Parsing - 1

- What is parsing?
- Shift-reduce parsing
  - Shift-reduce conflict
  - Reduce-reduce conflict
- Operator precedence parsing

Parsing

- **Parsing is the reverse of doing a derivation**
- **By looking at the terminal string, effectively try to build the parse tree from the bottom up**
- **Finding which sequences of terminals and nonterminals form the right hand side of production and *reducing* them to the left hand side nonterminal**

Shift-reduce Parsing

- ***Handle-*** **substring which is right hand side of some production; corresponds to the last expansion in a *rightmost derivation***
- **Replacement of handle by its corresponding nonterminal left hand side, results in reduction to the distinguished nonterminal by a *reverse rightmost derivation***
- **Parse works by shifting symbols onto the stack until have *handle* on top; then reduce; then continue**

Example

S	→	E	(1)
E	→	E + T	(2)
E	→	T	(3)
T	→	*id*	(4)

Rightmost derivation of a+b+c, handles in red

S → **E**

→ E + T

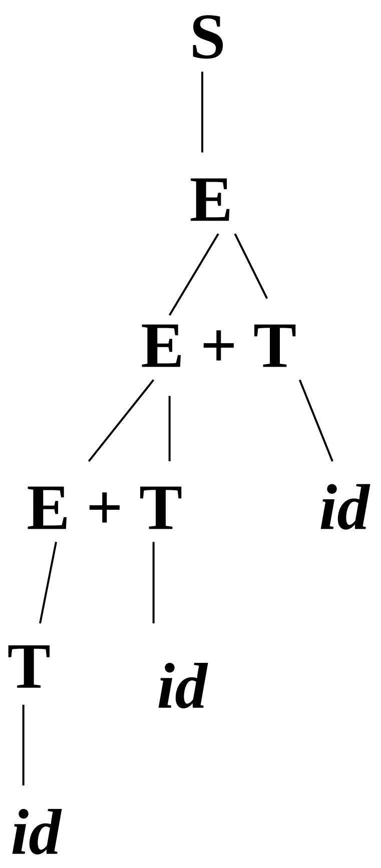
→ E + ***id***

→ E + T + ***id***

→ E + ***id*** + ***id***

→ **T** + ***id*** + ***id***

→ ***id*** + ***id*** + ***id***



Example

$S \rightarrow E$		(1)
$E \rightarrow E + T$		(2)
$E \rightarrow T$		(3)
$T \rightarrow$ *id*		(4)

Actions: shift, reduce, accept, error

Stack	Input	Action
$	id1 + id2 + id3 $	shift
$ id1	+ id2 + id3 $	reduce (4)
$ T	+ id2 + id3 $	reduce (3)
$ E	+ id2 + id3 $	shift
$ E +	id2 + id3 $	shift
$ E + id2	+ id3 $	reduce(4)
$ E + T	+ id3 $	reduce (2)
$ E	+ id3 $	shift
$ E +	id3 $	shift
$ E + id3	$	reduce (4)
$ E + T	$	reduce(2)
$ E	$	reduce (1)
$ S	$	accept

Possible Problems

- **Can get into conflicts where one rule implies *shift* while another implies *reduce***

  S → if E then S | if E then S else S

  On stack: if E then S

  Input: else

  Should *shift* trying for 2nd rule or *reduce* by first rule?

Possible Problems

- Can have two grammar rules with same right hand side which leads to *reduce-reduce* conflicts

  $A \rightarrow \alpha$ and $B \rightarrow \alpha$ both in grammar

  When α on top of the stack, how know which production choose? That is, whether to *reduce* to A or B?
- In both kinds of conflicts, problem is with the grammar, not necessarily the language
- Recall, there can be many context-free grammars corresponding to the same language!

Shift-Reduce Parsing

- Actions
  - *Shift* - push token onto stack
  - *Reduce* - remove handle from stack and push on corresponding nonterminal
  - *Accept* - recognize sentence when stack contains only the distinguished symbol and input is empty
  - *Error* - happens when none of the above is possible; means original input was not a sentence!

Handles

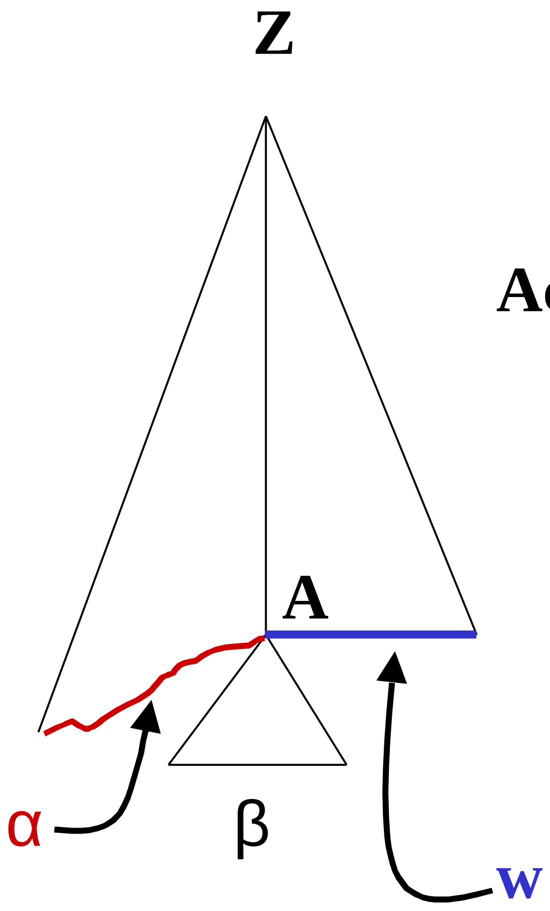
- **Any string of terminals and nonterminals derived from the distinguished nonterminal is called a *sentential form***
- **If grammar is unambiguous, then each right sentential form has a unique handle**

$$Z \overset{*}{\underset{rm}{\rightarrow}} \alpha\, A\, w \underset{rm}{\rightarrow} \alpha\, \beta\, w,$$

where α is a mixture of terminals and nonterminals; β is the handle;

and w is a string of terminals

A Handle in the Parse Tree



Action: reduce β to A

Ambiguity Example

Z → **E**

E → E *or* E | *a*

Two rightmost derivations (handles in red):

Z → E → E *or* E → E *or a* → E *or* E *or a* → E *or a or a* →

a or a or a

Z → E → E *or* E → E *or* E *or* E → E *or* E *or a* →

E *or a or a* → *a or a or a*

Shift *a*, reduce to E, shift *or*, shift *a*, reduce to E (now have E *or* E on stack). In deriv1, reduce E *or* E to E. In deriv2 shift *or* and *a* onto stack. SHIFT-REDUCE conflict.

Justification of Handle Use

- **How can we be sure that the handle will always be at the top of the stack?**
  - **Conventions: Greek letters for strings of terminals and nonterminals. Arabic letters for strings of terminals only. Capital letters are nonterminals.**
- **The following is a rightmost derivation:**

***Case 1*: A's production contains a rightmost nonterminal B.**

$$Z \overset{*}{\underset{rm}{\rightarrow}} \alpha\, A\, q \underset{rm}{\rightarrow} \alpha\, \beta\, B\, y\, q \underset{rm}{\rightarrow} \alpha\, \beta\, \gamma y\, q, \text{ where } B \rightarrow \gamma$$

Justification, cont.

Stack will contain $\$\alpha\beta\gamma$ with yq in the input.

This will be reduced to $\$\alpha\beta B$ with yq still in the input.

Handle can't be below B in the stack or else the derivation would have to have been:

…X…B $\rightarrow$ …δ…B with δ in the αβ on the stack. But this isn't a rightmost derivation, because B is to the right of X and X is being expanded first! #CONTRADICTION

Justification, cont.

Therefore handle must contain B and it is not "buried" in the stack.

Assume the handle is βBy (β or y may be empty)

Case 2: A's production does not contain a nonterminal

$Z \overset{*}{\underset{rm}{\rightarrow}} \alpha\ C\ x\ A\ r \underset{rm}{\rightarrow} \alpha\ C\ x\ y\ r \underset{rm}{\rightarrow} \alpha\ \gamma\ x\ y\ r$
where $A \rightarrow y$ and $C \rightarrow \gamma$

Justification, cont.

- **Stack will contain $\$\alpha\gamma$ with input xyr. This will be reduced to $\$\alpha C$, and then x and y will be shifted onto stack. Then $\$\alpha Cxy$ will be reduced to $\$\alpha CxA$ on the stack with r remaining in the input.**
- **So the handle is not buried in the stack.**

Operator Precedence Parsing

ASU, Ch 4.6

- A simplified bottom up parsing technique used for expression grammars
- Requires
  - No right hand side of rule is empty
  - No right hand side has 2 adjacent nonterminals
- Drawbacks
  - Small class of grammars qualify
  - Overloaded operators are hard (unary minus)
  - Parser correctness hard to prove

Operator Precedence

- **Define three precedence relations**
  - **a < b, a yields in precedence to b**
  - **a > b, a takes precedence over b**
  - **a = b, a has same precedence as b**
- **Find handle as <====> pattern at top of stack;**
- **Check relation between top of stack and next input symbol**
- **Basically, ignore nonterminals**

Example

Z → E

E → E   E | E + E | ***id***

Define precedence relations between + and *.

+ < *, * > +, + > +, * > * (last 2 ensure left associativity)

Form table of precedences.

Now parse using the table, and keep track of the operand nonterminals, too.

Sometimes can embed error handling in matrix.

	id	+	*	$
id		>	>	>
+	<	>	<	>
*****	<	>	>	>
$	<	<	<	

Example

Compare top of stack token to next input token.

Stack	**Compares**	**Input**
$	<	id1 + id2 * id3 $
$ < id1	>	+ id2 * id3 $
$ E	<	+ id2 * id3 $
$ E +	<	id2 * id3 $
$ E + < id2	>	* id3 $
$ E + E	<	* id3 $
$ E + E *	<	id3 $
$ E + E * < id3	>	$
$ E + < E * E	>	$
$ < E + E	>	$
$ < E	>	$
accept		

Making OP parsing practical

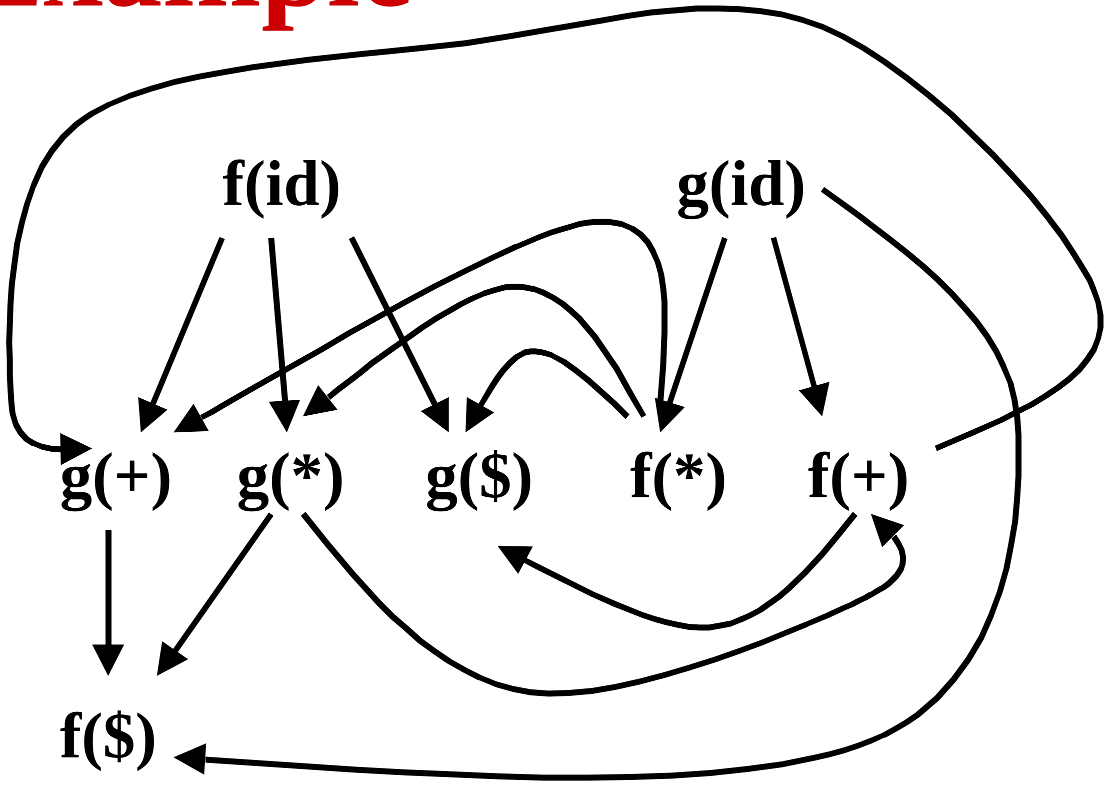
- How to store these precedences compactly?
- *Precedence functions*
  - Find functions f(), g() such that
    - f(token1) > g(token2) means token1 > token2
    - f(token1)=g(token2) means token1 = token2
    - f(token1) < g(token2) means token1 < token2
  - Graph partitioning algorithm to find f(),g() if possible.

Precedence Functions

- Form graph from table of precedences
  - Nodes formed by f(token1),f(token2),…,g(token1) etc.
    - Form equivalence classes of nodes based on the = relation (equal precedence, e.g., * /)
  - Edges show required relations between function values
    - If token1 > token2, then f(token1)-->g(token2)
    - If token1 < token2, then f(token1)<--g(token2)
  - If the graph is *acyclic,* then can find integer value assignments for the range values of f,g.
    - Let value of f(token1) be the length of the longest path from the node representing f(token1)

Example

	id	+	*	$
id		>	>	>
+	<	>	<	>
*	<	>	>	>
$	<	<	<	



Acyclic graph yields

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