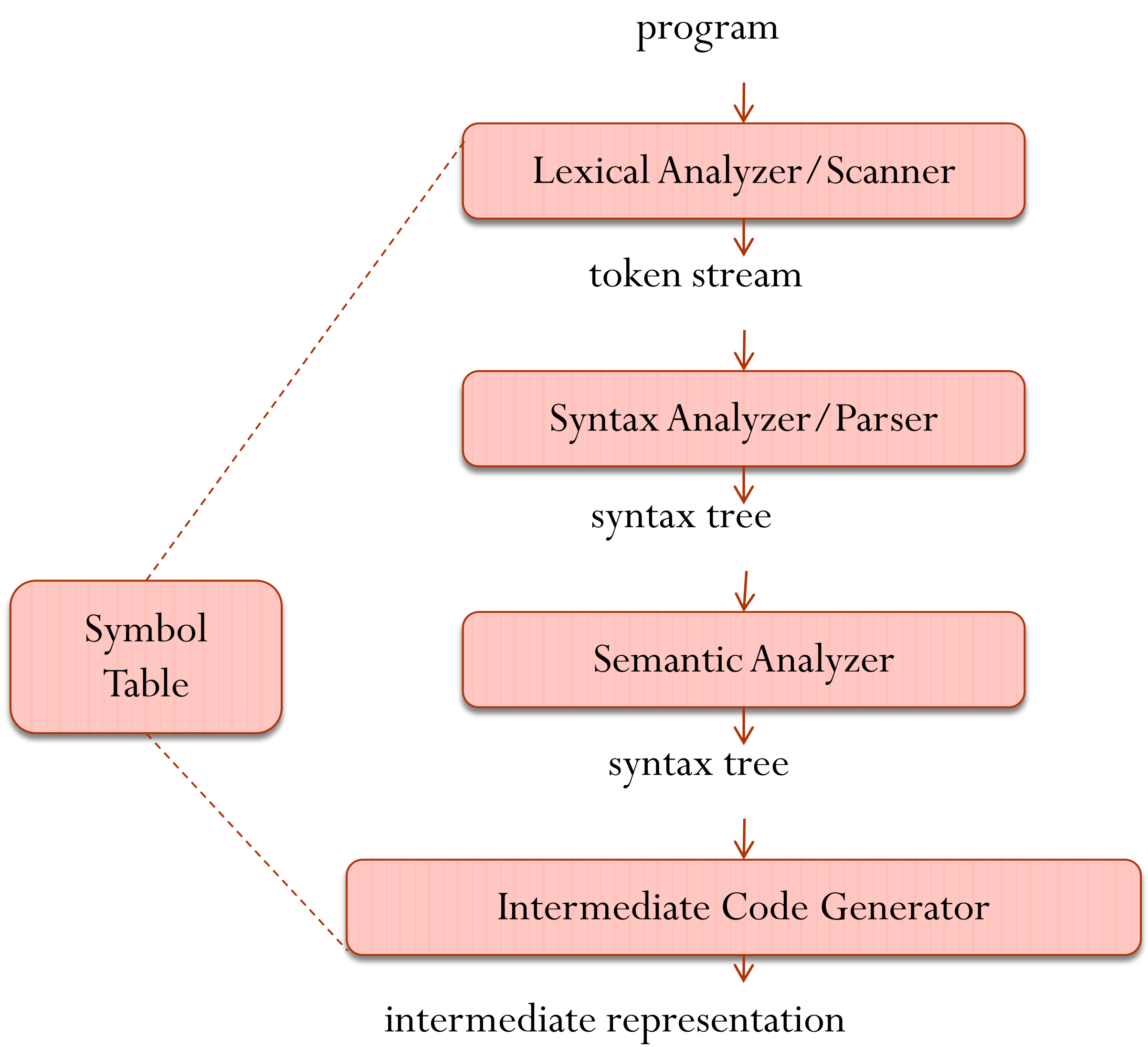
Compilers

Topic: Top Down Parsing

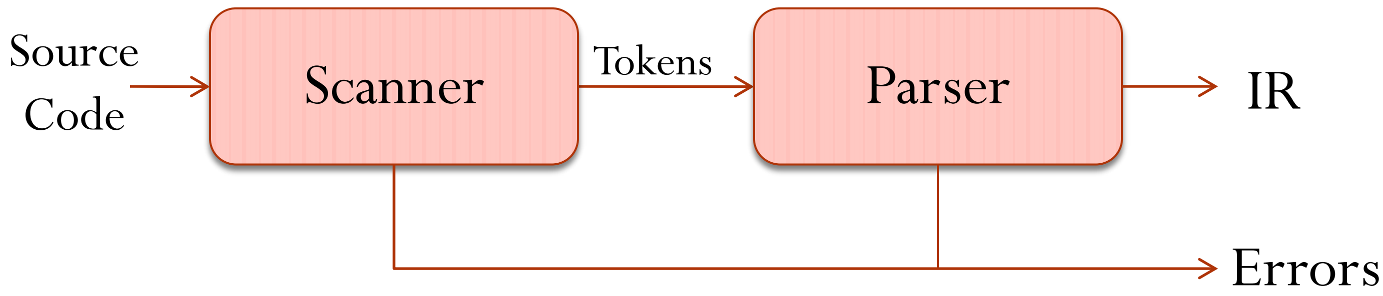
Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper's CS412 at Rice University

The Front End



The Front End: Scanner and Parser



Parser

- Takes as input a stream of tokens
- Checks if the stream of tokens constitutes a syntactically valid program of the language
- If the input program is syntactically correct
  - Output an intermediate representation of the code (like AST)
- If the input program has syntactic errors
  - Outputs relevant diagnostic information

Parsing Approaches

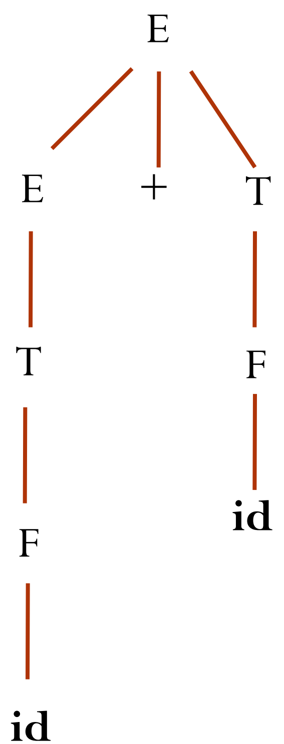
- Cocke-Younger-Kasami (CYK) algorithm can construct a parse tree for a given string and CFG in $\Theta(n^3)$ worst-case time.
- Earley's algorithm
  - $O(n^3)$ for general CFGs
  - $O(n^2)$ for unambiguous grammars
- We would like to have linear-time algorithms for parsing programs.

Parsing Techniques for Programming Langauges

- Key Idea: If we design the Syntax Rules for a Programming Language carefully, we may not require the full non-deterministic power of CFGs (and hence NPDAs).
  - For many Programming Languages this is the case.
- Two Parsing Techniques
  - Top-Down Parsing – Good for hand-coded parsers
  - Bottom-up Parsing – Good for Parsers generated by Automatic parser Generators

Top-Down Parsing versus Bottom-up Parsing

Top-down Parsing and Left-most Derivations



$$
\begin{aligned}
E &\Rightarrow E + T \\
&\Rightarrow T + T \\
&\Rightarrow F + T \\
&\Rightarrow id + T \\
&\Rightarrow id + F \\
&\Rightarrow id + id
\end{aligned}
$$

$E \rightarrow E + T \mid T$

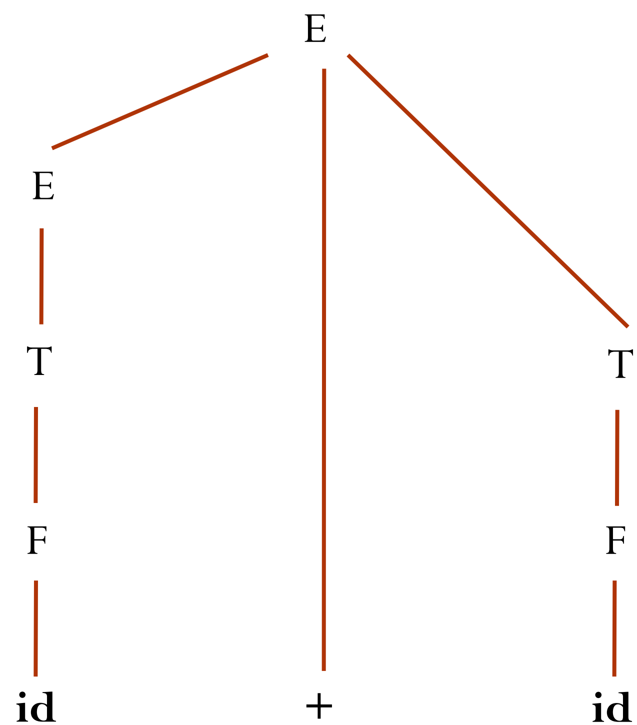
$T \rightarrow T * F \mid F$

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

$\mathbf{w = id + id}$

Lower fringe of the parse tree corresponds to left-most sentential forms.

Bottom-up Parsing and Right-most Derivations



$$
\begin{aligned}
&E \\
&\Rightarrow E + T \\
&\Rightarrow E + F \\
&\Rightarrow E + id \\
&\Rightarrow T + id \\
&\Rightarrow F + id \\
&\Rightarrow id + id
\end{aligned}
$$

Upper fringe of the parse tree corresponds to right-most sentential forms.

Parsing Techniques

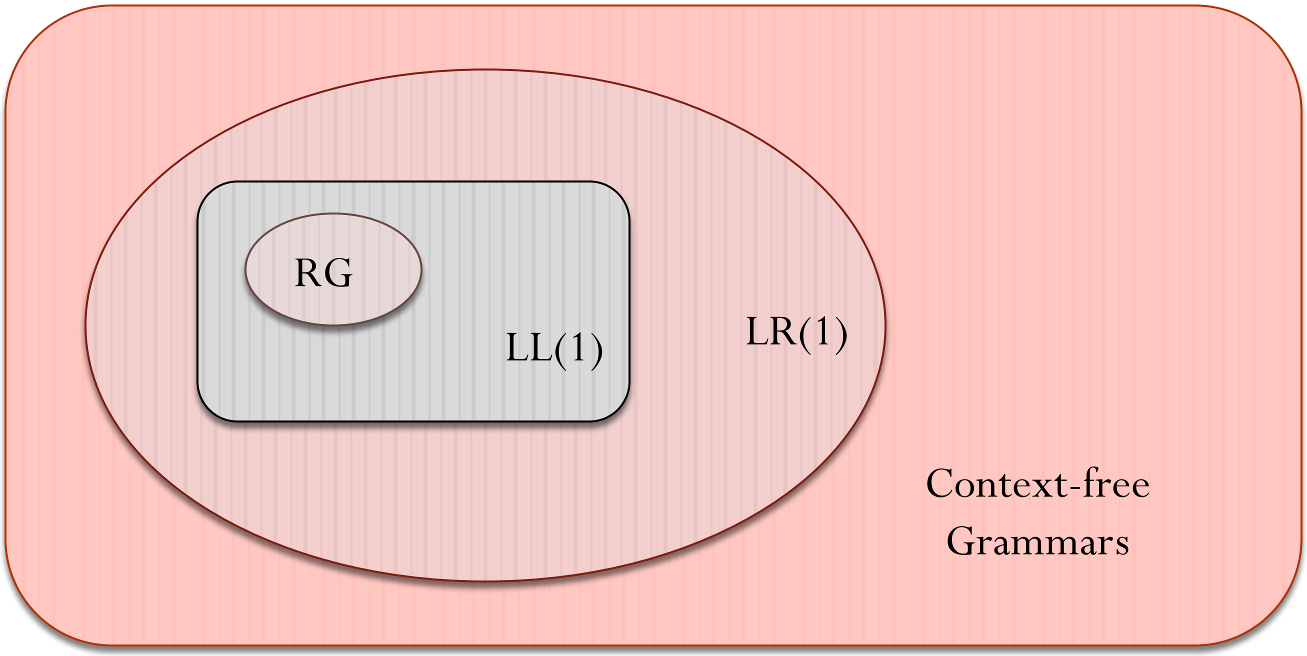
Top-down parsers (LL(1), recursive descent)

- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad "pick" ⇒ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)

- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars

Classes of CFGs



Top-Down Parsing – A Recursive-Descent Approach

Goal: Given an input string discover a derivation (or a parse tree) for it.

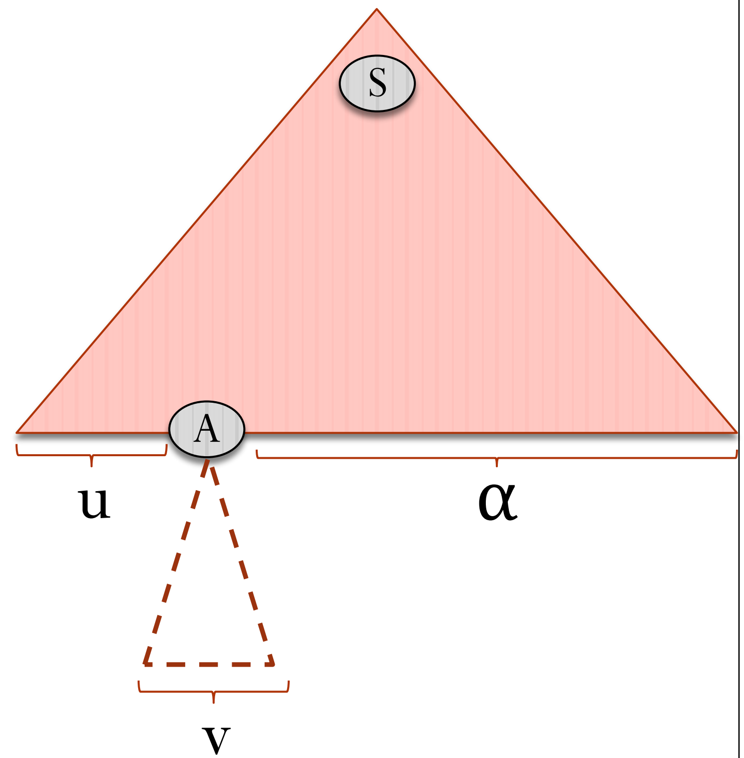
Approach: At step i

$$S \Rightarrow \gamma_1 \Rightarrow \gamma_2 \Rightarrow \ldots\ldots \gamma_{i-1} \Rightarrow \gamma_i \ldots\ldots \Rightarrow \gamma_{n-1} \Rightarrow \gamma_n \Rightarrow w$$

- Pick a non-terminal symbol A from the sentential form γ_{i-1}
  - Which one to pick? – In Top-Down Parsing we pick the left-most non-terminal.
- Apply a substitution rule corresponding to A
  - If we don't choose the right substitution rule we have to back-track later on.
- Key Idea: For certain CFGs (LL(1) Grammars) we can design a Bactrack-Free Parser.

Top-Down Parsing

1. Start with the Start Symbol as the root node.
2. At any point of time the leaves of the parse tree are labeled either as terminal or non-terminal symbols.
3. Pick the left-most leaf node which is labeled by a non-terminal.
4. Expand it using one of its substitution rules.
5. If there is an input mismatch, backtrack and try another production.



If the input string w = uy and v is not a prefix of y, then we have to backtrack.

Top Down Parsing – Example

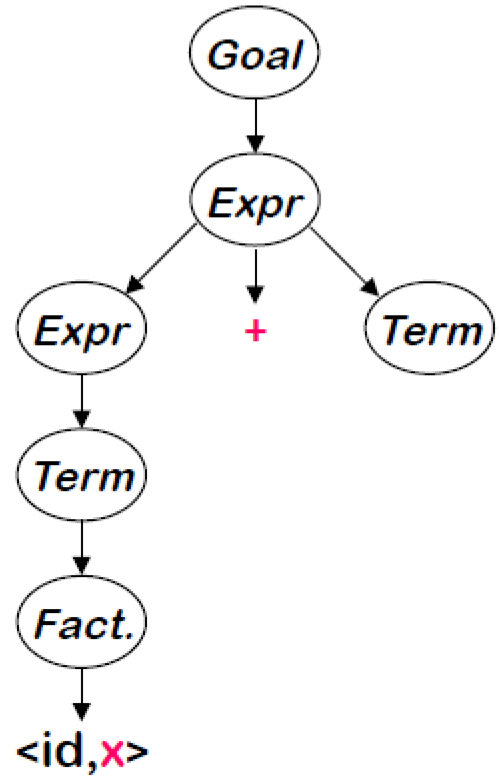
0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* * *Factor*
5		|	*Term* / *Factor*
6		|	*Factor*
7	*Factor*	→	<u>(</u> *Expr* <u>)</u>
8		|	<u>number</u>
9		|	<u>id</u>

And the input <u>x</u> - <u>2</u> * <u>y</u>

Top Down Parsing – Example

Let's try x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
1	*Expr* + *Term*	↑x - 2 * y
3	*Term* + *Term*	↑x - 2 * y
6	*Factor* + *Term*	↑x - 2 * y
9	<id,x> + *Term*	↑x - 2 * y
→	<id,x> + *Term*	x ↑- 2 * y



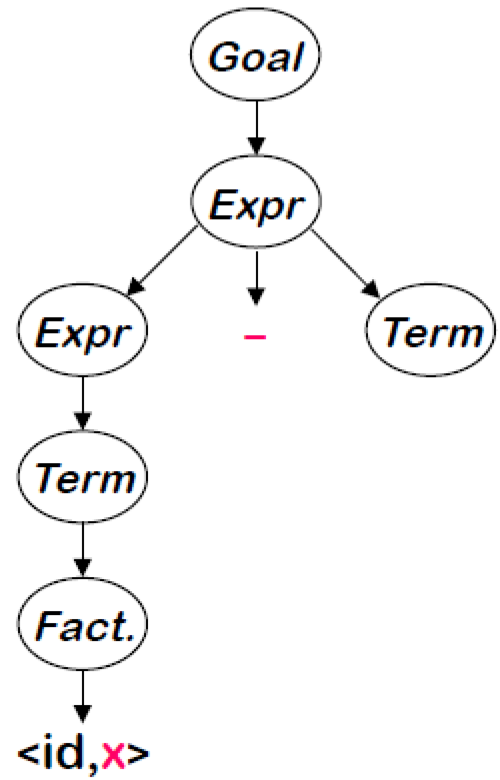
This worked well, except that "–" doesn't match "+"

The parser must backtrack to here

Top Down Parsing – Example

Continuing with x - 2 * y :

Rule	Sentential Form	Input
—	*Goal*	↑x - 2 * y
0	*Expr*	↑x - 2 * y
2	*Expr - Term*	↑x - 2 * y
3	*Term - Term*	↑x - 2 * y
6	*Factor - Term*	↑x - 2 * y
9	‹*id,x*› - *Term*	↑x - 2 * y
→	‹*id,x*› - *Term*	x ↑- 2 * y
→	‹*id,x*› - *Term*	x - ↑2 * y



Now, "-" and "-" match

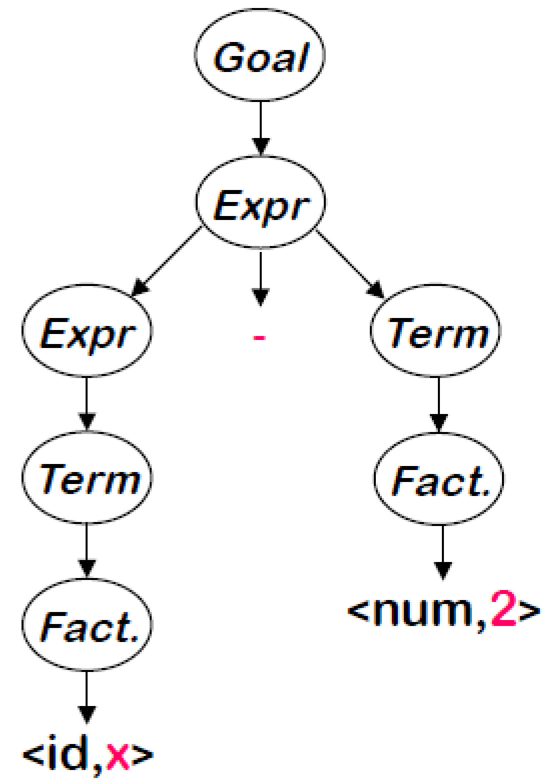
Now we can expand Term to match "2"

⇒ Now, we need to expand *Term* - the last *NT* on the fringe

Top Down Parsing – Example

Trying to match the "2" in x - 2 * y :

Rule	Sentential Form	Input
→	<id,x> - *Term*	x - ↑2 * y
6	<id,x> - *Factor*	x - ↑2 * y
8	<id,x> - <num,2>	x - ↑2 * y
→	<id,x> - <num,2>	x - 2 ↑* y



Where are we?

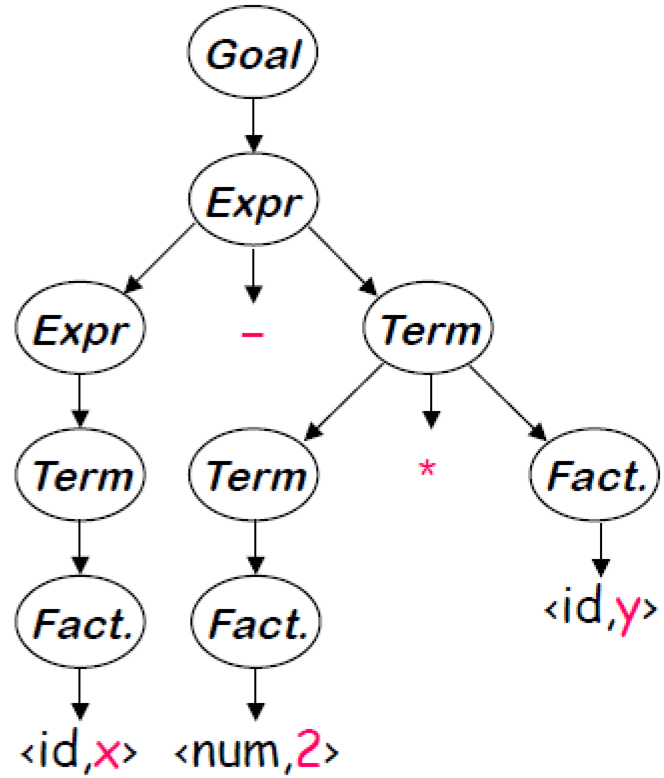
- "2" matches "2"
- We have more input, but no *NTs* left to expand
- The expansion terminated too soon

⇒ Need to backtrack

Top Down Parsing – Example

Trying again with "2" in x - 2 * y :

Rule	Sentential Form	Input
→	‹id,x› - *Term*	x - ↑2 * y
4	‹id,x› - *Term* * *Factor*	x - ↑2 * y
6	‹id,x› - *Factor* * *Factor*	x - ↑2 * y
8	‹id,x› - ‹num,2› * *Factor*	x - ↑2 * y
→	‹id,x› - ‹num,2› * *Factor*	x - 2 ↑* y
→	‹id,x› - ‹num,2› * *Factor*	x - 2 * ↑y
9	‹id,x› - ‹num,2› * ‹id,y›	x - 2 * ↑y
→	‹id,x› - ‹num,2› * ‹id,y›	x - 2 * y↑



The Point:

The parser must make the right choice when it expands a NT.
Wrong choices lead to wasted effort.

Recursive–Descent Parsing

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Fixing the function A()

Okay, after we fix the functions corresponding to each of the non-terminals, are we good?

1. How to choose the production at Step 1) (Iterate!)
2. How to implement the Backtracking?
3. When to report error?

Recursive Descent Parsing

$S \rightarrow A \mid B$

$A \rightarrow 0A1 \mid 01$

$B \rightarrow 1B0 \mid 10$

```
S( )
begin
  tempbufptr = bufptr
  if ( A( ) = success and *bufptr = EOF)
      then return success
  bufptr = tempbufptr  // bactrack
  if ( B( ) = success and *bufptr = EOF)
     then return success
  return fail  // all productions tried
end
```

bufptr is a global variable pointing to the current character or token in the input string.

Recursive Descent Parsing

$S \rightarrow A \mid B$

$A \rightarrow 0A1 \mid 01$

$B \rightarrow 1B0 \mid 10$

```
A( )
begin
  tempbufptr = bufptr
  if ( A_1( ) = success ) return success
  bufptr = tempbufptr // bactrack
  if ( A_2( ) = success ) return success
   bufptr = tempbufptr //backtrack
  return failure
end
```

```
B( )
begin
  tempbufptr = bufptr
  if ( B_1( ) = success ) return success
  bufptr = tempbufptr // bactrack
  if ( B_2( ) = success ) return success
   bufptr = tempbufptr //backtrack
  return failure
end
```

Recursive Descent Parsing

```
A_1( )
begin
 tempbuffptr = buffptr
 if ( *buffptr ≠ '0' ) then return failure
 buffptr = buffptr + 1
 if ( A( ) = failure ) then
   buffptr = tempbuffptr
   return failure
 end
 if ( *buffptr ≠ '1' ) then
   buffptr = tempbuffptr
   return failure
 end
 buffptr = buffptr + 1
 return success
end
```

```
A_2( )
begin
 tempbuffptr = buffptr
 if ( *buffptr ≠ '0' ) then return failure
 buffptr = buffptr + 1
 if ( *buffptr ≠ '1' ) then
   buffptr = tempbuffptr
   return failure
 end
 buffptr = buffptr + 1
 return success
end
```

Recursive Descent Parsing

```
B_1( )
begin
 tempbuffptr = buffptr
 if ( *buffptr ≠ '1' ) then return failure
 buffptr = buffptr + 1
 if ( B( ) = failure ) then
   buffptr = tempbuffptr
   return failure
 end
 if ( *buffptr ≠ '0' ) then
   buffptr = tempbuffptr
   return failure
 end
 buffptr = buffptr + 1
 return success
end
```

```
B_2( )
begin
 tempbuffptr = buffptr
 if ( *buffptr ≠ '1' ) then return failure
 buffptr = buffptr + 1
 if ( *buffptr ≠ '0' ) then
   buffptr = tempbuffptr
   return failure
 end
 buffptr = buffptr + 1
 return success
end
```

Recursive Descent Parsing

$E \rightarrow E + T \mid E - T$

$T \rightarrow T * F \mid T/F \mid F$

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

Can we do recursive descent parsing on the above grammar?

Problem: The grammar is Left-Recursive.

Left-Recursion

Top-down parsers cannot handle left-recursive grammars

Def: A grammar is left recursive if $\exists\ A \in NT$ such that

$$\exists \text{ a derivation } A \Rightarrow^{+} A\alpha, \text{ for some string } \alpha \in (NT \cup T)^{+}$$

- Our expression grammar is left recursive
- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Eliminating Left Recursion

Example: $A \rightarrow A\alpha \mid \beta$ where $\alpha, \beta \in (NT \cup T)^*$. Assume β doesn't start with the non-terminal A.

Grammar with left recursion eliminated:

$$\left.\begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}\right\} \quad \text{Key Idea: } A \Rightarrow^+ \beta\alpha^*$$

$$\left.\begin{array}{l} A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m \\ A \rightarrow \beta_1 A' \mid \ldots \mid \beta_m A' \\ A' \rightarrow \alpha_1 A' \mid \ldots \mid \alpha_n A' \mid \varepsilon \end{array}\right\} \quad A \Rightarrow^+ (\beta_1 + \ldots + \beta_m)(\alpha_1 + \ldots + \alpha_n)^*$$

Eliminating Left Recursion

Grammar with left-recursion

1. $E \rightarrow E + T \mid E - T \mid T$
2. $T \rightarrow T * F \mid T/F \mid F$
3. $F \rightarrow (E) \mid \mathbf{id}$

Grammar after eliminating Left-recursion

1. $E \rightarrow T E'$
2. $E' \rightarrow + T E' \mid - T E' \mid \varepsilon$
3. $T \rightarrow F T'$
4. $T' \rightarrow * F T' \mid / F T' \mid \varepsilon$
5. $F \rightarrow (E) \mid \mathbf{id}$

Eliminating Left Recursion

Eliminate left recursion from the following grammar.

$$S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$$

Indirect Left Recursion

Example 1: $S \rightarrow Aa \mid b$

$A \rightarrow Sd \mid cA \mid \varepsilon$

$$S \Rightarrow Aa \Rightarrow Sda$$

Example 2: $S \rightarrow aS \mid B$

$B \rightarrow Sb \mid b$

Reading Exercise: Left recursion elimination algorithm in the text book.

Eliminating Left Recursion

The transformation eliminates immediate left recursion

What about more general, indirect left recursion ?

The general algorithm:

arrange the NTs into some order $A_1, A_2, \ldots, A_n$

for $i \leftarrow 1$ to n

for $s \leftarrow 1$ to $i - 1$

replace each production $A_i \rightarrow A_s\gamma$ with $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,

where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for A_s

eliminate any immediate left recursion on A_i

using the direct transformation

Must start with 1 to ensure that $A_1 \rightarrow A_1\,\beta$ is transformed

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$), and no epsilon productions

Eliminating Left Recursion

How does this algorithm work?

1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding A_i has no non-terminal A_s in its *rhs*, for $s < i$
4. Last step in outer loop converts any direct recursion on A_i to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion

At the start of the i^{th} outer loop iteration

For all $k < i$, no production that expands A_k contains a non-terminal A_s in its rhs, for $s < k$

Example

- Order of symbols: *G*, *E*, *T*

1. $A_i = G$	2. $A_i = E$	3. $A_i = T$, $A_s = E$	4. $A_i = T$
$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$	$G \rightarrow E$
$E \rightarrow E + T$	$E \rightarrow T\,E'$	$E \rightarrow T\,E'$	$E \rightarrow T\,E'$
$E \rightarrow T$	$E' \rightarrow +\,T\,E'$	$E' \rightarrow +\,T\,E'$	$E' \rightarrow +\,T\,E'$
$T \rightarrow E \sim T$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
$T \rightarrow \underline{\text{id}}$	$T \rightarrow E \sim T$	$T \rightarrow T\,E' \sim T$	$T \rightarrow \underline{\text{id}}\,T'$
	$T \rightarrow \underline{\text{id}}$	$T \rightarrow \underline{\text{id}}$	$T' \rightarrow E' \sim T\,T'$
			$T' \rightarrow \varepsilon$

Using Look-ahead to Eliminate Backtracking

Question: Can we eliminate backtracking by using k look-ahead symbols ?

Note: k is constant and ideally we like k = 1.

Example 1: $S \rightarrow 0S \mid 11S \mid \varepsilon$

Example 2: $S \rightarrow 11S \mid 10S \mid \varepsilon$

Example 3: $S \rightarrow 0S1 \mid 1S0 \mid c$

Example 4: $S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$

Example 5: $S \rightarrow AeA \mid AfA \qquad A \rightarrow 0A \mid 1A \mid \varepsilon$

Example 6: $S \rightarrow 11S \mid 111S \mid \varepsilon$

Example 7: $S \rightarrow A \mid B \qquad A \rightarrow 0A \mid \varepsilon \qquad B \rightarrow 1B \mid \varepsilon$

Using Look-ahead to Eliminate Backtracking

Can we construct a non-backtracking parser for this expression grammar using constant amount of look-ahead?

1. $E \rightarrow T\ E'$
2. $E' \rightarrow +\ T\ E' \mid -\ T\ E' \mid \varepsilon$
3. $T \rightarrow F\ T'$
4. $T' \rightarrow *\ F\ T' \mid /\ F\ T' \mid \varepsilon$
5. $F \rightarrow (\ E\ ) \mid \textbf{id}$

Question: When to use null-production to quash a non-terminal symbol?

Using Look-ahead to Eliminate Backtracking

Can we construct a non-backtracking parser for this expression grammar?

1. $E \rightarrow T\ E'$
2. $E' \rightarrow +\ T\ E'\ |\ -\ T\ E'\ |\ \varepsilon$
3. $T \rightarrow F\ T'$
4. $T' \rightarrow *\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon$
5. $F \rightarrow (\ E\ )\ |\ \mathbf{id}\ |\ \mathbf{id}\ [\ Elist\ ]\ |\ \mathbf{id}\ (\ Elist\ )$
6. $Elist \rightarrow E\ ,\ Elist\ |\ E$

Can we transform the grammar so that we can construct a non-back tracking parser with just one symbol look ahead?

Using Look-ahead to Eliminate Backtracking

Original Grammar

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'\ |\ -\ T\ E'\ |\ \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon$

$F \rightarrow (\ E\ )\ |\ \mathbf{id}\ |\ \mathbf{id}\ [\ \text{Elist}\ ]\ |\ \mathbf{id}\ (\ \text{Elist}\ )$

$\text{Elist} \rightarrow E\ ,\ \text{Elist}\ |\ E$

Transformed Grammar

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E'\ |\ -\ T\ E'\ |\ \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T'\ |\ /\ F\ T'\ |\ \varepsilon$

$F \rightarrow (\ E\ )\ |\ \mathbf{id}\ \ \text{Args}$

$\text{Args} \rightarrow [\ \text{Elist}\ ]\ |\ (\ \text{Elist}\ )\ |\ \varepsilon$

$\text{Elist} \rightarrow E\ \text{Elist}'$

$\text{Elist}' \rightarrow ,\ \text{Elist}\ |\ \varepsilon$

1. This grammar transformation is called Left Factoring.
2. Hey, now can do it with just one symbol look ahead!

Left Factoring

Left Factoring is a grammar transformation which enables non-backtracking recursive descent parsing for certain grammars.

Example: $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Left Factored Grammar

$$A \rightarrow \alpha A'$$

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

Algorithm for Left-Factoring

For each non-terminal A

- Find the longest prefix α common to two or more of its alternatives
- Replace all of the A-productions

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

Represents the rest of productions

with

$$A \rightarrow \alpha A' \mid \gamma$$

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

- Repeatedly apply the above transformation until no two alternatives for a non-terminal have a common prefix.

Left Factoring

Example 1:

Bexpr → Bexpr **or** Bterm | Bterm

Bterm → Bexpr **and** Bfactor | Bfactor

Bfactor → **not** Bfactor | (Bexpr) | **true** | **false**

Left Factoring

Example 2:

$$S \rightarrow S\ S + \mid S\ S * \mid a$$

LL(1) Grammars

Predictive Parsing

- Problem: Given the left-most non-terminal to expand on the fringe of a parse tree, which production to use?
- Key Idea: Can we make use of the next k tokens to make that decision in a deterministic fashion?
- Grammars for which this is possible are called LL(k) grammars.
- Intuition: Trying to beat non-determinism using look-ahead information (Recall NPDAs are strictly more powerful DPDAs).
- It is undecidable to know whether or not a backtrack-free grammar exists for an arbitrary CFL.

LL(1) Grammars

Backtrack-free top down parsers can be constructed for LL(1) grammars.

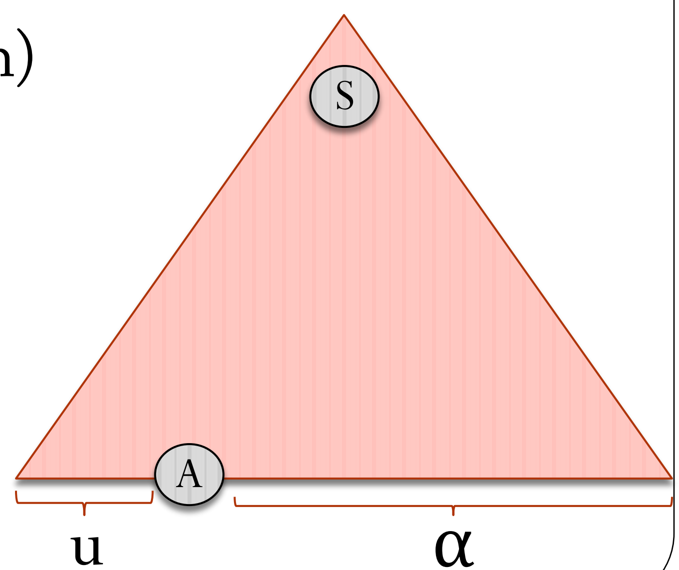
LL(1) – Scan the input from Left-to-Right.

LL(1) – Parser produces a left-most derivation.

LL(1) – 1 symbol lookahead.

Predictive Parsing

- Input string w = uav where 'a' is the current token.
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_m$
- Question: Can we choose the substitution rule to apply by using the look-ahead symbol (current token) a ? (LL(1) approach)
- More generally: Can we use the next k-tokens to decide which substitution look to apply? (LL(k) approach)



FIRST Function

Given a CFG $G = (NT, T, P, S)$

- For $X \in NT$,

$$FIRST(X) = \{a \in T \mid \exists\, X \Rightarrow^{*} a\beta\} \cup \{\varepsilon \mid \exists\, X \Rightarrow^{*} \varepsilon\}$$

- For $\alpha \in (NT \cup T)^{*}$, $FIRST(\alpha)$ consists of set of terminals that begin sentential forms derived from α.

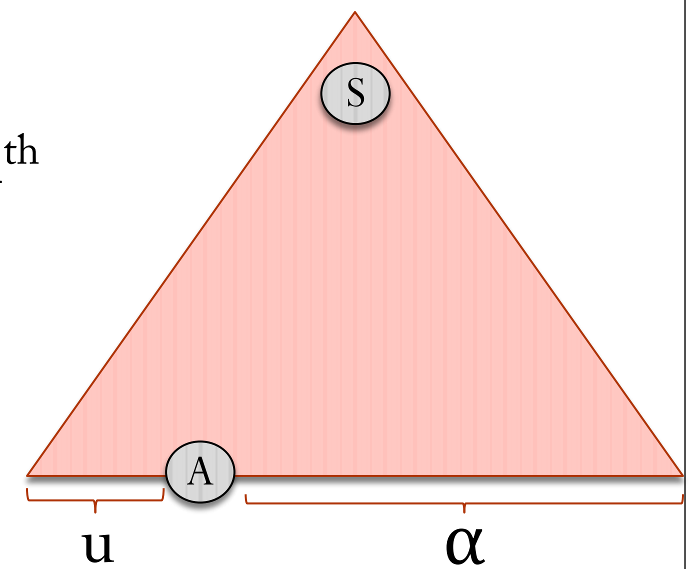
- If $\alpha \Rightarrow^{*} \varepsilon$, then ε is also in $FIRST(\alpha)$

$$FIRST(\alpha) = \{a \in T \mid \alpha \Rightarrow^{*} a\beta\} \cup \underbrace{\{\varepsilon\}}_{\text{only if } \alpha \Rightarrow^{*} \varepsilon}$$

Application of FIRST function

- Input string $w = uav$ where 'a' is the current token.
- $A \rightarrow \alpha_1 \mid \alpha_2 \mid .... \mid \alpha_n$

Our intuition: If $a \in \text{FIRST}(\alpha_i)$ then use the i^{th} production to expand A.



Problem Scenario: $a \in \text{FIRST}(\alpha_i), \text{FIRST}(\alpha_k)$ for $i \neq k$.

Question: Are we okay if for all $1 \leq i \neq k \leq n$

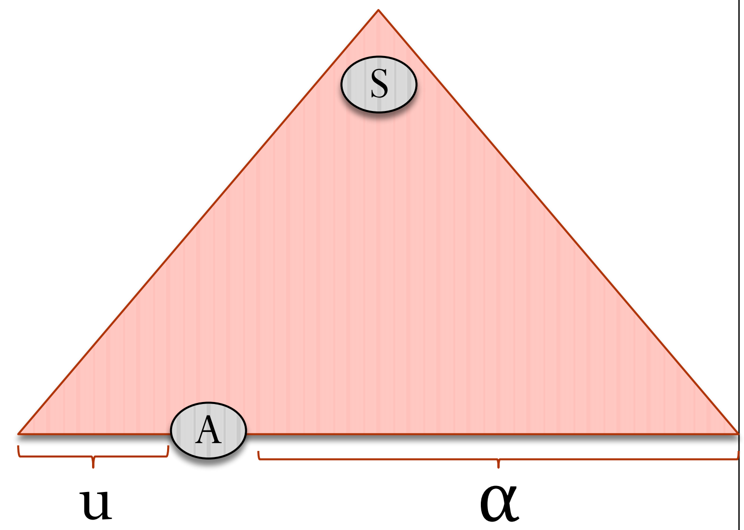
$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_k) = \emptyset \text{ ?}$$

Nope! Recall the grammar $S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$

Application of FIRST function

- Input string w = uav where 'a' is the current token.
- $A \rightarrow \alpha_1 \mid \alpha_2$
- $a \in FIRST(\alpha_1)$, $a \notin FIRST(\alpha_2)$ and

  $\varepsilon \in FIRST(\alpha_2)$

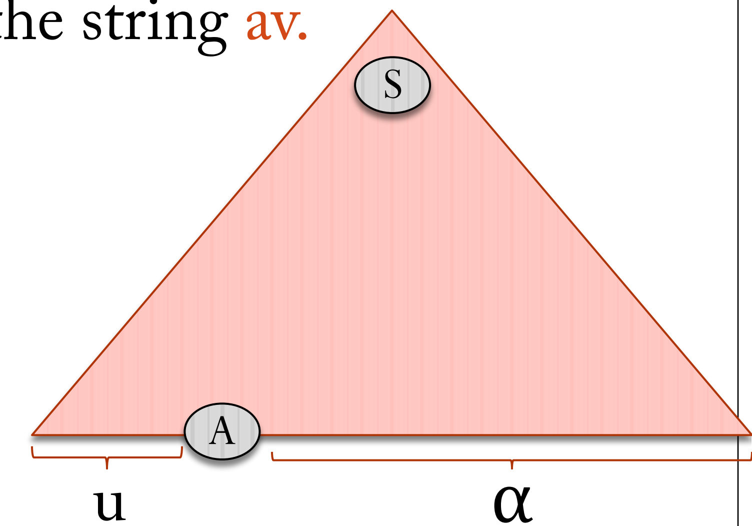


Two Possibilities

1. Case 1: Apply the production $A \rightarrow \alpha_1$ to generate 'a'
2. Case 2: Apply the production $A \rightarrow \alpha_2$ to reduce the non-terminal A to ε and hope α generates the rest of the string av.

Application of FIRST function

- Case 2: Apply the production $A \rightarrow \alpha_2$ to reduce the non-terminal A to ε and hope α generates the rest of the string av.



Necessary Condition for a Successful Case 2:

There exists a derivation of the form

$$S \Rightarrow^* uAa\beta \text{ for some } \beta \in (NT \cup T)^*.$$

FOLLOW Function

Given a CFG $G = (N, T, P, S)$

- For a non-terminal $A \in N$, FOLLOW(A) consists of set of terminals that can appear immediately to the right of A in some sentential form.
- In other words the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some $\alpha, \beta \in (V \cup T)^*$.
- If A can be the rightmost symbol in some sentential form (i.e., there exists a derivation $S \Rightarrow^* \alpha A$), then \$ is on FOLLOW(A)

LL(1) Grammars

Let G = (N, T, P, S) be a CFG. Consider the productions corresponding to a non-terminal A

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

Def: $$\mathrm{FIRST}^+(\alpha_i) = \begin{cases} \mathrm{FIRST}(\alpha_i) \text{ if } \varepsilon \notin \mathrm{FIRST}(\alpha_i) \\ \mathrm{FIRST}(\alpha_i) \cup \mathrm{FOLLOW}(A) \text{ if } \varepsilon \in \mathrm{FIRST}(\alpha_i) \end{cases}$$

LL(1) Grammars

Def: A CFG is said to LL(1) if for any non-terminal A with productions

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid .... \mid \alpha_n \text{ and for } 1 \leq i \neq j \leq n$$

$$\text{FIRST}^+(\alpha_i) \cap \text{FIRST}^+(\alpha_j) = \emptyset.$$

- Is this grammar $S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$ LL(1)?

LL(1) Grammars

Alternative Def: A CFG is said to be LL(1) if for any non-terminal A with productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid .... \mid \alpha_n$

1. $FIRST(\alpha_i) \cap FIRST(\alpha_j) = \emptyset$ (for $1 \leq i \neq j \leq n$)
2. If $\varepsilon \in FIRST(\alpha_i)$ then $FOLLOW(A) \cap FIRST(\alpha_j) = \emptyset$ (for $1 \leq i \neq j \leq n$)

Recursive–Descent Parsing

```
void A() {
        Choose an A-production, A → X1 X2 ··· Xk;
        for ( i = 1 to k ) {
                if ( Xi is a nonterminal )
                        call procedure Xi();
                else if ( Xi equals the current input symbol a )
                        advance the input to the next symbol;
                else /* an error has occurred */;
        }
}
```

Question: How can we fix this function if the grammar has LL(1) property?

Table Driven Approach for LL(1) Grammars

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr'*
2	*Expr'*	→	+ *Term Expr'*
3		|	- *Term Expr'*
4		|	ε
5	*Term*	→	*Factor Term'*
6	*Term'*	→	* *Factor Term'*
7		|	/ *Factor Term'*
8		|	ε
9	*Factor*	→	number
10		|	id
11		|	(*Expr*)

Prod'n	FIRST+
0	(,id,num
1	(,id,num
2	+
3	-
4	ε,), eof
5	(,id,num
6	*
7	/
8	ε,+,-,), eof
9	number
10	id
11	(

LL(1) Expression Parsing Table

	+	-	*	/	Id	Num	(	)	EOF
Goal	—	—	—	—	0	0	0	—	—
Expr	—	—	—	—	1	1	1	—	—
Expr'	2	3	—	—	—	—	—	4	4
Term	—	—	—	—	5	5	5	—	—
Term'	8	8	6	7	—	—	—	8	8
Factor	—	—	—	—	10	9	11	—	—

Construction of Predictive Parsing Table

	Id	+	-	*	/	(	)	$
E	E → T E’					E → T E’		
E’		E’ → +T E’	E’ → -T E’				E’ → ε	E’ → ε
T	T → F T’					T → F T’		
T’		T’ → ε	T’ → ε	T’ → *F T’	T’ → /FT’		T’ → ε	T’ → ε
F	F → id					F → (E)		

1. Rows indexed by Non-terminals
2. Columns indexed by terminals
3. For a production $A \rightarrow \alpha$, add this production to all table entries $M[A, a]$ for all $a \in \text{FIRST}^{+}(\alpha)$.

E → T E’
E’ → + T E’ | - T E’ | ε
T → F T’
T’ → * F T’ | / F T’ | ε
F → (E) | **id**

Can a table entry have multiple productions in it?

Table Driven Predictive Parser

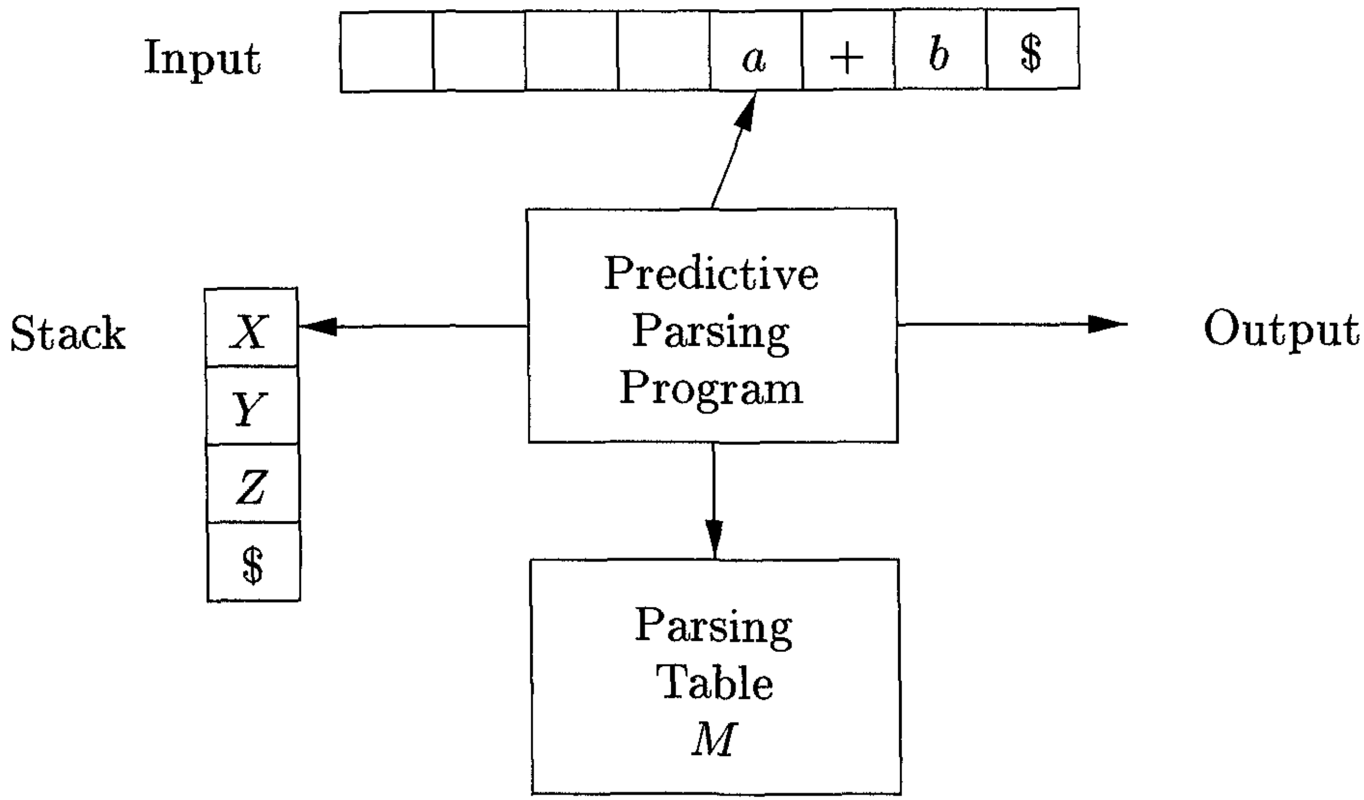


Table Driven Parsing Algorithm

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ... , Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Resolving Ambiguities in Certain LL(1) Grammars

Construct a parsing table for the following grammar?

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

1. Compute $FIRST^+$ sets
    1. $S \rightarrow iEtSS'$ $FIRST^+(iEtSS') = \{ i \}$
    2. $S \rightarrow a$ $FIRST^+(a) = \{ a \}$
    3. $S' \rightarrow eS$ $FIRST^+(eS) = \{ e \}$
    4. $S' \rightarrow \varepsilon$ $FIRST^+(\varepsilon) = \{ e, \$ \}$
    5. $E \rightarrow b$ $FIRST^+(b) = \{ b \}$

Resolving Ambiguities in Certain LL(1) Grammars

	i	t	e	b	a	$
S	S → iEtSS’				S → a	
S’			S’ → eS S’ → ε			S’ → ε
E				E → b		

1. FIRST$^+$(iEtSS’) = { i } S → iEtSS’
2. FIRST$^+$ (a) = { a } S → a
3. FIRST$^+$ (eS) = { e } S’ → eS
4. FIRST$^+$ (ε) = { e, $ } S’ → ε
5. FIRST$^+$ (b) = { b } E → b

1. Not an LL(1) Grammar.
2. There exists no equivalent LL(1) grammar.
3. However, we can easily resolve the ambiguity to suit our semantics.

LL(1) Grammars versus Regular Grammars

Question: Are LL(1) grammars more powerful than regular grammars?

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

What is not covered?

- Error Recovery Techniques

Computing FIRST and FOLLOW functions

FIRST Function

$$S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$$

Compute

1. FIRST(0)
2. FIRST(1)
3. FIRST(S)

FIRST Function

E → T E’

E’ → + T E’ | - T E’ | ε

T → F T’

T’ → * F T’ | / F T’ | ε

F → (E) | **id**

Compute

1. FIRST(E)
2. FIRST(E’)
3. FIRST(T’)
5. FIRST(F)
6. FIRST(T)

Computing FIRST

Computing FIRST(X)

- If X is a terminal, then FIRST(X) = { X }
- If X is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production
  - Include all non- ε symbols from FIRST(Y_1) in FIRST(X)
  - Include all non- ε from FIRST(Y_i) in FIRST(X) if ε is present in FIRST(Y_1), FIRST(Y_2), …., FIRST(Y_{i-1})
  - Include ε in FIRST(X) if $\varepsilon \in$ FIRST(Y_i), for all $1 \leq i \leq k$
- If $X \rightarrow \varepsilon$ is a production, then add ε to FIRST(X).

Note:This is not actually an algorithm.

FIRST Function

Notation: FST_X indicated FIRST(X).

$S \rightarrow A \mid B$

$A \rightarrow Ba$

$B \rightarrow Ab \mid \varepsilon$

Compute

1. FIRST(S)
2. FIRST(A)
3. FIRST(B)

Key Observation: Approximations at the end of iterations 3 and 4 are one and the same.

Iteration	FST_S	FST_A	FST_B
Initial Approximation	{}	{}	{}
1.1 (Update FST_S)	{}	{}	{}
1.2 (Update FST_A)	{}	{}	{}
1.3 (Update FST_B)	{}	{}	{ ε }
2.1 (Update FST_S)	{ ε }	{}	{ ε }
2.2 (Update FST_A)	{ ε }	{ a }	{ ε }
2.3 (Update FST_B)	{ε }	{ a }	{a, ε }
3.1 (Update FST_S)	{ a, ε }	{ a }	{ a, ε }
3.2 (Update FST_A)	{ a, ε }	{ a }	{ a, ε }
3.3 (Update FST_B)	{ a, ε }	{ a }	{ a, ε }
4.1 (Update FST_S)	{ a, ε }	{ a }	{ a, ε }
4.2 (Update FST_A)	{ a, ε }	{ a }	{ a, ε }
4.3 (Update FST_B)	{ a, ε }	{ a }	{ a, ε }

Iterative Algorithm for Computing FIRST

1. Does the initial approximation matters?
2. Does the algorithm converges?
3. How many iterations does it take to converge in the worst case?
4. Does the number of iterations depend on the order in which we compute FIRST(X) in a particular iteration?

FIRST Function

Notation: FST_X indicated FIRST(X).

$S \rightarrow A \mid B$

$A \rightarrow Ba$

$B \rightarrow Ab \mid \varepsilon$

Compute

1. FIRST(S)
2. FIRST(A)
3. FIRST(B)

Iteration	FST_S	FST_A	FST_B
Initial Approximation	{}	{}	{}
1.1 (Update FST_B)	{}	{}	{ε}
1.2 (Update FST_A)	{}	{a}	{ε}
1.3 (Update FST_S)	{a, ε}	{a}	{ ε }
2.1 (Update FST_B)	{a, ε}	{a}	{a, ε}
2.2 (Update FST_A)	{ a, ε }	{ a }	{a, ε}
2.3 (Update FST_S)	{a, ε }	{ a }	{a, ε }
3.1 (Update FST_B)	{ a, ε }	{ a }	{ a, ε }
3.2 (Update FST_A)	{ a, ε }	{ a }	{ a, ε }
3.3 (Update FST_S)	{ a, ε }	{ a }	{ a, ε }

Observation:
Algorithm stabilizes in 3 iterations instead of 4 iterations.

Fixed-Point Algorithms

- Solve the equation $x^2 - c = 0$ (or find the square root of c)
- Newton's approach: Find the fixed-point of the function

$$f(x) = (x + c/x)/2$$

- y is a fixed-point for a function f(x) if f(y) = y
- The fixed-point for the function $f(x) = (x+c/x)/2$ can be found using an iterative approach
- Newton-Raphson Method: To find the root of the equation f(x)=0 find the fixed-point of the function

$$g(x) = x - f(x)/f'(x)$$

Fixed-Point Algorithms and FIRST Function

- Given CFG $G = (N, T, P, S)$ where $N = \{ X_1, \dots , X_n \}$ we have n unknowns $FIRST(X_i)$ $1 \leq i \leq n$ and ….
- Corresponding to each non-terminal X_i and a production $X_i \rightarrow Y_1 \dots Y_k$ we have system of constraints (unknowns: $FIRST(X_1), \dots , FIRST(X_n)$)

$$\begin{aligned}
FIRST(Y_1) - \{\varepsilon\} &\subseteq FIRST(X) \\
\varepsilon \in FIRST(Y_1) \Longrightarrow FIRST(Y_2) - \{\varepsilon\} &\subseteq FIRST(X) \\
&\dots\dots\dots\dots \\
\varepsilon \in FIRST(Y_l) \text{ for } 1 \leq l \leq i\text{-}1 \Longrightarrow FIRST(Y_i) - \{\varepsilon\} &\subseteq FIRST(X) \\
&\dots\dots\dots\dots \\
\varepsilon \in FIRST(Y_l) \text{ for } 1 \leq l \leq k \Longrightarrow \{\varepsilon\} &\subseteq FIRST(X)
\end{aligned}$$

- Question: Does the System of Constraints precisely characterize the FIRST sets?

Fixed-Point Algorithms and FIRST Function

- This System of Constrains can have multiple solutions. Think of the solution

  FIRST(S) = { a, b, ε }, FIRST(A) = { a, b }, FIRST(B) = { a, b, ε }

- We want the Least or Smallest such solution. What does it mean for a solution to be Smallest?

Fixed-Point Algorithms and FIRST Function

- Define $U = \{ (S_1 \dots S_n) \mid \text{where } S_i \in 2^T \text{ for all } i \text{ and } n = |N| \}$
- Definition: Given Avec = $(A_1 \dots A_n)$ and Bvec = $(B_1 \dots B_n)$ we say

$$\text{Avec} \leq \text{Bvec iff } A_i \subseteq B_i \text{ for all } i.$$

- Among all the solutions to the System of Constraints we want the Least Solution.
- Define FIRSTV= $(\text{FIRST}(X_1) \dots (\text{FIRST}(X_n))$
- Question: Is it possible that there exists two minimal solution FIRSTV_1 and FIRSTV_2 such that

$$\text{FIRSTV}_1 \nleq \text{FIRSTV}_2 \text{ and } \text{FIRSTV}_2 \nleq \text{FIRSTV}_1$$

Side Note: Does the Diophantine equation $2x+3y = 5$ have multiple solutions? What is the least positive solutions? Are there infinitely many solutions?

Algorithm for Computing FIRST

- Define a function f(Xvec) procedurally as follows.

g(Xvec) { // Returns a vector from the set U.

for i = 1 to n

$Xvec_i = g_i(Xvec)$ // update FIRST(X_i) set

}

Algorithm for Computing FIRST

g_i (Xvec) {

for each product $X_i \rightarrow X_{j1} \ldots X_{jk}$ do

inclEPS = true;

$Xvec_i = Xvec_i \cup Xvec_{j1} - \{ \varepsilon \}$

for l = 2 to k do

if ($\varepsilon \notin Xvec_{j(l-1)}$) then { inclEps = false; break; }

$Xvec_i = Xvec_i \cup Xvec_{jl} - \{ \varepsilon \}$

end do

if (inclEPS = true and $\varepsilon \in Xvec_{jk}$) then { $Xvec_i = Xvec_i \cup \{ \varepsilon \}$ }

end do

}

Algorithm for Computing FIRST

- The solution to the system of constraints is given by the Least fixed-point to the function g() we defined.
- To reach the least fixed-point apply g() repeatedly until the solution doesn't move-up with ({} …. {}) as the initial approximation.
- Question 1: How many iterations does it take for the algorithm to converge to the fixed point in the worst-case?
- Question 2: Does the convergence time depends upon the order of the non-terminals X_1 , … X_n?

FOLLOW Function

Given a CFG $G = (N, T, P, S)$

- For a non-terminal $A \in N$, FOLLOW(A) consists of set of terminals that can appear immediately to the right of A in some sentential form.
- In other words the set of terminals a such that there exists a derivation of the form $S \Rightarrow^* \alpha A a \beta$ for some $\alpha, \beta \in (V \cup T)^*$.
- If A can be the rightmost symbol in some sentential form (i.e., there exists a derivation $S \Rightarrow^* \alpha A$), then \$ is on FOLLOW(A)

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

FOLLOW Function

- Compute FOLLOW(S)

$$S \rightarrow 0S1 \mid 1S0 \mid \varepsilon$$

FOLLOW(S) = { 0, 1, $ }

- Compute FOLLOW(S) and FOLLOW(A)

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sd \mid cA \mid \varepsilon$$

FOLLOW(S) = { d, $ }

FOLLOW(A) = { a }

Computing FOLLOW

Initialization Step: FOLLOW(S) = { \$ } and FOLLOW(A) = { } for all non-terminals A other than S.

Repeat the following steps until the FOLLOW sets stabilize

1. If there is a production $A \rightarrow \alpha B \beta$ then
   1. Everything in FIRST(β) except ε is in FOLLOW(B)
   2. If FIRST(β) contains ε then everything in FOLLOW(A) is in FOLLOW(B).
2. If there is a production $A \rightarrow \alpha B$ then
   1. Everything in FOLLOW(A) is in FOLLOW(B).

Computing FOLLOW

Compute FOLLOW for the non-terminals S, S’, A in

$S \rightarrow aAS'$

$S' \rightarrow \varepsilon \mid bS'$

$A \rightarrow aS$

Computing Follow

Notation: **FLW_X indicates the FOLLOW(X)**

Iteration	FLW_S	$FLW_{S'}$	FLW_A
Initial Approx.	{$}	{}	{}
1.1 (update FLW_S)	{$}	{}	{}
1.2 (update $FLW_{S'}$)	{$}	{$}	{}
1.3 (update FLW_A)	{$}	{$}	{b, $}
2.1 (update FLW_S)	{b, $}	{$}	{b,$}
2.2 (update $FLW_{S'}$)	{b,$}	{b,$}	{b,$}
2.3 (update FLW_A)	{b,$}	{b, $}	{b, $}
3.1 (update FLW_S)	{b, $}	{b, $}	{b,$}
3.2 (update $FLW_{S'}$)	{b,$}	{b,$}	{b,$}
3.3 (update FLW_A)	{b,$}	{b,$}	{b,$}

$S \to aAS'$
$S' \to bS' \mid \varepsilon$
$A \to aS$

FIRST(S) = { a }
FIRST(S') = { ε, b }
FIRST(A) = { a }

FOLLOW Function

$E \rightarrow T\,E'$

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

$T \rightarrow F\,T'$

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

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

Compute

1. FOLLOW(E)
2. FOLLOW(E')
3. FOLLOW(T)
4. FOLLOW(T')
5. FOLLOW(F)