



DISCLAIMER

Il materiale contenuto nel drive è stato raccolto e richiesto tramite autorizzazione ai ragazzi frequentanti il corso di studi di Informatica dell'Università degli Studi di Salerno. Gli appunti e gli esercizi nascono da un uso e consumo degli autori che li hanno creati e risistemati per tanto non ci assumiamo la responsabilità di eventuali mancanze o difetti all'interno del materiale pubblicato.

Il materiale sarà modificato aggiungendo il logo dell'associazione, in tal caso questo possa recare problemi ad alcuni autori di materiale pubblicato, tale persona può contattarci in privato ed elimineremo o modificheremo il materiale in base alle sue preferenze.

Ringraziamo eventuali segnalazioni di errori così da poter modificare e fornire il miglior materiale possibile a supporto degli studenti.



CoScienze
Associazione

Compilatori



Indice	i
Elenco delle figure	xv
Elenco delle tabelle	xviii
1 Introduzione	1
1.1 Definizione di un compilatore	1
1.1.1 Il linguaggio Ada & altri linguaggi	1
1.2 Linguaggi interpretati e compilati	2
1.3 Tipologie e caratteristiche dei linguaggi	2
1.4 Supportare o simulare una caratteristica	3
1.5 Struttura del compilatore	3
1.5.1 Fase dell'analisi lessicale (lexical analyzer)	3
1.5.2 Fase dell'analizzatore sintattico (Syntax Analyzer)	4
1.5.3 Fase analisi semantica (Semantic Analyzer)	5
1.5.4 Fase dell'intermediate code generator	6
1.5.5 Fase del code optimizer	7
1.5.6 Fase del code generator	7
1.6 Tipologie di compilatori	7
1.6.1 Il front end	7
1.6.2 Back end	7
1.6.3 Compilatori monolitici	8

1.7	Compilatori multiplatforma e multilinguaggi	8
1.8	Codice intermedio LLVM IR	8
1.9	Infrastrutture ibride	9
1.10	Codice intermedio ed impatto sui compilatori	10
1.11	Regole e fasi del compilatore	10
1.11.1	Regole lessicali	10
1.11.2	Regole sintattiche	11
1.11.3	Regole semantiche	11
1.12	Fasi logiche di Clang	12
1.13	Language processing system	12
1.14	Appartenenza del codice intermedio	13
1.15	Compilatori e Java	13
1.16	Compilatori e C	13
1.17	Problemi di Java e bytecode	14
1.18	Just in Time Compiler	14
1.19	Generazione codice intermedio	15
2	Struttura ed attività dei compilatori	16
2.1	Attività dell'analisi lessicale	16
2.1.1	Tipologia dei token	17
2.2	Scelta dei sottoalberi e realizzazione degli AST	17
2.3	Tabella dei simboli e tabella delle stringhe	18
2.3.1	Identificazione dei tipi nell'analisi semantica	18
2.4	Fasi dell'analisi semantica	18
2.5	Dall'analisi semantica al codice intermedio	19
3	Ciclo di vita dei linguaggi	20
3.1	Fasi del ciclo di vita di un linguaggio	20
3.2	Attori del linguaggio	21
3.3	Concetto di binding	21
3.3.1	Esempi di bindings	22
3.4	Concetto di Binding time	23
3.5	Pro e contro dei bindings	23
3.6	Da linguaggio compilato ad interpretato	24

4	Analisi lessicale	25
4.1	Introduzione all'analisi lessicale	25
4.2	Fasi dell'analisi lessicale	26
4.3	Famiglie dell'analisi lessicale	26
4.4	L'importanza dei pattern	27
4.5	jflex() ed il suo utilizzo	27
4.6	Famiglie e pattern	28
4.6.1	Famiglie: Parole chiave, operatori e simboli di punteggiatura	28
4.6.2	Famiglia: identificatori	28
4.6.3	Famiglia: costanti	28
4.7	Lessemi e operazioni sui lessemi	29
4.8	Espressioni regolari	29
4.9	Simboli dell'espressione regolare	29
4.10	Espressioni regolari ed altre notazioni	30
4.11	Differenza tra analizzatore e riconoscitore lessicale	30
4.12	Riconoscimento delle sottostringhe	31
4.13	Il tool Regex	31
4.14	Scorrere la stringa: forward e begin	31
4.15	Regole e tabella dei simboli	32
4.16	Compilatori bloccanti e non bloccanti	32
4.17	Espressioni regolari alternative	32
4.18	Grammatica dell'analisi sintattica	33
4.19	Funzionamento del Lexer	34
4.20	Diagrammi di transizione	34
4.20.1	Esempio: Diagramma di transizione per relop	35
4.20.2	Esempio: diagramma di transizione per gli id o parole chiave	36
4.20.3	Esempio: Diagramma di transizione di number	36
4.21	Passaggio tra automi	37
4.22	Diagrammi di transizione e gestione degli errori	38
4.23	Tecniche di distinzione degli identificatori	38
5	Jflex e Lex	39
5.1	Costruzione dell'analizzatore lessicale	39
5.2	Costruzione dell'analizzatore sintattico	39

5.3	Javacup e JFlex	40
5.4	Costruzione del Lexer	40
5.5	Overview di Lex, il precursore di JFlex	41
5.6	Sezione delle espressioni regolari di Lex	41
5.7	Sezione delle regole di Lex	41
5.8	Jflex e suddivisione in sezioni: Prima sezione di JFlex	42
5.9	Seconda sezione di JFlex	42
5.9.1	Definizioni regolari di JFlex	43
5.9.2	Tipi di commenti in Java	44
5.9.3	Identificatori e digit	44
5.10	Sezione delle regole e conflitti con JFlex	45
5.10.1	Stati ed appartenenza	45
5.10.2	Passaggio di stati	45
5.10.3	Gruppo YYINITIAL	45
5.10.4	Gruppo STRING	46
5.10.5	Stato di default	47
5.11	JFlex ed API	47
6	Analisi sintattica	48
6.1	Introduzione all'analisi sintattica	48
6.2	Grammatiche ed attributi	49
6.3	Albero di derivazione ed AST	49
6.4	Grammatiche context free	49
6.5	Concetti delle grammatiche	50
6.6	Forma sentenziale	50
6.7	Terminologia della grammatica	50
6.8	Linguaggio della grammatica	51
6.9	Differenza tra espressione e grammatica	52
6.9.1	Dimostrazione della grammatica	52
6.10	Albero di derivazione	53
6.11	Grammatiche differenti	53
6.12	Derivazione e non terminali	54
6.12.1	Non terminali multipli e alberi di derivazione	54
6.13	Breve riassunto sulla grammatica	54

6.14	Esempio delle grammatiche nell'ambito dei compilatori	55
6.14.1	Esempio: Partire dalla sentenza accettata	55
6.14.2	Esempio: Creare la grammatica che accetta la sentenza	55
6.14.3	Esempio: Sostituzioni e derivazioni	56
6.15	Nozioni generali sulle grammatiche	56
6.16	Grammatica per l'espressione aritmetica	57
6.16.1	Applicazione di sostituzione	58
6.17	Esempio di grammatiche ambigue	58
6.18	Definizione formale di grammatica ambigua	59
6.19	Grammatiche equivalenti e linguaggi ambigui	60
6.20	Problema del dangling else	60
6.21	Risoluzione dell'ambiguità della grammatica aritmetica	61
6.22	Ricorsione sinistra e destra	62
6.23	Eliminazione della ricorsione sinistra	63
6.24	Regola generale per l'eliminazione della ricorsione sinistra	63
6.25	Fattorizzazione a sinistra	64
6.26	Dall'albero di derivazione all'AST	65
6.27	Riassunto sulle grammatiche	65
6.28	Grammatica e cicli infiniti	66
6.28.1	Esempio: Ricorsione sinistra ed ambiguità	66
6.28.2	Esempio: Fattorizzazione sinistra	68
7	Parser	69
7.1	Capacità di riconoscimento del parser	69
7.2	Da input all'output del parser	69
7.3	Requisiti del parser	70
7.4	Costruzione dell'albero di parsing	70
7.5	Requisiti dei parser top down	72
7.5.1	Esempio: parser a discesa ricorsiva	72
7.6	Codifica dell'algoritmo della discesa ricorsiva	73
7.7	Nozioni sui parser	74
7.8	Preparazione sulle grammatiche	75
7.9	Parser non predittivo e predittivo	75
7.10	Parser predittivi e loro potenzialità	76

7.10.1 Esempio: Potenza e limiti dei parser non predittivi	76
7.11 Concetto di First	77
7.12 Simboli in first e follow	77
7.13 Concetto di Follow	78
7.13.1 Esempio: Presenza del valore epsilon	79
7.14 Calcolo di first	80
7.14.1 Esempio: calcolo di First	80
7.14.2 Notazioni e formalismi su first	80
7.15 Calcolo di follow	81
7.15.1 Notazioni e formalismi su follow	81
7.16 Utilità di follow e first	82
7.17 Riassunto sulle grammatiche	82
7.18 Scelte da effettuare durante la realizzazione dell'albero	83
7.19 Riassunto su FIRST e FOLLOW	83
7.19.1 Esempio: Calcolo del FOLLOW	84
7.20 Parser LL(1)	86
7.21 Costruzione formale della tabella LL(1)	87
7.22 Regole per evitare i conflitti	87
7.23 Passaggi per il controllo di una grammatica	88
7.23.1 Esempio: Tabelle con o senza conflitti	89
7.24 Come realizzare un parser	90
7.25 Riassunto sui parser	91
7.26 Schema di un parser	92
7.27 L'insieme delle grammatiche	92
7.28 Esercizio 16: Definire se una grammatica è LL(1)	93
8 Parser bottom up	94
8.1 Introduzione al parsing bottom up	94
8.2 Concetto di riduzione	94
8.2.1 Esempio: Grammatiche LR	95
8.3 Operazioni del parsing shift reduce	96
8.4 Parsing bottom up e derivazione rightmost	96
8.5 Handle	97
8.5.1 Esempio: handle	98

8.6	Automi push down	98
8.7	Esempio: Parser bottom up e grammatica ambigua	99
8.8	Conflitti della tabella LR	100
8.9	Items	100
8.9.1	Esempio: Items	101
8.10	Automa per gli items	101
8.11	Tabelle LR(0) ed impatto dei reduce	102
8.11.1	Esempio: automa per gli items	102
8.11.2	Esempio: Algoritmo ed uso dello stack	103
8.12	Operazioni di reduce e shift nell'esecuzione dell'automa	104
8.13	Differenza tra albero di parsing e albero di derivazione	105
8.14	Riassunto: parsing bottom up	105
8.14.1	Esempio: parser bottom up	106
8.15	Riassunto: Gli handle	107
8.16	LR(1) e funzionamento delle tabelle di parsing	107
8.16.1	Esempio: esecuzione del driver	108
8.17	Riassunto: costruzione dell'automa che riconosce gli handle	108
8.18	Riassunto: concetto di item	110
8.19	Parsing table o tabella di codifica dell'automa	110
8.20	Parser SLR(1)	112
8.21	Algoritmo SLR(1) e riconoscimento di una stringa	113
8.22	Conflitti per le grammatiche LR	114
8.22.1	Esempio di conflitto	114
8.23	Follow contestuali	115
8.23.1	Esempio: Impatto del follow contestuale	115
8.24	Algoritmo ed automa LR(1)	116
8.25	Item LR(1)	116
8.26	Chiusura e goto sugli item LR(1)	116
8.26.1	Esempio: esecuzione grammatica LR(1)	117
8.27	Confronto tra LR(1) items	118
8.28	Automi LALR(1)	118
8.28.1	Esempio automa LALR(1)	119
8.29	Item di produzioni vuote	120
8.30	Riassunto: Specifiche delle grammatiche	120

8.31 Riassunto: relazioni tra grammatiche	121
8.32 Riassunto: Algoritmo LR	122
8.32.1 Esempio: Realizzazione dell'automa LR(1) per la gramamtica postfissa	122
8.33 Conflitti shift/reduce negli automi LR(1)	124
9 Javacup	125
9.1 Introduzione: Javacup	125
9.2 Javacup e grammatiche ambigue	125
9.3 Notazione di javacup	126
9.4 Visione generale dell'analizzatore sintattico	126
9.5 Tools per i parser	126
9.6 Funzionalità ed usi di Javacup	127
9.7 Javacup lato codice	127
9.8 Grammatiche ad attributi e javacup	128
9.9 Operazioni di riduzione nelle grammatiche ad attributi	128
9.10 Comandi di CUP	129
9.11 Produzioni e precedenze	129
9.12 Aggiunta degli attributi alla grammatica	130
9.12.1 Esempio: Esecuzione del Parser data una stringa	130
9.13 Significato ed importanza delle azioni	131
10 Syntax-Directed Translation	133
10.1 Introduzione: Grammatica ad attributi	133
10.2 Calcolo degli attributi	133
10.3 Difficoltà delle grammatiche	134
10.4 Grammatiche S-attribuite	134
10.5 Traduzione diretta dalla sintassi	134
10.5.1 Funzionalità della traduzione diretta dalla grammatica	135
10.6 Tipi di traduzione diretta dalla grammatica	135
10.6.1 Esempio: Produzioni e regole semantiche	136
10.7 Tipi di Attributi e non terminali	136
10.8 Ordine topologico e parsing bottom up	137
10.9 Grammatica L-ereditata	137
10.10 Grammatiche S ed L attribuite	138
10.10.1 Esempio: Grammatica L-attribuita	138

10.10.2 Esempio: Albero di parse tree annotato	140
10.11 Passaggi per il calcolo degli attributi	141
10.11.1 Modi per calcolare gli attributi	141
10.11.2 Esempio: Grammatica circuit	142
10.12 Grammatiche S-attribuite, L-attribuite e grammatiche generali	142
10.12.1 Esempio: Grammatica con cicli	143
10.13 Navigazione dell'albero e calcolo degli attributi	143
10.14 Indipendenza di grammatiche e parser	143
10.15 Riassunto: Grammatiche ad attributi	144
10.16 Regole e produzioni	145
10.17 Riassunto: Da codice sorgente ad AST	145
10.18 Riassunto: Grammatiche ad attributi	145
10.19 Riassunto: Tipi di grammatiche	146
10.20 Regola generale di calcolo di attributi	146
10.20.1 Esempio: Definire se la grammatica sia L-attribuita	147
10.21 Grammatica postfissa (S-attribuita con schemi di traduzione)	148
10.22 Programmazione ad albero	149
10.22.1 Esempio: applicazione della programmazione ad albero	149
10.23 Esempio: dal codice sorgente all'AST	150
10.23.1 Esempio: Realizzazione dell'albero di derivazione	151
10.23.2 Esempio: Definizione della grammatica	151
10.23.3 Esempio: Definizione delle regole alle produzioni	152
10.23.4 Esempio: Grammatica con definizioni guidate dalla sintassi	153
10.23.5 Esempio: Grammatica con schema di traduzione	153
10.23.6 Esempio: Limiti della grammatica	154
10.23.7 Esempio: Forzatura della grammatica S-attribuita	155
10.24 Costruzione del Syntax Tree	155
10.24.1 Esempio: Costruzione di AST partendo da una grammatica aritmetica	156
10.25 Funzionamento dello stack semantico	159
10.26 Riferimento all'albero sintattico	160
10.27 Dalla grammatica all'albero	160
10.28 Funzionamento del Pattern visitor	160
10.29 Grammatica precisa o larga	161
10.30 Statements ed espressioni	161

10.31	Pattern Visitor	162
10.31.1	Esempio: pattern visitor	162
10.31.2	Esempio: Vistor ed operazioni matematiche	163
10.32	Direzione di propagazione	164
11	Analisi semantica	165
11.1	Ambiente e stato	165
11.2	Associazioni statiche e dinamiche per ambiente e stato	166
11.3	Compilatore e gestione della memoria	166
11.4	Scoping	167
11.5	Tipi di scoping	168
11.5.1	Esempio: Scoping e blocchi di codice	168
11.6	Scoping e tabella dei simboli	169
11.7	Analisi semantica e regole	169
11.8	Tecniche per la risoluzione dei problemi	170
11.8.1	Esempio: Problema di statement e dichiarazione	171
11.9	Semantica e regole semantiche	172
11.10	Definizione dell'analisi sintattica estesa (analisi semantica)	172
11.11	Definizione di scope	173
11.12	Regola del most-closely nested rule	173
11.13	Studiare lo scope	174
11.14	Regole di scoping	175
11.14.1	Esempi di regole semantiche	175
11.15	Scoping e tabelle dei simboli	175
11.16	Definizione di variabili e passi	176
11.16.1	Esempio: Programma e scoping	176
11.17	AST e albero delle tabelle di scoping	178
11.18	Implementazione della tabella di scoping	178
11.19	Implementazione della tabella dei simboli	179
11.20	Riassunto: Regole dell'analisi semantica	179
11.21	Fasi dell'analisi semantica e scoping	180
11.22	Visita dell'albero per la costruzione delle tabelle di scoping	180
11.23	Tabelle di scoping con stack ed alberi	181
11.24	Combinazione delle visite	181

11.25	Struttura ad albero dei costrutti	182
12	Regole di inferenza	183
12.1	Introduzione: Regole di inferenza	183
12.2	Ipotesi e conclusioni	184
12.3	Concetto di dimostrabilità	184
12.4	Tipi di regole	184
12.4.1	Regola per Int	184
12.4.2	Regola per Add	185
12.4.3	Esempio: Regola per Add	185
12.4.4	Regole per Bool	186
12.4.5	Regola per String	186
12.4.6	Regole per not	186
12.4.7	Regola per il While	186
12.4.8	Regola per identificatore	187
12.4.9	Regole per sequenze di statements	187
12.4.10	Regole per sequenze di statements	187
12.4.11	Regole per chiamate a procedura	188
12.4.12	Regole per l'assegnazione	189
12.4.13	Regole per If	189
12.4.14	Regole per while	190
12.5	Soundness del type system	190
12.6	Costrutti complessi ed operazioni	191
12.7	Regole e type environment	191
12.7.1	Esempio: Sottoalbero LT	192
12.8	Regole di inferenza, visitor e tabelle	194
12.9	Dichiarazioni e type environment	194
12.10	Riassunto: Regole di inferenza	195
12.11	Dichiarazioni ed aumento del Γ	196
13	Codice intermedio	197
13.1	Introduzione: codice intermedio	197
13.2	Forme intermedie di basso ed alto livello	198
13.3	Da AST a DAG	198
13.3.1	Esempio: Da AST a DAG	198

13.4	Codice intermedio ed ottimizzazione	199
13.4.1	Esempio: Da DAG a codice a 3 indirizzi	200
13.5	Da codice a tabella di quadruple	200
13.6	Tabella di triple	200
13.7	Triple indirette	201
13.8	Single Assignment Form	201
13.9	Problemi del Single Assignment Form	202
13.10	Specifica del codice a tre indirizzi	202
13.10.1	Esempio: Codice a tre indirizzi	203
13.11	Generazione del codice intermedio partendo dalla grammatica	203
13.11.1	Esempio: Considerazioni sulla grammatica del codice intermedio . . .	204
13.11.2	Esempio: Regole per il calcolo degli attributi	205
13.12	Traduzione incrementale o Generazione di codice incrementale	206
13.13	Istruzioni controllo di flusso	207
13.14	Espressioni booleane	207
13.15	Codice short circuit	208
13.16	Regole per l'uso delle espressioni booleane	209
13.17	Tipi di goto	209
13.18	Introduzione alle true e false list	210
13.19	Problema di goto ed indirizzi	211
13.20	Backpatching	211
13.21	Backpatch espressioni booleane	212
13.21.1	Esempio: Codice per le espressioni booleane	212
13.22	Esempio: Relazione tra espressioni	214
13.23	Blocchi di codice: Espressioni booleane	215
13.24	Riassunto: OR ed espressioni booleane	215
13.25	Blocchi di istruzione: Statements	216
13.26	Produzione dell'If e codice intermedio	217
13.27	Produzione dell'If-else e codice intermedio	217
13.28	Produzione del for e codice intermedio	218
14	Run-time environment	220
14.1	Introduzione: Run-time environment	220
14.2	Risorse a runtime	221

14.3	Obiettivi della generazione del codice	222
14.4	Assunzioni sull'esecuzione dei linguaggi	222
14.5	Chiamate a funzione e gerarchia	222
14.6	Concetti di attivazione	223
14.6.1	Lifetime di una funzione	223
14.6.2	Lifetime di una variabile	223
14.7	Albero di attivazione	223
14.7.1	Esempio: type environment e run-time environment	224
14.8	Run-time environment ed informazioni utili	225
14.9	Associazione tra variabile e memoria	225
14.10	Comportamento del record di attivazione	226
14.11	Gestione della memoria a tempo di esecuzione	226
14.12	Record di attivazione o frame	227
14.13	Contenuto del record di attivazione di una funzione chiamata	228
14.14	Sintesi sul contenuto del record di attivazione	228
14.15	Celle di memoria e record di attivazione	229
14.15.1	Esempio: chiamate a funzione e celle di memoria	230
14.16	Nozioni su i compilatori e gestione della memoria	230
14.17	Variabili globali	231
14.18	Heap	231
14.19	Concetto di alignment	232
14.20	Stack machine	232
14.21	Funzionamento dell'istruzione somma	233
14.22	Macchine a stack con accumulatore e schema uniforme	233
14.22.1	Esempio di stack ed accumulatore	234
14.23	Riassunto: gestione della memoria	235
14.24	Riassunto: heap e stack	235
14.25	Riassunto: stack ed accumulatore	236
14.26	Codice MIPS	236
14.26.1	Esempio: Stack e scrittura di un elemento	236
14.27	Da stack machine a MIPS	237
14.28	Istruzioni su MIPS	237
14.28.1	Esempio: MIPS e codice per la somma	238
14.29	Funzione fittizia cgen(e)	238

14.30 Implementazione dell'if nel MIPS	239
14.30.1 Esempio: MIPS ed if	239
14.31 Chiamate a funzione e generazione del codice	240
14.32 Frame pointer	240
14.33 Calling e return sequence	241
14.34 Comando per il jump della funzione	241
14.35 Passi della calling sequence	242
14.36 Passi della return sequence	243
14.37 Riassunto: Calling e return sequence	243
14.38 Generazione del codice per le variabili	243



Elenco delle figure

1.1	Struttura di un compilatore	3
1.2	LLVM IR	9
1.3	Fasi e regole del compilatore	11
1.4	JIT Compiler	14
4.1	Analizzatore lessicale (Lexer)	25
4.2	Diagramma di transizione per i white spaces	35
4.3	Tabella degli operatori	35
4.4	Diagramma di transizione per gli operatori	36
4.5	Diagramma di transizione per gli id o parole chiave	36
4.6	Diagramma di transizione per i numeri	37
4.7	Diagramma di transizione per la parola chiave then	38
6.1	Struttura del parser	48
6.2	Risoluzione dell'ambiguità della grammatica aritmetica	62
7.1	Parser a discesa ricorsiva	70
7.2	Esempio parser a discesa ricorsiva	72
7.3	Esempio parser a discesa ricorsiva	73
7.4	Codice per il parser a discesa ricorsiva	74
7.5	Concetto di follow e frontiera	79
7.6	Riassunto su FIRST e FOLLOW	84
7.7	Schema di un parser	92

7.8	L'insieme delle grammatiche	93
8.1	Esempio: Grammatiche LR	95
8.2	Esempio: Parser bottom up e grammatica ambigua	99
8.3	Esempio: automa per gli items	103
8.4	Esempio: parser bottom up	106
8.5	Riassunto: costruzione dell'automa che riconosce gli handle	110
8.6	Parser SLR(1)	112
8.7	Esempio di conflitto	114
8.8	Esempio: esecuzione grammatica LR(1)	117
8.9	Esempio: esecuzione grammatica LR(1)	118
8.10	Confronto tra LR(1) items	119
8.11	Confronto tra LR(1) items	120
8.12	Riassunto: Specifiche delle grammatiche	121
8.13	Realizzazione dell'automa LR(1)	123
8.14	Conflitti shift/reduce negli automi LR(1)	124
10.1	Esempio: Albero di parse tree annotato	140
10.2	Esempio: Grammatica con cicli	143
10.3	Esempio: Definire se la grammatica sia L-attribuita	147
10.4	Esempio: Realizzazione dell'albero di derivazione	151
10.5	Esempio: Grammatica con schema di traduzione	154
10.6	Esempio: Forzatura della grammatica S-attribuita	155
10.7	Esempio: Costruzione di AST partendo da una grammatica aritmetica	156
10.8	Esempio: Costruzione di AST partendo da una grammatica aritmetica	158
10.9	Esempio: Costruzione di AST partendo da una grammatica aritmetica	158
10.10	Esempio: pattern visitor	163
11.1	Ambiente e stato	166
11.2	Scoping	167
11.3	Scoping e tabella dei simboli	169
11.4	Studiare lo scope	174
11.5	Esempio: Programma e scoping	177
11.6	Esempio: Programma e scoping	177
12.1	Esempio: Sottoalbero LT	193

13.1	Introduzione: codice intermedio	197
13.2	Esempio: Da AST a DAG	199
13.3	Da codice a tabella di quadruple	200
13.4	Tabella di triple	201
13.5	Triple indirette	201
13.6	Single Assignment Form	202
13.7	Esempio: Codice a tre indirizzi	203
13.8	Generazione del codice intermedio partendo dalla grammatica	204
13.9	Esempio: Regole per il calcolo degli attributi	205
13.10	Regole per l'uso delle espressioni booleane	209
13.11	Backpatch espressioni booleane	212
14.1	Risorse a runtime	221
14.2	Esempio: type environment e run-time environment	224
14.3	Esempio: type environment e run-time environment	224
14.4	Gestione della memoria a tempo di esecuzione	226
14.5	Record di attivazione o frame	227
14.6	Esempio: chiamate a funzione e celle di memoria	230
14.7	Esempio: chiamate a funzione e celle di memoria	230
14.8	Heap	232
14.9	Funzionamento dell'istruzione somma	233
14.10	Macchine a stack con accumulatore	233
14.11	Esempio: Stack e scrittura di un elemento	237
14.12	Esempio: MIPS e codice per la somma	238
14.13	Esempio: MIPS e codice per la somma	238
14.14	Esempio: MIPS ed if	239
14.15	Frame pointer	241
14.16	Passi della calling sequence	242

Elenco delle tabelle



1.1 Definizione di un compilatore

Il compilatore è un **programma che ha caratteristiche molto stringenti**. Esso **traduce una serie di istruzioni scritte di un determinato linguaggio di programmazione** (codice sorgente) **in istruzioni di un altro linguaggio** (codice target). Quest'ultimo può essere un **linguaggio di alto livello**, uno **intermedio** o **assembly**, che **viene poi tradotto in linguaggio macchina tramite l'assembler**. Scegliere il linguaggio target è una scelta, avendo piena libertà.

1.1.1 Il linguaggio Ada & altri linguaggi

Ada è un **linguaggio usato per l'aeronautica militare** e, una volta scritto un programma in questo linguaggio, **bisognava poter dimostrare con un teorema che quel programma funzionasse e che faceva quello per cui era stato creato**. Ada era il massimo dello sviluppo di linguaggi di programmazione: doveva essere **facile, comprensibile e dimostrabile che fosse safe**. Con Ada però stavano bloccando il programmatore, perciò è stato **soppiantato dal C** che **dava più spazio di movimento al programmatore**. SQL è un **linguaggio domain specific non general purpose**, anche se la **maggior parte dei linguaggi general purpose possono diventare domain specific tramite librerie**. Un linguaggio di per sé non vale niente senza librerie. I linguaggi nascono tutti semplici. In generale, si sta cercando di **disincentivare**

sempre di più i linguaggi unsafe, come C. Ogni linguaggio di programmazione è più o meno l'evoluzione di un altro

1.2 Linguaggi interpretati e compilati

Python è un linguaggio interpretato. Il fatto di essere interpretato o compilato non è insito nel linguaggio, nel senso che **python non deve essere necessariamente interpretato**. Da una parte c'è l'annotazione (come Python) e dall'altra c'è chi implementa l'annotazione e la fa eseguire sul computer. Chi ha implementato Python all'inizio quindi ha deciso che dovesse essere interpretato, ma esiste JPython che compila in bytecode. Di norma, i linguaggi hanno delle caratteristiche che preferiscono l'interpretazione o compilazione. Ad esempio se vengono dichiarate tutte le variabili è probabile che tu sia un linguaggio compilato. Se non hai tutte le dichiarazioni potresti ancora esserlo ma è più complesso

1.3 Tipologie e caratteristiche dei linguaggi

I linguaggi possono essere **totalmente compilati, totalmente interpretati o ibridi**. **Linguaggi imperativi o procedurali sono linguaggi a cui dai un comando**. La maggiorparte dei linguaggi sono così, ovvero si dice **"metti 1 nella variabile x"**. Se devo spiegare come arrivare ad una finestra dico **"fai un passo avanti, fanne un altro..."**. **Si fornisce la procedura per svolgere un'attività**. Essi sono un'antitesi ai **linguaggi dichiarativi**, che sono più ad alto livello in cui si dice **"vai alla finestra"**, ovvero **non si spiega passo passo cosa fare**. In tal caso però deve esserci un **processore che dal comando "vai alla finestra" deve cacciare fuori i passi per arrivare alla finestra**. Un esempio di linguaggio dichiarativo è SQL. **Con il dichiarativo dico cosa fare ma non viene specificato come, dichiarando ciò che voglio** mentre negli imperativi **specifico cosa fare**, i passi da fare. Esempio di **linguaggio dichiarativo** è il **linguaggio naturale**. Ciò perché possiamo dire agli LLM cosa fare. Al momento, però, agli LLM manca la sicurezza che il programma scritto da quest'ultimo faccia ciò che deve fare. Tale problema potrebbe nascere anche dal fatto che il linguaggio umano è spesso ambiguo. Esistono poi linguaggi di prima generazione, seconda...

I linguaggi funzionali sono una specie di linguaggi dichiarativi. Un esempio è il LISP, linguaggio funzionale in quanto **basato sulla matematica, visto come funzione ed argomenti**. Alcune caratteristiche funzionali e logiche le stanno inserendo in linguaggi di programmazione. **Nei linguaggi funzionali si passa la funzione**

1.4 Supportare o simulare una caratteristica

Un linguaggio supporta una caratteristica o può essere simulata. Il C non supporta il passaggio delle funzioni ma si simula ciò tramite i puntatori a funzione. Essendo il C general purpose ci posso fare tutto. Allo stesso modo, C non supporta l'Object Orientation ma potrebbe comunque essere simulato se si è bravi a programmare

1.5 Struttura del compilatore

Questa mostrata in figura è una sequenza di passi logici ma l'implementazione di tali passi può avvenire in vari modi. Un compilatore ha come input il codice sorgente e come output il codice target. Le fasi potrebbero essere anche mischiate, ma ci saranno tutte

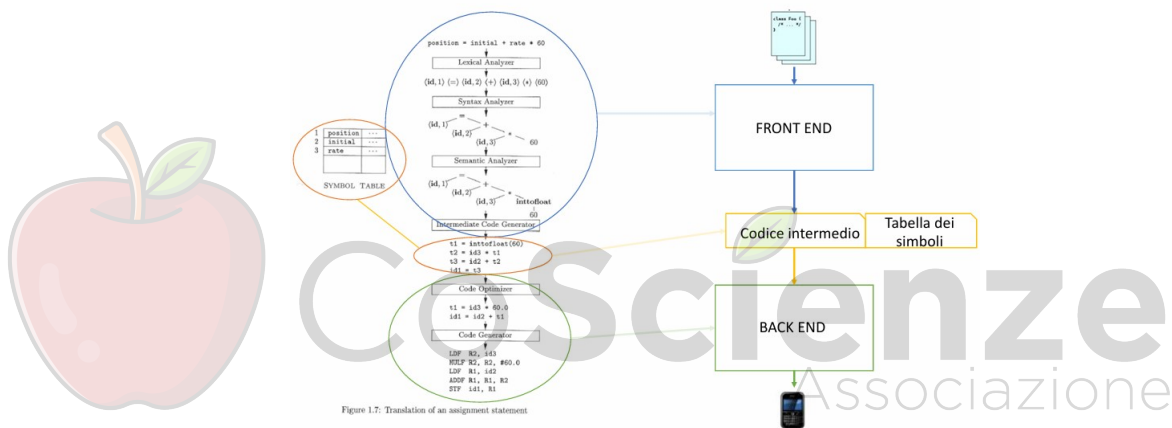


Figura 1.1: Struttura di un compilatore

1.5.1 Fase dell'analisi lessicale (lexical analyzer)

Esso prende la stringa che è il codice sorgente e produce i cosiddetti tokens. Ogni stringa di caratteri che ha un significato viene raggruppato e gli viene dato un nome logico, ovvero il token. Ad esempio alla variabile "position" viene dato il token "<id,1>". Si potrebbe usare come separatore di stringhe lo spazio, in quanto gli umani ragionano così, ma non è necessariamente quello il separatore. Quando devo riconoscere "position" come identificatore devo capire se lo spazio si può mettere o meno nella parola, se la virgola può far parte della parola o così via. Ci sono quindi due figure, chi inventa il linguaggio di programmazione e che implementa il linguaggio di programmazione. Se essi non si parlano la vita dell'utente diventa complesso. Più in generale, si prende la prima sequenza di caratteri del codice sorgente e sulla base delle regole definite nel box del Lexical Analyzer

si determinano gli identificatori. Ad ogni passaggio di fase **non posso perdere informazioni**. Il fatto che “position” ed “initial” siano due identificatori non vuol dire che siano diversi. **Siccome i nomi delle variabili sono state inventate dal programmatore, esse verranno prese e messe nella tabella dei simboli. Si assegnerà poi un token e mediante quest’ultimo saremo in grado di identificare il lessema di partenza.** Ciò che abbiamo di concreto all’inizio alla fine di questa fase viene trasformato in una rappresentazione logica. Ognuna delle fasi è un traduttore. L’analizzatore lessicale fa due cose: **riconosce la parola e la classifica.** è necessario vedere se la parola è presente nel linguaggio e se sia un identificatore o meno. **Se è una parola chiave del linguaggio allora tale parola non va messa nella tabella delle stringhe e spesso il nome del token equivale al nome alla parola chiave del linguaggio, come “if”, “int” ...** Bisogna fare attenzione alle parole chiave e parole inventate dal programmatore. Saranno le regole a dirci se una parola è una parola chiave o meno per linguaggio. **“main” è una parola riservata ma non chiave. Tutte le parole chiave sono riservate ma le parole riservate non sono tutte parole chiave. L’analisi delle parole riservate spetta poi all’analisi sintattica.** In generale, il lexical analyzer prende sempre le parole di senso compiuto più lunghe. **Simboli non significativi nel linguaggio, come lo spazio, vengono mangiati dal compilatore. Fasi di analisi lessicale, sintattica e semantica sono separate in quanto fasi logiche, ma potrebbero essere programmate con un unico programma o come moduli separati.** Ognuno di questi blocchi avrà un insieme di regole facenti riferimento al linguaggio sorgente.

1.5.2 Fase dell’analizzatore sintattico (Syntax Analyzer)

Mentre l’analisi lessicale riconosce le singole parole, l’analisi sintattica riconosce le frasi che devono essere sintatticamente corrette. Nel box syntax analyzer ci sono tutte le regole sintattiche. In questo box andiamo a mettere tutte le regole del nostro linguaggio, il che **risolve il problema delle frasi sintatticamente corrette. Non risolve però il problema della frase semanticamente corretta, ovvero se la frase abbia un senso.** Ad esempio, se nella variabile “a” che è una stringa vada a mettere un oggetto o faccio **“int i= ‘ciao’”** ciò è sintatticamente corretto in quanto c’è la dichiarazione, assegnazione e costante ma ciò è sbagliato in quanto si assegna una stringa alla variabile int. Tale fase mette l’interpretazione in un albero. L’assegnazione è la radice, l’operatore principale e a sua volta ha due figli sono gli argomenti: **l’identificativo dove devo andare a segnare qualcosa e l’espressione che devo assegnare.** Potrebbe accadere, come nell’immagine, che a sua volta l’espressione sia formata da più argomenti, come una somma ed un prodotto, che a sua

volta prende due argomenti. Gli argomenti della somma saranno il primo elemento da sommare e il risultato del prodotto. Ciò però non è detto, in quanto **è a discrezione di chi crea il compilatore scegliere.** Nell'immagine ad esempio, si preferisce quella che è umanamente accettata, pertanto si dice che la somma abbia come prodotto "initial" e il risultato del prodotto di "rate" e "60", il che rispetta l'accezione matematica. **Nella grammatica indico come leggere la frase. Riesco a dire chi abbia la precedenza nella frase in quanto essa è rappresentata da una regola contenuta nella box della Syntax Analyzer.** Il risultato di tale fase è un albero sintattico. **Output di questa fase è L'Abstract Syntax Tree.** Con questo albero non abbiamo perso nessuna informazione. **Unito ad esso ci sarà inoltre la tabella dei simboli mediante cui possiamo riconoscere gli identificatori.** Tutti i simboli sintattici come ";" o "(" e così via, vengono persi in questa fase in quanto utili per costruire l'albero mediante il quale si potrà poi risalire a tali simboli. Presa un'istruzione e costruitoci un sottoalbero, sarà inutile conoscere anche i simboli. Una volta capita che una determinata istruzione è un'istruzione condizionale, **creerò un sottoalbero in cui a sinistra c'è la condizione a destra l'istruzione che deve essere eseguita se la condizione è vera.** **Questi due sottoalberi avranno come nodo padre il nodo contenente la parola chiave "operazione if",** nodo che ha la semantica di tale istruzione. **Quando analizzo un AST e non trovo riferimento ai tipi vuol dire che sono ancora nella fase di analisi sintattica, in quanto è l'analisi semantica ad annoverare l'albero con i tipi.** I token possono anche avere degli attributi. **Solitamente i token hanno attributi se da soli non riuscirebbero a mantenere le informazioni necessarie,** ma si necessita di un attributo che mi distingua

1.5.3 Fase analisi semantica (Semantic Analyzer)

Ciò che non riesco a fare con l'analisi sintattica viene fatto qui. Nell'immagine, date per scontato che le variabili sono reali, il "60" deve essere risolto dal compilatore in quanto non si possono effettuare operazioni tra reali ed interi, causa la loro rappresentazione in memoria. **Posso quindi utilizzare una funzione che trasforma da intero a float la variabile.** Se però il mio compilatore non prevede una funzione che sappia modificare un floating con un intero, a tale operazione o si dovrebbe dare un errore. Nell'analisi semantica **devo andare a raccogliere tutte le informazioni di tipo sparse nel programma e poi dovrò utilizzarle.** Nell'immagine quindi, dovrò capire il tipo di "position" "initial" e "rate" e devo tradurre l'inter in float. Così facendo, **l'albero viene modificato.** Tutte le informazioni di tipo verranno raccolte con una visita del programma. **Alla fine dell'analisi semantica è stato tradotto in una forma logica interna.** Anche in questa fase non si potrà perdere il riferimento alla

tabella dei simboli. Si parte dalla creazione dell'albero mediante analisi sintattica mentre l'aggiunta dei tipi a funzioni o variabili è viene fatta dall'analisi semantica rivisitando l'albero. Quando legge un'operazione, l'analizzatore semantico prima vede il tipo del primo operatore poi del secondo. Va poi a vedere la regola in cui si specifica il tipo della somma noto il tipo dei due operatori e solo dopo definisce il tipo del risultato. L'assegnazione del tipo al risultato è quindi un qualcosa che viene fatta nell'ottica della bottom up: prima si vedono i figli (gli operatori) e poi si definisce il tipo del risultato. Prima di analizzare le istruzioni posso:

- **calcolare tutti i tipi dei nomi inventati dal programmatore:** vado esclusivamente ad analizzare i nodi DECL OP (nodi di dichiarazione). In questa fase imparo a capire quali sono i tipi. Questa visita viene fatta partendo dalla radice. In questa fase entro nei nodi DECL e sulla base dei figli di questo sottoalbero sarò in grado di capire il tipo del dato o la firma della funzione. Durante tale analisi si associa all'identificatore nella tabella dei simboli il relativo tipo
- **effettuare type checking:** vado a controllare che i tipi che ho visto siano ben usati. Una volta assegnati i tipi agli identificatori, bisogna vedere se siano stati usati bene. Controllo quindi l'uso dei tipi. Tale attività si fa bottom up propagando il tipo dai figli al padre. Tutti i token (che coincidono con le foglie) avranno un tipo. In questo controllo si parte quindi dalle foglie e si dà un tipo a tutti i nodi interni. Viene dato un tipo in base alle regole semantiche.

1.5.4 Fase dell'intermediate code generator

Successivamente all'analisi semantica, questo che ho scritto ha senso e può essere tradotto in un linguaggio macchina. Se passa l'analisi semantica il programma è traducibile. Non ci sono errori sintattici o semantici ma possono esserci errori come null pointer. Possiamo ora generare un codice intermedio. Il file iniziale si è perso in quanto ho lavorato sulle sue forme intermedie. Il codice intermedio che invece posso generare ora è un codice che è a metà tra sorgente e target. Se il target sarà un linguaggio che ha per ogni operatore 2 argomenti, dovrò tradurre tutto in operazioni binarie, anche se all'inizio il codice aveva operazioni con 3 argomenti. Si traduce quindi in codice intermedio che assomiglia al codice target ed è semanticamente equivalente al codice target. Il compilatore in questa fase analizzando l'albero può aggiungere altre variabili temporanee al fine di risolvere i calcoli.

1.5.5 Fase del code optimizer

Tale codice, per essere sicuri che sia equivalente, viene ottimizzato, in quanto il codice dato dalla fase di intermediate code generator potrebbe essere non ottimo. L'ottimizzazione può essere leggera o meno, però bisogna assicurarsi che il codice continui a funzionare.

1.5.6 Fase del code generator

Una volta che il codice viene ottimizzato può essere generato in linguaggio target, nella foto il linguaggio target è il linguaggio macchina

1.6 Tipologie di compilatori

1.6.1 Il front end

Esso è il primo passo. In questo si parte dal sorgente e si genera del codice intermedio. Il front end si preoccupa di interfacciarsi con il linguaggio sorgente e per fare ciò lo deve conoscere alla perfezione. Mi dice se è corretto o meno e lo deve poi tradurre in un codice intermedio con una tabella dei simboli. Se si perde la tabella dei simboli non si riesce a capire la differenza tra i vari identificatori, tra i vari token. Il codice intermedio è una forma che dipende anche dal codice target ma molte volte è definito. Al front end serve conoscere il codice intermedio, dell'hardware lui non ha conoscenza.

1.6.2 Back end

Chi ha una conoscenza profonda dell'hardware è il back end in quanto deve produrre un codice ottimizzato per l'hardware finale. Le ottimizzazioni e generazioni che vado a fare dipendono fortemente dall'hardware. La conoscenza di colui che scrive il front end si ferma al codice intermedio, non conoscendo l'hardware. Chi si preoccupa poi di tradurre in codice intermedio in codice macchina dell'hardware è colui che scrive il back end.

La difficoltà del compilatore non è il front end in quanto è assodata, ma il back end. Fin quando si lavorava solo con le CPU non c'era molto ricerca. Attualmente, invece, causa IA e GPU, processori diversi che lavorano con programmi con elevato parallelismo, si sta osservando una crescita di ricerca nei compilatori. Le GPU vengono utilizzate attualmente utilizzate sia per calcolo grafico che per calcolo tradizionale in parallelo, nello specifico nell'IA vengono usate sulle matrici. Esempio di scheda GPU è l'NVIDIA, che ha il suo linguaggio di programmazione (cuda) che prevede la conoscenza dell'hardware, non ponendosi quindi

come un linguaggio ad alto livello. È come un C che dipende dall'hardware. Attualmente la ricerca è creare un compilatore che prenda un linguaggio ad alto livello e che compili avendo come output cuda ottimizzato per il processore. **Si vuole creare compilatori che diano codice ottimo per la parallelizzazione**

1.6.3 Compilatori monolitici

Il problema del compilatore monolitico è che il risultato di tali compilatori non era riutilizzabile in altri contesti. Si è quindi preferito separare il monolitico in due parti: front end e back end. Il front end fissa il codice intermedio, pertanto il front end lo scrivo una volta e non scrivo più e più partendo dal codice intermedio scrivo il backend per i vari hardware. C'è quindi un codice intermedio standard. Il front end dipende dal codice sorgente mentre il backend dipende dall'hardware. Avendo 20 linguaggi e quindi 20 front end ed avendo 10 hardware e quindi 10 back end, avrò 200 combinazioni di compilatori. **Se cambio hardware cambio back end mentre se cambio il front end devo cambiare il sorgente**

1.7 Compilatori multiplatforma e multilinguaggi

Tale tipo di compilatore fissa un codice intermedio e lo rende noto a chi scrive front end ed a chi scrive back end. Al front end serve saperlo in quanto diventa il codice target, mentre al backend serve saperlo in quanto diventa codice sorgente. Il front end non deve preoccuparsi delle specifiche del linguaggio macchina target mentre il back end non deve preoccuparsi delle specifiche del linguaggio di programmazione sorgente.

Per poter fare il compilatore multiplatforma non c'è un solo codice intermedio ma possono essercene più di uno. Ci sarà parte di codice che può essere parallelizzato ed altro che non deve essere parallelizzato (parallelizzare non è sempre la soluzione). Si potranno avere al contempo più codici intermedi che sono più vicini alla macchina su cui devono eseguire. Lo stesso codice può essere segmentato in più parti ed ogni parte andare su un hardware diverso, questo perché ogni hardware è sempre e solo di una macchina.

1.8 Codice intermedio LLVM IR

Esso è un'infrastruttura che fornisce un codice intermedio. Possiamo scrivere in C in quanto avrà un processore Clang che lo traduce in questo linguaggio intermedio, potendo

scriverlo anche in Swift, C++, Rust... L'infrastruttura fornita insieme alla forma intermedia dell'LLVM IR è l'insieme di procedure per cui scrivere il back end diventa facile.

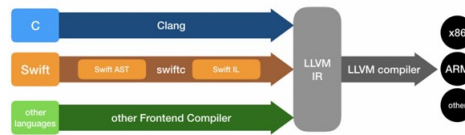


Figura 1.2: LLVM IR

1.9 Infrastrutture ibride

In tali infrastrutture il back-end è eliminato dal compilatore e sostituito da una macchina virtuale (interprete software) resa più efficiente da un JIT compiler. Esso implementa parti del back-end che però in questo caso sono eseguite durante l'esecuzione del codice intermedio. La macchina virtuale ed il suo jit compiler sono sempre specifici ad un'architettura hardware.

Quelle più note sono Java e .NET, che presentano rispettivamente i seguenti codici intermedi:

- Bytecode
- Common Intermediate Language (CIL)

Quando si esegue Java, viene compilato il bytecode e viene poi eseguito sulla macchina virtuale. Siccome tutte le architetture hanno la macchina virtuale Java, allora esso è un linguaggio multiplatforma. Esso è quindi multiplatforma in quanto qualcuno ha implementato la macchina, ma non nasce come tale. Java non è altro che una notazione. Al contempo Java è diventata multilinguaggi in quanto più linguaggi generano il bytecode. Avendo lo stesso target, essi possono essere mischiati.

Microsoft ha inventato un altro linguaggio, .NET il cui linguaggio intermedio è CIL, che ha la stessa filosofia di Java. Microsoft ha preso tutti i linguaggi che già sviluppava e li ha portati tutti a compilare in CIL. Su DART hanno fatto molti compilatori traducendolo in linguaggio macchina, in Javascript... Un linguaggio può essere fatto diventare interpretato, compilato, si potrebbe eseguirlo a volo (come fare una macchina virtuale per C)

1.10 Codice intermedio ed impatto sui compilatori

Inizialmente **non esistevano codici intermedi standard**, pertanto era **necessario fare compilatori monolitici** ed ogni linguaggio di programmazione aveva il suo codice intermedio, diverso da quello di un altro compilatore. Si è poi compreso che **si poteva definire un codice intermedio standard e separare backend e frontend per fare in modo che senza conoscere l'hardware mi posso inventare un linguaggio di programmazione e fare il compilatore**. Attualmente, invece, **passando per il front end arrivo all'LLVM IR e utilizzo il suo framework per ottimizzare il codice, attività che viene nel back end**. L'ottimizzazione è stata fondamentale da sempre per lo sviluppo di linguaggi ad alto livello.

Il codice macchina è un linguaggio interpretato in quanto viene eseguito riga per riga ed è interpretato dall'hardware. L'assembly invece è un linguaggio tradotto più che compilato, in quanto viene tradotto 1 ad 1 cosa che non può essere fatta con i linguaggi ad alto livello.

Si è inoltre provato a fare un **hardware che leggesse direttamente in bytecode per raggiungere velocità elevate**. Non ci si è però riusciti, per questo si è preferito avere la **suddivisione tra macchina virtuale (che legge bytecode) e macchina fisica (che legge codice macchina)**.

Va specificato che è **un problema non decidibile partire da un codice sorgente per andare a creare un programma che è ottimo**

1.11 Regole e fasi del compilatore

L'input di ogni fase non sarà solo l'input vero e proprio ma anche l'insieme di regole di ogni analyzer che dovranno essere poi applicate.

1.11.1 Regole lessicali

Esse **definiscono chiaramente come può essere fatto un identificatore, definendo cosa sia o meno una parola**. Sono contenute nel primo box "Lexical Analyzer". Alcune regole sono:

- **non possono esserci spazi**
- **non possono esserci simboli speciali**

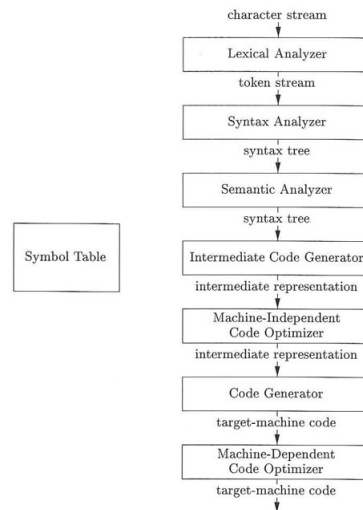


Figura 1.3: Fasi e regole del compilatore

Queste ed altre regole **vanno a definire chiaramente come può essere fatta o meno un identificatore**. Ogni **output di una fase è una forma intermedia**. Output di questa fase è un **token stream**

1.11.2 Regole sintattiche

Il Syntax Analyzer è un programma che **prende le regole sintattiche di come devono essere fatte le frasi e va a controllare che venga rispettata la struttura della frase**. Esempio va a **controllare se l'if è ben strutturato**. Com'è fatta la frase (if in questo caso) che sia esso in C o in Pascal **deve essere scritto come regola e dato in pasto al Syntax Analyzer**. A quest'ultimo arrivano **flussi di token e lui riconosce le frasi**. Output di questa fase è un **abstract syntax tree (AST)**

1.11.3 Regole semantiche

Le **regole che non riesco a mettere nell'Syntax Analyzer in quanto troppo complesse**, esempio sono le **regole di tipo, vengono messe nel Semantic Analyzer**. Se una variabile è dichiarata come intero deve essere usata come tale. Il **Semantic Analyzer effettua per lo più il type checking**. Output di questa fase è un **abstract syntax tree (AST)**. L'ultima forma **intermedia è proprio il codice intermedio** (prodotto dalla fase Intermediate Code Generator). L'operazione **dal linguaggio sorgente al linguaggio intermedio è detto abbassamento** in quanto la CPU è in basso e si scende dal linguaggio ad alto livello ad uno intermedio, scendendo.

1.12 Fasi logiche di Clang

Esso è un compilatore che compila sia C++ che ObjectiveC in codice intermedio LLVM IR. Le sue fasi logiche sono:

- **preprocessing:** tale fase **lavora sul sorgente e produce un sorgente**. Il preprocessore prende gli include dei "file.h" e li sostituisce con il contenuto di tali file. Il preprocessore quindi prende il sorgente e lo estende. Il vero programma che passa al compilatore è quello risultato di questa fase
- **Parsing ed analisi semantica:** nel parsing si effettua anche l'analisi lessicale. In questa fase è contenuto buona parte del front end, fermandosi però all'abstract syntax tree ed escludendo il codice intermedio
- **Code generation and optimization:** Prende l'AST, lo traduce in LLVM IR e poi lo trasforma in codice macchina. Tale fase **ottimizza anche il codice generato**. L'ottimizzazione però non sta solo in questa fase. Quando ad esempio si passa da 60 a 60.0, essa è un'operazione fatta dal compilatore che evita debba essere necessario utilizzare una funzione di cast da int a float. è un calcolo che si evita di fare a runtime. Da notare che tale operazione non ha cambiato la semantica del programma, pertanto è possibile ottimizzare anche nel front end, dove il compilatore cerca di fare tutte le operazioni sicure. L'ottimizzazione avviene sia sul codice intermedio che sul codice finale, in quanto il codice deve essere ottimale. **Generato il codice assembly, avremo un file ".s"**
- **assembler:** Prende il file.s e lo traduce in un linguaggio macchina che traduce il codice in un file ".o". Esso però non è ancora l'eseguibile. **Se il nostro file chiama funzioni esterne, esse devono essere aggiunte**. Ad esempio nell'oggetto ho la chiamata alla printf ma non ho la printf, che deve essere chiamata runtime
- **Linker:** prende il file .o e lo collega con tutti gli altri .o in cui ci sono i codici delle librerie.

1.13 Language processing system

Più in generale:

- **preprocessore:** esso **prende il sorgente e lo traduce in sorgente**, nonchè output di tale fase

- **compilatore:** composto dai vari step riportati precedentemente. Output di tale fase è il **target assembly program**
- **assembler:** utile se vogliamo generare **assembly**. Si potrebbe generare anche linguaggio macchina ma solitamente si genera l'assembly. Tale fase **produce codice macchina rilocabile**. Rilocabile in quanto il SO può essere messo dove vuole
- **loader/linker:** Il linker collega il ".o" con il ".o" delle librerie ed infine il loader si occupa della run, dell'esecuzione.

Il compilatore è solo una parte del processing system

1.14 Appartenenza del codice intermedio

Il codice intermedio può far parte del front end o back end, dipende da come lo si vede. Se è specifico dell'hardware è più del back end. Esso fa parte di uno o dell'altro in base a cosa dipende: se dipende dal linguaggio di programmazione allora è front end, altrimenti è back end. Se faccio del codice intermedio che dipende dalla CPU e quindi dall'hardware, il codice intermedio farà più parte del back end

1.15 Compilatori e Java

Il compilatore in Java aggiunge automaticamente e di default un costruttore a classi che non lo hanno. Il costruttore **non avrà parametri e richiamerà solo super()**. In Java, inoltre, **somma tra due int restituisce un int**. Questo è dato da una regola riportata nell'analisi semantica. Cambiando questa regola allora cambierebbe anche il risultato della fase dell'analisi semantica. In quest'ottica, **possiamo cambiare tutto di un linguaggio di programmazione**.

Con il bytecode, nella JVM gli argomenti sono in uno stack pertanto non è necessario avere gli argomenti, ma piuttosto operazioni di caricamento. Tutte le macchine virtuali sono basate su stack piuttosto che essere basate su RAM (a cui siamo abituati). Il codice è più compatto.

1.16 Compilatori e C

A differenza di Java, con C c'è stata invece sin dall'inizio la necessità del preprocessore. Si parte dal file sorgente e come risultato del preprocessing si vanno ad includere i file riportati con "include".

1.17 Problemi di Java e bytecode

Il problema di Java e bytecode è che essendo interpretati sono più lenti. Ciò che non è lento è il codice macchina che, seppur interpretato, essendo interpretato direttamente sull'hardware è più veloce. Tutti i linguaggi interpretati non su macchina ma su macchina virtuale sono più lenti in quanto le istruzioni devono essere prima lette e poi tradotte in codice macchina che viene poi eseguito. Il compilato va direttamente sulla CPU in esecuzione, mentre interpretato viene tradotto al volo ed eseguito. Per risolvere il problema della lentezza è stato introdotto il JIT compiler

1.18 Just in Time Compiler

Con questo compilatore cambia un po' l'architettura. Quando si lancia il bytecode sulla VM a tempo di esecuzione, se non ci fosse il JIT il codice andrebbe direttamente in esecuzione, venendo interpretato ed eseguito. Si è notato che durante le esecuzioni c'è un tempo di latenza, ovvero non si riesce ad usare tutti i cicli della CPU durante l'esecuzione di un programma

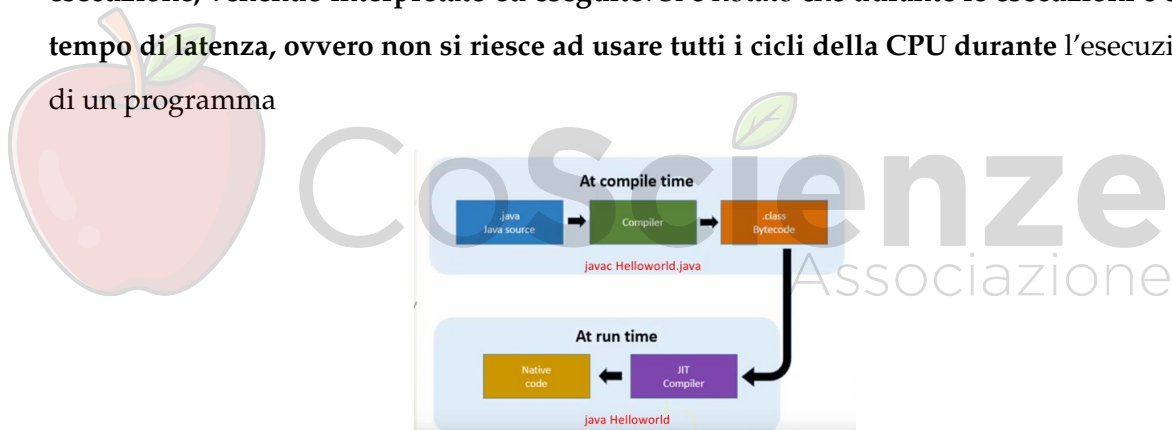


Figura 1.4: JIT Compiler

Ad esempio quando l'utente deve inserire la stringa, sono tempi enormi rispetto alla CPU perciò intanto la CPU dorme. Se però sta senza fare niente, mentre aspetta che l'utente dia l'input, nel frattempo potrebbe fare qualcosa. Le si fa lanciare un compilatore al volo. Vengono compilati i pezzi più frequenti del bytecode. Se un'istruzione è ripetuta x volte, dovrei interpretarla x volte. Piuttosto che fare ciò, compilo questa istruzione in codice macchina, per cui lanciando l'esecuzione alcune cose vengono interpretate ed eseguite, altre compilate e se la stessa cosa è stata già compilata viene eseguita direttamente la versione compilata. In base a ciò, è stato definito anche l'Ahead of time compiler è quello normale. Se un linguaggio ha sia un AOT Compiler, non bisogna spaventarsi in quanto è il

compilatore normale. Esso farà tutte le compilazioni prima che vada in esecuzione, mentre con JIT alcune cose si fanno durante.

Chiunque abbia una VM, ha anche un JIT Compiler in quanto è la soluzione a velocizzare i linguaggi interpretati. Il JIT è inoltre fondamentale per il C.

1.19 Generazione codice intermedio

Questo non è codice che potremo eseguire ma è piuttosto una struttura logica. Si chiamerà infatti **codice a tre indirizzi** dove l'indirizzo è quello alla tabella dei simboli. Anche il **codice intermedio** è un linguaggio che richiederà la dichiarazione della sintassi.

Quest'ultimo utilizzerà il "go to" (causa dello spaghetti code), con il quale è **molto difficile creare un albero ma non l'albero di esecuzione**. Esso è però utilizzato in quanto **semplifica la struttura dei compilatori e semplifica la vita dei programmatori**.



2.1 Attività dell'analisi lessicale

Nell'analisi lessicale ci sono le regole del linguaggio sorgente: come deve essere fatta la parola, cosa può contenere, come può iniziare (non con numeri)... Tali regole sono state scritte per risolvere 2 problemi:

- **semplicità di processing:** scrivere `3a per` come sono fatti gli analizzatori lessicali verrebbe diviso in 2 parti, ovvero `3` ed `"a"`.
- **interfacciamento umano:** si cerca di fare un linguaggio che sia abbastanza comprensibile

Solitamente, non si fa prima tutta l'analisi di token per poi passare all'analizzatore sintattico, ma in realtà è l'analizzatore sintattico che chiede a quello lessicale di passargli qualche token. Logicamente, però, al parser arriva on demand un flusso di token. Per quest'ultimi, non esistono standard di nomenclatura. L'importante è non avere lo stesso nome per simboli o identificatori differenti. L'analizzatore lessicale è in grado di riconoscere un intero da un reale in quanto le stringhe sono diverse. Da notare che le parole chiave sono riservate, anche se non sempre è stato così. Una cosa simile appesantiva di molto la fase di processing. Non sapendo il contesto non era possibile in fase di analisi lessicale comprendere se fosse una parola chiave o identificatore quel determinato elemento. Da ricordare che si cerca sempre di prendere la stringa più lunga di senso compiuto. A livello implementativo

conviene usare token senza attributi. L'analizzatore lessicale è un classificatore in quanto il suo compito è diviso in due parti:

- **riconoscitore:** riconoscere le sequenze di caratteri che hanno un senso
- **classificazione:** dopo riconosciute, deve classificarle.

Tali fasi possono essere fatte anche contemporaneamente, non necessariamente in modo sequenziale.

2.1.1 Tipologia dei token

L'analisi lessicale caccia quindi fuori dei token. Essi possono essere **singleton o composti**. Ogni linguaggio ha **gruppi di elementi** come: **identificatori, operatori binari...** Partendo da questo concetto, posso decidere se per ogni classe do un token o se per ogni elemento di quella classe dò un token. Se però dò il nome alla classe e in quella classe ho più di un elemento, allora devo accompagnare il token ad un attributo. In questo caso, l'attributo è un riferimento alla tabella pertanto non perdo informazioni. In questo processo perderò solo informazioni come spazi, commenti...

2.2 Scelta dei sottoalberi e realizzazione degli AST

Tutte le **parole chiave** (come **if** o **while**) che ci sono nei token **verranno perse durante l'analisi sintattica**. Questo perché avrò delle regole che mi indicheranno quella parola chiave da cosa è formata e pertanto potrò costruire un sottoalbero per tale costrutto del linguaggio. La radice mi andrà ad identificare il costrutto di cui sto parlando. Ciò mi permetterà di poter lasciarmi alle spalle le parole chiave. Il Syntax Analyzer è influenzato dalle regole sintattiche del linguaggio e dalla struttura che vogliamo dare agli alberi. Essi potrebbero essere binari, ternari, quaternari... **La forma dell'albero non è standard in quanto ogni compilatore si fa il suo AST**. Importante è che **ogni sottoalbero rappresenta un costrutto del linguaggio, taggato con il nome del costrutto**. Chi scrive l'albero sarà poi lo stesso a leggerlo quindi dovrà tenere conto di tutte le scelte fatte in fase di realizzazione dell'AST. **Prima di riportare la modifica sotto forma di albero, andiamo a controllare se la struttura della frase sia corretta**. Se è corretta e rispetta le regole, procede poi con altre modifiche. Importante è che **quando leggo la radice so cosa aspettarmi nel sottoalbero**. Anche il **“;”** può essere importante, in quanto può funzionare o come separatore fra istruzioni (in questo

caso ad esempio è inutile all'ultima riga) o come **conclusione di istruzione** (in questo caso è necessario anche all'ultima riga)

Partendo dall'analisi lessicale, l'**analizzatore sintattico legge il flusso di token e identifica categorie sintattiche** come: **funzioni, dichiarazioni...**

2.3 Tabella dei simboli e tabella delle stringhe

Differenza tra le due è che la **tabella delle stringhe** è composta solo da stringa ed **identificativo**, mentre la **tabella dei simboli** parte da una tabella delle stringhe per poi aggiungere **informazioni**. Se ad esempio una **variabile compare più volte**, essa sarà **ripetuta una sola volta nella tabella delle stringhe**. Essendo che però **tale variabile potrebbe essere definita una volta come stringa, una volta come int...** esse non saranno la stessa variabile ma bensì **variabili differenti con stesso nome**. La **tabella dei simboli** riporterà questa informazione e questa differenziazione, mentre **nella tabella delle stringhe il nome della variabile sarà riportata una volta sola**. Nella tabella dei simboli o delle stringhe non compariranno mai parole chiave, questo perché sono identificatori già noti. **Il compilatore mentre elabora tutte le parole tiene traccia solo delle parole inventate dal programmatore, in quanto delle parole chiave sa già tutto.**

2.3.1 Identificazione dei tipi nell'analisi semantica

Nell'analisi semantica **si parte l'AST, che rappresenta una sintesi del codice iniziale**. L'analisi semantica **mi arricchisce la tabella dei simboli con il tipo per ogni nome e l'albero viene esteso o arricchito**. Per capire che l'albero sia corretto con i tipi partiamo dai figli. In **presenza di operatori controlliamo le regole semantiche ed assegniamo il tipo e così via salendo**. Effettuiamo una visita bottom up

2.4 Fasi dell'analisi semantica

L'analisi semantica **può fare operazioni di type checking, controllo del main...** Essa consiste in **visite e si può effettuare un numero indefinito di visite**. Prima di analizzare le istruzioni posso:

- **calcolare tutti i tipi dei nomi inventati dal programmatore: vado esclusivamente ad analizzare i nodi DECL OP (nodi di dichiarazione)**. In questa fase **imparo a capire quali sono i tipi**. Questa visita viene fatta partendo dalla radice. In questa fase **entro**

nei nodi DECL e sulla base dei figli di questo sottoalbero sarò in grado di capire il tipo del dato o la firma della funzione. Durante tale analisi si associa all'identificatore nella tabella dei simboli il relativo tipo

- **effettuare type checking:** vado a controllare che i tipi che ho visto siano ben usati. Una volta assegnati i tipi agli identificatori, bisogna vedere se siano stati usati bene. Controllo quindi l'uso dei tipi. **Tale attività si fa bottom up propagando il tipo dai figli al padre. Tutti i token (che coincidono con le foglie) avranno un tipo. In questo controllo si parte quindi dalle foglie e si dà un tipo a tutti i nodi interni. Viene dato un tipo in base alle regole semantiche.**

Più in generale, **dovrei fare l'analisi semantica di tutto il file e non della singola funzione, questo in quanto integrando i file .h le firme di tali metodi verranno inseriti nel mio file** (se sono in C ad esempio). Mettere qualcosa nella tabella dei simboli significa che **poi posso usarla. Se io la uso ma non c'è nella tabella dei simboli, questo produrrà un errore. Alcuni controlli fatti in questa fase potrebbero essere integrati nell'analisi sintattica, andandola però a complicare.**

2.5 Dall'analisi semantica al codice intermedio

Si visita poi l'albero producendo un array. Esso sarà il codice intermedio. Prima ognuno si inventava il proprio codice intermedio, mentre **ora ci sono degli standard.** I codici intermedi, come LLVM IR, **possono essere accompagnati da framework che permette di generare codice per molte piattaforme.** Generalmente però **chi scrive un compilatore completo deve sapere chi sia il target.**

Il codice intermedio non è codice che potremo eseguire ma è piuttosto una struttura logica. Sarà però un linguaggio che richiederà la dichiarazione della sintassi.

3.1 Fasi del ciclo di vita di un linguaggio

Durante i vari momenti del ciclo di vita di un linguaggio di programmazione (a partire dalla definizione del linguaggio fino all'esecuzione dei suoi programmi) **ciascun elemento del linguaggio è legato a caratteristiche di un certo attore**. Dato un linguaggio, **ci sarà chi lo pensa, chi ne scrive il manuale, chi scrive il compilatore del linguaggio...** Nello specifico, le fasi sono:

- **definizione di un linguaggio**
- **implementazione di un linguaggio, ovvero la scrittura del compilatore**
- **uso del linguaggio**. Fino alla fase precedente avevamo visto solo **chi definiva il linguaggio e chi lo implementava**. Da questa fase in poi **escono fuori anche i programmatori**.

Le fasi sono:

- **scrittura del programma**
- **compilazione o traduzione del programma**
- **linking e caricamento in memoria del programma oggetto**
- **esecuzione del programma**

Tutto ciò che avviene subito prima dell'esecuzione avviene in modo statico, ovvero viene fatta una sola volta ed è invariabile. Tutto ciò che avviene dall'esecuzione in poi è runtime.

Il **linking/caricamento** è un aspetto statico perchè il file viene collegato con `printf` ed altre librerie, viene caricato in memoria ma ancora non viene eseguito. Dal momento in cui parte l'esecuzione ci troveremo in uno stato dinamico in quanto può cambiare la memoria in cui viene eseguito ed altre cose

3.2 Attori del linguaggio

Essi sono:

- **creatore del linguaggio**, collegato alla definizione del linguaggio
- **sviluppatore del compilatore**, collegato all'implementazione del linguaggio
- **programmatore**, collegato alla scrittura del programma
- **compilatore/traduttore**, collegato alla compilazione/traduzione
- **SO** collegato al caricamento mentre il **linking** è ad opera del compilatore
- **Processore** collegato all'esecuzione del programma eseguibile

In ogni momento del ciclo di vita ci sono dei legami tra elementi del linguaggio e caratteristiche o proprietà da parte di un certo attore.

3.3 Concetto di binding

In ogni momento del ciclo di vita c'è un **legame** come ad esempio il **collegamento tra variabile e allocazione di memoria** (che è una **caratteristica o proprietà di tale variabile**). Il **legame tra una variabile e la locazione di memoria** a cui fa riferimento viene chiamato **legame o binding**. Va notato che un linguaggio di programmazione potrebbe permettere o meno variabili globali, non potrebbe permettere la ricorsione (questo è possibile se ad ogni funzione si alloca un blocco di memoria statico, in quanto si andava a sovrascrivere sempre la stessa memoria)... **Un linguaggio può quindi scegliere di fare binding in un modo diverso rispetto all'altro**. Ad ogni momento del ciclo di vita: **elementi di un linguaggio di programmazione o di un suo programma sono legati a delle caratteristiche e delle scelte vengono definite**

In Pascal il simbolo `+` è legato all'operazione di addizione a tempo di definizione. Quando implemento il linguaggio devo implementare del codice che mi permetta di fare

somma o concatenazione, pertanto quando implemento il compilatore e allo stesso modo scopro se serve ad una somma o alla compilazione durante l'analisi semantica. Il simbolo + viene legato ad un'operazione in fase scrittura del codice e il valore finale del simbolo viene dato a tempo di esecuzione

3.3.1 Esempi di bindings

- la parola chiave **if** è legata (bound) al concetto di condizione a tempo di definizione del linguaggio.
- La scelta di dove memorizzare i parametri di una funzione è fatta a tempo di implementazione di un linguaggio. Se l'allocazione è statica lo faccio nel mio blocco, se invece è a stack ogni volta chiamo la funzione i suoi parametri andranno su una nuova area di memoria. Colui che decide se fare un'allocazione statica o dinamica è colui che crea il compilatore. Se decide di usare uno stack deve scrivere poi il codice per la gestione della memoria. Chi scrive il compilatore non solo traduce il codice del programmatore ma deve anche costruire un runtime environment (insieme di codici che gestiscono le risorse per permettere l'esecuzione del programma). Aggiunge al codice del programmatore il proprio codice di gestione. Il compilato conterrà quindi sia il codice del programmatore che il codice di gestione
- Il legame fra una variabile ed il suo tipo è definito a tempo di scrittura del programma (in linguaggi tipati senza inferenze di tipo) e realizzato (bound) a tempo di traduzione dal compilatore (analisi semantica)
- Il legame fra una variabile ed il suo indirizzo relativo è realizzato (bound) a tempo di traduzione dal compilatore. Se una variabile è statica sarà relativo rispetto allo 0, se dinamica, chi scrive il compilatore dovrà scrivere la gestione dello stack. Invece di mettere un numero (cosa che avviene per la statica) bisognerà scrivere un programma che calcola l'indirizzo di quando ci sarà la chiamata allo stack. Sarà quindi il compilatore ad occuparsene
- Il legame fra una variabile ed il suo indirizzo assoluto è realizzato (bound) a tempo di caricamento del programma in memoria
- Il legame fra una variabile ed il suo valore (ovvero quando una variabile prende un valore) è realizzato (bound) a tempo di esecuzione. In alcuni linguaggi ci sono operazioni tipo "const int x=10" in questo caso il legame tra variabile e valore è fatta

dal programmatore e realizzata dal compilatore. Quando vado a real time, quella x vale già 10, pertanto è un'operazione statica in quanto prima dell'esecuzione so già il suo valore.

3.4 Concetto di Binding time

Il binding time è **quando avviene il binding, ovvero quando ad esempio la variabile viene legata all'allocazione di memoria**. In tutto questo ciclo di vita tale legame avviene:

- **quando si carica il programma se la variabile è globale**, in quanto quando **carico il programma ho già l'indirizzo statico della variabile**. Durante l'esecuzione del programma la posizione in memoria della variabile non cambia. Il **compilatore darà un indirizzo che è però rilocabile, differente da quello vero in quanto il compilatore non sa dove il programma andrà eseguito**. Ad esempio, se però il blocco di memoria parte da 0 il compilatore allora dirà di mettere la variabile globale in posizione 10. Durante il caricamento, se il blocco di memoria parte da 100, l'allocazione della variabile globale sarà 104. Nelle funzioni ricorsive, essendo che chiamo gli stessi nomi e chiamando le stesse variabili, il nome non potrà essere collegato solo ad una variabile altrimenti ci sarebbe una riscrittura cancellando il valore di prima. C'è un'allocazione di memoria al volo sullo stack che viene fatta a runtime. La variabile avrà tanti indirizzi a seconda dell'esecuzione del programma ricorsivo
- **quando si è a runtime se la variabile è locale**.

3.5 Pro e contro dei bindings

Parlando di binding si parla di:

- **early binding (a tempo di compilazione)**: i legami li facciamo quanto prima possibile. Gli early binding sono i linguaggi compilati in quanto cercano di legare la variabile al tipo, alla memoria. . . La maggior parte dei bindings avviene a tempo di compilazione e ne restano pochi a tempo di esecuzione portando a maggiore efficienza. Il vantaggio è l'efficienza in quanto se so già da prima cosa il codice vuole fare riesco ad organizzarmi meglio.
- **late binding (a tempo di esecuzione)**: quanto più tardi possibile. I linguaggi interpretati sono tutti late binding in quanto viene fatto all'ultimo minuto. Solo quando

sto eseguendo scopro il tipo di una variabile e così via. La maggior parte dei bindings avviene a tempo di esecuzione portando a maggiore inefficienza ma maggiore flessibilità.

Non esiste un linguaggio solo compilato o solo interpretato. I nuovi linguaggi e loro implementazioni stanno eliminando il gap grazie anche ai JIT Compiler. Arrivare a tempo di esecuzione vuol dire arrivare all'ultimo, ovvero un late binding, mentre la compilazione è un early binding in quanto lo facciamo prima, cerchiamo di fare tutto prima. Quello che sta avvenendo oggi è che molti linguaggi di programmazione non sono early o late binding, ma misti. Typescript è tipizzato pertanto si è passati da un linguaggio interpretato con il Javascript in uno compilato con il Typescript. L'interprete alloca la memoria a tempo di esecuzione, mentre il compilatore alloca la memoria a tempo di compilazione. A tempo di compilazione so già l'indirizzo relativo della variabile. L'allocazione della memoria vera e propria avverrà a tempo di esecuzione. Il motivo che := sia stato scelto al posto di = è perchè il programmatore è pigro. Con la JIT Compiler mentre si esegue, si compila anche il codice in modo tale che non c'è necessità di reinterpretare del codice ma possiamo direttamente chiamarlo.

3.6 Da linguaggio compilato ad interpretato

Seppur meno efficienti, i linguaggi interpretati sono usati in quanto ci sono meno cose a cui pensare, tipo la definizione di tipo delle variabili. In un linguaggio compilato siamo obbligati a specificare il tipo della variabile mentre nei linguaggi interpretati no. Anche in molti linguaggi compilati non è necessario specificare precisamente il tipo, come Java che ha introdotto il tipo "var y". Ciò però non significa che il linguaggio è interpretato ma che il compilatore è più complesso, in quanto dovrà inferire il tipo di y. Nei linguaggi compilati non può essere effettuata operazione come "x=1" e poi alla riga dopo "x='ciao'" mentre con i linguaggi interpretati è possibile in quanto questo tipo di linguaggio non fa inferenza sui tipi a priori ma la fa al volo. Quello che si sta cercando di fare è prendere linguaggi interpretati e renderli compilati senza però cambiare la sintassi del linguaggio. In linea di massima è più semplice compilare un linguaggio che interpretarlo, poichè il codice dà più informazioni pertanto è possibile compilare il codice più facilmente, mentre l'interpretato è complesso in quanto dovremo ricavare informazioni run time. Si dovrebbe cercare di eseguire il programma prima per cercare di capire tipi ed altro

4.1 Introduzione all'analisi lessicale

Si parte dal programma sorgente, scritto in un qualsiasi linguaggio e visto come una sequenza di caratteri. La prima fase è quella dell'analisi lessicale. Quest'ultima si occupa di rispondere all'analizzatore sintattico, chiamato anche Parser. Quello che fa il parser è fare una chiamata getNextToken() al Lexer e gli viene restituito un token, che può essere un singleton o composto.

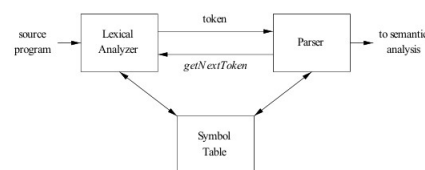


Figura 4.1: Analizzatore lessicale (Lexer)

L'analizzatore lessicale, detto anche **Lexer**, prende il flusso di caratteri dal programma sorgente e deve riconoscere le parole, definiti anche come lessemi. Il Parser ad un certo punto avrà bisogno di un input ed invece di leggere tale input dal file, demanda tale attività all'analizzatore lessicale. Esso costruisce il token e lo passa poi al Parser, che produrrà un AST. La richiesta dei token da parte del Parser viene fatta on demand e non successivamente all'analisi lessicale. Si avrà poi la tabella dei simboli in cui verranno inseriti i lessemi, ovvero la sequenza di caratteri che è stata riconosciuta. Anche le costanti devono essere scritte nella tabella dei simboli e se sto lavorando in C devo applicare la

funzione atoi() per passare da costante ad intero. Il lessema verrà poi tradotto in decimale, ottale... sulla base del linguaggio.

4.2 Fasi dell'analisi lessicale

Esse sono:

- **scanning**: fase in cui si **effettua la rimozione di commenti e unione di spazi consecutivi in uno solo**. Tali operazioni non richiedono la tokenizzazione.
- **analisi lessicale**: in questa fase **si producono token**

Più in generale, quello che fa il lexer è **prendere una stringa e vedere a quale famiglia essa appartenga, per poi assegnare un token**. Per tutti i nomi inventati dal programmatore andiamo ad inserire qualcosa nella tabella che possa identificarli. Da ricordare che i token diventano le foglie dell'albero sintattico

4.3 Famiglie dell'analisi lessicale

Famiglie in cui classificare le famiglie di stringhe sono:

- **commenti e spazi**: per essi però **non vengono riportati token**
- **parole chiave**: esse **dipendono dal linguaggio**, ma sono if, then, while... (il loro token è la **parola chiave stessa**)
- **operatori**: essi sono uguale, più, moltiplicazione, esponente, divisione
- **simboli di punteggiatura**: essi sono parentesi, punto e virgola, virgola (il loro token è spesso il loro simbolo)

Tali famiglie sono spesso indicati con un solo carattere o usando il nome stesso come token.

Altre famiglie sono: **identificatori**: in essi **rientrano tutti i nomi che non sono del linguaggio e che sono inventati dal programmatore**. Essi **hanno bisogno di un attributo**, pertanto avranno <nome famiglia, numero identificativo>. Con tale coppia **bisogna fare poi riferimento alla tabella delle stringhe, questo perché la coppia deve identificare univocamente il lessema**, non devo perdere informazioni. **costanti**: esse possono essere **valori numerici o stringhe**, come "a", "ciao," 5

I token sono i nomi logici che identificano i lessemi. Nel caso in cui ci sia “int x float x” finchè le due variabili sono in scope differenti non è un errore in quanto non è una dichiarazione della variabile. Nel caso in cui fossero nello stesso scope avremmo problemi in fase di analisi semantica, questo perchè operazioni di type checking e scoping è un qualcosa che viene controllato durante l'analisi semantica. Al contempo però, all'interno della tabella dei simboli ci sarà una sola x con puntatore a due tipi differenti. Di volta in volta scorriamo la tabella dei simboli e vediamo se il lessema che vogliamo inserire c'è già. I token diventano le foglie dell'albero sintattico. Quando definiamo un nuovo linguaggio di programmazione, dobbiamo definire la notazione, per fare in modo che possa scrivere dei programmi in tale linguaggio. A partire da questa notazione dobbiamo definire le classi o famiglie

4.4 L'importanza dei pattern

L'analizzatore lessicale scandisce i caratteri e cerca di comprendere di quale famiglia facciano parte. Dato il codice sorgente e riconosciuto il lessema, mediante pattern andiamo a capire di quale classe faccia parte il lessema e si definisce poi il token. Chi inventa il linguaggio definisce anche le regole per riconoscere le famiglie. Una volta che abbiamo i pattern posso fare l'analizzatore lessicale. Leggo i pattern e faccio il riconoscimento. A seconda della famiglia uso un token singleton o token con attributo inserendolo nella tabella. I pattern verranno forniti da colui che definisce il linguaggio

4.5 jflex() ed il suo utilizzo

Ogni espressione regolare può diventare un diagramma di transizione (automa). Tale passaggio possiamo farlo manualmente o in modo automatico mediante jflex(). Esso è un compilatore che prende in input un pattern e produce un analizzatore lessicale. Esso non è un analizzatore lessicale ma un generatore di analizzatore lessicale partendo da un file dei pattern. Per usarlo dobbiamo quindi programmare tale file contenente i pattern (file ad alto livello) ed una volta fatto, lo diamo in input a jflex() da in output l'analizzatore lessicale. jflex() si chiama così perchè produce codice java

4.6 Famiglie e pattern

4.6.1 Famiglie: Parole chiave, operatori e simboli di punteggiatura

Il lessema “if” matcha if come pattern che introduce il token IF, allo stesso modo il lessema “then” fa il match del pattern then introduce il token THEN. Per le parole chiave, le espressioni regolari sono banali, infatti l’espressione regola per il token IF è if. Lo stesso ragionamento può essere fatto con gli operatori. Per i simboli di punteggiatura, invece il lessema “;” fa match con il pattern ; che introduce il token SEMI o il token “;”. L’analizzatore lessicale cerca la parola più lunga che fa match con un pattern, a prescindere dalla categoria. Se un lessema rispetta più pattern, il lessema andrà a far parte della categoria in cui i pattern più lungo viene rispettato.

4.6.2 Famiglia: identificatori

Un esempio di espressione regolare è $[a-zA-Z0-9_]^+$. Tutto ciò che è tra [] indica un solo carattere e si potrà prendere un qualsiasi elemento contenuto al suo interno. Così facendo però verrebbe accettato anche: 1ab_3c. Per ovviare al problema di poter mettere come primo carattere un numero, l’espressione regolare che si usa è: `identificatore = [a-zA-Z_][a-zA-Z0-9_]*`. Importante è non mettere spazi, in quanto se vengono inseriti essi possono essere messi nella parola. Possiamo anche finire `letter_ = a|b|..|z|A...|Z|_` e definire: `identificatore = letter (letter_|digit)*`. Dare nomi come “letter” o “digit” viene chiamato **definizione regolare e semplifica la scrittura di pattern**.

4.6.3 Famiglia: costanti

Per le costanti, esse possono essere 5, 5.3, “ciao”, True, False, null... Allo stesso modo degli identificatori, possiamo definire `digit = 0|...|9` e facendo `intero = digit*`, vorrebbe dire che come intero posso mettere 00004. Ciò non sempre viene accettato dai linguaggi pertanto si può usare `digit0 = 1|...|9`, che non usa lo 0. Per assicurarci che però possa inserire solo lo 0, posso fare: `intero = (digit0 digit*)|0`. Tali annotazioni si possono definire anche mediante espressioni grammaticali. Consiglio è di leggere la specifica per qualsiasi linguaggio si vuole imparare.

4.7 Lessemi e operazioni sui lessemi

Il compilatore **parte dal codice sorgente, riconosce i lessemi** (il lessema è il **gruppo di caratteri che ha un senso e che è stata riconosciuta tramite pattern**). Una volta che è riconosciuto il lessema **o quest'ultimo si esclude quel lessema** (nel caso di spazi o commenti), **o si restituisce un token singleton o si restituisce un token con attributo. Non esistono 2 lessemi che hanno stesso token.** In generale, abbiamo tre elementi: **il lessema che viene riconosciuto, il pattern che riconosce il lessema** (realizzato mediante espressioni regolari) **ed un token.** In Java oltre ad avere un'espressione generale sarà necessario anche un look ahead

4.8 Espressioni regolari

Le **espressioni regolari dipendono da un alfabeto.** Un alfabeto potrebbe essere {a b c } e l'espressione regolare $a^*b^+c^?$. **Oltre alle espressioni regolari noi possiamo usare anche le definizioni regolari che permettono di programmare meglio.** Si passa da **espressione regolare alla definizione regolare dando un nome all'espressione regolare.** Nell'alfabeto il carattere "a" potrebbe essere rappresentato da un ASCII, mentre in Java si usa l'UNICODE, ovvero un'estensione dell'ASCII. **L'alfabeto cambierà a seconda delle esigenze**

4.9 Simboli dell'espressione regolare

Oltre al singolo simbolo e all'espressione di concatenazione, spesso omessa, altri simboli sono:

- Il simbolo * indica 0 o più
- Il simbolo + indica 1 o più
- Il simbolo ? indica 0 o 1
- Il simbolo | indica OR. **Esso rappresenta l'unione di due linguaggi e nel caso in cui la stringa sia riconosciuta da entrambe le espressioni regolari non è importante chi la riconosce.** In questo caso però si potrebbe **ottimizzare le espressioni regolari per evitare ridondanza**

è quindi necessario **definire il linguaggio delle stringhe generato dall'espressione regolare.** Le **stringhe che fanno match con l'espressione regolare fanno parte del linguaggio.** Le stringhe del linguaggio saranno le stringhe che **rispettano l'espressione regolare che verrà**

scritta sulla base dell'alfabeto. **Esempio:** l'espressione regolare $a^*b^+c?$ riconosce: aaabc, abc, b, bc, ab. Invece, aaa non è riconosciuto in quanto b deve sempre esserci per come è fatta l'espressione regolare. In questo esempio essendo * indica 0 o più, il linguaggio è infinito

4.10 Espressioni regolari ed altre notazioni

In un'altra notazione la `[]` indica la classe di carattere. Mettere `[abc]` è equivalente a: `(a|b|c)`. Ciò che è tra `[]` indica un singolo carattere. Operatori aggiuntivi sono:

- Nel caso all'interno delle `[]` venga messo `^`, allora possiamo selezionare tutto l'alfabeto tranne quello che è tra `[]`.
- Al contempo `^` fuori dalle `[]` indica l'inizio della stringa.
- Nel caso voglio imporre la fine della stringa devo mettere il `$`
- Il simbolo `-` usato con `x-y` indica che prendo dall'elemento `x` all'elemento `y`

Va notato che con `^` a non si sta indicando forzatamente che il lessema inizi con a, la stessa cosa vale per `c$`. Non si sta quindi imponendo che il lessema finisca per c. Si sta solo dicendo che il lessema deve essere all'inizio o alla fine di una stringa. Inoltre, se si ha sia il `$` che `^`, allora vuol dire che su di una linea deve esserci solo la parola riconosciuta. Il simbolo `^` riconosce solo l'inizio riga (inizio stringa), quindi anche se ci sono più lessemi sulla stessa riga esso non si preoccupa di riconoscere le parole successive. Possiamo suddividere in blocchi la stringa e prendere l'elemento che ci serve Java usa la libreria regex per riconoscere l'espressione. Le `[]` sono state introdotte da lex i primi generatori automatici di riconoscitori lessicali **Esempio:** `^a*b^+c?$`. In questo caso sto imponendo che il lessema riconosciuto sia all'inizio della stringa ed alla fine della stringa. Data questa espressione regolare `babcb`, `abc` non è riconosciuta in quanto la stringa non inizia con il lessema `abc`. `babcb` non è riconosciuta invece perchè dopo la `b` non può esserci una `a`. Un altro esempio pratico è che con `sp*$` sono interessato a cercare tutti i caratteri spazio che finiscono la linea. Ogni qual volta che vedo questo, posso fare un'operazione

4.11 Differenza tra analizzatore e riconoscitore lessicale

L'analizzatore deve produrre non solo il riconoscimento ma anche il token se necessario, mentre il riconoscitore lessicale deve rispondere solo "sì" e "no", si occupa solo di

riconoscere. L'analizzatore lessicale quindi, oltre a fare il lavoro del riconoscitore lessicale si occupa anche di assegnare i token ed inserire gli identificatori nella tabella delle stringhe. L'analizzatore lessicale è specifico del linguaggio di programmazione mentre il riconoscitore lessicale possiamo usarlo in qualsiasi codice.

4.12 Riconoscimento delle sottostringhe

Nel parsing dobbiamo riconoscere un lessema alla volta, quindi dobbiamo riconoscere anche le sottostringhe. Ciò differisce dalla teoria in cui non ci si interessa di riconoscere sottostringhe. Le sottostringhe nel pratico rappresenteranno lessemi che verranno tradotti un token. Si parla quindi di matching parziale sull'intera stringa. Quando l'analizzatore sintattico chiede un token, si iniziano a riconoscere i lessemi. Il parser potrebbe poi richiedere altri token e nel caso in cui ci siano simboli non conosciuti allora si dà errore.

4.13 Il tool Regex

Regex è uno strumento e non un compilatore. Il compilatore si avvale di strumenti che sono molto potenti. Esso è un tool che data un'espressione regolare indica se i lessemi fanno parte o meno del linguaggio. Definita un'espressione regolare, Regex scrive anche il codice in Java per implementarlo. Fornisce inoltre informazioni sul perché un lessema non sia stato riconosciuto. Con la caratteristica "multiplo" Regex mi permette di analizzare stringhe separate da spazi mentre con "multi linea" riconosce più righe. I compilatori utilizzano degli strumenti come regex, che è solo riconoscitore. In Java, si parte da un file, mediante espressione regolare si riconosce il lessema e se trova qualcosa dice su cosa ha fatto match.

4.14 Scorrere la stringa: forward e begin

Forward e begin sono due puntatori alla stringa. In fase iniziale, il Parser chiede il prossimo token al Lexer. Il lexer inizia a leggere: forward si muove in avanti e inizia a leggere, cercando di riconoscere un lessema che rispetti il pattern. Continua ad andare avanti fin quando il lessema non riconosce più nessun pattern. In questa situazione, fa l'operazione di retrack tornando al carattere precedente e definisce i caratteri tra begin e forward (estremi inclusi) come un lessema. Si effettuano le operazioni di tokenizzazione

necessarie e successivamente, begin e forward punteranno al carattere su cui forward ha fatto retrack. Nel caso in cui il lessema sia riconosciuto da 2 pattern, esso verrà riconosciuto dal pattern definito prima. Proprio per questo motivo, la famiglia degli identificatori e dei letterali va scritto alla fine. Tra begin e forward avrò il mio lessema. In buona sostanza, il forward va avanti fin quando non fallisce. Non appena fallisce fa retrack al carattere in cui non aveva fallito. L'idea è che forward va avanti fin quanto la parola fa match con un pattern. Non appena vede che la parola non fa match con nessun pattern fa retrack e torna ad uno stato in cui almeno un pattern era valido. Nel caso in cui valgono due regole si prende quella che riconosce il pattern più lungo. Se però i due pattern sono della stessa lunghezza si prende la regola che viene prima

4.15 Regole e tabella dei simboli

In alcuni casi, quello che facciamo è evitare di scrivere le regole legate alle parole chiave ma precarica tutte le parole chiave nella tabella dei simboli o in una tabella a parte. Ogni volta che legge un lessema va prima a vedere se esiste nella sezione della tabella dedicata alle parole chiave. Se esiste lì dentro allora essa non può essere un identificatore, restituendo il token della parola chiave. Se non c'è vado nella tabella dei simboli e se è presente restituisco il puntatore altrimenti aggiungo il lessema e ritorno il puntatore

4.16 Compilatori bloccanti e non bloccanti

Trovato un errore, dipende da come è stato programmato il compilatore il fatto che lui si fermi o meno. Se il compilatore è bloccante allora trovato l'errore si blocca, mentre se non è bloccante si inventa il token <ERROR> passando il problema all'analizzatore sintattico

4.17 Espressioni regolari alternative

Altro modo per definire number, id, letter... sono:

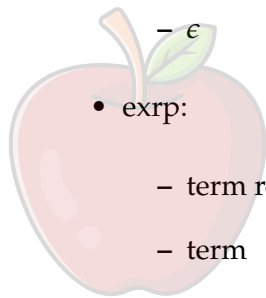
- **number** = digits (. digits)? (E [+]? digits)?
- **letter** = [A-Za-z]
- **id** = letter (letter | digits)*
- **relop** = < | > | <= | >= | = | <>

- **white space** = (blank | tab | newline)⁺. Tale espressione regolare serve per mangiare i whitespace senza che si restituisca i token

4.18 Grammatica dell'analisi sintattica

Mentre per l'analisi lessicale usiamo le espressioni regolari o definizioni regolari, per l'analisi sintattica usiamo le grammatiche. Mentre per l'analisi lessicale **definiamo come trovare le singole parole qui andremo a definire come sono fatte le frasi**. Non si parte però dalle parole del file sorgente, ma dalle **parole gestite dall'analisi lessicale**. **Quest'ultima manda al Parser un token**. Si descrivono quindi le frasi in funzione dei token. Esempi sono:

- stmt:
 - if expr then stmt
 - if expr then stmt else stmt
 - ϵ
- expr:
 - term relop term
 - term
- term
 - id
 - number



CoScienze
Associazione

Da notare che **stmt** è ricorsiva, in quanto al posto di **stmt** della frase “if expr then stmt” potresti mettere di nuovo “if expr then stmt”. Potrebbe anche non esserci uno statement e ciò succede se si usa la sintassi **;;**. Mentre **stmt** introduce il concetto **expr**, esso introduce a sua volta il termine **relop**. Esso rappresenta i possibili simboli di operazione. Il **relop** è il token ottenuto dall'analisi lessicale leggendo un lessema che rappresenta un operatore. Il parser non vedrà più il lessema ma vedrà il token “relop”. A sua volta, **expr** introduce il termine **term** che può essere un **id** o un **number**. Essi rappresentano i token dei lessemi. Nell'analisi sintattica uso i token che vengono mandati dal Lexer, astraendo le sequenze di caratteri.

4.19 Funzionamento del Lexer

L'analizzatore lessicale si può implementare in due modi differenti. Durante il riconoscimento del lessema, quello che succede è che il forward va avanti e quando riconosce un lessema. Sposta il forward sul primo carattere non consumato e si porta con sé il begin. Da notare che il lessema è una stringa riconosciuta da un pattern. Non sempre il lessema introduce un token, come nel caso dei white spaces. **L'approccio manuale della scrittura di un analizzatore lessicale non è usata nei reali compilatori, in quanto gli sviluppatori si affidano a generatori di analizzatori lessicali.** Il processo mentale però per l'analizzatore fatto a mano o automatico deve essere simile:

- definire le classi ed associare i token
- definire le espressioni regolari
- visualizzare le espressioni. Su quest'ultime si possono effettuare ottimizzazioni

4.20 Diagrammi di transizione

Parte complessa è riconoscere la regola. Per semplificare, possiamo visualizzare le espressioni regolari mediante diagrammi di transizione, simili ad automi. Negli automi però, non esistono operazioni come quella di retract. Ci sono però stato iniziale e stati finali. La struttura è simile ad un automa a stati finiti. Per ogni token che devo restituire posso fare il diagramma di transizione. Se però nello stato iniziale del diagramma non vedo niente che faccia match con il carattere che sto leggendo, devo passare al diagramma di transizione successivo. Così facendo andrò a scorrere tutti i diagrammi di transizione. Ciò viene fatto fin quando un diagramma di transizione accetta il lessema o i diagrammi finiscono. Ogni volta che vado in uno stato viene inviata una richiesta del prossimo simbolo perché bisogna fare una transizione. Una volta arrivato il simbolo, capisco come muovermi. A livello di codice ogni automa può essere rappresentato da uno switch. Nel case dello switch gestisco il nodo con archi uscenti. Ogni nodo è un case

L'automa degli spazi non restituisce token ma è comunque rappresentato. La sequenza di delimitatori finisce quando incontro un qualsiasi cosa che non sia un delimitatore. Anche in questo caso si va allo stato di fine e poi si fa retract, non tornando però nessun token.

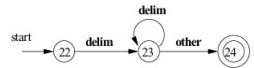


Figura 4.2: Diagramma di transizione per i white spaces

4.20.1 Esempio: Diagramma di transizione per relop

Voglio realizzare il diagramma di transizione che riconosce relop. Per fare ciò devo basarmi sulla tabella delle parole conosciute dal linguaggio. La tabella è presentata in foto

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any <i>ws</i>	–	–
if	if	–
then	then	–
else	else	–
Any <i>id</i>	id	Pointer to table entry
Any <i>number</i>	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE

Figura 4.3: Tabella degli operatori

Si parte dallo stato 0. Ci sono 3 simboli in relop che iniziano con < e quindi bisogna capire a quale lessema io faccia riferimento. Abbiamo = che è da solo e > che è prefisso di 2. Se vedo < devo avere 3 archi (cosa che accade con il nodo 1).

Parto da 0 e leggo <, andando in 1. Se leggo poi =, essendo che la combinazione <= non è prefisso di niente non ha senso continuare, pertanto si terminerebbe nello stato 2, restituendo (relop, LE). Se nel linguaggio esistesse <== avrebbe senso andare avanti con il rischio di sbagliare, ma siccome <= è prefisso di se stesso, allora mi fermo e restituisci (relop, LE). Se invece che = incontro > è la stessa cosa. Ciò cambia per 4. Essendo che non posso fermarmi al nodo 1 in quanto < da solo è prefisso di altro potendoci essere lessemi più lunghi, devo continuare. Se successivamente al minore non segue né uguale né maggiore potrebbe esserci un simbolo "other" (che potrebbe essere un numero, uno spazio, un carattere o altro) quindi vado a 4. Mi trovo però a leggere un simbolo in più di quanto avrei voluto, quindi bisogna fare retract. Il simbolo * indica retract quindi dal nodo 4 ritorna al nodo 1 e il lessema termina.

Parto da 0. Invece di leggere <, stavolta leggo =. Se mi trovo in questa condizione, siccome nessuno nel mio linguaggio ha un prefisso =, allora vuol dire che necessariamente = sarà un lessema, quindi vado nello stato 5 e restituisco (relop, GE)

Parto da 0. Se leggo >, siccome esso è prefisso >= o solo >, devo capire a quale esso appartenga. Leggo il prossimo simbolo e se è =, finisco il lessema >=, se non fosse stato

uguale, avrei preso solo > come e sarei andato nel nodo 8. Alternativamente, avrei fatto il retract e preso un simbolo in più

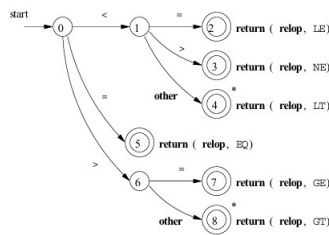


Figura 4.4: Diagramma di transizione per gli operatori

4.20.2 Esempio: diagramma di transizione per gli id o parole chiave

Un identificatore è fatto da una lettera e poi 0 o più lettere o digits. Mi fermo quando c'è qualcosa che non è né letter né digit. Parto da 9 e leggo una lettera e mi sposto a 10. Se continuo a leggere lettere o digit ritorno in 10, altrimenti, se leggo spazio o simboli vado in 11 che accetta. Una volta che sono in 11, mi fa fare il retract perché ho letto un simbolo in più. Mi conservo il lessema, aggiorni i cursori per la prossima parola e restituisco una sequenza di lettere.

Non so ancora se essa sia una parola chiave o meno. Per capire se lo sia uso la funzione getToken(). Mi prendo il token e vado nella tabella. Se il lessema lo trovo nella tabella dei token lo restituisce, altrimenti restituisce l'identificatore. InstallID() viene lanciata solo se il lessema non è una parola chiave. Se era un identificatore, prendo il lessema che mi ero conservato come stinga, lo metto nella tabella e restituisco il puntatore. InstallID viene quindi lanciata solo se riconosco l'id. Se invece stavamo analizzando if, il getToken() avrebbe trovato if nella tabella delle parole chiavi e non avrebbe lanciato installID()

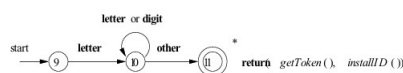


Figura 4.5: Diagramma di transizione per gli id o parole chiave

4.20.3 Esempio: Diagramma di transizione di number

Gli stati 12 e 13 rappresentano un numero intero, ovvero digits+ (un modo per consumare tutti i digits). Se fosse solo 513, dopo aver consumato tutti i digits mi fermerei andando a 20 e facendo retract.

Se invece il numero fosse 513.2, arrivato allo stato 13 mi fermo, leggo il . e vado al nodo 14. Mi leggo un altro numero e finisco in 15. Anche 15 serve a consumare tutti i digits. Se leggo un qualcosa che non sia un numero o "E" vado in 21, faccio retract e ritorno il numero.

Nel caso in cui il numero sia 513E+1, arrivato a 13 e salto da 13 a 16. Leggo poi + e vado a 17, poi a 18 dove inizio a consumare i digits. Quando leggo un qualcosa che non sia un numero vado in 19, faccio retract e torno un numero.

Supponendo di avere 1..3, potrebbe restituire error, oppure potrebbe restituire solo 1 in quanto almeno 1 è riconosciuto come numero

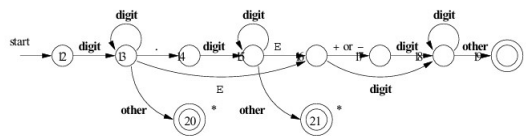


Figura 4.6: Diagramma di transizione per i numeri

4.21 Passaggio tra automi

Quando lancio il Lexer, metto in una variabile globale "state" lo stato iniziale del primo automa, ovvero 0. Se lo stato è 0 vado all'automa con stato iniziale 0 e vedo se ci sono archi uscenti che riconoscono il simbolo che sto leggendo. Se così non è, aggiornò la variabile con lo stato del prossimo diagramma di transizione e così via.

I vari diagrammi potrebbero essere eseguiti tutti in parallelo sul singolo carattere o alternativamente si crea un diagramma che collega tutti i diagrammi di transizione. Si parte sempre ad analizzare dal primo diagramma a scendere. Un carattere fallisce quando tutti gli automi falliscono nel riconoscerlo. Se però il compilatore è non blocking dopo aver fatto ciò si continua ad analizzare i prossimi lessemi. Bisogna fare attenzione però a fare sempre retract. Quando c'è un fallimento non solo dovrà aggiornare lo stato al prossimo diagramma ma dovrà gestire il fatto di riportare lo stato precedente alla visita. Il puntatore begin mi serve per fare un restore (riportare forward al punto di begin). Una volta che una lettura fallisce, bisogna fare il restore del forward. Per questo il cursore di begin non si muove. Oltre alla gestione delle parole chiave dell'identificatore, è possibile creare un diagramma di transizione per ogni parola chiave. Essi vanno messi prima dell'automa dell'identificatore.

Sull'ultimo arco è riportato non letter or digit in quanto se dopo "then" ci fosse una lettera o un numero esso verrebbe riconosciuto come identificatore (CHIEDERE)



Figura 4.7: Diagramma di transizione per la parola chiave then

4.22 Diagrammi di transizione e gestione degli errori

Si potrebbero mettere tutti i nodi fail in uno stato di errore, il quale non deve fare altro che spostare lo stato al prossimo digramma e resettare il cursore forward al cursore begin, facendo il restore. In alcuni compilatori, l'analizzatore lessicale cerca di correggere il lessema, cercando di capire perchè sia stato fatto l'errore. Potrei inoltre saltare l'errore e continuare da dopo, mettendo entrambi i cursori a dopo l'errore. Per fare ciò però dovrei riconoscere un lessema che contiene l'errore mediante pattern sbagliati. Esistono pattern sbagliati usati per simulare gli errori che spesso i programmatori fanno e poter definire specificatamente l'errore.

4.23 Tecniche di distinzione degli identificatori

Non c'è bisogno di mettere nella tabella dei simboli i simboli di punteggiatura. In questa tabella andremo a mettere solo i simboli inventati dal programmatore. Eccezionalmente, per evitare di fare un automa di transizione per ogni parola chiave, facciamo un'altra tabella solo per le parole chiave. Ci sono 2 tecniche per distinguere le parole chiavi dagli identificatori. Se il linguaggio si riserva le parole chiave, ogni volta che trovo if non devo restituire l'identificatore. Le tecniche sono:

- mettiamo una sola regola per identificare e quando riconosco una stringa, vado prima a vedere se è una parola chiave. Se lo è restituisco il token associato, se non lo è restituisco ID e lo installo nella tabella dei simboli. In questa tabella ci sono solo i simboli inventati dal programmatore
- creiamo dei diagrammi di transizione per ogni parola chiave

Logicamente tabella delle parole chiave e tabella delle stringhe sono due cose differenti. Spazi, tab e newline, devono essere comunque consumati dal compilatore senza però che venga restituito un token. Quando si crea un nuovo linguaggio di programmazione si cerca di essere sia più user friendly sia di semplificare la vita al compilatore

5.1 Costruzione dell'analizzatore lessicale

Parte della struttura del compilatore è composta dal Lexer e dal Parser. Il Parser fa una richiesta del prossimo token tramite `getNextToken()` e il Lexer legge da input, restituendo la coppia `<Token, attributo>`, dove l'attributo è opzionale. Il Lexer può essere costruito in due modi:

- mediante programmatore che tramite i diagrammi di transizione costruisce a mano il Lexer
- mediante tool come `jflex()`. L'utente passa a Jflex delle specifiche (regole lessicali) e JFlex si occupa di scrivere l'analizzatore lessicale. Esso è un transpiler in quanto traduce il file delle espressioni in linguaggio target.

5.2 Costruzione dell'analizzatore sintattico

Similmente al Lexer, possiamo:

- scrivere la grammatica ad attributi manualmente
- si scrive una specifica `k` che va in pasto a `javacup()` che produce il parser

5.3 Javacup e JFlex

Chi usa JFlex solitamente usa anche javacup. Si userà JFlex che dovrà generare codice che dovrà essere chiamato dal generato di javacup. Esso deve quindi sapere la funzione che deve chiamare per ottenere il token, in quanto essi sono due sorgenti creati da due tool diversi, necessitando quindi di conoscenza uno dell'altro. Il lessema in tale ottica sarà una variabile globale che verrà condivisa

I token verranno definiti e quando il Lexer invia un token al Parser, quest'ultimo deve sapere quale sia il token. Bisogna mettersi d'accordo sui identificativi dei token. Il parser genererà un file sym in cui saranno dichiarati tutti i token come costanti. Il Lexer dovrà essere a conoscenza in quanto quando restituisce il token dovrà generare un token che è dichiarato nel file sym.

Lexer definisce la funzione nextToken ed il Parser deve chiamare la funzione, dovendosi mettere d'accordo. Altra cosa che condividono è l'identificativo dei token, come il lessema. In un unico punto dovrà esserci scritto che ID è una costante che vale x, NUM, che vale y... e quando Lexer restituisce il valore x allora il Parser capisce che sta restituendo ID. La tabella di corrispondenza dei nomi con i valori la definisce javacup, mediante la creazione di un file chiamato sym.

Chiamo javacup poi chiamo JFlex e le due faranno riferimento alla stessa costante. In generale, JFlex non è usato solo nei compilatori, pertanto deve avere vita propria usando javacup. Se non si usa javacup il nome che JFlex dà alla funzione di richiesta dei token sarà differente

5.4 Costruzione del Lexer

Per costruire un lexer, i passaggi sono:

- studiare le specifiche lessicali del linguaggio
- creare un modello. Il modello può essere o sotto forma di diagrammi di transizione o di un file contenente definizioni regolari
- o si implementano i diagrammi di transizione (caso manuale) o si usa jflex passando in input il file delle definizioni regolari (caso automatico). I due casi sono equivalenti

5.5 Overview di Lex, i precursore di JFlex

JFlex si può usare sia da terminale sia con Maven. Il suo precursore è Lex. Esso è una specifica che produce codice C. Nella specifica Lex mancano le definizioni delle costanti per i token. Lavorando in stand alone, ovvero **non si ha bisogno di un parser con cui interagire, è necessario definire le costanti**. Tutto ciò che è **compreso tra `%{ %}` viene copiato pari pari nell'analizzatore lessicale**. Come nel caso di Lex, dato che si produce codice C, ciò che è all'interno di `%{ %}` **deve essere C**. La sintassi di Lex si divide in varie sezioni, di seguito riportate

5.6 Sezione delle espressioni regolari di Lex

In questo caso:

- **delimitatore** = `[\t\n]`
- **ws** = `{delimitatore}+`. Le vengono usate solo con i nomi delle definizioni regolari, per richiamare delle macro. In questo caso quindi **non sono interessato alle lettere che compongono la parola "delimitatore" ma alla definizione regolare**
- **letter** = singolo carattere
- **id** = `letter letter | digit`
- **carattere punto**: Se sono interessato al carattere `"."` devo mettere `\` davanti, questo perché l'operatore `.` indica qualsiasi carattere

5.7 Sezione delle regole di Lex

Ciò che abbiamo fatto fino ad ora è solo **definire le espressioni regolari**. In questa fase si vanno ad associare espressioni regolari ad azioni che bisogna compiere. L'analizzatore lessicale deve prima riconoscere e poi agire:

- **ws** -> `{/*non fa nulla*/}`
- **if** -> essa viene dichiarata prima con `define` essendo una parola chiave
- **then** -> essa viene dichiarata prima con `define` essendo una parola chiave

- **id** -> devo restituire l'identificatore del token ed il lessema, messo nella variabile globale **yytext**, ovvero la nostra variabile "lessema". L'attributo viene messo nella variabile globale condivisa con il generatore del parser. L'attributo lo mette nella variabile globale ed il token
- **num** -> riconosce il numero e lo restituisce
- In questo caso Lex da un unico token a tutti gli operatori relazioni, dando poi un attributo per far sì che si possano distinguere l'uno dall'altro.

Quando il parser vorrà leggere il lessema potrà farlo dalla variabile globale **yytext** dove si trova il token, la riga e la colonna in cui compare (**yycolumn** ed **yyrow**) e quanto sia lungo. Ciò è utile in quanto siamo abituati a capire a quale riga e quale colonna sia presente l'errore

5.8 Jflex e suddivisione in sezioni: Prima sezione di JFlex

Esso è la versione veloce di Lex che produce codice Java. La specifica che vedremo di JFlex deve lavorare con **cup**. Nello specifico, JFlex è suddiviso in tre parti separate da **%%** e sono:

- **usercode**
- **opzioni e dichiarazioni**
- **regole lessicali**

La prima parte contiene il codice che verrà copiato pari pari nel **Lexer**, come:

- **import** (come quello a **cup.runtime**). La libreria è utile in quanto i simboli dovranno essere poi gestiti dal generato di **Javacup** mentre JFlex ne fa solo uso.

5.9 Seconda sezione di JFlex

Essa contiene:

- **%class lexer**: JFlex genererà la classe java, potendo definire il nome della classe (che in questo caso è **lexer**). Se non definisco il nome uscirà un nome che JFlex ha scelto. Nelle opzioni di classe (**%class**) sto dando il nome alla classe. Posso però mettere anche **%implements** per far sì che la classe ne implementi un'altra o usare **%extends** se voglio che la mia classe ne estenda un'altra

- **%init** nella seconda sezione, **sto scrivendo qualcosa nel costruttore della classe lexer**
- **%unicode** **mi dice che il file che dovrà essere letto è un file contenente codice unicode**
- **%cup** **contiene le opzioni che servono per interagire con cup.** Jflex deve lavorare con il codice generato da javacup. Ad esempio **ci sarà codice che definisce il nome della funzione nextToken().** Grazie a questo %cup si può definire che nome vogliamo. **Il metodo generato all'interno della classe Lexer si chiamerà quindi come vogliamo.** il nome del **metodo che fa lo scanning di default è yylex.** Se voglio cambiare nome devo usare %cup, **cambiando il nome in nextToken().** Mettendo %cup stiamo implicitamente dicendo di implementare `java_cup.runtime.Scanner`, **di chiamare la function next_token,** che il tipo di ritorno della funzione di scanning deve essere `java_cup.runtime.Symbol` e definisce anche la funzione eofval. Esse viene chiamata quando il generat di lexer incontra l'EOF
- **%line e %column sono utili se vogliamo usare le variabili yyline e yycolumn** che sarebbero le **coordinate del lessema che viene riconosciuto.** Se si omette questa parte non potremo usare le variabili.
- **%{ e %}** **indicano il codice che viene scritto pari pari.** A tal fine il codice scritto lì dentro deve essere **java.** Tale codice definisce una **variabile string di tipo StringBuffer** (mi servirà per future operazioni) **e poi usa la funzione Symbol** (che si trova nella libreria di `java.cup.runtime`). La funzione definita **prende l'identificativo del token e restituisce un symbol a java cup in cui viene passato insieme a yyline e yycolumn.** Tale codice finirà in Lexer, con string che diventerà una variabile del Lexer. **Si può ridefinire la funzione Symbol per includere il passaggio dell'attributo del token.** Viene creata una funzione Symbol che non fa altro che restituire un oggetto di tipo symbol

5.9.1 Definizioni regolari di JFlex

Mentre nella prima sezione ci sono le funzioni scritte dal programmatore, nella seconda sezione ci sono le macro, le opzioni ed infine le definizioni regolari. Tali espressioni vengono dalla specifica di Java e definisce:

- **LineTerminator** = `\r|\n|\r\n`
- **InputCharacter** = `[^\r\n]` esso indica un qualsiasi carattere che non sia `\r o \n`

- **WhiteSpace** = {LineTerminator} | [\t\f]

5.9.2 Tipi di commenti in Java

Essi sono di diverso tipo:

- **Comment** = {TraditionalComment} | {EndOfLineComment} | {DocumentationComment}
- **TraditionalComment** = `"/" [*] ~ "*" /` | `"/" [*] "+" /`. In questa espressione non vogliamo che il terzo carattere sia un *. Questo perché se si avesse `/**` diventerebbe un **documentation comment**, che ha un'espressione regolare differente. Bisogna poi aggiungere il caso per `/***/` spesso usato come separatore.
- **EndOfLineComment** = `"/" /` {InputCharacter}* {LineTerminator}? Esso prende qualsiasi carattere in input e poi l'ultima cosa che prende è il line terminator. Così facendo non va oltre la riga.
- **DocumentationComment** = `"/" /**` {CommentContent} `*/" /`.
- **CommentContent** = `([^] | * + [^/])*`. Il commento può essere o un * ripetuto 0 o più volte o altro. Da notare che quando * è all'interno della classe di carattere [] non ha bisogno dello \ per indicarlo, mentre esternamente a [] ho bisogno di usare lo \.

Il simbolo `~` (upto) è utile in quanto consuma tutto finché non vede un carattere. Nel caso del **TraditionalComment** consuma tutto finché non vede `"/`. Ciò è comodo in quanto, avendo un commento lunghissimo, mangia tutto.

5.9.3 Identificatori e digit

Essi sono:

- **Identifier** = `[:jletter:] [:jletterdigit:]*` per fare il match jletter chiama la funzione di Java che riconosce le lettere mentre `:jletterdigit:` riconosce le letter o digit. Non le devo definire io ma sono già all'interno di JFlex
- **DecIntegerLiteral** = `0` | `[1-9][0-9]*` Essa accetta 0 o altri numeri che non iniziano per 0

Gli operatori usati da Jflex sono simili, introduce:

- simbolo `~` (upto), nel pratico `~a` che consuma fin quando non trova la stringa "a" specificata

- **negation !a**, che fa match con tutto tranne la stringa che fa match a

5.10 Sezione delle regole e conflitti con JFlex

Avendo molte regole, possono esserci dei conflitti. Quando due regole sono in conflitto tra di loro si ha una risoluzione, come prendere la regola più lunga o quella definita prima. Ciò potrebbe non piacerci. **Se potessi separare le regole conflittuali, quello che potrei fare è mettere la prima in un gruppo e la seconda in un altro gruppo di regole.** Così facendo, quando lavoro lo faccio su di solo gruppo, riuscendo a rompere il conflitto. Se le soluzioni offerte al conflitto non ci piacciono io posso separare le regole conflittuali, mettendole in due gruppi (dividendole in stati differenti).

5.10.1 Stati ed appartenenza

Si andrà a definire quindi prima la categoria e poi il pattern che bisogna rispettare. Tutti i **pattern** che hanno **x** come prefisso, saranno tutte nello stesso gruppo **x**. Se c'è una regola che fa conflitto potrei metterla nel gruppo **y**. Lo stato è un valore globale che è posto a default a **YYINITIAL**. Per generare il JFlex quando si inizia a lavorare si va nello stato **YYINITIAL**. Lo stato può essere **definito o prima di ogni pattern o può accorpate le regole comprese tra .** Tutte le regole che sono tra **<YYINITIAL>** saranno in tale gruppo e non potranno avere conflitti con le regole di **<STRING>**. La **variabile stato**, infatti, potrà essere solo in uno dei due gruppi per volta. Verranno contattate, inoltre, solo le regole dello stato in cui si è.

5.10.2 Passaggio di stati

è al contempo possibile passare da uno stato all'altro. Si avrà una funzione che si chiama **yybegin()** che quando vede il simbolo **\"** che indica semplicemente **"** fa lo switch da uno stato all'altro, quindi da un gruppo di regole all'altro. **Se mi trovo nello stato YYINITIAL verrà fatto yybegin(STRING) e viceversa.** Una volta che vedo le **"** non so se sto all'inizio o alla fine della stringa. Saprò cosa fare in quanto dipende dallo stato. In questo caso, infatti, entrambi gli stati (gruppi di regole) avranno lo stesso pattern

5.10.3 Gruppo YYINITIAL

Per le parole chiavi si ha:

- `<YYINITIAL> "abstract" { return symbol(sym.ABSTRACT); }`
- `<YYINITIAL> "boolean" { return symbol(sym.BOOLEAN); }`
- `<YYINITIAL> "break" { return symbol(sym.BREAK); }`

Si restituisce la chiamata a **symbol** passando come parametro **sym.parolachiave**. Per identificatori, commenti, letterali, si ha:

- `{Identifier} { return symbol(sym.IDENTIFIER); }` in questo caso non si fa la gestione della tabella dei simboli. la funzione usata restituisce un'istanza della classe `Symbol`
- `{DecIntegerLiteral} { return symbol(sym.INTEGERLITERAL); }`
- `\"` { `string.setLength(0); yybegin(STRING);` } essendo nello stato `YYINITIAL` vuol dire che sto iniziando a vedere una stringa. In questo caso, la variabile `string` è un buffer utile in quanto non riuscirà con un solo riconoscimento a vedere tutto il commento. Quando vede il pattern `"`, si porta dietro la variabile globale `yytext` che contiene il lessema. Averla sempre dietro è utile in quanto in presenza di identificatori bisogna poi metterli nella tabella delle stringhe. Operazione simile potrebbe essere fatta per `Identifier`. Avrò più `yytext` nello stack, uno per ogni regola che viene riconosciuta. Ho quindi bisogno di un buffer dove accumulo li `yytext` che formano il token. Non appena riconosco le `"` metto la lunghezza della stringa a 0 (in modo da vedere la lunghezza del lessema) e passo al gruppo o stato `STRING`
- `"=" { return symbol(sym.EQ); }`
- `"==" { return symbol(sym.EQEQ); }` Si trova in un conflitto in quanto sia `EQ` che `EQEQ` iniziano con lo stesso simbolo
- `"+" { return symbol(sym.PLUS); }`
- `Comment { /* ignore */ }`
- `WhiteSpace { /* ignore */ }`

5.10.4 Gruppo STRING

Essa contiene:

- `\"` { `yybegin(YYINITIAL); return symbol(sym.STRINGLITERAL, string.toString());` }
- In questo caso si fa anche un settaggio globale, che però non blocca l'esecuzione

degli altri comandi. Restituisco poi il token `STRINGLITERAL` passando come valore il lessema. La variabile `yytext` mantiene il lessema della regola riconosciuta. Siccome il lessema che voglio non è un singolo carattere ma tutta la stringa, io ho più regole per riconoscerlo e pertanto ogni volta che viene riconosciuto metto il carattere riconosciuto nella variabile `string` mediante `append(yytext())`. Quello che si restituisce è il token `STRINGLITERAL` e la costante, "" omesse, cambiando ulteriormente stato.

- `\\t { string.append('\\t'); }` per riconoscere `\t` deve mettere tale carattere nella stringa.
- `\\n { string.append('\\n'); }`
- `\\r { string.append('\\r'); }`
- `\\\" { string.append('\\\"'); }`
- `\\\\ { string.append('\\\\'); }`

Infine possiamo avere `[^\\n\\r\\\"\\\\]+ { string.append(yytext()); }`

5.10.5 Stato di default

Lo stato finale è: `[] throw new Error("Illegal character <"+ yytext()+">");` In questo caso entro e finisce. Se io scorro tutte le regole di `YYINITIAL` e falliscono tutte le regole, allora io vado nello stato che accetta tutti i caratteri quindi `[]` e fallisco. Qualsiasi carattere che non è stato catturato sopra verrà catturato da `[]`. Quando si è in tale stato quello che possiamo fare è restituire un messaggio di errore ed il lessema del carattere che mi ha dato problema.

5.11 JFlex ed API

Le API restituiscono valori come il lessema (`yytext`) o la lunghezza del lessema stesso (`yylength`) mentre `yystate` è quella che ha lo stato in cui ci troviamo. La variabile invece `yybegin` ci fa cambiare stato mentre `yypushback` ci fa tornare al carattere precedente

6.1 Introduzione all'analisi sintattica

Fino ad ora avevamo solo il codice sorgente ed il Lexer. A ciò possiamo aggiungere il Parser che deve restituire l'AST. Le specifiche lessicali vengono fuori dal manuale e vengono modellate con espressioni regolari, sia che sia fatto a mano o automaticamente. Il Lexer è quindi composto da una fase di riconoscimento del lessema tramite espressione regolare e una parte di azione, che spara il token inviandolo al Parser

Con il Parser abbiamo una struttura simile: da una parte devo riconoscere la frase e dall'altra devo fare un'azione. Nella fase di riconoscimento delle frasi non mi interessa degli attributi, andando a vedere solo se la frase è sintatticamente corretta. L'insieme di frasi corrette si va a definire tramite grammatica context-free.

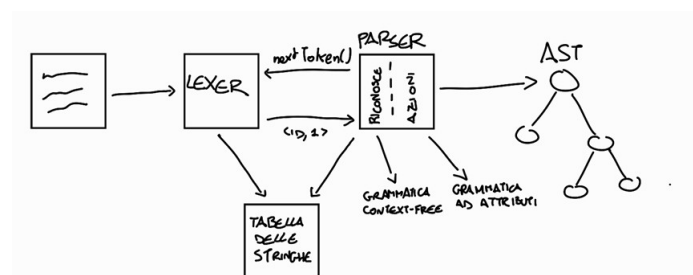


Figura 6.1: Struttura del parser

6.2 Grammatiche ed attributi

La parte del riconoscimento delle frasi nel Parser, lo faccio tramite le grammatiche **context-free** che sono regole come lo erano le espressioni regolari. Accanto ad ogni regola della grammatica, metterò delle azioni, dove l'azione sarà quella di costruire l'albero sintattico. Per le azioni, vedremo che alle grammatiche dobbiamo aggiungere gli attributi, passando dalla grammatica context-free alla grammatica ad attributi. Esso è un formalismo fortemente basato sulle grammatiche context-free

6.3 Albero di derivazione ed AST

Il passaggio dal riconoscimento alle azioni nel Parser avviene tramite un albero, definito **albero di derivazione** o **alberi di parsing**. Essi sono il risultato della grammatica context free. Esso indica le tracce di esecuzione della grammatica. L'albero di derivazione non è un albero fisico come l'AST (per cui c'è una struttura dati). Si potrebbe anche decidere di tradurre l'albero di derivazione nell'AST, ma non succede perchè l'albero di derivazione è molto più grande dell'AST.

All'albero di derivazione o albero di parsing si fa riferimento quando si parla di grammatica. L'AST è una strutturazione gerarchica ad albero del mio programma. L'AST non dipende dalla grammatica ma dipende solo dal programma, mentre l'albero di derivazione dipende dalla grammatica. L'albero di derivazione vive e muore con le grammatiche, in quanto serve solo per creare l'AST

6.4 Grammatiche context free

Essa è un formalismo per rappresentare le frasi. Le espressioni regolari riescono a definire linguaggi infiniti ma sono più grezzi nella definizione. Le grammatiche, essendo più potenti, possono fare tutti i linguaggi regolari (ottenuti dalle espressioni regolari) e coprire anche altri linguaggi per cui le espressioni regolari non riescono

Supponendo di prendere il linguaggio ottenuto da $a^n b^n$, il linguaggio sarà l'insieme di parole descritte dall'espressione regolare. In generale, linguaggi finiti sono facilmente riconoscibili, mentre sono complessi i linguaggi infiniti. Il linguaggio come quello che contiene tutti i programmi scritti in C, è un insieme finito ma che formalmente è indeterminato (in quanto posso aggiungere sempre programmi nuovi)

6.5 Concetti delle grammatiche

Le grammatiche si basano sul concetto di sostituzione e produzione. Vado a definire delle regole, come:

- 0) $S \rightarrow a S$
- 1) $S \rightarrow \epsilon$

Esse sono le regole che posso applicare nella sostituzione. Parto da S che rappresenta il simbolo iniziale e vado a sostituire S . La sostituzione posso farla andando a cercare una produzione che abbia la parte sinistra uguale al simbolo che voglio sostituire. Se esiste, sono libero di sostituire la parte sinistra della produzione con quella destra. Si parla di derivazione quando si parte da un simbolo presente nella parte sinistra della produzione ed arrivo o derivo una sentenza

Esempio: Date le regole definite precedentemente, sicuramente sarò in grado di generare aaa . Avendo solo le due produzioni, quello che faccio è $S \Rightarrow a S \Rightarrow a a S \Rightarrow a a a$. In questa operazione Se voglio raggiungere la sentenza $a a a$, essendo nello stato " $a a a S$ " se andassi ad usare la produzione (1) la S scomparirebbe in quanto andrebbe in ϵ , ovvero la parola vuota. Ciò mi va bene quindi la utilizzo

6.6 Forma sentenziale

Forme come " $a a S$ " dell'esempio precedente che sono a metà di una derivazione prendono il nome di forma sentenziale. Si chiama così in quanto c'è ancora qualcosa da sostituire. In quel caso devo continuare e sostituire la S . In generale, nella sostituzione devo prendere tutto il blocco " $a S$ " e devo sostituire solo la S , in quanto non devo andare a sostituire anche " a ". Il contesto devo portarmelo uguale. Nella derivazione, quando il simbolo vuoto è scritto con un'altra stringa, viene omessa, mentre nel caso il simbolo è da solo esso viene scritto.

6.7 Terminologia della grammatica

La grammatica G può essere vista come una quadrupla $G=(N, T, S, P)$:

- **Non terminali (N):** essi sono i sostituibili. Me ne accorgo che sono sostituibili in quanto si trovano nella parte sinistra della produzione. Esso è definito non terminale in

quanto non deve mai apparire in una sentenza. Se ci appare vuol dire che siamo in una forma sentenziale, pertanto è sostituibile. Nell'esempio il non terminale è {S}

- **Terminali (T):** sono coloro che appaiono nelle sentenze finali e non sono sostituibili. Non si vedrà mai un terminale nella parte sinistra della produzione. Nell'esempio di prima il terminale è {a}. ϵ non conta come terminale in quanto non si vedrà nella sentenza
- **Non terminale iniziale (S):** S si usa come convenzione per rappresentare il non terminale iniziale. Esso è il non terminale standard
- **Produzioni:** esse sono create da un non terminale \rightarrow una stringa di terminale/i e non terminale/i.

Ognuna delle quadruple è un insieme. L'unico operatore che abbiamo è la sostituzione. Ogni volta che c'è un non terminale, lo sostituisco usando la produzione. La sequenza di sostituzioni che parte da un non terminale iniziale e arriva ad una sentenza si chiama **derivazione completa**. Se arriva in una forma sentenziale si chiama **derivazione incompleta** (rimanendo però una derivazione). Il simbolo \Rightarrow indica un passo di derivazione ed è usato nelle derivazioni. Se esiste una derivazione che raggiunge una parola, tale parola farà parte del linguaggio. Le scelte che vengono fatte durante la derivazione possono portare ad una sentenza piuttosto che un'altra. Più sostituzioni mi danno una derivazione. Prendendo l'indice delle produzioni usate per produrre una sentenza (0001 nel caso precedente), esso può rappresentare una codifica della sentenza ottenuta alla fine della derivazione. In una grammatica ben formata le produzioni sono corrette, in una grammatica ben formata non si potrebbero avere problemi con le produzioni. La produzione ha un solo non terminale a sinistra. Nel caso in cui abbiano a sinistra un solo non terminale si parla di **context-free**, se ne ho più di 1 si parla di **grammatica context-sensitive** e nel tipo 0. Nel caso del **context sensitive**, ciò che metto da un lato deve essere minore di ciò che metto in un altro. Nel caso del tipo 0 si ha completa libertà. Nelle grammatiche mal formate possono esistere non terminali che non possono essere sostituiti, o in quanto non sono mai nella parte sinistra di una produzione o perchè creano cicli con altre produzioni

6.8 Linguaggio della grammatica

Un linguaggio generato dalla grammatica G è l'insieme di parole che posso derivare in 1 o più passi partendo da un non terminale iniziale. Se io riesco ad individuare una

derivazione che parte da un non terminale iniziale e termina nella sentenza "aaa", allora tale sentenza fa parte del linguaggio generato da G. $L(G) = \{ w \mid S \Rightarrow^+ w \}$ S indica un non terminale, mentre w è una qualsiasi sentenza che appartiene al linguaggio. Data una parola, se essa è derivabile in 1 o più passi partendo da un non terminale iniziale, allora essa appartiene al linguaggio generato dalla grammatica. Possono esistere anche sentenze che non appartengono a tale linguaggio. Date le produzioni di una grammatica, è facile inferire i terminali e non terminali. Tutti i simboli a sinistra delle produzioni sono non terminali. Possibilità è quello di avere le maiuscole come non terminali, mentre tutte le minuscole come terminali

6.9 Differenza tra espressione e grammatica

Con le espressioni, tutte le parole che fanno match con l'espressione fanno parte del linguaggio di quell'espressione. Nel caso delle grammatiche si parla di derivare e non matchare. Tutte le parole che riesco a derivare partendo da un non terminale fanno parte del linguaggio. Le espressioni regolari non hanno memoria, mentre le grammatiche context-free riescono a ricordare solo una cosa, pertanto anche il caso $a^n b^n c^n$ non può essere risolto mediante grammatiche, mentre $a^n b^n$ sì. Le espressioni regolari non sono abbastanza potenti per modellare alcuni linguaggi, quindi si prova con le grammatiche. Nella produzione, la posizione di S è importante poichè indica dove va sostituito il non terminale

Esempio: Non riesco a definire un'espressione regolare che riconosca tale stringa $a^n b^n$. Al più posso fare $(ab)^*$, ma da ciò otterrei abab... pertanto nell'ordine sbagliato. Non posso farlo in quanto le espressioni regolari non hanno memoria. Le produzioni che mi permettono di leggere questo linguaggio sono $S \rightarrow a S b$ e la seconda $S \rightarrow ab$. In questo modo $S \Rightarrow a S b \Rightarrow aa S bb \Rightarrow aaabbb$.

6.9.1 Dimostrazione della grammatica

Avendo una grammatica e volendo dimostrare che essa possa costruire una specifica stringa come $a^n b^n$, dovrei dimostrare che esiste una derivazione che parte da S ed arriva a $a^n b^n$. Con un'espressione simile devo assicurarmi che le a e b siano in egual numero e che siano nell'ordine giusto. In questo caso lo sono perchè ogni volta che sostituisco S attacco una "a" alla precedente ed una "b" alla successiva. Bisogna anche considerare che il linguaggio genera solo stringhe simili (nel caso sia richiesto) e non generi sia stringhe simili che altre.

Quando si scrive la grammatica, se non abbiamo chiaro il concetto di derivazione, diventa complesso scrivere la grammatica di un linguaggio. La parola vuota non viene considerata un terminale, però verrà considerato il ϵ come fine stringa

6.10 Albero di derivazione

In parallelo alla derivazione e ai passi di derivazione, posso creare un albero di derivazione, che non è altro che una rappresentazione differente della stessa derivazione. Si parte dal non terminale, visto come radice di un albero. Nel mio albero un nodo può essere o un terminale o un non terminale, espandendo il non terminale così come faccio la sostituzione.

Per le forme sentenziali posso considerare le foglie dell'albero. Vado quindi a guardare la frontiera (solo le foglie) ed ottengo o la forma sentenziale o la sentenza. Tale albero viene chiamato albero di derivazione. Nel caso di produzioni complesse, l'albero delle derivazioni potrebbe perdere delle informazioni.

6.11 Grammatiche differenti

Date due grammatiche, esse possono essere equivalenti se generano lo stesso linguaggio ma possono, al contempo, essere differenti in quanto hanno produzioni differenti. Grammatica e linguaggio sono due cose differenti. Per lo stesso linguaggio posso avere milioni di grammatiche che lo producono. Tali grammatiche sono equivalenti ma diverse in quanto hanno produzioni differenti. Alcune grammatiche sono facili da implementare, mentre altre no. Partendo da una grammatica che non può essere implementata nel linguaggio, non è detto che il linguaggio sia intrattabile. Dovrei solo produrre una grammatica migliore che genera lo stesso linguaggio

Esempio: data la grammatica con $S \rightarrow a S B$, $S \rightarrow ab$ e $B \rightarrow b$. Sia questo linguaggio che quello presentato prima, producono stringhe $a^n b^n$ ma hanno produzioni differenti. Iniziando la derivazione si ha $S \Rightarrow a S B$. Si hanno due non terminali e vanno tutti sostituiti se voglio raggiungere una sentenza. Usando la derivazione sinistra diventa $\Rightarrow a a S B B \Rightarrow a a b B B$. Faccio poi $\Rightarrow a a a b b B \Rightarrow a a a a b b b$

6.12 Derivazione e non terminali

Nelle grammatiche context-free, se in una derivazione ho che devo derivare più non terminali contemporaneamente, è ininfluente quale derivo prima, in quanto non cambia il risultato. Si è quindi introdotto il concetto di derivazione sinistra e derivazione destra:

- sinistra: scelgo sempre prima il non terminale sinistro. Nella derivazione sinistra, la sentenza inizia a comparire e si appaia da sinistra
- destra: scelgo sempre prima il non terminale destro.

6.12.1 Non terminali multipli e alberi di derivazione

In un caso in cui nella parte destra della produzione si ha più non terminali, facendo la derivazione e producendo l'albero di derivazione, si perde l'ordine con cui ho generato l'albero. Una volta che ho terminato l'albero, si perde il modo in cui io l'abbia generato. Il risultato dell'albero è lo stesso in quanto la grammatica è context-free. Non so se ho generato l'albero con depth first (da sinistra a destra) o search first. ... La struttura dell'albero rimane però la stessa

6.13 Breve riassunto sulla grammatica

Una grammatica è formata da terminali, non terminali, non terminale iniziale e produzioni. In una grammatica con:

- $S \rightarrow a S$
- $S \rightarrow a$

$\{S\}$ sono i non terminali, $\{a\}$ i terminali, S il non terminale iniziale e le produzioni sono le due presentate. Per usare le grammatiche, l'unico operatore è quello di sostituzione. Una grammatica la creiamo per generare un linguaggio o riconoscere un linguaggio, in linea di massima per caratterizzarlo. Il linguaggio riconosciuto dalla grammatica è a^+ . Le grammatiche, così come le espressioni regolari, hanno il loro successo o sono molto utili nel momento in cui rappresentano linguaggi infiniti, in quanto con poche regole posso avere milioni di frasi (o sentenze). Le regole però non fanno nulla da sole, pertanto devo includere le operazioni, che in questo caso è solo la sostituzione. Parto dal non terminale iniziale, prendo la produzione, vedo dove la S si trova nella parte sinistra e

vado a sostituire con la parte destra. Utilizzo le regole per generare delle derivazioni che partono dal non terminale. Posso fermarmi quando voglio con le sostituzioni. È a tempo di esecuzione che ho il numero infinito di frasi o sentenze che posso generare, ovvero quando applico vado ad applicare la grammatica con l'operazione di sostituzione. In generale, dato un programma, quando lavoro con il programma non lavoro con i lessemi ma bensì con i token, che in questa fase saranno sprovvisti di attributi. I token dell'analisi lessicale diventano l'alfabeto per il linguaggio che deve essere riconosciuto dalla grammatica. Quindi la T della quadrupla della grammatica verrà rappresentata da tutti i token che usiamo.

6.14 Esempio delle grammatiche nell'ambito dei compilatori

6.14.1 Esempio: Partire dalla sentenza accettata

Dato il codice: `if (x > 1) {if (y < 2) {a = 0}}`, non posso generare tale codice con una grammatica così com'è presentata, in quanto con la grammatica posso generare dei token. A tal fine, partendo dalla frase mi genero la sequenza di token: `IF LPAR ID GT NUM RPAR LBRACK IF LPAR ID LT NUM RPAR LBRACK ID ED NUM RBRACK RBRACK`. Da notare che in questa fase non si considerano gli attributi dei token, ma solo gli identificatori. Nella notazione precedente, le lettere maiuscole rappresentavano i non terminali (terminologia standard). Lavorando con il token però, avendo deciso che i token sono in maiuscolo bisogna invertire la cosa.

6.14.2 Esempio: Creare la grammatica che accetta la sentenza

I token dell'analisi lessicale diventano l'alfabeto per il linguaggio che deve essere riconosciuto dalla grammatica. Quindi la T della quadrupla della grammatica verrà rappresentata da tutti i token che usiamo. Tutti i token rappresenteranno i terminali, mentre i non terminali potranno essere degli statement, come `ifStat` o `Stat`:

- 0) `ifStat` -> `IF LPAR Cond RPAR LBRACK Stat RBRACK`
- 1) `Stat` -> `ifStat`
- 2) `Stat` -> `AssignStat`
- 3) ...

Esse definiscono delle produzioni. Nel caso di 0) io ho forzatamente messo le parentesi {}, pertanto se esse non verranno messe in fase di scrittura del codice sarà errore. Al contempo "Cond" e "Stat" sono altri non terminali, in quanto essi possono essere sviluppati a loro volta. Scrivere ad esempio `ifStat -> IF LPAR Cond RPAR LBRACK ifStat RBRACK`, avrebbe forzato la creazione di un linguaggio che è formato esclusivamente da if annidati.

Inserendo come produzione 3) `ifStat -> IF LPAR Cond RPAR Stat`, posso scrivere tramite codice anche degli if il cui body non deve necessariamente essere chiuso tra {}. Così facendo ho due produzioni `ifStat`, una che chiede necessariamente le graffe ed un'altra no. Siccome quando vedo una frase posso applicare o una o l'altra produzione, applicando di solito quella che ha successo, dipenderà dal modo in cui ho scritto il programma quale avrà successo.

L'OR non esiste nelle grammatiche formali ma è un'abbreviazione che può essere usata come nel caso di 1) e 2), scrivendo `Stat -> ifStat | AssignStat`. Più in generale, i non terminali della gramamtica sono `ifStat`, `Cond`, `Stat` ed `AssignStat`

6.14.3 Esempio: Sostituzioni e derivazioni

Per poter generare una frase, dovrò applicare le produzioni. In questo caso, la sequenza di produzioni applicate è: 0 2 0 3, questo perchè:

- La produzione 0 mi genera: `ifStat => IF LPAR Cond RPAR LBRACK Stat RBRACK`
- La produzione 2 mi genera: `=> IF LPAR Cond RPAR LBRACK ifStat RBRACK`
- La produzione 0 mi genera: `=> IF LPAR Cond RPAR LBRACK IF LPAR Cond RPAR LBRACK Stat RBRACK RBRACK`
- La produzione 3 mi genera: `=> IF LPAR Cond RPAR LBRACK IF LPAR Cond RPAR LBRACK AssignStat RBRACK RBRACK`

Nell'esempio in rosso sono segnate le cose che di volta in volta sono state sostituite rispetto l'iterazione precedente.

6.15 Nozioni generali sulle grammatiche

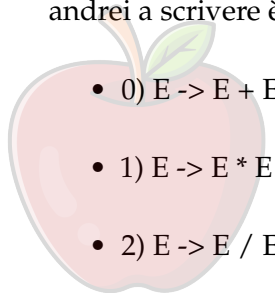
Quando in una grammatica non sappiamo cosa sia un non terminale, lo facciamo diventare token fittizio, ovvero un terminale fittizio. Mentre nell'analisi lessicale l'ordine delle regole era fondamentale, qui le regole verranno prese più volte, quando servono ed

in qualsiasi ordine. Sarà quindi una combinazione di produzioni. Le grammatiche saranno al momento in formato teorico, anche se il passaggio da teorico a quello dei compilatori è semplice, in quanto sia avranno terminali, non terminali, non terminale iniziale e produzioni

6.16 Grammatica per l'espressione aritmetica

Data un'espressione: $a + 4 / b * 3 - i$, voglio scrivere una grammatica per l'espressione aritmetica. La grammatica per tale espressione, dovrà accettare tutte le espressioni aritmetiche sulla somma, moltiplicazioni, divisioni e sottrazioni. Voglio poter rappresentare tutte e 4 le operazioni. La grammatica più semplice è una fortemente ricorsiva, ovvero una che mi dice che un'espressione aritmetica non è altro che la somma o la moltiplicazione o divisione o sottrazione di due altre operazioni. Si sta quindi dicendo che un'espressione non è altro che somma, moltiplicazione, divisione o sottrazione di due altre espressioni. Posso poi dire che lo stesso numero è un'espressione così come identificatore. La grammatica che andrei a scrivere è:

- 0) $E \rightarrow E + E$
- 1) $E \rightarrow E * E$
- 2) $E \rightarrow E / E$
- 3) $E \rightarrow E - E$



CoScienze
Associazione

Una grammatica del genere, per il momento non va da nessuna parte. Se faccio il gioco delle sostituzioni non mi porterebbe da nessuna parte. Quando si scrivono delle grammatiche quello che conviene fare è fare molte derivazioni, in quanto la grammatica vive quando si fanno le derivazioni. Se mi immagino le produzioni applicate in un qualsiasi ordine, allora inizio a ragionare. Posso facilmente scrivere una grammatica se so come viene derivata. Se io facessi una derivazione, la E non la toglierei mai. Da ricordare che la derivazione è un insieme di sostituzioni da una forma sentenziale all'altra fin quando non arrivano alla sentenza finale (la nostra frase). Se vado a sostituire E con gli altri terminali, sarei sempre con la forma sentenziale. Il non terminale E sarebbe un simbolo inutile (così come sarebbe inutile la grammatica). Il non terminale è inutile se non lo si può usare in nessuna derivazione di una frase. Per rendere un terminale utile possiamo dire che E può essere:

- 4) $E \rightarrow id$

- 5) $E \rightarrow \text{num}$
- 6) $E \rightarrow (E)$

Le **tonde mi servono a rompere la precedenza, rompere inteso che mi forza l'ordine delle operazioni**. Da notare che $+$, $-$, $*$ e $/$ sono terminali a cui può essere affidato un nome nel caso in cui il lessema diventi molto lungo

6.16.1 Applicazione di sostituzione

Volendo arrivare alla sentenza $\text{id} + \text{id}$ e volendo applicare la derivazione destra avrò:

- La produzione 0 mi genera: $E \Rightarrow E + E$
- La produzione 4 mi genera: $\Rightarrow E + \text{id}$
- La produzione 4 mi genera: $\Rightarrow \text{id} + \text{id}$

Successivamente alla prima produzione, **ho preso la forma sentenziale più a destra (dato che sto applicando la derivazione destra) e ho sostituito**. La **derivazione è destra se in tutte le forme sentenziali ho preso sempre il non terminale più a destra**. A tale derivazione corrisponde poi un **albero di derivazione**, che non è altro un albero che partendo dal non terminale iniziale, invece che mettere $E \Rightarrow E + E$, mi fa diventare i terminali e non terminali della parte destra della produzione dei figli del nodo radice. Da ricordare che è la **frontiera dell'albero a rappresentarmi la frase finale o la forma sentenziale attiva (frase)**. Tra le due rappresentazioni quasi equivalenti, l'unica cosa che si perde è l'ordine della derivazione, se destra o sinistra

6.17 Esempio di grammatiche ambigue

La grammatica precedente era ambigua. Una grammatica è ambigua se posso fare due derivazioni per ottenere la stessa frase. **Se io posso usare 2 alberi di derivazione diversi ed ottenere la stessa sentenza, allora la grammatica è ambigua**. Volendo realizzare l'espressione: $\text{id} + \text{id} * \text{id}$, possibili derivazioni sono:

- $E \Rightarrow E + E \Rightarrow \text{id} + E \Rightarrow \text{id} + E * E \Rightarrow \text{id} + \text{id} * E \Rightarrow \text{id} + \text{id} * \text{id}$. Le **produzioni utilizzate sono 0 4 1 4 4**. In tale caso si è usato la **left most derivation**. In questo caso l'albero di derivazione suggerisce di fare prima la moltiplicazione durante la visita bottom up, seppur io abbia risolto prima l'addizione

- $E \Rightarrow E * E \Rightarrow E * id \Rightarrow E + E * id \Rightarrow E + id * id \Rightarrow id + id * id$. Le produzioni utilizzate sono 1 4 0 4 4. In questo caso si è usato la right most derivation. In questo caso l'albero di derivazione suggerisce di fare prima l'addizione durante la visita bottom up, seppur io abbia risolto prima la moltiplicazione

Secondo le precedenze che dà la grammatica, in un'operazione $id + id * id$, bisognerebbe prima effettuare la moltiplicazione. Nel caso in cui abbia difficoltà a capire quale produzione utilizzare, si possono passare tutte in rassegna. Seppur la matematica dica di fare prima un'operazione e poi l'altra, la grammatica che stiamo usando non impone nessun vincolo. Nelle grammatiche però, più in generale, possono essere cablate le precedenze

6.18 Definizione formale di grammatica ambigua

Se vado a realizzare l'albero di derivazione sia per la prima che la seconda, scoprirò che ci sono due alberi diversi per la stessa sentenza, questo perché ci sono due derivazioni differenti. Per la stessa frase riconosciuta, la grammatica genera due parse tree. Ogni parse tree è un'interpretazione dell'input

Definizione: Una grammatica è ambigua quando esiste una frase (sentenza) che è riconosciuta dalla grammatica ma genera almeno due alberi di derivazione differenti. Una grammatica è ambigua se esiste almeno una frase che appartiene al linguaggio per cui è possibile costruire almeno due alberi di derivazione.

Definizione: Una grammatica è ambigua quando esiste una frase (sentenza) per cui esistono almeno due derivazioni sinistre (o destre) differenti che la producono

Dire che appartiene al linguaggio potrebbe risultare superfluo in quanto se esistono gli alberi di derivazione per quella sentenza, sicuramente essa appartiene al linguaggio. Il concetto di ambiguità non ci piace in quanto vogliamo che il nostro riconoscitore sia deterministico. Una grammatica che riconosce in più modi una sentenza, darebbe più alberi di derivazione per la stessa frase. Nel nostro caso, una volta potrebbe far scrivere nel compilatore che la somma ha precedenza sulla moltiplicazione e in un'altra che la moltiplicazione ha precedenza sulla somma.

L'idea è di costruire una grammatica che sia non ambigua, in quanto le grammatiche ambigue non sono deterministiche e non possono aiutarci a fare un Parser deterministico. Pertanto, ci conviene avere una grammatica non ambigua. Partendo da una grammatica, per dire se essa è ambigua o meno dobbiamo identificare la frase che abbia due alberi. Quando la grammatica è banale e non c'è ricorsione è semplice definire se un linguaggio è

ambiguo. Il tutto diventa complesso quando inizia la ricorsione. **Dato un linguaggio ed una grammatica, non è detto che io non possa avere un'altra grammatica, anche più semplice, che mi generi lo stesso linguaggio**

6.19 Grammatiche equivalenti e linguaggi ambigui

Due grammatiche sono equivalenti perché generano lo stesso linguaggio. Tali grammatiche però potrebbero essere differenti, in quanto una grammatica potrebbe essere ambigua mentre l'altra no.

Un linguaggio è ambiguo quando tutte le grammatiche collegate a quel linguaggio sono ambigue. Solo perché una grammatica è ambigua, invece, non è detto che il linguaggio di quella grammatica sia ambiguo. Una grammatica ambigua non implica un linguaggio ambiguo, mentre un linguaggio ambiguo implica che una qualsiasi grammatica per esso è ambigua. Da ricordare che la correlazione tra linguaggio e grammatica è molti ad 1, ovvero un linguaggio ha più grammatiche per quel linguaggio e non posso avere le proprietà del linguaggio da una sola grammatica, ovvero non è che il linguaggio ambiguo poiché una sua singola grammatica è ambigua

6.20 Problema del dangling else

Nei linguaggi di programmazione, ci sono state molte discussioni sul problema del **dangling else**. Data la frase: **if E1 then S1 else if E2 then S2 else S3** L'ultimo else dovrebbe essere legato all'if più vicino. Stessa cosa avviene per la dichiarazione di variabili. Con la grammatica:

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

Potrei generare una sentenza **"if E1 then if E2 then S1 else S2"** che può essere costruita mediante due derivazioni, avendo quindi due alberi di derivazione. Essendo ambigua, la grammatica non può essere scritta in questo modo. Alternativa a tale grammatica che costruisce lo stesso linguaggio essendo però non ambigua è:

- `stmt -> matchedstmt`

- $\text{stmt} \rightarrow \text{openstmt}$
- $\text{matchedstmt} \rightarrow \text{if expr then matchedstmt else matchedstmt}$
- $\text{matchedstmt} \rightarrow \text{other}$
- $\text{openstmt} \rightarrow \text{if expr then stmt}$
- $\text{openstmt} \rightarrow \text{if expr then matchedstmt else openstmt}$

La grammatica si è complicata di parecchio. Mentre l'altra era molto semplice, quasi intuitiva, togliendo l'ambiguità la grammatica diventa molto più complessa. Javacup ci permette di dire, al di fuori della grammatica quali siano le precedenze, pertanto viviamo con le grammatiche ambigue

6.21 Risoluzione dell'ambiguità della grammatica aritmetica

Volendo fare $\text{id} + \text{id} * \text{id}$, usando una grammatica che però non sia ambigua, possiamo usare le seguenti produzioni

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

Nel primo caso, **non siamo costretti a partire dalla somma o dalla sottrazione**. Se l'espressione non contiene nessuno dei due prendiamo T e T permetterà di effettuare una divisione o una moltiplicazione. Si considera direttamente l'albero questo perchè dato che l'ambiguità l'abbiamo definita sull'albero, possiamo controllare che ci sia un solo albero di derivazione per $\text{id} + \text{id} * \text{id}$. **Con le produzioni definite non riuscirò a fare più di un albero, pertanto la grammatica non è ambigua**. Per fare ciò non basta fare questo ma bisognerebbe anche dimostrare che non esiste altra frase per cui si ha più di un albero di derivazione ma al momento ci accontentiamo di questo. L'albero **prodotto** **permette la visita che ci piace**, in quanto **facendo una visita bottom up** dobbiamo fare prima la moltiplicazione poi la **somma**. Per la frase $\text{id} * \text{id} + \text{id}$ l'albero di derivazione è:

Allo stesso modo, **tale grammatica mi evita problemi con la sentenza $\text{id} + \text{id} + \text{id}$** , cioè rispetta il fatto che la somma è associativa a sinistra, mentre 2^{2^3} è associativa da destra. A Javacup noi daremo una grammatica ambigua in quanto è più semplice per noi. Oltre

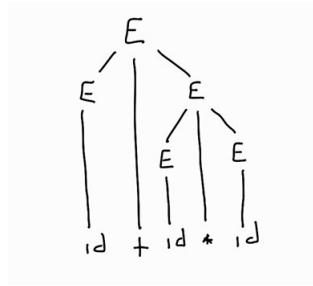


Figura 6.2: Risoluzione dell'ambiguità della grammatica aritmetica

alla grammatica ambigua daremo delle regole di precedenza, come ad esempio il fatto che la somma sia associativa da sinistra. ... Javacup prendendo la grammatica ambigua e tali informazioni, quando si troverà davanti un problema su cui deve decidere come operare, basandosi sulle informazioni aggiuntive, leggerà il modo giusto di continuare. La grammatica dovrebbe essere sempre non ambigua e perfetta, ma scrivere una grammatica non ambigua è complesso

6.22 Ricorsione sinistra e destra

La ricorsione sinistra non ci piace in quanto ci porta a cicli infiniti. Per ricorsione sinistra si intende che il non terminale di sinistra si ripete nella parte destra della produzione come primo carattere:

- $S \rightarrow a S$ rappresenta una ricorsione destra
- $S \rightarrow S a$ rappresenta una ricorsione sinistra

La ricorsione sinistra produce quindi un ciclo infinito. Dato ad esempio:

- 0) $S \rightarrow S a$
- 1) $S \rightarrow a$

Volendo arrivare alla frase *aaa* facendo una depth first da sinistra a destra (derivazione sinistra) quello che succede è che prendiamo la produzione 0 nel primo caso, in quanto siamo fiduciosi che la *a* la raggiungo. Nel secondo step scegliamo di nuovo la produzione 0, perchè siamo ancora fiduciosi e così via.

6.23 Eliminazione della ricorsione sinistra

Quello che devo fare è **eliminare la ricorsione sinistra**. Devo quindi generare una **grammatica equivalente ma senza ricorsione sinistra**. Non dovrò modificare tutta la grammatica ma solo le produzioni che hanno stessa parte sinistra della produzione, nonchè non terminale che causa la ricorsione sinistra, a dover essere cambiata. **Tutte le produzioni che non hanno ricorsione sinistra non devo toccarle**. Data una grammatica con produzioni:

- 0) $S \rightarrow S a$
- 1) $S \rightarrow b A$
- 2) $A \rightarrow a$

Saranno solo le produzioni 0 ed 1 a dover essere cambiate. Applicando tante volte la produzione 0, ottengo $S aaaaa$. Da notare che prima o poi dovrò arrivare alla base di S , in quanto S non è un simbolo inutile. In tale base potranno poi esserci altri non terminali, l'importante è che non ci sia S . Partendo da $S aaaa$ ed applicando la base (ovvero la produzione 1) ottengo $bA aaaaa$. Per tale linguaggio potrebbe però esserci un altro modo per essere generato, un modo che non prevede la ricorsione sinistra. Esempio per questa grammatica è:

- 0) $S \rightarrow bAR$
- 1) $R \rightarrow aR \mid \epsilon$

Ciò che viene dopo la base di S può essere chiamato come R , ovvero resto. Il valore ϵ è presente in quanto la grammatica di prima poteva produrre anche solo la sentenza ba . Le due grammatiche sono quindi equivalenti ma la seconda non ha ricorsione sinistra.

6.24 Regola generale per l'eliminazione della ricorsione sinistra

Data una produzione:

- $S \rightarrow S \alpha$
- $S \rightarrow \beta$

dove α e β sono delle stringhe, α può contenere a sua volta S , ma non ci interessa in quanto le S in α non producono una ricorsione sinistra. Essa potrà quindi essere un terminale

o non terminale, S incluso. β invece non può essere una ricorsione sinistra a sua volta altrimenti dovrebbe essere scritta come $S \gamma$

Ogni volta che troviamo una produzione di questo tipo bisogna iniziare sicuramente con β seguito da un resto. Il resto sarà poi formato dall'interazione di questo α ma R la mettiamo come ricorsione destra. R potrà poi scomparire con epsilon. Nel pratico:

- $S \rightarrow \beta R$
- $R \rightarrow \alpha R \mid \epsilon$

Su alcune grammatiche particolari, non serve applicare questa regola, ma applicandola saremo sicuri che funziona sempre la risoluzione della ricorsione. La regola più generale prevede di partire da: $S \rightarrow S \alpha_1 \mid S \alpha_2 \mid \dots \mid S \alpha_n \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$, dovendo poi giungere ad una forma senza ricorsione sinistra. Va notato che in questo caso, le frasi che posso generare sono β_1 seguite da tutte α_1 , β_1 seguite tutte da α_2 e così via.

6.25 Fattorizzazione a sinistra

Un'altra cosa che non piace avere nelle grammatiche sono due produzioni, con la stessa parte destra e che iniziano allo stesso modo. Il fatto che le parti destre abbiano un prefisso in comune per gli stessi non terminali, è poco efficiente. Ad esempio:

- 0) $S \rightarrow \alpha \beta$
- 1) $S \rightarrow \alpha \gamma$
- 2) $A \rightarrow \alpha x$

Quelle che ci danno fastidio sono solo la produzione 0 e la produzione 1. Vogliamo in questo caso ottimizzare, in quanto in questo esempio avremo S che produce sempre qualcosa che parte per α , che sia presa la produzione 0 o 1. In tal senso, possiamo ottimizzare le produzioni:

- $S \rightarrow \alpha R$
- $R \rightarrow \beta \mid \gamma$

Anche in questo caso si fa uso del resto. Le due grammatiche sono perfettamente equivalenti in quanto generano lo stesso linguaggio ma in modo differente

6.26 Dall'albero di derivazione all'AST

Dato l'albero della frase $id + id * id$ realizzato con la grammatica banale, avremo che la forma dell'albero di derivazione dipende dalla grammatica in quanto l'albero rappresenta le varie produzioni della grammatica che ho usato, e cambiando grammatica e mettendo altre produzioni anche l'albero cambierebbe, avendo sottoalberi differenti. Nel caso la grammatica abbia poche produzioni, l'albero risulterà più compatto, altrimenti può diventare grande in ampiezza o altezza. Da notare che nell'albero di derivazione tutti i nodi intermedi sono non terminali. Da notare che l'albero di derivazione è un trace di esecuzione. Man mano che applica delle produzioni può restituire qualcosa in output che è la costruzione del *syntax tree*. Sarà questo che fa il parser. Riconosce, mentre riconosce costruisce l'albero e mentre costruisce l'albero di derivazione per riconoscere, genera il *syntax tree*. Una frase appartiene al linguaggio se riesco a costruirci sopra un albero di derivazione. Se una grammatica riesce a costruire un albero di derivazione su di una frase, allora la frase appartiene al linguaggio della grammatica. Il nostro obiettivo è, data la grammatica, riconoscere e costruire alberi di derivazione. Il problema attuale è: dato un programma C, riesco a costruire un albero di derivazione secondo la grammatica? Se ci riesco il programma C è sintatticamente corretto

L'AST invece, astrae dalla grammatica ed è concentrato solo sul programma. Non interessa con quale grammatica sia stato costruito il *parse tree* e quanto sia diventato grande l'albero di derivazione. Ci interessa solo cosa rappresenta l'input che è stato riconosciuto. L'albero di derivazione quindi nasce e muore con la grammatica mentre l'AST è una rappresentazione che chi scrive i compilatori si inventa, indipendentemente dalla grammatica. Esse sono equivalenti come informazioni. Posso mettere delle regole nella grammatica che partendo dall'albero di derivazione mi costruisce una forma compatta che è il nostro AST

6.27 Riassunto sulle grammatiche

Ci sono varie tipologie di grammatiche. Per le ben formate è sempre garantito raggiungere i terminali partendo dai non terminali. Abbiamo poi visto le grammatiche ambigue. Una grammatica è tale se esiste almeno una frase per cui è possibile generare la frase stessa con almeno due alberi di derivazione differenti. Quando ci sono anche non terminali, si chiamerà forma sentenziale. Abbiamo le grammatiche ricorsive a sinistra e le grammatiche

che possono essere fattorizzate a sinistra.

$S \rightarrow SS + \mid SS * \mid a$ è sia derivabile a sinistra che fattorizzabile a sinistra. Essa contiene una notazione postfissa, ovvero una è una notazione che pone l'operatore dopo. Questa grammatica genera le postfisse, per cui invece di avere gli operatori nel mezzo si hanno dopo.

Una grammatica è ambigua se esiste una frase per cui è possibile costruire due alberi di derivazione o se ha due derivazioni sinistre (o destre) diverse. La derivazione sinistra è una visita da sinistra a destra depth first di un albero. Due alberi che hanno visite di questo tipo diverse, saranno sicuramente diversi. Se le leftmost sono diverse, sono diversi anche gli alberi. In generale, durante una derivazione per essere sinistra o destra devo sempre sviluppare il terminale a sinistra o destra, però il primo passaggio posso anche prendere una parte della stringa che sta a destra. La fattorizzazione mi garantisce una maggiore efficienza, non totale ma maggiore. Nella definizione formale di ambiguità del linguaggio, è importante specificare se siano right most o left most le derivazioni, in quanto in generale si possono avere più derivazioni per la stessa sentenza se non si segue una right most o una left most

6.28 Grammatica e cicli infiniti

Data la grammatica con produzioni $S \rightarrow SS + \mid SS * \mid a$ essa è:

- è sicuramente ambigua
- è anche ricorsiva sinistra in realtà sono le produzioni ad essere ricorsive sinistre e non l'intera grammatica. La produzione ad essere ricorsiva sinistra è tale quando il simbolo a sinistra compare come primo simbolo della parte destra. Esse non ci piacciono in quanto nel parser top down potremmo andare in un ciclo infinito
- è fattorizzabile a sinistra. Questo perché se due produzioni hanno stessa parte sinistra ed iniziano con la stessa parte destra allora sono fattorizzabili a sinistra. In generale, va notato che ogni non terminale genera di per sé un linguaggio

6.28.1 Esempio: Ricorsione sinistra ed ambiguità

Data una grammatica:

- $E \rightarrow E + E$
- $E \rightarrow E * E$

- $E \rightarrow id$

Anche essa è una grammatica ambigua in quanto la **sentenza “ $id + id * id$ ” può essere realizzata con due alberi differenti**. Consideriamo **due possibili derivazioni sinistre**:

- $E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$
- $E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$

In questo caso **abbiamo due derivazioni left most diverse che producono la stessa frase**. Nella **definizione formale di ambiguità del linguaggio**, è importante specificare se siano **right most** o **left most** le derivazioni, in quanto in generale si possono avere più derivazioni per la stessa sentenza se non si segue una **right most** o una **left most**. Al fine di renderla meno ambigua si è passati a:

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

Volendo ricostruire la stessa frase di prima, si avrà solo un albero di derivazione, pertanto la **grammatica non sarà più ambigua**. La derivazione è: $E \Rightarrow E + T \Rightarrow E + T * F \Rightarrow E + T * id \Rightarrow E + F * id \Rightarrow E + id * id \Rightarrow T + id * id \Rightarrow F + id * id \Rightarrow id + id * id$. Questa grammatica ha però un problema: è **ricorsiva a sinistra ma non è fattorizzabile a sinistra**. Le produzioni che mi danno problemi sono:

- $E \rightarrow E + T$
- $T \rightarrow T * F$

Partendo con risolvere $E \rightarrow E + T$ faccio: $E \Rightarrow E + T$ la prima volta, **arrivando ad avere alla fine ed applicando la seconda produzione ($E \rightarrow T$) a $\Rightarrow T + T + \dots + T + T$** . Per scrivere una grammatica che mi riconosce questo linguaggio, **sicuramente bisogna iniziare con T**. Devo **scrivere nuove produzioni che però siano equivalenti a quelle date, ovvero mi generano lo stesso linguaggio** (che in questo caso inizia sicuramente con T. Modifichiamo quindi la produzione con:

- $E \rightarrow T E'$
- $E' \rightarrow + T E' \mid \epsilon$ (metto una ricorsione di E' perché $+T$ si può ripetere più di una volta).
Avendo che α è $+T$ e β è T , applicando la formula generale: $S \rightarrow \beta R$ e $R \rightarrow \alpha R \mid \epsilon$, ci

troviamo proprio con quanto fatto. Fare $E \rightarrow T + E$ non bastava in quanto non avrei mai potuto iniziare con una somma

Allo stesso modo posso agire su T avendo che esso mi produce $F^* F^* \dots^* F$ quindi:

- $T \rightarrow F T'$
- $T' \rightarrow^* F T' \mid \epsilon$

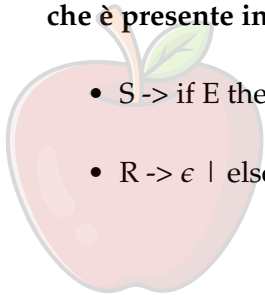
6.28.2 Esempio: Fattorizzazione sinistra

Date le produzioni:

- $S \rightarrow \text{if } E \text{ then } S$
- $S \rightarrow \text{if } E \text{ then } S \text{ else } S$

Si parla di fattorizzazione sinistra in questo caso perché **“if E then S” è un prefisso uguale che è presente in entrambe le produzioni di S**. Possiamo modificarle in:

- $S \rightarrow \text{if } E \text{ then } S R$
- $R \rightarrow \epsilon \mid \text{else } S$



CoScienze
Associazione

7.1 Capacità di riconoscimento del parser

Dall'analizzatore lessicale siamo passati al sintattico. L'analizzatore sintattico può essere diviso in due parti: la prima parte che riconosce e la seconda parte che fa l'azione. Nel nostro caso, nella prima parte abbiamo il parser che produce l'albero di derivazione (in questa parte ci sono anche i non terminali della grammatica) e nella seconda abbiamo il generatore di albero che produrrà l'AST, di dimensioni ridotte rispetto all'albero di derivazione

7.2 Da input all'output del parser

Nel parser sto usando le grammatiche. In input, partendo dall'analizzatore lessicale, arriva al parser un flusso di token su richiesta. Il parser prende la frase (o sentenza) e se riesce a costruirci l'albero allora quella frase è sintatticamente corretta. La grammatica può generare con le derivazioni ma può anche riconoscere, così come le espressioni regolari.

Con la grammatica devo sviluppare il parser. Il parser prende una frase w e su w deve costruire un albero che parte dal non terminale iniziale della grammatica. Se ci riesce, la frase appartiene al linguaggio della grammatica. Il parser fornisce due cose: la risposta che indica l'appartenenza o meno della frase e, se appartiene restituisce, anche l'albero di derivazione che ci dà informazioni sulla derivazione utilizzata. Esso è quindi

una piccola intelligenza artificiale explainable in quanto, indicando le produzioni usate, posso applicare le produzioni e vedere se effettivamente la frase è riconosciuta. Il punto è costruire l'albero di derivazione che in questo caso chiameremo albero di parsing dato che siamo proprio nella fase di parsing.

Data una grammatica e data una frase, il parser deve generare l'albero di derivazione. Il parser da solo non è nulla se non dà la grammatica

7.3 Requisiti del parser

Il parser si basa sulla grammatica. Noi vogliamo fare un parser generalista, nel senso che la grammatica non è incorporata nel parser ma è un input del parser. In alternativa, il parser si costruisce su quella grammatica e il parser non è generalizzabile. Esso prenderà una frase W e se riesce a costruire l'albero di parsing risponderà solo con "sì", altrimenti risponde "no". La prima cosa che deve fare il parser è, data la grammatica ed una frase, vedere se quella frase appartiene a quella grammatica.

7.4 Costruzione dell'albero di parsing

L'albero di parsing può essere costruito o top down oppure bottom up. Al termine della costruzione, l'albero dovrà riconoscere w . Finora, data la frase, avendola raggiunta dal non terminale iniziale abbiamo sempre applicato la tecnica top down. I parser bottom up sono costruiti partendo dalla frase e arrivando alla radice, opposto a come abbiamo fatto fino ad ora. Tipi di parser top down sono:

- parser a discesa ricorsiva: ogni volta che applico una produzione, io devo scegliere un non terminale da cui estendere l'albero. Nell'esempio di prima, parto da E e vedo

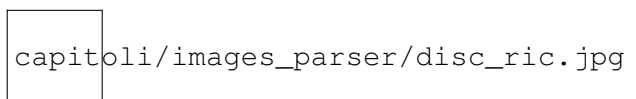


Figura 7.1: Parser a discesa ricorsiva

quali sono le produzioni, potendone essercene più di una. Ciò è una prima forma di non determinismo. Nel parser a discesa ricorsiva è previsto anche il backtrack, pertanto non è deterministico. Se metto il backtrack vuol dire che sviluppo prima il non terminale e solo dopo vedo se sia giusta. Se è quella giusta vado avanti, mentre se non lo è faccio backtrack, tornando indietro e provandone un'altra. Si ha necessità

del backtrack perchè questi parser, dell'input non sanno nulla pertanto, data la grammatica provo una produzione che non è detto sia quella giusta da applicare in quella situazione. Non bisogna partire dal supposto che il parser conosca la frase. Ad ogni passo potrei prendere la produzione che voglio, proprio perchè è non deterministico. Solo arrivato alla foglia vado a vedere se il simbolo corrente della mia derivazione è uguale alla parte dell'input che sto considerando. Se c'è un match tra input e simbolo della foglia, sono contento e vado avanti. Lo scanning dell'input deve essere da sinistra verso destra. Una volta che c'è stato il match sposto il cursore sull'input e consumo il simbolo per cui ho avuto un match. Se ad esempio il prossimo token che vedo nell'albero è * che però non ha match con l'input che è +, **devo tornare indietro alla scelta precedente** vedendo se posso farne un'altra di scelta, **in quanto ho commesso un errore. Parto dal non terminale iniziale prendo una produzione a caso e continuo così.** Quando scelgo la produzione sono cieco, non vedo l'input. Una cosa che potrei fare è usare il look ahead che mi permette di vedere il simbolo successivo a quello che sto considerando sull'input. Il parser a discesa ricorsiva può essere con backtrack e senza backtrack e può avere un look ahead. Il parser a discesa ricorsiva viene fatto con chiamate ricorsive. Il backtrack dipende dalla grammatica e dall'algoritmo che usiamo per la visita. Ciò che non ci piace del backtrack è che visitiamo l'albero più volte. è la grammatica però che mi costringe ad avere backtrack. Una grammatica che per ogni terminale ha solo una produzione sicuramente non avrà backtrack

- **parser predittivo:** altro tipo di parser. Esso usa il look ahead che rappresenta il prossimo simbolo dell'input da consumare. Il parser predittivo ha un input in più rispetto alla discesa ricorsiva con backtrack. Si può fare un parser di discesa ricorsiva anche senza backtrack ma predittivo. Il parser predittivo usa, inoltre, una tabella. Differenza vera e propria tra parser a discesa ricorsiva e predittivo è che il parser predittivo usa una tabella. Il parser predittivo usa il look ahead, usa una tabella ma per come è pensato non ha backtrack. Ci sono algoritmi differenti che permettono di guardare più caratteri durante il look ahead. Più caratteri vede più il programma è potente. Di solito si lavora sul look ahead di 1.

Data una grammatica, **provando a fare l'albero con il parser predittivo, prendo la prima produzione e vado avanti. Se mi fallisce vado alla seconda. Nel caso di una grammatica che ha come prima produzione una ricorsiva sinistra, l'algoritmo che prende la prima**

produzione mi fa andare in loop, in quanto non andrà mai alla seconda produzione

7.5 Requisiti dei parser top down

Non tutte le grammatiche sono adatte per la costruzione di un parser top down. Esse:

- Devono essere ben formate
- Non devono essere ambigue
- Non devono essere ricorsive sinistre
- Devono essere fattorizzate a sinistra


Per un parser recursive descend (a discesa ricorsiva), posso costruire un parser con eventuale backtracking su grammatiche che rispettano necessariamente tutte le regole 1-3, in quanto la 4 è più relativa all'efficienza. Per un parser predittivo, devo necessariamente rispettare tutte le regole 1-4, applicando la fattorizzazione sinistra. Le grammatiche sono componibili in quanto se un linguaggio è sottolinguaggio di un altro, basta includere le produzioni che ci interessano

7.5.1 Esempio: parser a discesa ricorsiva

Data la grammatica:

- $S \rightarrow c A d$
- $A \rightarrow ab$
- $A \rightarrow a$

E una frase $w = "cad"$, devo fare un parser a discesa ricorsiva. In questo esempio, ogni volta che dobbiamo scegliere una produzione, dobbiamo prendere quella che è scritta prima. Dato che nella grammatica non si ha produzione con ricorsione sinistra, non andrà in ciclo. Devo partire dalla radice S dell'albero, che produce i figli c , A e d . L'algoritmo



capitoli/images_parser/es_disc_ric_1.jpg

Figura 7.2: Esempio parser a discesa ricorsiva

va a sinistra e vede che c'è un terminale. Dato che è un terminale vado a vedere se mi fa

match con il simbolo corrente dell'input. Se c'è match sposto il cursore dell'input sul prossimo simbolo. Letto e fatto il match di c, torno indietro e vado ad A. Essendo un non terminale o sviluppo con la produzione e poi scendo nel sottoalbero appena creato. Faccio gli stessi passaggi nel caso siano un terminale o sviluppo se sia un non terminale. Se trovo un simbolo che non fa match allora torno indietro all'ultima scelta fatta, eliminando il sottoalbero e applicando la prossima produzione, se presente. Il cursore sull'input dovrà tornare indietro di tanti simboli pari al numero di terminali visti. Faccio ciò fin quando non finisco. L'input finito terminerà con \$. Il dollaro indica che non c'è altro nell'albero, faccio il match con fine stringa e pertanto mi fermo. Se avessi avuto una grammatica senza la seconda produzione non avrei avuto backtrack nell'albero. Se dò il look ahead ovvero la possibilità di guardare prima il simbolo e poi scegliere, ovvero la predizione, potrei evitare il backtrack, ma non sempre. Fattorizzando la grammatica ottengo:

- $S \rightarrow c A d$
- $A \rightarrow a R$
- $R \rightarrow b \mid \epsilon$

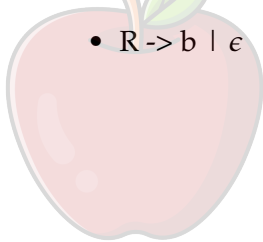


Figura 7.3: Esempio parser a discesa ricorsiva

Da notare che in questo caso il look ahead mi sarebbe servito. Da notare che in questo caso ho fatto backtrack ma meno di prima.

7.6 Codifica dell'algoritmo della discesa ricorsiva

Preso l'esempio di prima, se dovessi codificare farei una funzione per S, ovvero una funzione per il non terminale S. La S è una produzione che, a livello di codice, può essere strutturata come $S(\text{match}('c') A() \text{match}('d'))$. La prima cosa che fa è chiamare la funzione match con il terminale c. Se fa match, chiama la funzione A che rappresenta la produzione A, avendo $A(\text{match}('a') R())$. Il non terminale A dovrà gestire la ricorsione in quanto se c'è un errore bisogna tornare indietro. In generale, $\text{match}('e')$ dà sempre successo. Generalmente, per ogni produzione si scrive una funzione. Il codice ad alto livello è: In questo caso si prende una produzione A a caso. Per ogni elemento della parte destra della

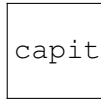
capitoli/images_parser/cod_disc_ric.jpg

Figura 7.4: Codice per il parser a discesa ricorsiva

produzione presa, se è terminale si fa il `match()` andando avanti se il `match` dà risultato corretto, altrimenti si chiama la funzione associata a quel non terminale. Tale algoritmo però non gestisce il `backtrack`. Il codice potrebbe essere modificato andando a chiedere di scorrere ogni produzione `A`, andando a provare per ognuna di esse. Un parser ha una grammatica cablata quando a livello di codice il parser si crea le funzioni sulla base delle produzioni, facendo delle funzioni per ogni terminale di quella grammatica. Le strutture che andremo a costruire sui non terminali restituiscono booleani. All'inizio della prima funzione, si mette il puntatore corrente nella variabile `fallback`. Se la funzione fallisce metterò il puntatore all'interno di `fallback`. Sarà questo a gestirmi la ricorsione, mettendo il puntatore all'inizio se devo annullare un'operazione.

Solitamente, a livello di codice, se ho una produzione composta da più elementi nella parte destra verifico se l'elemento corrente non corrisponda ad un elemento della produzione, mentre se la produzione ha a destra un solo elemento, si verifica se esso sia uguale all'aumento della produzione stessa. A livello di esame si chiede di prendere una grammatica e farla diventare un parser

7.7 Nozioni sui parser

La tecnica `leftmost` è collegata al parsing `top down`, dove `top down` fa riferimento al modo in cui andiamo a costruire l'albero. I terminali sono vincolati dalla frase. Quando abbiamo un linguaggio per cui vogliamo creare un parser, la prima cosa che facciamo è quella di tokenizzare. Quando parliamo di grammatiche omettiamo gli attributi dei token. Nelle grammatiche tradizionali non si mettono gli attributi. Essi però vengono inclusi nelle grammatiche ad attributi. Fino ad ora abbiamo visto un parser non predittivo. Il parser prima genera poi fa il `match` e se va bene continua altrimenti fa `backtrack`. Con il predittivo la produzione non si sceglie a caso ma si va a vedere con quale prossimo carattere devo fare `match`

7.8 Preparazione sulle grammatiche

Se io devo implementare un linguaggio da 0, la prima cosa che devo fare è scrivere la grammatica. Non posso scrivere una qualsiasi grammatica, ma una context free che sia ben formata in modo tale che i non terminali abbiano senso, ma deve essere anche non ambigua.

Se la grammatica è ambigua, il parser richiedere backtrack, pertanto rimuovendola produco un parser più efficiente. Al contempo, però, ottengo anche un parser che sia deterministico. Il problema dell'ambiguità è che crea più alberi. Avere più alberi significa, ad esempio per le espressioni aritmetiche, che avrò strade differenti percorribili, come fare prima somma e poi moltiplicazione e viceversa. In un linguaggio di programmazione non è ammissibile che si possa avere questa incertezza, pertanto bisogna essere sicuri che ad una frase corrisponda solo un'interpretazione. Le grammatiche ambigue non vanno bene proprio perché generano più alberi e pertanto generano più interpretazioni.

Facendo una discesa ricorsiva del parser, bisogna controllare se si hanno ricorsioni sinistre, che potrebbero portare a loop. Quest'ultime vanno pertanto rimosse. Infine, per ottimizzare la grammatica, bisogna fattorizzare la grammatica. Il prefisso che è in comune cerco di eliminarlo e fattorizzarlo. Fatto ciò, ho la grammatica che posso usare per fare il parser. Potrebbe, dopo tutte queste trasformazioni, diventare quasi leggibile rispetto a quella di prima. Rispettando le regole, avremo però un linguaggio equivalente a quello di prima. Partendo da G , ottengo una grammatica G' molto diversa ma equivalente.

7.9 Parser non predittivo e predittivo

Per ogni non terminale vado a fare una funzione che chiama gli altri non terminali o fa match con i terminali della parte destra in modo ricorsivo. Il parser è non predittivo in quanto quando si applica una produzione non si fa aiutare dall'input, ma è cieco. La scelta della prossima produzione avviene alla cieca

Con quello predittivo, invece, andremo a sbirciare l'input, avendo un'informazione in più. Ogni volta che abbiamo informazioni in più il nostro sistema è più potente. Per passare ai parser predittivi abbiamo bisogno di due funzioni: first e follow

7.10 Parser predittivi e loro potenzialità

Tramite **first** e **follow** verrà creata una tabella e in funzione di questa tabella, usando uno **stack**, riusciremo a costruire un parser predittivo deterministico. Data una frase lunga n passi, saprò dire se la frase appartiene, non appartiene e fornendo le derivazioni utilizzate. Avere la lista delle derivazioni è equivalente ad avere l'albero di parsing.

Quando facciamo l'albero di derivazione è possibile fare tanti passi per poi avere comunque un **syntax error**. Se ci fosse stato qualcuno che, prima della derivazione, fosse stato in grado di dirmi se sarei stato capace di arrivare o meno alla frase, mi sarei evitato possibilmente alcuni passi se la frase era raggiungibile o avrei evitato del tutto di fare la produzione se la frase mi avesse prodotto un **syntax error**.

Con i parser predittivi, ad ogni passo, **ogni volta che ho un simbolo, non solo so che c'è speranza di arrivare al simbolo, ma so anche quale sia la produzione che devo applicare**. Il parser predittivo dovrebbe dirmi quale sia la prossima produzione che ha speranza di raggiungere il simbolo dell'input

7.10.1 Esempio: Potenza e limiti dei parser non predittivi

Data una grammatica:

- $S \rightarrow cAD$
- $S \rightarrow A$
- $D \rightarrow Xa$
- $D \rightarrow Ac$
- $X \rightarrow b$
- $A \rightarrow ab$
- $A \rightarrow c$

Supponendo che l'input sia **babcc**, data e partendo dalla radice devo scegliere una **produzione**. Nella discesa ricorsiva non sappiamo quale sia la produzione migliore ma, se lo sapessimo, sarebbe tutto più facile. **Supponendo di non saperlo, inizio a prendere la prima produzione, questo fin quando non trovo un terminale**. Essendo che il **primo terminale della prima produzione è "c"** mentre la stringa inizia per **"b"**, la prima produzione non va bene, facendo **backtrack** e continuando la ricerca. Se non riesco con nessuna produzione a

trovare il terminale che volevo trovare, finendo le possibili scelte, allora viene restituito un syntax error. Supponendo di avere cabcc, la stringa avrebbe successo faccio $S \Rightarrow cAD$. Il primo simbolo fa match con c, potendo poi continuare con A. Nel caso in cui l'input fosse stato abc, ho possibilità di raggiungere il terminale a tramite la produzione $S \rightarrow A$.

7.11 Concetto di First

Se io so che dalla produzione S posso raggiungere il simbolo "a" presente nell'input, allora il simbolo a appartiene $FIRST(S)$. Per ogni non terminale mi calcolo il first di tutti i simboli che quel non terminale può raggiungere. Ciò mi dà una mano perchè ogni volta che arrivo su di un non terminale e vedo quale sia il prossimo simbolo che devo raggiungere, vado a vedere il $FIRST$ di quel non terminale. Se il simbolo appartiene al $FIRST$ del non terminale, allora vado a prendere la produzione corrispondente, mentre se non ci appartiene posso fare syntax error.

Un simbolo non appartiene al $FIRST$ di un non terminale quando non c'è una serie di produzioni che a partire da quel non terminale raggiunge quel simbolo.

Per far sì che a appartiene $FIRST(S)$ S con più derivazioni deve raggiungere a. Se $S \Rightarrow^* \alpha \beta$ allora a appartiene $FIRST(S)$. Il simbolo \Rightarrow^* indica più derivazioni. Per il non terminale S esiste una forma sentenziale che inizia con quel simbolo. Posso applicare questo a tutti i non terminali, non solo al non terminale iniziale

Definizione: Un simbolo appartiene al $FIRST$ di un non terminale se, partendo da quel non terminale, c'è una sequenza di derivazioni che mi raggiunge quel simbolo come simbolo più a sinistra.

Il $FIRST$ di ogni terminale sarà un insieme di elementi. Tali elementi, partendo dal non terminale iniziale e facendo una sequenza di derivazioni saranno raggiungibili come simboli più a sinistra.

7.12 Simboli in first e follow

Generalmente, a noi non interessa il $FIRST$ di un terminale in quanto è esso stesso per definizione, ma ci interessano di più i non terminali. Il $FOLLOW$ viene fatto solo sui non terminali. In generale, il follow può contenere \$ mentre il $FIRST$ può contenere ϵ . L'ultimo non terminale sarà sempre seguito da \$. Si ha anche come regola che il non terminale iniziale è sempre seguito da \$. Da S (non terminale iniziale) vedo tutta la frase e dopo

la frase c'è il \$, pertanto si avrà sempre $S' \rightarrow S \$$. Il FOLLOW non può avere ϵ mentre FIRST non può avere \$, in quanto esso non sarà mai all'interno di una produzione ma solo esternamente, alla fine delle forme sentenziali. Anche nel caso di tutte ϵ non si avrà epsilon in FOLLOW in quanto tale stringa terminerà comunque con \$.

Stesso caso per la forma sentenziale $aB \epsilon$. Il FOLLOW(B) non sarà epsilon in quanto essa non conta ma sarà \$. Mentre il FIRST mi serve per il parser predittivo, il FOLLOW si usa con gli ϵ

7.13 Concetto di Follow

Il FOLLOW mi indica, per quel non terminale, in una forma sentenziale da quale terminale può essere seguito. Nel FOLLOW(A), considerando l'esempio della grammatica di prima, ho: c,b,a appartiene FOLLOW(A). Questo perchè, effettuando più derivazioni si ha:

- $S \Rightarrow cAD \Rightarrow cAXa \Rightarrow cAba$
- $S \Rightarrow cAD \Rightarrow cAAc \Rightarrow cAab$
- $S \Rightarrow cAD \Rightarrow cAc$

Non sempre però è necessario fare tutte le derivazioni. Dal momento in cui A è sempre seguito da D, colui che mi dà idea su cosa viene dopo di A è FIRST(D). Il FOLLOW(A) \subset FIRST(D). Calcolare il FIRST vuol dire andare a fare tutte le derivazioni, mentre con il FOLLOW, avendo già i FIRST diventa più semplice. Nel FOLLOW devo trovare una forma sentenziale

Definizione: Il terminale b è nel FOLLOW(A) se esiste una forma sentenziale raggiungibile da S in cui b segue A, come $S \rightarrow Ab$. Quest'ultima è una forma sentenziale in quanto abbiamo sia non terminali come A che terminali come b

Tenendo in considerazione quanto detto, o diciamo che c,b,a appartiene FOLLOW(A) oppure che FOLLOW(A) = {b,a,c}. Nel FOLLOW(A) vado a mettere tutti quei terminali che possono seguire quel non terminale A in una forma sentenziale

Definizione: Il FOLLOW di un non terminale A, è un insieme di terminali che possono seguire quel non terminale in una qualsiasi forma sentenziale.

In un albero, la forma sentenziale è data da una frontiera, pertanto guardando un albero e la sua frontiera vuol dire che dopo A riuscirò a trovare quel non terminale. Se nella frontiera si trova ϵ , avendo l'esempio della foto, essendo che A va in epsilon avremo come

$\text{FIRST}(B) = x$. Il simbolo epsilon lo scriviamo ma non conta. Se A scompare ovvero va in ϵ , devo aggiungere anche il $\text{FOLLOW}(A)$ al $\text{FIRST}(B)$, non solo $\text{FIRST}(A)$



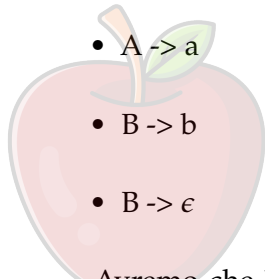
Figura 7.5: Concetto di follow e frontiera

Ci saranno casi in cui, avendo tipo $S \rightarrow EF$, non è vero che $\text{FOLLOW}(E) \subset \text{FIRST}(F)$ proprio a causa di ϵ , dovendo andare a calcolare il FIRST del terminale che mi viene dopo F , oppure fare il FOLLOW del terminale F (non sono certo della cosa)

7.13.1 Esempio: Presenza del valore epsilon

Data la grammatica:

- $S \rightarrow ABc$
- $A \rightarrow a$
- $B \rightarrow b$
- $B \rightarrow \epsilon$



CoScienze
Associazione

Avremo che $\text{FIRST}(S)$ viene calcolato andando a vedere tutte le produzioni di S e vedere la loro parte destra. Dato che il primo simbolo è il non terminale A , tutto ciò che può iniziare A può iniziare anche S . Un qualcosa che sarà nel $\text{FIRST}(A)$ sarà anche nel $\text{FIRST}(S)$. Abbiamo che:

- $S \Rightarrow ABc \Rightarrow aBc$

$\text{FIRST}(A) = \text{FIRST}(S) = \{a\}$. Allo stesso modo $\text{FIRST}(B)$ è banale a sua volta in quanto è $\{b, \epsilon\}$. Calcolando $\text{FOLLOW}(S)$, essendo che non è in nessuna posizione destra della produzione, avremo che $\text{FOLLOW}(S) = \{\$ \}$, questo perché è come se si avesse sempre una produzione teorica $S' \rightarrow S\$$. Per calcolare $\text{FOLLOW}(A)$ devo, partendo da S , cercare forme sentenziali a destra della produzione in cui A si trova. Vogliamo trovare i non terminali che seguono A nelle forme sentenziali. Dopo A ho due possibili produzioni:

- $S \Rightarrow ABc \Rightarrow Abc$
- $S \Rightarrow ABc \Rightarrow A\epsilon c$

Essendo che però ϵ è a metà della frase non devo considerarlo né scriverlo, saltando ed andando al prossimo carattere. Avremo quindi che $\text{FOLLOW}(A) = \{b, c\}$. In questo caso, non è vero che $\text{FOLLOW}(A) \subset \text{FIRST}(B)$ proprio a causa di ϵ , dovendo andare a calcolare il **FIRST** del terminale che mi viene dopo B, oppure fare il **FOLLOW** del terminale B (non certo)

Per i **FIRST** si guarda quali sono le produzioni che hanno come parte sinistra quel non terminale, per i **FOLLOW** si va a vedere in quale produzione quel non terminale appare nella parte destra della produzione

7.14 Calcolo di first

Data una produzione $A \rightarrow X_1 X_2 \dots X_n$, il $\text{FIRST}(X_1 X_2 \dots X_n) = \text{FIRST}(X_1)$ in quanto X_1 viene prima di tutti. Non sempre però è così, in quanto se X_1 può derivare ϵ allora bisogna considerare anche il $\text{FIRST}(X_2)$ e così via. Pertanto:

$$\text{FIRST}(X_1 X_2 \dots X_n) = \text{FIRST}(X_1) \cup_{\epsilon} \text{FIRST}(X_2) \cup_{\epsilon} \dots \cup_{\epsilon} \text{FIRST}(X_n) \cup_{\epsilon} \text{epsilon}$$

\cup_{ϵ} indica che l'unione viene fatta solo se c'è epsilon. Se non c'è ϵ ci si ferma prima. Se tutti gli X_i possono andare in epsilon è necessario aggiungere $\{\epsilon\}$. La catena si fermerà non appena si ha un non ϵ

7.14.1 Esempio: calcolo di First

Supponendo di avere una grammatica:

- $S \rightarrow AB$
- $A \rightarrow a \mid \epsilon$
- $B \rightarrow b \mid \epsilon$

$\text{FIRST}(S) = \text{FIRST}(AB) = \text{FIRST}(A) \cup_{\epsilon} \text{FIRST}(B) \cup_{\epsilon} \{\epsilon\} = \{a\} \cup \{b\} \cup \{\epsilon\}$. In questo caso, essendo che ϵ lo aggiungo già alla fine, $\text{FIRST}(A)$ e $\text{FIRST}(B)$ vanno a prendere tutto tranne ϵ

7.14.2 Notazioni e formalismi su first

Per il **FIRST**, se io ho una produzione come $A \rightarrow X_1 X_2 \dots X_n$ allora $\text{FIRST}(A)$ include $\text{FIRST}(X_1 X_2 \dots X_n)$. Contiene e non è uguale in quanto è possibile ci sia un'altra produzione $A \rightarrow Y_1 Y_2 \dots Y_m$ oltre a quella definita precedentemente. Ognuna di queste produzioni differenti contribuisce poi al $\text{FIRST}(A)$. Se io avessi ad esempio:

- $A \rightarrow a$
- $A \rightarrow b$

Non potrei dire che $FIRST(A) = \{a\}$ ma dovrebbe essere $\{a,b\}$. Da notare che un simbolo fa parte del FIRST di un non terminale solo se il simbolo è derivato dal non terminale stesso

7.15 Calcolo di follow

Per calcolare il follow devo andare a cercare il terminale nella parte destra. Se non trovo un non terminale nella parte destra in nessuna produzione metto $\$$. Se un non terminale è seguita da $\$$, quando vado ad applicare una produzione ad esempio $S \rightarrow cAD\$$, quello che viene dopo (o seguiva) S viene dopo (o seguirà) anche l'ultimo simbolo, ovvero D.

7.15.1 Notazioni e formalismi su follow

Per il FOLLOW, posso dire subito che $\$$ appartiene $FOLLOW(S)$, questo di default. Avendo la produzione che va in $A \rightarrow \alpha B \beta$, anche qui avrò che $FOLLOW(B) \subset FIRST(\beta)$ eccetto ϵ , se $\beta \Rightarrow^* \epsilon$. Se abbiamo ϵ nel $FIRST(\beta)$ lo andiamo a rimuovere dal $FOLLOW(B)$. Lettere greche come α e β indicano stringhe di terminali e non terminali.

Se invece abbiamo la produzione: $A \rightarrow \alpha B$, oppure $A \rightarrow \alpha B \beta$ ma abbiamo che $\beta \Rightarrow^* \epsilon$, allora abbiamo che $FOLLOW(B) \subset FOLLOW(A)$. Questo è ovvio in quanto avendo una forma sentenziale in cui ad un certo punto appare $A: \dots Ax \dots \Rightarrow \dots \alpha Bx \dots$ succede quindi che a B segue x, che seguiva a sua volta A. Segue quindi B tutto ciò che segue A nel caso in cui B sia l'ultimo non terminale o se sia seguito da terminali che però sono ϵ . In questo caso, B rimarrebbe scoperta diventando l'ultima.

Quando si ha ϵ nel mezzo della stringa, avendo $S \rightarrow X_1, X_2, \dots, X_n$, il $FOLLOW(X_1) \subset FIRST(X_2, \dots, X_n) = FIRST(X_2) \cup_{\epsilon} \dots \cup_{\epsilon} FIRST(X_n)$. Se però anche X_n va in ϵ , allora $FOLLOW(X_1) \subset FOLLOW(S)$, dove S è il non terminale che produce la forma sentenziale X_1, X_2, \dots, X_n . (Da controllare l'ultima cosa)

Supponendo che $X_2 \Rightarrow^* \epsilon$, dire che $FOLLOW(X_1) \subset FIRST(X_3)$ o dire che $FOLLOW(X_1) \subset FOLLOW(X_2)$ è la stessa cosa ma conviene usare sempre FIRST o sempre FOLLOW per evitare confusione.

7.16 Utilità di follow e first

Sulle informazioni ottenute dalla tabella posso costruirla un'altra. Con questa seconda tabella, se ho la stringa di n terminali saprò perfettamente quale produzione applicare, avendo l'efficienza massima tramite analisi statica della grammatica. La tabella per il caso precedente potrebbe essere:

Per ogni non terminale si fa una riga e per ogni terminale una colonna. Avendo la coppia (terminale, non terminale) per ogni terminale saprò quale non terminale devo espandere per raggiungerlo. Se non c'è la combinazione allora posso dare syntax error. Avendo la stringa in input $id+\$,$ partendo da "id" nonché primo simbolo della stringa in input, so che esso è raggiungibile partendo da E , pertanto parto da $E \rightarrow TE'$. Riuscirò a mettere tutte le produzioni nella tabella. Con questa tabella troverò subito l'errore, rendendo inutile l'espansione dell'albero, in quanto non si ha speranza di raggiungere un simbolo non specificato. Dove non c'è nessuna produzione avrò syntax error.

Non si costruirà questa tabella sia per il FOLLOW che per il FIRST. Volendo mettere $E' \rightarrow \epsilon$ devo metterlo nei simboli che sono nel FOLLOW(E'). $E' \rightarrow \epsilon$ è una produzione da usare e devo metterla nel FOLLOW(E'), ovvero del padre. Il padre ha $() \$$ pertanto a livello di tabella per raggiungere $)$ e $\$$ userò $E' \rightarrow \epsilon$ o un altro. Quando la parte destra può derivare la parola vuota, allora si mette quella produzione anche nel FOLLOW del simbolo sinistro, ed ecco perchè mi serve il FOLLOW. Mettendo ad esempio per $)$ $T' \rightarrow \epsilon$ o $E' \rightarrow \epsilon$ è comunque giusto in quanto entrambi hanno il simbolo $)$ nel loro FOLLOW

7.17 Riassunto sulle grammatiche

Una grammatica è una quadrupla (N, T, S, P) . Data la grammatica:

- 1) $E \rightarrow TE'$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$

- 8) F-> id

Abbiamo che:

- **non terminali E, E', T, T', F:** non possono comparire in una sentenza. Quando una grammatica è ben formata ha almeno per una produzione per ogni non terminale.
- **terminali +, *, (,), id:** qualunque carattere che non è un non terminale, è un terminale. L'ε non è un terminale, in quanto non esiste, indica la parola vuota
- **Non terminale iniziale:** in questo caso E
- **produzioni:** Le produzioni vengono applicate per sostituire.

Quando viene richiesto un albero top down, si parte dal non terminale iniziale e lo si pone come radice. Si applicano poi le regole o produzioni. Se con esse riusciamo a raggiungere la frase, allora la frase appartiene, altrimenti non appartiene.

7.18 Scelte da effettuare durante la realizzazione dell'albero

Le scelte da effettuare sono:

- **Quale non terminale vado a sviluppare?**
- **Se quel non terminale ha più produzioni, quale produzione vado a prendere?**

I parser non predittivi, per costruire la frase, prendono produzioni alla cieca. Nel caso del predittivo, possiamo guardare di un simbolo ed in base a quello scegliamo. In genere, un albero deve avere tutti i non terminali sviluppati fino alle foglie

7.19 Riassunto su FIRST e FOLLOW

Avendo parser predittivo ho bisogno di FIRST e FOLLOW:

- **FIRST:** applicata ad un terminale mi restituisce il terminale stesso. La funzione FIRST applicata ad un non terminale mi va a vedere tutte le produzioni che partono con quel non terminale e inserisce nel FIRST gli elementi che compaiono come primo elemento di quelle forme sentenziali. FIRST è l'insieme dei terminali dei FIRST che possono iniziare un sottoalbero. Ad esempio, data la foto, id fa parte del FIRST(T) in quanto, dato il sottoalbero di T e guardando la frontiera, il terminale più a sinistra, il

primo simbolo è proprio id. In generale, se ho il FIRST di più simboli, vado a fare il FIRST del primo \cup_ϵ al secondo così via. $\text{FIRST}(TE') = \text{FIRST}(T) \cup_\epsilon \text{FIRST}(E') \cup_\epsilon \epsilon$. Dobbiamo omettere ϵ ed usarla nell'unione finchè non arrivo alla fine o finchè mi fermo perché un FIRST di un non terminale non ha ϵ . La funzione FIRST è calcolabile su più non terminali tramite tale formula. A sua volta $\text{FIRST}(T) = \text{FIRST}(FT')$

- **FOLLOW:** Per il FOLLOW devo andare nell'albero e vedere un non terminale da cosa può essere seguito. Data la frontiera della foto, se T' scompare il prossimo simbolo che segue F è +. In generale, se ho una produzione $A \rightarrow \alpha B \beta$. $\text{FOLLOW}(B)$ contiene $\text{FIRST}(\beta)$. Se però $\beta \Rightarrow^* \epsilon$, ovvero β va in ϵ in più passi di derivazione, diventa $A \rightarrow \alpha B$ ed in questo caso $\text{FOLLOW}(B)$ contiene $\text{FOLLOW}(A)$. Se quello che viene dopo B non c'è o può scomparire e B si ritrova ad essere l'ultimo simbolo, allora si aggiunge il follow del padre, in quanto tutto ciò che segue il padre seguirà anche l'ultimo simbolo della forma sentenziale

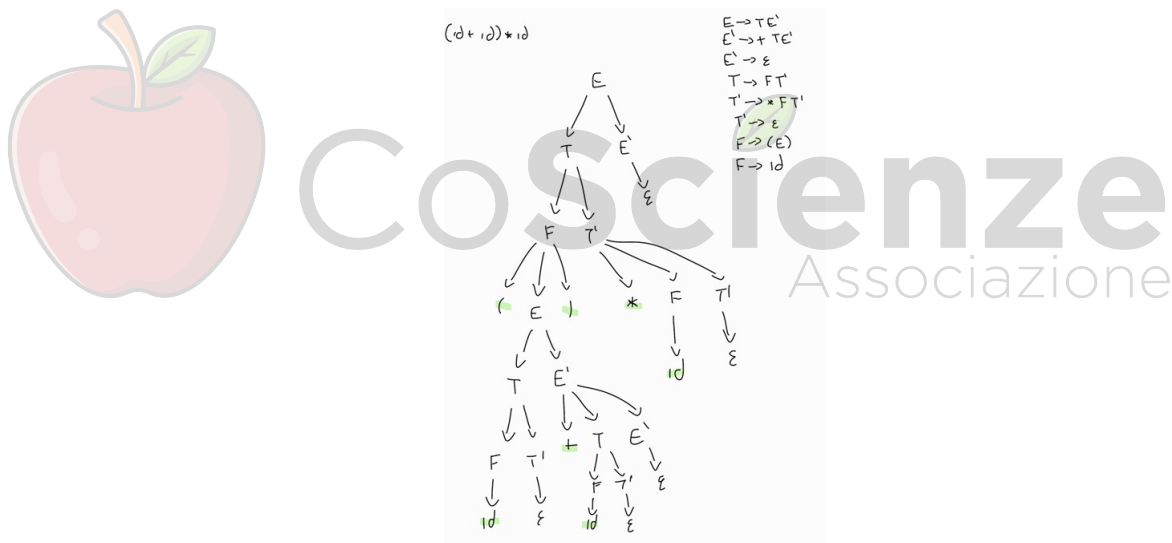


Figura 7.6: Riassunto su FIRST e FOLLOW

7.19.1 Esempio: Calcolo del FOLLOW

Data la grammatica:

- 1) $E \rightarrow TE'$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \epsilon$

- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow * FT'$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow id$

Partendo dai FIRST abbiamo:

- $FIRST(E) = FIRST(TE')$ dato che però T non può raggiungere ϵ in un qualsiasi numero di passi, **allora $FIRST(E) = FIRST(T)$** . Si parla di $=$ e non contiene in quanto E ha solo una produzione.
- $FIRST(E') = \{+, \epsilon\}$
- $FIRST(T) = FIRST(FT')$. Dato che F non va in ϵ ho che $FIRST(T) = FIRST(F)$
- $FIRST(T') = \{*, \epsilon\}$
- $FIRST(F) = \{(, id\}$

Ogni produzione contribuisce al calcolo dei FIRST e dei FOLLOW. Per i FOLLOW:

- $FOLLOW(E)$ è $\{\$, \})$ Essendo il simbolo iniziale, **$\$$ va sempre messo**
- **$FOLLOW(E')$** : Essendo l'ultimo simbolo di E , eredita il $FOLLOW(E)$
- **$FOLLOW(T) = \{+, \$, \})$** . Ciò è dato da **$FIRST(E')$** ed il fatto che annullandosi bisogna prendere $FOLLOW(E')$
- **$FOLLOW(T')$** : esso contiene il **$FOLLOW(T)$** in quanto è ultimo simbolo della forma sentenziale
- **$FOLLOW(F)$** : esso è sempre seguito da T' . Il $FIRST(T') = \{*, \epsilon\}$. Dato che non posso avere l' ϵ nella produzione $T \rightarrow FT'$ T' può scomparire. $FOLLOW(F)$ contiene $FOLLOW(T)$. Effettuo lo stesso ragionamento per la produzione $T' \rightarrow FT'$, questo perchè **$FOLLOW(T)$ potrebbe essere differente da $FOLLOW(T')$** . Devo considerare entrambe le produzioni, seppure F sia seguito in entrambi i casi da T' in quanto, **essendo che T' va in ϵ , F sarà l'ultimo elemento della forma sentenziale pertanto bisogna considerare i FOLLOW di T e T'**

7.20 Parser LL(1)

Un parser top down, dato un non terminale, deve scegliere la produzione. Per farlo in modo intelligente, ovvero con Parser predittivo, si usa il Parser LL(1). La prima L si indica che si sta scandendo l'input da sinistra a destra, la seconda L vuol dire che sto costruendo una derivazione left most. Il simbolo 1, invece, indica di quanti caratteri faccio il look ahead, in questo caso 1. Guardando il non terminale ed un simbolo, deve scegliere la produzione da usare, ovvero devo decidere quale produzione espandere. Tale decisione si traduce in una tabella che contiene:

- terminali
- non terminali
- produzioni

Data la grammatica:

- 1) $E \rightarrow TE'$
- 2) $E' \rightarrow +TE'$
- 3) $E' \rightarrow \epsilon$
- 4) $T \rightarrow FT'$
- 5) $T' \rightarrow *FT'$
- 6) $T' \rightarrow \epsilon$
- 7) $F \rightarrow (E)$
- 8) $F \rightarrow id$

Posso ottenere questa tabella:

	*	+	()	id	\$
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$		$E' \rightarrow \epsilon$		$E' \rightarrow \epsilon$
T			$T \rightarrow FT'$			$T \rightarrow FT'$
T'	$T' \rightarrow \epsilon$			$T' \rightarrow \epsilon$		$T' \rightarrow \epsilon$
F			$F \rightarrow (E)$		$F \rightarrow id$	

L'algoritmo di realizzazione della tabella funziona che prendo una produzione e vado ad inserirla. La produzione viene sicuramente inserita nella riga del non terminale a sinistra della produzione. La colonna in cui inserirla dipende dal FIRST di quel non terminale. Ad esempio, essendo che $\text{FIRST}(E) = \{ (, id \}$, allora la produzione $E \rightarrow TE'$ la metto nella colonna della (e dell'id. Questo significa che se leggo un qualsiasi simbolo che non siano (e id e devo espandere il non terminale E, darò sicuramente syntax error.

In un primo momento si ignorano le produzioni che hanno come parte destra ϵ . Questo perché bisogna calcolare cosa viene dopo le ϵ . Quando durante un albero di derivazione facciamo andare un non terminale in ϵ , quello che spesso succede è che stiamo aspettando un simbolo che quel non terminale non può produrre, pertanto se ho la possibilità lo faccio andare in ϵ e leggo il simbolo dopo. Colui che mi fa leggere il simbolo successivo del corrente è la funzione FOLLOW. Pertanto, per tutte le produzioni la cui parte destra va in ϵ , vado a considerare il FOLLOW del padre. Ad esempio, per $E' \rightarrow \epsilon$, vado a considerare il FOLLOW(E')

7.21 Costruzione formale della tabella LL(1)

Data una produzione $A \rightarrow \alpha$, a livello di tabella tale produzione verrà inserita in $[A, x]$ se x appartiene $\text{FIRST}(\alpha)$. Se però $\alpha \Rightarrow^* \epsilon$, allora tale produzione verrà inserita in $[A, x]$ se x appartiene FOLLOW(A). Ogni volta che abbiamo una produzione, andremo a mettere la produzione nella cella del simbolo x o se x appartiene $\text{FIRST}(\alpha)$ o x appartiene FOLLOW(A) se α va ad ϵ

Per una tabella LL(1), ogni volta che vedo un simbolo avrò solo un'azione da fare, o restituire syntax error o applicare la produzione. Ottenuta la tabella LL(1), la grammatica è cablata nella tabella. Per ogni produzione che troviamo nella grammatica, andiamo a porla nella tabella. Se il risultato è una tabella senza conflitti, ovvero una tabella in cui per ogni cella c'è un'unica azione, allora va bene. Ho una tabella LL(1) con conflitti quando nella stessa cella ho due o più produzioni.

7.22 Regole per evitare i conflitti

Se io ho due produzioni $A \rightarrow \alpha$ e $A \rightarrow \beta$:

- 1) $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$. Essi quindi non devono avere terminali in comune

- 2) Se $\alpha \Rightarrow^* \epsilon$ in 0 o più passi ma β non va in ϵ , $\text{FOLLOW}(A) \cap \text{FIRST}(\beta) = \emptyset$. Se una delle due va in ϵ bisogna allora considerare il $\text{FOLLOW}(A)$ e non più il FIRST di quel non terminale. Se $\text{FOLLOW}(A)$ e ad esempio $\text{FIRST}(\beta)$ hanno un simbolo in comune, allora finiranno nella stessa cella
- 3) Se sia α che β vanno in ϵ , entrambe le produzioni vanno negli stessi terminali del $\text{FOLLOW}(A)$, pertanto la grammatica non sarà LL(1) Se $\alpha \Rightarrow^* \epsilon$ ma anche $\beta \Rightarrow^* \epsilon$, allora avrà $\text{FOLLOW}(A) \cap \text{FOLLOW}(A)$ pertanto l'intersezione sarà sicuramente non vuota. Non posso avere sia α che β che va in \emptyset (Da chiedere)

In generale, quando si costruisce un parser predittivo deterministico è necessario non solo che la grammatica sia ben formata, senza fattorizzazione o ricorsione sinistra e non ambigua, ma deve anche rispettare le regole definite sopra. Con queste regole, potrò costruire un algoritmo deterministico. Una grammatica che rispetta solo il fatto che deve essere ben formata, senza fattorizzazione o ricorsione sinistra e non ambigua allora può essere sviluppata in modo top down. Per poter realizzare però un parser predittivo deterministico è necessario che si rispettino anche le ulteriori 3 regole prima definite

7.23 Passaggi per il controllo di una grammatica

Data la grammatica G , la grammatica G è LL(1)? Per rispondere alla domanda, prima cosa da fare è vedere se la grammatica è ambigua. Se essa è ambigua allora sicuramente non sarà LL(1) in quanto ha più alberi per la stessa sentenza e pertanto non è deterministica. Se la grammatica è ambigua allora ci sarà sicuramente almeno un conflitto nella tabella. Non è però vero il contrario. Se la tabella ha un conflitto non è detto che la grammatica sia ambigua. Se la grammatica è ambigua sicuramente c'è un conflitto, ma se c'è un conflitto non è detto che la grammatica sia ambigua, ma deve magari essere trasformata, magari bisogna fare una fattorizzazione. In questo caso, per rispondere alla domanda, se la grammatica è ambigua basta disegnare due alberi di derivazione per la stessa sentenza. Un linguaggio è LL(1) quando esiste una grammatica LL(1) per quel linguaggio. Se invece una grammatica non è LL(1) non è detto che il linguaggio non sia LL(1)

In generale, per rispondere alla domanda o si costruisce la tabella e si mostra che non ci sono doppie entrate, o si applicano queste 3 regole, andare a vedere tutte le produzioni che hanno lo stesso non terminale a sinistra:

- Se nessun FIRST ha gli ϵ , si va a vedere la prima regola, le altre due non servono

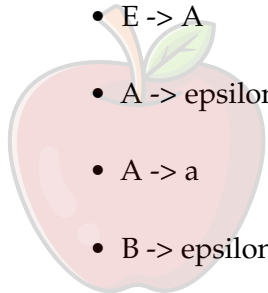
- Se vanno entrambi in ϵ , allora la grammatica non è LL(1)
- Se solo una produzione ha ϵ , vado a fare il FIRST di quella senza epsilon e FOLLOW del padre dell'altra controllando che l'intersezione sia vuota.

In alternativa, dopo aver calcolato FIRST e FOLLOW, invece di applicare le 3 regole, posso utilizzare la tabella LL(1). Se la tabella è senza conflitti o le tre regole sono rispettate, allora la grammatica sarà LL(1). Se la grammatica è LL(1) avremo un parser deterministico senza backtrack, altrimenti sarà con backtrack. Sapendo che la grammatica è LL(1) non c'è bisogno di backtrack

7.23.1 Esempio: Tabelle con o senza conflitti

Data una grammatica:

- $E \rightarrow AB$
- $E \rightarrow A$
- $A \rightarrow \epsilon$
- $A \rightarrow a$
- $B \rightarrow \epsilon$



CoScienze
Associazione

La tabella per FIRST e FOLLOW è:

	FIRST	FOLLOW
E	$a \epsilon$	\$
A	$a \epsilon$	\$
B	ϵ	\$

La tabella per verificare che la grammatica sia LL(1) è:

	a	\$
E	$E \rightarrow AB; E \rightarrow A$	$E \rightarrow AB; E \rightarrow A$
A	$A \rightarrow a$	$A \rightarrow \epsilon$
B		$B \rightarrow \epsilon$

Applicando le regole, vado a cercare i non terminali che hanno doppie produzioni, in questo caso $E \rightarrow AB$ e $E \rightarrow A$. In tal caso $\text{FIRST}(AB) \cap \text{FIRST}(A) = \{a, \epsilon\}$ diverso dal \emptyset . Ho trovato una coppia per cui l'intersezione non è vuota

7.24 Come realizzare un parser

Per fare un parser abbiamo bisogno di:

- **input**
- **stack**: rappresentato da una **stringa il cui top è a sinistra ed il bottom a destra**. Nel caso in cui si inverta top e bottom dello STACK, le produzioni dovranno essere scritte al contrario
- **tabella LL(1)**: essa **codifica la grammatica**
- **algoritmo di parsing**: esso **utilizza lo stack**.

Nella pratica, l'**algoritmo di parsing** esegue iterativamente una serie di operazioni. Nello stack, inizialmente su bottom ho solo non terminale iniziale "E" e \$. L'algoritmo ad ogni iterazione controlla il top dello stack. Se esso è un non terminale, con il look ahead (segnata in rosso nella colonna INPUT) vede quale produzione, partendo da quel non terminale mi permette di raggiungere quel simbolo, andandola a scegliere ed inserendola nella colonna AZIONE, che mi permette inoltre di **tenere traccia dell'albero**. Il look ahead si muove solo quando c'è match.

Nell'iterazione 3, in cui si ha un terminale e si ha un match con l'input, **sull'input** sposto il look ahead al prossimo simbolo e **rimuovo** inoltre il **terminale riconosciuto dallo stack** tramite pop. Quando arriviamo a \$ sia nello STACK che nell'INPUT, allora la frase è accettata. Se non ci arrivo, la frase non è accettata.

ITERAZIONE	STACK	INPUT	AZIONE
0	E\$	(id+id)*id\$	$E \rightarrow TE'$
1	TE'\$	(id+id)*id\$	$T \rightarrow FT'$
2	FT'E'\$	(id+id)*id\$	$F \rightarrow (E)$
3	(E)T'E'\$	(id+id)*id\$	match("(")
3	E)T'E'\$	(id+id)*id\$	$E \rightarrow TE'$
4	TE')T'E'\$	(id+id)*id\$...

Nella **colonna delle azioni** io mantengo traccia delle produzioni che vengono applicate. Da esse posso ricostruire l'albero. Esso è un parser in quanto **prende in input una stringa e restituisce l'albero di parsing o di derivazione**. Una **sentenza fa parte della grammatica solo se quella grammatica è capace di generare un albero di derivazione per quella sentenza**.

7.25 Riassunto sui parser

Un parser è un **programma che prende in input una sequenza di token o terminali e restituisce una risposta “si” o “no”**. Restituisce “si” e l’albero di derivazione se la sequenza è riconosciuta dalla grammatica. Se il parser corrispondente riesce a costruire un albero di derivazione per quella sentenza. Dalla **grammatica costruisco il parser e riesco a capire se quella frase è generabile dalla grammatica se e solo se il parser corrispondente riesce a costruire un albero di derivazione per la frase**. Il parser può dipendere fortemente dalla grammatica. In generale, esistono due tipi di parser:

- **top down**: l’albero di derivazione **viene generato dalla radice alle foglie**. A sua volta esso si divide in:
 - **parser a discesa ricorsiva**: esso può essere **non predittivo, predittivo, con backtrack o senza backtrack**. Essere con o senza backtrack è indipendente dall’essere predittivo o meno, in quanto avere backtrack o meno **dipende dalla grammatica**
 - **parser predittivi LL(1)**: esso è sempre **predittivo e sempre senza backtrack**, altrimenti è non LL(1) per definizione
- **bottom up**: l’albero di derivazione **viene generato dalle foglie alla radice**. Anche essi saranno predittivi senza backtrack. Tipi di parser bottom up sono:
 - LR(1)
 - SLR(1)
 - LALR(1)

La differenza tra i due è **il modo in cui viene generato l’albero di derivazione**. Sia LL(1) ed LR(1) sono basati su tabella. Prendo la grammatica e la traduco in una tabella che sarà alla base del parser.

A seconda della grammatica **posso decidere se applicare il backtrack o meno**. Nel dubbio si gestisce sempre il backtrack, è ciò viene fatto conservando il puntatore prima di andare avanti nell’input. Ogni volta che sono in un terminale, per essere sicuro che mi vada tutto a buon fine mi salvo il puntatore allo stato corrente e poi effettuo le mie operazioni. **Se qualcosa è andato male, faccio backtrack. Devo quindi conservare lo stato** in quanto, se qualcosa va storto, posso ripristinare lo stato. Lo stato in questo caso è un puntatore.

7.26 Schema di un parser

Se la discesa ricorsiva posso programmarla a mano, si avranno dei generatori automatici per la tabella. Gli diamo la grammatica e tramite algoritmi costruisce la tabella. Lo schema di un parser generico è il seguente:

- **input**
- **driver:** algoritmo generico che funziona sempre. esso legge la tabella, legge l'input, legge il top dello stack, va a vedere l'azione da fare sulla tabella e produce l'albero di parsing, sparando le produzioni che applica
- **tabella di parsing:** esse possono essere LL(1) o LR(1)...
- **stack**

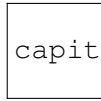
capitoli/images_parser/parser.jpg

Figura 7.7: Schema di un parser

Per questi algoritmi, lo schema generale è quello presentato in foto. Si prende la grammatica e la si trasforma in una tabella di parsing. Il driver che rimane lo stesso dato l'algoritmo LL(1) o LR(1), prende la tabella di parsing (unica cosa che deve cambiare) e viene agganciata al driver stesso. Il driver saprà come usare la tabella, leggere l'input ed aggiornare lo stack.

7.27 L'insieme delle grammatiche

L'algoritmo per LL(1) ed LR(1) è differente però il driver è capace di simulare entrambi. Se una grammatica è fattorizzabile, sicuramente non è LL(1), lo stesso vale per la ricorsione sinistra. Se vado a ritagliare tutte le possibile grammatiche, avrò un insieme LL(1) per cui è possibile avere una tabella senza conflitti. Si avrà poi l'insieme delle grammatiche LR(1) per cui è possibile fare una tabella di parsing LR(1). Se una grammatica è LR(1) allora essa sarà sicuramente SLR(1). Quando si parla di potenza dell'algoritmo si parla di quante grammatiche riesce a gestire senza doversi trasformare. Quello più potente è LR(1) in quanto riesce a gestire più grammatiche di tutti gli altri.



capitoli/images_parser/insiemi.jpg

Figura 7.8: L'insieme delle grammatiche

7.28 Esercizio 16: Definire se una grammatica è LL(1)

Data la grammatica:

- 0) $S \rightarrow SB$
- 1) $S \rightarrow y$
- 2) $B \rightarrow Bx$
- 3) $B \rightarrow Ax$
- 4) $A \rightarrow z$
- 5) $A \rightarrow zSy$

Quando si dice “questa grammatica” si intende grammatica non linguaggio. **Questa grammatica andrebbe trasformata in quanto si ha ricorsione sinistra, fattorizzazione sinistra...** ma applicando tali trasformazioni non si sta studiando più questa grammatica ma un'altra

8.1 Introduzione al parsing bottom up

Mentre con il top down **partivo dalla radice** ed andavo a cercare nella grammatica una produzione che nella parte sinistra avesse la radice, sostituendola poi con la parte destra, **nel bottom up facciamo il contrario. Dobbiamo partire dall'input e dobbiamo costruire sull'input, arrivando fino alla radice.**

8.2 Concetto di riduzione

La produzione andrà vista al contrario: **invece dal non terminale alla sentenza, ora andremo a vedere dalla sentenza al non terminale.** Il passaggio di prima si chiamava derivazione in quanto **vado a derivare cosa mi produceva il non terminale, mentre ora si chiama riduzione in quanto vedo la parte destra della produzione e la riduco alla parte sinistra.** Importante è che **non c'è bisogno né di fattorizzare né di rimuovere la ricorsione sinistra.** Parlando di parser bottom up stiamo parlando sempre di parser predittivi e tabellari. In questo caso, **non c'è l'analogo della discesa ricorsiva.** Anche qui avremo generatori automatici, **questo perchè è complesso fare un parser LR a mano.** Essa comunque non deve essere ambigua.

Nel parsing bottom up **non parliamo più di derivazioni, in quanto è un concetto top down, ma parleremo di riduzione.** Non derivo una produzione ma **riduco una produzione.**

Mentre prima dalla E dovevo derivare la frase, ora io dall'input devo ridurre e raggiungere il non terminale iniziale. Nel caso della riduzione, mentre prima la frontiera erano le foglie in basso, ora la frontiera sono i nodi senza un padre

8.2.1 Esempio: Grammatiche LR

La grammatica può essere ricorsiva sinistra in quanto non si presenta un problema:

- 1) $E \rightarrow E + T$
- 2) $E \rightarrow T$
- 3) $T \rightarrow T * F$
- 4) $T \rightarrow F$
- 5) $F \rightarrow (E)$
- 6) $F \rightarrow id$

Dato l'input $id * id$ devo costruire l'albero scorrendo l'input da sinistra verso destra ed applicando le regole al contrario..

Se vedo id devo ridurre in F , salendo. La frontiera al momento è $F * id$. Riduco poi F in T . Adesso, diventa complesso capire quando applicare la reduce e quando lo shift. Potrei ridurre T in E però in tal caso non riuscirei più a raggiungere il non terminale. La mia frontiera diventerebbe infatti $E * id$ che non può essere ridotto nel non terminale iniziale. L'algoritmo non farà backtrack pertanto in un caso reale lui saprà che non deve ridurre T in E . Non posso ridurre T in E pertanto tramite l'operazione di shift leggo $*$. Ora posso fare shift o reduce. Non avendo una produzione $T * id$ rifaccio shift e arrivo in id . Riduco poi id in F e faccio la riduzione che va da $T * F$ in T , che è anche la cosa più conveniente. Riduco poi T in E . Siccome ho raggiunto il non terminale iniziale posso lanciare l'accept

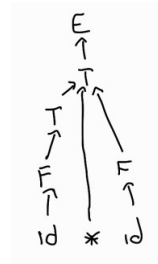


Figura 8.1: Esempio: Grammatiche LR

8.3 Operazioni del parsing shift reduce

Le operazioni che possono effettuare sono:

- **reduce:** mi permette di **sostituire la parte destra di una produzione con la sua parte sinistra**
- **shift:** mi permette di **andare nel prossimo simbolo dell'input. Quando faccio uno shift per leggere l'input non posso tornare indietro**
- **accept:** possibile lanciarla **quando arrivo al non terminale iniziale**
- **error:** operazione che **effettuo quando non posso fare le altre**

è complicato capire quando fare la shift e quando la reduce. **Ci vorrebbe una tabella che ci dica ad ogni passo se fare shift o reduce ed in caso quale produzione usare per la riduzione.** Quando facciamo la reduce, **abbiamo scoperto che alcune frontiere si possono ridurre ed altre no. Quelle che si possono ridurre ci danno garanzia di arrivare alla radice, le altre no.**

8.4 Parsing bottom up e derivazione rightmost

Andando a scrivere l'albero bottom up dell'esercizio di prima come una derivazione **rightmost** (in quanto per bottom up usiamo rightmost e top down leftmost), ottengo: $E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$. **Rifacendo l'albero con derivazione right most, è possibile vedere come con i passi fatti prima ho generato una derivazione rightmost al contrario.** Nell'albero, infatti:

- **Passo 1:** La forma sentenziale $id * id$ diventa $F * id$.
- **Passo 2:** La forma sentenziale $F * id$ diventa $T * id$
- **Passo 3:** La forma sentenziale $T * id$ diventa $T * F$
- **Passo 4:** La forma sentenziale $T * F$ diventa T
- **Passo 5:** La forma sentenziale T diventa E

Dalla costruzione bottom up partendo da sinistra dell'input a destra, non faccio altro che costruire una derivazione rightmost al contrario. Posso quindi verificare, tramite bottom up, se quella che ho costruito sia realmente una rightmost

Ad un certo punto, io avrei voluto sviluppare $T * id$ in $E * id$, riducendo quindi T in E . Come possiamo osservare, tale azione non si trova nella derivazione rightmost e infatti farlo avrebbe portato ad un errore.

8.5 Handle

Se qualcuno ci dicesse gli handle volta per volta, saprei perfettamente cosa prendere. Guardo la forma sentenziale e dico che gli handle sono dati da:

- **parte della forma sentenziale:** esso indica la **parte sentenziale a cui siamo interessati** (potrebbe essere anche l'intera sentenza) nell'esempio è id
- **posizione:** esso indica la **posizione della parte sentenziale a cui siamo interessati** nell'esempio è 0 in quanto 0 è la posizione della parte a cui siamo interessati nella stringa
- **produzione:** esso indica la **produzione in cui quella parte sentenziale compare come parte destra e che quindi ci permette di applicare la riduzione** nell'esempio è 6 in quanto è dato dalla produzione $F \rightarrow id$

L'handle quindi ($id\ 0\ 6$) mi dice che non appena vedo " id " in posizione 0 , posso utilizzare la **produzione 6** e fare la riduzione.

Siccome è un handle è la cosa giusta e sicura da fare. Mi troverò in una forma sentenziale che è quella precedente nella derivazione rightmost. Nel concetto di handle c'è già la correttezza. Gli handle mi garantiscono sempre di arrivare in una forma sentenziale raggiungibile dalla radice. La scansione da sinistra a destra dell'input avviene una sola volta, non posso spostarmi a piacimento. Un qualcosa è un handle, quindi, se la sua derivazione mi garantisce di arrivare al successo. Potrei poi sbagliare dopo ma sicuramente la forma sentenziale che raggiungerò con handle mi permette di essere raggiunta dalla radice. Come si arriva a calcolare l'handle non è dato saperlo. Ci possono essere più handle nella stessa forma sentenziale se la grammatica è ambigua. Entrambi, in questo caso, ci garantiranno di arrivare al non terminale iniziale in modo diverso. Seguendoli entrambi avrò due alberi diversi. Un handle mi garantisce, almeno localmente e per quello che sto facendo la frase sia giusta. Se successivamente la frase è sbagliata non è dato dall'handle saperlo.

Da ricordare che l'handle è afferente alla singola forma sentenziale, ovvero ogni forma sentenziale ha un singolo handle. Data una derivazione e calcolando l'handle per ogni derivazione ho una sequenza di handle

Se invece ad esempio l'handle sarebbe stato (id 2 6), ovvero sto puntando all'ultimo id della sentenza "id * id", ciò non sarebbe stato un handle in quanto non mi avrebbe portato alla forma sentenziale raggiungibile dalla radice.

8.5.1 Esempio: handle

Data la derivazione $E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$, gli handle per ogni forma sentenziale sono:

- Nell'ultima derivazione(id * id): l'handle è (id, 0, 6)
- Nella penultima derivazione(F * id): l'handle è (F, 0, 4)
- Nella terzultima derivazione(T * id): l'handle (id, 2, 6)
- Nella quartultima derivazione(T * F). l'handle è (T*F, 0, 3)
- Nella quintultima derivazione (T): l'handle è (T, 0, 2)
- Arrivato ad E, c'è l'accept

Tutto si risolve nel riconoscimento di E. Dobbiamo fare un algoritmo che sia in grado di riconoscere E. Dobbiamo realizzare un automa push down a partire dalla grammatica, dovendo simulare la derivazione della frase. Gli automi push down sono degli automi con degli stack

8.6 Automi push down

Tutto si risolve nel riconoscimento del non terminale iniziale, dovendo fare un algoritmo che sia in grado di riconoscerlo. Dobbiamo realizzare un automa push down a partire dalla grammatica, deve simulare la derivazione della frase. Gli automi push down sono degli automi con degli stack. L'automa LL(1) è un automa push down. I linguaggi context free sono riconoscibili da automi push down. Ogni volta che dobbiamo riconoscere una frase scritta in grammatica context free, andremo ad usare un automa push down. Gli automi regolari (diagrammi di transizione) sono automi senza stack che riconoscono la grammatica regolare. Con le grammatiche context free si può contare fino a 2 cose, mentre con gli automi regolari non si può ricordare nulla.

8.7 Esempio: Parser bottom up e grammatica ambigua

Data la grammatica ambigua:

- 1) $E \rightarrow E + E$
- 2) $E \rightarrow E * E$
- 3) $E \rightarrow id$

Data la sentenza $id * id + id$, i due alberi sono:

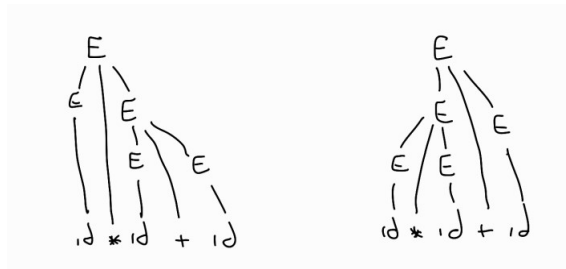


Figura 8.2: Esempio: Parser bottom up e grammatica ambigua

Le derivazioni rightmost sono:

- $E \Rightarrow E + E \Rightarrow E + id \Rightarrow E * E + id \Rightarrow E * id + id \Rightarrow id * id + id$ (albero di destra)
- $E \Rightarrow E * E \Rightarrow E * E + E \Rightarrow E * E + id \Rightarrow E * id + id \Rightarrow id * id + id$ (albero di sinistra)

In rosso è segnato il primo punto di in cui si differenziano nella derivazione leggendo da destra a sinistra. A livello di bottom up, invece, voglio confrontare le due derivazioni rightmost e vedere come si distinguono gli handle:

- **Step 1:** In questo caso, **seguire destra e sinistra è lo stesso**. Ho id che deve essere ridotto in E , pertanto il mio handle è $(id, 0, 3)$.
- **Step 2:** Anche in questo caso **seguire destra e sinistra è lo stesso**. Devo fare uno shift sul $*$ ed arrivare al prossimo id . L'operazione di shift non si vede nella riduzione.
- **Step 3:** vedendo l'albero di destra una volta che leggo il secondo id e faccio una riduzione in E . L'handle sarà $(id, 2, 3)$.
- **Step 4 (destra):** vedendo l'albero di destra ora ho la frontiera che è " $E * E + id$ ". **Sempre il destro mi dice che devo ridurre $E * E$ in E** , pertanto l'handle è $(E * E, 0, 2)$.

- **Step 4 (sinistra):** L'albero di sinistra, invece, qui **non vuole ridurre $E * E$, ma volendo ridurre prima la somma dovrebbe shiftare fino all'ultimo id e ridurre id in E**. L'handle sarà (id, 4, 3)
- **Step 5(destra):** ora avendo " **$E + id$** " faccio lo shift a id, e riduco in E. L'handle sarà (id, 2, 3)
- **Step 5(sinistra):** rispetto allo step destro, ho la forma " **$E * E + E$** " e riducendo diventa " **$E * E$** ". L'handle sarà (E+E, 2, 1)
- **Step 6(destra):** ora avendo " **$E + E$** " riduco in E, facendo la mia ultima riduzione. L'handle sarà (E+E, 0, 1)
- **Step 6(sinistra):** ora avendo " **$E * E$** " riduco in E, arrivando al non terminale iniziale. (E * E, 0, 2)

Avendo gli handle per i vari passaggi, possiamo ora controllare se le derivazioni siano **entrambe rightmost**. Per come sono scritte le derivazioni, possiamo vedere come entrambe lo siano. **Diventa derivazione rightmost in quanto si legge da sinistra a destra l'input.**

8.8 Conflitti della tabella LR

La grammatica risultata da questo esempio non sarà LR in quanto **quando andrò a realizzare la tabella che mi dirà cosa fare se shift o reduce in una cella ci saranno due azioni**: in un certo punto **avremo un conflitto shift reduce o reduce reduce**. Avrò un **conflitto shift reduce** se nella tabella, invece di trovare un'azione, ne trovo due: shift e reduce. Si parla di **conflitti reduce reduce** quando ci sono due produzioni differenti che posso usare per ridurre. L'handle mi dice sempre anche la produzione ma posso avere più produzioni che mi portano allo stesso punto. Se l'handle indica la produzione che bisogna usare per fare la riduzione, l'handle sarà collegata solo ad operazioni di reduce. Una volta che ho l'handle, derivo. Gli shift vengono fatti per raggiungere l'handle. Mi sposto fin quando non vedo l'handle. Gli shift sono fatti solo per la navigazione che può avvenire solo da sinistra a destra. Una volta che ho raggiunto la fine dell'input, non posso più fare shifts.

8.9 Items

Dobbiamo estrarre quante più informazioni possibili dalla grammatica, in quanto è da quest'ultima che dovrò ottenere informazioni sugli handle. Per analizzare una grammatica

nei minimi termini, dobbiamo definire il concetto di item: **Data una produzione che va in $A \rightarrow xyz$, se noi aggiungiamo \cdot , avendo $A \rightarrow \cdot xyz$, esso ci indicherà quanto è stato visto di quella produzione e quanto si deve ancora vedere. Una produzione simile si chiama item. Con \cdot all'inizio stiamo indicando che di quella produzione non ho visto nulla. Quando \cdot è alla fine della produzione, esso prende il nome di item completo. Questo è il momento in cui posso ridurre. Quando arrivo in uno stato in cui l'item è completo, quell'item lì è quello che mi fa fare la riduzione.**

Data una produzione, posso avere diversi tipi di item, ovvero posso avere \cdot in più punti. Il \cdot rappresenterà un'informazione di quanto è stato visto da quella produzione e quanto ancora bisogna vedere, pertanto in quale punto della produzione mi trovo. Durante l'algoritmo, potrei avere più produzioni attive e di ognuna avrò visto una parte.

8.9.1 Esempio: Items

Ad esempio:

- Con $A \rightarrow \cdot abc$ indico che non ho visto nulla
- Faccio lo shift di a e passo in uno stato in cui a è stato visto e mi aspetto di vedere bc
 $A \rightarrow a \cdot bc$
- Se vedo b vado in uno stato in cui b è stato visto e c mi resta vedere $A \rightarrow ab \cdot c$
- Se vedo anche c , allora andrò a finire nello stato $A \rightarrow abc \cdot$.

Se avessi invece ho $S \rightarrow a \cdot b$ e $S \rightarrow a \cdot c$ e ho letto a , vado in uno stato potrei o vedere b o vedere c . Sono entrambe in gioco. Potrei andare in uno stato in cui potrei vedere o una b o una c

8.10 Automa per gli items

Graficamente, mi porto indietro tutte le produzioni. Non appena una di esse diventa completa, ovvero un item completo, essa diventa la mia riduzione. Devo tenere traccia, in questo automa, di tutte le possibili produzioni ammissibili in quel punto. Un item è la produzione che ha un \cdot nella parte destra. Esso mi dice quanto ho visto della produzione e quanto ancora devo vedere. Gli item verranno usati per costruire gli stati.

Prima cosa da fare per costruire un automa è aumentare la grammatica con una produzione " $S' \rightarrow S''$ ". Essa mi garantisce che il non terminale iniziale non viene usato nel resto

della grammatica. Pertanto, quando riconosco S' non mi ritrovo in una produzione ma devo accettare. Nella realizzazione degli stati, quando ho una produzione ad esempio $A \rightarrow \alpha \cdot S$, pertanto il \cdot precede un non terminale, devo fare l'operazione di chiusura su quel non terminale, sviluppando per quel non terminale tutte le sue possibili produzioni. Vado avanti ad oltranza fin quando non ci sono più \cdot prima di un non terminale. Ogni stato sarà un insieme di items. In tale insieme ogni item deve ripetersi una sola volta. Se cerco di mettere un qualcosa uguale ad un'altra che c'è già, non la ripeterò.

Un'altra operazione è quella di goto. Essendo in uno stato e vedendo un simbolo, esso mi dice in quale altro stato devo spostarmi. I goto sui terminali generano gli shift. Con gli item completi è un'operazione di reduce. Ogni stato è un insieme di items, ed ogni automa è un insieme di insiemi di items, pertanto neanche gli stati possono ripetersi se uguali. Gli insiemi di items corrispondono agli stati dell'automa che riconosce una grammatica. L'automa automa avrà memoria e la memoria sarà data dallo stack.

8.11 Tabelle LR(0) ed impatto dei reduce

Se nella tabella associata all'algoritmo quando andiamo a fare la riduzione non andiamo a vedere il prossimo simbolo, la tabella sarà LR(0). Questo vale anche se con lo shift noi andiamo a sbirciare il prossimo simbolo. In questi casi, infatti, la differenza la fa la reduce. Se faccio la reduce senza preoccuparmi di andare a vedere il simbolo successivo, allora essa sarà LR(0). Quando anche le reduce guarderanno l'input, allora la tabella sarà LR(1). Allo stesso modo, gli items associati a questa tabella saranno LR(0). La tabella LR(1) metterà un look ahead sulla reduce e faremo una riduzione solo se il prossimo simbolo è quello che ci aspettiamo. Nel caso di LR(0), se il simbolo successivo alla forma sentenziale da ridurre fosse stato errato, avremmo comunque fatto la reduce. Con LR(1) ciò si può evitare

8.11.1 Esempio: automa per gli items

Data la grammatica ampliata:

- 0) $S' \rightarrow S$
- 1) $S \rightarrow 0 S 1$
- 2) $S \rightarrow 0 1$

Voglio costruire su questa grammatica un automa che la riconosca. Inizio a costruire uno stato 0. In questo stato non è visto nulla. Tutte le frasi che posso riconoscere devono

avere come radice S' . Parto da S' ma mi aspetto di vedere qualcosa che deriva da S . Applico la chiusura per la produzione $S' \rightarrow \cdot S$

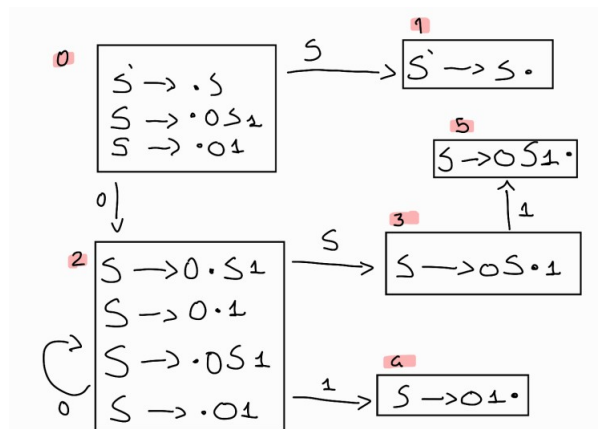


Figura 8.3: Esempio: automa per gli items

Partendo dallo stato 0, ed aggiungendo lo stato 1, in esso vado a portarmi tutte le produzioni che mi aspettavano S , in questo caso solo 1. Essendo che \cdot è alla fine $S' \rightarrow S \cdot$ è un item completo, si traduce con l'azione di accept

Un altro simbolo che stiamo aspettando nello stato 0 è proprio 0. Ho due produzioni che mi aspettavano 0 quindi sposto entrambe le produzioni, andando nello stato 2. Con " $S \rightarrow 0 \cdot S 1$ ", sto dicendo che di quella produzione ho visto lo 0 e voglio vedere una S , ma cosa si intende? Devo quindi fare una chiusura su S .

Se vedo una " S " stando nello stato 2, faccio l'operazione di goto e vado nello stato 3 con item 0. Avrò quindi l'item $S \rightarrow 0S \cdot 1$. Al contempo stando nello stato 2 e vedendo " 1 " faccio una goto allo stato 4. In tale stato, avrò $S \rightarrow 01 \cdot$, il che vuol dire che ho visto tutto. Ciò genera un reduce. Se arriviamo allo stato 4, sul top dello stack sarà garantito di vedere 01.

Nello stato 2, stiamo aspettando lo 0. Se io trovo uno stato che è uguale all'altro, non devo rimetterlo. Per cui, quello che faccio a questo punto, dallo stato 2 se rivedo " 0 " torno allo stato 2.

Lo stato 3 aspetta un 1, pertanto creo lo stato 5 con produzione $S \rightarrow 0S1 \cdot$. Io sono nello stato 3 che si aspetta un 1. Se nello stato 3 gli arriva " 1 ", allora andiamo nello stato 5, altrimenti se diamo ad esempio 0, andremo in errore

8.11.2 Esempio: Algoritmo ed uso dello stack

Data la grammatica ampliata di prima ho:

- 0) $S' \rightarrow S$
- 1) $S \rightarrow 0 S 1$
- 2) $S \rightarrow 0 1$

ITERAZIONE	STACK	INPUT	AZIONE
0	$\hat{0}\hat{0}\hat{2}$	0 011\$	SHIFT 2
1	$\hat{0}\hat{0}\hat{2}\hat{0}\hat{2}$	0 11\$	SHIFT 2
2	$\hat{0}\hat{0}\hat{2}\hat{0}\hat{2}\hat{1}\hat{4}$	1 1\$	SHIFT 4
3	$\hat{0}\hat{0}\hat{2}\hat{S}\hat{3}$	1 \$	REDUCE 2
4	$\hat{0}\hat{0}\hat{2}\hat{S}\hat{3}\hat{1}\hat{5}$	\$	SHIFT 5
5	$\hat{0}\hat{S}\hat{1}$	\$	REDUCE 1
6			ACCEPT

Il bottom dello stack sarà a sinistra, in quanto intuitivamente nel parser top down era al contrario. Rispetto all'esempio, data la ridondanza di simboli (0 può indicare sia uno stato che un terminale), $\hat{0}$ indicherà lo stato 0, mentre 0 indicherà il lessema 0. Ciò verrà fatto per tutti gli stati.

Dire $\hat{0}\hat{0}\hat{2}$ vuol dire che mi sposto dallo stato $\hat{0}$ allo stato $\hat{2}$ in quanto ho letto il simbolo 0 come prossimo elemento dell'input. Dire SHIFT 2 vuol dire che io faccio lo shift verso lo stato 2 mentre dire ad esempio REDUCE 2 vuol dire che sto usando la seconda produzione ovvero (2) $S \rightarrow 0 1$. Da notare che il reduce posso farlo solo quando l'handle è sullo stack. Avendo ad esempio 0 sullo stack e 1 nell'input non posso fare nulla. Dovrò infatti avere 01 nello stack per poter fare il reduce (cosa che avviene nell'interazione 3). Effettuare la reduce vuol dire fare il cammino a ritroso nell'automa. Tornando indietro dallo stato 4 allo stato 2, mi trovo in uno stato inconsistente in quanto avrò sul top del mio stack S, quando il top dello stack dovrebbe essere rappresentato da uno stato. Pertanto mi sposto nello stato $\hat{3}$. Nell'INPUT, in rosso sono indicati i simboli che di volta in volta sto leggendo e su cui sto shiftando, mentre in rosso sullo STACK sono indicate le riduzioni. Finita di calcolare la tabella, avrò quindi i due handle che mi portano all'accettazione della frase

8.12 Operazioni di reduce e shift nell'esecuzione dell'automa

Rispetto alle operazioni normali, reduce si divide in due passi:

- prima **sostituisce (riduce)** togliendo la forma sentenziale dallo stack e mettendo il **non terminale** al loro posto
- poi **completa poi il tutto aggiornando lo stato**. Lo stato in cui la forma sentenziale **finirà dopo la reduce** è indicato dallo stato precedente alla forma sentenziale.

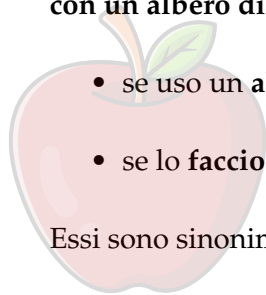
Se prima della reduce ho sullo stack $\overset{\wedge}{0}\overset{\wedge}{0}\overset{\wedge}{2}\hat{S}\overset{\wedge}{3}\overset{\wedge}{1}\overset{\wedge}{5}$, facendo una riduzione di $S \rightarrow 0S1$ avrò come risultato $\overset{\wedge}{0}\hat{S}$. Pertanto, dopo la riduzione lo stato in cui mi trovo sarà 0 dovendomi poi spostare in $\overset{\wedge}{1}$, in quanto se sto nello stato 0 e leggo S devo spostarmi nello stato 1

8.13 Differenza tra albero di parsing e albero di derivazione

In generale, l'insieme delle grammatiche LL(1) è contenuto nell'insieme delle grammatiche LL(2). Avendo grammatiche LL(2) o LL(3) avremo FIRST e FOLLOW modificati. Quando vado a fare una derivazione, essa posso farla o con una derivazione tradizionale o con un albero di derivazione. Nello specifico:

- se uso un albero per riconoscere una frase, esso prende il nome di albero di parsing
- se lo faccio per generare una frase allora si parla di albero di derivazione

Essi sono sinonimi



Coscienze
Associazione

8.14 Riassunto: parsing bottom up

Nel parsing bottom up l'albero lo costruiamo dalle foglie per arrivare alla radice. Andando a vedere la generazione partendo da S come albero e non come derivazione, vedrei una derivazione leftmost nel caso del parsing top down.

L'input per il parser bottom up è sempre una frase ed una grammatica. La frase viene riconosciuta dalla grammatica se io posso costruire un albero di parsing su quella frase. L'albero però devo costruirlo però partendo dalla sentenza e salendo fino ad arrivare alla radice. Per il parsing bottom up non ci interessa togliere la ricorsione sinistra o fattorizzare. Ciò che abbiamo con le riduzioni, è che la frontiera è sempre una forma sentenziale da cui derivo un'altra forma sentenziale. Nel caso del top down derivo una forma sentenziale mentre nel caso del bottom up, dalla forma sentenziale riduco un'altra forma sentenziale.

8.14.1 Esempio: parser bottom up

Data la grammatica:

- $S \rightarrow a B c$
- $B \rightarrow b$
- $B \rightarrow b c$

Abbiamo detto che **questa grammatica non è LL(1)**. Data una frase $abc\$$ quello che ci interessa è costruire una derivazione **rightmost inversa**, ovvero costruire l'albero dall'input alla radice. Nello specifico, se prima utilizzavo le produzioni espandendo la parte sinistra della produzione nella parte destra, io ora ho la parte destra della produzione e la devo ridurre nella parte sinistra. L'idea del parsing bottom up è che **devo ridurre l'input fino al non terminale iniziale**. La difficoltà è trovare cosa devo ridurre. Nell'esempio, io ho due produzioni che mi danno la stessa cosa: solo una sarà quella di successo

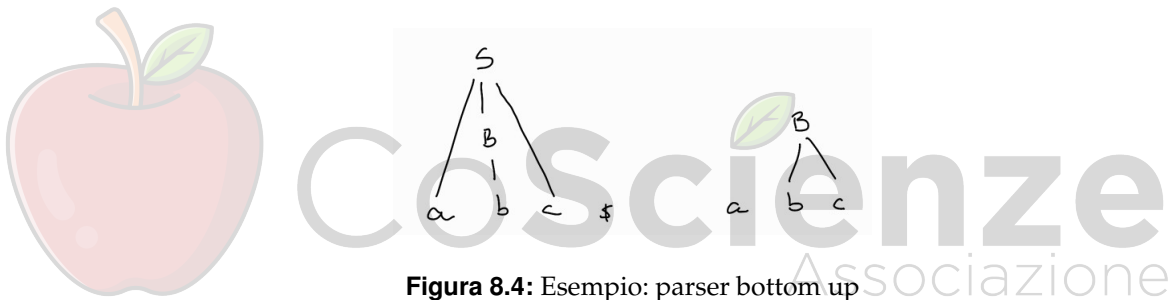


Figura 8.4: Esempio: parser bottom up

La frontiera adesso sarà composta dai nodi senza un padre. **Con la scelta della riduzione effettuata nell'albero di sinistra ho raggiunto il non terminale iniziale in due produzioni.** Facendo però la scelta di destra, ovvero riducendo bc in B otterrei una frontiera aB , da cui però non possiamo raggiungere il non terminale iniziale. Tramite l'albero di destra, sembrerebbe impossibile raggiungere il non terminale iniziale, ma ciò è dato solo dal fatto che abbiamo scelto una riduzione sbagliata. Facendo la derivazione rightmost dell'albero di sinistra ho: $abc \Rightarrow aBc \Rightarrow S$. Ciò che abbiamo con questa riduzione, è che la frontiera è sempre una forma sentenziale da cui derivo un'altra forma sentenziale. Nel caso del top down derivo una forma sentenziale mentre nel caso del bottom up, dalla forma sentenziale riduco un'altra forma sentenziale.

Essa viene ridotta come una **rightmost derivation**. Una **rightmost derivation** è: quando in una forma sentenziale ho due non terminali, prendo quella più a destra. Se parlo di **rightmost** in ambito bottom up la costruisco al contrario ma la vado a vedere costruita

come una **rightmost derivation**. L'handle in questo caso sono (b 1 2) e (aBc 0 0). Ogni volta che abbiamo una forma sentenziale dobbiamo capire quale sia l'handle.

8.15 Riassunto: Gli handle

Nella parola handle c'è già il successo. Se uso un handle ho fatto sicuramente la scelta giusta. Ogni volta che ho una forma sentenziale, devo capire quale sia l'handle. Un handle è una tripla formata da:

- parte della forma sentenziale a cui siamo interessati
- posizione della sottostringa
- produzione che vogliamo applicare

L'handle ha una proprietà, la riduzione indicata dall'handle ci avvicina al non terminale iniziale in un percorso rightmost al contrario. L'handle indica la scelta giusta per costruire una rightmost al contrario arrivando alla radice se parlo di derivazione. Se però sto costruendo un albero, essa indica la scelta giusta per raggiungere, con quell'albero, il non terminale iniziale. Ci vorrebbe qualcuno che vede la forma sentenziale e dica gli handle. In generale, il bottom up parsing è fatto in due parti:

- un qualcuno che fornisce gli handle
- l'algoritmo in sé che usa la tabella di parsing (che in questo caso rappresenta chi indica quali siano gli handle)

8.16 LR(1) e funzionamento delle tabelle di parsing

Stiamo lavorando sulla classe di parser LR(1). La L in questo caso indica che lo scanning dell'input viene fatto da sinistra a destra mentre la R sta per la costruzione di una rightmost inversa. Nel caso LL(1) la prima L indicava la stessa cosa mentre la seconda L indicava che si stava generando una leftmost derivation. L'algoritmo di parsing sarà predittivo e farà uso di tabella. Comunicherà con la parsing table, lo stack, l'input e genererà l'albero di parsing.

All'interno della parsing table LR(1) c'è l'intelligenza per il riconoscimento degli handle. Essi vengono riconosciuti nella parsing table e il driver è il parser vero e proprio che applica la conoscenza riportata nella tabella. In essa ci sarà un riconoscitore di handle. Esso

prevede quale sia l'handle. Sapere in un certo momento quale sia l'handle non significa che sicuramente arriveremo al non terminale iniziale, ma se c'è un modo solo per arrivare esso sarà sicuramente la strada. Nel nostro caso, la parsing table sarà un automa

8.16.1 Esempio: esecuzione del driver

Data la grammatica:

- $S \rightarrow a B c$
- $B \rightarrow b$
- $B \rightarrow b c$

Con input $abc\$$, l'esecuzione del driver è:

ITER	STACK	INPUT	AZIONE
0	\$a	abc\$	SHIFT a
1	\$ab	abc\$	SHIFT b
2	\$aB	abc\$	REDUCE 3
3	\$aBc	abc\$	SHIFT c
4	\$S	abc\$	REDUCE 1
5	\$S	abc\$	ACCEPT

Una riduzione posso farla solo se la stringa l'ho già messa nello stack. La prima cosa che devo fare se devo operare sopra una stringa è metterla nello stack. Siccome la scansione è da sinistra a destra, nello stack bisogna partire dal primo simbolo più a sinistra. L'operazione di SHIFT mi permette di inserire il simbolo che sto leggendo nello stack. Vado poi a leggere il prossimo simbolo. Una volta che un simbolo è sullo stack, devo vedere se ho un handle per quel simbolo ed in caso applicarlo. Nell'input in rosso è segnato di volta in volta il simbolo letto.

Nell'iterazione 5, siccome sullo stack ho il non terminale iniziale e nell'input leggo \$, allora ho raggiunto la radice ed ho letto tutto l'input, posso accettare. Se la frase non apparterrà, ad un certo punto invece di avere accept avrò error.

8.17 Riassunto: costruzione dell'automata che riconosce gli handle

Data la grammatica, ad esempio:

- 0) $S' \rightarrow S$
- 1) $S \rightarrow a A$
- 2) $A \rightarrow B b$
- 3) $B \rightarrow c$

Se qualcuno non mi dice quando effettuare **azioni di shift o reduce** abbiamo un problema. L'automa per il riconoscimento degli handle è un automa che ci indicherà le operazioni da **effettuare**. Per costruire un automa useremo due operazioni:

- closure
- goto

Si parte aggiungendo un'operazione alla grammatica, $S' \rightarrow S$. Con questa produzione **cambio il non terminale iniziale. Voglio un non terminale che si trovi in nessuna parte destra di nessuna produzione**, pertanto quando trovo il non terminale iniziale so che sarà finita, non dovendo cercare in nessuna produzione. Essa è una **produzione aumentata**. L'algoritmo si basa sul concetto di item. **Closure e goto agiscono sugli item e permettono di costruirli.**

Per l'algoritmo si **parte dal non terminale iniziale, ovvero la nostra produzione aumentata**. Essa inizialmente sarà l'unica produzione aumentata. Essere nello stato $S' \rightarrow \cdot S$ vuol dire che **non ho ancora visto nulla e mi aspetto di ridurre tutta la frase ad S**. Essendo che il \cdot precede un non terminale **dovrò fare la chiusura su quel non terminale. Vado quindi a prendere tutte le produzioni che hanno come parte sinistra S**. Se \cdot si trova prima di un terminale non devo fare nulla.

Una volta che ho fatto le chiusure fino ai terminali, **chiudo lo stato in cui sto. Ogni stato è dato da un insieme di items, pertanto due items uguali non possono comparire**. Dato il fatto che la grammatica può avere ricorsione sinistra, **il fatto che ogni stato è un insieme di items pertanto non possono comparire items uguali, mi risolverà in automatico la ricorsione**.

Quando sono in uno stato e mi aspetto di vedere un simbolo, **quando vedo quel simbolo vado in un altro stato aggiornando il \cdot** . Ciò mi viene dato dall'operazione di goto. In generale non possono esserci archi con la stessa label partenti dallo stesso stato. **Tutti gli items che aspettano un simbolo dovranno essere messi nello stesso stato. Ogni stato potrà generare operazioni come accept, reduce, shift o error. Su item completi non bisogna fare la closure o la goto in quanto dopo il \cdot non ci sarà nulla**. Dopo un'operazione di goto provo

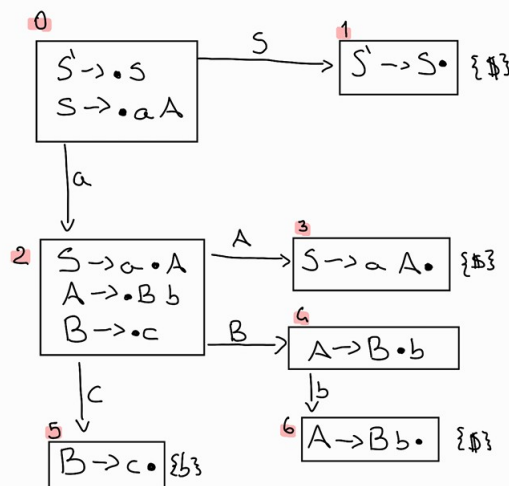


Figura 8.5: Riassunto: costruzione dell'automato che riconosce gli handle

a vedere se devo fare qualche chiusura nel nuovo stato. Per assicurarmi che analizzi tutto, prima di concentrarmi su di un altro stato conviene fare la goto e la closure di tutti gli items

8.18 Riassunto: concetto di item

Un item è una **produzione con un \cdot nella parte destra della produzione**. Se io prendo $A \rightarrow \cdot B b$ essa è una **produzione con un \cdot all'inizio e si chiama item**. In questo caso, il \cdot indica che di questa produzione non ho visto ancora nulla. **Tale produzione è però attiva e mi aspetto di vedere qualcosa che è derivabile da B**.

Tale produzione è attiva e mi aspetto di vedere nell'input qualcosa che è derivabile da B o riducibile a B. **Per ogni produzione posso avere il \cdot in qualsiasi parte della produzione**. Con $A \rightarrow B \cdot b$ indica che ho già visto B e mi aspetto di vedere b. Se riesco a vedere anche b, allora posso ridurre Bb in A. Sto quindi cercando di **costruire un handle su cui posso applicare poi quella produzione**. Quando per un qualsiasi motivo arrivo all'**item completo**, ovvero $A \rightarrow Bb \cdot$, allora sullo stack avrò Bb e potrò utilizzare la produzione $A \rightarrow Bb$ per la riduzione. In generale $A \rightarrow \epsilon$ è già un item completo, in quanto ϵ lo vedo sempre proprio perchè può essere qualsiasi cosa

8.19 Parsing table o tabella di codifica dell'automato

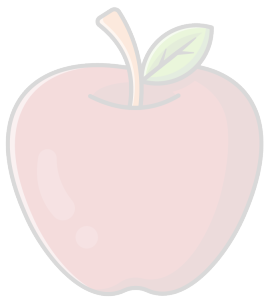
Posso **codificare l'automato push down per gli items in una tabella**: per ogni stato potrò dire se devo fare uno shift, un reduce o un goto su di un non terminale. Nello specifico ho

due sottotabelle: quella delle **action** e quella dei **goto**. Sulle **righe metto gli stati**, in questo caso da 0 a 6 mentre sulle colonne della tabella action metto i terminali e nella tabella goto metto i non terminali.

Tutte le informazioni necessarie all'automa le metto nella tabella. Guardando l'automa e vedendo il singolo stato ad esempio 0, vedendo "a" farò lo shift allo stato 2, pertanto nella cella [0, a] metterò S2. Per la tabella dei goto, sui non terminali non bisogna fare lo shift ma solo una transizione. Stando nello stato 0 e vedendo S mi sposto nello stato 1, pertanto in [0, S] metto 1, in quando è una transizione non uno shift.

Allo stesso modo, dato che $S' \rightarrow S \cdot$ sta nello stato 1, nella cella [1, \$] metterò accept. Ho messo accept quindi considerando il FOLLOW(S'). Il simbolo per decidere se fare o meno la reduce la vedo dopo il \cdot , ed in questo caso il simbolo dopo il \cdot non vedo nulla. Pertanto, devo simulare il simbolo successivo andando a fare il FOLLOW del non terminale. Banalmente, quello che seguirà quel non terminale seguirà anche \cdot .

Tabella action:



	a	b	c	\$
0	S2			
1				ACCEPT
2			S5	
3				R1
4	S6			
5				R3
6				R2

Tabella goto:

	S'	S	A
0		1	
1			
2			3
3			
4			
5			
6			

8.20 Parser SLR(1)

Gli shift sono sempre predittivi in quanto se non sono predittivi non guardando il terminale non riesco a decidere in quale stato andare a livello di automa. In caso di $S' \rightarrow S \cdot$ posso lanciare la reduce solo se l'item è completo. Per evitare potenziali errori per la reduce vado a controllare il prossimo simbolo. Nel caso di $S' \rightarrow S \cdot$ vuol dire che io sono a fine stringa pertanto il simbolo successivo deve essere necessariamente $\$$. Il simbolo per decidere se fare o meno la reduce la vedo dopo il \cdot , ed in questo caso il simbolo dopo il \cdot non vedo nulla. Pertanto, devo simulare il simbolo successivo andando a fare il FOLLOW del non terminale. Banalmente, quello che seguirà quel non terminale seguirà anche \cdot . Il FOLLOW verrà calcolato solo per gli item completi. Per ogni stato in cui ho un item completo, andrò a calcolare il FOLLOW del non terminale che sto considerando.

In questo caso avrò il look ahead anche sui reduce. Il parser SLR(1) è simile ad LR(0) con una differenza: esso va a controllare tramite look ahead gli item completi, andando a calcolare per il non terminale della parte sinistra dell'item il rispettivo FOLLOW. A livello grafico avrò:

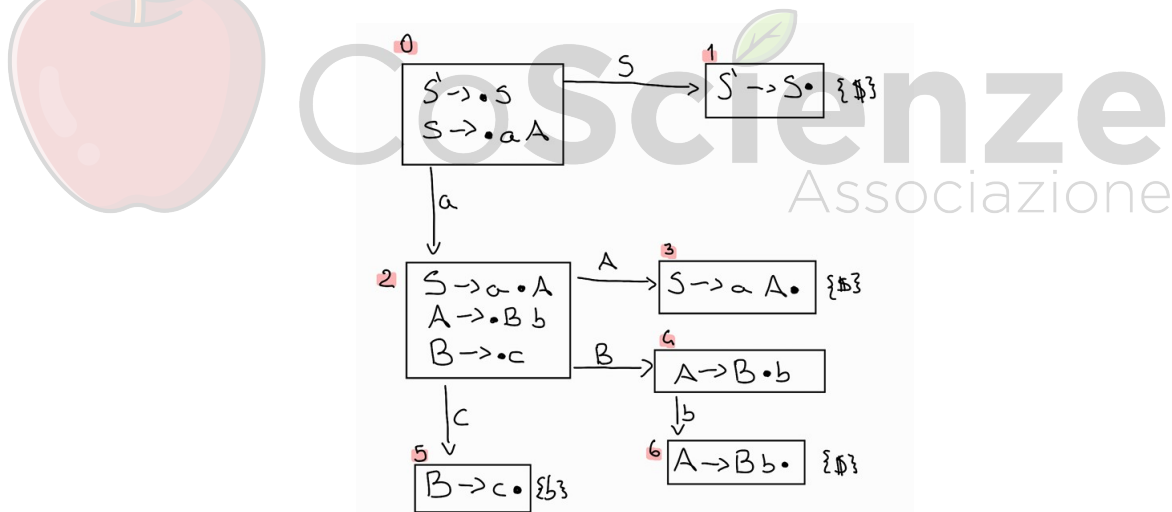


Figura 8.6: Parser SLR(1)

Il parser in questo caso si chiamerà Simple LR. La difficoltà sta nel come si calcola il look ahead delle reduce. Qui per farlo abbiamo messo il FOLLOW ma l'LR calcolerà il look ahead in modo più preciso, lasciando il resto invariato. In generale, sia LR(1) che SLR(1) lavorano costruendo l'automa LR(0). In alcuni casi, il FOLLOW potrebbe fallire, pertanto è necessario un qualcosa di meglio per calcolare gli LR(1)

8.21 Algoritmo SLR(1) e riconoscimento di una stringa

Data la grammatica:

- 0) $S' \rightarrow S$
- 1) $S \rightarrow a A$
- 2) $A \rightarrow B b$
- 3) $B \rightarrow c$

E dato l'input $acb\$,$ il funzionamento dell'algoritmo è:

ITER	STACK	INPUT	AZIONE
0	0	a cb\$	SHIFT 2
1	0a2	c b\$	SHIFT 5
2	0a2c5	b \$	REDUCE 3
3	0a2B	b \$	GOTO 4
3	0a2B4	b \$	SHIFT 6
4	0a2b4b6	\$	REDUCE 2
5	0a2A	\$	GOTO 3
5	0a2A3	\$	REDUCE 1
6	0S	\$	GOTO 1
6	0S1	\$	ACCEPT

Da considerare che **sul top dello stack deve esserci sempre uno stato. Per ogni iterazione, leggendo lo stack sul top dello stack (rappresentato dall'elemento più a destra) e il simbolo sull'input, vado a vedere nella parsing table quale operazione effettuare.** Con l'operazione di shift, metto il simbolo sullo stack e faccio la transizione al prossimo stato, consumando il simbolo sull'input e passando al successivo. L'operazione di reduce, invece, **leva dallo stack la parte destra della produzione e la sostituisce con la parte sinistra.** Sull'input sta indicato in rosso il simbolo che si sta leggendo. **Quando si fa un'operazione di reduce l'input si legge solo ma non si fa nessuna operazione su di esso**

L'operazione di goto che viene segnato nell'azione, è in realtà integrato nella reduce, perciò solitamente quando si fa la reduce si fa anche la goto. L'operazione di goto è quindi parte dell'operazione di reduce. Partendo dall'input ed applicando le azioni, potremo alla fine costruire una rightmost inversa ed ogni operazione di reduce corrisponderà ad un handle

8.22 Conflitti per le grammatiche LR

I conflitti che posso avere sono:

- **shift reduce (o reduce shift):** si ha un conflitto in quanto **nello stesso insieme di stati si può effettuare sia un'operazione di reduce che di shift sullo stesso simbolo**. Per la reduce bisogna sempre **calcolare il FOLLOW del non terminale della produzione**
- **reduce reduce:** si ha il conflitto in quanto **nello stesso insieme di stati si possono effettuare due riduzioni diverse sullo stesso simbolo nel FOLLOW**. Se i FOLLOW degli item completi sono **differenti, allora non si ha un conflitto**. In altri termini, il $\text{FOLLOW}(A) \cap \text{FOLLOW}(B)$ hanno un elemento in comune

In generale, **quando c'è l'item completo devo preoccuparmi, in quanto l'item completo produce la reduce e con la reduce potrebbero esserci dei conflitti**. Quando troviamo un item completo bisogna preoccuparsi in quanto potrebbe portarci ad un conflitto. Ciò però vale se l'item è in uno stato in cui ci sono altri item. Anche nel caso in cui l'item completo sia in uno stato con altri item, il conflitto si avrà solo se il FOLLOW del non terminale dell'item completo si sovrappone ad un altro FOLLOW di un altro item completo o ad un altro simbolo che è preceduto da \cdot in quel stato. Sarà possibile avere due archi con lo stesso nome nell'automa, fintanto però che non partano dallo stesso stato. Ogni item mi porta ad un'azione pertanto **quando ho un item da solo in uno stato non mi preoccupo**

8.22.1 Esempio di conflitto

Dato uno stato come il seguente: In questo caso, **arrivo in un insieme di items in cui**

$$\begin{array}{l} A \rightarrow a \cdot \\ B \rightarrow C \cdot d \end{array} \{ \# \}$$

Figura 8.7: Esempio di conflitto

sono attive due produzioni, di cui la prima ho visto la parte sinistra pertanto è un item completo, della **seconda invece è attiva** ma ho visto solo una parte.

Andando a fare la tabella su questo stato, **potrei ritrovarmi in una cella in cui, come in questo caso, leggendo "d" dovrei fare una reduce in quanto $A \rightarrow a \cdot$ deve essere ridotto proprio perché $\text{FOLLOW}(A) = \{d\}$ ma devo fare anche uno shift in quanto $B \rightarrow C \cdot d$ si aspetta una "d"**. La presenza di questo conflitto potrebbe essere notata prima di completare

l'intero automa, pertanto **non bisogna svilupparlo sempre fino alla fine**. A livello di tabella (in rosso è segnato il conflitto) avrei:

	=	\$
2	shift 6 reduce 5	reduce 5

8.23 Follow contestuali

Calcolare il **look ahead con il FOLLOW** è un approccio semplicistico. Esiste un approccio più preciso in cui i **FOLLOW** hanno conto del contesto ed essi prendono il nome di **FOLLOW CONTESTUALI**.

Conoscere il contesto ci può aiutare nell'essere più precisi. Quando si passa da **FOLLOW** a **FOLLOW CONTESTUALE** sto passando da **SLR(1)** a **LR(1)**. Il **FOLLOW** normale diventa quindi dozzinale, perchè **non fa altro che unire tutti i FOLLOW CONTESTUALI del non terminale che stiamo considerando**. È come se quel **non terminale** potesse essere seguito da **tutto**, ma così non è, in quanto dipende dal contesto, ovvero dallo stato in cui sono. In buona sostanza, il **FOLLOW** è dozzinale mentre il **FOLLOW CONTESTUALE** è più preciso.

8.23.1 Esempio: Impatto del follow contestuale

Data la grammatica:

- 0) $S \rightarrow CC$
- 1) $C \rightarrow c C$
- 2) $C \rightarrow c$

Come possiamo osservare, **nella produzione 0 ci sono due C** ma esse hanno contesti **differenti**. La seconda C, infatti, **potrebbe essere seguita nella prima fase solo dal fine stringa**. Facendo il $\text{FOLLOW}(C)$ in generale devo andare a considerare sia la prima che la seconda C ottenendo $\{=, \$\}$. **Tenendo però conto del contesto** (per semplicità aggiungo in questo caso gli indici alle C per una maggiore comprensione, facendo diventare la produzione $S \rightarrow C_1 C_2$) **ottengo** che:

- $\text{FOLLOW CONTESTUALE}(C_1) = \{c\}$
- $\text{FOLLOW CONTESTUALE}(C_2) = \{\$\}$

8.24 Algoritmo ed automa LR(1)

L'algoritmo LR(1) è identico all'SLR(1) per quanto riguarda la filosofia di costruire gli insiemi di items. Ciò che cambia è il calcolo del FOLLOW, ovvero il calcolo del look ahead delle reduce. Quando passo da un FOLLOW ad un FOLLOW CONTESTUALE sto passando da SLR(1) a LR(1) e dirò che **LR(1) è più potente dell'SLR(1) in quanto sarà più preciso sul quando fare le reduce**. Ci saranno quindi delle grammatiche che non sono SLR(1) ma che sono LR(1). La tabella ottenuta dagli automi LR(1) sarà strutturata allo stesso modo della SLR(1) e della LALR(1). Anche l'algoritmo di parsing sarà lo stesso per tutti e tre. Dato l'automato LR(1), costruisco la tabella di parsing LR(1) e su questa tabella verrò applicato l'algoritmo LR(1)

8.25 Item LR(1)

Finora, gli item $X \rightarrow \cdot X_1$ erano item LR(0) in quanto non ci interessiamo del look ahead che veniva messo solo dopo aver calcolato gli item. Nell'algoritmo LR(1) dobbiamo considerare gli item LR(1), pertanto verranno indicati anche i follow possibili quando l'item diventerà completo, avendo tipo $X \rightarrow \cdot X_1 \$$, dove $\text{FOLLOW}(X) = \{\$ \}$. Quando il \cdot è all'interno della produzione, non saremo interessati ai simboli del FOLLOW, in quanto il look ahead sarà il simbolo successivo al \cdot all'interno della produzione.

Ci portiamo comunque dietro queste informazioni in quanto ci serviranno quando lo stesso item diventa completo. Quando l'item diventa completo, le informazioni sul FOLLOW che prima non mi servivano a niente, diventano fondamentali. L'item LR(1) è quindi formato da un item LR(0) e dei terminali accanto, dove i terminali saranno utili solo quando l'item diventa completo. Questi non terminali me li porto però con me fin dall'inizio, trovandomeli accanto agli item completi.

In buona sostanza, andrò a fare il FIRST di ciò che segue, ma il FIRST lo vado a fare in modo contestuale, tenendo conto di quale non terminale sto espandendo in quel momento.

8.26 Chiusura e goto sugli item LR(1)

Se ad esempio sto facendo la chiusura di L partendo da $S \rightarrow \cdot L = R \$$, per calcolare il look ahead delle produzioni ottenute da tale chiusura dovrò fare $\text{FIRST}(=) \cup_{\epsilon} \text{FIRST}(R) \cup_{\epsilon} \text{FIRST}(\$)$. Nel peggiore dei casi, ovvero in cui sia $=$ che R vadano in ϵ , mi ritroverò sicuramente con il look ahead di $S \rightarrow \cdot L = R \$$, in quanto anche il look ahead deve essere

incluso nel FIRST. Ciò perché inserendo anche il look ahead nel FIRST, ad ϵ non ci arriverò mai

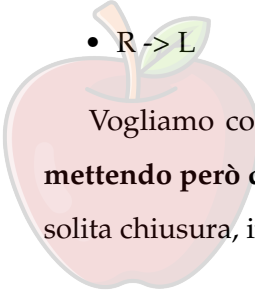
Facendo la goto, invece, io mi copio l'intero LR(1) item, spostato solo il \cdot . Nelle operazioni di goto, il look ahead viene copiato e nulla viene trasformato. In generale, l'aggiunta del look ahead rende diverse cose che prima erano uguali.

8.26.1 Esempio: esecuzione grammatica LR(1)

Data la grammatica:

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

Vogliamo costruire un automa LR(1) con item LR(1). Si parte sempre da $S' \rightarrow \cdot S$, mettendo però quello che servirà quando avrò l'item completo, ovvero $\$$. Applico poi la solita chiusura, inserendo per ogni produzione il follow contestuale



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

Figura 8.8: Esempio: esecuzione grammatica LR(1)

Per $S \rightarrow \cdot L = R$ ed $S \rightarrow \cdot R$, essendo che sto sviluppando la S di $S' \rightarrow \cdot S \$$, quando essa sarà completa, io vorrò vedere $\$$, pertanto anche per $S \rightarrow \cdot L = R$ ed $S \rightarrow \cdot R$ il look ahead sarà $\$$.

Allo stesso modo, devo fare la chiusura su L . Essendo che io sto espandendo L a partire da $S \rightarrow \cdot L = R$, facendo la chiusura avrò $L \rightarrow \cdot *R$ e $L \rightarrow \cdot id$. Quando sarà completa, dopo L in questo caso mi aspetto $=$, pertanto con il look ahead gli item diventano $L \rightarrow \cdot *R =$ e $L \rightarrow \cdot id =$.

Allo stesso modo, devo fare la chiusura su R. Essendo che sto sviluppando a partire da $S \rightarrow \cdot R$, facendo la chiusura ottengo $R \rightarrow \cdot L \$$. Mi ritrovo a dover fare la chiusura di L partendo però da $R \rightarrow \cdot L \$$ ed ottenendo $L \rightarrow \cdot * R \$$ e $L \rightarrow \cdot id \$$, che sono però diversi dagli item LR(1) precedenti.

Sviluppando ora l'insieme di item che con SLR(1) avrebbe avuto un conflitto, posso osservare come con LR(1) il conflitto non ci sia più, in quanto per l'item completo ho il look ahead su \$ e per l'item non completo ho il look ahead su =. Va infatti ricordato che il look ahead non serve sugli item non completi, seppure ce lo portiamo dietro a livello informativo.

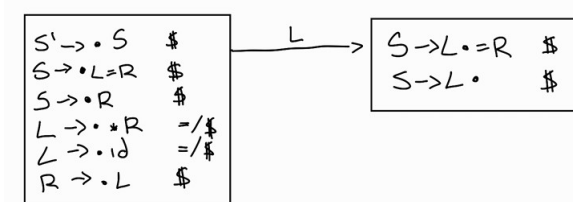


Figura 8.9: Esempio: esecuzione grammatica LR(1)

In questo caso la tabella sarà:

	=	\$
2	shift 6	reduce 5

8.27 Confronto tra LR(1) items

Due LR(1) items possono essere diversi anche solo a livello di look ahead. In generale quando io ho due items che hanno core o nucleo uguale e look ahead differente, non posso cancellare uno dei due items, in quanto differenti. A livello grafico, per semplicità, siccome il core dei due item sarà lo stesso, quello che facciamo è unire i due item LR(1) riportando i look ahead di entrambi gli items divisi da /. Ciò è visibile nel passaggio dallo stato sinistro a quello destro

8.28 Automi LALR(1)

A causa dell'aggiunta del look ahead, gli automi LR(1) potrebbero esplodere a livelli di stati in quanto hanno da tenere traccia di più cose. Al fine di minimizzare il numero di stati si può utilizzare LALR(1) compromesso tra SLR(1) ed LR(1), in quanto ha lo stesso numero di stati di SLR(1) ma è migliore di SLR(1)

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

$S' \rightarrow \cdot S$	\$
$S \rightarrow \cdot L = R$	\$
$S \rightarrow \cdot R$	\$
$L \rightarrow \cdot * R$	=/\$
$L \rightarrow \cdot id$	=/\$
$R \rightarrow \cdot L$	\$

Figura 8.10: Confronto tra LR(1) items

Per gli automi LALR(1) bisogna prima calcolare l'automa LR(1) e poi unire gli stati in cui ci sono item che si differenziano solo per il look ahead. In altre parole, per ottenere l'automa LALR(1) si fondono gli stati in cui ci sono items che hanno lo stesso core e si fa l'unione del look ahead. Facendo l'unione degli items, potrebbe sembrare che siamo tornati all'SRL(1) ma non è così, in quanto il numero di stati sarà uguale ma alcuni insiemi di look ahead saranno un sottoinsieme di quello di SLR(1).

Facendo la tabella per LALR(1) nel caso in cui ad esempio abbia fuso gli stati 4 e 7, nella tabella avrò lo stato 47. Facendo la fusione possiamo dimostrare che non possiamo generare conflitti shift reduce ma possiamo generare conflitti reduce reduce.

L'algoritmo di calcolare prima LR(1) e poi fondere gli stati non è ottimale, pertanto javacup utilizzerà un algoritmo più complesso che mentre costruisce fonde anche gli stati. Il fatto che una grammatica non sia né LL(1) né SLR(1) né LALR(1) né LR(1) non significa necessariamente che la grammatica sia ambigua. Potrebbe comunque non esserlo e non essere nessuna delle precedenti.

8.28.1 Esempio automa LALR(1)

Data la grammatica:

- 0) $S \rightarrow CC$
- 1) $C \rightarrow cC$
- 2) $C \rightarrow c$

L'automa LR(1) ottenuto sarà: In generale, prima di muovermi verso un insieme di items, io finisco di analizzare tutto lo stato precedente in termini di chiusure. In questo caso, possiamo fondere lo stato I_4 con lo stato I_7 . Posso fare lo stesso con I_8 ed i_9 e I_3 con I_6 .

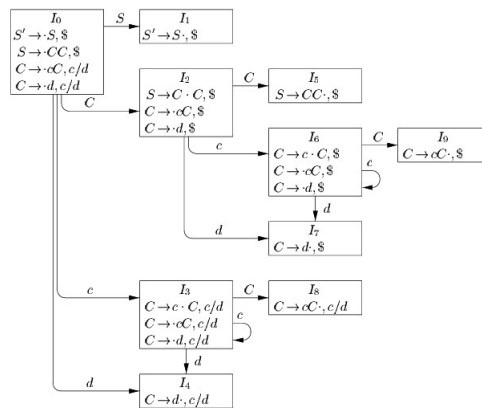


Figura 8.11: Confronto tra LR(1) items

8.29 Item di produzioni vuote

In generale, **item con produzioni che vanno in ϵ contano già come item completo** pertanto **avendo produzioni che vanno in ϵ è probabile che ci siano conflitti**. L'introduzione delle produzioni vuote, in sostanza, portano facilmente a conflitti, in quanto la produzione vuota è già un reduce. Se i look ahead collidono, avremo sicuramente qualche conflitto. Se c'è una reduce potrebbe esserci un conflitto mentre se c'è un conflitto c'è sicuramente una reduce.

Javacup fra shift e reduce **preferisce sempre le operazioni di shift**, mentre tra reduce/reduce sceglie sempre di ridurre la produzione scritta prima. Con il dangling else, esso è un conflitto shift reduce. Preferendo lo shift si attacca l'if all'else più vicino mentre preferendo la reduce al primo if.

8.30 Riassunto: Specifiche delle grammatiche

Abbiamo visto come i parser top down possano essere a discesa ricorsiva (con o senza backtrack o predittivo) o **LL(1) parsing che è sicuramente senza backtrack e sicuramente predittivo**. Per i parser bottom up abbiamo visto gli LR:

- **SLR(1)**: dove il look ahead è calcolato con follow
- **LR(1)**: dove i look ahead sono calcolati con follow contestuale
- **LALR(1)**: fusione di stati con lo stesso core nell'LR(1)

LR è sicuramente predittivo e non ha backtrack. Le grammatiche non hanno relazione 1 ad 1 con i linguaggi, pertanto se la grammatica è cattiva non è colpa del linguaggio. In generale,

le grammatiche si dividono in ambigue e non ambigue e tutto quello che abbiamo fatto era nel mondo delle non ambigue

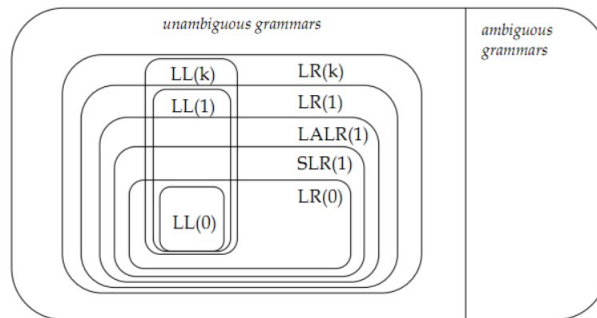


Figura 8.12: Riassunto: Specifiche delle grammatiche

8.31 Riassunto: relazioni tra grammatiche

Quando si parla di parser per le **grammatiche ambigue** si parla di **parser generali**, i quali producono foreste e non alberi. Una grammatica potrebbe essere non LR(1) ed essere comunque non ambigua. In generale, se la grammatica è SLR(1) sicuramente è sia LR(1) che LALR(1) ed è anche non ambigua. Ci sono grammatiche LL(1) che sono SLR(1) come grammatiche LL(1) che non lo sono. Avendo una grammatica ambigua, possiamo escludere il fatto che sia LR(1) e così via, dovendo però identificare la frase che rende quella grammatica ambigua. Una grammatica LR(1) non è detto che sia SLR(1). Si ha un automa SLR(1) quando si usano item LR(0) mentre si ha un automa LR(1) quando si usano item LR(1). Tali modelli sono sempre predittivi in quanto stando in uno stato e volendo spostare al successivo, devo sapere quale sia il prossimo terminale. Se non vedo il prossimo terminale non so cosa fare. Essendo predittivo, la scelta è influenzata dal simbolo che vedo. Negli item LR(1) i look ahead di item non completi saranno chiamati **look ahead di servizio**, in quanto non mi serviranno. I conflitti che si potranno avere sono solo shift/reduce e reduce/reduce (accept/reduce è un conflitto ma viene trattato come reduce/reduce).

In generale possono esistere grammatiche che sono LR(1) ma che non sono LALR(1), in quanto tramite l'unione di stati possono nascere conflitti reduce/reduce. Ciò capita con LALR(1) in quanto con l'aggregazione degli stati si modifica l'informazione. Una grammatica è ambigua se ci sono due leftmost diverse, e le leftmost non sono altro che una visita depth first da sinistra a destra.

Il numero di stati tra SLR(1) ed LALR(1) sono gli stessi ciò che cambia è che LALR(1) potrebbe avere i look ahead più rifiniti

8.32 Riassunto: Algoritmo LR

Per le grammatiche bottom up **bisogna applicare la riduzione e per applicare la riduzione su degli elementi è necessario che nello stack ci siano tutti gli elementi della produzione**. Sarà possibile **metterli nello stack tramite operazioni di shift sull'input**. Per l'automa bisogna parlare invece di items. Gli **item iniziali sono quelli il cui puntino è all'inizio**, mentre gli **item completi sono quelli il cui puntino è alla fine e che mi generano un'operazione di riduzione**. Ho visto tutto **pertanto la parte destra della produzione è sullo stack**. Le operazioni ammissibili sono **shift, reduce, error ed accept**. A livello di algoritmo **reduce si appoggia poi su goto per far tornare al top dello stack uno stato**.

8.32.1 Esempio: Realizzazione dell'automa LR(1) per la gramamtica postfissa

Data la grammatica:

- 0) $S \rightarrow SS +$
- 1) $S \rightarrow SS^*$
- 2) $S \rightarrow a$

Essa è la grammatica delle notazioni postfisse. **Per vedere se la grammatica sia LR(1) bisogna costruire l'automa e la tabella**. Le operazioni che useremo sono closure e goto. Per questo automa:

- **Stato 0:** Faccio la closure su S. **Il look ahead lo calcoleremo come il FIRST di tutti gli elementi che sono dopo il simbolo puntato**, in questo caso S. Continuo ad applicare closure fin quando aggiungo items nuovi. **Alla fine delle closure, posso andare ad unire gli items con stesso core o nucleo e look ahead differente**, in questo caso " $S \rightarrow \cdot SS +, \$$ " con " $S \rightarrow \cdot SS +, a$ ", " $S \rightarrow \cdot SS^*, \$$ " con " $S \rightarrow \cdot SS^*, a$ ", ed infine " $S \rightarrow \cdot a, \$$ " con " $S \rightarrow \cdot a, a$ ". Essendo solo un'abbreviazione e non una modifica dell'informazione, tale abbreviazione non mi cambia nulla, cosa che invece succede in **LALR(1) in quanto con l'aggregazione degli stati si modifica l'informazione**
- **Stato 1 (0, S):** Ho finalmente un item completo, ovvero " $S' \rightarrow S \cdot, \$$ ". **Mi porto anche le produzioni che aspettavano S e mi serve in questo caso calcolare la closure e così via**. Esso è un **insieme di item pericoloso proprio a causa del fatto che abbia una reduce ed altre operazioni**. Durante i goto i look ahead rimangono invariati. Lo shift si fa sempre sugli item che hanno il \cdot prima del simbolo terminale

- **Stato 2 (0, a):** anche in questo caso ho un item completo
- **Stato 3 (1, S):** mi porto gli items che si aspettavano S e faccio la closure. Causa delle prime due produzioni non posso però creare un ciclo nel caso in cui arrivi un'altra S.
- **Stato 4 (1, a):** ho un altro item completo. Esso ha lo stesso core dello stato 2 ma ha look ahead differenti
- **Stato 5 (3, +):** Ho un altro item completo. Esso ha lo stesso core dello stato 8 ma ha look ahead differente
- **Stato 6 (3, *):** Ho un altro item completo. Esso ha lo stesso core dello stato 9 ma ha look ahead differente
- **Stato 7 (3, S):** Su questo stato posso reiterare leggendo S in quanto volendo creare un nuovo stato ne creerei uno identico a 7 stesso. Essendo che non posso avere stati uguali ripetuti, reitero sullo stesso stato

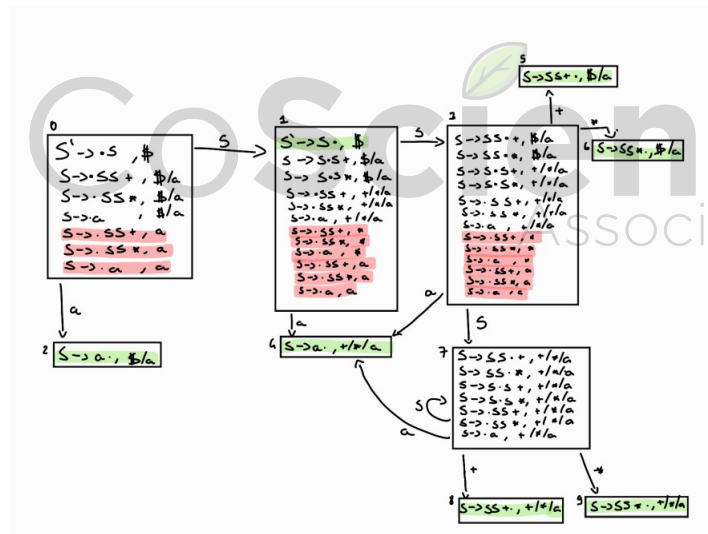
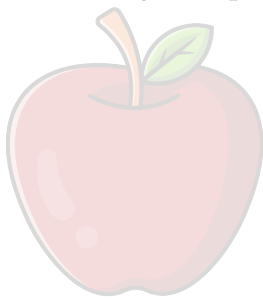


Figura 8.13: Realizzazione dell'automa LR(1)

La tabella è:

	+	*	a	\$	S
0			S2		1
1			s4	accept	3
2			r3	r3	
3	s5	s6	s4		7
4	r3	r3	r3		
5			r1	r1	
6			r2	r2	
7	s8	s9	s4		7
8	r1	r1	r1		
9	r2	r2	r2		

8.33 Conflitti shift/reduce negli automi LR(1)

Se la grammatica è LR(1) andando a creare l'automa LALR(1) non posso avere conflitti shift/reduce. Supponendo per assurdo che lo faccia, ciò dovrebbe essere il risultato di una fusione. Dovrei quindi avere due stati che si fondono e mi danno lo stato con shift reduce. Il look ahead che mi darà conflitto dovrà essere ereditato da uno dei due stati non fusi, ma in entrambi la sua presenza mi creerebbe un conflitto già a livello di LR(1). Ciò significa che anche la grammatica LR(1) dovrebbe avere un conflitto, pertanto la grammatica non sarebbe LR(1).

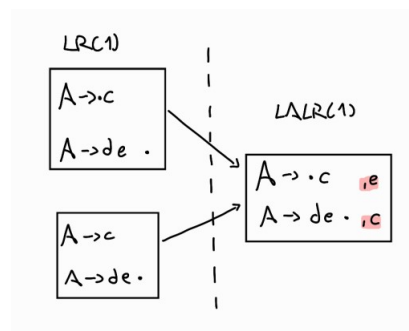


Figura 8.14: Conflitti shift/reduce negli automi LR(1)

9.1 Introduzione: Javacup

Esso ci permette di creare un parser bottom up LALR(1). Esso è meno potente di LR(1) ma ha meno stati. Su di esso non si potranno generare conflitti shift reduce. LR(1) gestisce più grammatiche parsabili, mentre SLR(1) ha molte meno grammatiche ed è meno potente.

Javacup prende una grammatica estesa con codice java e produce un parser LALR(1) in codice Java. Con javacup ci sono anche dei modi di eliminare i conflitti, come quello di definire le precedenze e sulla base di esse e la tabella dell'automa, si riescono a risolvere i conflitti.

9.2 Javacup e grammatiche ambigue

Con Javacup possiamo anche gestire grammatiche ambigue, basta che non ci siano conflitti. Possiamo specificare il numero di conflitti ammissibili e che javacup andrà poi a risolvere:

- tra shift e reduce prende sempre reduce
- tra reduce e reduce prende la produzione definita prima

Tocca poi al programmatore vedere se tale risoluzione dei conflitti va bene. Nelle grammatiche ad attributi, anche i terminali avranno un tipo. Gli attributi verranno usati per l'albero sintattico

9.3 Notazione di javacup

Alcune notazioni sono:

- per indicare la \rightarrow **bisogna usare $::=$** .
- Ogni non terminale verrà chiuso con $''$;". In generale, **ogni assegnazione ha $''$;**
- nel caso in cui vogliamo indicare ϵ , **non bisogna mettere nulla dopo la $|$** . In generale è buona norma mettere il commento `/* empty */`

9.4 Visione generale dell'analizzatore sintattico

Abbiamo visto come l'analizzatore lessicale **legge le stringhe dal file sorgente e risponde alle richieste dell'analizzatore sintattico**. L'analizzatore sintattico, come il lessicale è stato diviso in due parti: **riconoscimento ed azione**. Nell'analizzatore lessicale l'azione era **aggiungere una tabella dei simboli o delle parole e restituire il token all'analizzatore sintattico**. L'analizzatore sintattico **prende un flusso di token che possono avere o meno un attributo**. I **token nell'analizzatore sintattico vengono utilizzati per costruire l'albero sintattico**.

Abbiamo visto che una **sequenza di token diventa una sequenza di terminali, ovvero la sentenza**. Se il nostro **riconoscere riconosce la sentenza** è in grado di applicare le produzioni della grammatica e quindi ricostruire un albero di derivazione. Tutto ciò è la prima parte dell'analizzatore sintattico. **L'albero di derivazione non è altro che un trace di esecuzione, in quanto percorro varie produzioni fin quando non mi trovo la sentenza**.

L'albero di derivazione dipende dalla grammatica mentre l'albero sintattico no. Avendo **grammatica differente, avrei un albero di derivazione differente** mentre l'albero sintattico sarebbe rimasto invariato. **Come albero sintattico vogliamo la minima rappresentazione del nostro input**

9.5 Tools per i parser

I riconoscitori rispondono solo **"si"** e **"no"** mentre i parser restituiscono **gli alberi di derivazione o la sequenza di produzioni utilizzate**. In generale, i parser possono essere:

- I top down si **prestano per costruire un parser a mano ma è possibile costruirli anche in modo automatico tramite un tool chiamato Antlr** che però si basa su parser LL pertanto è necessario disambiguare la grammatica, fattorizzare la grammatica... anche

se **nelle ultime versioni antlr fa tutto da solo**, riuscendo a eliminare la ricorsione sinistra, fattorizzazione

- Per i **parser bottom up non c'è la versione manuale ma c'è solo una versione automatica**. Uno dei tool è Yacc (Yet Another Compiler Compiler), utile anche per costruire un intero compilatore monolitico (in tal caso si andrebbe a restituire non un albero sintattico ma bensì codice macchina). In generale, **quando nel nome c'è CC si intende Compiler Compiler, ovvero generatore di analizzatori sintattici**. Altro tool è Bison (lavorava in C++, mentre Yacc in C). Caratteristiche di questi tool è che prendevano un input una grammatica e davano in output il parser con la tabella. Per Java, esiste java cup o cup. Esso **prende in input una grammatica, costruisce la tabella di parsing LALR, aggiunge il parser che utilizza lo stack, input e tabella e produce una serie di classi java che fungono da parser**.

9.6 Funzionalità ed usi di Javacup

Javacup prende in input una grammatica e **genera il parser LALR, meno potente rispetto ad LR e più di un SLR**, pur mantenendo il suo stesso numero di stati. Javacup viene usato **sia per prendere una grammatica pulita e poi dare in output la sequenza di produzioni utilizzate, sia ha la possibilità di aggiungere alle proprie regole qualsiasi codice Java** pertanto possiamo fare molte cose.

Mentre con Jflex abbiamo aggiunto ad ogni espressione regolare un'azione in codice java, **lo stesso possiamo fare con javacup**. Tale codice si prende in input ciò che abbiamo riconosciuto e lo elabora. Nel caso dei compilatori tale codice viene usato **o per eseguire direttamente il codice o per generare l'albero sintattico**. Per poter aggiungere tali azioni la grammatica non sarà pura ma sarà una grammatica context free con delle azioni che si basano su attributi, uscendo fuori il concetto di grammatica ad attributi

9.7 Javacup lato codice

Una **grammatica con 0 conflitti non accetta alcun tipo di conflitto**. Se la grammatica presenta conflitti ma noi avevamo specificato 0 nel pom.xml durante la realizzazione del Parser, quest'ultimo non verrà costruito. **Verranno però evidenziati i conflitti e gli stati contenenti i conflitti e se non dovremo gestire nessun conflitto dovremo agire sulla grammatica affinché quegli stati non abbiano più conflitti**. In alcuni casi i conflitti non ci danno fastidio, questo

perché **javacup** può gestire conflitti **shift/reduce** o **reduce/reduce**. Un esempio di conflitto che non ci dà fastidio è quello legato al problema del dangling else

9.8 Grammatiche ad attributi e javacup

Inizialmente si scrive solo la grammatica per evitare complicazioni, poi se il programma è corretto sintatticamente iniziamo ad aggiungere altro. Nella grammatica ad attributi quello che cambia è che gli attributi non li hanno solo alcuni terminali ma anche i non terminali. ogni regola grammaticale avrà un'azione scritta in Java che calcola l'attributo del non terminale a sinistra della produzione

Per gli attributi, dovrà esserci consistenza tra il tipo che spara l'analizzatore lessicale come attributo ed il tipo dell'attributo che l'analizzatore sintattico riceve. Mi serve poi un meccanismo per accedere all'attributo del terminale o del non terminale

Avendo un parser **bottom up** ovvero in cui dai figli riduco il padre, partendo dagli attributi dei figli posso calcolare l'attributo del padre. I terminali o non terminali con attributo si presenteranno come: **RESISTOR:r** In questo caso, **RESISTOR** è il terminale ed **r** è l'attributo. Dovrò aggiungere ad ogni produzione una regola che mi calcola gli attributi dei non terminali. Un esempio potrebbe essere : **RESULT = new Double(r);** ∴ Il codice è semplice in quanto fa semplicemente un passaggio di **r** a **double**. Non è possibile che **r** non rappresenti il valore corretto poichè altrimenti non ci sarei arrivato prima

Usare **RESULT** indica che sto calcolando l'attributo del non terminale in cui sto riducendo. **RESULT** rappresenta, inoltre, uno standard e fa parte della specifica di **javacup**

9.9 Operazioni di riduzione nelle grammatiche ad attributi

Mentre io prima riducevo e basta, quando riduco nelle grammatiche ad attributi devo dare un attributo al padre, rispettando il tipo che il padre, ovvero il non terminale, si aspetta. Il non terminale va a quindi a sintetizzare, dal figlio ridotto, l'attributo. Il non terminale si va a calcolare l'attributo partendo da quello del figlio. Ogni riduzione va a calcolare l'attributo del padre e alla fine anche **S**, ovvero il non terminale iniziale, otterrà un attributo. Per leggere tale attributo basta aggiungere **“.value”** alla funzione **parse()**.

In generale, ciò che a noi interessa è prendere l'attributo, sia esso di un terminale o non terminale e lo si va ad usare. Non si capisce l'ordine delle azione. Non so quando un'azione verrà eseguita e quando verrà eseguita un'altra. Per capirlo prendiamo l'albero ed ogni

volta che facciamo una riduzione applichiamo una formula per calcolare l'attributo del padre.

9.10 Comandi di CUP

Comandi sono:

- **init with:** utile se vogliamo specificare cosa fare prima che venga eseguito il parser
- **scan with:** utile per importare una funzione precisa, specificando la chiamata da fare. Ciò è utile solo se apportiamo modifiche a qualcosa di Lexer o Parser. Se omettiamo tale codice la funzione richiama le solite funzioni
- **terminal:** specifica i terminali che per convenzione sono a lettere maiuscole. Non tutti i terminali devono avere attributi
- **non terminal:** specifica i non terminali, per convenzione a lettere minuscole. Non tutti i non terminali devono avere attributi
- **precedence:** ci permette di specificare le precedenze tra i vari operatori. Gli elementi che sono sulla stessa riga hanno la stessa precedenza. Chi è scritto prima ha precedenza minore, pertanto l'ordine con cui mettiamo le righe con "precedence" è importante
- **left:** sta ad indicare il tipo dell'associatività nelle operazioni come ad esempio moltiplicazione, somma, sottrazione... che hanno tutte associatività a sinistra, mentre l'esponenziale ha associatività a destra

Se all'interno di una produzione metto una chiamata a funzione o una produzione, devo metterla in action. Posso poi usare class per cambiare il nome della classe Parser

9.11 Produzioni e precedenze

Ogni produzione ha una precedenza. In caso di conflitti shift/reduce o reduce/reduce ho due items che si contendono un'azione e vince colui che ha la precedenza maggiore. La precedenza di una produzione è data dalla precedenza dell'ultimo terminale. Colui che definisce la precedenza per una produzione è il terminale più a destra in quella produzione.

Quando io ho **shift/reduce** e **reduce/reduce** ho due items che collidono. Sotto ogni item ho una produzione e per decidere quale eseguire vado a vedere la precedenza. Se una produzione ha un precedenza maggiore dell'altra vince lei. La precedenza di una produzione è data o con una keyword ovvero %prec oppure andando a considerare la precedenza dell'ultimo terminale di quella produzione. Sarò io quindi a decidere chi vince nello shift/reduce o reduce/reduce. Nel caso omettiamo le precedenze sarà il compilatore poi a risolvere da solo i conflitti

9.12 Aggiunta degli attributi alla grammatica

Una grammatica senza attributi restituisce esclusivamente l'albero ma non fa altro. Se io voglio trasformare il riconoscitore in un interprete, ovvero dato una frase non voglio che mi sia detto se la frase sia corretta o meno ma bensì voglio il risultato finale, nel caso in cui la frase sia un'espressione, devo aggiungere gli attributi, arrivando ad una grammatica ad attributi. Voglio costruire, partendo dalla grammatica, un piccolo interprete. Tutti non terminali avranno degli attributi

9.12.1 Esempio: Esecuzione del Parser data una stringa

Data l'espressione numerica: $5 + -(3*4)$; A livello di realizzazione di un parser bottom up, i passaggi che vengono effettuati sono:

- **Leggo il Simbolo 5:** In fase lessicale 5 diventa $\langle \text{NUMBER}, 5 \rangle$ mentre a livello di analisi sintattica avendo la produzione diventa (expr, 5)
- **Leggo il Simbolo +:** In fase lessicale esso diventa $\langle \text{PLUS} \rangle$ e così rimane anche in fase di analisi sintattica
- **Leggo il Simbolo -:** in fase lessicale esso diventa $\langle \text{MINUS} \rangle$ e così rimane anche in fase di analisi sintattica
- **Leggo il Simbolo (:** in fase lessicale esso diventa $\langle \text{LPAREN} \rangle$ e così rimane anche in fase di analisi sintattica
- **Leggo il Simbolo 3:** similmente a come successo con il simbolo 5, in fase di analisi lessicale diventa $\langle \text{NUMBER}, 3 \rangle$ mentre a livello sintattico applicando la produzione diventa (expr, 3)

- **Leggo il Simbolo *:** in fase lessicale esso diventa <TIMES> e così rimane anche in fase di analisi sintattica
- **Leggo il Simbolo 4:** similmente a come successo con il simbolo 5 e 3, in fase di analisi lessicale diventa <NUMBER, 4> mentre a livello sintattico applicando la produzione diventa (expr, 4)
- **Applico la produzione ed eseguo l'azione:** Con la produzione $\text{expr}:\text{e1 TIMES expr}:\text{e2}$ quello che posso fare, seguendo quanto mi dice l'azione, è eseguire la moltiplicazione. In questo caso, $\text{expr}:\text{e1}$ sarà (expr, 3) mentre $\text{expr}:\text{e2}$ sarà (expr, 4)
- **Leggo il Simbolo):** in fase lessicale esso diventa <RPAREN> e così rimane anche in fase di analisi sintattica
- **Applico la produzione ed eseguo l'azione:** Posso ora ridurre LPAREN expr:e RPAREN avendo che $\text{expr}:\text{e}$ in questo caso è (expr, 12)
- **Applico la produzione ed eseguo l'azione:** Posso ora applicare la produzione MINUS $\text{expr}:\text{e}$ dove $\text{expr}:\text{e}$ è (expr, (12))
- **Applico la produzione ed eseguo l'azione:** Posso ora applicare la produzione $\text{expr}:\text{e1 PLUS expr}:\text{e2}$ su e1 ovvero (expr, 5) e e2 ovvero (expr, -(12))
- **Leggo il Simbolo ;:** In fase lessicale esso diventa <SEMI> e così rimane anche in fase di analisi sintattica
- **Applico la produzione ed eseguo l'azione:** Posso ora ridurre applicando la produzione $\text{expr}:\text{e SEMI}$ che ha come azione quella di fare un print a schermo, in quanto ho finito l'espressione.

9.13 Significato ed importanza delle azioni

Le azioni associate alle produzioni sono pezzi di codice che noi scriviamo ma non sappiamo il loro ordine di esecuzione. Tale configurazione può essere visto come uno **switch** elaborato il cui ordine di esecuzione non è causale ma quello di un **parsing LR**. Non si possono scrivere le azioni se non si ha in mente l'idea dall'ordine con cui verranno eseguite. La prima produzione che definiamo sarà quella che verrà eseguita per ultima. Pertanto bisogna scrivere l'azione della prima produzione tenendo a mente ciò. Qualunque azione che metteremo alla prima produzione sarà l'ultima.

Le prime produzioni ad essere eseguite saranno quelle con i terminali. Saranno poi gli handle a dire cosa ridurre e pertanto in fase di definizione delle azioni sarà importante tenere a mente gli handle. La programmazione delle azioni è non sequenziale il cui ordine me lo dà il parsing bottom up. L'ordine sarà dato a priori ma non sarà quello a cui si è abituati. Andando a prendere tutte le istruzioni eseguite durante il parsing, ottengo un programma che è una permutazione delle azioni delle produzioni. Ogni frase che viene parsata genera un programma.

In un processo di sviluppo di un linguaggio di programmazione, **prima andiamo a scrivere la grammatica con i terminali e poi le espressioni regolari** (prima cup e poi flex). Sarà la grammatica a dirci quali sono i non terminali di cui abbiamo bisogno.



10.1 Introduzione: Grammatica ad attributi

I terminali, già di per sé potevano o no potevano avere attributi. Ciò che è nuovo è che **nelle grammatiche ad attributi anche i non terminali possono avere degli attributi**. Essi avranno un tipo pertanto quando vado a scrivere in javacup i terminali e non terminali **devo dire se hanno un attributo associato** (a volte anche più di uno ma in tal caso lato codice si costruisce una struttura).

Quando costruiamo l'albero sintattico il non terminale iniziale avrà come attributo il riferimento alla radice dell'AST. Da ricordare infatti che **l'albero di derivazione sarà costruito in modo bottom up, partendo dalle foglie e salendo fino alla radice, costruendo gli attributi di tutti i nodi dell'albero di parsing**.

Quando carico un albero di parsing, **costruisco un albero sintattico, levando tutta la ridondanza e lo faccio diventare compatto**, indipendente dalla grammatica ma che abbia le informazioni minime ed indispensabili del sorgente

10.2 Calcolo degli attributi

L'analisi lessicale calco gli attributi dei terminali. è l'analisi lessicale che **spara al parser sia il token che l'attributo**. Per i non terminali, li dovremo calcolare mentre facciamo il parsing, ovvero durante il parsing. Tali attributi **si calcolano tramite delle regole che**

vengono chiamate regole semantiche. La notazione di queste regole dipende da colui che le descrive.

Per javacup le regole sono espresse tra `< >` e per fare riferimento agli attributi si mettono dei nomi e tali nomi li si associano alle produzioni, ai non terminali. In generale, `RESULT` sarà una parola chiave che fa riferimento all'attributo del non terminale. Le operazioni definite tra `< >` hanno senso se, applicando la regola, ho già gli attributi che servono per calcolare l'operazione, altrimenti sto usando valori non definiti. In javacup gli attributi saranno chiamati con una label e tale label sarà poi usata per le operazioni

10.3 Difficoltà delle grammatiche

La difficoltà della grammatica ad attributi è proprio quella di **stare attenti, a quando si calcola un attributo, che tutti gli attributi che mi servono per calcolare l'attributo siano stati già calcolati**. Per il parsing bottom up, siccome so che si fanno le riduzioni e che derivo l'attributo del padre da quello dei figli, basta che mi calcolo prima l'attributo dei figli e poi del padre

10.4 Grammatiche S-attribuite

In generale, l'attributo del padre non dipende sempre da quello dei figli. Una grammatica nel quale esiste questa dipendenza è S-attribuita ed essa rappresenta un sottoinsieme delle grammatiche ad attributi. Tali grammatiche S-attribuite sono facili da calcolare per il parsing bottom up.

Una grammatica è S-attribuita se ogni volta che calcolo un attributo di un non terminale, lo calcolo a partire da quello dei figli. Il parsing bottom up mi garantisce che i figli li ho già visitati, in quanto nel parsing bottom up vedo prima i figli. A livello algoritmico essa è una visita post order, quando calcolo il nodo di un attributo, i figli li ho già visti

10.5 Traduzione diretta dalla sintassi

Esistono altri modi per annotare le grammatiche ad attributi oltre alle tecniche usate da javacup. In generale, si parla di **traduzione diretta dalla sintassi**, questo perché tutto ciò che facciamo con un qualsiasi programma non è altro che una traduzione. Anche una calcolatrice va a tradurre, ovvero un'espressione aritmetica viene tradotta nel suo valore.

L'insieme dei valori è quindi un altro linguaggio banale dato da tutti i numeri naturali. Si prende una stringa e la si traduce in un numero che è un linguaggio banale le cui sentenze sono solo numeri.

Con **traduzione diretta dalla sintassi** si intende che andiamo a mettere regole sulla grammatica. Con una grammatica ad attributi posso tradurre una frase in qualsiasi cosa voglio. Si usa la grammatica ad attributi per tradurre il flusso di token in albero sintattico. In altri casi si potrebbe tradurre direttamente in codice intermedio, scrivendo delle regole che non scrivono l'albero sintattico ma producono codice intermedio. Si potrebbe fare un controllo di analisi semantica...

Indipendentemente da cosa metto nelle regole, **posso far fare alla grammatica un qualsiasi tipo di traduzione.**

10.5.1 Funzionalità della traduzione diretta dalla grammatica

La traduzione diretta dalla grammatica fa due cose:

- **controlla che la frase sia corretta**
- **la traduce nel mio obiettivo**

La grammatica è quindi uno strumento come jflex che viene usato nei compilatori ma potrebbe essere usata anche in altri contesti. Per fare la traduzione diretta dalla grammatica abbiamo bisogno di una grammatica ad attributi. Ogni simbolo può avere un attributo

10.6 Tipi di traduzione diretta dalla grammatica

Possiamo avere una **definizione guidata dalla sintassi**. In tal caso gli attributi non sono inseriti nella produzione ma esternamente. La regola la posso associare dichiaratamente o posso dire quando quella regola deve essere eseguita, ovvero vado ad indicare in quale punto della visita io devo lanciare quella regola. In genere abbiamo:

definizione guidata della sintassi: io dichiaro e non specifico quando e come va applicata la regola. Io associo in modo dichiarativo, dichiarando a cosa sia associata una regola

schema di traduzione: Quando si usano le {}, vuol dire che è uno schema di traduzione, ovvero stiamo dicendo anche quando dobbiamo lanciare quella regola. Con lo schema di traduzione specifico anche quando vada eseguita durante la visita. In javacup si usa lo schema di traduzione. L'azione può essere messa anche dentro la produzione, non solo

esternamente. Le regole le attacco alla produzione. In tal caso la posizione dell'azione fa la differenza, mentre quella ottenuta con la definizione guidata dalla sintassi dovremo preoccuparci successivamente di dove e come implementare. In generale, la definizione guidata dalla sintassi ci serve quindi per ragionare, ragionando sull'algoritmo che deve essere implementato e una volta che mi è chiaro la definizione la faccio diventare operativa e la metto tra {}

10.6.1 Esempio: Produzioni e regole semantiche

Data la regola: $E \rightarrow E_1 + T$

Usando la **definizione guidata dalla sintassi** è: $E:\text{code} = E_1:\text{code} \parallel T:\text{code} \parallel '+'$

Usando lo **schema di traduzione** si ha: $E \rightarrow E_1 + T \{ \text{print } '+' \}$

Siccome è difficile distinguere il simbolo code tra le due E, in tal caso, per quella produzione, la seconda E diventa E_1 . Sono ancora lo stesso non terminale ma mi serve soltanto per capire a cosa faccia riferimento code. In tal caso con la regola semantica sto trasformando una notazione infissa in una postfissa. Ottengo un'espressione postfissa in quanto l'azione del print del + sta alla fine della produzione. Se l'avessi messa all'interno della produzione avrei ottenuto una notazione infissa. Se mettessi l'azione di print prima della E_1 , otterrei una notazione prefissa

10.7 Tipi di Attributi e non terminali

Gli attributi dei non terminali vanno calcolati. Posso calcolare l'attributo del non terminale sinistro o posso mettere una regola che mi calcola gli attributi. Dire se un attributo è sintetizzato o ereditato dipende dalla regola che lo calcola. A seconda di come la regola calcola gli attributi, essi saranno o sintetizzati o ereditati. Nello specifico:

- **attributi sintetizzati:** Se c'è una regola che mi calcola l'attributo del non terminale a sinistra, dirò che è sintetizzato, ovvero sintetizzo. Tali regoli sintetizzate dipendono dagli attributi dei simboli che stanno a destra della produzione. Facendo il parsing bottom up e calcolando il simbolo a sinistra sono sicuro che tutti i simboli a destra li ho già visti. Ciò è molto importante. Se la regola calcola l'attributo del padre in funzione dell'attributo dei figli, l'attributo viene chiamato sintetizzato
- **attributi ereditati:** si ha un attributo ereditato quando la regola calcola l'attributo di un simbolo che si trova a destra della produzione. Tale attributo da calcolare può

dipendere o dagli attributi dell'elemento sinistro o dagli attributi fratelli, ovvero coloro che si trovano nella parte a destra. Nell'esempio di prima con la produzione $E \rightarrow E_1 + T$, i non terminali E_1 e T sono fratelli, mentre E è il padre. Se la regola calcola l'attributo di un figlio in funzione di quello del padre o di quello dei fratelli, allora si chiama ereditato.

In generale, l'operazione di sintesi è un'operazione dal basso verso l'alto ovvero dai figli al padre, mentre l'operazione di ereditarietà è dall'alto verso il basso, ovvero dal padre ai figli o allo stesso livello, ovvero tra fratelli. La cosa naturale è usare bottom up per gli attributi sintetizzati e se si ha bisogno di attributi ereditati si utilizza un algoritmo top down, in quanto dal padre scende giù. Tale è una scelta mutuamente esclusiva. Questo però se voglio fare la traduzione durante il parsing, ovvero se mentre riconosco voglio anche generare l'albero. Un altro passo che si potrebbe fare è fare il riconoscimento, ricostruire l'albero di derivazione, metterci tutti gli attributi e poi leggere l'albero di derivazione e fare la traduzione

10.8 Ordine topologico e parsing bottom up

L'ordine topologico centra in quanto devo fare una visita dei nodi dell'albero di derivazione. Facendo il grafo delle dipendenze saprò che un attributo dipende da altri. Facendo l'ordine topologico saprò che quando raggiungo un nodo ho calcolato quelli da cui dipende. Se il grafo non è aciclico non potrò calcolare il valore degli attributi. Devo trovare l'ordine topologico affinché ogni volta che mi trovo a calcolare un attributo ho calcolato tutti quelli da cui lui dipende. Con la visita bottom up e gli attributi sintetizzati l'ordine topologico è proprio una visita bottom up e mi garantisce che ogni volta che raggiungo un nodo ho visto già quello dei figli.

10.9 Grammatica L-ereditata

Con gli attributi ereditati, il valore di un non terminale può dipendere dal padre o dai fratelli. Vedendo un non terminale B , esso sarà figlio di qualcuno e nel caso di attributo ereditato esso dipende o dai fratelli o dal padre. Qui diventa più difficile calcolare l'ordine topologico. Se l'attributo di B dipende solo dai fratelli sinistri, con una visita dell'albero depth first da sinistra a destra non ho problemi. Ogni volta che incontro un nodo ho incontrato sicuramente già il padre e già i fratelli sinistri. Una visita depth first da sinistra

a destra è proprio la tecnica usata dal parsing top down. Tale grammatica si chiamerà L-ereditata. Se B dipende da un nodo che dipende anche da un nodo a destra, la grammatica non sarà L-ereditata

è difficile implementare una grammatica L-attribuita con un parser bottom up. Ci sono però dei trucchi per particolari grammatiche per implementare con bottom up.

10.10 Grammatiche S ed L attribuite

Da notare che è S-attribuita se le regole calcolano solo gli attributi dei simboli a sinistra, mentre L-attribuita vuol dire che calcola anche gli attributi dei simboli a destra della produzione. Il concetto di L-attribuito include S-attribuito. Questo perché facendo una visita top down da sinistra a destra, ad un certo punto andremo a risalire l'albero. Facendo una depth first da sinistra a destra io scendo nei figli, mi calcolo gli attributi ma poi risalgo e nella risalita posso calcolare gli attributi sintetizzati.

In generale, S-attribuita è un sottoinsieme di L-attribuita. L-attribuita ha sia attributi ereditati che sintetizzati, mentre S-attribuita solo sintetizzati. Quando la regola calcola l'attributo del padre, esso è un attributo sintetizzato. Quando la regola calcola l'attributo del figlio, quell attributo è ereditato. Gli ereditati sono associati al top down con visita depth first da sinistra a destra, mentre i sintetizzati sono associati ai bottom up. Da notare che ogni nodo ha una regola associata

Ho potuto fare un post order in quanto la grammatica è S-attribuita. La posizione della regola è sempre alla fine. Vedo prima tutti i figli. In questo caso si ha una definizione guidata dalla sintassi ma posso farlo diventare uno schema di traduzione dicendo che le regole devono essere applicate dopo aver visto la parte destra. L'importante è che la regola si lancia dopo che tutti i figli destri che ci servono sono stati visti

10.10.1 Esempio: Grammatica L-attribuita

Un esempio di grammatica L-attribuita è:

Production	Semantic Rules
1) $T \rightarrow F T'$	$T'.inh = F.val$ $T.val = T'.syn$
2) $T' \rightarrow * F T'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
3) $T' \rightarrow \epsilon$	$T'.syn = T'.inh$
4) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

Da notare che la regola 1 calcola due attributi. Nello specifico:

- $T \rightarrow FT'$
 - $T'.inh$: è un attributo di un figlio. Esso dipende dal fratello sinistro. Con tale regola la grammatica diventa candidata per una L-attribuita
 - $T.val = T'.syn$: esso è un attributo del padre ed è sintetizzato
- $T' \rightarrow * F T'_1$
 - $T'_1.inh = T'.inh * F.val$: esso calcola un figlio che dipende dall'attributo del padre e di un fratello che sta a sinistra. è quindi un attributo ereditato
 - $T'.syn = T'_1.syn$: è un attributo sintetizzato in quanto si va a calcolare il padre
- $T' \rightarrow \epsilon$
 - $T'.syn = T'.inh$: sto calcolando l'attributo del padre in funzione di un attributo ancora del padre. Siccome però sto calcolando l'attributo del padre, allora è sintetizzato
- $F \rightarrow \text{digit}$
 - $F.val = \text{digit.lexval}$: sto calcolando un attributo del padre in funzione del figlio

La grammatica è quindi L-attribuita. Se faccio un albero di derivazione e lo annoto con gli attributi, mi basta fare per l'ordine topologico, una visita depth first da sinistra a destra, Posso decidere se farlo durante un parser top down o mi faccio prima l'albero e poi l'ordine topologico.

10.10.2 Esempio: Albero di parse tree annotato

Data la grammatica:

Production	Semantic Rules
1) $L \rightarrow E n$	$L.val = E.val$
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	$E.val = T.val$
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	$T.val = F.val$
6) $F \rightarrow (E)$	$F.val = E.val$
7) $F \rightarrow \text{digit}$	$F.val = \text{digit.lexval}$

data la frase $3 * 5 + 4 n$, l'albero che ne esce fuori è:

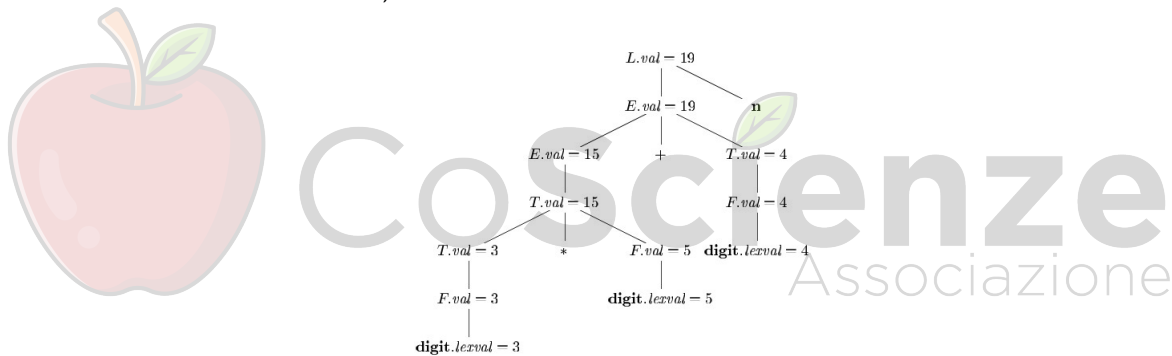


Figura 10.1: Esempio: Albero di parse tree annotato

Siccome la grammatica è S-attribuita, posso calcolare il valore finale facendo un algoritmo post order: scendo fino in fondo, calcolo il valore di un nodo che non ha figli e salgo. Quando ho visto tutti i figli calcolo:

- Parto da nodo **digit.lexval = 3** ottenuto dall'analisi lessicale
- Riduco **digit** in **F** applicando la regola, pertanto il valore 3 migra e sale al padre
- Riduco **F** in **T** e il 3 migra verso **T**
- Scendo poi in ***** ma non ho nulla da calcolare
- Scendo su **F** e scendo su **digit**. Mi calcolo il valore di **digit** tramite l'analisi lessicale che è 5

- Salgo ad F e siccome F ha visto tutti i figli, uso la regola e ottengo che $F.val = 5$
- Calcolati $T.val=3$ ed $F.val=5$, salgo al padre e uso la produzione $T \rightarrow T_1 * F$ che mi dice che $T.val = T_1.val * F.val$ pertanto $T.val=15$. Se la grammatica è S-attribuita e sto calcolando il valore di un padre sono sicuro che i figli li ho già calcolati
- Salgo poi in E che ha come regola $E.val = T.val$ quindi $E.val=15$
- Salgo poi in E, scendo in + e non ci sono operazioni pertanto scendo fino ad arrivare a `digit.lexval`
- Salgo in F e copio il valore di `digit.lexval` pertanto $F.val=4$
- Salgo in T e applicando la regola $T.val=4$
- Salgo in E e applicando la regola ho che $E.val = E_1.val + T.val$, pertanto $E.val=19$
- Salgo in L e applicando la regola ho che $L.val=E.val$ pertanto $L.val=19$, costruendo la

visita

Ho potuto fare un post order in quanto la grammatica è S-attribuita. La posizione della regola è sempre alla fine. Vedo prima tutti i figli. In questo caso si ha una definizione guidata dalla sintassi ma posso farlo diventare uno schema di traduzione dicendo che le regole devono essere applicate dopo aver visto la parte destra. L'importante è che la regola si lancia dopo che tutti i figli destri che ci servono sono stati visti

10.11 Passaggi per il calcolo degli attributi

In generale, costruisco l'albero di parsing. Fatto questo albero di derivazione posso aggiungere gli attributi per ogni nodo (ovvero ad ogni terminale o non terminale). Vado ad annotare tutti i terminali e non terminali con i rispettivi attributi. Solo dopo faccio una visita dell'albero. Quando incontro un sottoalbero che corrisponde ad una produzione, applico la regola semantica di quella produzione.

10.11.1 Modi per calcolare gli attributi

Posso scegliere due approcci:

- faccio l'albero di derivazione, lo annoto con gli attributi, mi faccio la post order guardando le regole della grammatica e mi calcolo gli attributi

- mentre faccio il parsing, quando faccio la riduzione sparo la regola semantica.

Posso far diventare una definizione basata sulla grammatica in uno schema di traduzione dicendo che le regole devono essere applicate dopo aver visto la parte destra. L'importante è che la regola si lancia dopo che tutti i figli destri che ci servono sono stati visti. Mettendo la regola sempre alla fine non si sbaglia mai

10.11.2 Esempio: Grammatica circuit

La grammatica Circuit risulta essere S-attribuita

10.12 Grammatiche S-attribuite, L-attribuite e grammatiche generali

Una grammatica L-attribuita una che può avere sia attributi sintetizzati che ereditati, ovvero calcolo l'attributo in funzione del padre o dei fratelli sinistri. Mentre le S-attribuite sono legate ad un ordine topologico bottom-up, le L-attribuite sono legate ad una top-down depth first da sinistra a destra. Se un simbolo dipende dal fratello deve dipendere dal fratello sinistro. Se prendo una grammatica ad attributi dove ci sono simboli che dipendono da attributi di fratelli destri, è comunque una grammatica ad attributi ma non è L-attribuita e non è legata ad un particolare ordine facile, non potendo farci né bottom up né depth first da sinistra a destra

Nei casi generici, non possiamo usare un parser. Il vantaggio è che posso fare sia parsing che traduzione mentre faccio riconoscimento, in parallelo. Nel caso però in cui la grammatica non sia né S-attribuita che L-attribuita, non potrò fare traduzione e parsing insieme ma dovrò fare i seguenti passaggi:

- un parser mi crea l'albero di derivazione
- lo annoto con gli attributi
- mi trovo l'ordine topologico

Se ci sono cicli, la grammatica ad attributi come è stata congeniata non va bene, non posso trovare un ordine topologico. Le grammatiche ad attributi con ciclo o dipendenza circolare sono grammatiche non gestibili. In generale, quando facciamo il parsing, dato che stiamo scollegando l'idea di parsing dalla grammatica, possiamo usare una tecnica qualsiasi di parsing, che sia bottom up o top down. Quando costruisco un albero prendo un sottoalbero

e mi vedo tutte le sue regole. **Le regole di una grammatica ad attributi non possono andare fuori dell'albero, come se fossero variabili locali. All'interno di una regola possiamo fare riferimento solo agli attributi della produzione, no fare riferimento ad altro.** I calcoli quindi saranno sempre locali

10.12.1 Esempio: Grammatica con cicli

Data una grammatica con **produzione $A \rightarrow B$** e con regole **$A.s = B.i$** e **$B.i = A.s + 1$** , non va bene. In quanto la prima regola mi dice che **l'attributo del padre dipende dal figlio**, la seconda dice che **l'attributo del figlio dipende dal padre**, creando un ciclo. Quando arrivo sul nodo A, per usare la prima regola devo vedere prima il figlio. Per calcolare però la seconda per vedere il padre, pertanto facendo il grafo delle dipendenze vedrei un ciclo

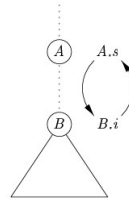


Figura 10.2: Esempio: Grammatica con cicli

Le grammatiche ad attributi hanno vita propria indipendente dai compilatori e dai parser. **Di tutte le grammatiche ad attributi ci interessano solo le L-attribuite e le S-attribuite**

10.13 Navigazione dell'albero e calcolo degli attributi

Per il calcolo degli attributi **parto dall'elemento più a fondo e più a sinistra** (depth first da sinistra a destra). Quando faccio la visita, **ogni volta che incontro un nodo calcolo i suoi ereditati ma non i sintetizzati. I sintetizzati li calcolo quando lascio il nodo.** Un nodo può essere **visto in entrata per la prima volta ed in uscita per l'ultima volta.** Quando l'incontro per la prima volta calcolo gli ereditati quando lo lascio per l'ultima volta calcolo i sintetizzati

10.14 Indipendenza di grammatiche e parser

Una grammatica che sia non LL(1) o che presenti ricorsione sinistra può essere L-attribuita. L'analisi se una grammatica sia L-attribuita viene fatta esclusivamente sugli attributi. **Le grammatiche ad attributi sono infatti indipendenti dai parser.** Una grammatica

può essere S-attribuita e avere una regola che calcola il padre tramite un altro attributo del padre stesso.

10.15 Riassunto: Grammatiche ad attributi

Se i terminali possono avere degli attributi, li possono avere anche i non terminali, solo che gli attributi dei terminali sono calcolati dall'analisi lessicale, mentre gli attributi non terminali devono essere calcolati dalla grammatica stessa. Dobbiamo quindi mettere delle regole per calcolarli. La produzione di una grammatica è fatta da una parte sinistra e da una parte destra. Quando calcolo un attributo della parte sinistra ho un attributo sintetizzato, e di solito dipende o dall'attributo stesso o da una parte dei figli. Quando si ha un attributo ereditato andremo a calcolare un attributo nella parte sinistra. Esso può dipendere o dal padre dai fratelli sinistri.

Se una grammatica ha solo attributi sintetizzati allora la grammatica è S-attribuita, se ha sia attributi sintetizzati che ereditati è L-attribuita. Tra tutte le possibili grammatiche, per essere calcolata una grammatica non può avere cicli, non devono esserci cicli: quando eseguo un albero di derivazione, devo garantire che ogni volta che calcolo un attributo in funzione di altri attributi, gli altri devono essere già calcolati.

Per calcolare il valore di una grammatica la si prende, si realizza l'albero di derivazione, lo si annota con gli attributi, si fa il grafo delle dipendenze ed infine si cerca poi l'ordine topologico. Per l'S-attribuita l'ordine topologico non va calcolato in quanto basta una visita bottom up o post order. Se la grammatica è invece L-attribuita, basta un'analisi depth first da sinistra a destra. Anche in questo caso l'ordine topologico è garantito ed è insito. Vantaggio è che posso lanciare un parser bottom up su di una grammatica ad attributi e fare in modo che mentre fa il riconoscimento fa anche la traduzione. Piuttosto che separare le fasi, faccio tutto insieme. In tal caso oltre ad avere uno stack avremo uno stack parallelo dove conserviamo gli attributi di terminali e non terminali. Esso prende il nome di stack semantico. Tutto ciò lo facciamo in quanto vogliamo un output trasformato nella sentenza. Vogliamo un albero sintattico che rappresenti l'input. Le regole che avrò devono costruire una gerarchia di nodi. Per la generazione del codice intermedio si possono utilizzare le grammatiche ad attributi come forma algoritmica per produrre codice intermedio. Tutti i compilatori moderni non fanno il salto ma realizzano prima l'AST

10.16 Regole e produzioni

Va notato che **quando definiamo una regola per una produzione essa non può cambiare**. Bisogna quindi **definire una regola che vada bene in tutti i casi**. Al contempo, **non si può assegnare allo stesso attributo due o più modi differenti di calcolare l'attributo stesso**.

10.17 Riassunto: Da codice sorgente ad AST

Siamo partiti dal file sorgente, riuscendo a fare l'analisi lessicale. Siamo ora nella **fase sintattica in cui abbiamo fatto la prima parte che riguardava il parsing vero e proprio**. Dobbiamo ora fare la **generazione degli alberi sintattici**, dovendoli produrre. Dopo le grammatiche abbiamo visto le grammatiche ad attributi, utili per fare gli alberi sintattici.

10.18 Riassunto: Grammatiche ad attributi

Per le grammatiche ad attributi abbiamo detto che per diventare ad attributi ha bisogno di:

- attributi
- regole per il calcolo degli attributi

A seconda di come queste regole vengono eseguite, ovvero a seconda di come vengono calcolati gli attributi, **parliamo di attributi sintetizzati ed attributi ereditati**. Tra gli **ereditati ci interessano quelli ereditati a sinistra**. Per i sintetizzati, data una produzione $A \rightarrow X_1 X_2 X_3$ gli **attributi sintetizzati sono tutti quelli calcolati per A**. **Non solo il padre può avere attributi sintetizzati. Tutti i simboli possono avere attributi sintetizzati solo che questi vengono calcolati quando si trovano che il simbolo della produzione a sinistra equivale a X_1 , X_2 o X_3** . Tali attributi si calcolano quando il simbolo si trova a sinistra. Gli attributi **ereditati vengono calcolati quando si trovano a destra**. **Se dipendono solo dagli attributi che si trovano a sinistra del non terminale o del padre sono ereditati a sinistra**. Gli attributi sintetizzati dipendono dai figli, mentre gli ereditati dipendono da fratelli (possibilmente a sinistra) o dal padre. Se parliamo di attributi ereditati a sinistra allora devono dipendere da attributi a sinistra o dal padre

10.19 Riassunto: Tipi di grammatiche

Le **grammatiche incidono sulla visita dell'albero di derivazione**. Possiamo dividere le grammatiche ad attributi in varie classi in funzione degli attributi:

- La più semplice è la **grammatica S-attribuita che ha solo attributi sintetizzati**. Gli attributi si calcolano su un input **visitando l'albero di derivazione sull'input in modo bottom up**. Il calcolo si può fare anche durante (e non dopo) il parsing bottom up
- Ci sono poi le **L-attribuite che hanno attributi sintetizzati e attributi ereditati da sinistra**. Tali grammatiche **dovrebbero avere almeno un attributo sintetizzato altrimenti non si riuscirebbe a calcolare l'attributo del non terminale iniziale**. Gli attributi si calcolano su un input **visitando l'albero di derivazione in modo depth first da sinistra a destra**. Il calcolo si può fare durante il parsing top down
- Ci sono le classi **non S-attribuite, non L-attribuita senza cicli di dipendenze**
- Ci sono le **non calcolabili con dipendenze cicliche**

Per le grammatiche e definizione di regole **abbiamo definizioni guidate dalla sintassi e schemi di traduzione**

10.20 Regola generale di calcolo di attributi

Il vantaggio di **lavorare con le classi S-attribuite ed L-attribuite è che posso lavorare durante il parsing, non dovendo suddividere in step la creazione di albero di parsing, annotarlo con gli attributi, mettere le dipendenze in funzione delle regole e quindi calcolare gli attributi**. Il calcolo generale per il calcolo di attributi in una grammatica ad attributi su di un dato input è:

- **costruire l'albero di derivazione**. Abbiamo a disposizione una grammatica ad attributi ed un input, pertanto costruiamo l'albero di derivazione
- **annotare terminali e non terminali nell'albero con i propri attributi**.
- **Usando le regole di ciascuna produzione della grammatica, creiamo il grafo delle dipendenze**
- **Se non ci sono cicli, andiamo a seguire l'ordine topologico per il calcolo degli attributi**

Nel caso di grammatica **S-attributo** o **L-attributo** il terzo e quarto passo sono esagerati. In generale, **dal padre non posso utilizzare gli attributi sintetizzati ma posso utilizzare solo gli attributi ereditati dal padre**, ovvero gli attributi che il padre ha ereditato, ma **un figlio non può prendere gli attributi sintetizzati del padre**. Quando incontro un attributo ereditato lo calcolo per la prima volta mentre l'attributo sintetizzato lo calcolo quando esco da quel nodo. Se vado a fare il grafo delle dipendenze e ci sono cicli non posso calcolare gli attributi

10.20.1 Esempio: Definire se la grammatica sia L-attribuita

Se io ho una grammatica:

- $S \rightarrow A B \{ B.x = A.y // A.y = 1 // S.z = B.x \}$
- $A \rightarrow a$
- $B \rightarrow b$

Tale grammatica è L-attribuita in quanto **andando a guardare le regole** (dobbiamo guardare solo quelle e non altro) **vedo che andiamo sì a sintetizzare l'attributo del padre S ma anche a ereditare l'attributo del figlio B tramite il fratello a sinistra**, pertanto la grammatica è L-attribuita

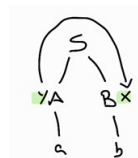


Figura 10.3: Esempio: Definire se la grammatica sia L-attribuita

Sull'albero ci sono delle dipendenze che vanno dal padre ai figli, dai figli ai padri o dai fratelli sinistri ai destri. Siccome la grammatica è L-attribuita potrei pensare di non fare i quattro passi ma:

- Partire da S, non posso calcolare quindi scendo in depth first da sinistra a destra
- vado in $A.y = 1$ che è un attributo neutro in quanto non è né sintetizzato né ereditato
- Torno ad S ma non ha ancora visto tutti i figli, quindi scendo in B
- B ha un attributo ereditato. L'attributo ereditato dipende o dagli attributi sinistri o dal padre. Avendo già visto entrambi posso calcolarlo. Dal padre non posso utilizzare

gli attributi sintetizzati ma posso utilizzare solo gli attributi ereditati dal padre, ovvero gli attributi che il padre ha ereditato, ma un figlio non può prendere gli attributi sintetizzati del padre

- Salgo su e vedo S

Quando incontro un attributo ereditato lo calcolo per la prima volta mentre l'attributo sintetizzato lo calcolo quando esco da quel nodo. Se vado a fare il grafo delle dipendenze e ci sono cicli non posso calcolare gli attributi

10.21 Grammatica postfissa (S-attribuita con schemi di traduzione)

Per il progetto, utilizzeremo: javacup -> parsing bottom up (LALR(1)) -> costruzione bottom up dell'albero di derivazione su un input (l'albero di derivazione è specifico di un input in quanto senza input non si ha l'albero di derivazione) -> grammatica S-attribuite utilizzando schemi di traduzione

La grammatica che viene utilizzata da javacup è la S-attribuita. Quando parliamo della S-attribuita e schemi di traduzione parliamo di una grammatica postfissa. Questo perchè abbiamo detto che lo schema di traduzione bisogna dire dove va messa la regola. Con la grammatica S-attribuita, siccome la regola calcola l'attributo del padre dopo aver visto i figli, la regola deve essere messa solo dopo che abbiamo visto tutti i figli, pertanto la regola viene messa alla fine della produzione. La grammatica postfissa è una grammatica S-attribuita e si usa uno schema di traduzione. In Java Cup, possiamo però aggiungere una regola all'interno della produzione: $S \rightarrow A \{ rule_1 \} B$. ciò però viene trasformata in: $S \rightarrow A X B$ ed $X \rightarrow \epsilon \{ rule_1 \}$. Non si cambia il linguaggio in quanto si aggiunge una produzione X che va in ϵ e che contiene la regola, traducendomi la grammatica da non postfissa a postfissa. Nei parsing bottom up abbiamo che la ϵ facilmente produce conflitti, pertanto conviene evitare la definizione di una regola al centro della produzione in quanto si va ad inserire nella grammatica produzioni che vanno in ϵ .

In generale, le regole per gli attributi ereditati sono sempre inseriti nella produzione e prima del simbolo a cui fanno riferimento. Le regole per gli attributi sintetizzati vengono messi alla fine mentre quelle per gli attributi ereditati vengono messi subito prima del simbolo che ne ha bisogno

10.22 Programmazione ad albero

Per lavorare con le grammatiche ad attributi, **dobbiamo usare un paradigma di programmazione, chiamato programmazione ad albero**. I passi della programmazione ad albero sono:

- **Creare una grammatica**
- **mi creo un input significativo che usi almeno tutte le produzioni**
- **Mi creo l'albero di derivazione**
- **annotare l'albero con attributi utili per i simboli e definire regole locali che rispettino il tipo di visita richiesto dal tipo di grammatica che si vuole creare. Se vogliamo fare una grammatica S-attribuita, dobbiamo effettuare una visita bottom up. Le regole devono essere locali in quanto posso usare in una regola solo simboli che si trovano in quella produzione. Se postfissa (S-attribuita con schemi di traduzione) la visita è bottom up se invece è L-attribuita bisogna fare una depth first da sinistra a destra. In generale, cerco di avere una grammatica postfissa applicando solo le regole che creano attributi sintetizzati. Se ci riesco mi fermo, se non riesco o cambio la grammatica o aggiungo attributi ereditati.**

In generale, ogni non terminale fa parte di due produzioni: una in cui è figlio ed una in cui è padre. L'attributo sintetizzato si ha sia quando il non terminale è padre che quando è figlio ma viene calcolato solo quando è padre. Le regole devono essere locali. Dovendo fare una grammatica S-attribuita bisogna pensare ad una visita bottom up. L'aggiunta dei tipi nella tabella dei simboli è una fase di analisi semantica, pertanto nel progetto la grammatica la usiamo per creare solo l'albero sintattico. La grammatica tramite postfissa dovrà produrre un AST. L'albero di derivazione è pur sempre un albero che contiene informazioni, ma contiene al suo interno anche non terminali della grammatica. Se si fa una grammatica lunga, l'albero diventerebbe enorme. Al contrario, l'AST è una rappresentazione sintetica e pulita (indipendente dalla grammatica), contenente solo la semantica, ovvero il significato dell'istruzione.

10.22.1 Esempio: applicazione della programmazione ad albero

Data una grammatica:

- $S \rightarrow S0$

- $S \rightarrow S1$
- $S \rightarrow 0$
- $S \rightarrow 1$

Dobbiamo scrivere una **grammatica postfissa che conti quanti 1 ci sono**. Seguendo la programmazione ad albero:

- Ho già la grammatica
- Scelgo l'input 0101
- mi creo l'albero
- **voglio arrivare ad un count al non terminale iniziale. Se dò l'attributo count ad S tutte le S lo avranno, potendo aggiungere tale attributo a tutte.** Dovendo lavorare localmente ad ogni produzione e **dovendo partire dal basso, la prima produzione dell'albero che incontro è $S \rightarrow 0$** . Il concetto è che **mi porto avanti il conteggio sia che vedo 0 o 1 ma incremento solo quando vedo 1.**

Nello specifico la grammatica diventa:

- $S \rightarrow 0 \{S.count = 0\}$
- $S \rightarrow 1 \{S.count = 1\}$
- $S \rightarrow S_1 1 \{S.count = S_1.count + 1\}$
- $S \rightarrow S_1 0 \{S.count = S_1.count\}$

Essendo attributi sintetizzati, **in ogni regola devo calcolare l'attributo del padre partendo dal figlio. L'ultima regola, ovvero $\{S.count = S_1.count\}$, seppur non aggiunga nulla alla somma, devo comunque portarla come regola in quanto mi propaga il valore.** L'aggiunta di un indice alla S, in generale, è per distinguere le varie S, è puramente letterale come cosa

10.23 Esempio: dal codice sorgente all'AST

Fino ad ora abbiamo visto una grammatica ad attributi che interpretava l'espressione nel suo valore, o una grammatica che interpretava la stringa binaria nel suo valore decimale. **Con la grammatica ad attributi posso però fare qualsiasi cosa. Dato un input che può essere rappresentato tramite una grammatica, con tale input posso farci quello che voglio.**

Dato il codice `"int i, j;"` vorrei che si aggiungesse alla tabella dei simboli il tipo alle variabili `i` e `j`. Devo fare una grammatica ad attributi che mi prende l'input, verifica che sia corretta la frase e la traduzione in questo caso è di riempire una tabella. Per fare ciò posso usare `addType(id.entry, type)` è una funzione che prende in input il tipo, ovvero `type` ed il riferimento al simbolo, quindi in questo caso sarebbe tipo `id.entry` dove `entry` è il riferimento alla tabella delle stringhe. Se `i` è stato già inserito in una tabella delle stringhe con indice `xx`, il valore di `entry` per `i` sarà proprio `xx`.

10.23.1 Esempio: Realizzazione dell'albero di derivazione

Per costruire la grammatica devo capire come sia fatto il linguaggio, che in questo caso è un tipo ed una sequenza di tipi. La prima cosa fare è imporre una struttura gerarchica (definire una grammatica). Devo poi mettere le regole che, quando vengono effettuate le riduzioni, mi permettono di lanciare la funzione `addType()` e costruire pertanto la tabella. È importante che la funzione la lanci quando ho sottomano sia il riferimento ad ID sia il tipo. È possibile che abbia solo un'informazione e non l'altra ma in questa condizione non devo fare niente. Devo lanciare la funzione quando tutto è chiaro. L'albero che la grammatica mi sottoporrebbe (da notare che manca un figlio partire da `D` ed arrivare in `;`) è:

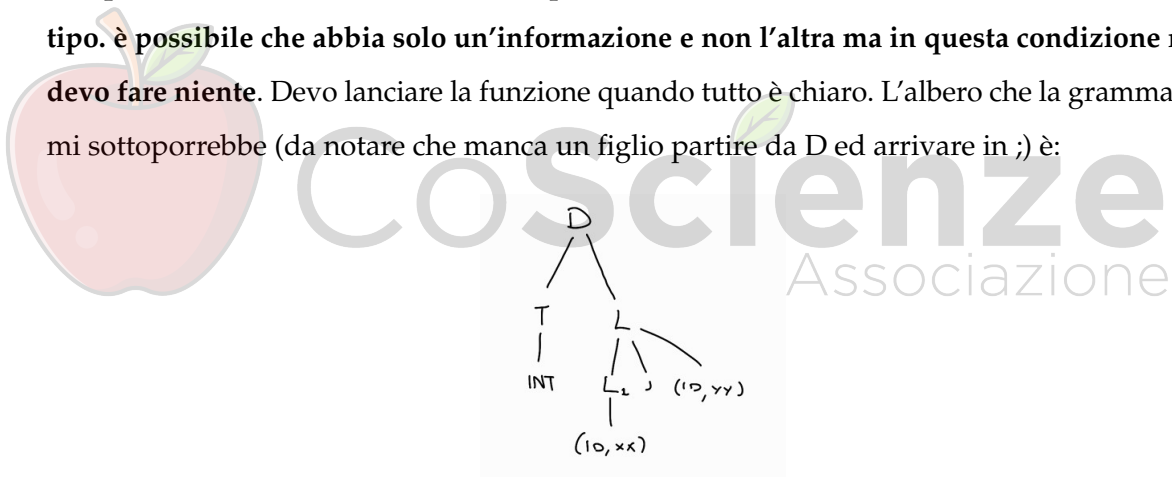


Figura 10.4: Esempio: Realizzazione dell'albero di derivazione

10.23.2 Esempio: Definizione della grammatica

Avendo il codice sorgente che è una dichiarazione di variabile, posso dire che la radice del mio albero sarà `D`. Una dichiarazione sarà fatta da un tipo ed una lista. Tale albero corrisponde già ad una produzione grammaticale, nel nostro caso `"D -> T L;"`. Successivamente, `T` può essere o il token `INT` o il token `FLOAT`, pertanto alla grammatica si potrebbe aggiungere la produzione `T -> INT` e `T -> FLOAT`. Lato codice avremo poi `i`, pertanto in fase di analisi lessicale sarà già riconosciuto come `(ID, xx)` e così viene passato al parser, dove `xx` è il riferimento alla tabella delle stringhe. Il parser usa la coppia `(ID, xx)` per fare il

parsing e per riconoscere l'attributo andando ad applicare le regole. Nella grammatica, L deve rappresentare una lista, pertanto le produzioni associate ad L potrebbero essere: $L \rightarrow L, ID$ ed $L \rightarrow ID$. La grammatica è quindi:

- $D \rightarrow T L$;
- $T \rightarrow INT$
- $T \rightarrow FLOAT$
- $L \rightarrow L, ID$
- $L \rightarrow ID$

10.23.3 Esempio: Definizione delle regole alle produzioni

Devo ragionare sull'albero per fare un algoritmo di visita che spari delle regole e chiami la funzione 2 volte `addType()`, una volta per ciascun ID. Provando a fare un bottom up, parto da sinistra e scende fino ad INT ho:

- **Riduzione $T \rightarrow INT$:** vediamo INT e sappiamo che possiamo ridurlo. Essendo che l'informazione INT mi serve, posso dare a T un attributo `type` la cui regola associata è `{ T.type=int }`
- **Riduzione $L \rightarrow ID$:** Il prossimo elemento che visitiamo è (ID, xx). Esso non può lanciare l'`addType` in quanto conosce l'attributo ma non il tipo, che è salvato nell'attributo `T.type`. Non posso però usare un attributo di un non terminale che si trova chissà dove nell'albero, potendo usare al più un attributo di L_1 , ovvero il padre di (ID, xx).

Non posso creare una grammatica postfissa con questa grammatica, in quanto non potrò mai chiamare l'`addType()`. Alternativa è quella di cambiare la grammatica o provare a vedere se una grammatica L-attribuita riesca a funzionare. Per far arrivare il tipo INT ad L_1 padre di (ID, xx), posso passare il tipo da T ad L (padre di L_1), ovvero passo l'informazione da fratello sinistro a fratello. Da L, passo poi l'informazione al figlio quindi da L ad L_1 (padre di (ID, xx)). A livello di schema di traduzione, il passaggio di T ad L è necessario che avvenga prima che visito il sottoalbero L. L'attributo di L che conterrà il tipo è "i". Essendo che ad una L abbiamo definito l'attributo i, lo stesso attributo deve essere definito per tutte le L. Quando L_1 ha visto sia `type` che `id`, può lanciare l'`addType()`. Posso poi effettuare la riduzione $L \rightarrow L_1, ID$ quindi ho sia `type` (ereditato) che (ID, yy) (sintetizzato) potendo

lanciare `addType()`. Fare il passaggio del tipo da T a D e da D ad L non andava bene in quanto l'attributo sarebbe sintetizzato per D pertanto verrebbe calcolato alla fine. Un figlio potrebbe, infatti, prendere solo un attributo ereditato dal padre e non anche sintetizzato. In generale, si avrebbe un passaggio in più.

10.23.4 Esempio: Grammatica con definizioni guidate dalla sintassi

A livello di grammatica, applicando la definizione guidata dalla sintassi avremo:

- 0) $D \rightarrow T L; \mid \mid L.i = T.type \mid \mid$
- 1) $T \rightarrow INT \mid \mid T.type = INT \mid \mid$
- 2) $T \rightarrow FLOAT \mid \mid T.type = FLOAT \mid \mid$
- 3) $L \rightarrow L_1, ID \mid \mid L_1.i = L.i \mid \mid addType(ID.entry, L.i)$
- 4) $L \rightarrow ID \mid \mid addType(ID.entry, L.i)$

Nella definizione guidata dalla sintassi per 0 sono costretto a fare il passaggio tra fratelli in quanto è l'unica produzione in cui T ed L sono insieme. Scrivendo una definizione guidata dalla sintassi come in 3, quello che mi garantisce che l'attributo i esiste è il fatto che ho una visita depth first da sinistra a destra, pertanto ho un ordine topologico. Nella mia mente, quando scrivo le regole so che sto seguendo la depth first da sinistra a destra. Avendo una visita differente scriverei probabilmente altre regole. In questo caso, l'ordine è quello della depth first da sinistra a destra. Se metto tutte le dipendenze negli attributi scopro che una depth first da sinistra a destra è un ordine topologico.

10.23.5 Esempio: Grammatica con schema di traduzione

Analizzando le definizioni guidate dalla sintassi:

- Affinchè tutto ciò funzioni, è importante che L abbia il tipo prima che si entri a visitare L, pertanto devo avere come regola " $D \rightarrow T \{L.i = T.type\} L ;$ ". A questo punto ho già visto T, pertanto ho già l'attributo T.type. Prima di entrare in L, metto $\{L.i = T.type\}$. L'attributo i è ereditato in quanto viene dato dal fratello. Facendo uno schema di traduzione, gli attributi ereditati sono sempre prima del simbolo a cui fanno riferimento.

- Per $T \rightarrow INT$ devo portare l'informazione del tipo dal figlio al padre, pertanto l'attributo è sintetizzato. Per andare sicuri la regola la mettiamo per ultima. Stesso ragionamento lo applichiamo per $T \rightarrow FLOAT$
- Prima di visitare in L_1 devo conoscere il tipo di $L_1.i$. Essendo $L_1.i$ ereditato, devo mettere la regola prima di L_1 . Le regole di `addType` possono essere visti come attributi sintetizzati, in quanto seppur non calcolano il padre usano comunque i figli. Per tale motivo `{ addType(ID.entry, L.i) }` va messo alla fine

La grammatica con schema di traduzione è:

- 0) $D \rightarrow T \{ L.i = T.type \} L ;$
- 1) $T \rightarrow INT \{ T.type = INT \}$
- 2) $T \rightarrow FLOAT \{ T.type = FLOAT \}$
- 3) $L \rightarrow \{ L_1.i = L.i \} L_1 ID \{ addType(ID.entry, L.i) \}$
- 4) $L \rightarrow ID \{ addType(ID.entry, L.i) \}$

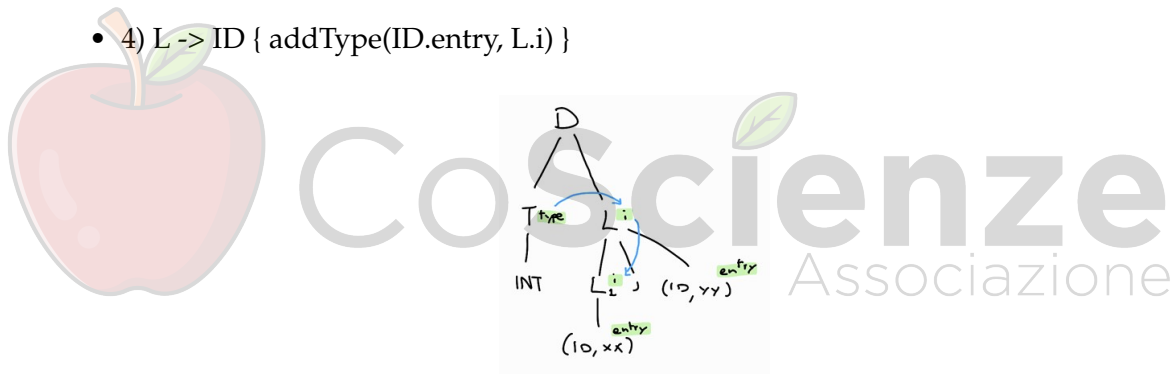


Figura 10.5: Esempio: Grammatica con schema di traduzione

10.23.6 Esempio: Limiti della grammatica

Tale grammatica ad attributi, essendo non postfissa, non può essere realizzata su `javacup`. Esso può fare solo bottom up ma in tal caso la grammatica necessita che gli attributi scendano. Tale grammatica potrebbe però essere realizzata a discesa ricorsiva. A livello di codice per il momento le chiamate ai non terminali vengono fatte senza attributi, mentre si potrebbero aggiungere i parametri alle funzioni in quanto essi mi rappresenterebbero gli attributi di quel non terminale.

La grammatica resta la stessa ma si mettono altre regole che creano nodi o foglie, tramite una libreria di gestione degli alberi. La visita sull'albero verrà fatta tramite `pattern visitor`, avendo il problema di portare su e giù gli attributi.

10.23.7 Esempio: Forzatura della grammatica S-attribuita

C'è un altro approccio che si potrebbe eseguire per l'esempio. Il vantaggio di come abbiamo costruito la grammatica è che l'addType() è distribuito. Tecnica alternativa è quella di portare tutto al nodo D. Non pretendo che L faccia l'operazione di addType ma far fare tutto alla radice dell'albero. Alla radice ci si troverà il tipo e la lista di ID. Dovrà scorrere la lista e fare l'addType() di tutti. L'idea sarà quella di avere degli attributi per L che salvano il riferimento al singolo ID o alla lista di ID. A livello di albero diventa:

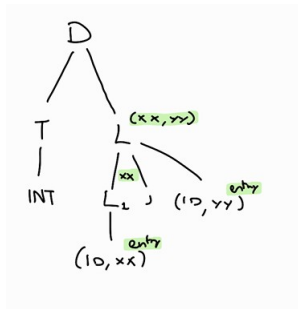


Figura 10.6: Esempio: Forzatura della grammatica S-attribuita

10.24 Costruzione del Syntax Tree

Per la costruzione di un albero partendo dalle regole di una grammatica abbiamo almeno due tipologie di classi:

- **Node("OP", left, right):** Essa prende il codice operativo (ovvero addOp, declOp) e poi i figli sinistro e destro. In tal caso l'albero sarà sempre binario, ma non deve essere sempre così. Cambiando i parametri si possono aggiungere più figli
- **Leaf(TOKEN, TOKEN_ATT).** Esso prende il token ed il suo attributo opzionale. Tutte le foglie dell'albero sintattico saranno sempre e solo token.

Node() essendo un costruttore mi costruisce un nodo e mi restituisce il riferimento al nodo, stessa cosa per Leaf(). Devo poi scrivere una grammatica le cui regole mi vanno a creare nodi e foglie. Ogni non terminale avrà come attributo il puntatore all'albero sintattico della sottofrase che lui vede. Metterò le regole alla grammatica in modo tale che ogni non terminale avrà un riferimento all'albero sintattico della sottofrase che il non terminale vede. Ciò andrà bene in quanto la radice avrà l'intero albero come albero ed avrà il riferimento all'intero albero

10.24.1 Esempio: Costruzione di AST partendo da una grammatica aritmetica

Data la grammatica:

- $E \rightarrow E + T$
- $E \rightarrow T$
- $T \rightarrow T * F$
- $T \rightarrow F$
- $F \rightarrow (E)$
- $F \rightarrow ID$
- $F \rightarrow NUM$

L'albero andrò a costruirlo dal basso verso l'alto. Dovendo fare $5 + 4 * 3$, l'AST è mostrato a destra mentre l'albero di derivazione a sinistra

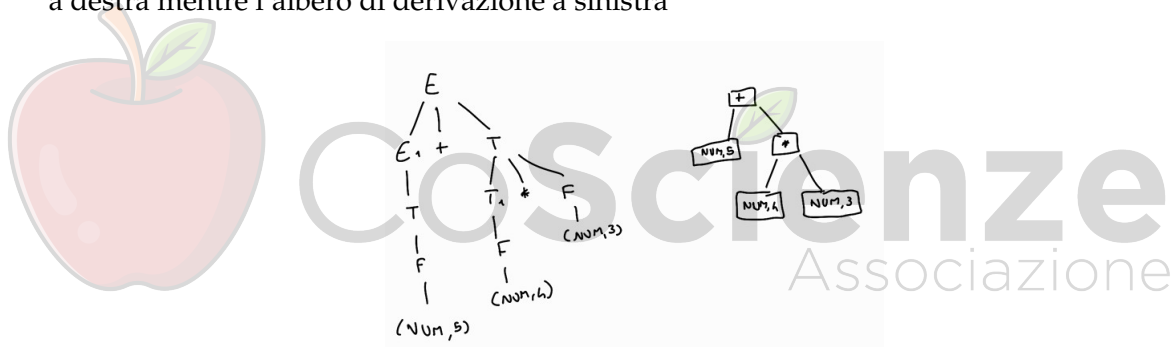


Figura 10.7: Esempio: Costruzione di AST partendo da una grammatica aritmetica

L'idea è che ogni **non terminale** abbia un **attributo** che si chiama **node**. Tale attributo conterrà il riferimento all'albero sintattico per ciò che lui vede. La F padre di (NUM,5) conterrà il puntatore a 5, in quanto è l'unica cosa che vede. F avrà come attributo il puntatore all'albero sintattico della sottofrase che lui vede. Tramite le regole, faremo ad esempio che in T vada a finire il riferimento al sottoalbero sintattico che rappresenta il suo input ovvero $4 * 3$. Ciò andrà bene in quanto E vedrà $5 + 4 * 3$ ed avrà l'intero albero. Le funzioni hanno dei parametri e **dobbiamo garantire che i parametri esistano**. Decidiamo di fare una grammatica S-attribuita, pertanto dò per scontato che la visita sia **bottom up**.

Le regole definite tramite visita bottom up sono:

- **Riduzione $F \rightarrow 5$:** Parto da (NUM, 5) devo costruire la foglia dell'albero sintattico. Lancio la funzione **Leaf(NUM, NUM.val)**, dove $NUM.val = 5$. Così solo non mi basta

pertanto devo salvarmi il riferimento. Essendo che tutti i non terminali hanno un attributo `node` che punta al sottoalbero avremo `F.node = new Leaf(NUM, NUM.val)`.

Il riferimento sarà `xx`

- **Riduzione $T \rightarrow F$:** Dato che T in questo caso vede solo 5, devo propagare il riferimento da F a T , avendo la regola `T.node = F.node`
- **Riduzione $E \rightarrow T$:** Devo propagare il valore pertanto `E1.node = T.node`
- **Riduzione $F \rightarrow 4$:** Qui mi trovo perchè la riduzione $E \rightarrow E + T$ non posso farla fin quando non ho visto tutto il sottoalbero di T , di cui $F \rightarrow 4$ fa parte. Ho già una regola nella grammatica pertanto devo verificare se la regola vada bene anche per questa produzione. Va bene quindi salgo. Mi creerà `new Leaf()` con riferimento `yy`. Il riferimento mi andrà a finire come valore di `F.node`, risalendo poi in `T.node`
- **Riduzione $F \rightarrow 3$:** Qui applico lo stesso ragionamento di prima. Anche in questo caso la regola va bene per la produzione. Mi crea una nuova `new Leaf()` che avrà riferimento `zz` che andrà a finire in `F.node`. Così facendo ora ho visto tutto il necessario per la produzione $T \rightarrow T_1 * F$
- **Riduzione $T \rightarrow T_1 * F$:** siccome di T conosco l'operatore ed il figlio sinistro e destro, posso lanciare la funzione `Node("...", T1.node, F.node)` e mi salvo il riferimento in `T.node`, ovvero `tt`
- **Riduzione $E \rightarrow E_1 + T$:** posso finalmente applicare la riduzione in quanto ho visto tutto il sottoalbero sia di E_1 che di T , chiamando la funzione `Node()` e salvandomi il riferimento `vv`

I riferimenti sopra citati sono fittizi e servono solo ad avere una visione del funzionamento delle funzioni `Node()` e `Leaf()` e come si propagano i riferimenti tra gli attributi `node`. Il riferimento `vv`, infine, è il riferimento all'intero albero. Dopo che avrò fatto il parsing, posso andare a prendere l'attributo di E ed avere un riferimento all'intero albero sintattico. Tale riferimento alla radice dell'albero lo passo alla funzione `seman()` che fa l'analisi semantica. La funzione `seman()` prende quindi un puntatore all'albero come parametro. Su tale riferimento scatta poi il pattern visitor. Faremo una prima visita per la stampa dell'albero, per vedere se l'albero sia come lo vogliamo. La seconda visita avrebbe l'analisi semantica. La terza potrebbe essere.

La grammatica modificata sarà:

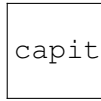

 capitoli/images_gram_attr/gr_att_alb5.jpg

Figura 10.8: Esempio: Costruzione di AST partendo da una grammatica aritmetica

- $E \rightarrow E_1 + T \{ E.\text{node} = \text{new Node}("+", E_1.\text{node}, T.\text{node}) \}$
- $E \rightarrow T \{ E.\text{node} = T.\text{node} \}$
- $T \rightarrow T_1 * F \{ T.\text{node} = \text{new Node}("*", T_1.\text{node}, F.\text{node}) \}$
- $T \rightarrow F \{ T.\text{node} = F.\text{node} \}$
- $F \rightarrow (E) \{ F.\text{node} = E.\text{node} \}$
- $F \rightarrow \text{ID} \{ F.\text{node} = \text{new Leaf}(\text{ID}, \text{ID.val}) \}$
- $F \rightarrow \text{NUM} \{ F.\text{node} = \text{new Leaf}(\text{NUM}, \text{NUM.val}) \}$

L'albero di derivazione è un albero che non è sintetico e che dipende dalla grammatica, rappresentando inoltre un trace di esecuzione. È stato quindi tradotto in qualcosa che non dipenda dalla grammatica. I simboli **()** servono per spezzare la priorità. Esse, se usate per **(5+4)*3** mi costringerebbero a scrivere un albero in cui ***** è la radice e non **+**, in quanto andrei a fare ***** come ultima operazione, in quanto andrei prima a fare la somma tra 5 e 4 e poi andrei a moltiplicare il risultato per 3.



 capitoli/images_gram_attr/gr_att_alb6.jpg

Figura 10.9: Esempio: Costruzione di AST partendo da una grammatica aritmetica

L'ultima operazione verrà messa come radice. Una volta che ho costruito l'albero, la tonda non mi serve più, in quanto l'informazione che mi portava la tonda è stata usata. Non ci saranno altri usi per la **()**. Lo stesso vale per **;** o **.,**. I commenti invece, terminano la loro utilità durante l'analisi lessicale. Simboli di punteggiatura una volta che costruisco l'albero non servono più, non portano altre informazioni.

10.25 Funzionamento dello stack semantico

La grammatica posso usarla in javacup che effettua un parser bottom up. ogni volta che javacup vede un token fa lo shift sullo stack e quando trova un handle sullo stack lo riduce. Per gestire gli attributi, oltre allo stack userà uno stack semantico:

LIVELLO	STACK	STACK SEMANTICO
6	E	vv
5	T	tt
4	NUM F	3 zz
3	*	
2	NUM F T	4 yy
1	+	
0	NUM F T E	5 xx

La prima cosa che fa è chiedere una coppia al Lexer, mettendo il token nello stack e l'attributo nello stack semantico. Nel primo caso parto da NUM e valore 5. Applico poi la riduzione $F \rightarrow \text{NUM}$ e salvo il valore aggiornando l'attributo di F, ovvero applicando la funzione Leaf e salvando il riferimento xx. In rosso mettiamo i token che andiamo a ridurre, arrivando per ogni riga all'ultimo token (in nero) che non viene ridotto ma che viene usato in altre produzioni:

- I livelli 2,3 e 4 vengono usati per la riduzione $T \rightarrow T_1 * F$ che viene salvata a livello 5 con riferimento tt. Una volta che applichiamo la riduzione andiamo ad eliminare i livelli 2, 3 e 4 ed andiamo a sostituirli con 5
- I livelli 0 1 e 5 vengono usati per la riduzione $E \rightarrow E_1 + T$ che viene salvata a livello 6 con riferimento vv, contenente il riferimento all'intero albero.

Useremo l'albero per vedere chi sia un handle. Quando applichiamo una riduzione come $T \rightarrow T_1 * F$, sapremo quale sia l'handle (F in questo caso) e sapremo sempre dove trovare i terminali come F e T_1 in quanto F sarà al top dello stack mentre T_1 sarà al livello top-2, questo perchè si trova due simboli prima di F nella produzione. La funzione di javacup quando legge T.node sa quale sia l'elemento sullo stack e sa dove andare a recuperare gli altri valori. Javacup prende gli attributi dallo stack e sa dove siano in quanto parte dall'ultimo simbolo sullo stack, ovvero il top, e da esso si va a calcolare la distanza

10.26 Riferimento all'albero sintattico

Dopo che avrò fatto il parsing, posso andare a prendere l'attributo del non terminale iniziale, nonchè radice dell'albero, ed avere un riferimento all'intero albero sintattico. Tale riferimento alla radice dell'albero lo passo alla funzione `seman()` che fa l'analisi semantica. La funzione `seman()` prende quindi un puntatore all'albero come parametro. Su tale riferimento scatta poi il pattern visitor. Faremo una prima visita per la stampa dell'albero, per vedere se l'albero sia come lo vogliamo. La seconda visita farebbe l'analisi semantica, mentre la terza potrebbe essere dedicata alla generazione del codice.

10.27 Dalla grammatica all'albero

Sulla grammatica mettiamo le azioni per costruire l'albero sintattico. Ho bisogno di classi per ogni tipologia di costrutto del programma. Ogni costrutto sarà un sottoalbero. Verrà specificato come dovranno essere fatti gli alberi e sottoalberi per i costrutti. Tale specifica dipenderà dal linguaggio. Ogni volta che riconosco un costrutto dovrò poi attaccare il sottoalbero di quel costrutto al padre. In casi normali, la forma del sottoalbero potrebbe essere differente. Tale forma la sceglie l'implementatore del linguaggio. Chi inventa il linguaggio si ferma alla specifica lessicale e sintattica, mentre la struttura dell'albero è problema di chi implementa il linguaggio. Ciò verrà fatto nel modo che chi implementa il linguaggio avrà un albero semplice.

Una volta che ho prodotto l'albero, il non terminale iniziale aveva come attributo un riferimento alla radice, pertanto la grammatica ha finito il suo lavoro e da ora in poi, tramite riferimento alla radice, posso visitarmi l'albero. La prima cosa che vorrei fare è stampare l'albero. La visualizzazione posso farla tramite xml o altre librerie come jtree. La visita dell'albero avviene tramite il pattern visitor

10.28 Funzionamento del Pattern visitor

La visita dell'albero viene fatta in depth first da sinistra a destra e tramite pattern visitor. Esso viene utilizzato quando si ha un albero. Il vantaggio principale di tale pattern è che cerca di separare l'obiettivo della visita dalla visita strutturale dell'albero. La struttura della visita è il modo in cui visitiamo, propagando ad esempio dai padri ai figli, mentre se devo stampare o generare codice ciò mi viene dalla semantica della visita

Scrivo un unico codice per la visita strutturale e poi metto dei moduli. Ogni modulo ha una semantica diversa della visita. Ho un unico modulo strutturale e più moduli semantici che fanno azioni differenti, come la stampa, il type checking... Il vantaggio di questo visitor è quello di separare la visita strutturale dell'albero che è fissa per tutti. Ogni volta che devo fare un visitor devo scrivere un modulo. Tale modulo corrisponde a dire cosa devo fare su ogni nodo. Se devo fare la stampa, scrivo dicendo che su quel nodo di cosa devo fare la stampa. Non mi preoccupa che quel nodo sia in un albero. Per ogni nodo specifico l'operazione da fare. La visita strutturale funziona con qualsiasi tipo di nodo

In Java abbiamo ad esempio la possibilità di definire le variabili successivamente ad averle utilizzate. Pertanto, **possiamo fare, tramite visitor, visite differenti: una per la dichiarazioni delle variabili, una visita che ne controlla l'utilizzo e così via.** Si potrebbe fare tutto in una sola visita utilizzando delle notazioni ed inserendo la variabile nella tabella dei simboli con un valore speciale ma tale tecnica è più complessa

10.29 Grammatica precisa o larga

Quando definiamo una grammatica **non stiamo scrivendo perfettamente il linguaggio che vogliamo. Dobbiamo farlo più largo, ovvero il linguaggio potrebbe avere frasi che non vogliamo, ma dobbiamo assicurarci che ci siano tutte quelle che vogliamo.** Sarà poi l'analisi semantica ad escludere le frasi che non ci piacciono. Stiamo **separando la difficoltà a livello lessicale specificando un linguaggio ristretto. Potrei scrivere una grammatica anche per il lessico ed aggiungere altre produzioni.** Quanto più facciamo prima meno faremo dopo e viceversa. **Il giusto equilibrio è quello di mettere il giusto in ogni fase senza complicarla troppo.** Uno dei vantaggi delle grammatiche ed espressioni regolari è che sono facilmente leggibili e non ci sono molte complessità. In tal modo, **aggiungere un nuovo costrutto risulterà semplice. In generale, cerco di scrivere pochi non terminali e di riusarli.** Se io scrivo una grammatica inizialmente complicata, posso successivamente modificarla senza però cambiare il linguaggio, mantenendo la semantica

10.30 Statements ed espressioni

Differenza tra espressione e statement è che **l'espressione è qualcosa che restituisce un risultato. Un'istruzione pura o statement puro non restituisce un risultato. Il concetto è lo stesso della differenza tra funzione e procedura.** Dato "a=b=c=1" essa è un'espressione

in quanto voglio che il valore di 1 sia propagato fino ad a. Il side effect così facendo è che a e b vengono modificati. **Se decido che lo statement non può restituire un valore allora “a=b=c=1” non potrà essere effettuato in quanto c=1 non restituirà 1 come valore.** L’if in C è uno statement puro e se voglio che restituisca qualcosa devo usare l’espressione condizionali

10.31 Pattern Visitor

L’idea è che **ogni nodo è una struttura, una classe.** Quello che andiamo a fare in Java è **andare a mettere un’istruzione di tipo accept(Visitor v) nelle classi che rappresentano i nodi.** Così facendo, quel nodo accetta un Visitor. Esso mi dice cosa fare con i dati che sono in quel nodo. **A seconda delle cose che voglio fare con quel nodo posso fare un visitor diverso:**

- un potrebbe interpretare l’esecuzione dell’albero
- uno potrebbe stampare la visione XML

Io nel nodo metto solo una funzione accept(Visitor v). **Non so quale sia ancora il visitor ma l’albero posso costruirlo lo stesso. Solo successivamente vado a creare un Visitor e posso creare quanti visitor voglio.** Una volta che definiamo la classe, essa non verrà toccata mai più, perchè **ha i dati ed accetta genericamente un visitor. Cambiano le visite ma non bisogna toccare nulla del nodo.** La cosa importante è che hanno estratto la semantica della visita dall’albero. **A seconda se lancio un visitor piuttosto che un altro, l’albero diventa qualcosa di differente.** L’albero è una struttura dati che non tocco. Devo andare a fare un nodo per ogni costrutto del linguaggio.

Ogni costrutto del linguaggio ha delle produzioni che lo riconoscono ed un sottoalbero con una radice. Se io prendo l’if, devo fare la classe per IfOp, in quanto è il nodo dell’operazione di if. Tutti i sottoalberi ifOp si comporteranno allo stesso modo pertanto se il visitor va bene per un nodo ifOp va bene per tutti. In altre parole, si parte dalla radice che manda il comando accept ai figli con lo stesso Visitor. Ciò non viene fatto dall’albero ma dal visitor stesso, e ciò mi permette di non dover cambiare l’albero.

10.31.1 Esempio: pattern visitor

Nel caso dell’esempio abbiamo una classe di nome plus che ha un operatore e due riferimenti ad altre classi. Il visitor è utilizzato per la valutazione dell’espressione. Si lancia

il processo andando a lanciare la root, ovvero il riferimento alla radice e vado a lanciare il metodo `accept(Visitor)`. Il codice è `root.accept(ev)`. Se avessi avuto un tipo di visitor differente, passando come parametro l'altro Visitor avrei cambiato la visita semantica dell'albero. A seconda di cosa passo come argomento di `accept()`, sto mandando un visitor differente

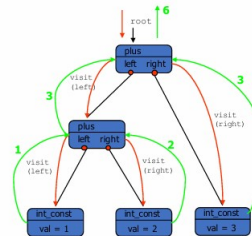


Figura 10.10: Esempio: pattern visitor

I nodi dell'albero sono `plus` ed `int_const`. In tali classi ci sono i dati che deve contenere il nodo ed il metodo `accept()` che accetta un visitor. I dati del nodo `plus` in questo caso sono due riferimenti al nodo espressione `left` e `right` ed il metodo `accept()`. La prima chiamata manda all'istanza `plus` il visitor `ev`. Il metodo `accept()` all'interno del nodo `plus` prende un Visitor e su tale visitor lancia il metodo `visit()` passando il nodo come argomento

La classe visitor ha un unico nome di metodo, ovvero `visit()` ma con firme diverse. Siccome la firma è diversa, quando chiamo visita con la classe `plus` andrà ad eseguire quel metodo con parametro `plus`. Quando il parametro è `int_const`, va ad eseguire il `visit` che ha `int_const` come parametro. Il metodo `visit` prende e propaga il visitor ai figli del nodo in cui stiamo se esso li ha altrimenti fa altre operazioni. In altre parole, si parte dalla radice che manda il comando `accept` ai figli con lo stesso Visitor. Ciò non viene fatto dall'albero ma dal visitor stesso, e ciò mi permette di non dover cambiare l'albero. Ogni volta che lancio la funzione `visit`, esso dà un risultato e prendo il valore di tale risultato. In `leftVal` avrò il risultato dell'albero sinistro la stessa cosa su `right`. Arrivato ad `int_const` posso calcolarmi il valore del nodo e propagare poi il valore al padre e così via fino ad arrivare alla radice.

10.31.2 Esempio: Visitor ed operazioni matematiche

Supponiamo di avere un albero sintattico che rappresenta espressioni aritmetiche ed i cui nodi foglia sono numeri. Quello che faccio è creare un'interfaccia `Node` generica in cui dichiaro la funzione `accept`. Ogni nodo deve avere i propri dati e questa funzione pertanto ogni classe andrà ad implementare `Node`, ad esempio `NumberNode` implementa `Node`.

Quello che si deve fare è chiedere al visitor di elaborare se stesso. Chiamo il metodo `visit()` del `Visitor` con la classe stessa e ci sarà una funzione che sa come lavorare. In questo esempio si ha anche `OperatorNode`. In questo caso voglio avere una visione XML dell'albero pertanto devo lavorare con `XmlBuilder`. Si richiama la `root` e su di essa viene chiamata `l'accept`. Il visitor richiama `visit()` su `Number` e `visit()` su `Operator`. Se il parametro passato alla funzione è `NumberNode` allora effettuo un'operazione, altrimenti se è `OperatorNode` ne effettuo un'altra.

10.32 Direzione di propagazione

Alcuni valori possono andare dal basso verso l'alto. Quando faremo l'analisi semantica, una prima visita sarà quella per costruire la tabella dei simboli e dalla seconda visita si avrà bisogno di tale tabella, anche se ogni nodo avrà poi una tabella differente in base allo scoping. Ci saranno sì dei valori che vanno dal basso verso l'alto, mentre ci saranno altri casi in cui devo fare scendere il valore ai figli. Lo scoping del costrutto che include un altro costrutto è facilmente gestibile. Il padre ha la tabella dei simboli e la deve passare al figlio ed il figlio fa l'analisi con quella tabella dei simboli. L'idea è che ci possono essere valori che salgono, altri che scendono o mischiati tutti insieme. Riusciamo a risolvere ciò con il `Visitor`

11.1 Ambiente e stato

Durante l'analisi sintattica l'unica cosa che devo fare è realizzare l'albero sintattico. Durante l'analisi semantica è importante che si vanno a discutere due concetti:

- **ambiente:** perlopiù associazione dinamica
- **stato:** perlopiù associazione dinamica

Data un'assegnazione come "a=0; b=1; a=b;" quello che succede è che a e b non sono la stessa cosa. In C a è un L-value (in quanto sta a sinistra dell'assegnazione quindi left value) mentre b è un R-value. L'L-value rappresenta la variabile, ovvero la cella di memoria riferita da a, mentre R-value è il valore, in questo caso 1. Anche se a e b sembrano uguali, quando dico l'elemento a sinistra intendo dire la cella, quando sta a destra intendo il suo valore, non la cella che si sposta in un'altra cella. In questo caso, come solitamente accade, abbiamo variabile e valore. Un compilatore deve associare ad un nome una variabile ed una variabile deve associare un valore. L'associazione del nome alla variabile mi definisce l'ambiente, mentre l'associazione della variabile a valore mi definisce lo stato.

L'ambiente è il collegamento tra nomi e variabili mentre lo stato è il collegamento tra R-value ed L-value. In generale, lo stato della memoria ci rappresenta il blocco di memoria ed i valori in esso contenuti. Prendendo tutti i valori di questo blocco di memoria avremo lo stato di cosa succede in memoria. A seconda dei momenti, ci possono essere associazioni

diversi. Lo stack è lo stato mentre i link sono l'ambiente, le variabili che sono collegate. Quando facciamo debugging non possiamo perdere l'ambiente. Se perdiamo l'ambiente non possiamo fare debugging. Potrei anche perdere il valore di una specifica cella di memoria ma mi serve l'environment, l'associazione nome/variabile

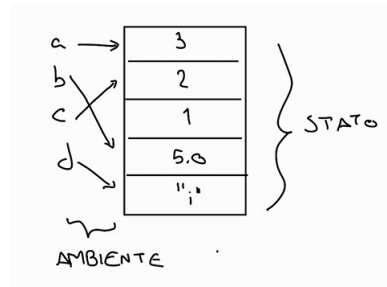


Figura 11.1: Ambiente e stato

11.2 Associazioni statiche e dinamiche per ambiente e stato

Il binding tra un nome ed una variabile a livello teorico dipende dal tipo di variabile. Anche per l'associazione di una variabile alla sua cella di memoria teoricamente statico. Se la variabile *a* è statica, l'indirizzo relativo lo decide il compilatore. Avendo 5 macchine diverse ed una variabile globale, essa però avrà 5 indirizzi diversi, o anche in momenti differenti sulla stessa macchina la sua posizione in memoria cambia.

L'associazione tra nome e variabile è un'associazione considerabile dinamica. Ciò è utile anche per le chiamate ricorsive, in quanto il blocco di memoria in cui lavora la funzione non è fisso. Se *F* chiama *F* non è che si sovrascrive. La stessa variabile *a* della funzione *f* ha due ruoli: sulla *F* chiamante al top dello stack e della *F* chiamata al top +1. La stessa *a* ha l'allocazione di memoria a tempo di esecuzione. Se la funzione *F* verrà chiamata molte volte, si avranno più istanze di *a* e solo una sarà attiva per ogni momento.

Lo stato, ovvero l'associazione tra variabile e valore sarà invece dinamico. Il valore che associa ad una variabile deve essere dinamico (nella maggioranza dei casi)

11.3 Compilatore e gestione della memoria

Durante lo scoping ci interessa che quando abbiamo un nome, il compilatore non saprà quale sia la memoria ma saprà calcolare, a partire da quel nome, dove si troverà in memoria. Ad esempio una variabile globale si troverà all'inizio del blocco di memoria.

Seppur l'allocazione di memoria per le variabili avviene in modo dinamico è il compilatore che deve fare in modo che le variabili non si sovrappongono. Se a e b avranno ad un certo punto la stessa memoria, ciò sarà un problema. Seppur il compilatore organizza, non saprà bene quale sarà la memoria associata, eppure si assicura che le variabili non creino problemi l'una con l'altra.

Per calcolare l'indirizzo di una variabile ci saranno delle formule matematiche da inserire nel codice macchina. Rispetto a dove ti trovi, ti devi sposare in una posizione. Il compilatore è un orchestratore della disposizione delle locazioni di memoria, ma non può assegnare una memoria fissa

11.4 Scoping

Nei linguaggi di programmazione, il nome può essere usato con tipi diversi e momenti diversi, quello che li distingue è lo scope. Se sono nello stesso scope ho un errore. Per evitare questo problema, durante l'analisi semantica, se sono nello stesso scoping, devo evitare che una stessa variabile venga dichiarata più volte. Scoping diversi sono accettabili, questo perché poi ai nomi devo dare una memoria e se sta nello stesso scoping avrà la stessa memoria.

Lo scoping ci permette di associare delle variabili a dei blocchi di codice. Esso ci dice in una certa parte del codice quali sono i nomi visibili e quali utilizzabili. Lo scoping non è un concetto intuitivo in quanto linguaggi differenti hanno scoping differenti.



```
public static void main(String args[])
{
    {
        int a = 2;
    }
    {
        int a = 3;
    }
}

public static void main(String args[])
{
    int a = 2;
    {
        int a = 3;
    }
}
```

Figura 11.2: Scoping

Questi due metodi sono corretti rispetto a C, in quanto le due variabili a non vanno in collisione tra loro. Le due variabili sono a livelli differenti, pertanto non è un problema. Al contrario, in Java la seconda immagine crea un conflitto. La keyword extern dichiara la variabile fuori dallo scope, come se fosse globale.

11.5 Tipi di scoping

Lo scoping può essere statico e dinamico. **Solitamente abbiamo solo uno scoping statico ma i primi linguaggi di programmazione avevano uno scoping dinamico.** Le differenze sono:

- **Scoping statico:** guardando solo il testo posso dire lo scope in cui si trova la variabile.
- **Scoping dinamico:** dipende dall'esecuzione.

Lo scoping che eredita una funzione nello statico è quello del blocco che lo racchiude, nel dinamico è quello che lo ha chiamato. A tempo di esecuzione, se io non trovo una variabile in uno scope, vado a cercare la variabile nello scope che lo racchiude. A livello statico vado a vedere il contesto, cosa racchiude quella variabile, mentre con il dinamico devo andare a vedere chi ha chiamato. In generale, lo scope è quella parte del codice in cui la variabile è definita ed usata

11.5.1 Esempio: Scoping e blocchi di codice

Supponendo di avere un programma con blocchi differenti. **B₁ include tutti, B₂ include B₃ e B₄, che sono paralleli.** Quando io vado in B₃ e B₄ ad esempio e in tale blocco viene usata una variabile che non è definita all'interno del blocco stesso, devo cercare nello scope più vicino che lo racchiude, in questo caso B₂. La classificazione è innestata, pertanto possiamo rappresentarla in modo gerarchico.

Se mi ritrovo ad analizzare un'istruzione in B₄ a livello di AST avrò un nodo per la stampa che fa riferimento alle variabili a, b. Se la variabile a non è nel mio scope, salgo da B₄ a B₂. Se in esso non è contenuta la variabile a salgo ancora a B₁ e mi vado a prendere il tipo della a che si trova nella tabella associata a B₁. A livello di esempio B₁ è la radice del sottoalbero ma sarà a sua volta contenuto nello scope della funzione che contiene B₁ e così via. Tutto deve avere un tipo, anche i nomi delle funzioni e variabili, in quanto non sono note a priori. Il compilatore deve estrarre dalle nostre tabelle i tipi delle variabili e funzioni. Se ad esempio B₁ racchiude B₂ come in questo caso, e io definisco a in B₁ che è definita anche in B₃, lo scope della a in B₁ sarà B₁ - B₃, in quanto in B₃ lo scope di a sarà differente. Lo stesso ragionamento si applica per tutte le variabili

11.6 Scoping e tabella dei simboli

La tabella dei simboli non sarà quindi visibile come una tabella unica ma dovrò fare una tabella dei simboli differente per ogni scope. Uscirà un albero in quanto tali tabelle saranno collegate perchè devo sapere chi mi racchiude le altre tabelle. La nostra tabella dei simboli avrà una struttura ad albero che non è la stessa dell'AST. Ogni nodo dell'albero punterà alla sua tabella dei simboli. Nell'analisi semantica non ci interessiamo dei valori ma solo che i tipi siano corretti.

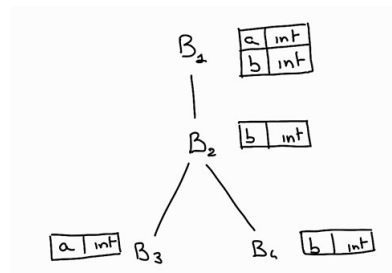


Figura 11.3: Scoping e tabella dei simboli

Quando facciamo l'analisi semantica, del valore della variabile, ovvero dello stato, non ci interessa nulla. Ci interessa solo l'ambiente, ovvero l'associazione nome/variabile e il compilatore deve orchestrare la memoria in modo tale che non si sovrappongono variabili che hanno lo stesso nome e scoping diversi. Devo chiarire bene quale sia il mio ambiente. Le tabelle dei simboli saranno una struttura gerarchica in cui si ha la radice. Tale radice sarà lo scope più esterno.

La confusione dell'analisi semantica è che si pensa all'esecuzione. Durante l'analisi semantica, però, non si sta eseguendo, ma si sta analizzando un testo. L'esecuzione guarda lo stato, ovvero cosa finisce nelle variabili, mentre noi qui dobbiamo capire quali variabili assegnare ai nodi e quanto devono essere grandi (in base al tipo), come devono essere formattate (ancora il tipo).

Se invece nella stessa tabella dei simboli dello stesso nodo ho la stessa variabile definita con due tipi differenti, avrò un conflitto. Scope differenti avranno memoria differente. A livello logico, in uno scope, ad ogni nome associamo una memoria sola

11.7 Analisi semantica e regole

Quando definiamo un linguaggio, abbiamo delle regole. Esse ci permettono di capire cosa sia ammissibile o meno. Non essendoci un linguaggio universalmente accettato, le

regole tra linguaggi differenti sono differenti. Seppur avremo una **notazione formale basata sulle regole di inferenza**, esse non sono implementabili. Servono soltanto per mettere le cose in chiaro. Cose più comuni potrebbero essere:

- un identificatore **deve essere dichiarato al più una volta all'interno di uno scope**
- un **identificatore non dovrebbe essere usato se non dichiarato** (tale regola non esiste nei linguaggi interpretati in quanto si va a fare inferenza di tipo). Il fatto che non dichiariamo la variabile prima di usarla **non vuol dire che il linguaggio sia interpretato ma solo che si fa inferenza di tipo. Tramite un motore di inferenza, ovvero qualcuno che fa inferenza di tipo, potrei non definire il tipo.** Sarà poi il compilatore che deve andare a cercare di capire il tipo. Ciò può essere fatta con l'esecuzione simbolica. **In base a come si utilizza la variabile si cerca di capire il tipo**
- il **tipo della parte sinistra dell'assegnamento dovrebbe fare il match della parte destra.** Ciò in C non è sempre vero in quanto possiamo fare la **coercizione di tipo** (assegnare un valore intero in un reale)
- **non si possono avere metodi con firme diverse** (quindi non si può avere il polimorfismo)

Quelle cose che ci sembravano normali da usare ora dobbiamo definirle. Tali regole sono **regole dell'analisi semantica ma messe in linguaggio naturale. Tale linguaggio non c'è un perché ma definisce queste regole semantiche.** Se non è dichiarato, deve essere almeno possibile inferire il tipo. Durante l'analisi semantica, **oltre allo scoping c'è il controllo delle dipendenze tra classi. Si va a fare un grafo con le classi e dipendenze e all'interno di tale grafo si vanno a cercare loop. Se ci sono loop si dà errore senza fare la visita dell'AST.**

11.8 Tecniche per la risoluzione dei problemi

Data una grammatica, in base al parsing che ci andiamo a fare abbiamo bisogno di modificarla, rispettando i prerequisiti. Quando abbiamo un conflitto, quello che possiamo fare è:

- **aggiungere le precedenze**
- **aggiustare la produzione.** In alcuni casi far passare la riduzione da destra a sinistra risolve il conflitto

- far **gestire a javacup il conflitto**. In tal caso tra shift/reduce javacup fa shift mentre reduce/reduce si fa a riduzione della produzione definita prima. **Ciò toglie sicuramente i conflitti, ma dobbiamo assicurarci che la semantica del linguaggio sia ancora giusta. In casi come il dangling else, javacup riesce a risolverlo e nel modo giusto.** Su problemi noti possiamo usarlo, mentre **su problemi non noti bisogna stare attenti che non stiamo andando ad accettare frasi che non volevamo accettare. Possiamo usare questa tecnica se siamo sicuri che il modo che ha javacup si risolvere il conflitto sia quello che volevamo.** Non dobbiamo perdere il controllo del generatore automatico di codice
- se ci sono troppi ϵ , si va a sostituire la produzione in ϵ copiando quella che non ha ϵ . **Non sempre gli ϵ vanno tolti, alcune volte potrebbero aiutare.**

Nel nostro linguaggio, se chiamo un'espressione essa non deve avere SEMI, altrimenti se la chiamo come istruzione devo averla

11.8.1 Esempio: Problema di statement e dichiarazione

Codice come "x:= y : integer; // dichiarazione x:= y; // istruzione" è accettabile dal nostro linguaggio. Tale **non è un problema della grammatica in quanto descrive proprio cosa vorremmo che facesse** Il problema è che quando l'andiamo a mettere con Javacup ci dà conflitto. A tal fine, bisogna modificare la grammatica.

In generale, **posso avere una lista di dichiarazioni seguita da una lista di istruzioni. Se dopo la dichiarazione ho un'istruzione, logicamente va bene in quanto la dichiarazione finisce con la prima linea ed inizia l'istruzione.** Per come è scritta la grammatica, il parser cerca di vedere la seconda come l'inizio di un'altra dichiarazione. **Devo cercare di far capire che la seconda non è una dichiarazione, seppur inizi allo stesso modo, ma l'inizio di un'istruzione.** Quello che possiamo fare è modellare la grammatica e farla diventare:

- $S' \rightarrow S$
- $S \rightarrow Ds Sts$
- $Ds \rightarrow D Ds$ // invertendo per avere ricorsione sinistra si eliminano conflitti
- $Ds \rightarrow "$
- $Sts \rightarrow St Sts$
- $Sts \rightarrow "$

- $D \rightarrow a b$
- $St \rightarrow a$

Avremo Ds che è una sequenza di singole dichiarazioni e Sts sarà una sequenza di statements. Il problema è che sia St che D iniziano con a , seppur in D dopo a ci sia b . Piuttosto che usare javacup per vedere il conflitto, possiamo farci aiutare dal tool lalr, inserita la grammatica e lanciandolo possiamo vedere dove siano i conflitti. Andando a trasformare la ricorsione destra in ricorsione sinistra, ho risolto il conflitto. $Ds \rightarrow D Ds$ diventa $Ds \rightarrow Ds D$. In tal modo posso fare dichiarazione e dopo istruzione in quanto non ho cambiato il linguaggio. Il problema nella grammatica è che c'è uno shift reduce sotto simbolo a . Ciò significa che c'è D che vuole shiftare sotto il simbolo a ed St che vuole ridurre.

11.9 Semantica e regole semantiche

Espressioni regolari e grammatiche possiamo usarli con tool. A livello di ingegneria del software, esse sono specifiche formali di alto livello. Esse sono non ambigue ed eseguibili in quanto esiste Jflex o javacup che le prende e le traduce. Nell'analisi semantica avremo una specifica formale, ovvero le regole di inferenza. Molti linguaggi di programmazione li utilizzano per specificare le regole dell'analisi semantica. Esse ci servono a livello di specifica formale non ambigua. Potrebbe esserci dell'ambiguità usando il linguaggio naturale.

Tutte le cose che non siamo riusciti a specificare nell'analisi sintattica, andiamo a specificare nell'analisi semantica. Il controllo dei tipi, non è un qualcosa che si è riusciti a fare in analisi sintattica. Esso non si può fare in quanto sono troppo lontane le cose. Nell'analisi lessicale gli elementi sono tutti nella stessa parola. Nell'analisi grammaticale possono stare un poco più lontani ma nell'analisi semantica e cose sono lontanissime. Si può addirittura importare un modulo che ha delle dichiarazioni e poi nel file si utilizzano dichiarazioni del modulo esterno.

11.10 Definizione dell'analisi sintattica estesa (analisi semantica)

L'analisi semantica o analisi sintattica estesa (in quanto ciò che non riesco a farlo nella sintattica la faccio nella semantica) cerca di controllare il significato del programma e fa dei controlli sia a lunga distanza che in profondità (il parser è a breve distanza). Una peculiarità della grammatica è che ogni produzione è locale. Se deve usare qualcosa della produzione posso farlo, se devo usare qualcosa del mondo esterno lo richiama e sarò poi compito di

quella produzione di gestire il problema. Con l'analisi semantica bisogna mischiare le cose. Essa viene implementata con una visita dell'AST.

11.11 Definizione di scope

Lo scope è la parte del programma in cui una variabile è visibile o accessibile. Si può avere uno scope ristretto (poche istruzioni) in cui identificatori possono riferire a cose diverse in parti diverse del programma. Lo stesso nome in una parte potrebbe essere una stringa e in un'altra un intero. Essendo in scoping diversi fanno cose diverse. Posso quindi usare lo stesso nome più volte. Gli scoping per lo stesso nome però non si possono sovrapporre. Non posso avere una parte di due scope in comune che usano la stessa variabile.

Non è che si va a sovrapporre lo scope vero e proprio. Per sovrapposizione si intende che la variabile è viva con due definizioni differenti. In general gli scope non si sovrappongono in quanto seppur uno scope ne contiene un altro, dallo scope grande vado a togliere il codice dello scope piccolo in cui è dichiarata, pertanto vado a togliere gli scope piccoli dallo scope globale (avendo uno scope globale con buchi)

11.12 Regola del most-closely nested rule

Gli identificatori seguono la regola del most-closely nested rule. Bisogna fare attenzione alla differenza di uso e definizione della variabile. Quando uso una variabile, se non trovo la dichiarazione nello stesso scope, vado a trovare la definizione nello scope più vicino che lo racchiude, nel confinante che lo racchiude. Non tutti però usano questa regola, ad esempio lo scoping dinamico.

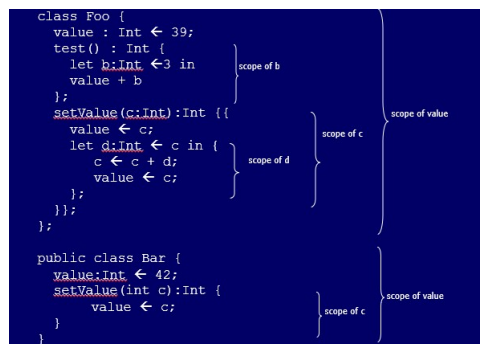
Con lo scoping statico si legge il sorgente ed in base a ciò si fanno le analisi, senza eseguire nulla. All'esecuzione vera è propria servono i dati ma i dati con lo scoping statico non sono noti. Possiamo immaginare come verrà eseguito un pezzo di codice e sulla base di ciò si fa la pianificazione della memoria e del resto. Lo scoping dinamico non segue questa regola. Si andrà ad eseguire il programma per capire quale sia lo scope o che valore abbia. A priori non potrò sapere il valore delle variabili non eseguendo lo scoping dinamico, in quanto dipende da quali dati ci sono, da quale sia il flusso di esecuzione, da quali chiamate verranno fatti e quali no

Lo scoping dinamico mi dice a tempo di esecuzione andando a vedere sullo stack delle chiamate, mentre con lo scoping statico avremo uno stack che viene generato durante la lettura del codice

11.13 Studiare lo scope

In generale, quando facciamo l'analisi semantica la prima cosa da fare è cercare di capire quali siano i costrutti che aggiungono dichiarazioni creano lo scope. È importante saperlo in quanto ogni volta che andremo a trovare chi crea lo scope, dovremo inserire le dichiarazioni nella tabella ed avere una nuova tabella per ogni scope. Alcuni esempi possono essere:

- **classe stessa che dichiara gli attributi:** Ogni volta che si entra in una classe bisognerebbe aggiungere tali attributi
- **espressione let:** esso può essere annidato, pertanto si potrebbe avere un le all'interno del let
- **tramite parametri formali**
- **definizione di attributi**
- **dichiarazioni di metodi e classi**



```

class Foo {
  value : Int ← 39;
  test() : Int {
    let b: Int ← 3 in
      value + b
  };
  setValue(c: Int): Int {
    value ← c;
    let d: Int ← c in {
      c ← c + d;
      value ← c;
    };
  };
};

public class Bar {
  value: Int ← 42;
  setValue(int c): Int {
    value ← c;
  }
}
  
```

Figura 11.4: Studiare lo scope

I parametri di una funzione sono nello stesso scope del corpo della funzione. Se un parametro sarebbe nascosto nello scope, non avrebbe senso passarlo alla funzione. In generale, {} in C creano sempre un nuovo scope, ma in Java no, pertanto dipende dal linguaggio. Se le classi sono globali, c'è uno scoping globale e nella tabella corrispondente devo mettere i nomi delle classi.

11.14 Regole di scoping

Le regole di scoping ci fanno fare il type checking, ovvero fare match delle dichiarazioni con l'uso. Lo stesso nome lo trovo 2 volte: **nella dichiarazione e nell'uso**. Tali operazioni o sono di seguito nello stesso scope o in scope differenti. In generale, la definizione deve essere scritta prima

11.14.1 Esempi di regole semantiche

Esempi di regole semantiche sono:

- **Le classi possono essere usate prima di essere definite. Posso avere una classe Foo che chiama la classe Bar anche se Bar viene dichiarata dopo.** Gli attributi di una classe, sono globali nella classe stessa, pertanto non possono essere usati in altre classi.
- **Un attributo posso dichiararlo comunque alla fine della classe ed usarlo prima.**
- **I metodi stessi possono essere definiti anche nella classe da cui si eredita.** Anche se la classe B non ha il metodo foo, dato che B eredita da A e A ha il metodo foo, allora anche B lo avrà
- **Se B eredita da A, lo può anche riscrivere pertanto anche richiamando B.foo() sto richiamando il metodo riscritto in B**
- **Le variabili locali di un metodo devono essere dichiarate prima delle istruzioni del metodo**
- **Le variabili non possono essere definite più volte nello stesso scope ma possono essere definite in scoping annidati**

11.15 Scoping e tabelle dei simboli

Si parla di scope diverso in quanto a livello di implementazione, quando diciamo scope diverso bisogna creare una nuova tabella. Gli scope sono implementati usando la tabella dei simboli. L'analisi lessicale mette i nomi degli identificatori all'interno della tabella delle stringhe e non in quella degli scope, in quanto quest'ultima dipende dagli scope, dovendo arrivare all'analisi semantica. Le tabelle dei simboli sono implementate come delle tabelle look-up, con coppie di (chiave, valore). Fin quando ho un programma ed i nomi delle variabili sono usati una sola volta, non ci sono problemi in quanto non ho

ambiguità. Se ci sono variabili con stesso nome nello stesso scope avrò problemi e dovrò usare tabelle per lo scope differenti. Quando vado a fare la lookup della tabella sul nome *a*, dove *a* è definita due volte, non so se fare il match della *a* con l'intero o con la stringa. Un'unica tabella come quella dei simboli non va bene per questo motivo. Bisogna creare una tabella per ogni scope

In generale, la tabella dei simboli potrà avere una colonna per il simbolo, una per il kind ed una per il type. Dovremo distinguere il "kind" dal "type", dove kind indica variable, method... e type indica int, float... Quando si metterà il type del metodo, si metterà la firma del metodo. Se ho una funzione che prende due interi e restituisce float, avrà `int*int -> float`. All'interno della tabella di scoping andrò a mettere solo le dichiarazioni. Posso o fare la visita delle dichiarazioni prima e poi faccio scope checking o lo faccio insieme. In generale, faccio puntare un nodo alla tabella

11.16 Definizione di variabili e passi

Se si introducono nuove dichiarazioni si introduce una nuova tabella. Supponendo di avere `"let x: int <- 0 in e"`, prima di entrare in `let` ho già delle variabili attive, sono già in uno scoping in cui la *x* potrebbe essere già dichiarata. Se io non la dichiarassi andrei ad usare la *x* definita prima, avrei errore. Ogni volta che entro in un nuovo scope, devo aggiungere le definizioni di un nuovo scope, alle correnti, sovrascrivere una definizione di *x* e ciò lo faccio per implementare la most-closely nested rule. Se io ho già una *x* quando la dichiaro, vado a nascondere quella e vado a prendermi l'ultima. Quello che bisogna fare prima di processare l'istruzione è:

- aggiungere la definizione a quelle correnti
- nascondere le altre definizioni della stessa *x*. Fatto tali operazioni posso entrare nell'espressione e posso andare a lavorare sugli usi delle variabili. Dopo aver analizzato tutti i nodi della *e*, la *x* si deve perdere in quanto ho finito lo scope
- Devo rimuovere la definizione di *x*
- riprendere la definizione della *x* precedente

11.16.1 Esempio: Programma e scoping

Lo scoping globale per questo esempio è rappresentato da *Foo*. Lavorando sulla classe *Foo*, io ho un attributo *value* e due metodi: *test* e *setValue*. Genero la tabella, per *Foo*,

```

class Foo {
  value : Int ← 39;
  test(b:Int) : Int {
    value + b
  };
  setValue(c:Int):Int {{
    value ← c;
    let d: Int ← c in {
      c ← c + d;
      value ← c;
    };
  }};
};

```

Figura 11.5: Esempio: Programma e scoping

vedendo poi i due figli ed andandoli a mettere nella tabella. Quando vado ad analizzare il sorgente della classe, mi troverò ad analizzare il sorgente della funzione test. Siccome test introduce una dichiarazione, lancio un'altra creazione di tabella e ci vado a mettere solo le cose che test ha dichiarato.

Entrando in test, una volta che vedo le dichiarazioni, potrei già analizzare il corpo e gli usi delle variabili. Quando vedo gli usi delle variabili, devo partire da test nel ricercare la definizione. Se trovo una b in test allora associo quell'uso a quella definizione. Finito di analizzare test passerò ad analizzare la funzione setValue. All'interno della tabella di scoping andrò a mettere solo le dichiarazioni. All'interno della funzione setValue ho un let che dichiara a sua volta una variabile. Siccome dichiara variabili devo creare una nuova tabella per il body di setValue.

Vedendolo da punto di vista di un AST banalmente entro nel nodo del metodo setValue e da ciò mi vado a visitare il nodo che dichiara le variabili. Quando incontro il let, avrò incontrato il nodo dell'albero. Mi immagino un nodo che ha a destra le istruzioni e a sinistra le istruzioni. Parto dalla sinistra e vado a segnarmi tutte le definizioni. Quando vado ad analizzare la parte destra dell'albero, troverò variabili definite nella parte sinistra o no. In caso no, vado a cercare la variabile nell'ultima tabella che ho creato. Parto con value e lo cerco in setValue. Non trovo value, seguo l'iter al contrario fino a trovarlo, in questo caso nella tabella di Foo. Lo stesso codice può essere visibile con AST.

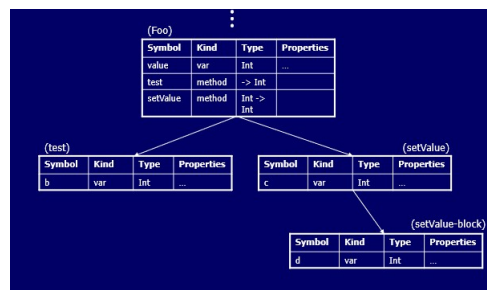


Figura 11.6: Esempio: Programma e scoping

11.17 AST e albero delle tabelle di scoping

Il nostro input è l'AST con i puntatori alla tabella delle stringhe per i nodi. Visito l'albero partendo dalla radice. **La radice l'ha restituita l'analizzatore sintattico. Il nostro main in javacup ha lanciato il parser e restituisce come valore finale il puntatore alla radice dell'albero.** Visitiamo la radice e sappiamo che è un nodo classe, pertanto sappiamo che esso è un nodo che genera scoping.

Prima di visitare il nodo, se esso è un nodo che genera scoping allora creo la tabella di quel nodo. Inserisco il nome del nodo nella tabella globale e creo una nuova tabella (figlia della tabella globale). Quando nell'AST mi sposto all'interno del nodo della classe e vedo ad esempio un metodo, essendo che anch'esso può generarmi delle dichiarazioni, inserisco il nome del metodo nella tabella della classe e creo una tabella di scoping per il metodo e così via. Visito poi l'espressione. Scrivo la variabile all'interno della tabella. L'espressione non è un nodo di scoping, quindi non genero una tabella. **Della variabile però non guardo nemmeno gli usi in quanto ciò lo faccio nella fase successiva.**

Ogni volta che vedo un **nodo che crea scoping, creo la tabella e fornisco al nodo un riferimento alla tabella e tutti i figli di quel nodo avranno un riferimento a quella tabella. Il padre deve dare ai figli il riferimento alla tabella corrente.** Tale riferimento viene passato dal padre ai figli. **Non è detto che ogni tabella generi una sola tabella, pertanto se all'interno di un metodo di una classe ci sono istruzioni e più metodi (entrambi generatori di scoping) andrò a mettere tutti i nomi dei metodi nella tabella del padre e poi vado a crearmi più tabelle che hanno come padre la tabella della classe in cui sono contenuti. Ogni volta che viene creata una tabella avrò un nuovo riferimento e i figli ereditano solo il riferimento allo scoping corrente, ovvero alla tabella del padre, non quella degli antenati del padre.**

11.18 Implementazione della tabella di scoping

La tabella dei simboli può anche essere uno stack. **In ogni momento di analisi, non vedo tutto l'albero ma solo una parte.** Posso pensare che prima di entrare nel **primo nodo** ho sullo stack la tabella global. Vedendo la classe vado a mettere quella sullo stack facendo lo stesso per i metodi e così via. Ci sono due tipologie di implementazioni:

- alberi con puntatori all'indietro
- non ho i puntatori tra le tabelle ma uno stack.

Ogni volta che vedo un **nodo** prendo la **tabella** e la metto sullo **stack**. Importante è che siano mantenuti i **legami**. Nella prima visita, ogni **nodo** punta alla sua **tabella**. Nella seconda visita, quando vedo un **nodo** con **tabella** metto la **tabella** nello **stack**. L'analisi lessicale non poteva fare la **tabella dei simboli** in quanto esiste una **tabella dei simboli** per ogni **scope**. Inoltre, l'analisi lessicale non vedeva nemmeno i **tipi delle variabili**. Pur vedendo i **tipi** avrebbe dovuto vedere anche lo **scope**. In generale, si può fare ma bisogna anticipare l'analisi semantica. Quello che si potrebbe fare è **mantenere traccia delle parentesi**, in quanto le **parentesi** danno il **blocco di scope**. La **key** diventa però non solo il **nome della variabile** ma anche il **blocco di scope** contando le **parentesi**.

11.19 Implementazione della tabella dei simboli

La **tabella dei simboli** è quindi un **hash table**, dovendo implementare i metodi. Alcuni sono:

- **enterScope()**: inizia un nuovo **scope**
- **lookup(x)**: mi trova la **x** corrente
- **addId(x)**: aggiunge l'elemento alla **tabella**
- **probe(x)**: vede se **x** è definito nello **scope** corrente
- **exitScope()**: esco dallo **scope** corrente. Quando torno su nell'albero devo tornare allo **scope** precedente e usando lo **stack** devo togliere lo **scope** corrente dallo **stack**. Ogni **tabella** farà riferimento però al **padre**

11.20 Riassunto: Regole dell'analisi semantica

L'input di questa fase è l'albero sintattico. Tutte le analisi verranno effettuate su tale albero. Le regole dell'analisi semantica saranno espresse in linguaggio naturale. Le riscriveremo tramite regole di inferenza in linguaggio più formale. Essa è una semplice traduzione da **linguaggio naturale** a **semantico**. Tali regole di inferenza le definisce chi definisce il **linguaggio**. Più sono chiare, concise e non ambigue e meglio è.

L'analizzatore semantico, in generale, andrà a costruire le **tabelle di scoping** e ciò dipende da come è costruito il **linguaggio**. Ci potrebbero essere espressioni che producono nuovi **scoping**, altri invece no. Una semplice espressione aritmetica non produce una **tabella** e non

dichiara variabili. Nel nostro programma tra `progam` e `begin` hanno una sezione di dichiarazione. Ogni dichiarazione deve andare nella tabella ed ogni funzione ha il proprio scoping. Lo stesso per `while` ed `if`. Anche tra `begin` ed `end` abbiamo una sezione di dichiarazione.

11.21 Fasi dell'analisi semantica e scoping

Stiamo usando lo scoping statico. Una volta che abbiamo gli scope essi vanno applicati nel type checking, ovvero utilizzo le tabelle per vedere se i tipi sono corretti. Le fasi sono quindi:

- **creazione degli scope:** creazione delle tabelle
- **type checking:** utilizzo delle tabelle per vedere se i tipi sono corretti. In altri termini, facciamo in modo che il type checking sia rispettato.

Non esiste un'unica tabella per un unico programma e bisogna costruire una tabella per ogni costruito. Abbiamo in input l'AST ed ogni costruito ha il suo nodo radice. Il nodo `program` avrà i figli per le dichiarazioni ed i figli per gli `statement`. Navigando le dichiarazioni, quando troviamo un nodo che genera scope creiamo subito la tabella e visitando le sue dichiarazioni popoliamo la tabella con le dichiarazioni. La tabella la leghiamo al nodo. Tutti i nodi figli che stanno in quello scoping punteranno alla tabella di quello scoping. Se incontro un nodo di scoping creo la tabella e vado a visitare le dichiarazioni nel corpo. Tale dichiarazioni le inserisco nella nuova tabella. Ciò lo faccio iterativamente. Faccio poi in modo che tutti i figli che usano variabili di quello scoping puntino a quella tabella. Annoto l'albero con la tabella di scoping, andando ad analizzare tutte le dichiarazioni, vado a fare il type checking vero e proprio. Abbiamo l'inizializzazione delle variabili durante le dichiarazioni. Anche su esse va fatto type checking. O il type checking lo facciamo durante la prima passata o dobbiamo fare la seconda passata. Logicamente sono due fasi: costruzione delle tabelle e type checking

11.22 Visita dell'albero per la costruzione delle tabelle di scoping

Devo mettere i nomi delle dichiarazioni. Se ho delle funzioni non entro in esse ma segno solo la loro firma. Non vado all'interno dei nodi ma leggo soltanto i nomi. Quando poi entro in un metodo, essendo che esso è un costruito che può creare dichiarazioni, ovvero può essere un nodo di scoping, avendo già il suo nome nella tabella del padre,

creo direttamente la nuova tabella. È importante che i nodi dell'albero puntino al loro scoping. Una volta che le tabelle sono create, vado a fare type checking. Quando io vado a considerare l'albero, nella seconda visita, potrei trovarmi ad analizzare un nodo, non interessandomi più le dichiarazioni, in quanto il nodo avrà già una tabella di riferimento

Ogni nodo avrà una tabella di riferimento, potendo vedere se la tabella contiene le variabili che utilizziamo. Se manca la definizione di una variabile, possiamo poi risalire l'albero. Parto dalla tabella di scoping di riferimento e salgo.

11.23 Tabelle di scoping con stack ed alberi

O ci conserviamo i link della struttura gerarchica, oppure man mano che andiamo avanti nell'analisi del type checking, tutte le tabelle dei simboli che troviamo le mettiamo su di uno stack (ciò avviene solo durante la seconda fase). Vado sul top dello stack perchè sono sicuro che trovo lo scope che serve a me. Trovando un nodo di scoping aggiunto la tabella collegata a quel nodo sullo stack. Importante è che quando finisco la visita del codice corrispondente ad una tabella di scoping, devo togliere dal top la tabella. In generale, importante è mantenere il riferimento del nodo dell'AST con quello della tabella.

Con lo stack succede che io incontro la classe e metto la tabella associata a quella classe. Incontro poi il metodo della classe e faccio lo stesso, mettendo la tabella del metodo nello stack. Lavoro su questa tabella e quando finisco tolgo la tabella del metodo dallo stack, andando a visitare il metodo successivo nella classe. Nello stack dobbiamo inserire la tabella quando incontriamo il nuovo costrutto e quando lo lasciamo dobbiamo toglierlo. Alternativamente, se sto facendo un'implementazione ad albero, ho un riferimento che va dal figlio al padre. La tabella di scoping sarà un hashtable con coppia (chiave, valore). La key sarà l'identificatore e il valore sarà un oggetto che mi dà le informazioni sull'identificatore, ovvero type e kind

11.24 Combinazione delle visite

Se volessi fare tutte e due le visite insieme e ho un pezzo di codice in cui la prima funzione ne chiama una ancora non definita, entrando nella funzione esterna vedo cosa usa. Essendo che essa usa una funzione ancora non definita avrei un errore. In tal senso, è meglio dividere le visite. Un'alternativa sarebbe quella di visitare l'albero e quando incontro la funzione interna ancora non definita inserisco un flag a true. Quando incontro

la dichiarazione, controllo che essa sia equivalente all'uso fatto ed in caso tolgo il flag, è come se si aspettasse prima di dare errore. Se alla fine del programma ho qualche tabella ancora con `flag=true`, vuol dire che qualcosa è stato definito e non usato, pertanto avrò un errore. Una sola visita va bene quando la variabile viene dichiarata prima dell'uso. Se avessi un linguaggio in cui le variabili sono dichiarate sempre prima dell'uso, una visita basta. Quando vado a vedere le espressioni sono sicuro che le dichiarazioni sono nello scope corrente in quanto dichiarate prima. Se non ci sono, non c'è speranza che esse vengano definite successivamente.

Quando le dichiaro dopo, si può usare la tecnica di segnarle se incontro prima l'uso della dichiarazione. Metto poi un flag. Con tali flag, comunque dovrei fare una visita successiva, in quanto non vado a controllare il flag stesso quando lascio la tabella. Alla fine di tutta l'analisi vado a cercare i `true`. Una seconda visita c'è, seppur parziale, ed è nella ricerca di risolvere i `true`. In tal caso è più vantaggioso l'albero in quanto mi mantiene la struttura

11.25 Struttura ad albero dei costrutti

I diagrammi di come devono essere costruiti vanno visti insieme alla grammatica. Per gli Op dovremo costruire una classe java che ha come attributi i figli. Essi avranno già i sottoalberi, ciò perché la costruzione è bottom up. I figli vengono costruiti man mano che si fa il parsing ed arrivando alla radice ho già i figli. L'ultimo nodo che verrà costruito è proprio la radice. Seguendo l'azione per costruire tale nodo, avremo già cosa serve con i figli, dovendoli passare come parametri.

Il visitor ci permette di scrivere codice per ogni nodo. La funzione `visit()` del visitor prende in input un nodo e ci lavora sopra, potendo controllare ad esempio nell'`assignOp` se ci siano più di una variabile a destra se a sinistra c'è una funzione... Ogni volta che il nodo va in input alla funzione `visit()` del visitor, all'interno della funzione `visit()` si deciderà come fare la stampa di quel nodo. In generale, il compilatore riuscirebbe a fare anche il controllo che una variabile sia passata per riferimento senza dover specificare tramite simbolo che si usi un simbolo

12.1 Introduzione: Regole di inferenza

Le regole di inferenza rappresentano un linguaggio formale. Le regole della semantica le abbiamo espresse fino ad ora in linguaggio naturale, che potrebbe però essere ambiguo. Non tutto si può formalizzare ma una buona parte sì. Le regole di inferenza rappresentano un formalismo con cui possiamo esprimere un qualcosa. Essa è un'espressione in cui si ha un IF seguita da un THEN. Se l'ipotesi è vera, allora lo sarà anche la conclusione. Le regole di inferenza sono una notazione compatta e matematica degli statement IF THEN.

Volendo fare type checking, possiamo dire: Se E_1 ed E_2 hanno dei certi tipi, allora E_3 ha un certo tipo. Se io faccio $5+4$, per sapere il tipo di 5 e 4 devo rivolgermi al tipo di 5 ed al tipo di 4. Se entrambi sono interi e stiamo parlando dell'operazione somma, allora il tipo del risultato sarà di tipo intero. Se io però faccio $5+1.3$, dove 5 è un intero e 1.3 è un reale, la somma di un intero ed un reale è un reale. Sarà l'analisi semantica a definirci il tipo del risultato. A livello di linguaggio macchina, la somma è implementata o solo per interi o solo per reali (in quanto per il reale c'è o shift della mantissa).

Dipende da chi definisce il linguaggio decidere se si può permettere una somma tra tipi differenti. Solitamente si fa una tabella per gli operatori, dove ogni riga ed ogni colonna è un tipo. In questa tabella possiamo definire che una somma tra reali e interi è un reale, ed in analisi semantica bisognerà fare di conseguenza il casting da intero a reale, in quanto alla generazione del codice macchina si dovrà avere una somma o solo tra interi o solo tra

reali. Il type checking non fa altro che segnare il tipo a tutti i nodi dell'albero, sia esso un tipo void, int... e lo fai funzione dell'albero sintattico.

12.2 Ipotesi e conclusioni

Simboli sono:

- \wedge : indica l'AND
- \Rightarrow indica "if-then", ovvero la conseguenza
- $x:T$ significa che x ha il tipo T

La sentenza "Se E_1 ed E_2 hanno dei certi tipi, allora E_3 ha un certo tipo", scritta tramite regola di inferenza diventa " $(e_1 : \text{Int} \wedge e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}$ ". **Quello che abbiamo scritto in linguaggio normale può diventare in linguaggio formale. Alla fine io posso scrivere la frase in linguaggio naturale tramite regola di inferenza.** La frase " $(e_1 : \text{Int} \wedge e_2 : \text{Int}) \Rightarrow e_1 + e_2 : \text{Int}$ " è un caso speciale di " $\text{Hypothesis}_1 \wedge \dots \wedge \text{Hypothesis}_2 \Rightarrow \text{Conclusion}$ ".

Per tradizione, le regole di inferenza, ovvero questa sequenza di ipotesi che implica una conclusione la si scrive come "

$$\frac{\text{Hypothesis}_1 \wedge \dots \wedge \text{Hypothesis}_2}{\text{Conclusion}}$$

". La freccia diventa la frazione. Tale regola **la si può leggere da sotto a sopra o da sopra a sotto a seconda di come si vuole interpretare. Tutte le regole di tipo hanno quindi ipotesi e conclusioni.**

12.3 Concetto di dimostrabilità

Quando abbiamo ' $e_1:T$ ', stiamo dicendo che "è dimostrabile che e_1 sia di tipo T ". Il nostro compilatore, guardando l'albero, deve arrivare su di un nodo e deve dimostrare che quel nodo è legato ad un costrutto che ha restituito un intero. Lo deve dimostrare nel senso che deve essere sicuramente vera come cosa

12.4 Tipi di regole

12.4.1 Regola per Int

Il fatto che i sia un intero implica che i sia di tipo int. " $i:\text{Int}$ " indica che i è di tipo intero, pertanto passiamo dalla matematica all'informatica. Se " i is an integer" allora " $i:\text{Int}$ ". Nella

notazione delle regole inferenziali diventa:

$$\frac{i \text{ is an integer}}{i:\text{int}}$$

Int è la convenzione, il tipo, che si dà ai numeri interi. Se i è un intero, posso dimostrare che i è di tipo Int. Se però definisco che invece di Int si debba usare Integer, allora la regola dovrà riflettere ciò, dovendo usare Integer e non int. Facendo questa scelta, sono già all'interno del lessico del linguaggio di programmazione. Tale regola è una postulare, ovvero una regola base

12.4.2 Regola per Add

In base ai postulati riesco a dimostrare che $e_1 : \text{Int}$ e $e_2 : \text{Int}$ allora l'espressione $e_1 + e_2 : \text{Int}$, ovvero che l'espressione è di tipo int. La regola è:

$$\frac{e_1 : \text{Int} \quad e_2 : \text{Int}}{e_1 + e_2 : \text{Int}}$$

Avendo una somma $5 + 4$, essendo che sono entrambe fissate come un intero, tale regola mi dimostra che $5+4$ avrà tipo intero. Quando arrivo a $5+4$, avrò già visto sia 5 che 4, pertanto so il tipo di entrambi i figli. Tale regola viene definita da chi definisce il linguaggio. Noi, facendo type checking, quando il type checker arriva sul nodo della somma, si implementa questa regola. Saranno le regole definite a fare la realtà e sulla base di tali regole si baseranno dimostrazioni future. Sia su e_1 che su e_2 dovrò andare poi ad applicare la regola per Int, in quanto sono entrambe Int. In generale, tali regole sono legate all'albero sintattico, infatti sono le regole che bisogna sapere per vedere cosa fare ogni volta che cadiamo su un nodo e che tipo deve avere quel nodo (se lo ha). Con $e_1 + e_2 : \text{Int}$ si sta indicando che è il risultato della somma ad essere di tipo Int

12.4.3 Esempio: Regola per Add

Avendo la regola

$$\frac{\frac{1 \text{ is an integer}}{1:\text{Int}} \quad \frac{2 \text{ is an integer}}{2:\text{Int}}}{1+2 : \text{Int}}.$$

Potrebbe risultare complesso da leggere. Immaginando $1+2$ come il nodo padre e le due ipotesi come figli. Tale regola avrà la forma di un albero. Andiamo sempre a vedere il tipo del nodo in funzione dei figli, che sono stati già calcolati. Se conosciamo i figli sapremo anche come sia il padre. Se 1 è un intero e 2 anche, allora anche $1+2$ che è il mio nodo addOp è un intero.

12.4.4 Regole per Bool

La costante false è di tipo Bool. Essa è **una regola base** che si presenta come

$$\frac{}{\text{false} : \text{Bool}}$$

In questo caso non abbiamo nulla come ipotesi pertanto **in tutti i casi è vero che false è di tipo Bool**.

12.4.5 Regola per String

La regola per la costante string è:

$$\frac{s \text{ is a string constant}}{s : \text{String}}$$

Mentre nel caso di false avrò proprio il token false, in questo caso, **avrò string_const e tale regola mi sta dicendo che string_const è di tipo String, pertanto lo sto legando al tipo del mio linguaggio**. Seppur banali, le cose vanno scritte in quanto devono essere dimostrabili

12.4.6 Regole per not

Se siamo riusciti a dimostrare che un'espressione è di tipo Bool, **andando ad applicare il not a tale espressione mi restituisce un tipo booleano**. Il simbolo \neg va ad indicare il not, la negazione. Ciò viene detto dalla regola:

$$\frac{e : \text{Bool}}{\neg e : \text{Bool}}$$

12.4.7 Regola per il While

Data la regola

$$\frac{e_1 : \text{Bool} \quad e_2 : T}{\text{while } e_1 \text{ do } e_2 : \text{true}}$$

Sull'espressione while **devo fare diversi controlli. Di solito, come conclusione si mette la sintassi del linguaggio che stiamo sviluppando**. Affinchè il while sia corretto semanticamente:

-

Su e_1 **devo controllare se sia un Bool**

e_2 sia qualsiasi tipo. Se e_1 è bool e e_2 è di un certo tipo, allora tale regola è semanticamente ben definita ed ha il tipo true (in questo caso). Se e_1 è stato calcolato ed è un intero, la regola del while non sarebbe corretta, in quanto e_1 dovrebbe essere corretto e per farlo dovrebbe essere di tipo Bool (in C sarebbe corretta una qualsiasi cosa diversa da 0). Per sapere se il while è corretto non mi interessa sapere il tipo di e_2

12.4.8 Regola per identificatore

Se x è un identificatore, di che tipo è? Entrano in gioco la tabella dei simboli, il type environment. Stiamo scrivendo il type system (ovvero l'insieme di regole) e se x è un identificatore, non posso dire che x sia di tipo identificatore, ma devo andare a vedere nello scope cosa sia la x . Partendo dall'albero, sono finito su un identificatore che mi viene fuori la coppia (ID, x).

Per arrivare alla variabile x , ho costruito la tabella dei simboli. Quando leggo x avrò il type environment, ovvero l'insieme delle tabelle vive in quel momento. Vado quindi a fare un lookup di x nel type environment. Se è nella tabella corrente mi fermo, se non c'è vado dal padre e così via fino a trovare la radice o la variabile. La regola:

$$\frac{x \text{ is an identifier}}{x: ?}$$

da sola non basta, in quanto mi devo portare il type environment.

12.4.9 Regole per sequenze di statements

La sequenza di due statement, affinché sia ben tipata, devo definire una regola come:

$$\frac{\Gamma \vdash \text{stmt}_1: \text{notype} \quad \Gamma \vdash \text{stmt}_2: \text{notype}}{\Gamma \vdash \text{stmt}_1; \text{stmt}_2: \text{notype}}$$

Il costrutto di cui mi interessa è " $\text{stmt}_1; \text{stmt}_2$ ", rappresentato dalla conclusione. Tale costrutto è ben tipato ed ha tipo void se il primo statement è ben tipato e non ha un tipo e il secondo anche è ben tipato. il fatto che entrambi siano notype non ci interessa. Gli statement non inseriscono dichiarazioni, pertanto l'esecuzione di uno statement non mi cambia un type environment

12.4.10 Regole per sequenze di statements

La sequenza di due statement, affinché sia ben tipata, devo definire una regola come:

$$\frac{\Gamma \vdash \text{stmt}_1: \text{notype} \quad \Gamma \vdash \text{stmt}_2: \text{notype}}{\Gamma \vdash \text{stmt}_1; \text{stmt}_2: \text{notype}}$$

Il costrutto di cui mi interessa è “ $\text{stmt}_1; \text{stmt}_2$ ”, rappresentato dalla conclusione. Tale costrutto è ben tipato ed ha tipo void se il primo statement è ben tipato e non ha un tipo e il secondo anche è ben tipato. il fatto che entrambi siano notype non ci interessa. Gli statement non inseriscono dichiarazioni, pertanto l'esecuzione di uno statement non mi cambia un type environment

12.4.11 Regole per chiamate a procedura

La chiamata a funzione è più complessa. La prima cosa che andiamo a fare quando effettuo una chiamata a funzione è:

- **vedere se nel type environment la funzione esiste** (esiste il nome della funzione)
- **se esiste, sarò in grado di vedere numero e tipi dei parametri.** Se il primo parametro è intero e la seconda è reale, la chiamata a funzione dovrà avere il primo parametro intero ed il secondo reale

Sono nella radice e voglio vedere se la chiamata a procedura è semanticamente corretta, ovvero:

- la funzione esiste
- ho una firma con lo stesso numero di parametri
- i parametri hanno lo stesso tipo

Tutto ciò voglio controllarlo in Γ . Tramite lookup, ovvero $\Gamma(f)$ voglio vedere se essa sia dichiarata, ed in caso voglio vedere se sa dichiarata con firma “ $\tau_1, \dots, \tau_n \rightarrow$ ” ovvero n parametri ed con n tipi di parametri (non necessariamente tutti diversi). Con $\tau_1, \dots, \tau_n \rightarrow$ sappiamo già che in Γ ci sia una funzione con lo stesso nome e con lo stesso numero di parametri. Ora devo controllare i tipi, ovvero che e_1 abbia il tipo del primo parametro e così via. Ciò devo farlo nello stesso environment Γ . Ogni e_i deve avere lo stesso tipo τ_i , dove τ_i indica l'iesimo tipo di parametro $e_i : \tau_i \ i = 1, 2, \dots, n$

$$\frac{\Gamma' f: \tau_1, \dots, \tau_n \rightarrow \sigma \quad \Gamma' e_i : \tau_i^{i \in 1 \dots n}}{\Gamma' f(e_1, \dots, e_n) : \sigma}$$

$$\frac{\Gamma' f: \tau_1, \dots, \tau_n \rightarrow \text{notype} \quad \Gamma' e_i : \tau_i^{i \in 1 \dots n}}{\Gamma' f(e_1, \dots, e_n) : \text{notype}}$$

Nelle regole, come nella programmazione dichiarativa, se metto 2 nomi uguali, intendo dire che sono lo stesso nome. Nella regola, la n non è una variabile, ma una costante,

pertanto è fondamentale che le due n siano uguali per dire che si hanno lo stesso numero di parametri. In generale, se vediamo lo stesso nome ripetuto più volte, esso fa riferimento allo stesso valore. Se uso Γ più volte, allora l'ambiente sarà lo stesso in quanto l'ambiente Γ è lo stesso.

I tipi dei parametri attuali, devono essere gli stessi dei parametri formali. Quando facciamo la dichiarazione della funzione parliamo di parametri formali, quando richiamiamo la funzione parliamo di parametri attuali. Essendo una procedura, **non avrò un tipo di ritorno. Volendo usare una regola per la funzione, alla firma della funzione devo aggiungere un valore di ritorno.** Tutto il resto non cambia. Mi immagino che la firma diventi " $\tau_1, \dots, \tau_n \rightarrow \tau_{n+1}$ ". Andrebbe fatta la lookup di f , pertanto $\Gamma(f)$

12.4.12 Regole per l'assegnazione

Per l'assegnazione multipla **se ho n elementi a sinistra, devo avere n elementi a destra. Vedendo così va già bene, in quanto ho n elementi a destra ed n a sinistra.** Avrei potuto mettere $m=n$ come ipotesi modificando la conclusione, mettendo m id a destra ed n elementi a sinistra. La regola che deve essere applicata è che **il tipo di e_i deve essere lo stesso tipo di id_i .** Le id_i devono essere nel type environment, pertanto le regole sono:

- **sia a destra che a sinistra ci deve essere lo stesso numero di elementi**
- **devo verificare che le variabili siano dichiarate**
- **che le variabili rispettino il tipo dell'espressione in corrispondenza**

Non si può cambiare il type environment, bisogna stare nello stesso scoping.

12.4.13 Regole per If

Un if è ben tipato quando è booleano. **Non mi interessa cosa sia contenuto nel corpo. L'if non ha un valore di ritorno.** Molte volte, il tipo dell'if è dato dal tipo dell'ultima istruzione. lo stesso vale anche per le funzioni, non necessitando il return.

In generale, anche se $body_1$ non ha tipo, è importante metterlo in quanto va a controllare sia se è ben tipato in gamma e sia se ha un tipo. **Anche se $body_1$ non ha un tipo, dato che il mio obiettivo è controllare anche che esso sia ben tipato, non devo ometterlo.** Se non ha un tipo deve essere comunque ben tipato. Ad esempio, **if non ha un tipo di ritorno ma se e non è booleano allora esso non è ben tipato. Dato l'ambiente O dovrò fare controlli quindi sia su e_1 , che e_2 che $body_1$.**

$$\frac{O' e: \text{boolean} \quad O' \text{ body}_1 : \text{notype} \quad O' \text{ body}_2 : \text{notype}}{O' \text{ if } (e) \text{ then } \text{body}_1 \text{ else } \text{body}_2 : \text{notype}}$$

12.4.14 Regole per while

$$\frac{\Gamma \vdash e: \text{boolean} \quad \Gamma \vdash \text{body}: \text{notype}}{\Gamma \vdash \text{while } e \text{ do } \text{body} \text{ endwhile}: \text{notype}}$$

Affinché ciò abbia senso, devo assicurarmi che **e sia booleano nello stesso type environment. Se sono nel nodo while ed andando a vedere il tipo del figlio vedo che esso non sia booleano dò errore semantico.** In generale:

- Controllo che tutto sia apposto
- devo poi calcolare il tipo di ritorno

Se “e” è booleano e body non ha tipo, significa che il **while è ben formato ed il suo tipo è notype, ovvero non ha un tipo. L’elemento body l’ho verificato nel type scoping Γ ed è tutto apposto. Se dal type environment Γ derivo body ed ho tipo notype e lo stesso vale per e che è boolean, l’espressione di while è apposto, non avendo errori semantici e posso assegnargli un tipo. In questo caso al while gli assegno il notype**

Quando vado a vedere un tipo di un costrutto, **lo vede sempre in un type environment che è vista come una funzione. Tale elemento può essere o nella tabella locale o in una padre. Si fa una lookup(x) e restituisce il tipo.**

Con $\Gamma(\text{id})$ mi sta dicendo che sto facendo la lookup nel type environment Γ su id. **Se tale lookup mi restituisce che id è di tipo , allora il tipo di id in quel type environment è proprio**

12.5 Soundness del type system

Un sistema è sound se **ogni volta che si dimostra che e_1 è di tipo T, quando a tempo di esecuzione si va a valutare quell’espressione, effettivamente e_1 è di tipo T.**

Se ciò non fosse vero, **se nel compilatore scopro che e_1 è una stringa, andrei ad assegnare la memoria per la stringa. Facendo il calcolo per quell’espressione, a tempo di esecuzione avrei però un intero o un float, e dovrei mettere il float nello spazio di memoria di una stringa**

Un type system è sound se, **quando si va ad eseguire il programma, i tipi che avevamo previsto analizzando il codice sono proprio quelli che escono quando viene eseguito. Un**

sistema deve essere per forza sound. Perciò bisogna dimostrare effettivamente le cose. Bisogna dimostrare che $1+2$, sapendo l'operazione di somma, mi dia effettivamente un intero.

In generale, un **type system** è l'insieme delle regole di inferenza che mi definiscono i tipi. Esso deve essere sound. Se con le regole mi esce fuori che $1+2$ è intero, quando vado a fare la somma, mi deve uscire un intero. Immaginando di fare una regola sballata, la cui somma di due interi è una stringa, ciò non sarà più sound, in quanto facendo l'operazione di sound si dovrebbe codificare una stringa in una memoria destinata ad interi.

Un **type system** per essere sound significa che i tipi previsti devono essere quelli che saranno ottenuti quando si esegue il programma. Per vedere se un **type system** è sound bisogna vederlo a tempo di esecuzione o immaginando cosa succeda. Il fatto che il **type system** sia sound non è un problema di chi fa i compilatori ma chi definisce il linguaggio. Per la stessa logica, anche la grammatica deve essere sound. Pensando di voler fare una cosa, se sbaglio la grammatica potrebbe non accettare mai le frasi che voglio accetti. Chi dà le grammatiche, le regole lessicali semantiche, deve darmi delle regole che siano sound.

12.6 Costrutti complessi ed operazioni

Dobbiamo non solo capire se è ben definita ma, una volta che lo abbiamo capito, le diamo un tipo. Operazioni semplici sono semanticamente ben definite di base. Per ogni costrutto cerchiamo quindi due cose:

- vediamo che la condizione sia semanticamente ben definita
- se è semanticamente ben definita, che tipo restituisce?

Se i tipi dei figli mi rendono semanticamente corretto posso definire il tipo del costrutto complesso. Oltre al tipo, per costrutti complessi bisogna verificare che esso sia semanticamente corretto, poi dare il tipo

12.7 Regole e type environment

Se non ho il **type environment** non potrò calcolare la variabile, pertanto devo usare un simbolo alla nostra regola che specifica il **type environment** in cui siamo. Dobbiamo mettere più informazioni alle regole, come il **type environment** che mi dà il tipo per le variabili non dichiarate. Il **type environment** è una funzione, ovvero un qualcosa che prende x e restituisce il tipo di x . Per semplificare ed astrarre le tabelle dei simboli, si va a definire

una funzione che, dato il nome della variabile, restituisce il suo tipo. Sappiamo come implementare questa funzione, in quanto **va a esplorare le tabelle dei simboli partendo dalla tabella corrente**. Tale funzione la chiamiamo O oppure Γ . Essa viene messa prima dell'espressione. Il type environment specifica O il type environment prima che venga letta l'espressione della regola. La regola per l'Add diventa:

$$\frac{O\ e_1 : \text{Int} \quad O\ e_2 : \text{Int}}{O\ e_1 + e_2 : \text{Int}}$$

Essa significa: **Come faccio a calcolare il tipo di $e_1 + e_2$ Dato che il mio il type environment è O ? Vado a calcolare il tipo di e_1 nel type environment O e vado a calcolare il tipo di e_2 nello stesso environment O .** Se sia e_1 che e_2 sono interi allora la somma sarà un intero. Per capire il tipo di $e_1 + e_2$ in un certo type environment **devo andare a calcolare sia il tipo di e_1 che di e_2 nello stesso ambiente O .** In generale " $O \vdash e : T$ " si legge come, dato l'ambiente O è dimostrabile che e ha tipo T . Sotto l'assunzione che le variabili abbiano il tipo dato da O , è dimostrabile che l'espressione ha il tipo T . È dimostrabile se applicando tutte le regole del type system posso dedurre che e ha tipo T . **Il costrutto non modifica il type environment, pertanto il type environment che abbiamo in ingresso va bene per tutti gli elementi.** Se si aggiungono delle dichiarazioni cambia anche il type environment. I visitor, per ogni nodo, dovranno implementare una regola semantica del linguaggio. Il visitor avrà già a disposizione le tabelle pertanto sarà nella seconda visita, in quanto nella prima si creano i type environment e si mettono i puntatori, nella seconda si stanno usando le tabelle e si stanno applicando le regole. Per effettuare il type checking, le regole di inferenza diranno cosa fare per ogni nodo. Il type system è l'insieme di regole mentre le tabelle creano il type environment. L'implementazione della funzione O non fa altro che cercare nelle tabelle. Se la x è dichiarata più volte nelle tabelle, la nostra funzione O prende la più vicina (most closely nested value).

12.7.1 Esempio: Sottoalbero LT

Devo verificare che la frase $(45 + x) < \text{neg } 34$ sia semanticamente corretto e quale sia il tipo finale. Siccome c'è una variabile x , per capire il suo tipo dovrò avere un type environment

Quando entro nella funzione, avrò già un type environment. Prima di valutare l'espressione di lt , ho già tabelle di scope e type environment. L'esplorazione diventa:

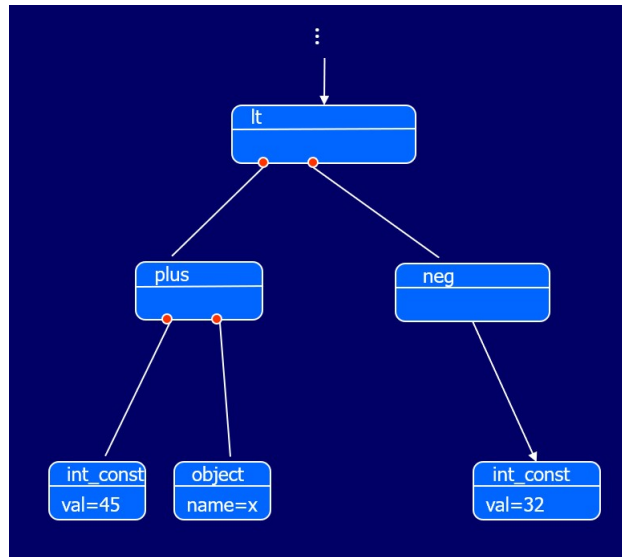


Figura 12.1: Esempio: Sottoalbero LT

- Entro in `lt` e scendo in `plus` e poi `int_const` (in quanto bottom up devo arrivare alle foglie)
- Arrivato ad `int_const`, devo avere una regola che mi associ `int_const` a qualcosa, una regola che mi dica cosa sia un `int_const`. Dato un certo type system l'`int_const` è sempre un `int`. Grazie a tale regola

$$\frac{}{\text{int_const} : \text{Int}}$$

`int_const` è di tipo intero

- Vado avanti e arrivo sulla `x`. Consulto la Γ e viene fatto un lookup del nome `x` al type environment. Se non è nella tabella risalgo. Se $\Gamma(x)$ restituisce `int`, allora nel type environment Γ `x` è di tipo `int`. La regola è

$$\frac{\Gamma(x) = \text{Int}}{\Gamma \vdash x : \text{Int}}$$

. Se l'operazione di ricerca su `x` mi restituisce `Int`, allora `x` è dichiarata di tipo `int`. $\Gamma(x)$ è un'operazione di utilizzo della funzione per capire cosa mi restituisce la funzione su `x`

- Salgo al nodo `plus` e ho bisogno della regola

$$\frac{\Gamma \vdash E1 : \text{Int} \quad \Gamma \vdash E2 : \text{Int}}{\Gamma \vdash E1 + E2 : \text{Int}}$$

per effettuare la somma. So già i tipi dei figli e sto nello stesso ambiente. Dato che non ci sono dichiarazioni, rimangono nello stesso ambiente. Se entrambi i figli hanno tipo `int`, allora anche la somma avrà tipo `int`

- Salgo in `lt` e scendo di nuovo in `neg`, scendendo fino alla foglia
- Arrivato in `int_const` restituisco `int` e salgo a `neg`
- mi serve una regola per calcolare il tipo della variabile. Per calcolare il tipo, richiama una funzione `visit()` su quel nodo. Nella funzione `visit` di quel nodo scriverò la regola. In questo caso essa è

$$\frac{\Gamma \vdash E1 : \text{Int}}{[\Gamma \vdash -E1 : \text{Int}]}$$

- Ho poi bisogno di una regola che mi dica come lavorare con `lt`, ovvero

$$\frac{\Gamma \vdash E1 : \text{Int} \quad \Gamma \vdash E2 : \text{Int}}{\Gamma \vdash E1 < E2 : \text{Bool}}$$

12.8 Regole di inferenza, visitor e tabelle

Qualcuno ci deve dire che **quando applico un operatore unario ad un intero ottengo un intero o altro. Posso implementare le regole di inferenza sotto forma di tabella, dove ogni cella può essere a sua volta scritta come regola di inferenza.** Le regole di inferenza le implementiamo nei nodi tramite `visitor()`. Per gli operatori, le regole di inferenza degli operatori potremmo implementarle tramite tabelle

Siccome il nostro linguaggio target non è il linguaggio macchina, **si potrebbe permettere il casting durante le operazioni di somma. I tipi di alcune scelte dipendono quindi dal linguaggio target.** Ogni qual volta mi trovo su `addOp` la svolgo facendo un accesso alla tabella che mi dice cosa fare (in quanto ogni cella rappresenta una regola di inferenza)

12.9 Dichiarazioni e type environment

A cambiare l'ambiente sono ad esempio le `{}`. Se ho una dichiarazione `d1` e poi un blocco `e1`, `e1` deve considerare la dichiarazione `d1`. La tabella dei simboli prima di entrare nel blocco, non è quella che devo considerare per l'istruzione, ma devo considerare un ambiente che è stato aumentato dal blocco. Dato il codice:

```
char * i="ciao"; int i; printf("%d", i);
```

Dato il type environment `O` che contiene la prima istruzione, voglio calcolare `printf`, ma essa devo calcolarla andando ad estendere `O` con la dichiarazione, avendo un ambiente `O [i/int]`, ovvero un ambiente esteso con la dichiarazione di `i` come `int`.

Il blocco sarà quindi ben tipato se l'istruzione è ben tipata in O esteso con l'intero i . Non avrò quindi lo stesso type environment. A livello di regola si ha:

$$\frac{\Gamma[d_1] \vdash e_1 : T}{\Gamma \vdash d_1 e_1 : \text{notype}}$$

Non posso calcolare il tipo di e_1 in Γ , altrimenti non è servita a niente la dichiarazione. Devo quindi prima **estendere Γ con d_1 e poi vado a vedere se e_1 è ben tipato nel nuovo ambiente. Se e_1 è ben tipato con l'ambiente esteso, allora $d_1 e_1$ è ben tipato sull'ambiente non esteso.** Se nel costrutto ci sono dichiarazioni, **quello che segue le dichiarazioni non può essere usato senza considerare la dichiarazione. Da notare che senza considerare d_1 , e_1 non sarebbe ben tipato in Γ** , in quanto nella print si aspetta che la i sia intera e senza considerare la int , i sarebbe stata una stringa.

Per e_1 vado ad estendere l'ambiente originario prima di entrare nell'istruzione con le nuove dichiarazioni. Se e_1 è ben tipato nell'ambiente esteso, allora è ben tipato anche il padre. A livello di albero, ho la tabella dei simboli che scende nel blocco, e questo blocco ha le dichiarazioni. Estendo la tabella dei simboli con le dichiarazioni e tale tabella estesa lo passo

12.10 Riassunto: Regole di inferenza

Il nodo $e_1 + e_2$ è rappresentato da **AddOp**. Se i suoi figli sono interi posso etichettarlo come intero. Questo perché se e_1 ed e_2 sono costanti intere, posso etichettare quel nodi di **tipo intero**. Passo poi al padre che sarà la somma di due costanti. Sulle due costanti è facile capire il tipo in quanto **abbiamo una regola che "se i è un integer, i : Int"**. Da notare che **integrer è il concetto in lingua naturale mentre Int è il nome che abbiamo dato all'interno del nostro compilatore**. Sulla costante applichiamo la regola.

Vado a calcolare il tipo di **AddOp**. Vado a vedere il tipo dei figli: $e_1 + e_2$ è il mio nodo **AddOp**. **Conoscendo sicuramente i figli e se i due figli sono entrambi int, tale regola mi dice il tipo che devo dare ad AddOp**. Tale regola non è banale in quanto è essa che decide il tipo. A sua volta questo **AddOp** sarà un'espressione che ha tipo **Int** e che potrà essere usata in altre occasioni. Data la regola

$$\frac{O e_1 : \text{Int} \quad O e_2 : \text{Int}}{O e_1 + e_2 : \text{Int}}$$

Essa può essere letta come: **Se è possibile dimostrare che il figlio sinistro è di tipo intero in O e il figlio destro è di tipo intero in O , allora $e_1 + e_2$ è ben tipata ed essendo ben tipata**

ha tipo Intero. Tale regola mi dice sia che quel nodo è ben tipato sia il tipo. Una volta che controllo che tutto sia ben tipato, devo anche calcolare il tipo di ritorno

12.11 Dichiarazioni ed aumento del Γ

Supponiamo di aver fatto un body che ha una parte dichiarativa di una variabile di tipo e e poi una parte di istruzioni, di statements. Esempio di codice è: `int x = 1 float x = 2.3; printf(x);`

Alla x nelle $\{\}$ abbiamo fatto una lookup sulla tabella che ci dice che x è un float, figlia della tabella che dice che x è un int. Prima di entrare in $\{\}$ sarà Γ esteso con float x , scritto come $\Gamma[x/\text{float}]$. Quando esco fuori dal blocco tolgo l'estensione ed ottengo Γ della tabella padre in cui x è definito come int. Se non avessi esteso Γ , entrando nel blocco mi sarei perso la definizione int x . Il codice nelle $\{\}$ lo vado a calcolare nell'ambiente esteso.

Se all'inizio del blocco mi trovo in un ambiente Γ , e questo blocco dichiara id di tipo e , allora lo statement deve essere ben tipato non rispetto a Γ ma Γ esteso. Nella parte del denominatore della frazione deve esserci il nostro linguaggio. Dobbiamo verificare, arrivando a quel nodo, se sia tutto apposto e se lo è, di che tipo sia quel nodo, tutto ciò conservando il corretto type environment. Quando entro nel corpo dell'if, devo considerare anche le dichiarazioni dell'if. **Non devo usare lo stesso ambiente di prima di entrare nell'if, ma devo estenderlo. La regola di inferenza è una formulazione matematica di ciò che facciamo sempre.** Dato il lessico, la grammatica e regole di inferenza abbiamo il linguaggio. Formalizzare è sempre meglio in quanto elimina le ambiguità. In alcuni casi, però, diventa complicato e difficile andare a definire le regole di inferenza

13.1 Introduzione: codice intermedio

Fino ad ora abbiamo studiato l'analisi lessicale, il parser e l'analisi semantica. Quello che dobbiamo fare è **generare il codice intermedio**. Dopo l'analisi semantica abbiamo il nostro **albero** e siamo sicuri che durante l'analisi semantica è tutto corretto.

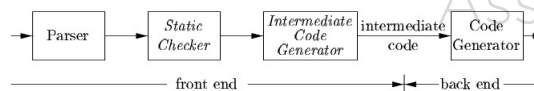


Figura 13.1: Introduzione: codice intermedio

Il codice intermedio posso generarlo:

- o con una **struttura interna come gli alberi**
- o lo posso **stampare in un file**.

L'approccio LLVM è quello di **costruire tramite api un albero** e alla fine esce un **formato del codice intermedio su cui possiamo fare un dump**, potendolo stampare. Esso è un linguaggio a sé stante, simile all'Assembler **Subito dopo la generazione del codice intermedio c'è l'ottimizzazione**. Il codice intermedio viene generato in modo automatico e dopo con l'ottimizzazione questo codice viene ridimensionato di parecchio ed è pronto per generare il **codice macchina**. L'ottimizzazione in realtà viene fatta quasi a tutti i livelli. Il **generatore**

di codice cerca di generare già codice ottimizzato. Più ci avviciniamo alla fine del backend più ci avviciniamo alla macchina

13.2 Forme intermedie di basso ed alto livello

Dal sorgente, abbiamo diverse trasformazioni del codice intermedio, ed ognuna di esse è una forma intermedia. Esistono forme intermedie di alto livello e di basso livello. L'albero sintattico, è una forma intermedia anch'essa ma viene dichiarata di alto livello. Già sull'albero si possono fare delle ottimizzazioni. Dal codice sorgente possiamo avere rappresentazioni intermedie di alto livello che rappresentazione intermedie più base, non ancora ottimizzate. In generale, sembrerebbe che ci sia una sola forma intermedia di basso livello. Nei compilatori correnti, essendo che essi arrivano a a GPU e a vari processori specializzati, le forme intermedie sono diverse. Si parla di lowering. Alcune ottimizzazioni si riescono a fare su alcune forme intermedie e non su altre. Nel nostro progetto stiamo usando C come linguaggio intermedio. Generando C, possiamo usare Clang che genera LLVM IR che può essere lanciato poi su qualsiasi architettura.

13.3 Da AST a DAG

Gli alberi sintattici potrebbero essere messi in una forma a DAG. Quando ci sono espressioni uguali, è inutile creare lo stesso codice due volte. Una volta che abbiamo capito come si fa un'operazione, come $(b-c)$, faccio in modo che ogni volta che accedo ad un nome, inserisco un riferimento. Si avranno alcune linee di codice per saltare all'allocazione che gli serve. Ciò significa che posso già ottimizzare a livello di AST

13.3.1 Esempio: Da AST a DAG

Data la frase $a + a * (b-c) + (b-c) * d$, voglio scrivere l'albero per questa espressione e poi il DAG, rispettando le precedenze. Per poter disegnare l'albero andiamo a capire le precedenze delle operazioni, diventando: $\{a + [a * (b-c)] + [(b-c) * d]\}$

Una visita di questo DAG non avrà nessuna differenza rispetto a prima, in quanto i nodi ci sono. Non fa differenza di quale nodo duplicato si cancella. Quello che succede, però, è che il mio visitor visita meno nodi. Le operazioni che faccio dovranno però essere equivalenti. Ragioniamo prima con le graffe e daranno già una struttura gerarchica.

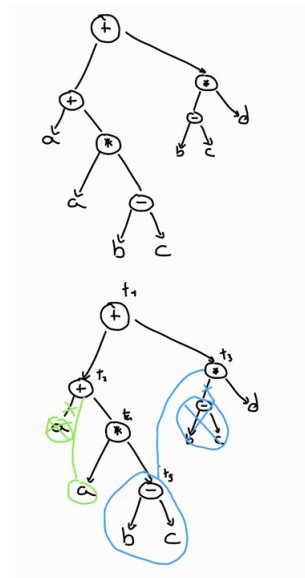


Figura 13.2: Esempio: Da AST a DAG

13.4 Codice intermedio ed ottimizzazione

L'ottimizzazione viene fatta ogni qual volta c'è possibilità di farla. **Passando al codice a basso livello, esso si chiama codice a tre indirizzi in quanto esso è un codice intermedio in quanto ha un'operatore binario ed un'assegnazione al massimo.** L'indirizzo fa riferimento al nome di una variabile. Sono arrivato all'analisi lessicale, sintattica e semantica avendo un albero, nonché un riferimento alla tabella dei simboli e quindi **l' x in una rappresentazione interna esso è un puntatore alla tabella dei simboli. Tutti i nomi che vediamo nel codice a 3 indirizzi fanno riferimento a tabelle dei simboli**

Data l'operazione $x+y*z$, **per rendere l'operazione semplificata sono costretto ad aggiungere dei nomi t_i .** Esse sono variabili temporanee che vengono generate dal compilatore e che finiranno nei registri. A livello di codice intermedio non mi interessa. **Tali variabili dovranno essere inserite nella tabella dei simboli. Essi non sono dei nomi inventati da noi ma dal compilatore per poter fare la riduzione di $x+y*z$ in 2 istruzioni a 3 indirizzi.**

Nell'ottimizzazione molte delle t_i scompaiono e a livello di codice macchina andranno ad usare la stessa cella di memoria in quanto non andranno in conflitto, pertanto quella cella di memoria potrà gestire casi differenti, per cui invece di mettere un registro per ogni variabile temporanea, cerca di ottimizzare il numero di registri.

13.4.1 Esempio: Da DAG a codice a 3 indirizzi

Per farla diventare codice 3 indirizzi, vado a mettere una temporanea t_i su ogni nodo di processo. Faccio una visita bottom up e inizio a scrivere il codice:

- $t_5 = b - c$
- $t_4 = a * t_5$
- $t_2 = a + t_4$
- $t_3 = t_5 * d$
- $t_1 = t_2 + t_3$

Tale codice sarà equivalente al DAG. Il calcolo di $b-c$ è unico, infatti si è andati a riutilizzare t_5 .

13.5 Da codice a tabella di quadruple

La visita la faremo tramite un visitor che ogni volta che incontra un nodo si inventa una temporanea e genera un codice. All'interno del compilatore, il nostro DAG diventerà un array di quadruple: operatore, primo e secondo argomento ed infine il risultato. All'interno della cella ci sarà non una stringa ma un riferimento alla variabile

	op	arg ₁	arg ₂	result
$t_1 = \text{minus } c$	0	minus	c	t ₁
$t_2 = b * t_1$	1	*	b t ₁	t ₂
$t_3 = \text{minus } c$	2	minus	c	t ₃
$t_4 = b * t_3$	3	*	b t ₃	t ₄
$t_5 = t_2 + t_4$	4	+	t ₂ t ₄	t ₅
$a = t_5$	5	=	t ₅	a
			...	

Figura 13.3: Da codice a tabella di quadruple

13.6 Tabella di triple

Invece di mettere le quadruple, si potrebbero utilizzare le triple. Si eliminano le colonne dei risultati. Al posto delle variabili temporanee, si metterà la regola che calcola quella variabile. Non saprò più t_1 quanto vale ma saprò la riga in cui essa viene calcolata. Il vantaggio di tale approccio è che abbiamo ridotto il codice a tre indirizzi. In generale però, l'ottimizzazione è pericolosa perché può cambiare il significato del programma, rendendolo non equivalente con quello dato. Sviluppando in C, non vogliamo perdere tempo nella compilazione ma vedere subito il risultato. Quando il codice va venduto, esso

	op	arg ₁	arg ₂
0	minus	c	
1	*	b	(0)
2	minus	c	
3	*	b	(2)
4	+	(1)	(3)
5	=	a	(4)
	...		

Figura 13.4: Tabella di triple

deve essere ottimizzato. Sono sicuro che funzioni ma può succedere che se facciamo ottimizzazioni spinte, potrebbe non comportarsi allo stesso modo di come si comportava prima. Una buona tecnica di ingegneria del software è fare testing quando sviluppi il software e quando faccio una modifica si ritesta

In generale, l'ottimizzazione potrebbe cambiare l'ordine delle quadruple. Se ciò succede, ho dei problemi. (0) non farà più all'istruzione che volevo. Infatti, (0) è posizionale. Se la riga 0 la metto in una cella precedente e sto usando le t_i non avrò problemi in quanto saprò quanto vale t_i . Usando invece (0), essa punterà ad un altro valore. dovrò cambiare tutte le istruzioni che facevano riferimento alla cella (0).

13.7 Triple indirette

Si è passati alle triple indirette, indipendenti dall'ordine. Essendo che il valore di (0) è ester-

instruction	op	arg ₁	arg ₂
35 (0)	minus	c	
36 (1)	*	b	(0)
37 (2)	minus	c	
38 (3)	*	b	(2)
39 (4)	+	(1)	(3)
40 (5)	=	a	(4)
...			

Figura 13.5: Triple indirette

no alla tabella, posso spostare la riga dove voglio ma non devo cambiare nulla, in quanto la posizione della 0 sarà la stessa. In generale, indipendentemente dall'implementazione, l'AST diventa un array.

13.8 Single Assignment Form

Tale forma non è da sottovalutare in quanto perfetta per le ottimizzazioni ed è adottata anche dall'LLVM.

In questo codice, non è possibile che la stessa variabile venga assegnata 2 volte. Si parte da una forma normale e se la stessa variabile viene assegnata una volta, la seconda volta

$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

(a) Three-address code. (b) Static single-assignment form.

Figura 13.6: Single Assignment Form

si va a cambiare nome alla variabile: $p = a + b$ diventa p_1 e quando cerco di assegnare a p un'altra volta, essa diventa un'altra variabile, ovvero p_2 .

Dal momento che riassegno p , la p iniziale l'avrò persa. Logicamente, non perdo nulla in quanto vado a collegare l'uso di una p con la sua ultima rinominazione. Ogni variabile viene assegnata una sola volta e devo fare uno studio, devo cercare di capire la p quale nome debba avere. Ogni volta che abbiamo una variabile e stiamo cercando di assegnarla (dopo averla già assegnata), le cambiamo nome

13.9 Problemi del Single Assignment Form

Dato il codice `if(flag) x = -1; else x = -1; y = x * a`, esso è un problema. Le due x dovranno diventare x_1 ed x_2 ma così facendo non so quale dovrà essere moltiplicata per a . Per risolvere il problema, si inventa una funzione $\phi(x_1, x_2)$ dinamica che restituisce il forward durante l'esecuzione. In base al percorso che abbiamo seguito restituisce una x_i piuttosto che l'altra. Avremo x_1 se il flusso passa per la parte vera della condizione, x_2 se passa nell'else. La funzione ϕ restituisce quindi il valore del suo argomento che corrisponde al sentiero seguito dal flusso di esecuzione. è un modo di programmare in funzione di come il codice viene eseguito. Usa un meccanismo di scrittura del codice in funzione di quale flusso stiamo eseguendo

13.10 Specifica del codice a tre indirizzi

Il codice a tre indirizzi ha una specifica, permettendo operazioni tipo $t_i = x \text{ op } y$ oppure $t_i = \text{op } x$. In generale, esso è composto da:

- **nomi:** esso è sostituito dal puntatore alla tabella dei simboli
- **costanti:**
- **temporanee:**

Possiamo usare i goto, gli if e gli ifFalse. Possiamo mettere o una singola x o un'espressione condizionale. La chiamata a funzione si può fare anche a livello di codice intermedio. Devo prima fissare i parametri e poi fare la chiamata con i parametri. Avremo poi anche * e & per i puntatori e operatori come y[i]. In tal caso, si usa y come indirizzo base ed i il numero di cella. In generale, si scrive prima a codice a tre indirizzi con delle label e poi si passa alle celle di memoria.

13.10.1 Esempio: Codice a tre indirizzi

Dato il codice `do i = i + 1; while (a[i] < v);` farlo diventare codice a 3 indirizzi. La prima cosa da fare è vedere come funziona il `do while`. si fa prima il corpo e poi si verifica se la condizione è vera. Se è vera, si salta all'inizio del corpo.

<pre> L: t₁ = i + 1 i = t₁ t₂ = i * 8 t₃ = a [t₂] if t₃ < v goto L </pre> <p>(a) Symbolic labels.</p>	<pre> 100: t₁ = i + 1 101: i = t₁ 102: t₂ = i * 8 103: t₃ = a [t₂] 104: if t₃ < v goto 100 </pre> <p>(b) Position numbers.</p>
--	---

Figura 13.7: Esempio: Codice a tre indirizzi

Molte volte, il codice a 3 indirizzi è generato in automatico, pertanto potrebbe fare cose assurde. A tal fine, l'ottimizzazione e la generazione del codice macchina si preoccupano di usare quanta meno memoria possibile. In tal caso, siccome `a[]` è un array di reali di 8 byte, si moltiplica `i*8` per avere l'indirizzo reale di ogni elemento `i`. Volendo prendere l'`i`esimo elemento siccome la `i` si muove a singolo byte, per `a[i]` bisogna muoversi ad `a[i*8]`. Tramite `t2` e `t3` riesco a beccare il valore di `a[i]`. Se `t3 < v` allora devo fare un goto ad `L`, se falso esco fuori. Tale codice era caratterizzato dalle label. Una volta fatto con la label, vado a piazzare il codice nell'array partendo ad esempio da 100 e la `L` diventa 100. Tale versione è meno astratta e più reale, in quanto si utilizzano celle ed indirizzi dell'array.

13.11 Generazione del codice intermedio partendo dalla grammatica

La forma algoritmica per generare il codice intermedio è quello delle grammatiche ad attributi. In un sistema reale se noi dovessimo generare codice intermedio, faremmo la visita dell'albero. Ogni nodo interno di un'espressione è una variabile temporanea.

PRODUCTION	SEMANTIC RULES
$S \rightarrow \text{id} = E ;$	$S.code = E.code \parallel$ $gen(top.get(\text{id}, lezeme)) ' = ' E.addr$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel E_2.code \parallel$ $gen(E.addr ' + ' E_1.addr ' + ' E_2.addr)$
$ - E_1$	$E.addr = \text{new Temp}()$ $E.code = E_1.code \parallel$ $gen(E.addr ' - ' E_1.addr)$
$ (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$ \text{id}$	$E.addr = top.get(\text{id}, lezeme)$ $E.code = ' '$

Figura 13.8: Generazione del codice intermedio partendo dalla grammatica

Tale grammatica **genera un codice intermedio per le espressioni aritmetiche fino all'assegnazione**. Se ho $x = a + b + - c$, tale frase è generata dalla grammatica. Il codice intermedio che vorrei generare per tale frase è:

- $t_1 = a + b$
- $t_2 = -c$
- $t_3 = t_1 + t_2$
- $x = t_3$

L'ultima è un'istruzione di copia in quanto **si copia una variabile in un'altra. Tutte le variabili sono indirizzi a tabelle dei simboli, perciò esso viene chiamata codice a 3 indirizzi**. Dobbiamo ora scrivere una grammatica ad attributi che mi crei questo codice a tre indirizzi.

13.11.1 Esempio: Considerazioni sulla grammatica del codice intermedio

La grammatica è riportata in foto. Essa usa definizione guidate dalle sintassi in quanto **lo schema di traduzione usa le graffe, mentre in questo caso non le usiamo**. Per calcolare se la grammatica sia S o L attributa vado a vado gli attributi per i non terminali. Essi sono:

- Per S code. **In code andiamo a cumulare il codice a tre indirizzi che man mano generiamo**
- Per E sia code che addr. **Code contiene il codice del sottoalbero mentre in addr andiamo a mettere l'indirizzo della variabile che contiene l'ultimo risultato**. Per un sottoalbero abbiamo un'espressione che ha generato del codice. Tale codice avrà il risultato in una certa variabile. **L'attributo addr è l'indirizzo a tale variabile contenente il risultato dell'espressione**.

Tutti gli attributi sono sintetizzati, pertanto **la grammatica è S attribuita**. Siccome ho una definizione guidata dalla sintassi, non ho specificato quando eseguire le regole, ma le ho solo associate. Essendo che la grammatica è S attribuita, **è facile fare lo schema di traduzione. Affinché S.code venga calcolata, devo aver visto tutti gli attributi dei figli. Se non ho visto i figli, non saprei come calcolare S.code**. Tutte le regole posso metterle dunque alla fine. Ciò **mi fa diventare la grammatica in postfissa**. Per poter eseguire la grammatica, devo eseguire le azioni quando vedo la parte destra della produzione

13.11.2 Esempio: Regole per il calcolo degli attributi

La funzione `new Temp()` genera una **variabile temporanea**. Per `addr`, se ho 7 istruzioni che mi rappresentano un'espressione, tali istruzioni avranno un risultato e tale risultato sarà in una variabile. L'attributo `addr` sarà l'indirizzo della variabile che mi mantiene il risultato. Data la frase: $x = a + b + -c$, posso fare l'albero:

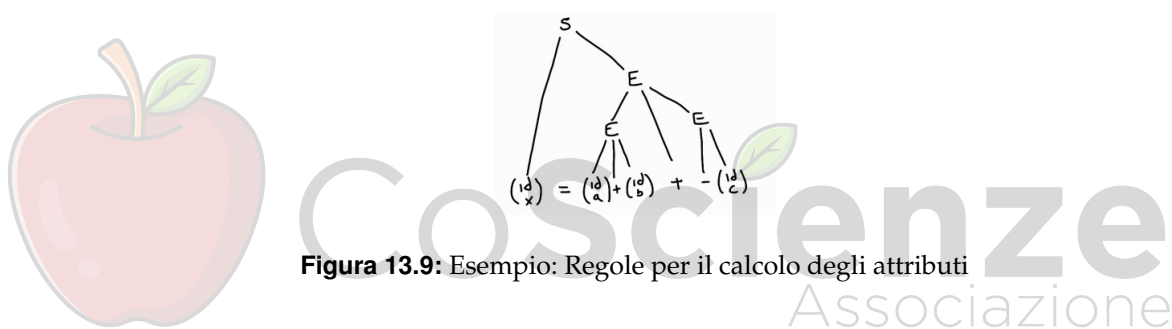


Figura 13.9: Esempio: Regole per il calcolo degli attributi

Per le riduzioni, per il **bottom up** avremo uno **stack semantico** ed uno **stack sintattico**. A livello di passi:

- La prima cosa che vado a **ridurre** è **"a"** (dopo aver fatto shift sia su `x` che su `=`). Avendo ridotto, **posso seguire la regola**. In `E` ho bisogno di calcolare sia `addr` che `code`
- **Shift** poi su `+` e leggo l'id **"b"**. Faccio lo stesso che ho fatto per `a`
- Quando **riduco** `E + E` in `E`, **creo una temporanea** t_1 e la salvo in `addr`, mentre il **codice** è dato dalla concatenazione del `code` di E_1 , E_2 e da $E_1.addr + E_2.addr$. Essendo che E_1 ed E_2 hanno `code` vuoto, avrò solo $E.code = E_1.addr + E_2.addr$
- Faccio poi lo **shift** su `-` e leggo l'id **"c"**. Faccio la **reduce** su `c`
- Faccio poi la **reduce** di `-E` in `E`. **Creo anche in questo caso una new Temp()** e come **code** avrò $- E_1.addr$

- Riduco poi $E_1 + E_2$ in E . Stavolta sia E_1 che E_2 hanno un contenuto in code pertanto concateno $E_1.code$, $E_2.code$ ed $E.code = E_1.addr + E_2.addr$. L'espressione mantiene traccia dell'indirizzo della variabile che avrà il risultato finale dell'intera espressione
- Quando vado in S , dopo aver visto $id = E$, devo eseguire il codice. Il codice di S mi sarà dato dal codice di E e poi bisogna generare una nuova istruzione. Bisogna prendere il lessema dell'id ed assegnargli l'addr di E che contiene il riferimento al valore dell'espressione

Man mano che visito l'albero bottom up, quando riduco eseguo l'azione corrispondente a quella produzione A livello di stack semantico e sintattico avremo:

LIVELLO	STACK	STACK SEMANTICO
10	S	$S.code = t_1 = a + b \parallel t_2 = -c \parallel t_3 = t_1 + t_2 \parallel x = t_3$
9	$E(5) + E(8)$ E	$E.addr = t_3, E.code = t_1 = a + b \parallel t_2 = -c \parallel t_3 = t_1 + t_2$
8	$-E(7)$ E	$E.addr = t_2, E.code = t_2 = -c$
7	id c E	$E.addr = c, E.code = vuoto$
6	-	
5	$E(2) + E(4)$ E	$E.addr = t_1, E.code = t_1 = a + b$
4	id b E	$E.addr = b, E.code = vuoto$
3	+	
2	id a E	$E.addr = a, E.code = vuoto$
1	=	
0	id x	

In rosso sono riportate le riduzioni e $E(2)$ indica che si fa riferimento alla E riportata nella riga con indice 2

13.12 Traduzione incrementale o Generazione di codice incrementale

Lo svantaggio di portarmi dietro sempre il codice è che il codice potrebbe essere lunghissimo, pertanto l'attributo code potrebbe diventare impraticabile. Un'altra soluzione potrebbe essere quella di scrivere una grammatica per cui non ho l'attributo code e man mano che trovo istruzioni da inserire in code, le vado a stampare in un file globale. Man mano che visito l'albero, invece di accumulare nei nodi i codici parziali, scrivo su di un file

globale, così che alla fine della visita dell'albero la radice non avrà il codice ma il nostro codice sì. In una grammatica simile, l'attributo `addr` rimane ma abbiamo una funzione `gen()` che genera il codice e manda in stampa. Si potrebbe anche generare un array di quadruple, triple o triple indirette. La funzione `gen()` è quindi una funzione generica, potendo aggiungere le istruzioni ad un array

13.13 Istruzioni controllo di flusso

Tutto ciò che abbiamo visto prima, lo faremmo con un visitor e non sulla grammatica. Albero di derivazione e sintattico sono equivalenti solo che il sintattico è sintetico e viene creato ad hoc, mentre sull'albero di derivazione bisogna lavorarci virtualmente (in quanto non esiste) mentre facciamo le riduzioni.

L'istruzioni che abbiamo, come `if`, `else`, `while`... sono istruzioni che modificano il control flow, in quanto non è più sequenziale ma vado a saltare un pezzo di codice. Anche il `while` mi permette di scrivere un codice più volte, pertanto il controllo del flusso viene gestito allo stesso modo. Tutte queste istruzioni fanno salti condizionati

13.14 Espressioni booleane

Le espressioni booleane hanno due funzioni:

- espressione vera e propria. Ho un'assegnazione e voglio calcolarne il valore
- controllo di flusso

Esse però vengono fatte e gestite allo stesso modo. Le espressioni booleane hanno una grammatica semplice, tipo:

- $B \rightarrow B \mid B$
- $B \rightarrow B \ \&\& \ B$
- $B \rightarrow ! \ B$
- $B \rightarrow (B)$
- $B \rightarrow E \ \text{rel} \ E$
- $B \rightarrow \text{true}$

- B -> false

Un'espressione booleana può essere anche una condizione fra due espressioni, come $5 > 4$. Sulla base delle ultime due, ovvero gli atomi delle espressioni booleane, si compongono le altre

13.15 Codice short circuit

Dovendo generare il codice per "if(x < 100 || (x > 200 && x !=y)) {x = 0;}":

- Se $x < 100$, non ha senso controllare ciò che viene dopo l'OR in quanto la condizione è già vera. Essendo che appena riesco a capire quanto vale l'espressione esco fuori, non ho bisogno di controllare altro. Ciò viene definito come short circuit, in quanto salto del codice. Problema dello short circuit è il side effect, ovvero se nel pezzo di codice che si salta si andrebbero a fare delle assegnazioni
- Se $x < 100$ però è falsa, il prossimo controllo che devo fare è $x > 200$.
- Se $x > 200$ devo controllare anche l'espressione dopo l'AND, mentre se $x > 200$ è falsa, allora sicuramente tutta l'espressione sarà falsa.

Devo scrivere il programma in codice a tre indirizzi in modo che venga implementato lo short circuit. Il codice è:

- if x < 100 goto L₂
- ifFalse x > 200 goto L₁
- ifFalse x !=y goto L₁
- L₂: x = 0
- L₁: //altro codice

Uscire fuori dall'if, non vuol dire necessariamente andare all'istruzione subito dopo. Immaginando che l'if sia l'unica istruzione all'interno del while, dopo aver finito l'if devo tornare a controllare la condizione del while. Uscire fuori da un'istruzione non è detto che ci porta ad un'istruzione più in basso, ma ci porta alla prossima istruzione, che essa sia prima o dopo dell'if stesso. In generale, si potrebbero avere delle istruzioni con side effect. In quel caso, non si vorrebbe lo short circuit in quanto andrebbe a saltare l'assegnazione. Dato ad esempio il codice "if (ptr == null || ptr.data = 5)", con side effect, se il ptr non è null allora io vorrei assegnare il valore del puntatore a 5, ma con short circuit non lo andrei a fare.

13.16 Regole per l'uso delle espressioni booleane

La grammatica per le espressioni booleane procede con:

- $S \rightarrow \text{if } (B) S_1$
- $S \rightarrow \text{if } (B) S_1 \text{ else } S_2$
- $S \rightarrow \text{while } (B) S_1$

Anche qui bisognerà scrivere delle regole semantiche per fare in modo che esca il codice. La logica dietro questi costrutti è:

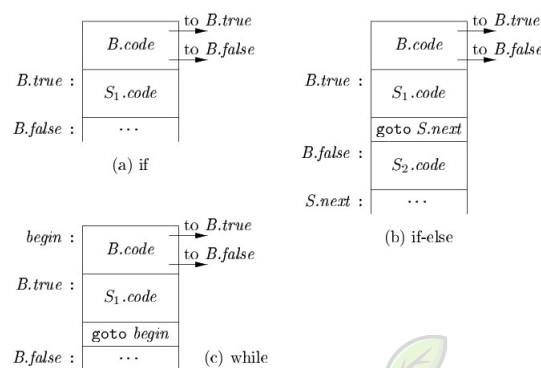


Figura 13.10: Regole per l'uso delle espressioni booleane

13.17 Tipi di goto

I goto possiamo classificarli come:

- quelli che **indicano se l'espressione è vera, quelli che vogliono saltare al codice per cui l'espressione è vera**
- quelli che **indicano se l'espressione è falsa, quelli che vogliono saltare quando l'espressione è falsa**

Dato il codice:

- $\text{if } x < 100 \text{ goto } L_2$
- $\text{ifFalse } x > 200 \text{ goto } L_1$
- $\text{ifFalse } x \neq y \text{ goto } L_1$
- $L_2: x = 0$

- L_1 : //altro codice

Il primo goto indica che l'espressione è vera mentre gli altri due indicano che l'espressione è falsa. Se avessi un array di quadruple, supponendo che la prima istruzione sia all'indirizzo 100, **potrei dire che l'indirizzo 100 dell'array, ha un goto che vuole andare a vero, ovvero vuole andare a quelle istruzioni nel caso in cui l'espressione è vera.** Il goto a 101, è un'istruzione che ha un goto che vuole andare a falso, ovvero vuole andare al pezzo di codice L_1 perchè ha capito che l'espressione è falsa. Lo stesso per 102. Avrò gli indirizzi dei goto. Potrei quindi fare true list ed una false list. Partendo dal codice, **potrei dire che la label true è L_2 mentre L_1 sarà false.** Le label sono sempre due: una dove vanno tutti i goto a true, un'altra dove vanno i goto a falso. Sono due approcci diversi.

B.code contiene il codice per l'espressione B. In B.true ho la label true e B.false ho la label dove vado a falso. B.false potrà essere anche più in alto. Per l'if else avrò una sequenza di goto, dovendo inserire anche l'istruzione "goto S.next". **Se vero esegue gli statement veri ma salta l'istruzione per falso. Se falsa, va direttamente all'istruzione per falso ed esce fuori.** Se viene omesso "goto S.next", quello che succede è che se la condizione è vera, dopo aver eseguito il codice della condizione vera, eseguo anche il codice della condizione falsa. Ci dobbiamo preoccupare che quando seguiamo le istruzioni per true, non eseguiamo quelle per false. Per il while, invece, inseriamo il codice per B e poi per S_1 . Dopo aver visto S_1 , devo riniziare il ciclo, saltando alla condizione. Se la condizione del ciclo è falsa esco dal ciclo, non necessariamente dopo

13.18 Introduzione alle true e false list

Nella true list ci metto le istruzioni che hanno i goto che vogliono andare a vero, mentre nella false list dei goto che vogliono andare a false. Lo posso già pensare come attributo. Avrei 100 nella true list e 101 e 102 nella false list. **Nel caso in cui si hanno le label, avrò B.true che è la label che vuole andare a vero e B.false che vuole andare a false.**

In generale, ho due codici: uno che calcola il codice con le label, ed uno che calcola lo stesso codice con gli indirizzi. Con quest'ultimo esce fuori una grammatica bottom up. Quella con le label verrebbe invece top down. Con le liste mi conservo gli indirizzi dei goto che vanno a vero e falso, con le label vado a segnare solo le label che vanno a vero e vanno a falso. Questo secondo approccio non verrà utilizzato in pratica. Se abbiamo label avremo gli attributi true e false, se useremo gli indirizzi avremo trueList e falseList.

13.19 Problema di goto ed indirizzi

Facendo l'approccio con le label, siamo costretti a fare due passi: nel primo scriviamo il codice con la label, con il secondo levo le label e metto gli indirizzi.

Se faccio direttamente con gli indirizzi, potrei avere un problema. Il problema è che quando devo scrivere un goto, non so il numero (l'indirizzo) da metterci. Non posso mettere un goto con un numero, in quanto tale numero lo saprò dopo aver scritto il codice. Nel caso della Label, mi invento la label e quando arrivo all'istruzione da eseguire metto la stessa label che avevo messo prima. Volendo usare direttamente gli indirizzi non posso obbligare un indirizzo

13.20 Backpatching

Al fine di risolvere il problema degli indirizzi, applico la tecnica di backpatching. Tale espressione mi conservo due liste:

- quelle dei goto che vogliono andare a vero ma non so dove andranno
- quelle dei goto che vogliono andare a falso

L'indirizzo del goto vuoto me lo salvo nella true list se il goto vuole andare a vero, nella false list se vuole andare a falso. In entrambi i casi, il goto non deve ancora avere l'indirizzo di dove andare. In buona sostanza, le istruzioni me le scrivo ma al contempo me le conservo. Quando arriverò all'istruzione da eseguire, saprò l'indirizzo dell'istruzione che dovrà essere eseguita nel caso della condizione vera. In tal caso, vado nelle true list e vado a mettere quell'indirizzo a tutte le istruzioni che sono nella true list, in quanto le istruzioni in questa lista hanno goto vuoto ma volevano andare all'istruzione vera.

Vado a fare un backpatch. Torno indietro in modo selettivo e vado a mettere l'indirizzo. Quando scrivo il codice per cui dovevo andare in caso di condizione falsa, allora faccio un backpatch sulla false list. Il backpatching mi permette di fare un'unica passata per scrivere il codice. Quando conosco il futuro e capisco quali siano gli indirizzi, salto all'istruzione e me le vado a riprendere. Nel caso ci siano più goto nella true o false list, l'indirizzo dell'istruzione che stiamo analizzando verrà messa a tutti i goto nella true. In generale, con il backpatch non si genera codice ma si va solo a mettere a posto i goto che sono vuoti

13.21 Backpatch espressioni booleane

Nelle espressioni booleane, possiamo aggiungere il backpatch. Si è aggiunto un non terminale M che mi conserverà l'indirizzo della prima tupla di B_2 . Dato ad esempio $B_1 \parallel B_2$, se B_1 è vera potevo finire, mentre se B_1 era falsa dovevo saltare a B_2 . Anche in questo caso, non so dove si troverà l'indirizzo iniziale di B_2 , potrebbe essere ovunque. Pertanto utilizzo il non terminale M per fare ciò. Mi porto dietro codici di espressioni booleane con goto vuoti e quando incontro le istruzioni da eseguire, saprò come riempire i goto. If saprà almeno i true list, mentre if else saprà sia true list che false list, in quanto sa cosa deve fare in caso di condizione rispettata e non rispettata

Data la grammatica per le espressioni booleane di prima a cui è stata aggiunta $M \rightarrow \epsilon$, le regole semantiche sono:

- 1) $B \rightarrow B_1 \parallel M B_2$ { $backpatch(B_1.falselist, M.instr);$
 $B.truelist = merge(B_1.truelist, B_2.truelist);$
 $B.falselist = B_2.falselist; \}$
- 2) $B \rightarrow B_1 \&\& M B_2$ { $backpatch(B_1.truelist, M.instr);$
 $B.truelist = B_2.truelist;$
 $B.falselist = merge(B_1.falselist, B_2.falselist); \}$
- 3) $B \rightarrow ! B_1$ { $B.truelist = B_1.falselist;$
 $B.falselist = B_1.truelist; \}$
- 4) $B \rightarrow (B_1)$ { $B.truelist = B_1.truelist;$
 $B.falselist = B_1.falselist; \}$
- 5) $B \rightarrow E_1 \text{ rel } E_2$ { $B.truelist = makelist(nextinstr);$
 $B.falselist = makelist(nextinstr + 1);$
 $gen('if' E_1.addr \text{ rel } E_2.addr 'goto -');$
 $gen('goto -');$ }
- 6) $B \rightarrow \text{true}$ { $B.truelist = makelist(nextinstr);$
 $gen('goto -');$ }
- 7) $B \rightarrow \text{false}$ { $B.falselist = makelist(nextinstr);$
 $gen('goto -');$ }
- 8) $M \rightarrow \epsilon$ { $M.instr = nextinstr; \}$

Figura 13.11: Backpatch espressioni booleane

Prendendo la prima produzione, per B_1 , B_2 o B ci saranno tanti goto che vogliono andare a vero e tanti che vogliono andare a false. Tutti i goto che vogliono saltare quando B_1 è falsa, dovranno saltare all'inizio di B_2 , ovvero M . Dal momento che i goto sono ancora vuoti, prendo e tramite backpatch metto l'indirizzo di M nella false list. Quando io sono arrivato dopo l'OR, so dove inizia B_2 in quanto la prima istruzione di B_2 è M , facendoli saltare tutti ad $M.instr$. Non posso però risolvere tutti i goto, ma solo quelli dell'OR. Visitor diversi mi permetteranno di fare operazioni totalmente diverse tra loro, compreso anche accettare codici differenti

13.21.1 Esempio: Codice per le espressioni booleane

Dato il codice: "if(a > 1 || b = 2){x=1}", lato codice avremo:

- 100: if a > 1 goto _
- 101: goto _
- 102: if b=2 goto _
- 103: goto _

Possiamo vedere le true list e false list di B, B₁ e B₂ come:

	TL	FL
B (a > 1 b = 2)	100	101
B₁ (a > 1)	102	103
B₂ (B = 2)	100, 102	103

Leggo B₁ e valuto se sia vera o falsa come condizione. Se falsa, devo andare a leggere B₂. Prima di farlo, io ho una M che conserva il cursore. Quando genero codice, lo scrivo nella cella corrente ed incrementa il cursore, che in questo caso viene salvato in M.instr. Prima di entrare in B₂, per questo esempio avrò M.instr = 102. M si annota l'indirizzo corrente nell'array. Mi annoto TL e FL di B₂, in quanto ho dei puntatori ma non so ancora dove mi porterà quel goto.

Se a > 1 io non so dove dovrò andare, in quanto uscirò fuori dalla condizione ma non so il contesto esterno. Per il contesto esterno, dipende da dove si B si trova, da cosa è esterno alla B. Se a > 1 è falsa, però so dove devo andare, potendo riempire il goto. L'indirizzo del goto da riempire la trovo nella falseList di B₁, ovvero 101, mentre l'indirizzo di B₂, ovvero 102, lo trovo in M.instr.

A livello di codice faccio **backpatch(B₁.falseList = M.instr)**. In altri termini, se la condizione di B₁ è falsa, il goto interessato è 101 e deve andare all'inizio della prossima istruzione, salvata in M.instr, ovvero 102

Alcuni goto li riesco a riempire, ma altri no, in quanto conosco solo la B ma non conosco altro del contesto. Tutto dipende da chi usa questa B. Per usare B, non posso perdere la conoscenza di quali possono andare a vero e quali a falso, dovendo generare la TrueList e FalseList di B stesso. Se B₁ è vera, tutta la B è vera. Ma anche se B₂ è vera è vera tutta l'espressione B, pertanto **B.TL = B₁.TL ∪ B₂.TL**. Se B₂ è falsa, considerando che arriviamo a B₂ se anche B₁ è falsa, allora B è falsa, quindi **B.FL = B₂.FL**. Il risultato a livello di codice è:

- 100: if a > 1 goto _
- 101: goto 102

- 102: if b=2 goto _
- 103: goto _

La regola la vado a lanciare dove aver visto la produzione, pertanto **conosco gli attributi degli elementi della parte destra della produzione**. Per il caso $B \rightarrow B_1 \ \&\& \ B_2$, se B_1 è vera mi serve saltare sul primo blocco di istruzione di B_2 . Se invece B_1 è falsa, allora anche B è falsa, pertanto $B.FL = B_1.FL \cup B_2.FL$. Se invece B_2 è vera allora B sarà vera, avendo $B.TL = B_2.TL$

13.22 Esempio: Relazione tra espressioni

Un'espressione booleana banale è: " $B \rightarrow E_1 \text{ rel } E_2$ ". Per scrivere il codice a 3 indirizzi di questa operazione, bisogna prima scrivere il codice a 3 indirizzi per le singole espressioni e ci conserviamo l'indirizzo della variabile che manterrà quell'espressione. Partendo dal codice: "if ((b+a) > 1) then {a=1} else {b=1}", l'espressione booleana è " $b+a > 1$ " dove " $b+a$ " è E_1 e " 1 " è E_2 . Il codice a 3 indirizzi sarà:

- $t_1 = b + a$
- if $t_1 > 1$ goto _
- goto _

Le ultime due istruzioni mi vengono fuori dal fatto che **sto analizzando una condizione**. così facendo, stiamo anche dando un contesto a $b+a > 1$. **Ogni volta che vedo una condizione, devo scrivere "if condizione goto" e alla riga dopo "goto" che mi va ad indicare l'else. Il primo goto vuole andare al codice se la condizione è vera, mentre il secondo vuole andare al codice se la condizione è falsa.**

In altre parole, **si genera il codice sequenziale da sinistra a destra, per cui quando vado a fare il codice per " $b+a$ " riesco a farlo tranquillamente**. Il simbolo " 1 " non ha codice e, avendo entrambe le mie espressioni, vado a sintetizzare in un'espressione booleana la relazione andando a scrivere il codice per la condizione. **Senza sapere molto della condizione, posso già scrivere "if condizione goto" e alla riga dopo "goto". Non saprò ancora dove andare in quanto andrò in una parte del programma di cui non so ancora nulla**. Se fossi all'interno di un if-else, il primo goto andrebbe nella parte del codice in cui si attende che la condizione sia vera ed il secondo goto nella parte di codice in cui si attende che la condizione sia falsa, ovvero nell'else. Scrivendo il codice d'esempio, **siccome non so la posizione dei**

codici, devo lasciare dei buchi. Sarà tramite la funzione `emit()` che mi permetterà di scrivere il codice ed incrementerà la variabile `nextInstr` che punta alla prossima istruzione da eseguire (mi tiene traccia di quale sia la prossima istruzione). Non solo devo creare codice ma mettendo dei goto vuoti devo anche conservare delle informazioni. Le operazioni da fare sono per generare il codice intermedio sono:

- `B.TL = new List(nextInstr)`
- `B.FL = new List(nextInstr+1)`
- `emit(if(E1.addr rel E2.addr) goto _)`
- `emit(goto _)`

Quando arrivo alla fine della produzione " $B \rightarrow E_1 \text{ rel } E_2$ ", io ho scritto, a livello di codice, solo " $t_1 = a + b$ ", ovvero il codice per E_1 . E_2 essendo costante non necessita nulla. La prossima istruzione da scrivere sarà `if t1 goto`

13.23 Blocchi di codice: Espressioni booleane

Ogni volta che ho un blocco di codice che mi implementa un'espressione booleana, posso pensare di avere tanti goto incompleti. Quando avrò finito l'intero codice avrò dei goto completi. Posso classificare i goto in true e false. Se usiamo per controllo di flusso e non calcolo dell'espressione i goto true ci faranno andare in una parte del codice quando l'espressione è vera e i goto false ci faranno andare in un blocco di codice differente quando l'espressione è falsa

Ad un certo punto sono costretto ad andare avanti nella mia analisi, per cui se non mi conservo delle informazioni non potrò più tornare su goto, pertanto è bene che io mi conservi l'id dell'istruzione che ha i goto incompleti. La true list è un insieme di indirizzi del codice che ho già scritto dove ci sono i goto incompleti che però devono saltare ad una parte del codice quando la condizione è vera. La false list sarà una lista di indirizzi che però saltano a del codice quando l'espressione è falsa

13.24 Riassunto: OR ed espressioni booleane

Dato " $B \rightarrow B_1 \text{ OR } B_2$ " quando mi trovo alla fine della produzione, B_1 e B_2 hanno scritto già il loro codice, pertanto nell'array avrò già un blocco di codice per B_1 ed uno per B_2 .

Essendo espressioni booleane sicuramente si avranno dei goto incompleti, classificandoli anche in questo caso come TL e nelle FL. Quando mi trovo alla fine della produzione tutto il codice è stato già scritto. So inoltre che B_1 e B_2 avranno delle liste che mi tengono traccia di dove i goto vogliono andare dove le condizioni sono vere.

Siccome però ho un OR, se B_1 è falsa devo saltare alla condizione B_2 . Una cosa che posso fare, essendo che sto alla fine ormai di B_2 , sarebbe quello di usare una M che indichi la prima istruzione di B_2 . Andrei a dare un valore a questa M non alla fine della produzione ma intanto che la sto leggendo. La produzione diventerebbe " $B \rightarrow B_1 \text{ OR } M B_2$ ". La produzione M va a vuoto e l'attributo di M conterrà il valore della nextInstr. Tutte le istruzioni che vogliono saltare a falso in B_1 , le faccio saltare ad M.instr, in quanto è la prima istruzione di B_2 . Riesco a riempire qualche goto ma non riuscirò a riempirli tutti.

Siccome tutto diventerà B, posso raccogliere le liste. Non potendo perdere l'informazione, devo dare true list e false list di B_1 e B_2 a B. Nella true list di B ci devono essere i goto che fanno rendere vera la condizione, mentre nella false list di B ci andranno gli indirizzi della false list di B_2

13.25 Blocchi di istruzione: Statements

Una cosa che dobbiamo ricordarci sono i possibili goto che un'istruzione può avere. L'espressione booleana avrà sicuramente dei goto che vanno a vero e dei goto che vanno a falso. Per uscire dal codice di un'espressione booleana non posso seguire un flusso logico sequenziale. Se entriamo nel codice di un'espressione booleana siamo costretti a vedere un salto. Con l'espressione booleana, si becca sicuramente un goto e si esce fuori. Si esce fuori da un'espressione booleana solo con i goto, e sicuramente si incontrerà un goto per uscire ed andare da qualche parte. Con gli statement, non è così.

Una volta che ho fatto questo blocco statement, posso andare alla prossima istruzione. Non è detto che la prossima istruzione sia successiva, ma potrebbe anche essere precedente (ad esempio se lo statement che stiamo analizzando è l'ultimo di un while). In generale, essa andrà alla next instruction

Il modo sequenziale però non è l'unico modo con cui posso uscire dal blocco. Se ci fossero solo assegnazioni, sì. Ma non tutte le istruzioni sono così. Il while ad esempio è uno statement, pertanto non è che vedo tutte le istruzioni ed esco alla fine del flusso sequenziale, ma esco dall'istruzione del while

L'istruzione che mi fa saltare fuori, potrebbe non essere l'ultima, ciò significa che non solo l'espressione booleana può avere dei goto incompleti ma anche un'istruzione potrebbe avere dei goto incompleti. Vogliono andare alla prossima istruzione ma non sanno dove sia. Le istruzioni conterranno una next list, ovvero i goto che vogliono portarmi alla prossima istruzione. Dal blocco booleano esco solo con i goto, classificandoli o in veri o falsi, mentre dal blocco statement posso uscire o con goto che vogliono andare alla prossima istruzione o esco fuori in caduta sequenziale, senza goto

13.26 Produzione dell'If e codice intermedio

La produzione sarà " $S \rightarrow \text{if } (B) M S_1$ ". Volendo scrivere l'azione per tale produzione, arrivato alla fine della produzione devo tenere in mente che ho già scritto il codice per B (per cui ho goto vuoti) e per S_1 (per cui potrei avere goto vuoti).

Possiamo riempire i goto della TL di B. Per riempire tale lista dovrei avere l'indirizzo del primo statement di S , pertanto metto un marker. Tutti i goto che vogliono andare a vero, vogliono andare in S_1 , dove la prima istruzione di S è definita da $M.instr$, pertanto inserisco la regola $\{\text{backpatch}(B.TL, M.instr)\}$ e vado a risolvere i goto vuoti nella true list con la prima istruzione di S_1 .

Per i goto a falso, o li riempio, non sapendo però dove saltare, o me li annoto nel padre. Essendo che il padre è uno statement avrà un attributo nextList, pertanto inserisco la regola $\{S.NL = B.FL \text{ Unito } S_1.NL\}$. In essa inserisco anche la nextList di S_1 in quanto una volta svolto il codice dello statement vero dell'if, voglio comunque eseguire il prossimo blocco di codice. Per sapere se la regola sia corretta, devo usare tutti gli attributi. Gli attributi o li riempio perchè ho l'informazione per riempirli o li passo al padre

13.27 Produzione dell'If-else e codice intermedio

Data la produzione " $S \rightarrow \text{if } (B) M_1 S_1 N \text{ else } M_2 S_2$ " ed aggiungendo l'else, quando finiamo di leggere l'espressione booleana ho sicuramente scritto il codice per B, con relativi goto. Incontro M_1 che non fa altro che conservarsi un puntatore alla prima istruzione di B_1 . Visito poi S_1 e scrivo il codice per S_1 e vado avanti. Incontro S_2 e a questo punto scrivo il codice per S_2 . Per essere conservativo e considerare le possibili uscite da S_1 ed S_2 , considero il caso in cui anch'essi abbiano delle next list insieme alla caduta sequenziale (ovvero l'uscita dopo aver letto in modo sequenziale gli statements)

Quando B è vero, devo andare in M_1 , pertanto la true list di B devo riempirla con l'indirizzo in $M_1.instr$, in quanto se vera la condizione vado agli statements S_1 . A livello di azione avrò $\{backpatch(B.TL, M_1.instr)\}$. Essendo in if-else, se la condizione non è vera io so che devo andare ovvero in S_2 . A fine di non perdermi l'indirizzo di inizio di S_2 , uso un altro marker M_2 . A livello di azione avrò $\{backpatch(B.FL, M_2.instr)\}$. Tutti i falsi andranno all'inizio di F_2 .

Ho ancora le next list sia di S_1 che S_2 che non so riempire, in quanto quando entrambe finiscono vanno fuori dall'if-else. A tal fine, per non perdermi il goto, me li salvo nel padre tramite l'azione $\{S.NL = S_1.NL \text{ Unito } S_2.NL\}$.

Se S_1 però non mi incontra il salto, dopo aver eseguito il blocco di codice, andrei a fare prima S_1 e poi S_2 . Prima di entrare in S_2 dovrei mettere un pezzo di codice N che mi permette di fare un salto e di evitare che dopo aver eseguito S_1 vada ad eseguire anche S_2 . Tale codice N , farà l'azione $\{emit(goto)\}$. Il non terminale N , come M , non deve aggiungere nulla al linguaggio. Anche tale goto deve essere messo nella NL di S , in modo tale che non si perdano informazioni. Prima di mettere il goto, posso fare $\{N.NL = \text{next instr}\}$. Il goto di N sarà incompleto in quanto non ho conoscenza di quale sia la prossima istruzione

13.28 Produzione del for e codice intermedio

Data la produzione " $S \rightarrow \text{FOR}(E_1 \ B \ M_1 \ E_2 \ N_1) \ M_2 \ S_1 \ N_2$ ". Anche se scritto in modo sequenziale, quando si esegue il for, le istruzioni non vengono eseguite in modo sequenziale: si esegue prima E_1 , poi faccio l'espressione booleana B , poi faccio S_1 , poi E_2 ed infine riparto da B .

Quando però scriviamo il codice intermedio, non dobbiamo interessarci del modo in cui vengono eseguite le istruzioni (flusso di esecuzione). Dobbiamo generare del codice che garantisce quel flusso di esecuzione.

A tal fine scriverò il codice per E_1 , poi incontro l'espressione booleana B e scrivo subito dopo il codice per B . Si scriverà in sequenza anche il codice per E_2 e poi il codice per S_1 . L'espressione booleana B avrà i suoi goto e S_1 avrà i suoi goto: uno alla next list ed un goto di caduta sequenziale. Devo aggiungere goto in modo che l'esecuzione segua il flusso di esecuzione.

Parto da E_1 e B . In B devo gestire i salti. Se la condizione è vera, devo mandarla all'inizio di S_1 , dovendo mettere un marker M_2 all'inizio di S_1 . L'azione sarà $\{backpatch(B.TL, M_2.instr)\}$.

Se B è falsa devo uscire dal for ed andare alla prossima istruzione, dovendomi portare $B.FL$ alla next list del padre (per evitare di perdermi la false list). La B non può mai cadere sequenzialmente al blocco successivo. Per S_1 , il goto della caduta sequenziale, così come la next list devono andare all'inizio di E_2 , segnato con M_1 . L'azione per fare ciò è $\{\text{backpatch}(S_1.NL \text{ Unito } N_2.NL, M_1.instr)\}$.

Bisogna poi intercettare E_2 che altrimenti cadrebbe naturalmente a S_1 , dovendolo far tornare a M_3 , ovvero la prima istruzione di B . l'azione sarà $\{\text{backpatch}(N_1.NL, M_3.instr)\}$. L'unica uscita per andare alla prossima istruzione è quando la condizione B è falsa, quindi l'azione che devo inserire $\{S.NL = B.FL\}$. Tale codice che presenta molti jump sarà poi ottimizzato successivamente, arrivando addirittura alla teoria del punto fisso per i cicli. Siamo costretti ad utilizzare molti goto in quanto il flusso di esecuzione del for è complesso



14.1 Introduzione: Run-time environment

Già con la generazione del codice, abbiamo visto al codice scritto in un modo diverso: **durante l'analisi semantica guardavamo il codice per analizzarlo, per vedere dove fossero variabili, scoping...** quando andiamo a generare il codice intermedio, invece, noi **dobbiamo pensare all'esecuzione di quel codice.**

In questo caso, **quando si parla di runtime non eseguiamo il codice, ma dobbiamo scrivere il codice macchina finale e pensare a come verrà seguito il test.** L'analisi è sempre statica ma gli occhi sono diversi: **mentre prima cercavamo gli scoping, ovvero le aree in cui le variabili erano dichiarate, adesso andiamo a vedere le chiamate a funzione, l'esecuzione, come dovrà essere eseguito il programma...**

Guardando il codice devo orchestrare la memoria. Per eseguire una funzione ha bisogno di codice e memoria: il codice lo generiamo e la memoria è dove la funzione ha i dati. Nel codice che ho della funzione **devo inserire delle istruzioni che non sono scritte dal programmatore ma sono delle istruzioni che gestiscono i dati.** Facendo chiamate a funzione, sotto avremo uno stack che però non vediamo. ciò significa che **bisogna aggiungere alla traduzione del codice quelle istruzioni che gestiscono lo stack.** In Java, il codice sarà il bytecode, sotto il quale c'è altro codice che gestisce il garbage collector.

L'insieme di tutti i programmi che gestiscono l'esecuzione del nostro programma, ovvero le istruzioni che vanno aggiunte, si chiama run-time environment, ovvero l'ambiente

di esecuzione in cui il nostro programma si trova ad eseguire.

Prima della generazione del codice macchina, **chi fa la generazione del codice macchina, dovrà sia tradurre il codice scritto dal programmatore ma dovrà anche aggiungere il codice per la gestione runtime. Il codice generato è tutt'uno. La gestione dello stack viene mischiata al codice del programmatore e diventa un unico programma.** In generale, ci sono tecniche standard per strutturare il codice eseguibile:

- gestione delle risorse a runtime
- corrispondenza fra statico-dinamico
- organizzazione della memoria

14.2 Risorse a runtime

L'esecuzione di un programma **parte dal SO. Quest'ultimo alloca lo spazio, carica il codice e poi salta al codice. Avrò il mio .exe, lo mando in esecuzione, da ciò il SO capirà quanta memoria occupa il codice e si avranno anche delle specifiche per sapere quanta memoria aggiuntiva serve.**

Nel codice non c'è solo il codice corrispondente al codice scritto dal programmatore, **ma ci sarà anche il codice per gestire lo spazio aggiuntivo. Solitamente vediamo le singole variabili... una visione ad alto livello della memoria.** Tale codice è misto, tra quello corrispondente al programma e tra quello per la gestione delle risorse. **Tradizionalmente si parte da indirizzi bassi e si cresce scendendo**

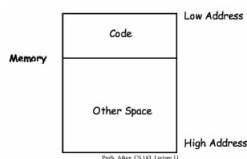


Figura 14.1: Risorse a runtime

il compilatore:

- è importante che generi il codice
- che orchestri l'uso dello spazio dei dati

Se non sappiamo come orchestrare la memoria, non possiamo generare codice. Bisogna prima pensare a come vogliamo organizzare la memoria

14.3 Obiettivi della generazione del codice

I goal sono:

- **correttezza:** il programma che esce fuori deve essere equivalente nell'esecuzione a quello dato. Non è equivalente 1:1 con il codice. Generando il codice macchina per un programma, infatti stiamo generando il programma per un'idea di esecuzione che ha bisogno stack e di altra roba che il programmatore non ha specificato
- **velocità:** possiamo anche creare un programma che a livello di complessità di tempo è ottimo, ma l'uso della memoria che ne fa il compilatore è pessimo, per cui il nostro programma potrebbe andare lentissimo

14.4 Assunzioni sull'esecuzione dei linguaggi

Per poter orchestrare la memoria, bisogna prendere delle decisioni obbligate da chi ha definito il linguaggio. Alcune assunzioni che di solito i linguaggi di programmazione tradizionali adottano sono:

- l'esecuzione è sequenziale e il controllo si muove da un punto di controllo all'altro in un ordine ben definito, sapendo perfettamente dove andare
- quando una procedura viene chiamata, il controllo va alla funzione chiamata e ritorna al chiamante solo dopo che il chiamato ha terminato. Casi in cui questa assunzione non è vera è nelle chiamate asincrone

A seconda dell'assunzione cambia sia come gestire la memoria che l'esecuzione. Se queste assunzioni sono vere, vuol dire che possiamo organizzare le chiamate a funzione in un modo gerarchico. La funzione `f()` chiama `g()` ed `f()` continua solo quando `g()` ha finito. Tali assunzioni guideranno l'orchestrazione della memoria. Il compilatore è statico quindi noi parlando di esecuzione stiamo pensando all'esecuzione

14.5 Chiamate a funzione e gerarchia

Tale organizzazione gerarchica l'abbiamo vista già nello scoping. Lo scoping utilizza stack ed alberi per loro natura, per le chiamate a funzione si usano sempre gli stack ma il problema è differente. Le chiamate a funzione entrano in gioco a runtime, nello scoping invece guardate solo il codice, leggendo il codice. Con lo scoping io leggo il testo riga

per riga, mentre con l'esecuzione, vado a cercare il main e parto da lì quando scrivo il compilatore, ma non lo sto eseguendo

14.6 Concetti di attivazione

14.6.1 Lifetime di una funzione

Ogni volta che faccio una chiamata a funzione ho un'attivazione di quella funzione. Per lifetime di un'attivazione si intendono tutti i passi per eseguire P. I passi includono quelli che non ha pensato il programmatore andandoli ad estendere, dovendo pensare già a livello macchina.

Il lifetime dell'attivazione di P sono tutti i passi per eseguire P inclusi i passi nelle chiamate a procedure P. Ogni volta che una funzione chiama un'altra nella semplice call abbiamo un blocco di istruzioni da dover eseguire. Alcune istruzioni vengono lasciate nel corpo del chiamante, altre vengono assegnate al corpo del chiamato (calling procedure)

14.6.2 Lifetime di una variabile

Il lifetime della variabile è la parte di esecuzione in cui x è definita. A livello dinamico, parlando di lifetime si parla di quando viene eseguito il codice, ovvero delle istruzioni in cui x viene definita, pertanto il lifetime è un concetto dinamico.

14.7 Albero di attivazione

L'assunzione che ci diceva che una funzione deve finire prima di continuare l'esecuzione del padre, ci permette di avere la struttura ad albero. Le strutture di attivazione sono correttamente annidate

Parlando di type environment si parla delle tabelle che sono attive in un punto, ed anche in questo caso si ha un albero, dallo scope globale allo scope finale. Quando facciamo lo scoping partiamo dalla prima riga, dal primo nome, scendendo ed andando a creare tabelle di scoping. Quando vado a parlare di attivazione delle funzioni, la prima cosa che andiamo a pensare di eseguire è il main, pertanto parto dal main(). Sto pensando all'esecuzione. Se main() chiama una funzione, quando quest'ultima finisce torno a main. La lettura del codice, quando stiamo pensando in modo dinamico, è casuale, saltando da destra a sinistra. Mentre sto eseguendo una funzione, saranno attive solo le chiamate del sentiero che conduce a quella funzione

14.7.1 Esempio: type environment e run-time environment

partendo dal codice:

```
Class Main {
  g(): Int { 1 };
  f(x: Int): Int { if x = 0 then g() else f(x - 1) f();
  main(): Int { f(3); }
}
```

Figura 14.2: Esempio: type environment e run-time environment

il type environment e il runtime environment sono: Dovendo fare il type environment,

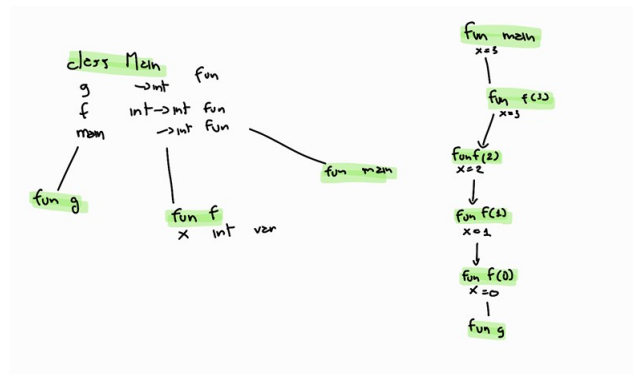


Figura 14.3: Esempio: type environment e run-time environment

possiamo immaginarci di avere una tabella global che contiene la classe main. La classe main contiene poi a sua volta le funzioni f(), g() e main() con le rispettive firme:

- g() è una funzione con firma $\rightarrow \text{int}$
- f() è una funzione con firma $\text{int} \rightarrow \text{int}$ e che ha un parametro in input, pertanto la tabella di scoping contiene la variabile x definita come int
- main() è una funzione con firma $\rightarrow \text{int}$

Se l'if dichiara variabili, dovrebbe esserci una tabella vuota, come dovrebbe esserci anche per l'else e così via.

Nel caso del run-time, parto dal main come funzione: essa sarà la radice dell'albero. In questo caso, siamo a tempo di esecuzione, pertanto stiamo eseguendo il main. Quando eseguiamo il main, esso ha bisogno dei propri dati, del proprio activation record. La memoria che vado a dare al main() è la memoria di cui il main ha bisogno per poter eseguire. main() chiama poi f(), che avrà a sua volta bisogno del suo blocco di memoria con la sua variabile ed il valore della variabile, di cui prima non avevamo bisogno. f() a sua volta chiama f(x-1), quindi f() fa una chiamata a se stesso, con parametro differente.

La funzione `f()` andrà a chiamare se stessa fin quando `x` non sarà uguale a 0. In tal caso, si andrà a chiamare `g()`. A livello di generazione del codice, non vado a specificare `x` ma i valori mi andranno a finire in un particolare punto che il mio programma sarà capace di associare alla variabile `x`. Quando `g()` finisce, si deve ritornare subito dopo la `g()`. Si esce fuori da `f()` e a catena si arriva al `main()`.

14.8 Run-time environment ed informazioni utili

Qui sto eseguendo il programma, simulando, sto pensando all'esecuzione. Se non mi ricordo il valore delle variabili che una funzione ha, tale funzione non potrà essere eseguita. Io che devo generare il codice macchina, devo fare in modo di organizzare dove dati reali dell'esecuzione dovranno andare. Non posso fare l'esecuzione del programma. Non posso fare l'esecuzione del programma ma posso dire in quale blocco di memoria metto i dati, qualunque essi siano. Non posso pensare quali saranno i valori veri e propri ma posso orchestrare la memoria. So solo che le variabili dovranno stare nella memoria del record di attivazione della funzione in cui sono usati. Anche se io parlo della stessa funzione, posso richiamare la funzione più volte, avendo attivazioni differenti della stessa funzione. Per la tabella dei simboli, invece, avrò una tabella unica.

Metto il parametro `x` in una cella che avrà un determinato indirizzo. Il mio codice, che dovrà aggiungere prima di accedere alla `x`, dovrà calcolarsi prima `x` ed una volta trovato l'indirizzo, lavoro sulla cella. Il linguaggio macchina deve prima tradurre la variabile `x` nell'indirizzo di memoria effettivo.

Terminata la funzione chiamata, dobbiamo ritornare prima della chiamata, pertanto ognuno bisogna ricordarsi il punto in cui il padre lo ha chiamato. Ognuno si deve ricordare il punto dove il padre lo ha chiamato, in quanto una volta che il figlio ha terminato, il padre deve ripartire da dove si era fermato. Nella memoria associata al figlio, dovrà essere anche l'indirizzo da dove il padre deve riprendere. Ci si mantiene l'indirizzo di dove il padre deve riprendere.

14.9 Associazione tra variabile e memoria

Nello scoping la memoria l'abbiamo usata per conservare i nomi e l'associazione ai tipi, nel run-time ci serve per conservare i dati che dovranno essere usati durante l'esecuzione. Li organizzo ma non vado ad inserirli. Si avrà l'istruzione che fa l'assegnazione che va a

calcolare poi il valore. Devo pianificare l'environment, ovvero pianificare come dal nome si raggiunge la cella di memoria. Se sbaglio questa associazione si avranno dei problemi, altrimenti si parte da una variabile e si potrebbe andare a prendere la memoria associata ad un'altra variabile.

Ogni blocco di memoria avrà un puntatore intermedio che sarà noto ed in base alla distanza da questo punto, saprò dove stanno x, y e z e farò in modo che tale associazione sia univoca. Siccome tutte memorie saranno sullo stack, la stessa memoria potrebbe essere assegnata a variabili differenti in momenti differenti. Quando la funzione finisce, quel blocco di memoria assegnatogli per eseguire dovrà essere rilasciato. Siccome la struttura è ad albero e a stack, quelle zone di memoria verranno messe in uno stack.

14.10 Comportamento del record di attivazione

Il record di attivazione dipende dal comportamento a runtime. L'albero di attivazione è dinamico e dipende dal tempo di esecuzione. A seconda dei valori che ci sono dentro l'albero potrà cambiare, a differenza dello scoping. Il compilatore non genera alberi di attivazione ma il compilatore crea il codice che creerà a tempo di esecuzione i record di attivazione. Devo pianificare come devono essere fatti i record di attivazione, come possono essere messi sullo sack, come possono essere tolti... dovendo scrivere un programma che, quando eseguito, mi darà l'albero di attivazione

14.11 Gestione della memoria a tempo di esecuzione

Partendo dal codice: Quando chiamo la procedura Main, viene messa sullo stack la

```
Class Main {
  g() : Int { 1 };
  f(x: Int): Int { if x = 0 then g() else f(x - 1) f() };
  main(): Int { f(3) };
}
```

Figura 14.4: Gestione della memoria a tempo di esecuzione

memoria associata al main(). Quando il main chiama g(), sullo stack verrà messo g() e quando g() finisce la sua esecuzione, il blocco di memoria che serviva a g() per eseguire viene rimosso dallo stack e viene sostituito dal record di attivazione nel blocco f(). In sostanza f() e g() useranno gli stessi indirizzi per gestire le proprie memorie, ma siccome g() è morta quello spazio è dato ad f()

Non ho un grosso spreco di memoria in quanto posso usare la memoria che altri non usano più. Dopo che $f()$ ha eseguito, richiamo $g()$ e metterò nello stack anche la memoria per l'esecuzione di $g()$. Il record di attivazione di $g()$ avrà le stesse variabili ma una volta è stata seguita all'indirizzo di memoria di $f()$, ora si trovano dopo. Non posso quindi dare degli indirizzi assoluti alle variabili di $g()$. A seconda di dove mi trovo nello stack, avrò una variabile x che cambierà indirizzo. L'environment per $g()$ cambia a seconda di quando viene chiamata, per cui tutto deve essere relativo e non assoluto. Quando $g()$ finisce e anche $f()$ finisce, si ritorna al main che lascerà lo stack vuoto quando anch'esso finisce. Non c'è una cancellazione, ma una sovrascrittura della memoria. Si sovrascrive la memoria per risparmiare tempo. Sullo stack non cancelliamo ma sovrascrivere solo e facciamo in modo che se un dato non mi serve, non ci accedo proprio

Il blocco di memoria che il sistema operativo di memoria deve dare al programma, sarà un blocco di memoria dove mette il codice macchina compreso il codice per la gestione della memoria. Lo stack lo si fa crescere verso il basso. Basta spostare il top e si sta togliendo e mettendo nello stack. L'informazione necessaria per gestire un'attivazione di una procedura lo chiamiamo record di attivazione o frame

14.12 Record di attivazione o frame

L'informazione necessaria per gestire un'attivazione di una procedura lo chiamiamo record di attivazione o frame. Quel blocco di memoria è il frame o il record di attivazione.

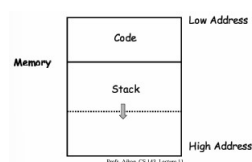


Figura 14.5: Record di attivazione o frame

Se una procedura F chiama G , allora il record di attivazione di G contiene il misto di informazione tra F e G . Ogni volta che facciamo una chiamata a funzione, il codice macchina corrispondente sarà il blocco di istruzioni che serve a fare la chiamata di F . Un pezzo di tale codice lo mettiamo nel corpo del chiamante e parte nel corpo del chiamato.

14.13 Contenuto del record di attivazione di una funzione chiamata

Nel record di attivazione di G ci sono sicuramente i parametri. Avendo che F chiama G, stiamo cercando di capire cosa abbia il chiamato. F è sospeso fin quando G è in esecuzione. Abbiamo bisogno, in G, delle informazioni per risvegliare F, ovvero abbiamo bisogno di sapere quale sia la prossima istruzione che deve fare F quando G ha finito.

F sta nell'istruzione 20. F poi chiama G che si conserverà il numero 21, cosicché quando G finisce, dirà che bisogna riprendere dalla riga 21. Non può essere F a dirlo in quanto essa è dormiente. Tale informazione è conservata in G

Se G è una funzione, allora deve restituire un dato ad F. Tale dato lo mettiamo nel record di attivazione di entrambi, in quanto serve ad F ed è un dato che sta in G. Il valore che la funzione deve restituire, la crea il chiamato ma deve usarla il chiamante, pertanto la mettiamo a metà fra le due, ovvero alla fine del record di attivazione del chiamante ed è anche la prima del chiamato. Quando il chiamato G finisce, la sua memoria viene rilasciata, ovvero spostato il top, non cancellando nulla. La memoria del valore di ritorno si troverà a top+1 ed il chiamante tranquillamente userà il top per andare ad accedere al valore. I parametri attuali di G saranno nel chiamato insieme alle variabili interne a G stesso.

14.14 Sintesi sul contenuto del record di attivazione

Le cose contenute in una funzione chiamata sono:

- il valore di ritorno lo mettiamo in alto, adiacente al record di attivazione del padre. Il valore di ritorno è un'area di memoria in cui G scrive ed F può leggere
- i parametri attuali passati dal padre. è il chiamante F che scrive nel record di attivazione del chiamato
- control link: il puntatore al record di attivazione precedente. F ha il suo record di attivazione. G si conserva un puntatore a quello del record di attivazione del padre. Ciò prende il nome di control link. il record di attivazione di G lo prepara F e poi gli passa il controllo. Tutta la memoria deve essere preparata da chi chiama, perciò c'è un blocco di istruzioni che deve essere eseguita ogni volta che si ha una chiamata a funzione. Devo predisporre il record di attivazione del chiamato e poi gli passo il controllo

- **il machine status:** il machine status mantiene lo stato della macchina. La funzione F ad un certo punto smette la sua esecuzione, avendo magari i suoi registri principali tra cui il program counter, ovvero l'indirizzo dell'istruzione del programma che deve eseguire. La funzione F sta eseguendo ed ha un certo stato dei registri con cui sto lavorando e ho il program counter che mi dice la prossima istruzione da eseguire. Quando chiamo G, congelo ciò che ho e lo memorizzo nel record di attivazione del chiamato (ovvero G). Lo metto nel record di attivazione del chiamato cosicché quando il chiamato smette, prima di smettere, ristabilisce il machine status, cosicché il chiamante F si ritrova la situazione identica a prima che avesse chiamato. è un problema di F che dovrà gestire G, che deve ristabilire la situazione com'era prima, con la prossima istruzione
- deve salvarsi le variabili locali
- deve conservarsi le variabili temporanee

In generale, si deve conservare qualcosa del padre e qualcosa di suo. Quando si chiama una funzione, il padre deve: allocare spazio per il valore di ritorno, mettere i parametri attuali, mettere i puntatori allo stesso record, conservo il machine status e mi fermo, in quanto le variabili locali se le vede la funzione G. Questi campi dipendono dall'implementazione del compilatore stesso. Anche l'ordine è casuale. Per il risultato, anche se all'inizio è vuoto devo comunque lasciare lo spazio per salvarmi il risultato. Gli argomenti, il control link ed il return address non saranno vuote come celle di memoria

14.15 Celle di memoria e record di attivazione

Importanti sono il valore di ritorno, gli argomenti che servono alla funzione per eseguire, il contro link, il return address, machine status e variabili locali e temporanee. Siccome le temporanee sono variabili, le mettiamo per ultime. Quando la funzione è attiva, il resto dello stack sotto è campo aperto, non è utilizzato da nessuno. In generale:

- **il risultato ha una taglia fissa:** se la funzione restituisce un intero,
- **gli argomenti anche hanno un tipo noto,** pertanto si userà un numero fisso di byte.
- **Il control link anche in quanto è un puntatore alla struttura interna**

- anche il **return address** e **machine status** hanno **taglia fissa** La taglia di tali oggetti è nota al compilatore e non dipende dall'esecuzione. Non so cosa ci sia dentro durante l'esecuzione ma so il loro tipo e le loro dimensioni.

14.15.1 Esempio: chiamate a funzione e celle di memoria

```

Class Main {
  g() : Int { 1 };
  f(x: Int): Int { if x=0 then g() else f(x-1)(**)fi);
  main(): Int { f(3); (*)
};}

```

AR for f:

result
argument
control link
return address

Figura 14.6: Esempio: chiamate a funzione e celle di memoria

Nel main non ci sono variabili temporanee né locali. Quando il main chiama f(3), a tempo di esecuzione, il main lascia lo spazio del risultato, il 3 dovrà finire nella cella dopo, avremo poi il control link che punta all'activation del padre e avremo poi il return address indicato da (**). Nel mondo reale esso sarà il program counter. Quando f richiama se stessa, viene messo sullo stack un blocco identico con x=2. Il control link punterà all'inizio del chiamante e così via. Quando sono all'interno del codice, mi troverò sempre in un record di attivazione che sta eseguendo e userò le variabili di quel record di attivazione. Quando la funzione finisce, (**) mi indicherà dove andare. Il chiamato dovrà cambiare il program counter della macchina

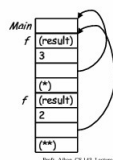


Figura 14.7: Esempio: chiamate a funzione e celle di memoria

14.16 Nozioni su i compilatori e gestione della memoria

Tale progettazione, non è quella di record di attivazione di tutti i linguaggi. Alcuni linguaggi, ad esempio il fortran, **davano un record di attivazione fisso per ogni funzione, parlando di memoria statica**. Con ciò non si poteva fare la ricorsione.

Il compilatore deve **determinare a tempo di compilazione, ovvero guardando solo il codice, lo schema di record di attivazione e generare codice che accede correttamente ai record di attivazione, facendo mapping tra nome della variabile e locazione di memoria**. Il

vantaggio di porre il valore di ritorno all'inizio è che il chiamante può trovarlo ad un punto fisso dal proprio frame. Quando la funzione *f* restituisce il suo valore, lo metterà in un punto fisso. Se la funzione chiamante deve usare un valore del chiamato, se lo trova ad una distanza prefissata. Quando una funzione rilascia il proprio record di attivazione, esso non viene cancellato ma semplicemente il top dello stack viene arretrato e la sua zona diventa zona allocabile a qualcun altro. In altri termini, si va ad estendere la zona di attivazione del chiamante. L'ordine degli elementi può essere cambiato, si può decidere cosa fa il chiamante e cosa il chiamato... In generale, l'organizzazione è migliore di un'altra se migliora la velocità e semplifica la generazione del codice.

I compilatori reali, cercano di mantenere le informazioni nel frame quanto più possibile nei registri in quanto sono più veloci e più vicini alla CPU.

14.17 Variabili globali

tutti i riferimenti alle variabili globali puntano allo stesso oggetto. Essi non devono poter accedere ad una memoria di allocazione diversa. Deve esistere un unico punto accessibile da tutti. Le globali non vanno nel record di attivazione in quanto quando il record di attivazione va via, perderei la globale. Le si dà un indirizzo fisso una volta per tutte e sono staticamente allocate.

Subito dopo il codice che ha una taglia fissa e nota, avremo la memoria per le variabili statiche. Essa dovrà essere la stessa per tutte le funzioni e per tutta la durata del programma. Quando accedo a tale memoria, posso addirittura fare un'associazione 1:1 e non serve che faccio un calcolo per capire dove si trovi una variabile

14.18 Heap

Per la memoria, dinamicamente allocata, serve l'heap. Un valore potrebbe sopravvivere alla vita della funzione. Se quando la funzione finisce, libero la sua memoria, c'è il rischio che liberi anche questa variabile che invece doveva sopravvivere. Tutte le memorie allocate con *new*, *malloc*... le vado a mettere nell'heap. Linguaggi che allocano dinamicamente usano un heap per memorizzare i dati dinamici

Heap e stack non sono a taglia fissa, ma sono stati messi in un'area a cui entrambi accedono. Se mi serve molto lo stack e non l'heap, potrei anche espandermi, e viceversa. Se bloccavo la memoria, poteva bloccare il programma mentre c'era altra memoria disponibile.

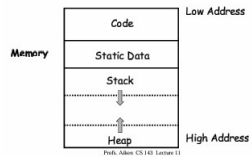


Figura 14.8: Heap

L'area di codice, invece, contiene il codice oggetto e per la maggiorparte del programma è fissata e read only. Anche i dati statici sono a taglia fissa, mentre stack ed heap sono a taglia variabile. Bisogna tener cura che non crescano uno dentro l'altro, pertanto heap e stack vengono messi in modo opposto.

14.19 Concetto di alignment

La maggiorparte delle macchine hanno parole di 32 bit. L'accesso all'inizio di ogni parola è più veloce: se vogliamo mettere dei dati, conviene mettere multipli di parole. Se però abbiamo una stringa più corta, sarebbe efficiente portarla ad un numero fisso, aggiungendo del padding ed andando ad allinearla. Il padding non è parte della memoria ma memoria non utilizzata. C'è però un certo spreco.

14.20 Stack machine

La gestione delle temporanee può essere usata tramite stack machine. Esso è un modello di valutazione semplice, che serve per valutare le espressioni. Non usa né variabili né registri, quindi non c'è la memoria per una variabile specifica. Anche i valori intermedi finiscono sullo stack. Per fare la somma di due numeri, abbiamo un'allocazione per i due numeri. Vi si accede e si fa poi la somma tramite l'indirizzo delle due variabili (avendo 3 indirizzi)

In questo caso, con lo stack, le istruzioni lavorano sullo stack e non sulla ram. Lavorano con pop e push che avranno al più un argomento. Il push mi dice cosa mettere ma non dove, in quanto lo metto sempre al top dello stack. Vengono a mancare gli indirizzi degli argomenti.

14.21 Funzionamento dell'istruzione somma

Ciascuna istruzione prende gli operandi dal top dello stack. Ho lo stack con una certa configurazione:

- incontro la somma, che è un operatore binario, che ha bisogno di due argomenti
- faccio la pop dei primi due elementi
- mi porto gli elementi nella cpu e faccio l'addizione
- faccio il push sullo stack del risultato

Ciò che stava sotto i due elementi è rimasto inalterato

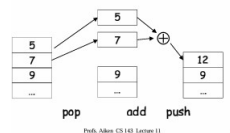


Figura 14.9: Funzionamento dell'istruzione somma

I due elementi li devo prima mettere nello stack, poi faccio l'add. Non devo scrivere nemmeno gli argomenti di add in quanto saranno i due più al top. Le istruzioni sono più corte. Ciascun operazione prende gli operandi dallo stesso posto e mette i risultati nello stesso posto, ovvero al top dello stack. Gli operandi si trovano sempre al top dello stack, non c'è bisogno né specificare dove sono gli operandi né dove mettere il risultato

14.22 Macchine a stack con accumulatore e schema uniforme

Per l'istruzione somma, ci sono 3 operazioni di memoria, 2 pop ed un push. Per ottimizzare questa cosa, possiamo usare una cella alternativa e lo chiamiamo accumulatore.

Per poter fare una somma, vado a prendere nell'accumulatore un argomento e l'altro argomento dal top dello stack, mettendo tutto nell'accumulatore. Con questo tipo di istruzione, ho evitato di fare una lettura al top dello stack ed una scrittura del risultato al top dello stack. Avrò solo un'operazione sulla memoria

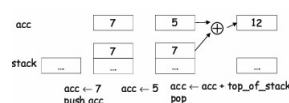


Figura 14.10: Macchine a stack con accumulatore

Il record di attivazione continua ad esistere. Al posto della gestione delle temporanee mettiamo questo stack, in quanto le temporanee le introducevamo per fare i calcoli. In questo caso, si parla di macchine a stack con accumulatore, dove l'accumulatore mantiene il risultato nello stack. Qualsiasi operazione debba fare faremo sempre la stessa cosa. Il risultato lo metteremo sempre nell'accumulatore

Quando c'è l'operazione su n argomenti, metto il risultato nell'accumulatore dopo aver calcolato $n-1$ elementi. Tutte le $n-1$ espressioni verranno prima messe nell'accumulatore e poi scaricate sullo stack e alla fine avrò e_n nell'accumulatore. per calcolare il risultato finale, faccio il pop degli $n-1$ elementi nello stack, e_n sarà già nell'accumulatore quindi non serve altro, ed infine vado a salvare il risultato nell'accumulatore. Nella somma anche 5 o 7 sono esse stesse espressioni. è bello notare che la configurazione dello stack a termine dell'operazione sarà identica a com'era prima, e l'operazione avrà il proprio risultato nell'accumulatore. Avendo stack ed operazione, alla fine dell'operazione avrò lo stesso stack ed il risultato nell'accumulatore. Questa è un'assunzione che posso fare nel mio calcolo successivo. In generale, si parla di schema uniforme in quanto mi lascia lo stack invariato e ciò lo so anche da prima. Sarò facile scrivere il codice per ottenere ciò in quanto posso fidarmi del fatto che dopo l'operazione lo stack sarà lo stesso, non avendo side effect sullo stack, ma solo sull'accumulatore, potendo continuare a lavorarci

14.22.1 Esempio di stack ed accumulatore

Volendo fare $7+5$, metto nel mio accumulatore 7. Essendo che devo mettere anche 5, scarico 7 nello stack e 5 nell'accumulatore. Leggo poi l'operazione di somma e vado a prendere 7 dallo stack e 5 dall'accumulatore. Faccio la somma e salvo il risultato nell'accumulatore

Lo stesso concetto è applicabile per $3 + (7+5)$, dove $7+5$ è un'espressione a sua volta. Parto a mettere 3 nell'accumulatore. Lo scarico poi nello stack e metto 7 nell'accumulatore. Faccio lo stesso con 7 e metto 5 nell'accumulatore. Calcolo poi $7+5$ e salvo il risultato nell'accumulatore. Faccio poi $12 + 3$ e salvo il risultato nell'accumulatore. Avrò il risultato dell'operazione nell'accumulatore e lo stack così com'era prima. Sarò facile scrivere il codice per ottenere ciò in quanto posso fidarmi del fatto che dopo l'operazione lo stack sarà lo stesso, non avendo side effect sullo stack, ma solo sull'accumulatore, potendo continuare a lavorarci

14.23 Riassunto: gestione della memoria

Quando andiamo a generare un codice, dobbiamo pensare alla sua esecuzione. Devo scrivere un programma che genera un codice che fa determinate cose e questo codice generato deve fare quello per cui è stato scritto, dovendo pensare alla doppia esecuzione. Non stiamo eseguendo nulla ma stiamo pianificando l'esecuzione

Ogni volta che una funzione viene chiamata, viene messo sullo stack un record di attivazione. Il record di attivazione contiene alcuni campi come: risultato, argomenti, control link e return address. Ogni volta che la funzione viene chiamata, verrà allocato un blocco di memoria nella zona stack della RAM. In tal caso, nel runtime environment, andiamo subito a cercare il main e vedere l'esecuzione.

Le variabili del nostro codice dovranno essere associate a delle celle di memoria. L'indirizzo fisso verrà messo dopo il codice e sarà unico, ovvero viene messo nella zona statica. Se la variabile è locale o appartiene alla funzione allora essa andrà a finire sul record di attivazione che andrà messa nello stack. L'associazione della variabile con la locazione di memoria verrà fatta a runtime, a priori.

Parlando di runtime intendiamo che noi dobbiamo scrivere un codice che sia in grado di calcolare l'indirizzo di quella variabile. Dobbiamo sapere perfettamente dove si trova nell'activation record. Ci muoviamo relativamente all'activation record, in quanto saranno indirizzi relativi a dove l'activation record viene messo nello stack. Pianifichiamo come deve essere la memoria, dove devono essere messe le variabili in modo preciso, in modo che quando devo raggiungere una variabile so perfettamente come muovermi per trovare la locazione di quella variabile

14.24 Riassunto: heap e stack

Discorso diverso è per l'heap che contiene variabili che possono sopravvivere alla funzione. Possiamo usare gli stack in quanto siamo partiti dall'assunzione che le funzioni, quando ne chiamano un'altra, aspettano che il figlio finisca, pertanto abbiamo una chiamata sincrona. Ciò ci permette di avere una gerarchia di chiamate e quindi di avere un albero di attivazione. Potremo poi rappresentare ogni path attivo come uno stack.

Tutto ciò che abbiamo detto sul record di attivazione vale sia per le macchine RAM che le macchine a stack (bytecode di Java). Nelle stack machine, c'è in più il fatto che alla fine del record di attivazione, le variabili temporanee e locali possono essere gestite ancora

sullo stack, usando solo pop e push. Siccome il top dello stack ha il maggior numero di accessi, pertanto è critica dal punto di vista della complessità di tempo, possiamo effettuare delle ottimizzazioni. La differenza tra macchine RAM a stack è data dal codice, che per le macchine a stack è più snello

14.25 Riassunto: stack ed accumulatore

Per poter generare il codice sulla macchina a stack, una proprietà importante è quella dell'accumulatore. Qualsiasi espressione vogliamo calcolare quando generiamo il codice, deve lasciare l'espressione finale nell'accumulatore e deve lasciare lo stack invariato. L'accumulatore alla fine riceve il risultato finale in quanto rispetta la proprietà ricorsiva. Tale tipo di programmazione ordinata potrebbe darci un po' di inefficienza ma sarà perfetto per continuare a programmare in modo ordinato

14.26 Codice MIPS

Il codice MIPS simula lo stack. Il funzionamento del MIPS è che mette l'accumulatore nel registro 0. Questo è possibile in quanto il MIPS è una macchina RISC, ovvero con molti registri, essendo inoltre capace di mantenere le istruzioni molto semplici.

L'accumulatore sarà a registro 0, mentre lo stack pointer sarà un puntatore allo stack. Sarà lui a fare l'allocazione andando ad incrementare lo stack pointer. Ogni volta che incrementiamo lo stack pointer stiamo lasciando spazio all'activation record. Da tenere ben presente è che lo stack cresce verso gli indirizzi bassi e lo stack pointer punta alla prima cella libera. Essendo che le operazioni sul MIPS sono solo su registri, solo il load e store mi fanno le operazioni sui registri. Quando io faccio il pop, passo da un valore di SP minore ad uno maggiore. La cella precedente diventa disponibile a scrittura e qualsiasi cosa conteneva non mi interessa più in quanto non sarà raggiungibile. Non viene cancellato nulla ma sarà tutto lasciato così com'è e sarà SP che decide quale sia la memoria allocata e non allocata. È importante preservare lo stack ed avere il risultato dell'espressione, anche se a livello di codice non è la cosa più efficiente ma a livello di logica rende tutto più facile

14.26.1 Esempio: Stack e scrittura di un elemento

Supponendo che SP punti inizialmente a 16, aggiungendo un altro elemento faccio salire SP a 12. SP punterà quindi alla prima cella libera. Questo vale sempre. Per 0(sp) si

intende che la mia locazione di memoria sarà data da $0 + sp$. Supponendo di essere a $SP=16$ e volendo pushare il valore 1, devo fare un'operazione di *li* (load immediate), ovvero il load di una costante.

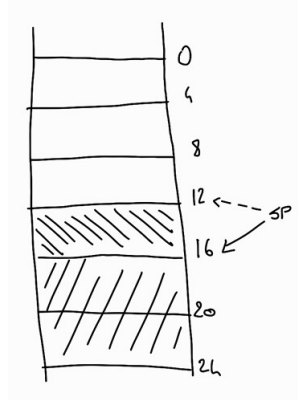


Figura 14.11: Esempio: Stack e scrittura di un elemento

14.27 Da stack machine a MIPS

Il compilatore genera codice per uno stack machine con accumulatore. Vogliamo eseguire il risultato su simulatore MIPS. Dobbiamo simulare la stack machine. L'accumulatore sarà a 0, lo stack point a SP ed il return address lo metteremo anch'esso in un registro. Avremo anche un frame pointer per raggiungere le variabili. Lo stack cresce verso indirizzi più bassi e l'indirizzo della prossima locazione sullo stack è mantenuto dal MIPS in SP ed il top-1 dello stack è all' $SP+4$ (essendo che cresce verso indirizzi più bassi, il top-1 è a $SP+4$)

14.28 Istruzioni su MIPS

Il MIPS ha una struttura RISC, quindi con molti registri. Load e store usano operandi e risultati in memoria. Avremo che \$sp è lo stack pointer, \$a0 l'accumulatore e \$t1 come registro temporaneo. Alcune istruzioni sono:

- **lw**: serve per caricare un elemento dato un registro di partenza. Nel codice "**lw reg₁ offset(reg₂)**" carico il contenuto di $offset+reg_2$ in reg_1
- **add**: mi permette di fare la somma. Dato "**add reg₁ reg₂ reg₃**" metto la somma di reg_2 reg_3 in reg_1
- **sw**: serve per salvare un elemento dato un registro di partenza. Nel codice "**sw reg₁ offset(reg₂)**" salvo il contenuto di reg_1 in $offset+reg_2$

- **addiu**: mi permette di fare la somma di un registro ed una costante. Dato “addiu reg₁ reg₂ imm” metto la somma di reg₂ imm in reg₁
- **li**: mi permette di mettere una costante in un registro. Nel codice “li reg imm” metto imm in reg

14.28.1 Esempio: MIPS e codice per la somma

```

acc ← 7
push acc

acc ← 5
acc ← acc + top_of_stack
pop

li $a0 7
sw $a0 0($sp)
addiu $sp $sp -4
li $a0 5
lw $t1 4($sp)
add $a0 $a0 $t1
addiu $sp $sp 4

```

Figura 14.12: Esempio: MIPS e codice per la somma

Parto da **mettere 7 nell’accumulatore, ovvero \$a0**. Dovendo prendere anche 5 salvo 7. Tramite “sw \$a0 0(\$sp)” prendo il contenuto dell’accumulatore e lo salvo in un registro 0(\$sp). Vado poi ad aggiornare lo stack pointer tramite il codice “addiu \$sp \$sp -4”. Vado poi a fare il **load della costante 5 in \$a0**, il load del contenuto di 4(\$sp) in \$t1, faccio poi la **somma dell’accumulatore e della variabile contenuta nella cella temporanea \$t1** e lo salvo in \$a0 stesso. Infine, faccio il **pop di 7 dallo stack** tramite addiu “\$sp \$sp 4”

Per la variabile 7, faccio i passaggi: **accumulatore-> stack-> registro \$t1**. Ciò viene fatto perchè nelle macchine a registri (MIPS simula la macchina a stack ma è a registri) per poter fare la somma tra 7 e 5 devo metterle entrambe nei registri

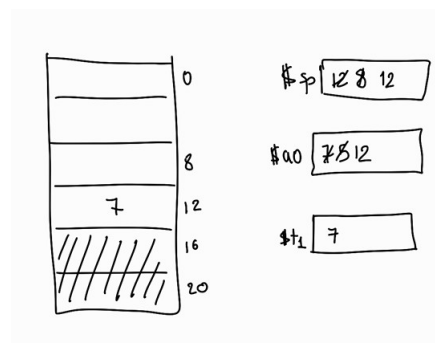


Figura 14.13: Esempio: MIPS e codice per la somma

14.29 Funzione fittizia cgen(e)

Quando mi trovo al **nodo al top**, non saprò che valore ha, sapendo solo che essa sia un’espressione. Per ciascuna espressione e, noi generiamo il codice MIPS che calcola il

valore in \$a0 e preserva lo stack. Preservare lo stack vuol dire che manterremo lo stesso stack pointer che avevamo prima di effettuare l'operazione

Andiamo a definire una funzione `cgen(e)` che ha come risultato il codice da generare. Essa rappresenterà un interprete e non dovrà eseguire il codice ma solo stamparlo. In tal caso, la somma di due espressioni sarà data da `cgen(e1 + e2)` che si va a scrivere il codice per `e1` e per `e2` tramite `cgen(e1)` e `cgen(e2)`. Alla fine di ogni espressione, avrò il risultato della stessa nell'accumulatore e lo stack invariato. Ogni espressione potrà generare quanto codice vorremo ma so che alla fine dell'espressione lo stack rimarrà invariato. Qualunque codice mi venga generato, lo stack non sarà cambiato. Posso continuare a programmare continuando ad avere una chiara visione dello stato della memoria. Del codice che verrà generato da `e2` avrò solo il risultato salvato nell'accumulatore. Quando sarò in `e2`, avrò il risultato di `e1` che è nell'accumulatore. Se serve l'accumulatore libero potrò portarmi il risultato di `e1` nello stack e alla fine, il risultato lo rimetterò nell'accumulatore facendo il `pop`.

La semplicità è importante, in quanto il nostro codice più che efficiente deve essere corretto. La generazione può essere scritta come una discesa ricorsiva dell'AST

14.30 Implementazione dell'if nel MIPS

Per fare l'if, possiamo utilizzare un'istruzione chiamata `branch to label`. Il codice sarà `"beq reg1 reg2 label"` che ci permetterà di saltare alla label se i due registri sono uguali, o per un salto incondizionato potremo usare `"b label"`.

14.30.1 Esempio: MIPS ed if

```

cgen(if e1 = e2 then e3 else e4) =
  cgen(e1)
  sw $a0 0($sp)
  addiu $sp $sp -4
  cgen(e2)
  lw $t1 4($sp)
  addiu $sp $sp 4
  beq $a0 $t1 true_branch

false_branch:
  cgen(e4)
b end_if
true_branch:
  cgen(e3)
end_if:

```

Figura 14.14: Esempio: MIPS ed if

Avendo `cgen(if e1 = e2 then e3 else e4)`, come prima cosa lancio `e1`, conservando il valore. Vado poi a calcolare il valore di `e2` lasciando lo stack identico a come era prima. Mi porto in `t1` il risultato che sta all'indirizzo `4($sp)`, faccio la somma per il `pop`. Posso poi effettuare il controllo tra `$a0` e `$t1`. In caso siano uguali salto a `true_branch` e se non è vera la condizione

vado direttamente a `false_branch`. Avrò poi bisogno di `"b end_if"` per saltare al codice successivo all'`if` tramite salto incondizionato

14.31 Chiamate a funzione e generazione del codice

La prima cosa da fare, è capire come organizzare l'`activation record`. Informazioni che sicuramente devono esserci sono:

- il risultato finale
- gli argomenti della funzione
- il control link, ovvero il puntatore al padre
- l'indirizzo di ritorno a cui mi serve saltare

Applicando tali concetti alla stack machine, il risultato finale lo terremo nell'accumulatore, potendolo togliere come elemento da salvare. Gli argomenti devono esserci per forza, mentre il control link che mi fa in modo di trovare il padre è superfluo, questo perché con lo stack pointer, quando una funzione finisce, ritorna alla precedente. Il return address, invece, mi serve e momentaneamente possiamo metterlo in un registro. Il record di attivazione mantiene i parametri attuali. Se una funzione è x_1, \dots, x_n sullo stack metteremo tutto al contrario in modo tale che x_1 sia al top. Se in un linguaggio non vengono definite variabili locali, nel record di attivazione saranno escluse. Al record di attivazione sullo stack andiamo ad aggiungere un frame pointer che mi servirà per raggiungere le variabili.

Se io so l'offset tra il frame pointer e la variabile x , potrò facilmente raggiungere tale variabile. Ciò è necessario in quanto se uso lo stack pointer ed esso cresce e si muove, avrei problemi a trovare la variabile, Devo avere un punto dove non c'è variabilità

14.32 Frame pointer

Anche il frame pointer me lo devo conservare, pertanto non ho il control link ma avrò il frame pointer nel record di attivazione. Chi fa la chiamata $f(x, y)$ si conserva la chiamata al vecchio frame pointer che garantisce al padre come raggiungere determinate variabili. Io devo conservarmi il frame pointer in quanto lo avrò in un determinato registro

Siccome devo sovrascrivere il frame pointer con uno nuovo, mi conservo il vecchio e mi mantengo il nuovo. Quando finisco mi riprendo il vecchio e il padre saprà come raggiungere determinate variabili. Avrò poi le variabili.

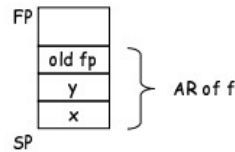


Figura 14.15: Frame pointer

14.33 Calling e return sequence

Quando si effettua una chiamata a funzione, tale chiamata diventa un blocco di istruzioni. Un pezzo lo deve fare chi chiama, un altro pezzo è accorpato al corpo del chiamato. Tutte le istruzioni di chiamata si chiamano *calling sequence* mentre le istruzioni di ritorno si chiamano *return sequence*. Durante la *calling sequence* il padre prepara il record di attivazione del chiamato per poi fare un jump al chiamato. C'è una *calling sequence* che prepara il record di attivazione, gli argomenti... e poi fa il jump alla funzione.

La *calling sequence* può continuare nel figlio. Il chiamato potrebbe finire le operazioni di chiamato e poi dopo eseguire il proprio codice. Quando ha finito, dovrà fare una *return sequence*. Ci sarà del codice che viene quindi diviso tra padre e figlio. Anche la *return sequence* può essere completata dal chiamante.

Il codice di una chiamata a funzione è suddivisibile in *calling sequence* e *return sequence*. La *calling sequence* può essere un pezzo del chiamante prima che si faccia il jump al chiamato ed un pezzo del chiamato prima di iniziare l'esecuzione. Quando il chiamato ha finito di fare, deve fare al minimo un jump al padre o in alternativa una *return sequence*. Le sequenze fanno spesso riferimento all'aggiornamento del record di attivazione. Parte della *return sequence* del chiamante è ad esempio quella di prendersi il risultato finale del chiamato per metterla nella variabile. La *return sequence* non termina fin quando il chiamante non prende il risultato finale dal chiamato e la mette nella sua memoria.

14.34 Comando per il jump della funzione

L'istruzione "jal label" è come l'istruzione "b label" ma si conserva la prossima istruzione da eseguire. Se ho questa istruzione alla riga 100, si conserva 101 nel registro RA e poi fa il salto. Una volta che è stato fatto il salto, il chiamato dovrà conservare tale RA in quanto se fa un'altra chiamata, il nuovo jal potrebbe sovrascrivere il vecchio. La funzione chiamata, può a sua volta chiamarne un'altra, potendo lanciare a sua volta un jal. La prima

cosa che fa il chiamato è conservare l'RA e poi inizia a lavorare. Ciò fa parte della calling sequence, mentre la return sequence predispone la memoria e mette la situazione com'era prima.

14.35 Passi della calling sequence

La prima cosa che fa il chiamante è salvarmi il frame pointer nello stack, facendo un push del frame pointer sullo stack. Mi vado a calcolare i valori dell'espressione. Si va a generare il codice per e_n (in quanto calcola all'inverso), lo mette sullo stack e aggiorna lo stack pointer. Ciò lo fa fino ad e_1 . Alla fine dell calling sequence, avrò il frame pointer con il risultato degli argomenti. Faccio poi il jal che mi conserva nel registro RA il prossimo indirizzo. A questo punto, il chiamante ha finito la sua parte di calling sequence

Facciamo un salto dove la funzione è definita, ovvero un salto al chiamato. La prima cosa che fa il chiamato è portarsi il frame pointer a puntare allo stack pointer corrente, in modo tale che esso sia attaccato alle variabili. Il chiamato si conserva poi il return address, ovvero fa il push del return address. L'informazione dove siano le variabili è data dalla tabella dei simboli che indicherà il tipo delle variabili, pertanto a loro size SP punterà ora alla nuova cella libera mentre FP. Si blocca FP in quanto esso non si muoverà durante l'esecuzione della funzione (a contrario di SP) e potrò ora raggiungere tutte le variabili.

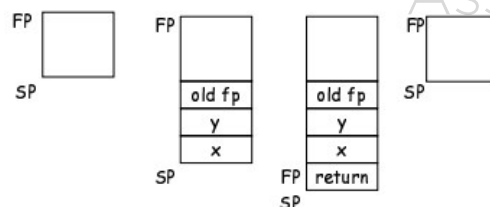


Figura 14.16: Passi della calling sequence

Nella tabella dei simboli, devo anche mettere il tipo e pertanto saprò quale sia la dimensione, potendo specificare anche gli offset. La dimensione allocata per ogni variabile sarà nota. La mia funzione chiamata, può eseguire ora il codice per il corpo della funzione. Una volta eseguita, troverò nell'accumulatore il risultato, il return value. Se il chiamante vuole un risultato dal chiamato, dopo aver fatto il jal, quando gli ritorna il controllo, la funzione chiamante potrà accedere all'accumulatore e prendersi il risultato. Nella return sequence, il chiamante si prende il valore e lo mette nella sua memoria (in questo caso non necessario in quanto abbiamo un registro).

14.36 Passi della return sequence

Una volta che la **funzione ha eseguito il suo lavoro**, ci sono delle istruzioni in più che devono gestire la restituzione del controllo al chiamante. Mettiamo tutto com'era prima e in aggiunta l'accumulatore avrà il risultato finale.

Durante questa sequenza, il **chiamato mette il return address dov'era prima**, tolgo il contenuto del padre, metto a posto anche il frame pointer e la situazione ritorna ad essere quella del punto di partenza. Una volta fatto ciò il chiamato deve restituire il controllo, e possiamo restituire il controllo in quanto si può fare il jump al contenuto del return address. L'indirizzo sarà lo stesso che al label si è salvato, quindi si ritorna al padre.

In questa implementazione, **non c'è una vera e propria return sequence del chiamante**, ma nel caso generale cosa deve fare chiamato e chiamante è specifico del compilatore

14.37 Riassunto: Calling e return sequence

In altri termini, la calling sequence del chiamante prende il registro FP, lo memorizzo nello stack poi genero il codice per i parametri e metto il risultato nell'accumulatore e poi nello stack al contrario e faccio la chiamata. Il chiamato, conserva il return address nello stack, aggiunge allo stack quello che gli è stato dato dal chiamante, aggiorna l'FP a puntare al return e lo stack pointer viene aggiornato anch'esso. Finisco la calling sequence del chiamato, eseguo il codice e dopo la generazione del codice il chiamato fa la return sequence. Fa il pop del return e lo mette nel registro RA. Leva gli argomenti, fa il restore del vecchio FP (e ne fa anche il pop), pertanto FP ed SP si ritrovano dove erano prima e il risultato si troverà nell'accumulatore.

14.38 Generazione del codice per le variabili

Non possiamo usare SP in quanto **può muoversi durante il calcolo dell'espressione**. L'espressione fa riferimento a variabili pertanto sarà necessario arrivarci. **Quello che si fa è usare il Frame Pointer che punta sempre al return address sullo stack**. Poichè non si muove, può essere usato per trovare le variabili. Se x_i è l' i -esimo parametro formale della funzione per cui il codice è generato, se tutte le variabili hanno taglia 4, andrò a prendere $4*i$ e lo aggiungo al frame pointer e poi lo metto all'accumulatore. Il frame pointer ci serve ad accedere a tutti i dati che sono nel contesto della funzione. Gli offset possono essere

poi salvati nella tabella dei simboli. Idea è quella di vedere la tabella dei simboli in funzione dei tipi indipendentemente dalla tabella.

Quando gestiamo il for, dobbiamo tenere a mente che possiamo definire una variabile all'interno del for stesso

