Concepts of Programming Languages – Recitation 1: Predictive Parsing

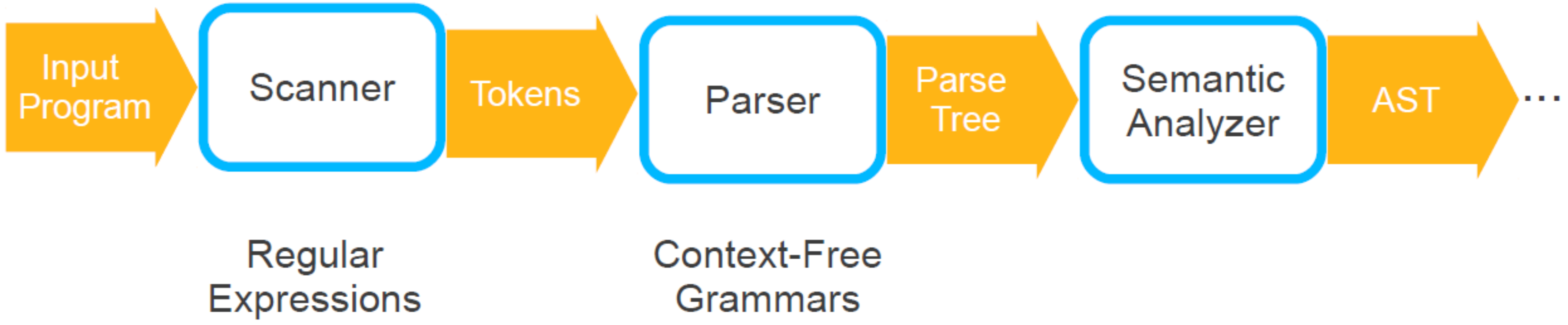
Oded Padon
odedp@mail.tau.ac.il

Reference:
Modern Compiler Implementation in Java by Andrew W. Appel – Ch. 3

Administrative

- Course website and forum (**not using moodle**):
  - http://cs.tau.ac.il/~msagiv/courses/pl17.html
  - https://groups.google.com/a/mail.tau.ac.il/d/forum/taupl17-group
- Recitation groups:
  - Wednesday 13:10-14:00 Melamed
  - Wednesday 14:10-15:00 Melamed
  - Wednesday 15:10-16:00 Melamed
- Grade: 50% Exercises, 50% Exam
- Exercises: **all mandatory**, teams of 1 or 2 or 3 students
- First exercise is published
- My reception hours: Wednesdays, 17:00-18:00
  - email before you come!

Role of Parsing



Context Free Grammars

- Terminals (tokens)
- Non-terminals
  - Start non-terminal
- Derivation rules (also called productions)
  <Non-Terminal> → Symbol Symbol … Symbol

Example Context Free Grammar

- Terminals (tokens)
- Non-terminals
  - Start non-terminal

1 <S> → <S> ; <S>
2 <S> → id := <E>
3 <S> → print (<L>)
4 <E> → id
5 <E> → num
6 <E> → <E> + <E>
7 <E> → (<S>, <E>)
8 <L> → <E>
9 <L> → <L>, <E>

Example Parse Tree

<S>

<S> ; <S>

<S> ; id := E

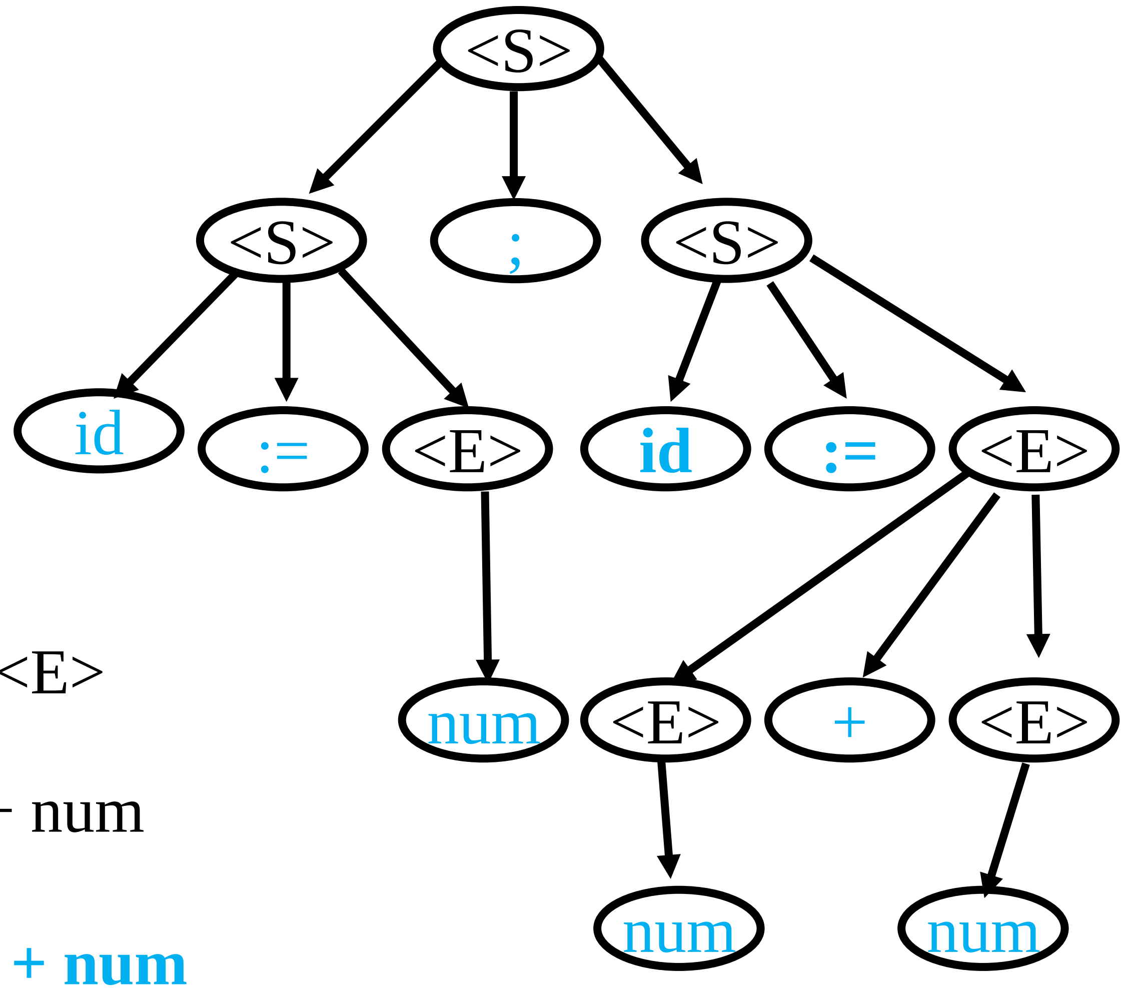
id := <E> **; id :=** <E>

id := num ; id := <E>

id := num ; id := <E> + <E>

id := num ; id := <E> + num

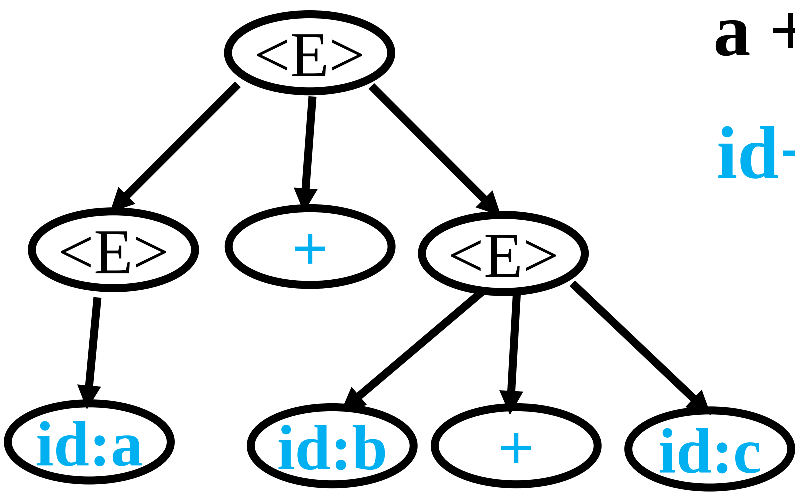
id := num ; id :=num + num



Ambiguous Grammars

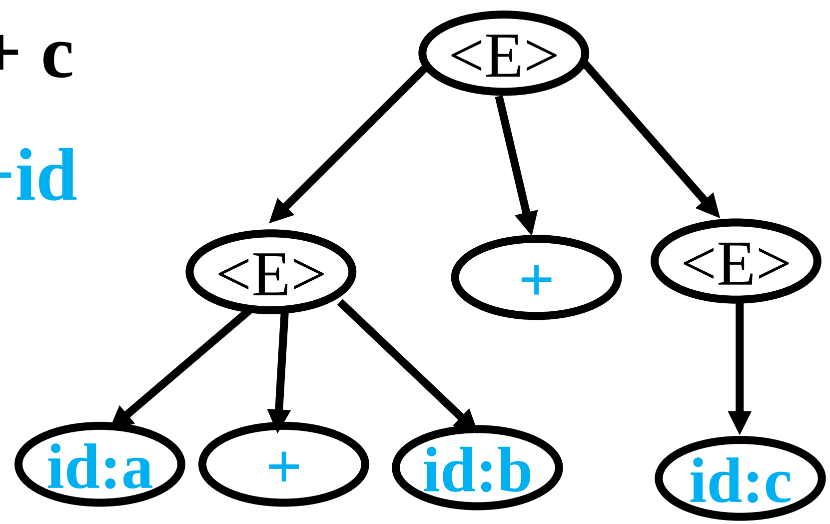
- Two leftmost derivations
- Two rightmost derivations
- Two parse trees

1 <E> → <E> + <E>
2 <E> → <E> * <E>
3 <E> → id
4 <E> → (<E>)



a + b + c

id+id+id



Non Ambiguous Grammars for Arithmetic Expressions

Ambiguous grammar:

1 <E> → <E> **+** <E>
2 <E> → <E> ***** <E>
3 <E> → **id**
4 <E> → **(**<E>**)**

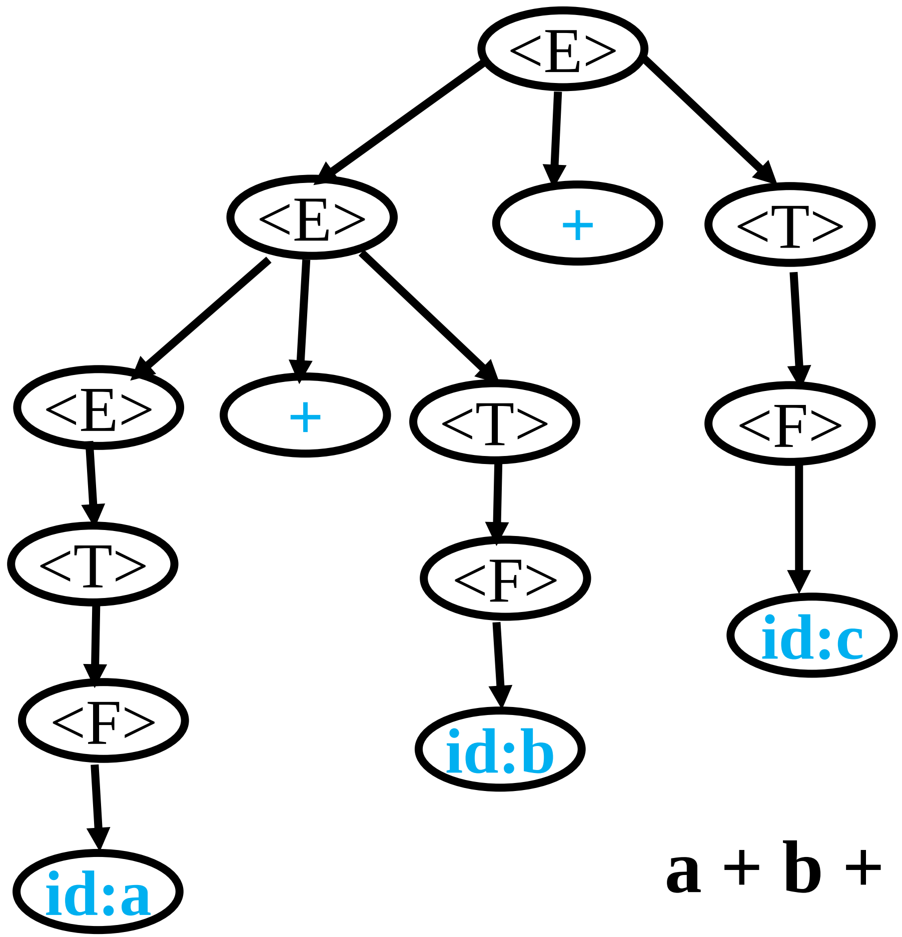
Non-Ambiguous grammar:

1 <E> → <E> **+** <T>
2 <E> → <T>
3 T → <T> ***** <F>
4 T → <F>
5 F → **id**
6 F → **(**<E>**)**

Non Ambiguous Grammars for Arithmetic Expressions

Non-Ambiguous grammar:

1 <E> → <E> + <T>
2 <E> → <T>
3 T → <T> * <F>
4 T → <F>
5 F → **id**
6 F → **(**<E>**)**



a + b + c

id+id+id

Predictive Parsing – LL(1)

- **L**eft to right, **L**eft most derivation, **1** token look ahead
- We must uniquely predict the next rule by the current non-terminal and 1 next token
  - Grammar must be non ambiguous
  - Grammar must not contain left recursion (right recursion is ok)
  - Grammar must be left factored
  - These conditions are necessary but not sufficient

Example Predictive Parser

<S> → **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

```
def parse_S():
  if next(id):
    match(id)
    match(assign)
    parse_E()
  elif next(if_tok):
    match(if_tok)
    match(lp)
    parse_E()
    match(rp)
    parse_S()
    match(else_tok)
    parse_S()
  else:
    syntax_error()
```

```
def parse_E():
  parse_T()
  parse_EP()

def parse_T():
  if next(id):
    match(id)
  elif next(lp):
    match(lp)
    parse_E()
    match(rp)
  else:
    syntax_error()
```

```
def parse_EP():
  if next(plus):
    match(plus)
    parse_E()
  elif (next(rp) or
        next(else_tok) or
        next(eof)):
    return  # ε
  else:
    syntax_error()
```

Example Predictive Parser

<S> → **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

<S> → <S> **;** <S> ?

```
def parse_S():
  if next(id):
    match(id)
    match(assign)
    parse_E()
  elif next(if_tok):
    match(if_tok)
    match(lp)
    parse_E()
    match(rp)
    parse_S()
    match(else_tok)
    parse_S()
  else:
    syntax_error()
```

```
def parse_E():
  parse_T()
  parse_EP()

def parse_T():
  if next(id):
    match(id)
  elif next(lp):
    match(lp)
    parse_E()
    match(rp)
  else:
    syntax_error()
```

```
def parse_EP():
  if next(plus):
    match(plus)
    parse_E()
  elif (next(rp) or
        next(else_tok) or
        next(eof)):
    return  # ε
  else:
    syntax_error()
```

Nullable Non-Terminals

- Non-terminal A is **nullable** iff A $\rightarrow^*$ ε

Example:

<S>→ **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

- Is S nullable?
- Is E nullable?
- Is T nullable?
- Is EP nullable?

Computing Nullable

- Non-terminal A **nullable** iff A $\rightarrow^*$ ε

```
Nulllable = ∅
for each rule A → ε:
    add A to Nullable
while changes occur do:
    for each rule A → A1 A2 … An :
        if {A1, A2 ,… , An} ⊆ Nullable:
            add A to Nullable
```

Example – Computing Nullable

<S>→ **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

```
Nulllable = ∅
for each rule A → ε:
    add A to Nullable
while changes occur do:
    for each rule A → A1 A2 … An :
        if {A1, A2 ,… , An} ⊆ Nullable:
            add A to Nullable
```

Nulllable = ∅

Nullable = {EP}

Fix point reached

First Sets

- First(α) – set of all tokens that can be the start of α
- $\text{First}(\alpha) = \{\, t \mid \exists \beta : \alpha \rightarrow^{*} t\, \beta \,\}$

Example:

<S> → **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

- First(S) =
- First(E) =
- First(T) =
- First(EP) =

Computing First

- First(α) – set of all tokens that can be the start of α
- First(α) = $\{ t \mid \exists\beta: \alpha \rightarrow^* t\ \beta \}$

```
For each token t: First(t) = {t}
For each non-terminal A: First(A) = ∅
while changes occur do:
    for each rule A → A1 A2 … An :
        for each 1 ≤ i ≤ n:
            if {A1, A2 ,… , Ai-1} ⊆ Nullable:
                add First(Ai) to First(A)
```

- For convenience, we defined First(t) = {t} for tokens

Example – Computing First

1 <S>→ **id :=** <E>
2 <S> → **if (**<E>**)** <S> **else** <S>
3 <E> → <T> <EP>
4,5 <T> → **id** | **(**<E>**)**
6,7 <EP> → ε | **+** <E>

```
Nullable = {EP}
```

```
For each token t: First(t) = {t}
For each non-terminal A: First(A) = ∅
while changes occur do:
  for each rule A → A1 A2 … An :
    for each 1 ≤ i ≤ n:
      if {A1, A2 ,… , Ai-1} ⊆ Nullable:
        add First(Ai) to First(A)
```

```
First(t) = {t} for t in {id, ass, if, lp, rp, else, plus}

add id to First(S)      # 1
add if to First(S)      # 2
                        # 3
add id to First(T)      # 4
add lp to First(T)      # 5
                        # 6
add plus to First(EP)   # 7

add id,lp to first(E)   # 3

Fix point reached
```

A	**First(A)**
S	id, if
E	id, lp
T	id, lp
EP	plus

Follow Sets

- Follow(A) – set of all tokens that can appear after A
- Follow(A) = { t | ∃β,γ: <S> $\rightarrow^*$ β <A> t γ}

Example:

<S>→ **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

- Follow(S) =
- Follow(E) =
- Follow(T) =
- Follow(EP) =

Computing Follow

- Follow(A) – set of all tokens that can appear after A
- $\text{Follow}(A) = \{ t \mid \exists \beta,\gamma: <S> \rightarrow^{*} \beta <A> t\ \gamma \}$

For each non-terminal A: Follow(A) = ∅
For the start non-terminal S: Follow(S) = {eof}
while changes occur do:
    for each rule $<A> \rightarrow A_1\ A_2 \ldots A_n$:
        for each $1 \le i \le n$:
            if $\{A_{i+1}, \ldots, A_n\} \subseteq$ Nullable:
                add Follow(A) to Follow(A_i)
        for each $1 \le i < j \le n$:
            if $\{A_{i+1}, \ldots, A_{j-1}\} \subseteq$ Nullable:
                add First(A_j) to Follow(A_i)

Example – Computing Follow

1 <S>→ **id :=** <E>
2 <S> → **if (**<E>**)** <S> **else** <S>
3 <E> → <T> <EP>
4,5 <T> → **id** | **(**<E>**)**
6,7 <EP> → ε | **+** <E>

```
For each non-terminal A≠S: Follow(A) = ∅
For the start non-terminal: Follow(S) = {eof}
while changes occur do:
  for each rule <A> → A1 A2 … An :
    for each 1 ≤ i ≤ n:
      if {Ai+1,…, An} ⊆ Nullable:
        add Follow(A) to Follow(Ai)
    for each 1 ≤ i < j ≤ n:
      if {Ai+1,…, Aj-1} ⊆ Nullable:
        add First(Aj) to Follow(Ai)
```

```
Nullable = {EP}
First(S) = {id, if}
First(E) = First(T) = {id, lp}
First(EP) = {plus}
```

```
Follow(A) = ∅ for A in {E, T, EP}
Follow(S) = {eof}

add eof to Follow(E)        # 1
add rp to Follow(E)         # 2
add else to Follow(S)       # 2
add eof, rp to Follow(T)    # 3
add eof, rp to Follow(EP)   # 3
add plus to Follow(T)       # 3

add else to Follow(E)       # 1
add else to Follow(T)       # 3
add else to Follow(EP)      # 3

Fix point reached
```

A	Follow(A)
S	eof, else
E	eof, rp, else
T	eof, rp, plus, else
EP	eof, rp, else

Select Sets

- For each rule $A \rightarrow \alpha$:
  - If α is not nullable: $Select(A \rightarrow \alpha) = First(\alpha)$
  - If α is nullable: $Select(A \rightarrow \alpha) = First(\alpha) \cup Follow(A)$

- If we need to parse A, we can decide which rule to apply according to the next token and the select sets

- The grammar is LL(1) iff:
  for every two grammar rules <A> $\rightarrow \alpha$ and <A> $\rightarrow \beta$:
  $Select(A \rightarrow \alpha) \cap Select(A \rightarrow \beta) = \emptyset$

Example – Computing Select

<S>→ **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<EP> → ε | **+** <E>

A	Nullable	First(A)	Follow(A)
S	no	id, if	eof, else
E	no	id, lp	eof, rp, else
T	no	id, lp	eof, rp, plus, else
EP	yes	plus	eof, rp, else

For each rule A → α:

If α is not nullable:

Select(A→ α) = First(α)

If α is nullable:

Select(A→ α) = First(α) ∪ Follow(A)

A→ α	Select(A→ α)
<S>→ **id :=** <E>	id
<S> → **if (**<E>**)** <S> **else** <S>	if
<E> → <T> <EP>	id, lp
<T> → **id**	id
<T> → **(**<E>**)**	lp
<EP> → ε	eof, rp, else
<EP> → **+** <E>	plus

Generating a Predictive Parser

- Compute: Nullable, First, Follow, Select
  - If the grammar is not LL(1) report an error

- Create a procedure for every non-terminal A:
  - For every rule A → α:
    - If the next token is in Select(A → α), apply the rule <A> → α: this contains recursive calls to procedures of other non-terminals
  - If the next token is not in any Select(A → α), report syntax error

Putting It All Together

A→ α	Select(A→ α)
<S>→ **id :=** <E>	id
<S> → **if (**<E>**)** <S> **else** <S>	if
<E> → <T> <EP>	id, lp
<T> → **id**	id
<T> → **(**<E>**)**	lp
<EP> → ε	eof, rp, else
<EP> → **+** <E>	plus

```
def parse_S():
  if next(id):
    match(id)
    match(assign)
    parse_E()
  elif next(if_tok):
    match(if_tok)
    match(lp)
    parse_E()
    match(rp)
    parse_S()
    match(else_tok)
    parse_S()
  else:
    syntax_error()
```

```
def parse_E():
  parse_T()
  parse_EP()

def parse_T():
  if next(id):
    match(id)
  elif next(lp):
    match(lp)
    parse_E()
    match(rp)
  else:
    syntax_error()
```

```
def parse_EP():
  if next(plus):
    match(plus)
    parse_E()
  elif (next(rp) or
        next(else_tok) or
        next(eof)):
    return  # ε
  else:
    syntax_error()
```