Compilation

0368-3133

Lecture 3a:

Syntax Analysis:

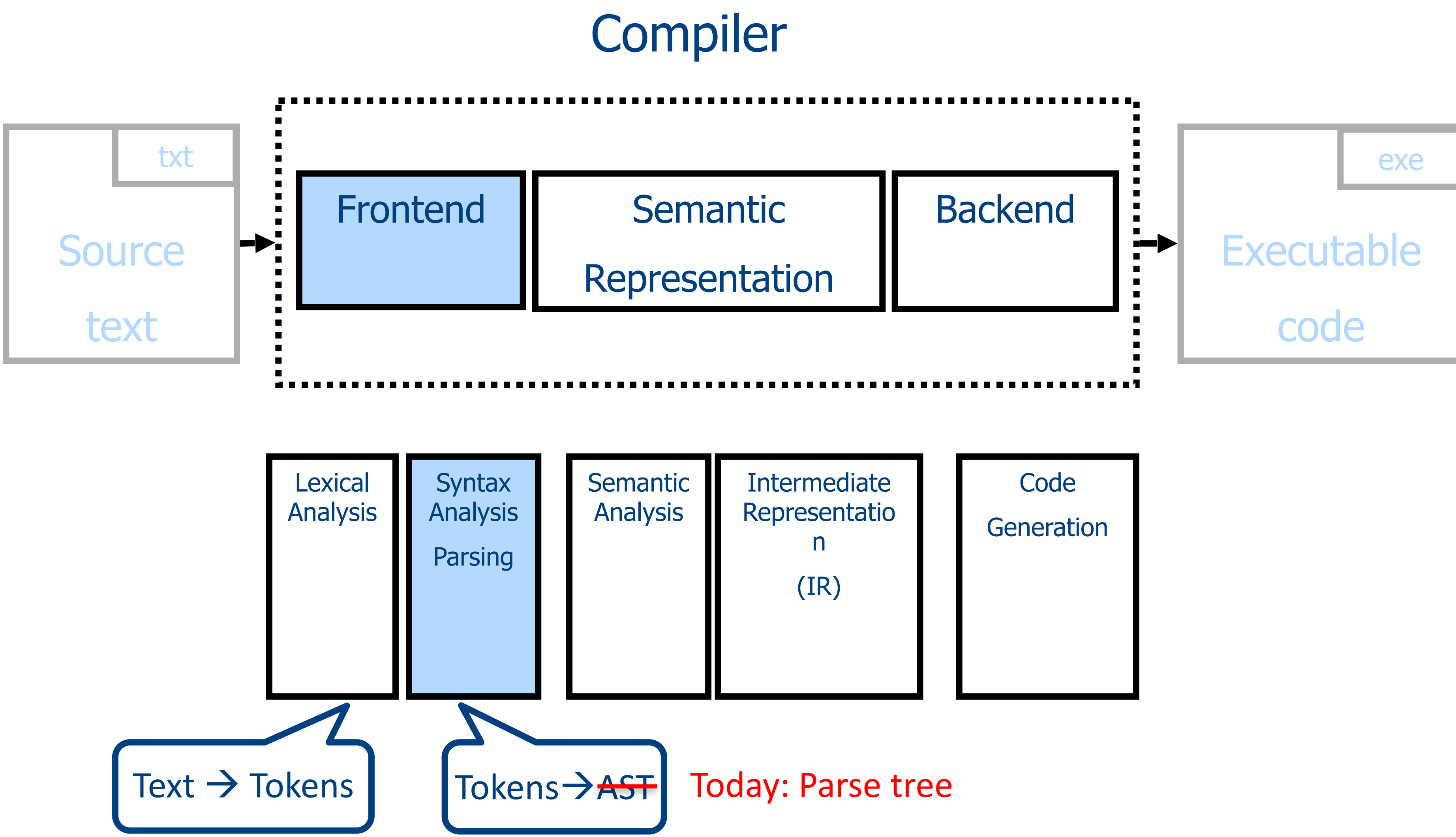
Top-Down parsing (Part 2)

Slides' credit: Shachar Itzhaky, Eran Yahav

Today

- Top-down (predictive) parsing - Part 2
- Bottom-up parsing

Conceptual Structure of a Compiler

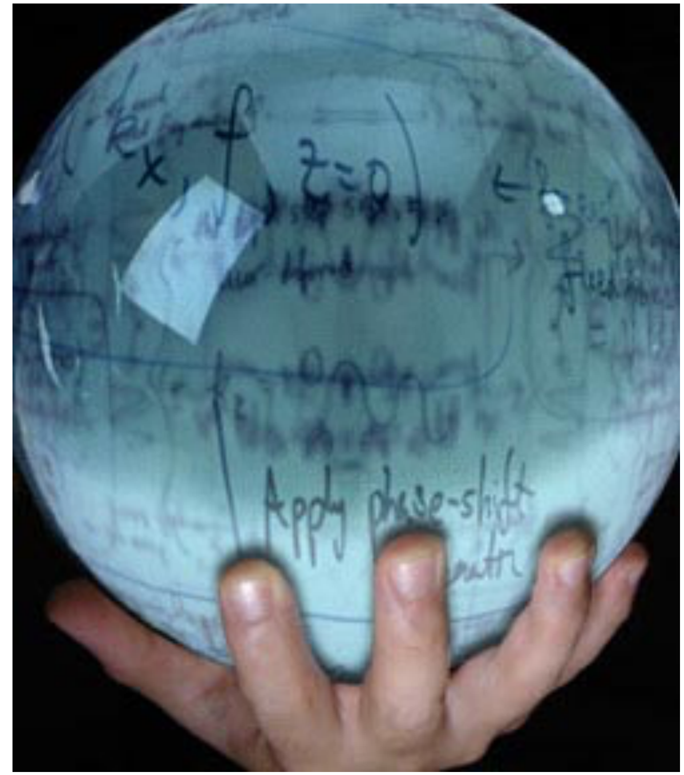


Parsing

- Goals
  - Decide whether the sequence of tokens a syntactically valid program in the source language
  - Construct a structured representation of the input
    - i.e, the parse tree
  - Error detection and reporting

Predictive Parsing

- ✓ LL(k) grammars
- ✓ Recursive descent
  - ✓ Manually written
- PDA-based parsers
  - Automatically generated
- "Fixing" LL(1) conflicts
- Bulding the parse tree
- Error handling



Reminders

FIRST Sets

$$G = (V, T, P, S)$$

- For a sequence of symbols $\alpha \in (V \cup T)^*$
  - $t \in \text{FIRST}(\alpha) \;\Leftrightarrow\; \alpha \Rightarrow^* t\,w$
  - Special case: $\varepsilon \in \text{FIRST}(\alpha) \;\Leftrightarrow\; \alpha \Rightarrow^* \varepsilon$ (α is *nullable*)

(not really
a terminal)

FOLLOW Sets

$$G = (V, T, P, S)$$

- For a nonterminal A
  - t ∈ FOLLOW(A) ⇔ S $\Rightarrow^* \alpha A t \beta$
  - Special case: $\$$ ∈ FOLLOW(A) ⇔ S $\Rightarrow^* \alpha A$

(not really
a terminal)

LL(1) grammars

- A grammar is in the class LL(1) iff
  - For every two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ we have
    - $FIRST(\alpha) \cap FIRST(\beta) = \{\}$ // including ε
    - If $\varepsilon \in FIRST(\alpha)$ then $FOLLOW(A) \cap FIRST(\beta) = \{\}$
    - If $\varepsilon \in FIRST(\beta)$ then $FOLLOW(A) \cap FIRST(\alpha) = \{\}$

Recursive Descent (for LL(1) Grammars)

```
void A() {
    find an A-production, A → X1X2...Xk ,   (α = X1X2...Xk)
        such that  current ∈ FIRST(α)
               or  ε ∈ FIRST(α) and current ∈ FOLLOW(A)
  if  no such production exists error;
  if (α == ε) return;
  for (i = 1; i ≤ k; i++) {
    if (Xi is a nonterminal)    call function Xi();
    else if (Xi == current)     current = next_token();
    else                        error;
  }
}
```

This is the basis for an LL(1) parser.

(but it is still recursive — stay tuned)

Computing FIRST & FOLLOW Sets

- ✓ Computing FIRST Sets
- Computing FOLLOW Sets

FOLLOW set - Definition

FOLLOW(A) = $\{ t \in T \mid S \Rightarrow^* \alpha A t \beta \} \cup$

$\{ \$ \mid S \Rightarrow^* \alpha A \}$

FOLLOW set - Constraints

$$\text{FOLLOW}(A) = \{\, t \in T \mid S \Rightarrow^* \alpha A t \beta \} \cup \{\, \$ \mid S \Rightarrow^* \alpha A \}$$

- $\$ \in \text{FOLLOW}(S)$

- For each $A \rightarrow \alpha\ B\ \beta$

  $\text{FIRST}(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}(B)$

if $S \Rightarrow^* \gamma A \mu \Rightarrow \gamma \alpha B \beta \mu \ \wedge\ \beta \Rightarrow^* t\delta$ then $S \Rightarrow^* \gamma \alpha B t \delta \mu$

- For each $A \rightarrow \alpha\ B\ \beta$ and $\varepsilon \in \text{FIRST}(\beta)$

  $\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$

if $S \Rightarrow^* \gamma A t \mu \Rightarrow \gamma \alpha B \beta t \mu \ \wedge\ \beta \Rightarrow^* \epsilon$ then $S \Rightarrow^* \gamma \alpha B t \mu$

Computing FOLLOW sets

$$G = (V, T, P, S)$$

1. Foreach ($B \in V$)

   FOLLOW(B) = { $t \in$ FIRST(β) | $A \to \alpha B \beta \in P$ } \ { ε }

2. FOLLOW(S) = FOLLOW(S) ∪ {\$}

3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \to \alpha B \beta \in P$)

     if ($\varepsilon \in$ FIRST(β))

       FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Yet another fixed-point computation

Example: FOLLOW sets

$S \rightarrow TZ$

$Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid \text{int } Y$

$Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   FOLLOW(B) = { $t \in$ FIRST(β) | $A \rightarrow \alpha B \beta \in P$ } \ { ε }

2. FOLLOW(S) = FOLLOW(S) ∪ {\$}
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if (ε ∈ FIRST(β))

   FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Non. Term.	S	T	Z	Y
FOLLOW				

Example: FOLLOW sets

$S \rightarrow TZ$

$T \rightarrow (S) \mid \text{int } Y$

$Z \rightarrow + S \mid \varepsilon$

$Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   $\text{FOLLOW}(B) = \{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$

2. $\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in \text{FIRST}(\beta)$)

   $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Non. Term.	S	T	Z	Y
FOLLOW	)			

Example: FOLLOW sets

$S \rightarrow TZ$ $\qquad$ $Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid \text{int } Y$ $\qquad$ $Y \rightarrow * \, T \mid \varepsilon$

1. Foreach ($B \in V$)

   FOLLOW(B) = { $t \in$ FIRST(β) | $A \rightarrow \alpha B \beta \in P$ } \ { ε }

2. FOLLOW(S) = FOLLOW(S) ∪ {\$}

3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

     if (ε ∈ FIRST(β))

       FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Non. Term.	**S**	**T**	**Z**	**Y**
FOLLOW	)	+		

Example: FOLLOW sets

$S \rightarrow TZ$

$T \rightarrow (S) \mid \text{int}\ Y$

$Z \rightarrow +S \mid \varepsilon$

$Y \rightarrow *T \mid \varepsilon$

1. Foreach ($B \in V$)

   $\text{FOLLOW}(B) = \{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$

2. $\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in \text{FIRST}(\beta)$)

   $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Non. Term.	S	T	Z	Y
FOLLOW	) $	+		

Example: FOLLOW sets

$S \rightarrow TZ$ $\qquad$ $Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid \text{int}\ Y$ $\qquad$ $Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   FOLLOW(B) = $\{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$
2. FOLLOW(S) = FOLLOW(S) $\cup$ {\$}
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in \text{FIRST}(\beta)$)

   FOLLOW(B) = FOLLOW(B) $\cup$ FOLLOW(A)

Non. Term.	**S**	**T**	**Z**	**Y**
FOLLOW	) \$	+	) \$	

Example: FOLLOW sets

$S \rightarrow TZ$

$T \rightarrow (S) \mid \text{int}\ Y$

$Z \rightarrow +S \mid \varepsilon$

$Y \rightarrow *T \mid \varepsilon$

1. Foreach ($B \in V$)

   $\text{FOLLOW}(B) = \{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$
2. $\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in \text{FIRST}(\beta)$)

   $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Non. Term.	S	T	Z	Y
FOLLOW	) $	+	) $	+

Example: FOLLOW sets

$S \rightarrow TZ$ $\qquad$ $Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid \text{int } Y$ $\qquad$ $Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   FOLLOW(B) = { $t \in$ FIRST(β) | $A \rightarrow \alpha B \beta \in P$ } \ { ε }

2. FOLLOW(S) = FOLLOW(S) ∪ {\$}

3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

      if ($\varepsilon \in$ FIRST(β))

         FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Non. Term.	**S**	**T**	**Z**	**Y**
FOLLOW	) \$	+) \$	) \$	+

Example: FOLLOW sets

$S \rightarrow TZ$ $\qquad$ $Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid int\ Y$ $\qquad$ $Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   $FOLLOW(B) = \{ t \in FIRST(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$

2. $FOLLOW(S) = FOLLOW(S) \cup \{\$\}$
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in FIRST(\beta)$)

   $FOLLOW(B) = FOLLOW(B) \cup FOLLOW(A)$

Non. Term.	**S**	**T**	**Z**	**Y**
FOLLOW	) \$	+) \$	) \$	+

Example: FOLLOW sets

$S \longrightarrow TZ$ $\qquad$ $Z \longrightarrow + S \mid \varepsilon$

$T \longrightarrow (S) \mid \text{int } Y$ $\qquad$ $Y \longrightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   $\text{FOLLOW}(B) = \{ t \in \text{FIRST}(\beta) \mid A \rightarrow \alpha B \beta \in P \} \setminus \{ \varepsilon \}$
2. $\text{FOLLOW}(S) = \text{FOLLOW}(S) \cup \{\$\}$
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in \text{FIRST}(\beta)$)

   $\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup \text{FOLLOW}(A)$

Non. Term.	S	T	Z	Y
FOLLOW	) \$	+) \$	) \$	+) \$

Example: FOLLOW sets

$S \rightarrow TZ$ $\qquad Z \rightarrow + S \mid \varepsilon$

$T \rightarrow (S) \mid \text{int } Y$ $\qquad Y \rightarrow * T \mid \varepsilon$

1. Foreach ($B \in V$)

   FOLLOW(B) = { $t \in$ FIRST(β) | $A \rightarrow \alpha B \beta \in P$ } \ { ε }
2. FOLLOW(S) = FOLLOW(S) ∪ {\$}
3. Repeat until FOLLOW(X) does not change for any X

   foreach ($A \rightarrow \alpha B \beta \in P$)

   if ($\varepsilon \in$ FIRST(β))

   FOLLOW(B) = FOLLOW(B) ∪ FOLLOW(A)

Non. Term.	S	T	Z	Y
FOLLOW	) \$	+) \$	) \$	+) \$

Fixpoint!

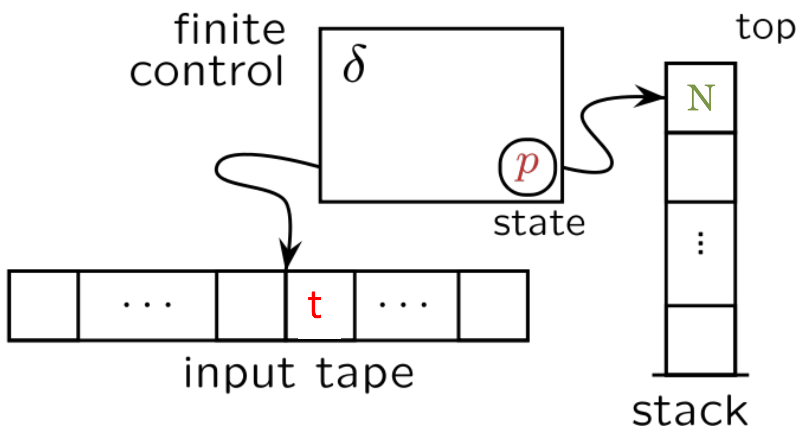
Pushdown Automata-based Parsers

LL(k) Parsers

- Recursive Descent
  - Manual construction
  - Uses recursion
- Wanted
  - A parser that can be generated automatically
  - Does not use recursion
  - Uses a deterministic computational model

LL(1) Parsing with Pushdown Automata

- A PDA uses
  - Prediction stack
  - Input stream
  - Transition table



- δ : nonterminals × tokens ⟶ production alternative (right-hand side)
- Entry δ[N,t] for nonterminal N and terminal t contains the alternative of N that would be predicted when the current input starts with t

The Transition Table

- Constructing the transition table is easy
  - It relies on FIRST and FOLLOW
  - Based on the concept of our recursive descent earlier

Building The Transition Table

$$G = \langle V, T, P, S \rangle$$

Foreach $A \longrightarrow \alpha \in P$

- $\delta[A,t] = \alpha$ if $t \in FIRST(\alpha) \setminus \{\varepsilon\}$
- $\delta[A,t] = \alpha$ if $\varepsilon \in FIRST(\alpha)$ and $t \in FOLLOW(A)$
  - t can also be \$

δ is not well defined ➔ the grammar is not LL(1)

Example Transition Table

(1) E ⟶ LIT

(2) E ⟶ **(** E OP E **)**

(3) E ⟶ not E

(4) LIT ⟶ true

(5) LIT ⟶ false

(6) OP ⟶ and

(7) OP ⟶ or

(8) OP ⟶ xor

which rule should be used

Input tokens

Nonterminals

δ	(	)	not	true	false	and	or	xor	$
E									
LIT									
OP									

Example Transition Table

(1) E ⟶ LIT

(2) E ⟶ **(** E OP E **)**

(3) E ⟶ not E

(4) LIT ⟶ true

(5) LIT ⟶ false

(6) OP ⟶ and

(7) OP ⟶ or

(8) OP ⟶ xor

which rule should be used

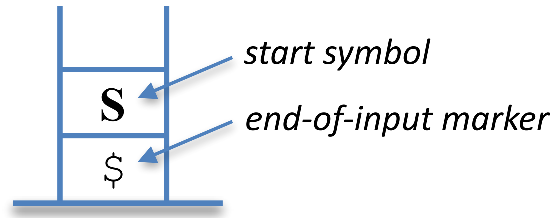
Input tokens

δ	(	)	not	true	false	and	or	xor	$
E	2		3	1	1				
LIT				4	5				
OP						6	7	8	

Nonterminals

LL(k) parsing with pushdown automata

- Initial state —



- Two possible moves

(σ = next token)

  - Predict

    When top of stack is nonterminal **N**, pop **N**, lookup δ[**N**,σ].
    If δ[**N**,σ] is defined, push δ[**N**,σ] on prediction stack.
    Otherwise – syntax error.

  - Match

    When top of stack is a terminal t, it must be equal to next input token.
    If t = σ, pop t and consume σ.
    If t ≠ σ – syntax error.

- Parsing terminates when prediction stack is empty.
  - At this point the input *must be finished* ⇒ success.

Simple Example

aacbb$

$A \rightarrow aAb \mid c$

(top is left)

Stack content	Remaining input	Action
A$	**a**acbb$	predict(A,a) = aAb
aAb$	**a**acbb$	match(a,a)
Ab$	**a**cbb$	predict(A,a) = aAb
aAbb$	**a**cbb$	match(a,a)
Abb$	**c**bb$	predict(A,c) = c
cbb$	**c**bb$	match(c,c)
bb$	**b**b$	match(b,b)
b$	**b**$	match(b,b)
$	**$**	match($,$)

	a	b	c
A	aAb		c

LL(1) Conflicts

- When the grammar is not LL(1) we might be able to "fix" it
- No algorithm
  - e.g., consider ambiguous grammars
- A few tricks/guidlines can help

LL(1) Conflicts

term → `ID` | indexed
indexed → `ID` `[` expr `]`

- FIRST(`ID`) = { `ID` }
- FIRST(indexed) = { `ID` }

Both are alternatives for term → α

FIRST/FIRST conflict

⇒ This grammar is not in LL(1). Can we "fix" it?

Left Factoring

- Rewrite the grammar to be in LL(1)

term → `ID` | indexed
indexed → `ID` `[` expr `]`

⇓

term → `ID` after_ID
after_ID → `[` expr `]` | ε

Intuition: just like factoring $x \cdot y + x \cdot z$ into $x \cdot (y + z)$

Left Factoring

another example

$$S \to \texttt{if}\ E\ \texttt{then}\ \{\ S\ \}\ \texttt{else}\ \{\ S\ \}$$
$$\quad \mid\ \texttt{if}\ E\ \texttt{then}\ \{\ S\ \}$$
$$\quad \mid\ \ldots$$

$$S \to \texttt{if}\ E\ \texttt{then}\ \{\ S\ \}\ S'$$
$$\quad \mid\ \ldots$$
$$S' \to \texttt{else}\ \{\ S\ \}\ \mid\ \varepsilon$$

Cheat

Are We Done?

$S \to A\ a\ b$

$A \to a \mid \varepsilon$

- Select a rule for A with a in the look-ahead:
  - Should we pick (1) $A \to a$ or (2) $A \to \varepsilon$?

- FIRST(a) = { 'a' }
- FIRST(ε) = { ε }   FOLLOW(A) = { 'a' }

alternatives for $A \to \alpha$

FIRST/FOLLOW conflict

$\Rightarrow$ The grammar is not in LL(1). Can we fix *that*?

Grammatical Substitution

$S \rightarrow A\ a\ b$

$A \rightarrow a \mid \varepsilon$

Substitute A in S

$S \rightarrow a\ a\ b \mid a\ b$

Left factoring

$S \rightarrow a\ after\_a$

$after\_a \rightarrow a\ b \mid b$

So Far

- Can determine if a grammar is in LL(1)
  - The FIRST and FOLLOW sets
  - Algorithms for computing these
- Have some techniques for modifying a grammar to find an equivalent grammar in LL(1)
  - Left factoring
  - Assignment
- Now let's look at a third example and present one more such technique

Left Recursion

$$E \rightarrow E \mathbf{+} \text{ term } \mid \text{ term}$$

- Left recursion cannot be handled with a **bounded** lookahead
- What can we do?
- <u>Theorem</u>: Any grammar with left recursion has an equivalent grammar without left recursion

Left Recursion Removal

p. 130

$N \rightarrow N\alpha \mid \beta$

G_1

$N \rightarrow \beta N'$
$N' \rightarrow \alpha N' \mid \varepsilon$

G_2

Can be done algorithmically. Problem: grammar becomes mangled beyond recognition

$L(G_1) = \{\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots\}$

$N' \Rightarrow^* \varepsilon, \alpha, \alpha\alpha, \alpha\alpha\alpha, \ldots$

$L(G_2) = \{\beta, \beta\alpha, \beta\alpha\alpha, \beta\alpha\alpha\alpha, \ldots\}$

- For our example:

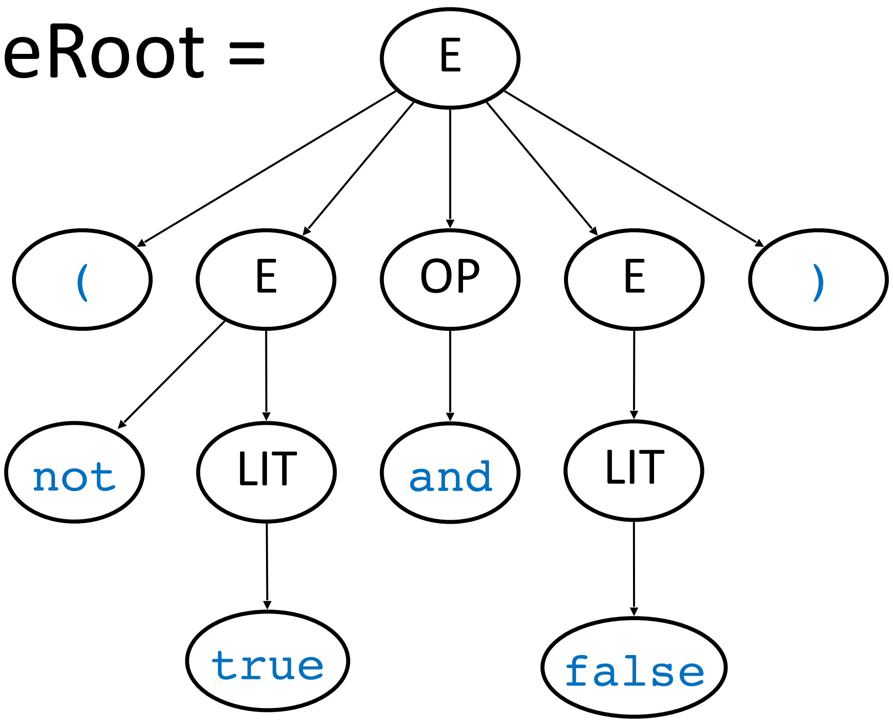
$E \rightarrow E + \text{term}$
$\quad \mid \text{term}$

$E \rightarrow \text{term}\ E'$
$E' \rightarrow +\ \text{term}\ E' \mid \varepsilon$

Building the Parse Tree

Building the Parse Tree

- Input = "`( not true and false )`";
- `Node` treeRoot =



Building the parse tree with recursive descent parsers

- Can add an action to perform on each production rule to build the parse tree
  - Every function returns an object of type Node
  - Every Node maintains a list of children
  - Function calls can add new children

Building the parse tree with recursive descent parsers

```
Node E() {
  result = new Node();
  result.name = "E";
  if (current ∈ {TRUE, FALSE}) // E
    result.addChild(LIT());
  else if (current == LPAREN)  // E
    result.addChild(match(LPAREN));
    result.addChild(E());
    result.addChild(OP());
    result.addChild(E());
    result.addChild(match(RPAREN));
  else if (current == NOT)         // E → not E
    result.addChild(match(NOT));
    result.addChild(E());
  else error;
    return result;
}
```

```
Node match(Token t) {

  result = new Node();
  result.name = "T";
  result.val = "t";
  if (current == t)
    return result;
  else error;
}
```

Building the Syntax Tree with PDA-based Parsers

- Can add an action to perform on each production — in the LL(k) case, at predict

- Every symbol on the parser stack (except $) represents a node in the syntax tree.
  - The initial entry for S is the root.
  - When the top of the stack is a nonterminal N and parser performs predict(N, $X_1X_2 \ldots X_k$):
    - Create nodes for X_i's as child nodes of N

Simple Example

aacbb$

$A \rightarrow aAb \mid c$

Stack content (top is left)	Remaining input	Action
A$	**a**acbb$	

A

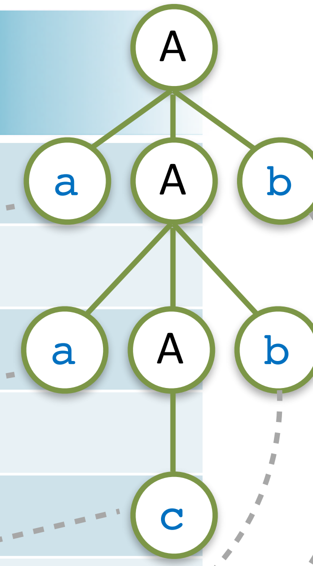
	a	b	c
A	aAb		c

Simple Example

aacbb$

$A \to aAb \mid c$

Stack content (top is left)	Remaining input	Action
A$	**a**acbb$	predict(A,a) = aAb
aAb$	**a**acbb$	match(a,a)
Ab$	**a**cbb$	predict(A,a) = aAb
aAbb$	**a**cbb$	match(a,a)
Abb$	**c**bb$	predict(A,c) = c
cbb$	**c**bb$	match(c,c)
bb$	**b**b$	match(b,b)
b$	**b**$	match(b,b)
$	**$**	match($,$) accept



	a	b	c
A	aAb		c

Error Handling

- Types of errors
  - Lexical errors
  - Syntax errors
  - Semantic errors (e.g., type mismatch)
  - Runtime errors (e.g., division by zero)

Error Handling

- x = a * (p+q * (-b * (r-s) ;
  - Where should we report the error?
  - The valid prefix property

The Valid Prefix Property

- For every prefix tokens
  - t_1, t_2, …, t_i that the parser identifies as legal:
    - there exists tokens t_{i+1}, t_{i+2}, …, t_n such that t_1, t_2, t_i, t_{i+1}, … , t_n is a syntactically valid program

- Parser continues normally as long as the read tokens constitue a valid prefix

Error Handling

- Types of errors
  - Lexical errors
  - Syntax errors
  - Semantic errors (e.g., type mismatch)
  - Runtime errors (e.g., division by zero)

  - Requirements
    - Report the error clearly
    - Recover and continue so more errors can be discovered
    - Be efficient
      - Do not get into an infinite loop

Recovery is tricky

- Heuristics for adding tokens, dropping tokens, skipping to semicolon, etc.

Simple Example, Inifnite Loop

b $

A → aAb | c

report: “missing token **a** inserted in line YYY”

Stack content	Remaining input	Action
A $	**b** $	predict(A,b) = error
A $	**a** b $	predict(A,a) = aAb
a A b $	**a** b $	match(a,a)
A b $	**b** $	predict(A,b) = error
A b $	**a** b $	predict(A,a) = aAb
a A b b $	**a** b $	match(a,a)
…	…	…

fix=push ‘a’

fix=push ‘a’

	a	b	c
A	aAb		c

Error Handling in LL Parsers

MCD
S3.4.5

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

A = FIRST(stack content)

don't need to do anything

panic mode

Simple Example, Bad Word

abcbb$

$A \rightarrow aAb \mid c$

Stack content	Remaining input	Action
A $	**a** b c b b $	predict(A,a) = aAb
a A b $	**a** b c b b $	match(a,a)
A b $	**b** c b b $	predict(A,b) = error

	a	b	c
A	$A \rightarrow a\ A\ b$		$A \rightarrow c$

Simple Example, Bad Word

abcbb$

$A \rightarrow aAb \mid c$

Stack content	Remaining input	Action
A $	a b c b b $	predict(A,a) = aAb
a A b $	a b c b b $	match(a,a)
A b $	b c b b $	predict(A,b) = ✗ skip
A b $	c b b $	predict(A,a) = A → c
c b $	c b b $	match(c,c)
b $	b b $	match(b,b)
$	b $	match($,b) = ✗ skip
$	$	match($,$)

	a	b	c
A	A → a A b		A → c

Error Handling in LL Parsers

FOLLOW-set error recovery

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

A = FOLLOW(Currently predicted non-terminal)

pop the stack symbols used to derive that non-terminal

Error Handling in LL Parsers

FOLLOW-set
error recovery

- The **acceptable-set method**
  - Step 1: construct a set A of *acceptable* terminals based on the current state of the parser.
  - Step 2: discard input tokens until a token $t_A \in A$ is encountered.
  - Step 3: advance the parser to the next state in which it can consume t_A.

Choose A manually

A = ~~FOLLOW(Currently processed non-terminal)~~

pop the stack symbols used to derive that non-terminal

Error Handling

- Different compilers adopt different approaches.
- Some examples
  - Acceptable-set method (e.g., panic mode): drop tokens until reaching a synchronizing token, like a semicolon, a right parenthesis, end of file, etc.
  - Phrase-level recovery: attempting local changes: replace "," with ";", eliminate or add a ";", etc.
  - Error production: anticipate errors and automatically handle them by adding special rules to the grammar
  - Global correction: find the minimum modification to the program that will make it derivable in the grammar
    - Not a practical solution, except with very small grammars

A Note About ANTLR

- ANTLR = ANother Tool for Language Recognition
- LL(*) algorithm
  - Like LL(*k*) on steroids
  - Notable extensions:
    - Repeat operators — like in regex
    - Lookahead predicates — allow for unbounded lookahead at the cost of backtracking

✷ There's a nice demo at lab.antlr.org

* *LL(*): The Foundation of the ANTLR Parser Generator*, Parr and Fischer, PLDI 2011

Summary

- Top-down parsing
  - ✓ LL(k) grammars
  - ✓ LL(k) parsing with recursive descent
  - ✓ LL(k) parsing with pushdown automata
- LL(k) parsers
  - ✓ Cannot deal with common prefixes and left recursion
  - ✓ Left-recursion removal might result in a complicated grammar


That's all Folks