Compilers

module of the course “Professional English”

Yulia Burkatovskaya

Department of Computer Engineering

Associate professor

5. Bottom-up parsing

- Reduction
- Handle pruning
- Shift-reduce parsing
- Conflicts during shift-reduce parsing
- Introduction to LR parsing: simple LR
- Items and the LR(0) automaton
- Use of the LR(0) automaton
- Viable prefix

5. Bottom-up parsing

Bottom-up parsing is the construction of a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

E → T

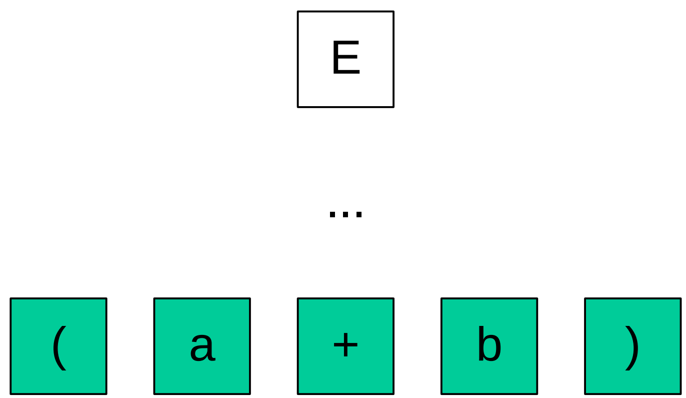
T → E+T

T → E*T

T → ID

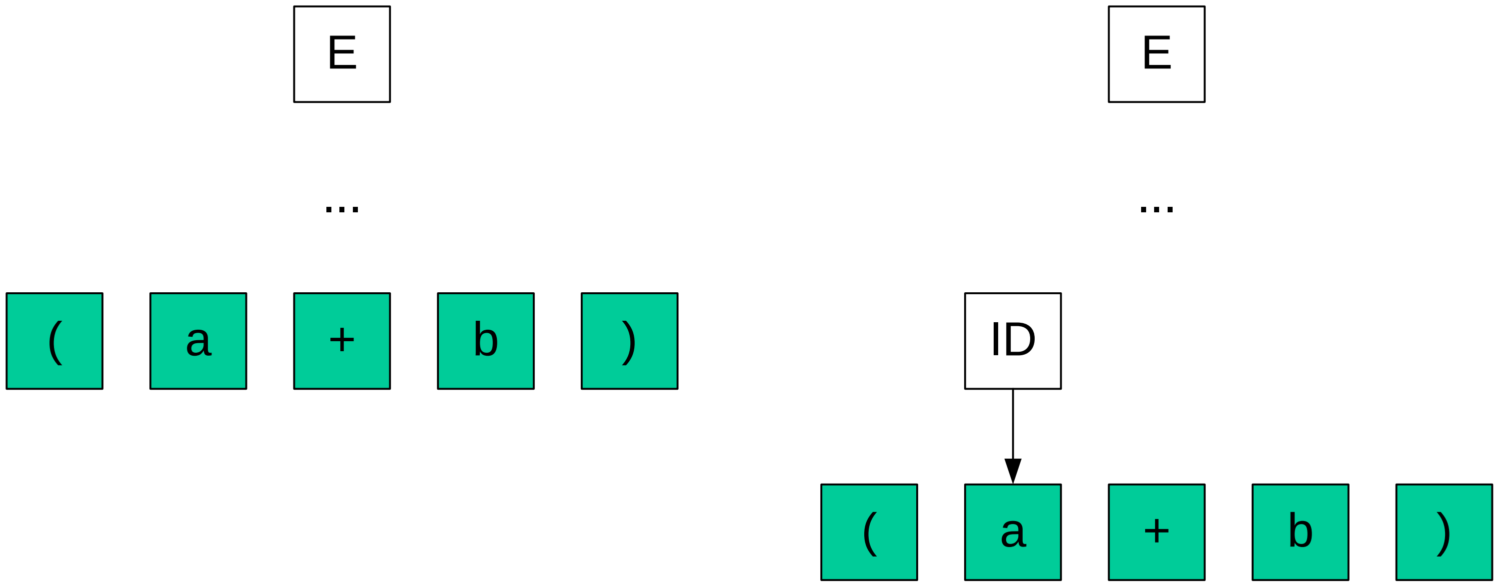
T → (E)

ID → a | … | z

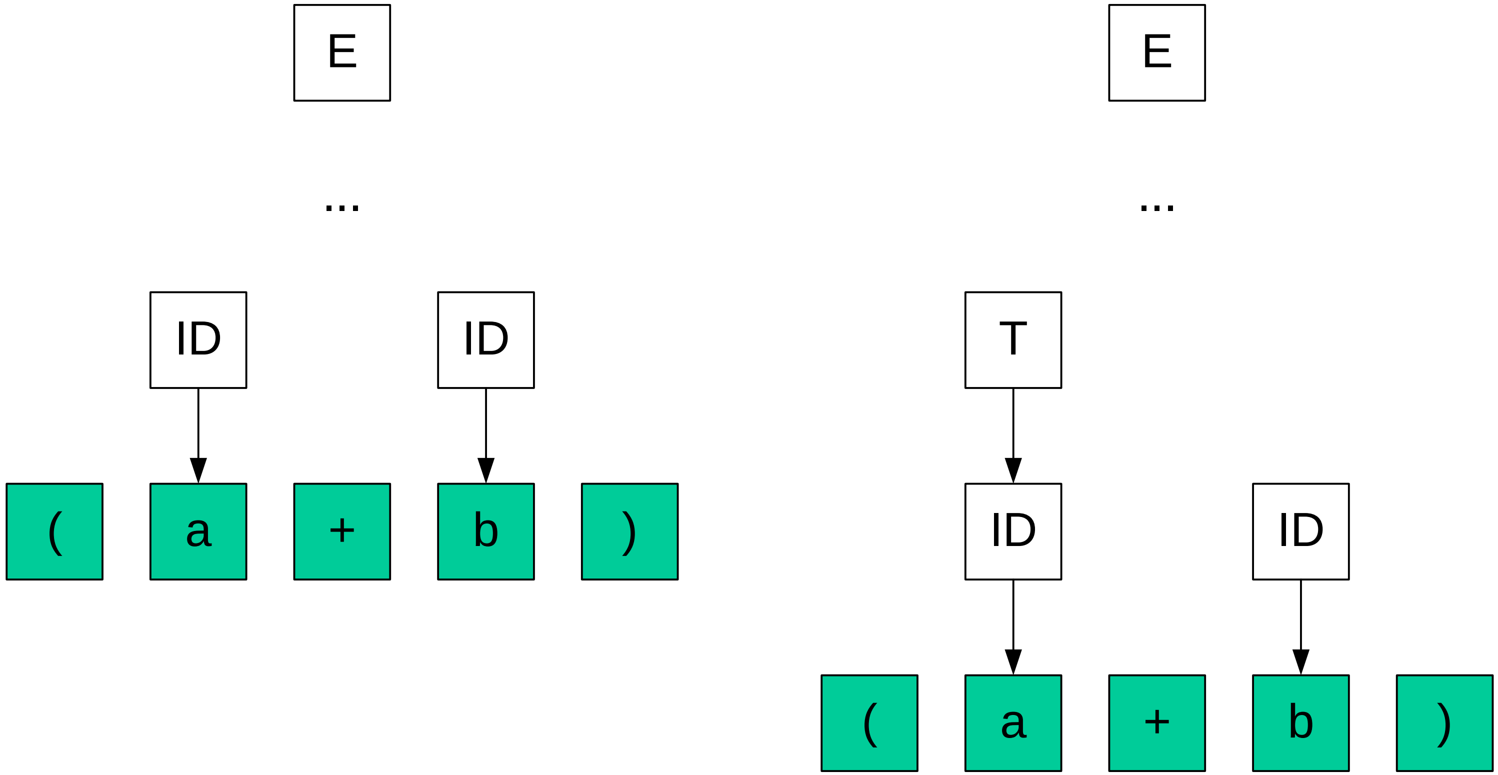


(a+b) → (ID+b) → (T+b) → (E+b) → (E+ID) → (E+T) → (T) → (E) → T → E

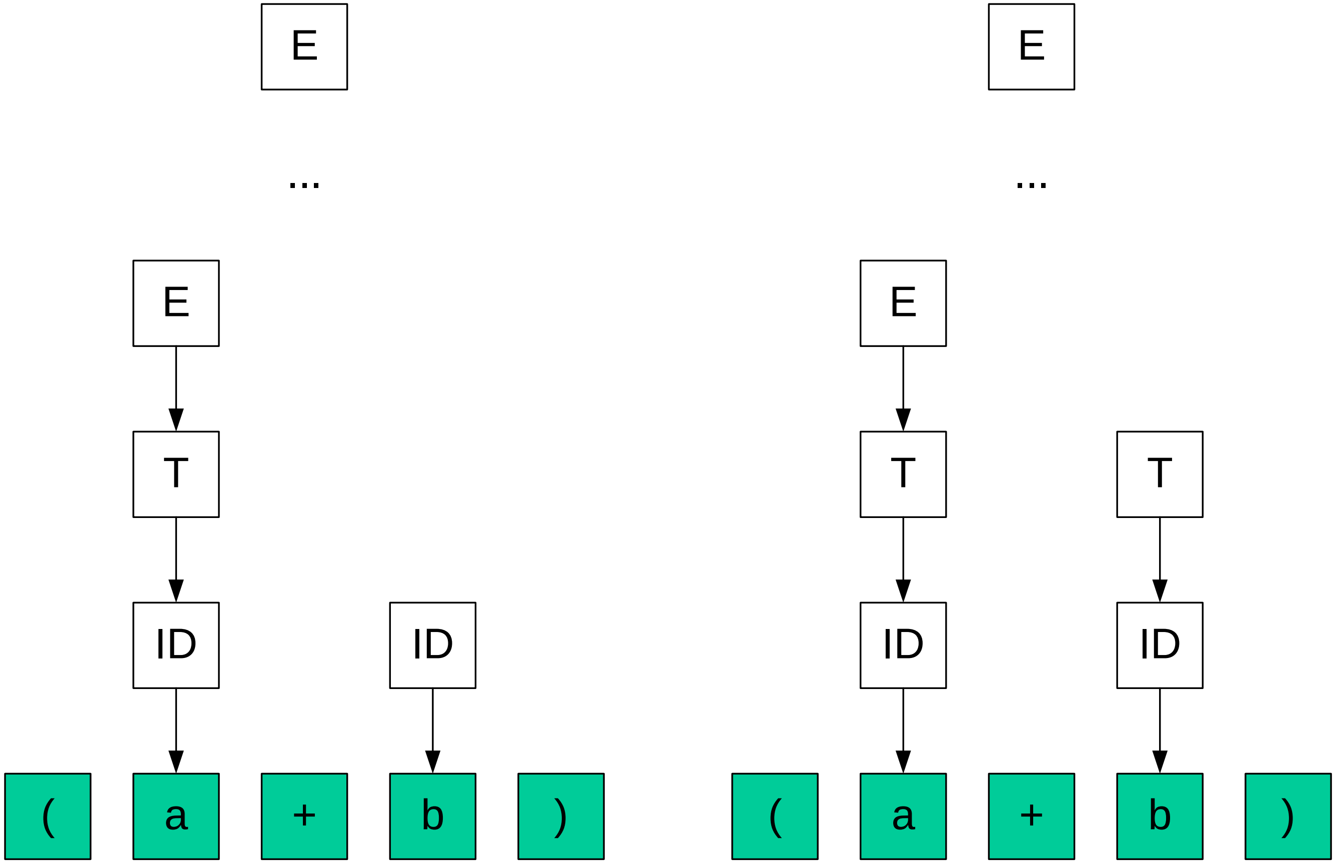
Bottom-up parsing



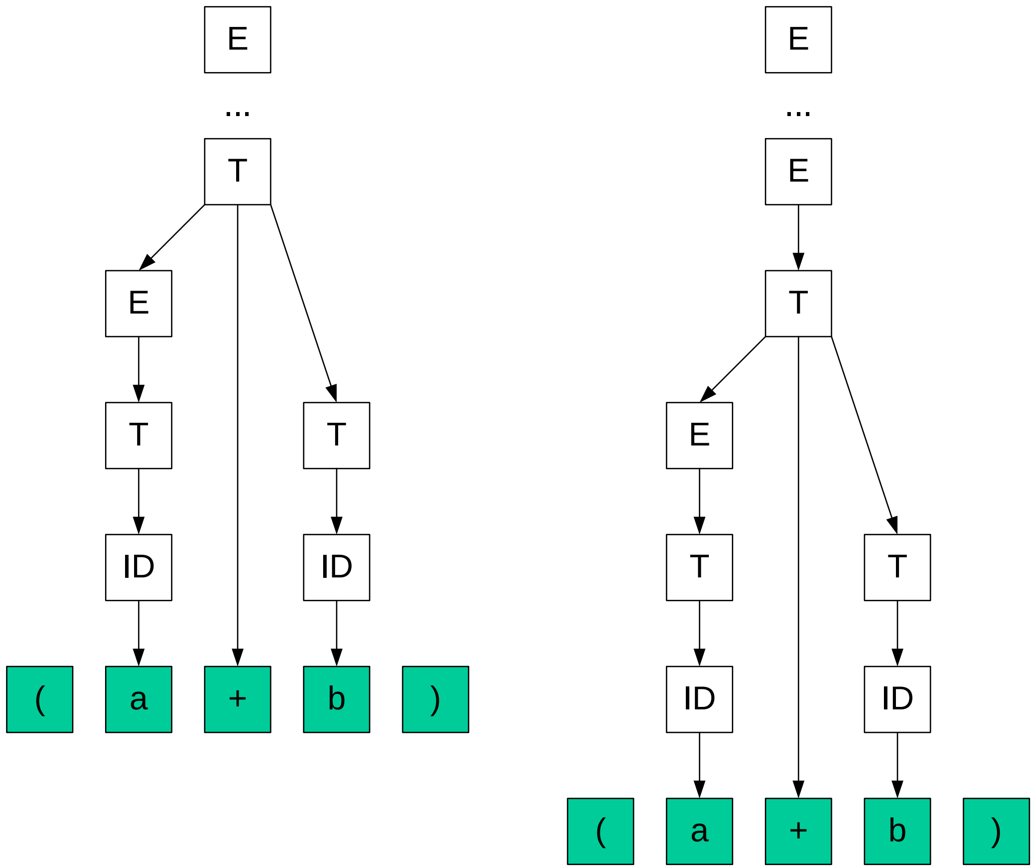
Bottom-up parsing



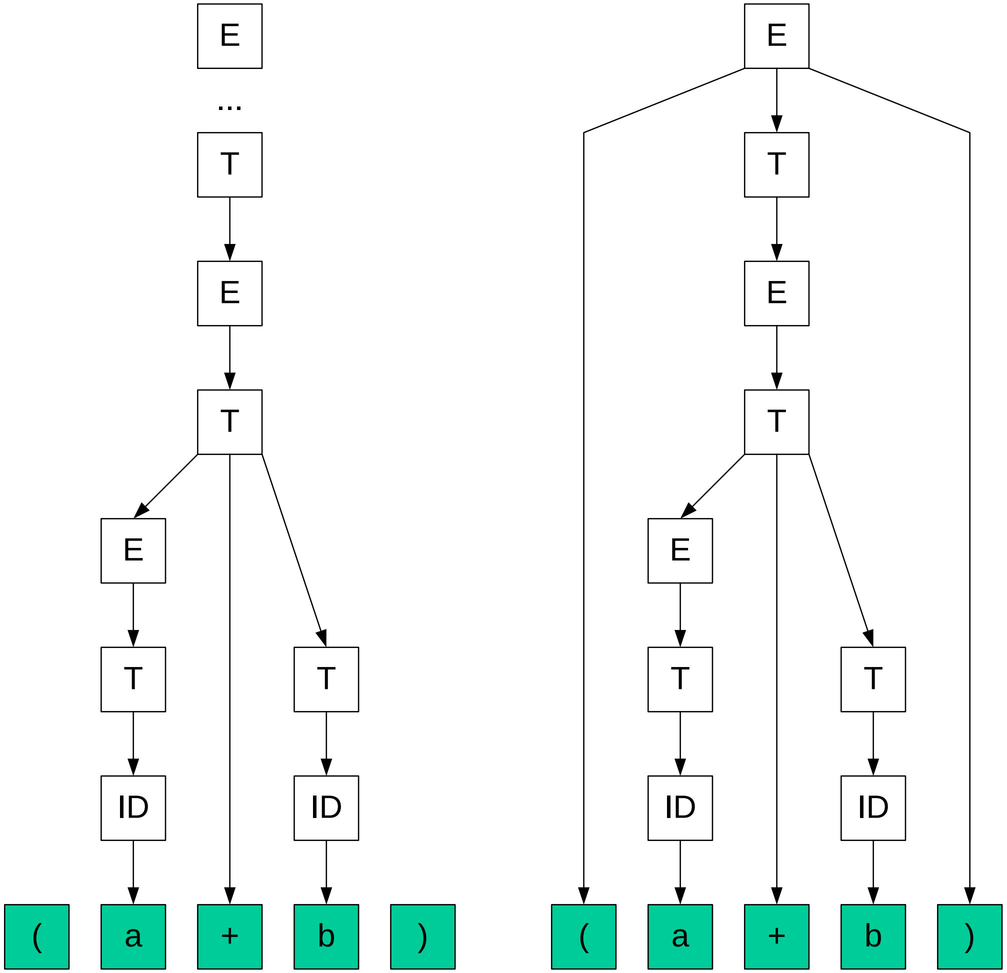
Bottom-up parsing



Bottom-up parsing



Bottom-up parsing



5.1. Reduction

- We can think of bottom-up parsing as the process of **"reducing"** a string *w* to the start symbol of the grammar.
- At each **reduction** step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.
- The key decisions during bottom-up parsing are about **when** to reduce and about **what** production to apply, as the parse proceeds.

Reduction

E → T

T → E+T

T → E*T

T → ID

T → (E)

ID → a | … | z

(a+b)*c → (ID+b)*c → (T+b)*c → (E+b)*c → (E+ID)*c → (E+T)*c
→ (T)*c → (E)*c → T*c → E*c → E*ID → E*T → T → E

Here the leftmost substring is replaced.

5.2. Handle pruning

- Bottom-up parsing during a left-to-right scan of the input constructs a rightmost derivation in reverse.
- Informally, a **"handle"** is a substring that matches the body of a production, and whose reduction represents one step along the reverse of a rightmost derivation.

Right sequential form	Handle	Reducing production
a+b*c	a	ID → a
ID+b*c	ID	T → ID
E+T*c	E+T	T → E+T
T*c	E	E → T

5.3. Shift-Reduce Parsing

- A stack holds grammar symbols and an input buffer holds the rest of the string to be parsed.
- The handle always appears at the top of the stack just before it is identified as the handle.
- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production.
- The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty.

Shift-Reduce Parsing

E → T
T → E+T
T → E*T
T → ID
T → (E)

STACK	INPUT	ACTION
$	ID+ID*ID$	shift
ID$	+ID*ID$	reduce by T →ID
T$	+ID*ID$	reduce by E →T
E$	+ID*ID$	shift
+E$	ID*ID$	shift
ID+E$	*ID$	reduce by T →ID
T+E$	*ID$	reduce by T →E+T

Shift-Reduce Parsing

$E \to T$

$T \to E+T$

$T \to E*T$

$T \to ID$

$T \to (E)$

STACK	INPUT	ACTION
T+E$	*ID$	reduce by $T \to E+T$
T$	*ID$	reduce by $E \to T$
E$	*ID$	shift
*E$	ID$	shift
ID*E$	$	reduce by $T \to ID$
T*E$	$	reduce by $T \to E*T$
T$	$	reduce by $E \to T$
E$	$	accept

Shift-Reduce Parsing

While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make:

- **Shift.** Shift the next input symbol onto the top of the stack.
- **Reduce.** The right end of the string to be reduced must be at the top the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
- **Accept.** Announce successful completion of parsing.
- **Error.** Discover a syntax error and call an error recovery routine.

Shift-Reduce Parsing

The handle will always eventually appear on top of the stack, never inside.

- $A \rightarrow \alpha A z \rightarrow \alpha\beta B y z \rightarrow \alpha\beta\gamma y z$

STACK	INPUT	ACTION
$\$\alpha\beta\gamma$	$yz\$$	reduce by $B \rightarrow \gamma$
$\$\alpha\beta B$	$yz\$$	shift (B is the rightmost non-terminal!)
$\$\alpha\beta B y$	$z\$$	reduce by $A \rightarrow \beta B y$
$\$\alpha A$	$z\$$	shift
$\$\alpha A z$	$\$$	reduce by $A \rightarrow \alpha A z$
$\$A$	$\$$	accept

Shift-Reduce Parsing

- $A \rightarrow \alpha\beta xAz \rightarrow \alpha\beta xyz \rightarrow \alpha\beta\gamma yz$

STACK	INPUT	ACTION
$\$\alpha\gamma$	xyz$	reduce by $B \rightarrow \gamma$
$\$\alpha B$	xyz$	shift (B is the rightmost non-terminal!)
$\$\alpha Bx$	yz$	shift
$\$\alpha Bxy$	z$	reduce by $A \rightarrow y$
$\$\alpha BxA$	z$	shift
$\$\alpha BxAz$	$	reduce by $A \rightarrow \alpha BxAz$
$A	$	accept

5.4. Conflicts During Shift-Reduce Parsing

- Whether to shift or to reduce (a **shift/reduce conflict**)?
- Which of several reductions to make (a **reduce/reduce conflict**)?

An ambiguous grammar (for example, consider the dangling-else grammar):

- STMT → if STMT then STMT
- STMT → if STMT then STMT else STMT
- STMT → OTHER

Conflicts During Shift-Reduce Parsing

- If "if STMT then STMT" is a handle?
- A shift/reduce conflict.

STACK	INPUT	ACTION
$...if STMT then STMT	else ...$	Reduce by STMT → if STMT then STMT? Shift "else"?

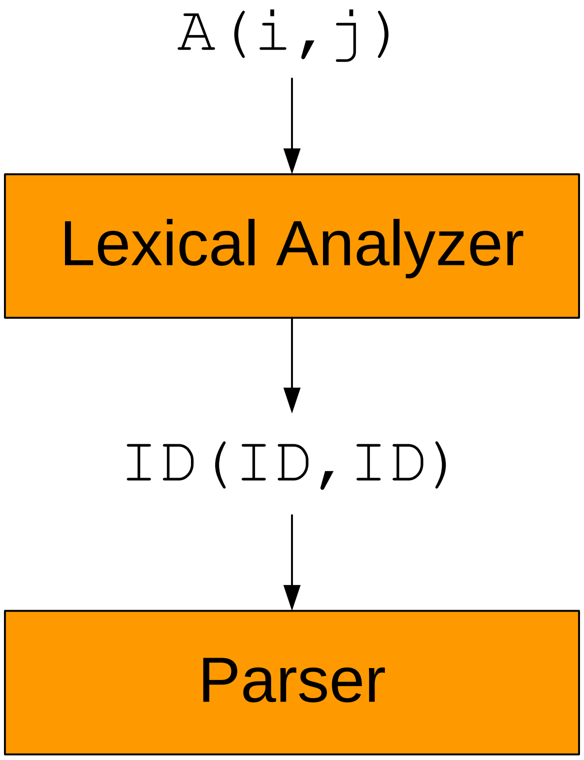
A possible decision: always shift!

Conflicts During Shift-Reduce Parsing

- Suppose we have a lexical analyzer that returns the token name id for all names, regardless of their type.
- Suppose also that procedures and arrays have the same syntax, for example:
- A(i,j): procedure A with parameters i,j;
- A(i,j): element of the array A with the indices i, j.
- Since the translation of indices in array references and parameters in procedure calls are different, we want to use different productions to generate lists of actual parameters and indices.

Conflicts During Shift-Reduce Parsing

- STMT → ID(PAR_LIST)
- STMT → EXPR:=EXPR
- PAR_LIST → PAR_LIST,PAR
- PAR_LIST → PAR
- PAR → ID
- EXPR → ID(EXPR_LIST)
- EXPR → ID
- EXPR_LIST → EXPR_LIST,EXPR
- EXPR_LIST → EXPR



Conflicts During Shift-Reduce Parsing

- Procedure or array?
- A reduce/reduce conflict.

STACK	INPUT	ACTION
\$...ID(ID	,ID) …\$	Reduce by “PAR → ID” or “EXPR → ID”?

.

Conflicts During Shift-Reduce Parsing

Possible decision:

- Use

STMT → PROCID(PAR_LIST)

instead of

STMT → ID(PAR_LIST).

- Use parsing table during lexical analysis.

STACK	INPUT	ACTION
$...ID(ID	,ID) …$	Reduce by “PAR → ID”
$...PROCID(ID	,ID) …$	Reduce by “EXPR → ID”

5.5. Introduction to LR-parsing: simple LR

LR(k)-grammars

- L – scanning the input from left to right;
- L – producing a rightmost derivation in reverse;
- k – using k input symbols of lookahead at each step to make parsing action decisions.

- The practical interest: k=0 and k=1.
- LR=LR(1).

Introduction to LR-parsing: simple LR

Why LR-parser?

- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written. Non-LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.

- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.

Introduction to LR-parsing: simple LR

Why LR-parser?

- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive or LL methods.

Introduction to LR-parsing: simple LR

Drawback

- Too much work to construct an LR parser by hand for a typical programming-language grammar. A specialized tool, an LR parser generator, is needed.

- Fortunately, many such generators are available (i.e.Yac**c)**. Such a generator takes a context-free grammar and automatically produces a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator locates these constructs and provides detailed diagnostic messages.

5.6. Items and the LR(0) Automaton

- When to shift and when to reduce?

$E \rightarrow T$
$T \rightarrow E+T$
$T \rightarrow E*T$
$T \rightarrow ID$
$T \rightarrow (E)$

STACK	INPUT	ACTION
$	ID+ID*ID$	shift
…	…	…
T+E$	*ID$	reduce by T →E+T
T$	*ID$	reduce by E →T

- Why reduce?

Items and the LR(0) Automaton

- An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse. States represent sets of "items“.

- An **LR(0) item** (item for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production A→XYZ yields the four items:
  - A→ · XYZ
  - A→X · YZ
  - A→XY · Z
  - A→XYZ ·

- The production A →ε generates only one item, A→ ·.

Items and the LR(0) Automaton

Intuitively, an item indicates how much of a production we have seen at a given point in the parsing process. For example,

- the item $A \rightarrow \cdot XYZ$ indicates that we hope to see a string derivable from XYZ next on the input;
- the item $A \rightarrow X \cdot YZ$ indicates that we have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ.
- Item $A \rightarrow XYZ \cdot$ indicates that we have seen the body XYZ and that it may be time to reduce XYZ to A.

Items and the LR(0) Automaton

- One collection of sets of LR(0) items, called the **canonical LR(0) collection**, provides the basis for constructing a deterministic finite automaton that is used to make parsing decisions. Such an automaton is called an **LR(0) automaton.**
- In particular, each state of the LR(0) automaton represents a set of items in the canonical LR(0) collection.

Items and the LR(0) Automaton

- To construct the canonical LR(0) collection for a grammar, we define an **augmented grammar** and two functions, **CLOSURE** and **GOTO**.
- If *G* is a grammar with start symbol *S,* then G', the **augmented grammar** for G, is *G* with a new start symbol S’ and production $S' \to S$. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of the input. That is, acceptance occurs when and only when the parser is about to reduce by $S' \to S$.

Items and the LR(0) Automaton

Closure of item sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

- Initially, add every item in I to CLOSURE(I).
- If $A\alpha \cdot B\beta \in$ CLOSURE(I) and $B \rightarrow \gamma$ is a production, then add the item $B \rightarrow \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

Items and the LR(0) Automaton

An augmented expression grammar:

- $E' \rightarrow E$
- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid ID$

- $I=\{E' \rightarrow \cdot E\}$

Rule 1:

- add $E' \rightarrow \cdot E$ to the CLOSURE(I);

Rule 2:

- $E \rightarrow E+T \in P$ so add $E \rightarrow \cdot E+T$ to the CLOSURE(I);
- $E \rightarrow T \in P$ so add $E \rightarrow \cdot T$ to the CLOSURE(I);
- $T \rightarrow T*F \in P$ so add $T \rightarrow \cdot T*F$ to the CLOSURE(I);
- …

Items and the LR(0) Automaton

An augmented expression grammar:

- $E' \to E$
- $E \to E+T \mid T$
- $T \to T*F \mid F$
- $F \to (E) \mid ID$

- $I=\{[E' \to \cdot E]\}$

CLOSURE(I):

- $E' \to \cdot E$
- $E \to \cdot E+T$
- $E \to \cdot T$
- $T \to \cdot T*F$
- $T \to \cdot F$
- $F \to \cdot (E)$
- $F \to \cdot ID$

Items and the LR(0) Automaton

A convenient way to implement the function *CLOSURE* is to keep a boolean array *added,* indexed by the nonterminals of *G,* such that **added[B]** is set to **true** if and when we add the item $B \rightarrow \cdot\gamma$ for each B-production $B \rightarrow \cdot\gamma$.

Items and the LR(0) Automaton

Note that if one B-production is added to the CLOSURE(I) with the dot at the left end, then all B-productions will be similarly added to the closure. Hence, it is not necessary to list the items $B \rightarrow \cdot\gamma$ added to the CLOSURE(I). A list of the nonterminals *B* whose productions were so added will suffice.

We divide all the sets of items of interest into two classes:

- **Kernel items:** the initial item, $S' \rightarrow \cdot S$, and all items whose dots are not at the left end.
- **Nonkernel items:** all items with their dots at the left end, except for $S' \rightarrow \cdot S$.

Items and the LR(0) Automaton

Each set of items of interest is formed by taking the closure of a set of kernel items; the items added in the closure can never be kernel items. Thus, we can represent the sets of items we are really interested in with very little storage if we throw away all nonkernel items, knowing that theycould be regenerated by the closure process.

Items and the LR(0) Automaton

The function GOTO

The second useful function is GOTO(I,X) where

- I is a set of items;
- X is a grammar symbol;
- GOTO(I,X) is defined to be the CLOSURE of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha X \cdot \beta] \in I$.

Intuitively, the GOTO function is used to define the transitions in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input X.

Items and the LR(0) Automaton

An augmented expression grammar:

- $E' \to E$
- $E \to E+T \mid T$
- $T \to T*F \mid F$
- $F \to (E) \mid ID$

- $I=\{[E' \to \cdot\, E],[E \to E \cdot +T]\}$
- Construct GOTO(I,+)

$[E \to E \cdot +T]$ contains +:

- move the dot the + and obtain $[E \to E+ \cdot T]$;
- construct the function CLOSURE($[E \to E+ \cdot T]$):
- $E \to E+ \cdot T$
- $T \to \cdot\, T*F$
- $T \to \cdot\, F$
- $F \to \cdot\, (E)$
- $F \to \cdot\, ID$

Items and the LR(0) Automaton

An augmented expression grammar:

- $E' \rightarrow E$
- $E \rightarrow E+T \mid T$
- $T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid ID$

- $I=\{[E' \rightarrow \cdot E],[E \rightarrow E \cdot +T]\}$

GOTO(I,+):

- $E \rightarrow E+ \cdot T$
- $T \rightarrow \cdot T*F$
- $T \rightarrow \cdot F$
- $F \rightarrow \cdot (E)$
- $F \rightarrow \cdot ID$

Items and the LR(0) Automaton

C – the canonical collection of sets of LR(0) items for an augmented grammar G‘:

- Add the CLOSURE([S’→ · S]) to C;
- For each I∈C and for each X: GOTO(I,X)≠Ø and GOTO(I,X)∉C add GOTO(I,X) to C;

- Repeat the last step until it is possible.

5.7. Use of the LR(0) Automaton

The central idea behind «Simple LR», or SLR, parsing is the construction from the grammar of the LR(0) automaton.

- The states of this automaton are the sets of items from the canonical LR(0) collection;
- the transitions are given by the GOTO function;
- the start state is the CLOSURE([S'→ · S]);
- all state are accepting states.

Use of the LR(0) Automaton

How can LR(0) automata help with shift-reduce decisions?

- Suppose that the string α of grammar symbols takes the LR(0) automaton from the start state 0 to some state j. Then, shift on next input symbol a if state j has a transition on a.
- Otherwise, we choose to reduce; the items in state j will tell us which production to use.

5.8. Viable prefixes

- The prefixes of right sentential forms that can appear on the stack of a shiftreduce parser are called **viable prefixes.** A viable prefix is a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form.
- By this definition, it is always possible to add terminal symbols to the end of a viable prefix to obtain a right-sentential form.
- LR(0) automaton recognizes viable prefixes.

Viable prefixes

$A \to \alpha A z$

$A \to \beta B y$

$B \to \gamma$

$A \to \alpha A z \to \alpha \beta B y z \to \alpha \beta \gamma y z$

STACK	INPUT	ACTION
$\$\alpha\beta\gamma$	yz$	reduce by $B \to \gamma$
$\$\alpha\beta B$	yz$	shift
$\$\alpha\beta B y$	z$	reduce by $A \to \beta B y$
$\$\alpha A$	z$	shift
$\$\alpha A z$	$	reduce by $A \to \alpha A z$
$\$A$	$	accept