Concepts of Programming Languages – Recitation 1: Predictive Parsing

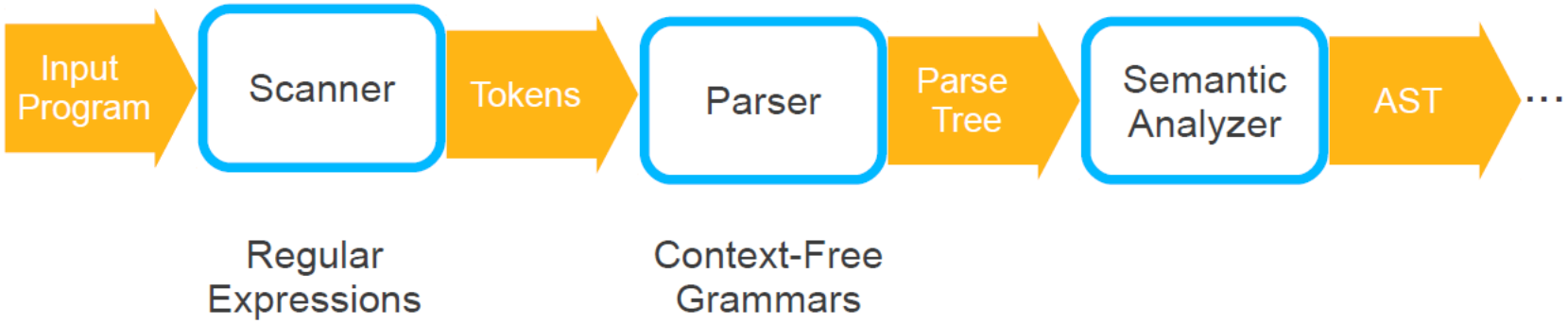
Oded Padon
odedp@mail.tau.ac.il

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

Administrative

- Grade: 50% Exercises, 50% Exam
- Exercises: all mandatory, teams of 1 or 2 or 3
- First exercise will be published soon
- Course website: http://cs.tau.ac.il/~msagiv/courses/pl15.html
- Course forum – link from website
- My reception hours: Wednesdays, 18:00-19:00
  – email before you come!

Role of Parsing



Context Free Grammars

- Terminals (tokens)
- Non-terminals
  - Start non-terminal
- Derivation rules (also called productions)
  <Non-Terminal> $\rightarrow$ 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

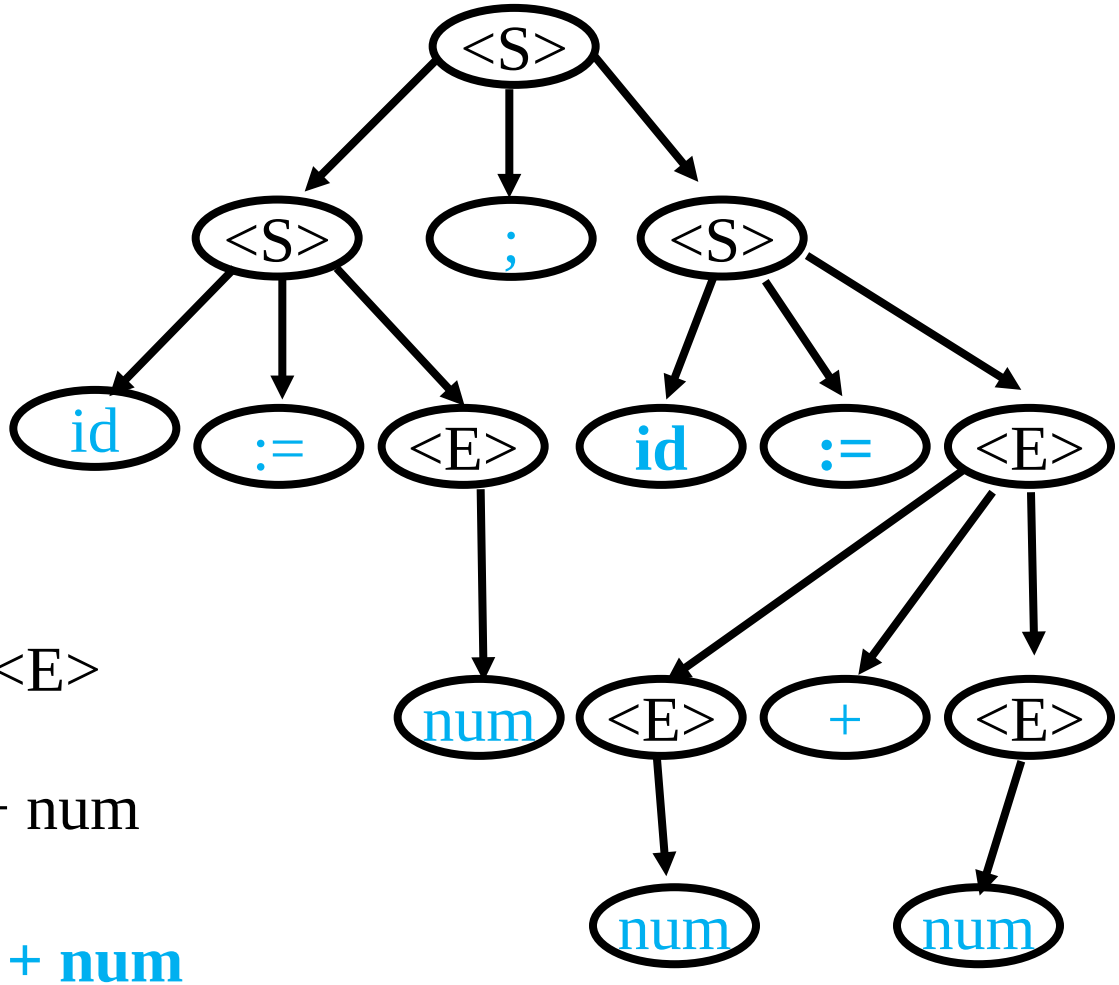
id := <E> **; id :=** <E>

id := num ; id := <E>

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

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

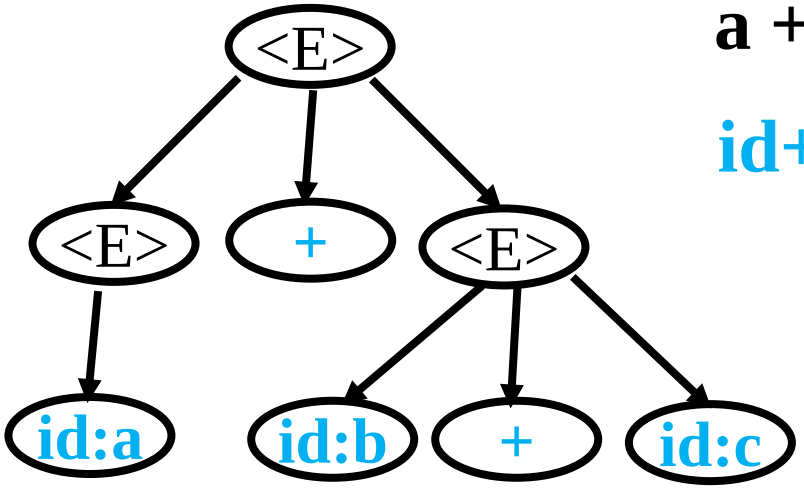
id := num ; id :=num + num



Ambiguous Grammars

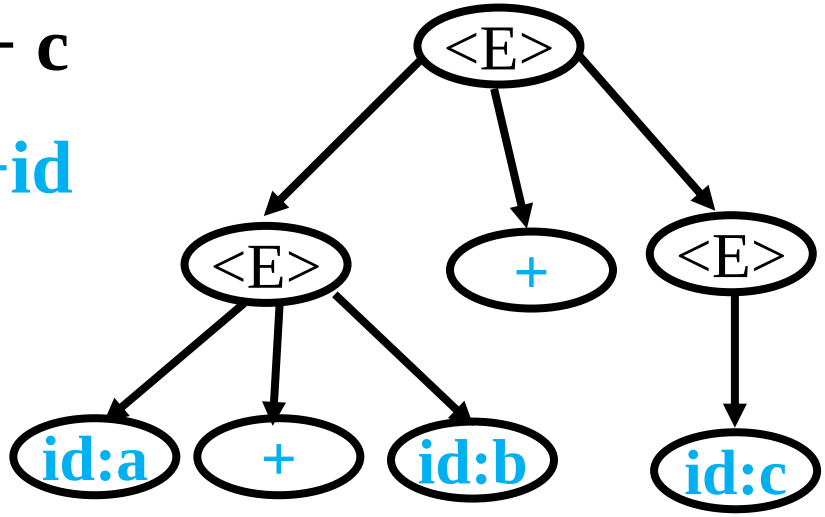
- Two leftmost derivations
- Two rightmost derivations
- Two parse trees

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>)

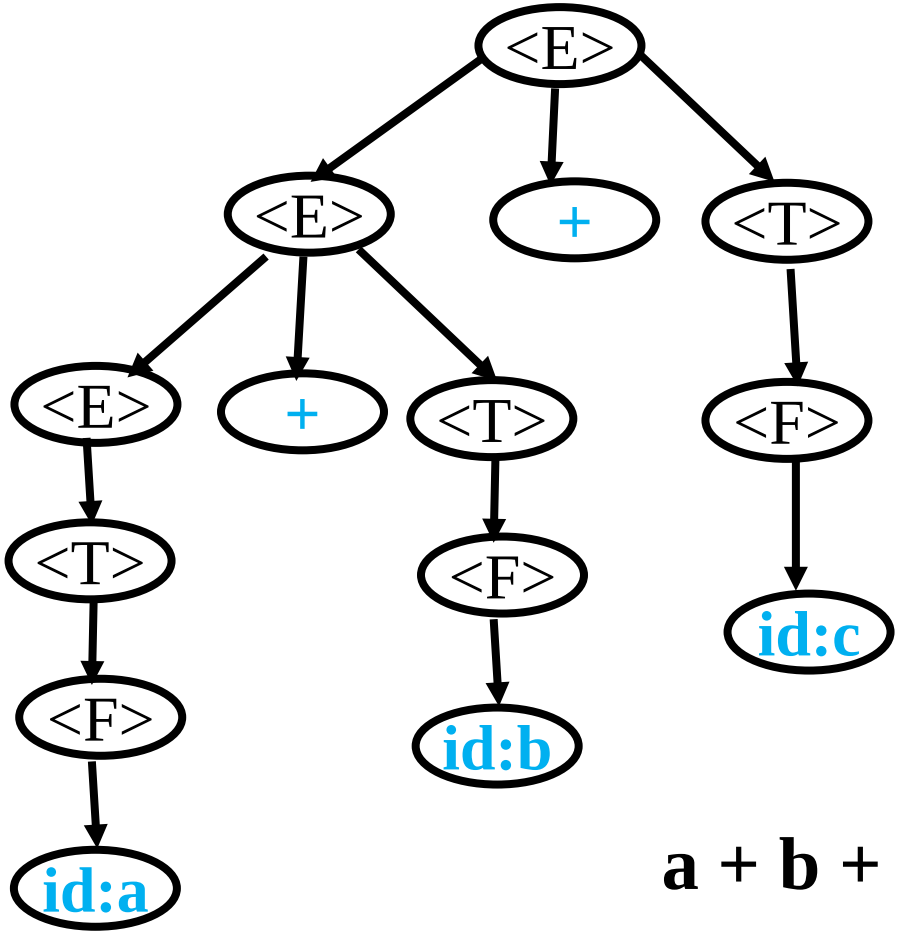
Non-Ambiguous grammar:

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 id(input):
    match(input, id)
    match(input, assign)
    parse_E()
  elif if_tok(input):
    match(input, if_tok)
    match(input, lp)
    parse_E()
    match(input, rp)
    parse_S()
    match(input, else_tok)
    parse_S()
  else:
    syntax_error()
```

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

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

```
def parse_EP():
  if plus(input):
    match(input, plus)
    parse_E()
  elif
    rp(input) or
    else_tok(input) or
    eof(input):
    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 id(input):
    match(input, id)
    match(input, assign)
    parse_E()
  elif if_tok(input):
    match(input, if_tok)
    match(input, lp)
    parse_E()
    match(input, rp)
    parse_S()
    match(input, else_tok)
    parse_S()
  else:
    syntax_error()
```

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

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

```
def parse_EP():
  if plus(input):
    match(input, plus)
    parse_E()
  elif
    rp(input) or
    else_tok(input) or
    eof(input):
    return // ε
  else:
    syntax_error()
```

Nullable Non-Terminals

- Non-terminal A **nullable** iff A $\rightarrow^{*} \varepsilon$
- Computing Nullable:

```
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
- First(α) = { t | $\exists\beta$: $\alpha \rightarrow^* t\ \beta$ }
- Computing First:

```
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 i from 1 to n:
            if {A1, A2 ,… , Ai-1} ⊆ Nullable:
                add First(Ai) to First(A)
```

Example – Computing First

<S> → **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<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 → A_1 A_2 … A_n :
    for i from 1 to n:
      if {A_1, A_2 ,… , A_{i-1}} ⊆ Nullable:
        add First(A_i) to First(A)

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

add id to First(S)
add if to First(S)
add id to First(T)
add lp to First(T)
add plus to First(EP)

add id,lp to first(E)

Fix point reached

Example – Computing First

<S> → **id :=** <E>
<S> → **if (**<E>**)** <S> **else** <S>
<E> → <T> <EP>
<T> → **id** | **(**<E>**)**
<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 → A_1 A_2 … A_n :
    for i from 1 to n:
      if {A_1, A_2 ,… , A_{i-1}} ⊆ Nullable:
        add First(A_i) to First(A)

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

add id to First(S)
add if to First(S)
add id to First(T)
add lp to First(T)
add plus to First(EP)

add id,lp to first(E)

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> →* β <A> t γ}
- Computing Follow:

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

Example – Computing Follow

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

For each symbol X: Follow(X) = ∅
while changes occur do:
  for each rule <A> → $A_1\ A_2\ \ldots\ A_n$:
    for i from 1 to n, for j from i+1 to n:
      if $\{A_{i+1}, \ldots, A_{j-1}\} \subseteq$ Nullable:
        add First(A_j) to Follow(A_i)
      if $\{A_{i+1}, \ldots, A_n\} \subseteq$ Nullable:
        add Follow(A) to Follow(A_i)

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

Follow(t) = ∅ for t in {id, ass, if, lp, rp, else, plus}
Follow(S) = Follow(E) = Follow(T) = Follow(EP) = ∅

add eof to Follow(S)
add rp to Follow(E)
add else to Follow(S)
add rp, plus to Follow(T)

add eof, else to Follow(E)
add eof, else to Follow(T)
add rp, eof, else to Follow(EP)

Fix point reached

Example – Computing Follow

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

```
For each symbol X: Follow(X) = ∅
while changes occur do:
  for each rule <A> → A1 A2 … An :
    for i from 1 to n, for j from i+1 to n:
      if {Ai+1,…, Aj-1} ⊆ Nullable:
        add First(Aj) to Follow(Ai)
      if {Ai+1,…, An} ⊆ Nullable:
        add Follow(A) to Follow(Ai)
```

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

```
Follow(t) = ∅ for t in {id, ass, if, lp, rp, else, plus}
Follow(S) = Follow(E) = Follow(T) = Follow(EP) = ∅

add eof to Follow(S)
add rp to Follow(E)
add else to Follow(S)
add rp, plus to Follow(T)

add eof, else to Follow(E)
add eof, else to Follow(T)
add rp, eof, else to Follow(EP)

Fix point reached
```

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

Select Sets

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

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

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	rp, eof, else
T	no	id, lp	rp, plus, eof, else
EP	yes	plus	rp, eof, 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	rp, eof, 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	rp, eof, else
<EP> → **+** <E>	plus

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

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

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

```
def parse_EP():
  if plus(input):
    match(input, plus)
    parse_E()
  elif
    rp(input) or
    else_tok(input) or
    eof(input):
    return // ε
  else:
    syntax_error()
```