Compilers

by
Marwa Yusuf

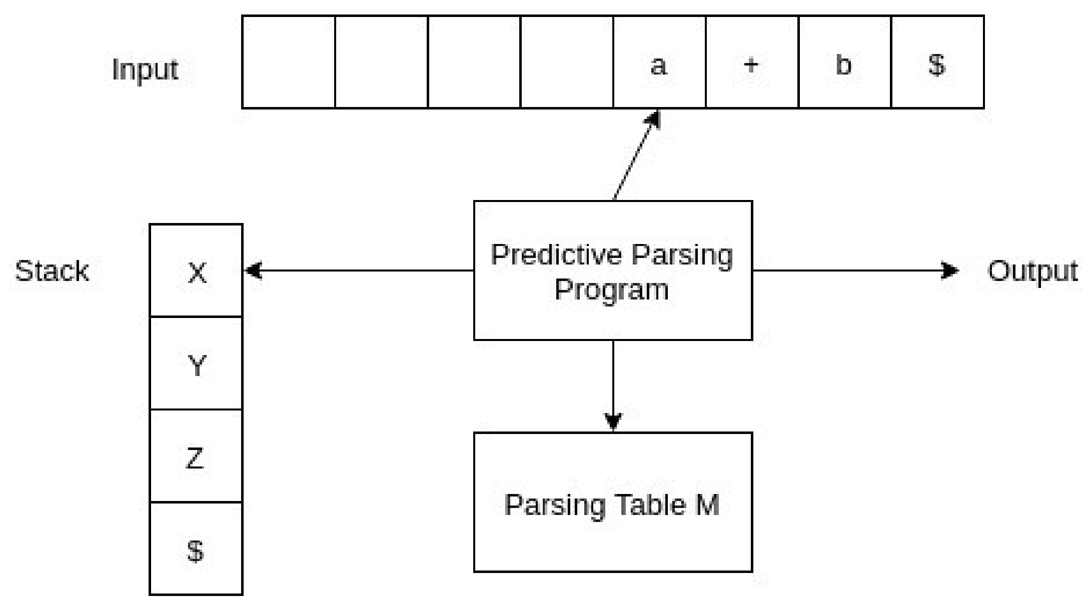
Lecture 7
Mon. 12-4-2021

Chapter 4 (4.4.4 to 4.6.2)

Syntax Analysis

Nonrecursive Predictive Parsing

- Maintain a stack explicitly.
- Mimics a leftmost derivation.
- If *w is the* input read so far, then the stack hold α such that $S \Rightarrow^*_{lm} w\alpha$
- Table driven parser:
- The parser behavior is described in terms of its configurations (the stack contents and the remaining input).



Nonrecursive Predictive Parsing Algorithm

- **INPUT**: A string w and a parsing table M for grammar G.
- **OUTPUT**: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.
- **METHOD**: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The following program uses the predictive parsing table M to produce a predictive parse for the input.

Nonrecursive Predictive Parsing Code

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*] = $X \rightarrow Y_1\ Y_2 \cdots Y_k$) {
    output the production $X \rightarrow Y_1\ Y_2 \cdots Y_k$;
    pop the stack;
    push $Y_k, Y_{k-1}, \ldots, Y_i$ onto the stack, with Y_1 on top;
  }
  set *X* to the top stack symbol;
}

Example

MATCHED	STACK	INPUT	ACTION
	E$	**id + id * id$**	
	TE'$	**id + id * id$**	output $E \rightarrow TE'$
	FT'E'$	**id + id * id$**	output $T \rightarrow FT'$
	id T'E'$	**id + id * id$**	output $F \rightarrow$ **id**
id	T'E'$	**+ id * id$**	match **id**
id	E'$	**+ id * id$**	output $T' \rightarrow \varepsilon$
id	+ TE'$	**+ id * id$**	output $E' \rightarrow +TE'$
id +	TE'$	**id * id$**	match +
id +	FT'E'$	**id * id$**	output $T \rightarrow FT'$
id +	**id** T'E'$	**id * id$**	output $F \rightarrow$ **id**
id + id	T'E'$	*** id$**	match **id**
id + id	* FT'E'$	*** id$**	output $T' \rightarrow *FT'$
id + id *	FT'E'$	**id$**	match *
id + id *	**id** T'E'$	**id$**	output $F \rightarrow$ **id**
id + id * id	T'E'$	**$**	match **id**
id + id * id	E'$	**$**	output $T' \rightarrow \varepsilon$
id + id * id	$	**$**	output $E' \rightarrow \varepsilon$

	Input Symbol					
	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**			$F \rightarrow (E)$		

Error Recovery in Predictive Parsing

- An error happens when the terminal on top of stack does not match the next input symbol, or when non-terminal A is on top of stack, a is next input symbol, and M[A,a] is **error**.

Panic Mode

- Skip input symbols until a synchronizing token is reached.
- Effectiveness depends on choice of Synchronizing tokens.
- Some options:
  - Place all FOLLOW(A) into synchronizing set of A.
  - Place symbols beginning higher level constructs into synchronizing set of lower level constructs. (expressions within statements).
  - Place all FIRST(A) into synchronizing set of A.
  - If $A \Rightarrow^* \varepsilon$, then use the production deriving ε as a default. May postpone error detection, but no error is lost.
  - If top of stack is terminal, pop it, report, and continue (place all other tokens in the synchronizing set of a token).

Example

$E \rightarrow T\,E'$

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

$T \rightarrow F\,T'$

$T' \rightarrow *\,\mathrm{F}\,T' \mid \varepsilon$

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

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

- If blank, skip symbol.
- If synch, pop top non-terminal.
- If top token not matched, pop it.

Non-Terminal	Input Symbol					
	id	+	*	(	)	$
E	$E \rightarrow TE'$			$E \rightarrow TE'$	synch	synch
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	synch		$T \rightarrow FT'$	synch	synch
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \textbf{id}$	synch	synch	$F \rightarrow (E)$	synch	synch

Example

STACK	INPUT	REMARK
E $	+ **id** * + **id** $	error, skip +
E $	**id** * + **id** $	**id** is in FIRST(E)
T E' $	**id** * + **id** $	
F T' E' $	**id** * + **id** $	
id T' E' $	**id** * + **id** $	
T' E' $	* + **id** $	
* F T' E' $	* + **id** $	
F T' E' $	+ **id** $	error, M[F, +] = synch
T' E' $	+ **id** $	F has been popped
E' $	+ **id** $	
+ T E' $	+ **id** $	
T E' $	**id** $	
F T' E' $	**id** $	
id T' E' $	**id** $	
T' E' $	$	
E' $	$	
$	$	

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

Panic Mode

- Note: The compiler designer must supply informative error message (what and where).

Phrase Level Recovery

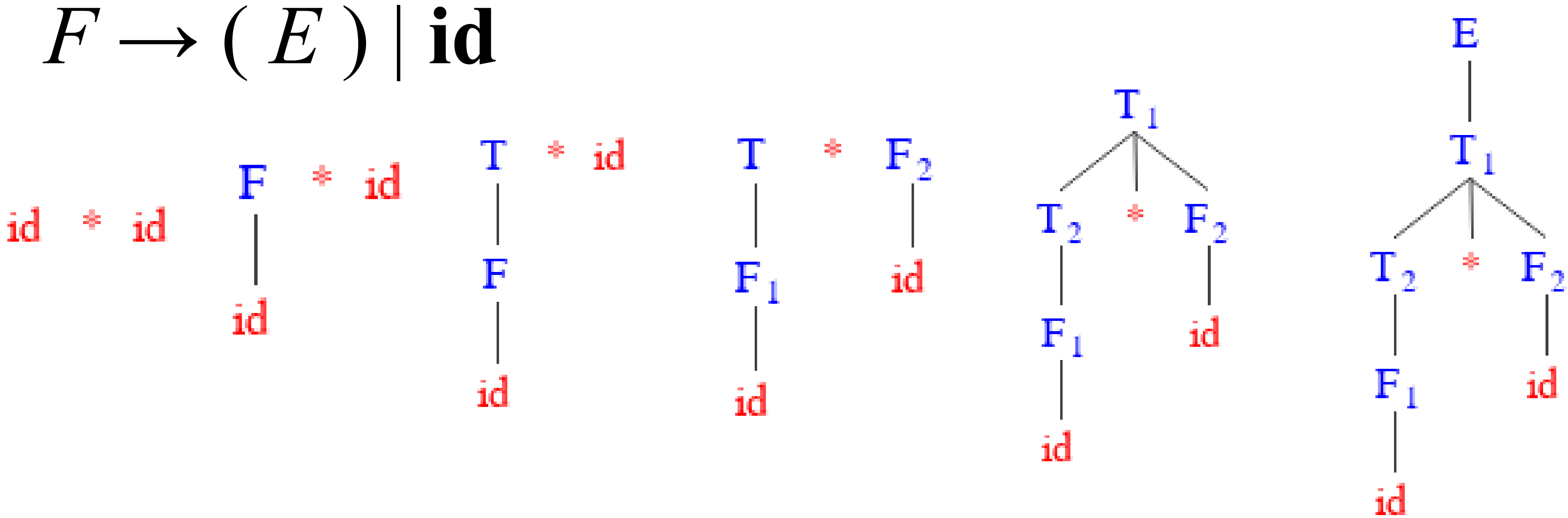
- Filling in blank entries in the table with pointers to error routines.
  - Change, insert or delete symbols in the input and report.
  - Pop from the stack.
- Alteration (or pushing) stack symbols is questionable:
  - May result in no valid derivation.
  - Possible infinite loop: checking that an input symbol is consumed, (or stack shortened) can be used as a protection.

Bottom-Up Parsing

- From leaves and up to the root.
- Shift-reduce parsing, for LR grammar, hard to build by hand, easy using generators.
- $E \to E + T \mid T$

  $T \to T * F \mid F$

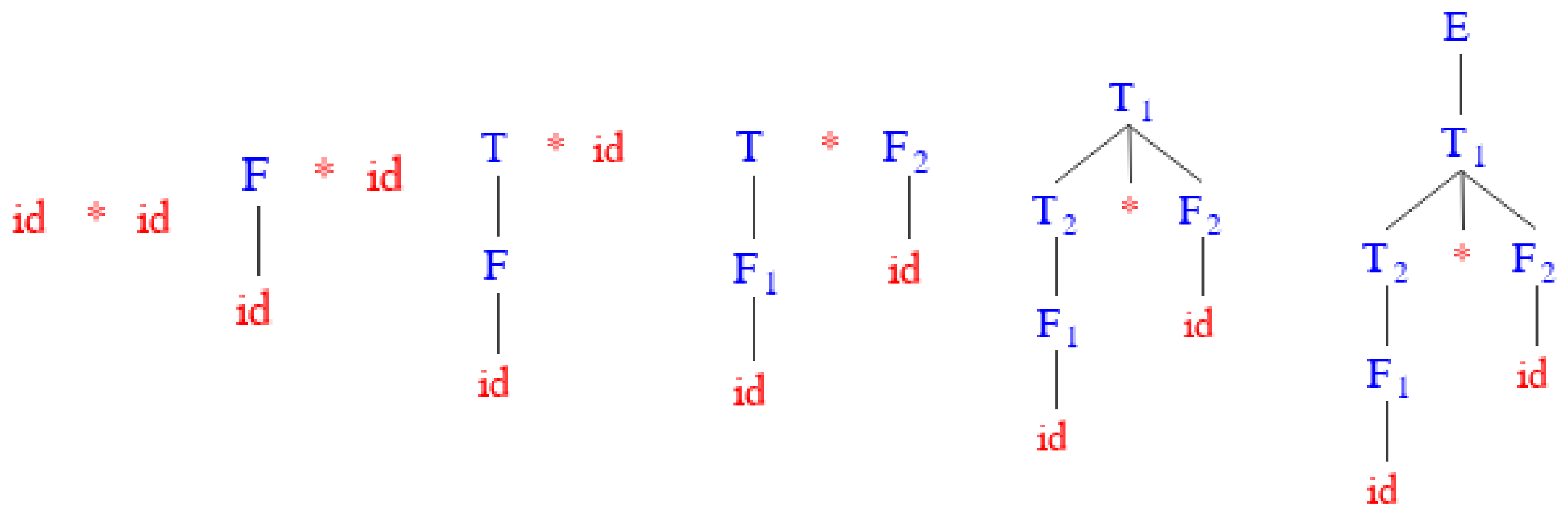
  $F \to (\ E\ ) \mid \mathbf{id}$



Reductions

- Reducing the input string to the start symbol. (at each step, a substring is replaced by a non-terminal)
- The decision: when to reduce and what production to use.

Example



- **id** * **id**, F * **id**, T * **id**, $T * F$, T, E (root)
- A reduction is the reverse of a derivation.
- The prev. reduction is the reverse of a rightmost derivation.
- $E \Rightarrow T \Rightarrow T * F \Rightarrow T * \textbf{id} \Rightarrow F * \textbf{id} \Rightarrow \textbf{id} * \textbf{id}$

Handle Pruning

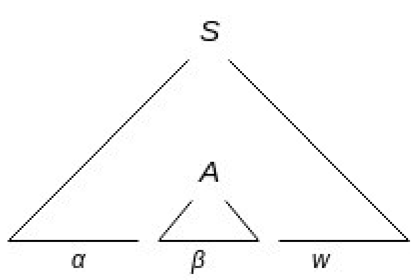
- A “handle” is a substring matching the body of a production, and its reduction is a step in the reverse of rightmost derivation.

Right sentential Form	Handle	Reducing Production
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \rightarrow F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

- T is not a handle in $T * \mathbf{id}_2$ (If replaced by T would give wrong) (leftmost substring that matches some body need not be a handle).

Handle Definition (formal)

- If $S \Rightarrow^*_{\mathrm{rm}} \alpha A w \Rightarrow_{rm} \alpha\beta w$, then production A → β in the position following α is a handle of $\alpha\beta w$.
- Alternatively, a handle of a right-sentential form γ is a production $A \rightarrow \beta$ and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ.
- w to the right of the handle must contain only terminals.
- For convenience, we refer to the body β rather than $A \rightarrow \beta$ as a handle.
- Ambiguous grammar → "a handle”.
- Unambiguous grammar → every right-sentential form has exactly one handle.



Handle Pruning

- A rightmost derivation can be obtained by handle pruning.
- $S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \ldots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = w$
- Find β_n in γ_n, replace β_n by the head of $A \rightarrow \beta_n$ to obtain γ_{n-1}
- Repeat till reach S, then successful.

Shift-Reduce Parsing

- A bottom-up parsing: a stack holding grammar symbols.
- Handle always on top of the stack (at the right, conventionally).
- Initially: Stack Input

  $ *w* $

- Finally, either ERROR or : Stack Input

  $ *S* $

- Operations: *Shift, Reduce, Accept, Error.*

Example

STACK	INPUT	ACTIONS
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	shift
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	reduce by $F \rightarrow \mathbf{id}$
$ F	$* \mathbf{id}_2$ $	reduce by $T \rightarrow F$
$ T	$* \mathbf{id}_2$ $	shift
$ $T *$	$\mathbf{id}_2$ $	shift
$ $T * \mathbf{id}_2$	$	reduce by $F \rightarrow \mathbf{id}$
$ $T * F$	$	reduce by $T \rightarrow T * F$
$ T	$	reduce by $E \rightarrow T$
$ E	$	accept

Conflicts

- There are context-free grammars for which shift reduce parsing cannot be used, not in LR(k) class (non-LR grammar).
- There are *shift/reduce* conflict and *reduce/reduce* conflict.
- Example: Ambiguous G cannot be LR:

$stmt \rightarrow$ **if** *expr* **then** *stmt*

| **if** *expr* **then** *stmt* **else** *stmt*

| **other**

Stack	Input
… **if** *expr* **then** *stmt*	**else** … $

shift/reduce conflict

- We can favor *shift*, as a workaround in this case.

Conflicts

- Example:
  1) $stmt \rightarrow$ **id** ($parameter\_list$)
  2) $stmt \rightarrow expr := expr$
  3) $parameter\_list \rightarrow parameter\_list, parameter$
  4) $parameter\_list \rightarrow parameter$
  5) $parameter \rightarrow$ **id**
  6) $expr \rightarrow$ **id** ($expr\_list$)
  7) $expr \rightarrow$ **id**
  8) $expr\_list \rightarrow expr\_list , expr$
  9) $expr\_list \rightarrow expr$

Stack	Input
… **id** (**id**	, **id**) … $

- Could use symbol table.
- Change **id** in (1) to **procid**, and rely on lexical analyzer (with help of symbol table)

Stack	Input
… **procid** (**id**	, **id**) … $

- Note that 3rd symbol in stack determines which production.

Intro to LR Parsing

- LR(k) parsing: L for scanning left to right, R for rightmost derivation in reverse, K look-ahead symbols.
- We will only consider K<= 1. LR by default is LR(1).

LR Grammar

- LR parser is table driven.
- LR Grammar: a grammar for which you can construct a parsing table (as will be shown).
- For a grammar to be LR, sufficient that a left-to right shift reduce can recognize handles of right-sentential forms when they appear on top of the stack.

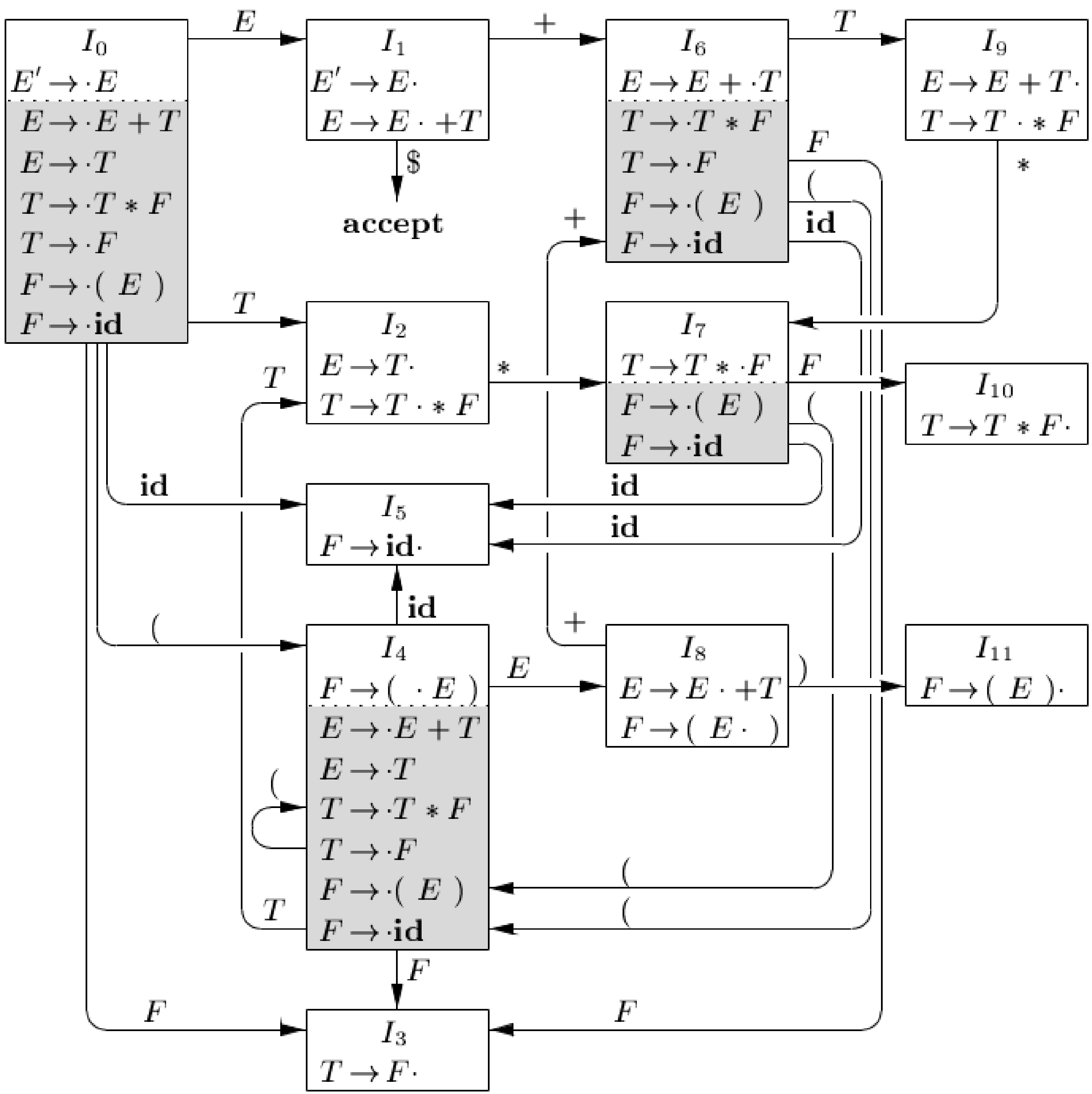
Why LR Parser?

- For reading.

Items & LR Automaton

- Shift-reduce decisions: states to keep track of parsing position.
- States represent sets of items.
- An (LR(0)) item of a grammar G: a production with a dot at some position in body.
- $A \rightarrow XYZ$ yields:
  - $A \rightarrow \cdot XYZ$ (hope to see string derivable from XYZ on the input.
  - $A \rightarrow X \cdot YZ$ (saw a string derivable from X and hope to see a string derivable from YZ).
  - $A \rightarrow XY \cdot Z$
  - $A \rightarrow XYZ \cdot$ (*saw* a string derivable from XYZ and may be the time to reduce XYZ to A).
- $A \rightarrow \varepsilon$ yields: $A \rightarrow \cdot$
- *Canonical* LR(0): one collection of sets of LR(0) items.
  - Provides the basis for constructing a DFA (LR(0) automaton), used for parsing decisions.
  - Each state represents a set of items.

Example Automaton



Constructing canonical LR(0) collection C

- Augmented grammar G' for grammar G = G with new start symbol S' and production $S' \rightarrow S$.
- Acceptance occurs when and only when about to reduce by $S' \rightarrow S$.
- To construct canonical LR(0) collection we need augmented grammar and CLOSURE and GOTO functions.
- If I is a set of items, CLOSURE(I):
  - Add every item in I to CLOSURE(I).
  - If $A \rightarrow \alpha \cdot B\beta$ is in CLOSURE(I), then add each $B \rightarrow \cdot\gamma$ until no more items can be added.

Example

- $E' \rightarrow E$

  $E \rightarrow E + T \mid T$

  $T \rightarrow T * F \mid F$

  $F \rightarrow (E) \mid \textbf{id}$
- If I is the set $\{[E' \rightarrow \cdot E]\}$ then CLOSURE(I) is I_0 in the prev. figure.
- It may be sufficient to list non-terminals, not productions.

1) Kernel items: S' → ·S and all items with no dots on the left.

2) Nonkernel items: the rest.

- Nonkernel are shaded in figure.

CLOSURE computation algorithm

```
SetOfltems CLOSURE(I) {
  J = I;
  repeat
    for ( each item A → α·Bβ in J )
      for ( each production B → γ of G )
        if ( B → γ is not in J )
          add B → γ to J;
  until no more items are added to J on one round;
  return J;
}
```

GOTO function

- The transition from the state for I under input X.
- $GOTO(I, X)$: the closure of the set of all items $[A \rightarrow \alpha X \cdot \beta]$ such that $[A \rightarrow \alpha \cdot X \beta]$ is in I.
- If $I := \{[E' \rightarrow E\cdot], [E \rightarrow E\cdot + T]\}$

  then GOTO(I, +) contains:

  $E \rightarrow E +\cdot T$

  $T \rightarrow \cdot T * F$

  $T \rightarrow \cdot F$

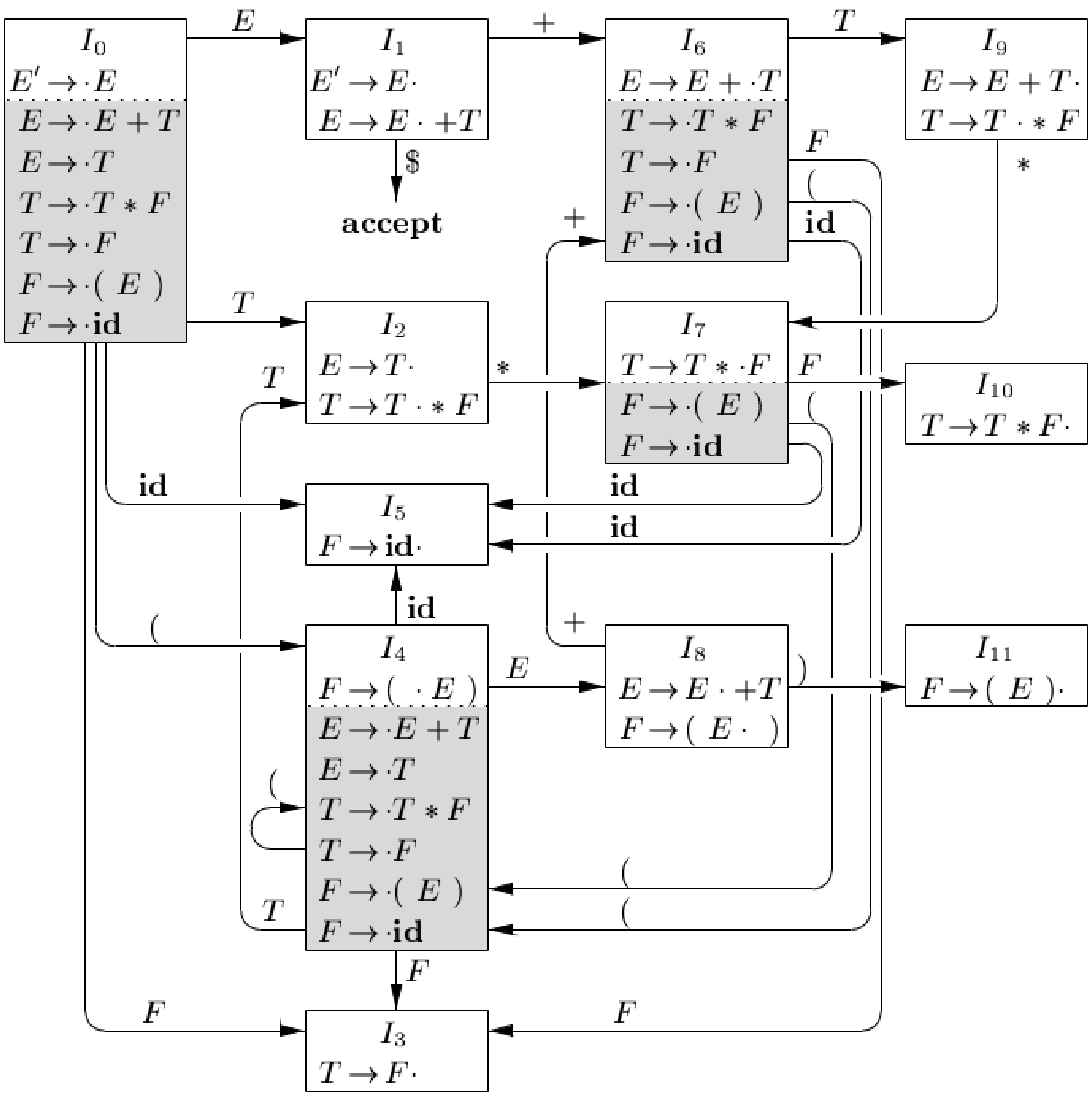
  $F \rightarrow \cdot(E)$

  $F \rightarrow \cdot\mathbf{id}$
- Find items with + immediately to the right of the dot.

Algorithm to construct C

```
void items(G'){
  C = CLOSURE({[S' -> ·S]});
  repeat
    for ( each set of items I in C )
      for ( each grammar symbol X )
        if ( GOTO(I, X) is not empty and not in C )
          add GOTO(I, X) to C;
  until no new sets of items are added to C on a
  round;
}
```

Example



I0
E' → ·E
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I1
E' → E·
E → E · +T
accept
I2
E → T·
T → T · * F
I3
T → F·
I4
F → (· E)
E → ·E + T
E → ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I5
F → id·
I6
E → E + ·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id
I7
T → T * ·F
F → ·(E)
F → ·id
I8
E → E · +T
F → (E ·)
I9
E → E + T·
T → T · * F
I10
T → T * F·
I11
F → (E)·