Parsing

Part I

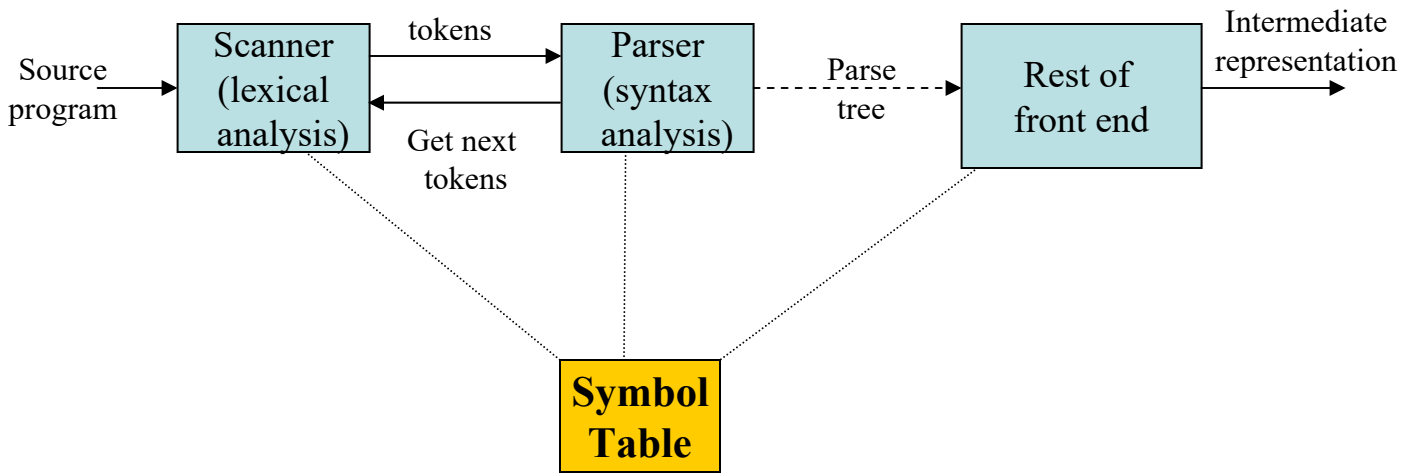
Language and Grammars

- Every (programming) language has precise rules
  - In English:
    - Subject Verb Object
  - In C
    - programs are made of functions
      - » Functions are made of statements etc.

Parsing

- A.K.A. Syntax Analysis
  - Recognize sentences in a language.
  - Discover the structure of a document/program.
  - Construct (implicitly or explicitly) a tree (called as a parse tree) to represent the structure.
  - The above tree is used later to guide translation.

Role of the parser



- Verifies if the string of token can be generated from the grammar
- Error?
    - Report with a good descriptive, helpful message
    - Recover and continue parsing!
- Build a parse tree !!

Rest of Front End

- Collecting token information
- Type checking
- Intermediate code generation

Errors in Programs

- **Lexical**

  if x<1 thenn y = 5:

  “Typos”

- **Syntactic**

  if ((x<1) & (y>5))) ...

  { ... { ... _ ... }

- **Semantic**

  if (x+5) then ...

  Type Errors

  Undefined IDs, etc.

- **Logical Errors**

  if (i<9) then ...

  Should be <= not <

  Bugs

  Compiler cannot detect Logical Errors

Error Detection

- Much responsibility on Parser
  - Many errors are syntactic in nature
  - Precision/ efficiency of modern parsing method
  - Detect the error as soon as possible
- Challenges for error handler in Parser
  - Report error clearly and accurately
  - Recover from error and continue..
  - Should be efficient in processing
- Good news is
  - Simple mechanism can catch most common errors
- Errors don't occur that frequently!!
    - 60% programs are syntactically and semantically correct
    - 80% erroneous statements have only 1 error, 13% have 2
    - Most error are trivial : 90% single token error
    - 60% punctuation, 20% operator, 15% keyword, 5% other error

Adequate Error Reporting is Not a Trivial Task

- Difficult to generate clear and accurate error messages.

Example

```
function foo () {
...
if (...) {
...
} else {
   ...
...
}
<eof>
```

Missing } here

Not detected until here

Example

```
int myVarr;
...
x = myVar;
...
```

Misspelled ID here

Not detected until here

Error Recovery

- After first error recovered
  - Compiler must go on!
    - Restore to some state and process the rest of the input

- **Error-Correcting Compilers**
  - Issue an error message
  - Fix the problem
  - Produce an executable

Example

```
Error on line 23: “myVarr” undefined.
“myVar” was used.
```

May not be a good Idea!!

- Guessing the programmers intention is not easy!

Error Recovery May Trigger More Errors!

- Inadequate recovery may introduce more errors
  - Those were not programmers errors

- Example:

```
int myVar flag ;
...
x := flag;
...
...
while (flag==0)
...
```

Declaration of flag is discarded

Variable flag is undefined

Variable falg is undefined

Too many Error message may be obscuring
- May bury the real message
- Remedy:
  - allow 1 message per token or per statement
  - Quit after a maximum (e.g. 100) number of errors

Error Recovery Approaches: Panic Mode

- Discard tokens until we see a “synchronizing” token.

Example

Skip to next occurrence of
`} end ;`
Resume by parsing the next statement

- The key...
  - Good set of synchronizing tokens
  - Knowing what to do then
- Advantage
  - Simple to implement
  - Does not go into infinite loop
  - Commonly used
- Disadvantage
  - May skip over large sections of source with some errors

Error Recovery Approaches: Phrase-Level Recovery

- Compiler corrects the program
  by deleting or inserting tokens
  ...so it can proceed to parse from where it was.

Example

while (x==4) y:= a + b

Insert do to fix the statement

- The key...
  Don't get into an infinite loop
  ...constantly inserting tokens and never scanning the actual source
- Generally used for error-repairing compilers
  - Difficulty: Point of error detection might be much later the point of error occurrence

Error Recovery Approaches: Error Productions

- Augment the CFG with “Error Productions”
- Now the CFG accepts anything!
- If “error productions” are used...

  Their actions:

  { print (“Error...”) }

- Used with...
  - LR (Bottom-up) parsing
  - Parser Generators

Error Recovery Approaches: Global Correction

- Theoretical Approach
- Find the minimum change to the source to yield a valid program
  - Insert tokens, delete tokens, swap adjacent tokens
- Global Correction Algorithm

  Input: grammatically incorrect input string x; grammar G

  Output: grammatically correct string y

  Algorithm: converts x → y using minimum number changes (insertion, deletion etc.)
- Impractical algorithms - too time consuming

Context Free Grammars (CFG)

- A **context free grammar** is a formal model that consists of:
- **Terminals**

  Keywords

  Token Classes

  Punctuation
- **Non-terminals**

  Any symbol appearing on the lefthand side of any rule
- **Start Symbol**

  Usually the non-terminal on the lefthand side of the first rule
- **Rules (or "Productions")**

  BNF: Backus-Naur Form / Backus-Normal Form

  Stmt **::= if** Expr **then** Stmt **else** Stmt

Rule Alternative Notations

$E \rightarrow E + E$
$E \rightarrow (E)$
$E \rightarrow - E$
$E \rightarrow ID$

$E \rightarrow E + E$
$\rightarrow (E)$
$\rightarrow - E$
$\rightarrow ID$

$E \rightarrow E + E$
$\mid (E)$
$\mid - E$
$\mid ID$

$E \rightarrow E + E \mid (E) \mid - E \mid ID$

All Notations are Equivalent

Notational Conventions

Terminals

a b c ...

Nonterminals

A B C ...

S

Expr

Grammar Symbols (Terminals or Nonterminals)

X Y Z U V W ...

Strings of Symbols

α β γ ...

A sequence of zero
Or more terminals
And nonterminals

Strings of Terminals

x y z u v w ...

Including ε

Examples

$A \rightarrow \alpha B$

A rule whose righthand side ends with a nonterminal

$A \rightarrow x\ \alpha$

A rule whose righthand side begins with a string of terminals (call it "x")

Derivations

1. $E \rightarrow E + E$
2. $\rightarrow E * E$
3. $\rightarrow (E)$
4. $\rightarrow - E$
5. $\rightarrow ID$

A "Derivation" of "(id*id)"

$$E \Rightarrow (E) \Rightarrow (E*E) \Rightarrow \underbrace{(id*E)} \Rightarrow (id*id)$$

"Sentential Forms"

A sequence of terminals and nonterminals in a derivation

(id*E)

Derivation

If $A \rightarrow \beta$ is a rule, then we can write

$$\underbrace{\alpha A \gamma}_{} \Rightarrow \alpha\beta\gamma$$

Any sentential form containing a nonterminal (call it A)

... such that A matches the nonterminal in some rule.

Derives in zero-or-more steps $\Rightarrow^*$

$$E \Rightarrow^* \texttt{(id*id)}$$

If $\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow \gamma$, then $\alpha \Rightarrow^* \gamma$

Derives in one-or-more steps $\Rightarrow^+$

Given

G A grammar

S The Start Symbol

Define

L(G) The language generated

$L(G) = \{ w \mid S \Rightarrow^{+} w \}$

"Equivalence" of CFG's

If two CFG's generate the same language, we say they are "equivalent."

$G_1 \approx G_2$ whenever $L(G_1) = L(G_2)$

In making a derivation...

Choose which nonterminal to expand

Choose which rule to apply

Leftmost Derivation

In a derivation... always expand the *leftmost* nonterminal.

$$
\begin{aligned}
& E \\
\Rightarrow\ & E+E \\
\Rightarrow\ & (E)+E \\
\Rightarrow\ & (E*E)+E \\
\Rightarrow\ & (\underline{id}*E)+E \\
\Rightarrow\ & (\underline{id}*\underline{id})+E \\
\Rightarrow\ & (\underline{id}*\underline{id})+\underline{id}
\end{aligned}
$$

1.	E	$\rightarrow$ E + E
2.		$\rightarrow$ E * E
3.		$\rightarrow$ (E)
4.		$\rightarrow$ - E
5.		$\rightarrow$ ID

Let $\Rightarrow_{LM}$ denote a step in a leftmost derivation ($\Rightarrow_{LM}^{*}$ means zero-or-more steps)

At each step in a leftmost derivation, we have

$$w A \gamma \Rightarrow_{LM} w \beta \gamma \qquad \text{where } A \rightarrow \beta \text{ is a rule}$$

(Recall that w is a string of terminals.)

Each sentential form in a leftmost derivation is called a "left-sentential form."

If $S \Rightarrow_{LM}^{*} \alpha$ then we say α is a "left-sentential form."

Rightmost Derivation

In a derivation... always expand the *rightmost* nonterminal.

$$
\begin{aligned}
& E \\
\Rightarrow\ & E+E \\
\Rightarrow\ & E+\underline{id} \\
\Rightarrow\ & (E)+\underline{id} \\
\Rightarrow\ & (E*E)+\underline{id} \\
\Rightarrow\ & (E*\underline{id})+\underline{id} \\
\Rightarrow\ & (\underline{id}*\underline{id})+\underline{id}
\end{aligned}
$$

1.	E	$\rightarrow$ E + E
2.		$\rightarrow$ E * E
3.		$\rightarrow$ (E)
4.		$\rightarrow$ - E
5.		$\rightarrow$ ID

Let $\Rightarrow_{RM}$ denote a step in a rightmost derivation ($\Rightarrow_{RM}^{*}$ means zero-or-more steps)

At each step in a rightmost derivation, we have

$\alpha A w \Rightarrow_{RM} \alpha\beta w$ where $A \rightarrow \beta$ is a rule

(Recall that w is a string of terminals.)

Each sentential form in a rightmost derivation is called a "right-sentential form."

If $S \Rightarrow_{RM}^{*} \alpha$ then we say α is a "right-sentential form."

Parse Tree

Two choices at each step in a derivation...

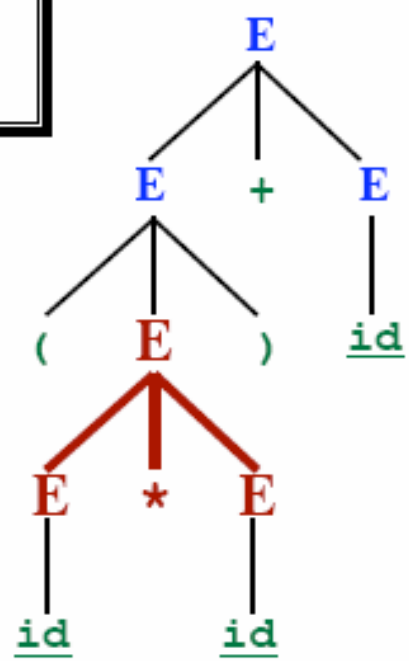
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

E
⇒ E+E
⇒ (E) +E
⇒ (E*E) +E
⇒ (id*E) +E
⇒ (id*id) +E
⇒ (id*id) +id

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID



Parse Tree

Two choices at each step in a derivation...

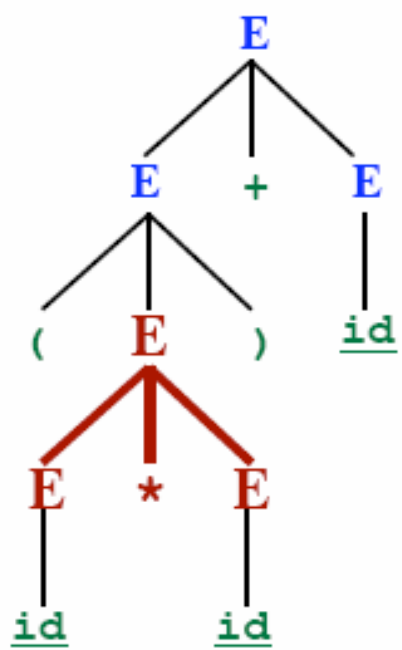
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID



Parse Tree

Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

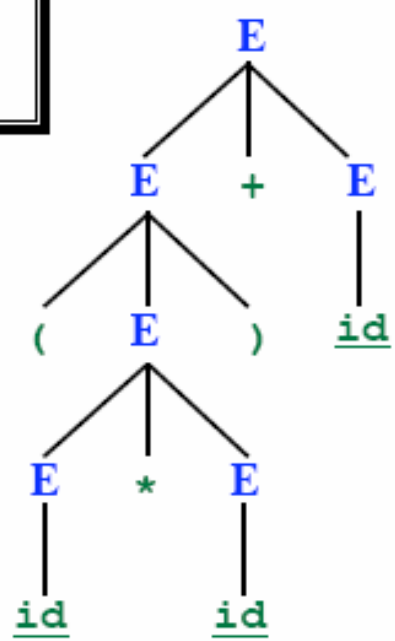
Leftmost Derivation:

E
⇒ E+E
⇒ (E)+E
⇒ (E*E)+E
⇒ (id*E)+E
⇒ (id*id)+E
⇒ (id*id)+id

Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID



Parse Tree

Given a leftmost derivation, we can build a parse tree.
Given a rightmost derivation, we can build a parse tree.

Lefttmost Derivation of `(id*id)+id`

Rightmost Derivation of `(id*id)+id`

Same Parse Tree

Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

Therefore:

- Several leftmost derivations
- Several rightmost derivations

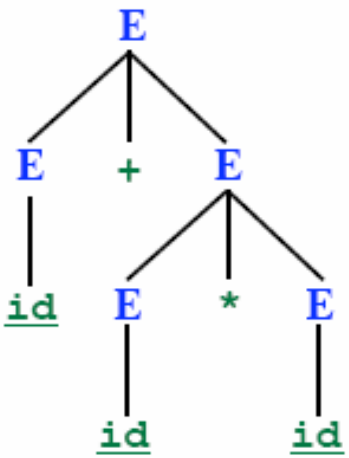
Ambiguous Grammar

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Input: id+id*id

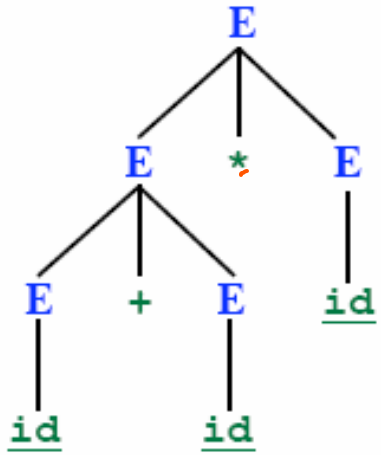
Leftmost Derivation #1

E
⇒ E+E
⇒ id+E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Leftmost Derivation #2

E
⇒ E*E
⇒ E+E*E
⇒ id+E*E
⇒ id+id*E
⇒ id+id*id



Ambiguous Grammar

- More than one Parse Tree for some sentence.
  - The grammar for a programming language may be ambiguous
  - Need to modify it for parsing.

- Also: Grammar may be left recursive.
- Need to modify it for parsing.

Parsing

Part II

Writing Grammars

When writing a grammar (or RE) for some language, the following must be true:

1. All strings generated are in the language.
2. Your grammar produces all strings in the language.

Example:

$S \rightarrow (S)\ S \mid \varepsilon$

Generates all strings of balanced parentheses

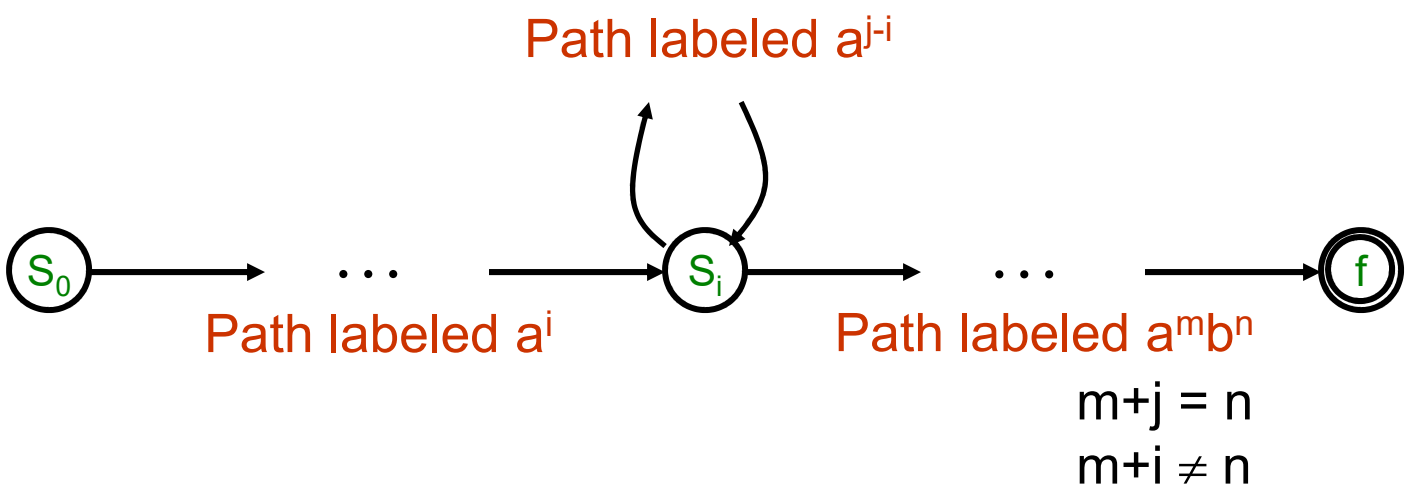
Using induction show that

Every sentence derivable from S is balanced

Every balanced string is derivable from S

- $L=\{ a^n b^n \mid n \geq 1\}$

- Show that L can be described by a grammar not by a regular expression
- Construct a DFA D with k states to accept L
- For $a^n b^n$ ($n>k$) some state (s_i) of D must be entered twice



Elimination of Ambiguity

Ambiguous Grammar

- A Grammar is ambiguous if there are multiple parse trees for the same sentence
- For the most parsers, the grammar must be unambiguous

Unambiguous grammar

unique selection of the parse tree for a sentence

Disambiguation

- Express Preference for one parse tree over others
  - Add disambiguating rule into the grammar

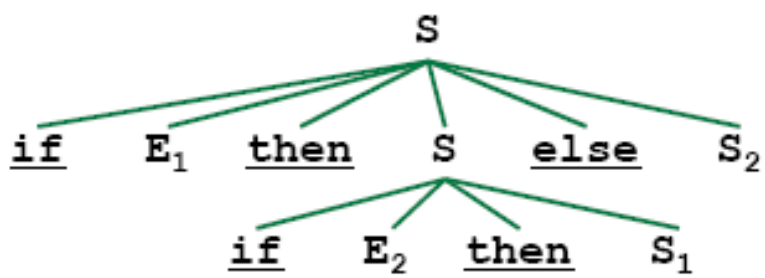
Dangling-else grammar

This grammar is ambiguous!

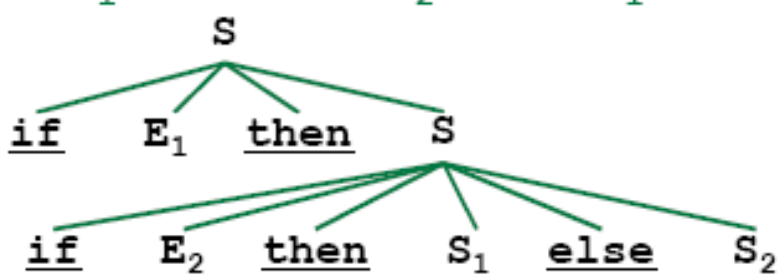
Stmt → **if** Expr **then** Stmt
→ **if** Expr **then** Stmt **else** Stmt
→ ...Other Stmt Forms...

Example String: **if** E_1 **then if** E_2 **then** S_1 **else** S_2

Interpretation #1: **if** E_1 **then** (**if** E_2 **then** S_1) **else** S_2



Interpretation #2: **if** E_1 **then** (**if** E_2 **then** S_1 **else** S_2)



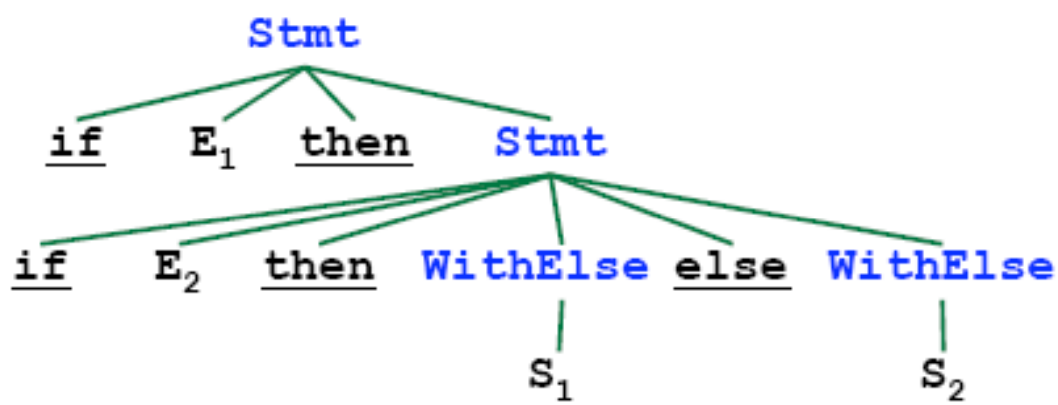
Dangling-else grammar

Goal: "Match **else**-clause to the closest **if** without an **else**-clause already."
Solution:

Stmt	→	**if** Expr **then** Stmt
	→	**if** Expr **then** WithElse **else** Stmt
	→	...Other Stmt Forms...
WithElse	→	**if** Expr **then** WithElse **else** WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between **then** and **else** must have an **else**.
i.e., the Stmt must not end with "**then** Stmt".

Interpretation #2: **if** E_1 **then** (**if** E_2 **then** S_1 **else** S_2)



Dangling-else grammar

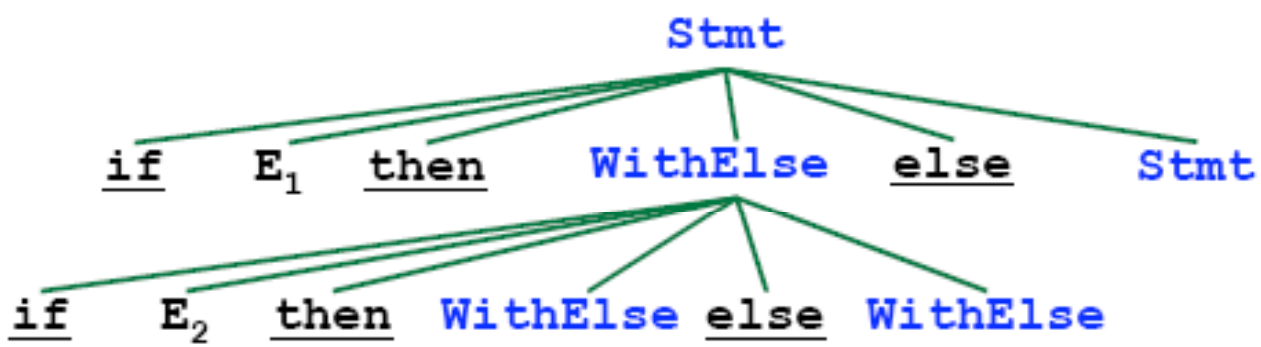
Goal: "Match **else**-clause to the closest **if** without an **else**-clause already."

Solution:

Stmt	→	**if** Expr **then** Stmt
	→	**if** Expr **then** WithElse **else** Stmt
	→	...Other Stmt Forms...
WithElse	→	**if** Expr **then** WithElse **else** WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between **then** and **else** must have an **else**.
i.e., the Stmt must not end with "**then** Stmt".

Interpretation #1: **if** E_1 **then** (**if** E_2 **then** S_1) **else** S_2



Left Recursion

Whenever

$$A \Rightarrow^{+} A\alpha$$

Simplest Case: Immediate Left Recursion

Given:

$$A \rightarrow A\alpha \mid \beta$$

Transform into:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

where A' is a new nonterminal

More General (but still immediate):

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$$

Transform into:

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' \mid \ldots$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \ldots \mid \varepsilon$$

Immediate Left Recursion Elimination: example

- Grammar

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid \textbf{id}$

- **Left recursion Eliminated**

  $E \rightarrow T E'$

  $E' \rightarrow + T E' \mid \varepsilon$

  $T \rightarrow F T'$

  $T' \rightarrow * F T' \mid \varepsilon$

  $F \rightarrow (E) \mid \textbf{id}$

Left Recursion in More Than One Step

Example:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid e$

Is A left recursive? Yes.

Is S left recursive? Yes, but not immediate left recursion. $S \Rightarrow Af \Rightarrow Sdf$

Approach:

Look at the rules for S only (ignoring other rules)... No left recursion.

Look at the rules for A...

Do any of A's rules start with S? Yes.

$A \rightarrow Sd$

Get rid of the S. Substitute in the righthand sides of S.

$A \rightarrow Afd \mid bd$

The modified grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Afd \mid bd \mid e$

Now eliminate immediate left recursion involving A.

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid eA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow A\underline{c} \mid S\underline{d} \mid B\underline{e}$

$B \rightarrow A\underline{g} \mid S\underline{h} \mid \underline{k}$

So Far:

$S \rightarrow A\underline{f} \mid \underline{b}$

$A \rightarrow \underline{bd}A' \mid B\underline{e}A'$

$A' \rightarrow \underline{c}A' \mid \underline{fd}A' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow Ag \mid Sh \mid k$

Look at the B rules next; Does any righthand side start with "S"?

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow Ag \mid Afh \mid bh \mid k$

Substitute, using the rules for "S"

$Af\ldots \mid b\ldots$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \epsilon$

$B \rightarrow Ag \mid Afh \mid bh \mid k$

Does any righthand side start with "A"?

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**
A → A**c** | S**d** | B**e**
B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**
A → **bd**A' | B**e**A'
A' → **c**A' | **fd**A' | ε
B → **bd**A'**g** | B**e**A'**g** | A**fh** | **bh** | **k**

Substitute, using the rules for "A"
bdA'... | B**e**A'...

Left Recursion in More Than One Step

The Original Grammar:

S → A**f** | **b**
A → A**c** | S**d** | B**e**
B → A**g** | S**h** | **k**

So Far:

S → A**f** | **b**
A → **bd**A' | B**e**A'
A' → **c**A' | **fd**A' | ε
B → **bd**A'**g** | B**e**A'**g** | **bd**A'**fh** | B**e**A'**fh** | **bh** | **k**

Substitute, using the rules for "A"
bdA'... | B**e**A'...

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be$

$B \rightarrow Ag \mid Sh \mid k$

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow bdA'g \mid BeA'g \mid bdA'fh \mid BeA'fh \mid bh \mid k$

Finally, eliminate any immediate Left recursion involving "B"

Next Form

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA'$

$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow bdA'gB' \mid bdA'fhB' \mid bhB' \mid kB'$

$B' \rightarrow eA'gB' \mid eA'fhB' \mid \varepsilon$

Left Recursion in More Than One Step

The Original Grammar:

$S \rightarrow Af \mid b$

$A \rightarrow Ac \mid Sd \mid Be \mid C$

$B \rightarrow Ag \mid Sh \mid k$

$C \rightarrow Bkm A \mid AS \mid j$

If there is another nonterminal, then do it next.

So Far:

$S \rightarrow Af \mid b$

$A \rightarrow bdA' \mid BeA' \mid CA'$

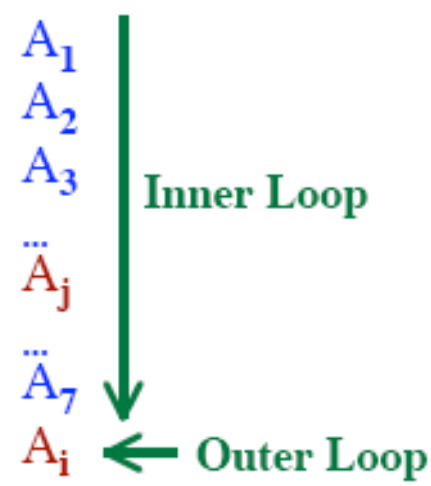
$A' \rightarrow cA' \mid fdA' \mid \varepsilon$

$B \rightarrow bdA'gB' \mid bdA'fhB' \mid bhB' \mid kB' \mid CA'gB' \mid CA'fhB'$

$B' \rightarrow eA'gB' \mid eA'fhB' \mid \varepsilon$

Algorithm for Eliminating Left Recursion

```
Assume the nonterminals are ordered A1, A2, A3,...
        (In the example: S, A, B)
for each nonterminal Ai (for i = 1 to N) do
  for each nonterminal Aj (for j = 1 to i-1) do
    Let Aj → β1 | β2 | β3 | ... | βN be all the rules for Aj
    if there is a rule of the form
       Ai → Ajα
    then replace it by
       Ai → β1α | β2α | β3α | ...  | βNα
    endIf
  endFor
  Eliminate immediate left recursion
        among the Ai rules
endFor
```



Left Factoring

Problem:

Stmt → `if` Expr `then` Stmt `else` Stmt

→ `if` Expr `then` Stmt

→ OtherStmt

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

Stmt → `if` Expr `then` Stmt ElsePart

→ OtherStmt

ElsePart → `else` Stmt | ε

General Approach:

Before: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Left Factoring

Problem:

$\underbrace{\text{Stmt}}_{A} \rightarrow \underbrace{\texttt{if}\ \text{Expr}\ \texttt{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\texttt{else}\ \text{Stmt}}_{\beta_1}$

$\rightarrow \underbrace{\texttt{if}\ \text{Expr}\ \texttt{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\varepsilon}_{\beta_2}$

$\rightarrow \underbrace{\text{OtherStmt}}_{\delta_1}$

With predictive parsing, we need to know which rule to use!
(While looking at just the next token)

Solution:

$\underbrace{\text{Stmt}}_{A} \rightarrow \underbrace{\texttt{if}\ \text{Expr}\ \texttt{then}\ \text{Stmt}}_{\alpha}\ \underbrace{\text{ElsePart}}_{C}$

$\rightarrow \underbrace{\text{OtherStmt}}_{\delta_1}$

$\underbrace{\text{ElsePart}}_{C} \rightarrow \underbrace{\texttt{else}\ \text{Stmt}}_{\beta_1} \mid \underbrace{\varepsilon}_{\beta_2}$

General Approach:

Before: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \ldots \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

After: $A \rightarrow \alpha C \mid \delta_1 \mid \delta_2 \mid \delta_3 \mid \ldots$

$C \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \ldots$

Parsing

Part III

Top Down Parsing

- Find a left-most derivation
- Find (build) a parse tree
- Start building from the root and work down…
- As we search for a derivation
  - Must make choices:
    - Which rule to use
    - Where to use it

- May run into problems!!

Top-Down Parsing

- Recursive-Descent Parsing
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient
- Predictive Parsing
  - no backtracking
  - efficient
  - needs a special form of grammars (LL(1) grammars).
  - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.
  - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.

Recursive Descent Parsing (Backtracking)

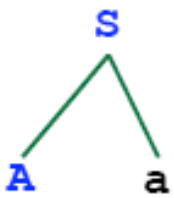
Input: aabbde

S

1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

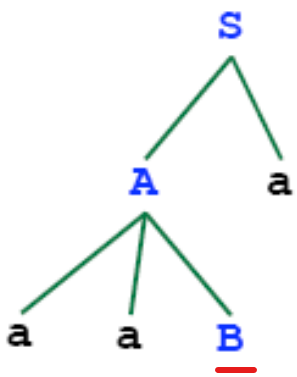
Input: **aabbde**



1. S → Aa
2. → Ce
3. A → aaB ✓
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

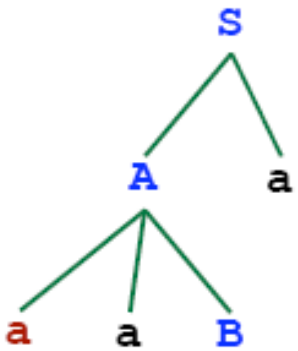
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

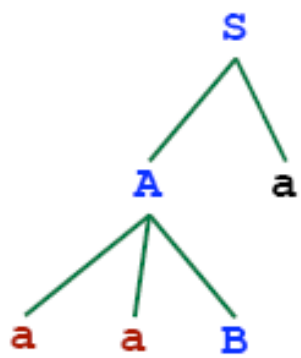
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

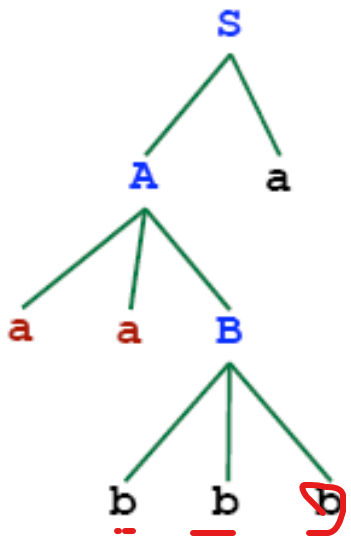
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

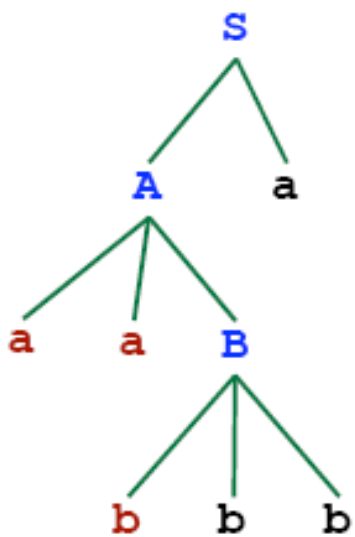
Input: aabbde



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

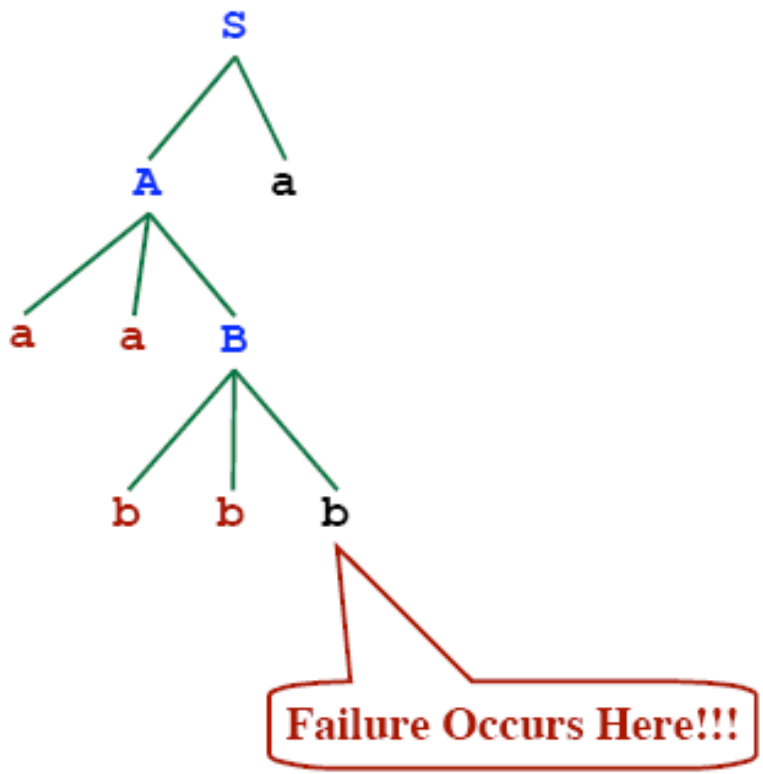
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

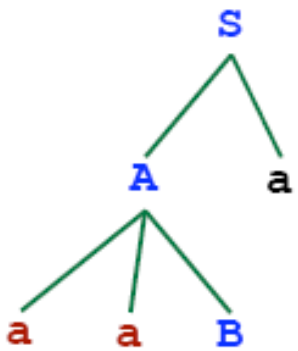
Input: **aabbde**



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

Input: **aabbde**

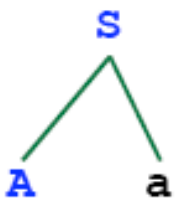


1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

We need an ability to back up in the input!!!

Recursive Descent Parsing (Backtracking)

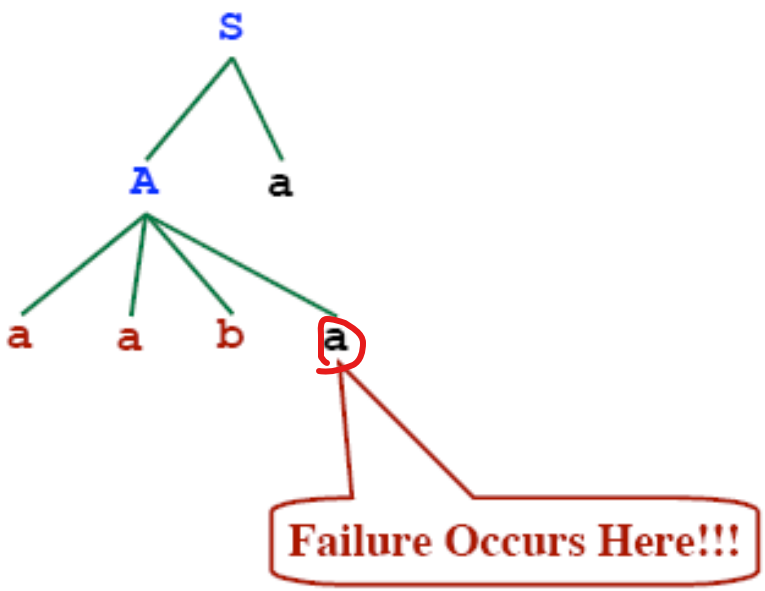
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba** ✓
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

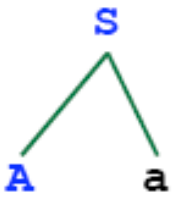
Input: **aabbde**



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

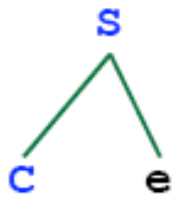
Input: **aabbde**

S

1. S → Aa
2. → Ce ✓
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

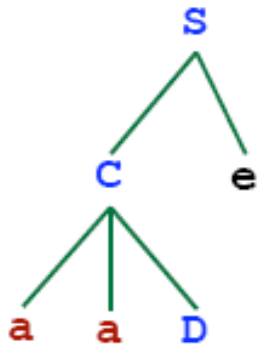
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

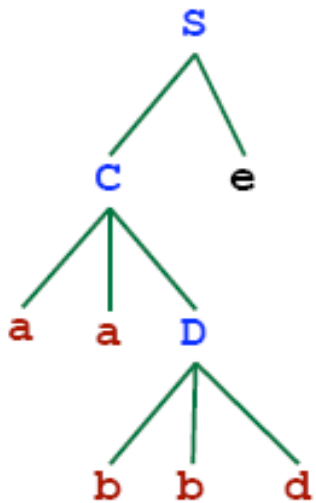
Input: **aabbde**



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Recursive Descent Parsing (Backtracking)

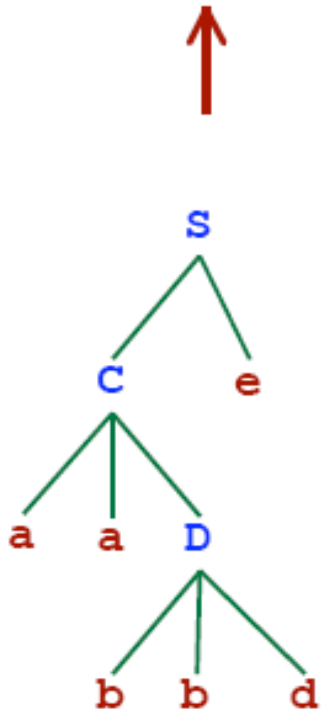
Input: **aabbde**



1. S → A**a**
2. → C**e**
3. A → **aa**B
4. → **aaba**
5. B → **bbb**
6. C → **aa**D
7. D → **bbd**

Recursive Descent Parsing (Backtracking)

Input: **aabbde**



1. S → Aa
2. → Ce
3. A → aaB
4. → aaba
5. B → bbb
6. C → aaD
7. D → bbd

Successfully parsed!!

Recursive-Descent Parsing Algorithm

- A recursive-descent parsing program consists of a set of procedures – one for each non-terminal
- Execution begins with the procedure for the start symbol
  - Announces success if the procedure body scans the entire input

```
void A(){
    for (j=1 to t){ /* assume there is t number of A-productions */
        Choose a A-production, A_j → X_1 X_2 … X_k;
        for (i=1 to k){
            if (X_i is a non-terminal)
                call procedure X_i();
            else if (X_i equals the current input symbol a)
                advance the input to the next symbol;
            else backtrack in input and reset the pointer
        }
    }
}
```

Predictive Parser

When re-writing a non-terminal in a derivation step, a predictive parser can uniquely choose a production rule by just looking the current symbol in the input string.

$A \rightarrow \alpha_1 \mid \ldots \mid \alpha_n$

input: ... a .....

current token

Predictive Parser (example)

stmt → `if` ..... |
`while` ..... |
`begin` ..... |
`for` .....

- When we are trying to write the non-terminal *stmt*, if the current token is `if` we have to choose first production rule.
- When we are trying to write the non-terminal *stmt*, we can uniquely choose the production rule by just looking the current token.
- We eliminate the left recursion in the grammar, and left factor it. But it may not be suitable for predictive parsing (not LL(1) grammar).

Recursive Predictive Parsing

- Each non-terminal corresponds to a procedure.

Ex: $A \rightarrow aBb$ (This is only the production rule for A)

```
proc A {
        - match the current token with a, and move to the next token;
        - call 'B';
        - match the current token with b, and move to the next token;
    }
```

Recursive Predictive Parsing (cont.)

$A \rightarrow aBb \mid bAB$

```
proc A {
    case of the current token {
        'a':  - match the current token with a, and move to the next token;
              - call 'B';
              - match the current token with b, and move to the next token;

        'b':  - match the current token with b, and move to the next token;
              - call 'A';
              - call 'B';
    }
}
```

Recursive Predictive Parsing (cont.)

- When to apply ε-productions.

  $A \rightarrow aA \mid bB \mid \varepsilon$

- If all other productions fail, we should apply an ε-production. For example, if the current token is not a or b, we may apply the ε-production.
- Most correct choice: We should apply an ε-production for a non-terminal A when the current token is in the follow set of A (which terminals can follow A in the sentential forms).

Recursive Predictive Parsing (Example)

$A \rightarrow aBe \mid cBd \mid C$

$B \rightarrow bB \mid \varepsilon$

$C \rightarrow f$

```
proc A {
    case of the current token {
        a:  - match the current token with a,
              and move to the next token;
            - call B;
            - match the current token with e,
              and move to the next token;
        c:  - match the current token with c,
              and move to the next token;
            - call B;
            - match the current token with d,
              and move to the next token;
        f:  - call C
    }
}
```

first set of C

```
proc C {   match the current token with f,
           and move to the next token; }
```

```
proc B {
    case of the current token {
        b:  - match the current token with b,
              and move to the next token;
            - call B
        e,d:  do nothing
    }
}
```

follow set of B

First Function

Let α be a string of symbols (terminals and nonterminals)

Define:

FIRST (α) = The set of terminals that could occur first in any string derivable from α

$= \{ a \mid \alpha \Rightarrow^* aw$, plus ε if $\alpha \Rightarrow^* \varepsilon \}$

Example:

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

FIRST (F) = { (, id }
FIRST (T') = { *, ε}
FIRST (T) = { (, id }
FIRST (E') = { +, ε}
FIRST (E) = { (, id }

Computing the First Function

For all symbols X in the grammar...

```
if X is a terminal then
  FIRST(X) = { X }

if X → ε is a rule then
  add ε to FIRST(X)

if X → Y1 Y2 Y3 ... YK is a rule then
  if a ∈ FIRST(Y1) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and a ∈ FIRST(Y2) then
    add a to FIRST(X)
  if ε ∈ FIRST(Y1) and ε ∈ FIRST(Y2) and a ∈ FIRST(Y3) then
    add a to FIRST(X)
  ...
  if ε ∈ FIRST(Yi) for all Yi then
    add ε to FIRST(X)
```

Repeat until nothing more can be added to any sets.

To Compute the FIRST(X1X2X3...XN)

```
Result = {}
Add everything in FIRST(X1), except ε, to result
if ε ∈ FIRST(X1) then
  Add everything in FIRST(X2), except ε, to result
  if ε ∈ FIRST(X2) then
    Add everything in FIRST(X3), except ε, to result
    if ε ∈ FIRST(X3) then
      Add everything in FIRST(X4), except ε, to result
      ...
        if ε ∈ FIRST(XN-1) then
          Add everything in FIRST(XN), except ε, to result
          if ε ∈ FIRST(XN) then
            // Then X1 ⇒* ε, X2 ⇒* ε, X3 ⇒* ε, ... XN ⇒* ε
            Add ε to result
          endIf
        endIf
      ...
    endIf
  endIf
endIf
```

First - Example

- P → i | c | n T S
- Q → P | a S | b S c S T
- R → b | ε
- S → c | R n | ε
- T → R S q

P = i, c, n

Q = a, b, i, c, n

R = b, ε

S = c, ε, b, n

T = b, c, n, q

- FIRST(P) = {i,c,n}
- FIRST(Q) = {i,c,n,a,b}
- FIRST(R) = {b,ε}
- FIRST(S) = {c,b,n,ε}
- FIRST(T) = {b,c,n,q}

First - Example

- $S \rightarrow a\ S\ e \mid S\ T\ S$
- $T \rightarrow R\ S\ e \mid Q$
- $R \rightarrow r\ S\ r \mid \varepsilon$
- $Q \rightarrow S\ T \mid \varepsilon$

- $FIRST(S) = \{a\}$
- $FIRST(R) = \{r, \varepsilon\}$
- $FIRST(T) = \{r, a, \varepsilon\}$
- $FIRST(Q) = \{a, \varepsilon\}$

2. FOLLOW Sets

- FOLLOW(A) is the set of terminals (including end marker of input - $) that may follow non-terminal A in some sentential form.
- FOLLOW(A) = $\{c \mid S \Rightarrow^{+} \ldots Ac \ldots\} \cup \{\$\}$ if $S \Rightarrow^{+} \ldots A$
- For example, consider $L \Rightarrow^{+} (())(L)L$

  Both ')' and end of file can follow L
- NOTE: ε is ***never*** in FOLLOW sets

Computing FOLLOW(A)

1. If A is start symbol, put \$ in FOLLOW(A)
2. Productions of the form $B \rightarrow \alpha\ A\ \beta$,

   Add FIRST(β) – {ε} to FOLLOW(A)

INTUITION: Suppose $B \rightarrow AX$ and FIRST(X) = {c}

$$S \Rightarrow^{+} \alpha\ B\ \beta \Rightarrow \alpha\ A\ X\ \beta \Rightarrow^{+} \alpha\ A\ c\ \delta\ \beta$$

= FIRST(X)

3. Productions of the form $B \rightarrow \alpha A$ or

$B \rightarrow \alpha A \beta$ where $\beta \Rightarrow^* \varepsilon$

Add FOLLOW(B) to FOLLOW(A)

INTUITION:

- Suppose $B \rightarrow Y A$

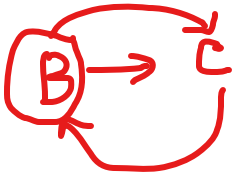
  $S \Rightarrow^+ \alpha B \beta \Rightarrow \alpha Y A \beta$

  FOLLOW(B)

- Suppose $B \rightarrow A X$ and $X \Rightarrow^* \lambda$

  $S \Rightarrow^+ \alpha B \beta \Rightarrow \alpha A X \beta \Rightarrow^* \alpha A \beta$

  FOLLOW(B)



Example

- S → a S e | B
- B → b B C f | C
- C → c C g | d | ε

- FIRST(C) = {c,d,~~ε~~}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) = g, f, c, d, e, $
- FOLLOW(B) = {c, d, f, $, e}
- FOLLOW(S) = {$}, ~~e~~

Using rule #1

Assume the first non-terminal is the start symbol

Example

- S → a <u>S e</u> | B
- B → b <u>B C f</u> | C
- C → c <u>C g</u> | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) = {f,g}
- FOLLOW(B) = {c,d,f}
- FOLLOW(S) = {$,e}

Using rule #2

Example

- S → a S e | <u>B</u>
- B → b B C f | <u>C</u>
- C → c C g | d | ε

- FIRST(C) = {c,d,ε}
- FIRST(B) = {b,c,d,ε}
- FIRST(S) = {a,b,c,d,ε}

- FOLLOW(C) =

  $\{f,g\} \cup \text{FOLLOW}(B)$
  $= \{c,d,e,f,g,\$\}$

- FOLLOW(B) =

  $\{c,d,f\} \cup \text{FOLLOW}(S)$
  $= \{c,d,e,f,\$\}$

- FOLLOW(S) = {$, e }

Using rule #3

Example

- S → (A) | ε
- A → T E
- E → & T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {&, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) = {$}
- FOLLOW(A) = {)}
- FOLLOW(E) =)
- FOLLOW(T) = &,)

Example

- S → (A) | ε
- A → T E
- E → & T E | ε
- T → (A) | a | b | c

- FIRST(T) = {(,a,b,c}
- FIRST(E) = {&, ε }
- FIRST(A) = {(,a,b,c}
- FIRST(S) = {(, ε}

- FOLLOW(S) = {$}
- FOLLOW(A) = {) }
- FOLLOW(E) =

FOLLOW(A) = {) }

- FOLLOW(T) =

FIRST(E) ∪ FOLLOW(A) ∪ FOLLOW(E) = {&,)}

Predictive Parsing

Will never backtrack!

Requirement:

For every rule:

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3 \mid \ldots \mid \alpha_N$$

We must be able to choose the correct alternative by looking only at the next symbol

May peek ahead to the next symbol (token).

Example

A → aB
→ cD
→ E

Assuming a,c $\notin$ FIRST (E)

Example

Stmt → `if` Expr ...
→ `for` LValue ...
→ `while` Expr ...
→ `return` Expr ...
→ ID ...

Predictive Parsing

- **LL(1) Grammars**
  - Can do predictive parsing
  - Can select the right rule
  - Looking at only the next 1 input symbol
    - First L : Left to Right Scanning
    - Second L: Leftmost derivation
    - 1 : one input symbol look-ahead for predictive decision

- **LL(k) Grammars**
  - Can do predictive parsing
  - Can select the right rule
  - Looking at only the next k input symbols

- **Techniques to modify the grammar:**
  - Left Factoring
  - Removal of Left Recursion

- **LL(k) Language**
  - Can be described with an LL(k) grammar

Table Driven Predictive Parsing

Assume that the grammar is LL(1)

i.e., Backtracking will never be needed

Always know which righthand side to choose (with one look-ahead)

- No Left Recursion
- Grammar is Left-Factored.

Example

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \varepsilon$

$F \rightarrow (\ E\ ) \mid \underline{id}$

Term ...

+Term +Term + ... +Term

Factor ...

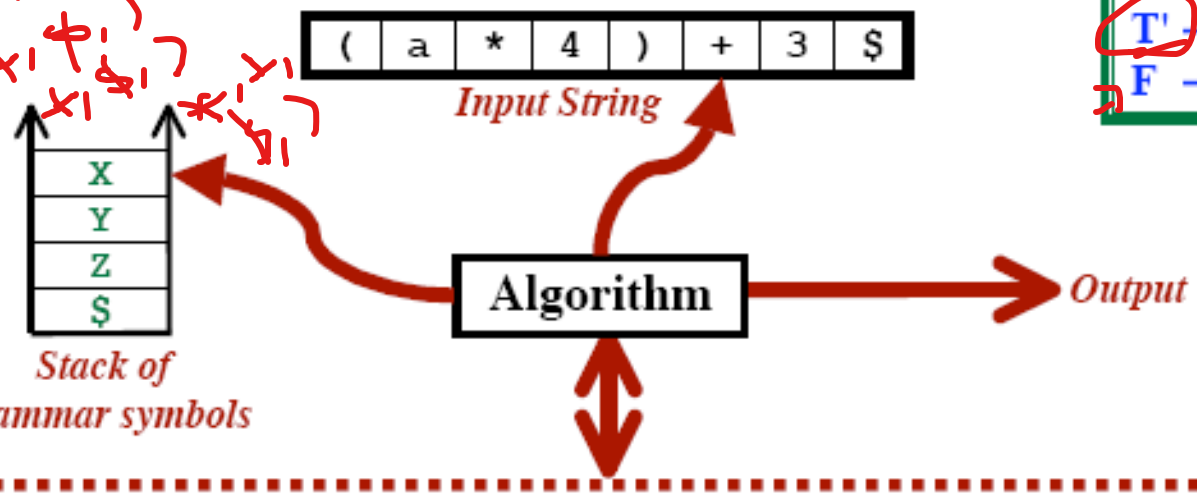
*** Factor * Factor * ... * Factor**

Step 1: From grammar, construct table.

Step 2: Use table to parse strings.

Table Driven Predictive Parsing

	Fir	Fol
E	(, id	$,)
E'	+, ε	$,)
T	(, id	+, $,)
T'	*, ε	+, $,)
F	(, id	*, +, $,)



$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

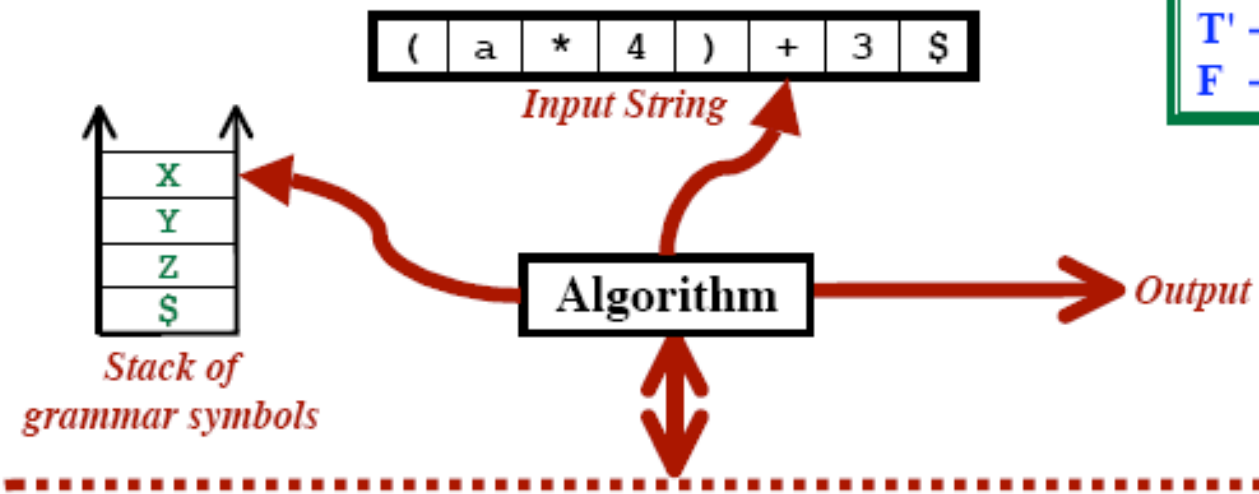
Pre-Computed Table:

3.

	(	id	+	*	)	$
E	E → TE	-				
E'			E' → +TE'		E' → ε	—
T	—	—				
T'			—	*	—	—
F	—	—				

Table Driven Predictive Parsing

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$



Pre-Computed Table:

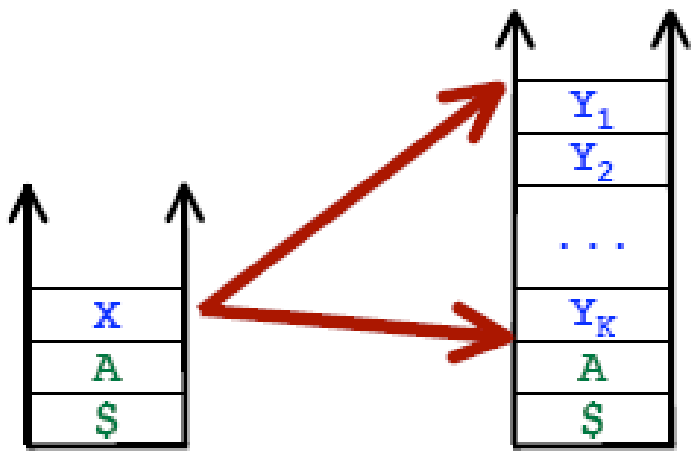
Input Symbols, plus $

Nonterminals

	id	+	*	(	)	$
E	E→TE			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing Algorithm

```
Set input ptr to first symbol; Place $ after last input symbol
Push $
Push S
repeat
  X = stack top
  a = current input symbol
  if X is a terminal or X = $ then
    if X == a then
      Pop stack
      Advance input ptr
    else
      Error
    endIf
  elseIf Table[X,a] contains a rule then   // call it X → Y1 Y2 ... YK
    Pop stack
    Push YK
    ...
    Push Y2
    Push Y1
    Print ("X → Y1 Y2 ... YK")
  else   // Table[X,a] is blank
    Syntax Error
  endIf
until X == $
```



Predictive Parsing

Input:

(id*id)+id

Output:

Example

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

(	id	*	id	)	+	id	

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

(	id	*	id	)	+	id	$

Add $ to end of input
Push $
Push E

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

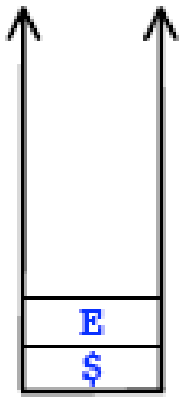
Input:

(id*id)+id

Output:

Example

E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id



(	id	*	id	)	+	id	$

Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

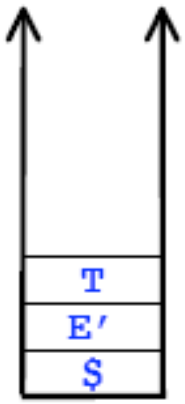
E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

Input:

(id*id)+id

Output:

E → T E'



(	id	*	id	)	+	id	$

Look at Table [E, '(']
Use rule E→TE'
Pop E
Push E'
Push T
Print E→TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

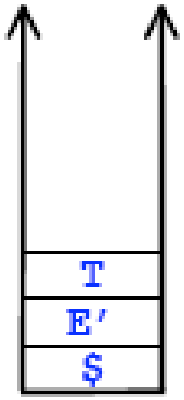
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$



(	id	*	id	)	+	id	$

Table [T, '('] = $T \rightarrow FT'$
Pop T
Push T'
Push F
Print $T \rightarrow FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

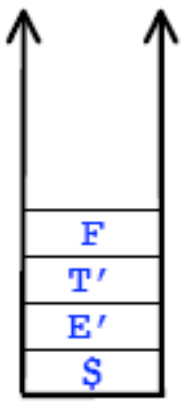
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$



(	id	*	id	)	+	id	$

Table [T, '('] = $T \rightarrow FT'$
Pop T
Push T'
Push F
Print $T \rightarrow FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

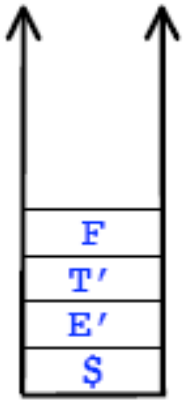
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$



(	id	*	id	)	+	id	$

Table [F, '('] = $F \rightarrow (E)$
Pop F
Push (
Push E
Push)
Print $F \rightarrow (E)$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

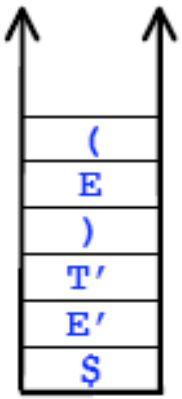
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)



(	id	*	id	)	+	id	$

Table [F, '('] = F→(E)
Pop F
Push)
Push E
Push (
Print F→(E)

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

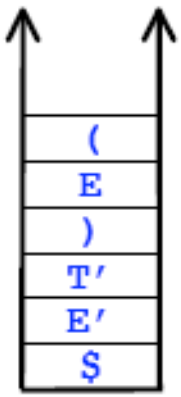
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

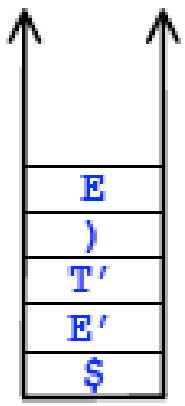
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$



(	id	*	id	)	+	id	$

Table [E, id] = $E \rightarrow TE'$
Pop E
Push E'
Push T
Print $E \rightarrow TE'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

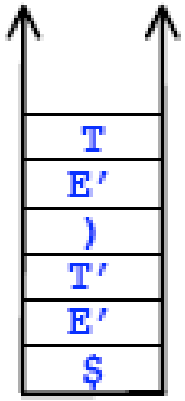
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

`(id*id)+id`

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$



(	id	*	id	)	+	id	$

Table [E, id] = $E \rightarrow TE'$
Pop E
Push E'
Push T
Print $E \rightarrow TE'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

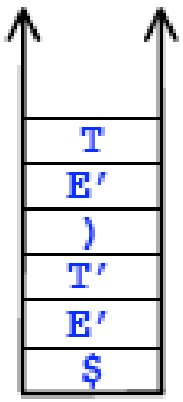
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$



(	id	*	id	)	+	id	$

Table [T, id] = $T \rightarrow FT'$
Pop T
Push T'
Push F
Print $T \rightarrow FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

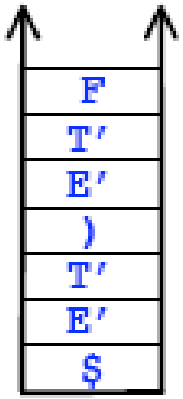
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$



(	id	*	id	)	+	id	$

Table [T, id] = $T \rightarrow FT'$
Pop T
Push T'
Push F
Print $T \rightarrow FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

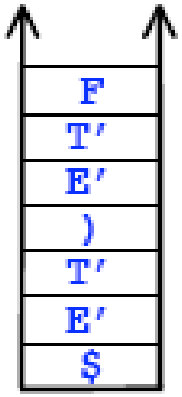
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$



(	id	*	id	)	+	id	$

Table [F, id] = $F \rightarrow \underline{id}$
Pop F
Push id
Print $F \rightarrow \underline{id}$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

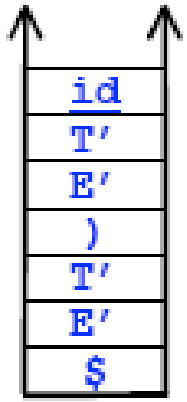
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$



(	id	*	id	)	+	id	$

Table [F, id] = $F \rightarrow \underline{id}$
Pop F
Push id
Print $F \rightarrow \underline{id}$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

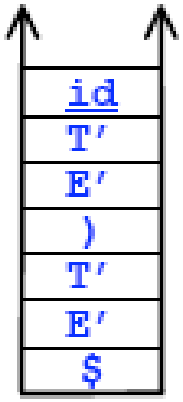
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

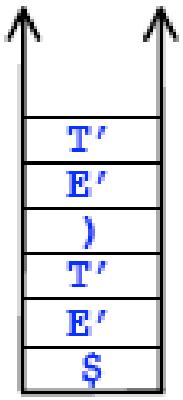
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$



(	id	*	id	)	+	id	$

Table [T', ''] =* $T' \rightarrow *FT'$
Pop T'
Push T'
Push F
Push ''*
Print $T' \rightarrow *FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

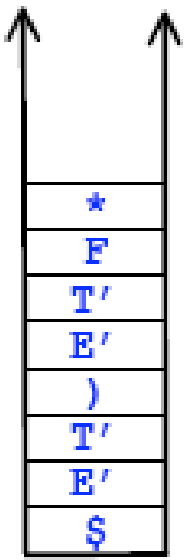
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow *\ F\ T'$



(	id	*	id	)	+	id	$

Table [T', ''] =* $T' \rightarrow *FT'$
Pop T'
Push T'
Push F
Push ''*
Print $T' \rightarrow *FT'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

Example

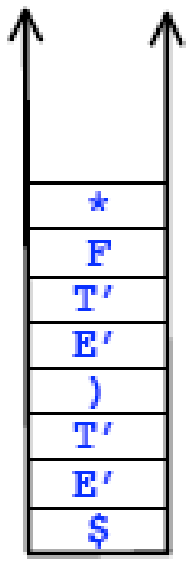
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

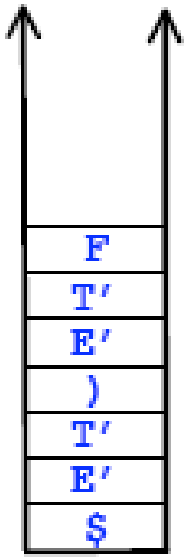
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

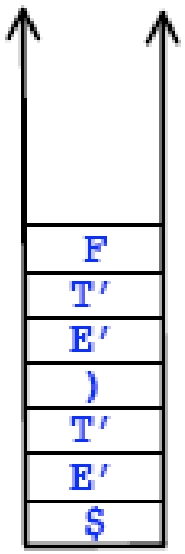
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'



(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

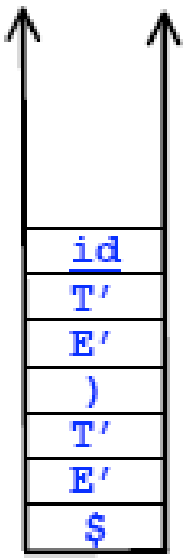
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

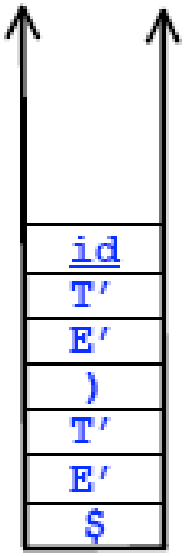
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

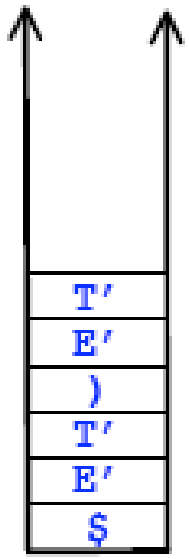
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

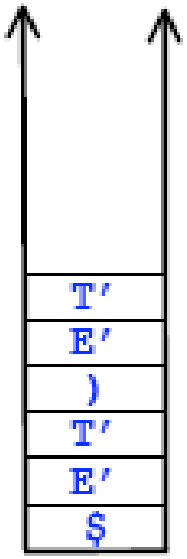
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

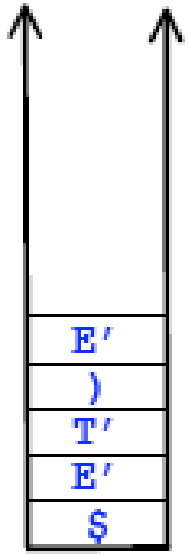
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [T', ')'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

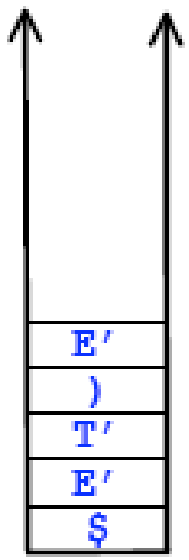
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

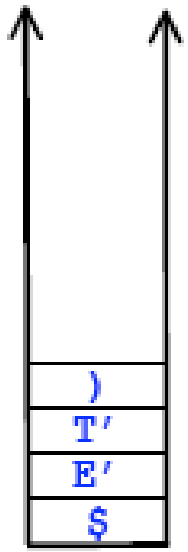
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [E', ')'] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

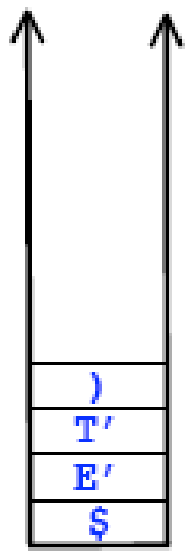
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

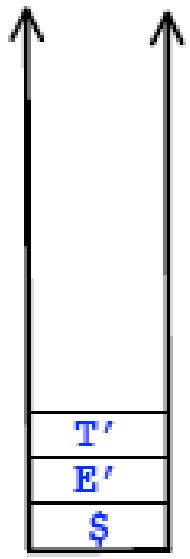
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

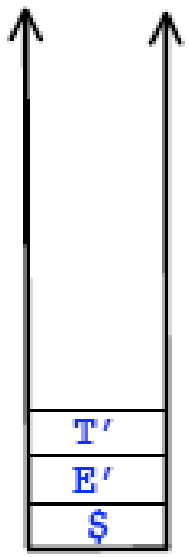
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [T', '+'] = T' → ε
Pop T'
Push <nothing>
Print T' → ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

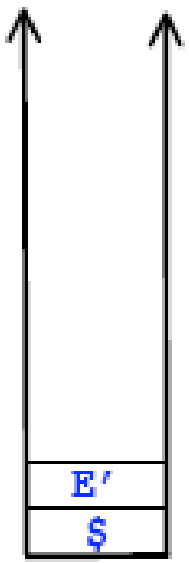
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε



(	id	*	id	)	+	id	$

Table [T', '+'] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

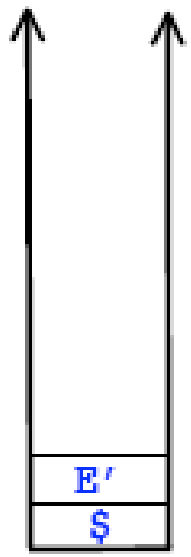
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$
$T' \rightarrow *\ F\ T'$
$F \rightarrow id$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$
$T' \rightarrow \varepsilon$



(	id	*	id	)	+	id	$

Table [E', '+'] = $E' \rightarrow +TE'$
Pop E'
Push E'
Push T
Push '+'
Print $E' \rightarrow +TE'$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

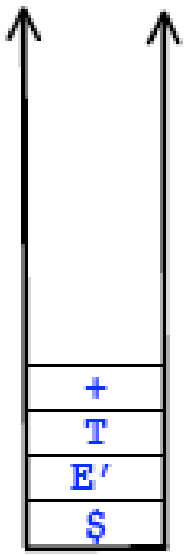
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'



(	id	*	id	)	+	id	$

Table [E', '+'] = E'→+TE'
Pop E'
Push E'
Push T
Push '+'
Print E'→+TE'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

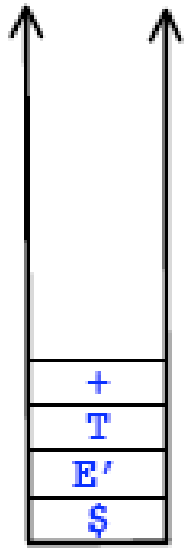
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

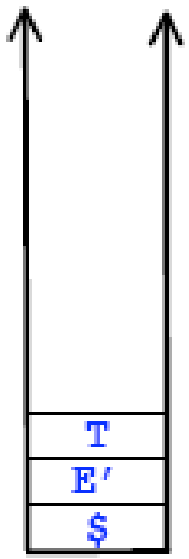
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

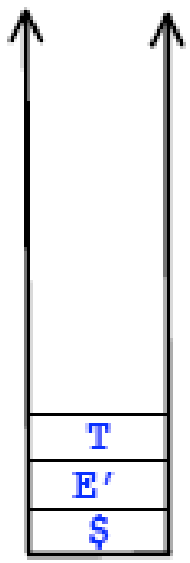
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

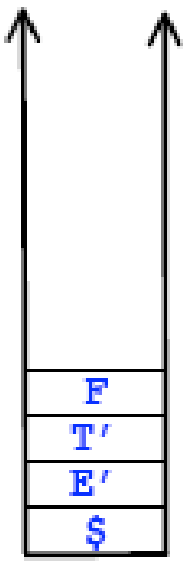
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'
T	→	F T'



(	id	*	id	)	+	id	$

Table [T, id] = T→FT'
Pop T
Push T'
Push F
Print T→FT'

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

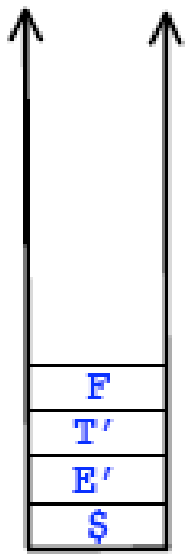
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow *\ F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$
$T' \rightarrow \varepsilon$
$E' \rightarrow +\ T\ E'$
$T \rightarrow F\ T'$



(	id	*	id	)	+	id	$

Table [F, id] = $F \rightarrow \underline{id}$
Pop F
Push id
Print $F \rightarrow \underline{id}$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \underline{id}$			$F \rightarrow (E)$		

Predictive Parsing

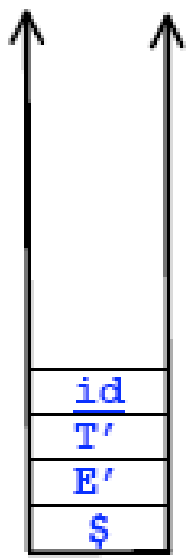
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [F, id] = F→id
Pop F
Push id
Print F→id

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

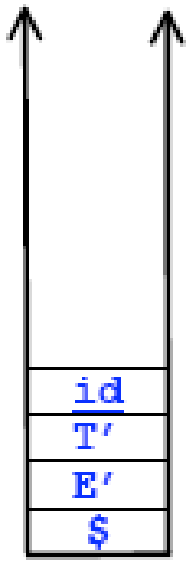
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

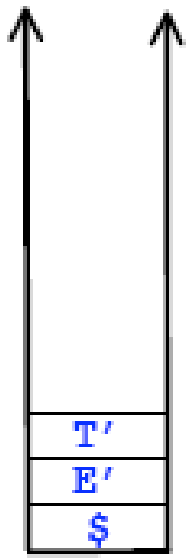
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E	→	T E'
T	→	F T'
F	→	(E)
E	→	T E'
T	→	F T'
F	→	id
T'	→	* F T'
F	→	id
T'	→	ε
E'	→	ε
T'	→	ε
E'	→	+ T E'
T	→	F T'
F	→	id



(	id	*	id	)	+	id	$

Top of Stack matches next input
Pop and Scan

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

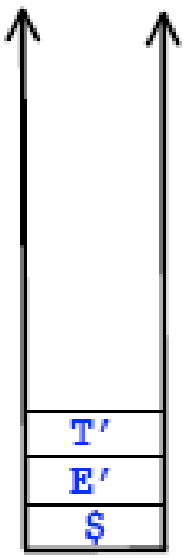
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$
$T' \rightarrow *\ F\ T'$
$F \rightarrow id$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$
$T' \rightarrow \varepsilon$
$E' \rightarrow +\ T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$



(	id	*	id	)	+	id	$

Table [T', $] = $T' \rightarrow \varepsilon$
Pop T'
Push <nothing>
Print $T' \rightarrow \varepsilon$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

Example

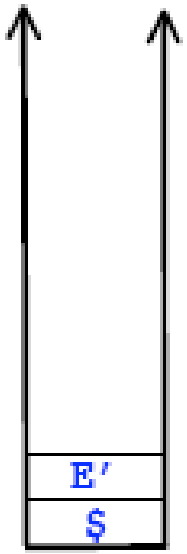
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε



(	id	*	id	)	+	id	$

Table [T', $] = T'→ε
Pop T'
Push <nothing>
Print T'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

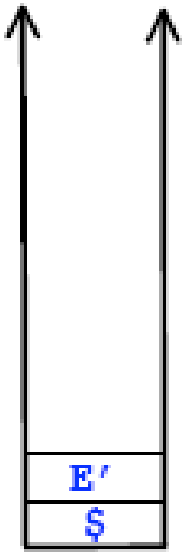
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid id$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$
$T' \rightarrow *\ F\ T'$
$F \rightarrow id$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$
$T' \rightarrow \varepsilon$
$E' \rightarrow +\ T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow id$
$T' \rightarrow \varepsilon$



(	id	*	id	)	+	id	$

Table [E', $] = $E' \rightarrow \varepsilon$
Pop E'
Push <nothing>
Print $E' \rightarrow \varepsilon$

	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

Predictive Parsing

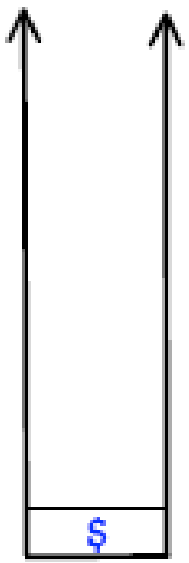
Example

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



E → T E'
E' → + T E' | ε
T → F T'
T' → * F T' | ε
F → (E) | id

(	id	*	id	)	+	id	$

Table [E', $] = E'→ε
Pop E'
Push <nothing>
Print E'→ε

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

Predictive Parsing

Example

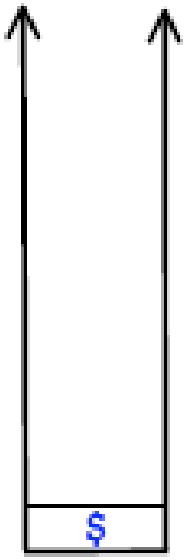
$E \rightarrow T\ E'$
$E' \rightarrow +\ T\ E' \mid \varepsilon$
$T \rightarrow F\ T'$
$T' \rightarrow *\ F\ T' \mid \varepsilon$
$F \rightarrow (\ E\ ) \mid \underline{id}$

Input:

(id*id)+id

Output:

$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow (\ E\ )$
$E \rightarrow T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow *\ F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$
$T' \rightarrow \varepsilon$
$E' \rightarrow +\ T\ E'$
$T \rightarrow F\ T'$
$F \rightarrow \underline{id}$
$T' \rightarrow \varepsilon$
$E' \rightarrow \varepsilon$



(	id	*	id	)	+	id	$

Input symbol == $
Top of stack == $
Loop terminates with success

	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

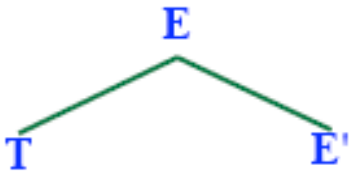
Reconstructing the Parse Tree

Input:

`(id*id)+id`

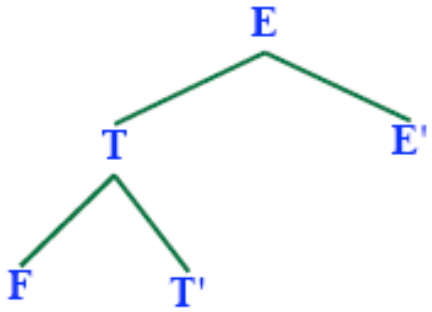
Output:

E → T E'



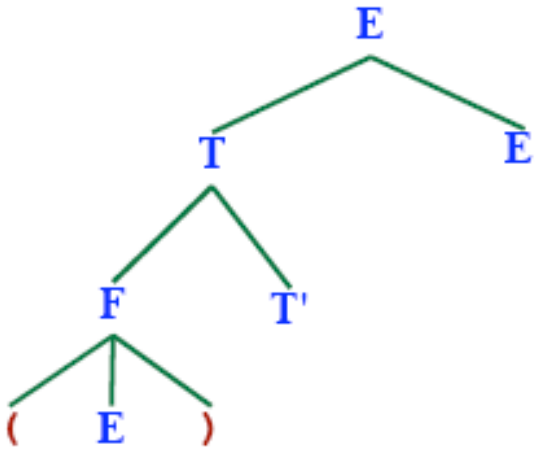
Output:

E → T E'
T → F T'



Output:

E → T E'
T → F T'
F → (E)



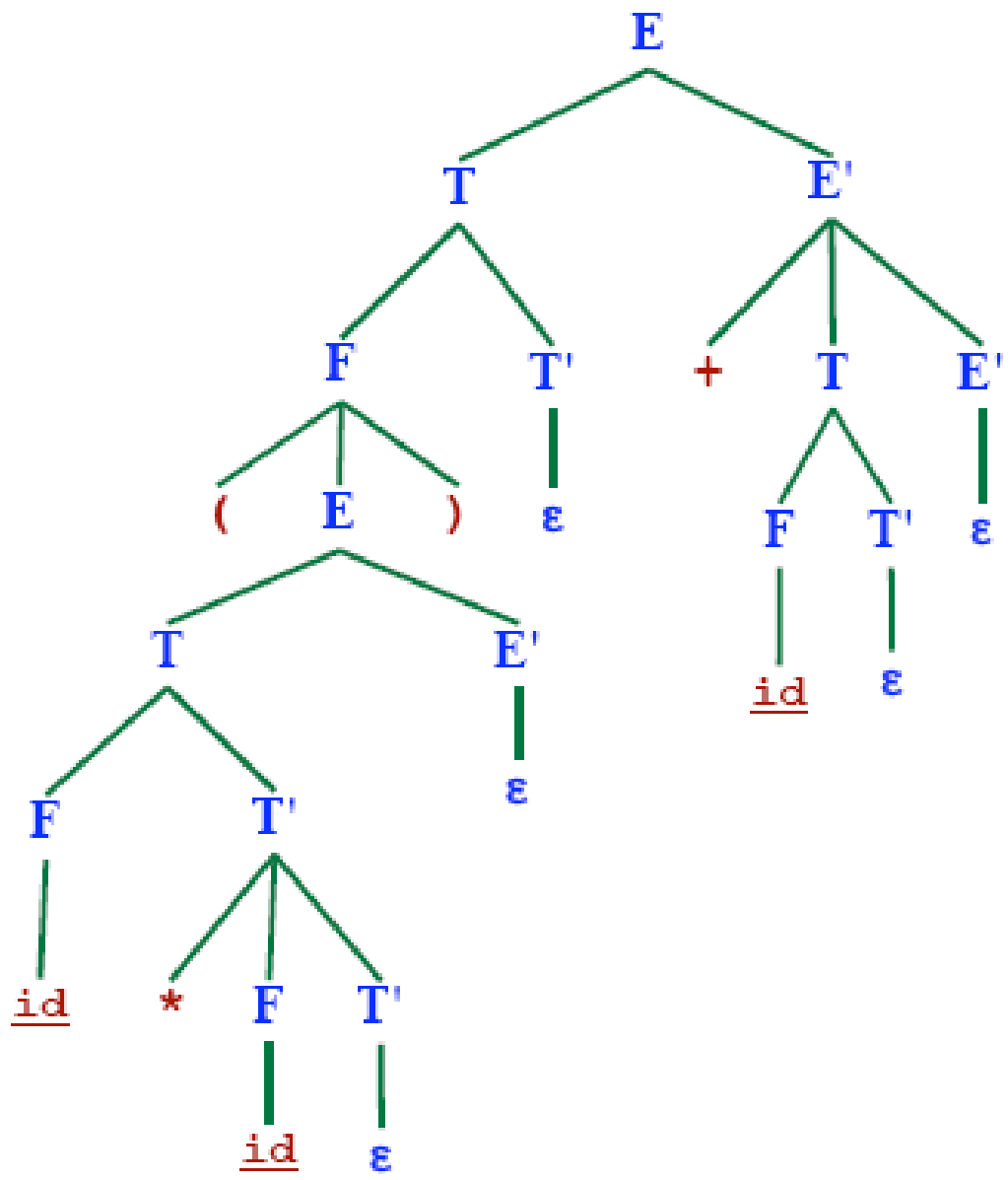
Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε



Reconstructing the Parse Tree

Input:

(id*id)+id

Output:

E → T E'
T → F T'
F → (E)
E → T E'
T → F T'
F → id
T' → * F T'
F → id
T' → ε
E' → ε
T' → ε
E' → + T E'
T → F T'
F → id
T' → ε
E' → ε

Leftmost Derivation:

E
T E'
F T' E'
(E) T' E'
(T E') T' E'
(F T' E') T' E'
(id T' E') T' E'
(id * F T' E') T' E'
(id * id T' E') T' E'
(id * id E') T' E'
(id * id) T' E'
(id * id) E'
(id * id) + T E'
(id * id) + F T' E'
(id * id) + id T' E'
(id * id) + id E'
(id * id) + id

Transition Diagram for Predictive Parsers

- Useful for visualizing predictive parsers.
- To construct Transition Diagram from a grammar
  - Eliminate left recursion
  - Left factor the grammar
  - Then for each nonterminal A
    - Create an initial and final state
    - For each production $A \rightarrow X_1X_2\ldots X_k$, create a path from the initial to the final state, with edges labeled X_1, X_2, …, X_k. If $A \rightarrow \varepsilon$, the path is an edge labeled ε.

Transition Diagram for Predictive Parsers

- Predictive parser begins in the start state for the start symbol
- Suppose at any time it is in state s with an edge
  - labeled by a terminal a to state t

    - If the next input is a the parser advances in input and moves to state t
    - If the edge from s to t is labeled by ε, then the parser moves immediately to state t without advancing the input
  - labeled by a nonterminal A

    - Parser goes to the start state for A
    - If it ever reaches the final state of A it will immediately go back to state t

Transition Diagram for Predictive Parsers

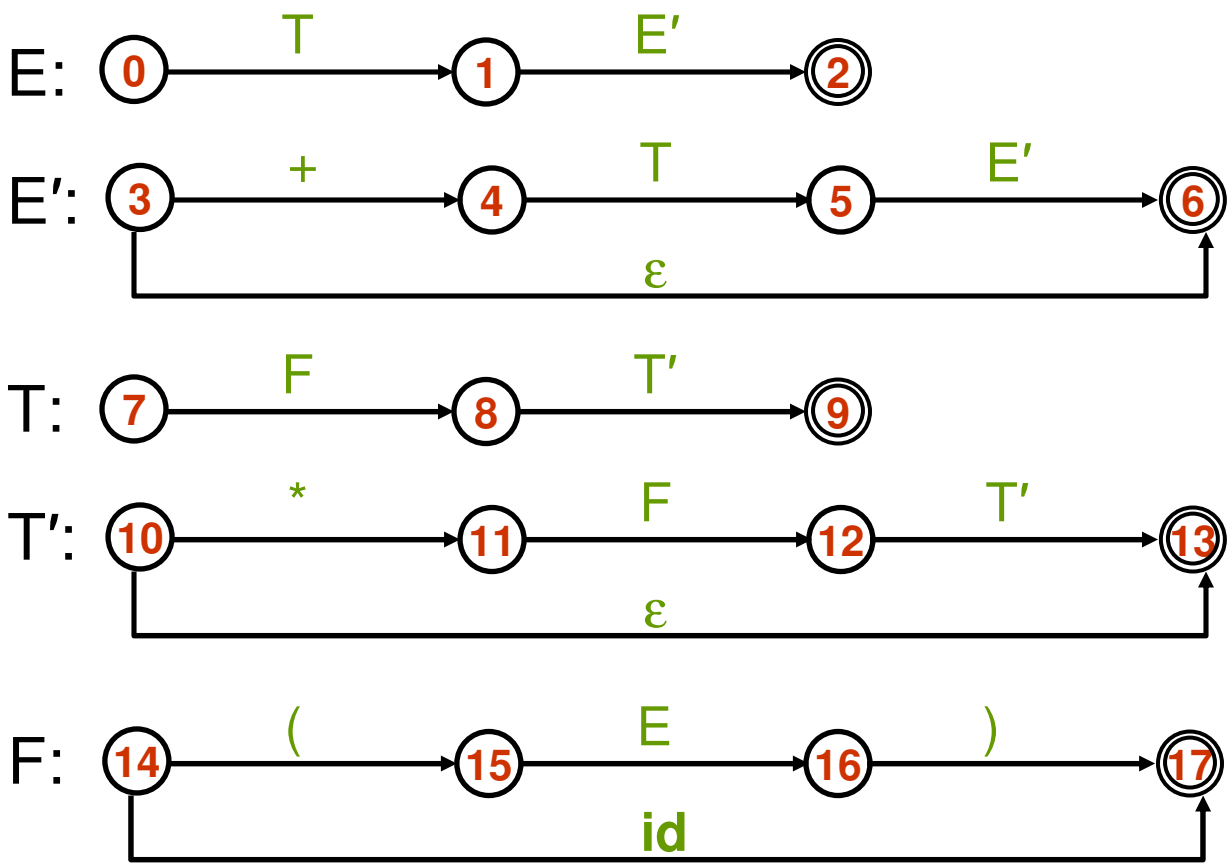
$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

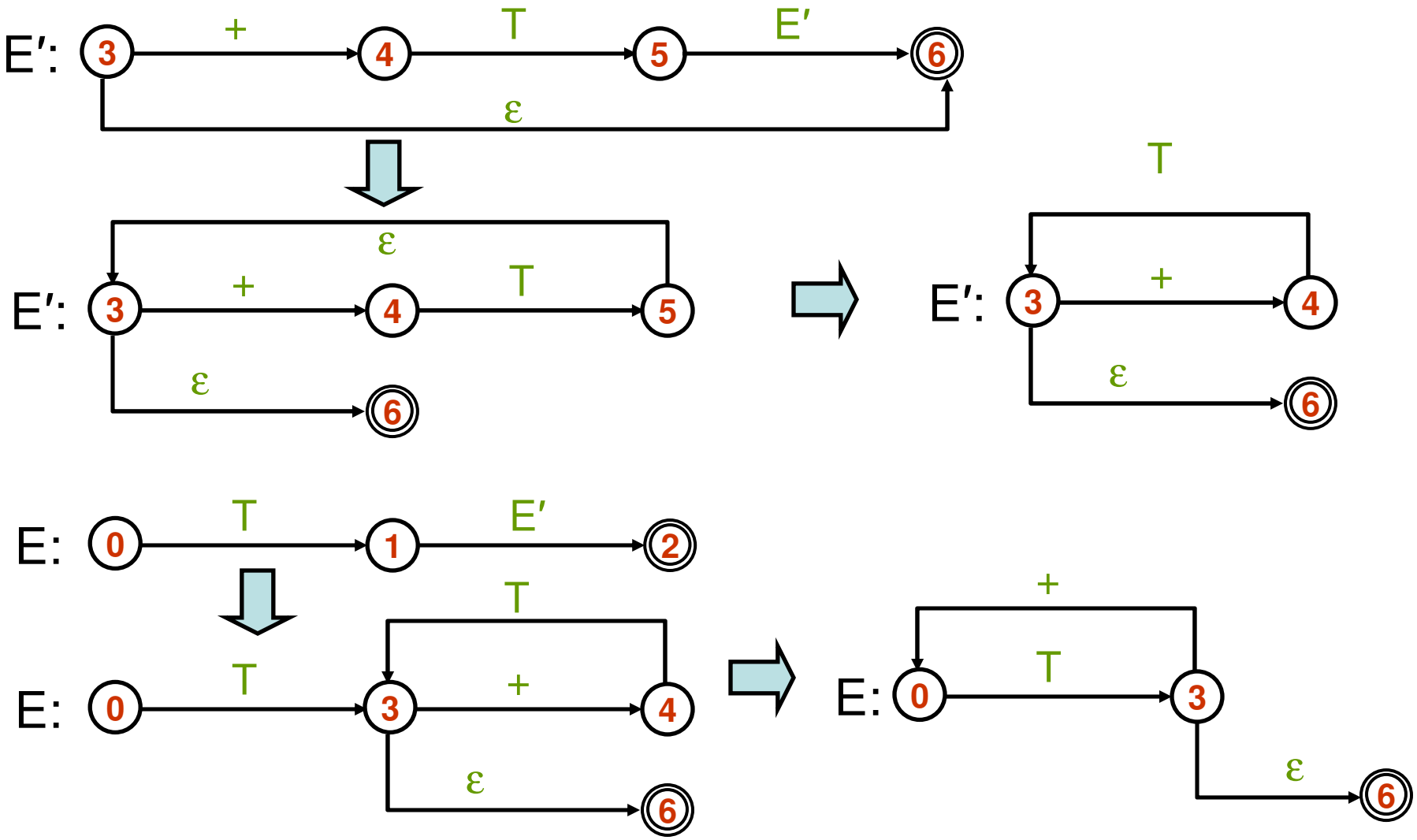
$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid \mathbf{id}$



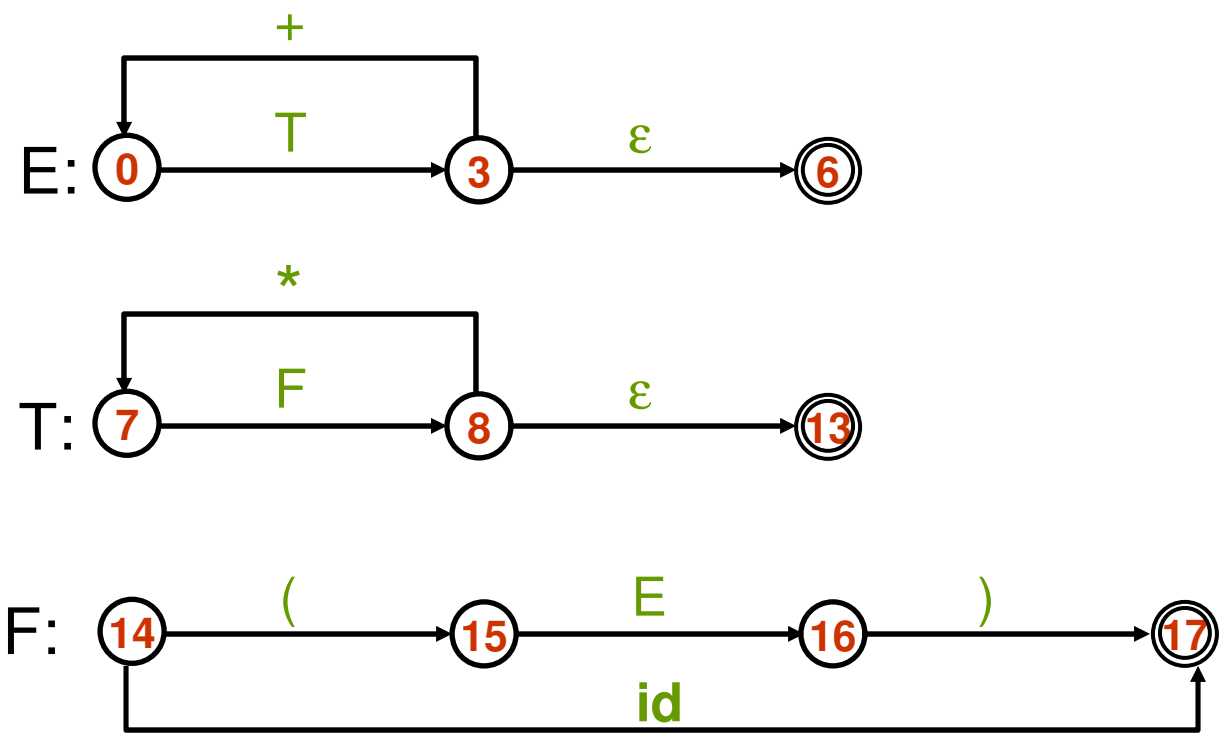
Simplification of Transition Diagrams

- TDs can be simplified by substituting one in another



Simplification of Transition Diagrams

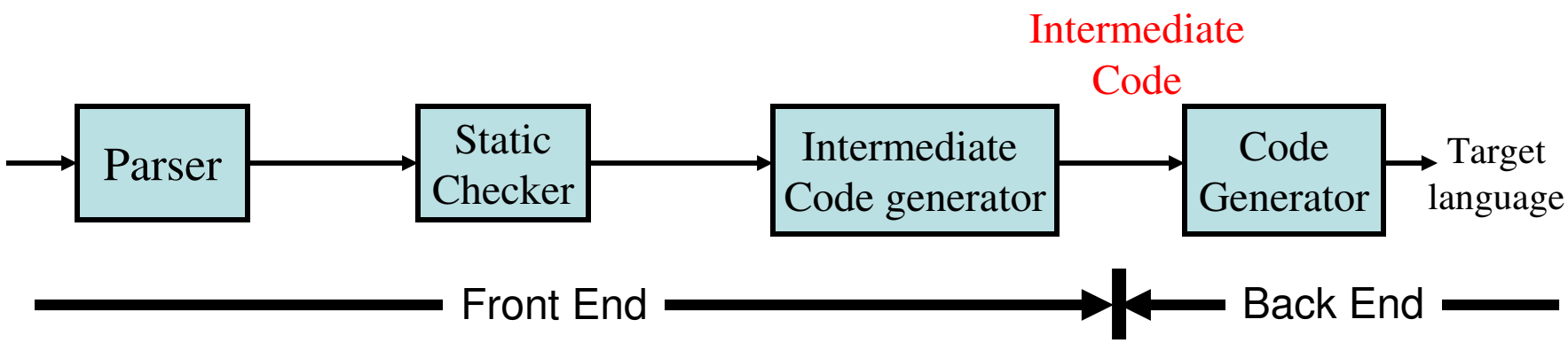
- Complete set of TDs
- A C implementation of this simplified version of the parser runs 20-25% faster than the original version



Intermediate Code Generation

Part I

Compiler Architecture



- m × n compliers can be built by writing m front ends and n back ends – save considerable amount of effort
- We assume parsing, static checking and IC generation is done sequentially
  - These can be combined and done during parsing
- Static checking
  - Operator operand compatibility
  - Proper placement of break/continue keywords etc.

Intermediate Code (IC)

- The given program in a source language is converted to an equivalent program in an intermediate language by the IC generator.
- Ties the front and back ends together
- Language and Machine neutral
- Many forms
- Level depends on how being processed
- More than one intermediate language may be used by a compiler

- Intermediate language can be many different languages, and the designer of the compiler decides this intermediate language.
  - syntax trees can be used as an intermediate language.
  - postfix notation can be used as an intermediate language.
  - three-address code (Quadraples) can be used as an intermediate language
    - we will use quadraples to discuss intermediate code generation
    - quadraples are close to machine instructions, but they are not actual machine instructions.
  - some programming languages have well defined intermediate languages.
    - java – java virtual machine
    - prolog – warren abstract machine
    - In fact, there are byte-code emulators to execute instructions in these intermediate languages.

Intermediate language levels

- **High**

T1 ← a[i,j+2]

- **Medium**

t1 ← j + 2
t2 ← i * 20
t3 ← t1 + t2
t4 ← 4 * t3
t5 ← addr a
t6 ← t5 + t 4
t7 ← *t6

- **Low**

r1 ←[fp-4]
r2 ← r1 + 2
r3 ← [fp-8]
r4 ← r3*20
r5 ← r4 + r2
r6 ← 4 * r5
r7 ← fp – 216
f1 ← [r7+r6]

Intermediate Languages Types

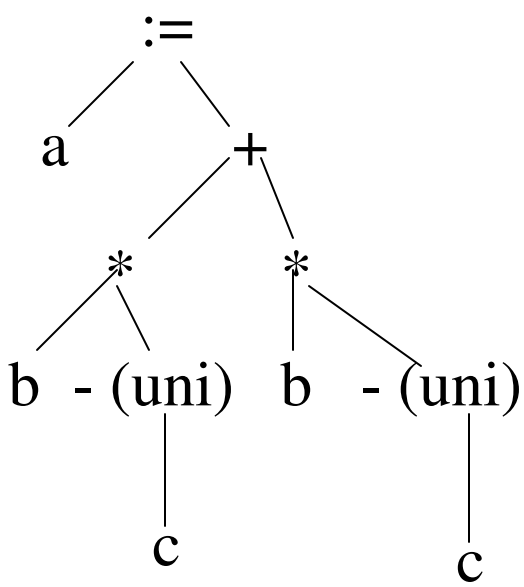
- Graphical IRs:
  - Abstract Syntax trees
  - Directed Acyclic Graphs (DAGs)
  - Control Flow Graphs
- Linear IRs:
  - Stack based (postfix)
  - Three address code (quadruples)

Graphical IRs

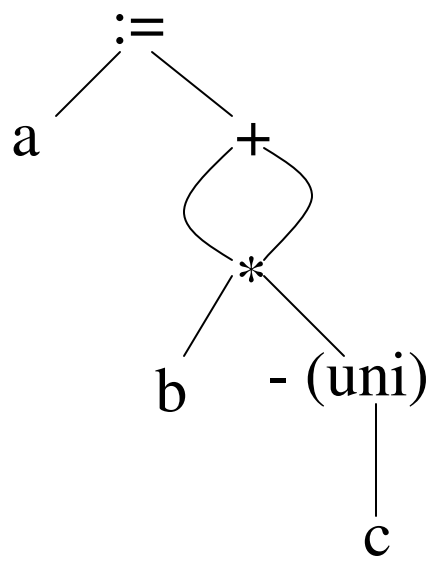
- Abstract Syntax Trees (AST) – retain essential structure of the parse tree, eliminating unneeded nodes.
- Directed Acyclic Graphs (DAG) – compacted AST to avoid duplication – smaller footprint as well
- Control flow graphs (CFG) – explicitly model control flow

ASTs and DAGs:

a := b *-c + b*-c



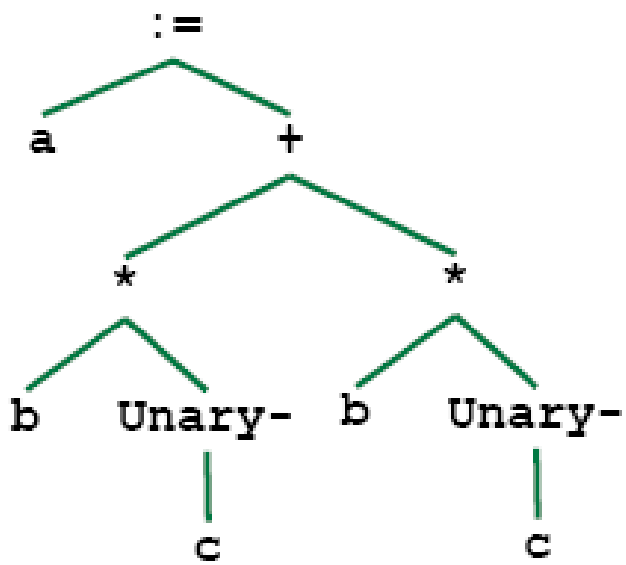
AST



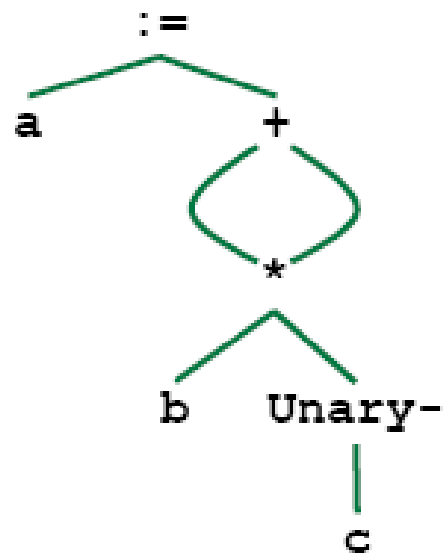
DAG

Implementation of DAG/AST: Value Number Method

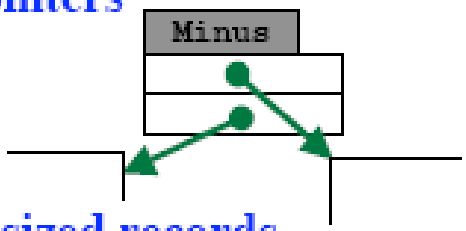
Tree:



DAG:



Structures and Pointers



An array of fixed-sized records

0	id	b	-
1	id	c	-
2	unary-	1	-
3	mult	0	2
4	id	b	-
5	id	c	-
6	unary-	5	-
7	mult	4	6
8	add	3	7
9	id	a	-
10	assign	9	8

Q: How to build DAG's (i.e., trees with shared, common parts)?

A: When you are about to allocate a new node; look to see if you already have one with the same info.

Three-Address Code

- A three-address code is:

  `x := y op z`

  where x, y and z are names, constants or compiler-generated temporaries; **`op`** is any operator.

- But we may also the following notation for three-address code (it looks like a machine code instruction)

  `op y,z,x`

  apply operator `op` to `y` and `z`, and store the result in `x`.

- We use the term “three-address code” because each statement usually contains three addresses (two for operands, one for the result).

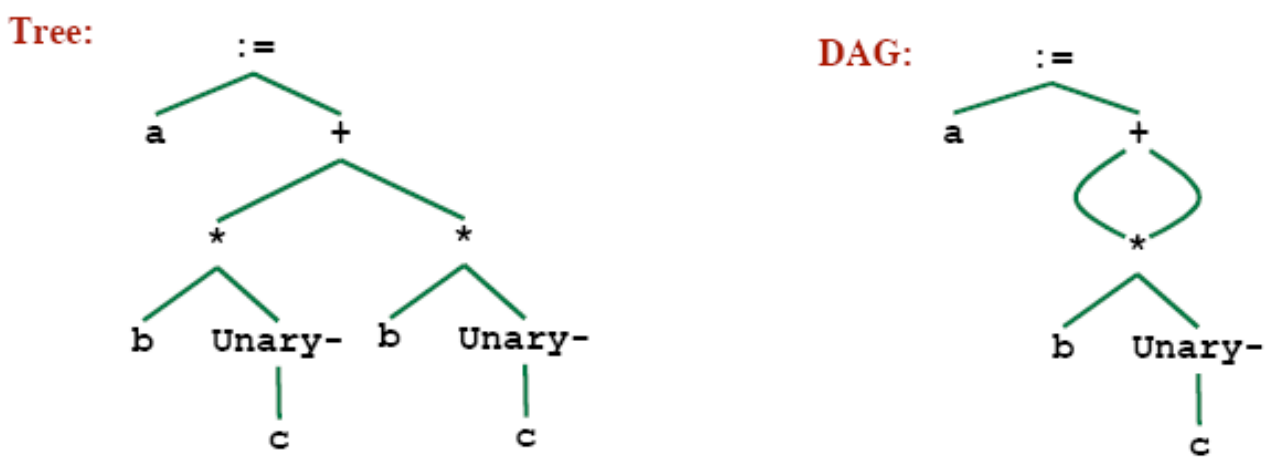
Linearized Representation of DAG/AST

- Source Code
  - a = b * -c + b * -c
- Three address code

Each instruction has (up to) 3 operands.

```
t1 := -c          neg     c      ⇒ t1
t2 := b * t1      mult    b,t1   ⇒ t2
t3 := -c          neg     c      ⇒ t3
t4 := b * t3      mult    b,t3   ⇒ t4
t5 := t2 + t4     add     t2,t4  ⇒ t5
a := t5           move    t5     ⇒ a
```

- Tree Representation



Three-Address Statements

Binary Operator: `op y,z,result` or `result := y op z`

where `op` is a binary arithmetic or logical operator. This binary operator is applied to `y` and `z`, and the result of the operation is stored in `result`.

Ex:

```
add  a,b,c
gt   a,b,c
addr a,b,c
addi a,b,c
```

Unary Operator: `op y,,result` or `result := op y`

where `op` is a unary arithmetic or logical operator. This unary operator is applied to `y`, and the result of the operation is stored in `result`.

Ex:

```
uminus    a,,c
```

Three-Address Code

- Two concepts
  - Address
  - Instruction
- Address
  - Name: source-program names to appear as addresses
  - Constant: Different types of constants
  - Compiler Generated temporary:

Three-Address Instruction

Assignment Type 1: x `:=` `y op z`

op is a binary arithmetic or logical operation

x, *y* and *z* are addresses

Assignment Type 2: x `:=` `op z`

op is a unary arithmetic or logical operation

x and *z* are addresses

Copy Instruction: x `:=` `y`

x and *z* are addresses and x is assigned the value of y

Three-Address Instructions

Unconditional Jump: `goto L`

We will jump to the three-address code with the label `L`, and the execution continues from that statement.

Ex:

```
goto  L1      // jump to L1
jmp   7       // jump to the statement 7
```

Conditional Jump 1: `if x goto L` and `ifFalse x goto L`

We will jump to the three-address code with the label `L` if `x` is TRUE and FALSE, respectively. Otherwise, the following three-address instruction in sequence is executed next.

Conditional Jump 2: `if x` ***`relop`*** `y goto L`

We will jump to the three-address code with the label `L` if the result of `y relop z` is true, and the execution continues from that statement. If the result is false, the execution continues from the statement following this conditional jump statement.

Three-Address Statements (cont.)

Procedure Parameters: `param x`

Procedure Calls: `call p,n`

where `x` is an actual parameter, we invoke the procedure `p` with `n` parameters.

Ex:

```
param x1
param x2
......
param xn
call  p,n,
```

➔ $p(x_1, ..., x_n)$

n is necessary because call can be nested

```
f(x+1,y) ➔   add   x,1,t1
             param t1,,
             param y,,
             call  f,2,
```

Three-Address Statements (cont.)

Indexed Assignments:

`x := y[i]`

sets x to the value in location i memory units beyond location y

`y[i] := x`

sets contents of the location i memory units beyond location y to the value of x

Address and Pointer Assignments:

`x := &y`

sets the r-value of x to l-value of y

`x := *y` where y is a pointer whose r-value is a location

sets the r-value of x equal to the contents of that location

`*x := y`

sets the r-value of the object pointed by x to the r-value of y

Three address code example

do i=i+1; while (a[i] < v)

```
L:      t1=i+1
        i=t1
        t2=i*8
        t3=a[t2]
        if t3 < v goto L
```

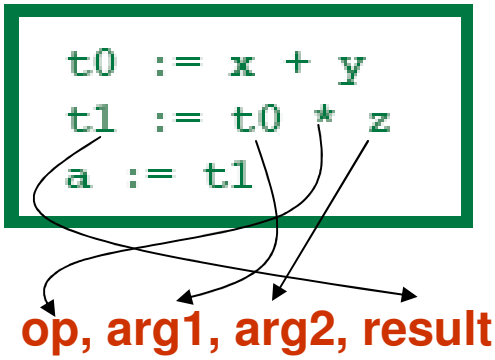
(A) Symbolic Labels

```
100:  t1 = I +1
101:  i = t1
102:  t2=i*8
103:  t3=a[t2]
104:  if t3 < v goto 100
```

(B) Position Numbers

Representing 3-Address Statements

- Quadruples ("Quads")
- Triples
- Indirect Triples



- `x = minus y`
  - Does not use arg2
- `x = y`
  - Op is =
- `param a1`
  - Uses neither arg2 nor result
- Conditional/Unconditional jumps
  - Put the target label in result

Quadruples

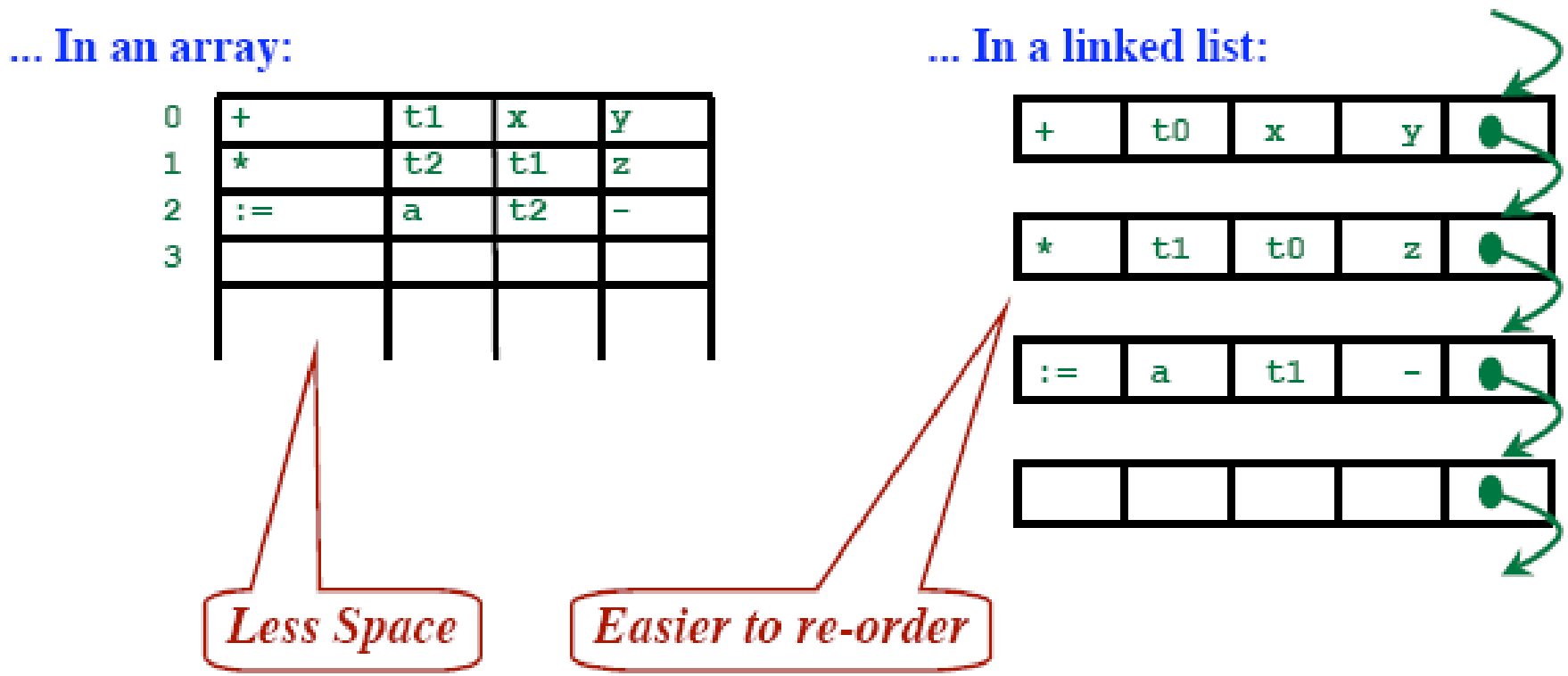
- a = b * -c + b * -c

```
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Instr	Operation	Arg 1	Arg 2	Result
(0)	uminus	c		t_1
(1)	mult	b	t_1	t_2
(2)	add	a	t_2	t_3
(3)	move	t_3		a

Quadruples

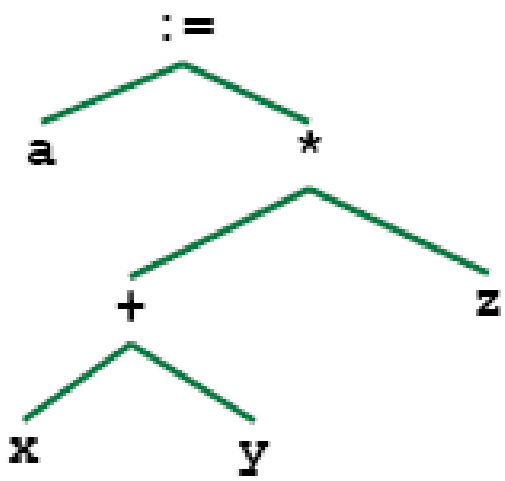
- Store each fields directly



Triples

Don't store the result directly.
Implicitly associate a temporary result with each triple.

```
t0 := x + y
t1 := t0 * z
a := t1
```



Avoids creating the temporaries.
Saves storage.
Difficult to re-order instructions.

0	+	x	y
1	*	0	z
2	:=	a	1
3			

The following instruction is difficult

```
x[i] := y
```

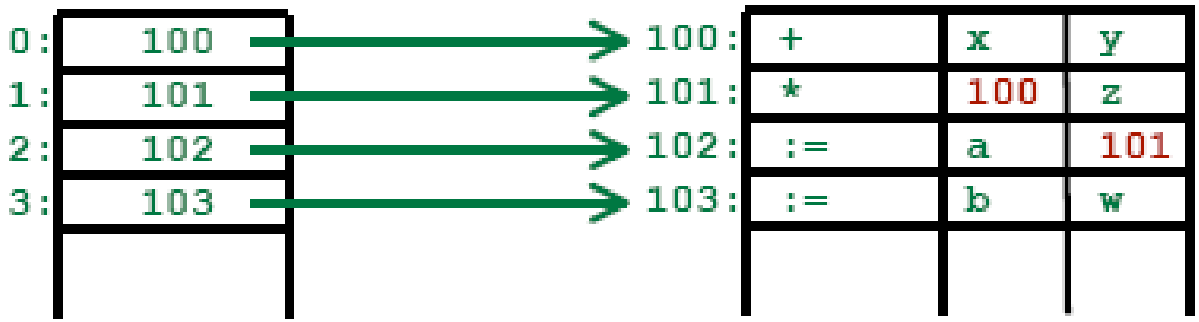
It takes 2 triples.

	op	arg1	arg2
0	[ ]=	x	i
1	:=	0	y

Indirect Triples

Get around the re-ordering problem

... by introducing another data structure.



Quadruples

- Less indirection, simpler
- Easier to manipulate, reorder

Triples

Indirect Triples

- About same amount of space as quadruples
- May save space when lots of shared sub-expressions
- More complex

Code Generation

Part I

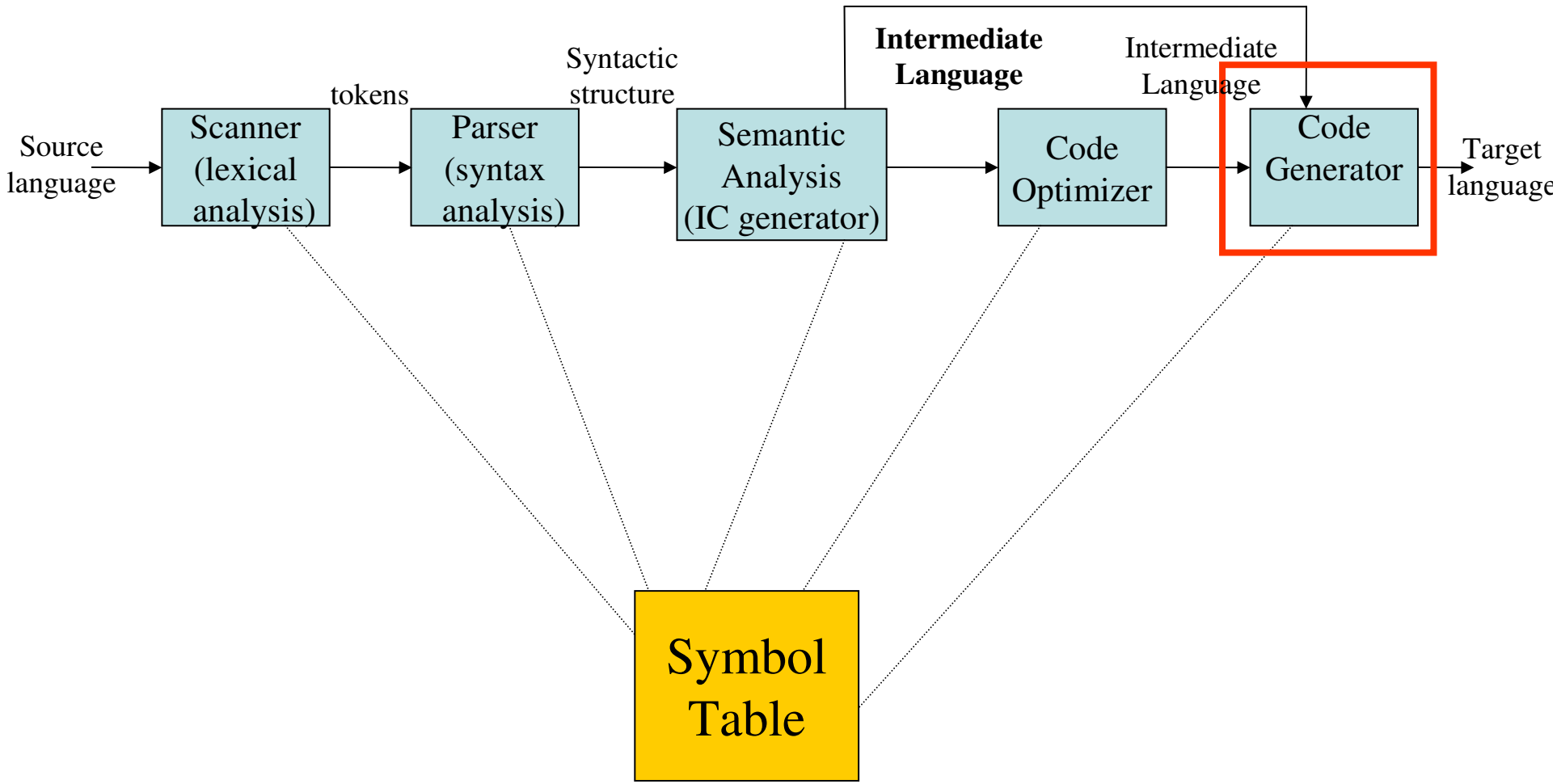
Code Generation

The code generation problem is the task of mapping intermediate code to machine code.

Requirements:

- Correctness
  - Must preserve semantic meaning of source program
- Efficiency
  - Make effective use of available resources
  - Code Generator itself must run efficiently

Compiler Architecture



Code Generation

- Want ***optimal*** code sequences?

  NP-Complete

  Generate all correct code sequences

  ... and see which is best

- We should be content with heuristic techniques that generate good code

- **Optimal?**

  The target program...

  ... executes **faster**

  ... takes less memory

Tasks of a Code Generator:

- Input language:
  - intermediate code (optimized or not)
    - Syntax and semantic errors have been removed
    - Type checking done and type conversion operators inserted
  - Information in the symbol table
- Target architecture: must be **well** understood
  - Significantly influences the difficulty of code generation
  - RISC, CISC
- Interplay between
  - Instruction Selection
  - Register Allocation
  - Instruction Scheduling

Instruction Selection

- There may be a large number of ‘candidate’ machine instructions for a given IR instruction
  - Level of IR
    - High: Each IR translates into many machine instructions
    - Low: Reflects many low-level details of machine
  - Nature of the instruction set
    - Uniformity and completeness
  - Each has own cost and constraints
    - Accurate cost information is difficult to obtain
  - Cost may be influenced by surrounding context

Instruction selection: Machine Idioms

IR Code: `x := x + 5`

Target Code:

```
mov   x,r0
add   5,r0
mov   r0,x
```

IR Code: `x := x + 1`

Target Code:

```
mov   x,r0
add   1,r0
mov   r0,x
```

Target Code:

```
mov   x,r0
inc   r0
mov   r0,x
```

Target Code:

```
inc   x
```

Register Allocation

- How to best use the bounded number of registers.
- Use or registers
  - Register allocation
    - We select a set of variables that will reside in registers at each point in the program
  - Register assignment
    - We pick the specific register that a variable will reside in.
- Complications:
  - special purpose registers
  - operators requiring multiple registers.
- Optimal assignment is NP-complete

Register Allocation

Multiply Instruction

```
mul    y,r4
```

Must specify an even numbered register

$r5 \times y \rightarrow [r4,r5]$

Multiply Instruction

```
div    y,r4
```

Must specify an even numbered register

$[r4,r5] \div y \Rightarrow [r4,r5]$

SRDA: Shift Right Double Arithmetic

```
srda   32,r6
```

Register Allocation

IR Code:

```
t := a + b
t := t * c
t := t / d
```

Target Code:

```
mov   a,r1
add   b,r1
mul   c,r0
div   d,r0
mov   r1,t
```

IR Code:

```
t := a + b
t := t + c
t := t / d
```

Target Code:

```
mov   a,r0
add   b,r0
add   c,r0
srda  32,r0
div   d,r0
mov   r1,t
```

Conclusion:

Where you put the result of t:=a+b (either r0 or r1) depends on how it will be used later!!!

[A "chicken-and-egg" problem]

Instruction Scheduling

- Choosing the order of instructions to best utilize resources
- Picking the optimal order is NP-complete problem

- Simplest Approach
  - Don’t mess with re-ordering.
  - Target code will perform all operations in the same order as the IR code
- Trickier Approach
  - Consider re-ordering operations
  - May produce better code

    ... Get operands into registers just before they are needed

    ... May use registers more efficiently

Moving Results Back to Memory

- When to move results from registers back into memory?
  - After an operation, the result will be in a register.
- **Immediately**
  - Move data back to memory just after it is computed.
  - May make more registers available for use elsewhere.
- ***Wait as long as possible* before moving it back**
  - Only move data back to memory “at the end”
    - or “when absolutely necessary”
  - May be able to avoid re-loading it later!

Code Generation Algorithm #1

Simple code generation algorithm:

Define a target code sequence to each intermediate code statement type.

Example Target Machine

2-Address Architecture

source destination

```
mov    x,r0
add    y,r0
mov    r0,z
```

r0 := r0 + y

store back into memory

Code Generation Algorithm #1

Statement-by-statement generation
Code for each IR instruction is
generated independently of all other IR instructions.

IR Code:

```
a := b + c
d := a + e
```

ALSO: Registers are not used effectively.

Target Code:

```
...
mov   b,r0
add   c,r0        a := b + c
mov   r0,a
mov   a,r0
add   e,r0        d := a + e
mov   r0,d
...
```

This instruction is totally unnecessary!!!

Example Target Machine

A 2-address Architecture

```
op source,destination
```

2 operands, at most

Op-Codes:

```
mov
add
sub
mul
...
```

Address Modes:

Absolute Memory Address

```
mov    x,y
sub    x,y
```

x → y
y–x → y

Register

```
mov    r0,r1
sub    r2,r3
```

r3–r2 → r3

Literal

```
mov    39,r1
sub    47,r2
```

Data is included in the instruction directly

Indirect Register

```
mov    r0,[r1]
```

Register contains an address. Moves data in to word pointed to by r1

Indirect plus Index

```
mov    r0,[r1+48]
```

Double Indirect

```
mov    r0,[[r1+48]]
```

Use r1+48 as an address. Go to memory and fetch a second address, "p". "p" points to the word.

Evaluating A Potential Code Sequence

- Each instruction has a "*cost*"

  Cost = Execution Time
- Execution Time is difficult to predict.

  Pipelining, Branches, Delay Slots, etc.
- **Goal:** Approximate the real cost

  A "***Cost Model***"

Simplest Cost Model:

Code Length ≈ Execution Time

Just count the instructions!

A Better Cost Model

Look at each instruction.
Compute a cost (in "units").
Count the number of memory accesses.

Cost = 1 + Cost-of-operand-1 + Cost-of-operand-2 + Cost-of-result

	example	cost
Absolute Memory Address	`x`	1
Register	`r0`	0
Literal	`39`	0
Indirect Register	`[r1]`	1
Indirect plus Index	`[r1+48]`	1
Double Indirect	`[[r1+48]]`	2

Example: `sub 97,r5` r5 – 97 → r5
Cost = 1 + 0 + 0 + 0 = 1

Example: `sub 97,[r5]` [r5] – 97 → [r5]
Cost = 1 + 1 + 0 + 1 = 3

Example: `sub [r1],[[r5+48]]` [[r5+48]] – [r1] → [[r5+48]]
Cost = 1 + 2 + 1 + 2 = 6

Cost Generation Example

IR Code: `x := y + z`

Translation #1:

```
mov   y,x   3
add   z,x   4
```

Cost = 7

Translation #2:

```
mov   y,r1  2
add   z,r1  2
mov   r1,x  2
```

Cost = 6

Lesson #1:
Use Registers

Translation #3:

Assume "y" is in r1 and "z" is in r2
Assume "y" will not be needed again

```
add   r2,r1  1
mov   r1,x   2
```

Cost = 3

Lesson #2:
Keep variables in registers

Lesson #3:
Avoid or delay storing into memory.

Translation #4:

Assume "y" is in r1 and "z" is in r2
Assume "y" will not be needed again.
Assume we can keep "x" in a register.

```
add   r2,r1  1
```

Cost = 1

Lesson #4: *(not illustrated)*
Use different addressing modes effectively.

Basic Blocks

Break IR code into blocks such that...

The block contains NO transfer-of-control instructions

... except as the last instruction

- **A sequence of consecutive statements.**
- **Control enters only at the beginning.**
- **Control leaves only at the end.**

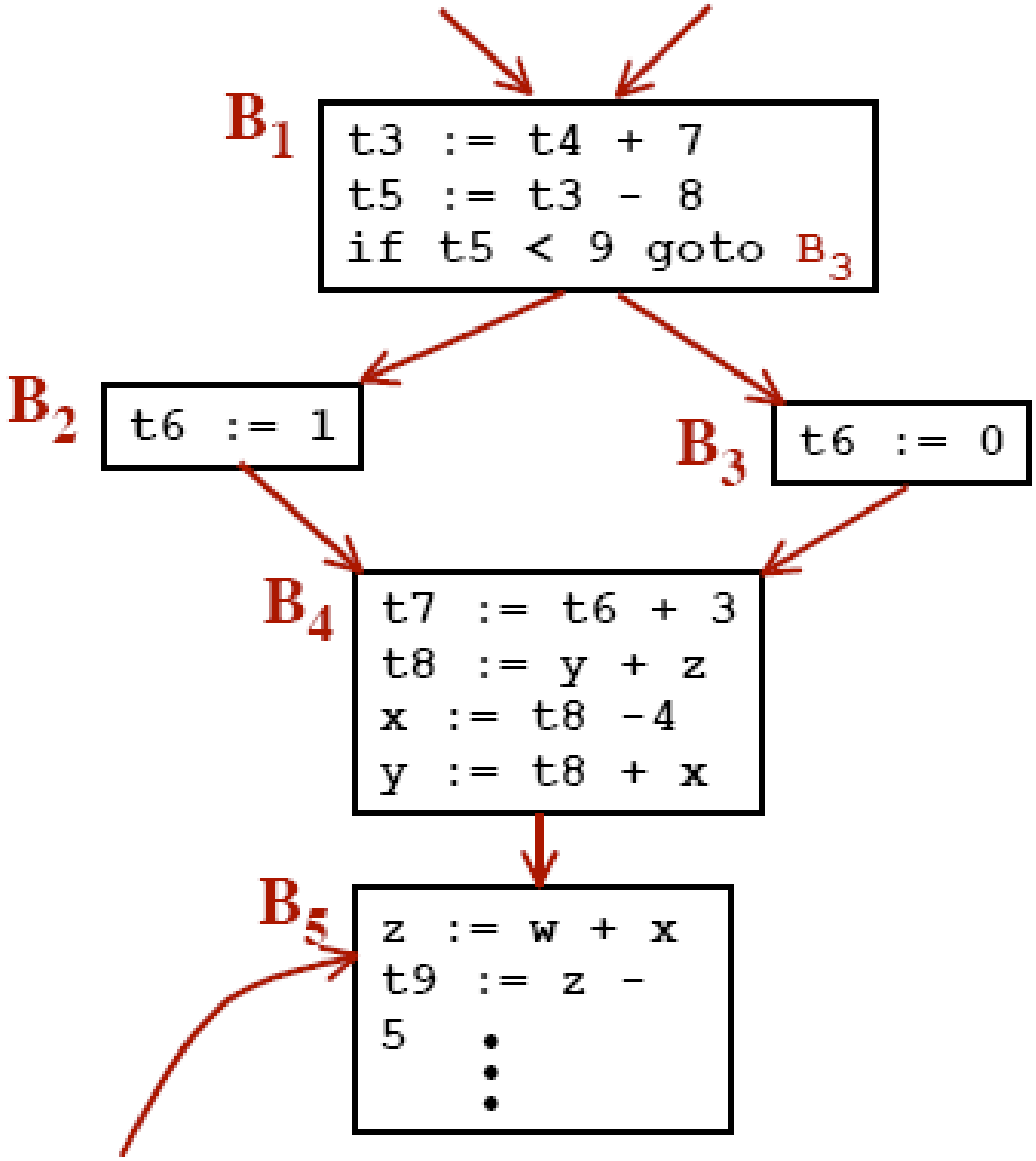
Basic Blocks

```
                 ⋮
Label_43:    t3 := t4 + 7
             t5 := t3 - 8
             if t5 < 9 goto Label_44
             t6 := 1
             goto Label_45
Label_44:    t6 := 0
Label_45:    t7 := t6 + 3
             t8 := y + z
             x := t8 -4
             y := t8 + x
Label_46:    z := w + x
             t9 := z - 5
                 ⋮
```

Basic Blocks

```
                 ⋮
Label_43:        t3 := t4 + 7
                 t5 := t3 - 8                    B1
                 if t5 < 9 goto Label_44
                 t6 := 1                         B2
                 goto Label_45
Label_44:        t6 := 0                         B3
Label_45:        t7 := t6 + 3
                 t8 := y + z                     B4
                 x := t8 -4
                 y := t8 + x
Label_46:        z := w + x                      B5
                 t9 := z - 5
                 ⋮
```

Control Flow Graph



Algorithm to Partition Instructions into Basic Blocks

Concept: "Leader"

The first instruction in a basic block

Idea:

Identify "leaders"

- The first instruction of each routine is a leader.
- Any statement that is the target of a branch / goto is a leader.
- Any statement that immediately follows
  - a branch / goto
  - a call instruction

  ... is a leader

A Basic Block consists of

A leader and all statements that follow it

... up to, but not including, the next leader

Identify Leaders

```
                 ⋮
Label_43:     t3 := t4 + 7
              t5 := t3 - 8
              if t5 < 9 goto Label_44
              t6 := 1
              goto Label_45
Label_44:     t6 := 0
Label_45:     t7 := t6 + 3
              t8 := y + z
              x := t8 -4
              y := t8 + x
Label_46:     z := w + x
              t9 := z - 5
                 ⋮
```

Identify Leaders

```
                 ⋮
Label_43:    t3 := t4 + 7
             t5 := t3 - 8
             if t5 < 9 goto Label_44
             t6 := 1
             goto Label_45
Label_44:    t6 := 0
Label_45:    t7 := t6 + 3
             t8 := y + z
             x := t8 -4
             y := t8 + x
Label_46:    z := w + x
             t9 := z - 5
                 ⋮
```

Targets of GOTOs

Identify Leaders

```
                 ⋮
Label_43:    t3 := t4 + 7
             t5 := t3 - 8
             if t5 < 9 goto Label_44
             t6 := 1
             goto Label_45
Label_44:    t6 := 0
Label_45:    t7 := t6 + 3
             t8 := y + z
             x := t8 -4
             y := t8 + x
Label_46:    z := w + x
             t9 := z - 5
                 ⋮
```

Follows a GOTO

Identify Leaders

```
                 ⋮
Label_43:        t3 := t4 + 7
                 t5 := t3 - 8
                 if t5 < 9 goto Label_44
                 t6 := 1
                 goto Label_45
Label_44:        t6 := 0
Label_45:        t7 := t6 + 3
                 t8 := y + z
                 x := t8 -4
                 y := t8 + x
Label_46:        z := w + x
                 t9 := z - 5
                 ⋮
```

Look at Each Basic Block in Isolation

Use (B)

The set of variables used (i.e., read) by the Basic Block

(... before being written / updated)

The "inputs" to the BB

Def (B)

The set of variables in the Basic Block that are written / assigned to.

The "outputs" of the BB

B_7

```
x := y + v
z := x * y
v := z + 5
if w < v goto B9
```

Use (B_7) = y, v, w

Def (B_7) = x, z, v

View the basic block as a function

< x, z, v > := f (y, v, w)

Okay to transform the block!

(as long as it computes the same function)

Common Sub-Expression Elimination

Transform:

```
x := b + c
y := a - d
d := b + c
```

We compute "b+c" twice!

Into:

```
x := b + c
y := a - d
d := x
```

Common Sub-Expression Elimination

Transform:

```
x := b + c
y := a - d
d := b + c
z := a - d
```

What about "a-d"...
Do we need recompute?

Into:

```
x := b + c
y := a - d
d := x
z := a - d
```

Yes!
"d" has been changed since "a-d" computed!
Now, "a-d" may compute a different value!

Reordering Instructions in a Basic Block

Sometimes we can change the order of instructions...

```
x := b + c          x := b + c          a := x + y
d := e + f          a := x + y          x := b + c
a := x + y          d := e + f          d := e + f
```

Not Okay!

But some changes would change the program!

When can we exchange these two instructions?

$$x := \ldots v_1 \ldots v_2 \ldots$$
$$y := \ldots v_3 \ldots v_4 \ldots$$

If and only if...

$v_1 \neq y$

$v_2 \neq y$

$v_3 \neq x$

$v_4 \neq x$

Any variables (including possibly "x" and "y")

Live Variables

"Is some variable x live at some point P in the program?"

Could the value of "x" at point P ever be needed later in the execution?

"Point in a program"

A point in a program occurs between two statements.

```
...
a := b + c
d := e * f
c := b - 5
...
```

Point P

Is it possible that the program will ever read from x along a path from P?

[... before "x" is written / stored into]

Dead Variables

A Variable is "*Dead at point P*"
= Not Live

Value will <u>definitely</u> never be used.
No need to compute it!
If value is in register, no need to store it!

Liveness Example

```
a := b + c
d := e * f
c := b - 5
```

At this point...

Is b live? YES

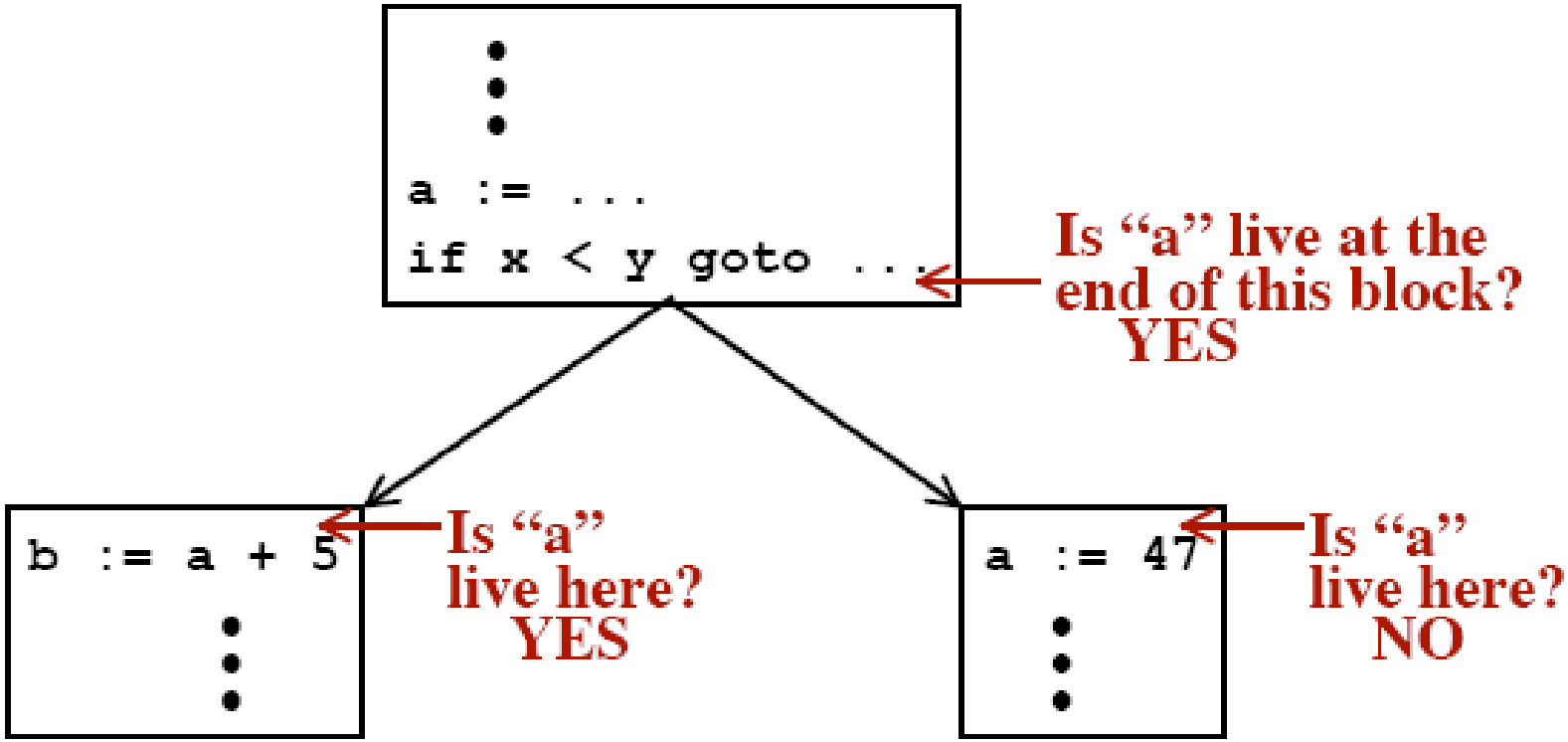
Is c live? NO

Is a live? Don't Know

Is g live? Possibly!

Liveness Example

Must look at the whole "control flow graph" to determine liveness.



Live Variable Analysis

Input:

The Control Flow Graph

$$\left.\begin{array}{l}\text{Use}(B_i)\\ \text{Def}(B_i)\end{array}\right\}\text{for all } B_i$$

Output:

$\text{Live}(B_i)$ = a list of all variables live at the end of B_i

Live Variable Analysis missing?

Assume <u>all</u> variables are live at the end of each basic block.

Temporaries

Assumption:

Each temporary is used in only one basic block

(True of temps for expression evaluation)

```
    . . .
t5  := xxxx + xxxx
    . . .
xxxx := t5 + xxxx
    . . .
```

More precisely:
No temp will ever be in Use(B_i) for any BB

Conclusion:

Temps are never live at the end of a basic block.

If Live-Variable-Analysis is missing...

this assumption can at least identify many dead variables.

Dead Code

"Dead Code" (first meaning)

Any code that cannot be reached.
(Will never be executed.)

```
        x := y + z
        goto Label_45
        a := b + c
        d := e * f
  Label_45:
        z := x - a
```

Dead Code (unreachable)

"Dead Code" (second meaning)

A statement which computes a dead variable.

Example:

```
        b := x * y
        a := b + c
        ...
```

If "a" is not live here...

Then eliminate this statement!!!

Temporaries

If you can identify a variable which is not in Use(B_i) for any basic block

(e.g., a temporary used only in this basic block)

Then you may...

- **Rename the variable**
- **Keep the variable in a register instead of in memory**
- **Eliminate it entirely (during some optimization)**

Must be careful that the variable is not used in other routines

(i.e., accessed as a non-local from another routine)

Algebraic Transformation

Watch for special cases.
Replace with equivalent instructions
... that execute with a lower cost.

Examples

```
x := y + 0   ⇒   x := y
x := y * 1   ⇒   x := y
x := y ** 2  ⇒   x := y * y
x := y + 1   ⇒   x := incr(y)
x := y - 1   ⇒   x := decr(y)
...etc...
```

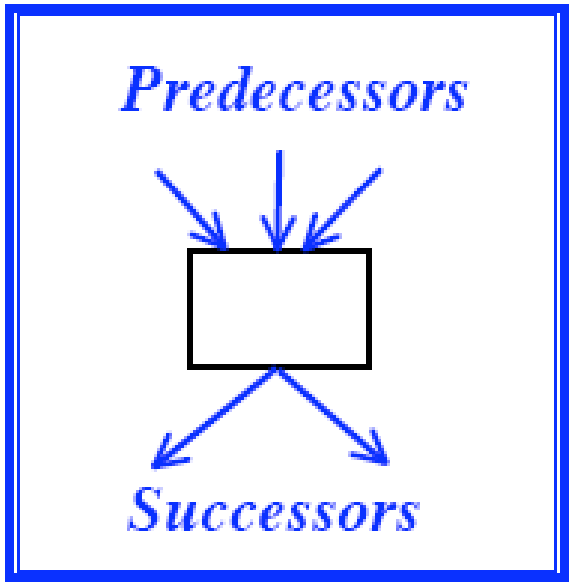
May do some transformations during "Peephole Optimization."

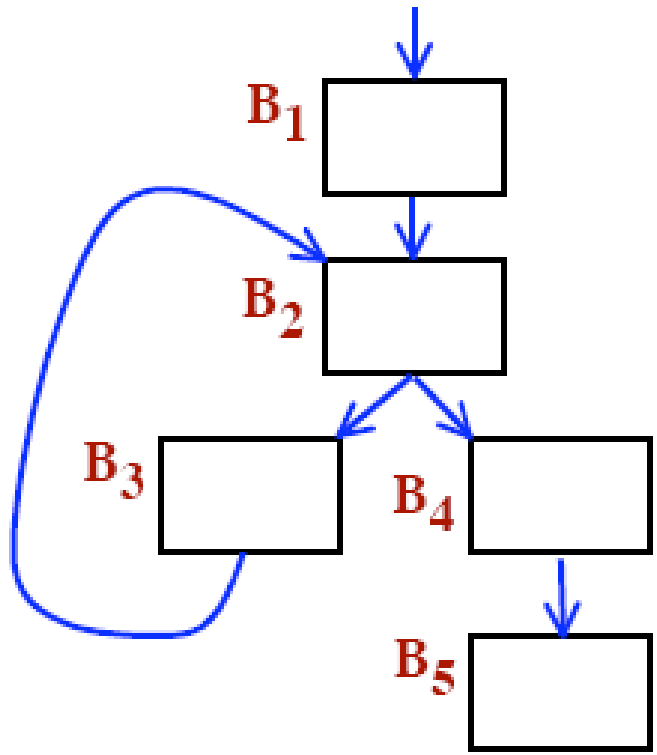
Other transformations may be Target Architecture Dependent
(use your "cost model" to determine when to transform)

Control Flow Graphs

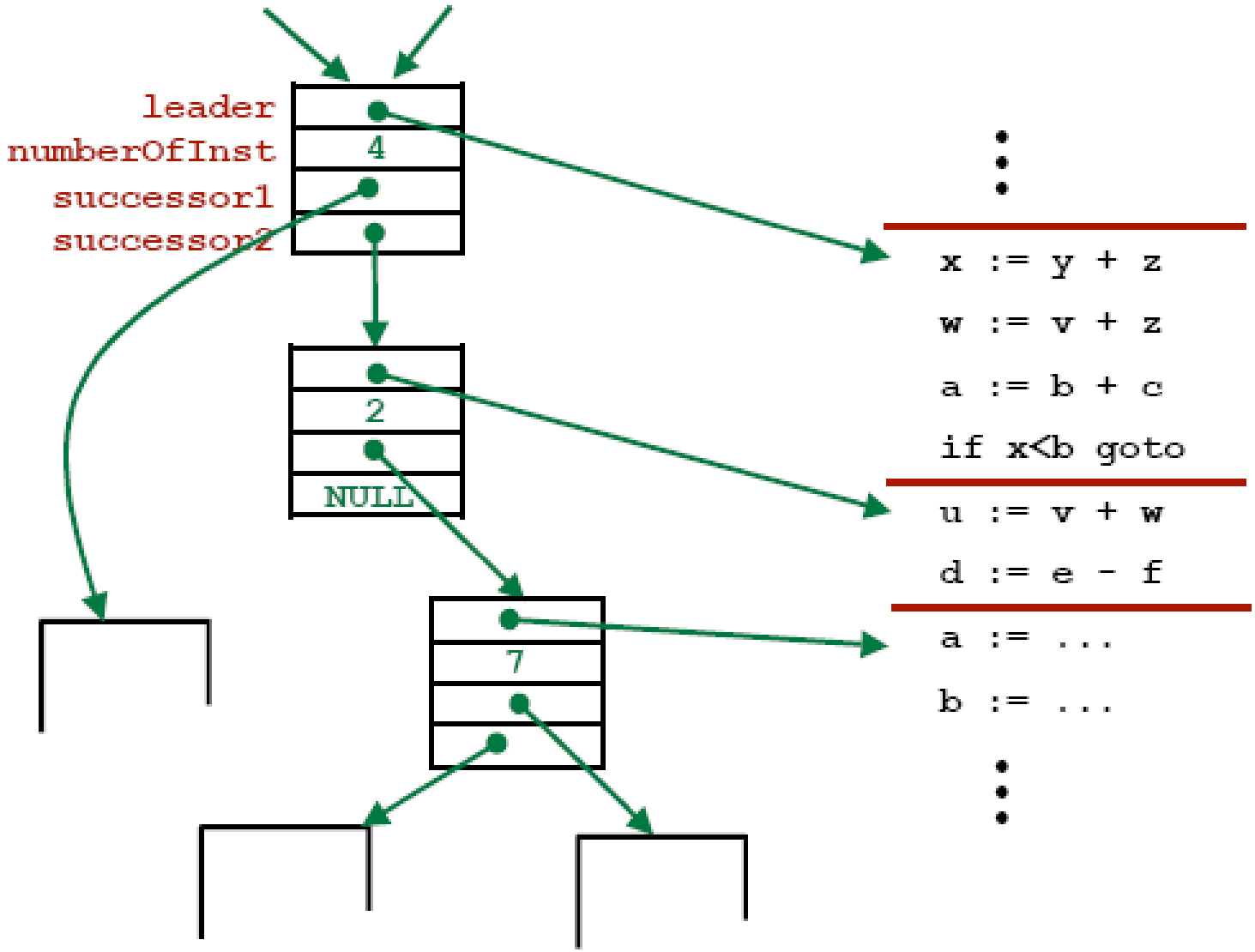
Definitions:

- Initial Block
- Predecessor Blocks
- Successor Blocks

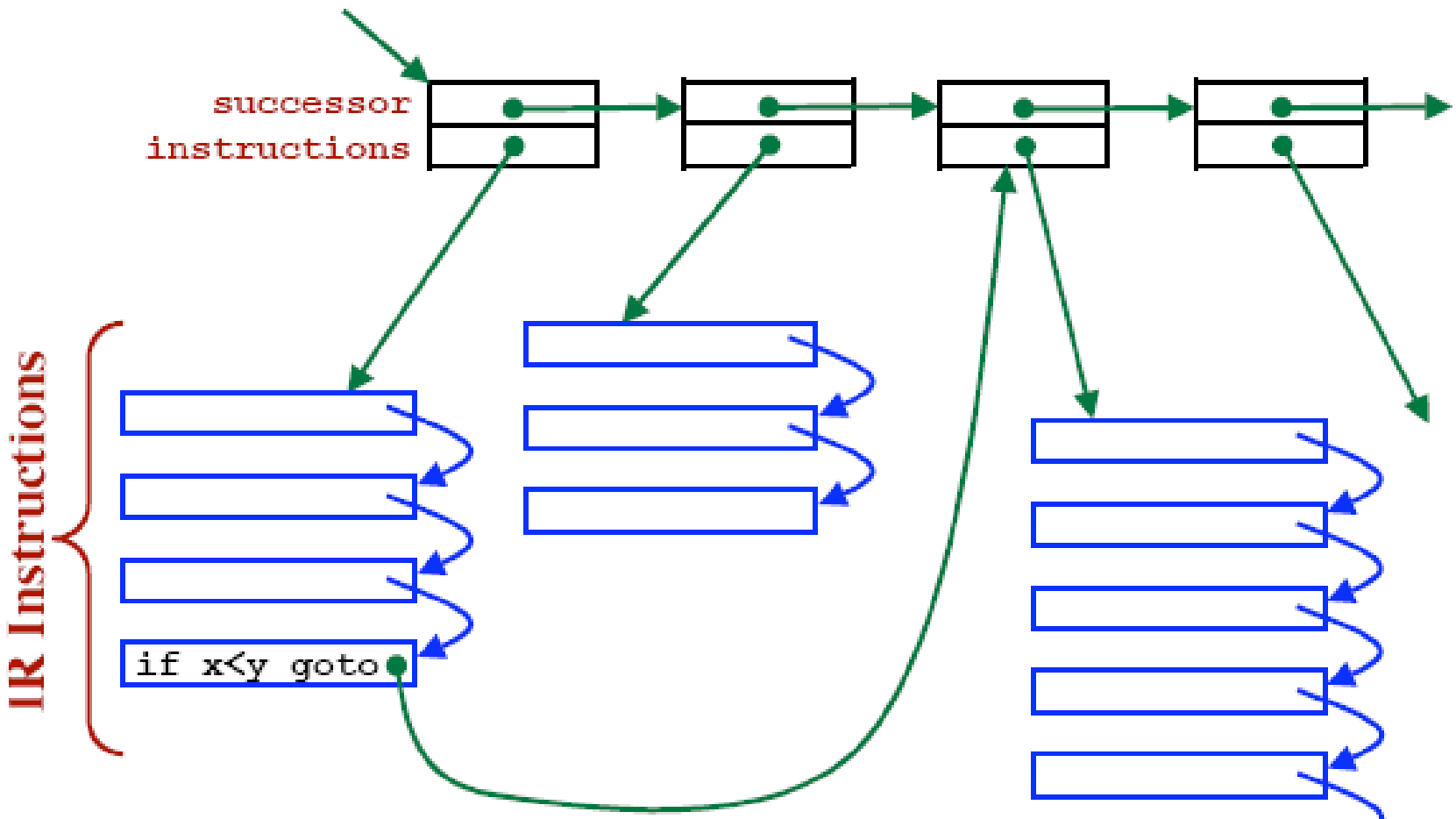




Representing Flow Graphs: Idea 1



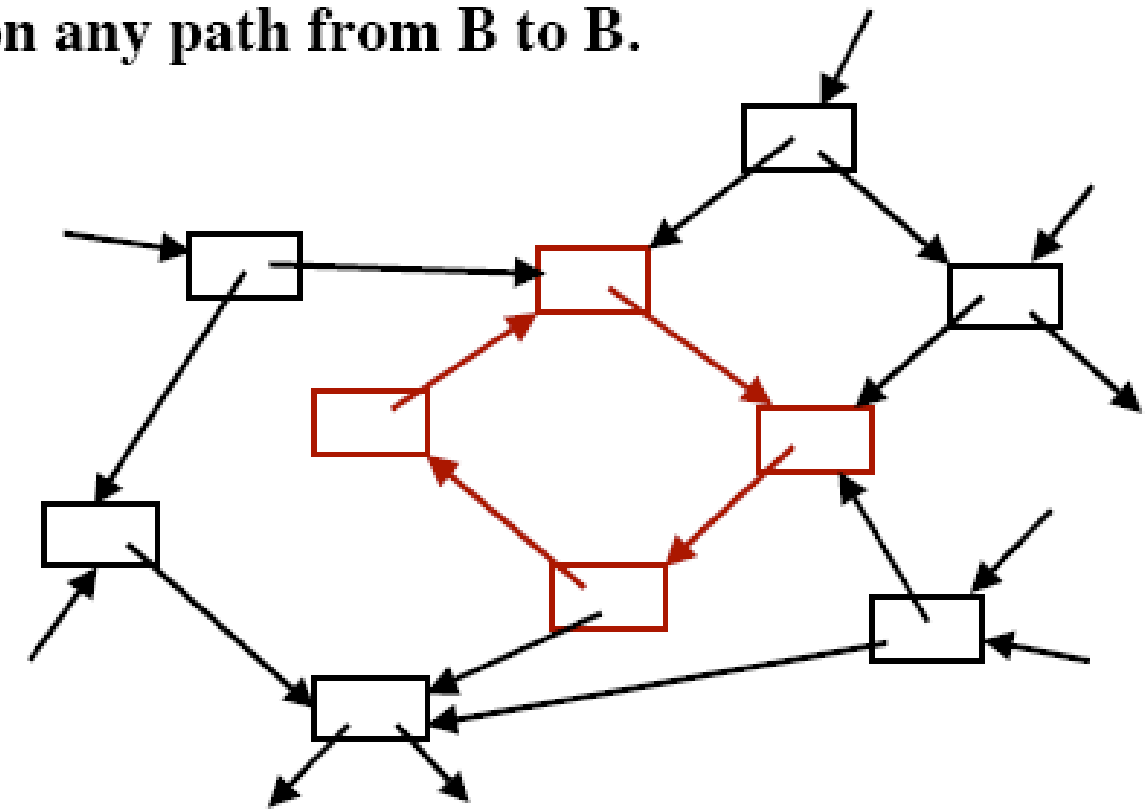
Representing Flow Graphs: Idea 2



Branch instructions now point to Basic Blocks
(...not to IR instructions, as before)

What is Loop?

A *cycle* in the flow graph.
Can go from B back to B.
A *path* from B to B.
All blocks on any path from B to B.



Natural Loop

Each loop has a unique entry (its "Header Block")

To reach any block in the loop (from outside the loop)
you must first go through the header block

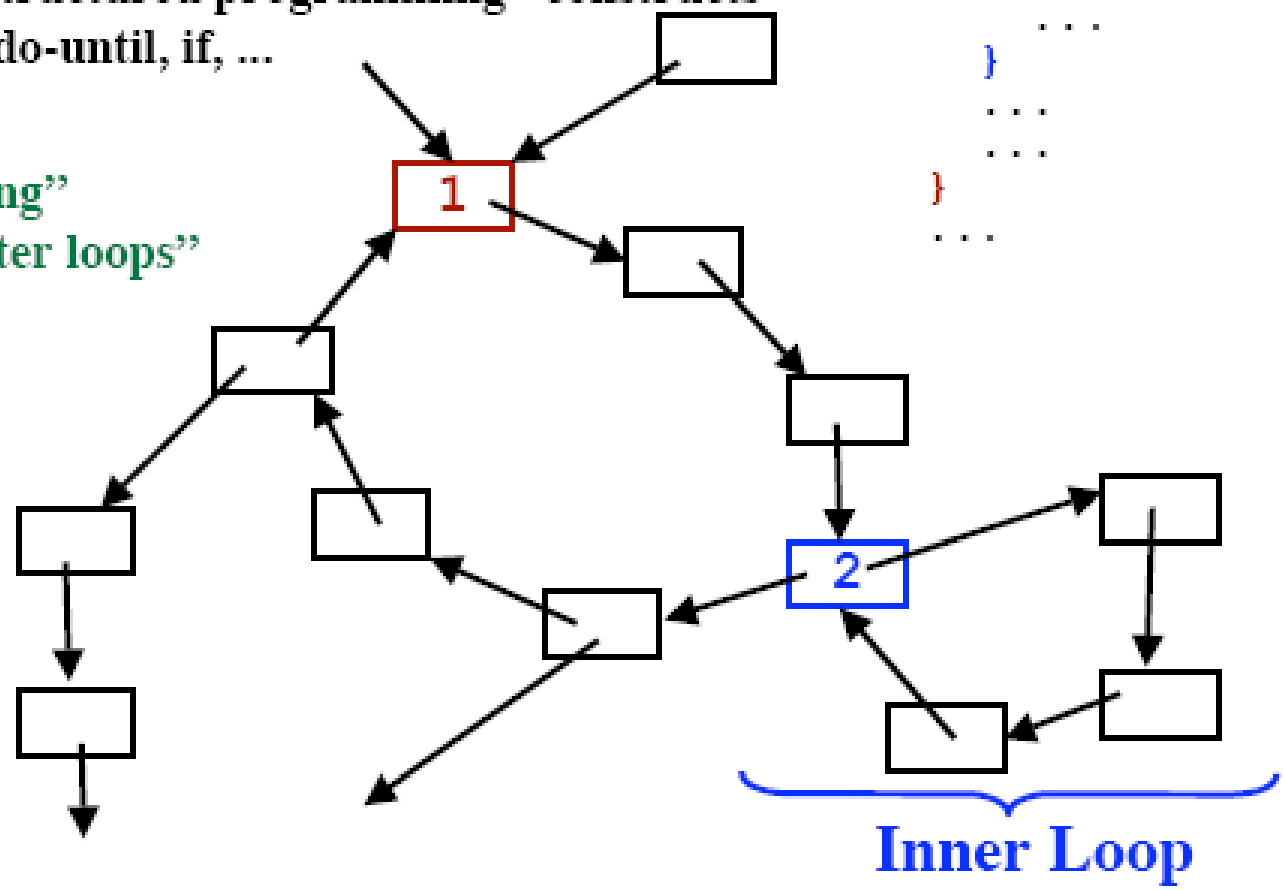
Result from "structured programming" constructs
while, for, do-until, if, ...

<u>Concepts:</u>

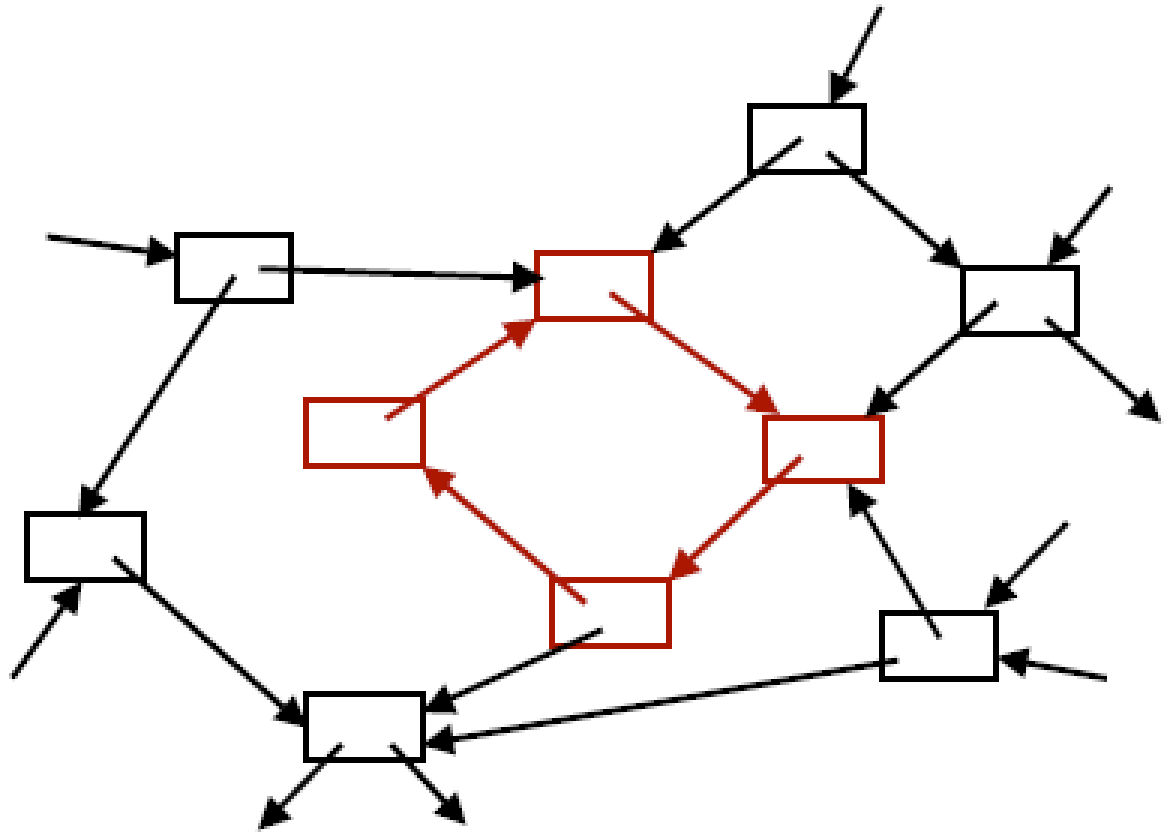
"loop nesting"

"inner / outer loops"

```
...
...
while(...) {
   ...
   ...
   while(...) {
      ...
      ...
   }
   ...
   ...
}
...
```



Loops with Multiple Entries

- Very un-natural
- Rare in assembly language programs
- Impossible in many programming languages

```
goto Lab45;
...
while (x<y) {
   ...
   Lab45:
   ...
}
```