# CPSC 323 Compilers and Languages

Instructor: Susmitha Padda

Credits : Doina Bein

# Chapter 4. Syntax Analysis II
(Bottom-Up Parsers)

4.1 Introduction

4.2 Operator Precedence Parser

4.3 LR Parsers

4.3.1) Simple LR (SLR)

4.3.2) Canonical LR (CLR)

4.3.3) Look-Ahead LR (LALR)

4.4 Error Handling

4.5 Symbol Table

- There are fairly simple unambiguous grammars having no LL(1) parser:
- Example: Σ={a,b,c,+,-,-,*,(,) }

E -> E+T | E-T | T

T-> T*F | F

F -> a | b | c | ( E )

- But one can use a bottom-up parser, also called shift-reduce parser
- It uses a stack:
  - We will push input tokens into the stack until we have something we can reduce on the top of the stack; this is called *shift*
  - Reduce means substituting one or more top entries in the stack that form the RHS of a rule with the LHS of that rule; this is called *reduce*
- A *handle* is a string of terminals and /or nonterminals that is the RHS of a rule
- *Additionally, in the rightmost derivation, there are never any nonterminals to the right of a handle*
- At each step we need to decide whether to shift or reduce
- Making a shift-reduce parser to recognize a handle is the major concern: a handle must be the correct RHS of a rule, such that reducing that handle will take us one step back in the derivation from the starting nonterminal

# LR Parsing

- LR parsers are almost always table-driven:
  - like a table-driven LL parser, an LR parser uses a big loop in which it repeatedly inspects a two-dimensional table to find out what action to take
  - unlike the LL parser, however, the LR driver has non-trivial state (like a DFA), and the table is indexed by current input token and current state
  - the stack contains a record of what has been seen SO FAR (NOT what is expected)

# LR Parsing

- A scanner is a DFA
  - it can be specified with a state diagram
- An LL or LR parser is a PDA
  - Early's & CYK algorithms do NOT use PDAs
  - a PDA can be specified with a state diagram and a stack
    - the state diagram looks just like a DFA state diagram, except the arcs are labeled with <input symbol, top-of-stack symbol> pairs, and in addition to moving to a new state the PDA has the option of pushing or popping a finite number of symbols onto/off the stack

# LR Parsing

- An LL(1) PDA has only one state!
  - well, actually two; it needs a second one to accept with, but that's all (it's pretty simple)
  - all the arcs are self loops; the only difference between them is the choice of whether to push or pop
  - the final state is reached by a transition that sees EOF on the input and the stack

# 4.1 Introduction

Disadvantages of Top-Down parser:

- Remove left recursion and thus changing the grammar by introducing new nonterminals
- Hard to generate the intermediate code

Q: is there a better way of parsing without changing the grammar?

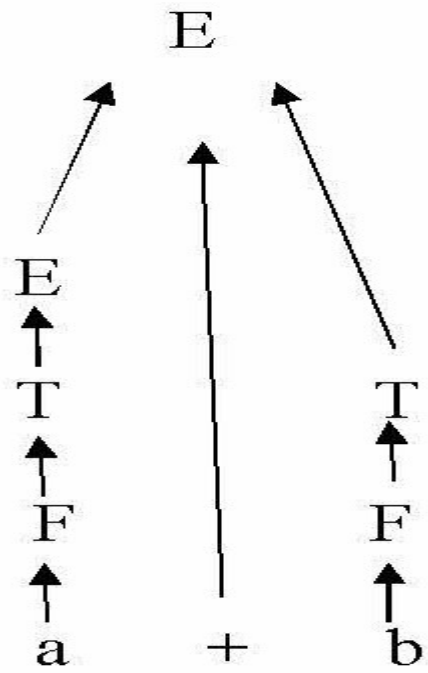
A: Yes, parse the string bottom up

(Start from the bottom of the tree and build up)

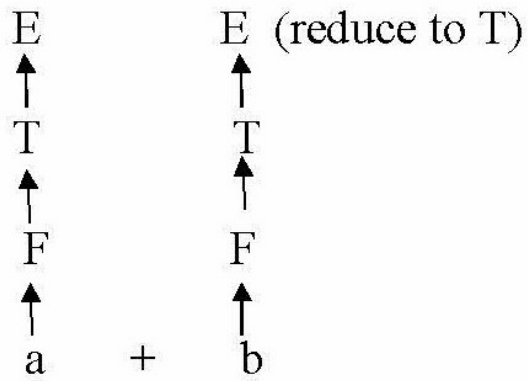
Ex: Let's consider the following grammar:

R1) E -> E + T
R2) E -> T
R3) T -> T * F
R4) T -> F
R5) F -> id

And parse the string a + b




But we also could have done this way!!




But it leads to nowhere

How do we know which way???

Consider the CFG:

R1) E -> E + T

R2) E -> T

R3) T -> T * F

R4) T -> F

R5) F -> id

Now let's consider a RMD for a + b from E:

**1) => E + T**

**2) => E + F**

**3) => E + id (b)**

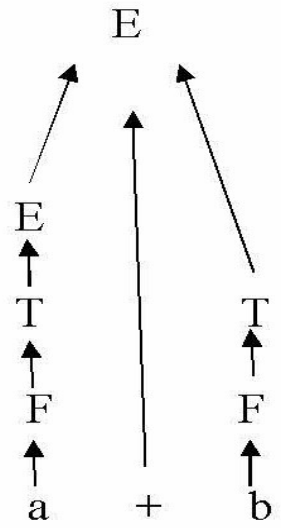
**4) => E + b**

**5) => T + b**

**6) => F + b**

**7) => id(a) + b**

**8) => a + b**




RMD shows how to reduce in reverse. It tells us that in (1) that we need to reduce according to E -> E + T and NOT E -> T.

**Conclusion:** We have to look at the RMD to see which reduction we need to apply in order to parse the tree bottom-up.

# Parsing with a Stack

- We will push tokens into the stack until at the top of the stack is the right hand side of a production (i.e we have something we can reduce on the top of the stack)
- When we have, we will do it by popping the right-hand side of a production off the stack and pushing the left-hand side of the production on its place
- We will continue pushing and reducing until the input is used up and the stack contains only the starting symbol
- **Handle**: A handle is the ***right-hand side*** of a production which we can reduce to get the preceding step in the RMD
- Additionally, in the rightmost derivation, there are never any nonterminals to the right of a handle

- In general, for β to be a handle of the sentential form αβω (ω is reserved for all terminal symbols), we must have

  αAω => αβω in the RMD, and

  S =*> αAω

This implies that A -> β exists and reduction leads to the preceding step of the RMD.

- Example: consider the following grammar:

S -> S+T | T

T -> T*a | a

and we want to parse a + a * a

- We add $ as endmarker:

A -> S$

S -> S+T | T

T -> T*a | a

and we want to parse a + a * a$

- The rightmost derivation:

A => S$ => S+T$=> S+T*a$ => S+a*a$=>T+a*a$=>a+a*a$

- In bottom-up manner, the parser moves are:

| move | stack | Unread input | production |
|---|---|---|---|
| - | $ | a+a*a$ | |
| shift | $a | +a*a$ | |
| reduce | $T | +a*a$ | T->a |
| reduce | $S | +a*a$ | S->T |
| shift | $S+ | a*a$ | |
| shift | $S+a | *a$ | |
| reduce | $S+T | *a$ | T->a |
| shift | $S+T* | a$ | |
| shift | $S+T*a | $ | |
| reduce | $S+T | $ | T->T*a |
| reduce | $S | $ | S->S+T |
| shift | $S$ | epsilon | |
| reduce | $A | | A->S$ |

- This parser is nondeterministic because we cannot decide whether to shift an input symbol onto the stack or to reduce a terminal on top of the stack, or which reduction is the correct one to apply
- There are certain combinations of top of the stack and input symbols for which a reduction is always appropriate, and a shift is correct for all other combinations; the set of such pairs is called *precedence relations*
- *Precedence grammars*: grammars in which precedence relations can be used to obtain a deterministic shift-reduce parser

| Ex. | E (RMD) |
| --- | --- |
| E | |
| E + T (handle: T or E+T? ) | 1) => E + T |
| | 2) => E + F |
| E + F (F) | 3) => E + id (b) |
| E + id(b) (id) | 4) => E + b |
| E + b (E) | 5) => T + b |
| T + b (T) | 6) => F + b |
| F + b (F) | 7) => F + id(b) |
| id(a) + b (id) | 8) => a + b |
| a + b | |

So, bottom-up parser is about finding the handles and reducing until the top of the tree is found.

# 4.2 Operator Precedence Parser

- Simplest bottom-up parser, however recognizes the smallest set of CFL.
- It uses precedence table, a stack and a driver to recognize a sentence.
- The precedence table consists of precedence relations <., .= , and .>
- We say that for two terminals a and b:
  - a <. b if a has a lower precedence than b
  - a .> b if a has a higher precedence than b
  - a .= b if a and b have the same precedence

- The precedence table will resolve all uncertain cases (dealing with ambiguity).
- Ex: for the following grammar

1) E -> E + E

2) E -> E * E

3) E -> id

The precedence table is

| stack \ token | + | * | Id | $ |
|---|---|---|---|---|
| + | .> | <. | <. | .> |
| * | .> | .> | <. | .> |
| id | .> | .> | | .> |
| $ | <. | <. | <. | |

Blank entries indicate error conditions, except the lower hand-right corner ([$,$]) which is the termination condition.

# Please note:

- We mean by a symbol = nonterminal or terminal or $
- Even if we call it the stack, we treat it as an array of symbols so we can access elements below the top of the stack
- When using the operator precedence table, we start at the top of the stack and we look down in the stack until we find a terminal
- The stack always has a terminal, in the worst case will be the $, the bottom of the stack
- If Table[stack, token] = <. , i.e. stack <. token then Execute: insert the operator precedence on top of the terminal in the stack (that was the top most terminal) and push token into the stack. We have cases when inserting is different from pushing, see next example
- If Table[stack, token] = >., i.e. stack >. token then Execute: reduce the top of the stack by selecting the top set of symbols up to a operator precedence; remove the operator precedence from the stack
- The role of the operator precedence in the stack is to tell you which is the handle to be reduced

**Driver:**

Push $ onto the stack

Append $ at the end of the input string

Repeat

- Let t = top most TERMINAL symbol in the stack and i = the incoming terminal
- Find in Table [t, i] the relationship between t and i
- If NO entry then error
- Else if t .< i or t .= i then push <. and i onto the stack (skip all NTs for <.)
- Else { /* t .> i */

  There is a handle on the stack delimited by <. and .>.

  Reduce the handle by popping it from the stack and pushing the left-hand side of the production onto the stack.

  If the symbols popped from the stack do not match the RHS of any production, then error

Until (t = $ and i = $) or error found

| token / stack | + | * | Id | $ |
|---|---|---|---|---|
| + | .> | <. | <. | .> |
| * | .> | .> | <. | .> |
| id | .> | .> | | .> |
| $ | <. | <. | <. | |

1) E -> E + E
2) E -> E * E
3) E -> id

**Ex. Parse id+id*id using OPP**

| Stack | Compare | Input | Production used |
|---|---|---|---|
| $ | <. | Id+id*id$ | |
| $<.id | .> | +id*id$ | E -> id |
| $E | <. | +id*id$ | |
| $<.E+ | <. | Id*id$ | |
| $<.E+<.id | .> | *id$ | E -> id |
| $<.E+ E | <. | *id$ | |
| $<.E+<E* | <. | id$ | |
| $<.E+<.E*<.id | .> | $ | E -> id |
| $<.E+<.E*E | .> | $ | E -> E * E |
| $<.E + E | .> | $ | E -> E + E |
| $E | | $ | Finished |

# Operator Precedence Parser (OPP) Evaluation

- The table can be build based on precedence and associativity, and it is hard to build it directly from the grammar
- Not all grammars lend themselves to operator precedence parsing (OPP)
- Restrictions:
  - No two or more consecutive nonterminals on the RHS of any productions
  - No two distinct nonterminals may have the same RHS
  - For any two terminals in the grammar, at most one of <., =., or .> must hold
  - We cannot have any Ɛ-productions

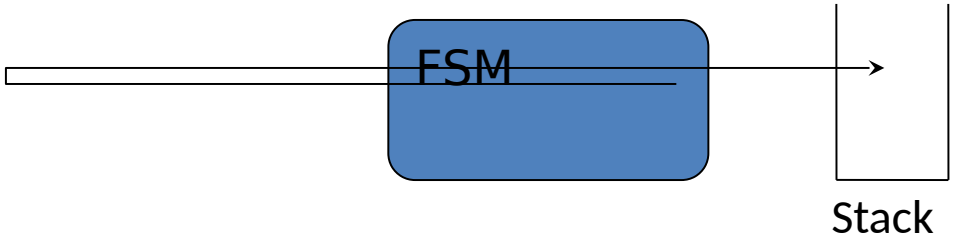
# 4.3 The LR Parser

- LR parser is the most powerful parser we will consider, i.e. can be build for any grammar we presented in class
- It handles the widest variety of CFGs, including all grammars that have predictive parsers or precedence parsers
- It works fast and it detects errors as soon as the first incorrect token is encountered
- It is easy to extend an LR parser to incorporate intermediate code generation
- General LR parser are called *LR(k)* parsers: Left-to-right scan of tokens, Rightmost derivation, look ahead *k* characters in the input
- Hopcroft and Ullman (1979) showed that any deterministic CFL can be handled by an LR(1) parser

- An LR parser uses a table; the table has two parts, the *action* part and the *go-to* part

| LR Parsing Table (consists of 3 parts) | | |
|---|---|---|
| States | Terminals (Action part) | Nonterminals (GoTo part) |
| | | |
| | | |
| | | |
| | | |

- Recall that a CFL is a Type 2 language and is accepted by a pushdown automaton (also called a *stack automaton*)
- A PDA has a finite-state controller and a stack



- In the LR parser, the parser table is the state table of the finite-state controller
- Each stack entry consists of a state and a grammar symbol, but in a practical LR parser, only the state is stored
- We include the production number in the table and not the actual production

Productions:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

| | Action Part | | | | | | GOTO Part | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

4 types of entries:

1. `Sn`: shift the input token into the stack and enter state n
2. `Rn`: reduce by means of Rule #n by popping the RHS with states as well up to the end of the RHS, then consider the state which is now on top of the stack (called tss), push the LHS, change the state by looking into go-to part of the table[tss, LHS] and push the new state into the stack
3. `n`: the new state due to some reduce action
4. `ACCT`: accept
5. Blank entry: error

**Driver:**

Place $ at the end of the input string

Push state 0 (the starting state) onto the stack

Repeat

Let $q_m$ be the current state (at the top of the stack) and i the incoming token

Find x = Table [$q_m$ , i];

Case x of

$S(q_n)$ : Shift (i.e. push) i onto stack and enter state $q_n$, i.e., push ($q_n$);

R(n): Reduce by means of Production #n by popping the symbols of the RHS of Rule #n (together with the states following them)

Let $q_j$ be the top of the stack state after reduce

Push the LHS L onto the stack

Push $q_k$ = Table [$q_j$ , L] onto the stack

ACCT: Accept. Parsing is complete

Error: error condition

Until ACCT or Error

Productions:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

| | Action Part | | | | | | GOTO Part | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

**Example of Parsing id+id$**

| Stack | Input | Table Entry | Action |
|---|---|---|---|
| 0 | id+id$ | S5 | |
| Push(id); push(5) | | | |
| 0id5 | +id$ | | R6 |
| F-> id; Table[0,F]=3 | | | |
| 0F3 | +id$ | | R4 |
| T -> F; Table[0,T] = 2 | | | |
| 0T2 | +id$ | | R2 |
| E -> T; Table[0,E] = 1 | | | |
| 0E1 | +id$ | | S6 |
| Push(+); push(6) | | | |
| 0E1+6 | id$ | | S5 |
| Push(id); push(5) | | | |

# Constructing an LR Parser

- Three stages, of increasing complexity:
  - Simple LR (SLR)
  - Canonical LR
  - Lookahead LR (LALR)
- SLR Parser is powerful enough to generate the parse table shown in the previous example
- Concepts needed: states of a parser, items, closure, state transitions
- States of the parser:
  - the stack stores states and symbols
  - a state corresponds to a particular sequence of symbols at the top of the stack
  - The current state depends on the topmost stack symbol and the current input token

# Items

- In the process of parsing a input string, handles need to be built in the top of the stack
- Handles appear in stages, not all at once, and sometimes building some part of the handle involves applying multiple rules while the rest of the handle sits in the stack
- To keep track how far we have gotten through the growth of a handle we use *items*
- An *item* is a production with a place marker, written as a dot, and inserted somewhere in the RHS; items are enclosed in [ …] to not be confused with rules
- An item whose RHS starts with a . is called *an initial item*
- An item whose RHS starts with a . is called *a completed item*
- Example: consider the production E -> E + T then

[E -> .E + T], [E -> E. + T], [E -> E + .T], [E -> E + T.] are items.

Out of them:

[E -> .E + T] is an *initial Item*

[E -> E + T.] is a *completed Item*

# Closure

- If I is a set of items, then *Closure(I)* is given by
  1) I ϵ Closure (I)
  2) If [A -> α.Bγ] is in the Closure(I) so far and B->β then
     [B -> .β] is in the Closure(I)
- Example: consider the following "Expression Grammar"

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

- Compute: Closure ( [T -> .T *F])

Closure ( I = { [T -> .T *F]}) = { [T -> .T *F], [T -> .T * F], [T -> .F],
[F -> .id], [F -> .(E)] } = { [T -> .T * F], [T -> .F], [F -> .id], [F -> .(E)] }

- Compute Closure ( [E -> E .+T] )

Closure ( [E -> E .+T] ) = { [E -> E .+ T] }

- Compute Closure ( [T -> T * .F], [ F-> id.] )

Closure ( [T -> T * .F], [ F-> id.] ) = { [T -> T * .F], [F -> id.], [F-> . (E)], [F -> .id] }

# State Transitions

- Transition are determined by the rules of the grammar and the items obtained from it
- We look now how we can have transitions in between items
- Moving the dot after a terminal symbol is straightforward: match the top of the stack to the current input token and process both
- But moving the dot preceding a nonterminal is not straightforward: For example, if we have [E->.E+T]
- Then we need to determines the closure of the item and we select one of the items in the closure that will help us process correctly the current input token
- Nonterminals have rules and we need to find out which set of rules need to be applied to reach a string of symbols that start with a terminal
- Recall the Closure of a rule: Consider a set to contain the rule only. Then recursively apply until no more changes: For all items in the set, if the dot precedes another nonterminal D, then add to the set all initial items built from the rules with D in the LHS , unless they are already in the set.
- We look at what symbols (terminals and nonterminals) come after the dot and we consider them when writing the transition table

# Simple LR (SLR)

- Now we are ready to construct the Parsing Table

**Step 1)** Augment the Grammar: we need to add an extra rule using a new nonterminal when we have more than one rule for the starting symbol (i.e. having the starting symbol in the LHS)

- Initial grammar: (1) E->E+T (2) E->E-T (3) E->T (4) T->T*F (5) T->T/F (6) T->F (7) F->(E) (8) F->i
- Augmented grammar: (0) Z->E (1) E->E+T (2) E->E-T (3) E->T (4) T->T*F (5) T->T/F (6) T->F (7) F->(E) (8) F->i

**Step 2)** Construct the transition sets:

- We start with State 0 which corresponds to the empty stack and the closure of the initial item derived from Rule 0: Closure([Z->.E])
- We need to determine the transitions from State 0 for various symbols of the grammar
- Some symbols have no transition, and the next state is the error state
- In other cases, we will respond to the next input token by a reduction and moving to a different state

- Consider the earlier grammar to which we add: Z->E

(0) Z->E (1) E->E+T (2) E->T (3) T->T*F (4) T->F (5) F->(E) (6) F->id

- We look now how we can have transitions in between items
  - Between the items [F->.(E)] and [F->(.E)] the transition is dictated by the symbol '(': moving the dot means shifting '(' onto the stack and have a change of states
  - But if the current state is [Z->.E] which represents *the starting state* or state 0, and in case our input is "(id)" then out of the two initial items [F->.(E)] and [F->.id] that are in Closure ([Z->.E]) we choose to select [F->.(E)].
- State 0: Closure([Z->.E]) = { [Z->.E], [E->.E+T], [E->.T], [T->.T*F], [T->.F], [F->.(E)], [F->.id] }
- But transitioning from [Z->.E] to [F->.(E)] is done without "consuming" any input symbol, so it can be considered as an **ε**-transition for a NFA; Closure ([Z->.E]) is also called **ε**-*closure*([Z->.E])
- We look at what symbols (terminals and nonterminals) come after the dot and we consider them when writing the transition table
- For state 0, the symbols are: E, T, F, 'id', and '('

- To construct then a row in the state table, for every symbol X in the grammar, we find the state reachable via that symbol as follows:
  - If the current state is V[$\alpha$] then we find V[$\alpha$X] by processing the items in V[$\alpha$]:
    1. For every item [C->a.Xb], include [C->aX.b] in V[$\alpha$X] (move the dot past X)
    2. Apply the closure rule for V[$\alpha$X] until nothing can be added.
- Transition out of state 0 and which a dot is followed by E are: [Z->.E], [E->.E+T], and

V[E]={ [Z->E.], [E->E.+T] }. State 1.

V[T]={ [E->T.], [T->T.*F] }. State 2.

V[F] = { [T → F.] }. State 3.

For '(', we have [F->.(E)]. Advancing the dot gives us [F->(.E)]. But now we have a nonterminal after the dot so we need to compute Closure([F->(.E)]):

V[( ) = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .i] }. State 4.

V[id] = { [F → i.] }. State 5.

- We note that when we apply Rule 1 above, we move the dots. Rule 1 above corresponds to transitions that push symbols onto the stack and we show that by moving the dot.
- When we apply Rule 2 above, no symbols are pushed onto the stack so the dot stays.

| State | i | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | | | 4 | | | 1 | 2 | 3 |
| 1 | To be completed later | | | | | | | | |
| 2 | | | | | | | | | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |

- ~~We do now the row for State 1. The transitions from State 1 are { [Z-> E.], [E -> E.+T] }~~ so the only symbol is + since there is nothing after the dot in [Z->E.]:

V[E+] = { [E -> E+.T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .id] }. State 6.

- We do now the row for state 2. The transitions from State 2 are { [E -> T.], [T -> T.*F] } so the only symbol is * since there is nothing after the dot in [E->T.]:

V[T*] = { [T -> T*.F], [F -> .(E)], [F -> .id] }. State 7.

- We do now state 3. The transition from State 3 is only V[F]={ [T -> F.] } so row 3 is blank.
- Transitions from State 4 are { [F -> (.E)], [E -> .E+T], [E -> .T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .id] }

V[E] = { [F -> (E.)], [E -> E.+T] }. State 8.

V[T] = { [E -> T.], [T -> T.*F] }. State 2

V[F] = { [T -> F.] }. State 3.

V[( ) = State 4

V[i] = { [F -> i.] } = State 5.

- Transitions from State 6 = { [E -> E+.T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .i] }

V[T] = { [E -> E+T.], [T -> T.*F] }. State 9.

V[F] = { [T -> F.] }. State 3.

V[( ] = State 4.

V[i]= State 5.

- Transitions from State 7 = { [T -> T*.F], [F -> .(E)], [F -> .i] }

V[F] = { [T -> T*F.] }. State 10.

V[( ] = State 4.

V[i) = State 5.

- Transitions from State 8 = { [F -> (E.)], [E -> E.+T] }

V[) ] = { [F -> (E).] }. State 11.

V[+ ] = State 6

- Transitions from State 9 = { [E -> E+T.], [T -> T.*F] }

V[*] = { [T -> T*.F], [F -> .(E)], [F -> .id] } = State 7.

- No transitions from States 10 and 11. FINISHED.

# Step3) Construct the Transition Table

**i1 = N (i0, E) = { [E' □ E.], [E' □ E.+T] }**
**i2 = N (i0, T) = { [E □ T.], [T □ T.*F] }**
**i3 = N (i0, F) = { [T □ F.] }**
**i4 = N (i0, ( ) = { [F □ (.E)], [E □ .E+T], [E □ .T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }**
**i5 = N (i0, id) = { [F □ id.] }**
**i6 = N (i1, +) = { [E □ E+.T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }**
**i7 = N (i2, *) = { [T □ T*.F], [F □ .(E)], [F □ .id] }**
**i8 = N (i4, E) = { [F □ (E.)], [E □ E.+T] }**
**N (i4, T) = { [E □ T.], [T □ T.*F] } = i2, N (i4, F) = i3, N (i4, ( ) = i4, N (i4, id) = i5**
**i9 = N (i6, T) = { [E □ E+T.], [T □ T.*F] }, N (i6, F) = i3, N (i6, ( ) = i4, N (i6, id) = i5**
**i10 = N (i7, F) = { [T □ T*F.] }, N (i7, ( ) = i4, N (i7, id) = i5**
**i11 = N (i8, ) ) = { [F □ (E).] }, N (i8, + ) = i6, N (i9, *) = i7**

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | | | 4 | | | 1 | 2 | 3 |
| 1 | | 6 | | | | | | | |
| 2 | | | 7 | | | | | | |
| 3 | | | | | | | | | |
| 4 | 5 | | | 4 | | | 8 | 2 | 3 |
| 5 | | | | | | | | | |
| 6 | 5 | | | 4 | | | | 9 | 3 |
| 7 | 5 | | | 4 | | | | | 10 |
| 8 | | 6 | | | 11 | | | | |
| 9 | | | 7 | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |

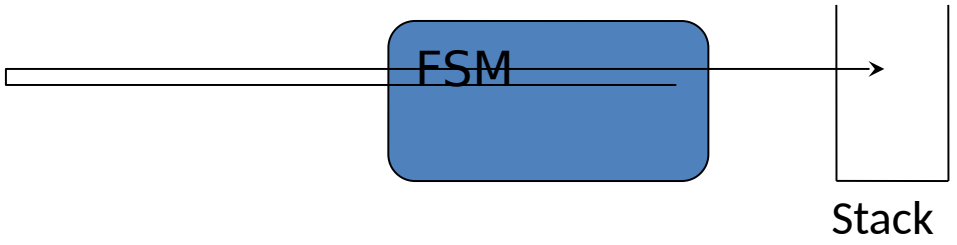
# 4.3 The LR Parser

- LR parser is the most powerful parser we will consider, i.e. can be build for any grammar we presented in class
- It handles the widest variety of CFGs, including all grammars that have predictive parsers or precedence parsers
- It works fast and it detects errors as soon as the first incorrect token is encountered
- It is easy to extend an LR parser to incorporate intermediate code generation
- General LR parser are called *LR(k)* parsers: Left-to-right scan of tokens, Rightmost derivation, look ahead *k* characters in the input
- Hopcroft and Ullman (1979) showed that any deterministic CFL can be handled by an LR(1) parser

- An LR parser uses a table; the table has two parts, the *action* part and the *go-to* part

| LR Parsing Table (consists of 3 parts) | | |
|---|---|---|
| States | Terminals (Action part) | Nonterminals (GoTo part) |
| | | |
| | | |
| | | |
| | | |

- Recall that a CFL is a Type 2 language and is accepted by a pushdown automaton (also called a *stack automaton*)
- A PDA has a finite-state controller and a stack



- In the LR parser, the parser table is the state table of the finite-state controller
- Each stack entry consists of a state and a grammar symbol, but in a practical LR parser, only the state is stored
- We include the production number in the table and not the actual production

Productions:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

| | Action Part | | | | | | GOTO Part | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

4 types of entries:

1. `Sn`: shift the input token into the stack and enter state n
2. `Rn`: reduce by means of Rule #n by popping the RHS with states as well up to the end of the RHS, then consider the state which is now on top of the stack (called tss), push the LHS, change the state by looking into go-to part of the table[tss, LHS] and push the new state into the stack
3. `n`: the new state due to some reduce action
4. `ACCT`: accept
5. Blank entry: error

**Driver:**

Place $ at the end of the input string

Push state 0 (the starting state) onto the stack

Repeat

Let $q_m$ be the current state (at the top of the stack) and i the incoming token

Find x = Table [$q_m$ , i];

Case x of

$S(q_n)$ : Shift (i.e. push) i onto stack and enter state $q_n$, i.e., push ($q_n$);

R(n): Reduce by means of Production #n by popping the symbols of the RHS of Rule #n (together with the states following them)

Let $q_j$ be the top of the stack state after reduce

Push the LHS L onto the stack

Push $q_k$ = Table [$q_j$ , L] onto the stack

ACCT: Accept. Parsing is complete

Error: error condition

Until ACCT or Error

Productions:

1. E --> E + T
2. E --> T
3. T --> T * F
4. T --> F
5. F --> (E)
6. F --> id

| | Action Part | | | | | | GOTO Part | | |
|---|---|---|---|---|---|---|---|---|---|
| State | id | + | * | ( | ) | $ | E | T | F |
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

**Example of Parsing id+id$**

| Stack | Input | Table Entry | Action |
|---|---|---|---|
| 0 | id+id$ | S5 | Push(id); push(5) |
| 0id5 | +id$ | R6 | F-> id; Table[0,F]=3 |
| 0F3 | +id$ | R4 | T -> F; Table[0,T] = 2 |
| 0T2 | +id$ | R2 | E -> T; Table[0,E] = 1 |
| 0E1 | +id$ | S6 | Push(+); push(6) |
| 0E1+6 | id$ | S5 | Push(id); push(5) |
| 0E1+6id5 | $ | R6 | F->id; Table[6,F]= 3 |
| 0E1+6F3 | $ | R4 | T->F; Table[6,T] = 9 |
| 0E1+6T9 | $ | R1 | E -> E+T; Table[0,E]=1 |
| 0E1 | $ | | ACCT |

# Constructing an LR Parser

- Three stages, of increasing complexity:
  - Simple LR (SLR)
  - Canonical LR
  - Lookahead LR (LALR)
- SLR Parser is powerful enough to generate the parse table shown in the previous example
- Concepts needed: states of a parser, items, closure, state transitions
- States of the parser:
  - the stack stores states and symbols
  - a state corresponds to a particular sequence of symbols at the top of the stack
  - The current state depends on the topmost stack symbol and the current input token

# Items

- In the process of parsing a input string, handles need to be built in the top of the stack
- Handles appear in stages, not all at once, and sometimes building some part of the handle involves applying multiple rules while the rest of the handle sits in the stack
- To keep track how far we have gotten through the growth of a handle we use *items*
- An *item* is a production with a place marker, written as a dot, and inserted somewhere in the RHS; items are enclosed in [ …] to not be confused with rules
- An item whose RHS starts with a . is called *an initial item*
- An item whose RHS starts with a . is called *a completed item*
- Example: consider the production E -> E + T then

[E -> .E + T], [E -> E. + T], [E -> E + .T], [E -> E + T.] are items.

Out of them:

[E -> .E + T] is an *initial Item*

[E -> E + T.] is a *completed Item*

# Closure

- If I is a set of items, then *Closure(I)* is given by

  1) I ϵ Closure (I)

  2) If [A -> α.Bγ] is in the Closure(I) so far and B->β then [B -> .β] is in the Closure(I)

- Example: consider the following "Expression Grammar"

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

- Compute: Closure ( [T -> .T *F])

Closure ( I = { [T -> .T *F]}) = { [T -> .T *F], [T -> .T * F], [T -> .F], [F -> .id], [F -> .(E)] } = { [T -> .T * F], [T -> .F], [F -> .id], [F -> .(E)] }

- Compute Closure ( [E -> E .+T] )

Closure ( [E -> E .+T] ) = { [E -> E .+ T] }

- Compute Closure ( [T -> T * .F], [ F-> id.] )

Closure ( [T -> T * .F], [ F-> id.] ) = { [T -> T * .F], [F -> id.], [F-> . (E)], [F -> .id] }

# State Transitions

- Transition are determined by the rules of the grammar and the items obtained from it
- We look now how we can have transitions in between items
- Moving the dot after a terminal symbol is straightforward: match the top of the stack to the current input token and process both
- But moving the dot preceding a nonterminal is not straightforward: For example, if we have [E->.E+T]
- Then we need to determines the closure of the item and we select one of the items in the closure that will help us process correctly the current input token
- Nonterminals have rules and we need to find out which set of rules need to be applied to reach a string of symbols that start with a terminal
- Recall the Closure of a rule: Consider a set to contain the rule only. Then recursively apply until no more changes: For all items in the set, if the dot precedes another nonterminal D, then add to the set all initial items built from the rules with D in the LHS , unless they are already in the set.
- We look at what symbols (terminals and nonterminals) come after the dot and we consider them when writing the transition table

# Simple LR (SLR)

- Now we are ready to construct the Parsing Table

**Step 1)** Augment the Grammar: we need to add an extra rule using a new nonterminal when we have more than one rule for the starting symbol (i.e. having the starting symbol in the LHS)

- Initial grammar: (1) E->E+T (2) E->E-T (3) E->T (4) T->T*F (5) T->T/F (6) T->F (7) F->(E) (8) F->i
- Augmented grammar: (0) Z->E (1) E->E+T (2) E->E-T (3) E->T (4) T->T*F (5) T->T/F (6) T->F (7) F->(E) (8) F->i

**Step 2)** Construct the transition sets:

- We start with State 0 which corresponds to the empty stack and the closure of the initial item derived from Rule 0: Closure([Z->.E])
- We need to determine the transitions from State 0 for various symbols of the grammar
- Some symbols have no transition, and the next state is the error state
- In other cases, we will respond to the next input token by a reduction and moving to a different state

- Consider the earlier grammar to which we add: Z->E

(0) Z->E (1) E->E+T (2) E->T (3) T->T*F (4) T->F (5) F->(E) (6) F->id

- We look now how we can have transitions in between items
  - Between the items [F->.(E)] and [F->(.E)] the transition is dictated by the symbol '(': moving the dot means shifting '(' onto the stack and have a change of states
  - But if the current state is [Z->.E] which represents *the starting state* or state 0, and in case our input is "(id)" then out of the two initial items [F->.(E)] and [F->.id] that are in Closure ([Z->.E]) we choose to select [F->.(E)].
- But transitioning from [Z->.E] to [F->.(E)] is done without "consuming" any input symbol, so it can be considered as an **ε**-transition for a NFA; Closure ([Z->.E]) is also called **ε**-*closure*([Z->.E])
- We look at what symbols (terminals and nonterminals) come after the dot and we consider them when writing the transition table
- For state 0, the symbols are: E, T, F, 'id', and '('

- To construct then a row in the state table, for every symbol X in the grammar, we find the state reachable via that symbol as follows:
  - If the current state is V[$\alpha$] then we find V[$\alpha$X] by processing the items in V[$\alpha$]:
    1. For every item [C->a.Xb], include [C->aX.b] in V[$\alpha$X] (move the dot past X)
    2. Apply the closure rule for V[$\alpha$X] until nothing can be added.
- Transition out of state 0 and which a dot is followed by E are: [Z->.E], [E->.E+T], and

V[E]={ [Z->E.], [E->E.+T] }. State 1.

V[T]={ [E->T.], [T->T.*F] }. State 2.

V[F] = { [T → F.] }. State 3.

For '(', we have [F->.(E)]. Advancing the dot gives us [F->(.E)]. But now we have a nonterminal after the dot so we need to compute Closure([F->(.E)]):

V[( ) = { [F → (.E)], [E → .E+T], [E → .T], [T → .T*F], [T → .F], [F → .(E)], [F → .i] }. State 4.

V[id] = { [F → i.] }. State 5.

- We note that when we apply Rule 1 above, we move the dots. Rule 1 above corresponds to transitions that push symbols onto the stack and we show that by moving the dot.
- When we apply Rule 2 above, no symbols are pushed onto the stack so the dot stays.

| State | i | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | | | 4 | | | 1 | 2 | 3 |
| 1 | To be completed later | | | | | | | | |
| 2 | | | | | | | | | |
| 3 | | | | | | | | | |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |

- ~~We do now the row for State 1. The transitions from State 1 are { [Z-> E.], [E -> E.+T] }~~ so the only symbol is + since there is nothing after the dot in [Z->E.]:

V[E+] = { [E -> E+.T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .id] }. State 6.

- We do now the row for state 2. The transitions from State 2 are { [E -> T.], [T -> T.*F] } so the only symbol is * since there is nothing after the dot in [E->T.]:

V[T*] = { [T -> T*.F], [F -> .(E)], [F -> .id] }. State 7.

- We do now state 3. The transition from State 3 is only V[F]={ [T -> F.] } so row 3 is blank.
- Transitions from State 4 are { [F -> (.E)], [E -> .E+T], [E -> .T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .id] }

V[E] = { [F -> (E.)], [E -> E.+T] }. State 8.

V[T] = { [E -> T.], [T -> T.*F] }. State 2

V[F] = { [T -> F.] }. State 3.

V[( ) = State 4

V[i] = { [F -> i.] } = State 5.

- Transitions from State 6 = { [E -> E+.T], [T -> .T*F], [T -> .F], [F -> .(E)], [F -> .i] }

V[T] = { [E -> E+T.], [T -> T.*F] }. State 9.

V[F] = { [T -> F.] }. State 3.

V[( ] = State 4.

V[i]= State 5.

- Transitions from State 7 = { [T -> T*.F], [F -> .(E)], [F -> .i] }

V[F] = { [T -> T*F.] }. State 10.

V[( ] = State 4.

V[i) = State 5.

- Transitions from State 8 = { [F -> (E.)], [E -> E.+T] }

V[) ] = { [F -> (E).] }. State 11.

V[+ ] = State 6

- Transitions from State 9 = { [E -> E+T.], [T -> T.*F] }

V[*] = { [T -> T*.F], [F -> .(E)], [F -> .id] } = State 7.

- No transitions from States 10 and 11. FINISHED.

# Step3) Construct the Transition Table

**i1 = N (i0, E) = { [E' □ E.], [E' □ E.+T] }**
**i2 = N (i0, T) = { [E □ T.], [T □ T.*F] }**
**i3 = N (i0, F) = { [T □ F.] }**
**i4 = N (i0, ( ) = { [F □ (.E)], [E □ .E+T], [E □ .T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }**
**i5 = N (i0, id) = { [F □ id.] }**
**i6 = N (i1, +) = { [E □ E+.T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }**
**i7 = N (i2, *) = { [T □ T*.F], [F □ .(E)], [F □ .id] }**
**i8 = N (i4, E) = { [F □ (E.)], [E □ E.+T] }**
**N (i4, T) = { [E □ T.], [T □ T.*F] } = i2, N (i4, F) = i3, N (i4, ( ) = i4, N (i4, id) = i5**
**i9 = N (i6, T) = { [E □ E+T.], [T □ T.*F] }, N (i6, F) = i3, N (i6, ( ) = i4, N (i6, id) = i5**
**i10 = N (i7, F) = { [T □ T*F.] }, N (i7, ( ) = i4, N (i7, id) = i5**
**i11 = N (i8, ) ) = { [F □ (E).] }, N (i8, + ) = i6, N (i9, *) = i7**

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 5 | | | 4 | | | 1 | 2 | 3 |
| 1 | | 6 | | | | | | | |
| 2 | | | 7 | | | | | | |
| 3 | | | | | | | | | |
| 4 | 5 | | | 4 | | | 8 | 2 | 3 |
| 5 | | | | | | | | | |
| 6 | 5 | | | 4 | | | | 9 | 3 |
| 7 | 5 | | | 4 | | | | | 10 |
| 8 | | 6 | | | 11 | | | | |
| 9 | | | 7 | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |

# Step4) Write out the table

1) Change the entry m in the Action Part to "Sm"

2) If a state contains a completed item [C->α.] of the rule C->α which is the $n^{th}$ rule in the grammar, then for all the inputs in Follow(C), the action is "Reduce n".

(*) Special Case: If a state q contains the item [Z –> S.] which is the completed item for R0, then the entry for [state q, $] is "ACCT"

# 1. Change  m to Sm in Action Part

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | | | | |
| 2 | | | S7 | | | | | | |
| 3 | | | | | | | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | | | | | | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | | S7 | | | | | | |
| 10 | | | | | | | | | |
| 11 | | | | | | | | | |

i1 = N (i0, E) = { [E' □ E.], [E' □ E.+T] } (Special case)
i2 = N (i0, T) = { [E □ T.], [T □ T.*F] }
i3 = N (i0, F) = { [T □ F.] }
i4 = N (i0, ( ) = { [F □ (.E)], [E □ .E+T], [E □ .T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }
i5 = N (i0, id) = { [F □ id.] }
i6 = N (i1, +) = { [E □ E+.T], [T □ .T*F], [T □ .F], [F □ .(E)], [F □ .id] }
i7 = N (i2, *) = { [T □ T*.F], [F □ .(E)], [F □ .id] }
i8 = N (i4, E) = { [F □ (E.)], [E □ E.+T] }
i9 = N (i6, T) = { [E □ E+T.] , [T □ T.*F] }
i10 = N (i7, F) = { [T □ T*F.] }
i11 = N (i8, ) ) = { [F □ (E).] }

R0. E' -> E
R1. E -> E + T
R2. E -> T
R3. T -> T * F
R4. T -> F
R5. F -> (E)
R6. F -> id

**2) Find for completed item in each state**

- **State 2 has a completed item [E -> T.] for R2**
  Follow (E) = {+, $, )} => Table [2, {+,$,)}] = R2
- **State 3 has a completed item [T -> F.] for R4**
  Follow (T) = { *, +,), $} => Table[3, { *, +,), $} ] = R4
- **State 5 has a completed item [F -> id.] for R6**
  Follow(F) = { *, +,), $} => Table[5, { *, +,), $} ] = R6
- **State 9 has a completed item [E -> E+T.] for R1**
  Follow (E) = {+, $, )} => Table [9, {+,$,)}] = R1
- **State 10 has a completed item [T -> T*F.] for R3**
  Follow (T) = { *, +,), $} => Table[10, { *, +,), $} ] = R3
- **State 11 has a completed item [F -> (E).] for R5**
  Follow(F) = { *, +,), $} => Table[11, { *, +,), $} ] = R5

(*) State 1 has a completed item [E ' -> E.] => Table [1, $] = ACCT

**Result:**

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| **2** | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

**Ex 2)**

**Consider the following Grammar**

**R0) S' -> S**
**R1) S -> E = E**
**R2) S -> id**
**R3) E -> E + id**
**R4) E -> id**

**Step2) Transition Sets**

**I0 = Closure ( [S' -> .S] ) = { [S' ->.S], [ S -> .E = E], [S -> .id],**
**[E -> . E + id], [E -> .id]}**
**I1 = N (I0, S) = { [S' -> S.]}**
**I2 = N (I0, E) = { [S -> E . = E], [E -> E. + id]}**
**I3 = N (I0, id) = { [S -> id.], [E -> id.] }**
**I4 = N (I2, =) = { [S -> E = .E], [E ->. E + id], [E -> .id] }**
**I5 = N (I2, +) = { [E -> E + .id] }**
**I6 = N (I4, E) = { [S -> E = E.], [E -> E . + id]}**
**I7 = N (I4, id) = { [E -> id.]}**
**I8 = N (I5, id) = { [E -> E + id.] }**
**N (I6, +) = { [E -> E + .id] } = I5**

**I0 = Closure ( [S' -> .S] ) = { [S' ->.S], [ S -> .E = E], [S -> .id],**
**[E -> . E + id], [E -> .id]}**

**I1 = N (I0, S) = { [S' -> S.]}**

**I2 = N (I0, E) = { [S -> E . = E], [E -> E. + id]}**

**I3 = N (I0, id) = { [S -> id.], [E -> id.] }**

**I4 = N (I2, =) = { [S -> E = .E], [E ->. E + id], [E -> .id] }**

**I5 = N (I2, +) = { [E -> E + .id] }**

**I6 = N (I4, E) = { [S -> E = E.], [E -> E . + id]}**

**I7 = N (I4, id) = { [E -> id.]}**

**I8 = N (I5, id) = { [E -> E + id.] }**

**I5 = N (I6, +) = { [E -> E + .id] }**

## Step 3) Transition Table

| | Id | = | + | $ | S | E |
|---|---|---|---|---|---|---|
| 0 | 3 | | | | 1 | 2 |
| 1 | | | | | | |
| 2 | | 4 | 5 | | | |
| 3 | | | | | | |
| 4 | 7 | | | | | 6 |
| 5 | 8 | | | | | |
| 6 | | | 5 | | | |
| 7 | | | | | | |
| 8 | | | | | | |

## Step4) Write out the Table

**1) m => Sm in Action Part**

**2) Complete the Reduce rules: Let's consider State 3**

**State 3 = { [S -> id.], [E -> id.] } has two completed items:**

**[S -> id.] for R2 and [E -> id.] for R4**

**Compute Follow (S) = {$} and Follow (E) = {=, +, $}**

**=> Table [3, $] = R2 and Table[3, {=, +, $}] = R4**

**R0) S' -> S**
**R1) S -> E = E**
**R2) S -> id**
**R3) E -> E + id**
**R4) E -> id**

| | Id | = | + | $ | S | E |
|---|---|---|---|---|---|---|
| 0 | S3 | | | | 1 | 2 |
| 1 | | | | ACCT | | |
| 2 | | S4 | S5 | | | |
| 3 | | R4 | R4 | R2/R4 | | |
| 4 | S7 | | | | | 6 |
| 5 | S8 | | | | | |
| 6 | | | S5 | | | |
| 7 | | | | | | |
| 8 | | | | | | |

**There is a conflict in Table[3, $] = {R2/R4} Called Reduce/Reduce Conflict.**

**That is, if the stack has State 3 and input is $, there are two possibilities R2 and R4**

| | Id | = | + | $ | S | E |
|---|---|---|---|---|---|---|
| 0 | S3 | | | | 1 | 2 |
| 1 | | | | ACCT | | |
| 2 | | S4 | S5 | | | |
| 3 | | R4 | R4 | (R2/R4) | | |
| 4 | S7 | | | | | 6 |
| 5 | S8 | | | | | |
| 6 | | | S5 | | | |
| 7 | | | | | | |
| 8 | | | | | | |

**Let's examine further: Consider a string "a" and try to parse**

| Stack | Input | Action |
|---|---|---|
| 0 | a$ | S3 |
| 0a3 | $ | R2 or R4 (Choose R2) S -> id |
| 0S1 | $ | ACCT |

| Stack | Input | Action |
|---|---|---|
| 0 | a$ | S3 |
| 0a3 | $ | R2 or R4 (Choose R4) E -> id |
| 0E2 | $ | Error???? |

**What does it mean ? => R4 should NOT be allowed in the first place**

**The conflict occurred because the technique we utilized is not powerful enough. We need a more powerful method**

**Canonical LR**

# LR Parsing

- An LL(1) PDA has only one state!
  - well, actually two; it needs a second one to accept with, but that's all (it's pretty simple)
  - all the arcs are self loops; the only difference between them is the choice of whether to push or pop
  - the final state is reached by a transition that sees EOF on the input and the stack

# LR Parsing

- An SLR/LALR/LR PDA has multiple states
  - it is a "recognizer," not a "predictor"
  - it builds a parse tree from the bottom up
  - the states keep track of which productions we *might* be in the middle
- SLR parsing is based on
  - Shift
  - Reduce

  and also
  - Shift & Reduce (for optimization)

# 4.4 Error Handling

**If there is a syntax error, your parser should do**

**a) Generate a "meaningful" error message**

**b) Recover from the error**

**a) Error message**

**Assume LR parser, current state is N and token = i, then**

**Table[N, i] = empty is an error.**

**In this case, we could look at the Table [N, x] = Not empty and announce all x tokens**

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

Ex. Current state = 5 and token ( => error => look for entry

**b) Error recovery is more "tricky"**

**One way to do is to insert a fake token e.g., a(b+c) => will generate an error message of missing an operator => Insert any operator and move on**

| State | Id | + | * | ( | ) | $ | E | T | F |
|---|---|---|---|---|---|---|---|---|---|
| 0 | S5 | | | S4 | | | 1 | 2 | 3 |
| 1 | | S6 | | | | ACCT | | | |
| 2 | | R2 | S7 | | R2 | R2 | | | |
| 3 | | R4 | R4 | | R4 | R4 | | | |
| 4 | S5 | | | S4 | | | 8 | 2 | 3 |
| 5 | | R6 | R6 | | R6 | R6 | | | |
| 6 | S5 | | | S4 | | | | 9 | 3 |
| 7 | S5 | | | S4 | | | | | 10 |
| 8 | | S6 | | | S11 | | | | |
| 9 | | R1 | S7 | | R1 | R1 | | | |
| 10 | | R3 | R3 | | R3 | R3 | | | |
| 11 | | R5 | R5 | | R5 | R5 | | | |

Ex. Current state = 5 and token ( => fake token +, *

**But what if the parser assumes wrong => But what if the parser assumes wrong => (e.g., state 4, multiple different entries => Will generate message which will be hard to understand**

# 4.5 Symbol Table

**A Database for symbol that occur during the compilation process.**
**It also provides set of procedures such as, lookup(), insert(), remove(), list()  etc**

## 4.5.1 Organization of the table

- **Array :**

Adv: Simplest and no overhead (pointers)
Disadv: fixed size, search O(N)

- **Linked List**

Adv: Expandable
DisAdv: Overhead (Pointers), search  O(N)

- **Binary Search Tree**

Adv: Expandable, Search  O(logN)
Disavd: Overhead (2 pointers), House-keeping for  Binary Search Tree

- **Hash Table (Most compiler use this)** = Bucket + LL

Adv: Expandable, Search  O(λ) = average bucket size
If hash function h(x) is good the bucket size will be good.
Some h(x) = mode function, middle square function etc

## 4.5.2 Scope issues

**To make variables visible correctly**

- **No Scope => All variables are global (Basic)**
- **Totally Separate => All local = No nesting is allowed**
- **Nested scope : Nesting allowed**

**How Symbol table handles the scopes:**

- **No scope: All names (variables must be distinct)**
- **Totally Separate: Same names are allowed if scopes are different**

- **Nest Scope (many PLs)**

**Use of dynamic (growing and shrinking) symbol table, i.e.,**

**If a scope is entered the table grows by the symbols of the scope**

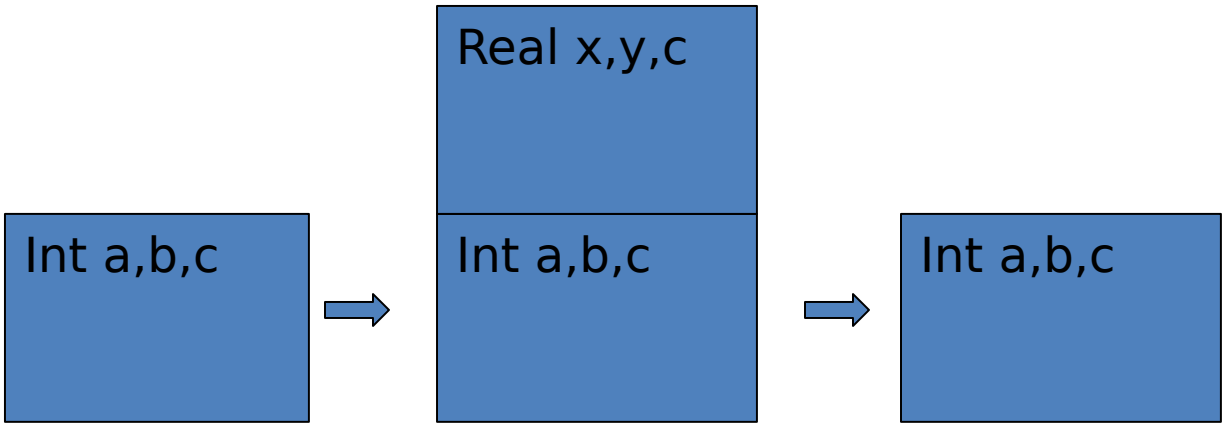
**And if scope is left the table shrinks (STACK)**

**Example:**

```
{
    int a,b, c;
    …….
        {
            real x,y,c;
            …..
            c= /* real */

        }

   …c = /* int */
}
```



# The Symbol Table

- Needed at every point of the compilation process
- It needs to be organized in such a way to obtain fast access to the information it holds and reflect the organization of the program
- Can be organized as:
  - a linked list
  - A BST
  - An hash table (an array accessed by hashing)
- Trade-off between the space memory needed and the access time
- Operations that need to be supported: lookup(), insert(), remove(), list()  etc

| | Advantage | Disadvantage |
|---|---|---|
| Array | simplest, minimum space among all other data structures because it uses no overhead (no pointers) | requires a fixed size, search must be sequential so O(N) |
| Linked List | expandable; it can self-organize and move up to the head of the list the item you searched for | overhead (stores pointers), most frequently used symbols will be at the front of the list but the time complexity for search remains O(N) |
| Binary Search Tree | expandable, search O(height) that could be in best case O(logN) but at random is approx. 1.39 lg(n), can be displayed in alphabetical order using inorder traversal | overhead (2 pointers per entry table), deleting makes the tree unbalanced and increases the search time. AVL or red-black trees can be used but they require a lot of house-keeping. |
| Hash table using chaining | Expandable, Search O(λ) | Depends on the hash function |

Hash table using chaining (most compiler use this) = array of pointers, and each pointer is the head of a linked list (a linked list is called chain)

The array size is fixed; a chain can have arbitrary length

Let λ be the load factor, i.e. the ratio between the total number of items stored and the number of entries in the array; then the average length of a chain is λ

# Storing Identifiers

- If the name is very long, one alternative is to concatenate all names into one long array, by marking the end of the identifier by a special character, and provide a pointer to the start of the identifier
- **Symbol table must provide either the identifier name or a pointer to the name, and a code indicating the data type of the object to which the entry refers**
- Other many additional fields: number of dimensions, their bounds, other specific data (e.g. for a record would be the number of fields, size of the record, additional info for variant record)
- A problem is the scope of the identifier in a program (discussed next)

# Scope Issues

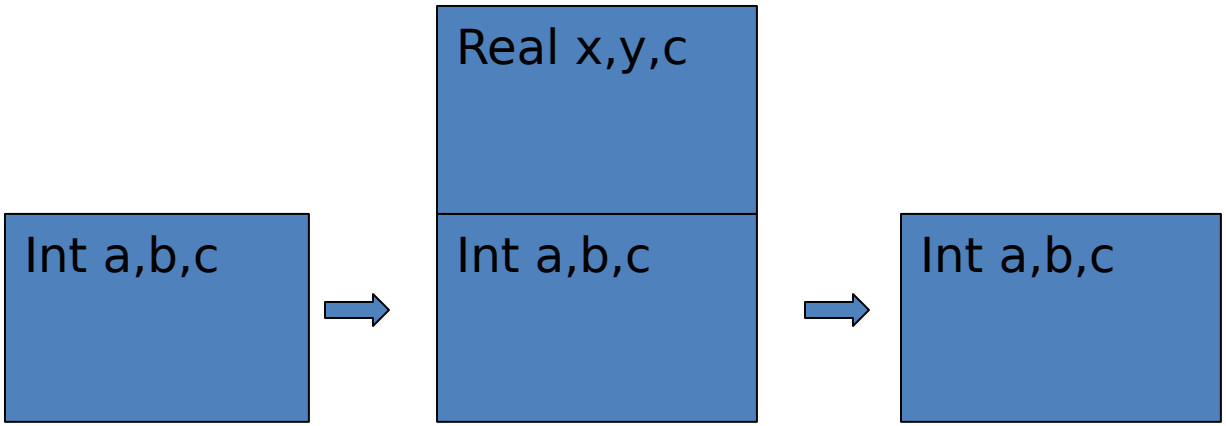
- To make variables visible correctly
  - No Scope => All variables are global (Basic)
  - Totally Separate => All local = No nesting is allowed
  - Nested scope : Nesting allowed
- How Symbol table handles the scopes:
  - No scope: All names (variables must be distinct)
  - Totally Separate: Same names are allowed if scopes are different
  - Nest Scope (many PLs)
    - Use of dynamic (growing and shrinking) symbol table, i.e. STACK
    - If a scope is entered the table grows by the symbol table of the scope
    - And if scope is left, the table shrinks (STACK)

**Example:**

```
{
    int a,b, c;
    …….
        {
            real x,y,c;
            …..
            c= /* real */

        }

   …c = /* int */
}
```



# The Symbol Table

- Needed at every point of the compilation process
- It needs to be organized in such a way to obtain fast access to the information it holds and reflect the organization of the program
- Can be organized as:
  - a linked list
  - A BST
  - An hash table (an array accessed by hashing)
- Trade-off between the space memory needed and the access time
- Operations that need to be supported: lookup(), insert(), remove(), list()  etc

| | Advantage | Disadvantage |
|---|---|---|
| Array | simplest, minimum space among all other data structures because it uses no overhead (no pointers) | requires a fixed size, search must be sequential so O(N) |
| Linked List | expandable; it can self-organize and move up to the head of the list the item you searched for | overhead (stores pointers), most frequently used symbols will be at the front of the list but the time complexity for search remains O(N) |
| Binary Search Tree | expandable, search O(height) that could be in best case O(logN) but at random is approx. 1.39 lg(n), can be displayed in alphabetical order using inorder traversal | overhead (2 pointers per entry table), deleting makes the tree unbalanced and increases the search time. AVL or red-black trees can be used but they require a lot of house-keeping. |
| Hash table using chaining | Expandable, Search O(λ) | Depends on the hash function |

Hash table using chaining (most compiler use this) = array of pointers, and each pointer is the head of a linked list (a linked list is called chain)

The array size is fixed; a chain can have arbitrary length

Let λ be the load factor, i.e. the ratio between the total number of items stored and the number of entries in the array; then the average length of a chain is λ

# Storing Identifiers

- If the name is very long, one alternative is to concatenate all names into one long array, by marking the end of the identifier by a special character, and provide a pointer to the start of the identifier
- **Symbol table must provide either the identifier name or a pointer to the name, and a code indicating the data type of the object to which the entry refers**
- Other many additional fields: number of dimensions, their bounds, other specific data (e.g. for a record would be the number of fields, size of the record, additional info for variant record)
- A problem is the scope of the identifier in a program (discussed next)

# Scope Issues

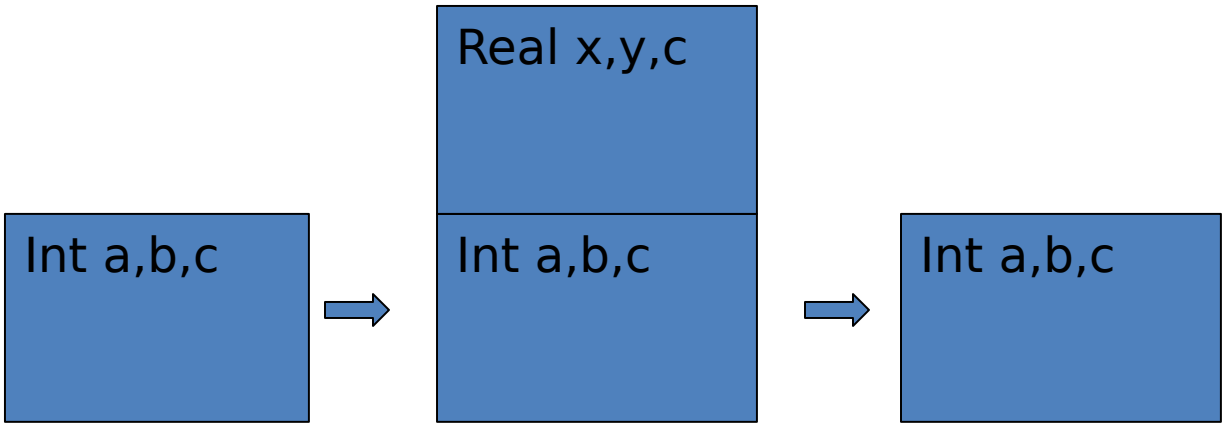
- To make variables visible correctly
  - No Scope => All variables are global (Basic)
  - Totally Separate => All local = No nesting is allowed
  - Nested scope : Nesting allowed
- How Symbol table handles the scopes:
  - No scope: All names (variables must be distinct)
  - Totally Separate: Same names are allowed if scopes are different
  - Nest Scope (many PLs)
    - Use of dynamic (growing and shrinking) symbol table, i.e. STACK
    - If a scope is entered the table grows by the symbol table of the scope
    - And if scope is left, the table shrinks (STACK)

**Example:**

```
{
    int a,b, c;
    …….
        {
            real x,y,c;
            …..
            c= /* real */

        }

   …c = /* int */
}
```



# Name, Scope, and Binding

- Fundamental to all programming languages is the ability to name data, i.e., to refer to data using symbolic identifiers rather than addresses
- Not all data is named! For example, dynamic storage in C or Pascal is referenced by pointers, not names
- A name is exactly what you think it is
  - Most names are identifiers
  - symbols (like '+') can also be names
- A binding is an association between two things, such as a name and the thing it names
- The *scope* of a binding is the part of the program (textually) in which the binding is active
- Binding Time is the point at which a binding is created or, more generally, the point at which any implementation decision is made
  - language design time
    - program structure, possible type
  - language implementation time
    - I/O, arithmetic overflow, type equality (if unspecified in manual)

# Scope Rules

- A *scope* is a program section of maximal size in which no bindings change, or at least in which no re-declarations are permitted (see below)
- In most languages with subroutines, we OPEN a new scope on subroutine entry:
  - create bindings for new local variables,
  - deactivate bindings for global variables that are re-declared (these variable are said to have a "hole" in their scope)
  - make references to variables

# Scope Rules

- On subroutine exit:
  - destroy bindings for local variables
  - reactivate bindings for global variables that were deactivated
- Algol 68:
  - ELABORATION = process of creating bindings when entering a scope
- Ada (re-popularized the term elaboration):
  - storage may be allocated, tasks started, even exceptions propagated as a result of the elaboration of declarations

# Scope Rules

- With STATIC (LEXICAL) SCOPE RULES, a scope is defined in terms of the physical (lexical) structure of the program
  - The determination of scopes can be made by the compiler
  - All bindings for identifiers can be resolved by examining the program
  - Typically, we choose the most recent, active binding made at compile time
  - Most compiled languages, C and Pascal included, employ static scope rules

# Scope Rules

- Note that the bindings created in a subroutine are destroyed at subroutine exit
  - The modules of Modula, Ada, etc., give you closed scopes without the limited lifetime
  - Bindings to variables declared in a module are inactive outside the module, not destroyed
  - The same sort of effect can be achieved in many languages with *own* (Algol term) or *static* (C term) variables (see Figure 3.5)

# Scope Rules

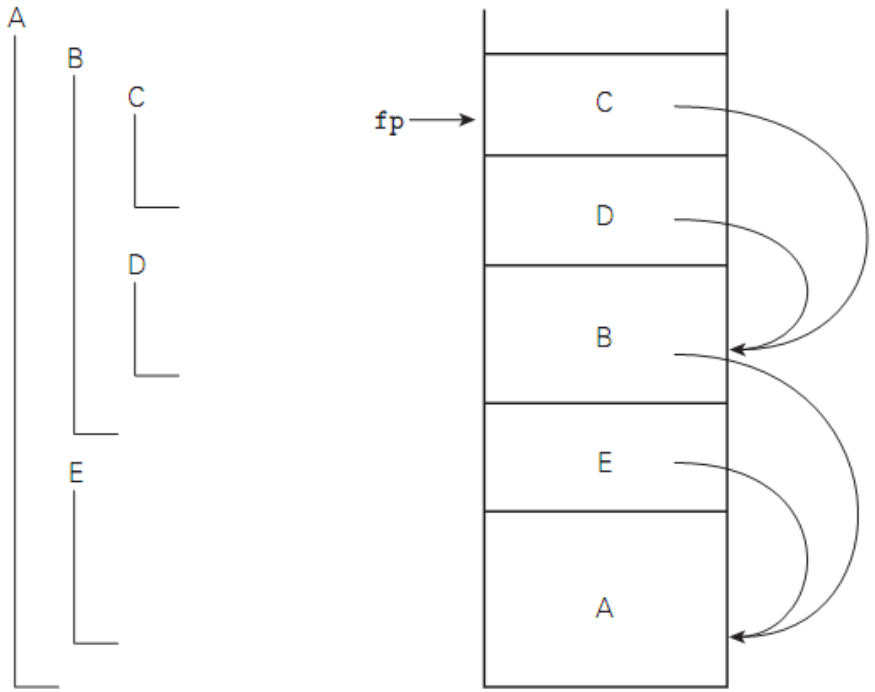



**Figure 3.5** **Static chains.** Subroutines A, B, C, D, and E are nested as shown on the left. If the sequence of nested calls at run time is A, E, B, D, and C, then the static links in the stack will look as shown on the right. The code for subroutine C can find local objects at known offsets from the frame pointer. It can find local objects of the surrounding scope, B, by dereferencing its static chain once and then applying an offset. It can find local objects in B's surrounding scope, A, by dereferencing its static chain twice and then applying an offset.

# Scope Rules
# Example: Static vs. Dynamic

- Static scope rules require that the reference resolve to the most recent, compile-time binding, namely the global variable `a`
- Dynamic scope rules, on the other hand, require that we choose the most recent, active binding at run time
  - Perhaps the most common use of dynamic scope rules is to provide implicit parameters to subroutines
  - This is generally considered bad programming practice nowadays
    - Alternative mechanisms exist
      - static variables that can be modified by auxiliary routines
      - default and optional parameters

# Scope Rules

- Dynamic scope rules are usually encountered in interpreted languages
  - early LISP dialects assumed dynamic scope rules.
- Such languages do not normally have type checking at compile time because type determination isn't always possible when dynamic scope rules are in effect

# Scope Rules
# Example: Static vs. Dynamic

```
program scopes (input, output );
var a : integer;
procedure first;
        begin a := 1; end;
procedure second;
     var a : integer;
     begin first; end;
begin
     a := 2; second; write(a);
end.
```

# Scope Rules
# Example: Static vs. Dynamic

- If static scope rules are in effect (as would be the case in Pascal), the program prints a `1`
- If dynamic scope rules are in effect, the program prints a `2`
- Why the difference? At issue is whether the assignment to the variable `a` in **`procedure`** `first` changes the variable `a` declared in the main program or the variable `a` declared in **`procedure`** `second`

# Lifetime and Storage Management

- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is *active* is its scope
- In addition to talking about the *scope of a binding*, we sometimes use the word *scope* as a noun all by itself, without an indirect object

# Binding

- The terms STATIC and DYNAMIC are generally used to refer to things bound before run time and at run time, respectively
- In general, early binding times are associated with greater efficiency
- Later binding times are associated with greater flexibility
- Compiled languages tend to have early binding times
- Interpreted languages tend to have later binding times

# Lifetime and Storage Management

- The period of time from creation to destruction is called the LIFETIME of a binding
  - If object outlives binding it's garbage
  - If binding outlives object it's a dangling reference
- The textual region of the program in which the binding is *active* is its scope
- In addition to talking about the *scope of a binding*, we sometimes use the word *scope* as a noun all by itself, without an indirect object

# Lifetime and Storage Management

- Storage Allocation mechanisms
  - –Static
  - –Stack
  - –Heap
- Static allocation for
  - –code
  - –globals
  - –static or own variables
  - –explicit constants (including strings, sets, etc)
  - –scalars may be stored in the instructions

# Lifetime and Storage Management

- Central stack for
  - –parameters
  - –local variables
  - –temporaries
- Why a stack?
  - –allocate space for recursive routines
    (not necessary in FORTRAN – no recursion)
  - –reuse space
    (in all programming languages)

# Lifetime and Storage Management

- Contents of a stack frame (cf., Figure 3.1)
  - arguments and returns
  - local variables
  - temporaries
  - bookkeeping (saved registers, line number static link, etc.)
- Local variables and arguments are assigned fixed OFFSETS from the stack pointer or frame pointer at compile time

# Lifetime and Storage Management

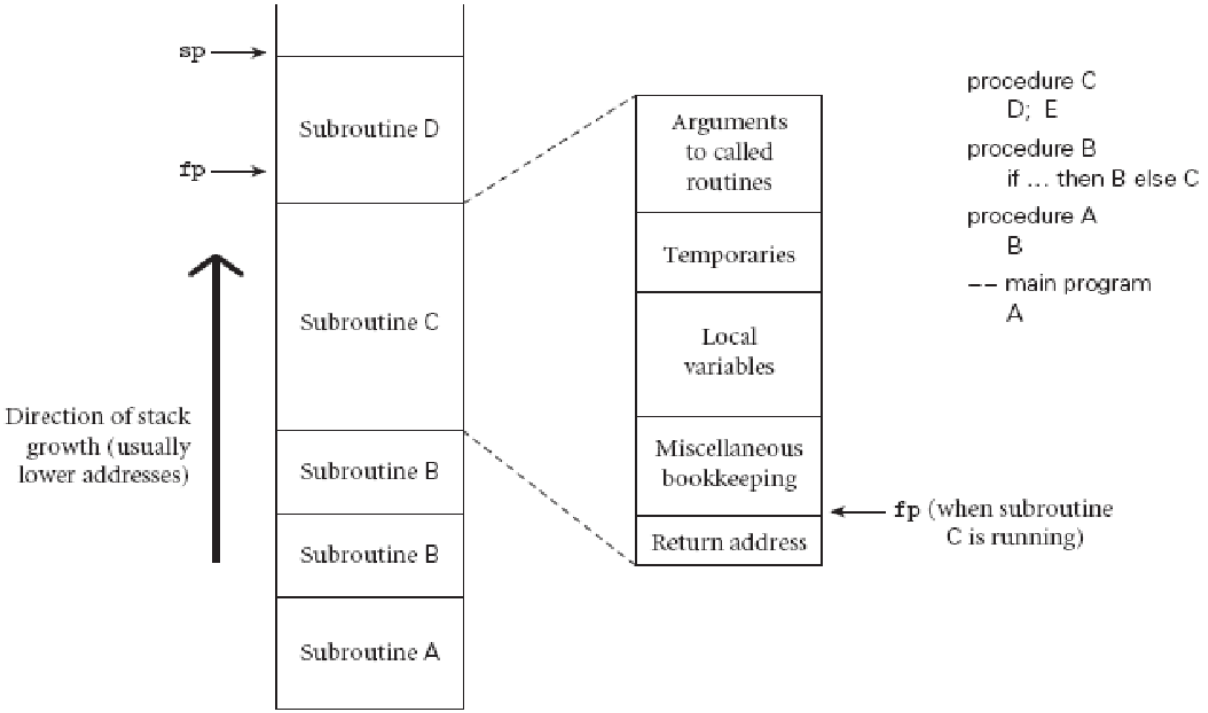



**Figure 3.1** Stack-based allocation of space for subroutines. We assume here that subroutines have been called as shown in the upper right. In particular, B has called itself once, recursively, before calling C. If D returns and C calls E, E's frame (activation record) will occupy the same space previously used for D's frame. At any given time, the stack pointer (`sp`) register points to the first unused location on the stack (or the last used location on some machines), and the frame pointer (`fp`) register points to a known location within the frame of the current subroutine. The relative order of fields within a frame may vary from machine to machine and compiler to compiler.

# Lifetime and Storage Management

- Heap for dynamic allocation

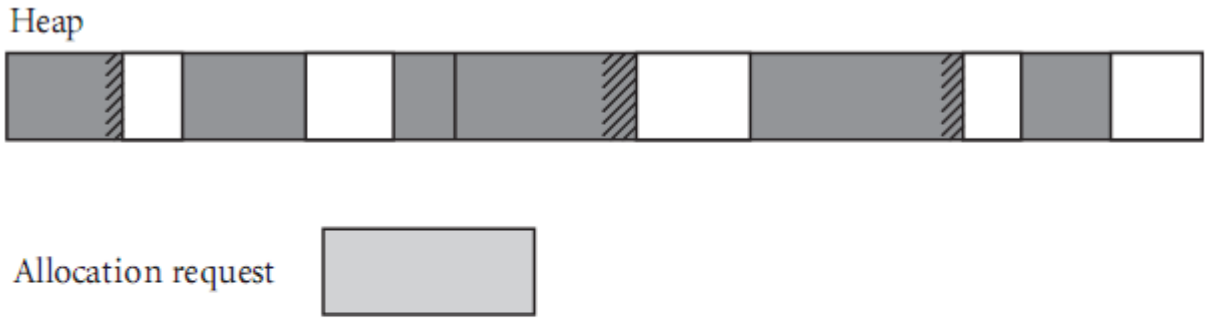



**Figure 3.2 Fragmentation.** The shaded blocks are in use; the clear blocks are free. Cross-hatched space at the ends of in-use blocks represent internal fragmentation. The discontiguous free blocks indicate external fragmentation. While there is more than enough total free space remaining to satisfy an allocation request of the illustrated size, no single remaining block is large enough.

# The Meaning of Names within a Scope

- Overloading
  - some overloading happens in almost all languages
    - integer + vs. real +
    - read and write in Pascal
    - function return in Pascal
  - some languages get into overloading in a big way
    - Ada
    - C++

# The Meaning of Names within a Scope

- It's worth distinguishing between some closely related concepts
  - overloaded functions - two different things with the same name; in C++
    - overload norm

```
int norm (int a){return a>0 ? a : -a;)
complex norm (complex c ) { // ...
```

  - polymorphic functions -- one thing that works in more than one way
    - in Modula-2: function min (A : array of integer); …
    - in Smalltalk

END