CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall

Phases of a compiler

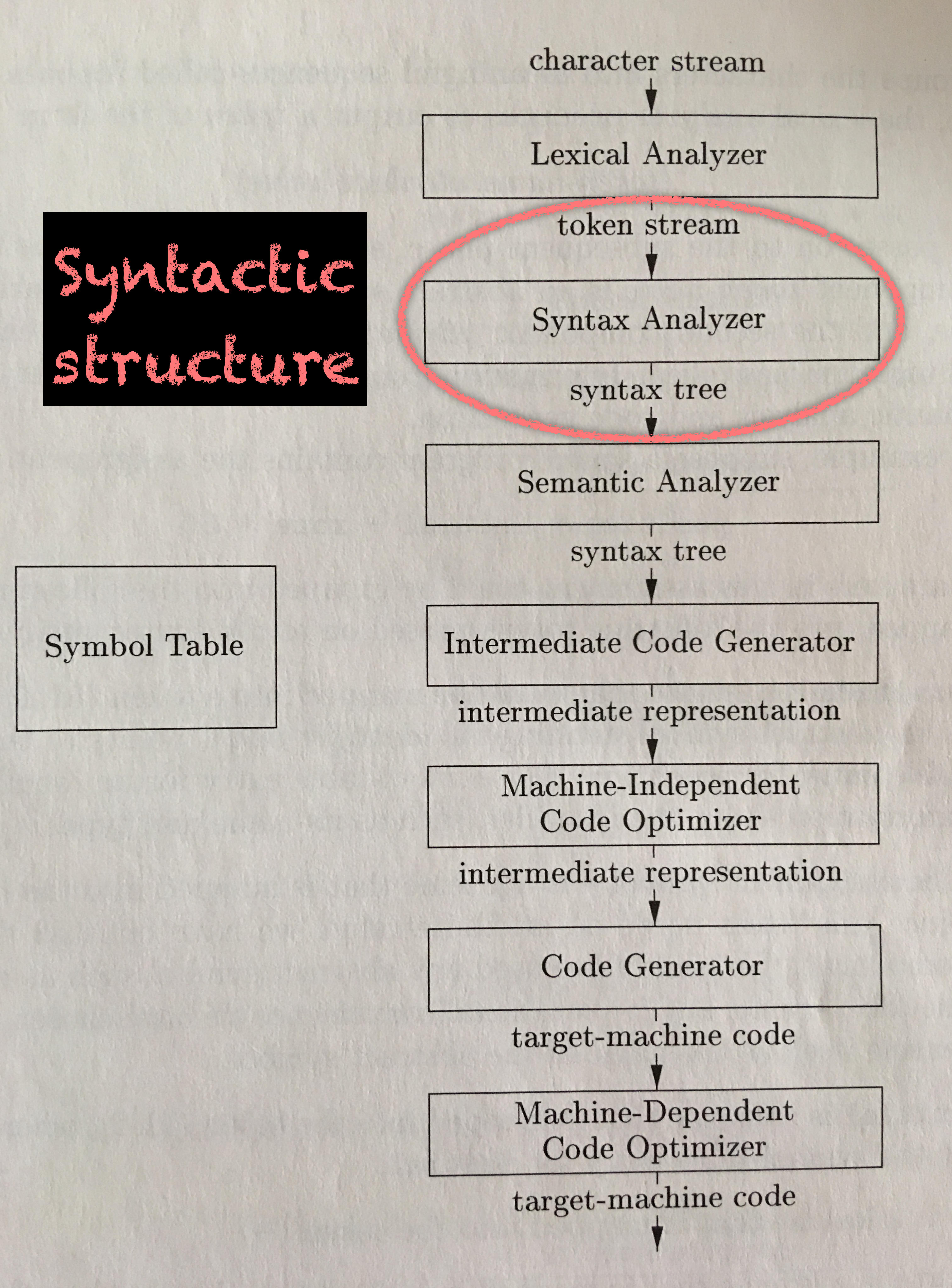


Figure 1.6, page 5 of text

Bottom-up parsing

Top-down predictive parsing gave us a quick overview of issues related to parsing.

With this context we can more easily describe bottom-up parsing.

Example grammar

E -> E + T

E -> T

T -> T * F

T -> F

F -> (E)

F -> id

Same expression grammar we used for top-down presentation.

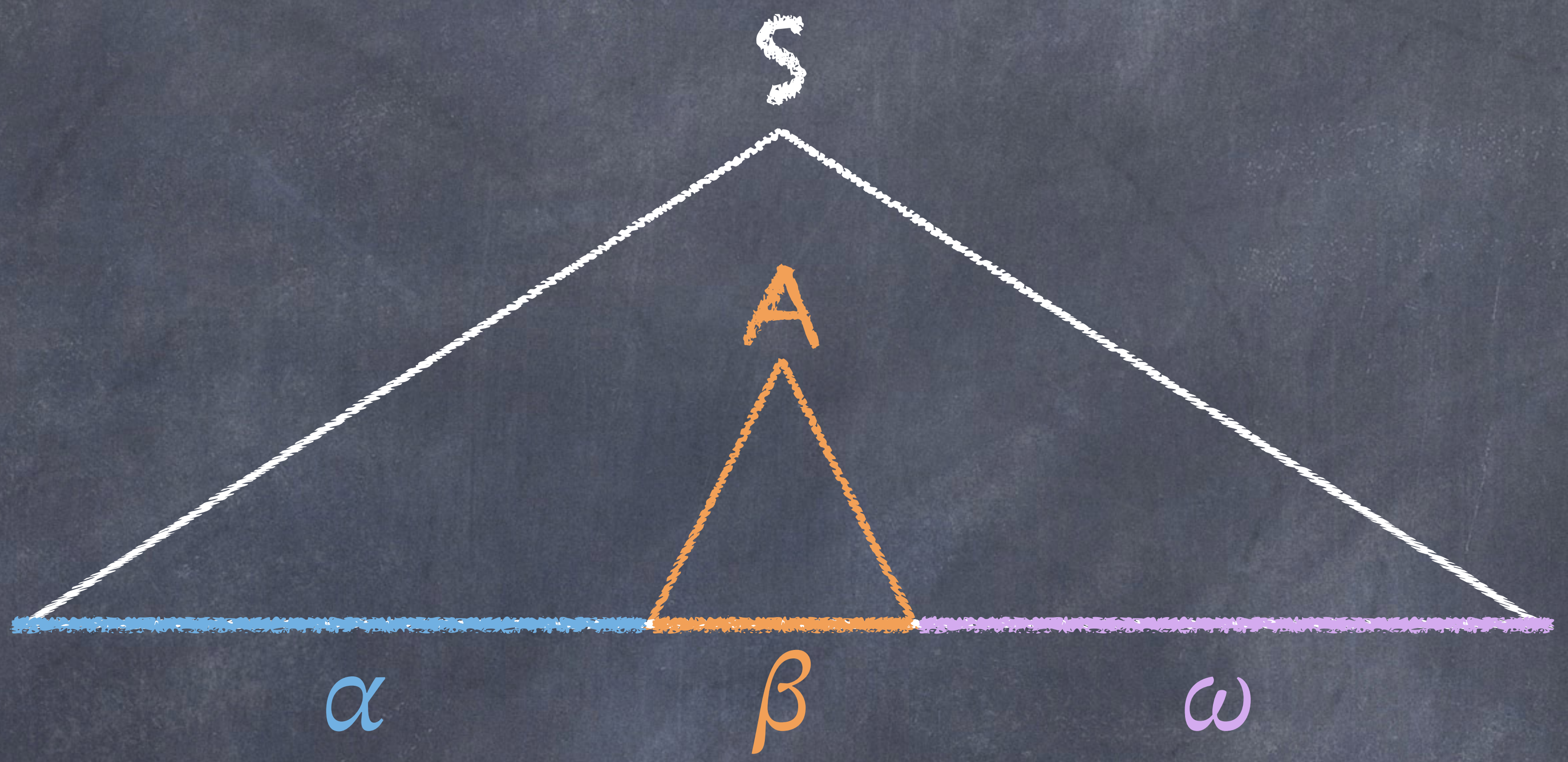
Terminology

- If $S \Rightarrow^*_{lm} \alpha$ then we call α a left-sentential form of the grammar (lm means leftmost)
- If $S \Rightarrow^*_{rm} \alpha$ then we call α a right-sentential form of the grammar (rm means rightmost)

handle

- "Informally, a 'handle' is a substring that matches the body of a production and whose reduction represents one step along the reverse of a rightmost derivation." [p. 235]
- "Formally, if $S \Rightarrow^*_{rm} \alpha A \omega \Rightarrow_{rm} \alpha\beta\omega$, then the production A -> β in the position following α is a handle of $\alpha\beta\omega$" [p. 235]
- "Alternatively, a handle of a right-sentential form γ is a production A -> β and a position of γ where the string β may be found, such that replacing β at that position by A produces the previous right-sentential form in a rightmost derivation of γ." [p. 235]

As a picture



"A handle A -> β in the parse tree for αβω" Fig 4.27 [p. 236]

A rightmost derivation of the string

id * id

Rightmost derivation	Production
E	
⇒ T	E -> T
⇒ T * F	T -> T * F
⇒ T * id	F -> id
⇒ F * id	T -> F
⇒ id * id	F -> id

Recall grammar

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

[p.235]

A bottom-up parse: what we're aiming for!

Table is reverse of that on previous slide.

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

id * id has handle id

(or more formally F -> id is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

F * id has handle F

(or more formally T -> F is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T * id has handle id

(or more formally F -> id is a handle after T *)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T * F has handle T * F

(or more formally T -> T * F is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

T has handle T

(or more formally E -> T is a handle)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id
T * F	T * F	T -> T * F
T	T	E -> T
E		

figure 4.26 [p.235]

What happens if we reduce something that's not a handle?

T * id has handle id

(or more formally F -> id is a handle after T *)

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	id	F -> id

Consider this point in the previous table.

We identified F -> id as a handle.

figure 4.26 [p.235]

Example - figure 4.26 [p.235]

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	T	E -> T

What if …

… we made a different choice?

Example - figure 4.26 [p.235]

Right sentential form	Handle	Reducing production
id * id	id	F -> id
F * id	F	T -> F
T * id	T	E -> T
E * id	id	F -> id
E * F	F	T -> F
E * T	T	E -> T
E * E	*FAIL*	

T * id could be reduced to E * id using production E -> T, but E -> T is not a handle since that reduction does not represent "one step along the reverse of a rightmost derivation."

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

Basic idea

If we know what the handle is for each right sentential form, we can run the rightmost derivation in reverse!

Handle pruning [p 235]

- "A rightmost derivation in reverse can be obtained by 'handle pruning' "
- If $\omega \in \mathcal{L}(G)$:

Rightmost derivation →

$$S = \gamma_0 \Rightarrow_{rm} \gamma_1 \Rightarrow_{rm} \gamma_2 \Rightarrow_{rm} \cdots \Rightarrow_{rm} \gamma_{n-1} \Rightarrow_{rm} \gamma_n = \omega$$

← Handle pruning

Big question

How do we figure out the handles?

Big question

How do we figure out the handles?

We'll answer this in a bit, but first let's consider how a parse will proceed in a bit more detail.

Shift-reduce parsing

STACK [Bottom...Top]	INPUT
$	ω $
↓	
$ S	$

[modified from fig 4.28, p 237]

Revisit example, with input: id * id $

Stack	Lookahead	Handle	Action
$	id * id $		Shift
$ id	* id $	id	Reduce F -> id
$ F	* id $	F	Reduce T -> F
$ T	* id $		Shift
$ T *	id $		Shift
$ T * id	$	id	Reduce F -> id
$ T * F	$	T * F	Reduce T -> T * F
$ T	$	T	Reduce E -> T
$ E	$		Accept

Observations [p 235]

- ω, the string after the handle, must be $\in T^*$
- We say "a handle" rather than "the handle" since the grammar may be ambiguous and may therefore allow more than one rightmost derivation of $\alpha\beta\omega$.
- If a grammar is unambiguous, then every right-sentential form of the grammar has exactly one handle.

Items

- "How does a shift-reduce parser know when to shift and when to reduce?" [p 242]
- "...by maintaining states to keep track of where we are in a parse."
- Each state is a set of items.
- An item is a grammar rule annotated with a dot, •, somewhere on the RHS.

Rules and items

A -> X Y Z
A -> • X Y Z
A -> X • Y Z
A -> X Y • Z
A -> X Y Z •

A -> ε
A -> •

The • shows where in a rule we might be during a parse.

Building the finite control for a bottom-up parser

- Build a finite state machine, whose states are sets of items
- Build a table (M) incorporating shift/reduce decisions

Augment grammar

Given a grammar

$G = (N,T,P,S)$

we augment to a grammar

$G' = (N \cup \{S'\}, T, P \cup \{S' \to S\}, S')$, where $S' \notin N$

G' has exactly one rule with S' on left.

We need two operations to build our finite state machine

CLOSURE(I)

GOTO(I,X)

CLOSURE(I)

- I is a set of items
- CLOSURE(I) fixed point construction

$CLOSURE_0(I) = I$

repeat {

$CLOSURE_{i+1}(I) =$

$CLOSURE_i(I) \cup \{ B \rightarrow \bullet\gamma \mid A \rightarrow \alpha\bullet B\beta \in CLOSURE_i(I) \text{ and } B \rightarrow \gamma \in P \}$

} until $CLOSURE_{i+1}(I) = CLOSURE_i(I)$

CLOSURE(I)

- I is a set of items
- CLOSURE(I) fixed point construction

Intuition: an item like A -> X • Y Z conveys that we've already seen X, and we're expecting to see a Y followed by a Z.

The closure of this item is all the other items that are relevant at this point in the parse.

For example, if Y -> R S T is a production, then Y -> • R S T is in the closure because if the next thing in the input can derive from Y, it can derive from R.

GOTO(I,X)

- GOTO(I,X) is the closure of the set of items A -> αX•β s.t. A -> α•Xβ $\in$ I
- GOTO(I,X) construction for G' (figure 4.32):

```
set-of-items CLOSURE(I) {
    J = I
    repeat {
        for each item A -> α•Bβ ∈ J
            for each production B -> γ ∈ P
                if B->•γ not already in J
                    add B->•γ to J
    } until no more items are added to J
    return J
}
```

Building the LR(0) automaton

```
void items(G') {
   C = { CLOSURE( { S' -> •S } ) }
   repeat {
      for each set of items I ∈ C and
      for each grammar symbol X ∈ (N∪T)
      if ( GOTO(I,X) is not empty and not already in C )
         add GOTO(I,X) to C
   } until no new sets of items are added to C
}
```

C is a set of sets of items

Example [p 245]

Grammar G	Augmented Grammar G'
	S' -> E
E -> E + T	E -> E + T
E -> T	E -> T
T -> T * F	T -> T * F
T -> F	T -> F
F -> (E)	F -> (E)
F -> id	F -> id

Compute items(G')

S' -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> id

SET OF ITEMS (I)	i	$CLOSURE_i(I)$
{ S' -> • E }	0	{ S' -> • E }

Compute items(G')

S' -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> id

SET OF ITEMS (I)	i	$CLOSURE_i(I)$
{ S' -> • E }	0	{ S' -> • E }
	1	$CLOSURE_0(I)$ ∪ { E -> • E + T , E -> • T }

Compute items(G')

S' -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> id

SET OF ITEMS (I)	i	$CLOSURE_i(I)$
{ S' -> • E }	0	{ S' -> • E }
	1	$CLOSURE_0(I)$ ∪ { E -> • E + T , E -> • T }
	2	$CLOSURE_1(I)$ ∪ { T -> • T * F , T -> • F }

Compute items(G')

S' -> E
E -> E + T
E -> T
T -> T * F
T -> F
F -> (E)
F -> id

SET OF ITEMS (I)	i	CLOSURE$_i$(I)
{ S' -> • E }	0	{ S' -> • E }
	1	CLOSURE$_0$(I) ∪ { E -> • E + T , E -> • T }
	2	CLOSURE$_1$(I) ∪ { T -> • T * F , T -> • F }
	3	CLOSURE$_2$(I) ∪ { F -> • (E) , F -> • id }

Compute items(G')

$S' \to E$
$E \to E + T$
$E \to T$
$T \to T * F$
$T \to F$
$F \to (E)$
$F \to id$

SET OF ITEMS (I)	i	$CLOSURE_i(I)$
$\{ S' \to \bullet E \}$	0	$\{ S' \to \bullet E \}$
	1	$CLOSURE_0(I) \cup \{ E \to \bullet E + T , E \to \bullet T \}$
	2	$CLOSURE_1(I) \cup \{ T \to \bullet T * F , T \to \bullet F \}$
	3	$CLOSURE_2(I) \cup \{ F \to \bullet (E) , F \to \bullet id \}$
	4	$CLOSURE_3(I) \cup \varnothing$

Terminology

- Kernel items: S' -> • S and all items with • not at left edge
- Non-kernel items: all items with • at left edge, except S' -> • S

This gives us the first state of the finite state machine, I_0

I_0		
	S' -> • E	kernel item
	E -> • E + T	non-kernel items
	E -> • T	are computed from
	T -> • T * F	CLOSURE(kernel),
	T -> • F	and therefore do
	F -> • (E)	not need to be
	F -> • id	explicitly stored

Next we compute GOTO(I_0,X) $\forall$ X $\in$ N $\cup$ T

N $\cup$ T = { E , T , F , + , * , (,) , id }

N.B. - augmented start symbol S' can be ignored

GOTO(I_0,E) = CLOSURE({ S' -> E •, E -> E • + T })

= { S' -> E • , E -> E • + T }

N.B. there is no non-terminal after the •, so no new items are added by CLOSURE operation

I_1

S' -> E •
E -> E • + T

only kernel items