Overview

30 September 2019 12:23

Course Homepage
https://bb.imperial.ac.uk/webapps/blackboard/content/listContent.jsp?course_id=_17445_1&content_id=_1582845_1

Book(s)

Aims:

To introduce students to the concepts underlying the design and implementation of language processors.

Learning Outcomes:

By the end of the course, students will be able to answer these questions:

(1) What language processors are, and what functionality do they provide to their users?

(2) What core mechanisms are used for providing such functionality?

(3) How are these mechanisms implemented?

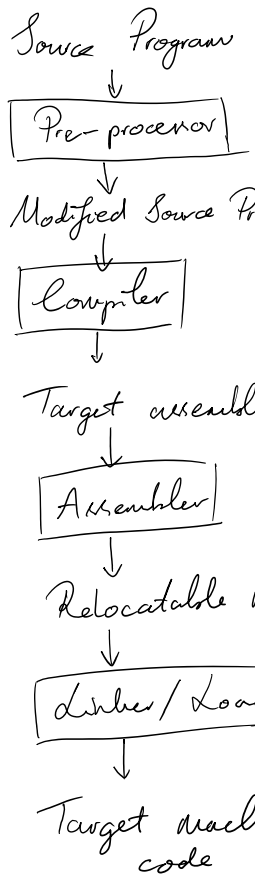
Students will also be able to construct language processors.

Syllabus:

From <http://intranet.ee.ic.ac.uk/electricalengineering/eecourses_t4/course_content.asp?c=EE2-15&s=I2#start>

Indtroduction to Compiling

03 February 2020 18:15



Import/includes the other programs/program sections so that the source program can run
May expand some shorthand's (macro)
In C programs these are starting with #

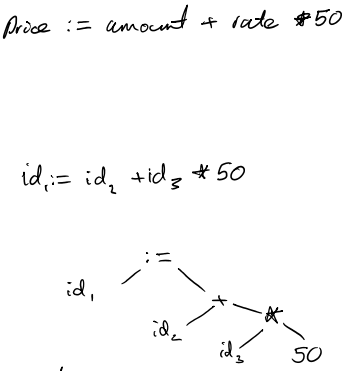
Produces program in assembly language, as easy to debug and translate later

Coverts assembly to machine code

Links each model of the program so that the program is executable and create header for the program
Loader checks available memory blocks for execution

Complier Phases

price := amount + rate *50



Coverts input into a series of individual tokens

Checks if the tokens are in logical order / grammar

Meaning error e.g. adding different types float + double

Each statement is written in 3 address code

Generation of code for the target architecture (e.g. register numbers etc)

$id_1 := id_2 + id_3 * 50$


```
temp1 := inttoreal(50)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
```

```
temp1 := id3 *50.0
id1 := id2 + temp1
```

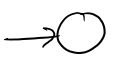
```
MOVF  R2, id3
MULF  R2, #50
MOVF  R1, id2
ADDF  R1, R2
MOVF  id, R1
```

Finite Automata and Formal Language

03 February 2020 18:17

Q - finite non-empty set of states
Σ - finite non-empty set of input alphabets
δ- the transition function which maps QXE into Q. One of the states in Q and one of the symbols or aphabets in E will map one of the states of Q.
$q_0 \in Q$ - the initial state belonging to Q
$F \leq Q$ - the set of final states

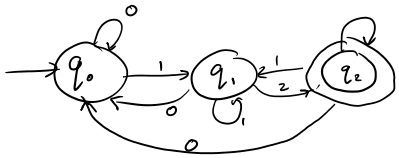
A transition system is a directed graph



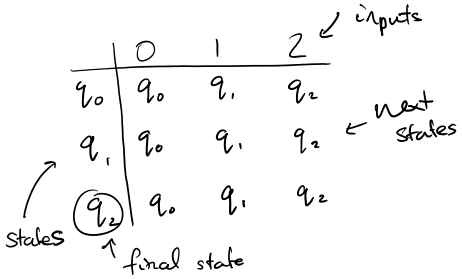
Start state

Final state

Example: Transition Diagram



Example: Transition Table



Set of states

$Q = \{q_0, q_1, q_2\}$

Set of input symbols

$\Sigma = \{0, 1, 2\}$

Final states

$F = \{q_2\} \quad F \subseteq Q$

The transition functions:

$Q \times \Sigma = 3 \times 3 = 9$ There will be 9 delta functions, we know this from the Cartesian product of amount of states in Q and sigma

$\delta(q_0, 0) = q_0 \quad \delta(q_1, 0) = q_0 \quad \delta(q_2, 0) = q_0$
$\delta(q_0, 1) = q_1 \quad \delta(q_1, 1) = q_1 \quad \delta(q_2, 1) = q_1$
$\delta(q_0, 2) = q_2 \quad \delta(q_1, 2) = q_2 \quad \delta(q_2, 2) = q_2$

Acceptability of a string by DFA / NFA

DFA:
The string $x \in \Sigma^*$ is accepted by a Finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ if $\delta(q_0, x) = q$ for some $q \in F$

Properties of Transition Functions
Property 1 :

$\delta(q, \Lambda) = q$

Property 2:
For $\omega \in \Sigma^*$ for a :

$\delta(q, a\omega) = \delta(\delta(q, a), \omega)$
$\delta(q, \omega a) = \delta(\delta(q, \omega), a)$

NFA:
NFA is the same as DFA with one exception:
Delta transition function maps from QxE into 2^Q . This means that there can be more than one next state. In DFA there can only be one next state.

The string $w \in \Sigma^*$ is accepted by a Finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ if $\delta(q_0, w)$ contains some final state.

NFA to DFA conversion

Given input:

$M = (\{q_1, q_2, q_3, q_4\}, \{0,1\}, \delta, q_1, \{q_4\})$

δ is:

Q \ Σ	0	1
→ q_1	q_1, q_2	q_1
q_2	q_2	q_4
q_3	q_4	q_3
(q_4)		

Conversion:

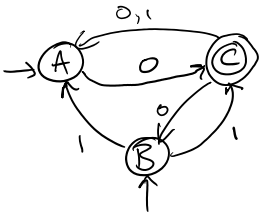
Q \ Σ	0	1
→ [q_1] (initial state)	[q_1, q_2]	[q_1]
[q_1, q_2]	[q_1, q_2, q_3]	[q_1, q_4]
[q_1, q_2, q_3]	[q_1, q_2, q_3, q_4]	[q_1, q_2, q_4]
[q_1, q_2, q_3, q_4]	[q_1, q_2, q_3, q_4]	[q_1, q_2, q_3, q_4]
[q_1, q_2, q_4]	[q_1, q_2, q_3]	[q_1, q_2, q_4]

final states

Steps:

1. Start with initial state, write the outputs as a single node
2. Write a new node equal to each new (not previously defined) output
3. According to the rules from the NFA table, find the set of nodes that this will output
4. Repeat until no new outputs are created
5. The initial state will be the same as NFA
6. The final state will be any node with a combination including the final state node on the NFA.

Given input:



Convert to transition table:

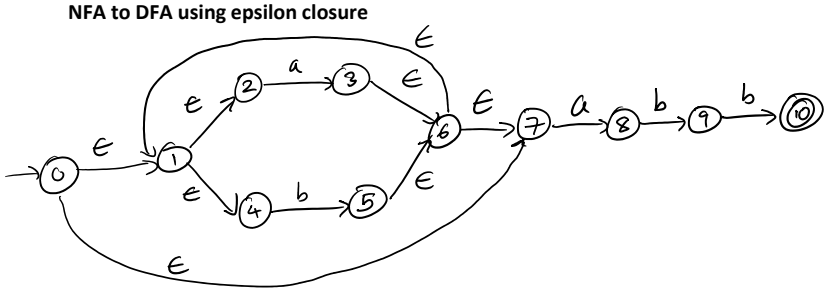
	0	1
→ A	C	—
→ B	—	AC
C	AB	A

Use same rules as above

	0	1
→ AB	C	AC
C	AB	A
AC	ABC	A
A	C	ϕ
ABC	ABC	AC
ϕ	ϕ	ϕ

- When you have multiple start states, clump them together into one node/start state
- When you have no transition on the NFA (e.g. here if you try input 1 on state A) and it is the only output from that state in the DFA table then write phi (or another symbol) to indicate null

NFA to DFA using epsilon closure



NFA State	DFA State	a	b
{0,1,2,4,7}	A	B	C
{1,2,3,4,6,7,8}	B	B	D
{1,2,4,5,6,7}	C	B	C
{1,2,4,5,6,7,9}	D	B	E
{1,2,3,5,6,7,10}	E	B	C

$A = \epsilon\text{-closure}(\{0\}) = \{0,1,2,4,7\}$ apply a/b to these states to get next ϵ closure output

Aa $B = \epsilon\text{-closure}(\{3,8\}) = \{1,2,3,4,6,7,8\}$ Ba = no new comb

Ab $C = \epsilon\text{-closure}(\{5\}) = \{1,2,4,5,6,7\}$ no new comb

Bb $D = \epsilon\text{-closure}(\{5,9\}) = \{1,2,4,5,6,7,9\}$ Da = no new comb

Db $E = \epsilon\text{-closure}(\{5,10\}) = \{1,2,4,5,6,7,10\}$ no new comb

DFA states

NFA states

Minimisation of finite automata

Definitions

1. Two states are equivalent if both $d(q_1,x)$ and $d(q_2,x)$ are final states or both are non-final states for

all possible inputs in the set.

2. Two states are k equivalent if both $d(q_1,x)$ and $d(q_2,x)$ are final states or both are non-final states for all inputs of length k or less in the set.

Properties

1. The definitions above are reflexive, symmetric and transitive
2. If q_1 and q_2 are K-equivalent for all k, then they are equivalent
3. If q_1 and q_2 are k+1 equivalent, then they are k equivalent

- [ ] WRITE EXAMPLE HERE

Grammars & Regex

03 February 2020 21:08

Definition of a Grammar

A grammar is: (V_n, Σ, P, S)
Where:

- V_n is a finite non empty set whose elements are variables
- Sigma or V_T is a finite non-empty set whose elements are terminals
- $V_n \cap \Sigma = \varphi$
  - This means you cannot find a single grammar symbol that is both terminal and non terminal at the same time
- S is a start symbol, where $S \in V_n$
- P is a finite set whose elements are $\alpha \to \beta$, known as production rules where $\alpha, \beta \in (V_n \cup \Sigma)^*$ but alpha should contain at least one symbol from V_n.

Left linear grammar
All productions are in the form:
$A \to B\alpha$
$A \to \alpha$

Example:
$A \to Aa|Bb|b$

Right linear grammar
All productions are in the form:
$A \to \alpha B$
$A \to \alpha$

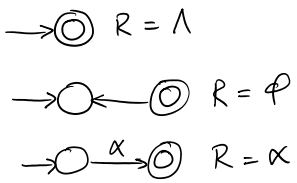
Example:
$A \to aA|bB|b$

Classification of languages

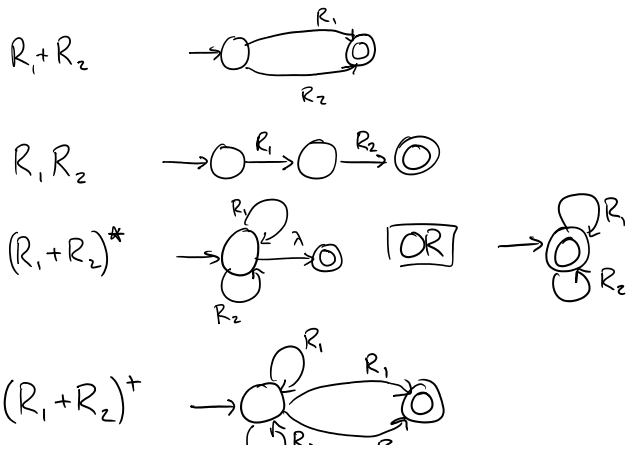
Regular Expressions

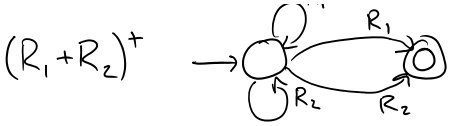
Identities
R* = Λ + R + RR + RRR …
R⁺ = R + RR + RRR …

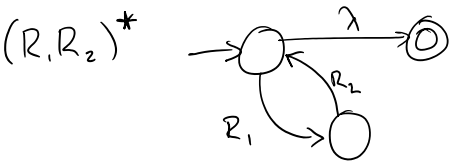
I_1	Φ + R = R
I_2	ΦR = RΦ = Φ
I_3	ΛR = RΛ = R
I_4	Λ* = Λ and Φ* = Λ
I_5	R + R = R
I_6	R*R* = R*
I_7	RR* = R*R
I_8	(R*)* = R*
I_9	Λ + RR* = R* = Λ + R*R
I_{10}	(PQ)*P = P(QP)*
I_{11}	(P+Q)* = (P*Q*)*
I_{12}	(P+Q)R = PR + QR

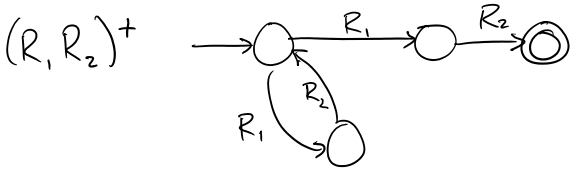


Regex to transition systems





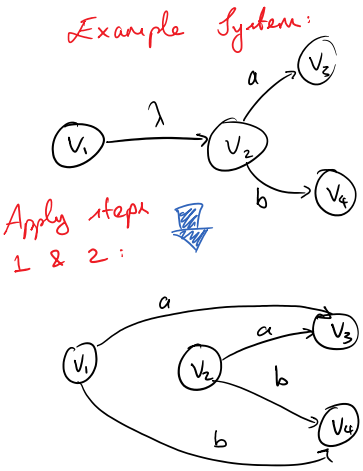


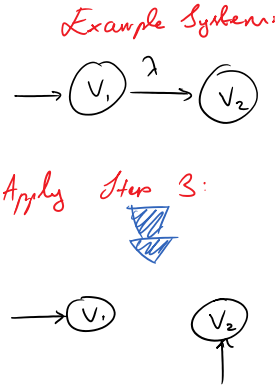


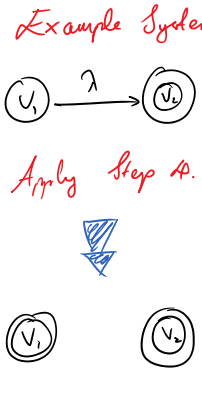
Lambda transition elimination

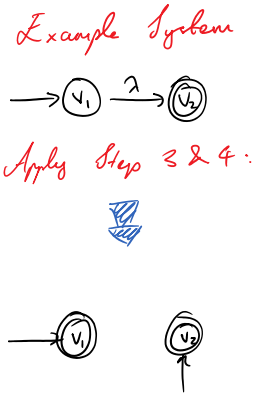
Rules:

1. Find all the edges starting from v_2
2. Duplicate all these edges starting from v_1 keeping the same edge labels
3. If v_1 is the initial state, make v_2 the initial state also
4. If v_2 is the final state, make v_1 the final state also









Leftmost and Rightmost derivation in CFG

- $A \Rightarrow \omega$ is leftmost derivation if we apply a production only to the left most variable at every step
- It is the rightmost derivation if we apply a production only to the right most variable at every step

Example: ① $S \to aAS$, ② $S \to a$, ③ $A \to SbA$, ④ $A \to SS$, ⑤ $A \to ba$

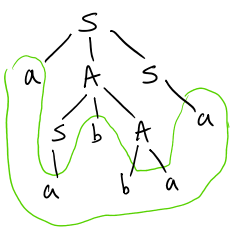
input string 'aabbaa'

1) Leftmost derivation

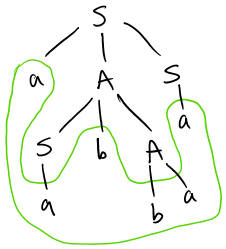
$S \Rightarrow aAS$ by (1)
$\Rightarrow aSbAS$ by (3)
$\Rightarrow aabAS$ by (2)
$\Rightarrow aabbaS$ by (5)
$\Rightarrow aabbaa$ by (2)

2) Rightmost derivation

$S \Rightarrow aAS$ by (1)
$\Rightarrow aAa$ by (2)
$\Rightarrow aSbAa$ by (3)
$\Rightarrow aSbbaa$ by (5)
$\Rightarrow aabbaa$ by (2)



aabbaa



aabbaa

Derivation tree is the same!

Derivation tree is the same!

Augmented Grammar

If G is a grammar with start symbol S, the augmented grammar for G is G# with a new start symbol S' and production S' -> S will be included.

The purpose of the new starting production is to indicate to the parser when it should stop parsing and accept the input (i.e. when the parser was about to reduce by S' -> S).

Grammar G

E → E + T | T

T → T * F | F

F → (E) | id

G'

" "

" "

" "

E' → E

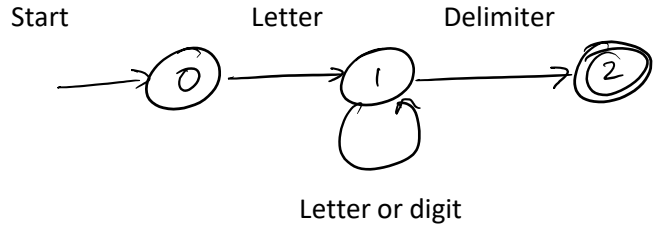
Lexical analysis

03 February 2020 18:17

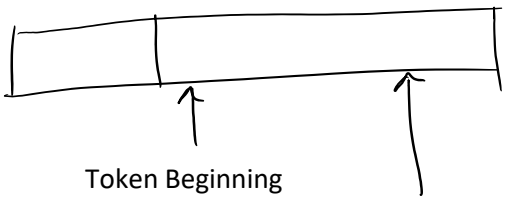
Introduction

A lexical analyser acts as a subroutine or coroutine, which is called by the parser whenever it needs a new token.

In most programs a variable name must start with a character, and then contain any number of letters or digits



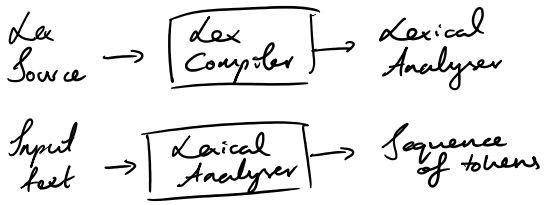
Input buffer:



Lookahead pointer checks for a delimiter, which indicates end of token. If a delimiter is detected, then it will detract to the previous character.

Lex program

A LEX program is a specification of a lexical analyser, consisting of a set of regular expressions together with an action for each regular expression.



Lex program contains

- A sequence of auxiliary definitions
- A sequence of translation rules

Auxiliary definitions:

Statements in the form

$D_1 = R_1$

…

$D_n = R_n$

Where D_i is a distinct name and each R_i is a regular expression, whose symbols are chosen from $\Sigma \cup (D_1, D_2 \cdots D_{i-1})$ i.e. characters or previously defined names

Translation rules:

Statements in the form

P_1 {A_1}

…

P_m {A_m}

P_i is a regular expression called pattern, Each A_i is a code fragment describing what

action the lexical analyser should take when token P is found.

Examples:
Number of tokens in the following C- statement:

printf (" i = %d , &i = %x " , i , &i) ; ⇒ 10 tokens

Parsing Techniques

03 February 2020 18:18

Operator Precedence Parsing

- Grammars must have no epsilon production on the RHS
- Grammars must have no two adjacent non-terminals on the RHS

Example: $E \to EAE \mid (E) \mid -E \mid id$

$A \to + \mid - \mid * \mid / \mid \uparrow$

$E \to E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid -E \mid id$

using A, and expanding $E(+ \mid - \mid * \mid / \mid \uparrow)E$

Rules:

1. If precedence of θ_1 is greater than θ_2:
   a. $\theta_1 > \theta_2$, $\theta_2 < \theta_1$
2. If precedence's of θ_1 and θ_2 are equal then:
   a. $\theta_1 > \theta_2$, $\theta_2 < \theta_1$ if left associative
   b. $\theta_2 > \theta_1$, $\theta_1 < \theta_2$ if right associative
3. Other rules:

$(\doteq)$ $\quad \$ \lessdot ($ $\quad \$ \lessdot id$

$(\lessdot ($ $\quad id \gtrdot \$$ $\quad) \gtrdot \$$ $\quad \to$

$(\lessdot id$ $\quad id \gtrdot)$ $\quad) \gtrdot)$

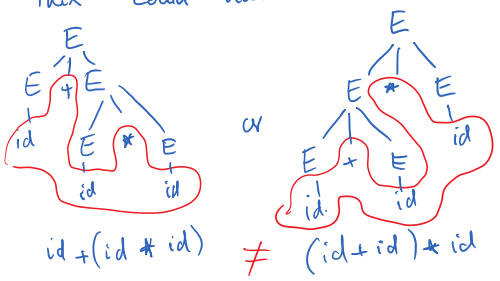
$a \gtrdot b$ a has precedence higher than b

these are used to reduce ambiguity

$\$$ = end of string marker

Example $E \to E+E \mid E*E \mid id$

This could have 2 trees



$id + (id * id) \neq (id + id) * id$

for $id + id * id$

Using operator precedence:

$\uparrow$

$* \quad /$

$+ \quad -$

$id + id * id$

$\$ \; id + id * id \; \$$

$\$ \lessdot id \gtrdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$

$\$ \lessdot + \lessdot id \gtrdot * \lessdot id \gtrdot \$$

$\$ \lessdot + \lessdot * \lessdot id \gtrdot \$$

$\$ \lessdot + \lessdot * \gtrdot \$$

Go from the start until the first > is reached, from there go back until the first < is reached. What is enclosed between these will be the handle. Remove this token, and repeat the process until all handles are gone.

Calculation of FIRST

Rules:

1. If X is a terminal then FIRST(X) = {X}
2. If X is non-terminal and X->aα, then add a to FIRST(X)
   a. If X->ε is a production, then add ε to FIRST(X)
3. If $X \to Y_1 Y_2 \ldots Y_k$ is a production, then for all I such that all of $Y_1 Y_2 \ldots Y_{i-1}$ are non-terminals and FIRST(Y_j) contains ε for j= 1 to (i-1) then add every non epsilon symbol in FIRST(Y_j) to FIRST(X). If epsilon is in FIRST(Y_j) for all j, then add epsilon to FIRST(X).

Example:

$E \to TE'$ $\quad E' \to TE' \mid \epsilon$ $\quad T \to FT'$ $\quad T' \to *FT' \mid \epsilon$ $\quad F \to (E) \mid id$

$E \to TE'$: $X \to Y$

$T \to FT'$: $X \to Y$

$T' \to *FT' \mid \epsilon$: $X \to Y$ and Y has epsilon so first(X) has ε too

$F \to (E) \mid id$: brackets

everything non epsilon in first(T) will be in first(E)

$-\epsilon$

First (E) = { (, id }

First (E') = { +, ε }

First (T) = { (, id }

$-\epsilon$

First (T') = { *, ε }

First (F) = { (, id }

= rule 3

= rule 2

Calculation of Follow

Rules:

1. If $ is in FOLLOW(s), where S is the start symbol.
2. If there is a production A -> αBβ, β ≠ ε, then everything in FIRST(β) but ε is in FOLLOW(B) FOLLOW(B) ← FIRST(β) − {ε}
3. If there is a production A -> αB, or a production A -> αBβ where FIRST(β) contains ε (i.e. β => ε), then everything in FOLLOW(A) is in FOLLOW(B)

follow (E) = { $,) }

follow (E') = { $,) }

= rule 2

then everything in FOLLOW(A) is in FOLLOW(B)

follow(E) = { $,) }
follow(E') = { $,) }
follow(T) = { + , $,) }
follow(T') = { + }
follow(F) = { * , + , $,) }

= rule 2
= rule 3

Predictive parsing table construction

1. For each production A->α of the grammar do steps 2 & 3
2. For each terminal a in FIRST(α), add A->ε to M[A,b] for each terminal b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A) add A->α to M[A, $]
3. Make each of the undefined entry of M error

from before:

E->TE'
E'-> +TE' | ε
T-> FT'
T'-> *FT' | ε
F -> (E) | id

FIRST
first(E) = first(T) = first(F) = { (, id }
first(E') = { +, ε }
first(T') = { *, ε }

FOLLOW
follow(E) = follow(E') = { $,) }
follow(T) = follow(T') = { +, $,) }
follow(F) = { *, +, $,) }

	id	+	*	(	)	$
E	E->TE'			E->TE'		
E'		E'->+TE'		E'-> ε		E'-> ε
T	T->FT'			T->FT'		
T'		T'-> ε	T'->*FT'		T'-> ε	T'-> ε
F	F->id			F->(E)		

= using step 2: e.g. E->TE' ⇒ first(E) = { (, id } ⇒ write E->TE' in E row, 'id' column and '(' column
= using step 1: e.g. E'->+TE' ⇒ first(terminal) = terminal itself ⇒ write E'->+TE' in terminal column and E' row.
= using step 3: eg. E'-> ε

Predictive Parser

Using the predictive parsing table, we can now parse an input by following the following instructions:

```
Repeat
    Begin
        Let X be the top stack symbol and a the next input symbol
        If X is a terminal or $ then
            If X = a then
                Pop X from the stack and remove a from the input
            Else ERROR
        Else
            If M[X, a] = X -> Y1 Y2 ... Yk then
                Begin
                    Pop X from the stack
                    Push Yk ... Y2 Y1 onto the stack, Y1 on top
                End
            Else ERROR
    End
Until X = $
```

Example:

Stack	Input	Output
$E	id + id * id $	
$E'T	id + id * id $	E->TE'
$E'T'F	id + id * id $	T->FT' !
$E'T'id	id + id * id $	F -> id
$E'T'	+ id * id $	no output – id is a terminal
$E'	+ id * id $	T'-> ε

$E'T	+ id * id $	no output — id is a terminal
$E'	+ id * id $	T' → ∈
$E'T+	+ id * id $	E' → +TE'
$E'T	id * id $	no output — + is a terminal
$E'TF	id * id $	T → FT' — Repeating from !
$E'T'id	id * id $	F → id
$E'T'	* id $	no output — id is a terminal
$E'T'F*	* id $	T' → * FT'
$E'T'F	id $	no output — * is a terminal
$E'T'id	id $	F → id
$E'T'	$	no output — id is a terminal
$E'	$	T' → ∈
$	$	E' → ∈ end when X = $

Shift Reduce Parser

Viable prefixes and valid items

Viable prefixes:

The prefixes of right sentential (any string derivable from the start symbol) forms that can appear on the stack of a shift reduce parser

It does not continue past the right end of the rightmost handle (the leftmost simple phase of a sentential form) of that sentential form.

Valid items:

For the item $A \rightarrow \beta_1 \cdot \beta_2$ and $A \Rightarrow \alpha A \beta \Rightarrow \alpha\beta_1\beta_2\beta$

Since the dot is in between B_1 and B_2, aB_1 will be on top of the stack and so is a viable prefix. This means that the item $A \rightarrow \beta_1 \cdot \beta_2$ is a valid item. Every viable prefix is associated with a valid item, and an item is usually valid for many viable prefixes.

Set of items construction

Closure:

Begin

Repeat

For each item$A \rightarrow \alpha \cdot B\beta$ in I and each production $B \rightarrow \gamma$ in G such that $B \rightarrow \cdot \gamma$ is not in I

Add $B \rightarrow \cdot \gamma$ to I

Until no more items can be added to I

Return I

End

Set of items:

Begin

C = {Closure({S' -> .S})}

Repeat

For each set of items I in C and each grammar symbol X such that GOTO(X,X) is not empty and not already in C

Add GOTO(I, X) into C

Until no more sets of items can be added

End

Grammar G'

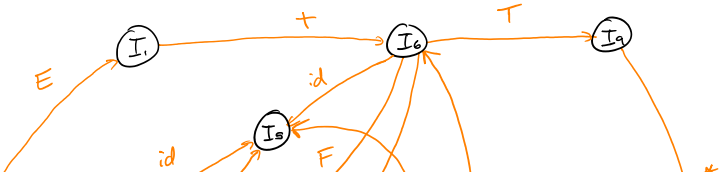
E' → E

E → E+T | T

T → T*F | F

F → (E) | id

We can draw a DFA for this



$T \to T * F \mid F$

$F \to (E) \mid id$

I_0: finding closure of E'

$E' \to \cdot E$

$E \to \cdot E + T \mid \cdot T$

$T \to \cdot T * F \mid \cdot F$

$F \to \cdot (E) \mid \cdot id$

$Goto(I_0, E) = I_1$

$E' \to E \cdot$

$E \to E \cdot + T$

$Goto(I_0, T) = I_2$

$E \to T \cdot$

$T \to T \cdot * F$

⋮ do all the I_0 gotos

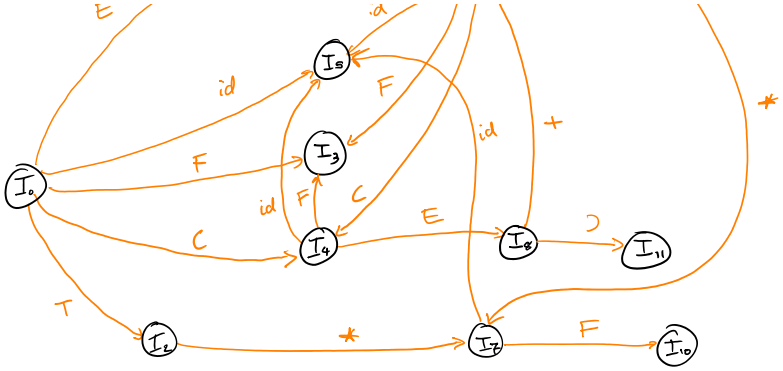
F, (, id, +

$Goto(I_2, *) = I_7$

$T \to T * \cdot F$

$F \to \cdot (E) \mid \cdot id$

continue until the dot is at the end of all productions



Parsing table construction

Action parsing rules

1. If GOTO (I_i , a) = I_j then set action [i, a] to shift j
2. If [A -> α·] is in I_i, then set action[i, a] to reduce A -> α for all α in FOLLOW(A)
3. If [S' -> S·] in I_i then set action [i, $] to accept

GOTO parsing rules

1. If GOTO(I_i, A) = I_j then goto [I, A] = j
2. All other entries error
3. Initial state is from items containing S' -> ·S

Production Rule Numbers

① $E \to E + T$

② $E \to T$

③ $T \to T * F$

④ $T \to F$

⑤ $F \to (E)$

⑥ $F \to id$

Parsing Table

State	Action						Goto		
	id	+	*	(	)	$	E	T	F
0							1	2	3
1									
2	—	r2	s7	—	r2	r2	—	—	—
3									
⋮									

First

$E' = \{(, id\}$

$E = \{(, id\}$

$T = \{(, id\}$

$F = \{(, id\}$

Follow

$E' = \{\$\}$

$E = \{\$, +,)\}$

$T = \{\$, +, *,)\}$

$F = \{\$, +,)\}$

Example $Goto(I_2, *) = I_7$ rule 1. says set [2, *] = shift 7

Example in I_2 : $E \to T \cdot$ rule 2. says [2, x] = reduce 2 where x = follow E

Example $GOTO(I_0, E) = I_1$, $GOTO(I_0, T) = I_2$, $GOTO(I_0, F) = I_3$ GOTO rule 1. says [2,E] = 1, [2,T] = 2, [2,F] = 3

- [ ] **LR parser/parsing table construction**
- [ ] **LALR parser/parsing table construction**