TDDD16 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, 2008.

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
  - $S =>^*_{lm} uY\alpha => u\beta\alpha =>^* ux$ and
  - $S =>^*_{lm} uY\alpha => u\gamma\alpha =>^* uy$

  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

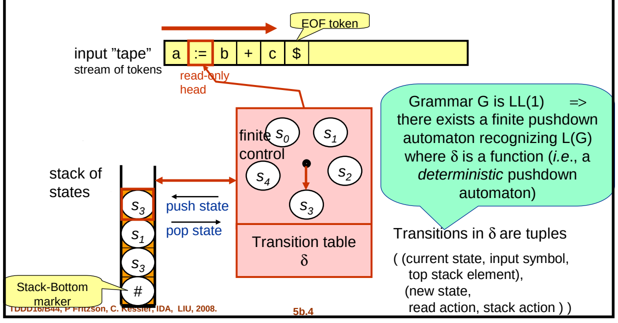
- The following grammar is LL(1) (terminals are bold-face):

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



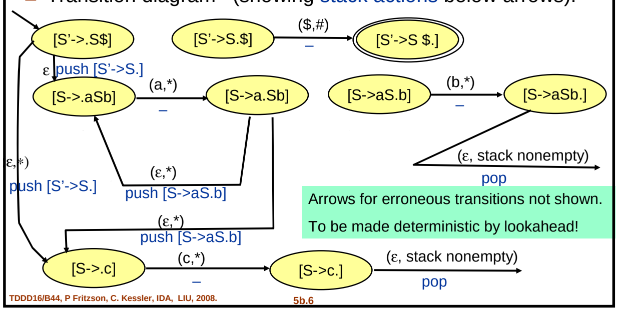
Context-Free Items

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

- Add new start symbol S' with S' –> S $
- For each production $A -> \alpha_1...\alpha_k$ 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)

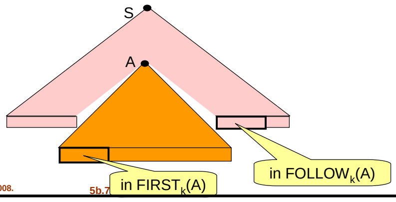
Example

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



Computing FIRST = $FIRST_1$

For all grammar symbols X:

- If X is a terminal, then FIRST(X) = { X }.
- If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).
- If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_q$ 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).

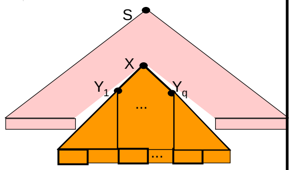
Apply these rules until no more terminals or ε can be added to any FIRST set.

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 $X_1 X_2 \ldots X_n$ of grammar symbols:

- Add to FIRST($X_1 X_2 \ldots X_n$) 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($X_1 X_2 \ldots X_n$)

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

FIRST(abc) = {a}

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

Computing FOLLOW

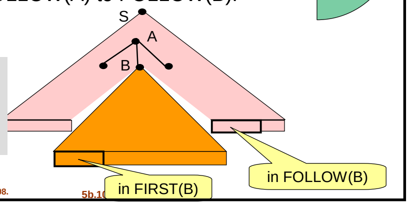
Compute FOLLOW(B) for each nonterminal B:

- Add $ to FOLLOW(S)
- If there is a production $A \rightarrow^* \alpha B \beta$ for arbitrary α, β then add all of FIRST(β) except ε to FOLLOW(B)
- If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where ε in FIRST(β), i.e. $\beta \Rightarrow^* \varepsilon$, 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.

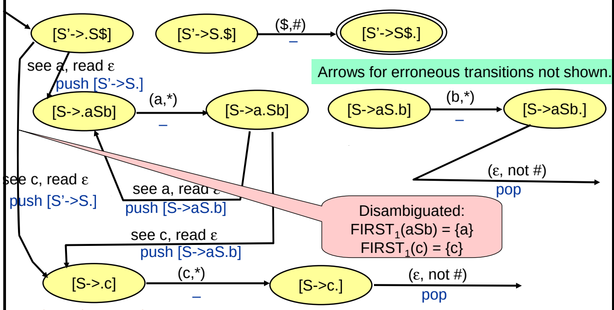
For the example grammar
S -> aSb; S->c

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.