

APPUNTI DI

**TEORIA E TECNICHE DI  
COMPILAZIONE**

A.A. 2010/2011

Prof.ssa Marinella Sciortino

# Indice

<b>Indice</b> . . . . .	1
<b>1. Introduzione</b> . . . . .	7
1.1. Linguaggi di Programmazione . . . . .	7
1.2. Processore di un linguaggio di programmazione . . . . .	7
1.3. Processori come software separati . . . . .	7
1.4. Processori o ambienti di sviluppo integrati . . . . .	7
1.5. Obiettivi del corso . . . . .	8
1.6. Perché e quando nascono i compilatori . . . . .	8
1.6.1. Linguaggio macchina: linguaggi a basso livello della I generazione . . . . .	8
1.6.2. Linguaggio assembly: linguaggi a basso livello della II generazione . . . . .	8
1.6.3. L'evoluzione dei linguaggi di programmazione: nascono i linguaggi ad alto livello . . . . .	8
1.6.4. Nascono i compilatori . . . . .	9
1.6.5. Nascono importanti linguaggi ad alto livello . . . . .	9
1.6.6. Nascono i linguaggi specializzati . . . . .	9
1.6.7. Nascono altri paradigmi, per esempio il paradigma logico . . . . .	9
1.6.8. Evoluzione dei linguaggi e compilatori . . . . .	9
1.7. Compilatore . . . . .	10
1.8. Interpreti . . . . .	10
1.9. Compilatori ibridi . . . . .	10
1.10. Macchine virtuali . . . . .	10
1.10.1. Codice per macchina virtuale . . . . .	11
1.10.2. Come lavora una macchina virtuale . . . . .	11
<b>2. Struttura di un compilatore. Le fasi della compilazione</b> . . . . .	12
2.1. Linguaggi compilati: dal sorgente all'eseguibile . . . . .	12
2.2. Il preprocessore . . . . .	12
2.3. Linker e Loader . . . . .	12
2.4. Il compilatore lavora con linguaggi simbolici . . . . .	13
2.5. La struttura di un compilatore . . . . .	13
2.6. Le fasi della compilazione . . . . .	13
2.7. Front end & Back end . . . . .	14
2.8. Analisi lessicale (lexical analysis o scanning) . . . . .	14
2.9. Analisi sintattica (syntax analysis o parsing) . . . . .	14
2.10. Analisi semantica (semantic analysis) . . . . .	14
2.11. Semantica statica e semantica dinamica . . . . .	15
2.12. Generazione del codice intermedio . . . . .	15
2.13. Three-address code . . . . .	15
2.14. Ottimizzazione del codice . . . . .	15
2.14.1. Esempi di ottimizzazioni indipendenti dalla macchina . . . . .	16
2.15. Generazione del codice target o oggetto . . . . .	16
2.16. Compiliamo un piccolo programma . . . . .	17
2.17. Gestione degli errori . . . . .	17
2.18. Fasi e passate . . . . .	17
2.19. Realizzare un compilatore . . . . .	18
2.20. Realizzare automaticamente un compilatore . . . . .	18
2.21. Evoluzione della Compiler Technology . . . . .	18
<b>3. Analisi Lessicale (Come costruire uno scanner)</b> . . . . .	19
3.1. Analisi lessicale . . . . .	19
3.2. Token . . . . .	19
3.3. Lessema (Lexeme) . . . . .	20

3.4. Pattern . . . . .	20
3.5. Attributi . . . . .	20
3.6. Implementazioni della symbol table . . . . .	20
3.7. Ruolo dello scanner . . . . .	21
3.8. Errori lessicali . . . . .	21
3.9. Come implementare uno scanner . . . . .	21
3.10. Come descrivere i token . . . . .	22
3.11. Linguaggi formali (definizioni) . . . . .	22
3.12. Espressioni regolari . . . . .	22
3.13. Definizioni regolari . . . . .	23
3.14. Ambiguità . . . . .	23
3.15. Esistono linguaggi non regolari . . . . .	24
3.16. Come riconoscere i token . . . . .	24
3.17. Come riconoscere i token rellop . . . . .	24
3.18. Riconoscimento di keyword e identificatori . . . . .	25
3.19. Tecniche per la gestione dell'input . . . . .	25
3.20. Automi a stati finiti . . . . .	25
3.20.1. Simulare un DFA con diagrammi di transizione . . . . .	26
3.20.2. Simulare un DFA con matrice di transizione . . . . .	26
3.21. Il problema è un po' più complesso... . . . . .	26
3.22. Come costruire l'automa . . . . .	27
3.23. Automi a stati finiti non-deterministici (NFA) . . . . .	27
3.24. Espressioni regolari ed automi . . . . .	27
3.25. Da espressione regolare a NFA . . . . .	28
3.26. Da NFA a DFA . . . . .	28
3.27. Simulare un NFA . . . . .	29
3.28. NFA o DFA ? . . . . .	29
<b>4. Generatori automatici di scanner (FLEX)</b> . . . . .	30
4.1. Generatori di scanner . . . . .	30
4.2. Come funziona Flex . . . . .	30
4.3. Programmare in Flex . . . . .	30
4.3.1. Come definire i pattern nella sezione delle regole . . . . .	32
4.3.2. Come avviene il match . . . . .	33
4.3.3. Come definire le azioni . . . . .	33
4.3.4. Variabili e routine disponibili all'utente . . . . .	33
4.3.5. Esempi: . . . . .	34
4.3.6. Start Conditions . . . . .	37
4.4. Prova in itinere 2010 . . . . .	38
<b>5. Scanner del linguaggio TINY</b> . . . . .	40
<b>6. Analisi sintattica e grammatiche CF</b> . . . . .	43
6.1. Grammatiche context-free . . . . .	43
6.2. Regole ricorsive . . . . .	44
6.3. Alberi di derivazione o di parsing . . . . .	44
6.4. Grammatiche ambigue . . . . .	44
6.4.1. Alcuni tipi di ambiguità . . . . .	44
6.4.2. Eliminare l'ambiguità . . . . .	45
6.4.3. Ambiguità dell'else "pendente" . . . . .	45
6.5. Costrutti non context-free . . . . .	45
6.6. Compito principale del parser . . . . .	45
6.7. Ruolo del parser nel processo di compilazione . . . . .	45
6.8. Perché usare le grammatiche . . . . .	46
6.9. Grammatiche CF e automi a pila . . . . .	46
6.9.1. Dalla grammatica CF all'automa a pila indeterministico con un solo stato . . . . .	46

6.10. Parser deterministici e non deterministici . . . . .	47
6.11. Tipi di parser . . . . .	48
6.12. Algoritmo di Earley . . . . .	48
6.12.1. Scansione, completamento e predizione . . . . .	48
6.12.2. Esempi . . . . .	49
6.12.3. Complessità dell'algoritmo . . . . .	49
6.13. Gestione degli errori . . . . .	50
<b>7. Parser discendenti e ascendenti</b> . . . . .	51
7.1. Parser deterministici top down (dalla radice alle foglie) . . . . .	51
7.2. Trasformazione della grammatica per l'analisi top-down . . . . .	51
7.3. Fattorizzazione sinistra . . . . .	51
7.4. Eliminazione ricorsione sinistra . . . . .	52
7.4.1. Algoritmo per eliminare le ricorsioni sinistre . . . . .	52
7.5. Parser a discesa ricorsiva . . . . .	52
7.5.1. Procedura tipica per un parser top-down a discesa ricorsiva . . . . .	53
7.5.2. Costruzione delle procedure ricorsive . . . . .	53
7.6. Parser LL(1) . . . . .	53
7.6.1. I Parser LL(1) sono parser discendenti non ricorsivi . . . . .	54
7.6.2. Costruzione della parsing table di un LL(1) . . . . .	54
7.7. First . . . . .	55
7.7.1. Esempio di calcolo di First . . . . .	55
7.7.2. Algoritmo in pseudo-C per calcolare FIRST . . . . .	55
7.7.3. FIRST può non bastare . . . . .	56
7.8. Follow . . . . .	56
7.8.1. Algoritmo in pseudo-C per calcolare FOLLOW . . . . .	56
7.8.2. Esempio di calcolo di Follow . . . . .	57
7.9. Grammatica LL(1) . . . . .	57
7.10. Come costruire la tabella . . . . .	57
7.11. Altra definizione di Grammatica LL(1) . . . . .	57
7.12. Algoritmo di parsing LL(1) . . . . .	58
7.12.1. Complessità di calcolo . . . . .	58
7.12.2. Esempio . . . . .	58
7.13. Esempi di grammatiche LL(1) e non . . . . .	59
7.14. Come ottenere grammatiche LL(1) . . . . .	60
7.15. Limiti della famiglia LL(k) . . . . .	60
7.16. Relazione tra grammatiche LL(k) . . . . .	60
<b>8. Parsing ascendenti</b> . . . . .	61
8.1. Parsing Bottom-up . . . . .	61
8.1.1. Esempio di parsing bottom up . . . . .	61
8.2. Parser shift-reduce (SR) . . . . .	61
8.2.1. Conflitti durante il Parsing SR . . . . .	62
8.3. Tipico SR parsing: LR(k) . . . . .	62
8.3.1. Perché usare i parser LR? . . . . .	63
8.3.2. Schema di un parser LR . . . . .	63
8.3.3. Configurazione e funzionamento di un parser LR . . . . .	63
8.4. Parser LR(0) . . . . .	63
8.4.1. Automa LR(0) . . . . .	64
8.4.2. Costruzione della tabella LR(0) . . . . .	64
8.4.3. Algoritmo di Parsing LR(0) . . . . .	66
8.4.4. Grammatiche LR(0) . . . . .	66
8.5. Conflitti shift-reduce o reduce-reduce . . . . .	66
8.6. Esempio di grammatica non LR(0) . . . . .	67
8.7. Tabella ambigua . . . . .	67

8.8. LL(k) vs. LR(k) . . . . .	67
8.9. LR Parsing . . . . .	67
8.10. Esempi . . . . .	68
8.11. Confronto tra grammatiche e linguaggi LR(0) e LL(1) . . . . .	68
8.12. Parser LR(0), SLR, LR(1), LALR(1): cosa hanno in comune? . . . . .	68
8.13. Simple LR parser (SLR o SLR(1)) . . . . .	69
8.13.1. Tabella SLR . . . . .	69
8.13.2. Esercizi . . . . .	69
8.13.3. Esempio di grammatica SLR . . . . .	70
8.13.4. Algoritmo di parsing SLR(1) . . . . .	70
8.14. Esistono grammatiche non SLR . . . . .	70
8.15. Esempio di conflitto reduce/reduce . . . . .	71
8.16. Parser LR(1) . . . . .	71
8.16.1. Costruzione della tabella LR(1) . . . . .	71
8.16.2. Esercizio . . . . .	72
8.17. Parsing LALR(1) . . . . .	73
8.17.1. Condizioni LALR(1) . . . . .	73
8.17.2. Esempio di LR(1) ma non LALR(1) . . . . .	73
8.18. Regole per risolvere l'ambiguità . . . . .	73
8.19. Proprietà dei linguaggi e delle grammatiche LR(k) <b>IMPORTANTE</b> . . . . .	74
8.20. Confronto tra le grammatiche LL(k) e LR(k): . . . . .	74
<b>9. Generatori di Parser (BISON)</b> . . . . .	75
9.1. Grammatiche GLR . . . . .	75
9.2. Uso di Yacc o Bison . . . . .	75
9.3. Uso di Flex e Bison . . . . .	75
9.4. Programmare in Bison . . . . .	76
9.5. Specifiche Bison . . . . .	76
9.6. Azioni e valore semantico . . . . .	76
9.7. Esempio . . . . .	77
<b>10. Analisi semantica</b> . . . . .	84
10.1. Semantica statica . . . . .	84
10.2. Semantica dinamica . . . . .	84
10.3. Analisi semantica statica . . . . .	85
10.4. Esempi di semantica statica e semantica dinamica . . . . .	85
10.5. Analisi semantica statica . . . . .	85
10.6. Attributi e Grammatiche con attributi . . . . .	85
10.6.1. Binding time . . . . .	86
10.6.2. Grammatiche con attributi . . . . .	86
10.6.3. Rappresentazione di un attributo . . . . .	86
10.6.4. Grammatiche con attributi e semantica guidata dalla sintassi . . . . .	87
10.6.5. Esempi . . . . .	88
10.6.6. Definire e calcolare gli attributi . . . . .	90
10.6.7. Dipendenze funzionali degli attributi . . . . .	91
10.6.8. Grafo delle dipendenze . . . . .	91
10.6.9. Algoritmo generale per il calcolo degli attributi . . . . .	91
10.6.10. Si restringe la classe delle grammatiche con attributi . . . . .	92
10.6.10.1. Attributi sintetizzati . . . . .	92
10.6.10.2. Attributi ereditati . . . . .	92
10.6.11. Algoritmi per il calcolo degli attributi . . . . .	92
10.6.12. Grammatiche con L-attributi . . . . .	92
10.6.13. Passate per il calcolo degli attributi . . . . .	93
10.7. Analisi sintattico-semantica . . . . .	93

<b>11. Tabella dei simboli (Symbol Table)</b>	94
11.1. Quando costruire ed interagire con la TS	94
11.2. Struttura della Tabella dei simboli	94
11.3. Operazioni principali sulla TS	94
11.4. Problemi	95
11.5. Struttura della Tabella dei simboli	95
11.6. Realizzazione della Tabella dei simboli	95
11.6.1. La Tabella dei simboli come <b>ARRAY</b>	95
11.6.2. La TS come <b>lista concatenata</b>	95
11.6.3. La TS come ABR	96
11.6.4. La TS ad accesso diretto	96
11.6.5. La TS come <b>tabella hash</b>	97
11.6.5.1. La funzione hash	97
11.6.5.2. Collisioni	98
11.6.5.3. Scelta di M e fattore di carico	98
11.6.5.4. Funzioni hash (hash table di taglia M)	98
11.6.5.5. Metodi classici di risoluzione delle collisioni	99
11.6.5.6. Lista di collisione (hash table with chaining)	99
11.6.6. Comportamento e implementazione della ST	99
11.7. Dichiarazioni	100
11.8. Attributi di visibilità	100
11.9. Visibilità e struttura a blocchi	100
11.9.1. Regole di visibilità e struttura a blocchi	100
11.9.2. Risoluzione del problema	101
11.9.2.1. Una TS per ogni ambiente	101
11.9.2.1.1. La pila degli ambienti	101
11.9.2.1.2. Problemi con la pila degli ambienti	101
11.9.2.2. Una TS globale	101
11.10. Osservazioni sul contenuto della TS	102
11.10.1. Nome dell'identificatore	102
11.10.2. La soluzione	102
<b>12. Type checking (controllo dei tipi)</b>	103
12.1. Compatibilità o equivalenza di tipi	103
12.2. Dichiarazioni	104
12.3. Regole per il type checking	104
12.4. Realizzazione di un type checker	104
12.4.1. Le regole della grammatica	104
12.4.2. Type checking delle dichiarazioni	105
12.4.3. Type checking degli statement	105
12.4.4. Type checking delle espressioni	105
12.4.5. Type coercion (conversione automatica di tipo)	105
12.4.6. Type coercion delle espressioni	106
12.5. Overloading	106
12.6. Polimorfismo	106
12.7. Equivalenza strutturale	106
<b>13. generazione del Codice Intermedio</b>	107
13.1. Forma del codice intermedio	107
13.2. Più codici intermedi	108
13.3. Codice a tre indirizzi	108
13.3.1. Non esiste un 3AC standard	108
13.4. Indirizzi e istruzioni	109
13.5. Esempio in linguaggio SimplePas	110
13.6. Strutture dati per l'implementazione del 3AC	110

13.7. Implementazione delle istruzioni in 3AC . . . . .	110
13.8. Esempio . . . . .	111
13.9. Codice per macchina virtuale . . . . .	111
13.9.1. Un esempio: il P-Code . . . . .	112
13.9.1.1. P-code per il programma per il calcolo del fattoriale . . . . .	113
13.9.2. Confronto tra P-code e 3AC . . . . .	113
13.10. Tecniche per la generazione del codice intermedio . . . . .	113
13.11. Esempio: traduzione delle espressioni . . . . .	114
13.12. Grammatica con attributi per la generazione di P-code . . . . .	114
13.13. Grammatica con attributi per la generazione di 3AC . . . . .	115
13.14. Considerazioni . . . . .	115
13.15. Criteri generali per il C.I. . . . .	116
13.15.1. if then . . . . .	116
13.15.2. if then else . . . . .	116
13.15.3. while . . . . .	116
13.15.4. assegnazione di valori booleani . . . . .	116
13.15.5. switch . . . . .	117
13.15.6. chiamata di funzione . . . . .	117
13.16. Fase di sintesi . . . . .	117
<b>14. Generazione del codice oggetto . . . . .</b>	<b>118</b>
14.1. Ottimizzazione del codice . . . . .	118
14.2. Esempi di ottimizzazioni indipendenti dalla macchina . . . . .	118
14.3. Generatori di codice oggetto . . . . .	119
14.4. Input per un generatore di codice oggetto . . . . .	119
14.5. Il programma target . . . . .	119
14.6. Codice macchina assoluto e rilocabile . . . . .	119
14.7. Selezione delle istruzioni . . . . .	119
14.8. Allocazione dei registri . . . . .	120
14.9. Ordine di valutazione . . . . .	120
14.10. Linguaggio target . . . . .	120
14.11. Indirizzi nel codice target . . . . .	120
14.12. Blocchi base . . . . .	121
14.12.1. Algoritmo di partizionamento in blocchi base . . . . .	121
14.13. Algoritmo per il prossimo uso . . . . .	121
14.14. Diagramma di flusso . . . . .	121
14.15. Semplice generazione del codice per un blocco base . . . . .	122
14.16. Istruzioni macchina per le operazioni e per la copia . . . . .	122
14.17. Esempi di ottimizzazioni dipendenti dalla macchina . . . . .	122

# 1. Introduzione

## 1.1. Linguaggi di Programmazione

I linguaggi di programmazione sono gli strumenti di base dei programmati. Tutti i software che girano su tutti i computer sono scritti in un qualche linguaggio di programmazione. Prima che un programma possa "girare" ovvero essere mandato in esecuzione in un computer deve essere prima trasformato (tradotto) in sequenze di istruzioni elementari eseguibili da un computer. Si usano spesso, a volte "inconsapevolmente" nei processori di linguaggi di programmazione.

## 1.2. Processore di un linguaggio di programmazione

È un qualsiasi sistema che manipola programmi espressi in un particolare linguaggio. Consente di scrivere i programmi e prepararli per l'esecuzione.

Può incorporare una varietà di sistemi tra cui:

› **Editors:** Consente la digitazione, modifica e salvataggio in un file.

› **Traduttori e Compilatori:** Un traduttore traduce un testo scritto in un linguaggio in un altro. In particolare, un compilatore traduce programmi da linguaggio ad alto livello a linguaggio a basso livello, quindi prepara per l'esecuzione su una macchina. Prima di effettuare la traduzione un compilatore controlla la presenza di errori sintattici e contestuali.

› **Interpreti:** Un interprete prende un programma espresso in un particolare linguaggio e lo esegue immediatamente. Questa modalità di esecuzione, che omette la fase di compilazione, è preferibile in ambienti interattivi (ad es. molti linguaggi di interrogazione dei database sono interpretati).

## 1.3. Processori come software separati

La filosofia dei "tool separati" è ben esemplificata dal SO Unix. Ad Esempio, un utente UNIX che sviluppa un'applicazione sul gioco degli scacchi in Java usando SUN JDK.

- |   |                  |
|---|------------------|
| › Usa l'editor vi   | vi Chess.java    |
| › Invoca il compilatore javac                                 | javac Chess.java |
| › Chiama l'interprete java sul codice in bytecode Chess.class | java Chess       |

## 1.4. Processori o ambienti di sviluppo integrati

Esistono vari IDE. Per Java alcuni esempi sono Eclipse, NetBeans, Borland Jbuilder, per C alcuni esempi sono Visual Studio, DEVC++, ...

L'utente può aprire, editare, compilare ed eseguire il programma. L'editor consente di distinguere tra variabili, parole chiave, commenti. Il compilatore è integrato con l'editor, nel caso di errori di compilazione rimanda alla frase contenente presumibilmente l'errore. Se ci sono errori di esecuzione, si rimanda nell'editor alla frase del codice oggetto in cui è fallita l'esecuzione.

## 1.5. Obiettivi del corso

- Illustrare la struttura, le tecniche di costruzione e il funzionamento di un moderno compilatore

- Usare le tecniche per la realizzazione di compilatori mediante l'uso di strumenti automatici:

- Flex                   (Fast Lexical analyzer generator)
- Bison                 (Parser generator)

- Applicare principi e tecniche per la progettazione dei compilatori in vari contesti legati all'analisi di testi.

Lo studio dei compilatori richiede particolari conoscenze che coinvolgono:

- ›Linguaggi di programmazione (in particolare il linguaggio C)
- ›Informatica Teorica (linguaggi, automi, espressioni regolari e grammatiche)
- ›Algoritmi (liste, pile, alberi, grafi, tabelle hash)

## 1.6. Perché e quando nascono i compilatori

L'interesse verso i compilatori è profondamente legato all'evoluzione dei linguaggi di programmazione. Esistono linguaggi a vari livelli di astrazione. I compilatori includono tecniche per sopperire alle inefficienze introdotte da astrazioni sempre più ad alto livello.

### 1.6.1. Linguaggio macchina: linguaggi a basso livello della I generazione

Il linguaggio macchina è il linguaggio nativo del computer su cui un certo programma viene eseguito. Guida l'esecuzione della macchina. Esso consiste di sequenze di bit che sono interpretate (eseguite passo dopo passo) da un meccanismo interno al computer. Può essere espresso in binario o in esadecimale.

Quando il primo computer elettronico apparve (nel 1945, modello di John von Neumann), era possibile usare solo il linguaggio macchina. Le operazioni realizzabili erano dette di basso livello erano strettamente legate al tipo di macchina e consistevano in:

- Spostare dati da una locazione all'altra;
- Confrontare due valori;
- Sommare il contenuto di 2 registri;
- ...

*Ad esempio sul processore Intel 80x86, il codice **C7 06 0000 0002** dice alla CPU di inserire 2 nella locazione 0000.*

### 1.6.2. Linguaggio assembly: linguaggi a basso livello della II generazione

I **linguaggi assembly** sono nati negli anni '50. Inizialmente erano soltanto rappresentazioni mnemoniche di istruzioni in linguaggio macchina, ovvero abbreviazioni o parole che rendevano più facile ricordare le sequenze di bit. Dopo, vennero aggiunte anche le istruzioni macro.

*Ad esempio il codice precedente potrebbe diventare **movx,2**.*

Un programma in linguaggio assembly è tradotto in linguaggio macchina da un programma chiamato **assembler** che traduce una alla volta le istruzioni in assembly in codice macchina.

### 1.6.3. L'evoluzione dei linguaggi di programmazione: nascono i linguaggi ad alto livello

I software per i primi computer erano scritti in linguaggio assembly. L'esigenza dei linguaggi ad alto livello (indipendenti cioè dalla macchina) nacque quando il vantaggio di riusare software su diversi tipi di macchine divenne significativamente maggiore del costo della scrittura di un compilatore.

*L'istruzione precedente potrebbe diventare **x=2**.*

Nel 1957 venne introdotto il FORTRAN per il calcolo scientifico da John Backus dell'IBM, da cui derivò successivamente il BASIC (1964). Tali linguaggi prevedono:

- istruzione IF
- cicli WHILE e FOR
- istruzioni CASE e SWITCH

Il COBOL nacque nel 1961 per la gestione degli archivi commerciali, il LISP nel 1959 per il calcolo simbolico, l'ALGOL nel 1960. Nel 1967 nacque il SIMULA, che introdusse per primo il concetto (allora appena abbozzato) di oggetto software.

#### 1.6.4. Nascono i compilatori

Il termine **compilatore** fu coniato da Grace Murray Hopper nel 1950. E' stata una matematica, informatica e militare statunitense che ebbe un ruolo fondamentale nello sviluppo del COBOL. Il primo compilatore reale fu un compilatore FORTRAN sviluppato alla fine degli anni cinquanta.

#### 1.6.5. Nascono importanti linguaggi ad alto livello

Nel 1970 Niklaus Wirth pubblica il Pascal, il primo linguaggio strutturato, a scopo didattico; nel 1972 dal BCPL nascono prima il B (rapidamente dimenticato) da cui Dennis Ritchie sviluppò il C, che invece fu fin dall'inizio un grande successo. Insieme a FORTRAN, ALGOL e COBOL costituiscono esempi del **paradigma di programmazione imperativo**

Nel 1983 vede la luce Smalltalk, il primo linguaggio realmente e completamente ad oggetti, che si ispira al Simula e al Lisp. Esempi di linguaggi object-oriented odierni sono Eiffel, C++ nel 1986, e successivamente Delphi, JAVAnel 1995.

In media un'istruzione in linguaggio ad alto livello corrisponde a 10 istruzioni in assembly.

**Nota:** Contestualmente alla nascita dei primi compilatori Noam Chomsky cominciò il suo studio dei linguaggi naturali alla classificazione dei linguaggi in base alla potenza di calcolo delle grammatiche.

#### 1.6.6. Nascono i linguaggi specializzati

Sono i linguaggi progettati per applicazioni specifiche, come per esempio lo sviluppo di applicazioni commerciali. Hanno come obiettivo quello di ridurre al minimo gli sforzi per la programmazione, il tempo per lo sviluppo di un software e il relativo costo.

Ad esempio:

- › SQL, per query in database
- › Postscript, per generare report
- › Mathematica, Matlab, per la manipolazione e l'analisi di dati
- › ...

#### 1.6.7. Nascono altri paradigmi, per esempio il paradigma logico

Si risolve un problema usando i vincoli dati al problema stesso piuttosto che usare un l'algoritmo scritto da un programmatore. Usati principalmente in intelligenza artificiale, istruiscono il computer per trovare la soluzione.

Ad esempio:

- › Linguaggi basati su linguaggi logici come Prolog, OPS5
- › Mercury
- › ...

#### 1.6.8. Evoluzione dei linguaggi e compilatori

L'evoluzione dei linguaggi di programmazione è profondamente in relazione con quella dei compilatori.

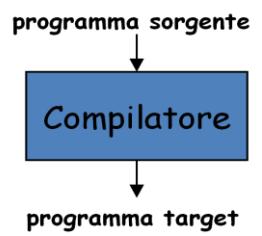
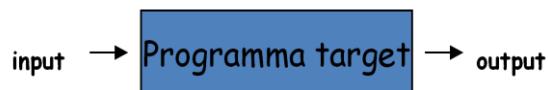
- La progettazione di un compilatore richiede enormi sforzi legati alla scelta dei modelli da utilizzare e delle implementazioni. ☺
- Un compilatore deve tradurre in modo corretto un insieme potenzialmente infinito di programmi scritti in linguaggio sorgente. Il codice prodotto deve essere il più possibile efficiente, sia da un punto di vista del tempo di esecuzione che della memoria utilizzata. ☺
- La maggior parte degli utenti vedono un compilatore come una "blackbox". I compilatori consentono ai programmatore di ignorare i dettagli machine-dependent della programmazione. ☺

## 1.7. Compilatore

Un compilatore è un programma capace di leggere un programma in un linguaggio (linguaggio sorgente) e tradurlo in un programma equivalente in un altro linguaggio (linguaggio destinazione o target).

Un ruolo importante del compilatore consiste nel riportare gli eventuali errori nel programma sorgente incontrati durante il processo di traduzione.

Se il programma target è programma in un linguaggio-macchina eseguibile, allora esso può essere chiamato dall'utente per processare un input e produrre un output.



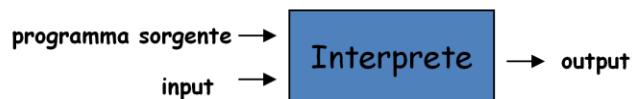
## 1.8. Interpret

Invece di produrre il programma target come traduzione, un interprete esegue direttamente le operazioni specificate nel programma sorgente su input forniti dall'utente.

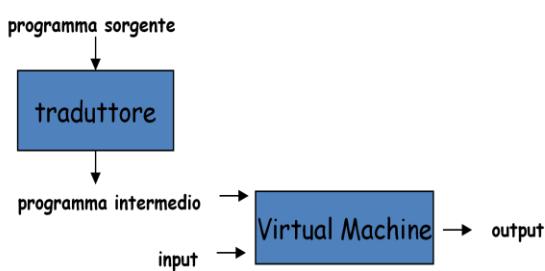
Il programma target prodotto da un compilatore è solitamente 10 volte più veloce di quello prodotto da un interprete. Un interprete comunque può meglio eseguire una diagnostica degli errori poiché esegue il programma sorgente statement dopo statement.

Esempi di linguaggi interpretati sono:

- Javascript
- Perl
- PHP
- Python
- ...



## 1.9. Compilatori ibridi



Esistono anche compilatori ibridi che combinano compilazione e interpretazione. Ad esempio, un programma sorgente in JAVA viene prima compilato e trasformato in una lingua intermedia (linguaggio di bytecode). I bytecode vengono poi interpretati o compilati da una virtual machine.

Alcuni compilatori JAVA, detti compilatori just in time, traducono il bytecode in linguaggio macchina immediatamente prima di doverlo eseguire.

## 1.10. Macchine virtuali

E' un software che crea un ambiente virtuale in cui l'utente può eseguire alcune applicazioni senza tener conto del sistema operativo sottostante.

Dal momento che esistono macchine virtuali scritte per diverse piattaforme, il programma compilato può "girare" su qualsiasi piattaforma su cui "gira" la macchina virtuale ("Write once, run everywhere").

Progenitore delle odierne VM è la "macchina p", cioè il calcolatore astratto per cui venivano (e vengono tuttora) compilati i programmi in Pascal nelle prime fasi della compilazione (producendo il cosiddetto p-code).

### 1.10.1. Codice per macchina virtuale

- **p-code** prodotto dal Pascal p-compiler per una **Virtual Stack Machine**
- **bytecode** prodotto dai compilatori Java per una **Virtual Java Machine** progettata da Sun Microsystems.
- **Common Intermediate Language (CIL)** della piattaforma .NET eseguito dal **Common Language Runtime (CLR)** la macchina virtuale .NET progettata da Microsoft per funzionare con qualsiasi sistema operativo.

### 1.10.2. Come lavora una macchina virtuale

I programmi applicativi vengono scritti in un linguaggio che viene compilato per questo calcolatore immaginario (cioè tradotti nelle sue istruzioni native) e, una volta compilati, vengono eseguiti sulla macchina virtuale software, che può agire o come interprete o come compilatore "al volo" (compilazione just in time: traducono il codice intermedio in linguaggio macchina immediatamente prima che venga processato l'input).

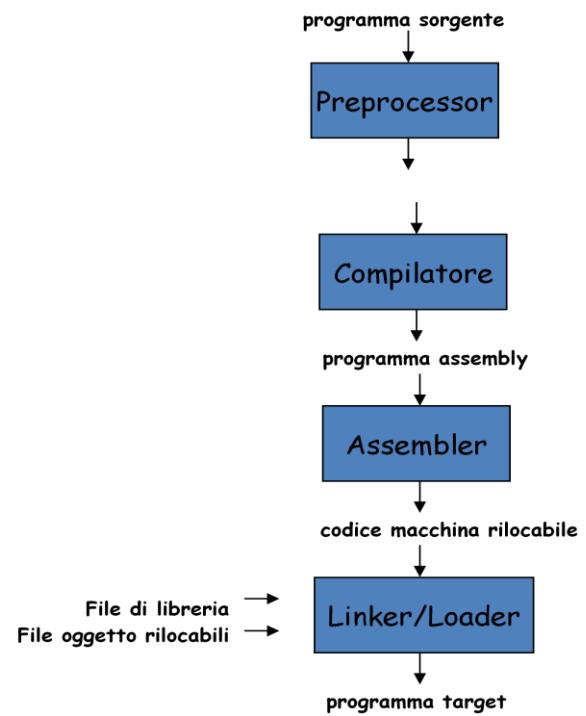
Le più recenti implementazioni di virtual machine incorporano un JIT compiler:

- La macchina virtuale Java HotSpot di Sun Microsystem
- La macchina virtuale Common Language Runtime di Microsoft per la piattaforma .NET
- La macchina virtuale Parrot per il linguaggio Perl
- ...

## 2. Struttura di un compilatore. Le fasi della compilazione

### 2.1. Linguaggi compilati: dal sorgente all'eseguibile

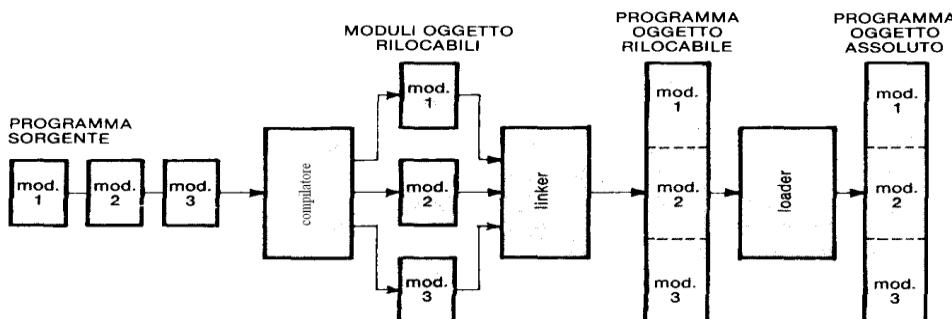
- Il programma target prodotto dal compilatore normalmente non è pronto per l'esecuzione e ciò per diverse ragioni.
- Un programma sorgente può essere diviso in moduli memorizzati in file separati. Il compito di mettere assieme il codice sorgente è a volte affidato al preprocessor o precompilatore. Esso può anche espandere le macro in statement del linguaggio sorgente.
- Il programma sorgente modificato viene dato in pasto al compilatore.
- Il compilatore può produrre un codice in linguaggio assembly che è più facile da produrre ed è più facile fare il debug. Allora il file viene processato dall'assembler che produce un codice macchina rilocabile.
- I programmi lunghi sono spesso compilati a pezzi, così il codice macchina rilocabile deve essere linkato con altri file in codice rilocabile e file di libreria al fine di produrre un codice che giri sulla macchina. Il linker assegna e risolve gli indirizzi di memoria esterna, poiché un codice in un file può riferirsi ad una locazione in un altro file. Il loader mette insieme tutti i file oggetto eseguibili in memoria per l'esecuzione.



### 2.2. Il preprocessore

- In alcuni casi il codice sorgente subisce una trasformazione (**precompilazione**) prima di essere compilato.
- La precompilazione può essere definita come la trasformazione del codice sorgente in un altro.
- Esempi:
  - Traduzione del Lisp in C
  - Prime versioni del C++ (C con classi)
  - Utilizzo dell'SQL embedded
  - Adattare il programma in funzione dell'installazione (Compilazione condizionale)
  - Espansione delle macro

### 2.3. Linker e Loader



Questi strumenti sono presenti anche negli ambienti integrati di sviluppo

## 2.4. Il compilatore lavora con linguaggi simbolici

Ogni linguaggio simbolico è formato da:

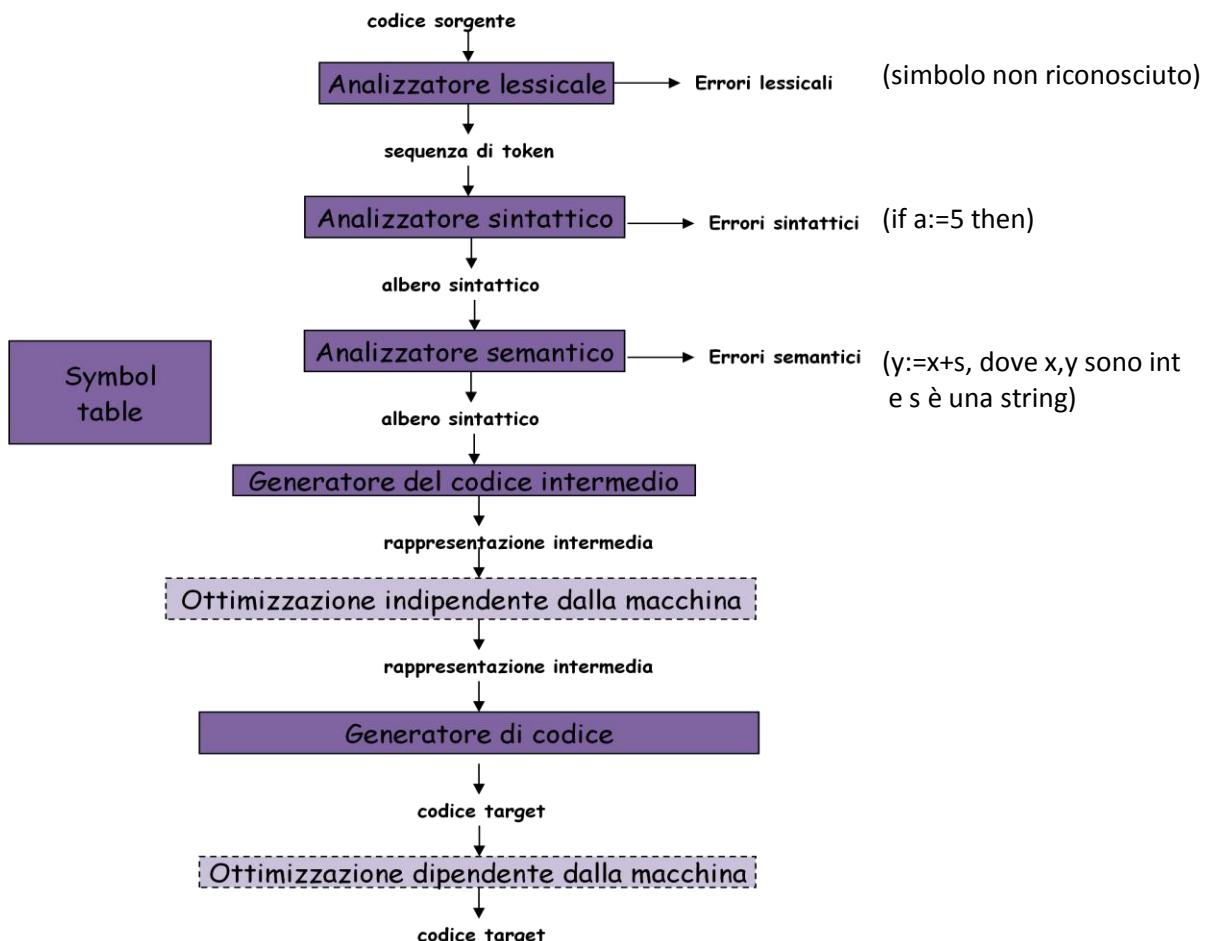
- Alfabeto, insieme dei simboli usati
- Lessico, insieme delle parole che formano il linguaggio (vocabolario)
- Sintassi, insieme delle regole per la formazione delle frasi (istruzioni)
- Semantica, significato da associare ad ogni parola ed istruzione

## 2.5. La struttura di un compilatore

Il processo di compilazione consiste in una serie di passi chiamati **fasi della compilazione**

- Analisi lessicale
  - Analisi sintattica
  - Analisi semantica
  - Generazione del codice intermedio
- } Analisi
- 
- Ottimizzazione
  - Generazione del codice target (codice oggetto) Sintesi
- } Sintesi

## 2.6. Le fasi della compilazione



## 2.7. Front end & Back end

La compilazione è stata suddivisa in due parti:

- Analisi
- Sintesi

La prima parte, che dipende solo dal tipo di linguaggio che deve essere tradotto viene anche detta "front end" del compilatore.

La seconda, insieme alle ottimizzazioni dipendenti dalla macchina viene detta "back end" del compilatore.

## 2.8. Analisi lessicale (lexical analysis o scanning)

In questa fase l'analizzatore lessicale (**scanner**) suddivide il codice sorgente in **lessemi** (elementi lessicali o terminali del linguaggio) che memorizza in una symbol table. Per ogni lessema produce un **token**.

I vari **lessemi**, una volta isolati, vengono memorizzati in una o più tabelle (tabelle dei descrittori o **symbol table**) dove saranno contenute tutte le indicazioni necessarie per identificare e caratterizzare i singoli **lessemi** (nome, tipo, visibilità,...; nel caso di nomi di funzioni, il numero di parametri, come sono passati, il tipo restituito,...)

Un token può essere pensato come una coppia <nome del token, attributo>

Il nome del token è un nome astratto utile durante l'analisi sintattica, l'attributo è il puntatore nella symbol table alla entry contenente i dati di questo specifico token.

I compiti dello scanning sono:

- Costruzione delle symbol table
- Trasformazione del codice sorgente in forma semplificata, pronta per l'analisi sintattica.

## 2.9. Analisi sintattica (syntax analysis o parsing)

In questa fase l'analizzatore sintattico (**parser**) usa i token (i nomi dei token) per definire la struttura grammaticale della sequenza di token, ovvero determina la struttura delle singole istruzioni e controlla se sono rispettate le regole della sintassi del linguaggio.

Il parser adopera il codice semplificato e la symbol table costruiti nella fase precedente e genera per ogni istruzione il corrispondente albero sintattico (**syntax tree**).

L'output della fase di analisi sintattica è l'insieme degli alberi sintattici che descrivono il programma.

## 2.10. Analisi semantica (semantic analysis)

Usa il syntax tree e la symbol table del programma sorgente per controllarne la consistenza semantica con le definizioni del linguaggio. L'analisi semantica consiste nell'interpretazione del "significato" delle strutture prodotte nella fase precedente controllando che esse siano legali e significative.

Si controlla, per esempio, che le variabili coinvolte siano dichiarate e definite, che i tipi siano corretti, che gli operatori siano usati correttamente, ....

Una parte importante dell'analisi semantica è il type checking dove il compilatore controlla che ogni operatore abbia gli operandi giusti.

Per esempio:

- controlla che l'indice di un array sia un intero
- applica le regole di visibilità degli identificatori
- applica le conversioni di tipo nel caso di operatori che possono essere applicati ad operandi di diverso tipo (coercions)

## 2.11. Semantica statica e semantica dinamica

- Semantica statica: indipendente dai dati su cui opera il programma sorgente.

```
var i : real;  
a : array [1..100] of integer;  
.....  
i:=3.5;  
a[i]:=3;
```

- Semantica dinamica: dipendente dai dati su cui opera il programma sorgente.

```
var i : integer;  
a : array [1..100] of integer;  
.....  
read(i);  
a[i]:=3;
```

- L'analizzatore semantico si occupa della semantica statica, mentre la semantica dinamica spetta all'interprete o al supporto esecutivo.

## 2.12. Generazione del codice intermedio

In questa fase si provvede alla generazione del codice intermedio che solitamente è una rappresentazione di basso livello (più vicina alla macchina).

Esistono diversi modelli usati per la generazione del codice intermedio (execution model). In ogni caso deve avere due proprietà importanti: essere facile da produrre e facile da tradurre in codice target.

Uno dei più usati è il *three-address-code* in cui tutte le istruzioni hanno la forma di istruzioni in assembly con tre operandi per istruzione:

*id := id1 operator id2*

## 2.13. Three-address code

In 3-AC :

- Ogni istruzione ha al più un operatore sul lato destro.
- Le istruzioni fissano l'ordine con cui vengono eseguite le operazioni
- Vengono generati nomi temporanei per memorizzare i valori intermedi
- Alcune istruzioni hanno meno di 3 operandi

## 2.14. Ottimizzazione del codice

In questa fase il codice intermedio viene analizzato e trasformato in codice ad esso equivalente ma più efficiente.

Questa fase può essere molto complessa e lenta.

La maggior parte dei compilatori permette di disattivare l'ottimizzazione per velocizzare la traduzione; altri non hanno ottimizzazione.

In questa fase si tratta di ottimizzazioni *indipendenti dalla macchina*.

Esistono anche ottimizzazioni *dipendenti dalla macchina* operate dal *post-ottimizzatore*.

### 2.14.1. Esempi di ottimizzazioni indipendenti dalla macchina

- Calcolare una sola volta le espressioni ripetute, purché le variabili non mutino di valore tra le ripetizioni:

$a[i*k] := a[i*k] + 1$  ( $i*k$  si può calcolare una sola volta)

- Eliminare le moltiplicazioni per 1 e le somme di 0.

- Portare fuori da un ciclo le operazioni che non dipendono dal valore dell'indice:

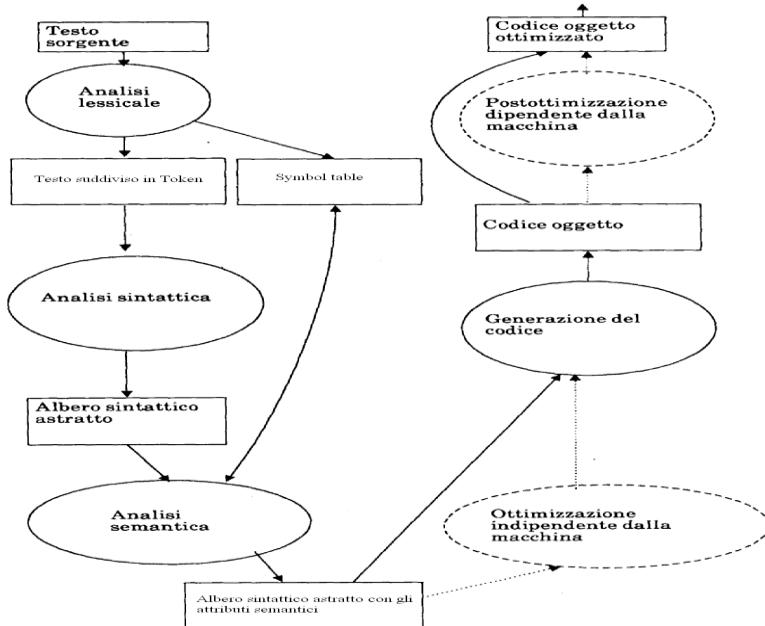
```
for i:=1 to N do
begin
```

...

j:=r/s;

...

end;                   (j:=r/s si può portare fuori)



N.B. Il problema di generare il codice target ottimale è indecidibile.

### 2.15. Generazione del codice target o oggetto

In questa fase si provvede alla traduzione del codice intermedio eventualmente ottimizzato nel linguaggio della "target machine".

Se il linguaggio target è il codice-macchina, allora per ogni variabile del programma sono assegnati registri e locazioni di memoria.

NOTA: Le decisioni sull'allocazione della memoria vengono prese o durante la generazione del codice intermedio o durante la generazione del codice oggetto.

## 2.16. Compiliamo un piccolo programma

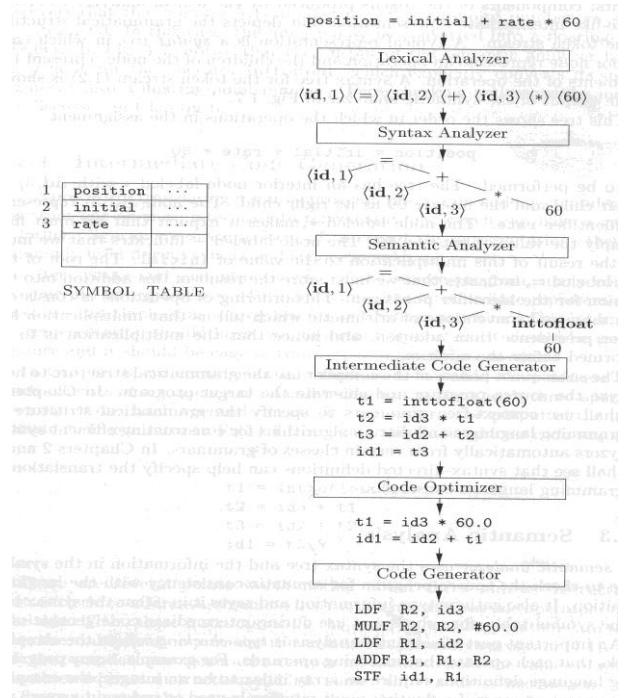
**position = initial + rate \* 60 ;**

- Individuazione dei lessemi:

### Token

	Lessema
identificatore	position, initial, rate
operatore somma (+)	+
operatore prodotto (*)	*
operatore di assegnazione (=)	=
costante	60
terminatore	;

### Lessema



- Type checking:

*position, initial e rate* sono floating-point e *60* è un intero.

- 3-AC:

```
t1=inttofloat(60)
t2=id3 * t1
t3=id2 + t2
id1=t3
```

- Generazione del codice target intermedio:

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```

La lettera F indica che si trattano floating-point

## 2.17. Gestione degli errori

Durante tutte le fasi della compilazione possono venir riscontrati degli errori. Essi possono essere tali da bloccare o meno il processo. Spesso accade che un errore ne provochi in conseguenza tanti altri. Conviene non preoccuparsi se alla prima compilazione l'elenco degli errori è molto lungo. Il problema di trovare tutti gli errori di un programma è indecidibile

## 2.18. Fasi e passate

Pur avendo descritto separatamente e sequenzialmente le varie fasi della compilazione, in realtà queste possono essere svolte con modalità e in tempi diversi.

Inoltre la compilazione può avvenire in una o più “passate”.

Ad esempio, le fasi del front-end possono essere raggruppate in una sola passata, l'ottimizzazione del codice intermedio in un'altra passata e poi un'ultima passata per il back-end.

In generale, una passata può coincidere con:

- una singola fase o parte di essa;
- più fasi;
- parti di più fasi;

Ma come stabiliamo il numero delle passate?

Un compilatore ad una sola passata è normalmente più veloce.

Alcuni linguaggi sono stati progettati perché potessero essere compilati in una sola passata (Pascal).

In altri casi le possibilità offerte dai linguaggi di programmazione impediscono la compilazione in una sola passata.

Uno svantaggio della compilazione in singola passata è che non è possibile effettuare delle ottimizzazioni "sofisticate" necessarie per generare codice di alta qualità.

Può essere difficile, infatti, valutare il numero di volte che un'espressione deve essere analizzata per produrre una sua buona ottimizzazione.

## 2.19. Realizzare un compilatore

Il primo compilatore fu scritto in linguaggio assembler (e non c'erano altre alternative)

Se si dispone di un compilatore C\_1 per un determinato linguaggio L\_1 è possibile scrivere un compilatore C\_2 per un altro linguaggio L\_2.

E' possibile scrivere un compilatore utilizzando una versione minimale dello stesso linguaggio (bootstrapping).

Il compilatore può girare sulla stessa macchina sulla quale girerà il codice target; oppure il compilatore gira su una macchina diversa da quella su cui girerà il codice target (cross compiler).

## 2.20. Realizzare automaticamente un compilatore

Esistono strumenti software per generare automaticamente parti di un compilatore. Ad esempio:

- Flex        generatore di analizzatori lessicali
- Bison        generatore di analizzatori sintattici

## 2.21. Evoluzione della Compiler Technology

La rapida evoluzione delle architetture propone continue sfide ai creatori di compilatori.

- Parallelismo a livello di istruzioni (Intel IA64). I compilatori possono ordinare le istruzioni in modo da rendere tale parallelismo più efficiente.
- Parallelismo a livello di processori. I compilatori possono generare codici paralleli per multiprocessori a partire da un codice sequenziale.
- Gerarchie di memorie, al livello di velocità di accesso e di spazio. Il compilatore può cambiare l'ordine delle istruzioni che accedono ai dati

Nello sviluppo delle architetture moderne i compilatori sono sviluppati nella fase di design dell'architettura stessa.

- RISC (Reduced Instruction-Set Computer)

### 3. Analisi Lessicale (Come costruire uno scanner)

#### 3.1. Analisi lessicale

E' una fase durante la quale l'analizzatore lessicale (scanner) scandisce la sequenza di caratteri che costituisce il codice sorgente (source code), li raggruppa in lessemi e produce una sequenza di token corrispondenti a tali lessemi.

Viene spesso chiamata TOKENIZZAZIONE (Tokenizing).

Per esempio, consideriamo la seguente linea di codice:

```
for index:=1 to N do a[index] := 4 + 2;
```

Questo codice contiene 30 caratteri non-blank ma solo 15 token:

Keywords:	for, to, do	Operatori:	:=, +
Identifieri:	index, a, N	Punteggiatura:	;
Costanti:	1, 4, 2	Parentesi:	[, ]

A meno che ciò non venga gestito da un precompilatore, lo scanner deve tener conto di:

- Rimuovere i commenti: i commenti sono individuabili con simboli speciali. Lo scanner deve individuare tali simboli.
- Case Conversion: Molti linguaggi di programmazione (ad es. Pascal) ignorano la capitalizzazione. Lo scanner deve convertire tutte le lettere in uppercase, per esempio.
- Rimuovere gli spazi: gli spazi bianchi (blank, tab, invio, ...) servono per separare certi tipi di token. Uno scanner deve solo riconoscere i token.
- Tenere traccia del numero di linea: nel caso eventuali messaggi d'errore.
- Preparazione di un output listing: Alcuni scanner creano una versione annotata del codice sorgente, contenente per es. il numero di linea, messaggi di errore o di warning, ...

Talvolta gli analizzatori lessicali applicano i seguenti due processi uno dietro l'altro:

- **Scanning:** non richiede la tokenizzazione ma solo la rimozione dei commenti e la compattificazione degli spazi bianchi consecutivi in uno solo;
- **Analisi lessicale:** lo scanner produce la sequenza di token.

Si potrebbe scaricare il ruolo di analizzatore lessicale sul parser?

I vantaggi dell'uso dello scanner sono:

- ciascuna delle due fasi viene semplificata (anche gli spazi bianchi diventerebbero simboli terminali della grammatica complicandola ulteriormente);
- miglioramento dell'efficienza (esistono tecniche specializzate per l'analisi lessicale);

#### 3.2. Token

È una coppia costituita da un nome del token e da un attributo opzionale; il nome del token è un simbolo astratto che rappresenta un tipo di unità lessicale o unità logica di informazione nel programma sorgente.

Per esempio:

- le parole chiave di un linguaggio (keywords) come per esempio: if, while, do, then, else, ...
- gli identifieri: stringhe definite dall'utente costituite da lettere e numeri che iniziano con una lettera.
- Numeri
- stringhe di caratteri: sequenze di caratteri comprese tra virgolette
- operatori: simboli come \*, +, ... o simboli mult(carattere come >=, <=, <>, ...
- simboli speciali: parentesi, ., ;, ...

(Spesso quando si parla di token si intende il nome del token)

### 3.3. Lessema (Lexeme)

È la sequenza di caratteri nel programma sorgente identificati dallo scanner, ovvero una specifica istanza di un token.

### 3.4. Pattern

È la descrizione compatta della forma che il lessema di un token può assumere. Nel caso delle keyword, il pattern è proprio la sequenza dei caratteri della keyword. Nel caso di identifieri o altri token, il pattern è una struttura in grado di matchare con molte stringhe.

TOKEN	PATTERN	LESEMMI
if	Caratteri i,f	if
else	Caratteri e,l,s,e	else
Op.confronto	< or > or <= or >= or == or !=	<=, !=
Identificatore	Lettera seguita da lettera o cifra	pi, rate, d3
Numero	Qualsiasi costante numerica	3.14, 67, 5.02e23
Stringa	Qualsiasi sequenza di caratteri diversi da "", circondati da ""	"ciao ciao"

### 3.5. Attributi

Sono particolari informazioni relative ad uno specifico lessema di un particolare token. E' necessario quando più lessemi possono matchare con un pattern. E' quindi importante identificare in qualche modo il lessema. Il nome del token influenza le decisioni di parsing, l'attributo la traduzione del token dopo il parsing.

- Esempio:  $E=M*C^2$ 
  - <id, punt. nella symbol table alla entry relativa a E>
  - <op\_ass>
  - <id, punt. nella symbol table alla entry relativa a M>
  - <op\_molt>
  - <id, punt. nella symbol table alla entry relativa a C>
  - <op\_exp>
  - <num, valore intero 2>

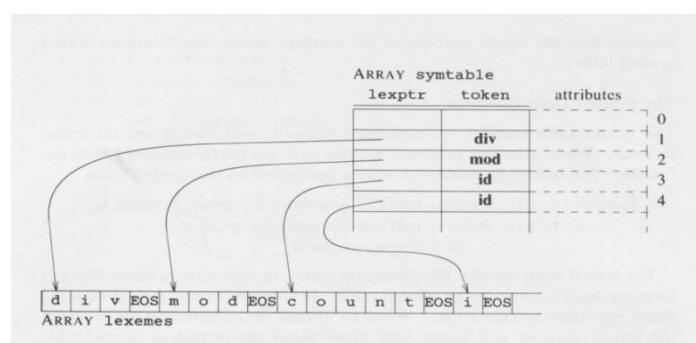
### 3.6. Implementazioni della symbol table

Spesso non è opportuno assegnare uno spazio fissato ad un lessema (troppo per alcuni lessemi, troppo poco per altri).

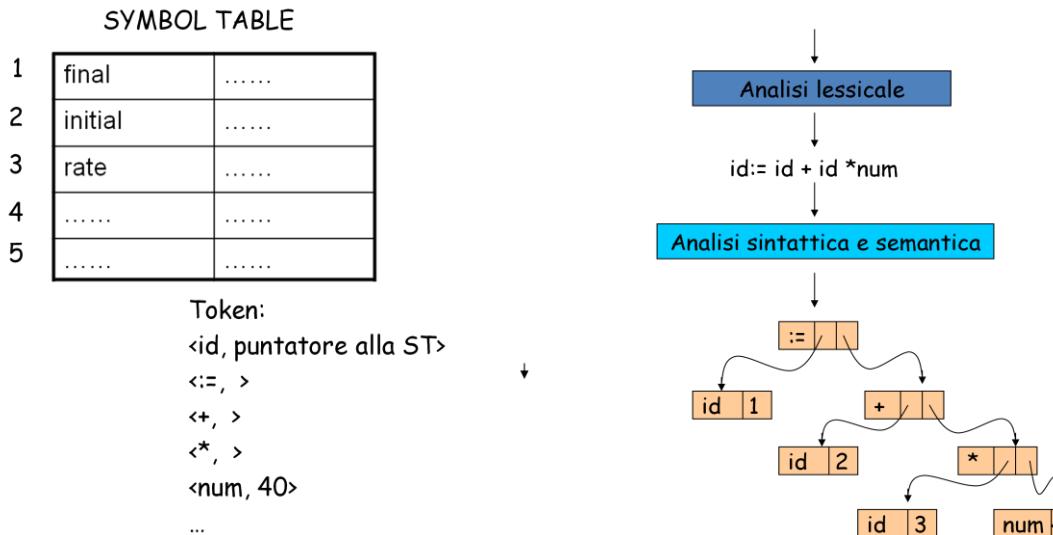
Un array di lessemi separato mantiene le stringhe che formano gli identifieri.

Lo scanner comunica con la ST con operazioni del tipo

- insert(s,t): restituisce l'indice di una nuova entry per la stringa s, token t
- lookup(s): restituisce l'indice della entry per s, 0 se non la trova. Può riconoscere le keywords



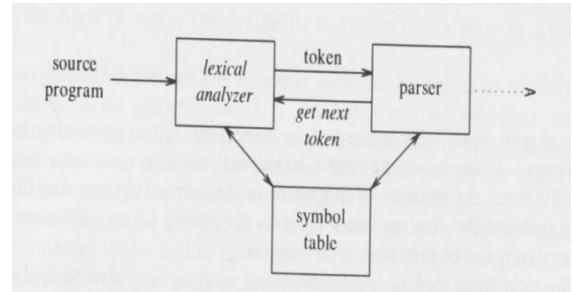
Esempio:



### 3.7. Ruolo dello scanner

I token prodotti dallo scanner saranno usati dal parser per l'analisi sintattica.

Questa interazione, descritta in figura, è comunemente implementata rendendo lo scanner una procedura o routine che legge un carattere per volta finché non identifica un token. Tale procedura agisce su richiesta dell'analizzatore sintattico.



### 3.8. Errori lessicali

Un analizzatore lessicale ha una visione molto localizzata del codice sorgente, quindi è in grado di riconoscere pochi errori.

Per esempio dato il codice “`fi a=f(x) then ...`”, in genere uno scanner non può sapere se “`fi`” è un misspelling di “`if`”, oppure se è un identificatore non dichiarato.

Se uno scanner non riconosce una sequenza di caratteri come token può evidenziare l'errore oppure potrebbe essere dotato di strategie di recupero dell'errore:

- Cancellare caratteri successivi dal resto dell'input finché non riconosce un token;
- Cancellare, inserire, sostituire un carattere nel resto dell'input;
- Effettuare correzioni locali sulla base di un calcolo del criterio minimum-distance (è un metodo costoso da implementare poiché consiste nel trovare il più piccolo numero di trasformazioni necessarie per ottenere un lessema valido.).

### 3.9. Come implementare uno scanner

Esistono 3 approcci:

- Usare un generatore automatico di analizzatori lessicali, come Lex, Flex, .... Il generatore fornisce anche le routine per la lettura dell'input.
- Scrivere l'analizzatore lessicale come un programma in un linguaggio di programmazione convenzionale. L'input si gestisce sfruttando le caratteristiche del linguaggio.
- Scrivere l'analizzatore lessicale come un programma in linguaggio assembly. In tal caso l'input deve essere gestito a basso livello.

### 3.10. Come descrivere i token

Definiremo le regole lessicali per un linguaggio di programmazione attraverso le espressioni regolari.

Le espressioni regolari sono un'importante strumento per specificare i pattern. Rappresentano un modo compatto di denotare quali caratteri possono costituire un lessema che appartiene ad una certa classe di token.

Un analizzatore lessicale è uno strumento in grado di riconoscere i pattern, ovvero un automa a stati finiti.

### 3.11. Linguaggi formali (definizioni)

- Alfabeto: Insieme finito di simboli
- Stringa o parola: sequenza finita di simboli dell'alfabeto
- Lunghezza di una parola: numero dei caratteri che la compongono
- Parola vuota: parola di lunghezza 0 (denotata con  $\epsilon$ )
- Linguaggio: insieme finito o infinito di parole su un dato alfabeto
- Operazioni sui linguaggi:

$L \cup M$	unione	$\{s \mid s \in L \text{ o } s \in M\}$
$LM$	concatenazione	$\{st \mid s \in L \text{ e } t \in M\}$
$L^*$	chiusura di Kleene	zero o più concatenazioni di $L$
$L^+$	chiusura positiva	una o più concatenazioni di $L$

### 3.12. Espressioni regolari

Una espressione regolare è una delle seguenti:

- l'espressione  $\epsilon$  denota il linguaggio  $L=\{\epsilon\}$
- l'espressione  $\phi$  denota il linguaggio vuoto
- l'espressione  $a$  denota il linguaggio  $L=\{a\}$
- se  $r$  ed  $s$  sono espressioni regolari che denotano i linguaggi  $L(r)$  e  $L(s)$ ,
  - $r|s$  denota  $L(r) \cup L(s)$
  - $rs$  denota  $L(r)L(s)$
  - $r^*$  denota  $L(r)^*$
  - $(r)$  denota  $L(r)$

Regole di precedenza: operatore  $*$ , poi concatenazione e infine  $|$ . Tali operazioni sono associative a sinistra. I linguaggi definiti mediante espressioni regolari sono detti linguaggi regolari.

Esempi:

- $bbb$  è un'espressione regolare che denota ...
  - $\{bbb\}$ .
- $ab|bbb$  è un'espressione regolare che denota ...
  - $\{ab, bbb\}$ .
- $ba^*$  è un'espressione regolare che denota ...
  - $\{b, ba, baa, baaa, \dots\}$ .
- $b(a|b)^*b$  |  $b$  è un'espressione regolare che denota ...
  - {stringhe che cominciano e finiscono per  $b$ }.
- $(a|b)(a|b)$  è un'espressione regolare che denota ...
  - $\{aa, ab, ba, bb\}$ .

Notazioni:

- **Una o più ripetizioni:** data un'espressione regolare  $r$ , denotiamo con  $r^+$  una o più concatenazioni di  $r$ ;
- **Zero o una istanza:** denotiamo con  $r?$  l'espressione  $r|\epsilon$  ;
- **Carattere jolly:** denotiamo con  $"."$  un'espressione che individua un qualsiasi carattere dell'alfabeto. Per es.  $".*b.*"$

- **Range di caratteri:**  $[a-z]$  denota l'espressione  $a|b|\dots|z$ ;  $[a-zA-Z]$  denota un range multiplo
- **Carattere non presente in un dato insieme:**  $[^abc]$  denota un carattere diverso da a, b e c;  $[^a]$  denota un carattere diverso da a

### 3.13. Definizioni regolari

Sia A l'alfabeto dei simboli. Una definizione regolare è una sequenza di definizioni della forma:

$$\begin{array}{lll} d_1 & \rightarrow & r_1 \\ d_2 & \rightarrow & r_2 \\ d_3 & \rightarrow & r_3 \\ \dots & & \\ d_n & \rightarrow & r_n \end{array}$$

dove ogni  $d_i$  ha un nome distinto e ogni  $r_i$  è un'espressione regolare sull'alfabeto  $A \cup \{d_1, d_2, d_3, \dots, d_{i-1}\}$ .

Con queste notazioni definiamo...

- I numeri naturali, interi, reali in notazione esponenziale  
 $\text{nat} = [0-9]^+$   
 $\text{signedNat} = (+|-)? \text{nat}$   
 $\text{number} = \text{signedNat}(\text{.} \text{nat})? (\text{E signedNat})?$
- Keywords  
 $\text{reserved} = \text{if} | \text{while} | \text{do} | \dots$
- Identificatore  
 $\text{letter\_} = [a-zA-Z]_$   
 $\text{digit} = [0-9]$   
 $\text{id} = \text{letter\_}(\text{letter\_} | \text{digit})^*$
- Commento in Pascal  
 $\{\{\}^*$

### 3.14. Ambiguità

- Un lessico può essere definito mediante espressioni regolari multiple
  - Ciò può condurre ad ambiguità
- Per l'input "begin", la RE  $[a-z]^+$  individua
  - Ogni prefisso: "b", "be", "beg", etc.
  - Quali token scegliere?
- Per l'input "begin", entrambe le due RE vanno bene
  - begin
  - $[a-z]^+$

Quali espressioni regolari applicare?

Eliminiamo l'ambiguità:

- **Longest Match:** Viene considerata come lessema la più lunga stringa che ha un match con un'espressione regolare.  
 Esempio  
 $[a-z]^+$  prosegue la ricerca del match con caratteri minuscoli, non si ferma al primo match.
- **Regole di priorità:** Se due espressioni regolari hanno entrambe un match, allora la prima espressione regolare è quella che determina il match e quindi il tipo di token.  
 Esempio  
 "if" è tokenizzato come IF, non come ID.

### 3.15. Esistono linguaggi non regolari

Il linguaggio  $S=\{ab, aabb, aaabbb, \dots\}$  non può essere definito mediante un'espressione regolare.

Le espressioni regolari possono essere usate per denotare solo un numero fissato di ripetizioni o un numero non specificato di ripetizioni di un dato costrutto. Non possono essere usate invece per denotare costrutti bilanciati o annidati.

Uno strumento utile per mostrare che un dato linguaggio non è regolare è il Pumping Lemma.

### 3.16. Come riconoscere i token

Nel 1956 Kleene ha dimostrato l'equivalenza tra le espressioni regolari e gli automi a stati finiti deterministici.

Gli automi a stati finiti sono un strumento efficace per riconoscere i token.

Un analizzatore lessicale è uno strumento in grado di riconoscere i pattern (espressioni regolari), ovvero un **automa a stati finiti**.

Esempio:

Supponiamo di conoscere la grammatica che regola il nostro linguaggio e soprattutto i suoi simboli terminali

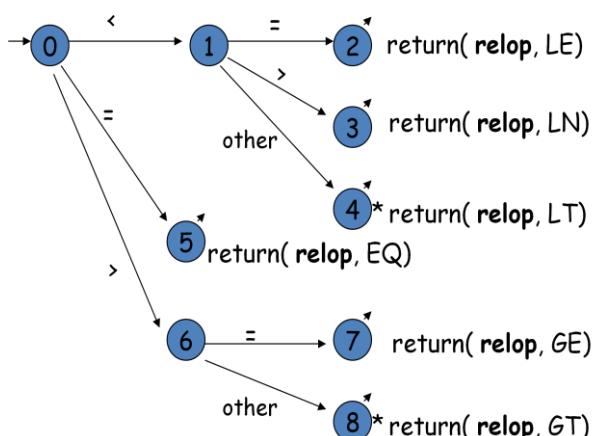
```
stmt -> if expr then stmt | if expr then stmt else stmt | ε  
expr -> term relop term | term  
term -> id | number
```

I simboli terminali della grammatica diventano i token

Ad es:

```
nat = [0-9]+  
signedNat = (+|-)? nat  
number = signedNat("." nat)? (E signedNat)?  
letter_=[a-zA-Z]  
id = letter_(letter_|nat)*  
if= if  
then=then  
else=else  
relop=<|>|<=|>|=|<>  
ws=(blank|tab|newline)+
```

### 3.17. Come riconoscere i token relop



Il simbolo \* indica che il puntatore di lettura deve ritrarsi di una posizione.

Ad ogni stato finale è associata un'azione.

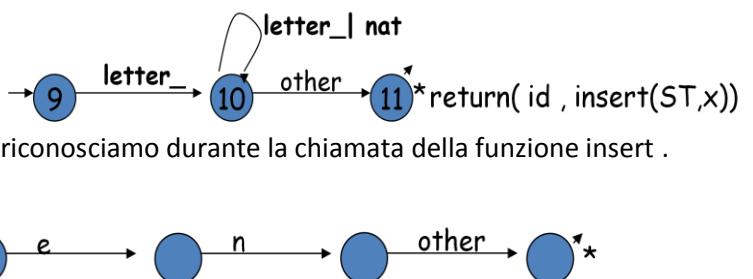
### 3.18. Riconoscimento di keyword e identificatori

La funzione insert inserisce il token nella ST se non è già presente e restituisce il puntatore alla locazione.

Per le keywords:

O si inseriscono nella ST inizialmente e quindi li riconosciamo durante la chiamata della funzione insert .

Altrimenti:



Bisogna fare in modo di dare priorità al riconoscimento delle keyword.

### 3.19. Tecniche per la gestione dell'input

In teoria lo scanner analizza la stringa sorgente un carattere per volta. In pratica deve essere in grado di accedere a sottostringhe della sorgente ovvero di tornare indietro di un blocco di caratteri, prima di poter annunciare un match.

- Un identificatore viene riconosciuto solo dopo che si incontra un carattere diverso da numeri e lettere.
- Gli operatori -, =, < sono prefissi di ->, ==, <=.

Visto l'elevato numero di caratteri e il tempo necessario ad elaborarli, non sarebbe conveniente invocare comandi di lettura per ogni carattere di input e tornare eventualmente indietro. Normalmente può essere più conveniente leggere N caratteri di input con un unico comando di lettura del sistema. Quindi, la stringa sorgente è letta attraverso un'area buffer cosicché lo scanner possa anche tornare indietro più facilmente. Normalmente si usa una tecnica chiamata "double buffering", ovvero due buffer ciascuno di dimensione N che vengono riempiti alternativamente. In genere N è scelto pari alla dimensione del blocco del disco (per es. 4K).

Per esempio:

```

If dist >= rate * (end - start) then dist := max;
For i:=1 to r do
  rate := dist * i;
  ↓
  If dist >= rate * (end - s | tart) then dist := max; Fo
  ↑↑↑↑↑↑
  | Carattere corrente
  ↓
  Inizio del lessema
  ↓
  r i:=1 to r do rate:=di | tart) then dist := max; Fo
  
```

Per ogni carattere, per fare avanzare il puntatore corrente bisogna testare 2 condizioni (verifica se si è in posizione finale e verificare quale carattere è stato letto). Possiamo ridurre i 2 test a 1 introducendo un carattere sentinella (eof) alla fine di ogni metà. Solo se il carattere è eof si effettua il test ulteriore.

### 3.20. Automi a stati finiti

Un Automa Finito Deterministico (DFA) è una quintupla  $(Q, A, \delta, q_0, F)$

- $Q$  è l'insieme di stati
- $A$  è un insieme di simboli (alfabeto di input)
- $\delta$  è la funzione di transizione dell'automa ed associa alle coppie stato-simbolo uno stato  $\delta: Q \times A \rightarrow Q$
- $q_0$  è uno stato particolare detto lo stato iniziale dell'automa
- $F$  è un insieme di stati detti stati finali (o di accettazione) dell'automa.

Funzione di stato su una stringa  $x$

- $\delta'(q, e) = q$  per ogni  $q$  in  $Q$ ;  $\delta'(q, xa) = \delta(\delta'(q, x), a)$
- $\delta'(q, x) = \text{stato in cui si trova l'automa dopo aver letto tutti caratteri di } x$ .

Linguaggio accettato dall'automa:  $L = \{x \in S^*: \delta'(q_0, x) \in F\}$

Un DFA può essere implementato:

- Mediante il diagramma di transizione
  - Può essere facilmente implementato con un programma (si usa una variabile che tiene conto dello stato in cui ci si trova)
- Mediante la matrice di transizione (le righe sono gli stati, le colonne i simboli; ogni cella contiene lo stato di arrivo corrispondente)
  - Può essere implementato in modo semplice con un programma
  - La taglia del codice è ridotta
  - È facile da cambiare
  - Lo svantaggio è che le tabelle possono diventare molto grandi

### 3.20.1. Simulare un DFA con diagrammi di transizione

```
....  
switch (state) {  
    case 0: c=nextchar();  
              if (c=='<') state =1;  
              else if (c=='>') state =2;  
              else ...  
    case 1: ....  
    ....  
    case 5: return ('yes');  
}
```

### 3.20.2. Simulare un DFA con matrice di transizione

```
state=s0  
c=nextchar();  
while (c!=eof) {  
    state=tab(state,c);  
    c=nextchar();  
}  
if (state in F) return 'yes';  
else return 'no';
```

## 3.21. Il problema è un po' più complesso...

I DFA sono un modo per rappresentare algoritmi che riconoscono stringhe in accordo con certi pattern.

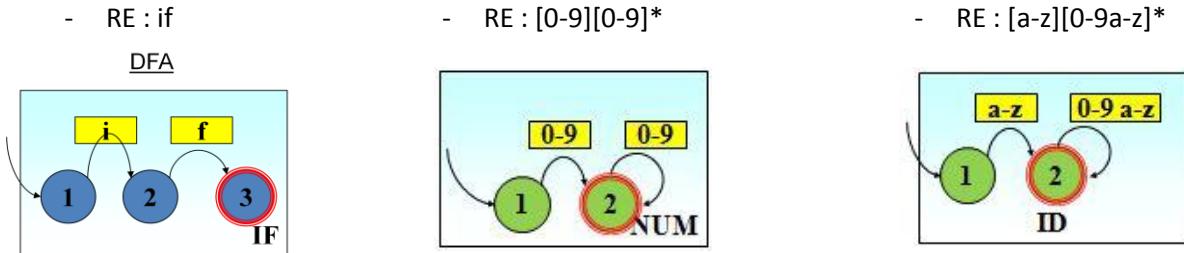
I diagrammi o le tabelle non descrivono tutti gli aspetti del comportamento di un algoritmo DFA. Per esempio

- non descrivono cosa succede in presenza di errore.
- non descrivono cosa succede in caso di arrivo su uno stato di accettazione.
- dovrebbero tener conto anche delle operazioni di Lookahead e Backtracking, ovvero applicare le regole per eliminare le ambiguità.
- Bisogna arricchirli con le azioni corrispondenti.

### 3.22. Come costruire l'automa

- Definire le espressioni regolari che definiscono i token.
- Trovare il DFA corrispondente (teorema di Kleene)

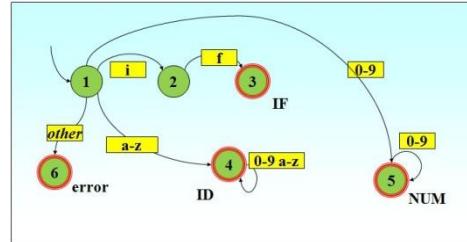
Esempio 1:



Scrivere il codice corrispondente al diagramma di transizione è molto semplice.

Ma comporre DFA non è così immediato:

Si ottengono automi finiti non deterministici



### 3.23. Automi a stati finiti non-deterministici (NFA)

Sono quasi come i DFA, tranne che hanno più di una transizione per input e  $\epsilon$ -transizioni.

Gli automi non-deterministici

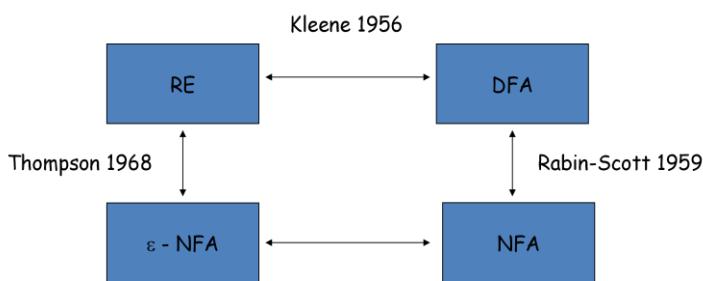
- Più semplici da creare.
- Difficili da eseguire in modo efficiente.

Automati deterministici

- Non hanno  $\epsilon$  transizioni.
- Da uno stesso stato non escono due archi con la stessa etichetta.
- Sono più facili da implementare.

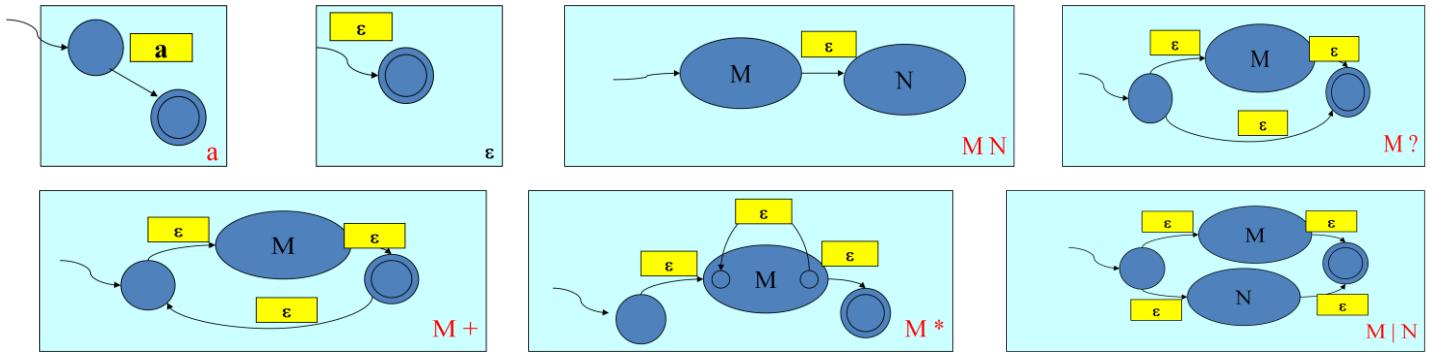
E' possibile creare l'automa non deterministico e poi passare al deterministico.

### 3.24. Espressioni regolari ed automi



Ancora prima dello sviluppo di UNIX, Ken Thompson aveva studiato l'impiego delle espressioni regolari in comandi come grep-Global (search for) regular expression and print

### 3.25. Da espressione regolare a NFA

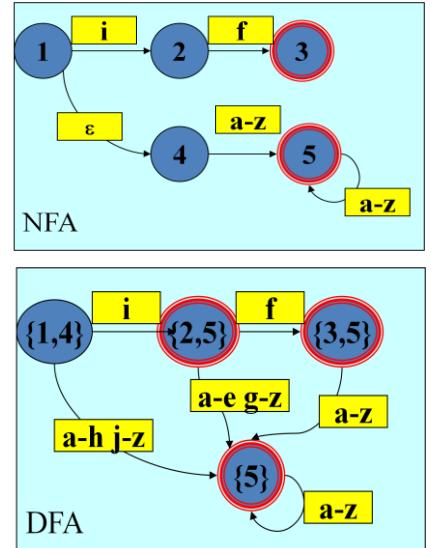


### 3.26. Da NFA a DFA

- Ogni stato in DFA corrisponde ad un insieme di stati in NFA.
- Può produrre un DFA con  $2^N$  stati
- Esistono algoritmi efficienti per trasformare un NFA in DFA.
  - I generatori di analizzatori lessicali li usano!

Subset Construction:

- Calcola la  $\epsilon$ -chiusura dello stato 1 in NFA: {1,4}
  - Gli stati Raggiungibili da 1 attraverso  $\epsilon$
- Crea le transizioni da ogni stato di {1,4}
  - su "i" si ha: {2,5}
  - calcolare la  $\epsilon$ -chiusura di {2,5} non ci dà un nuovo stato in DFA
  - {2,5} è uno stato finale in DFA poiché 5 è finale in NFA
- Ripeti finchè non hai preso in considerazione tutti gli stati



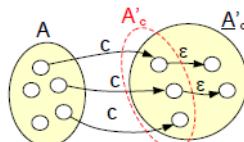
$$\text{NFA } \xrightarrow{\text{DFA}} \text{M' } \left\{ \begin{array}{l} M = (\Sigma, S, T, s_0, F) \\ M' = (\Sigma, S', T', s'_0, F'), \text{ in cui} \end{array} \right. \left\{ \begin{array}{l} S' \subseteq 2^S \\ T': S' \times \Sigma \rightarrow S' \\ s'_0 = s_0 \\ F' \subseteq S' \end{array} \right.$$

- Computazione di  $S'$ ,  $T'$ ,  $F'$ :

```

S' := { s'_0 }; T' := ∅;
repeat
  Scegli un nodo A ∈ S' non marcato;
  for each c ∈ Σ che marca una transizione di M uscente da un nodo in A do
    begin
      A'_c := { z | s ∈ A, s →^c z ∈ T };
      A'_c := ε-chiusura(A'_c);
      if A'_c ∉ S' then S' := S' ∪ { A'_c };
      T' := T' ∪ { A →^c A'_c }
    end;
    Marca A
  until tutti gli elementi in S' sono marcati;
  F' := { A | A ∈ S', A ∩ F ≠ ∅ }.

```



### 3.27. Simulare un NFA

Abbiamo bisogno di 2 pile:

**Oldstates**, memorizza l'insieme corrente di stati (parte dx, linea 4).

**Newstates**, memorizza gli stati successivi (parte sx, linea 4).

- Per ogni iterazioni Newstates diventa Oldstates.
- Un array booleano indica quali stati stanno in Newstates.
- Una tabella di transizione in cui nell'elemento  $(t,x)$  ci possono essere liste di stati.
- N stati, M transizioni  
 $O(k(N+M))$ ,  $k = |x|$

```

    Stati
    correnti
S=ε-closure(s0)
c=nextchar();
while (c!=eof) {
    S=ε-closure(move(S,c));
    c=nextchar();
}
if (S ∩ F != ∅) return 'yes';
else return 'no';

```

### 3.28. NFA o DFA ?

Data una RE  $r$  e una stringa  $s$ , esistono due strategie per testare se  $s$  sta in  $L(r)$ :

- Costruire l'NFA da  $r$  in  $O(|r|)$  tempo.  
Infatti il numero di stati è proporzionale ad  $|r|$  e ci sono al più 2 transizioni per ogni stato. La tabella di transizione viene memorizzata in  $O(|r|)$  spazio.  
Il riconoscimento di una stringa  $s$  mediante un NFA con  $|r|$  stati può essere simulato efficientemente mediante due stack in  $O(|r| \times |s|)$ .
- Costruire il DFA da  $r$  applicando la subset construction ed eventualmente la minimizzazione. La subset construction costa un tempo  $O(|r|^2 M)$  dove  $M$  è il numero di stati del DFA ottenuto. Nel caso medio in cui  $M$  è circa  $|r|$ , allora  $O(|r|^3)$ .  
Il riconoscimento di una stringa  $s$  mediante il DFA può essere simulato in  $O(|s|)$ . Lo spazio può diventare  $O(2^{|r|})$ .

Automa	Costo iniziale	Costo per stringa
NFA	$O( r )$	$O( r  \times  s )$
DFA, caso medio	$O( r ^3)$	$O( s )$
DFA, caso peggiore	$O( r ^2 2^{ r })$	$O( s )$

- I generatori di analizzatori lessicali e altri sistemi di string processing partono da espressioni regolari.
- Se lo string-processor deve essere usato più volte (è il caso dell'analizzatore lessicale) allora il costo della costruzione del DFA è sopportabile. Quindi si preferisce costruire il DFA o passando dal NFA o direttamente e poi applicando algoritmi di minimizzazione.
- Nel caso di applicazioni come grep in cui l'utente specifica un'espressione regolare, allora conviene simulare direttamente un NFA.
- Esistono anche strategie miste, in cui si comincia con la simulazione del NFA, memorizzando però gli insiemi degli stati e le transizioni via via calcolati. Ad ogni passo si vede se una data transizione è stata già calcolata.

Si può costruire un DFA direttamente a partire da un'espressione regolare senza passare per il NFA. In alcuni casi si ottengono DFA con minor numero di stati

Si può minimizzare il DFA ottenendo il DFA minimale. Il tempo è  $O(n \log n)$ .

Rappresentazione compatta delle tabelle di transizione.

## 4. Generatori automatici di scanner (FLEX)

### 4.1. Generatori di scanner

Un generatore automatico di scanner prende in input un file che specifica il lessico di un certo linguaggio:

- solitamente nella forma di espressioni regolari
- ...e includendo altre funzioni ausiliarie, definizioni di token, ...

Produce in output un codice (scritto in un certo linguaggio) che implementa il ruolo dello scanner.

Esistono molti generatori automatici:

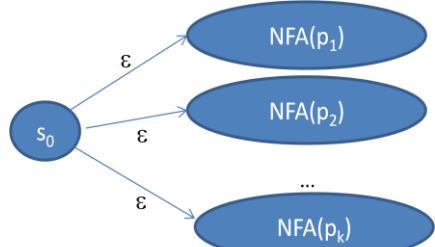
- Lex, Flex, ScanGen, ... (generano un codice in C)
- JLex, Sable, Cup (generano un codice in Java)

Lex fu il primo generatore di scanner. Esso fu inventato da Mike Lesk e Eric Shmidt (AT&T Bell Lab) nel 1975.

Esistono tanti software alternativi al Lex. Uno dei più conosciuti ed usati è Flex (Fast Lexical analyser generator) introdotta da Vern Paxson intorno al 1987 per risolvere problemi di efficienza.

### 4.2. Come funziona Flex

- Si descrivono tutte le espressioni regolari che definiscono i token;
- Converte le espressioni regolari in automi non deterministici;
- Combina tutti i NFAs in un unico NFA;
- Converte il NFA in un automa deterministico (DFA) mediante la subset construction;
- Quando uno stato del DFA contiene uno o più stati di accettazione è necessario stabilire qual è il primo;
- Ottimizza l'automa deterministico;
- Produce codice per simulare l'azione del DFA.



Come si realizza il riconoscimento del longest best match:

- Si legge il testo un carattere per volta e mantiene memoria degli stati intermedi
- Appena si arriva in uno stato pozzo o non ci sono le transizioni uscenti, si cerca a ritroso lo stato di accettazione.
- È possibile usare la pila in cui si inseriscono via via gli stati di accettazione incontrati durante il processo di matching.

### 4.3. Programmare in Flex

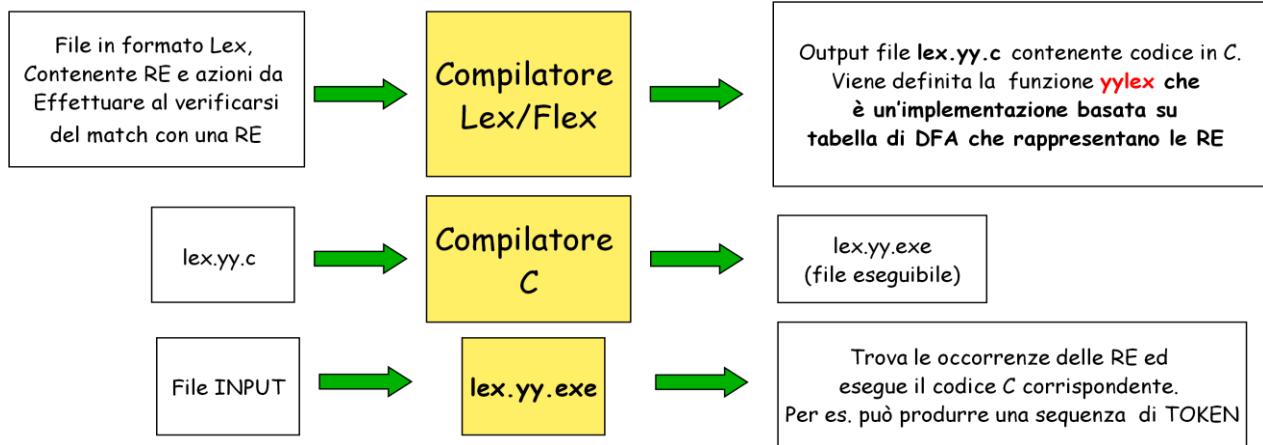
Flex è un free software. Pur non essendo un software GNU, il GNU Project ne distribuisce un manuale.

- <http://flex.sourceforge.net/> (linux)
- <http://gnuwin32.sourceforge.net/packages/flex.htm> (windows)

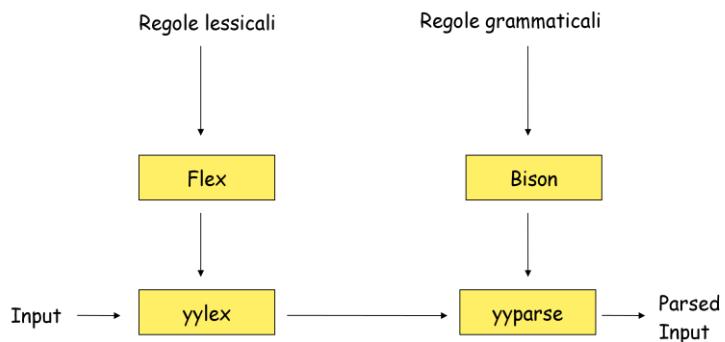
Una volta installati Flex e Bison bisogna impostare le variabili d'ambiente. In Windows7 :

- dx su Computer -> Proprietà -> Cambia Impostazioni -> Avanzate -> Variabili d'ambiente
- In "variabili di sistema" selezionate "Path", quindi "Modifica"; in "Valore variabile" aggiungete ";" (punto e virgola) seguito dalle path degli eseguibili dei programmi precedentemente installati, ovvero ;C:\Program Files\GnuWin32\bin

Uso del Flex:



Uso di Flex e Bison:



Un file in formato Lex o Flex consiste di 3 sezioni, separate da %%:

#### DEFINIZIONI

(contiene:

- definizioni di variabili o definizioni regolari
- segmento di codice C, indentato oppure delimitato da %{ e %}, che deve comparire nel file di output )

%%

#### REGOLE

(contiene una sequenza di regole contenenti:

- i pattern espressi mediante RE
- codice C da eseguire in corrispondenza del match con un certo pattern)

%%

#### FUNZIONI AUSILIARIE (opzionale)

(contiene codice C che deve essere copiato nel file di output)

Anche nella sezione REGOLE è possibile inserire codice tra %{ e %}. Ciò deve avvenire prima della prima regola. Può servire per esempio per dichiarare variabili locali usate nella routine di scanning.

Il primo esempio: (primo.fl):

%%

Esiste la regola di default: il testo che non matcha con nulla viene ricopiat o sull'output. Quindi il nostro primo programma (con scritto solamente %%) genera uno scanner che semplicemente copia il suo input (un carattere per volta) nel suo output.

Sia nella sezione delle definizioni che in quella delle regole:

- ogni testo indentato o racchiuso tra %{ e %} viene copiato parola per parola sull'output.
- %{ e %} devono apparire a inizio di linea (non indentati).
- nella sezione delle definizioni, un commento non-indentato che comincia con /\* e finisce con \*/ diventa un commento nel file di output

Come si usa:

1. flex primo.fl
2. gcc lex.yy.c -o primo
3. primo < prova.txt >output.txt

#### 4.3.1. Come definire i pattern nella sezione delle regole

x	match con il carattere 'x'
.	qualsiasi carattere escluso il newline
[xyz]	una "classe di caratteri"; il pattern matcha o con 'x', 'y', o 'z'
[abj-oZ]	una "classe di caratteri" contenente un range; matcha con 'a', 'b', una qualsiasi lettera da 'j' a 'o', oppure 'Z'
[^A-Z]	una "classe negata di caratteri", i.e., qualsiasi carattere escluso quelli della classe. In questo caso, un qualsiasi carattere escluso le lettere maiuscole.
[^A-Z\n]	qualsiasi carattere tranne le maiuscole e il newline

Con le parentesi quadre solo \ - e ^ sono caratteri speciali; per includere il carattere – questo deve apparire come primo o ultimo carattere.

r*	zero o più espressioni r
r+	una o più espressioni r
r?	zero o una r
r{2,5}	da due a cinque r
r{2,}	due o più r
r{4}	esattamente 4 r
{name}	espansione della definizione "name"
"[xyz]\\"foo"	la stringa: [xyz]"foo

il carattere \ ha il ruolo di carattere di escape.

\x	se x è 'a', 'b', 'f', 'n', 'r', 't', o 'v', allora \x ha la stessa interpretazione che per ANSI-C. Altrimenti, sta per il carattere x. (usato per l'escape di operatori come '*')
\0	carattere NULL (ASCII code 0)
\123	carattere con valore ottale 123
\x2a	carattere con valore esadecimale 2a
(r)	Espressione r; le parentesi servono per imporre delle precedenze
rs	espressione r seguita da s
r s	o r oppure s
r/s	una r ma solo se seguita da s. Il testo matchato da s è considerato incluso per effetto del "longest match", ma è restituito all'input prima che venga eseguita l'azione. Questo tipo di pattern è chiamato trailing context".

Valgono le regole di precedenza per le espressioni regolari; ad es. ciao|bu\* è equivalente a (ciao)|(b(u)\*)

<code>^r</code>	r, ma solo all'inizio di una linea	
<code>r\$</code>	r, ma solo alla fine di una linea.	Equivalente a "r/\n".
<code>&lt;s&gt;r</code>	r, ma solo nella start condition s	
<code>&lt;s1,s2,s3&gt;r</code>	stessa cosa, ma in una qualsiasi start condition s1, s2, o s3	
<code>&lt;*&gt;r</code>	r in una qualsiasi start condition, anche in una esclusiva.	
<code>&lt;&lt;EOF&gt;&gt;</code>	end-of-file	
<code>&lt;s1,s2&gt;&lt;&lt;EOF&gt;&gt;</code>	end-of-file in start condition s1 o s2	
<code>[:digit:]</code>	indica tutti i caratteri per cui "isdigit" restituisce true ES: <code>[[:alpha:][:digit:]]</code> è equivalente a <code>[a-zA-Z0-9]</code>	

#### 4.3.2. Come avviene il match

Quando viene determinato un match, il testo corrispondente al match viene reso disponibile attraverso la variabile globale **yytext** e la sua lunghezza viene memorizzata in **yyleng**.

**Vengono quindi eseguite le azioni corrispondenti al pattern per il quale avviene il match.**

Prosegue la scansione dell'input alla ricerca di altri match.

Se non viene trovato alcun match, viene eseguita la *default rule*: il carattere dell'input viene considerato un match e copiato nell'output

Nota: la variabile yytext può essere definita come char pointer o array; E' possibile controllare tale definizione attraverso la direttiva %pointer o %array nella sezione delle definizioni. Se si usa l'opzione -l (compatibilità con Lex), yytext è un array. % pointer consente scansioni più veloci

#### 4.3.3. Come definire le azioni

##### pattern      azione

Se l'azione è vuota (contiene solo ;), al match del pattern non succede nulla.

Se l'azione contiene un {, allora verranno eseguite le azioni fino al raggiungimento di }.

Un'azione che consiste solo di | indica che è definita nella stesso modo della regola che segue.

Es1:

```
|  
\t |  
\n ;
```

Es2:

`[a-z]+ {printf("%s", yytext);}` è equivalente a `[a-z]+ {ECHO;}`

Direttive speciali che possono essere incluse in un'azione:

ECHO: copia yytext nell'output dello scanner

BEGIN: seguito dal nome di una condizione START posiziona lo scanner in tale condizione

REJECT: dirige lo scanner a procedere sul secondo longest best match (di solito un prefisso dell'input)

#### 4.3.4. Variabili e routine disponibili all'utente

<code>yylex()</code>	routine di scanning
<code>yytext</code>	stringa con la quale avviene il match
<code>yyin</code>	input file (default: stdin)
<code>yyout</code>	output file (default: stdout)
<code>yyleng</code>	lunghezza di yytext
<code>unput(c)</code>	rimette il car c nella stringa dei simboli da esaminare
<code>input()</code>	legge il carattere successivo
...	

#### 4.3.5. Esempi:

- Costruire uno scanner che conta il numero di caratteri e il numero di linee dell'input:

```
/* Nella sezione definitions i commenti vengono copiati nel file di
output*/
%{
    int num_lines = 0, num_chars = 0;
%
%option noyywrap
%%
\n      {++num_lines; ++num_chars;}
.      {++num_chars;}
%%
void main()
{
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
           num_lines, num_chars );
}
```

- Costruire uno scanner che conta il numero di caratteri e il numero di linee di un file dato in input:

```
%{
    int num_lines = 0, num_chars = 0;
%
%option noyywrap
%%
\n      ++num_lines; ++num_chars;
.      ++num_chars;
%%
main(int argc, char *argv[])
{
    --argc; /* skip over program name */
    if ( argc > 0 )
        yyin = fopen( argv[1], "r" );
    else
        yyin = stdin;
    yylex();
    printf( "# of lines = %d, # of chars = %d\n",
           num_lines, num_chars );
}
```

- raddoppi tutte le occorrenze delle vocali in un file

```
%option main
vocali      [aeiouAEIOU]
%%
{vocali}    {ECHO;ECHO;}
```

- Mantenga solo le linee che finiscono o cominciano con una consonante, cancellando le altre.

```
cons          [b-df-hl-np-tv-z]
%option main
%%
^{cons}.*   ECHO;
.*{cons}$   ECHO;
.          ;
```

- comprima gli spazi e i tab in unico singolo blank, e rimuova quelli alla fine di una linea.

```
%option main
%%
[ \t]+      {printf(" ");}
[ \t]+[\n]  {printf("\n");}
```

- conti le occorrenze dei numeri multipli di 3 e dei multipli di 5

```
%{
int n_mult3=0;
int n_mult5=0;
%
%option noyywrap
%%
[^0-9]      ;
[0-9]*       {int num=atoi(yytext);
               if (num%3==0) ++n_mult3;
               if (num%5==0) ++n_mult5; }
%
main()
{   yylex();
    printf(" # multipli di 3 = %d\n # multipli di 5 = %d",
           n_mult3,n_mult5);
}
```

- Elimini il testo inserito tra { e }

```
%option main
%%
\{[^{}]+\}  {printf("{}");}
```

- Riscriva solo le occorrenze di begin e di be riportando il numero di riga.

```
%{
int cont=0;
%
%option noyywrap
%%
be | "begin"     {ECHO; printf(" linea num=%d\n",cont);REJECT; }
[\n]                cont=cont+1;
.
;
%
int main()
{ yylex(); }
```

- Riscrivere occorrenze begin e gin riportando il num.di riga

```
%option main
%{
int numlinea=0;
%
%option noyywrap
%%
gin | begin     {ECHO;printf(" linea:%d\n",numlinea);REJECT; }
[\n]                {numlinea++;}
.
;
```

- Sostuisce i caratteri A, \, B, ^, - con "carattere speciale"

```
%option main
%%
[A\\B\\^\\-]  {printf("carattere speciale\n");}
```

- Converte tutte le lettere maiuscole di un file in minuscole, ad eccezione di quelle racchiuse tra “ e ”

```
%option main
%%
[""] [A-Za-z]+[""] {ECHO; }
[A-Z]      {printf("%c", tolower(yytext[0])); }
```

- Restituisca il più grande dei numeri naturali presenti nel testo

```
%{
int num=0, num_max=0;
%
nat      0|[1-9][0-9]*
%option noyywrap
%%
{nat}+    {ECHO; num=atoi(yytext); if (num>num_max) num_max=num; }
%%
int main() {
yylex();
printf("\nIl numero naturale piu' grande e':%d", num_max);
}
```

- Conti le occorrenze dei numeri pari e di quelli dispari

```
%{
int numpari=0, numdispari=0;
%
%option noyywrap
%%
[0-9]*[13579]  {ECHO; numdispari++; }
[0-9]*[02468]  {ECHO; numpari++; }
%%
int main() {
yylex();
printf("\n # di numpari : %d \n # di numdispari : %d", numpari,
numdispari); }
```

- Converte un testo in alfabeto farfallino (Es: ciao -> cifialfafo)

```
%option main
vocali [aeiou]
%%
{vocali}  {printf("%sf%s", yytext, yytext); }
```

#### 4.3.6. Start Conditions

Flex fornisce un meccanismo di regole che si attivano in modo condizionato.

Le condizioni START sono dichiarate nella sezione con %s o %x (condizioni inclusive o esclusive).

Una condizione Start è attivata usando l'azione BEGIN (lo scanner si trova nella cond start);

Se la cond. è inclusiva le regole che non si trovano in nessuna delle condizioni START saranno attive. Se la cond. è esclusiva le sole regole ad essere attive saranno quelle qualificate con la cond. START attivata.

- Esempio di utilizzo di condizioni inclusive:

Copiare l'input cambiando la parola magic in first in tutte le linee che cominciano per a, in second in quelle che cominciano per b e in third in quelle che cominciano per c.

```
%s  AA BB CC
%option main
%%
^a  {ECHO; BEGIN AA;}
^b  {ECHO; BEGIN BB;}
^c  {ECHO; BEGIN CC;}
\n  {ECHO; BEGIN 0;}
<AA>magic  {printf("first");}
<BB>magic  {printf("second");}
<CC>magic  {printf("third");}
```

- Esempio di utilizzo di condizioni inclusive:

Scanner che riconosce e scarta i commenti mantenendo un contatore delle linee.

```
%x comment
|{
int linea=1;
}
%option main
%%
/*                      BEGIN(comment);
<comment>[^*\n] ; /* rimuove ciò che non è '*' */
<comment>*[^\n]* ;
<comment>\n      ++linea;
<comment> /*          BEGIN(INITIAL);
```

NOTA: BEGIN(0) o BEGIN(INITIAL) sono equivalenti e riportano lo scanner allo stato originale in cui nessuna condizione start è attiva.

- Scrivere un programma in FLEX che trasformi in maiuscole tutte le parole delle righe che cominciano con una parola di caratteri tutti minuscoli, in minuscole tutte le parole delle righe che cominciano con una parola di caratteri tutti maiuscoli. Le altre siano lasciate inalterate.

```
int i;
%option main
%x min2max max2min
%%
^[a-z]+/[ ]+ |
^[a-z]+$      {for(i=0;i<yylen;i++)
                printf("%c",toupper(yytext[i])); BEGIN(min2max);}
^([A-Z]+/[ ]+ |
^([A-Z]+$      {for(i=0;i<yylen;i++)
                printf("%c",tolower(yytext[i])); BEGIN(max2min);}
<min2max>[a-z] {printf("%c",toupper(yytext[0]));}
<min2max>\n    {ECHO;BEGIN(0);}
<max2min>[A-Z] {printf("%c",tolower(yytext[0]));}
<max2min>\n    {ECHO;BEGIN(0);}
```

#### 4.4. Prova in itinere 2010:

Analizzare codici crittografati è l'obiettivo del programma da realizzare. In particolare, con l'ausilio di Flex, si deve realizzare uno strumento che aiuti uno studioso a decifrare alcuni codici che venivano usati dai popoli Maya per comunicarsi strategie di attacco e di difesa durante i periodi di guerra.

Le lettere maiuscole singole sono articoli, le lettere minuscole singole sono preposizioni. Una riga valida può cominciare solo con articoli.

Per confondere l'avversario un codice potrebbe contenere anche righe non valide, che quindi non rispettano la precedente condizione. Ogni riga è composta da parole separate da spazi o caratteri di tabulazione. Per alcune di tali parole è stata trovata una strategia di decifrazione.

I sostantivi sono successioni di caratteri minuscoli consecutivi che cominciano per vocale. Se a partire dalla seconda lettera il numero di vocali è maggiore del numero dei consonanti il sostantivo è maschile, altrimenti è femminile.

I verbi sono successioni di cifre consecutive che cominciano per 0.

Gli aggettivi sono successioni di caratteri minuscoli consecutivi che cominciano per consonante. In particolare, un aggettivo è femminile se le vocali appaiono in ordine non decrescente e le vocali uguali compaiono consecutivamente. In caso contrario, è maschile.

Qualsiasi altro identificatore è non riconosciuto. Non si è ancora scoperto, per esempio, come si codificano gli avverbi.

Il programma deve produrre un file di testo che riporti l'elenco degli identificatori incontrati nelle righe valide nell'ordine in cui compaiono e per ogni identificatore devono essere descritte le seguenti informazioni:

Lessema dell'identificatore

Tipo (cioè articolo, preposizione, sostantivo, verbo, aggettivo o non riconosciuto)

Genere (femminile o maschile nel caso di sostantivi o aggettivi)

Infine il file deve riportare il numero di righe valide e il numero di righe non valide.

```
%{
int cont_riga_v=0,voc=0,cons=0,cont_riga_nonv=0;
%}
cons [b-df-hj-np-tv-z]
%x RIGAV SOST RIGANV

%option noyywrap
%%
^ [A-Z] [ \t]+          {printf("Lessema: %c \nTipo: articolo\n",
yytext[0]); cont_riga_v++; BEGIN RIGAV; }
^ [A-Z] [ \t]*\n        {printf("Lessema: %c \nTipo: articolo\n",
yytext[0]); cont_riga_v++; }
^\n                    {cont_riga_nonv++; }
^.                     {cont_riga_nonv++; BEGIN RIGANV; }

<RIGAV>[a-z] [ \t]+    {printf("Lessema: %c \nTipo: preposizione\n",
yytext[0]); }
<RIGAV>[a-z] [ \t]*\n    {printf("Lessema: %c \nTipo: preposizione
\n",yytext[0]); BEGIN 0; }
<RIGAV>[aeiou]/[a-z]+   {printf("Lessema: %c",yytext[0]); BEGIN SOST; }

<SOST>[aeiou]      {ECHO;voc++; }
<SOST>{cons}        {ECHO;cons++; }
<SOST>[ \t]+         {printf("\nTipo: sostantivo\n") ;
if (voc>cons) printf("Genere: maschile\n");
else printf("Genere: femminile\n");
voc=0;cons=0; BEGIN RIGAV; }
```

```

<SOST>[ \t]*\n  {printf("\nTipo: sostantivo\n") ;  

   if (voc>cons) printf("Genere: maschile\n");  

   else printf("Genere: femminile\n");  

   voc=0;cons=0; BEGIN 0;}  

<RIGAV>[0] [0-9]+[ \t]+    {printf("Lessema:%s", yytext);  

                           printf("\n Tipo: verbo\n");}  

<RIGAV>[0] [0-9]+\n      {printf("Lessema:%s", yytext);  

                           printf("Tipo: verbo\n");BEGIN 0;}  

<RIGAV>{cons}+a*{cons}*e*{cons}*i*{cons}*o*{cons}*u*{cons}*[ \t]+  

          {printf("Lessema :%s\n", yytext);  

           printf("Tipo: aggettivo\nGenere: femminile\n");}  

<RIGAV>{cons}+a*{cons}*e*{cons}*i*{cons}*o*{cons}*u*{cons}*[ \t]*\n  

          {printf("Lessema :%s", yytext);  

           printf("Tipo: aggettivo\n Genere: femminile\n");BEGIN 0;}  

<RIGAV>{cons}+[a-z]+[ \t]+    {printf("Lessema :%s\n", yytext);  

                           printf("Tipo: aggettivo\n Genere: maschile\n");}  

<RIGAV>{cons}+[a-z]+[ \t]*\n    {printf("Lessema :%s", yytext);  

                           printf("Tipo: aggettivo\nGenere: maschile\n"); BEGIN 0;}  

<RIGAV>.  

<RIGANV>.*  

<RIGANV>[ \t]*\n      BEGIN 0;  

%%  

int main() {  

yylex();  

printf("\n # righe valide = %d\n # righe non valide = %d",  

     cont_riga_v,cont_riga_nonv);  

}

```

## 5. Scanner del linguaggio TINY

Tiny è un linguaggio di programmazione molto semplice, usato solitamente in Computer Science nei corsi di Compilatori.

Usato e descritto in **Compiler Construction Principles and Practice by Kenneth C. Louden(1997)**

Il linguaggio contiene la lettura di numeri interi, somma, sottrazione aritmetiche e stampa dei valori numerici.

Il lessico di TINY è il seguente:

- Keywords:           **if else then repeat until read write end**
- Separatori:         `;` `()`
- Operatoria carattere singolo: `+` `-` `*` `/` `<` `=`
- Un identificatore consiste di una lettera seguita da lettere e/o cifre. Per esempio: `x`, `x2`, `xx2`, `x2x`, `End`, `END2`. Nota che `End` è un identificatore mentre `end` è una keyword. Inoltre `7x` non è un identificatore. Le stringhe all'interno di commenti non sono identificatori. (Nota che nella implementazione che vedremo in seguito gli identificatori sono sequenze di una o più lettere)
- I commenti sono stringhe comprese tra `{` e `}` e possono essere più lunghi di una linea.

La grammatica di TINY è la seguente:

- `<program>`       $\rightarrow$     `<stmt list>`
- `<stmt list>`      $\rightarrow$     `<stmt list> ; <stmt> | <stmt>`
- `<stmt>`           $\rightarrow$     `<if_stat> | <repeat_stat> | <assign_stat> | <read_stat> | <write_stat>`
- `<if_stat>`        $\rightarrow$     `if <expr> then <stmt list> end | if <expr> then <stmt list> else <stmt list> end`
- `<repeat_stat>`    $\rightarrow$     `repeat <stmt list> until <expr>`
- `<assign_stat>`    $\rightarrow$     `identificatore := <expr>`
- `<read_stat>`      $\rightarrow$     `read identificatore`
- `<write_stat>`     $\rightarrow$     `write <expr>`
- `<expr>`            $\rightarrow$     `<s_expr> <comp_op> <factor> | <s_expr>`
- `<comp_op>`       $\rightarrow$     `< | =`
- `<s_expr>`         $\rightarrow$     `<s_expr> <addop> <term> | <term>`
- `<addop>`         $\rightarrow$     `+ | -`
- `<term>`           $\rightarrow$     `<term> <mulop> <factor> | <factor>`
- `<mulop>`         $\rightarrow$     `* | /`
- `<factor>`       $\rightarrow$     `( <expr> ) | numero | identificatore`

Esempio di un programma in TINY:

```
{ Sample program
in TINY language-
computes factorial
}
read x;           { input an integer }
if 0 < x then    { don't compute if x <= 0 }
    fact:= 1;
    repeat
        fact:= fact*x;
        x := x-1
    until x = 0;
    write fact      { output factorial of x }
end
```

## Analizzatore lessicale in Flex

digit	[0-9]
number	{digit}+
letter	[a-zA-Z]
identifier	{letter}+
newline	\n
whitespace	[ \t]+
%%	
"if"	{returnIF;}
"then"	{returnTHEN;}
"else"	{returnELSE;}

Struttura del Compilatore TINY proposto da Louden:

globals.h	main.c
util.h	util.c
scan.h	scan.c
parse.h	parse.c
symtab.h	symtab.c
analyze.h	analyze.c
code.h	code.c
cgen.h	cgen.c

## Opzioni del compilatore

- NO\_PARSE Solo uno scanner
  - NO\_ANALYZE Solo parsere scanner
  - NO\_CODE Esegue l'analisi semantica ma non genera il codice

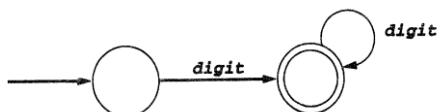
#### Opzioni per le informazioni da dare in output

- EchoSource Riscrive il codice del programma con i relativi numeri di linea
  - TraceScan Informazioni sui token incontrati
  - TraceParse Stampa il parse treelinearizzato
  - TraceAnalyze Informazioni su symboltable e typechecking
  - TraceCode Tiene traccia della generazione del codice

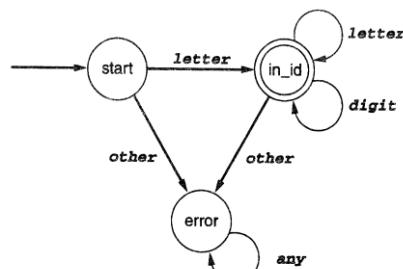
## Identificatori



### I numeri

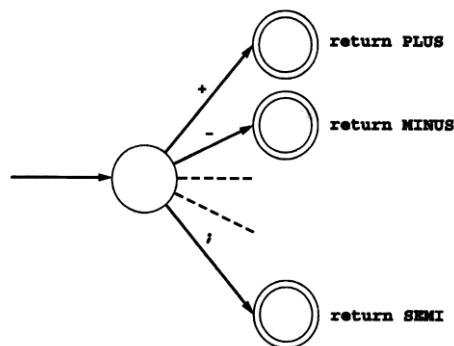
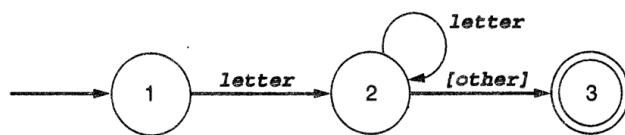


Se consideriamo anche gli errori

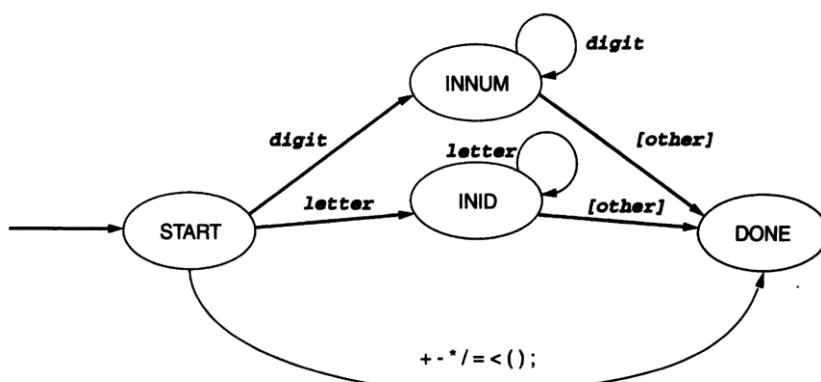


### Operatori e separatori

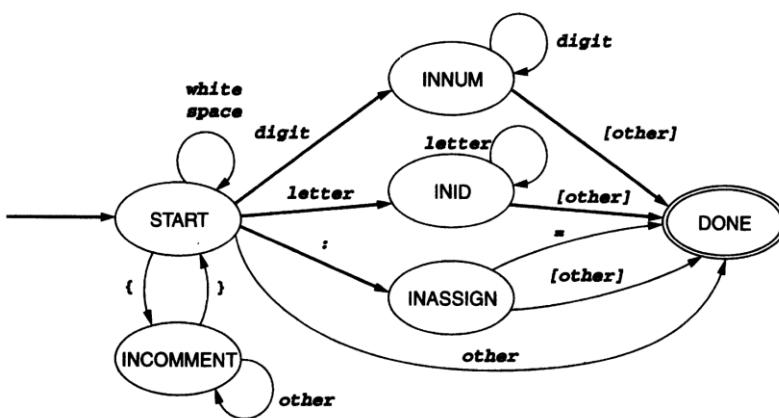
Se volessimo implementare il long est match  
[other] indica che la testina deve poi tornare indietro



Se usiamo altri indicatori per il token da produrre (ad es. una variabile nel codice) basta un solo stato



Per concludere



Le parole chiave possono avere automi a parte, oppure su può creare una look-up table con le parole chiave da controllare ogni volta che si trova un identificatore.

## 6. Analisi sintattica e grammatiche CF

### 6.1. Grammatiche context-free

Nell'ambito dei compilatori parliamo di linguaggi context-free visto che la sintassi dei costrutti di un comune linguaggio di programmazione può essere descritta da una grammatica context-free. Un linguaggio context-free è generato da una grammatica context-free.

Una grammatica context-free (CFG) è una quadrupla  $G=(T,N,S,P)$  dove:

- $T$  è l'alfabeto dei simboli terminali (= token lessicali);
- $N$  è l'alfabeto dei simboli intermedi o variabili o non terminali (= categorie grammaticali);
- $S \in N$  è l'assioma;
- $P$  è l'insieme delle produzioni della forma  $A \rightarrow \alpha$ , dove  $A \in N$  e  $\alpha \in (N \cup T)^*$

Esempi :

*instr* → if expr then instr else instr  
*frase* → soggetto verbo complemento

Il linguaggio generato da  $G$  è l'insieme delle stringhe di simboli terminali ottenute a partire dall'assioma con una o più derivazioni.

Una derivazione consiste nell'applicazione di una sequenza di una o più produzioni:  $\eta A \delta \Rightarrow \eta \alpha \delta$

Esempi:

- Grammatica che genera la struttura di un libro:  
 $S \rightarrow fA$  (f frontespizio, A serie di capitoli)  
 $A \rightarrow AtB \mid tB$  (t titolo, B serie di righe)  
 $B \rightarrow rB \mid r$  (r riga)
- Il linguaggio generato è l'insieme di tutte le stringhe che rappresentano la struttura corretta di un libro, ftrrrr
- Tale linguaggio è anche regolare,  $L=f(tr^+)^+$
- Esistono linguaggi context-free non regolari, per esempio:  $L=\{a^n b^n, n>0\}$   
 $S \rightarrow aSb \mid ab$

Linguaggi finiti e infiniti

- Linguaggio finito:  
 $S \rightarrow aBc$   
 $B \rightarrow ab \mid Ca$   
 $C \rightarrow c$   
 $L=\{aabc, acac\}$
- Linguaggio infinito  
 $G = (\{E, T, F\}, \{i, +, *, (), ()\}, P, E)$   
 $E \rightarrow E+T \mid T \quad T \rightarrow T^*F \mid F \quad F \rightarrow (E) \mid i$   
 $L = \{i, i+i, i+i+i, i^*i, (i+i)^*i, \dots\}$

- Analizziamo le tre derivazioni:  
 $E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow i+T \Rightarrow i+T^*F \Rightarrow i+F^*F \Rightarrow i+i^*F \Rightarrow i+i^*i$   
 $E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow E+T^*i \Rightarrow E+F^*i \Rightarrow E+i^*i \Rightarrow T+i^*i \Rightarrow F+i^*i \Rightarrow i+i^*i$   
 $E \Rightarrow E+T \Rightarrow E+T^*F \Rightarrow T+T^*F \Rightarrow T+F^*F \Rightarrow T+F^*i \Rightarrow F+F^*i \Rightarrow F+i^*i \Rightarrow i+i^*i$

## 6.2. Regole ricorsive

Sia  $G$  una grammatica pulita o ridotta, cioè:

1. ogni simbolo non terminale  $A$  è raggiungibile dall'assioma;
2. ogni simbolo non terminale  $A$  genera un linguaggio non vuoto;
3. non sono consentite derivazioni circolari:  $A \Rightarrow^* A$

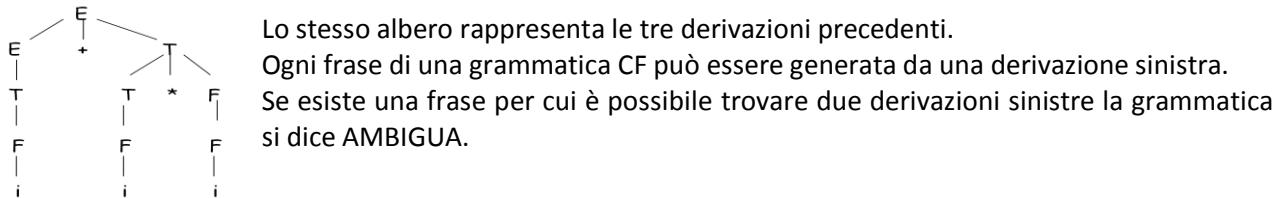
Condizione necessaria e sufficiente affinché  $L(G)$  sia infinito è che  $G$  permetta derivazioni ricorsive, ovvero del tipo  $A \Rightarrow^n xAy$

## 6.3. Alberi di derivazione o di parsing

Un **albero di derivazione** è una rappresentazione ad albero di una derivazione, in cui

- ogni simbolo è un nodo
- ogni simbolo è connesso al simbolo che l'ha generato

Nel caso dell'esempio precedente:



## 6.4. Grammatiche ambigue

Una grammatica si dice ambigua quando ci sono due alberi di parsing differenti per la stessa frase (o, equivalentemente, due derivazioni leftmost o sinistre per la stessa frase).

Un linguaggio per cui esistono solo grammatiche ambigue si dice **inherentemente ambiguo**;

Stabilire se una data CFG sia ambigua o se un dato linguaggio sia inherentemente ambiguo sono problemi indecidibili.

Per alcune applicazioni si usano metodi che coinvolgono grammatiche ambigue unite alle regole che servono per eliminare le ambiguità.

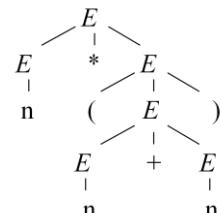
I costrutti per cui il parsing è difficile coinvolgono per lo più regole di precedenza o associatività.

Esempio di grammatica ambigua:

Espressioni aritmetiche

$$\begin{aligned} E &\rightarrow n \\ E &\rightarrow ( E ) \\ E &\rightarrow E + E \\ E &\rightarrow E - E \\ E &\rightarrow E * E \\ E &\rightarrow E / E \end{aligned}$$

Parse tree  $n * n + n$



Esistono altri alberi di derivazione per  $n * n + n$

### 6.4.1. Alcuni tipi di ambiguità

- Ricorsione sinistra e destra nella stessa regola:  
 $E \rightarrow E + E \mid i$        $i + i + i$  ha due derivazioni sinistre  
Si elimina stabilendo un ordine di derivazione, per esempio:  
 $E \rightarrow i + E \mid i$       oppure       $E \rightarrow E + i \mid i$
- Ricorsione sinistra e destra in regole diverse:  
 $A \rightarrow aA \mid Ab \mid c$        $L = a^* cb^*$   
Si stabilisce un ordine tra le derivazioni:  
 $S \rightarrow aS \mid X$        $X \rightarrow Xb \mid c$

#### 6.4.2. Eliminare l'ambiguità

**Grammatica non ambigua** per espressioni aritmetiche (operatori postfissi):

$$E \rightarrow EE + | EE - | EE * | EE / | \text{number}$$

**Grammatica non ambigua** per espressioni aritmetiche:

$$E \rightarrow E + T | E - T | T$$

$$T \rightarrow T * F | T / F | F$$

$$F \rightarrow ( E ) | \text{number}$$

Scelta: associatività a sinistra, precedenza di \* e / su + e -

#### 6.4.3. Ambiguità dell'else "pendente"

Un comune tipo di ambiguità riguarda le frasi condizionali. Si consideri la grammatica:

$$\text{stmt} \rightarrow \text{if expr then stmt} | \text{if expr then stmt else stmt} | \text{other}$$

Ad esempio: **if E<sub>1</sub> then if E<sub>2</sub> then S<sub>1</sub> else S<sub>2</sub>** ha due parse tree.

Idea: basta distinguere gli statement in matched o completi (statement che contengono sia **then** che **else** e tale che sia dopo il **then** che dopo l'**else** ci siano statement matched) e statement open (statement con condizionali semplici o tali che il primo statement sia matched e il secondo open). Solo gli statement marche possono precedere l'**else**.

Quindi la grammatica diventa:

$$\text{stmt} \rightarrow \text{matched\_stmt} | \text{open\_stmt}$$

$$\text{matched\_stmt} \rightarrow \text{if expr then matched\_stmt else matched\_stmt} | \text{other}$$

$$\text{open\_stmt} \rightarrow \text{if expr then stmt} | \text{if expr then matched\_stmt else open\_stmt}$$

#### 6.5. Costrutti non context-free

- $L = \{wcw \mid w \in (a|b)^*\}$   
Rappresenta il problema di controllare che gli identificatori siano dichiarati prima del loro uso.
- $L = \{a^n b^m c^n d^m \mid n, m > 0\}$   
Rappresenta il problema di controllare il numero dei parametri formali di due funzioni coincida con quello dei parametri attuali delle rispettive funzioni.

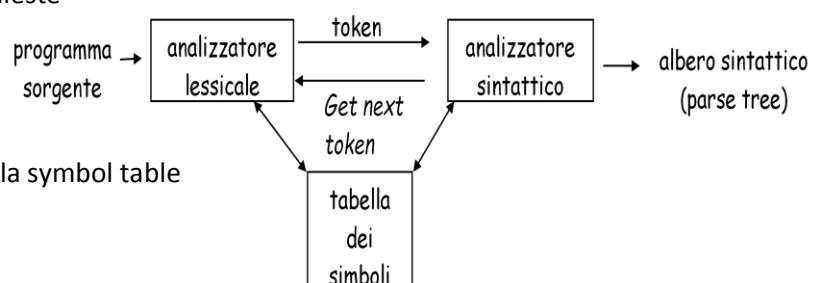
#### 6.6. Compito principale del parser

- Data una grammatica CF che genera un linguaggio L e data una frase x, rispondere alla domanda: x appartiene a L?
- Se la risposta è sì, esibire un albero di derivazione di x
- Se la risposta è no, esibire eventuali errori sintattici

#### 6.7. Ruolo del parser nel processo di compilazione

Svolge un ruolo centrale nel front-end:

- attiva l'analizzatore lessicale con richieste
- verifica la correttezza sintattica
- costruisce l'**albero di parsing**
- gestisce gli errori comuni di sintassi
- prepara e anticipa la traduzione
- colleziona informazioni sui token nella symbol table
- realizza alcuni tipi di analisi semantica
- genera il codice intermedio.



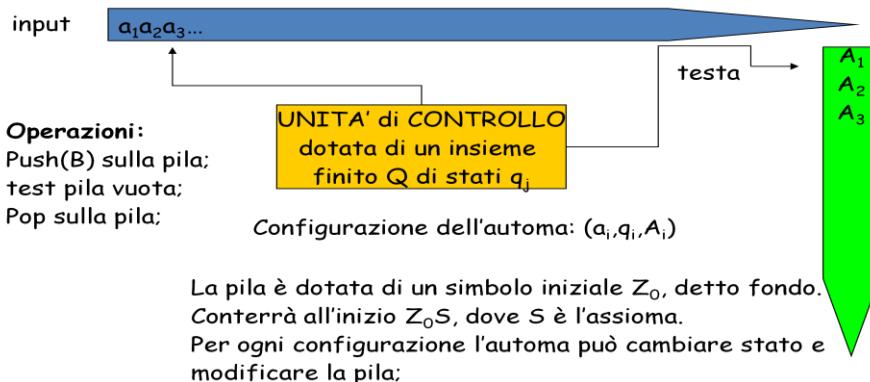
## 6.8. Perché usare le grammatiche

- Una grammatica fornisce in modo semplice e facile da capire una specifica della sintassi di un linguaggio di programmazione
- A partire da certe classi di grammatiche è possibile costruire in modo automatico parser efficienti in grado di stabilire se un certo programma è ben formato.
- Un parser può rivelare alcune ambiguità sintattiche difficili da notare in fase di progettazione del linguaggi.
- Una grammatica progettata adeguatamente impedisce una struttura ad un linguaggio di programmazione che è utile per la traduzione in codice oggetto e per la ricerca degli errori.
- Aggiungere nuovi costrutti a linguaggi descritti mediante grammatiche è più facile.

## 6.9. Grammatiche CF e automi a pila

- La classe dei linguaggi CF coincide con quella dei linguaggi accettati da automi dotati di una memoria ausiliaria a pila.
- Un primo algoritmo di riconoscimento potrebbe basarsi sulla simulazione di un automa a pila.
- Tuttavia il modello da considerare è non deterministico poiché la classe dei linguaggi riconosciuti da automi a pila deterministici è strettamente inclusa in quella dei linguaggi CF.
- Ad ogni passo, l'automa sceglie in modo non deterministico una delle regole applicabili in funzione dello stato, del simbolo corrente e del simbolo in cima alla pila.

Gli automi a pila sono automi finiti dotati di una memoria ausiliaria organizzata come una pila illimitata:



### 6.9.1. Dalla grammatica CF all'automa a pila indeterministico con un solo stato

Regola	Mossa	
$A \rightarrow B\alpha_1\alpha_2\dots\alpha_n$	if testa=A then pop; push( $\alpha_n\dots\alpha_1B$ )	(mossa spontanea)
$A \rightarrow b\alpha_1\alpha_2\dots\alpha_n$	if car=b and testa=A then pop; push( $\alpha_n\dots\alpha_1$ ); avanza testina lettura;	
$A \rightarrow \epsilon$	if testa=A then pop;	(mossa spontanea)
Per ogni carattere b	if car=b and testa=b then pop; avanza testina lettura;	
---	if car=\$ and testa= $Z_0$ then accetta; alt;	

Esempio

$$L = \{a^n b^m \mid n \geq m \geq 1\}$$

Regole

1. S->aS
2. S->A
3. A->aAb
4. A->ab
- 5.
- 6.

Mosse

- if car=a and testa=S then pop; push(S); avanza;
- if testa=S then pop; push(A);
- if car=a and testa=A then pop; push(bA); avanza;
- if car=a and testa=A then pop; push(b); avanza;
- if car=b and testa=b then pop; avanza;
- if car=\$ and testa=Z<sub>0</sub> then accetta; alt;

Esempio: aaabb

La scelta tra 1 e 2 non è deterministica;

- L'automa costruito riconosce una stringa se e solo se la grammatica la genera;
- L'automa simula le derivazioni sinistre della grammatica;
- Si dimostra che la famiglia dei linguaggi CF coincide con quella dei linguaggi riconosciuti da automi a pila indeterministici con un solo stato.

Se la grammatica è nella forma di Greibach (ogni regola inizia con un terminale e non contiene altri terminali) non ci sono mosse spontanee e la pila non impila mai simboli terminali.

Data una stringa x di lunghezza n, se la derivazione da S esiste, avrà lunghezza n.

Se K è il numero massimo di alternative per ogni non terminale A,

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_K$$

allora ad ogni passo si hanno al più K scelte

Per esplorare tutte le scelte, si ha una complessità esponenziale  $O(K^n)$

Si può fare molto meglio!

In un generico automa a pila sono presenti tre forme di indeterminismo:

- Incertezza tra più mosse di lettura: per stato q, carattere a e simbolo A della pila, d(q,a,A) ha più di un valore;
- Incertezza tra una mossa spontanea (senza lettura) e una mossa di lettura;
- Incertezza tra più mosse spontanee;

Se nessuna delle tre forme è presente, l'automa a pila è deterministico, e il linguaggio riconosciuto è detto context-free deterministico.

## 6.10. Parser deterministici e non deterministici

Un parser è l'implementazione di un automa a pila che riconosce il linguaggio generato da una certa grammatica.

Se un parser è deterministico allora ogni frase è riconosciuta con un solo calcolo (l'automa a pila è deterministico).

Un linguaggio riconosciuto da parser deterministico si dice linguaggio context-free deterministico e può essere generato da una grammatica non ambigua.

La famiglia dei linguaggi deterministici è strettamente contenuta in quella dei linguaggi context-free.

Se un linguaggio è inerentemente ambiguo allora il parser non può mai essere deterministico.

Per esempio:  $L = \{a^i b^j c^k \mid i=j \text{ oppure } j=k\}$

$$L = \{a^i b^j c^* \mid i >= 0\} \cup \{a^* b^i c^j \mid i >= 0\}.$$

Esistono linguaggi non ambigui ma il cui parsing è effettuato da parser non deterministici (due calcoli diversi ma solo uno termina con successo):

$$\text{Esempio: } L = \{a^n b^n \mid n > 0\} \cup \{a^n b^{2n} \mid n > 0\}$$

## 6.11. Tipi di parser

### ➤ Metodi Universali

- (Cocke-Younger-Kasami 1967): usa la programmazione dinamica per stabilire se una data stringa appartiene ad un dato linguaggio context-free. Richiede che la grammatica sia in forma normale di Chomsky. L'algoritmo richiede tempo  $O(n^3)$ .
- (Earley 1970): in grado di trattare qualsiasi grammatica CF. L'algoritmo richiede tempo  $O(n^3)$ .

### ➤ Metodi Lineari in $O(n)$ , su certe grammatiche riconosciute da automi a pila deterministici:

- analisi discendente (top-down), più intuitiva, ben adatta a grammatiche semplici;
- analisi ascendente (bottom-up), più sofisticata, più utilizzata dai generatori automatici di analizzatori sintattici, poiché necessita poche manipolazioni della grammatica.

## 6.12. Algoritmo di Earley

L'idea è quella di costruire progressivamente tutte le possibili derivazioni leftmost o rightmost compatibili con la stringa in input.

Durante il procedimento si analizza la stringa in input da sinistra verso destra scartando via via le derivazioni in cui non vi sia corrispondenza tra i simboli derivati e quelli della stringa.

Se esistono due derivazioni leftmost o rightmost possibili l'algoritmo restituisce i due alberi di derivazione.

La complessità di calcolo è proporzionale al cubo della lunghezza della stringa da analizzare e si riduce al quadrato se la grammatica non è ambigua e ancora di più se è deterministica.

In dettaglio:

- Data una stringa  $x_1x_2\dots x_n$ , l'algoritmo la scandisce da sinistra verso destra e per ogni  $x_i$  costruisce l'insieme  $S[i]$ , costituito da coppie (dot\_rule, puntatore).
- Una dot\_rule è una produzione di  $G$  avente sul lato destro un punto che ne marca una posizione.
- Il puntatore è un intero che indica la posizione dell'input a partire dalla quale è iniziato l'esame della produzione contenuta nella dot\_rule.

In altre parole se un elemento di  $S[j]$  è del tipo:  $(A \rightarrow a.b, i)$  con  $0 <= i <= j$

Ciò significa che:

- si è iniziato l'esame della produzione  $A \rightarrow ab$  a partire dalla posizione  $i+1$  dell'input;
- è già stata esaminata la parte che precede il punto;
- è già stato verificato che  $a$  genera  $x_{i+1}\dots x_j$

Per semplicità si aggiunge \$ a x e la produzione  $S' \rightarrow \$\$$ .

```
Input x=x1...xn
xn+1=$
S[0]={ (S' -> . $$, 0) }
for j=0 to n+1 do
    elabora ogni coppia (A->a.b, i) di S[j] applicando una
    delle 3 operazioni: scansione, completamento, predizione.
if S[n+1]={ (S' -> $$., 0) } then
    accetta
else rifiuta.
```

### 6.12.1. Scansione, completamento e predizione

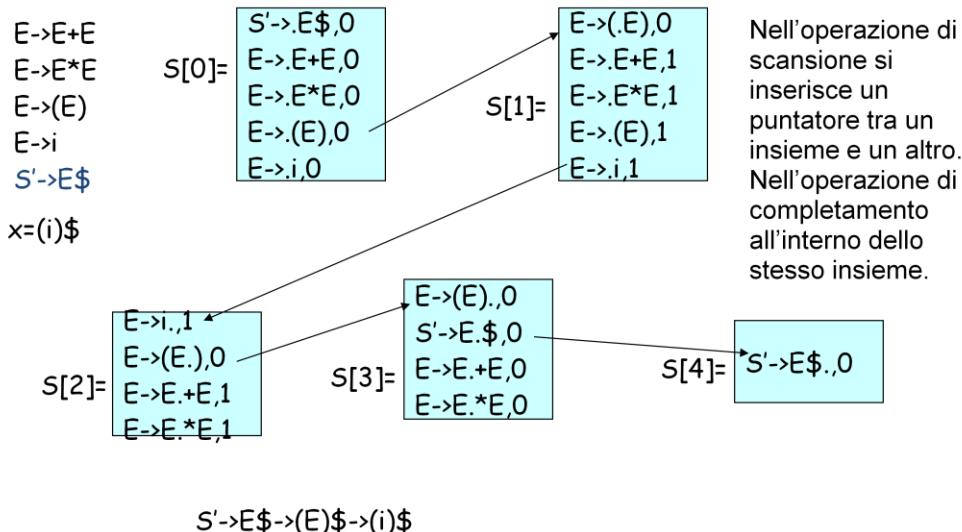
Esaminiamo uno stato  $(A \rightarrow \alpha\beta, i)$  appartenente a  $S[j]$ .

- SCANSIONE: Se  $\beta$  inizia con un terminale  $a$ , cioè  $\beta = a\beta'$ , allora se  $a = x_{j+1}$ , aggiungi la coppia  $(A \rightarrow \alpha a. \beta', i)$  a  $S[j+1]$ .
- PREDIZIONE: Se  $\beta$  inizia con un non terminale  $B$ , cioè  $\beta = B\beta'$ , allora, per ogni produzione  $B \rightarrow \gamma$ , aggiungi la coppia  $(B \rightarrow. \gamma, j)$  a  $S[j]$ . (si predice l'espansione di  $B$  a partire dalla posizione  $j$ )

- COMPLETAMENTO: Se  $\beta$  è la parola vuota allora data la coppia  $(A \rightarrow \alpha . , i)$ , per ogni coppia  $(C \rightarrow \eta . A\delta, h)$  di  $S[i]$  si aggiunge  $(C \rightarrow \eta A. \delta, h)$  a  $S[j]$ , (la predizione  $A \rightarrow \alpha$  si è verificata quindi si può allungare la parte riconosciuta. Si individuano allora in  $S[i]$  tutti gli stati che avevano fatto la predizione  $\eta \Rightarrow x_{h+1} \dots x_j$ . Siccome  $A \Rightarrow a \Rightarrow x_{i+1} \dots x_j$ , allora  $\eta A \Rightarrow x_{h+1} \dots x_j$

Proseguendo a ritroso si costruisce la derivazione e quindi l'albero di derivazione.

### 6.12.2. Esempi

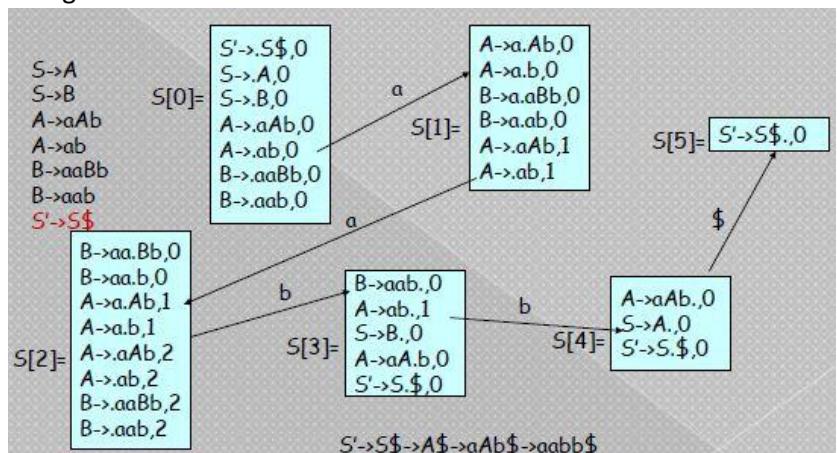


Esercizio:

Costruire l'algoritmo di Early per la grammatica:

$S \rightarrow A \mid B$   
 $A \rightarrow aAb \mid ab$   
 $B \rightarrow aaBb \mid aab$   
e la stringa aabb

Nota: Se ci sono produzioni  $A \rightarrow \epsilon$ , si effettua subito il completamento (la dot\_rule è  $A \rightarrow .$ )



Esercizio:

Siano date la grammatica  $G$  e la stringa  $aaaa$ . Si esegua il riconoscimento della stringa utilizzando l'algoritmo di Early.

$G: S \rightarrow aaS \mid Saaa \mid a \mid \epsilon$

### 6.12.3. Complessità dell'algoritmo

- Ciascun insieme  $S[j]$  può avere un numero di coppie che cresce linearmente con  $j$ , quindi  $O(n)$ ;
- Le operazioni di scansione e predizione su ogni coppia sono indipendenti da  $n$ ;
- L'operazione di completamento richiede  $O(j)$  per ogni coppia, quindi in totale  $O(n^2)$ ;
- Sommando i passi per ogni  $i$  si ha  $O(n^3)$ .

### 6.13. Gestione degli errori

Un parser deve essere in grado di *scoprire*, *diagnosticare* e correggere gli errori in maniera efficiente, per *riprendere* l'analisi e scoprire nuovi errori.

Alcuni parser (LL e LR) hanno la proprietà “viable prefix”: sono in grado di rilevare un errore non appena si presenta perché sono in grado di riconoscere i prefissi validi del linguaggio

Le strategie di riparazione sono:

- “panic mode”: scoperto l'errore il parser riprende l'analisi in corrispondenza di alcuni token sincronizzanti predefiniti (es.: delimitatori begin end) *scartando* alcuni caratteri. Svantaggi: può essere *scartato* molto input.
- “phrase level”: correzioni locali ottenute inserendo, modificando, cancellando alcuni terminali per poter riprendere l'analisi (es.: ‘,’ -> ‘;’).  
Svantaggi: difficoltà quando la distanza dall'errore è notevole.
- “error productions”: uso di produzioni che estendono la grammatica per generare gli errori più comuni. Metodo efficiente per la diagnostica.
- “global correction”: si cerca di “calcolare” la migliore correzione possibile alla derivazione errata (minimo costo di interventi per inserzioni/cancellazioni). Metodo globale poco usato in pratica, ma tecnica usata per ottimizzare la strategia “phrase level”.

## 7. Parser discendenti

A seconda che si consideri una derivazione sinistra o destra e in base all'ordine con cui tale derivazione è ricostruita si ottengono due classi di parser:

- **Discendenti**: si costruisce la derivazione sinistra partendo dall'assioma; l'albero di derivazione si costruisce dalla radice alle foglie;
  - **Ascendenti**: si costruisce la derivazione destra ma nell'ordine riflesso, cioè l'albero si costruisce dalle foglie alla radice.

## Esempio:



### 7.1. Parser deterministici top down (dalla radice alle foglie)

Studieremo due tipi di parser top-down deterministici:

- quelli **a discesa ricorsiva**, molto versatili e semplici da scrivere a mano
  - quelli **LL(1)**, semplici da gestire, usano una pila e servono a introdurre gli algoritmi più complessi di parsing bottom-up.

Il termine LL(1) ha il seguente significato:

1. la prima L, significa che l'input è analizzato da sinistra verso destra
  2. la seconda L, significa il parser costruisce una derivazione leftmost per la stringa di input
  3. il numero 1, significa che l'algoritmo utilizza soltanto un solo simbolo dell'input per risolvere le scelte del parser (ci sono varianti con k simboli)

## 7.2. Trasformazione della grammatica per l'analisi top-down

Due aspetti rendono una grammatica inadatta all'analisi top-down: la ricorsione sinistra e la presenza di prefissi comuni in più parti destre di regole associate allo stesso simbolo non terminale.

- $A \rightarrow y\alpha^1 | \dots | y\alpha^n$  *Rimedio: fattorizzazione sinistra.*
  - $A \rightarrow A\alpha$  ( $A$  non terminale). *Rimedio: eliminazione ricorsione sinistra.*

Può essere possibile adattare una grammatica in modo che sia applicabile un parser predittivo.

### 7.3. Fattorizzazione sinistra

Idea: quando non è chiaro quale produzione usare per espandere un non terminale  $A$ , si possono riscrivere le produzioni in modo da “posticipare” la scelta, introducendo un non terminale supplementare.

Eempio:  $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma$   
*fattorizzazione sinistra:*

$A \rightarrow \alpha A' \mid \gamma$      $A'$  è un nuovo simbolo non terminale

$A' \rightarrow \beta_1 \mid \beta_2$

`<stmt> -> if <expr> then <stmt> | if <expr> then <stmt> else <stmt>`

*fattorizzazione sinistra:*

**<stmt>** -> **if** **<expr>** **then** **<stmt>** **<S>**

$\langle S \rangle \rightarrow \varepsilon \mid \text{else } \langle \text{stmt} \rangle$

## 7.4. Eliminazione ricorsione sinistra

Metodo: date le produzioni “ricorsive sinistre” e non, di un non terminale  $A$ :

$$A \rightarrow A\alpha_1 | \dots | A\alpha_n | \beta_1 | \dots | \beta_n$$

si sostituiscono con (eliminazione ricorsione sinistra immediata)

$$A \rightarrow \beta_1 A' | \dots | \beta_n A'$$

$$A' \rightarrow \alpha_1 A' | \dots | \alpha_n A' | \epsilon$$

Esempi espressioni aritmetiche

$$E \rightarrow E+T \mid T$$

$$E \rightarrow TE'$$

$$T \rightarrow T*F \mid F$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid \text{id}$$

NOTA: Non è detto che le ricorsioni sinistre siano solo immediate

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \epsilon$$

### 7.4.1. Algoritmo per eliminare le ricorsioni sinistre

- INPUT: Grammatica senza cicli né  $\epsilon$ -produzioni
- OUTPUT: Una grammatica equivalente senza ricorsioni sinistre (Tale grammatica potrebbe contenere  $\epsilon$ -produzioni)

Algoritmo:

1. Ordina i simboli non terminali  $A_1, A_2, \dots, A_n$ ;
2. for ( $i=1$  to  $n$ ) {
  - for ( $j=1$  to  $i-1$ ) {
    - sostituire ogni produzione  $A_i \rightarrow A_j \gamma$  con le produzioni  $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$  dove  $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$  sono tutte le  $A_j$  produzioni
    - eliminare le ricorsioni sinistre immediate

Nell'esempio:  $S \rightarrow Aa \mid b$       Si ottiene:  $S \rightarrow Aa \mid b$   
 $A \rightarrow Ac \mid Sd \mid \epsilon$        $A \rightarrow bdA' \mid A'$   
 $A' \rightarrow cA' \mid adA' \mid \epsilon$

## 7.5. Parser a discesa ricorsiva

L'idea: le regole grammaticali per un non-terminale  $A$  sono viste come costituenti una procedura che riconosce un  $A$

- la parte destra di ogni regola specifica la struttura del codice per questa procedura
- la sequenza di terminali e non terminali nelle varie regole corrisponde a un controllo che i terminali siano presenti nell'input e a invocazioni delle procedure dei simboli non terminali
- la presenza di diverse regole per  $A$  è modellata da **case o if**
- può richiedere backtracking (può richiedere di leggere più di una volta parte della stringa in ingresso).

NOTA: Prima di procedere è necessario assicurarsi che non ci siano ricorsioni sinistre e fattori sinistri.

ESEMPIO:  $F \rightarrow (E) \mid \text{id}$   
 $E \rightarrow TE'$   
 $E' \rightarrow +TE' \mid \epsilon$   
 $T \rightarrow FT'$   
 $T' \rightarrow *FT' \mid \epsilon$

### 7.5.1. Procedura tipica per un parser top-down a discesa ricorsiva

```
void A() {
    scegli, per A, una produzione A->X1X2...Xk
    for (da i a k) {
        if (Xi è un non terminale)
            richiama Xi();
        else if (Xi è uguale al simbolo in input corrente)
            procedi al simbolo successivo;
        else si è verificato un errore;
    }
}
```

### 7.5.2. Costruzione delle procedure ricorsive

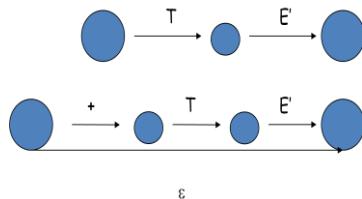
$$F \rightarrow (E) | id$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' | e$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' | e$$



void E();

{

T();

E'();

}

void E'();

{ If (tok==+) then  
avanza input;  
T();  
E'();  
}

}

void S () { /\* funzione per S \*/

if (tok == IF) then {

avanza(IF); E(); avanza(THEN); S(); avanza(ELSE); S();

} else if (tok == BEGIN) {

avanza(BEGIN); S(); L();

} else if (tok == PRINT) {

avanza(PRINT); E();

} else error();

}

void L () { /\* funzione per L \*/

if (tok == END) then

avanza(END);

else if (tok == PUNTVIRG) {

avanza(PUNTVIRG); S(); L();

} else error();

}

void E () { /\* funzione per E \*/

avanza(NUM); avanza(EQ); avanza(NUM);

}

**Esempio:**

S -> if E then S else S

S -> begin S L

L -> end

L -> ; S L

S -> print E

E -> num = num

token { IF, THEN, ELSE, BEGIN,  
PRINT, PUNTVIRG, NUM, EQ };

### 7.6. Parser LL(1)

Nella tecnica “a discesa ricorsiva” l’analisi sintattica viene effettuata attraverso una cascata di chiamate ricorsive.

I parser discendenti deterministici possono anche essere guidati da una tabella. In tal caso si parla di parser predittivi, cioè parser a discesa ricorsiva senza backtracking.

Nel parser **LL(1)**, la pila delle chiamate ricorsive viene esplicitata nel parser, e quindi **non si fa più uso di ricorsione**.

**Esempio:** il linguaggio delle parentesi bilanciate

$$S \rightarrow ( S ) S$$

$$S \rightarrow \epsilon$$

e vediamo come opera il parser **LL(1)** per riconoscere la stringa “( )”

Il parser consiste di una **pila**, che contiene inizialmente il simbolo “\$” (fondo della pila), ed un **input**, la cui fine è marcata dal simbolo “\$” (EOF generato dallo scanner)

<b>pila</b>	<b>input</b>	<b>azione</b>
\$	( )\$	

- il parsing inizia inserendo il simbolo iniziale in testa alla pila
 

pila	input	azione
\$	)\$	
- il parser **accetta** una stringa di input se, dopo una sequenza di azioni, la pila contiene “\$” e la stringa di input è “\$”
 

pila	input	azione
...	...	
\$	\$	accept
- ogni volta che in testa alla pila c’è un simbolo non terminale X, lo si espande secondo una produzione  $X \rightarrow \gamma$ , che viene scelta a seconda del simbolo in testa all’input e ai valori di una tabella (la parte destra della produzione viene invertita sulla pila)
 

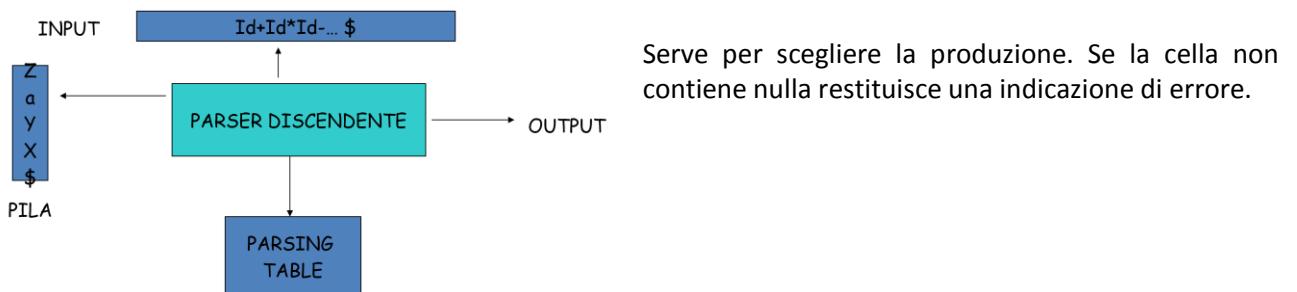
pila	input	azione
\$ \$	)\$	$S \rightarrow ( S ) S$
- ogni volta che sulla pila c’è un simbolo terminale **t**, si controlla che in testa all’input ci sia anche lo stesso simbolo, nel qual caso lo si elimina sia dalla pila che dall’input; altrimenti è **errore**

Per costruire un parser **LL(1)**, bisogna costruire una tabella – la **tabella LL(1)** – che determina la regola da usare per l’espansione, dati il simbolo non-terminale e il carattere in input

pila	input	azione
\$ \$	)\$	$S \rightarrow ( S ) S$
\$ S ) S (	)\$	match
\$ S ) S	)\$	$S \rightarrow \epsilon$
\$ S )	)\$	match
\$ S	\$	$S \rightarrow \epsilon$
\$	\$	accept

Se l’input è generato dalla grammatica questo parsing fornisce una derivazione leftmost, altrimenti produce un’indicazione d’errore.

### 7.6.1. I Parser LL(1) sono parser discendenti non ricorsivi



### 7.6.2. Costruzione della parsing table di un LL(1)

1. la tabella ha simboli non-terminali come righe, e simboli terminali come colonne
2. per ogni regola  $X \rightarrow \gamma$  di G, si inserisce tale regola nella casella  $(X, t)$ , per ogni  $t$  tale che  $\gamma \Rightarrow^* t \beta$
3. per ogni regola  $X \rightarrow \gamma$  di G, per cui  $\gamma \Rightarrow^* \epsilon$ , si inserisce nella casella  $(X, t)$  tale regola, per ogni  $t$  tale che  $S \Rightarrow^* \beta X t \alpha$

Bisogna conoscere questi simboli **t** in grado di far effettuare la scelta della produzione.

Le regole 2 e 3 sono difficili da implementare: gli algoritmi per risolverle sono discussi in seguito vedi **FIRST** e **FOLLOW**.

per esempio: la tabella per la grammatica

$$S \rightarrow (S)S, \quad S \rightarrow \epsilon$$

	(	)	\$
S	$S \rightarrow (S)S$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

## 7.7. First

E' una funzione che consente di costruire le entries della tabella, quando possibile.

**FIRST( $\gamma$ ):** insieme dei simboli terminali che si trovano all'inizio delle stringhe derivate da  $\gamma$  ( $\gamma$  è una stringa di simboli terminali e non).

L'algoritmo:

1. se  $X$  è un terminale, allora  $\text{FIRST}(X) = \{ X \}$ ,
2. se  $X \rightarrow \epsilon$  appartiene alla grammatica, allora aggiungi  $\epsilon$  a  $\text{FIRST}(X)$ ,
3. se  $X \rightarrow Y_1Y_2\dots Y_k$  appartiene alla grammatica, allora:
  - se  $a \in \text{FIRST}(Y_i)$  per qualche  $i$  ed  $a$  sta in  $\text{FIRST}(Y_1), \text{FIRST}(Y_2), \dots, \text{FIRST}(Y_{i-1})$ , aggiungi  $a$  in  $\text{FIRST}(X)$ ;
  - se tutti gli insiemi  $\text{FIRST}(Y_1), \text{FIRST}(Y_2), \dots, \text{FIRST}(Y_k)$ , contengono  $\epsilon$ , aggiungi  $\epsilon$  a  $\text{FIRST}(X)$ ;
4. per definire l'insieme  $\text{FIRST}(\gamma)$ , dove  $\gamma = X_1X_2\dots X_k$  (una stringa di terminali e non), si procede reiterando le regole seguenti:
  - aggiungi  $\text{FIRST}(X_1) \setminus \{ \epsilon \}$  a  $\text{FIRST}(\gamma)$
  - se per qualche  $i < k$ , tutti gli insiemi  $\text{FIRST}(X_1), \dots, \text{FIRST}(X_i)$  contengono  $\epsilon$ , allora aggiungi  $\text{FIRST}(X_{i+1}) \setminus \{ \epsilon \}$  a  $\text{FIRST}(\gamma)$
  - se tutti gli insiemi  $\text{FIRST}(X_1), \dots, \text{FIRST}(X_k)$  contengono  $\epsilon$ , allora aggiungi  $\epsilon$  a  $\text{FIRST}(\gamma)$

Se  $X \rightarrow^* \epsilon$  allora  $\epsilon \in \text{FIRST}(X)$  ( $X$  è detto annullabile).

### 7.7.1. Esempio di calcolo di First

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$$\begin{array}{ll} E \rightarrow E+T \mid T & E \rightarrow TE' \\ T \rightarrow T^*F \mid F & E' \rightarrow +TE' \mid \epsilon \\ F \rightarrow (E) \mid \text{id} & T \rightarrow FT' \\ & T' \rightarrow *FT' \mid \epsilon \\ & F \rightarrow (E) \mid \text{id} \end{array}$$

$$\begin{array}{llll} \text{FIRST}(\text{id})=\{\text{id}\} & \text{FIRST}(\text{ })=\{\text{ }\} & \text{FIRST}(+) = \{+\} & \text{FIRST}(*)=\{* \} \\ \text{FIRST}(\text{ })=\{\text{ }\} & \text{FIRST}(F)=\{\text{id}, \text{ }\} & \text{FIRST}(T)=\text{FIRST}(E)=\text{FIRST}(F) & \\ \text{FIRST}(E')=\{+, \epsilon\} & \text{FIRST}(T')=\{*, \epsilon\} & & \end{array}$$

### 7.7.2. Algoritmo in pseudo-C per calcolare FIRST

```

for all terminals X and  $\epsilon$  do  $\text{FIRST}(X) = \{X\}$  ;
for all non-terminals X do  $\text{FIRST}(X) = \{ \}$  ;
while (there are changes to any  $\text{FIRST}(X)$ ) do
  for each production  $X \rightarrow Y_1 Y_2 \dots Y_k$  do
    {
      i := 1 ; continue := true ;
      while (continue == true && i <= k) do
        if  $Y_i \in \text{FIRST}(X)$  then
          for all terminals  $a$  such that  $a \in \text{FIRST}(Y_i)$  do
            add  $a$  to  $\text{FIRST}(X)$ 
        else if  $Y_i \rightarrow^* \epsilon$  then
          add  $\epsilon$  to  $\text{FIRST}(X)$ 
        end if
        i := i + 1
      end while
    end if
  end for
end while

```

```

{
    add FIRST(Yi) \ { ε } to FIRST(X) ;
    if (ε is not in FIRST(Yi)) continue := false ;
    i := i+1 ;
}
if (continue == true) add ε to FIRST(X) ;
}

```

### 7.7.3. FIRST può non bastare

**Osservazione:** se la grammatica contiene due produzioni

$X \rightarrow \gamma_1$

$X \rightarrow \gamma_2$

(stesso simbolo non-terminale a sinistra, due sequenze differenti a destra)

e  $\text{FIRST}(\gamma_1) \cap \text{FIRST}(\gamma_2)$  è non vuota, allora la grammatica non può essere analizzata col parsing top-down:

se  $t \in \text{FIRST}(\gamma_1) \cap \text{FIRST}(\gamma_2)$

allora, il parsing discendente non saprà cosa fare quando il primo simbolo dell'input è  $t$

## 7.8. Follow

Dato un non terminale  $X$ , l'insieme **FOLLOW**( $X$ ) è l'insieme dei simboli terminali, eventualmente  $\$$ , che appaiono alla destra di  $X$  in qualche forma sentenziale ed è definito come segue:

1. se  $X$  è l'assioma, allora aggiungi  $\$$  a **FOLLOW**( $X$ )
2. se c'è una produzione  $A \rightarrow \alpha X \gamma$ , allora aggiungi  $\text{FIRST}(\gamma) \setminus \{\epsilon\}$  a **FOLLOW**( $X$ )
3. se c'è una produzione  $A \rightarrow \alpha X \gamma$  per cui  $\epsilon \in \text{FIRST}(\gamma)$ , allora aggiungi **FOLLOW**( $A$ ) a **FOLLOW**( $X$ )

**Osservazioni:**

1. quando l'assioma non compare a destra delle produzioni, il “ $\$$ ” è l'unico simbolo nel suo insieme **FOLLOW**
2. l'insieme **FOLLOW** non contiene mai “ $\epsilon$ ”
3. **FOLLOW** è definito soltanto per non-terminali: potremmo generalizzare la definizione ma ciò è inutile per la tabella **LL(1)**
4. la definizione di **FOLLOW** lavora “alla destra” di una produzione, mentre quella di **FIRST** lavora “alla sinistra”: una produzione  $X \rightarrow \alpha$  non ha alcuna informazione su **FOLLOW**( $X$ ), se  $X$  non è presente in  $\alpha$

### 7.8.1. Algoritmo in pseudo-C per calcolare FOLLOW

```

FOLLOW(Axiom) = { $ } ;
for all (nonterminal(X) && X != Axiom) do FOLLOW(X) = { } ;
while (there are changes to any FOLLOW set) do
    for each production X -> Y1 Y2... Yk do
        for each nonterminal(Yi) do {
            add FIRST(Yi+1... Yk) \ { ε } to FOLLOW(Yi) ;
        /* Note: if i=k then Yi+1... Yk = ε */
        if ε is in FIRST(Yi+1... Yk)
            add FOLLOW(X) to FOLLOW(Yi);
        }
    }
}

```

### 7.8.2. Esempio di calcolo di Follow

Nel caso della grammatica delle espressioni aritmetiche senza ricorsioni sinistre:

$$\begin{array}{ll} E \rightarrow E+T \mid T & E \rightarrow TE' \\ T \rightarrow T^*F \mid F & E' \rightarrow +TE' \mid \epsilon \\ F \rightarrow (E) \mid \text{id} & T \rightarrow FT' \\ & T' \rightarrow *FT' \mid \epsilon \\ & F \rightarrow (E) \mid \text{id} \end{array}$$

$$\begin{array}{llll} \text{FIRST(id)} = \{\text{id}\} & \text{FIRST('')}' = \{''\} & \text{FIRST('')} = \{+\} & \text{FIRST(*)} = \{*\} \\ \text{FIRST(')} = \{'\}' & \text{FIRST(F)} = \{\text{id}, ''\} & & \\ \text{FIRST(T)} = \text{FIRST(E)} = \text{FIRST(F)} & & & \\ \text{FIRST(E')} = \{+, \epsilon\} & \text{FIRST(T')} = \{*, \epsilon\} & & \end{array}$$

$$\begin{array}{ll} \text{FOLLOW}(E) = \{\$\}, \} & \text{FOLLOW}(E') = \{\$\}, \} \\ \text{FOLLOW}(T) = \{\$\}, +\} & \text{FOLLOW}(T') = \{\$\}, +\} \\ \text{FOLLOW}(F) = \{\$\}, +, *\} & \end{array}$$

### 7.9. Grammatica LL(1)

Una grammatica è LL(1) se e solo se per ogni produzione del tipo  $A \rightarrow \alpha \mid \beta$  si ha:

- $\alpha$  e  $\beta$  non derivano stringhe che cominciano con lo stesso simbolo  $\alpha$ .
- Al più uno tra i due può derivare la stringa vuota.
- Se  $\beta \rightarrow^* \epsilon$  allora  $\alpha$  non deriva stringhe che cominciano con terminali che stanno in  $\text{FOLLOW}(A)$ . Analogamente per  $\alpha$

Equivalentemente affinchè una grammatica sia LL(1) deve avvenire che per ogni coppia di produzioni  $A \rightarrow \alpha \mid \beta$

1.  $\text{FIRST}(\alpha)$  e  $\text{FIRST}(\beta)$  devono essere disgiunti
2. Se  $\epsilon$  è in  $\text{FIRST}(\beta)$  allora  $\text{FIRST}(\alpha)$  e  $\text{FOLLOW}(A)$  devono essere disgiunti.

### 7.10. Come costruire la tabella

1. Per ogni regola  $X \rightarrow \alpha$  di  $G$ , si inserisce nella casella  $(X, t)$  la regola  $X \rightarrow \alpha$ , per ogni  $t$  tale che  $t \in \text{FIRST}(\alpha)$
2. Per ogni regola  $X \rightarrow \alpha$  di  $G$ , per cui  $\alpha \Rightarrow^* \epsilon$  ( $\epsilon \in \text{FIRST}(\alpha)$ ), si inserisce nella casella  $(X, t)$  la regola  $X \rightarrow \alpha$ , per ogni  $t$  tale che  $t \in \text{FOLLOW}(X)$ . Se  $\epsilon \in \text{FIRST}(\alpha)$  and  $\$ \in \text{FOLLOW}(X)$ , si inserisce la regola  $X \rightarrow \alpha$  in  $(X, \$)$ .
3. Le caselle non definite definiscono un errore.

NOTA: Se  $G$  è ricorsiva sinistra o ambigua, la tabella avrà caselle con valori multipli.

### 7.11. Altra definizione di Grammatica LL(1)

Una grammatica la cui tabella **LL(1)** non contiene più di un elemento nelle caselle è detta **LL(1)** (**leftmost derivation with 1 symbol lookahead**)

**osservazione:** per costruzione una grammatica **LL(1)** non è ambigua, né ricorsiva sinistra

## 7.12. Algoritmo di parsing LL(1)

```

LL1_parser( stack p, input i, LL1_table M, initial S )
{
    /* p, i sono pile, S è il simbolo iniziale */
    int error = 0 ;
    p = push(p, $) ;
    p = push(p, S) ;
    while ( top(p) != $ && top(i) != $ && !error)
    {
        if isterminal(top(p))
        {
            if top(p) == top(i)
            { p = pop(p) ; i = pop(i) ; }
            else
                error = 1 ;
        }
        else /* top(p) è un non-terminal, bisogna espandere */
        {
            if isempty(M[top(p),top(i)])
                error = 1;
            else
                { /* M[top(p),top(i)] == X à X1 ... Xn; */
                    p = pop(p) ;
                    for (j = n ; j > 0 ; j = j-1)
                        p = push(p, Xj) ;
                }
        }
        if (!error)
            accept() ;
        else
            raise_error() ;
    }
}

```

### 7.12.1. Complessità di calcolo

E' lineare nella lunghezza n della stringa sorgente. Viene consumato un carattere per volta oppure viene effettuata una mossa spontanea. Il numero di mosse spontanee è limitato dal numero di simboli non terminali della grammatica (poiché non ci sono ricorsioni sinistre).

### 7.12.2. Esempio

costruire il parser **LL(1)** per la grammatica

$$S \rightarrow ( S ) \quad S \rightarrow [ S ] \quad S \rightarrow \{ S \}$$

$$S \rightarrow \epsilon$$

insiemi **FIRST, FOLLOW:**

	<b>FIRST</b>	<b>FOLLOW</b>
S	(, [, {, ε	), ], }, \$

la tabella **LL(1)**:

	(	[	{	)	]	}	\$
S	$S \rightarrow (S)$	$S \rightarrow [S]$	$S \rightarrow \{S\}$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

### 7.13. Esempi di grammatiche LL(1) e non

- G:  $S \rightarrow aSb \mid \epsilon$

FIRST(S)={a,  $\epsilon$ }, FIRST(a)={a}, FIRST(b)={b}  
FOLLOW(S)={b, \$}



	a	b	\$
S	$S \rightarrow aSb$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$

- G:  $S \rightarrow +SS \mid *SS \mid id$

Per semplicità consideriamo solo i simboli non terminali

FIRST(S)={+, \*, id}  
FOLLOW(S)={\$, +, \*, id}



	+	*	id	\$
S	$S \rightarrow +SS$	$S \rightarrow *SS$	$S \rightarrow id$	

- G:  $S \rightarrow aSb \mid aSc \mid \epsilon$

Metodo della fattorizzazione sinistra:

G':  $S \rightarrow aSA \mid \epsilon$   
A  $\rightarrow b \mid c$

Per semplicità consideriamo solo i simboli non terminali

FIRST(S)={a,  $\epsilon$ }, FIRST(A)={b, c}  
FOLLOW(S)={\$, b, c} = FOLLOW(A)

	a	b	c	\$
S	$S \rightarrow aSA$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$	$S \rightarrow \epsilon$
A		A $\rightarrow b$	A $\rightarrow c$	

- G:  $S \rightarrow aSa \mid bSb \mid a \mid b$

Metodo della fattorizzazione sinistra:

G':  $S \rightarrow aA \mid bB$   
A  $\rightarrow Sa \mid \epsilon$   
B  $\rightarrow Sb \mid \epsilon$

Per semplicità consideriamo solo i simboli non terminali

FIRST(S)={a, b}, FIRST(A)={a, b,  $\epsilon$ } = FOLLOW(B)  
FOLLOW(S)={a, b, \$} = FOLLOW(A) = FOLLOW(B)

	a	b	\$
S	$S \rightarrow aA$	$S \rightarrow bB$	
A	A $\rightarrow Sa$ A $\rightarrow \epsilon$	A $\rightarrow Sa$ A $\rightarrow \epsilon$	A $\rightarrow \epsilon$
B	B $\rightarrow Sb$ B $\rightarrow \epsilon$	B $\rightarrow Sb$ B $\rightarrow \epsilon$	B $\rightarrow \epsilon$

Non è un linguaggio LL(1)

#### 7.14. Come ottenere grammatiche LL(1)

- Verificare, se è possibile, che non sia ambigua. In caso contrario, se si può si rimuova l'ambiguità;
- Controllare che non presenti ricorsioni sinistre. In caso contrario trasformarle in ricorsioni destre.
- Se un simbolo non terminale ammette più derivazioni con lo stesso prefisso applicare il metodo della fattorizzazione sinistra.
- In alternativa, può essere necessario allungare la lunghezza della prospezioni. (Equivale a considerare parser LL(k), k>1)

#### 7.15. Limiti della famiglia LL(k)

Non tutti i linguaggi verificabili da parser deterministici sono generabili da grammatiche LL(k).

Per esempio:

$L=\{a^*a^n b^n \mid n \geq 0\}$  Linguaggio deterministico ma non LL(k)

È generato dalla grammatica

$S \rightarrow A \mid aS$

$A \rightarrow aAb \mid \epsilon$

Altro esempio:

$S \rightarrow R \mid (S)$

$R \rightarrow E=E$

$E \rightarrow a \mid (E+E)$

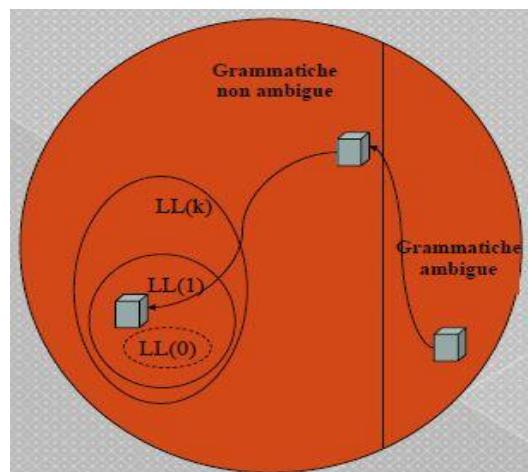
Definisce relazioni di uguaglianza tra espressioni aritmetiche additive.

Una stringa che inizia con "(" può essere una relazione R parentesizzata oppure un'espressione.

Il linguaggio non è LL(k) ma è deterministico.

#### 7.16. Relazione tra grammatiche LL(k)

E' possibile generalizzare la nozione di **FIRST** nel parsing predittivo in modo da restituire i primi  $k$  token in input, e costruire una tabella predittiva in cui le righe sono i simboli non-terminali e *le colonne sono sequenze di  $k$  terminali*. Le grammatiche non ambigue corrispondenti sono le **LL( $K$ )**



## 8. Parsing ascendenti

### 8.1. Parsing Bottom-up

Gli algoritmi di parsing bottom-up o ascendente analizzano una stringa di input cercando di ricostruire i passi di una derivazione rightmost.

Sono detti bottom-up perchè costruiscono l'albero sintattico relativo ad una stringa prodotta dalla grammatica dalle foglie alla radice.

Un modello generale di parsing bottom-up è il parsing shift-reduce, chiamato comunemente SR-parsing.

La classe più grande di grammatiche per cui è possibile usare un SR-parsing è data dalle grammatiche LR. Costruire un parser LR a mano è molto difficile ma tale metodo è utile nel caso di generazione automatica di parser.

- Obiettivo: Costruire un parse tree di una stringa in input partendo dalle foglie fino ad arrivare alla radice. Questo processo può essere pensato come una riduzione di una stringa all'assioma della grammatica.
- Metodo (intuitivo): Ad ogni passo di riduzione una particolare sottostringa che ha un match con la parte destra di una qualche regola di produzione viene sostituita con la parte sinistra di quella produzione. Se la sottostringa è scelta in modo corretto, allora viene così prodotta una derivazione rightmost in ordine inverso.

#### 8.1.1. Esempio di parsing bottom up

Si consideri la grammatica

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

La sequenza abcd è può essere ridotta ad S nel modo seguente:

$$ab\mathbf{cd} \rightarrow a\mathbf{Abcd} \rightarrow a\mathbf{Ad} \rightarrow a\mathbf{A} \rightarrow S$$

Handle: è una sottostringa di una forma sentenziale destra che coincide con la parte destra di una produzione e la cui riduzione rappresenta un passo della inversa della derivazione rightmost.

Un handle può essere seguito solo da simboli terminali.

### 8.2. Parser shift-reduce (SR)

Il parser usa una **pila**, che contiene inizialmente il simbolo “\$” (fondo della pila), ed un **input**, la cui fine è marcata dal simbolo “\$” (è l'EOF generato dallo scanner).

ESEMPIO:

$$\begin{aligned} S &\rightarrow aABe \\ A &\rightarrow Abc \mid b \\ B &\rightarrow d \end{aligned}$$

La frase abcd\$ viene valutata come segue:

- nello stato iniziale la pila è vuota

pila	input	azione
\$	ab\mathbf{cd}\$	

- il parser esegue azioni, che oltre ad “accept” sono di tre ulteriori tipi:

- **shift**: un simbolo terminale è spostato dalla stringa di input sulla pila (si indica scrivendo la parola “shift”)

- **reduce**: una stringa  $\alpha$  sulla pila è ridotta al non-terminale A, secondo la regola  $A \rightarrow \alpha$  (si indica scrivendo “**reduce**  $A \rightarrow \alpha$ ”)
- **error**: errore di sintassi

	<b>pila</b>	<b>input</b>	<b>azione</b>
1	\$	abbcde\$	<b>shift</b>
2	\$a	bbcde\$	<b>shift</b>
3	\$ab	bcde\$	<b>reduce</b> $A \rightarrow b$
4	\$aA	bcde\$	<b>shift</b>
5	\$aAb	cde\$	<b>shift</b>
6	\$aAbc	de\$	<b>reduce</b> $A \rightarrow Abc$
7	\$aA	de\$	<b>shift</b>
8	\$aAd	e\$	<b>reduce</b> $B \rightarrow d$
9	\$aAB	e\$	<b>shift</b>
10	\$aABe	\$	<b>reduce</b> $S \rightarrow aABe$
11	\$S	\$	<b>accept</b>

**Prefissi Ammissibili:** Insieme dei prefissi delle forme sentenziali destre che possono apparire sullo stack di uno SR parser. Ovvero, sono i prefissi delle forme sentenziali destre di derivazioni rightmost, che non contengono sottostringhe interne che sono handle (tali sottostringhe possono apparire solo come suffisso).

Come riconoscere un handle sullo stack?

Come scegliere la produzione o l'azione opportuna?

### 8.2.1. Conflitti durante il Parsing SR

Esistono grammatiche CF per le quali il parsing SR non può essere usato.

In questi casi ogni SR-parser può raggiungere una configurazione in cui sfruttando l'intero stack e il simbolo da leggere non si riesce a decidere se eseguire uno shift o un reduce, o non si sa quale riduzione applicare.

Esempio: una grammatica ambigua non può essere LR

```
stmt -> if expr then stmt
      | if expr then stmt else stmt
      | other
```

<b>pila</b>	<b>input</b>	
...if expr then stmt	else ...\$	Shift o Reduce?

### 8.3. Tipico SR parsing: LR(k)

Il termine LR(k) ha il seguente significato:

1. la L significa che l'input è analizzato da sinistra verso destra
2. la R significa che il parser produce una derivazione rightmost per la stringa di input
3. il numero k significa che l'algoritmo utilizza k simboli dell'input per decidere l'azione del parser.

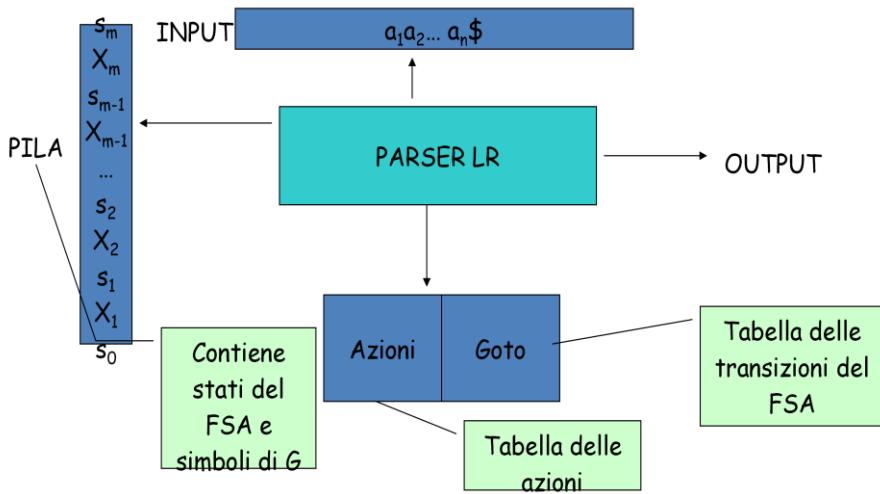
### 8.3.1. Perché usare i parser LR?

- Possono essere costruiti per riconoscere virtualmente tutti i costrutti di linguaggi di programmazione definiti da grammatiche CF.
- E' il più generale parsing SR che non richiede backtracking.
- Riconosce velocemente gli errori sintattici

- La classe delle grammatiche LR contiene propriamente le grammatiche LL. Infatti un parser  $LR(k)$  deve riconoscere l'occorrenza di una parte destra di una produzione in una forma sentenziale destra con  $k$  simboli di prospezione. Un parser  $LL(k)$  deve scegliere una produzione in base ai primi  $k$  simboli che riesce a derivare.
- Scrivere un parser LR a mano è difficile, ma esistono generatori automatici.

### 8.3.2. Schema di un parser LR

Si costruisce il FSA che riconosce l'insieme dei prefissi ammissibili. Sia  $s_0, s_1, s_2, \dots, s_p$  il suo insieme degli stati (ciascuno di essi rappresenta lo stato della pila).



### 8.3.3. Configurazione e funzionamento di un parser LR

- Una configurazione di un LR parser è :  $(s_0 \dots X_{m-1} X_m s_m, a_i \dots a_n \$)$  che rappresenta la forma sentenziale destra  $X_1 \dots X_{m-1} X_m a_i \dots a_n \$$
- L'azione del parser è determinata dallo stato che si trova in cima alla pila ed eventualmente dal simbolo (o un gruppo di simboli) dell'input.
- Lo stato successivo dipende dalla tabella goto.

### 8.4. Parser LR(0)

- I parser **LR(0)** analizzano la stringa di input considerando solamente il simbolo in testa alla pila:
- La classe delle grammatiche corrispondenti non è interessante,
- la tecnica sì

Il comportamento del parser **LR(0)** si basa sulla tabella **LR(0)** definita come segue

**Caselle vuote rappresentano errori**

		Stati	Azioni	Goto
		1 2 3 4 ...	Shift Reduce A->b Reduce B->a Shift Accept ...	a b ( ) c 3 2 4 ...
				A B S S'
				4 2

#### 8.4.1. Automa LR(0)

1° passo: Si dota la grammatica della produzione  $S' \rightarrow S$

Operazione CLOSURE:

se  $I$  è un insieme di item, allora  $\text{CLOSURE}(I)$  si costruisce come segue:

1.  $I \subseteq \text{CLOSURE}(I)$ ;

2. Se  $A \rightarrow a.Bb$  è in  $\text{CLOSURE}(I)$  e  $B \rightarrow \gamma$  è una produzione, allora si aggiunge  $B \rightarrow \gamma$  in  $\text{CLOSURE}(I)$ . Si applica questa regola fino a quando non aggiungo altri item.

```
Function Closure(I);
begin
  J:=I;
  repeat
    for each item  $A \rightarrow a.Bb$  in  $J$  and each  $B \rightarrow \gamma$  such that  $B \rightarrow \gamma$  is not in  $J$  do
      add  $B \rightarrow \gamma$  to  $J$ ;
    until no more items can be added to  $J$ ;
end
```

ESEMPIO: data la seguente grammatica, il primo stato è  $\text{CLOSURE}(S' \rightarrow .S)$

$S' \rightarrow S$   
 $S \rightarrow ( L )$   
 $S \rightarrow x$   
 $L \rightarrow S$   
 $L \rightarrow L, S$

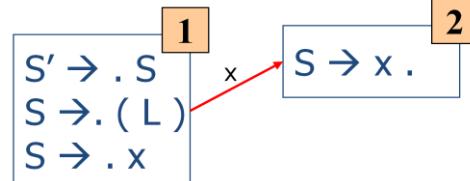
**1**  
 $S' \rightarrow .S$   
 $S \rightarrow .( L )$   
 $S \rightarrow .x$

Operazione GOTO( $I, X$ ): se  $I$  è un insieme di item e  $X$  un simbolo di  $G$ , allora  $\text{GOTO}(I, X)$  si definisce come la chiusura dell'insieme di tutti gli item  $A \rightarrow \alpha X \beta$  tale che  $A \rightarrow \alpha X \beta$  è un item di  $I$ .

Intuitivamente se  $I$  è l'insieme degli item validi per un prefisso riducibile  $\gamma$ , allora  $\text{GOTO}(I, X)$  è l'insieme degli item validi per  $\gamma X$ .

```
Function GOTO(I, X);
begin
  J:=emptyset;
  for each item  $A \rightarrow a.Xb$  in  $I$  do add  $A \rightarrow aX.b$  to  $J$ ;
  Return CLOSURE(J);
end
```

Consente di costruire le transizioni dell'automa.



#### 8.4.2. Costruzione della tabella LR(0)

Un LR(0) item di una grammatica  $G$  è una produzione di  $G$  insieme con un punto in una posizione della parte destra.

Ad esempio, data la produzione  $A \rightarrow XYZ$ , gli item sono:

$A \rightarrow .XYZ$   
 $A \rightarrow X.YZ$   
 $A \rightarrow XY.Z$   
 $A \rightarrow XY.Z$

La produzione  $A \rightarrow \epsilon$  genera l'item  $A \rightarrow .$

Un item indica quanto di una produzione è stato visto ad una certa fase del processo di parsing.

Gruppi di item costituiscono gli stati dell'automa che riconosce i prefissi ammissibili.

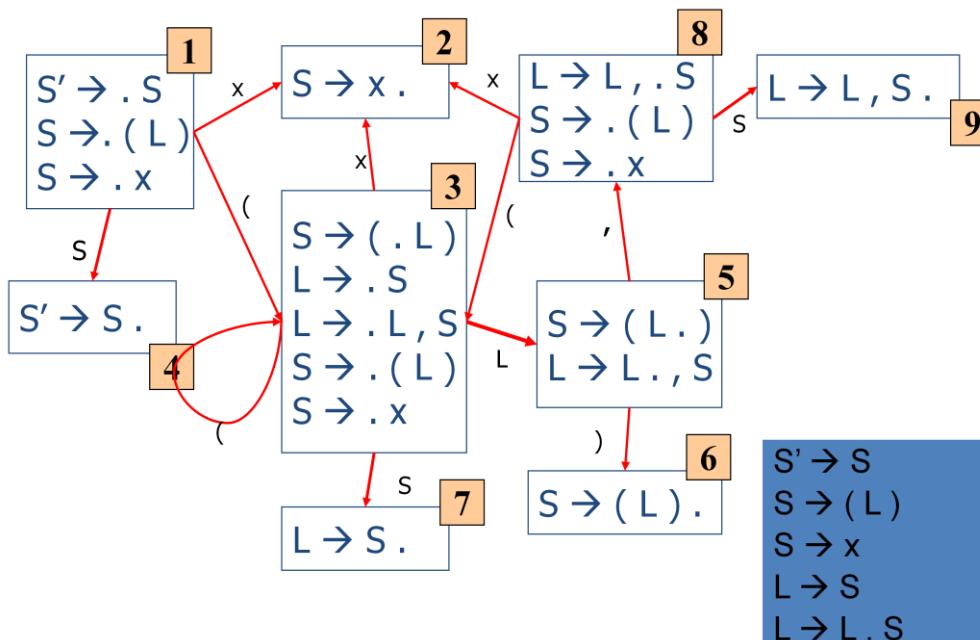
Tale costruzione è alla base di tutti i parser LR.

La tabella ha come righe il numero degli stati e come colonne i simboli della grammatica (prima i terminali e poi i non-terminali).

- **azioni shift:** poichè dallo stato 1 esiste una transizione “(“ (un simbolo terminale) nello stato 3, ciò significa eseguire una operazione di “shift 3” in corrispondenza di ( 1 , “(“ ).
- **azioni goto:** poichè dallo stato 1 esiste una transizione “S” nello stato 4 (un simbolo non-terminale), ciò significa eseguire una operazione di “goto 4” in corrispondenza di ( 1 , “S” )

Continuando in questo modo si completa la tabella, per le azioni di shift e goto

- **azioni reduce:** ogni stato che contiene un elemento LR(0) del tipo  $X \rightarrow \gamma$ . deve avere una operazione di riduzione con tale regola per ogni simbolo terminale.
- **azione acc:** l'operazione di reduce  $S' \rightarrow S$ . quando il simbolo terminale è “\$” equivale ad accettazione e si sostituisce nella tabella con “acc” in corrispondenza di \$.



Costruzione della tabella

- (0)  $S' \rightarrow S$
- (1)  $S \rightarrow ( L )$
- (2)  $S \rightarrow x$
- (3)  $L \rightarrow S$
- (4)  $L \rightarrow L, S$

	(	)	x	,	\$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2				
4					acc		
5			s6		s8		
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2				
9	r4	r4	r4	r4	r4	g9	
action						goto	

#### 8.4.3. Algoritmo di Parsing LR(0)

Sia  $\text{TAB}_{\text{LR}0}$  la tabella  $\text{LR}(0)$  e sia  $u$  lo stato corrente (*quello in testa alla pila*), le azioni sono:

- se  $\text{TAB}_{\text{LR}0}[u, t] = \text{reduce } k$  ( $t$  è un qualunque terminale), dove  $k$  rappresenta la produzione  $A \rightarrow \alpha$ . allora si rimuove la stringa  $\alpha$  dalla pila, assieme a tutti gli stati corrispondenti (fino allo stato immediatamente prima di  $a$ ) e si inserisce il non-terminale  $A$  ( $2^* | \alpha |$  operazioni di pop) siano  $u'A$  gli elementi in testa alla pila, e sia  $\text{TAB}_{\text{LR}0}[u', A] = \text{goto } u''$ , allora si inserisce sulla pila lo stato  $u''$
- altrimenti, se  $\text{TAB}_{\text{LR}0}[u, t] = \text{shift } u'$  si sposta sulla pila il token " $t$ " in testa all'input e si inserisce nella pila lo stato  $u'$
- altrimenti, se  $\text{TAB}_{\text{LR}0}[u, t] = \text{accept}$  allora termina
- altrimenti si rileva un **errore**

Esempio: il riconoscimento di  $(x,(x)) \$$  per la grammatica precedente.

<i>passo</i>	<i>automa</i>	<i>input</i>	<i>azione</i>
1	1	$(x,(x)) \$$	s3
2	1 ( 3	$x,(x)) \$$	s2
3	1 ( 3 x 2	$,(x)) \$$	r2 (S à x) + g7
4	1 ( 3 S 7	$,(x)) \$$	r3 (L à S) + g5
5	1 ( 3 L 5	$,(x)) \$$	s8
6	1 ( 3 L 5 , 8	$(x)) \$$	s3
7	1 ( 3 L 5 , 8 ( 3	$x)) \$$	s2
8	1 ( 3 L 5 , 8 ( 3 x 2	$)) \$$	r2 (S à x) + g7
9	1 ( 3 L 5 , 8 ( 3 S 7	$)) \$$	r3 (L à S) + g5
10	1 ( 3 L 5 , 8 ( 3 L 5	$)) \$$	s6
11	1 ( 3 L 5 , 8 ( 3 L 5 ) 6	$) \$$	r1 (S à (L) ) + g9
12	1 ( 3 L 5 , 8 S 9	$) \$$	r4 (L à L,S) + g5
13	1 ( 3 L 5	$) \$$	s6
14	1 ( 3 L 5 ) 6	$\$$	r1 (S à (L) ) + g4
15	1 S 4	$\$$	accept

#### 8.4.4. Grammatiche LR(0)

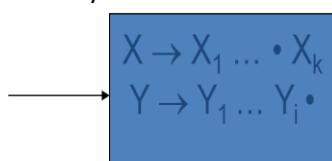
- E' una grammatica in cui ogni cella della tabella LR(0) contiene al più un solo valore.
- Equivalentemente, gli stati dell'automa sono o contenenti solo produzioni che presuppongono shift o solo produzioni che presuppongono reduce.
- Gli stati che presuppongono operazioni di reduce, inoltre, contengono una sola produzione.

Proprietà dell'automa LR(0):

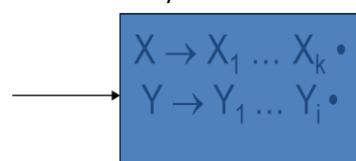
1. tutte le frecce entranti in uno stato hanno la stessa etichetta;
2. Uno stato di reduce non ha successori;
3. Uno stato di shift ha almeno un successore.

#### 8.5. Conflitti shift-reduce o reduce-reduce

- shift/reduce

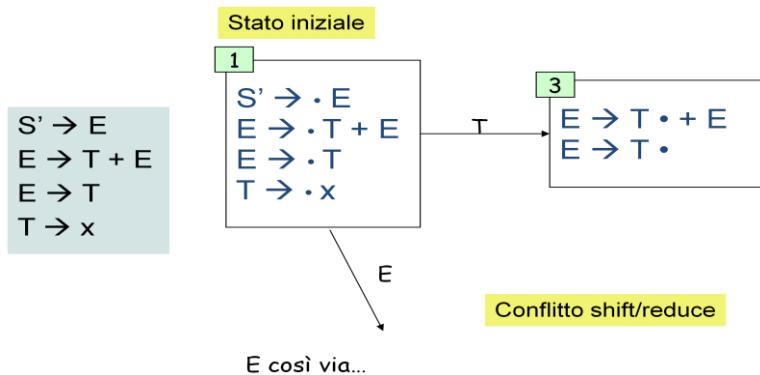


- reduce/reduce



In questi casi si tratta di una grammatica non LR(0).

## 8.6. Esempio di grammatica non LR(0)



## 8.7. Tabella ambigua

- (0)  $S' \rightarrow E$
- (1)  $E \rightarrow T + E$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow x$

Dallo stato 3 leggendo "+" posso o shiftare su 4 o ridurre con  $E \rightarrow T$ .

Ovvero, il modello diventa non deterministico

Abbiamo bisogno di strumenti più potenti

	x	+	\$	E	T
1	s5			g2	g3
2			acc		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

## 8.8. LL(k) vs. LR(k)

Left to Right parse

Leftmost derivation

k-token look ahead

→ LL(k)

- Predice quale produzione usare dopo aver visto  $k$  tokens dalla parte destra derivata.
- Usati sia nei compilatori scritti a mano (discendenti ricorsivi) sia costruiti con strumenti automatici.
- Sono considerati veloci ma potrebbero avere bisogno del backtracking.
- Recentemente sono stati ripresi.
  - ANTLR e javacc for Java
- Necessitano di modificare la grammatica.

Left to Right parse

Rightmost derivation

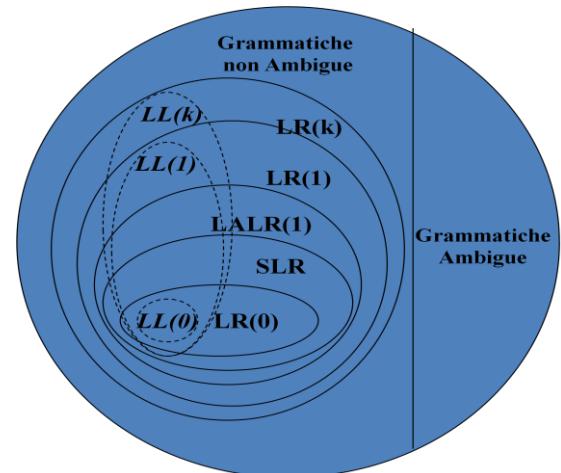
k-token look ahead

→ LR(k)

- Riesce a riconoscere le occorrenze del lato destro di una produzione avendo visto i primi  $k$  simboli di ciò che deriva da tale lato destro
- Usati tipicamente in modo automatico.
- I più usati per il parsing reale
  - YACC, BISON, CUP for Java, sablecc
- Necessitano solo di aggiungere la produzione  $S' \rightarrow S$ .

## 8.9. LR Parsing

- Le grammatiche LR sono più potenti delle LL.
- LR(0) ha esclusivo interesse didattico.
- Simple LR (SLR) consentono il parsing di famiglie di linguaggi interessanti.
- La maggior parte dei linguaggi di programmazione ammettono una grammatica LALR(1).
  - Molti generatori di parser usano questa classe.
- LR(1) fornisce un parsing molto potente.
  - L'implementazione è poco controllabile.
  - Si cercano grammatiche LALR(1) equivalenti.



## 8.10. Esempi

La grammatica  
 $S \rightarrow A \mid aS$   
 $A \rightarrow aAb \mid \epsilon$   
 è LR(0)? E' LL(1)?

- Non è LL(1)
- ( Il linguaggio è deterministico ma non LL(k) )
- Non è LR(0)
- ( La presenza di regole vuote viola la condizione LR(0) )

La grammatica  
 $S \rightarrow a \mid ab$   
 è LR(0)? E' LL(1)?

- È LL(1) modificando la grammatica (ha un fattore sx)
- Non è LR(0) (conflitto shift/reduce nel secondo stato)

In un linguaggio LR(0), se una stringa appartiene al linguaggio, nessun prefisso di esso può appartenervi

Esercizi:

La grammatica  
 $S \rightarrow aSb \mid \epsilon$   
 è LR(0)? E' LL(1)?

Non è LR(0) (abbiamo  $\epsilon$ )  
 È LL(1)

La grammatica  
 $S \rightarrow SA \mid A$   
 $A \rightarrow aAb \mid ab$   
 è LR(0)? E' LL(1)?

Che conclusioni trarre sulle relazioni tra grammatiche LR(0) e LL(1)?

## 8.11. Confronto tra grammatiche e linguaggi LR(0) e LL(1)

Le classi di grammatiche LR(0) e LL(1) non sono incluse una nell'altra

- Una grammatica con regole vuote non è LR(0) ma può essere LL(1)
- Una grammatica con ricorsioni sinistre non è LL(1) ma può essere LR(0)

Le famiglie dei linguaggi LR(0) e LL(1) sono distinte e incomparabili.

- Un linguaggio chiuso per prefissi non è LR(0) ma può essere LL(1)
- Esistono linguaggi LR(0) ma non LL(1)

## 8.12. Parser LR(0), SLR, LR(1), LALR(1): cosa hanno in comune?

Usano azioni di "shift" e "reduce";

Sono macchine guidate da una tabella:

- Sono raffinamenti di LR(0)
  - Calcolano un FSA usando la costruzione basata sugli item
  - SLR: usa gli stessi "item" di LR(0).
    - Usa anche le informazioni dell'insieme FOLLOW
  - LR(1)/LALR(1): un item contiene anche informazioni date dai simboli lookahead.
    - LALR(1) è una semplificazione di LR(1) per ridurre il numero degli stati
- Consentono di definire classi di grammatiche
  - se il parser LR(0) (o SLR, LR(1), LALR(1)) calcolato dalla grammatica non ha conflitti shift/reduce o reduce/reduce, allora G è per definizione una grammatica LR(0) (o SLR, LR(1), LALR(1)).

### 8.13. Simple LR parser (SLR o SLR(1))

E' un modo semplice di costruire parser più potenti di **LR(0)** utilizzando il prossimo token di input per decidere su alcune azioni e costruire la tabella.

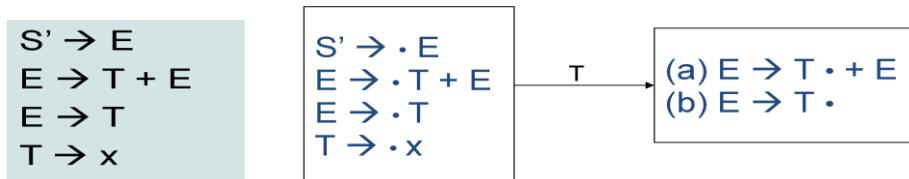
Sia s lo stato corrente:

1. se lo stato s contiene un item della forma  $A \rightarrow \alpha.x\beta$  e x è l'etichetta di una transizione uscente, allora  $TAB_{SRS}[s,x]=\text{shift } u$ , dove u è lo stato che contiene  $A \rightarrow \alpha.x.\beta$ .
2. se lo stato s contiene  $A \rightarrow \gamma$ . allora  $TAB_{SRS}[s,t]=\text{reduce } A \rightarrow \gamma$  per tutti i token t contenuti in  $\text{ FOLLOW}(A)$ . Ciò vale nel caso in cui A sia diverso da  $S'$ .
3. se lo stato s contiene  $S' \rightarrow S$ . allora  $TAB_{SRS}[s,\$]=\text{accept}$ .

Le grammatiche per cui le tabella di analisi prodotte dai parser **SLR(1)** non contengono ambiguità sono dette **grammatiche SLR(1)**.

Molte grammatiche dei linguaggi di programmazione sono **SLR(1)**

$$\text{Follow}(E) = \{ \$ \}$$



Viene eliminata l'ambiguità dalla tabella poichè:

- reduce (b) sul token "\$"
- shift (a) sul token "+"

#### 8.13.1. Tabella SLR

- (0)  $S' \rightarrow E$
- (1)  $E \rightarrow T + E$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow x$

La riduzione avviene solo se il token successivo è un simbolo valido nella riduzione.

Una grammatica è SLR se e solo se per ogni stato s:

1. Per ogni item  $A \rightarrow \alpha.x\beta$  con x terminale, non c'è l'item  $B \rightarrow \gamma$  in s con x in  $\text{Follow}(B)$ .
2. Per ogni coppia di item  $A \rightarrow \alpha.$  e  $B \rightarrow \beta.$   $\text{Follow}(A)$  e  $\text{Follow}(B)$  sono disgiunti

	x	+	\$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

#### 8.13.2. Esercizi

1. Costruire la tabella SLR per la grammatica

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E+n \mid n \end{aligned}$$

2. Costruire la tabella SLR per la grammatica

$$\begin{aligned} S' &\rightarrow S \\ S &\rightarrow (S)S \mid e \end{aligned}$$

3. Costruire la tabella SLR per la grammatica

$$\begin{array}{lll} E \rightarrow T & E \rightarrow E+T & T \rightarrow (E) \\ T \rightarrow k & & \end{array}$$

4. Costruire la tabella SLR per la grammatica

$$D \rightarrow tL; \quad L \rightarrow i \quad L \rightarrow L,i$$

che schematizza la dichiarazione di una serie di identificatori. E' LR(0)? Genera un linguaggio regolare?

### 8.13.3. Esempio di grammatica SLR

- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + T \mid T$
- (2)  $T \rightarrow T^* F \mid F$
- (3)  $F \rightarrow (E) \mid id$

	id	+	*	(	)	\$	E	T	F
0	s5			s4			g1	g2	g3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			g8	g2	g3
5		r6	r6		r6	r6			
6	s5			s4			g9	g3	
7	s5			s4					g10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

### 8.13.4. Algoritmo di parsing SLR(1)

Sia  $\text{TAB}_{\text{SLR}}$  la tabella **SLR(1)** e sia  $u$  lo stato corrente (quello in testa alla pila), sia “ $t$ ” il prossimo token di input, sia “ $u'$ ” lo stato sulla pila, le azioni sono:

- **azione di reduce:** se  $\text{TAB}_{\text{SLR}}[u, t] = \text{reduce } k$ , dove  $k$  rappresenta la produzione  $A \rightarrow \alpha$ . Allora si rimuove la stringa  $\alpha$  dalla pila, assieme a tutti gli stati corrispondenti (fino allo stato immediatamente prima di  $\alpha$ ) e si inserisce il non-terminale  $A$ . Siano  $u'A$  gli elementi in testa alla pila, e sia  $\text{TAB}_{\text{SLR}}[u', A] = \text{goto } u''$ , si inserisce sulla pila lo stato  $u''$
- **azione di shift:** se  $\text{TAB}_{\text{SLR}}[u, t] = \text{shift } u'$  allora si sposta sulla pila il token “ $t$ ” in testa all’input e si inserisce sulla pila lo stato  $u'$
- altrimenti si rileva un **errore**

### 8.14. Esistono grammatiche non SLR

Ogni grammatica SLR è non ambigua, ma esistono molte grammatiche non ambigue che non sono SLR.

Per esempio:

$S \rightarrow L = R$

$S \rightarrow R$

$L \rightarrow *R$

$L \rightarrow id$

$R \rightarrow L$

Essa ammette un conflitto shift/reduce;

Molti SR parser risolvono automaticamente i conflitti shift/reduce privilegiando lo shift al reduce (cioè incorpora la regola dell’annidamento più vicino nel problema dell’else pendente). Per esempio una versione semplificata della grammatica è:

$S \rightarrow L \mid \text{other}$

$L \rightarrow \text{if } S \mid \text{if } S \text{ else } S$  (ambiguità in corrispondenza del token else)

if a if s1 else s2

(1) if a { if s1 else s2}      (2) if a { if s1 } else s2

Privilegiare lo shift, dà l’interpretazione 1, privilegiare il reduce dà la 2

### 8.15. Esempio di conflitto reduce/reduce

La seguente grammatica modella statement che possono rappresentare chiamate a procedure senza parametri o assegnazione di espressioni a variabili

stmt->call-stmt | assign-stmt

call-stmt -> **identifier**

assign-stmt -> var := esp

var -> identifier

esp -> var | **number**



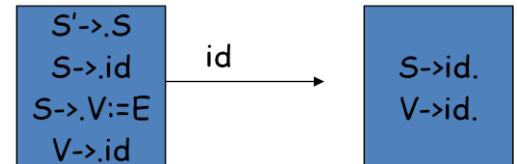
S -> id | V := E

V -> id

E -> V | n

Follow(S)={\$} Follow(V)={:=, \$}

In realtà una variabile si può trovare alla fine di un input, ma solo dopo aver trovato il token :=



### 8.16. Parser LR(1)

Consente di ovviare a molte ambiguità dei parser **SLR(1)** al prezzo di una crescita sostanziale della complessità dell'algoritmo ma poco usato in pratica poiché poco efficiente:

- si preferisce il più semplice **LALR(1)** poichè è efficiente come **SLR(1)**
- è simile agli automi **LR(0)**: cambiano gli **item** e le operazioni **closure**, **goto** e **reduce**
- **Fu il primo ad essere introdotto [Knuth 1965]**

NOTA:

1. il **problema** del parser **SLR(1)** è che utilizzava il lookahead **dopo** la costruzione del DFA

2. nei parser **LR(1)** il lookahead è utilizzato **durante** la costruzione

dell'automa (Quindi si prendono in considerazione i simboli che veramente possano seguire un certo handle)

#### 8.16.1. Costruzione della tabella LR(1)

**Gli item LR(1)** hanno la forma A->  $\alpha$ .  $\beta$  , t

dove A ->  $\alpha$ .  $\beta$  è un item **LR(0)** e “t” è un token (il lookahead) oppure t=\$ (Ciò indica il fatto che la sequenza a si trova in cima alla pila e che alla testa dell’input c’è la stringa derivabile da  $\beta$ t).

```

Function Closure (I) ;
begin
  J:=I;
  repeat
    for each item [A->  $\alpha$ . $\beta$ , z] in J
      for each X-> $\gamma$ 
        for each w $\in$ FIRST( $\beta$ z)
          add [X->. $\gamma$ , w] to J;
    until no more items can be added to J;
    return J;
end
  
```

Esempio:

S'-> S

S-> CC

C-> cC|d

Closure ([S'->S, \$])=

{[S'->S, \$], [S->.CC,\$]}

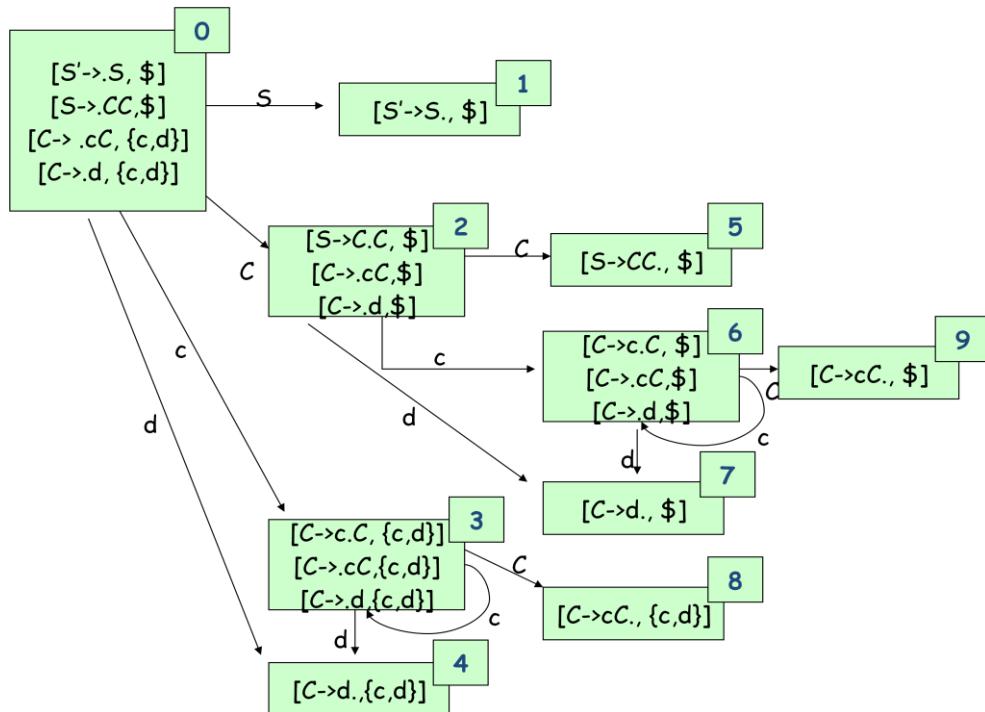
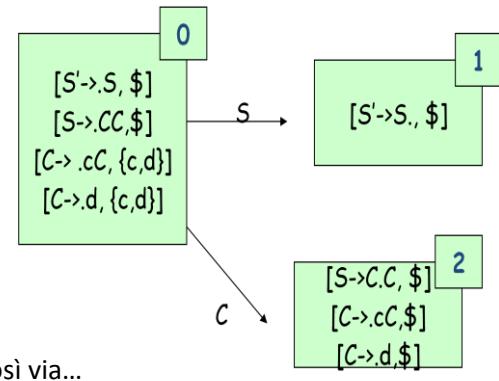
[C-> .cC, {c,d}],

[C->.d, {c,d}] }

```

Function Goto(I, X);
begin
    J := insieme degli item [A-> αX.β, α]
    tali che [A-> α.Xβ, α] è in I;
    return CLOSURE(J);
end

```



La tabella è strutturalmente simile a quella LR(0)

- azioni shift:** se dallo stato  $s$  esiste una transizione “ $t$ ” (“ $t$ ” simbolo terminale) nello stato  $s'$ , inserire “shift  $s'$ ” in corrispondenza di ( $s$ , “ $t$ ”)
- azioni goto:** se dallo stato  $s$  esiste una transizione “ $S$ ” (“ $S$ ” simbolo non-terminale) in  $s'$ , inserire “goto  $s'$ ” in corrispondenza di ( $s$ , “ $S$ ”)
- azioni reduce:** se lo stato  $s$  contiene un item LR(1) del tipo  $[X \rightarrow \gamma. . t]$ , con  $t$  simbolo terminale e  $X$  diverso da  $S'$  e “ $k$ ” identifica la produzione  $X \rightarrow g$ , allora inserire “reduce  $k$ ” in corrispondenza di ( $s$ , “ $t$ ”)
- azione accept:** se lo stato  $s$  contiene l’item  $[S' \rightarrow S., \$]$ , allora inserire “accept” in corrispondenza di ( $s$ , “ $\$$ ”)

**osservazione:** il parser LR(1) ridurrà soltanto quando il simbolo in testa all’input sarà “ $t$ ”

Lo stato iniziale è quello costruito da  $[S' \rightarrow S, \$]$

### 8.16.2. Esercizio

Creare la tabella LR(1) per la grammatica

$S \rightarrow L = R$   
 $S \rightarrow R$   
 $L \rightarrow *R$   
 $L \rightarrow id$   
 $R \rightarrow L$

## 8.17. Parsing LALR(1)

Le tabelle di analisi **LR(1)** sono di solito di **ordini di grandezza maggiori** di quelle **SLR(1)**:

se una tabella **SLR(1)** di un linguaggio di programmazione è intorno ai 10 KB, una tabella **LR(1)** dello stesso linguaggio è intorno ai MB, con il doppio della memoria per costruirla.

**osservazione:** se consideriamo l'automa **LR(1)** della grammatica  $S' \rightarrow S$ ,  $S \rightarrow CC$ ,  $C \rightarrow cC|d$  e ignoriamo i lookahead, alcune coppie di stati sono identici:

gli stati 8 e 9,

gli stati 4 e 7,

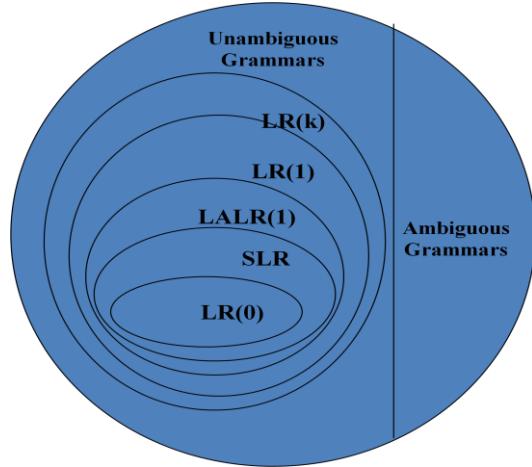
gli stati 3 e 6,

Il parser **LALR(1)** consiste nell'identificare questi stati, combinando i loro lookahead, con l'obiettivo di ottenere un DFA **LR(1)** simile al DFA **LR(0)**.

**Infatti:**

1. Le prime componenti degli item LR(1) sono item LR(0)
2. Se due stati  $s$  e  $t$  LR(1) hanno la stessa prima componente e se da  $s$  esce una transizione con  $X$  verso lo stato  $s'$ , allora anche da  $t$  uscirà una transizione con  $X$  verso  $t'$  e  $t$  e  $t'$  avranno le stesse prime componenti

- Un parsing LALR(1) potrebbe generare conflitti che il LR(1) corrispondente non genererebbe (ciò non accade in pratica).
- Si dimostra che se una grammatica è LR(1), la tabella LALR(1) non può avere conflitti shift/reduce ma solo reduce/reduce.
- È possibile computare il DFA del LALR(1) direttamente dal DFA del LR(0) attraverso un processo chiamato **lookahead propaganti**



### 8.17.1. Condizioni LALR(1)

Nota che una grammatica soddisfa la condizione LALR(1) se valgono entrambe le condizioni:

- Ogni candidata di riduzione ha un insieme di prospezione disgiunto dalle etichette terminali uscenti;
- Se vi sono due candidate di riduzione i loro insiemi di prospezione sono disgiunti;

### 8.17.2. Esempio di LR(1) ma non LALR(1)

```

S->A | Ba |bAa |bB
A->a
B->a
  
```

## 8.18. Regole per risolvere l'ambiguità

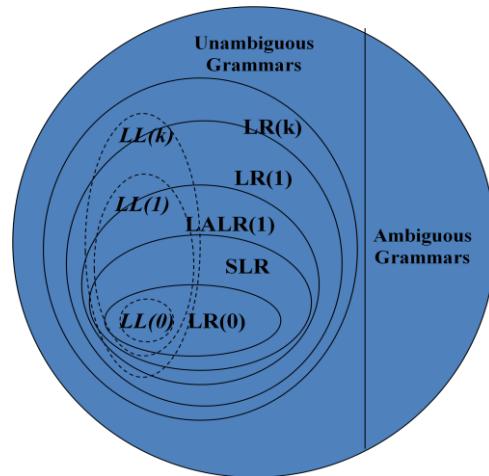
- Spesso può essere comodo usare grammatiche ambigue ed usare delle regole per risolvere l'ambiguità.
- Precedenza ed associatività  $E \rightarrow E+E | E^*E | (E) | id$  (è ambigua poiché non specifica la precedenza e l'associatività tra gli operatori)
- La grammatica non ambigua equivalente è:  $E \rightarrow E+T$   $T \rightarrow T^*F$
- La prima è preferibile perché:
  - si possono cambiare precedenza e associatività senza cambiare le produzioni
  - il parser con meno produzioni è più veloce.
- Stabilire delle regole di precedenza e di associatività fa risolvere al parser i conflitti.
- Un altro esempio riguarda l'ambiguità del "dangling-else".

### 8.19. Proprietà dei linguaggi e delle grammatiche LR(k) **IMPORTANTE**

- La famiglia dei linguaggi verificabili da parser deterministici coincide con quella dei linguaggi generati dalle grammatiche LR(1).  
Ciò non significa che ogni grammatica il cui linguaggio è deterministico, sia necessariamente LR(1); potrebbe essere ambigua o richiedere una prospezione di lunghezza  $k>1$ ; esisterà una grammatica equivalente LR(1);
- La famiglia dei linguaggi generati dalle grammatiche LR(k) coincide con quella dei linguaggi generati da LR(1). Quindi un linguaggio context free ma indeterministico non può avere una grammatica LR(k).
- Per ogni  $k>1$ , esistono grammatiche LR(k) ma non LR(k-1).
- Data una grammatica, è indecidibile se esista un  $k>0$  per cui tale grammatica risulti LR(k); di conseguenza non è decidibile se il linguaggio generato da una grammatica CF è deterministico. E' decidibile soltanto se  $k$  è fissato.

### 8.20. Confronto tra le grammatiche LL(k) e LR(k):

- Ogni linguaggio regolare è LL(1);
- Per ogni  $k>0$ , una grammatica LL(k) è anche LR(k);
- Le grammatiche LL(1) e LR(0) non sono incluse una nell'altra;
- Quasi tutte le grammatiche LL(1) sono LALR(1)



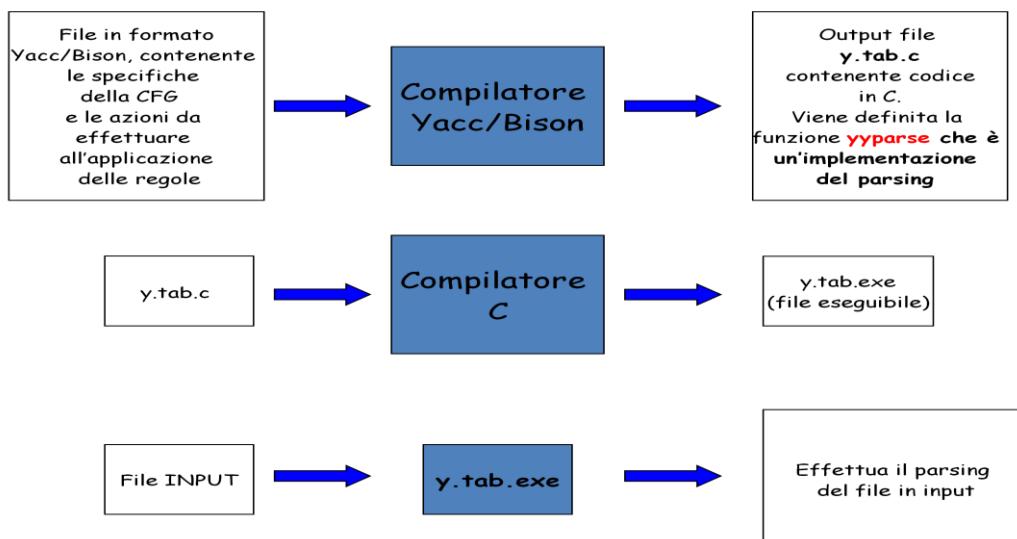
## 9. Generatori di Parser (BISON)

- Un generatore automatico di parser prende in input un file che specifica la sintassi di un certo linguaggio:
  - solitamente nella forma delle regole di una grammatica CF
  - ...e includendo altre funzioni ausiliarie, definizioni di token, ...
- ...e produce in output un codice (scritto in un certo linguaggio) che implementa il ruolo del parser
- Erano chiamati compilatori di compilatori poiché tradizionalmente tutti gli step della compilazione venivano realizzate dal parser. Adesso il parser è considerato solo uno step del processo.
  - Yacc (Yet another compiler-compiler) è un generatore largamente usato che incorpora la tecnica del parsing LALR(1)
  - Una delle implementazioni più usate di Yacc, nonché di pubblico dominio è BISON
- Il linguaggio deve essere descritto mediante una grammatica context-free
- Bison è ottimizzato per grammatiche LALR(1)
- Con dichiarazioni opportune, Bison è anche capace di fare il parsing per una classe più estesa di grammatiche.

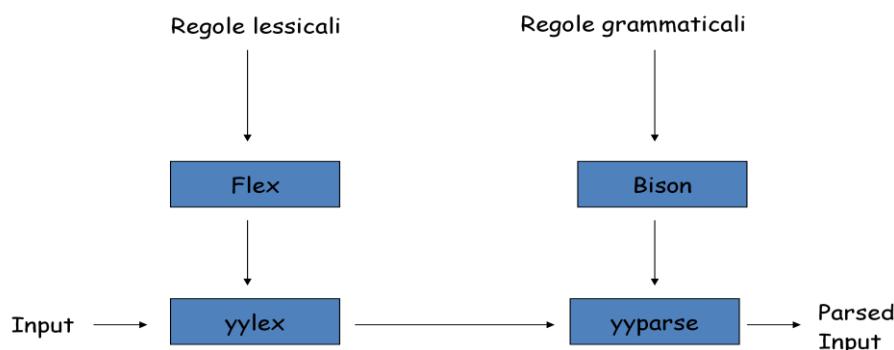
### 9.1. Grammatiche GLR

Se si usa l'opzione %glr-parser Bison produrrà un parser LR generalizzato che alla presenza di un conflitto shift-reduce o reduce-reduce, clonerà se stesso per seguire entrambe le possibilità

### 9.2. Uso di Yacc o Bison



### 9.3. Uso di Flex e Bison



## 9.4. Programmare in Bison

Un file in formato Bison consiste di 3 sezioni, separate da %%:

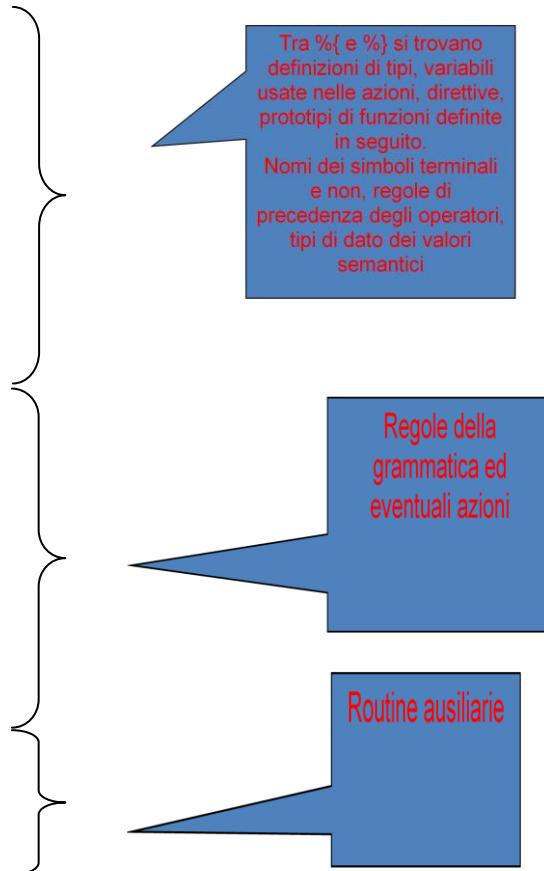
```
DEFINIZIONI
%%
REGOLE della GRAMMATICA
%%
FUNZIONI AUSILIARIE
```

La routine yyparse ha bisogno di altre funzioni:

1. l'analizzatore lessicale (yylex) che può essere scritto a mano o prodotto da Flex
2. La funzione che riporta gli errori (yyerror)
3. La funzione main che deve contenere la chiamata yyparse

## 9.5. Specifiche Bison

```
%{
#include <stdio.h>
int regs[26];
int base;
%
%start espr
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS
%%
espr: /*empty */
      |
      espr istr DIGIT
      |
      espr error LETTER
      {
          funzione();
      }
      ;
%
main() { ... }
```



## 9.6. Azioni e valore semantico

Ogni token potrebbe portare con se il corrispondente lessema (eventualmente memorizzato nella variabile yylval di tipo YYSTYPE)

Il proposito delle azioni è spesso quello di calcolare il valore semantico di un certo costrutto  
expr : expr '+' expr { \$\$ = \$1 + \$3; }

Sequenza di comandi per generare l'analizzatore sintattico:

- \* flex espr.fl
- \* bison -d espr.y
- \* gcc lex.yy.c espr.tab.c -o espr  
(il parametro -d crea l'header file (bison -d espr.y) )

## 9.7. Esempio

Espressioni aritmetiche: scrivere un parser bottom-up (ed un analizzatore lessicale) per le seguenti grammatiche per espressioni aritmetiche (rispettivamente grammatica non ambigua ed ambigua):

E -> E + T	E-> E + E
E -> E - T	E-> E-E
E -> T	E-> E * E
T -> T * F	E-> E/E
T -> T / F	E-> numero
T -> F	E-> (E)
F -> numero	
F -> (E)	

dove numero è un intero senza segno;

### ESPR.FL

```
/* Analizzatore lessicale per le espressioni aritmetiche (non ambigue)
 *
 * Questo file (espr.fl) e' da usare insieme all'analizzatore sintattico
 * realizzato con BISON a partire da espr.y.
 */

%{
#include <stdio.h>
#include "espr.tab.h"
%}

/* regular definitions */

delim [ \t\n]
ws    {delim}+
digit [0-9]
number   {digit}+
%option noyywrap
%%
{ws}      ;
{number}   { return(NUM); }
\+          { return(PIU); }
\‐          { return(MENO); }
\*          { return(PER); }
[/]         { return(DIVISO); }
\(          { return(PAR_AP); }
\)          { return(PAR_CH); }
%%
```

## ESPR.Y

```
/* Analizzatore sintattico per espressioni aritmetiche (non ambigue)
 * (il programma effettua soltanto il riconoscimento della correttezza
 * sintattica della stringa di input: l'output e' "Errore Sintattico" se
 * c'e' un errore sintattico, altrimenti l'output e' nullo.)
 *
 * Questo file (espr.y) e' da usare insieme all'analizzatore lessicale
 * realizzato con FLEX a partire da espr.fl.
 *
 * Riconosce espressioni aritmetiche secondo la seguente grammatica:
 *
 *     E -> E + T | E - T | T
 *     T -> T * F | T / F | F
 *     F -> ( E ) | numero
 *
 *
 * Sequenza di comandi per generare l'analizzatore sintattico:
 *
 *     flex espr.fl
 *     bison -d espr.y
 *     gcc lex.yy.c espr.tab.c -o espr
 */
%{
#include <stdio.h>
%}

%token NUM PIU MENO PER DIVISO PAR_AP PAR_CH
%start Expr
%error-verbose

%%
Expr: Expr PIU Term | Expr MENO Term | Term

Term: Term PER Factor | Term DIVISO Factor | Factor

Factor: NUM | PAR_AP Expr PAR_CH
%%

main()
{
    yyparse();
}
yyerror (char *s)
{
    printf ("Errore Sintattico\n");
}
```

Se invece volessimo effettuare anche la valutazione della stringa:

**ESPR2.Y**

```
%{
#include <stdio.h>
%}
%token NUM PIU PER PAR_AP PAR_CH NEWL
%start Input
%%
Input:    /* empty */
          | Input Line ;
Line:     NEWL
          | Expr NEWL {printf("%d\n", $1); }
          ;
Expr:    Expr PIU Term   {$$=$1+$3; }
          | Term   { $$ = $1; }
          ;
Term:    Term PER Factor {$$=$1*$3; }
          | Factor { $$ = $1; }
          ;
Factor:  NUM
          | PAR_AP Expr PAR_CH
          ;
%%
main()
{
    yyparse();
}
yyerror (char *s)
{
    printf ("Errore Sintattico\n");
}
```

Esercizi:

1. Scrivere un programma in Bison che valuti un'espressione in forma postfissa.

### **POSTFISSA.FL**

```
%{
#include "post.tab.h"
%
/* Definizioni regolari */
delim [ \t]
ws    {delim}+
digit [0-9]
number   {digit}+
%option noyywrap
%%
{ws} ;
{number}  {yyval=atoi(yytext); return(NUM); /* yyval di default è intero */}
\+      {return(PIU);}
\*      {return(PER);}
\n      {return(NEWL);}
%%
```

### **POSTFISSA.Y**

```
%{
#include <stdio.h>
%
%token NUM PIU PER NEWL
%start Assioma
%%
Assioma:   /* empty (regola vuota) */
          | Assioma Line
          ;
Line:     NEWL |
          Expr_p NEWL {printf("Valore : %d\n", $1);}
          ;
Expr_p:   NUM        /* Si dovrebbe scrivere ($$=$1) ma è superflua perchè è
          | Expr_p Expr_p PIU {$$=$1+$2;}
          | Expr_p Expr_p PER {$$=$1+$2;}
          ;
%%
main()
{
yyparse();
}
yyerror (char *s)
{ printf("Errore Sintattico"); }
```

2. Scrivere un programma in Bison che converta in notazione postfissa, le espressioni in forma infissa corretta.

### **POST\_INF.FL**

```
%{
#include "pos_inf.tab.h"
%
/* Definizioni regolari */
delim [ \t]
ws    {delim}+
digit [0-9]
number   {digit}+
%option noyywrap
%%
{ws} ;
{number} {yyval=atoi(yytext); return(NUM);}
\+      {return(PIU);}
\*      {return(PER);}
\(
\)
\n      {return(NEWL);}
%%
```

### **POST\_INF.Y**

```
%token NUM PIU PER PAR_AP PAR_CH NEWL
%left PIU
%left PER
%start Assioma
%error-verbose
%%
Assioma: /* empty (regola vuota) */
        | Assioma Line
        ;
Line:   NEWL |
        Expr NEWL {printf("\n");}
Expr:   NUM   {printf("%d ",$1);}
        | Expr PIU Expr {printf("+ ");}
        | Expr PER Expr {printf("* ");}
        | PAR_AP Expr PAR_CH { };
%%
main()
{
yyparse();
}
yyerror (char *s)
{ printf("Error : %s",s); }
```

3. Scrivere un programma che testi se una parola è palindroma

#### PALINDROMA.FL

```
%{  
#include "palind.tab.h"  
%}  
delim [ \t]  
ws {delim}+  
%option noyywrap  
%%  
{ws} ;  
a {return(A);} ;  
b {return(B);} ;  
\n {return(NEWL);} ;  
. {printf("Token non riconosciuto\n");}  
%%
```

#### PALINDROMA.Y

```
%token  
A B NEWL  
%start Input  
%error-verbose  
%glr-parser  
%%  
Input: Pal|  
      Input Pal;  
Pal: Expr NEWL {printf("Palindroma\n");}  
Expr: A Expr A |  
     B Expr B |  
     A |  
     B |  
     ;  
%%  
int main()  
{ yyparse(); }  
yyerror (char *s)  
{ printf(" Non Palindroma\n"); }
```

4. Scrivere un programma in Bison che converta in notazione infissa, le espressioni in forma postfissa corretta.



# 10. Analisi semantica

Il processo di compilazione consiste in una serie di passi chiamati **fasi della compilazione**

- |   |         |
|---|---------|
| <ul style="list-style-type: none"><li><input type="checkbox"/> Analisi lessicale</li><li><input type="checkbox"/> Analisi sintattica</li><li><input checked="" type="checkbox"/> <b><u>Analisi semantica</u></b></li><li><input type="checkbox"/> Generazione del codice intermedio</li></ul> | Analisi |
| <ul style="list-style-type: none"><li><input type="checkbox"/> Ottimizzazione</li><li><input type="checkbox"/> Generazione del codice oggetto</li></ul>   | Sintesi |

L'analisi semantica interpreta il significato associato alla struttura sintattica e verifica che le regole di impiego del linguaggio siano soddisfatte.

Obiettivi dell'analisi semantica:

- Raccogliere le informazioni relative agli identificatori introdotti nella tabella dei simboli;
- Verificare la correttezza d'impiego degli identificatori e dei costrutti del linguaggio;
- Segnalare gli errori in modo chiaro e senza ridondanze.

## 10.1. Semantica statica

Indipendente dai dati su cui opera il programma sorgente:

- verificare che una variabile sia dichiarata una sola volta e che venga usata coerentemente al tipo dichiarato;
- rispetto delle regole che governano i tipi degli operandi nelle espressioni e negli assegnamenti (controllo sui tipi);
- rispetto delle regole di visibilità e univocità degli identificatori;
- correttezza delle strutture di controllo del linguaggio;
- rispetto delle regole di comunicazione fra i vari moduli (interni e/o esterni) che costituiscono il programma.
- verificare che le chiamate dei sottoprogrammi siano congruenti con le loro dichiarazioni.

## 10.2. Semantica dinamica

Dipendente dai dati su cui opera il programma sorgente:

- controllo sui cicli infiniti;
- controllo sulla dereferenziazione di puntatori NULL;
- controllo sui limiti degli array (per esempio, l'indice di un array non superi i limiti stabiliti dalla sua dichiarazione);
- un dato letto in input sia compatibile con il tipo della variabile a cui è destinato.

L'analizzatore semantico si occupa della semantica statica, mentre la semantica dinamica spetta all'interprete o al supporto esecutivo.

## 10.3. Analisi semantica statica

L'analisi semantica statica, come quella lessicale e sintattica, consta di due fasi:

- la descrizione delle analisi da eseguire
- l'implementazione delle analisi mediante opportuni algoritmi.

## 10.4. Esempi di semantica statica e semantica dinamica

Semantica statica: indipendente dai dati su cui opera il programma sorgente.

```
var i : real;  
a : array [1..100] of integer;  
...  
i:=3.5;  
a[i]:=3;
```

Semantica dinamica: dipendente dai dati su cui opera il programma sorgente.

```
var i : integer;  
a : array [1..100] of integer;  
...  
read(i);  
a[i]:=3;
```

L'analizzatore semantico si occupa della semantica statica, mentre la semantica dinamica spetta all'interprete o al supporto esecutivo.

## 10.5. Analisi semantica statica

Nell'analisi sintattica esistono formalismi standard per descrivere la sintassi e i vari algoritmi di parsing per implementare la sintassi stessa.

Nell'analisi semantica la situazione non è così definita:

- non esiste una metodologia standard per definire o descrivere la semantica statica di un linguaggio;
- esiste un'enorme varietà di controlli semanticici statici nei vari linguaggi.

L'idea di base, è in ogni caso, quella di aumentare il lavoro del parser con azioni speciali di tipo semantico.

## 10.6. Attributi e Grammatiche con attributi

Esistono diversi approcci per la descrizione della semantica statica di un linguaggio. Uno tra questi si definisce attraverso **le grammatiche con attributi (attribute grammars)**.

Sono grammatiche context-free in cui sono state aggiunte proprietà delle entità sintattiche del linguaggio (**attributi**) e regole di valutazione di tali proprietà (**regole semantiche o equazioni di attributi**).

Una grammatica con attributi specifica quindi sia azioni sintattiche che semantiche.

Le grammatiche con attributi sono utilizzabili in tutti i linguaggi di programmazione che obbediscono al principio della "SEMANTICA GUIDATA DALLA SINTASSI" che asserisce che la semantica non dipende dal contesto ma è strettamente legata alla sintassi.

Ciò accade per tutti i moderni linguaggi di programmazione.

Un attributo è qualunque proprietà delle entità sintattiche di un linguaggio.

Gli attributi possono variare molto rispetto al loro contenuto, alla loro complessità e principalmente in relazione al momento in cui essi sono calcolati.

Gli algoritmi per l'implementazione dell'analisi semantica non sono chiaramente esprimibili come quelli di parsing.

Esempi di attributi sono:

- Il tipo di una variabile
- Il valore di una espressione
- La locazione di una variabile in memoria
- Il codice oggetto di una procedura
- .....

Gli attributi possono essere:

- STATICI: calcolati al tempo di compilazione (i tipi di dati, il numero delle cifre significative...)
- DINAMICI: calcolati al tempo di esecuzione (valore di una espressione, le locazioni di memoria di una struttura dati dinamica)

Il calcolo di un attributo e l'associazione del suo valore ad un costrutto sintattico è detto **binding** (legame) dell'attributo.

Il momento in cui questo legame avviene è detto **binding time**. Differenti attributi possono avere binding time diversi, o anche lo stesso attributo può avere differenti binding time che variano da linguaggio a linguaggio. Noi siamo interessati agli attributi statici.

#### 10.6.1. Binding time

In linguaggi dichiarativi come C e Pascal, il tipo di una variabile o di un'espressione è un attributo definito al tempo di compilazione. Il type checker (analizzatore semantico che calcola tale attributo per tutte le entità del linguaggio per le quali è definito e verifica che tali tipi siano conformi alle regole dei tipi del linguaggio) in C e in Pascal agisce durante la fase di compilazione, mentre in linguaggi come LISP o alcuni linguaggi ad oggetti tale processo avviene durante l'esecuzione.

Il valore di un'espressione è generalmente calcolato al tempo di esecuzione. In alcuni casi però (3+4\*5 per esempio) l'analizzatore semantico può scegliere di valutare l'espressione durante la compilazione.

L'allocazione di una variabile può essere sia statica che dinamica e dipende dal linguaggio e dal tipo di variabile: in FORTRAN tutte le variabili sono statiche, in LISP sono tutte dinamiche mentre in C e Pascal possono essere sia statiche che dinamiche.

Il codice oggetto di una procedura è un attributo statico.

#### 10.6.2. Grammatiche con attributi

Sono grammatiche context-free in cui sono state aggiunte proprietà delle entità sintattiche del linguaggio (**attributi**) e regole di valutazione di tali proprietà (**regole semantiche o equazioni di attributi**).

Una grammatica con attributi è quindi una terna (**G, A, R**) dove :

- **G** è una grammatica context-free
- **A** è l'insieme degli attributi associati ad ogni simbolo terminale e non
- **R** è l'insieme di regole associate alle varie produzioni di G.

#### 10.6.3. Rappresentazione di un attributo

In queste ipotesi gli attributi sono associati direttamente ai simboli della grammatica (terminali e non).

Se X è un simbolo sintattico e 'a' è un attributo associato ad X scriveremo:

X.a

per accedere al valore corrispondente.

#### 10.6.4. Grammatiche con attributi e semantica guidata dalla sintassi

Dato un insieme di attributi

$$a_1, a_2, \dots, a_k$$

il principio della **semantica guidata dalla sintassi** afferma che per ogni produzione del tipo

$$X_0 \rightarrow X_1 \dots X_n$$

i valori degli attributi  $X_i.a_j$  per ogni simbolo sintattico  $X_i$  sono legati ai valori degli attributi degli altri simboli presenti nella produzione.

Questo legame è definito nella forma:

$$X_i.a_j = f_{ij}(X_0.a_0, X_0.a_1, \dots, X_0.a_k, \dots, X_n.a_0, X_n.a_1, \dots, X_n.a_k)$$

e costituisce una **regola semantica (attribute equation)**.

Si definisce grammatica con attributi l'insieme di tali regole per ogni produzione del linguaggio.

Le grammatiche con attributi sono specificate mediante tabelle, in cui, accanto ad ogni produzione, sono elencate le regole semantiche associate.

regola grammaticale	regole semantiche
regola 1	equazione 1.1 equazione 1.2 equazione 1.3
regola n	equazione n.1 equazione n.2

#### Osservazioni

- Sono uno strumento molto potente per l'analisi semantica.
- Pur non essendo semplici da usare, le grammatiche con attributi sono meno complesse di quanto sembrano:
  - di solito gli attributi sono pochi
  - le regole semantiche dipendono raramente da tutti gli attributi
  - spesso gli attributi possono essere separati in sottoinsiemi e le regole semantiche possono essere scritte separatamente per ogni sottoinsieme
- I manuali dei linguaggi di programmazione non definiscono la grammatica con attributi, sicché il progettista del compilatore deve scriversela a mano.
- Nonostante ciò, è importante studiare le grammatiche con attributi perché consentono di definire analisi semantiche più semplici, concise, con meno errori, e che consentono una più semplice comprensione del codice.

### 10.6.5. Esempi

#### ➤ Esempio 1

Si consideri la seguente grammatica per esprimere un numero in binario:

$$\begin{aligned} \text{number} &\rightarrow \text{number digit} \mid \text{digit} \\ \text{digit} &\rightarrow 0 \mid 1 \end{aligned}$$

Un attributo significativo potrebbe essere il suo valore in decimale.

Definiamo un attributo **val** per i simboli non-terminali **number** e **digit** (**number.val** e **digit.val**).

Come diventa la grammatica con attributo **val** ?

regola grammaticale	regole semantiche
$\text{number}_1 \rightarrow \text{number}_2 \text{ digit}$	$\text{number}_1.\text{val} = 2 * \text{number}_2.\text{val} + \text{digit}.\text{val}$
$\text{number} \rightarrow \text{digit}$	$\text{number}.\text{val} = \text{digit}.\text{val}$
$\text{digit} \rightarrow 1$	$\text{digit}.\text{val} = 1$
$\text{digit} \rightarrow 0$	$\text{digit}.\text{val} = 0$

- La regola **digit → 1** implica che digit ha il valore che la cifra stessa rappresenta, e quindi **digit.val = 1**
- Analogamente **digit → 0** implica che **digit.val=0**
- Se il numero è derivato usando la regola **number → digit** allora il suo valore è **number.val=digit.val**
- Consideriamo che il numero sia derivato usando la regola

$$\text{number} \rightarrow \text{number digit}$$

Riscriviamo la regola distinguendo le due occorrenze di number:

$$\text{number}_1 \rightarrow \text{number}_2 \text{ digit}$$

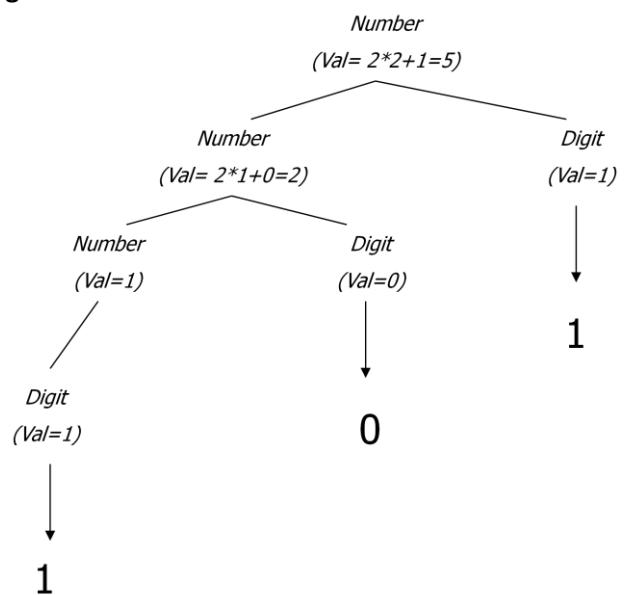
da cui si ottiene

$$\text{number}_1.\text{val} \rightarrow 2 * \text{number}_2.\text{val} + \text{digit}.\text{val}$$

Occorre notare la differenza tra la rappresentazione sintattica di digit e il suo contenuto semantico (valore). Nella regola **digit → 1** il simbolo 1 è un token mentre in **digit.val = 1** il simbolo 1 il valore numerico.

Albero sintattico decorato

- Il significato delle regole semantiche per una particolare stringa può essere descritto usando l'albero sintattico associato agli attributi (**albero sintattico decorato**).
- Per esempio descriviamo l'albero sintattico decorato per la stringa 101.
- Il calcolo dell'apposita regola semantica è indicato all'interno del nodo.
- E' importante osservare come avviene il calcolo degli attributi per avere un'idea di come possano funzionare gli algoritmi di calcolo degli attributi stessi.



#### ➤ Esempio 2

Consideriamo la seguente grammatica per semplici espressioni aritmetiche:

$$\begin{aligned} \text{exp} &\rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term} \\ \text{term} &\rightarrow \text{term} * \text{factor} \mid \text{factor} \\ \text{factor} &\rightarrow (\text{exp}) \mid \text{number} \end{aligned}$$

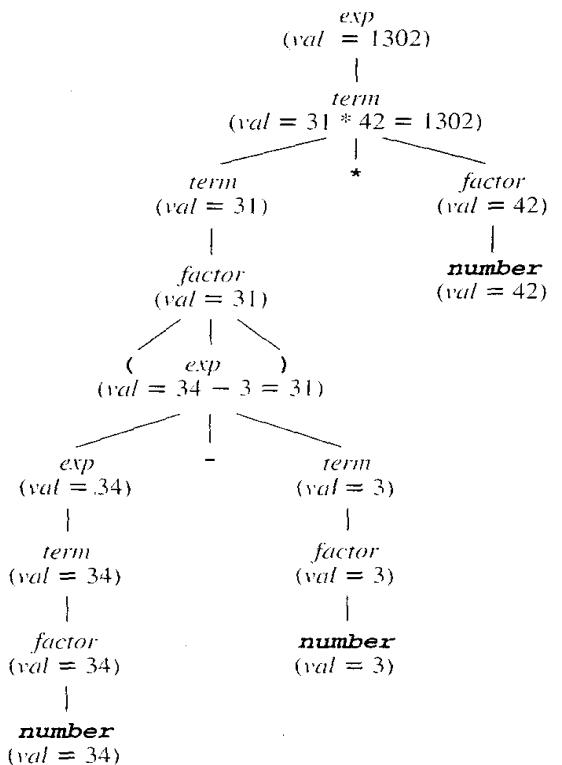
L'attributo considerato è sempre **val** cioè il valore numerico di **exp** (**term** o **factor**).

La grammatica con attributi corrispondente è:

regola grammaticale	regole semantiche
$exp_1 \rightarrow exp_2 + term$	$exp_1.val = exp_2.val + term.val$
$exp_1 \rightarrow exp_2 - term$	$exp_1.val = exp_2.val - term.val$
$exp \rightarrow term$	$exp.val = term.val$
$term_1 \rightarrow term_2 * factor$	$term_1.val = term_2.val * factor.val$
$term \rightarrow factor$	$term.val = factor.val$
$factor \rightarrow ( exp )$	$factor.val = exp.val$
$factor \rightarrow number$	$factor.val = number.val$

**number** è considerato come simbolo terminale; ciò significa che sarà l'analizzatore lessicale, ad esempio, ad inizializzare opportunamente il campo **number.val**. Alternativamente bisogna introdurre produzioni esplicite e corrispondenti regole semantiche (per esempio le regole dell'esempio precedente).

La figura a destra descrive l'albero sintattico decorato per la stringa  $(34-3)*42$



### ➤ Esempio 3

Completiamo la grammatica dell'esempio 1 in modo da esprimere numeri in codice binario (b) e ternario (t):

```

based_num → number base
base → b | t
number → number digit | digit
digit → 0 | 1 | 2
  
```

Gli attributi sono:

based_num	:	val
base	:	val
number	:	val , base
digit	:	val , base

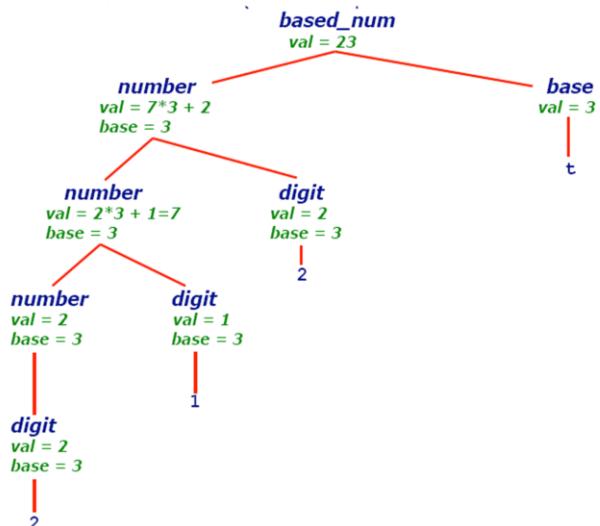
La grammatica con attributi è:

regola grammaticale	regole semantiche
$based\_num \rightarrow number\ base$	$based\_num.val = number.val$ $number.base = base.val$
$base \rightarrow b$	$base.val = 2$
$base \rightarrow t$	$base.val = 3$
$number_1 \rightarrow number_2\ digit$	$number_2.base = number_1.base$ $digit.base = number_1.base$ $number_1.val =$ $\text{if } ( digit.val == \text{error} ) \text{ or } ( number_2.val == \text{error} )$ then error else $number_2.base * number_2.val + digit.val$
$number \rightarrow digit$	$number.val = digit.val$ $digit.base = number.base$
$digit \rightarrow 2$	$digit.val = \text{if } ( digit.base == 2 ) \text{ then error}$ else $digit.val = 2$
$digit \rightarrow 1$	$digit.val = 1$
$digit \rightarrow 0$	$digit.val = 0$

Osservazioni:

- Questa grammatica può generare stringhe corrette sintatticamente ma non semanticamente: **21b** (l'analizzatore semantico dovrebbe rilevare questo errore).
- Nelle regole semantiche è stato utilizzato il costrutto "if-then-else".
- Ciò è perfettamente lecito perché le regole semantiche saranno scritte in un linguaggio di programmazione (sarà quindi possibile invocare anche funzioni).

La figura a destra descrive l'albero sintattico decorato per la stringa 212t



#### 10.6.6. Definire e calcolare gli attributi

Negli primi due esempi il calcolo degli attributi è avvenuto con una semplice visita dell'albero di parsing. In alcuni casi invece può servire completare l'analisi sintattica e la costruzione dell'albero di parsing per poi procedere con l'analisi semantica. Ciò implica che il compilatore deve effettuare più di una passata.

Problemi:

- Come definire gli attributi?
- Come definire le regole semantiche?
- Come specificare l'ordine in cui calcolarli?
- Quali algoritmi utilizzare per calcolarli?

La definizione degli attributi è relativamente semplice: basta inserire le opportune dichiarazioni nella parte iniziale di qualunque analizzatore sintattico.

Le parti destre delle regole devono essere espressioni effettivamente calcolabili al momento della derivazione della produzione. Questo vuol dire che gli attributi coinvolti devono essere già disponibili.

In alcuni casi il calcolo degli attributi è semplice:

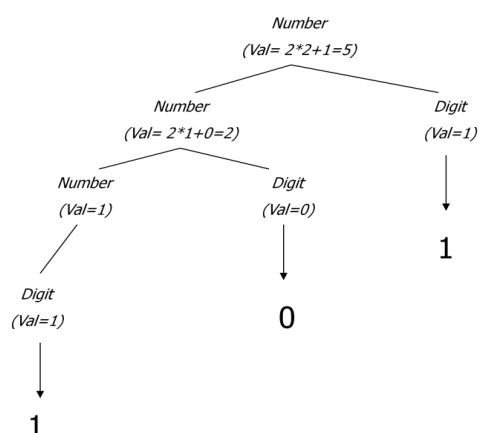
Dipendono dal tipo di visita dell'albero sintattico decorato.

Esempio:

*number → number digit | digit*  
*digit → 0 | 1*

regola grammaticale	regole semantiche
<i>number<sub>1</sub> → number<sub>2</sub> digit</i>	<i>number<sub>1</sub>.val = 2*number<sub>2</sub>.val + digit.val</i>
<i>number → digit</i>	<i>number.val = digit.val</i>
<i>digit → 1</i>	<i>digit.val = 1</i>
<i>digit → 0</i>	<i>digit.val = 0</i>

In questo caso il calcolo del valore dell'attributo val si ottiene con una visita in ordine posticipato dell'albero decorato.



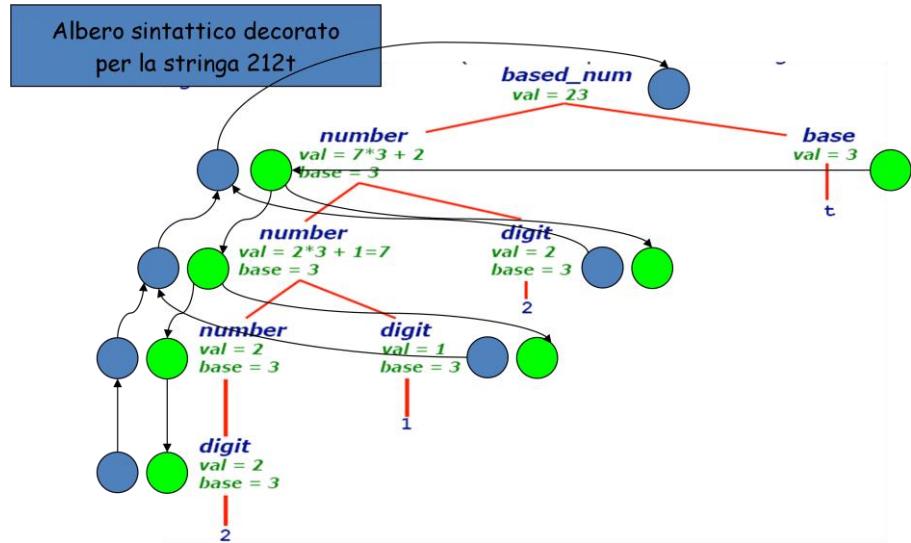
### 10.6.7. Dipendenze funzionali degli attributi

- Per capire come il diverso modo di visitare l'albero sintattico decorato può influire nella corretta valutazione degli attributi occorre introdurre il concetto di dipendenze funzionali degli attributi.
- Una regola semantica assegna un valore ad un attributo di un nodo dell'albero sintattico, in funzione dei valori di altri attributi del nodo stesso e dei nodi vicini (padre, fratelli e figli).
- L'attributo dipende dunque funzionalmente da altri attributi, i cui valori devono essere noti per consentire il calcolo.

### 10.6.8. Grafo delle dipendenze

- Data una grammatica con attributi, ad ogni regola è associato un grafo delle dipendenze.
- Per ogni nodo dell'albero di parsing etichettato con il simbolo X, il grafo delle dipendenze avrà un nodo per ogni attributo di X.
- Se una regola definisce il valore dell'attributo A.b in funzione di X.c, allora esisterà un arco da X.c a A.b.

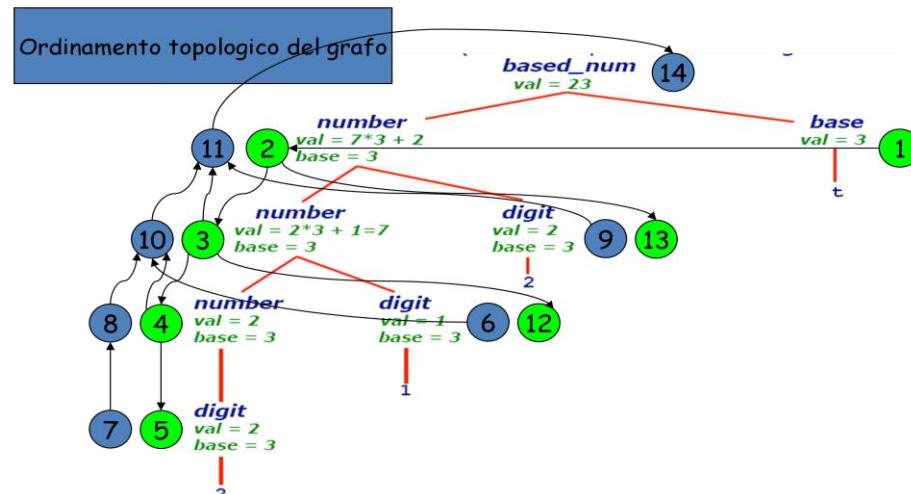
Per esempio:



### 10.6.9. Algoritmo generale per il calcolo degli attributi

- L'algoritmo deve calcolare l'attributo di un nodo prima del calcolo dell'attributo del nodo successore; bisogna cioè trovare un ordinamento topologico del grafo.
- Il grafo deve essere aciclico.
- Il tempo di calcolo può diventare eccessivo.
- Cos'è: sequenza dei vertici in modo tale che se esiste un arco da u a v, allora u precede v nell'ordinamento.

Per esempio:



## 10.6.10. Si restringe la classe delle grammatiche con attributi

Esiste una classe di grammatiche con attributi non circolari:

- Attributi sintetizzati (synthesized attributes)
- Attributi ereditati (inherited attributes)

### 10.6.10.1. Attributi sintetizzati

- Un attributo associato ad un nodo dell'albero sintattico si dice **sintetizzato** se il suo valore dipende solo dai valori degli attributi dei nodi figli.

**Formalmente:** un attributo  $a$  è **sintetizzato** se data una regola  $A \rightarrow X_1 \dots X_n$ , l'unica equazione con  $a$  nella parte sinistra ha la forma:  $A.a = f(X_1.a_1, \dots, X_k.a_k, \dots, X_n.a_1, X_n.a_k)$

- Una grammatica che ha tutti attributi sintetizzati è detta grammatica **puramente sintetizzata** o grammatica con **S-attributi**.

### 10.6.10.2. Attributi ereditati

- Un attributo associato ad un nodo dell'albero sintattico si dice **ereditato** se il suo valore dipende dai valori degli attributi del nodo padre e/o dei nodi fratelli.
- Gli attributi ereditati sono utili per esprimere le dipendenze di un costrutto di un linguaggio rispetto al suo contesto.

## 10.6.11. Algoritmi per il calcolo degli attributi

Nel caso di grammatiche con **S-attributi** il calcolo del valore degli attributi si ottiene con una sola visita in ordine posticipato dell'albero sintattico decorato.

Schematicamente ciò può essere rappresentato dal seguente pseudocodice:

```
procedure PostEval (T: treenode);  
begin  
    for each child C of T do  
        PostEval(C);  
    Compute all synthesized attributes of T;  
end;
```

Nel caso di grammatiche con **attributi ereditati** non è chiaro l'algoritmo di visita dell'albero: infatti in questo caso bisogna ritardare l'applicazione delle regole semantiche fino al momento in cui le informazioni contestuali e il valore degli altri attributi lo consentono.

### 10.6.12. Grammatiche con L-attributi

Si tratta di grammatiche in cui gli attributi di un nodo sono:

- sintetizzati  
*oppure*
- ereditati che dipendono:
  - dagli attributi ereditati del nodo padre  
*oppure*
  - dagli attributi ereditati o sintetizzati dei nodi fratelli che lo precedono.

In quest'ultimo caso il calcolo del valore degli attributi si ottiene con una sola visita anticipata sinistra dell'albero sintattico decorato.

#### 10.6.13. Passate per il calcolo degli attributi

- Le strategie di calcolo degli attributi in una passata (descendente o ascendente) sono applicabili quindi quando le dipendenze fra gli attributi soddisfano le condizioni piuttosto restrittive viste in precedenza.
- In certi casi non è possibile ricondursi a tali condizioni per cui è necessario ricorrere a strategie più potenti come quelle a più passate.
- Ogni passata visita l'albero parzialmente decorato con i valori degli attributi calcolati dalle passate precedenti, e calcola quel (sotto)insieme degli attributi per cui gli insiemi delle dipendenze sono disponibili nell'albero.
- A seconda delle modalità di visita (ascendenti, descendenti, da sx a dx o da dx a sx) si possono trattare le diverse classi di grammatiche ad attributi.

#### 10.7. Analisi sintattico-semantică

- Fino a questo momento abbiamo supposto che l'albero sintattico sia già costruito al momento dell'analisi semantica.
- Abbiamo separato di fatto l'analisi lessicale da quella semantica.
- In alcuni casi abbiamo visto che è possibile incorporare l'analisi semantica entro la procedura di analisi sintattica:
  - nelle grammatiche con S-attributi il calcolo degli attributi rispetta l'ordine di costruzione dell'albero da parte di un analizzatore sintattico ascendente
  - nelle grammatiche con L-attributi il calcolo degli attributi rispetta l'ordine di costruzione dell'albero da parte di un analizzatore sintattico discendente.

# 11. Tabella dei simboli (Symbol Table)

Il compilatore, nelle sue varie fasi, deve tenere traccia:

- degli identificatori utilizzati nel codice sorgente;
- dei loro attributi (tipo e caratteristiche).

Queste informazioni sono memorizzate in un'apposita struttura dati detta

## ***Tabella dei simboli (Symbol table)***

da consultare ogni volta che si incontra un identificatore:

- se esso non è presente nella tabella (prima occorrenza), allora si introduce come nuovo elemento con una serie di attributi;
- se esso è già presente, allora se ne aggiornano, eventualmente i suoi attributi.

### 11.1. Quando costruire ed interagire con la TS

La TS è consultata in tutte le fasi del processo di compilazione.

La costruzione della TS avviene, almeno in linea teorica, a partire dall'analisi lessicale e completata nelle fasi successive.

➤ Nell'analisi lessicale:

- ❑ è creata una entry nella tabella per ogni nuovo identificatore incontrato (variabili, tipi, nomi di funzioni, etichette, campi, ...);
- ❑ a questa entry non è associato alcun valore: di solito non si conosce né il tipo né le caratteristiche.

➤ Nell'analisi sintattica e semantica:

- ❑ la TS viene riempita con informazioni sui tipi e sulle caratteristiche degli identificatori.

Viene coinvolta anche nella fase di generazione del codice

### 11.2. Struttura della Tabella dei simboli

La TS è una struttura dati astratta per rappresentare insiemi di coppie

< nome\_id, lista\_attributi >

ove:

**nome\_id**, che di solito è una stringa, rappresenta la chiave per accedere a **lista\_attributi** cioè alle informazioni associate in modo univoco all'identificatore stesso.

### 11.3. Operazioni principali sulla TS

La TS, che può essere considerata una struttura dati "dizionario", ha tre operazioni di base:

- **Insert** (inserimento)  
per memorizzare le informazioni associate ad un identificatore.
- **Lookup** (accesso e/o ricerca)  
per ritrovare le informazioni associate ad un determinato identificatore.
- **Delete** (cancellazione)  
per rimuovere le informazioni associate ad un determinato identificatore quando esso non è più visibile.

#### 11.4. Problemi

- Il problema principale è quello di individuare l'organizzazione più idonea della struttura dati che implementa la TS.
- In genere, due sono i principali fattori che possono influenzare la scelta dell'organizzazione di una struttura dati:
  - lo spazio di memoria occupato
  - la velocità di accesso
- Poiché la TS normalmente è consultata migliaia di volte nel corso della compilazione, la velocità di accesso avrà maggiore priorità rispetto allo spazio di memoria occupato.
- Bisogna considerare, inoltre, che anche le specifiche del linguaggio sorgente possono influenzare l'organizzazione della TS.

#### 11.5. Struttura della Tabella dei simboli

Illustreremo le principali organizzazioni della TS e vedremo come le caratteristiche del linguaggio sorgente possono influenzare tali strutture.

#### 11.6. Realizzazione della Tabella dei simboli

La TS può essere organizzata come:

- un semplice array
- una lista concatenata
- un albero binario di ricerca
- una tabella hash

##### 11.6.1. La Tabella dei simboli come **ARRAY**

- Vantaggi:
  - semplicità di implementazione
  - facile accesso alle sue componenti
  - spazio occupato in memoria relativo ai soli dati
- Svantaggi:
  - dimensione della struttura da fissare a priori (ciò implica di fissare un limite sul numero di identificatori che si possono gestire)
  - ricerche non particolarmente efficienti
  - La struttura ad array è raramente usata tranne che per semplici compilatori.

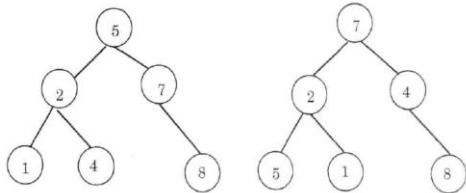
##### 11.6.2. La TS come **lista concatenata**

- In questo caso la struttura è dinamica.
  - La ricerca resta lineare ma è possibile ottimizzarla:
    - mantenendo la lista ordinata
    - utilizzando il metodo della **riorganizzazione dinamica**
- Secondo questo metodo le voci a cui si accede sono spostate via via verso la testa della lista cosicché i simboli più usati avranno un tempo di accesso inferiore (si presume che le istruzioni vicine nel sorgente potranno farvi riferimento).
- Gestione semplice ma inefficiente per grosse tabelle.

### 11.6.3. La TS come ABR

- Gli ABR sono particolari alberi binari tali che per ogni nodo mantengono un ordinamento ponendo nel sottoalbero di sinistra gli elementi minori e in quello di destra tutti gli elementi maggiori del nodo considerato.
- Si dimostra che se l'albero è costruito in modo casuale ha un altezza di approssimativamente  $1.39\log n$ , dove  $n$  è il numero dei nodi.  
Le tre operazioni hanno un costo proporzionale all'altezza. ☺  
Frequenti cancellazioni possono sbilanciare l'albero. ☹
- Supportano efficienti operazioni di ordinamento.

Albero binario di ricerca:



Albero binario:

```

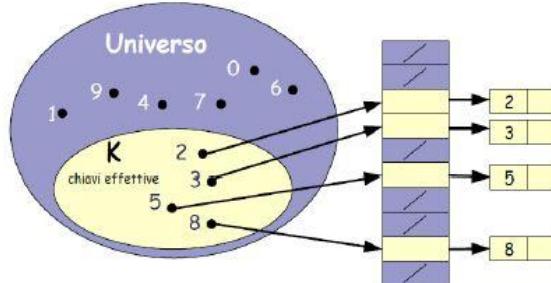
struct treenode {
    int key;
    struct treenode *sx;
    struct treenode *dx;
};

typedef struct treenode TREE;
typedef TREE *TREEPTR;

```

### 11.6.4. La TS ad accesso diretto

- Si suppone che le chiavi associate agli  $n$  elementi da memorizzare sono interi in  $[0..m-1]$  con  $n \leq m$
- Assumiamo che le chiavi siano tutte distinte
- Si definisce un array  $V$  di dimensione  $m$  tale che se un certo elemento  $x$  ha chiave  $k$ , allora  $V[k]=x$
- Le caselle che non corrispondono ad alcuna chiave avranno NULL
- Le chiavi sono usate come indici per spostarci nella struttura dati
- *Il tempo richiesto per ogni azione è costante*
- *Se  $m$  è molto grande può diventare impraticabile!!!*
- *Può comportare un enorme spreco di memoria!!!*



Misuriamo il grado di riempimento di una tabella introducendo il fattore di carico:

$$\alpha = N/M$$

dove  $M$  è la dimensione della tabella e  $N$  è il numero di chiavi effettivamente utilizzate.

Per esempio la tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre

$$N=100, M=10^6, \alpha=0,0001=0,01\%$$

E se le chiavi non sono interi?

### 11.6.5. La TS come **tabella hash**

Idea: Dimensionare la tabella in base al numero di elementi attesi ed utilizzare una speciale funzione (funzione hash) per indicizzare la tabella.

La maggior parte dei compilatori usa questo tipo di organizzazione: in particolare la TS è una “hash table with chaining”.

Questo tipo di organizzazione consente di implementare le tre operazioni fondamentali con un numero molto contenuto di accessi indipendente dalla dimensione della tabella.

Una ricerca basata su hashing è completamente diversa da quella basata su confronti:

invece che spostarci nella struttura dati in funzione dell'esito dei confronti fra chiavi, cerchiamo di accedere agli elementi della tabella in modo diretto tramite operazioni aritmetiche che trasformano le chiavi in indirizzi della tabella.

E' necessario determinare la **funzione di hash** che trasforma una chiave in un indirizzo della tabella.

#### 11.6.5.1. La funzione hash

Un indirizzo hash è un possibile valore dell'indice di un vettore di elementi destinati ad ospitare gli elementi della tabella.

La corrispondenza **nome\_id - posizione** viene calcolata da un'apposita funzione detta funzione hash:

$$h : S \rightarrow \{0..M-1\}$$

dove  $S$  è l'insieme delle chiavi.

Una funzione hash ideale è facilmente calcolabile e approssima una funzione casuale: per ogni valore di input, i possibili valori di output dovrebbero essere in qualche senso equiprobabili.

L'idea è quella di implementare la TS come un array così definito:

```
#define N .. ; //lunghezza massima della tabella
struct Item{
    int key;
    . .
};

typedef struct Item ITEM;
typedef ITEM *HASH_TAB;
void HT_init(HASH_TAB *st;int M);
{int i;
*st=(HASH_TAB)malloc(M*sizeof(ITEM));
for (i=0;i<M;i++) *st[i]=NullItem;
}
```

Oppure

```
struct ITEM *hashtab[N];
```

Con l'hashing, un elemento con chiave  $k$  viene memorizzato nella cella  $h(k)$ .

**Pro:** riduciamo lo spazio necessario per memorizzare la tabella.

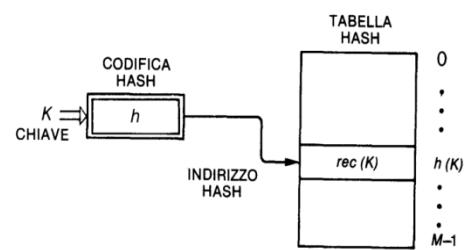
**Contro:** perdiamo la corrispondenza tra chiavi e posizioni in tabella.

Operazioni:

```
insert(ITEM e; key k);
st[h(k)]=e;

delete (key k);
st[h(k)]=NULL;

ITEM search (key k);
return st[h(k)]
```



### 11.6.5.2. Collisioni

La situazione ideale si ha quando la funzione  $h$  è iniettiva:

$$\text{per ogni } K, K' \in S: K \neq K' \Rightarrow h(K) \neq h(K')$$

In questo caso ogni chiave ha il suo indirizzo hash distinto da quello di tutte le altre chiavi: ogni elemento della tabella può essere raggiunto con un solo accesso (funzione hash perfetta).

Naturalmente si richiede che  $M$  sia non minore del numero delle possibili chiavi, cosa che in genere non accade.

Ciò potrebbe non sempre essere possibile, per cui si accetta il fatto che due o più chiavi “collidano” cioè abbiano lo stesso indirizzo hash.

- Le chiavi possibili sono in genere moltissime e non note a priori (per esempio le variabili che un utente inserirà in un programma, o i cognomi dei clienti di una ditta), quindi si ha  $|Insieme delle chiavi| >> M$ .
- Dunque è inevitabile che nell'applicazione si presentino due o più chiavi  $K_1, K_2 \dots$  con lo stesso indirizzo hash, cioè  $h(K_1)=h(K_2)=\dots$   
Nasce quindi un problema di *collisioni*: solo una delle chiavi (tipicamente la prima che si è presentata, diciamo  $K_1$ ) potrà essere allocata in  $A[h(K_1)]$ , e le altre saranno poste altrove.
- Dobbiamo quindi affrontare tre problemi:
  - 1) come si sceglie la dimensione  $M$ ;
  - 2) come si calcola la funzione  $h$ ;
  - 3) come si risolvono le collisioni.

### 11.6.5.3. Scelta di $M$ e fattore di carico

- Se  $N$  è il numero variabile degli elementi e  $M$  è la dimensione fissata del vettore, si definisce fattore di carico  $\alpha=N/M$  e si dovrebbe scegliere presumibilmente  $M$  in modo tale che  $\alpha$  non superi 0,9. Se ciò non dovesse essere possibile si potrebbe raddoppiare la dimensione della tabella e riallocare tutte le chiavi nel nuovo vettore.
- Spesso,  $M$  si sceglie come potenza di due o come numero primo. Ciò dipende dalla funzione hash.

### 11.6.5.4. Funzioni hash (hash table di taglia $M$ )

- Se le chiavi sono numeri reali casuali in  $[0,1)$ , la funzione hash è  $h(k)=\lfloor kM \rfloor$
- Se le chiavi sono numeri reali casuali in  $[s,t)$ , allora  $h(k)=\lfloor (k-s)/(t-s) * M \rfloor$
- Se le chiavi sono intere,  $h(k)=k \% M$  (hashing modulare). Se  $M=2^p$ ,  $h(k)$  rappresenta i  $p$  bit meno significativi. Una buona scelta sarebbe  $M$  un primo non troppo vicino a potenza di 2.
- Se le chiavi sono stringhe, allora la funzione hash può essere:

```
int hash(char *s) {
    /* calcola il valore HASH di s;*/
    int h=0,a=127;
    for(;*s]!='\0';s++)
        h=(a*h+s)%HASHSIZE; /*HASHSIZE deve essere un primo*/
    return h;}
```

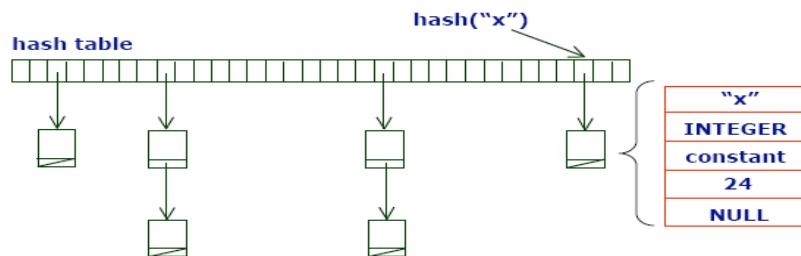
Ogni carattere contribuisce in modo unico al risultato finale. Per avere una buona (cioè uniforme) distribuzione degli oggetti nella tabella (e avere collisioni distribuite) è conveniente utilizzare dei numeri primi per la base delle potenze e per la dimensione delle tabelle.

#### 11.6.5.5. Metodi classici di risoluzione delle collisioni

- Liste di collisione (hash table with chaining): gli elementi collidenti sono contenuti in liste esterne alla tabella;  $T[i]$  punta alla lista di elementi tali che  $h(k)=i$ .
- Indirizzamento aperto: tutti gli elementi sono contenuti nella tabella; se una cella è occupata, se ne cerca un'altra libera.

#### 11.6.5.6. Lista di collisione (hash table with chaining)

La TS sarà un array di puntatori ciascuno dei quali punta ad una lista concatenata (chain o bucket) in cui ogni nodo memorizza le informazioni relative ad ogni identificatore il cui indirizzo hash collide.



- La probabilità che il numero di chiavi in ciascuna lista sia pari al fattore di carico è molto vicina a 1.
- E' un metodo molto efficiente per  $\alpha > 1$ .
- Caso della ricerca (caso medio): nell'ipotesi di uniformità semplice della funzione hash, una ricerca richiede in media un tempo  $\Theta(1+\alpha)$
- Se n è proporzionale a M allora diventa  $\Theta(1)$

Osservazioni:

- Alcuni compilatori scelgono di utilizzare una sola tabella dei simboli (che contiene qualunque informazione), che è dunque abbastanza complessa.
- Altri compilatori scelgono di separare le informazioni in differenti tabelle (tabella per i tipi, per la portata delle dichiarazioni, etc.)

#### 11.6.6. Comportamento e implementazione della ST

- Il comportamento di una symbol table dipende pesantemente dalle proprietà delle dichiarazioni del linguaggio che deve essere tradotto;
- Il funzionamento delle operazioni di insert e delete, quando devono essere chiamate, quali attributi inserire, variano da linguaggio a linguaggio.

## 11.7. Dichiarazioni

Esistono 4 principali tipi di dichiarazioni in un linguaggio di programmazione:

- **Dichiarazioni di costanti:** `const int A=2;`

(in C esiste anche `#define` che però viene gestito dal preprocessore)

Associano valori a nomi; In Pascal si richiede che siano statiche, ovvero computabili in fase di compilazione; il compilatore sostituisce il nome col valore. Quando sono dinamiche (per es. C) si può assegnare valore solo una volta.

- **Dichiarazioni di tipi:**

In Pascal `type ciccio=array[1..6] of char;`

In C ci sono dichiarazioni `struct` e `union` che definiscono delle vere e proprie strutture di tipo:

```
struct nodo {  
    char *dname;  
    struct nodo * next; }
```

Oppure `typedef`: `typedef struct nodo *ptrnodo;`

Associano nomi a tipi nuovi o alias di tipi.

- **Dichiarazioni di variabili:**

In Pascal `var c:integer;`

In C `int c;`

Associano nomi a tipi. Comunemente fanno uso di attributi relativi alla visibilità (scope).

- **Dichiarazioni di procedure/funzioni:**

Sono dichiarazioni di costanti di tipo procedure o funzioni;

Può essere usata una sola TS per tutti i tipi di dichiarazione, diverse TS per ogni tipo di dichiarazione ma esistono linguaggi (come il C e il Pascal) che hanno differenti TS per diverse regioni del programma (procedure o funzioni).

## 11.8. Attributi di visibilità

Nel caso di variabili, un attributo implicito è la visibilità di una dichiarazione, o la regione del programma dove la dichiarazione si applica.

Un attributo collegato alla visibilità è la locazione di memoria per la variabile dichiarata.

Un altro è il tempo di vita della variabile: in C tutte le variabili esterne a funzioni sono allocate staticamente (prioritaria all'inizio dell'esecuzione). Il tempo di vita è quello del programma.

Nel caso di variabili dichiarate dentro una funzione, il tempo di vita dura quanto la funzione.

## 11.9. Visibilità e struttura a blocchi

- Nei linguaggi senza dichiarazioni annidate (Fortran) la TS così come l'abbiamo descritta è più che sufficiente.
- Molti linguaggi di programmazione moderni sono strutturati a blocchi. Un blocco è un costrutto che può contenere dichiarazioni. E' possibile quindi avere anche dichiarazioni annidate. Pertanto bisogna gestire con la TS anche il fatto che una dichiarazione ha una sua portata (il tempo di vita).

### 11.9.1. Regole di visibilità e struttura a blocchi

- **Dichiarazione prima dell'uso:** una variabile deve essere dichiarata prima di essere usata. Ciò consente alla symbol table di essere costruita in fase lessicale e di generare errori durante la fase di parsing.
- **Regola del blocco annidato più vicino:** date diverse dichiarazioni con lo stesso nome la dichiarazione valida è quella del blocco più interno.

## 11.9.2. Risoluzione del problema

Esistono due approcci per tener conto delle regole di visibilità nella TS:

- avere una TS per ogni ambiente o blocco;
- avere una TS globale.

### 11.9.2.1. Una TS per ogni ambiente

E' probabilmente più semplice pensare ad una TS per ogni ambiente.

Infatti quando la compilazione di un determinato blocco è finita, la TS contenente l'ambiente locale, può essere cancellata.

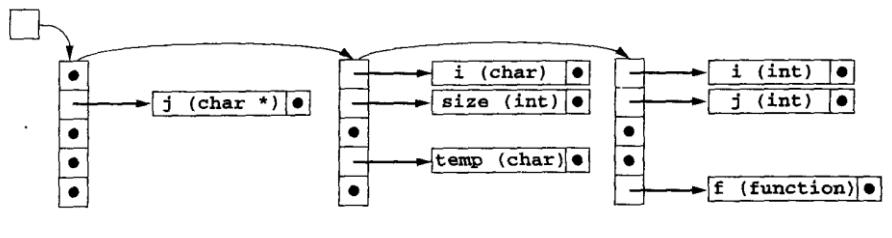
#### 11.9.2.1.1. La pila degli ambienti

Ciò suggerisce di utilizzare una "pila degli ambienti": ogni volta che si definisce un nuovo ambiente una nuova TS è aggiunta nella pila.

Quando il compilatore termina con tale ambiente, la TS relativa può essere eliminata.

L'uso della pila assicura che vengano rispettate le regole di visibilità.

```
int i,j;
int f (int size)
{ char i; temp;
  ...
  { char *j;
    ...
  }
}
```



#### 11.9.2.1.2. Problemi con la pila degli ambienti

- Nei compilatori a più passate si potrebbe avere la necessità di rivisitare gli ambienti già cancellati;
- Alcuni compilatori creano la "cross-reference table" che contiene per ogni variabile dove è definita e dove è referenziata. Tale tabella è ovviamente costruita alla fine della compilazione e alcune informazioni potrebbero non essere più disponibili. La soluzione potrebbe essere quella di salvare opportunamente gli ambienti cancellati (per es. in un file temporaneo) ;
- Se sono presenti più TS abbiamo bisogno di più hash table normalmente tutte della stessa dimensione: può capitare, quindi, che per alcuni ambienti con pochi identificatori lo spazio di memoria inutilizzato sia veramente tanto.

### 11.9.2.2. Una TS globale

- In questo caso bisogna associare all'identificatore un'informazione relativa all'ambiente a cui appartiene.
- Ciò può essere fatto associando ad ogni ambiente un numero che verrà incrementato ogni volta che inizia un nuovo ambiente.
- Ogni nuovo identificatore sarà inserito all'inizio della lista in modo tale che in cima ci saranno gli identificatori dell'ultimo ambiente analizzato e ciò in accordo con le regole di visibilità.
- La cancellazione degli ambienti già analizzati non presenta particolari difficoltà, fermo restando le osservazioni fatte in precedenza.

## 11.10. Osservazioni sul contenuto della TS

Per ogni elemento:

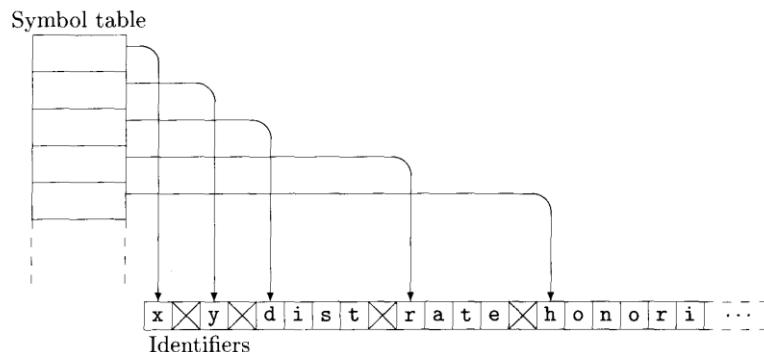
- nome dell'identificatore
- elenco degli attributi

### 11.10.1. Nome dell'identificatore

- In alcuni linguaggi di programmazione la lunghezza degli identificatori è limitata.
- In questo caso è possibile memorizzare il nome dell'identificatore in un vettore di caratteri di lunghezza massima pari a quella consentita per la lunghezza degli identificatori stessi.
- Lo spreco di memoria risulta irrisonio se il limite per la lunghezza degli identificatori è ragionevole (per es. 8 caratteri).
- Tale soluzione non può essere accettata per quei linguaggi che ammettono identificatori di lunghezza arbitraria.

### 11.10.2. La soluzione

- I nomi degli identificatori vengono memorizzati, opportunamente delimitati, in un unico vettore di caratteri di grandi dimensione (detto "array dei nomi").
- Nella TS non sarà presente il nome dell'identificatore ma un puntatore all'array dei nomi.



## 12. Type checking (controllo dei tipi)

La TS, o meglio il suo contenuto, è molto importante per uno dei principali compiti della fase di analisi semantica: il **type checking**.

In effetti, con “Type checking” si fa riferimento a due attività separate ma strettamente correlate fra di loro (eseguite insieme):

- calcolo e mantenimento delle informazioni sui tipi dei dati;
- controllo che ogni parte del programma abbia un senso per le regole dei tipi ammessi nel linguaggio.

Le informazioni sui tipi dei dati possono essere **statiche** o **dinamiche**.

In alcuni linguaggi (LISP) la gestione dei tipi dei dati è interamente dinamica. In questo caso il **Type checking** verrà eseguito durante l'esecuzione.

In altri linguaggi (C, Pascal) le informazioni sono prevalentemente statiche. In questo caso il **Type checking** (type inference + type checking) verrà eseguito durante il processo di compilazione.

Le informazioni statiche sono anche utili per determinare lo spazio di memoria necessario per allocare le variabili.

Ovviamente noi ci occuperemo solo del **Type checking** statico.

### 12.1. Compatibilità o equivalenza di tipi

- Il Type checker controlla che in un'espressione gli operandi siano compatibili fra di loro e, se necessario, che il risultato sia compatibile con la variabile destinata a riceverlo.
  - Domanda: “quando due operandi sono compatibili?”
  - Non basta dire: “quando sono dello tipo”.
  - Infatti l'introduzione delle dichiarazioni di tipo nei moderni linguaggi di programmazione pone una serie di questioni che il progettista di compilatori (e non solo) deve tenere in considerazione.
  - La risposta sulla compatibilità dei tipi dipende infatti dalla nozione di compatibilità o equivalenza fra tipi che viene adottata.
  - Si considerano due diverse nozioni di equivalenza:
    - Equivalenza nominale
    - Equivalenza strutturale
- **Equivalenza nominale**: secondo questa nozione due variabili sono dello stesso tipo se e solo se appaiono nella stessa lista di una dichiarazione oppure se sono state dichiarate con lo stesso **<nometipo>** (sia esso predefinito o definito dall'utente).
- **Equivalenza strutturale**: secondo questa nozione due variabili sono dello stesso tipo se hanno la stessa struttura.

L'**equivalenza nominale** risulta molto più restrittiva dell'**equivalenza strutturale**: due tipi equivalenti nominalmente lo sono anche strutturalmente ma non è generalmente vero il contrario.

Solo da alcuni anni, per evitare differenti interpretazioni da parte dei compilatori si è ritenuto essenziale includere nelle descrizione di un linguaggio anche le regole di equivalenza dei tipi.

## 12.2. Dichiarazioni

- Supponiamo di poter dichiarare una sola variabile per volta.
- Una grammatica semplificata potrebbe essere la seguente:

$$\begin{aligned} D \rightarrow & T \text{ id; } D \mid \epsilon \\ T \rightarrow & B \text{ C } \mid \text{record } \{ \text{ } D \text{ } \} \\ B \rightarrow & \text{int } \mid \text{float} \\ C \rightarrow & \epsilon \mid \text{['num']} \text{ } C \end{aligned}$$

- Il tipo e l'indirizzo relativo di un certo identificatore sono memorizzati nella ST.

## 12.3. Regole per il type checking

Può assumere due forme:

- Sintesi: costruzione di un tipo di un'espressione a partire dal tipo delle sue sottoespressioni. Richiede che tutti i nomi siano stati dichiarati prima.
- Inferenza: determina il tipo di un costrutto di un linguaggio in base al modo in cui è usato.

## 12.4. Realizzazione di un type checker

- Faremo adesso un esempio di un semplice linguaggio per il quale descriveremo il Type checking in termini di azioni semantiche.
- Introdurremo i due attributi: (name, type)
- Faremo uso della TS.

### 12.4.1. Le regole della grammatica

```

program → var-decls ; stmts
var-decls → var-decls ; var-decl | var-decl
var-decl → id : type-exp
type-exp → int | bool
stmts → stmts ; stmt | stmt
stmt → if exp then stmt | id := exp
exp → exp + exp | exp or exp
exp → true | false | id

```

Grammar Rule	Semantic Rules
<b>var-decl → id : type-exp</b>	insert( <b>id.name</b> , type-exp.type)
<b>type-exp → int</b>	type-exp.type:=integer
<b>type-exp → bool</b>	type-exp.type:=boolean
<b>stmt → if exp then stmt</b>	<b>if not</b> typeEqual(exp.type, boolean) <b>then</b> type-error(stmt)
<b>stmt → id := exp</b>	<b>if not</b> typeEqual( <b>lookup(id.name)</b> , exp.type) <b>then</b> type-error(stmt)
<b>Exp<sub>1</sub> → Exp<sub>2</sub> + Exp<sub>3</sub></b>	<b>if not</b> (typeEqual(exp <sub>2</sub> .type, integer) <b>and</b> typeEqual(exp <sub>3</sub> .type, integer)) <b>then</b> type-error(exp <sub>1</sub> ); exp <sub>2</sub> .type:=integer
<b>Exp<sub>1</sub> → Exp<sub>2</sub> or Exp<sub>3</sub></b>	<b>if not</b> (typeEqual(exp <sub>2</sub> .type, boolean) <b>and</b> typeEqual(exp <sub>3</sub> .type, boolean)) <b>then</b> type-error(exp <sub>1</sub> ); exp <sub>2</sub> .type:=integer
<b>exp → true</b>	exp.type:= boolean
<b>exp → false</b>	exp.type:= boolean
<b>exp → id</b>	exp.type:= <b>lookup(id.name)</b>

#### 12.4.2. Type checking delle dichiarazioni

$\text{var-decl} \rightarrow \text{id} : \text{type-exp}$	$\text{insert}(\text{id.name}, \text{type-exp.type})$
$\text{type-exp} \rightarrow \text{int}$	$\text{type-exp.type} := \text{integer}$
$\text{type-exp} \rightarrow \text{bool}$	$\text{type-exp.type} := \text{boolean}$

La regola semantica

$\text{insert}(\text{id.name}, \text{type-exp.type})$

associa all'identificatore inserito nella TS il suo tipo.

#### 12.4.3. Type checking degli statement

Gli statement non hanno un loro tipo ma bisogna sempre verificarne la correttezza.

$\text{stmt} \rightarrow \text{if exp then stmt}$	$\text{if not typeEqual(exp.type, boolean) then type-error(stmt)}$
---	--

Nel caso del costrutto **if** si richiede che l'espressione condizionale sia di tipo boolean.

La funzione **typeEqual** stabilisce se i tipi rappresentati dai suoi due parametri sono equivalenti (per nome).

La funzione **type-error** segnalerà opportunamente un errore in funzione dello statement esaminato.

Nel caso dell'assegnazione si richiede che il risultato dell'espressione sia di tipo compatibile con quello della variabile assegnata.

$\text{stmt} \rightarrow \text{id} := \text{exp}$	$\text{if not typeEqual}(\text{lookup}(\text{id.name}), \text{exp.type}) \text{ then type-error(stmt)}$
---	---

La funzione **lookup** restituisce il tipo associato all'identificatore passato come parametro.

#### 12.4.4. Type checking delle espressioni

$\text{Exp}_1 \rightarrow \text{Exp}_2 + \text{Exp}_3$	$\text{if not (typeEqual}(\text{exp}_2.\text{type, integer}) \text{ and typeEqual}(\text{exp}_3.\text{type, integer})$ $\text{then type-error(exp}_1\text{);}$ $\text{exp}_2.\text{type} := \text{integer}$
$\text{Exp}_1 \rightarrow \text{Exp}_2 \text{ or Exp}_3$	$\text{if not (typeEqual}(\text{exp}_2.\text{type, boolean}) \text{ and typeEqual}(\text{exp}_3.\text{type, boolean})$ $\text{then type-error(exp}_1\text{);}$ $\text{exp}_2.\text{type} := \text{integer}$
$\text{exp} \rightarrow \text{true}$	$\text{exp.type} := \text{boolean}$
$\text{exp} \rightarrow \text{false}$	$\text{exp.type} := \text{boolean}$
$\text{exp} \rightarrow \text{id}$	$\text{exp.type} := \text{lookup}(\text{id.name})$

#### 12.4.5. Type coercion (conversione automatica di tipo)

In alcuni linguaggi di programmazione è possibile che alcune espressioni, che coinvolgono operandi di tipo diverso, vengano valutate correttamente dopo opportune conversioni (ogni linguaggio specifica opportune regole di conversione).

Le conversioni possono riguardare sia le espressioni che le assegnazioni.

Le conversioni possono essere:

- implicite: effettuate dal type checker automaticamente.
- esplicite: è il programmatore che forza la conversione (cast).

#### 12.4.6. Type coercion delle espressioni

Consideriamo le seguenti regole:

- $\text{exp}_1 \rightarrow \text{exp}_2 + \text{exp}_3$
- $\text{type-exp} \rightarrow \text{int}$
- $\text{type-exp} \rightarrow \text{real}$

L'azione semantica che esegue la conversione di tipo può essere:

```
if typeEqual(exp2.type, integer) and typeEqual(exp3.type, integer) then exp1.type:=integer
else
  if typeEqual(exp2.type, integer) and typeEqual(exp3.type, real) then exp1.type:=real
  else
    if typeEqual(exp2.type, real) and typeEqual(exp3.type, integer) then exp1.type:=real
    else
      if typeEqual(exp2.type, real) and typeEqual(exp3.type, real) then exp1.type:=real
      else type-error(exp1)
```

Usa due funzioni:

- Max(t<sub>1</sub>, t<sub>2</sub>) restituisce il maggiore dei due tipi t<sub>1</sub> e t<sub>2</sub> secondo la gerarchia di promozione.
- Byte \_\_\_\_ Short \_\_\_\_ int \_\_\_\_ long \_\_\_\_ float \_\_\_\_ double  
Char \_\_\_\_ /
- Widen(a,t,w) genera la conversione di tipo necessaria per promuovere un indirizzo di tipo t in uno di tipo w

#### 12.5. Overloading

Un operatore (o una funzione) è overloaded se lo stesso nome di operatore ha diverso significato a seconda del contesto.

Es.1            2+3 (somma tra interi)  
                2.1 +3.0 (somma tra reali)  
Es.2            int max(int x,y);  
                float max (float x,y);

Un modo per eliminare l'ambiguità consiste nell'effettuare le operazioni di lookup nella TS controllando anche la lista dei parametri.

#### 12.6. Polimorfismo

Una funzione è polimorfa se può avere parametri di qualsiasi tipo.

Procedure swap(var x,y:anytype);

- A differenza dell'overloading in cui ci sono funzioni diverse con lo stesso nome, nel polimorfismo una stessa funzione può essere applicata a variabili di più tipi.
- Esistono particolari type checker che, sfruttando particolari e sofisticate tecniche di pattern matching risolvono il problema.

#### 12.7. Equivalenza strutturale

L'unificazione è il problema di determinare se due espressioni s e t possono essere rese identiche sostituendo alle variabili di s e t nuove espressioni.

Si risolve attraverso algoritmi su grafi.

La verifica dell'equivalenza strutturale è un particolare problema di unificazione.

## 13. Generazione del Codice intermedio

Il punto della situazione:

- Processo di compilazione:
  - Analisi (lessicale, sintattica, semantica)
  - Sintesi (generazione del codice oggetto)
- La generazione del codice oggetto è la fase più complessa di un compilatore perché dipende:
  - dalle caratteristiche del linguaggio sorgente
  - dalle caratteristiche della **target machine** (architettura, sistema operativo, ..)
- La generazione del codice può prevedere, inoltre, una fase di ottimizzazione che può dipendere anche da particolari caratteristiche della **target machine** come registri, modelli di indirizzamento, memoria, ....

Per questo motivo prima di procedere alla fase di generazione del codice oggetto, nella maggior parte dei compilatori è prevista la **generazione di un codice intermedio** che sarà poi utilizzato dal back-end per generare il codice oggetto.

Questa fase farà parte del front-end del compilatore.

Vantaggi: l'introduzione del codice intermedio permette di ottenere notevoli benefici:

- **Indipendenza del front-end dal back-end**

Nella rappresentazione intermedia non si fa alcuna ipotesi sulla target machine: è possibile quindi definire un back-end per ogni architettura destinataria (re-targeting)

- **Maggiore semplicità di traduzione nel codice oggetto**

La rappresentazione intermedia presenta, nella maggior parte dei casi, affinità, in termini di istruzioni, con il linguaggio assembler (che è spesso il linguaggio utilizzato per produrre il codice oggetto)

- **Maggiore semplicità per le ottimizzazioni**

Gli algoritmi di ottimizzazione risultano più semplici se operanti su sequenze di istruzioni piuttosto che su strutture più complesse come gli alberi sintattici. Ciò consente di produrre codice più efficiente.

La struttura di un compilatore

**Fasi della compilazione**

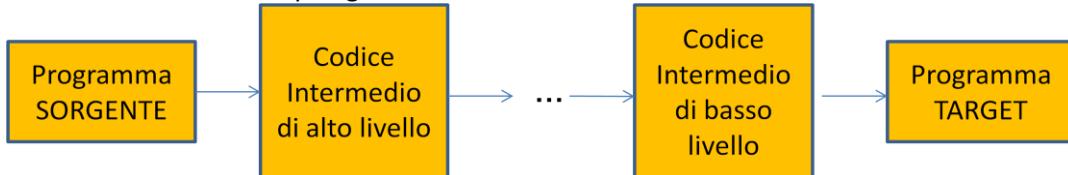
- |   |                        |
|---|------------------------|
| <input type="checkbox"/> Analisi lessicale                        | Analisi<br>(front-end) |
| <input type="checkbox"/> Analisi sintattica                       |                        |
| <input type="checkbox"/> Analisi semantica                        | Sintesi<br>(back-end)  |
| <input type="checkbox"/> <b>Generazione del codice intermedio</b> |                        |
| <input type="checkbox"/> Ottimizzazione                           |                        |
| <input type="checkbox"/> Generazione del codice oggetto           |                        |

### 13.1. Forma del codice intermedio

- Esistono diversi modelli usati per la generazione del codice intermedio.
- Tutti, comunque, si basano sul principio della linearizzazione degli alberi sintattici.
- Il codice intermedio può essere di alto livello o essere più vicino al codice oggetto.
- Può incorporare o no le informazioni contenute nella tabella dei simboli (regole di visibilità, livelli di annidamento,...). Se non lo fa, tali informazioni dovranno essere gestite in fase di generazione del codice oggetto.
- Uno dei più importanti ed usati è il codice a tre indirizzi: **three-address-code (3AC)**

## 13.2. Più codici intermedi

Possono esistere vari step di generazione del codice intermedio

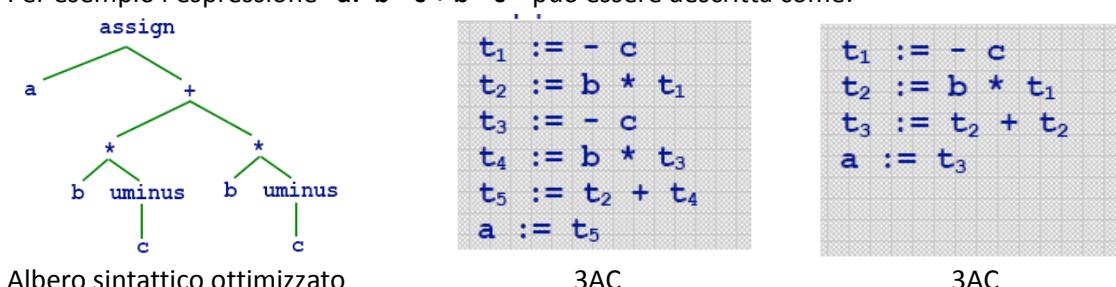


## 13.3. Codice a tre indirizzi

- Il nome “codice a tre indirizzi” deriva dal fatto che di solito ogni istruzione contiene tre indirizzi: due per gli operandi e uno per il risultato.
- Il codice a tre indirizzi è una sequenza di istruzioni generalmente del tipo:  
 $\text{id} := \text{id}_1 \operatorname{operator} \text{id}_2$   
dove  $\text{id}$ ,  $\text{id}_1$  e  $\text{id}_2$  sono nomi, costanti (tranne  $\text{id}$ ), o nomi temporanei generati dal compilatore, ed **operator** è un operatore.
- Ad esempio l'espressione  $x + y * z$  potrebbe essere tradotta:  
 $t_1 := y * z$   
 $t_2 := x + t_1$   
dove  $t_1$  e  $t_2$  sono nomi temporanei generati dal compilatore.

Il codice a tre indirizzi consente di linearizzare le rappresentazioni ad albero sintattico.

Per esempio l'espressione  $a := b * -c + b * -c$  può essere descritta come:

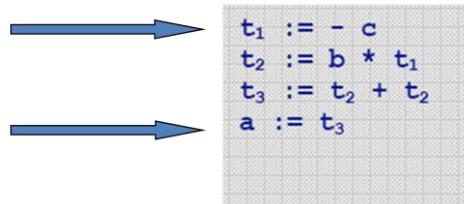


Ovviamente la forma dell'istruzione:

$\text{id} := \text{id}_1 \operatorname{operator} \text{id}_2$

è insufficiente per rappresentare tutte le caratteristiche di un linguaggio di programmazione.

Per esempio gli operatori unari, come la negazione o l'assegnazione, richiedono una variante del 3AC che contiene due soli indirizzi.



### 13.3.1. Non esiste un 3AC standard

- Per rappresentare, quindi, in 3AC tutti i costrutti di un linguaggio di programmazione sarà necessario adattare la forma del 3AC per ogni costrutto.
- Se il linguaggio ha caratteristiche insolite, bisognerà “inventarsi” nuove forme di 3AC.
- Questa è una delle ragioni per cui non esiste una forma standard per il 3AC.

## 13.4. Indirizzi e istruzioni

Un indirizzo può essere:

- un nome;  
nell'implementazione tale nome è sostituito da un puntatore alla TS;
- una costante;  
può essere di vario tipo; il compilatore potrebbe operare opportune conversioni di tipo;
- indirizzo temporaneo;  
è utile per le ottimizzazioni usare variabili temporanee con nomi diversi

Ogni istruzione può essere contrassegnata da una label simbolica. Le istruzioni possono essere:

- Assegnazioni della forma **x=y op z**, dove op è un'operazione logica o aritmetica;
- Assegnazioni della forma **x=op y**, dove op è un'operazione unaria;
- Istruzioni di copia **x=y**;
- Salto incondizionato **goto L**, dove L è la label di un'istruzione;
- Salti condizionati della forma **if x goto L** o **ifFalse x goto L**;
- Salti condizionati della forma **if x relop y goto L**;

- Chiamate di procedure: **param x** per i parametri, **call p,n** per le procedure, **y=call p,n** per le funzioni, **return y** è opzionale;

```
param x1
param x2
...
param xn
call p,n
```

genera una chiamata della procedura p(x<sub>1</sub>,x<sub>2</sub>,...,x<sub>n</sub>)

- Copie indicizzate della forma **x=y[i]** e **x[i]=y**; (la prima pone x al valore della locazione che si trova i unità dopo y, la seconda pone uguale a y il contenuto della locazione che si trova i unità dopo x.)
- Assegnazioni di indirizzi della forma **x=&y** (pone il valore di x uguale alla locazione di y)
- Assegnazioni di puntatori della forma **\*x=y** e **x=\*&y**;

Esempio

```
do i=i+1;
while (a[i]<v);
```

L:  $t_1 = i + 1$   
 $i = t_1$   
 $t_2 = i * 8$   
 $t_3 = a[t_2]$   
if  $t_3 < v$  goto L

ETICHETTE SIMBOLICHE

100:  $t_1 = i + 1$   
101:  $i = t_1$   
102:  $t_2 = i * 8$   
103:  $t_3 = a[t_2]$   
104: if  $t_3 < v$  goto 100

INDICI DI POSIZIONE

La dimensione di ogni elemento dell'array ha dimensione pari a 8 unità

### 13.5. Esempio in linguaggio SimplePas

#### Programma per il calcolo del fattoriale

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 0
    fact := 1;
    repeat
        fact := fact * x;
        x := x - 1
    until x = 0;
    write fact { output factorial of x }
end
```

#### Tree address code

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

#### Esempio di codice 3AC

```
read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt
```

**if\_false**: istruzione di salto condizionale a 2 indirizzi  
(usata per tradurre sia il costrutto if che il repeat)

**label**: istruzione a un indirizzo, segnano la posizione di indirizzi di salto;

**halt**: istruzione senza indirizzi che serve per marcare

### 13.6. Strutture dati per l'implementazione del 3AC

Il 3AC, normalmente, non è implementato in forma testuale ma:

- ogni istruzione in 3AC è realizzata attraverso un record con campi per gli operatori ed operandi;
- la sequenza delle istruzioni in 3AC è implementata attraverso un vettore o una lista concatenata.

### 13.7. Implementazione delle istruzioni in 3AC

- Le istruzioni vengono implementate con record con 4 campi (uno per l'operatore e tre per gli indirizzi degli operandi).
- Questi record vengono chiamati **quadruple**.
- Per quelle istruzioni che hanno bisogno di meno indirizzi i campi relativi saranno lasciati “vuoti”.

### 13.8. Esempio

Consideriamo l'espressione:  $a := b^* - c + b^* - c$

```

 $t_1 := -c$ 
 $t_2 := b * t_1$ 
 $t_3 := -c$ 
 $t_4 := b * t_3$ 
 $t_5 := t_2 + t_4$ 
 $a := t_5$ 

```

Codice 3AC

	op	arg1	arg2	result
(0)	uminus	c		$t_1$
(1)	*	b	$t_1$	$t_2$
(2)	uminus	c		$t_3$
(3)	*	b	$t_3$	$t_4$
(4)	+	$t_2$	$t_4$	$t_5$
(5)	$:=$	$t_5$		a

Quadruple

Tree address code

```

read x
t1 = x > 0
if_false t1 goto L1
fact = 1
label L2
t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
write fact
label L1
halt

```

Quadruple

```

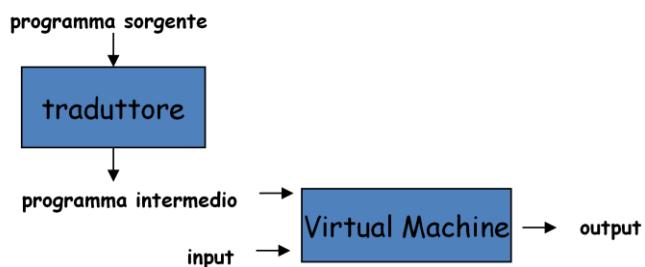
(rd,x,_,_)
(gt,x,0,t1)
(if_f,t1,L1,_)
(asn,1,fact,_)
(lab,L2,_,_)
(mul,fact,x,t2)
(asn,t2,fact,_)
(sub,x,1,t3)
(asn,t3,x,_)
(eq,x,0,t4)
(if_f,t4,L2,_)
(wri,fact,_,_)
(lab,L1,_,_)
(halt,_,_,_)

```

### 13.9. Codice per macchina virtuale

Codice oggetto in forma intermedia (è considerato codice intermedio):

- **p-code** prodotto dal Pascal p-compiler per una **Virtual Stack Machine**
- **bytecode** prodotto dai compilatori Java per una **Virtual Java Machine**
- **Common Intermediate Language (CIL)** della piattaforma .NET eseguito dal **Common Language Runtime (CLR)**, la macchina virtuale .NET progettata da Microsoft per funzionare con qualsiasi sistema operativo.



### 13.9.1. Un esempio: il P-Code

Questo codice è stato progettato negli anni 70-80 per essere eseguito da una **Virtual Stack Machine (P-machine)**.

Contiene descrizioni e informazioni specifiche legate alla struttura della P-machine

Supponiamo che la **P-machine** sia costituita da:

- una memoria per il codice
- una memoria per le variabili
- uno stack per i dati temporanei
- una serie di registri per la gestione dello stack e il supporto dell'esecuzione.

Un esempio di P-Code:

Consideriamo l'espressione: **2\*a+(b-3)**

Una versione in P-code per questa espressione è la seguente:

```
ldc 2      ; load constant 2
lod a      ; load value of variable a
mpi        ; integer multiplication
lod b      ; load value of variable b
ldc 3      ; load constant 3
sbi        ; integer subtraction
adi        ; integer addition
```

Consideriamo l'assegnazione: **x := y+1**

che corrisponde alle seguenti istruzioni in P-code:

```
lda x      ; load address of x
lod y      ; load value of y
ldc 1      ; load constant 1
adi        ; add
sto        ; store top to address
            ; below top & pop both
```

### 13.9.1.1. P-code per il programma per il calcolo del fattoriale

```
read x; { input an integer }
if 0 < x then { don't compute if x <= 1
    fact := 1;
repeat
    fact := fact * x;
    x := x - 1
until x = 0;
write fact { output factorial of x }
end
```

```
lda x      ; load address of x
rdi        ; read an integer, store to
           ; address on top of stack (& pop it)
lod x      ; load the value of x
ldc 0      ; load constant 0
grt        ; pop and compare top two values
           ; push Boolean result
fjp L1    ; pop Boolean value, jump to L1 if false
lda fact   ; load address of fact
ldc 1      ; load constant 1
sto        ; pop two values, storing first to
           ; address represented by second
lab L2    ; definition of label L2
lda fact   ; load address of fact
lod fact   ; load value of fact
lod x      ; load value of x
mpi        ; multiply
sto        ; store top to address of second & pop
lda x      ; load address of x
lod x      ; load value of x
ldc 1      ; load constant 1
sbi        ; subtract
sto        ; store (as before)
lod x      ; load value of x
ldc 0      ; load constant 0
equ        ; test for equality
fjp L2    ; jump to L2 if false
lod fact   ; load value of fact
wri        ; write top of stack & pop
lab L1    ; definition of label L1
stp
```

### 13.9.2. Confronto tra P-code e 3AC

- Il P-code in alcuni aspetti è più vicino, rispetto al 3AC, agli attuali linguaggi assemblativi.
- Le istruzioni in P-code richiedono meno indirizzi (uno o zero indirizzi).
- P-code è meno compatto del 3AC in termini di numero di istruzioni.
- P-code non è “self-contained” in quanto le istruzioni operano implicitamente sullo stack (e questi indirizzi impliciti sullo stack sono proprio gli indirizzi che mancano nelle istruzioni).
- Il vantaggio dell’uso dello stack sta nel fatto che il compilatore non ha bisogno di assegnare nomi ai valori che necessitano in ogni punto del codice così come accade nelle istruzioni in 3AC.
- Si possono usare strutture dati analoghe a quelle usate per il 3AD

### 13.10. Tecniche per la generazione del codice intermedio

- La generazione del codice intermedio (o direttamente il target code senza codice intermedio) può essere vista come il calcolo di un attributo.
- Infatti, se il codice generato è visto come un attributo stringa, allora questo codice diventa un attributo sintetizzato che può essere definito usando una grammatica con attributi.
- Il codice intermedio potrà quindi essere generato direttamente durante il parsing.

### 13.11. Esempio: traduzione delle espressioni

Consideriamo la seguente grammatica che rappresenta un piccolo sottoinsieme delle espressioni in C.

$$\begin{aligned} \text{exp} &\rightarrow \text{id} = \text{exp} \mid \text{aexp} \\ \text{aexp} &\rightarrow \text{aexp} + \text{factor} \mid \text{factor} \\ \text{factor} &\rightarrow (\text{exp}) \mid \text{num} \mid \text{id} \end{aligned}$$

Questa grammatica contiene solo due operazioni: l'assegnazione (=) e l'addizione (+).

Il token **id** rappresenta un identificatore e il token **num** rappresenta un numero intero.

Entrambi i token hanno un attributo **strval**, di tipo stringa che rappresenta il lesema (il nome nel caso di **id** e il valore numerico nel caso di **num**).

### 13.12. Grammatica con attributi per la generazione di P-code

Grammar Rule	Semantic Rules
$\text{exp}_1 \rightarrow \text{id} = \text{exp}_2$	$\text{exp}_1.\text{pcode} = "1da" \parallel \text{id}.\text{strval}$ ++ $\text{exp}_2.\text{pcode}$ ++ "stn"
$\text{exp} \rightarrow \text{aexp}$	$\text{exp}.\text{pcode} = \text{aexp}.\text{pcode}$
$\text{aexp}_1 \rightarrow \text{aexp}_2 + \text{factor}$	$\text{aexp}_1.\text{pcode} = \text{aexp}_2.\text{pcode}$ ++ $\text{factor}.\text{pcode}$ ++ "adi"
$\text{aexp} \rightarrow \text{factor}$	$\text{aexp}.\text{pcode} = \text{factor}.\text{pcode}$
$\text{factor} \rightarrow (\text{exp})$	$\text{factor}.\text{pcode} = \text{exp}.\text{pcode}$
$\text{factor} \rightarrow \text{num}$	$\text{factor}.\text{pcode} = "1dc" \parallel \text{num}.\text{strval}$
$\text{factor} \rightarrow \text{id}$	$\text{factor}.\text{pcode} = "1od" \parallel \text{id}.\text{strval}$

- L'attributo **pcode** conterrà l'istruzione in P-code
- Il simbolo **++** concatena due stringhe inserendo un newline tra di loro
- Il simbolo **||** concatena due stringhe inserendo tra di loro uno spazio

Esempio

Consideriamo l'espressione: **(x=x+3)+4**

Grammar Rule	Semantic Rules	P-code
$\text{exp}_1 \rightarrow \text{id} = \text{exp}_2$	$\text{exp}_1.\text{pcode} = "1da" \parallel \text{id}.\text{strval}$ ++ $\text{exp}_2.\text{pcode}$ ++ "stn"	<b>1da x</b>
$\text{exp} \rightarrow \text{aexp}$	$\text{exp}.\text{pcode} = \text{aexp}.\text{pcode}$	<b>1od x</b>
$\text{aexp}_1 \rightarrow \text{aexp}_2 + \text{factor}$	$\text{aexp}_1.\text{pcode} = \text{aexp}_2.\text{pcode}$ ++ $\text{factor}.\text{pcode}$ ++ "adi"	<b>1dc 3</b>
$\text{aexp} \rightarrow \text{factor}$	$\text{aexp}.\text{pcode} = \text{factor}.\text{pcode}$	<b>adi</b>
$\text{factor} \rightarrow (\text{exp})$	$\text{factor}.\text{pcode} = \text{exp}.\text{pcode}$	<b>stn</b>
$\text{factor} \rightarrow \text{num}$	$\text{factor}.\text{pcode} = "1dc" \parallel \text{num}.\text{strval}$	<b>1dc 4</b>
$\text{factor} \rightarrow \text{id}$	$\text{factor}.\text{pcode} = "1od" \parallel \text{id}.\text{strval}$	<b>adi</b>

### 13.13. Grammatica con attributi per la generazione di 3AC

Grammar Rule	Semantic Rules
$exp_1 \rightarrow id = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++$ $id.strval \parallel "=" \parallel exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode ++ factor.tacode$ $+ + aexp_1.name \parallel "=" \parallel aexp_2.name$ $\parallel "+" \parallel factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow ( exp )$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow num$	$factor.name = num.strval$ $factor.tacode = ""$
$factor \rightarrow id$	$factor.name = id.strval$ $factor.tacode = ""$

- L'attributo *tacode* conterrà l'istruzione in 3AC

- La funzione *newtemp()* genera nomi temporanei iniziando con t<sub>1</sub>

- L'attributo *name* tiene conto di questo nome

#### Esempio

Consideriamo l'espressione: (x=x+3)+4

Grammar Rule	Semantic Rules
$exp_1 \rightarrow id = exp_2$	$exp_1.name = exp_2.name$ $exp_1.tacode = exp_2.tacode ++$ $id.strval \parallel "=" \parallel exp_2.name$
$exp \rightarrow aexp$	$exp.name = aexp.name$ $exp.tacode = aexp.tacode$
$aexp_1 \rightarrow aexp_2 + factor$	$aexp_1.name = newtemp()$ $aexp_1.tacode =$ $aexp_2.tacode ++ factor.tacode$ $+ + aexp_1.name \parallel "=" \parallel aexp_2.name$ $\parallel "+" \parallel factor.name$
$aexp \rightarrow factor$	$aexp.name = factor.name$ $aexp.tacode = factor.tacode$
$factor \rightarrow ( exp )$	$factor.name = exp.name$ $factor.tacode = exp.tacode$
$factor \rightarrow num$	$factor.name = num.strval$ $factor.tacode = ""$
$factor \rightarrow id$	$factor.name = id.strval$ $factor.tacode = ""$

#### 3AC

**t1 = x + 3**

**x = t1**

**t2 = t1 + 4**

### 13.14. Considerazioni

- La generazione del codice intermedio come calcolo di attributi sintetizzati è semplice e vantaggioso per comprendere i diversi metodi per la generazione del codice.
- Di fatto non è una tecnica molto usata nei moderni compilatori per diverse ragioni:
  - la concatenazione di stringhe è un'operazione che ha un notevole peso computazionale e che richiede una notevole quantità di memoria;
  - nel caso di grammatiche con attributi ereditati la generazione del codice è più complessa.
- Ecco perché spesso la generazione del codice intermedio avviene operando, con appositi moduli, direttamente sull'albero sintattico.

### 13.15. Criteri generali per il C.I.

#### 13.15.1. if then

Gli operatori booleani si traducono in salti.

```
if (x<100 || x>20 && x!=y) x=0;
```



```
if x<100 goto L2  
ifFalse x>20 goto L1  
ifFalse x!=y goto L1  
L2: x=0  
L1:
```

Il compilatore può ottimizzare la valutazione, calcolando la porzione minima necessaria a stabilire se tutta l'espressione sia vera o falsa.

#### 13.15.2. if then else

Gli operatori booleani si traducono in salti.

```
if (x<100) x=0;  
else x=1;
```



```
if x<100 goto L2  
ifFalse x<100 goto L1  
L2: x=0  
     goto L3  
L1: x=1  
L3:
```

#### 13.15.3. while

Gli operatori booleani si traducono in salti.

```
while (x<100) x=x+1;
```



```
L3: if x<100 goto L2  
     ifFalse x<100 goto L1  
L2: t1=x+1  
     x=t1  
     goto L3  
L1:
```

#### 13.15.4. assegnazione di valori booleani

```
x=a<b && c<d;
```



```
ifFalse a<b goto L1  
ifFalse c<d goto L1  
t=true  
goto L2  
L1: t=false  
L2: x=t
```

### 13.15.5. switch

Switch statements

```
switch (E) {
    case V1:S1
    case V2:S2
    ...
    case Vn-1:Sn-1
    default: Sn
}
```

```
valuta E in t
      goto test
L1:codice per S1
      goto next
L2:codice per S2
      goto next
      ...
Ln:codice per Sn
test:if t=V1 goto L1
      if t=V2 goto L2
      ...
      goto Ln
next:
}
```

### 13.15.6. chiamata di funzione

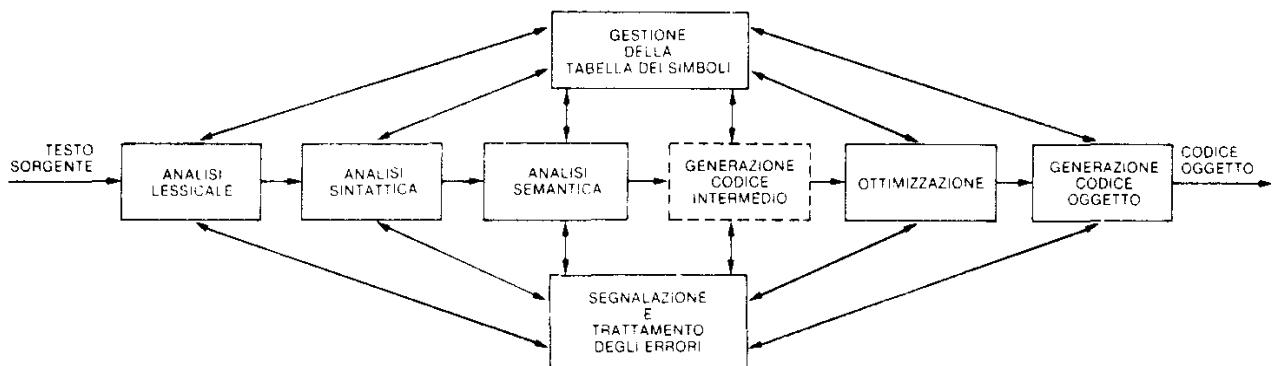
n=f(a)



```
t1=a
param t1
t2=call f,1
n=t2
```

### 13.16. Fase di sintesi

In generale, a questo punto è possibile passare alla fase della generazione del codice oggetto (target code) che può essere direttamente in linguaggio macchina o, come accade spesso, in linguaggio assemblativo. In questa fase è possibile prevedere eventuali azioni di ottimizzazione del codice.



# 14. Generazione del codice oggetto

La struttura di un compilatore

## Fasi della compilazione

- |  |                        |
|--|------------------------|
| <input type="checkbox"/> Analisi lessicale                     | Analisi<br>(front-end) |
| <input type="checkbox"/> Analisi sintattica                    |                        |
| <input type="checkbox"/> Analisi semantica                     | Sintesi<br>(back-end)  |
| <input type="checkbox"/> Generazione del codice intermedio     |                        |
| <input type="checkbox"/> Ottimizzazione                        | Sintesi<br>(back-end)  |
| <input type="checkbox"/> <u>Generazione del codice oggetto</u> |                        |

## 14.1. Ottimizzazione del codice

- La traduzione di un programma in codice intermedio e la successiva generazione del codice, producono un codice oggetto solitamente inefficiente rispetto a quello che potrebbe scrivere direttamente nel linguaggio assemblativo un esperto programmatore.
- Ecco perché molti compilatori cercano di applicare delle trasformazioni migliorative, dette ottimizzazioni, allo scopo di rendere il codice prodotto più efficiente.
- Le trasformazioni possono operare a livello del codice intermedio (ottimizzazioni *indipendenti dalla macchina*).
- Esistono anche ottimizzazioni *dipendenti dalla macchina* operate dal *post-ottimizzatore* direttamente sul codice oggetto a valle del generatore.

Il problema di generare un codice target ottimo è indecidibile.

Molti problemi si incontrano in questa fase, come l'allocazione dei registri, sono computazionalmente intrattabili.

Ci si deve accontentare di tecniche euristiche che generano un buon codice non necessariamente ottimo.

## 14.2. Esempi di ottimizzazioni indipendenti dalla macchina

- **Propagazione dei valori (copy propagation)**
  - Sostituzione di una variabile con il più recente valore ad essa assegnato.
- **Ripiegamento delle costanti (constant folding)**
  - In un'espressione in cui gli argomenti sono costanti si sostituisce il risultato.
- **Riduzione dei salti**
  - Per esempio un salto ad un'altra istruzione di salto.
- **Eliminazione del codice morto**
  - Eliminazione di codice che non può mai essere eseguito perché irraggiungibile.
- **Eliminazione di sottoespressioni comuni**
  - Sostituzione di espressioni già calcolate.
- **Eliminazione degli assegnamenti ridondanti o inutili**
  - $x=x+0; x=x*1; x=x-0; \dots$
- **Estrazione del codice invariante da un ciclo**
- **Riduzione del costo delle operazioni**
  - Sostituzione di un'operazione costosa per mezzo di una più veloce.  
Per esempio  $x^3 \rightarrow x*x*x$      $2*x \rightarrow x+x$      $x/2 \rightarrow x * 0,5$
- **Espansione in linea delle chiamate di procedure o eliminazione delle ricorsioni in coda (dove l'ultima operazione è la chiamata a procedura).**

```

int gcd(int u, int v)      int gcd(int u, int v)
{if (v==0) return u;       {begin:
else return gcd(v,u%v);    if (v==0) return u;
                           else
                           {int t1=v, t2=u%v;
                           u=t1; v=t2;
                           goto begin;}}
}

```

### 14.3. Generatori di codice oggetto

Un generatore di codice ha 3 obiettivi principali:

- selezione delle istruzioni (scelta delle istruzioni in linguaggio macchina per implementare il C.I.)
- allocazione e assegnazione dei registri (stabilire quali valori mantenere in quali registri)
- ordine delle istruzioni (decidere in quale ordine devono essere eseguite le istruzioni)

### 14.4. Input per un generatore di codice oggetto

- L'input è il codice intermedio, unitamente alle informazioni contenute nella tabella dei simboli.
- Può essere il codice a tre indirizzi, rappresentato come quadruple o triple
- Può essere il codice per macchine virtuali, come il p-code o il bytecode, ...
- Rappresentazioni lineari di alberi
- ...

### 14.5. Il programma target

- È strettamente dipendente dall'architettura a livello dell'instruction set della macchina target.
- Architetture più comuni
  - RISC (reduced instruction set computer) – molti registri, istruzioni a 3 indirizzi, insieme semplice di istruzioni
  - CISC (complex instruction set computer) – pochi registri, istruzioni a due indirizzi, istruzioni complesse
  - a stack
- Spesso è espresso in linguaggio assembly.

### 14.6. Codice macchina assoluto e rilocabile

Un programma in codice macchina assoluto ha il vantaggio di poter essere caricato in memoria in un posizione fissa e immediatamente e rapidamente eseguito.

Un programma in codice rilocabile consente di compilare separatamente vari sottoprogrammi. Tali oggetti rilocabili devono poi essere composti insieme mediante il linking caricati in memoria mediante il loader. Ciò dà un vantaggio in termini di flessibilità e di riuso del codice.

### 14.7. Selezione delle istruzioni

- La complessità di questa attività è influenzata da:
  - Il livello di astrazione del codice intermedio
  - La natura dell'instruction set
  - La qualità del codice oggetto che si vuole ottenere (velocità e dimensione)

- Può essere modellizzato come un problema di matching tra gli alberi utilizzati per rappresentare il codice intermedio e le istruzioni del linguaggio target.
- Esistono algoritmi di programmazione dinamica per cercare sequenze di codice vicino all'ottimo
- Il problema di trovare il codice ottimo è indecidibile.

#### 14.8. Allocazione dei registri

- I registri sono gli elementi di memorizzazione più veloce, ma in genere non sono tanti.
  - Allocazione dei registri: si seleziona l'insieme delle variabili che risiederanno nei registri
  - Assegnazione dei registri: Si seleziona lo specifico registro per ogni variabile
- Il problema di assegnazione ottima è NP-completo
- Dipende dall'architettura

#### 14.9. Ordine di valutazione

La scelta di un ordine con cui eseguire le istruzioni può richiedere l'uso di un numero inferiore di registri.  
La scelta del miglior ordine è un problema NP-Completo.

#### 14.10. Linguaggio target

Consideriamo un modello di una semplice macchina target, che dispone di n registri di uso generale. Le tipiche istruzioni sono:

- Operazioni di caricamento o load  
LD dest, addr → carica in dest il valore contenuto in addr
- Operazioni di store  
ST x, r → salva in x il valore del registro r
- Operazioni di calcolo  
Op dest, src1, src2
- Salti incondizionati  
BR L → branch su etichetta L
- Salti condizionati  
Bcond r , L → branch su L se il valore del registro r soddisfa la condizione cond

#### 14.11. Indirizzi nel codice target

Ogni programma in esecuzione ha a disposizione uno spazio di indirizzamento logico partizionato in 4 aree per dati e codice:

- Code: definita staticamente, contiene il codice target eseguibile, la dimensione del codice target può essere calcolata a compile-time
- Static: definita staticamente, contiene le costanti globali e gli altri dati generati dal compilatore, la dimensione può essere calcolata a compile-time
- Heap: gestita dinamicamente, dedicata agli oggetti creati e distrutti durante l'esecuzione. La dimensione non è determinata a compile-time
- Stack: gestito dinamicamente, dedicato alla memorizzazione dei record d'attivazione corrispondenti alle chiamate delle procedure. La dimensione non è determinata a compile-time

## 14.12. Blocchi base

- Molto generatori di codice partizionano le istruzioni in C.I. in blocchi base che saranno sicuramente eseguito in sequenza
- Ogni blocco base è una sequenza di istruzioni a 3 indirizzi
- Il flusso di controllo può entrare nel blocco solo attraverso la prima istruzione.
- I blocchi base diventano nodi del diagramma di flusso.

### 14.12.1. Algoritmo di partizionamento in blocchi base

- Si individuano le istruzioni “leader” che costituiscono l’inizio di un blocco base
- La prima istruzione del C.I. è leader
- Ogni istruzione di destinazione di un salto incondizionato o condizionato è un leader
- Ogni istruzione immediatamente successiva a un salto è un leader

Esempio

```
→ read x
    t1 = x > 0
    if_false t1 goto L1
→ fact = 1
label L2
→ t2 = fact * x
fact = t2
t3 = x - 1
x = t3
t4 = x == 0
if_false t4 goto L2
→ write fact
label L1
→ halt
```

## 14.13. Algoritmo per il prossimo uso

- Conoscere quando il valore di una variabile sarà utilizzato la prossima volta è essenziale per generare un buon codice - il registro può essere usato per memorizzare altre variabili
- In un blocco base si considerano le istruzioni dall’ultima alla prima
- Per ogni istruzione i:x=y+z
  - Nella tabella dei simboli si aggiornano le info relative a x: non “viva” e prossimo uso “nessuno”
  - Si aggiornano le informazioni relative a y e z indicandole come vive e con prossimo uso in i

## 14.14. Diagramma di flusso

Una volta partizionato il C.I. in blocchi, questi diventano i nodi di un grafo (il diagramma di flusso)

#### 14.15. Semplice generazione del codice per un blocco base

Assumiamo che le istruzioni macchina abbiano la forma

- LD reg, mem
- ST mem, reg
- OP reg, reg, reg

Per ogni registro si ha un descrittore di registro che tiene traccia dei nomi delle variabili il cui nome si trova in quel momento nel registro.

Per ogni variabile si ha un descrittore di indirizzo che tiene traccia della locazione in cui si trova il valore di quella variabile.

#### 14.16. Istruzioni macchina per le operazioni e per la copia

Per ogni istruzione a tre indirizzi  $x=y+z$

- Si selezionano i registri per x, y, z. Siano Rx, Ry, Rz
- Se y non si trova già in Ry si genera l'istruzione LD Ry, y', dove y' indica una delle locazioni associate a y
- Se z non si trova già in Rz si genera l'istruzione LD Rz, z'
- Si genera l'istruzione ADD Rx,Ry,Rz

Per le istruzioni del tipo  $x=y$  si procede in modo analogo e si generano le istruzioni del tipo LD Ry,y

#### 14.17. Esempi di ottimizzazioni dipendenti dalla macchina

Le ottimizzazioni dipendenti dalla macchina possono essere suddivise in due classi:

- Ottimizzazioni relative ad una attenta selezione delle istruzioni e all'efficiente allocazione dei registri
- Ottimizzazioni relative all'uso di particolari istruzioni in linguaggio macchina (ottimizzazioni a feritoia o peephole optimization) :
  - Eliminazione operazioni load/store ridondanti
  - Eliminazione del codice irraggiungibile
  - Eliminazione di salti superflui
  - Semplificazioni algebriche ( $x=x+0$ )
  - Uso di modalità d'indirizzamento con auto-incremento o auto-decremento, nel caso che la macchina target lo consenta.