Bottom Up Parsing

- Parsing algorithms which proceed from the bottom of the derivation tree and apply grammar rules (in reverse) **.**
- A bottom up parse is similar to a derivation in reverse

Bottom Up Parsing

- Each time a grammar rule is applied to a sentential form, the rewriting rule is applied backwards.
- Consequently, derivation trees are constructed, or traversed, from bottom to top.

Bottom Up Parsing

Shift* Reduce *Parsing

Bottom up parsing involves two fundamental operations.

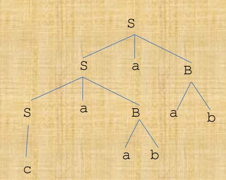
✓The process of moving an input symbol to the stack is called a shift operation.

✓The process of replacing symbols on the top of the stack with a nonterminal is called a reduce operation (it is a derivation step in reverse).

Bottom Up Parsing

Shift Reduce Parsing

```
G19:
1. S → S a B
2. S → c
3. B → a b
```



A Derivation Tree for the String `caabaab`, Using Grammar G19

Bottom Up Parsing

Shift Reduce Parsing

```
G19:
1. S → S a B
2. S → c
3. B → a b
```

S ⇒ S a B ⇒ S a a b ⇒ S a B a a b
⇒ S a a b a a b ⇒ c a a b a a b

The string of symbols being reduced is called a handle, and it is imperative in bottom up parsing that the algorithm be able to find a handle whenever possible.

Bottom Up Parsing

▽	caabaab↵	shift
▽c	aabaab↵	reduce using rule 2
▽S	aabaab↵	shift
▽Sa	abaab↵	shift
▽Saa	baab↵	shift
▽Saab	aab↵	reduce using rule 3
▽SaB	aab↵	reduce using rule 1
▽S	aab↵	Shift
▽Sa	ab↵	Shift
▽Saa	b↵	Shift
▽Saab	↵	reduce using rule 3
▽SaB	↵	reduce using rule 1
▽S	↵	Accept

Shift Reduce Parsing

```
G19:
1. S → S a B
2. S → c
3. B → a b
```

Sequence of Stack Frames Parsing `caabaab` using Grammar G19

Bottom Up Parsing

Shift Reduce Parsing

- ✓A shift/reduce conflict occurs when the parser does not know whether to shift an input symbol or reduce the handle on the stack.
- ✓A reduce/reduce conflict occurs when there is more than one grammar rule whose right hand side matches the top of the stack.

Bottom Up Parsing

Shift Reduce Parsing

How to resolve these conflicts?

1. Making an assumption.
2. Rewrite the grammar.
3. Looking ahead at additional input characters LR(k).

Bottom Up Parsing

Shift Reduce Parsing

A bottom up programming languages generally defines the language using LR(1) grammar because an ambiguous grammar will always produce conflicts when parsing bottom up with the shift reduce algorithm.

Bottom Up Parsing

Shift Reduce Parsing

▽	aaab↵	shift
▽a	aab↵	reduce using rule 2
▽S	aab↵	shift
▽Sa	ab↵	shift/reduce conflict
		reduce using rule 2 (incorrect)
▽SS	ab↵	shift
▽SSa	b↵	shift
▽SSab	↵	reduce using rule 3
▽SSB	↵	Syntax error (incorrect)

```
G20:
1. S → S a B
2. S → a
3. B → a b
```

An Example of a Shift/Reduce Conflict Leading to an Incorrect Parse Using Grammar G20

Bottom Up Parsing

Shift Reduce Parsing

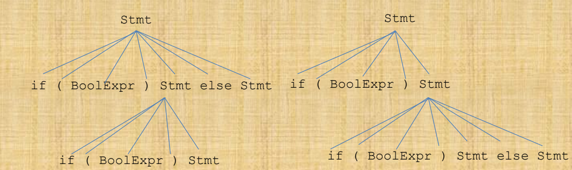
A Reduce/Reduce Conflict

```
G21:
1. S → S A
2. S → a
3. A → a
```

▽	aa↵	shift
▽a	a↵	reduce/reduce conflict (rules 2 and 3)
		reduce using rule 3 (incorrect)
▽A	a↵	shift
▽Aa	↵	reduce/reduce conflict (rules 2 and 3)
		reduce using rule 2 (rule 3 will also yield a syntax error)
▽AS	↵	Syntax error

Bottom Up Parsing

Shift Reduce Parsing



Two Derivation Trees for

```
if (BoolExpr) if (BoolExpr) Stmt else Stmt

   1. Stmt → if (BoolExpr) Stmt else Stmt
   2. Stmt → if (BoolExpr) Stmt
```

Bottom Up Parsing

Shift Reduce Parsing

```
1. Stmt → if (BoolExpr) Stmt else Stmt
2. Stmt → if (BoolExpr) Stmt
```

```
if (BoolExpr) if (BoolExpr) Stmt else Stmt
```

Stack	Input
∇... if (BoolExpr) Stmt	else ... ↵

Parser Configuration Before Reading the else Part of an if Statement

Bottom Up Parsing

Shift Reduce Parsing

Sample Problem 5.1
Show the sequence of stack and input configurations as the string caab is parsed with a shift reduce parser, using grammar G19.
Solution:

∇	caab↵	shift
∇c	aab↵	reduce using rule 2
∇S	aab↵	shift
∇Sa	ab↵	shift
∇Saa	b↵	shift
∇Saab	↵	reduce using rule 3
∇SaB	↵	reduce using rule 1
∇S	↵	Accept

```
G19:
1. S → S a B
2. S → c
3. B → a b
```

Bottom Up Parsing

LR Parsing With Tables

- ✓This technique makes use of two tables to control the parser.
- ✓The first table, called the action table, determines whether a shift or reduce is to be invoked.
- ✓If it specifies a reduce, it also indicates which grammar rule is to be reduced.
- ✓The second table, called a goto table, indicates which stack symbol is to be pushed on the stack after a reduction.

Bottom Up Parsing

LR Parsing With Tables

For example, suppose we have the following stack and input configuration:

Stack	Input
∇S	ab↵

The action shift will result in the following configuration:

Stack	Input
∇Sa	b↵

The a has been shifted from the input to the stack.

Bottom Up Parsing

LR Parsing With Tables

Suppose, then, that in the grammar, rule 7 is:
`7. B → Sa`
Select the row of the goto table labeled ∇, and the column labeled B.

If the entry in this cell is push `X`, then the action reduce 7 would result in the following configuration:

Stack	Input
∇X	b↵

Bottom Up Parsing

LR Parsing With Tables

The operation of the LR parser can be described as follows:

1. Find the action corresponding to the current input and the top stack symbol.

2. If that action is a shift action:
   a. Push the input symbol onto the stack.
   b. Advance the input pointer.

Bottom Up Parsing

LR Parsing With Tables

3. If that action is a reduce action:

   a. Find the grammar rule specified by the reduce action.
   b. The symbols on the right side of the rule should also be on the top of the stack – pop them all off the stack.
   c. Use the nonterminal on the left side of the grammar rule to indicate a column of the goto table, and use the top stack symbol to indicate a row of the goto table. Push the indicated stack symbol onto the stack.
   d. Retain the input pointer.

Bottom Up Parsing

LR Parsing With Tables

4. If that action is blank, a syntax error has been detected.

5. If that action is Accept, terminate.

6. Repeat from step 1.

Bottom Up Parsing

LR Parsing With Tables

Action and Goto Tables to Parse Simple Arithmetic Expressions

▽

Initial Stack

```
G 5
1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor →( Expr )
6. Factor → var
```

Bottom Up Parsing

LR Parsing With Tables

Action and Goto Tables to Parse Simple Arithmetic Expressions

	Action Table					
	+	*	(	)	var	↲
▽			shift (		shift var	
Expr1	shift +					Accept
Term1	reduce 1	shift *		reduce 1		reduce 1
Factor3	reduce 3	reduce 3		reduce 3		reduce 2
(			shift (		shift var	
Expr5	shift +			shift)		
)	reduce 5	reduce 5		reduce 5		reduce 5
+			shift (		shift var	
Term2	reduce 2	shift *		reduce 2		reduce 2
*			shift (		shift var	
Factor4	reduce 4	reduce 4		reduce 4		reduce 4
var	reduce 6	reduce 6		reduce 6		reduce 6

Bottom Up Parsing

LR Parsing With Tables

Action and Goto Tables to Parse Simple Arithmetic Expressions

	Goto Table		
	Expr	Term	Factor
▽	push Expr1	push Term2	push Factor4
Expr1			
Term1			
Factor3			
(	push Expr5	push Term2	push Factor4
Expr5			
)			
+		push Term1	push Factor4
Term2			
*			push Factor3
Factor4			
var			

Bottom Up Parsing

LR Parsing With Tables

```
G5
1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor →( Expr )
6. Factor → var
```

Sequence of configurations when Parsing
(var+var)*var

▽	(var+var)*var↲	shift (	
▽(	var+var)*var↲	shift var	
▽(var	+var)*var↲	reduce 6	push Factor4
▽(Factor4	+var)*var↲	reduce 4	push Term2
▽(Term2	+var)*var↲	reduce 2	push Expr5
▽(Expr5	+var)*var↲	shift +	
▽(Expr5+	var)*var↲	shift var	
▽(Expr5+var	)*var↲	reduce 6	push Factor4
▽(Expr5+Factor4	)*var↲	reduce 4	push Term1
▽(Expr5+Term1	)*var↲	reduce 1	push Expr5

Bottom Up Parsing

```
G5
1. Expr → Expr + Term
2. Expr → Term
3. Term → Term * Factor
4. Term → Factor
5. Factor →( Expr )
6. Factor → var
```

LR Parsing With Tables

Sequence of configurations when Parsing `(var+var)*var`

▽(Expr5	)*var↵	shift)	
▽(Expr5)	*var↵	reduce 5	push Factor4
▽Factor4	*var↵	reduce 4	push Term2
▽Term2	*var↵	shift *	
▽Term2*	var↵	shift var	
▽Term2*var	↵	reduce 6	push Factor3
▽Term2*Factor3	↵	reduce 3	push Term2
▽Term2	↵	reduce 2	push Expr1
▽Expr1	↵	Accept	

LR Parsing With Tables

Sample Problem 5.2
Show the sequence of stack, input, action, and goto configurations for the input var∗var using the parsing tables of Figure 5.7.

▽	var*var↵	shift var	
▽var	*var↵	reduce 6	push Factor4
▽Factor4	*var↵	reduce 4	push Term2
▽Term2	*var↵	Shift*	
▽Term2*	var↵	shift var	
▽Term2*var	↵	reduce 6	push Factor3
▽Term2*Factor3	↵	reduce 3	push Term2
▽Term2	↵	reduce 2	push Expr1
▽Expr1	↵	Accept	

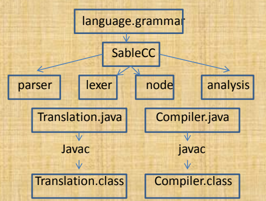
SableCC
Overview of SableCC

- ✓The user of SableCC prepares a grammar file as well as two java classes: Translation and Compiler.
- ✓These are stored in the same directory as the parser, lexer, node, and analysis directories.
- ✓Using the grammar file as input, SableCC generates java code the purpose of which is to compile source code as specified in the grammar file.

SableCC
Overview of SableCC

- ✓SableCC generates a lexer and a parser which will produce an abstract syntax tree as output.
- ✓If the user wishes to implement actions with the parser, the actions are specified in the Translation class.

SableCC
Overview of SableCC



Generation and Compilation of a Compiler Using SableCC

SableCC
Structure of the SableCC Source Files

There are six sections in the grammar file:

1. Package
2. Helpers
3. States
4. Tokens
5. Ignored Tokens
6. Productions

SableCC
Structure of the SableCC Source Files

- The Ignored Tokens section gives you an opportunity to specify tokens that should be ignored by the parser (typically white space and comments).

- The Productions section contains the grammar rules for the language being defined. This is where syntactic structures such as statements, expressions, etc. are defined.

SableCC
Structure of the SableCC Source Files

- Each definition consists of the name of the syntactic type being defined (i.e. a nonterminal), an equal sign, an EBNF definition, and a semicolon to terminate the production.

- All names in this grammar file must be lower case.

SableCC
Structure of the SableCC Source Files

An example of a production defining a while statement

```
stmt = while l_par bool_expr r_par stmt ;
```

The semicolon at the end is not the token for a semicolon, but a terminator for the stmt rule.

SableCC
Structure of the SableCC Source Files

Productions may use EBNF-like constructs.
If x is any grammar symbol, then:

```
x? // An optional x (0 or 1 occurrences of x)

x* // 0 or more occurrences of x

x+ // 1 or more occurrences of x
```

SableCC
Structure of the SableCC Source Files

Alternative definitions, using |, are also permitted and must be labeled with names enclosed in braces.

The following defines an argument list as 1 or more identifiers, separated with commas:

```
arg_list = {single} identifier
       | {multiple} identifier ( comma identifier ) + ;
```

SableCC
Structure of the SableCC Source Files

Labels must also be used when two identical names appear in a grammar rule.

Each item label must be enclosed in brackets, and followed by a colon:

```
for_stmt = for l_par [init]: assign_expr [s1]:semi bool_expr
          [s2]:semi [incr]: assign_expr r_par stmt ;
```

SableCC
An Example Using SableCC

The purpose of this example is to translate infix expressions involving addition, subtraction, multiplication, and division into postfix expressions, in which the operations are placed after both operands.

```
Infix                  Postfix
2 + 3 * 4              2 3 4 * +
2 * 3 + 4              2 3 * 4 +
( 2 + 3 ) * 4          2 3 + 4 *
2 + 3 * ( 8 – 4 ) - 2  2 3 8 4 - * + 2 -
```

SableCC
An Example Using SableCC

G 5:

```
1. Expr → Expr + Term
2. Expr → Expr - Term
3. Expr → Term
4. Term → Term * Factor
5. Term → Term / Factor
6. Term → Factor
7. Factor → ( Expr )
8. Factor → number
```

SableCC
An Example Using SableCC

The grammar file is shown below:

```
Package postfix;
Tokens
 number = ['0'..'9']+;
 plus = '+';
 minus = '-';
 mult = '*';
 div = '/';
 l_par = '(';
 r_par = ')';
 blank =
  (' ' | 10 | 13 | 9)+;
 semi = ';' ;
Ignored Tokens
 blank;
```

```
Productions
 expr =
      {term} term |
      {plus} expr plus term |
      {minus} expr minus term
      ;
 term =
      {factor} factor |
      {mult} term mult factor |
      {div} term div factor
      ;
 factor =
      {number} number |
      {paren} l_par expr r_par
      ;
```

SableCC
An Example Using SableCC

SableCC will produce a class called `DepthFirstAdapter`, which has methods capable of visiting every node in the syntax tree.

In order to implement actions, all we need to do is extend `DepthFirstAdapter` (the extended class is usually called `Translation`), and override methods corresponding to rules (or tokens) in our grammar.

SableCC
An Example Using SableCC

Since our grammar contains an alternative, Mult, in the definition of Term, the `DepthFirstAdapter` class contains a method named `outAMultTerm`.

It will have one parameter which is the node in the syntax tree corresponding to the Term.

SableCC
An Example Using SableCC

```
public void outAMultTerm (AMultTerm node)
```

This method will be invoked when this node in the syntax tree, and all its descendants, have been visited in a depth-first traversal.

SableCC
An Example Using SableCC

```
package postfix;
import postfix.analysis.*; // needed for DepthFirstAdapter
import postfix.node.*;  // needed for syntax tree nodes.
class Translation extends DepthFirstAdapter
{
public void outAPlusExpr(APlusExpr node)
{// out of alternative {plus} in expr, we print the plus.
System.out.print ( " + ");
}
public void outAMinusExpr(AMinusExpr node)
{// out of alternative {minus} in expr, we print the minus.
System.out.print ( " - ");
}
```

SableCC
An Example Using SableCC

```
public void outAMultTerm(AMultTerm node)
{// out of alternative {mult} in term, we print the minus.
System.out.print ( " * ");
}
public void outADivTerm(ADivTerm node)
{// out of alternative {div} in term, we print the minus.
System.out.print ( " / ");
}
public void outANumberFactor (ANumberFactor node)
// out of alternative {number} in factor, we print the
// number.
{ System.out.print (node + " "); }
}
```

SableCC
An Example Using SableCC

Other methods in the `DepthFirstAdapter` class.

An '`in`' method for each alternative, which is invoked when a node is about to be visited.

```
public void inAMultTerm (AMultTerm node)
```

SableCC
An Example Using SableCC

A '`case`' method for each alternative.

This is the method that visits all the descendants of a node, and it is not normally necessary to override this method.

```
public void caseAMultTerm (AMultTerm node)
```

A '`case`' method for each token; the token name is prefixed with a '`T`' as shown below:

```
public void caseTNumber (TNumber token)
{ // action for number tokens }
```

SableCC
An Example Using SableCC

SableCC tends to alter names that were included in the grammar.

This is done to prevent ambiguities.

For example, `l_par` becomes `LPar`, and `bool_expr` becomes `BoolExpr`.

SableCC
An Example Using SableCC

- ✓An important problem to be addressed is how to invoke an action in the middle of a rule (an embedded action).
- ✓All the user needs to do is to copy the case method from `DepthFirstAdapter`, and add the action at the appropriate place.

SableCC
An Example Using SableCC

```
public void caseAWhileStmt (AWhileStmt node)
{ inAWhileStmt(node);
  if(node.getWhile() != null)
  { node.getWhile().apply(this) }
    System.out.println ("LBL");
   // embedded action
  if(node.getLPar() != null)
  { node.getLPar().apply(this); }
  if(node.getBoolExpr() != null)
  { node.getBoolExpr().apply(this); }
  if(node.getRPar() != null)
  { node.getRPar().apply(this); }
  if (node.getStmt() != null)
  { node.getStmt().apply (this) ; }
    outAWhileStmt (node);
}
```

SableCC
An Example Using SableCC

```
package postfix;
import postfix.parser.*;
import postfix.lexer.*;
import postfix.node.*;
import java.io.*;
public class Compiler
{
  public static void main(String[] arguments)
  { try
    { System.out.println("Type one expression");
      // Create a Parser instance.
      Parser p = new Parser ( new Lexer ( new PushbackReader
                  ( new InputStreamReader(System.in), 1024)));
      // Parse the input.
      Start tree = p.parse();
      // Apply the translation.
      tree.apply(new Translation());
      System.out.println();
    }
    catch(Exception e)
    { System.out.println(e.getMessage()); }
    }
}
```

SableCC
An Example Using SableCC

Sample Problem 5.3
Use SableCC to translate infix expressions involving addition, subtraction, multiplication,and division of whole numbers into atoms. Assume that each number is stored in a temporary memory location when it is encountered.

SableCC
An Example Using SableCC

For example, the following infix expression:

```
34 + 23 * 8 - 4
```

should produce the list of atoms:

```
MUL T2 T3 T4
ADD T1 T4 T5
SUB T5 T6 T7
```

Here it is assumed that `34` is stored in `T1`, `23` is stored in `T2`, `8` is stored in `T3`, and `4` is stored in `T6`

SableCC
An Example Using SableCC

Solution:

Since we are again dealing with infix expressions, the grammar given in this section may be reused. Simply change the package name to `exprs`.

SableCC
An Example Using SableCC

```
package exprs;
import exprs.analysis.*;
import exprs.node.*;
import java.util.*; // for Hashtable
import java.io.*;
class Translation extends DepthFirstAdapter
{
  // Use a Hashtable to store the memory locations for exprs
  // Any node may be a key, its memory location will be the
  // value, in a (key,value) pair.
  Hashtable hash = new Hashtable();

  public void caseTNumber(TNumber node)
  // Allocate memory loc for this node, and put it into
  // the hash table.
  { hash.put (node, alloc()); }

  public void outATermExpr (ATermExpr node)
  { // Attribute of the expr same as the term
    hash.put (node, hash.get(node.getTerm()));
  }
```

SableCC
An Example Using SableCC

```
public void outAPlusExpr(APlusExpr node)
{ // out of alternative {plus} in Expr, we generate an
  // ADD atom.
  Integer i = alloc();
  hash.put (node, i);
  atom ("ADD", (Integer)hash.get(node.getExpr()),
         (Integer)hash.get(node.getTerm()), i);
}
public void outAMinusExpr(AMinusExpr node)
{ // out of alternative {minus} in Expr,
  // generate a minus atom.
  Integer i = alloc();
  hash.put (node, i);
  atom ("SUB", (Integer)hash.get(node.getExpr()),
         (Integer)hash.get(node.getTerm()), i);
}
public void outAFactorTerm (AFactorTerm node)
{ // Attribute of the term same as the factor
  hash.put (node, hash.get(node.getFactor()));
}
```

SableCC
An Example Using SableCC

```
public void outAMultTerm(AMultTerm node)
{ // out of alternative {mult} in Factor, generate a mult
  // atom.
  Integer i = alloc();
  hash.put (node, i);
  atom ("MUL", (Integer)hash.get(node.getTerm()),
          (Integer) hash.get(node.getFactor()) , i);
}
public void outADivTerm(ADivTerm node)
{ // out of alternative {div} in Factor,
  // generate a div atom.
  Integer i = alloc();
  hash.put (node, i);
  atom ("DIV", (Integer) hash.get(node.getTerm()),
          (Integer) hash.get(node.getFactor()), i);
}
```

SableCC
An Example Using SableCC

```
    public void outANumberFactor (ANumberFactor node)
    { hash.put (node, hash.get (node.getNumber()));
    }

    public void outAParenFactor (AParenFactor node)
    { hash.put (node, hash.get (node.getExpr()));
    }

    void atom (String atomClass, Integer left, Integer right,
               Integer result)
    { System.out.println (atomClass + " T" + left + " T" +
           right + " T" + result);
    }

    static int avail = 0;
    Integer alloc()
    { return new Integer (++avail);
    }
}
```

SableCC
An Example Using SableCC

```
// decaf.grammar
// SableCC grammar for decaf, a subset of Java.
// March 2003, sdb
Package decaf;
Helpers                                              // Examples
  letter = ['a'..'z'] | ['A'..'Z'] ;                 // w
  digit = ['0'..'9'] ;                               // 3
  digits = digit+ ;                                  // 2040099
  exp = ['e' + 'E'] ['+' + '-']? digits;             // E-34
  newline = [10 + 13] ;
  non_star = [[0..0xffff] - '*'];
  non_slash = [[0..0xffff] - '/'];
  non_star_slash = [[0..0xffff] - ['*' + '/']];
```

```
Tokens
  comment1 = '//' [[0..0xffff]-newline]* newline ;
  comment2 = '/*' non_star* '*'
             (non_star_slash non_star* '*'+)* '/' ;
  space = ' ' | 9 | newline ;       // '\t'=9 (tab)
  clas = 'class' ;                  // key words (reserved)
  public = 'public' ;
  static = 'static' ;
  void = 'void' ;
  main = 'main' ;
  string = 'String' ;
  int = 'int' ;
  float = 'float' ;
  for = 'for' ;
  while = 'while' ;
  if = 'if' ;
  else = 'else' ;
  assign = '=' ;
  compare = '==' | '<' | '>' | '<=' | '>=' | '!=' ;
```

```
  plus = '+' ;
  minus = '-' ;
  mult = '*' ;
  div = '/' ;
  l_par = '(' ;
  r_par = ')' ;
  l_brace = '{' ;
  r_brace = '}' ;
  l_bracket = '[' ;
  r_bracket = ']' ;
  comma = ',' ;
  semi = ';' ;
  identifier = letter (letter | digit | '_')* ;
  number = (digits '.'? digits? | '.'digits) exp? ;
                              // Example: 2.043e+5
  misc = [0..0xffff] ;

Ignored Tokens
  comment1, comment2, space;
```

```
Productions
  program = clas identifier l_brace public static
            void main l_par string l_bracket
            r_bracket [arg]: identifier r_par
            compound_stmt r_brace ;

  type = {int} int
       | {float} float ;

  declaration = type identifier identlist* semi;

  identlist = comma identifier ;

  stmt = {dcl} declaration
       | {stmt_no_trlr} stmt_no_trailer
       | {if_st} if_stmt
       | {if_else_st} if_else_stmt
       | {while_st} while_stmt
       | {for_st} for_stmt ;
```

```
stmt_no_short_if = {stmt_no_trlr} stmt_no_trailer
                 | {if_else_no_short} if_else_stmt_no_short_if
                 | {while_no_short} while_stmt_no_short_if
                 | {for_no_short} for_stmt_no_short_if;

stmt_no_trailer = {compound} compound_stmt
                | {null} semi
                | {assign} assign_stmt;

assign_stmt = assign_expr semi;

for_stmt = for l_par  assign_expr? semi bool_expr? [s2]:
semi [a2]: assign_expr? r_par stmt;

for_stmt_no_short_if = for l_par assign_expr? semi
                       bool_expr? [s2]: semi [a2]:
                       assign_expr? r_par
                       stmt_no_short_if;
```

```
while_stmt = while l_par bool_expr r_par stmt;

while_stmt_no_short_if = while l_par bool_expr r_par
                         stmt_no_short_if;

if_stmt = if l_par bool_expr r_par stmt;

if_else_stmt = if l_par bool_expr r_par stmt_no_short_if
               else stmt;

if_else_stmt_no_short_if = if l_par bool_expr r_par
                          [if1]: stmt_no_short_if else
                          [if2]: stmt_no_short_if;
compound_stmt = l_brace stmt* r_brace;

bool_expr = expr compare [right]: expr;

expr = {assn} assign_expr
     | {rval} rvalue;
```

```
 assign_expr = identifier assign expr ;

 rvalue = {plus} rvalue plus term
        | {minus} rvalue minus term
        | {term} term;

 term = {mult} term mult factor
      | {div} term div factor
      | {fac} factor;

 factor = {pars} l_par expr r_par
        | {uplus} plus factor
        | {uminus} minus factor
        | {id} identifier
        | {num} number;
```