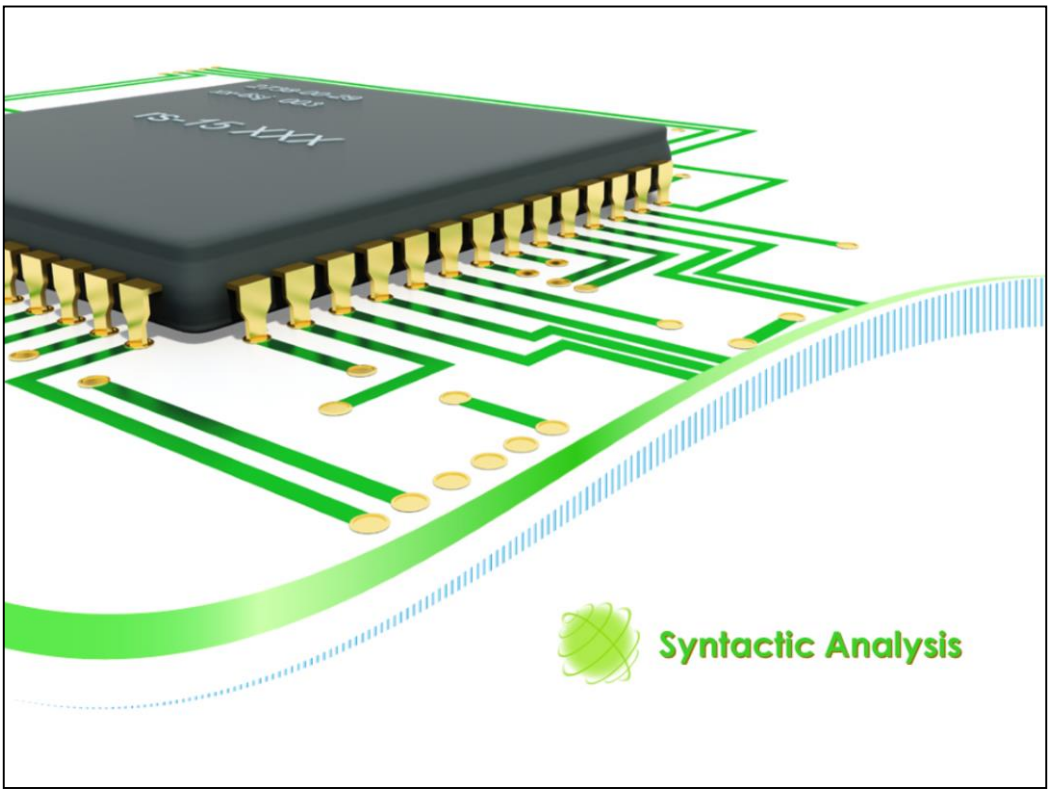
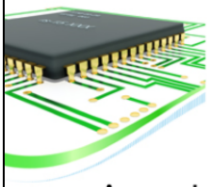

Syntactic Analysis



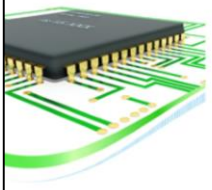
Definition

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.

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.

Typical implementations receive the sequence of the tokens generated by the lexical analyzer. In such cases, the sequence of tokens is fixed for a given input. There may be other approaches, especially in parsing strategies using top-down processing, where the tokens are recognized as the parsing proceeds. For such cases, back-tracking and alternative sequence of tokens may be observed.



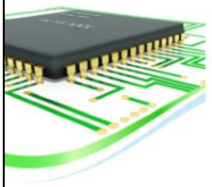
Syntactic Validity

Yes, valid program is a sentence.

Usually, validity transcends syntax.

- **Declarative Integrity**
- **Parameter Matching**
- **Type Conformance**
- **Operator Applicability**
- **Statement Applicability**
- ...

Thinking of our experiences with the programming languages which we are accustomed to, event without consulting to the programming language definitions we can safely state that syntactic compliance is necessary to validate a program but not enough. Validity of a given program is governed by the rules that transcends the grammatical constraints. Can you give examples of context dependent properties of a programming language that you know?



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)**

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

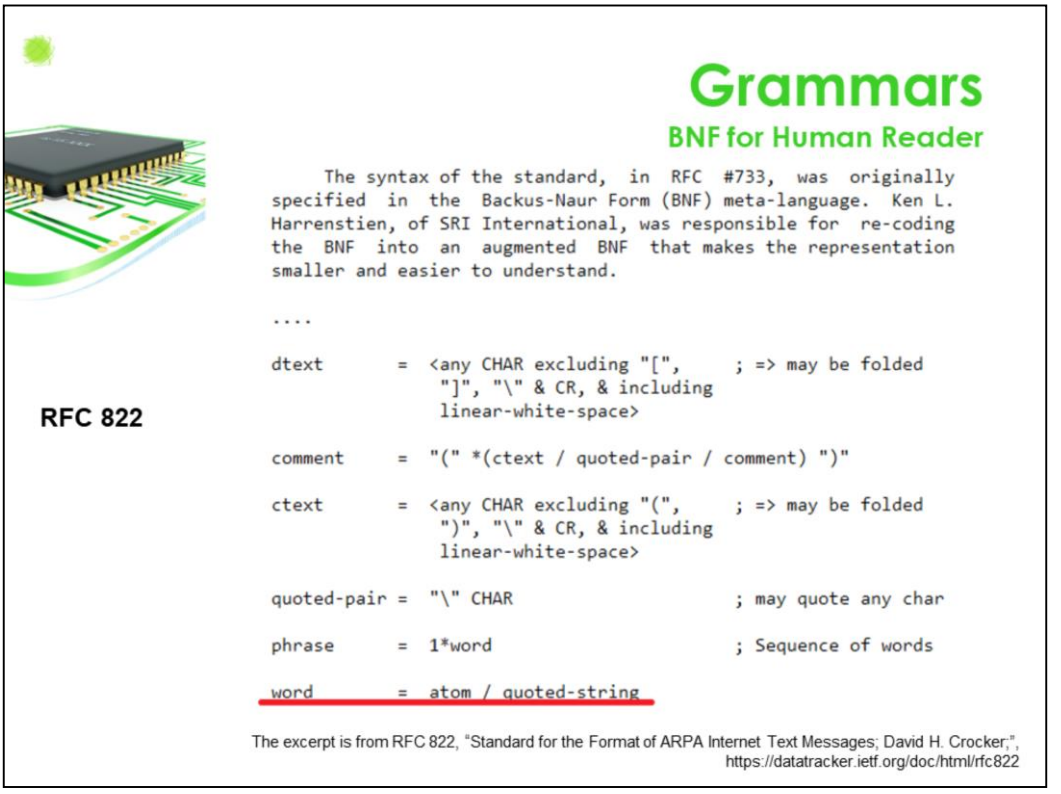
It is a common practice to use formal notations to describe the syntactic rules of languages for a bunch of reasons such as precise-expressive specification, ease of maintenance and improvement, automatic generation recognizers, and similar. For most of the languages that need a parser, the syntactic specification has a property of recursion which is not supported by the type-3 languages i.e. regular expressions.

Any BNF like notation defines common constructs as noted below.

Non-terminals: These are the symbols for the atomic components of the syntax definition reported by the lexical analyzer.

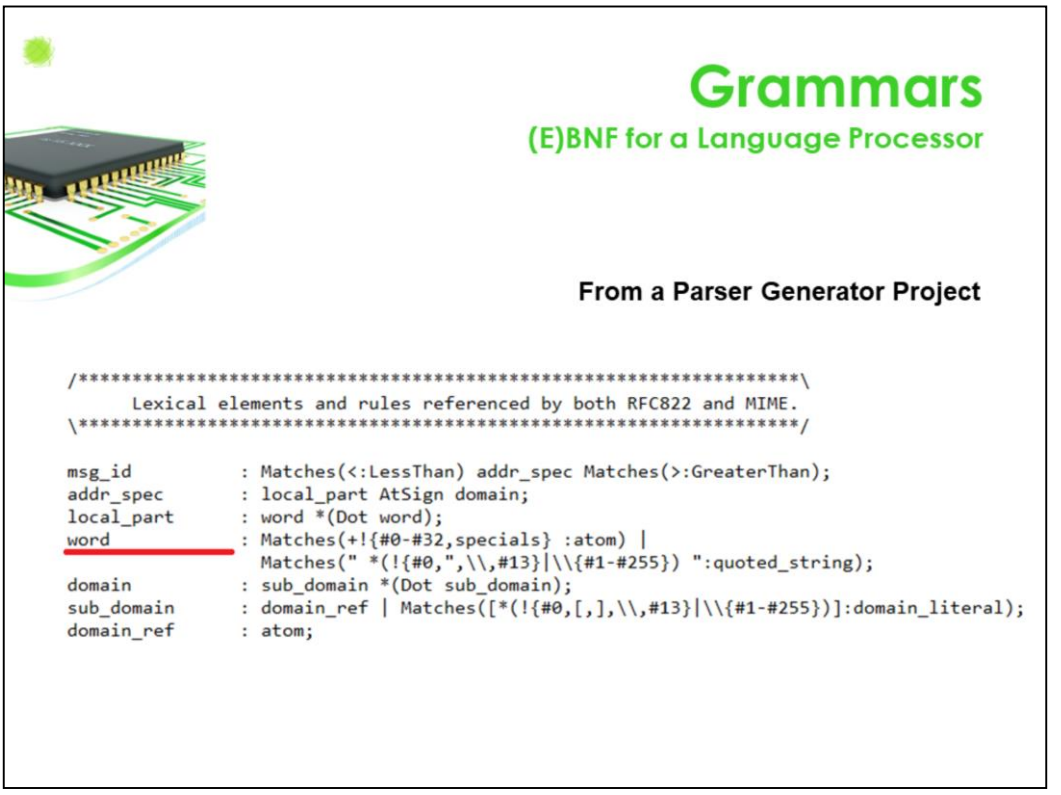
Terminals: These are the symbols making reference to the derivation rules. This references allow the designer to define recursion / cycles in a controlled manner.

Rules: These are the constructs combining the symbols with simple operators such as alternation, Kleene star and derivatives, and helper elements such as Epsilon, Parentheses, and similar. In some contexts, Kleene star and its derivatives are avoided for sake of simplicity as they can be mapped to expressions with help of interim symbols, alternation and Epsilon without loss of expressivity.

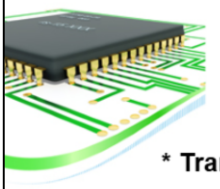


A common notation to specify the intended syntax is known as Backus-Naur Form (BNF), which dates back to the 1960s. BNF and similar notations are widely applied to define the grammar of the intended language as type-2 grammars which are called as context free grammars (CFGs). The notations used in books or standard documents (RFC's for example) have a certain level of flexibility as their main concern is to describe the grammar for human readers. Automatic parser generators on the other hand must be strict in their meta-language (notation) definition, which is similar to the BNF notation at some level. Each project defines its own way of building declarations.

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



The excerpt is from a proprietary parser definition language.



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

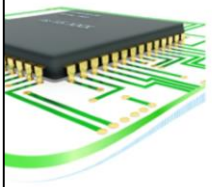
u e? w → u e' w
e' : e | ε

Parentheses

u (e) w → u e' w
e' : e

Alternative Notations

[e], {e}, prefix, postfix, etc…



Derivation

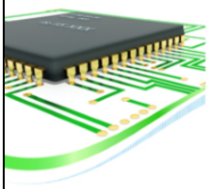
Definition

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"

If uAv is a sentential form in derivation of string and production rule A exists such that A->w, then uAv -> uwv . A Î V, and u, v, w Î S*, where S* is the set of the sentential forms allowed by the R.

Basically, the choice of A in a sentential form can be made arbitrarily. But, there are two basic ways of making this choice that we are interested in. The first way is choosing the leftmost non-terminal, the second way is choosing the rightmost non-terminal. If a derivation chooses the leftmost non-terminals only, it is called leftmost derivation (LMD). Similarly, if a derivation chooses the rightmost non-terminals only, it is called rightmost derivation (LMD). A sentential form can be defined alternatively as an intermediate form that is reachable from S. So, a left sentential form is a sentential form that exists as a step in LMD; a right sentential form is a sentential form that exists as a step in RMD.



Sentential Form

Definition

Sentential form is a string of symbols that occurs as a step 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.**

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

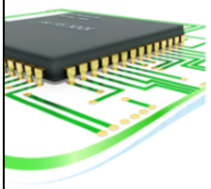
$$f_n = uAv \qquad A \in V \qquad A \rightarrow w \in R \qquad f_{n+1} = uwv \qquad uAv, uwv \in S^*$$

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

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

If uAv is a sentential form in derivation of string and production rule A exists such that A->w, then uAv -> uwv . A Î V, and u, v, w Î S*, where S* is the set of the sentential forms allowed by the R.

Basically, the choice of A in a sentential form can be made arbitrarily.



Derivation

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

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

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

$\alpha \Rightarrow^* \beta$ and $\beta \Rightarrow^* \gamma$ then $\alpha \Rightarrow^* \gamma$

Example: Input: aaba

G_1 :

$S \rightarrow aA \mid abC$

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

$C \rightarrow c$

G_1 :

$S \rightarrow aA$

$S \rightarrow abC$

$A \rightarrow aA$

$A \rightarrow bA$

$A \rightarrow C$

$A \rightarrow \varepsilon$

$C \rightarrow c$

S by S-> aA

aA by A-> aA

aaA by A -> bA

aabA -> aA

aaba This is a sentence. This is rightmost derivation by the way.

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

But, there are two basic ways of making this choice that we are interested in. The first way is choosing the leftmost non-terminal, the second way is choosing the rightmost non-terminal. If a derivation chooses the leftmost non-terminals only, it is called leftmost derivation (LMD). Similarly, if a derivation chooses the rightmost non-terminals only, it is called rightmost derivation (LMD). A sentential form can be defined alternatively as an intermediate form that is reachable from S. So, a left sentential form is a sentential form that exists as a step in LMD; a right sentential form is a sentential form that exists as a step in RMD.

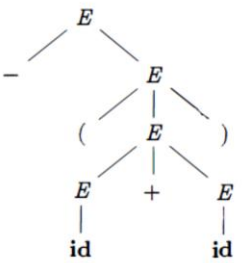
Parse Trees

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"

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

Ambiguity

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

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

E → E **+** E
E → E * E
E → **-** E
E → **(** E **)**
E → **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"

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

A well-known ambiguity problem is the dangling-else problem!

Ambiguity
Elimination

Rewriting

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

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

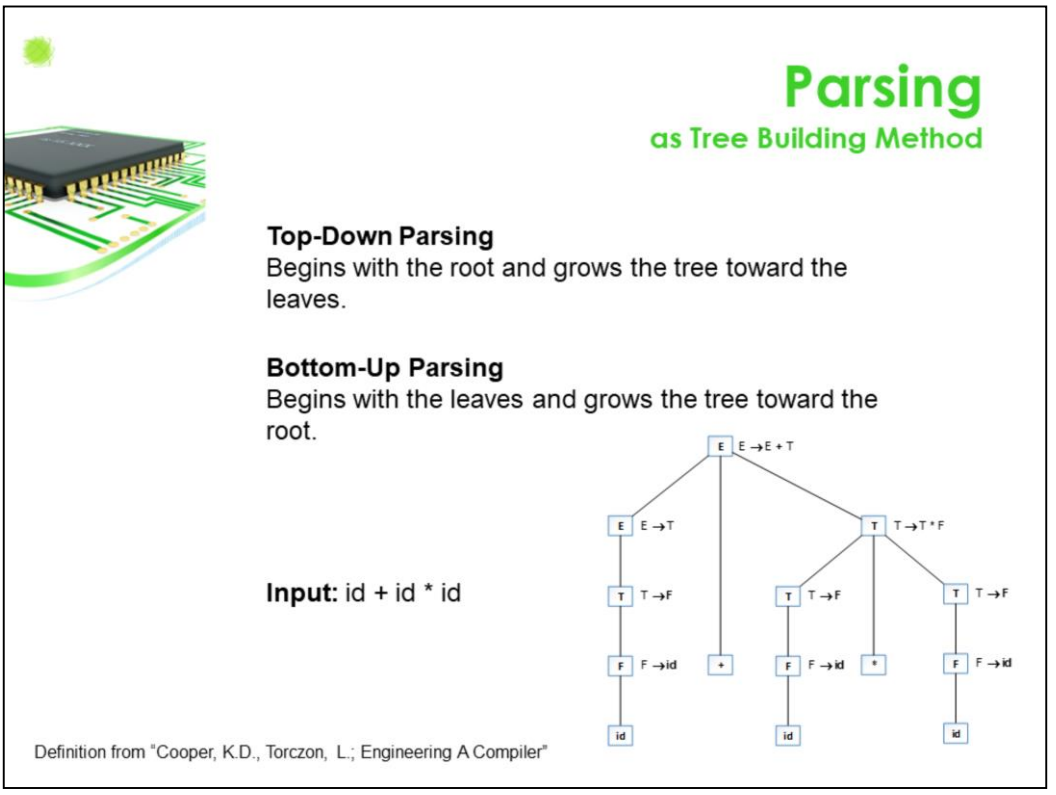
G:
E → E **+** T | T
T → T * F | F
F → **-** E | **(** E **)** | **id**

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

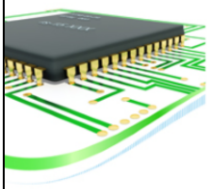
Definition and example from "Aho, A.V, Ullman J.D, Sethi R., Lam M.S; Dragon Book"

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



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

Create the parse tree top-down and bottom-up fashion.



Top Down Parsing

The Left Recursion Problem

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**

Input: id + id * 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"

A top down parser establishes the parse tree starting from the root and tries each production of the variable by expanding it as a child.

Use leftmost derivations to create the parse tree. When you apply the rules in the order they were declared, you cannot avoid endless recursion.

Left Recursion

Elimination

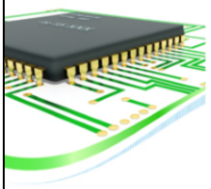
When Grammar has no Cycles and no ε-productions

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 $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ 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"

The algorithm given above does not guarantee when the preconditions are not met.



Left Recursion

Elimination

General Elimination Algorithm

G:

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

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

Input: id + id * id

G:

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

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

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

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

When the algorithm applied to the grammar G, it is transformed to a more complicated form. But now, it is recursion free.

Use leftmost derivations again to create the parse tree.

Backtracking

The leftmost, top-down parser

```
root ← node for the start symbol, S;
focus ← root;
push(null);
word ← NextWord( );

while (true) do;
   if (focus is a nonterminal) then begin;
      pick next rule to expand focus (A → β1,β2,...,βn);
      build nodes for β1,β2...βn as children of focus;
      push(βn, βn−1, ..., β2);
      focus ← β1;
   end;
   else if (word matches focus) then begin;
      word ← NextWord( );
      focus ← 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) ← ∅ ;
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:

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

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

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

$F \rightarrow - E$
$F \rightarrow (E)$
$F \rightarrow id$

Build the first sets!

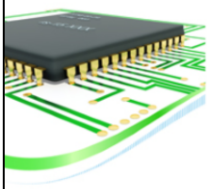
Build the first sets.

Follow Set

Construction

```
for each A ∈ NT do;
    FOLLOW(A) ← ∅;
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**

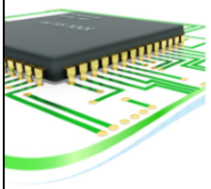
E → T E'
E' → +T E'
E' → ε

T → F T'
T' → * F T'
T' → ε

F → - E
F → (E)
F → **id**

Build the follow sets!

Build the first sets.



Backtracking

Backtrack-free Test

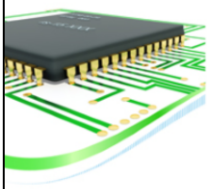
Test each non-terminal with multiple RHS!

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

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

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

To test if a grammar is backtrack-free any non-terminal having more than one RHS (alternatives) must be tested. For a backtrack-free grammar, the First+ sets generated for alternatives are distinct.



Left Factoring

G:

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

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

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

$F \rightarrow - E$
$F \rightarrow (E)$
$F \rightarrow id$
$F \rightarrow id (E)$
$F \rightarrow id [E]$

When array and function call like structures added!

G:

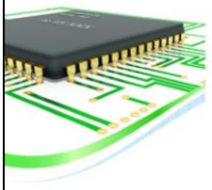
$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow - E \mid (E) \mid id$

. . .

$F \rightarrow - E$
$F \rightarrow (E)$
$F \rightarrow id\ G$

$G \rightarrow (E)$
$G \rightarrow [E]$
$G \rightarrow \varepsilon$

When we add two more productions for F to create a flavor of function call and array access backtracking test for F fails! Rewriting with left factoring helps eliminate common First+ sets.



Predictive Parsers

Stages of Test & Elimination

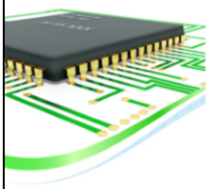
- Ambiguity
- Left Recursion
- Backtracking
- Left Factoring

Can you revise the leftmost, top-down parser?

Now, we can focus on

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

When we add two more productions for F to create a flavor of function call and array access backtracking test for F fails! Rewriting with left factoring helps eliminate common First+ sets.



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"

Recursive descent parsing is one way of implementing top down parsing strategy. It is easy to construct and follow. It can be built by hand or automatically. A standard debugger becomes a tool to diagnose and fix parsing related problems!

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

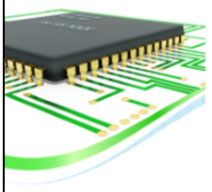
(non-predictive)

```
root ← node for the start symbol, S;
focus ← root;
push(null);
word ← NextWord( );

while (true) do;
   if (focus is a nonterminal) then begin;
      pick next rule to expand focus (A → β1,β2,....,βn);
      build nodes for β1,β2...βn as children of focus;
      push(βn,βn−1,....,β2);
      focus ← β1;
   end;
   else if (word matches focus) then begin;
      word ← NextWord( );
      focus ← 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



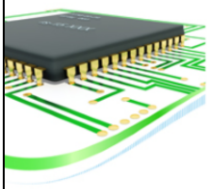
LL(1) Parser

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 → B1B2···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

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

LL(1) Parser

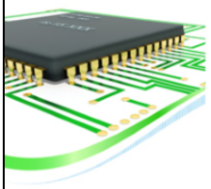
Predictive

No backtracking

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)

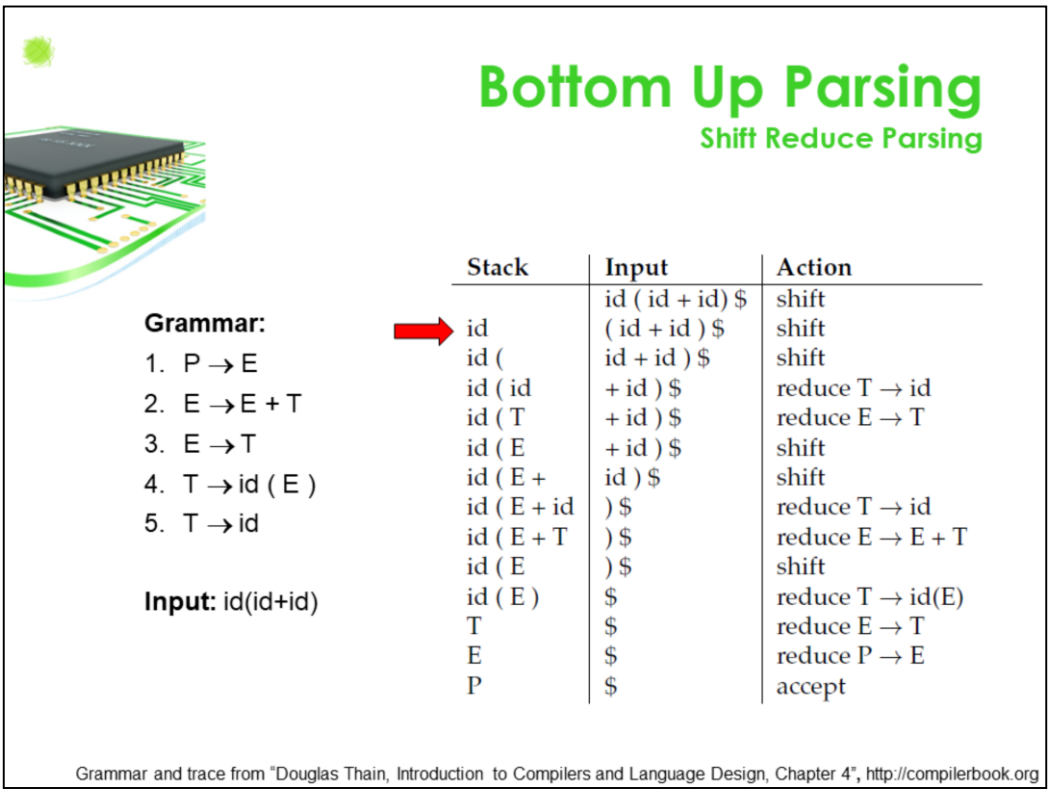
- n lookahead token
- Rightmost derivation
- Left to right scanning

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

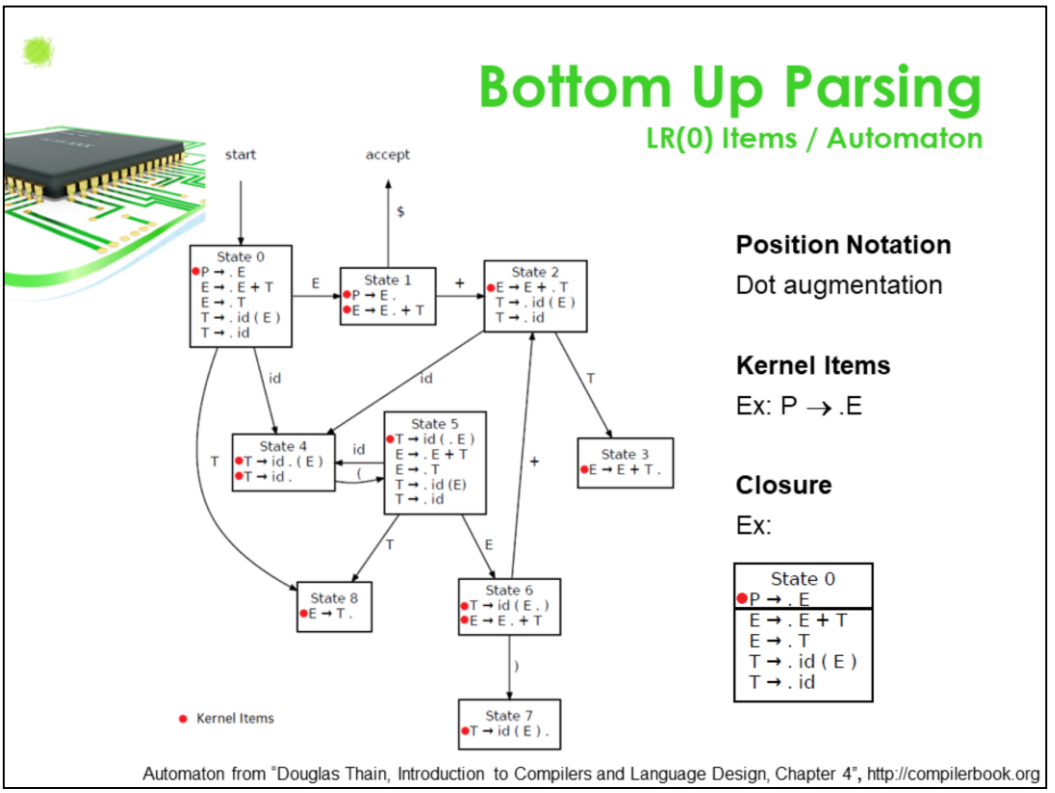
We will start with shift reduce parsing, which conveys the basic operation of the bottom up parsing.



This is a sample to illustrate the operation of the shift reduce parsing, which applies a strategy to start from the input and reaching the goal symbol by applying reductions. The sequence of operations applied is result of purposeful selection to achieve successful completion.

The second action (‘shift’) could also be (‘reduce’), which would not lead us a result. In other words, “id(“ is a viable prefix for reduction while “T(“, “E(“ are not. It is critical to keep the left side in viable prefix forms. Another relevant definition to give at this point is the “handle”, which is a substring that matches a production at certain form in right sentential form. See the Stack and Input to observe right sentential forms in reverse order. Handles are reduced (or pruned) in the process of recognition.

A strategy with backtracking would be proposed! Like we did in top-down parsing, we will seek the ways of making good choices with the clues (or the context) that can be obtained from the input stream so that predictive actions can be taken. In this regard, we will define requirements of the well-formed grammars. To start with a rule, let’s note that the grammar must not be ambiguous in which case there is more than one handle for some stages of shift reduce parsing.



The LR(0) items are the productions augmented with a dot that signifies the position on the rule. The substring on the left is the recognized part, the substring on the right are the remaining substring to be recognized.

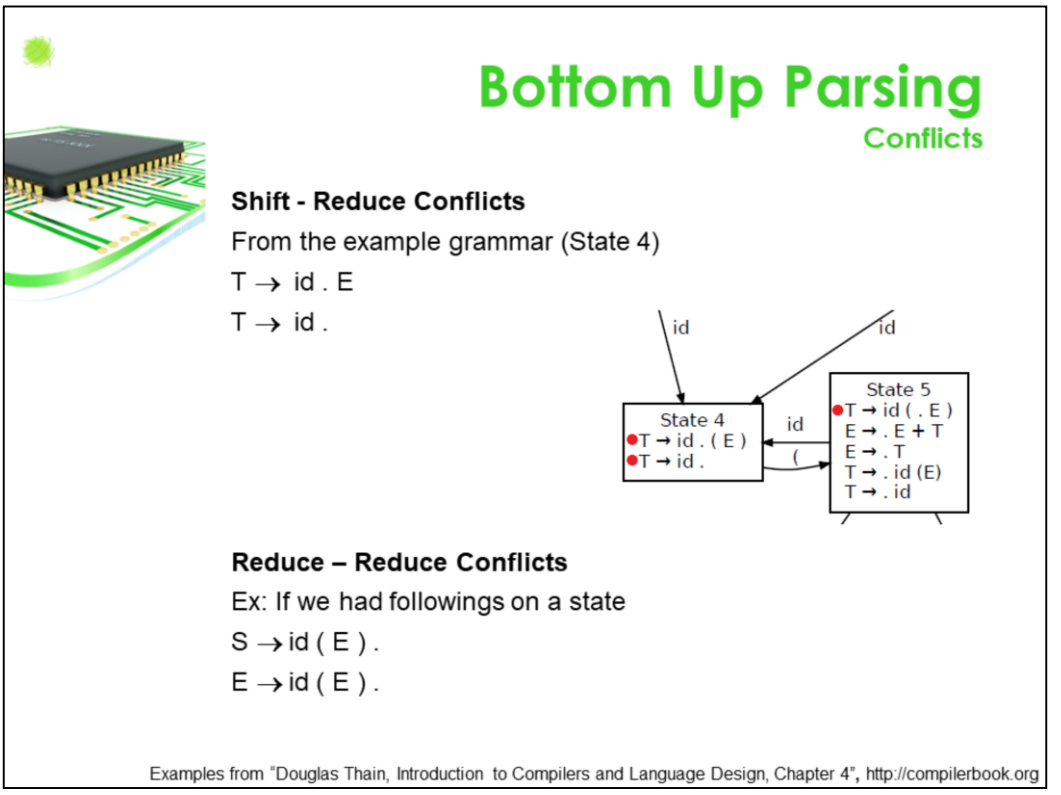
1. If there is alternatives for the starting production create a root production S' –> Σ, ανδ ασσυμε Σ□ ασ σταρτινγ προδυχτιον.

2. To create start state, create the kernel item using the starting production

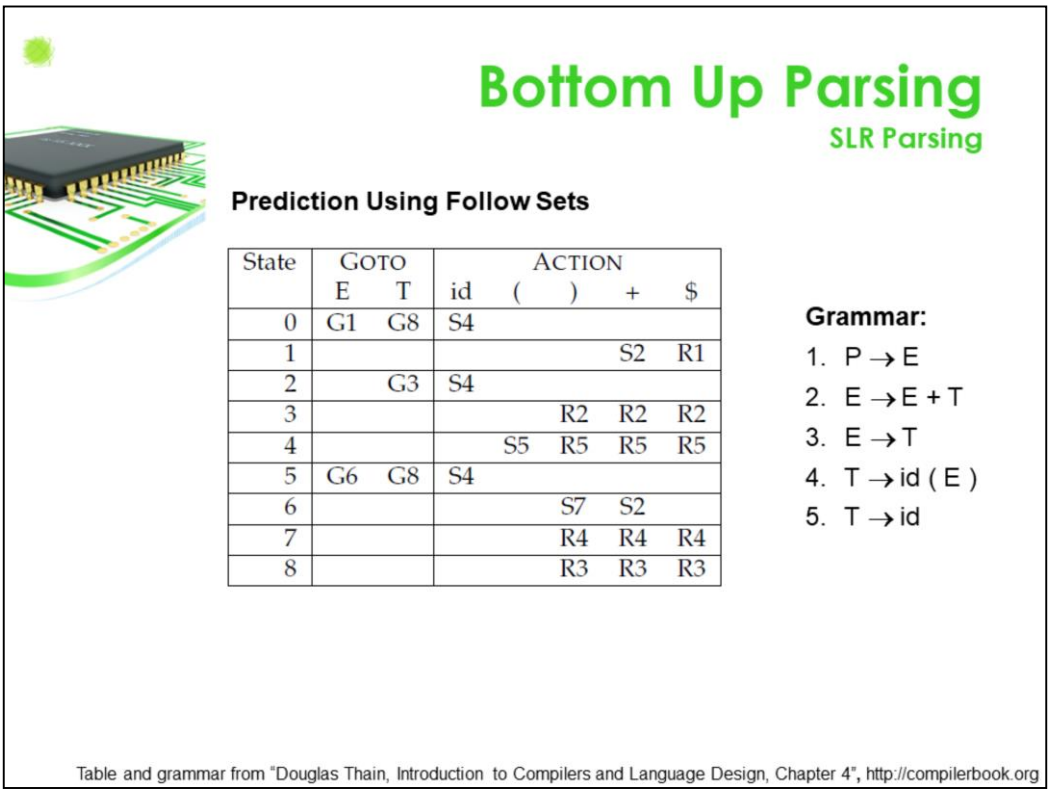
3. Choose a state to process and create closure. To create closure of a state, kernel items are the starting points. Scan items in the closure, on each item, for each non-terminal appearing on the right of dot, include the productions for that nonterminal with a dot at start. Repeat the process until the closure becomes stable .i.e. no new items added.

4. Scan items and for each item in the closure, for each w appearing on the right of dot (A->u.wv) try to find a transition and destination state on the diagram. If not found create a transition and a destination state. Create a kernel item in the form of A->uw.v .

5. Goto step 3 if there is at least one state that was not processed.



Conflicts have be dealt with by using predictions. Our first step for predictive actions is the SLR Parsing.



State	GOTO		ACTION				
	E	T	id	(	)	+	$
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

Create the First and Follow Sets.

Create S (Shift) actions for each terminal transition edge using the destination state number.

Create G (Goto) actions for each non-terminal transition edge using the destination state number.

Create R (Reduce) actions for each terminal in the follow set. Use the production number to note R.

Bottom Up Parsing

SLR Parsing

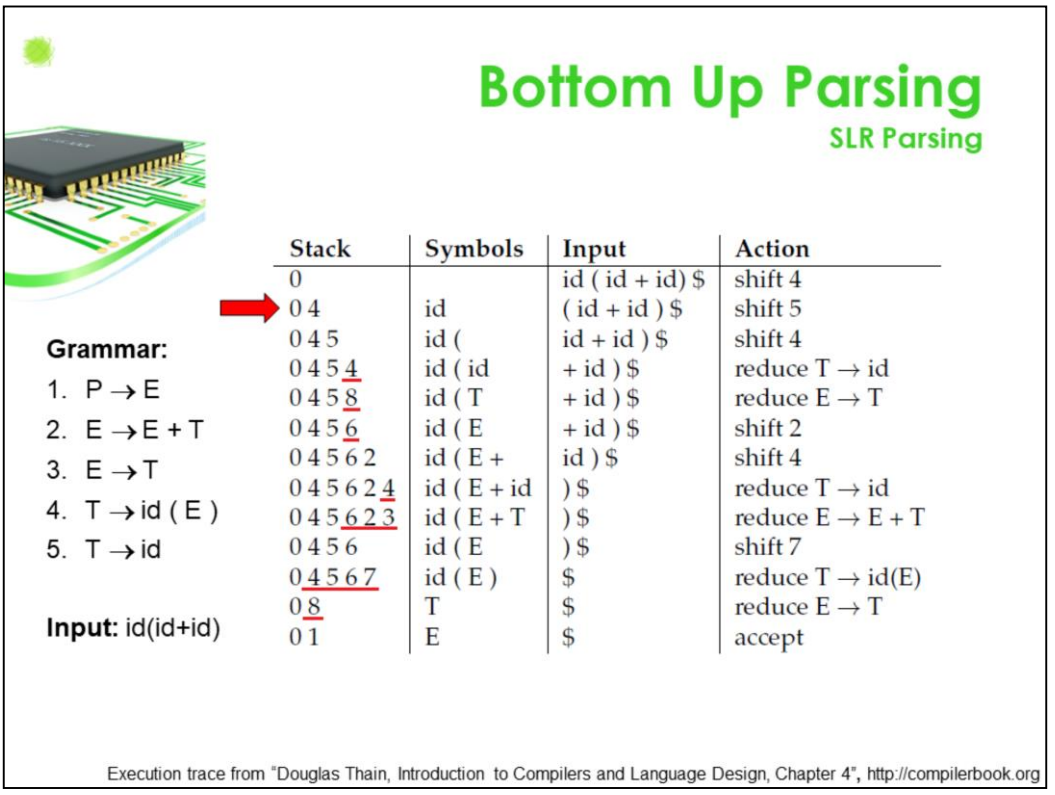
SLR Parsing Algorithm.

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

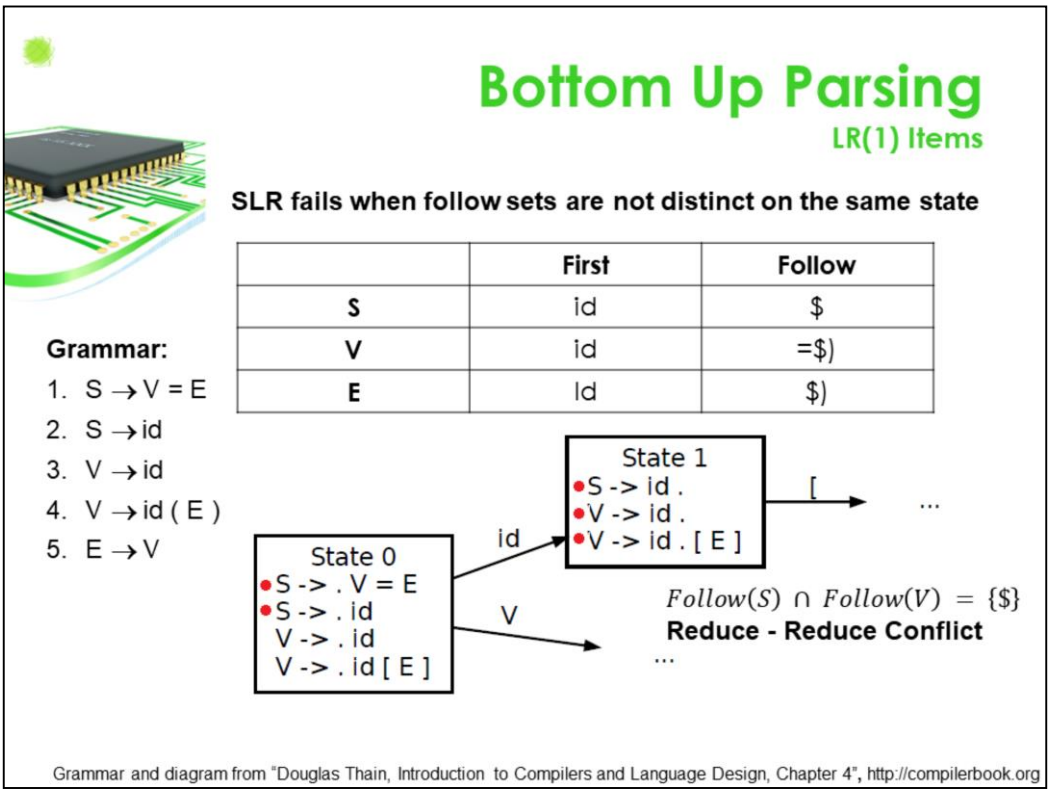
Loop:
- Let s be the top of the stack.
- If ACTION$[s, a]$ is **accept**:
  - Parse complete.
- Else if ACTION$[s, a]$ is **shift** t:
  - Push state t on the stack.
  - Let a be the next input token.
- Else if ACTION$[s, a]$ is **reduce** $A \rightarrow \beta$:
  - Pop states corresponding to β from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

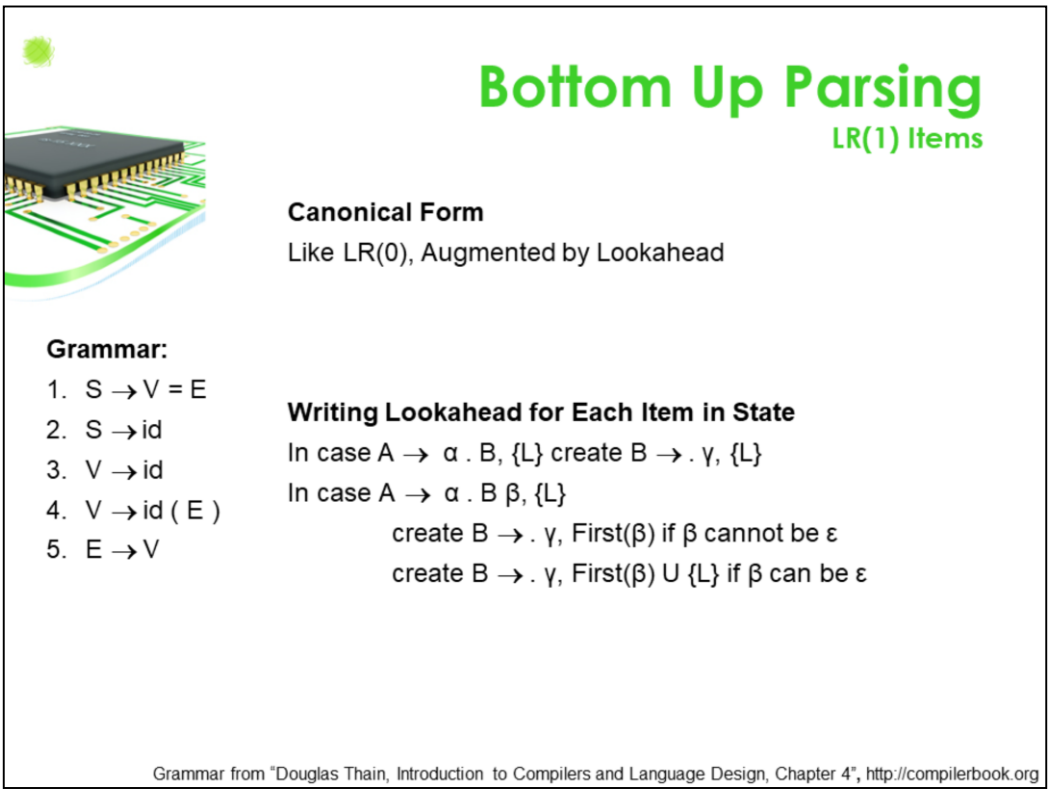
This is the SLR parsing algorithm driven by the SLR table and the input.



Use of SLR table eliminated the dilemma that occurred in the execution that we examined while introducing shift-reduce parsing. Note that, the symbols and input are displayed to make visible the sentential forms. The next token and the stack of the states are the essential parts to implement table driven shift reduce algorithm.



The state 1 has two productions with follow sets having a common element causing reduce-reduce conflict. Use of lookeaheads instead of follow may help improve the power of the bottom-up parsing.



The LR(0) items are the productions augmented with a dot that signifies the position on the rule. The substring on the left is the recognized part, the substring on the right are the remaining substring to be recognized.

1. If there is alternatives for the starting production create a root production S' –> Σ, ανδ ασσυμε Σ□ ασ σταρτινγ προδυχτιον.

2. To create start state, create the kernel item using the starting production(s) with lookahead {$}

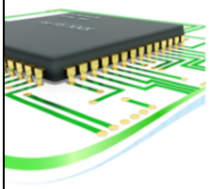
3. Choose a state to process and create closure. To create closure of a state, kernel items are the starting points. Scan items in the closure, on each item, for each non-terminal appearing on the right of dot, include the productions for that nonterminal with a dot at start and the calculated lookahead. Repeat the process until the closure becomes stable .i.e. no new items added.

4. Merge items having LR(0) items with different lookaheads by unioning lookeahead sets on the finalized closure.

5. Scan items and for each item in the closure, for each w appearing on the

right of dot (A->u.wv, {L}) try to find a transition and destination state on the diagram. If not found create a transition and a destination state. Create a kernel item in the form of A->uw.v {L} .

6. Goto step 3 if there is at least one state that was not processed.



Bottom Up Parsing

LR(1) Items

Grammar:

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

Example Case for P (State 0)

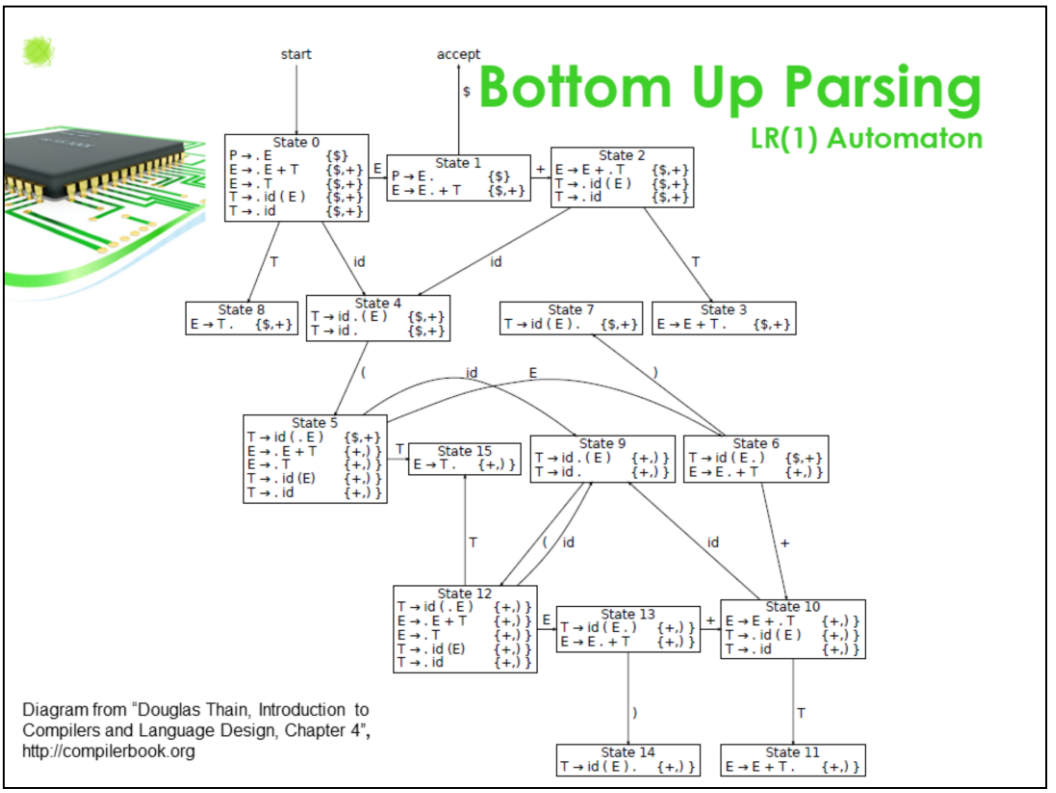
As kernel item add (1) P → . E, {$}
Using (1) add (2) E → . E + T, {$}
Using (1) add (3) E → . T, {$}
Using (2) add (4) E → . E + T, {+}
Using (3) add (5) T → . id (E), {$}
Using (3) add (6) T → . id, {$}
Using (4) add (7) E → . T, {+}
No new item using (5), (6)
Using (7) add (8) T → . id (E), {+}
Using (7) add (9) T → . id, {+}
No new items using (8), (9)
Merge the items having common LR(0) items.

Closure

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

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

This is step by step evolution of closure for the state 0.



Applying the principles similar to the LR(0) case, the LR(1) automaton is built as seen. Note that there are many states (16 in this case) for even a small piece of grammar. Developing the LR(1) states and items by hand is time consuming and it is easy to make mistakes. The parser generators like Yacc, Bison take care of handling generation of the tables and error free code.

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 → . E + T {$+}	E → . E + T {)+}	E → . E + T {$)+}
E → . T {$+}	E → . T {)+}	E → . T {$)+}

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

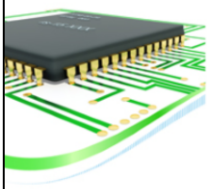
Reconstruct the diagram!

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

LALR parsers weaker than the LR(1) parsers but stronger than the SLR parsers.



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

Studying LL(1), SLR, LALR, LR(1) parsers are useful to understand the insights and the challenges in syntactic analysis. The study helps as relate the syntactic analysis with context free grammars along with the properties and limitations. Note that even syntactic analysis may transcend the boundaries of context free grammars. Even though the parsers are mostly generated with help of parser generators, manual interventions may bee needed to obtain the intended parsers. The strategies of implementing parsers are far more richer than the strategies introduced in this presentation.

https://www.antlr.org/papers/LL-star-PLDI11.pdf

https://bford.info/pub/lang/peg.pdf