Bottom-Up Parsing

- "Shift-Reduce" Parsing
- Reduce a string to the start symbol of the grammar.
- At every step a particular substring is matched (in left-to-right fashion) to the right side of some production and then it is substituted by the non-terminal in the left hand side of the production.

Consider:

1 $\mathbf{S} \rightarrow \mathbf{aABe}$

2-3 $\mathbf{A} \rightarrow \mathbf{Abc} \mid \mathbf{b}$

4 $\mathbf{B} \rightarrow \mathbf{d}$

$$
\begin{array}{l}
abbcde \\
\downarrow 3 \\
aAbcde \\
\downarrow 2 \\
aAde \\
\downarrow 4 \\
aABe \\
\downarrow 1 \\
S
\end{array}
$$

Rightmost Derivation:

$$S \underset{rm}{\overset{1}{\Rightarrow}} \mathbf{aABe} \underset{rm}{\overset{4}{\Rightarrow}} \mathbf{aAde} \underset{rm}{\overset{2}{\Rightarrow}} \mathbf{aAbcde} \underset{rm}{\overset{3}{\Rightarrow}} \mathbf{abbcde}$$

Handles

- Handle of a string = substring that matches the RHS of some production AND whose reduction to the non-terminal on the LHS is a step along the reverse of some rightmost derivation.
- Formally:
  - A phrase is a substring of a sentential form derived from exactly one Non-terminal
  - A simple phrase is a phrase created in one step
  - handle is a simple phrase of a right sentential form
- i.e. $A \to \beta$ is a handle of $\alpha\beta x$, where x is a string of terminals, if:

$$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A x \underset{rm}{\Rightarrow} \alpha\beta x$$

- A certain sentential form may have many different handles.
- Right sentential forms of a non-ambiguous grammar have one *unique* handle [but many substrings that look like handles potentially !].

Example

Consider:

$$S \to aABe$$
$$A \to Abc \mid b$$
$$B \to d$$

$$S \underset{rm}{\Rightarrow} \underline{aABe} \underset{rm}{\Rightarrow} aA\underline{d}e \underset{rm}{\Rightarrow} a\underline{Abc}de \underset{rm}{\Rightarrow} a\underline{b}bcde$$

It follows that:

$(S \to)$ aABe is a handle of $\underline{aABe}$

$(B \to)$ d is a handle of $aA\underline{d}e$

$(A \to)$ Abc is a handle of $a\underline{Abc}de$

$(A \to)$ b is a handle of $a\underline{b}bcde$

Example, II

Grammar:

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Consider aAbcde (it is a right sentential form)

Is [$A \rightarrow b$, $aA\underline{b}cde$] a handle?

if it is then there must be:

$$S \Rightarrow_{rm} \ldots \Rightarrow_{rm} aAAbcde \Rightarrow_{rm} aAbcde$$

no way ever to get two consecutive A's in this grammar. => Impossible

Example, III

Grammar:

$$S \rightarrow aABe$$
$$A \rightarrow Abc \mid b$$
$$B \rightarrow d$$

Consider aAbcde (it is a right sentential form)

Is [$B \rightarrow d$, $aAbc\underline{d}e$] a handle?

if it is then there must be:

$$S \Rightarrow_{rm} \ldots \Rightarrow_{rm} aAbcBe \Rightarrow_{rm} aAbcde$$

we try to obtain **aAbcBe**

$$S \Rightarrow_{rm} aABe \Rightarrow_{??} aAbcBe$$

not a right sentential form

Shift Reduce Parsing with a Stack

- The “big” problem : given the sentential form locate the handle
- General Idea for S-R parsing using a stack:
  1. “shift” input symbols into the stack until a handle is found on top of it.
  2. “reduce” the handle to the corresponding non-terminal.
  3. “accept” when the input is consumed and only the start symbol is on the stack.
  4. “error” call the error handler
- Viable prefix: prefix of a right sentential form that appears on the stack of a Shift-Reduce parser.

What happens with ambiguous grammars

Consider:

$$E \rightarrow E + E \mid E * E \mid$$
$$\mid (E) \mid id$$

Derive id+id*id
By two different Rightmost derivations

Example

STACK	INPUT	Remark
\$	id + id * id\$	Shift
\$ id	+ id * id\$	Reduce by $E \rightarrow id$
\$ E	+ id * id\$	Shift
\$ E +	id * id\$	Shift
\$ E + id	* id\$	Reduce by $E \rightarrow id$
\$ E + E		

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

Both reduce by $E \rightarrow E + E$, and Shift can be performed: Shift/reduce conflict

Conflicts

- Conflicts [appear in ambiguous grammars] either "shift/reduce" or "reduce/reduce"

- Another Example:

$$\begin{aligned} \textit{stmt} \rightarrow\ & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ \mid\ & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \textbf{ else } \textit{stmt} \\ \mid\ & \textbf{other} \ \ \textbf{(any other statement)} \end{aligned}$$

Stack	Input	
if … then	else …	Shift/ Reduce conflict

More Conflicts

stmt → **id (** ***parameter-list*** **)**

stmt → ***expr*** **:=** ***expr***

parameter-list → ***parameter-list*** **,** ***parameter*** **|** ***parameter***

parameter → **id**

expr-list → ***expr-list*** **,** ***expr*** **|** ***expr***

expr → **id | id (** ***expr-list*** **)**

Consider the string A(I,J)
Corresponding token stream is id(id, id)
After three shifts:
Stack = id(id   Input = , id)

Reduce/Reduce Conflict … what to do?
(it really depends on what is A,
an array? or a procedure?

Removing Conflicts

- One way is to manipulate grammar.
  - cf. what we did in the top-down approach to transform a grammar so that it is LL(1).
- Nevertheless:
  - We will see that shift/reduce and reduce/reduce conflicts can be best dealt with after they are discovered.
  - This simplifies the design.

Operator-Precedence Parsing

- problems encountered so far in shift/reduce parsing:
  - IDENTIFY a handle.
  - resolve conflicts (if they occur).
  - operator grammars: a class of grammars where handle identification and conflict resolution is easy.

- Operator Grammars: no production right side is $\in$ or has two adjacent non-terminals.

$$E \rightarrow E - E \mid E + E \mid E * E \mid E / E \mid E \wedge E \mid - E \mid (E) \mid id$$

- note: this is typically ambiguous grammar.

Basic Technique

- For the terminals of the grammar, define the relations <. .> and .=.
- a <. b means that *a* yields precedence to *b*
- a .=. b means that *a* has the same precedence as *b*.
- a .> b means hat *a* takes precedence over *b*
- E.g. * .> + or + <. *

- Many handles are possible. We will use <. .=. And .> to find the correct handle (i.e., the one that respects the precedence).

Using Operator-Precedence Relations

- GOAL: delimit the handle of a right sentential form
- <. will mark the beginning, .> will mark the end and .=. will be in between.
- Since no two adjacent non-terminals appear in the RHS of any production, the general form sentential forms is as:
  $\beta_0\ a_1\ \beta_1\ a_2\ \beta_2\ \ldots\ a_n\ \beta_n$, where each β_i is either a nonterminal or the empty string.
- At each step of the parse, the parser considers the top most terminal of the parse stack (i.e., either top or top-1), say a, and the current token, say b, and looks up their precedence relation, and decides what to do next:

Operator-Precedence Parsing

1. If a .=. b, then shift b into the parse stack
2. If a <. b, then shift <. And then shift b into the parse stack
3. If a .> b, then find the top most <. relation of the parse stack; the string between this relation (with the non-terminal underneath, if there exists) and the top of the parse stack is the handle (the handle should match (weakly) with the RHS of at least one grammar rule); replace the handle with a typical non-terminal

Example

STACK	INPUT	Remark
$	id + id * id$	$ <. id
$ <. id	+ id * id$	id >. +
$ E	+ id * id$	$ <. +
$ E <. +	id * id$	+ <. id
$ E <. + <. id	* id$	id .> *
$ E <. + E	* id$	+ <. *
$ E <. + E <. *	id$	* <. id
$ E <. + E <. * <. id	$	id .> $
$ E <. + E <. * E	$	* .> $
$ E <. + E	$	+ .> $
$ E	$	accept
	$	

	+	*	(	)	id	$
+	.>	<.	<.	.>	<.	.>
*	.>	.>	<.	.>	<.	.>
(	<.	<.	<.	.=.	<.	
)	.>	.>		.>		.>
id	.>	.>		.>		.>
$	<.	<.	<.		<.	.=.

Parse Table

1-2 E → E + T | T

3-4 T → T * F | F

5-6 T → (E) | id

Producing the parse table

- $FirstTerm(A) = \{a \mid A \Rightarrow^{+} a\alpha \text{ or } A \Rightarrow^{+} Ba\alpha\}$
- $LastTerm(A) = \{a \mid A \Rightarrow^{+} \alpha a \text{ or } A \Rightarrow^{+} \alpha aB\}$
- a .=. b iff $\exists\, U \rightarrow \alpha ab\beta$ or $\exists\, U \rightarrow \alpha aBb\beta$
- a <. b iff $\exists\, U \rightarrow \alpha aB\beta$ and $b \in$ FirsTerm(B)
- a .> b iff $\exists\, U \rightarrow \alpha Bb\beta$ and $a \in$ LastTerm(B)

Example:

- FirstTerm (E) = {+, *, id, (}
- FirstTerm (T) = {*, id, (}
- FirstTerm (F) = {id, (}

- LastTerm (E) = {+, *, id,)}
- LastTerm (T) = {*, id,)}
- LastTerm (F) = {id,)}

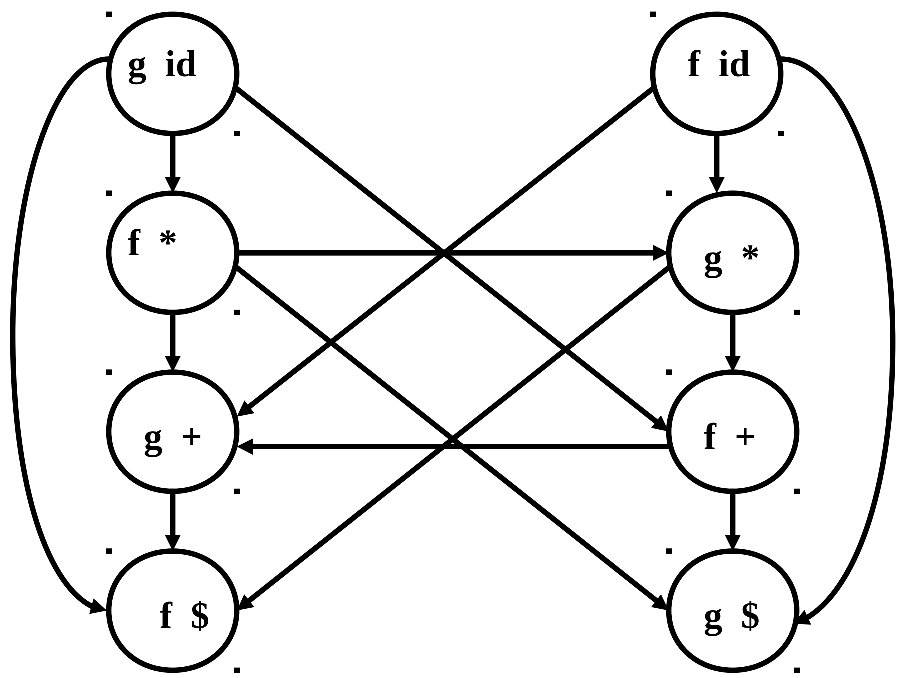
1-2 $E \rightarrow E + T \mid T$
3-4 $T \rightarrow T * F \mid F$
5-6 $T \rightarrow (E) \mid id$

Precedence Functions vs Relations

	+	-	*	/	↑	(	)	id	$
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

- f(a) < g(b) whenever a <. b
- f(a) = g(b) whenever a .=. b
- f(a) > g(b) whenever a .> b

Constructing precedence functions



	+	*	id	$
f	2	4	4	0
g	1	3	5	0

Handling Errors During Reductions

- Suppose *abEc* is poped and there is no production right hand side that matches *abEc*
- If there were a rhs *aEc*, we might issue message
  illegal b on line x
- If the rhs is *abEdc*, we might issue message
  missing d on line x
- If the found rhs is *abc*, the error message could be
  illegal E on line x,

  where E stands for an appropriate syntactic category represented by non-terminal E

Handling shift/reduce errors

	id	(	)	$
id	e3	e3	.>	.>
(	<..	<.	.=.	e4
)	e3	e3	.>	.>
$	<.	<.	e2	e1

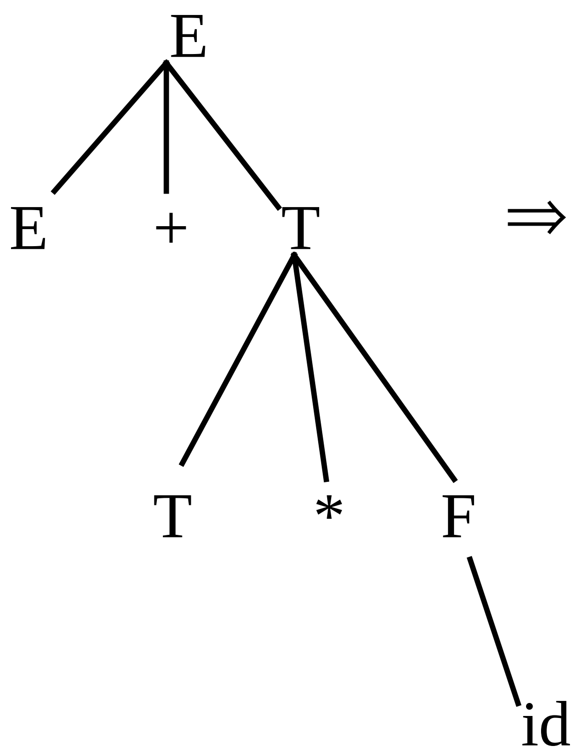
e1: /* called when whole expression
    is missing */
    insert id onto the input
    print “missing operand

e2: /* called when expression begins
    with a right parenthesis */
    delete) from the input
    print “unbalanced right parenthesis”

e3”: /* called when id or) is followed by id or (*/
    insert + onto the input
    print “missing operator

e4: /* called when expression ends with a left parenthesis */
    pop (from the stack
    print “missing right parenthesis”

Extracting Precedence relations from parse tables

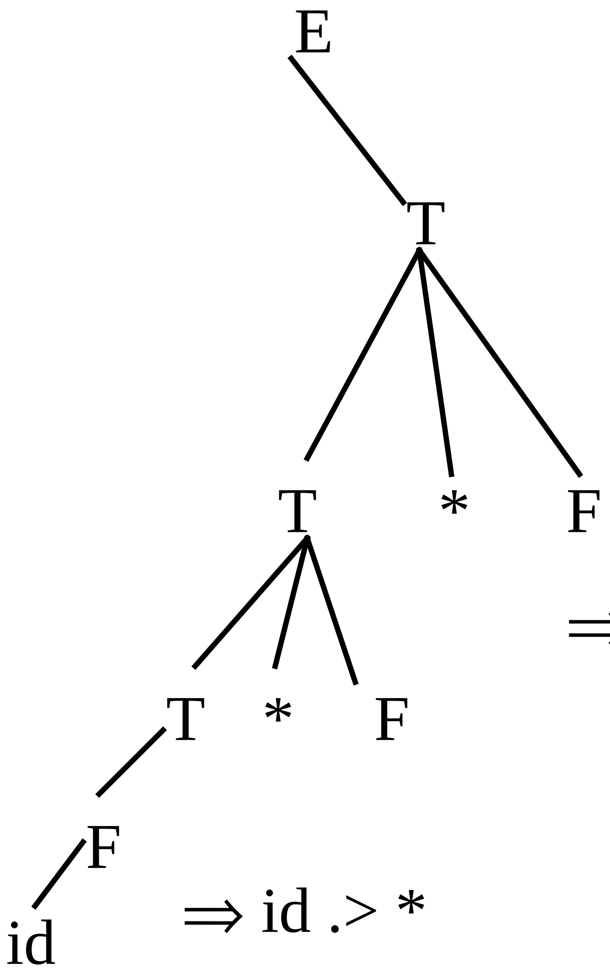


$\Rightarrow + <\!\cdot\, *$

$\Rightarrow * <\!\cdot\, id$

1-2 E → E + T | T
3-4 T → T * F | F
5-6 T → (E) | id

Extracting Precedence relations from parse tables



$\Rightarrow * \cdot\!> *$

$\Rightarrow id \cdot\!> *$

1-2 $\mathbf{E \rightarrow E + T \mid T}$

3-4 $\mathbf{T \rightarrow T * F \mid F}$

5-6 $\mathbf{T \rightarrow (E) \mid id}$

Pros and Cons

- + simple implementation
- + small parse table
- - weak (too restrictive for not allowing two adjacent non-terminals
- - not very accurate (some syntax errors are not detected due weak treatment of non-terminals)

- Simple precedence parsing is an improved form of operator precedence that doesn’t have these weaknesses