Parsing

CSE 530 Design of Compilers

2025

Dar-jen Chang

Table of Contents

1 Introduction to Parsing

2 Top-Down and LL(1) Recursive Descent Parsing

3 Grammar Analysis

4 Coco/R Parsing Functions

5 Handling LL(1) Conflicts

Structure of a Compiler

character stream

v a l = 1 0 * v a l + i

⇩

lexical analysis (scanner)

⇩

token stream

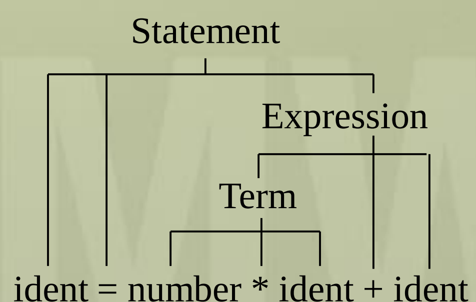
1	3	2	4	1	5	1	token number (type)
(ident)	(assign)	(number)	(times)	(ident)	(plus)	(ident)	
"val"	-	10	-	"val"	-	"i"	token value

⇩

syntax analysis (parser)

⇩

syntax tree



Parsing and PDA

- Given a CFG, parsing is a process (technique or algorithm) to determine if a given string of terminals is a sentence of the CFG and if yes, construct a syntax tree for the sentence.

- (Equivalence of CFG and PDA) Given any CFG, G, there exists a Push-Down Automaton (PDA) M such that L(M) = L(G).

Context-Free Grammars

Example

```
Expr = Term { ( "+" | "-" ) Term }.
Term = Factor { ( "*" | "/" ) Factor }.
Factor = id | "(" Expr ")".
```

indirect central recursion

Context-free grammars can be recognized by ***push-down automata***

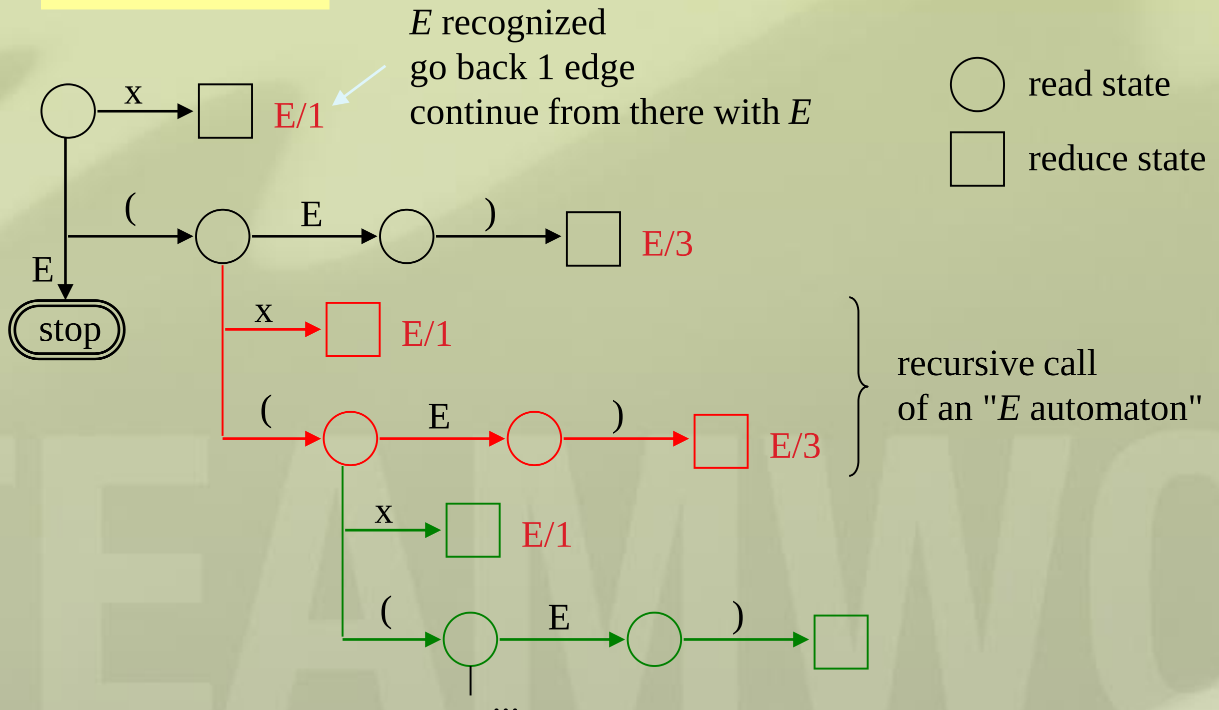
Push-Down Automaton (PDA)

Characteristics

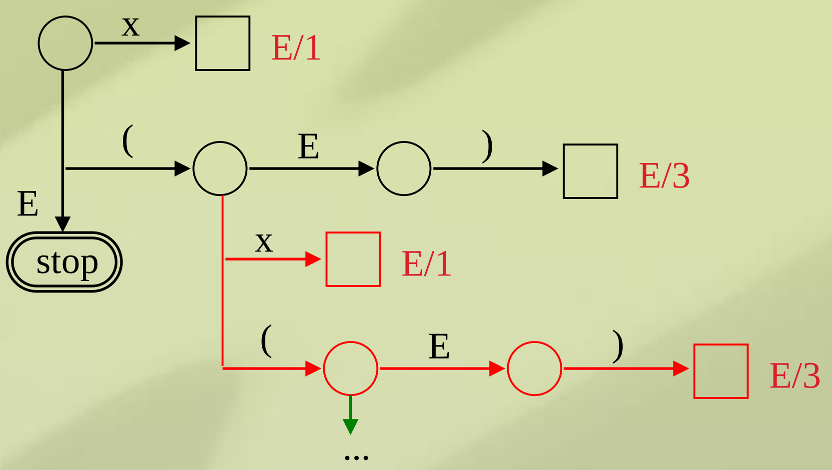
- Allows transitions with terminal symbols <u>and</u> nonterminal symbols
- Uses a stack to remember the visited states

Example

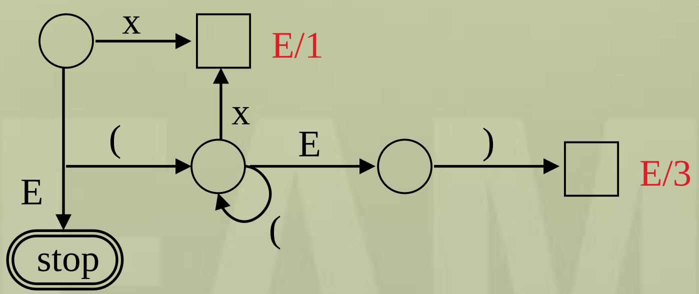
E = x | "(" E ")".



Push-Down Automaton (cont'd)



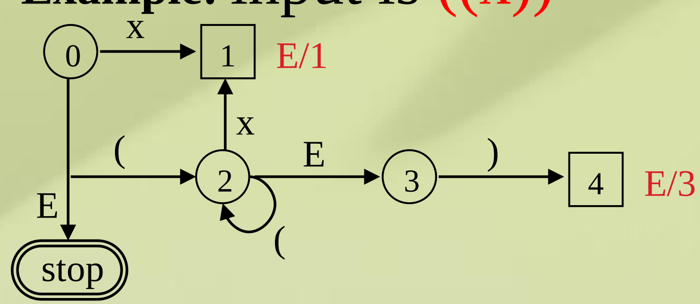
Can be simplified to



Needs a stack in order to find its way back through the visited states

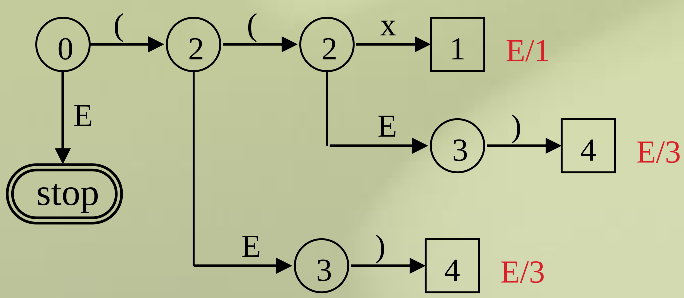
How a PDA Works

Example: input is ((x))



Visited states are stored in a stack

stack	*remaining input*
0 .	((x))
0 2 .	(x))
0 2 2 .	x))
0 2 2 1 .	))
0 2 2 .	E))
0 2 2 3 .	))
0 2 2 3 4 .	)
0 2 .	E)
0 2 3 .	)
0 2 3 4 .	
0 .	E



Limitations of Context-Free Grammars

CFGs cannot express *context conditions*, for example:

- ***Every name must be declared before it is used***
  The declaration belongs to the context of the use; the statement

  ```
  x = 3;
  ```

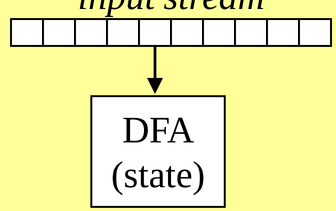
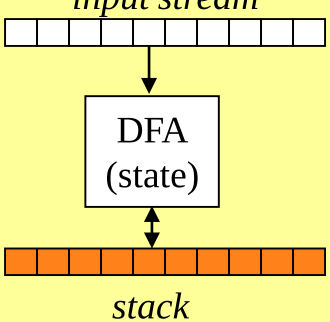
  may be right or wrong, depending on its context

- ***The operands of an expression must have compatible types***
  Types are specified in the declarations, which belong to the context of the use

Possible solutions

- ***Use context-sensitive grammars***
  too complicated
- ***Check context conditions during semantic analysis***
  i.e. the syntax allows sentences for which the context conditions do not hold
  int x = "three"; syntactically correct
  semantically wrong
  The error is detected during semantic analysis (not during syntax analysis).

Regular versus Context-free Grammars

	Regular Grammars	Context-free Grammars
Used for	Scanning	Parsing
Recognized by	**DFA** (no stack) 	**PDA** (stack) 
Productions	A = a | b C.	A = α.
Problems	nested language constructs	context-sensitive constructs (e.g. type checks, ...)

Parsing Algorithms (Techniques)

- General PDA may be nondeterministic which may not be practical for compiler uses.

- Consideration of Parsing Algorithms

  Generality vs. Efficiency

General Parsing Algorithms (Techniques)

- Parsing algorithms work for any CFG

- Examples of general parsing algorithms

  - Earley parsing algorithm (1968)

  - Cocke–Younger–Kasami algorithm (alternatively called CYK, or CKY)

Note: Both can be used for ambiguous CFG.

Earley Parsing Algorithm

- By Jay Earley (1968, 1970)
- Works for any CFG, even ambiguous CFG (i.e. it can find all syntax trees of a sentence in an ambiguous CFG).
- Useful for natural language parsing (note natural languages are inherently ambiguous)
- Not useful for programming language compilers.

CKY Parsing Algorithm

- The algorithm is named after some of its rediscoverers: John Cocke, Daniel Younger, Tadao Kasami, and Jacob T. Schwartz. It employs bottom-up parsing and dynamic programming.
- CYK operates only on context-free grammars given in Chomsky normal form (CNF).
- A CFG is in CNF if every production is either A -> B C or A -> t (A, B, and C are non-terminals; t is terminal).
- Any CFG may be transformed into a CNF grammar expressing the same language.

Practical Parsing Techniques for Compilers

- Two classes of commonly used parsing techniques:

  Top-down (e.g., LL(1), LL(k))
  Bottom-up (e.g., LR(1), LR(k), LALR(1))

Practical Parsing Techniques

- Top-Down Parsing
  For example, Coco/R is an LL(1) top-down recursive descent parser generator (with user–defined LL(1) conflict resolvers).

- Bottom-Up Parsing
  For example, yacc uses LALR(1) bottom-up parsing technique.

Table of Contents

1 Parsing and Push-Down Automata

2 Top-Down and LL(1) Recursive Descent Parsing

3 Grammar Analysis

4 Coco/R Parsing Functions

5 Handling LL(1) Conflicts

Top-Down Parsing

- Top-down syntax tree construction, i.e. from the start symbol to derive the input sentence using left-to-right scan of the input and leftmost derivations.
- Two techniques:

(1) Top-down parsing with backtracking (exhaustive depth-first search), which works for all non left-recursive CFGs.

(2) LL(*k*) using *k* look ahead tokens to guide the derivation (e.g., LL(1) using one look ahead token).

Top-Down with Backtracking Parsing

- To expand the letfmost non-terminal with alternative productions, try the alternatives one by one and backtrack if needed.
- <u>Example</u> Consider the following CFG G_b

[S.1] S = '(' L ')'

[S.2] | 'a' .

[L.1] L = S ',' L

[L.2] | S .

Top-Down with Backtracking Parsing

- Works for non-left-recursive CFGs.
- There are algorithms that can be used to rewrite a left-recursive CFG to an equivalent non-left recursive CFG.
- As a result, the top-down with backtracking parsing can be used to parse any CFG language (like PDA can do).
- But, it is non-deterministic.

Top-Down with Backtracking Parsing (cont'd)

- Difficult to implement: requires all prefixes of the input string (note most scanners only return one token, i.e. the next look ahead token, at a time)
- Backtracking (non-deterministic) is not efficient.
- Alternatives : deterministic top-down parsing techniques

Deterministic Top-Down Parsing Techniques

- Using look-ahead tokens to determine which alternative production should be used (e.g. LL(k)). Note LL(k), for any k, does not work for the grammar G_b given before, but G_b can be rewritten to work with LL(1).
- They only work for a subset of CFGs.

LL(1) Parsing

- From the start symbol to derive the input sentence using Left-to-right scan of the input and Leftmost derivations with one (1) look ahead token.
- LL(1) parsing only works for LL(1) grammars (i.e. not every CFG).
- However, almost all common programming language constructs can be specified in LL(1) grammars (maybe with the help of some kind of LL(1) conflict resolvers like that supported in Coco/R).

LL(1) Parsing (cont'd)

- Two LL(1) parsing implementations
  - Parsing-Table driven
  - Recursive descent recursive functions
- Coco/R generates recursive descent parsers.
- Moreover, Coco/R allows the users to define LL(1) conflict resolvers to extend the expressiveness power of LL(1) grammars (effectively LL(*k*) and more).

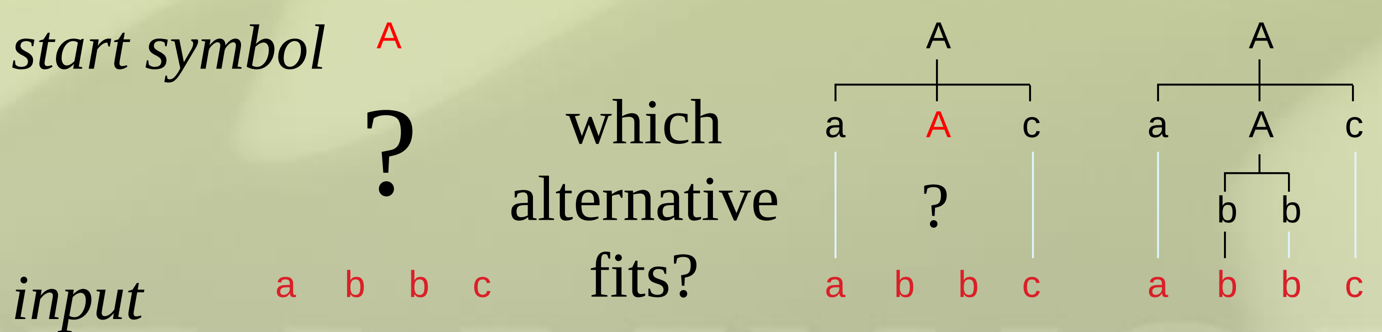
Recursive Descent Parsing

- Top-down parsing technique
- The syntax tree is build from the start symbol to the sentence (top-down)

Example *grammar* *input*

A = a A c | b b.

a b b c



The correct alternative is selected using ...

- the look ahead token from the input stream
- the terminal starting symbols of the alternatives

Recursive Descent Parser Structure

- The parser is a collection of functions, one for each non-terminal symbol in the grammar. The start symbol function is the entry point of the parser.
- Since most grammars are recursive, the resulting functions are also usually recursive, hence the name "recursive descent".

Recursive Descent Parser Structure (cont'd)

- The parser maintains a global variable (e.g. Token la) which contains the next look-ahead token in the input that has not been examined by the parser.
- The body of a parsing function for a non-terminal B is constructed by considering all the (alternative) productions with B on their left-hand side.

Recursive Descent Parser Structure (cont'd)

- Simple demo (A, B, and C non-terminals; t terminal)

A = B t C

Right-hand side of A

```
void A()    { // body of function A

                B(); // call B

                // match token t and get next token
                ......................

                C(); // call C
            }
```

Recursive Descent Parser Structure (cont'd)

- Complications – alternative productions

$$
\begin{aligned}
A \;=\; & \quad \alpha_1 \\
| & \quad \alpha_2 \\
& \ldots\ldots \\
| & \quad \alpha_n \\
. &
\end{aligned}
$$

There are *n* alternative productions of A. Need to determine which of these alternative productions to replace A.

Recursive Descent Parser Structure (cont'd)

- The parser decide which alternative production should be used by examining the look-ahead token (i.e., LL(1)).
- A non-terminal on the right-hand side turns into a call to the parsing function for that non-terminal.
- A token on the right-hand side turns into a test to make sure that the current look-ahead token matches the token. If they match, the parser calls the scanner to get the next token from the input. Otherwise a syntax error is detected.

Recursive Descent Parser Example

- Consider this CFG :

```
IdentList = ident (. n++ .) L.
L         = comma ident (. n++ .) L
          |
          .
```

- The scanner is generated by Coco/R:

```
ident = letter { letter | digit }.
comma = ',' .
```

Recursive Descent Parser Example (cont'd)

- Parser Implementation

```
public class Parser
{
    Scanner scanner;
    Token la;        // Look-ahead token, i.e. next token to be examed
    int n = 0;

public Parser(Scanner scan)
{
        scanner = scan;
        la = scanner.Scan();
}
```

Recursive Descent Parser Example (cont'd)

- Parser Implementation

```
public void Parse()
{
    if (IdenList())
        Console.WriteLine ("There are {0} idents.", n);
    else Console.WriteLine ("Invalid input!");
}
bool IdentList()     // IdentList =   ident L.
{
    if (la.kind == 1)          // ident
    {
        n++;
        la = scanner.Scan();
        return L();
    }
    else return false;         // syntax error; expecting ident
}
```

Recursive Descent Parser Example (cont'd)

Parser Implementation

```
public bool L()      // L  =   comma ident L  | .
{
    if (la.kind == 2) // comma (',')
    {
            la = scanner.Scan ();
            if (la.kind == 1) // ident
            {
                    n++;
                    la = scanner.Scan ();
                    return L();
            }
            else return false; // syntax error; expecting ident
    }
    else if (la.kind == 0) return true; // EOF

    return false; // syntax error: expecting ident or EOF
}
```

Recursive Descent Parser Example (cont'd)

Driver (i.e. Main function) Implementation

```
string input;
Console.Write("Enter ident list > ");

input = Console.ReadLine();

ASCIIEncoding en = new ASCIIEncoding();
byte[] inputBytes = en.GetBytes(input);
Scanner scanner = new Scanner(new MemoryStream(inputBytes));

// Use our parser
Parser parser = new Parser(scanner);

parser.Parse();
```

LL(1) Recursive Descent Parser Theory

- In a leftmost derivation, we need to expand the leftmost non-terminal A.
- Suppose A has these alternative productions:

$$
\begin{array}{lll}
A = & & \alpha_1 \\
 & | & \alpha_2 \\
 & \ldots\ldots & \\
 & | & \alpha_n \\
 & . &
\end{array}
$$

Which alternative production is the right one to expand (rewrite) A in the leftmost derivation?

LL(1) Recursive Descent Parser Theory (cont'd)

- Idea : using Select sets of productions to select one from several alternative productions to expand the leftmost non-terminal.

- The select set of an alternative production is a set of terminals which are used to match the look ahead token to determine if this alternative production is the right one to use.

Select Sets

- Example (which alternative should be used to expand A?)

```
A =   'b'  B        // Select set of  this alternative = { b }
    |
      'c'  C        // Select set of  this alternative = { c }
```

```
void A()
{
    if (la.kind == b)
    {
        Get(); // Get next token
        B();
    } else if (la.kind == c)
    {
        Get();
        C();
    } else SynErr();
}
```

Select Sets and LL(1)

- Example

```
S =    A 'a' .
A =    'a'  B        // Select set of  this alternative = { a }
     |
       C 'd' D       // Select set of  this alternative = { c, d }
     .
B =    'b' B    |   .
C =    'c'  C   |   .
D =    'd' D    |  .
```

- LL(1) parsing works for this CFG

Select Sets and LL(1) (cont'd)

- Example (simple modification of the last example)

```
S =    A 'a' .
A =    'a'  B        // Select set of  this alternative = { a }
       |
       C  D a        // Select set of  this alternative = { a, c, d }
       .
B =    'b' B   |  .
C =    'c'  C   |  .
D =    'd' D   |   .
```

- LL(1) parsing does not work for this CFG. Why?
- The problem is called LL(1) Coflicts

Table of Contents

1 Introduction to Parsing

2 Top-Down and LL(1) Recursive Descent Parsing

3 Grammar Analysis

4 Coco Parsing Functions

5 Handling LL(1) Conflicts

Grammar Analysis

- Analyze the properties of a given CFG.
- Remove unreachable symbols and useless productions.
- Determine what nonterminals can derive ε. These nonterminals are called deletable (or nullable). Since such nonterminals may disappear during a parse process, they must be handled carefully.

Grammar Analysis (cont'd)

- Compute FIRST sets. FIRST(α), where α is a string of symbols, is the set of terminals that can begin a string derivable from α.
- Compute FOLLOW sets. FOLLOW(A), where A is a nonterminal, is the set of terminals that may follow A in some sentential form (a string of terminal or non-terminal symbols that can be derived from the start symbol).

Grammar Analysis (cont'd)

- Compute SELECT sets of productions and find LL(1) conflicts. We will define SELECT sets and LL(1) conflicts later on.
- Modify (or transform) grammar to eliminate undesirable grammar properties. For example, G1 -> G2 -> G3 and so on such that L(G1) = L(G2) = L(G3) = …, and each step of grammar modification will remove some undesirable properties like left-recursion and LL(1) conflicts.

Unreachable nonterminals and terminals

- Example unreachable (useless) symbols

```
S  =  A  .
A  =  t  .
B  =  s  .
If S is the start symbol, clearly we
cannot reach B and s.
```

- Remove such useless symbols, how?

Useless Productions

- Example

```
S   = A   .
A   = t   .
B   = s   .
```

S is the start symbol. The last production is not useful. If a nonterminal can not be reached from the start symbol, any production with the nonterminal on its left-hand side also is useless.

- Algorithms?

Deletables (nullables)

- A non-terminal A is deletable (or nullable) if A can derive ε (empty string), i.e. A =>* ε. A terminal is always non-nullable. A string is nullable if every symbol in the string is nullable

<u>Example</u>

```
S =      A 'a' .
A =     'a' B    |  C  D  .
B =     'b' B    |    .
C =     'c'  C   |    .
D =     'd' D   |    .
```

-- Non-terminals A, B, C, and D are deletable.

-- The string CD is deletable.

First Sets

- Given a string α consisting of non-terminals and terminals in a CFG, First(α) is the set of the leading terminal (token) of any string that can be derived from α.
- Example:

```
S =    A 'a' .

A =   'a' B   | C  D
B =   'b' B   |  .
C =   'c'  C   |  .
D =   'd' D   |  .
```

```
--  First (S) = { a, c, d} since
            S => A a => a B
            S => A a => C D a => c C D a
            S => A a => C D a => D a => d D a
-- First (CD) = { c, d } since
            C D => c C D
            C D => D => d D
```

Follow Sets

- Given a non-terminal, N, in a CFG, Follow(N) is the set of all terminals (tokens) that follow B in some string that can be derived from the start symbol S. EOF always belongs to Follow(S).
- Example:

```
S =   A 'a' .
A =   'a' B   | C  D .
B =   'b' B   |  .
C =   'c'  C  |  .
D =   'd' D   |  .
```

-- Follow(S) = { EOF }
-- Follow(A) = { a } since
    S => A a   (a follows A)
-- Follow(B) = Follow(D) = { a } since
    S => A a => a B a and S => A a => C D a
-- Follow(C) = { a, d } since
    S => A a => C D a => C a and S => A a => C D a = > C d D a

Select Sets (Cont‘d)

- Given a production $A \rightarrow \alpha$, the select set of the production, denoted by Select($A\rightarrow\alpha$), is defined as follows:

  (1) if α is not deletable,

  $$\text{Select}(A\rightarrow\alpha) = \text{First}(\alpha)$$

  (2) if α is deletable,

  $$\text{Select}(A\rightarrow\alpha) = \text{First}(\alpha) \cup \text{Follow}(A)$$

Select Sets (Cont‘d)

<u>Example</u>

S = A 'a' .

A = 'a' B | C D .
B = 'b' B | .
C = 'c' C | .
D = 'd' D | .

Consider the alternative productions of A.

(1) A → 'a' B

Since the RHS (‘a’ B) is not deletable, the Select set of this alternative is First (‘a’ B) = { a }.

(2) A → C D

Since the RHS (C D) the Select set of this alternative is First (C D) ∪ Follow(A) = { c, d } ∪ { a } = { a, c, d}

Select Sets and LL(1) Conflicts

Consider all alternative productions of A :

$$
\begin{aligned}
A \quad = \quad & \quad \alpha_1 \\
& \mid \ \alpha_2 \\
& \mid \ \alpha_3 \\
& \text{----------} \\
& \mid \ \alpha_n
\end{aligned}
$$

If the next look ahead token *t* is in Select(A→ α_i), use the *i*-th alternative to expand A.

<u>Problem</u> The select sets of all alternative productions of A may not be disjoint.

LL(1) Conflicts Defined

For any two alternative productions of A :

A = α
| β

If Select(A→ α) ∩ Select(A→ β) ≠ ø (empty set), then there is an LL(1) conflict between these two alternative productions.

In case of LL(1) conflicts, coco parser generator will use the first listed production when the look-up token causes the conflict. The correctness of the choice is not predictable.

LL(1) Grammar Defined

A CFG, G, is an LL(1) grammar, if there are no LL(1) conflicts in the productions of G.

Note: When coco/r detects LL(1) conflicts from the input grammar, it will only give LL(1) conflicts as warnning messages, not errors, and continue to generate parser. As a result, it is the user's responsiblity to check the LL(1) conflict warning messages to make sure those conflicts are accepable.

Table of Contents

1 Introduction to Parsing

2 Top-Down and LL(1) Recursive Descent Parsing

3 Grammar Analysis

4 Coco/R Parsing Functions

5 Handling LL(1) Conflicts

TokenVariables of the Parser

Most recently recognized token and lookahead token

The parser has two tokens variables:

```
Token t;          // Most recently recognized token
Token la;         // Lookahead token (still unrecognized)
```

The parser call the scanner to get the next token from the input stream

	t	la				
token stream	ident	assign	ident	plus	ident

already recognized

The scanner is called at the beginning of parsing to get the first token in la.

How to Parse Terminal Symbols

Match Input token with the expected token

Input token: la
parsing action: Expect(int n);

```
void Expect (int n)
{
  if (la.kind == n) Get();  // recognized => get the next token
  else SynErr(n);
}
```

The names of the terminal symbols are declared as constants (in the Parser class)

```
public const int _EOF = 0,
                 _IDENT = 1, _NUMBER = 2, ...,
                 _PLUS = 4, _MINUS = 5, ... ;
```

How to Parse Nonterminal Symbols

Nonterminal -> Parsing Method

symbol to be parsed: A
parsing method: A(); // call of the parsing method A

Every nonterminal symbol is recognized by a parsing method with the same name

```
private void A()
{
    ... parsing actions for the right-hand side of A ...
}
```

How to Parse Sequences

production: A = a B c.

parsing method:

```
void A () {
    // la contains a terminal start symbol of A
    Expect(a);
    B();
    Expect(c);
    // la contains a follower of A
}
```

Simulation

A = a B c.
B = b b.

	remaining input
void **A** () {	a b b c
Expect(a);	b b c
B();	c
Expect(c);	
}	
void B() {	b b c
Expect(b);	b c
Expect(b);	c
}	

How to Parse Alternatives

Pattern $\alpha \mid \beta \mid \gamma$ α, β, γ are arbitrary non-deletable strings

Parsing action

```
if (la in First(α)) { ... parse α ... }
else if (la in First(β)) { ... parse β ... }
else if (la in First(γ)) { ... parse γ ... }
else Error("..."); // find a meaninful error message
```

Example

```
A = a B | B b.
B = c | d.
```

First(aB) = {a}
First(Bb) = First(B) = {c, d}

```
void A () {
    if (la == a) {
        Expect(a);
        B();
    } else if (la == c || la == d) {
        B();
        Expect(b);
    } else Error ("invalid start of A");
}
```

```
static void B () {
    if (la == c) Expect(c);
    else if (la == d) Expect(d);
    else Error ("invalid start of B");
}
```

examples:parse a d and c b
parse b b

How to Parse EBNF Options

Pattern [α] α is an arbitrary EBNF expression

Parsing action

```
if (la in First(α)) { ... parse α ... }  // no error branch!
```

Example

```
A = [ a b ] c.
```

```
void A () {
   if (la == a) {
      Expect(a);
      Expect(b);
   }
   Expect(c);
}
```

Example: parse a b c

parse c

How to Parse EBNF Iterations

Pattern { α } α is an arbitrary EBNF expression

Parsing action

```
while (la in First(α)) { ... parse α ... }
```

Example

```
A = a { B } b.
B = c | d.
```

```
void A () {
    Expect(a);
    while (la == c || la == d) B();
    Expect(b);
}
```

alternatively ...

```
void A () {
    Expect(a);
    while (la != b && la != Token.EOF) B();
    Expect(b);
}
```

without EOF: danger of an infinite loop, if *b* is missing in the input

Example: parse a c d c b

parse a b

Optimizations

Avoiding multiple checks

A = a | b.

```
void A () {
   if (la == a) Get();  // no Expect(a);
   else if (la == b) Get();
   else Error("invalid A");
}
```

A = { a | B d }.
B = b | c.

```
void A () {
   while (la == a || la == b || la == c) {
      if (la == a) Get();
      else { // no Expect any more
         B(); Check(d);
      } // no error case
   }
}
```

More efficient scheme for parsing alternatives in an iteration

A = { a | B d }.

```
void A () {
   for (;;) {
      if (la == a) Get();
      else if (la == b || la == c) { B(); Expect(d); }
      else break;
   }
}
```

Optimizations

Frequent iteration pattern

α { separator α }

```
for (;;) {
    ... parse α ...
    if (la == separator) Get(); else break;
}
```

Example

ident { "," ident }

```
for (;;) {
    Expect(ident);
    if (la == Token.COMMA) Get(); else break;
}
```

input e.g.: a , b , c :

Computing Terminal Starting Symbols Correctly

Grammar

terminal start symbols

```
A = B a.
B = { b } c
  |  [ d ]
  |  e.
```

b and *c*

d and *a* (!)

e

```
C = D e
  |  f.
D = { d }.
```

d and *e* (*D* is deletable!)

f

Parsing methods

```
static void A () {
    B(); Expect(a);
}
```

```
static void B () {
    if (la == b || la == c) {
        while (la == b) Get();
        Expect(c);
    } else if (la == d || la == a) {
        if (la == d) Get();
    } else if (la == e) {
        Get();
    } else Error("invalid B");
}
```

```
static void C () {
    if (la == d || la == e) {
        D(); Expect(e);
    } else if (la == f) {
        Get();
    } else Error("invalid C");
}
```

```
static void D () {
    while (la == d) Get();
}
```

Table of Contents

1 Introduction to Parsing

2 Top-Down and LL(1) Recursive Descent Parsing

3 Grammar Analysis

4 Coco/R Parsing Functions

5 Handling LL(1) Conflicts

LL(1) Property

LL(1) ... can be analyzed from **L**eft to right
with **L**eft-canonical derivations (leftmost NTS is derived first)
and **1** lookahead symbol

Definition

1. A grammar is LL(1) if all its productions are LL(1).
2. A production is LL(1) if for all its alternatives in the production

$$\alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

the following condition holds:

$$\text{Select}(\alpha_i) \cap \text{Select}(\alpha_j) = \{\} \quad (\text{for any } i \text{ and } j)$$

If this condition doesn't hold, the production is said to have LL(1) conflict(s).

In other words

- The select sets of all alternatives of a production must be pairwise disjoint.
- The parser must always be able to select one of the alternatives by looking at the look ahead token.

How to Remove LL(1) Conflicts

Factorization

```
IfStatement =  "if" "(" Expr ")" Statement
            |  "if" "(" Expr ")" Statement "else" Statement.
```

Extract common start sequences

```
IfStatement =  "if" "(" Expr ")" Statement  (   |  "else" Statement  ).
```

… or in EBNF

```
IfStatement =  "if" "(" Expr ")" Statement [ "else" Statement ].
```

Sometimes nonterminal symbols must be inlined before factorization

```
Statement   =  Designator "=" Expr ";"
            |  ident "(" [ ActualParameters ] ")" ";".
Designator  =  ident { "." ident }.
```

Inline *Designator* in *Statement*

```
Statement   =  ident { "." ident } "=" Expr ";"
            |  ident "(" [ ActualParameters ] ")" ";".
```

then factorize

```
Statement   =  ident  (  { "." ident } "=" Expr ";"
                      |  "(" [ ActualParameters ] ")" ";"
                      ).
```

How to Remove Left Recursion

Left recursion is always an LL(1) conflict

For example

IdentList = ident | IdentList "," ident.

generates the following phrases

ident
ident "," ident
ident "," ident "," ident
...

can always be replaced by iteration

IdentList = ident { "," ident }.

Note: compare syntax trees of these two grammars (yellow and orange).

Hidden LL(1) Conflicts

EBNF options and iterations are hidden alternatives

A = [α] β. $\leftarrow\rightarrow$ A = α β | β. Where α and β are arbitrary EBNF expressions

Rules

A = [α] β.	First(α) $\cap$ First(β) must be {}
A = { α } β.	First(α) $\cap$ First(β) must be {}

A = α [β].	First(β) $\cap$ Follow(A) must be {}
A = α { β }.	First(β) $\cap$ Follow(A) must be {}

A = α | .	First(α) $\cap$ Follow(A) must be {}

Removing Hidden LL(1) Conflicts

Name = [ident "."] ident.

Where is the conflict and how can it be removed?

Name = ident ["." ident].

Is this production LL(1) now?

We have to check if First("." ident) ∩ Follow(Name) = {}

Prog = Declarations ";" Statements.
Declarations = D { ";" D }.

Where is the conflict and how can it be removed?

Inline *Declarations* in *Prog*

Prog = D { ";" D } ";" Statements.

First(";" D) ∩ First(";" Statements) ≠ {}

Prog = D ";" { D ";" } Statements.

We still have to check if First(D ";") ∩ First(Statements) = {}

Dangling Else

If statement

```
Statement =  "if" "(" Expr ")" Statement [ "else" Statement ]
          |  ... .
```

This is an LL(1) conflict!

First("else" Statement) ∩ Follow(Statement) = {"else"}

It is even an ambiguity which cannot be removed

Statement

Statement

if (expr1) if (expr2) stat1; else stat2;

Statement

Statement

We can build 2 different syntax trees!

Can We Ignore LL(1) Conflicts?

An LL(1) conflict is only a warning

The parser selects the first matching alternative

```
A = a b c
  | a d.
```

if the lookahead token is *a* the parser selects this alternative

Example: Dangling Else

```
Statement = "if" "(" Expr ")" Statement [ "else" Statement ]
          | ... .
```

If the lookahead token is "else" here
the parser starts parsing the option;
i.e. the "else" belongs to the innermost "if"

$$\underbrace{\text{if (expr1) }\underbrace{\text{if (expr2) stat1; else stat2;}}_{\text{Statement}}}_{\text{Statement}}$$

Luckily this is what we want here.

Other Requirements for a Grammar (Preconditions for Parsing)

Completeness

For every NTS there must be a production

```
A = a B C .
B = b b .
```

error!
no production for *C*

Derivability

Every NTS must be derivable (directly or indirectly) into a string of terminal symbols

```
A = a B | c .
B = b B .
```

error!
B cannot be derived into a string of terminal symbols

Non-circularity

A NTS must not be derivable (directly or indirectly) into itself ($A \Rightarrow B_1 \Rightarrow B_2 \Rightarrow \ldots \Rightarrow A$)

```
A = a b | B .
B = b b | A .
```

error!
this grammar is circular because of $A \Rightarrow B \Rightarrow A$