TDDD55 Compilers and interpreters

TDDB44 Compiler Construction

Syntax Analysis, Parsing

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

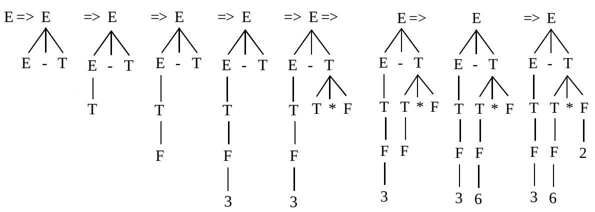
Parser

- A parser for a CFG (*Context-Free Grammar*) is a program which determines whether a string w is part of the language L(G).
- **Function**
  - Produces a parse tree if w ∈ L(G).
  - Calls semantic routines.
  - Manages syntax errors, generates error messages.
- **Input:**
  - String (finite sequence of tokens)
  - Input is read from left to right.
- **Output:**
  - Parse tree / error messages

Top-Down Parsing

- Example: **Top-down** parsing with input: 3 - 6 * 2

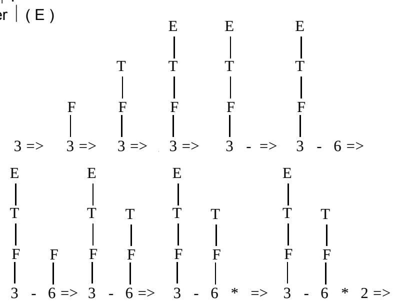
E → E - T | T
T → T * F | F
F → Integer | (E)



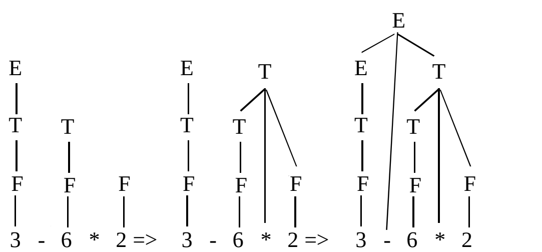
Bottom-up Parsing

- Example: **Bottom-up** parsing with input: 3 - 6 * 2

E → E - T | T (same CFG as in previous example)
T → T * F | F
F → Integer | (E)



Bottom-up Parsing cont.

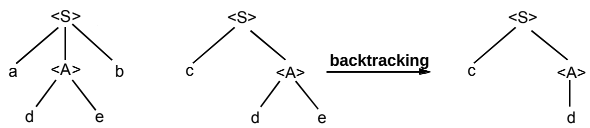


Top-Down Analysis

- How do we know in which order the string is to be derived?
  - Use one or more tokens lookahead.
- Example: **Top-down analysis with backtracking**

<S> → a <A> b *1 token lookahead works well*
| c <A> *1 token lookahead works well*
<A> → d e *test right side until something*
| d *fits*

- a) adeb b) cd



Top-down Analys with Backtracking, cont.

- Top-down analys with backtracking is implemented by writing a **procedure or a function for each nonterminal** whose task is to find one of its right sides:

```
bool A() {  /* A → d e | d */
  char* savep;
  savep = inpptr;

  if (*inpptr == 'd') {
    scan();  /* Get next token, move inpptr a step */
    if (*inpptr == 'e') {
      scan();
      return true;   /* 'de' found */
    }
  }
  inpptr = savep;
  /* 'de' not found, backtrack and try 'd'*/
  if (*inpptr == 'd') {
    scan(); return true;   /* 'd' found, OK */
  }
  return false;
}
```


Top-down Analys with Backtracking, cont.

```
bool S() {  /* S -> a A b | c A */
  if (*inpptr == 'a') {
    scan();
    if A() {
      if (*inpptr == 'b') {
        scan();
        return true;
      } else return false;
    } else return false;
  }
  else if (*inpptr == 'c') {
    scan();
    if A() return true; else return false;
  }
  else return false;
}
```


Construction of a top-down parser

- Write a procedure for each nonterminal.
- Call scan directly *after each token is consumed*.
  - Reason: The look-ahead token should be available
- Start by calling the procedure for the start symbol.

At each step check the leftmost non-treated vocabulary symbol.

- If it is a *terminal symbol*
  - Match it with the current token, and read the next token.
- If it is a *nonterminal symbol*
  - Call the routine for this nonterminal.
- In case of error call the error management routine.

Example: An LL(1) grammar which describes binary numbers

S→ BinaryDigit BinaryNumber

BinaryNumber→ BinaryDigit BinaryNumber

| ε

BinaryDigit→ 0 | 1

Sketch of a Top-Down Parser (recursive descent)

```
void TopDown(input,output) {

/* main program */
  scan();
  S();
  if not eof then error(...);
}
```

Grammar:

S→ BinaryDigit BinaryNumber
BinaryNumber→ BinaryDigit BinaryNumber
| ε
BinaryDigit→ 0 | 1

```
void BinaryDigit()
{
  if (token==0 || token==1) scan();
  else error(...);
} /* BinaryDigit */

void BinaryNumber()
{
  if (token==0 || token==1)
  {
    BinaryDigit();
    BinaryNumber();
  } /* OK for the case with ε */
} /* B' */

void S()
{
  BinaryDigit();
  BinaryNumber();
} /* S */
```


A Top-Down Parser that does not Work, Infinite Recursion:

```
void TopDown(input,output)
{

/* main program */
  scan();
  S();
  if not eof then error(...);
}
```

Grammar:

S→ BinaryDigit BinaryNumber
BinaryNumber→ BinaryNumber BinaryDigit
| ε
BinaryDigit→ 0 | 1

```
void BinaryDigit()
{
  if (token==0 || token==1) scan();
  else error(...);
} /* BinaryDigit */

void BinaryNumber()
{
  if (token==0 || token==1)
  {
    BinaryNumber(); /*Infinite Recursion here */
    BinaryDigit();
  } /* OK for the case with ε */
} /* B' */

void S()
{
  BinaryDigit();
  BinaryNumber();
} /* S */
```

Non-LL(1) Structures in a Grammar:

- Left recursion, example:

  E → E - T

  | T

- Productions for a nonterminal with the same prefix in two or more right-hand sides, example:

  arglist → ( )

  | (args)

  or

  A → a b

  | a c

- The problem can be solved in most cases by rewriting the grammar to an LL(1) grammar

Convert a grammar for top-down parsing?
1. Eliminate left recursion
a) Transform the grammar to iterative form

EBNF (*Extended BNF) Notation:*

- {β} same as the regular expression: β*
- [β] same as the regular expression: β | ε
- ( ) left factoring, e.g. A → ab | ac in EBNF is rewritten: A → a (b | c)

Transform the grammar to be iterative using EBNF

- A → A α | β (where β may not be preceded by A)

in EBNF is rewritten:

- A → β {α}

1b) Transform the Grammar to Right Recursive Form Using a Rewrite Rule:

- A → A α | β (where β may not be preceded by A)

is rewritten to

A → β A'

A' → α A' | ε

Generally:

- A → A α1 | A α2 | ... | A αm | β1 | β2 | ... | βn (where β1, β2, ... may not be preceded by A)

is rewritten to:

A → β1 A' | β2 A' | ... | βn A'

A'→ α1 A' | α2 A' | ... | αm A' | ε

2. Left Factoring Using ( ) or [ ]

Original Grammar:

<stmt> → if <expr> then <stmt>

| if <expr> then <stmt> else <stmt>

Solution using EBNF:

<stmt> → if <expr> then <stmt>
[else <stmt>]

Solution using rewriting:

<stmt> → if <expr> then <stmt> <rest-if>

<rest_if> → else <stmt> | ε

Original Grammar

- A → ab | ac

Solution using EBNF:

- A → a (b | c)

Summary LL(1) and Recursive Descent

Summary of the LL(1) grammar:

- Many CFGs are not LL(1)
- Some can be rewritten to LL(1)
- - The underlying structure is lost (because of rewriting).

Two main methods for writing a top-down parser

- Table-driven, LL(1)
- Recursive descent

LL(1)	Recursive Descent
Table-driven	Hand-written
+ fast	- Much coding, + fast
+ Good error management and restart	+ Easy to include semantic actions; good error mgmt

Small Rewriting Grammar Exercise

Example: A recursive Descent Parser for Pascal Declarations, Orig. Grammar

<declarations> → <constdecl> <vardecl>
<constdecl> → CONST <consdeflist>
| ε
<consdeflist> → <consdeflist> <constdef>
| <constdef>
<constdef> → id = number ;
<vardecl> → VAR <vardeflist>
| ε
<vardeflist> → <vardeflist> <idlist> : <type> ;
| <idlist> : <type> ;
<idlist> → <idlist> , id
| id
<type> → integer
| real

Rewrite in EBNF so that a Recursive Descent Parser can be Written

<declarations> → <constdecl> <vardecl>
<constdecl> → CONST <consdef> { <consdef> }
| ε
<constdef> → id = number ;
<vardecl> → VAR <vardef> { <vardef> }
| ε
<vardef> → id { , id } : (integer | real) ;

A Recursive Descent Parser for the New Pascal Declarations Grammar in EBNF

- We have one character lookahead.
- scan should be called when we have consumed a character.

```
void declarations()
/*<declarations>→<constdecl><vardecl> */
{
  constdecl();
  vardecl();
} /* declarations */
```

```
void constdecl()
/* <constdecl> → CONST <consdef>
{ <consdef> }
| ε */
{
    if (token == CONST) {
      scan();
      If (token == id)
        constdef();
      else
        error("Missing id after CONST");

     while (token == id) constdef();
    }
} /* constdecl */
```


Pascal Declarations Parser cont 1

```
void constdef()
/* <constdef> → id = number ; */
{
  scan(); /* consume ID, get next token */
  if (token == '=')
    scan();
  else
    error("Missing '=' after id");
  if (token == NUMBER) then
    scan();
  else
    error("Missing number");

  if (token == ';')
    scan(); /* consume ';', get next token */
  else
    error("Missing ';' after const decl");
} /* constdef */
```

```
void vardecl()
/* <vardecl> → VAR <vardef> { <vardef> }
|ε */
{
  If (token == VAR) {
    scan();

    if (token == ID)
      vardef();
    else
      error("Missing id after VAR");

    while (token == ID) {
      vardef();
    }
  }
} /* vardecl */
```


Pascal Declarations Parser cont 2

```
void vardef() /* <vardef> → id { , id } : ( integer | real ) ; */
{
  scan();
  while (token == ',') {
    scan();
  if (token == ID)
    scan();
  else error("id expected after ',' ");
  } /* while */

  if (token == ':') {
    scan();
    if ((token == INTEGER) || (token == REAL))
      scan();
    else error("Incorrect type of variable");
    if (token ==';')
      scan();
    else error("Missing ';' in variable decl.");

  } else error("Missing ':' in var. decl.");
} /* vardef */
```

```
{ /* main */
  scan(); /* lookahead token */
  declarations();
  if (token!=eof_token) then error(...);
} /* main */
```


TDDD55 Compilers and interpreters
TDDB44 Compiler Construction

LL Parsing Issues Beyond Recursive Descent

LL(*k*)
LL items
Finite pushdown automaton
FIRST and FOLLOW
Table-driven Predictive Parser

Peter Fritzson, Christoph Kessler,
IDA, Linköpings universitet, 2011.

LL(*k*)

- Given:
  - Context-free grammar G = (N, Σ, P, S)
  - Integer $k > 0$
- G is (in) **LL(*k*)** if:

  for any two leftmost derivations
  - $S \Rightarrow^*_{lm} uY\alpha \Rightarrow u\beta\alpha \Rightarrow^* ux$ and
  - $S \Rightarrow^*_{lm} uY\alpha \Rightarrow u\gamma\alpha \Rightarrow^* uy$

  with $x[1{:}k] = y[1{:}k]$ — the *k* first tokens of *x* and *y* are equal

  it holds $\beta = \gamma$.
- That is, for fixed left context *u*, the choice for the "right" production to apply to Y is uniquely determined by the next *k* input tokens.

Example

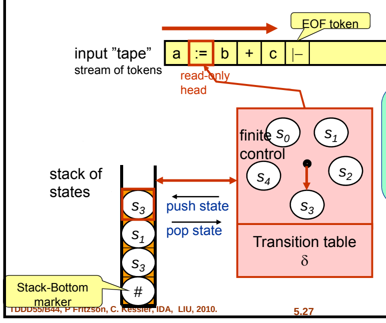
- The following grammar is LL(1) (terminals are bold-face):

  S -> **if** **ident** **then** S **else** S **fi**
  | **while** **ident** **do** S **od**
  | **begin** S **end**
  | **ident** := **ident**

Automaton Model for Parsing Context-Free Languages

Finite pushdown automaton (FPA)

- a finite automaton with a stack of states



Grammar G is LL(1) => there exists a finite pushdown automaton recognizing L(G) where δ is a function (*i.e.*, a *deterministic* pushdown automaton)

Transitions in δ are tuples
((current state, input symbol, top stack element), (new state, read action, stack action))

Context-Free Items

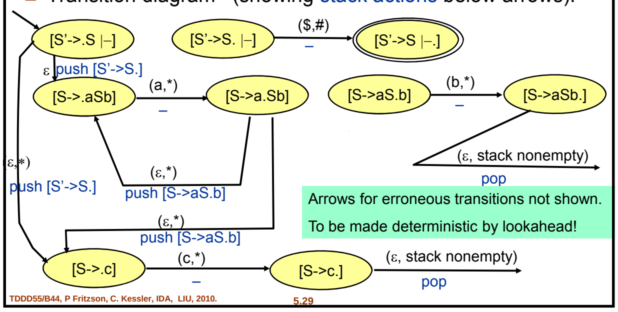
Given CFG G, construct states of the finite pushdown automaton:

- Add new start symbol S' with S' –> S |– (|– means End-of-Input)
- For each production $A \to \alpha_1 \ldots \alpha_k$ e.g. A -> aBc

  create k+1 **context-free items** (= states)
  - e.g., [A->.aBc], [A->a.Bc], [A->aB.c], [A->aBc.]
- Construct a **predictive parser** as finite pushdown automaton:
  - start in state [S'->.S |–] with empty stack (#)
  - halt and accept in state [S'->S |–.] with empty stack (#)
  - at [A->α.bγ]: read input symbol, i.e., [A->α.bγ] –> [A->αb.γ]
  - at [A->α.Bγ]: push [A->αB.γ], determine new production B–>β and start from [B->.β] — Prediction!
  - at [B->β.]: pop state [A->αB.γ] to restore context (if #, error)

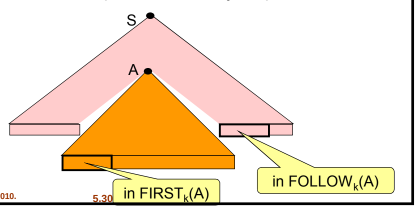
Example

- Grammar with productions { S –> aSb | c }
- Add new start symbol S': { S' –> S; S –> aSb; S–>c }
- Transition diagram (showing stack actions below arrows):



FIRST and FOLLOW

- For a sentential form α in $(N \cup S)^+$, FIRST(α) denotes the set of all terminals with can be ***first*** in a string derived from α.
- For a nonterminal A in N, FOLLOW(A) denotes the set of all terminals (e.g. a) that could appear immediately ***after*** A in a sentential form i.e., there exists $S \Rightarrow^* \alpha A a \beta$ for arbitrary α, β



Small FIRST and FOLLOW Exercise

Computing FIRST = $FIRST_1$

For all grammar symbols X:

- If X is a terminal, then FIRST(X) = { X }.
- If X → ε is a production, then add ε to FIRST(X).
- If X is a nonterminal and X → Y_1 Y_2 ... Y_q is a production,
  - then place all those a of Σ in FIRST(X) where for some i, a is in FIRST(Y_i) and ε is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}) (that is, Y_1, ..., Y_{i-1} all may derive ε).
  - If ε is in FIRST(Y_j) for *all* j=1,2,...,q then add ε to FIRST(X).

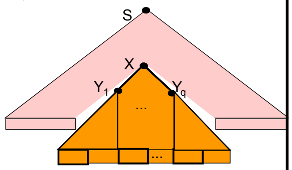
Apply these rules until no more terminals or ε can be added to any FIRST set.

For the example grammar
S' -> S; S -> aSb; S->c

FIRST(a) = {a}, FIRST(b) = {b}, FIRST(c) = {c}

FIRST(S') = FIRST(S)

FIRST(S) = { a, c }



Computing FIRST (cont.)

For any string X_1 X_2 ... X_n of grammar symbols:

- Add to FIRST(X_1 X_2 ... X_n) all non-ε symbols of FIRST(X_1).
- If ε in FIRST(X1), add also all non-ε symbols of FIRST(X_2), otherwise done.
- If ε also in FIRST(X_2), add also all non-ε symbols of FIRST(X_3), otherwise done.
- ...
- If ε also in FIRST(X_n), add ε to FIRST(X_1 X_2 ... X_n)

For the example grammar
S' -> S; S -> aSb; S->c

FIRST(abc) = {a}

FIRST(Sb) = FIRST(S) = {a,c}

Computing FOLLOW

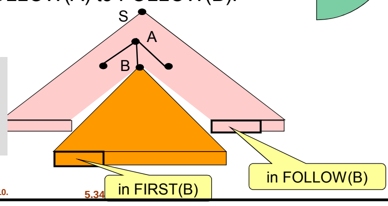
Compute FOLLOW(B) for each nonterminal B:

- Add |– to FOLLOW(S)
- If there is a production A –>* αBβ for arbitrary α, β then add all of FIRST(β) except ε to FOLLOW(B)
- If there is a production A –> αB, or a production A –> αBβ where ε in FIRST(β), i.e. β =>* ε, then add all of FOLLOW(A) to FOLLOW(B).

Apply these rules until no more terminals or ε can be added to any FOLLOW set.

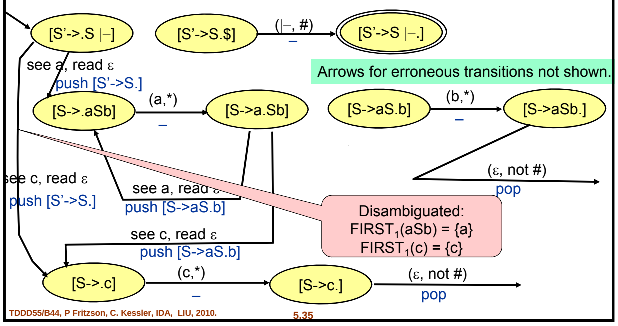
For the example grammar
S -> aSb; S->c

FOLLOW(S) = {|–, b}



Example Cont.: Finite Pushdown Automaton (FPA) Made Deterministic

- Grammar with productions { S –> aSb | c }
- Added new start symbol S': { S' -> S |–; S -> aSb; S->c }



Example (cont.): Transition table (*k*=1)

state	**final ?**	lookahead **a**	lookahead **b**	lookahead **c**	lookahead |–
[S'->.S |–]	no	push [S'->S.$]; [S->.aSb]	[Error]	push [S'->S.$]; [S->.c]	[Error]
[S'->S. |–]	no	[Error]	[Error]	[Error]	read |–; [S'->S |–.]
[S'->S |–.]	yes				
[S->.aSb]	no	read a; [S->a.Sb]	[Error]	[Error]	[Error]
[S->a.Sb]	no	push [S->aS.b]; [S->.aSb]	[Error]	push [S->aS.b]; [S->.c]	[Error]
[S->aS.b]	no	[Error]	read b; [S->aSb.]	[Error]	[Error]
[S->aSb.]	no	[Error]	pop state	[Error]	pop state
[S->.c]	no	[Error]	[Error]	read c; [S->c.]	[Error]
[S->c.]	no	[Error]	pop state	[Error]	pop state

General Approach: Predictive Parsing

At any production A -> α

- If ε is not in FIRST(α)):
  - Parser expands by production A -> α
    if current lookahead input symbol is in FIRST(α).
- otherwise (i.e., ε in FIRST(α)):
  - Expand by production A -> α
    if current lookahead symbol is in FOLLOW(A)
    or if it is |– and |– is in FOLLOW(A).

Use these rules to fill the transition table.
(pseudocode: see [ASU86] p. 190, [ALSU06] p. 224)

Summary: Parsing LL(*k*) Languages

- **Predictive LL parser**
  - iterative, based on finite pushdown automaton
  - transition-table-driven
  - can be generated automatically
- **Recursive-descent parser**
  - recursive
  - manually coded
  - easier to fix intermediate code generation, error handling
- **Both require lookahead** (or backtracking)
  to predict the next production to apply
  - Removes nondeterminism
  - Necessary checks derived from FIRST and FOLLOW sets
  - FIRST and FOLLOW are also useful for syntax error recovery

Homework

- Now, read again the part on recursive descent parsers
  and find the equivalent of
  - Context-free items (Pushdown automaton (PDA) states)
  - The stack of states
  - Pushing a state to stack
  - Popping a state from stack
  - Start state, final state

  in a recursive descent parser.