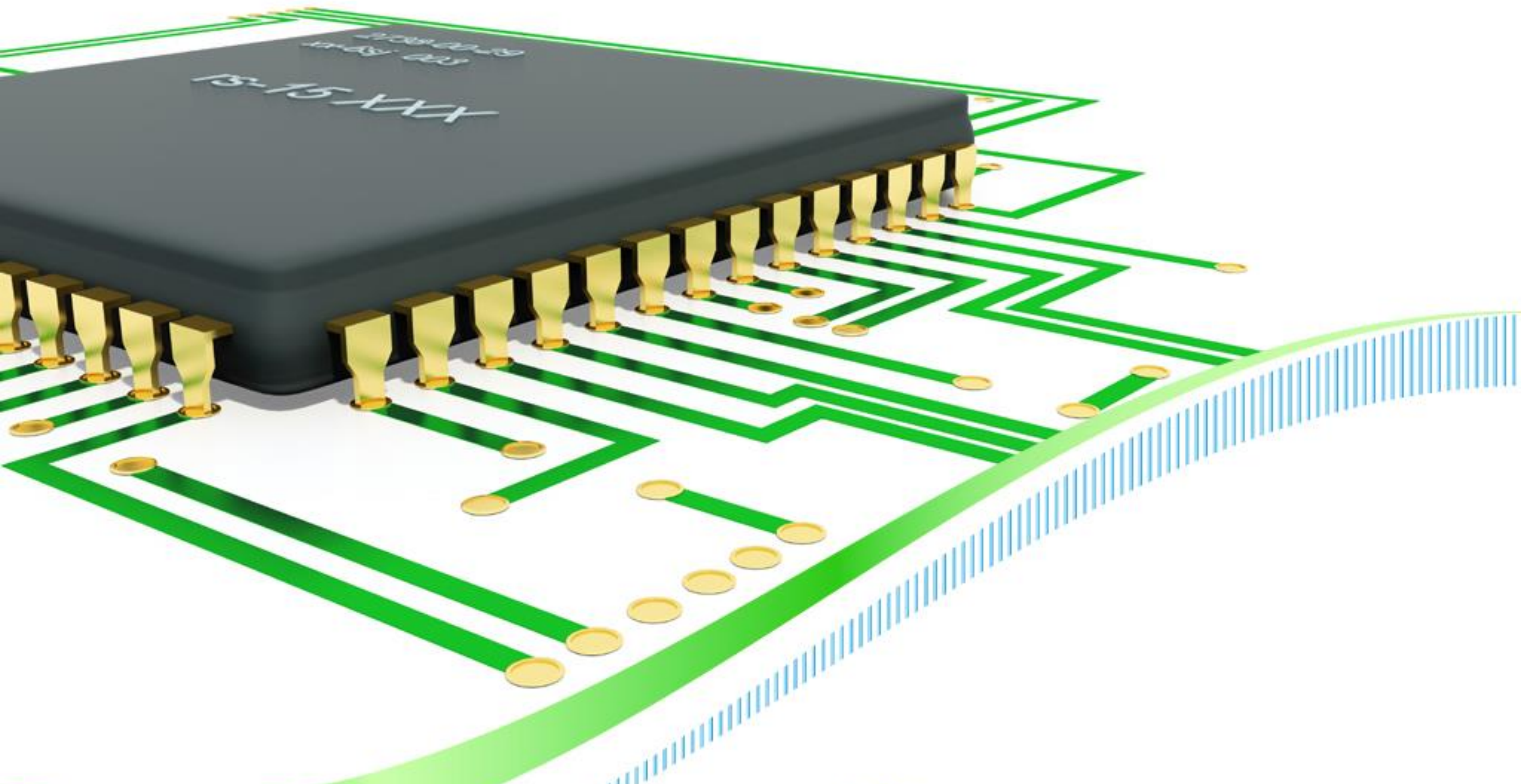
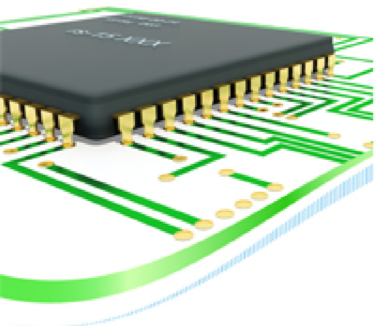


Syntactic Analysis

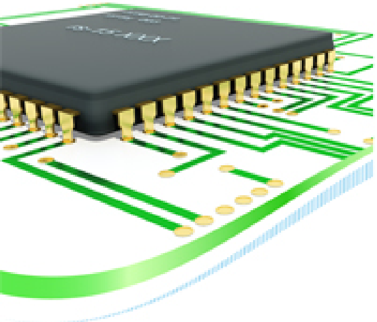


Definition

In a typical language processor implementation, syntactic analysis is the attempt to recognize string of tokens as a sentence in a given language

Parser is the general term for the component that performs the syntactic analysis.

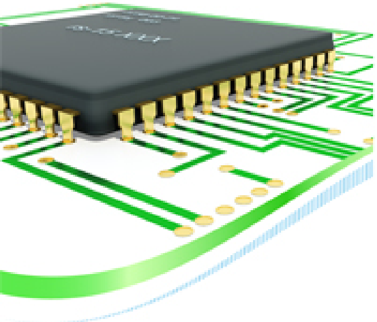
Syntactic Validity



Yes, valid program is a sentence.

Usually, validity transcends syntax.

- Declarative Integrity
- Parameter Matching
- Type Conformance
- Operator Applicability
- Statement Applicability
- …



Grammars

CFG and (E)BNF

- **Context Free Grammar (Type 2: Grammars)**

  G = (V, Σ, R, S)

  V: Set of non-terminal (Variables)

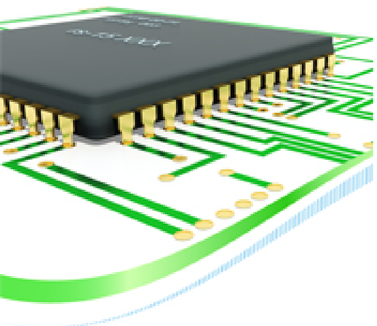
  Σ: Set of terminals (Reported by lexical analyzer)

  R: Rule set (Production rules, productions)

  S: Start variable

- **Backus-Naur Form (Notation)**

$$
\begin{array}{lcl}
rexpr & \rightarrow & rexpr + rterm \mid rterm \\
rterm & \rightarrow & rterm\ rfactor \mid rfactor \\
rfactor & \rightarrow & rfactor * \mid rprimary \\
rprimary & \rightarrow & \mathbf{a} \mid \mathbf{b}
\end{array}
$$



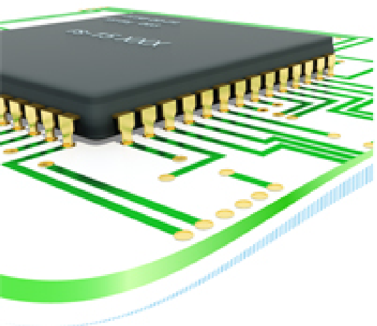
Grammars

BNF for Human Reader

RFC 822

```
     The syntax of the standard,  in  RFC  #733,  was  originally
specified  in  the  Backus-Naur Form (BNF) meta-language.  Ken L.
Harrenstien, of SRI International, was responsible for  re-coding
the  BNF  into  an  augmented  BNF  that makes the representation
smaller and easier to understand.

....

dtext       =  <any CHAR excluding "[",     ; => may be folded
                "]", "\" & CR, & including
                linear-white-space>

comment     =  "(" *(ctext / quoted-pair / comment) ")"

ctext       =  <any CHAR excluding "(",     ; => may be folded
                ")", "\" & CR, & including
                linear-white-space>

quoted-pair =  "\" CHAR                     ; may quote any char

phrase      =  1*word                       ; Sequence of words

word        =  atom / quoted-string
```

The excerpt is from RFC 822, “Standard for the Format of ARPA Internet Text Messages; David H. Crocker;”, https://datatracker.ietf.org/doc/html/rfc822



Grammars

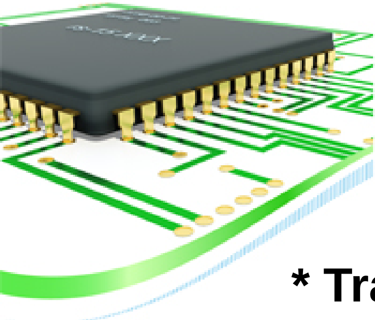
(E)BNF for a Language Processor

From a Parser Generator Project

```
/*********************************************************************\
      Lexical elements and rules referenced by both RFC822 and MIME.
\*********************************************************************/

msg_id          : Matches(<:LessThan) addr_spec Matches(>:GreaterThan);
addr_spec       : local_part AtSign domain;
local_part      : word *(Dot word);
word            : Matches(+!{#0-#32,specials} :atom) |
                  Matches(" *(!{#0,",\\,#13}|\\{#1-#255}) ":quoted_string);
domain          : sub_domain *(Dot sub_domain);
sub_domain      : domain_ref | Matches([*(!{#0,[,],\\,#13}|\\{#1-#255})]:domain_literal);
domain_ref      : atom;
```

Grammars

BNF Simplified



* Transformation

u e* w → u e’ w
e' : e’e | ε (Left recursive form)
e' : e e’ | ε (Right recursive form)

+ Transformation

u e+ w → u e e’ w
e' : e’e | ε (Left recursive form)
e' : e e’ | ε (Right recursive form)

? Transformation

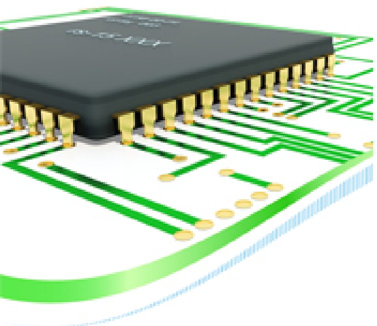
u e? w → u e’ w
e' : e | ε

Parentheses

u (e) w → u e’ w
e' : e

Alternative Notations

[e], {e}, prefix, postfix, etc...



Derivation

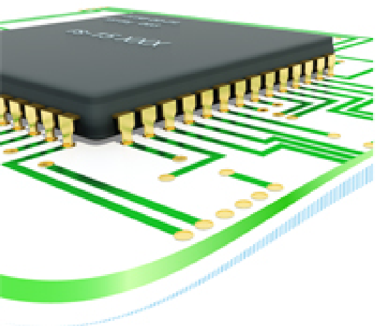
Definition

Derivation is a sequence of rewriting steps that begins with the grammar’s start symbol and ends with a sentence in the language.

Definition from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

Sentential Form

Definition



Sentential form is a string of symbols that occurs as a <u>step</u> in a valid derivation.

A step is making use of a rule on the latest sentential form.

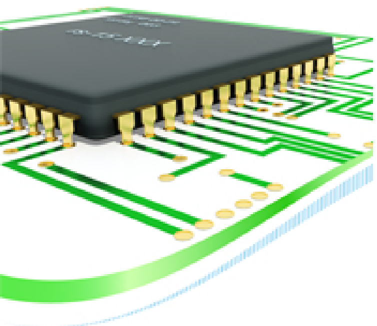
- **Choose a variable.**
- **Replace with the associated production.**

$$G = (V, \Sigma, R, S)$$

$$f_n = uAv \qquad A \in V \qquad A \rightarrow w \in R \qquad f_{n+1} = uwv \qquad uAv,\ uwv \in S^*$$

$$u, v \in (V \cup \Sigma)^*$$

Definition from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



Derivation

$S^* \Rightarrow \alpha$ means α can be found as a sentence starting from S by zero or more derivation steps.

$\alpha \overset{*}{\Rightarrow} \beta$ means β is reachable from α by zero or more derivation steps.

$\alpha \overset{+}{\Rightarrow} \beta$ means β is reachable from α by one or more derivation steps.

$\alpha \overset{*}{\Rightarrow} \beta$ and $\beta \overset{*}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\Rightarrow} \gamma$

Example: Input: aaba

G_1 :

$S \to aA \mid abC$

$A \to aA \mid bA \mid C \mid \varepsilon$

$C \to c$

G_1 :

$S \to aA$

$S \to abC$

$A \to aA$

$A \to bA$

$A \to C$

$A \to \varepsilon$

$C \to c$

Derivations

LMD, RMD, Left and Right Sentential Forms

Leftmost Derivation (LMD) chooses the leftmost variable.
Rightmost Derivation (RMD) chooses the rightmost variable.
Any step in LMD is leftmost sentential form.
Any step in RMD is rightmost sentential form.

Finding sentence aaba by G1

S use S → aA
aA use A → aA
aaA use A → bA
aabA use A → aA
aabaA use A → ε
aaba

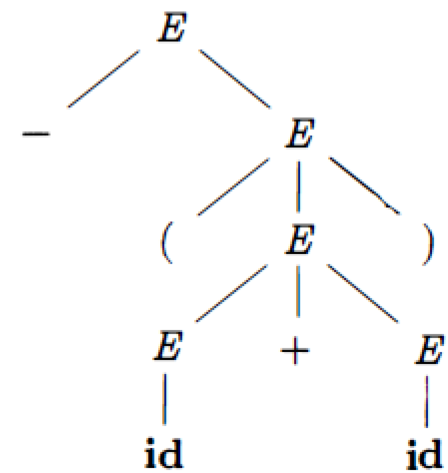
Parse Trees

Successive steps taken in a derivation can be represented with help of trees, which are known as parse trees.

G: E → E **+** E | E ***** E | **-** E | **(** E **)** | **id**

E → E **+** E
E → E ***** E
E → **-** E
E → **(** E **)**
E → **id**

Input: - (id + id)



Example from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

G: $E \rightarrow E \mathbf{+} E \mid E \mathbf{*} E \mid \mathbf{-} E \mid \mathbf{(} E \mathbf{)} \mid \mathbf{id}$

$E \rightarrow E \mathbf{+} E$
$E \rightarrow E \mathbf{*} E$
$E \rightarrow \mathbf{-} E$
$E \rightarrow \mathbf{(} E \mathbf{)}$
$E \rightarrow \mathbf{id}$

Input: id + id * id

Create leftmost derivations for this sentence!

Definition and example from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

Ambiguity
Elimination

Rewriting

G:

E → E **+** E | E ***** E | **-** E | **(** E **)** | **id**

E → E **+** E

E → E ***** E

E → **-** E

E → **(** E **)**

E → **id**

G:

E → E **+** T | T

T → T ***** F | F

F → **-** E | **(** E **)** | **id**

E → E **+** T

E → T

T → T ***** F

T → F

F → **-** E

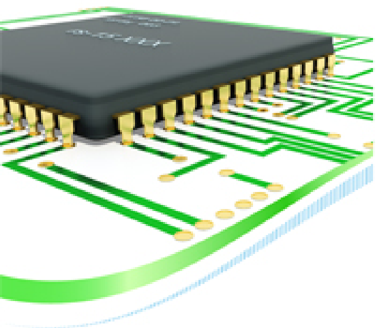
F → **(** E **)**

F → **id**

Inputs: id + id * id, id + id + id, id * id * id

Definition and example from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

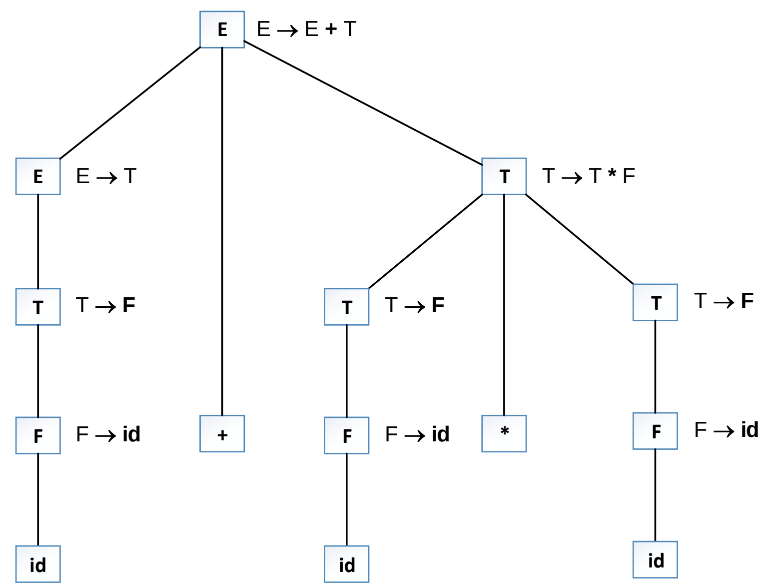
Parsing
as Tree Building Method



Top-Down Parsing
Begins with the root and grows the tree toward the leaves.

Bottom-Up Parsing
Begins with the leaves and grows the tree toward the root.

Input: id + id * id



Definition from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

Top Down Parsing

The Left Recursion Problem

Input: id + id * id

G:

E → E **+** T I T
T → T ***** F | F
F → **-** E I **(** E **)** I **id**

E → E **+** T
E → T
T → T ***** F
T → F
F → **-** E
F → **(** E **)**
F → **id**

Immediate Recursions

A → A α | β

Elimination

A → β A’
A’ → α A’ | ε

Elimination rule from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

Left Recursion

Elimination

When Grammar has no Cycles and no ε-productions

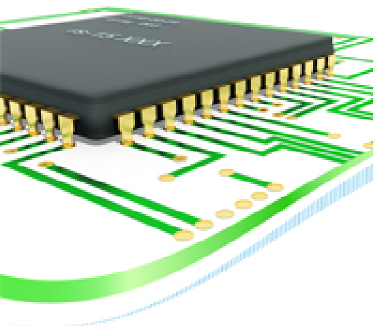
General Elimination Algorithm

arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
for (each i from 1 to n) {
    for (each j from 1 to $i-1$) {
        replace each production of the form $A_i \to A_j\gamma$ by the
            productions $A_i \to \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where
            $A_j \to \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ are all current A_j-productions
    }
    eliminate the immediate left recursion among the A_i-productions
}

Algorithm from “Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book”

Left Recursion
Elimination

General Elimination Algorithm



G:

E → E **+** T l T
T → T ***** F | F
F → **-** E l **(** E **)** l **id**

E → E **+** T
E → T
T → T ***** F
T → F
F → **-** E
F → **(** E **)**
F → **id**

Input: id + id * id

G:

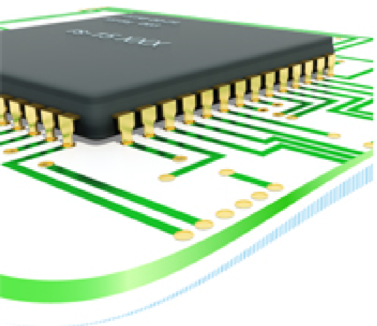
E → E **+** T l T
T → T ***** F | F
F → **-** E l **(** E **)** l **id**

E → T E'
E' → +T E' | ε

T → F T'
T' → * F T' | ε

F → **-** E
F → **(** E **)**
F → **id**

Backtracking



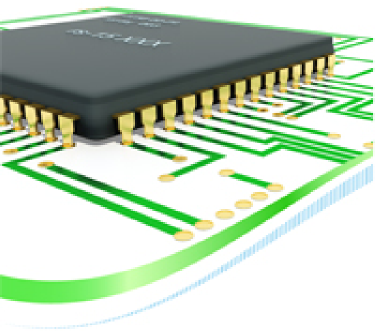
The leftmost, top-down parser

root $\leftarrow$ *node for the start symbol,* S*;*
focus $\leftarrow$ *root;*
push(null);
word $\leftarrow$ *NextWord( );*

while (true) do;
    if (focus is a nonterminal) then begin;
        pick next rule to expand focus ($A \rightarrow \beta_1, \beta_2, \ldots, \beta_n$)*;*
        build nodes for $\beta_1, \beta_2 \ldots \beta_n$ *as children of focus;*
        push($\beta_n, \beta_{n-1}, \ldots, \beta_2$)*;*
        focus $\leftarrow \beta_1$*;*
    end;
    else if (word matches focus) then begin;
        word $\leftarrow$ *NextWord( );*
        focus $\leftarrow$ *pop( )*
    end;
    else if (word = eof and focus = null)
        then accept the input and return root;
        else backtrack;
end;

Algorithm from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

Backtracking



Backtrack-free Grammar / Predictive Grammar
a CFG for which the leftmost, top-down parser can always predict the correct rule with lookahead of at most one word.

Elimination Using First and Follow Sets

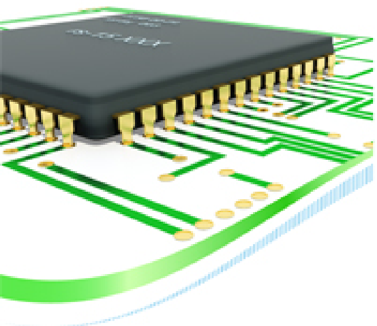
FIRST Set
For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

FOLLOW Set
For a nonterminal α, FOLLOW(α) contains the set of words that can occur immediately after α in a sentence.

Definitions from "Cooper, K.D., Torczon, L.; Engineering A Compiler"

First Set

Construction



```
for each α ∈ (T ∪ eof ∪ ϵ) do;
    FIRST(α) ← α;
end;

for each A ∈ NT do;
    FIRST(A) ← Ø ;
end;

while (FIRST sets are still changing) do;
   for each p ∈ P, where p has the form A→β do;
       if β is β1β2...βk, where βi ∈ T ∪ NT, then begin;
         rhs ← FIRST(β1) − {ϵ};
         i ← 1;
         while (ϵ ∈ FIRST(βi) and i ≤ k-1) do;
             rhs ← rhs ∪ (FIRST(βi+1) − {ϵ}) ;
             i ← i + 1;
         end;
       end;
      if i = k and ϵ ∈ FIRST(βk)
          then rhs ← rhs ∪ {ϵ};
      FIRST(A) ← FIRST(A) ∪ rhs;
   end;
end;
```

G = (V, Σ, R, S)

What are T, NT, P?

Algorithm from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

First Set
Example

G:

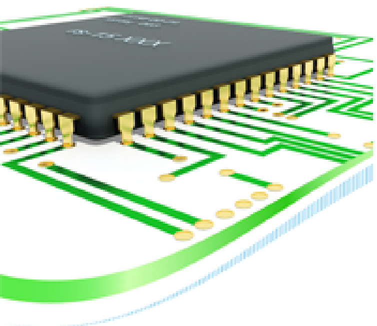
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow - E \mid (E) \mid \textbf{id}$

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

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

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

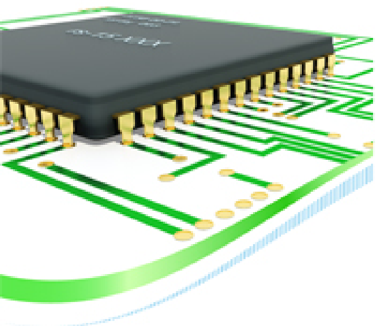
Build the first sets!



Follow Set
Construction

```
for each A ∈ NT do;
    FOLLOW(A) ← Ø ;
end;

FOLLOW(S) ← {eof};

while (FOLLOW sets are still changing) do;
   for each p ∈ P of the form A → β1β2···βk do;
      TRAILER ← FOLLOW(A);

      for i ← k down to 1 do;
         if βi ∈ NT then begin;

            FOLLOW(βi) ← FOLLOW(βi) ∪  TRAILER;

            if ϵ ∈ FIRST(βi)
               then TRAILER ← TRAILER ∪ (FIRST(βi) − ϵ);
               else TRAILER ← FIRST(βi);
         end;
         else TRAILER ← FIRST(βi);     // is {βi}
      end;
   end;
end;
```

Algorithm from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



Follow Set

Example

G:

E → E **+** T | T
T → T ***** F | F
F → **-** E | **(** E **)** | **id**

E → T E’
E’ → +T E’
E’ → ε

T → F T’
T’ → * F T’
T’ → ε

F → **-** E
F → **(** E **)**
F → **id**

Build the follow sets!

Backtracking

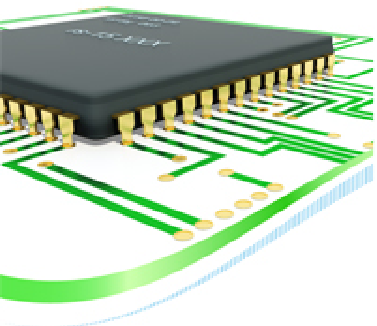
Backtrack-free Test

Test each non-terminal with multiple RHS!

$$First^{+}(A \longrightarrow \beta) = \begin{cases} First(\beta), & \varepsilon \notin First(\beta) \\ First(\beta) \cup Follow(A), & otherwise \end{cases}$$

$$First^{+}(A \longrightarrow \beta_i) \cap First^{+}(A \longrightarrow \beta_j) = \emptyset, \quad \forall\, 1 \leq i, j \leq n, \quad i \neq j$$

Test method from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



Left Factoring

G:

E → E **+** T | T
T → T ***** F | F
F → **-** E | **(** E **)** | **id**

E → T E’
E’ → +T E’
E’ → ε

T → F T’
T’ → * F T’
T’ → ε

F → **-** E
F → **(** E **)**
F → **id**
F → **id (** E **)**
F → **id [** E **]**

When array and function call like structures added!

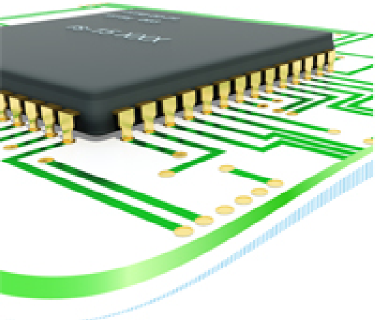
G:

E → E **+** T | T
T → T ***** F | F
F → **-** E | **(** E **)** | **id**

.
.
.
.

F → **-** E
F → **(** E **)**
F → **id** G

G → **(** E **)**
G → **[** E **]**
G → ε



Predictive Parsers

Stages of Test & Elimination

- Ambiguity
- Left Recursion
- Backtracking
- Left Factoring

Can you revise the leftmost, top-down parser?

Now, we can focus on

- Recursive Descent Parsers
- Table Driven LL(1) Parser

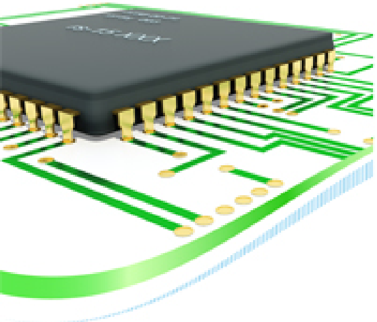
Recursive Descent Parsing

```
Expr( )
    /* Expr → Term Expr′  */
    if ( Term( ) )
        then return EPrime( );
        else Fail();

EPrime( )
    /* Expr′→ + Term Expr′  */
    /* Expr′→ - Term Expr′  */
    if (word = + or word = - )
        then begin;
           word ← NextWord( );
           if ( Term() )
               then return EPrime( );
               else Fail();
        end;
```

A procedure for each NT

- Hand coded
- Or, automatically generated based on grammar
- Ease of customization, debugging

Sample pseudo-code from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



LL(1) Parsing

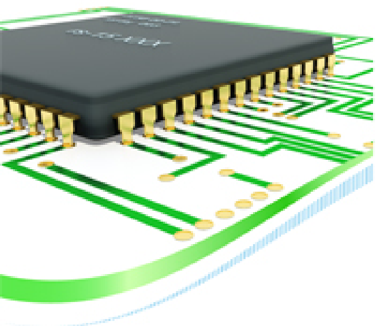
LL(1)

- 1 lookahead token
- Leftmost derivation
- Left to right scanning

LL(1) Parsers

- Use standard skeleton procedure
- Use parse tables
- Can be generated by parser generators

LL(1) Parsing



The leftmost, top-down parser

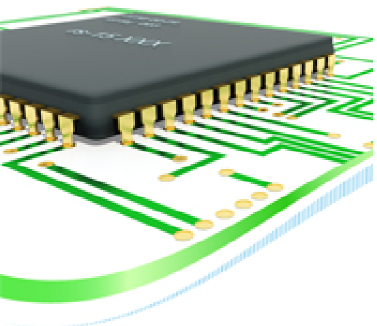
(non-predictive)

root $\leftarrow$ node for the start symbol, S;
focus $\leftarrow$ root;
push(null);
word $\leftarrow$ NextWord( );

while (true) do;
    if (focus is a nonterminal) then begin;
        pick next rule to expand focus ($A \rightarrow \beta_1, \beta_2, \ldots, \beta_n$);
        build nodes for $\beta_1, \beta_2 \ldots \beta_n$ as children of focus;
        push($\beta_n, \beta_{n-1}, \ldots, \beta_2$);
        focus $\leftarrow \beta_1$;
    end;
    else if (word matches focus) then begin;
        word $\leftarrow$ NextWord( );
        focus $\leftarrow$ pop( )
    end;
    else if (word = eof and focus = null)
        then *accept the input and return root*;
        else backtrack;
end;

Sample pseudo-code from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

LL(1) Parsing



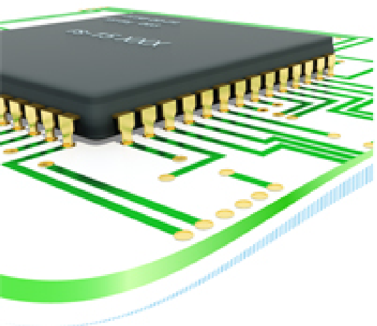
LL(1) Parser

Predictive

No backtracking

```
word ← NextWord( );
push eof onto Stack;
push the start symbol, S, onto Stack;
focus ← top of Stack;
loop forever;
   if (focus = eof and word = eof)
       then report success and exit the loop;
   else if (focus ∈ T or focus = eof) then begin;
      if focus matches word then begin;
         pop Stack;
         word ← NextWord( );
      end;
      else report an error looking for symbol at top of stack;
   end;
   else begin; /* focus is a nonterminal */
      if Table[focus,word] is A → B1 B2 ··· Bk then begin;
         pop Stack;
         for i ← k to 1 by -1 do;
           if (Bi ≠ ε)
              then push Bi onto Stack;
         end;
      end;
      else report an error expanding focus;
    end;
    focus ← top of Stack;
end;
```

Sample pseudo-code from “Cooper, K.D., Torczon, L.; Engineering A Compiler”



LL(1) Parsing

Parse Table

LL(1) Parser

Predictive

No backtracking

0	*Goal*	→	*Expr*
1	*Expr*	→	*Term Expr′*
2	*Expr′*	→	+ *Term Expr′*
3		|	- *Term Expr′*
4		|	ϵ
5	*Term*	→	*Factor Term′*
6	*Term′*	→	x *Factor Term′*
7		|	÷ *Factor Term′*
8		|	ϵ
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

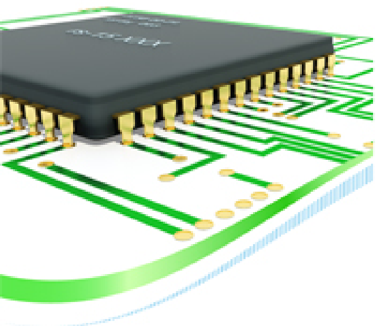
Use First+ to create the parse table!

	eof	+	-	×	÷	(	)	name	num
Goal	—	—	—	—	—	0	—	0	0
Expr	—	—	—	—	—	1	—	1	1
Expr′	4	2	3	—	—	—	4	—	—
Term	—	—	—	—	—	5	—	5	5
Term′	8	8	8	6	7	—	8	—	—
Factor	—	—	—	—	—	9	—	11	10

Grammars and table from “Cooper, K.D., Torczon, L.; Engineering A Compiler”

Bottom Up Parsing

LR Grammars



LR(n)

- n lookahead token
- Rightmost derivation
- Left to right scanning

Bottom-Up Parsing

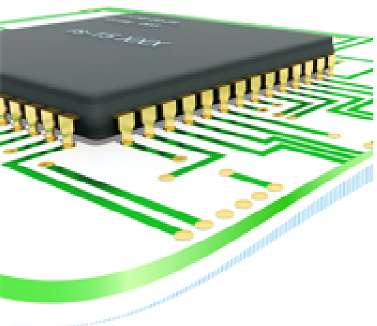
Begins with the leaves and grows the tree toward the root.

LR Grammars

- Shift Reduce Parsing
- LR(0) Automaton
- Using Follow: SLR Parsing
- Using Lookahead: LR(1) Canonical Items
- Saving Resources: LALR

Bottom Up Parsing

Shift Reduce Parsing



Grammar:

1. P $\rightarrow$ E
2. E $\rightarrow$ E + T
3. E $\rightarrow$ T
4. T $\rightarrow$ id (E)
5. T $\rightarrow$ id

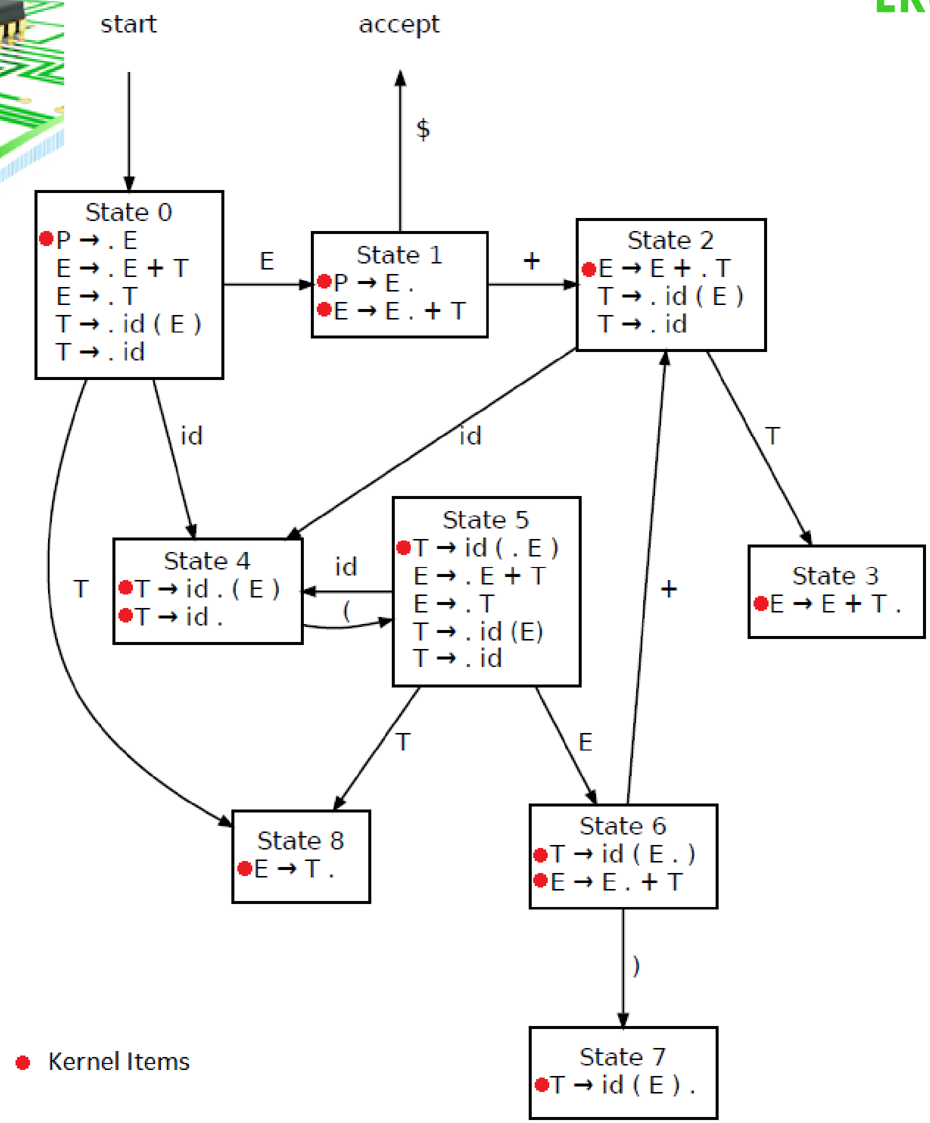
Input: id(id+id)

Stack	Input	Action
	id (id + id) $	shift
id	(id + id) $	shift
id (	id + id) $	shift
id (id	+ id) $	reduce T $\rightarrow$ id
id (T	+ id) $	reduce E $\rightarrow$ T
id (E	+ id) $	shift
id (E +	id) $	shift
id (E + id	) $	reduce T $\rightarrow$ id
id (E + T	) $	reduce E $\rightarrow$ E + T
id (E	) $	shift
id (E)	$	reduce T $\rightarrow$ id(E)
T	$	reduce E $\rightarrow$ T
E	$	reduce P $\rightarrow$ E
P	$	accept

Grammar and trace from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

LR(0) Items / Automaton



Position Notation

Dot augmentation

Kernel Items

Ex: P → .E

Closure

Ex:

State 0
P → . E
E → . E + T E → . T T → . id (E) T → . id

Automaton from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 4”, http://compilerbook.org

Bottom Up Parsing

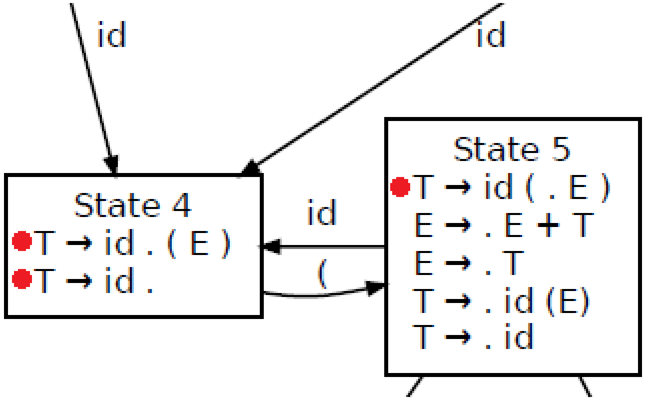
Conflicts

Shift - Reduce Conflicts

From the example grammar (State 4)

T $\rightarrow$ id . E

T $\rightarrow$ id .



Reduce – Reduce Conflicts

Ex: If we had followings on a state

S $\rightarrow$ id (E) .

E $\rightarrow$ id (E) .

Examples from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

SLR Parsing

Prediction Using Follow Sets

State	GOTO E	GOTO T	ACTION id	ACTION (	ACTION)	ACTION +	ACTION $
0	G1	G8	S4				
1						S2	R1
2		G3	S4				
3					R2	R2	R2
4				S5	R5	R5	R5
5	G6	G8	S4				
6					S7	S2	
7					R4	R4	R4
8					R3	R3	R3

Grammar:

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

Table and grammar from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 4”, http://compilerbook.org

Bottom Up Parsing

SLR Parsing

SLR Parsing Algorithm.

Let S be a stack of LR(0) automaton states. Push S_0 onto S.
Let a be the first input token.

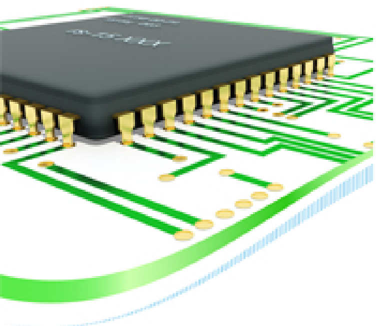
Loop:

- Let s be the top of the stack.
- If ACTION[s, a] is **accept**:
  - Parse complete.
- Else if ACTION[s, a] is **shift** t:
  - Push state t on the stack.
  - Let a be the next input token.
- Else if ACTION[s, a] is **reduce** A $\rightarrow \beta$:
  - Pop states corresponding to β from the stack.
  - Let t be the top of stack.
  - Push GOTO[t, A] onto the stack.
- Otherwise:
  - Halt with a parse error.

Algorithm from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

SLR Parsing



Grammar:

1. P → E
2. E → E + T
3. E → T
4. T → id (E)
5. T → id

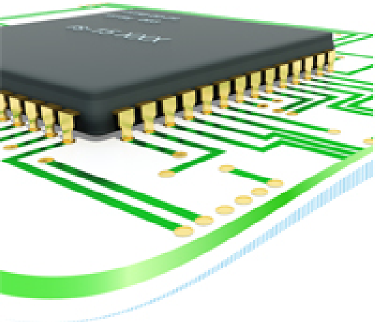
Input: id(id+id)

Stack	Symbols	Input	Action
0		id (id + id) $	shift 4
0 4	id	(id + id) $	shift 5
0 4 5	id (	id + id) $	shift 4
0 4 5 4	id (id	+ id) $	reduce T → id
0 4 5 8	id (T	+ id) $	reduce E → T
0 4 5 6	id (E	+ id) $	shift 2
0 4 5 6 2	id (E +	id) $	shift 4
0 4 5 6 2 4	id (E + id	) $	reduce T → id
0 4 5 6 2 3	id (E + T	) $	reduce E → E + T
0 4 5 6	id (E	) $	shift 7
0 4 5 6 7	id (E)	$	reduce T → id(E)
0 8	T	$	reduce E → T
0 1	E	$	accept

Execution trace from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

LR(1) Items

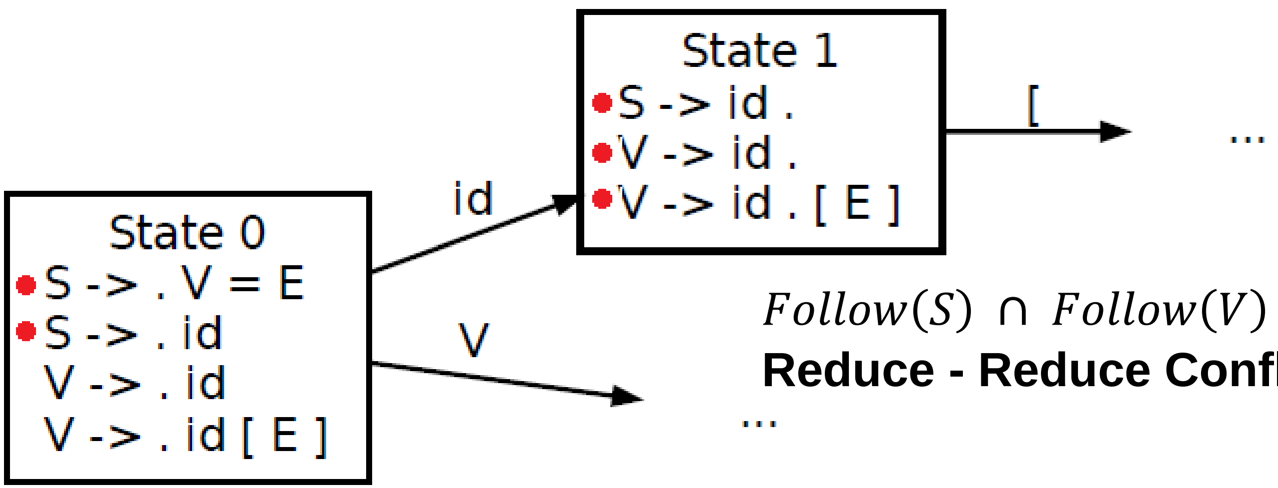


SLR fails when follow sets are not distinct on the same state

Grammar:

1. S → V = E
2. S → id
3. V → id
4. V → id (E)
5. E → V

	First	**Follow**
S	id	$
V	id	=$)
E	Id	$)



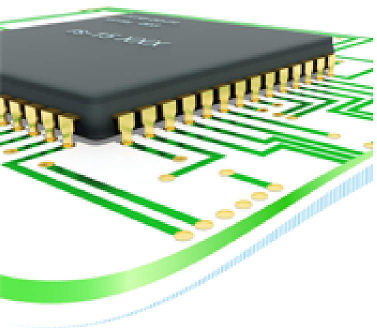
$Follow(S) \cap Follow(V) = \{\$\}$

Reduce - Reduce Conflict

Grammar and diagram from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

LR(1) Items



Canonical Form

Like LR(0), Augmented by Lookahead

Grammar:

1. $S \rightarrow V = E$
2. $S \rightarrow id$
3. $V \rightarrow id$
4. $V \rightarrow id\ (\ E\ )$
5. $E \rightarrow V$

Writing Lookahead for Each Item in State

In case $A \rightarrow\ \alpha\ .\ B$, {L} create $B \rightarrow .\ \gamma$, {L}

In case $A \rightarrow\ \alpha\ .\ B\ \beta$, {L}

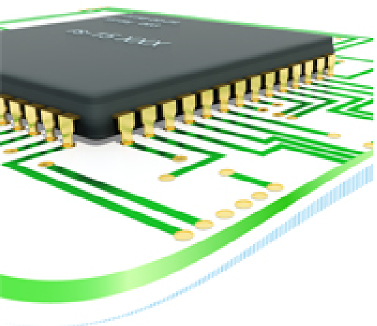
create $B \rightarrow .\ \gamma$, First(β) if β cannot be ε

create $B \rightarrow .\ \gamma$, First(β) U {L} if β can be ε

Grammar from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 4”**,** http://compilerbook.org

Bottom Up Parsing

LR(1) Items



Grammar:

1. P $\rightarrow$ E
2. E $\rightarrow$ E + T
3. E $\rightarrow$ T
4. T $\rightarrow$ id (E)
5. T $\rightarrow$ id

Example Case for P (State 0)

As kernel item add (1) P $\rightarrow$. E, {$}

Using (1) add (2) E $\rightarrow$. E + T, {$}

Using (1) add (3) E $\rightarrow$. T, {$}

Using (2) add (4) E $\rightarrow$. E + T, {+}

Using (3) add (5) T $\rightarrow$. id (E), {$}

Using (3) add (6) T $\rightarrow$. id, {$}

Using (4) add (7) E $\rightarrow$. T, {+}

No new item using (5), (6)

Using (7) add (8) T $\rightarrow$. id (E), {+}

Using (7) add (9) T $\rightarrow$. id, {+}

No new items using (8), (9)

Merge the items having common LR(0) items.

Closure

P $\rightarrow$. E, {$}
E $\rightarrow$. E + T, {$ +}
E $\rightarrow$. T, {$ +}
T $\rightarrow$. id (E), {$+}
T $\rightarrow$. id, {$ +}

Grammar and diagram from "Douglas Thain, Introduction to Compilers and Language Design, Chapter 4", http://compilerbook.org

Bottom Up Parsing

LR(1) Automaton

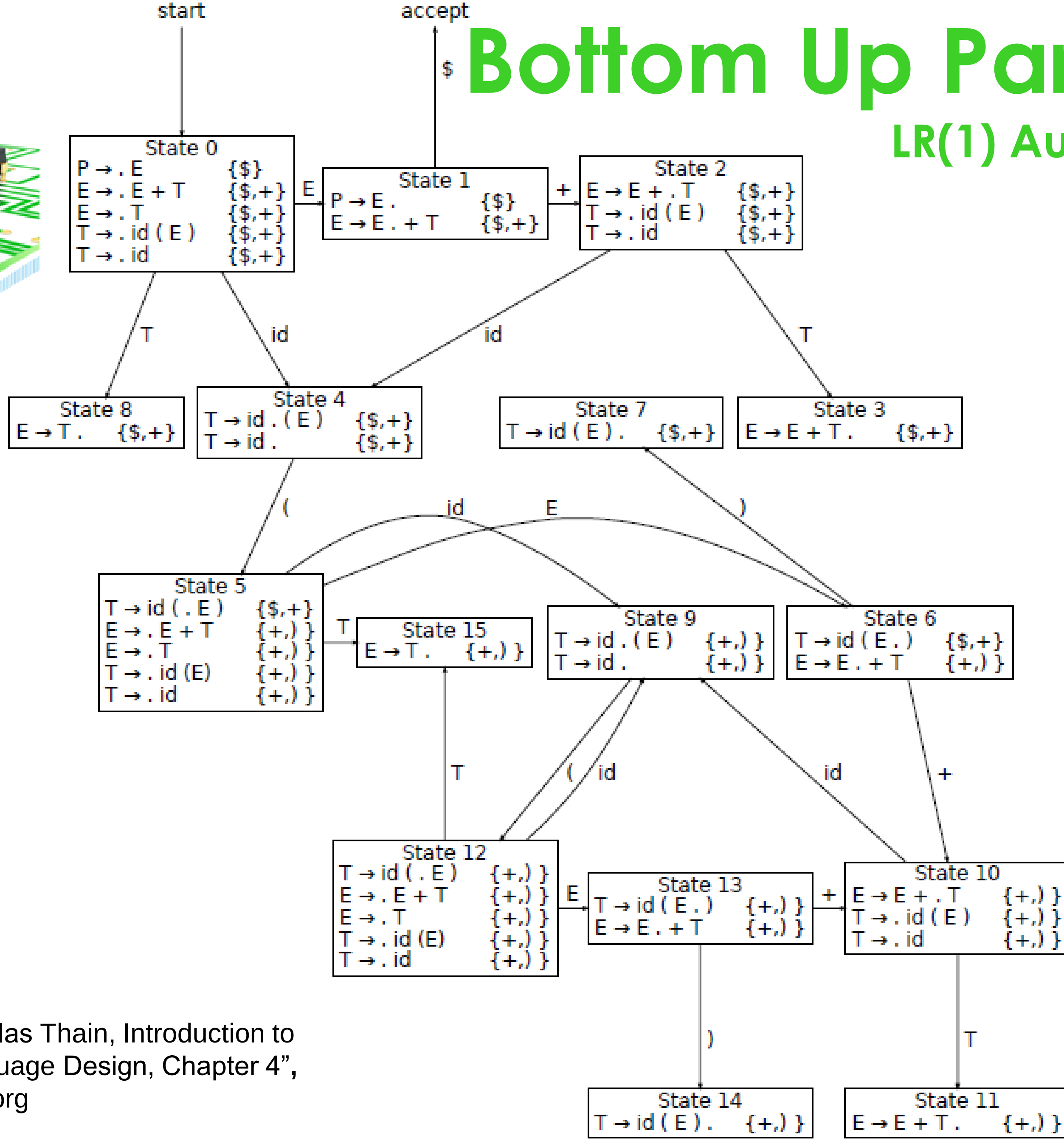
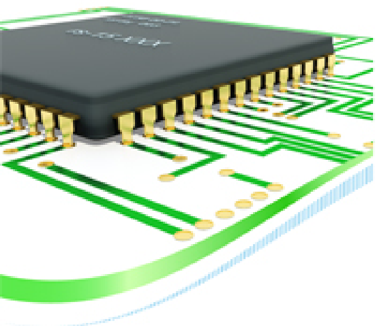


Diagram from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 4”, http://compilerbook.org



Bottom Up Parsing

LALR Parsing

Merge States Having Common Cores

Identify the States Having Common Cores

Write a New State with LR(0) items and Union Lookaheads

Example: Not from the Previous Diagram!

$E \rightarrow .\ E + T$	$\{\$+\}$
$E \rightarrow .\ T$	$\{\$+\}$

$E \rightarrow .\ E + T$	$\{)+\}$
$E \rightarrow .\ T$	$\{)+\}$

$\longrightarrow$

$E \rightarrow .\ E + T$	$\{\$)+\}$
$E \rightarrow .\ T$	$\{\$)+\}$

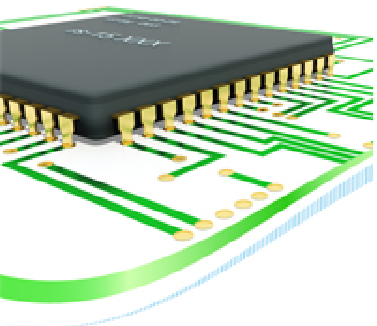
Identify the states that can be merged in the previous diagram!

Reconstruct the diagram!

Note on Grammars:

$$LL(1) \subset SLR \subset LALR \subset LR(1) \subset CFG$$

Example and comparison from “Douglas Thain, Introduction to Compilers and Language Design, Chapter 4”**,** http://compilerbook.org



Final Words

Other Parsing Strategies

- LL(*)
- Parse Expression Grammars (PEG)
- …more

“BOTTOM LINE: We do all this math to ensure that the source code is translated to internal representation unambiguously, predictably.” (*)

(*) Remark from Cem Bozşahin, notes on Syntactic Analysis