(n) grammar, where n is unbounded, also known as Generalized LR (GLR) parsing. GLR grammars are the most expressive context-free grammars. Any context-free grammar can be parsed by GLR. However, there is a price to pay. At points in the grammar where LR(1) would produce a conflict, Bison will "split" the parser into several parallel parsers, each taking one of the forks in the road. The token stream is also cloned and fed to each parallel parser as demanded. During this split, the semantic actions (see later in this unit) are not triggered but they are remembered. Eventually a point is reached where a tie-breaking rule can be applied and one of the parallel parsers is then chosen as the correct one. At this time, the semantic actions that the chosen parser would have taken are performed. This approach breaks the coupling between the lexer and semantic actions, and may require push-back of one or more tokens if the parallel parsers result in rules that evaluate to different numbers of terminals.

Some languages have such poorly designed syntax and complicated semantic rules (such as C++) that using GLR parsing is almost inevitable. But many languages, including C, can be parsed using LALR. This simpler approach, and fixing up a few corner cases through a second pass in semantic analysis, is usually easier. Therefore we will not be covering GLR parsing in this course, but the student is invited to explore that chapter in the Bison documentation.

LALR(1) parser operation

Most real-world programming languages can be adequately parsed using a parsing technique known as LALR(1). The "Dragon Book" text contains a full explanation of how the LALR(1) parser state machine may be generated automatically from the set of rules.

A grammar which is said to be LALR(1) is one which can be parsed by this method, the essentials of which are:
• The LALR(1) parser is a DFA state machine.
• Each of the states represents one or more LR(1) items

• An LR(1) item consists of three things: (1) a grammar rule, (2) a "cursor" which represents how much of that rule we may have already seen, (3) one or more lookahead tokens to match against. I.e. a state represents what we've seen so far and what rules it might possibly match.
• All symbols, both terminals and non-terminals, are represented by integer symbol numbers.
• All states are represented by integer state numbers.
• All rules are represented by integer rule numbers. When a non-terminal has several possible rules, each is given a distinct rule number.
• The parser maintains a stack of state numbers, with the top of the stack representing the current state number. This stack is initialized by pushing state #0.
• There is a lock-step correspondence between states and symbols. The state stack therefore can also be viewed as a symbol stack. However, there is no explicit symbol stack in the LALR parser.
• A pair of 2-dimensional tables are created by the parser generator (such as YACC or bison) and these control the state machine. The ACTION table describes the action to take after consuming a token from the lexer. It is indexed by token number and state number. The GOTO table determines what state to enter when a prior state is returned to during a reduction action. It is indexed by the prior state number and the rule number. Equivalently, since each rule is associated with just one left-hand-side non-terminal, we can think of the GOTO table as being indexed by state number and symbol.

Bison generates a function called `yyparse` from an input file which has a `.y` extension. The yyparse function is called once and consumes the entire input (internally calling yylex to obtain tokens from the input stream) and returns 1 if the input is accepted or 0 if it contains a syntax error.

The LALR algorithm as implemented by Bison/Yacc follows, in pseudo-code:

```
yyparse()
{
  initialize:
        push state 0;
        current_state=0;
        token=yylex();
        for(;;)
        {
                action=ACTION_TABLE[current_state][token];
                switch(action)
                {
                 case ERROR:
                        yyerror(descriptive error message);
                        return 0;
                 case ACCEPT:
                        return 1;
                 case SHIFT(new_state):
                        push new_state;
                        current_state=new_state;
                        token=yylex();
                        /**** SEMANTIC VALUE HANDLING : ***/
                                push yylval on the semantic value stack
                        break;
                 case REDUCE(rule_num):
                        nrhs=LENGTH[rule_num];  //how many symbols RHS of rule
                        pop nrhs states from the stack
                        exposed_state=new top of stack
                        new_state=GOTO[exposed_state][rule]
                        push new_state
                        current_state=new_state
                        /**** SEMANTIC VALUE HANDLING : ***/
                          execute the embedded semantic action C code
                            associated with this rule.
                          pop nrhs values from the value stack and replace
                            with the new $$ value computed in the embedded code.
                }
        }
}
```

• The ERROR action means that there is no may to make further progress towards the goal symbol. The default action is to report the error and halt (return from yyparse). In a later unit we'll see how to modify this action to allow error recovery.
• The ACCEPT action occurs when the parser is in state 0 and the look-ahead token is `$end`, the end of input.

Canonical LR(1) vs LALR(1)

The LALR(1) method is a refinement of what the text calls "Canonical LR(1) parsing." The latter produces a large number of states and very large ACTION and GOTO tables.

LALR combines states which share certain properties. We'll see later that in some rare cases, this causes a problem and a grammar can be LR(1) but fail to be LALR(1).

Running Bison

Let us use the following simple grammar and see what bison does with it, by using the −v option and examining the .output file:

```
/* Grammar 5 */
%token NUM
%%
expr:   '(' expr ')'
        | NUM
        | NUM '+' NUM
        ;
%%
```

Bison internalizes the rules, giving them a number and adding a rule 0 which represents the entire language:

```
    0 $accept: expr $end

    1 expr: '(' expr ')'
    2     | NUM
    3     | NUM '+' NUM
```

We will use bison's notation of a . character to represent the cursor which tracks how much of a given rule we have already consumed. In the initial state 0, we have not seen any input. Therefore:

```
state 0

    0 $accept: . expr $end

    NUM  shift, and go to state 1
    '('  shift, and go to state 2

    expr  go to state 3
```

The line after "state 0" tells us the place (or places) we could possibly be in. Here, there is only one possible place, before the beginning of the top-level symbol. This place is associated with rule #0. Had the `--report=all` option been given, the output file would list all possible places we could be, including derived places:

```
state 0

    0 $accept: . expr $end
    1 expr: . '(' expr ')'
    2     | . NUM
    3     | . NUM '+' NUM

    NUM  shift, and go to state 1
    '('  shift, and go to state 2
```

```
    expr  go to state 3
```

This is the difference between what is known in LR parser theory as the entire set of LR items vs just the kernel. It is a property of LALR(1) parsers that each state is associated uniquely with a particular set of kernel items.

After the blank line, the next group of lines give the possible actions for tokens. If a token other than one of those listed is encountered, this produces an ERROR action. In some cases, it is possible to make a reduction without looking at the next token, and a special token `$default` will appear in the listing. This is equivalent to a wildcard in the ACTION table. The last group of lines give the GOTO actions if this state is exposed after a reduction. These are listed by the symbol which has been reduced. Effectively, the GOTO table tells us what state to go to after the cursor dot has advanced through a non-terminal.

Let's trace out the steps which the Bison LALR parser generator algorithm takes in constructing the states and tables. In state 0, there are two possible tokens which can occur next and be valid, NUM and (. These lead to states 1 and 2 respectively:

```
state 1

    2 expr: NUM .
    3     | NUM . '+' NUM

    '+'  shift, and go to state 4

    $default  reduce using rule 2 (expr)
```

Having reached state 1 by having seen a NUM, a '+' token would lead us to believe that we are in the middle of rule #3, as depicted by the cursor dot. But any other lookahead token (the `$default` action) would lead us to believe that we have just seen an instance of rule #2. Note that there is no GOTO action in state 1, because the cursor dot is not positioned directly to the left of any non-terminal in any of the rules included in that state.

State 2 is reached after having seen a '(':

```
state 2

    1 expr: '(' . expr ')'

    NUM  shift, and go to state 1
    '('  shift, and go to state 2

    expr  go to state 5
```

We are almost in the same boat as state 0. We are beginning a new `expr`, and thus the same tokens are valid and lead to the same actions as in state 0. Let's follow from state 1 to state 4:

```
state 4

    3 expr: NUM '+' . NUM
```

```
    NUM  shift, and go to state 7
```

Again, NUM is the only valid token. If we see it, then we go to state 7:

```
state 7

    3 expr: NUM '+' NUM .

    $default  reduce using rule 3 (expr)
```

In state 7, it doesn't matter what the look-ahead token is, we always perform this default reduction. One might ask why the state exists? Why not simply reduce in state 4? The answer is that the parser must preserve the one-to-one correspondence between states and symbols. In order to perform a reduction `expr: NUM '+' NUM`, which has 3 rhs symbols, there must be 3 corresponding states on the stack. Thus we shift in state 4 and push another state (state 7) on the stack, "representing" the second instance of NUM. Then we pop 3 symbols from the stack during the reduction, and we have reduced to `expr`. Where can `expr` be found in a GOTO action? It can be found in state 2 or in state 0. In both cases, we see that the cursor was immediately to the left of `expr` in the rule, and now when we return back to that state, having reduced `expr`, it is only logical that we advance to a state where the cursor has stepped past `expr`:

```
state 3

    0 $accept: expr . $end

    $end  shift, and go to state 6

state 5

    1 expr: '(' expr . ')'

    ')'  shift, and go to state 8
```

In state 3, we must see `$end`, the end of input. This would take us to state 6:

```
state 6

    0 $accept: expr $end .

    $default  accept
```

in which the only action is to accept the input. Again, the parser appears to make an extraneous move (why not simply accept in state 3?) and this is to maintain consistency of the stack.

Looking at state 5, having seen `expr` following '(', the only valid token is ')', and that will lead to state 8:

```
state 8

    1 expr: '(' expr ')' .
```

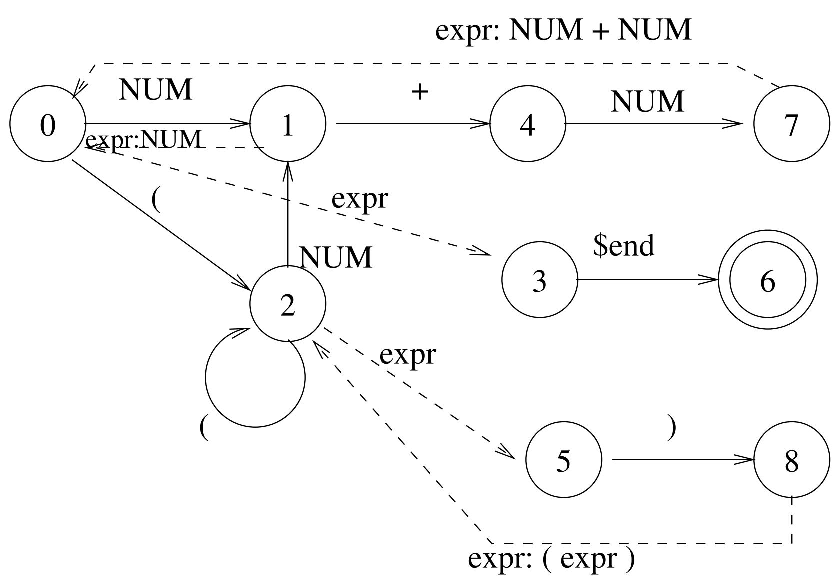
```
    $default  reduce using rule 1 (expr)
```

in which the default action for any token is to reduce by rule #1.

We have now seen all 8 states generated by this grammar. The ACTION and GOTO tables may be summarized:

Sym:	NUM	+	(	$end	)	expr
	ACTION TABLE					GOTO TABLE
State						
0	S1	err	S2	err	err	GOTO3
1	R2	S4	R2	R2	R2	n/a
2	S1	err	S2	err	err	GOTO5
3	err	err	err	S6	err	n/a
4	S7	err	err	err	err	n/a
5	err	err	err	err	S8	n/a
6	accept	accept	accept	accept	accept	n/a
7	R3	R3	R3	R3	R3	n/a
8	R1	R1	R1	R1	R1	n/a

A state transition diagram:



Let us follow the operation of the parser when the input is `( ( NUM ) )`. To disambiguate which token is which, we'll call the first left paren (1 and the second one (2, etc.

```
State stack      token          action
0                (1             SHIFT(2)
0 2              (2             SHIFT(2)
0 2 2            NUM            SHIFT(1)
0 2 2 1          )1             REDUCE(2)  expr: NUM
0 2 2            )1             GOTO (2,expr)==>5
0 2 2 5          )1             SHIFT(8)
0 2 2 5 8        )2             REDUCE(1) expr: ( expr )
0 2              )2             GOTO (2,expr)==>5
0 2 5            )2             SHIFT(8)
```

```
0 2 5 8        $end          REDUCE(1) expr: ( expr )
0              $end          GOTO (0,expr)==>3
0 3            $end          SHIFT(6)
0 3 6                        ACCEPT
```

We see that the LALR parser made as many shifts as there were input tokens, and as many reductions as there were non-terminals in the derivation. From this we can conclude that the LALR parser operates in linear time with respect to input length.

Lists

How would we write a grammar for a list of identifiers delimited by commas? One way is:

```
id_list:
          ID
          |id_list ',' ID
```

vs:

```
id_list:
          ID
          |ID ',' id_list
```

In LL parsing the first form is not acceptable since it is left-recursive. But in LR parsing, the second form, while usable, is sub-optimal. When faced with a right-recursive rule, the LR parser must shift the entire sequence of tokens making up that list, and then make a series of reductions. Since the list is potentially unbounded, this might overflow the LR parser stack. For this reason, lists are almost always written in left-recursive form when intended for LR parsing.

Conflicts

If, in a particular state, the LALR(1) parser generated by yacc or bison can not determine what action to take based on the single lookahead token, then there is a "conflict" in that state. Conflict comes in two forms:

• Shift/Reduce: The input seen up to this point matches a rule, but the lookahead token gives promise that we could continue to shift it (and possibly more tokens) and eventually match a different, "better" rule.

• Reduce/Reduce: There are two or more rules which match the input seen thus far, and the lookahead token doesn't help us in determining which one to apply.

Conflicts may arise because the grammar as written is truly ambiguous, or the grammar may be perfectly clear but it fails to meet the criteria for LALR(1). yacc/bison resolves shift/reduce conflicts in favor of shifting (unless some other means of resolution overrides), and considers these to be a warning. However, reduce/reduce conflicts have no default resolution because it isn't clear which semantic action to trigger. Bison picks the first rule mentioned, but this is almost certainly not what you want. It is a good idea to eliminate all shift/reduce or reduce/reduce conflicts, using the techniques described below.

Operator Precedence

Consider this expression grammar:

```
/* Grammar 6 */
%token NUM
%%
expr:   NUM
    |   expr '+' expr
    |   expr '-' expr
    |   expr '/' expr
    |   expr '*' expr
    |   '(' expr ')'
```

running this through bison yields 4 states with 4 shift/reduce conflicts each, for a total of 16. Let's look in the `.output` file created by `bison -v --report=look-ahead` at one of the conflicting states:

```
state 14

    2 expr: expr . '+' expr
    3     | expr . '-' expr
    4     | expr . '/' expr
    5     | expr . '*' expr
    5     | expr '*' expr .  [$end, '+', '-', '/', '*', ')']

    '+'  shift, and go to state 6
    '-'  shift, and go to state 7
    '/'  shift, and go to state 8
    '*'  shift, and go to state 9

    '+'       [reduce using rule 5 (expr)]
    '-'       [reduce using rule 5 (expr)]
    '/'       [reduce using rule 5 (expr)]
    '*'       [reduce using rule 5 (expr)]
    $default  reduce using rule 5 (expr)
```

Consider the input `2*3+4` at the point of reaching the +. The parser will see 2, reduce it to expr, shift the *, see 3 and reduce that to expr. Now, we *could* be at `expr '*' expr .` in which case we should reduce to expr. However, we *could* also be in the middle of `expr . '+' expr`, in which case we should shift the + and continue. The precedence is not defined in the grammar, which gives rise to this shift/reduce conflict between Shifting to state 6 and Reducing with rule 5. A similar precedence conflict is seen on the

Likewise, there would be an associativity conflict in an expression like `2*3*4` which can be seen with Shift-9 vs Reduce-5 on the '*' token or Shift-8 vs Reduce-5 on the '/' token.

We see by the fact that the conflicting reduce action is inside of square brackets that the parser has resolved the shift/reduce conflict in favor of shift, which is the default behavior. This has only a 50% chance of being correct, though, in terms of preserving the semantics of the language. Therefore it behooves us to resolve the conflict explicitly.

This conflict appeared earlier, as we were considering LL(1) grammars, and we solved it by re-writing the grammar to be unambiguous. However, yacc and bison have some nice conflict resolution mechanisms which are technically beyond the scope of LALR(1) parsing but come in handy. (In the listing above, the square brackets after the two instances of rule 5 list the possible look-ahead tokens for which a reduction by that rule in that state might be considered. This information is supplied only with the `--report=look-ahead` or `--report-all` options to bison. Note that while there are two LR(1) items in state 14 that involve rule #5, only the second one can lead to a reduction. A reduction can only take place when the "cursor" is at the end of the rule!)

In this case, we can assign precedence levels and associativity to the operator tokens. A new yacc/bison grammar is:

```
/* Grammar 7 */
%token NUM
%right '='
%left '+' '-'
%left '*' '/'
%%
expr:   NUM
    |   expr '+' expr
    |   expr '-' expr
    |   expr '/' expr
    |   expr '*' expr
    |   '(' expr ')'
    |   IDENT '=' expr              /* Added to illustrate right associativity */
    ;
```

Precedence resolves the conflict a `X b Y c`, where a,b and c are symbols, such as expr, and X/Y are (differing) operators. The symbol b is experiencing a conflict. When it has been shifted or appears at the top of the parser symbol stack, does it pull towards operator X, meaning REDUCE, or does it pull towards Y, meaning SHIFT? b should pull in the direction of the higher precedence operator. So consider a * b + c. b should pull towards the *, and a * b should reduce to expr.

Associativity resolves the conflict a `Z b Z c`, where Z is the same operator appearing on both sides of symbol b. If Z is a left-associative operator, then b pulls towards the left instance of Z (REDUCE). If Z is right- associative, b pulls to towards the right (SHIFT). So in the expression a - b - c, b should be pulled to the left (because subtraction is left-associative) and a - b is reduced to expr first. But in a = b = c, b pulls to the right and the assignment b = c is reduced to expr first.

The `%left` and `%right` declarations specify the associativity and precedence of the tokens. Tokens mentioned earlier have lower precedence. All tokens mentioned in the same line have the same precedence and associativity, i.e. it is impossible to have left-and right-associative tokens in the same precedence level.

It is also possible to specify `%nonassoc` for tokens which do not have any associativity. A non-associative operator is one where expr OP expr OP expr does not make sense. In the C language, there are no true examples. However, consider a language where the construct `a<b<c` is equivalent to the C expression `(a<b) && (b<c)`. In such a language, it would be proper to declare < and > as non-associative operators, to prevent the above expression from being parsed (as it would be in C) `(a<b)<c`.

Not only do tokens have precedence, but a rule inherits precedence from the **right-most token** on its rhs. The precedence of a rule can also be established explicitly with the `%prec` tag.

When there is a shift/reduce conflict, operator precedence/associativity resolves it by considering the conflicting rule(s) along with the look-ahead token, according to the following algorithm:

- If prec(token) > prec(rule), then shift
- If prec(token) < prec(rule), then reduce
- If prec(token)==prec(rule) and token is left-associative, reduce
- If prec(token)==prec(rule) and token is right-associative, shift
- If prec(token)==prec(rule) and token is non-associative, error
- If either the rule or the token has no defined precedence, shift (and Bison issues a shift/reduce conflict warning)

Now, let's look at the pesky state again (in Bison 2.3 or 2.4, it was necessary to specify `-v --report=all` to get this reporting, and with the disambiguating features, the state numbers changed):

```
state 16

    2 expr: expr . '+' expr
    3     | expr . '-' expr
    4     | expr . '/' expr
    5     | expr . '*' expr
    5     | expr '*' expr .  [$end, '+', '-', '*', '/', ')']

    $default  reduce using rule 5 (expr)

    Conflict between rule 5 and token '+' resolved as reduce ('+' < '*').
    Conflict between rule 5 and token '-' resolved as reduce ('-' < '*').
    Conflict between rule 5 and token '*' resolved as reduce (%left '*').
    Conflict between rule 5 and token '/' resolved as reduce (%left '/').
```

The input 2*3+4 is resolved as a reduce action, because the precedence of + is lower than the precedence of rule 5 (which inherits the precedence of *). Likewise, 2*3*4 is a reduce because * is left-associative.

The Dangling ELSE

A classic problem in block-structured programming languages is the nesting of if-then-else statements. Consider the grammar:

```
/* Grammar 8 */
stmt:   expr
  |     if_stmt
  ;

if_stmt:        IF '(' expr ')' stmt
   |            IF '(' expr ')' stmt ELSE stmt
   ;

expr:   NUM
          ;                  /* the ; terminates the rule, it is not a token! */
```

which is an abstraction of the C grammar. If we were to run this grammar through Bison, it would report 1 shift/reduce conflict. The `.output` file shows:

```
state 10

    3 if_stmt: IF '(' expr ')' stmt .  [$end, ELSE]
    4        | IF '(' expr ')' stmt . ELSE stmt

    ELSE  shift, and go to state 11

    ELSE      [reduce using rule 3 (if_stmt)]
    $default  reduce using rule 3 (if_stmt)
```

Consider the input `if (1) if (2) 3 else 4`, and the moves which the parser will make:

Symbol stack	LA token	Action
	IF	SHIFT
IF	(	SHIFT
IF (	NUM{1}	SHIFT
IF (NUM{1}	)	REDUCE: expr:NUM
IF (expr	)	SHIFT
IF (expr)	IF	SHIFT
IF (expr) IF	(	SHIFT
IF (expr) IF (	NUM{2}	SHIFT
IF (expr) IF (NUM{2}	)	REDUCE: expr:NUM
IF (expr) IF (expr	)	SHIFT
IF (expr) IF (expr)	NUM{3}	SHIFT
IF (expr) IF (expr) NUM{3}	ELSE	REDUCE: expr:NUM
IF (expr) IF (expr) expr	ELSE	REDUCE: stmt:expr
IF (expr) IF (expr) stmt	ELSE	conflict

We are now in state 10, and there are two possibilities. We could be at the end of rule #3, in which case we should reduce, or we could be in the middle of rule #4, in which case we should shift. The fact that the `reduce using rule3` action is in brackets tells us that bison has applied the default shift/reduce conflict resolution in favor of shifting. In this case, this is actually what we want. (Colloquially, we say that in the C language, an else associates with the nearest if.) However, it would be nice to say so explicitly and not have to deal with the warning. We could re-write the grammar to remove the ambiguity, but doing so would be awkward. However, we can apply the operator precedence mechanism:

```
/* Grammar 9 */
%left IF
%left ELSE
%token NUM
%%
stmt:   expr
  |     if_stmt
  ;

if_stmt:        IF '(' expr ')' stmt                    %prec IF
   |            IF '(' expr ')' stmt ELSE stmt            %prec ELSE
   ;

expr:   NUM;
```

Now we have listed IF and ELSE so that IF has lower precedence. We need the `%prec IF` on the first rule because otherwise the rule would inherit the precedence of the rightmost token which is ')', and that has no defined precedence here. The `%prec ELSE` on the second rule is redundant and purely stylistic. The use of operator precedence forces a shift on ELSE in rule 3, eliminating the warning:

```
state 10

    3 if_stmt: IF '(' expr ')' stmt .  [$end]
    4        | IF '(' expr ')' stmt . ELSE stmt

    ELSE  shift, and go to state 11

    $default  reduce using rule 3 (if_stmt)

    Conflict between rule 3 and token ELSE resolved as shift (IF < ELSE).
```

Mysterious Lookahead ambiguities

Sometimes yacc/bison can not generate a parser for a grammar because of limited lookahead. We have seen an LR(2) grammar and clearly LALR(1) can not deal with it because it needs two tokens of lookahead. But consider this grammar (from the Bison manual):

```
/*Grammar 10*/
%token ID
%%
def:      param_spec return_spec ','
          ;

param_spec:
          type
          | name_list ':' type
          ;
```

```
name_list:
          name
          | name_list ',' name
          ;

return_spec:
          type
          | name ':' type
          ;

type:     ID;

name:     ID;
```

The intent of this contrived grammar is to parse function prototype declarations with a sort of inside-out and backwards syntax. Thus `a,b,c:int foo:int,` declares a function called foo taking integer arguments a, b and c. Likewise `int int,` is an abstract declarator giving a type of a function (of abstract name) returning an integer and accepting a single integer argument.

Consider the input `ID ','`. It should be possible to parse this without ambiguity. It is not a problem that there are two rules which have the exact same right-hand-side. Because `param_spec` can not derive to ε, and because `return_spec` can not contain a comma, then the token sequence ID ',' occurring at the start of input must be `param_spec -> name_list -> name`. Even though this grammar is LR(1), in yacc or bison it produces a reduce/reduce conflict:

```
state 1

    8 type: ID .  [ID, ',']
    9 name: ID .  [',', ':']

    ','       reduce using rule 8 (type)
    ','       [reduce using rule 9 (name)]
    ':'       reduce using rule 9 (name)
    $default  reduce using rule 8 (type)
```

The problem is that LALR has combined two kernels into state 1, and thus can't decide whether ID is a `type` or a `name`. It shouldn't have done that, but this is a deficiency of the LALR algorithm. This grammar is LR(1) but not LALR(1). One way to fix this is to introduce an invalid production, now:

```
/* Grammar 11 */
return_spec:
          type
          | name ':' type
          | ID BOGUS
          ;
```

Where BOGUS is a dummy token which will never be returned by the lexer. This forces some of the states to be split and fixes the ambiguity. Another approach might be to flatten the grammar and remove the `type` and `name` rules, inlining their definition where they appear in the `param_spec`, `return_spec` and `name_list` rules.

One might ask what is the purpse of having two different symbols, `type` and `name` which are the same thing? For purely parsing purposes, this is pointless. But of course parsing is not the end goal of the compiler. We may want to take different semantic actions (see below) depending on what symbol is recognized.

Other conflict problems

There are other cases where LALR(1) parsers get in to trouble. Thankfully, in most real-world cases, they can be fixed, which is why LALR(1) parsers continue to be the most widely used. If the conflict can not be resolved entirely within the parser, it might be necessary to simplify the grammar and try to fix things in the semantic stage. Alternatively, it might be possible to pass information back down to the lexer to cause it to alter behavior and return different token codes to resolve the problem. The Yacc and Bison documentation, as well as the O'Reilly Book, give examples of this approach.

Semantic Actions

We have seen that the yacc/bison state stack corresponds to the symbols (terminals or non-terminals) in progress. In addition to this state, yacc/bison maintains a semantic value stack in lock-step with the state stack. Every symbol has a semantic value. Values of terminals are set by the lexer through the variable `yylval`, while values of non-terminals are computed by embedded actions. When a rule is reduced, the value stack is accessible from within the embedded action by use of `$` macros. It is easiest to give an example:

```
/* Grammar expr.y */
/* %debug generates code to print out debugging actions */
%debug
/* %error-verbose causes bison to generous meaningful syntax error messages */
%error-verbose
%{
#define YYDEBUG 1       /* enable debug code to compile */
%}
%token NUM
%token NL
%left '+' '-'
%left '*' '/'
%nonassoc '('
%%
exprlist:       expr            {printf("EXPR VALUE IS %d\n",$1);}
        | exprlist NL expr      {printf("EXPR VALUE IS   %d\n",$3);}
        ;

expr:   NUM             {$$=$1;}
   |    expr '+' expr   {$$ = $1 + $3;}
   |    expr '-' expr   {$$=$1 - $3;}
   |    expr '/' expr   {if ($3 ==0) fprintf(stderr,
                        "/0 err\n"); else $$=$1 / $3;}
   |    expr '*' expr   {$$=$1*$3;}
   |    '-' expr %prec '(' {$$= -1 * $2; }
/* The %prec above forces unary minus to have a precedence higher than
        multiplication or division.  Technically, the expression
        -B/C should be parsed (-B)/C, not -(B/C) */
   |    '(' expr ')'    {$$=$2;}
   ;

%%
main()
{
        yydebug=1;      /* turn on run-time debugging output from Bison */
        yyparse();
}

yyerror(char *err)
{
        fprintf(stderr,"syntax error:%s\n",err);
}
```

Consider the input `4+5`. The parser will make the following moves:

```
State/Symbol Stack              LA token        Value stack     ACTION
                                NUM                             SHIFT
NUM                             +               4               REDUCE
expr                            +               4               SHIFT
expr +                          NUM             4 ??            SHIFT
expr + NUM                      $end            4 ?? 5          REDUCE
expr + expr                     $end            4 ?? 5          REDUCE
expr                            $end            9               ACCEPT
```

When a token is shifted, the value of the global variable `yylval`, which has been set by the lexer, is pushed onto the value stack. (In the example above, tokens such as '+' have no meaningful value. The lexer would probably not assign to `yylval` in this case, and its value would remain the same as the last token seen with an actual semantic value. Since the value of '+' is of no consequence to us, we leave it undefined. Beware: referencing the wrong semantic value is a common yacc/bison error)

During a reduction, as many value stack entries are popped from the stack as there are symbols on the rhs of the corresponding rule. The code embedded in curly braces after the rule is executed. This code is copied verbatim into the `yyparse` function, except that occurrences of `$1` are translated to an expression which accesses the first (leftmost) of the popped values, i.e. it corresponds to the value of the first rhs symbol. Likewise, `$2` corresponds to the second symbol, etc. Or in general, $n is

```
value_stack_top[n-NRHS]
```

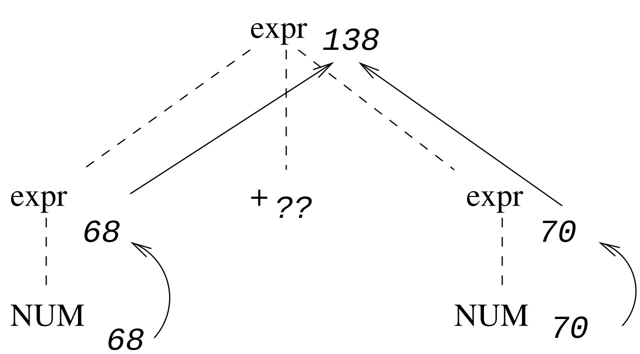
where NRHS is the number of symbols on the right-hand-side of the rule.

`$$` represents a temporary variable which is used to hold the computed value which will become the semantic value of the symbol on the lhs of the rule. After the rule has executed, this `$$` value will be pushed onto the value stack.

Yacc/Bison implicitly assign $$=$1 before any explicit embedded action code. Therefore, if there is no embedded action, or the action does not contain an explicit assignment to `$$`, then the default action `$$=$1` takes place, i.e. the value of the lhs is the value of the first rhs symbol.

Synthesized and Inherited Attributes

Another way to view this process is to draw the parse tree, annotated with the semantic value associated with each node of the tree.



The arrows indicate data flow. The leaves of the parse tree are the terminals (tokens), and their semantic values are established by the lexer. Note that in the tree above, the value of any non-terminal node depends solely on the values of its children. I.e. all of the dataflow arrows point "up." Such values are called **Synthesized Attributes**, and when a grammar with embedded actions relies solely on synthesized attributes, it is known as an **S-Attributed Grammar**.

Note also that since the LR parser always makes a depth-first traversal of the parse tree, evaluation of S-attributed grammar comes naturally.

Now consider this grammar and semantic actions, intended to represent a simplified C-style declaration syntax.

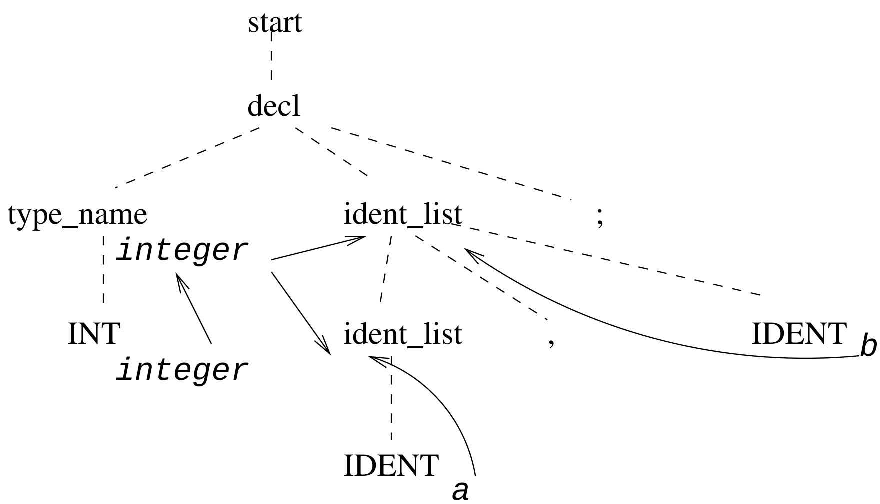
```
/* Grammar 13 */
%token INT
%token DOUBLE
%token IDENT
%{
#define YYSTYPE char*
%}


%%
start: decl
        |start decl
        ;

decl: type_name ident_list ';'
        ;

ident_list:
        ident_list ',' IDENT    {printf("%s declared as %s\n",$3,$0);$$=NULL;}
        |IDENT                  {printf("%s declared as %s\n",$1,$0);$$=NULL;}
        ;

type_name:
        INT                             {$$="integer";}
        |DOUBLE                         {$$="double";}
        ;
%%
```

Consider an input of `int a,b;` and the moves which the parser makes:

```
State/symbol stack              LA              Value stack     ACTION
                                INT                             SHIFT
INT                             IDENT           ?               REDUCE
type_name                       IDENT           integer         SHIFT
type_name IDENT                 ,               integer a       REDUCE
        (prints "a declared as integer")
type_name ident_list            ,               integer NULL    SHIFT
type_name ident_list ,          IDENT           integer NULL ?  SHIFT
type_name ident_list , IDENT    ;               integer NULL ? b REDUCE
         (prints "b declared as integer")
type_name ident_list            ;               integer NULL    SHIFT
```

The notation `$0` refers to the semantic value just to the left of (i.e. just below in the value stack) the value of the leftmost symbol on the rhs of the rule. Call the rule being reduced C. The symbol C associated with the rule (the lhs) must appear in some other rule which is higher in the parse tree. Call that rule P. Then `$0` is the value of the symbol just to the left of C in rule P. I.e. to locate `$0`, we go up one level and one step to the left. (If we go up one level and there are no symbols to the left, we go upwards until there are...the reasoning behind this is apparent from examination of the semantic value stack). Likewise, `$-1` would refer to the symbol two places to the left of C in rule P.

Again, visualizing the attributed parse tree:



We see that the data flow arrows point either up or down. Those attributes which depend on nodes which are not children are termed **Inherited Attributes**. If an attributed grammar contains only either synthesized attributes or attributes inherited from above and to the left, that is known as an `L-attributed grammar`.

While we could conceive of other ways of attributing a parse tree in which the dataflow does not satisfy L-attributed, such schemes would be difficult to evaluate practically (or impossible if the dataflow graph contained circular dependencies), because we would have attributes which depend on symbols which we have not yet seen.

In yacc or bison, caution is suggested when using inherited attributes. This is because they depend on the position of the rule in another rule. If the rule which is using inherited attributes can appear in multiple parent rules, the attribute in which we are interested might be a different number of steps to the left, and then we can not write an action which will consistently reference it (correctly). We might also rewrite part of the grammar and inadvertently change the position of inherited attributes, leading to incorrect results.

In the "declaration" example above, if we wanted to avoid inherited attributes, we could instead process the entire declaration, storing the `ident_list` using some internal format (e.g. a linked list), and then taking a second pass at the end of the rule. Then the type and the identifiers list would both be accessible as synthesized attributes.

Note that in the actions for the two `ident_list` rules, we assigned `$$=NULL`, even though the semantic value for `ident_list` is never referenced elsewhere in the code. This is a good defensive coding practice. Yacc/Bison implicitly do `$$=$1` before any explicit embedded action. Therefore, had we not made the explicit assignment, `ident_list` would receive the semantic value associated with either `ident_list` or `IDENT` in those rules. This could lead to very subtle and elusive errors. Say, in another embedded action, we mistakenly refer to the value of `ident_list`, which has no meaningful value. Without our explicit NULL, we would get a value which is probably a valid pointer, but of course is the wrong valid pointer! This pointer may in turn get buried and referenced in other data structures, creating a data corruption bug that will drive one crazy! By setting it to NULL, we will access an invalid pointer and the compiler will die. On the surface this sounds worse, but it is better to fail and let the end-user (or invoking make program) know it, than to seemingly work but produce incorrect output.

Mid-rule actions

It is possible in yacc or bison to embed an action in the middle of a rule:

```
rule: sym1 sym2
          {printf("sym1=%d sym2=%d\n",$1,$2);$$=$1+$2;} sym4
          {printf("midrule=%d\\n",$3);}
          ;
```

Recall that in the LALR parser algorithm, the embedded actions are executed when a REDUCE is performed. Yacc/Bison therefore must introduce a dummy non-terminal symbol at the point of the mid-rule action. This dummy symbol must always reduce, i.e. it must be ε. So the mid-rule action above is equivalent to:

```
rule: sym1 sym2 midrule sym4
          ;

midrule:  /*empty*/
                    ;  {printf("sym1=%d sym2=%d\n",$-1,$0);$$=$-1+$0;}
```

The ε production may cause what had been a perfectly fine LALR(1) grammar to become LR(2). Consider above, if after having seen `sym1 sym2`, if it can not be determined based on the one look-ahead token whether we are about to see `sym4`, then the introduction of the mid-rule action will break the grammar. It can be shown that if the grammar (or the portion of it to which we wish to apply the mid-rule action) was formerly LL(1), then the addition of the ε production will make it LR(1). But if it was not LL(1) before, then it will not be LR(1) afterwards, and we can not introduce the action.

As a programming convenience, the semantic values of the symbols to the left of a mid-rule action can be referenced using $1, $2, etc. Yacc/Bison will translate that to $-1, $0, etc., because it recognizes that this is a mid-rule action. However, if the mid-rule action is moved to an explicit ε dummy non-terminal, then those exact same values must be referenced using $0, $-1, etc. In the example above, `sym2` is immediately to the left in the rule in which `midrule` appears, therefore it is $0. Likewise `sym1` is two symbols to the left in the containing rule, so it is referenced using $-1.

The mid-rule action may set $$. This does not set the semantic value of the overall rule, but just the value of the dummy symbol which represents the mid-rule action. This value can be referenced by the final reduce action in the containing rule (or by other mid-rule actions which appear later in that same rule). In the example above, the mid-rule action is the third symbol in `rule`, and the end-rule action references its value using $3. If the mid-rule action does not set $$, the resulting value is **undefined**, because the implicit $$=$1 assignment is referencing a non-existent symbol (the empty production has 0 symbols).

The introduction of a mid-rule action into an existing rule of course increments the $n nomenclature for the symbols to the right of it. Other actions in that rule need to be manually adjusted for the new $n values. This is a frequent source of error.

Typically a mid-rule action is used in a compiler when it is desired to establish some kind of global state change before shifting the remaining part of a rule. For example, upon seeing the opening brace which starts a new scope in the C language, one might want to adjust the symbol table to be in that scope.

```
compound_statement:
          '{' {enter_block();} decl_or_stmt_list '}' {leave_block();}
```

Another common source of error is failing to realize that when the mid-rule action is triggered, the first token of the next symbol has already been processed by the lexer. In the above example, if it is important that the first token of a `decl_or_stmt_list` be processed in the new scope, the mid-rule action must be moved to before the opening brace token. However, depending on the rest of the grammar, this may create a LALR conflict.

Using the Yacc/Bison %union directive

The type of the value stack of Yacc/Bison is represented by a typedef name `YYSTYPE`. By default, this is defined to be `int`. In practical compilers, this is rarely what is desired. By using the `%union {` directive in the .y file, yacc/bison will include a union declaration for `YYSYTPE`. Everything which appears between the opening brace of the %union and its closing brace is copied verbatim (comments too) into the .tab.h file, inside of a `union` typedef declaration.

Typically, inside the %union, one declares a union of all the possible data types to be held on the stack. Different grammar symbols have different data types associated with them. The value stack is an array of unions. I.e. each slot of this array is of fixed size in memory, and is large enough to hold the largest data type.

Yacc/Bison does not actually understand the C language. This point is often overlooked by novices. One can write anything inside of the %union directive. Yacc/Bison can't tell if it is valid C. It also can't parse the union declaration and figure out what the valid names are. One must associate data types with each terminal or non-terminal symbol, by using the < type > notation. This is applied to tokens by attaching the type to a %left, %right, %nonassoc or %token directive. For non-terminals, the %type directive is used. Each symbol must have only one type defined for it.

When a semantic value is referenced and the grammar had a %union directive, access to that value translates to C code (in the .tab.c file) similar to:

```
value_stack_pointer[offset].type_name
```

Because Yacc/Bison can't parse the union declaration, it is up to the programmer to correctly associate the Yacc/Bison type names with the corresponding field definitions in the union. Errors will not be detected until the .tab.c file is compiled.

When using a %union, every symbol in the grammar must have a type, even if its value is never explicitly set or referenced, because of the implicit $$=$1; action. Mid-rule actions do not have a symbol name, so if they need to set their own $$ value, the $<type_name>$ notation must be used. Similarly, if a mid-rule action value is referenced in and end-of-rule action, say as the third symbol, then the notation needs to be $<type_name>3. Consult the Bison manual for more comprehensive information.