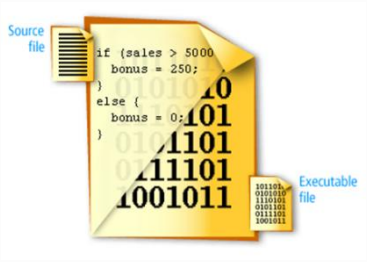
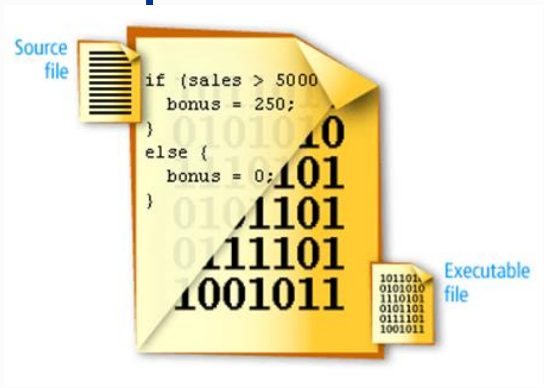
Compilatori

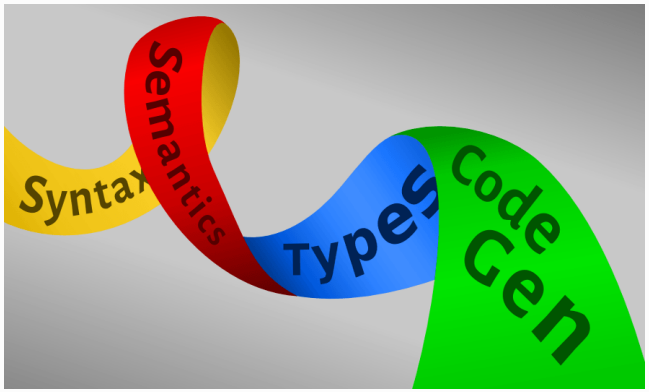


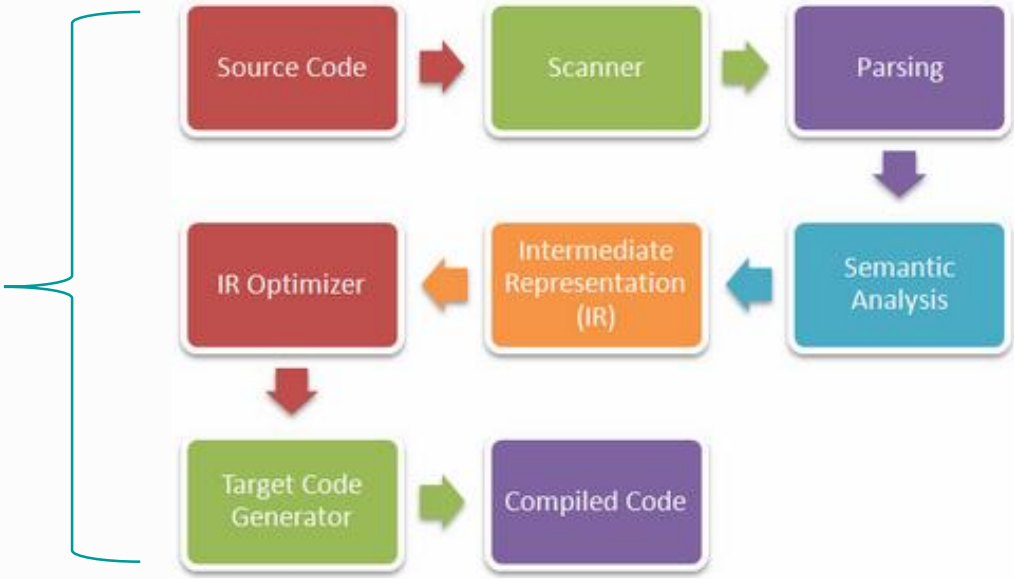
Compilers.
You can't live with them..
You can't live without them
Sounds familiar?

http://ksat.me/why-fantom-matters-1-the-compiler/

Compilatori







Compilatori-Esempio

Il problema attraverso un esempio

x := a * 2 + b * (x *3)

- Che cosa è sintatticamente questa frase?
- È un comando valido?
- Come si determina il significato?
  - Divisione in parole
  - Conversione in parole in comandi
  - Interpretazione del significato dei comandi

Trasformazioni successive

Compilatori-Esempio

Divisione in parole: ***Analisi Lessicale***

Input: X := a * 2 + b * (x *3)

Output: id<x> assign id<a> times int<2> plus id<b> times lpar id<x> times int<3> rpar

Fasi

- Raggruppare i caratteri della stringa di input in token
- Eliminare i caratteri superflui (spazi, new-lines, commenti…)
- Usare le espressioni regolari per la specifica e i DFA per l'implementazione
- **flex**

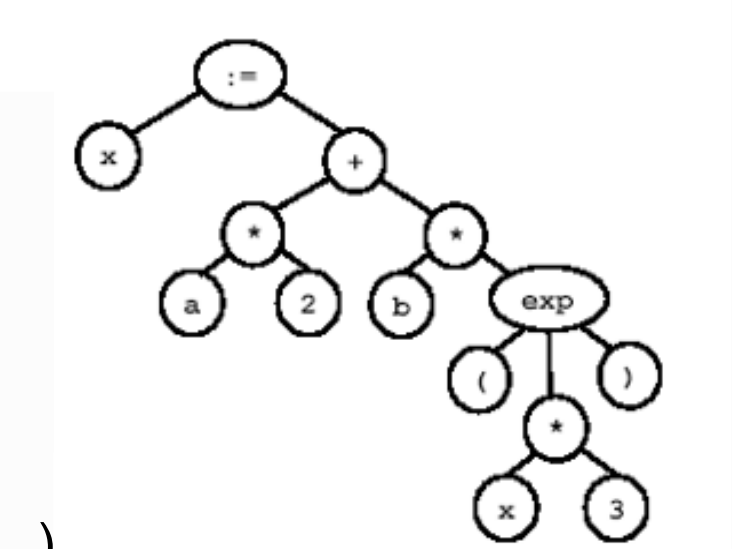
DFA: automi a stati finite deterministici: per ogni coppia di stato e simbolo in ingresso c'è una ed una sola transizione allo stato successivo.

Compilatori-Esempio

Conversione delle parole in comandi: Analisi Sintattica:

Input: id<x> assign id<a> times int<2> plus id<b> times lpar id<x> times int<3> rpar

Output: alberi sintattici



Fasi

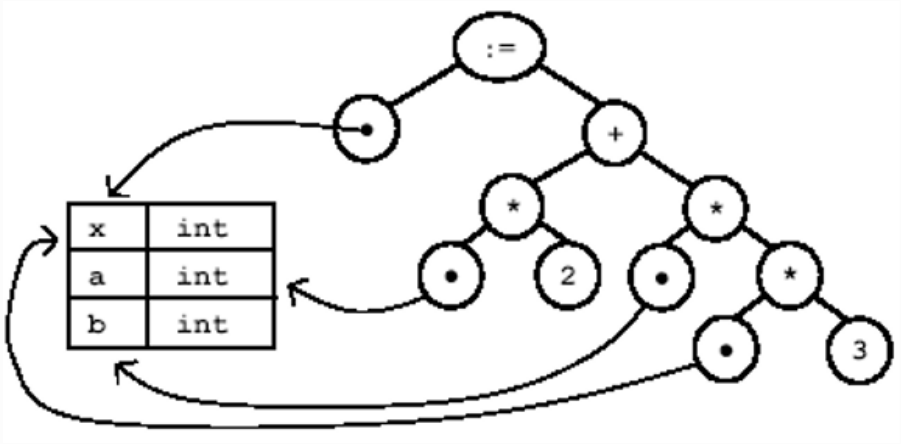
- Raggruppare i token della stringa di input in comandi
- Eliminare i token superflui (parentesi…)
- Usare i linguaggi context-free per la specifica e i push-down automata per l'implementazione
- **bison**

Compilatori-Esempio

Significato dei comandi: Analisi Semantica

Input: alberi sintattici

Output: alberi sintattici con tabella dei simboli



Fasi

- Connette le def. di variabili

al loro uso

- Verifica che ogni

espressione abbia il tipo coretto

- Controlla se è legale leggere/scrivere x
- Usa grammatiche con attributi, symbol tables, …

Compilatori-Esempio

Generazione di codice intermedio astratto: Traduzione

Input: alberi sintattici con tabella dei simboli

Output: rappresentazione intermedia più adatta per generare codice macchina (bytecode)

Fasi

- Definizione della sintassi astratta intermedia
- Diversi tipi di rappresentazione intermedia (gerarchica, lineare, ad albero, a triple)

Compilatori-Esempio

Generazione di codice macchina concreto: Selezione delle istruzioni

Input: rappresentazione intermedia

Output: codice macchina (target)

Fasi

- Implementazione della rappresentazione intermedia in un dato linguaggio assembler
- Scelta delle istruzioni tra quelle possibili
- Tipi di salti
- Utilizza grammatiche ad alberi e programmazione dinamica

```
r1 ← load   M[fp+x]
r2 ← loadi  3
r3 ← mul    r1, r2
r4 ← load   M[fp+b]
r5 ← mul    r3, r4
r6 ← load   M[fp+a]
r7 ← sl     r6, 1
r8 ← add    r6, r5
     store M[fp+x] ← r8
```

Compilatori-Esempio

generazione di codice macchina ottimizzato: Ottimizzazioni

```
INPUT:   codice macchina
OUTPUT:  codice macchina ottimizzato
```

fasi:

- migliorare il codice attraverso metriche: parametri possibili
  - ✓ lunghezza del codice
  - ✓ numero di registri
  - ✓ velocità
- strumenti: analisi del flusso, liveness, loop optimizations, etc.
- tipi di ottimizzazioni: leaf-functions, rimozione della tail recursion, etc.

Minimizzazione dei registri utilizzati

```
r1 ← load   M[fp+x]
r2 ← loadi  3
r1 ← mul    r1, r2
r2 ← load   M[fp+b]
r1 ← mul    r1, r2
r2 ← load   M[fp+a]
r2 ← sl     r2, 1
r1 ← add    r1, r2
     store M[fp+x] ← r1
```

Compilatori-Esempio

Allocazione dei Registri

```
INPUT:   codice macchina (ottimizzato)
OUTPUT:  codice in cui il numero dei registri utilizzato è
         inferiore a quello della macchina
```

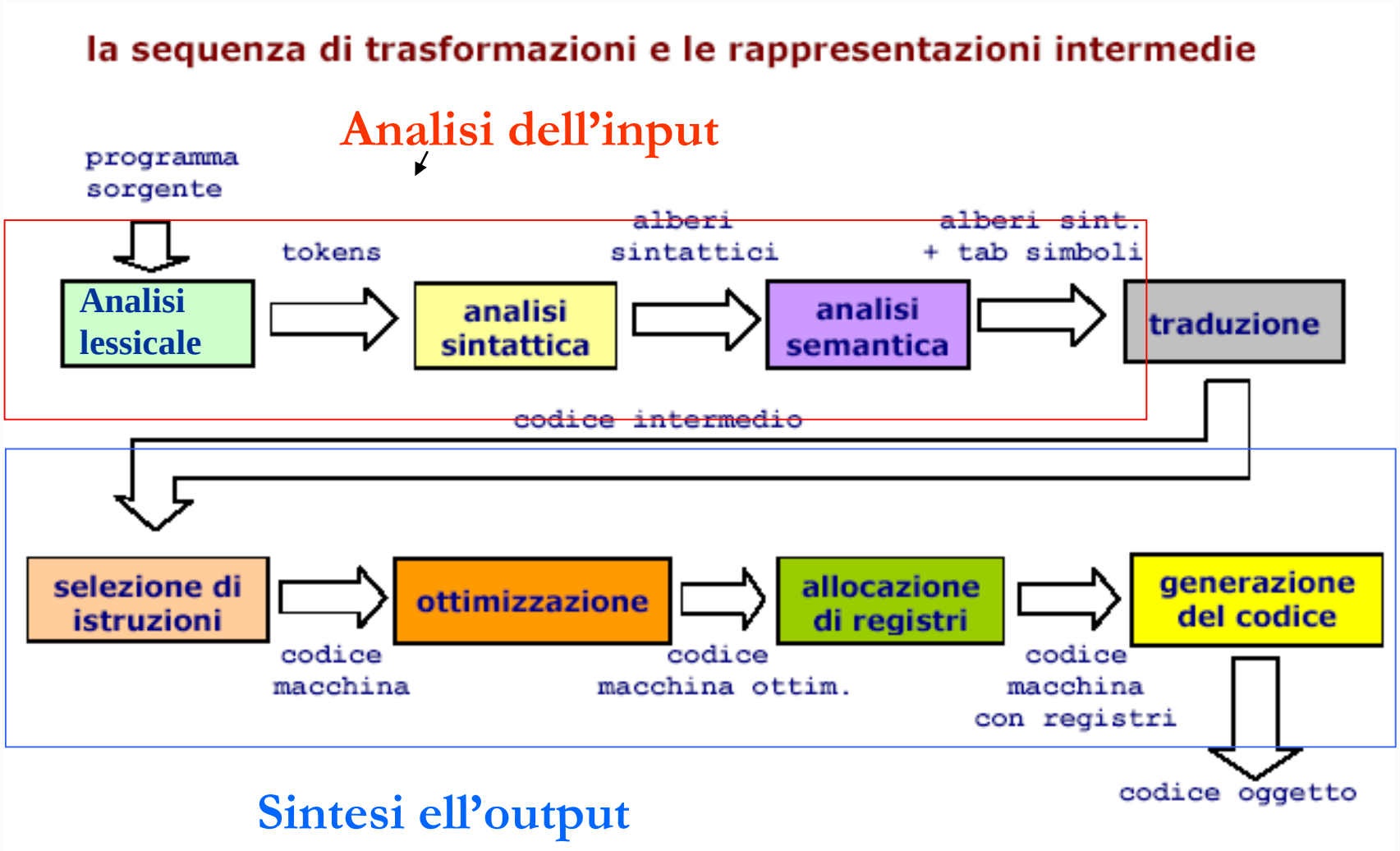
fasi:

- le fasi precedenti assumevano la presenza di un numero illimitato di registri
- utilizzare un numero finito di registri
- quando possibile, assegnare `SOURCE` e `DESTINATION` di una istruzione `MOVE` allo stesso registro, in modo da eliminarla
- tecnica utilizzata: grafi colorati

Generazione del Codice

si genera il codice oggetto, nel linguaggio assembler del processore

Compilatori-Esempio



Compilatori-Esempio

che cosa rende un compilatore "buono"?

- ✓ correttezza
- ✓ performance dei programmi tradotti
- ✓ scalabilità del compilatore
  - il compilatore è veloce
  - compilazione separata
- ✓ facile da modificare
- ✓ ausilio alla programmazione
  - messaggi di errore intelligibili
  - supporto per il debugging

Compilatori : analisi dell'input

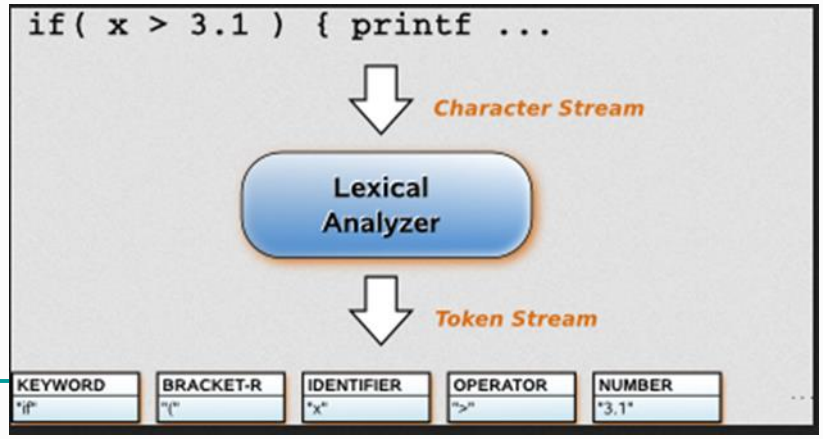
- √ Analisi Lessicale
  - Espressioni regolari
  - Lex
- √ Analisi Sintattica
  - Parser top-down e bottom-up
  - Yacc
- √ Analisi semantica
  - Guidata dalla sintassi nei L.d.P.

Analisi Lessicale

Analisi lessicale

Scopi

- data una sequenza di caratteri (stringa) di un alfabeto, verificare se la sequenza può essere decomposta in una sequenza di token (parole), tale che ogni token appartiene al lessico:
  - in caso positivo, restituire in uscita la sequenza di token
  - in caso negativo, restituire un errore lessicale



Analisi lessicale

Esempio

- *e*1: (a|b|...|z) (a|b|...|z|0|1|...|9)
- *e*2: if
- *e*3: else
- *e*4: {
- *e*5: }
- *e*6: *b* (spazio bianco)

➢ stringa in input:

if *b* pippo *b* {pluto} *b* else *b* paperino

➢ l'analizzatore restituisce la sequenza di token:

if, *b*, pippo, *b*, {, pluto, }, *b*, else, *b*, paperino

Analisi lessicale

Spesso, alcuni token non hanno importanza per la successiva fase di analisi sintattica e possono essere quindi non restituiti dall'analizzatore lessicale

- *nell'esempio precedente, è possibile che i simboli di spaziatura corrispondenti all'espressione e6 possano non essere restituiti (perché non significativi per la successiva fase di analisi sintattica)*
- in tal caso, la sequenza di token restituita dall'analizzatore lessicale è:
  - if, pippo, {, pluto, }, else, paperino

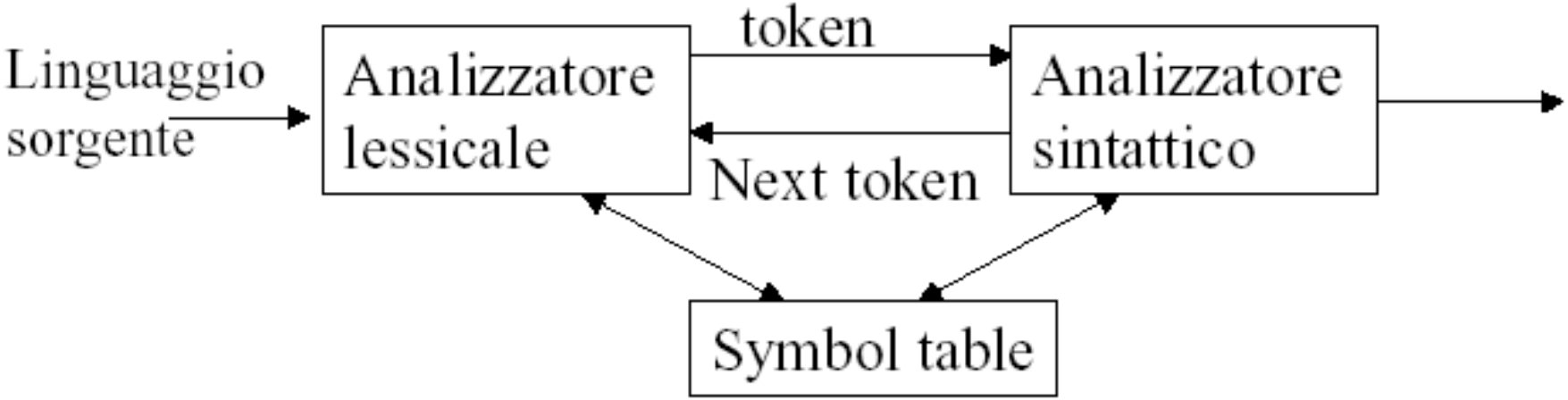
Analisi lessicale

In generale, nell'analisi lessicale vengono distinti diversi tipi (o classi) di elementi lessicali

- nel lessico dell'esempio precedente, ci sono 6 tipi di token diversi, corrispondenti alle 6 *espressioni regolari e1; : : : ; e6*
- ad ogni token viene associato il tipo corrispondente all'espressione regolare che riconosce il token
  - l'attribuzione dei tipi ai token facilita la successiva analisi sintattica

Analisi lessicale: applicazioni

L'analizzatore lessicale nel compilatore:



Strumenti: Lex e Yacc (flex/bison)

- Possono servire per risolvere problemi di analisi di formato
  - LEx analizzatore lessicale
  - Yacc analizzatore sintattico

- Lex e Yacc non sono in grado di generare un compilatore completo: richiedono codice C/C++ di supporto per le parti non strettamente sintattiche.
- Entrambi sono dei generatori di codice ad alto livello: normalmente generano codice C che poi deve essere compilato insieme al codice di supporto scritto esplicitamente.

Analisi lessicale: flex (Fast Lexical Analyzer)

- Flex utilizza per la specifica dell'analizzatore lessicale le espressioni regolari
- Flex genera uno ***scanner:***
  - Prende in input un file in un formato speciale (*flex.l*)
  - Genera un file C chiamato lex.yy.c che definisce una funzione ylex()
  - Per ottenerlo si deve compilarlo (lanciando il programma flex: flex flex.l)
  - E si genera l'eseguibile passando questo output al compilatore (gcc- o scanner lex.yy.c) che si può eseguire (./scanner)

https://github.com/westes/flex

Analisi lessicale: flex

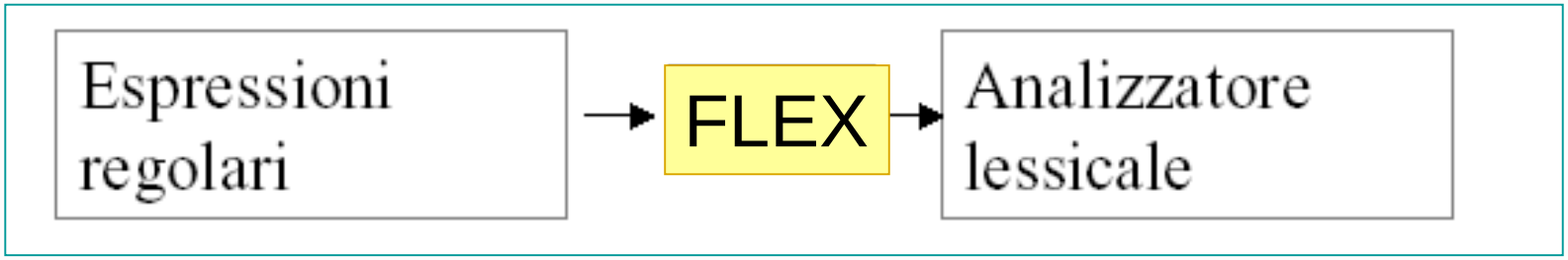
Azioni di riconoscimento in flex

- Ad ogni espressione regolare è associabile in flex un'azione che viene eseguita all'atto del riconoscimento (istruzioni in codice c, { }).
- Azione nulla si associa il carattere '; '.

Output del riconoscitore

- Il testo riconosciuto viene conservato nella variabile yytext,(char *).
- Il numero di caratteri riconosciuti viene memorizzato nella variabile yyleng, (int).
- Il testo non descritto da nessuna espressione regolare viene ricopiato in uscita, carattere per carattere

Analisi lessicale: Lex



L'input di flex viene diviso in tre parti, separate da %%.

dichiarazioni
%%
regole di transizione
%%
procedure ausiliarie

Flex produce un programma C, l'analizzatore lessicale che è dato dalla funzione int yylex().

Analisi Lex

Regole di transizione

- Comprende le espressioni regolari che vengono riconosciute dall'analizzatore lessicale.
  - Associato ad ogni espressione regolare c'è un pezzo di codice C che viene eseguito ogni volta che una espressione regolare viene riconosciuta.

esempio: *scanner di espressioni aritmetiche*

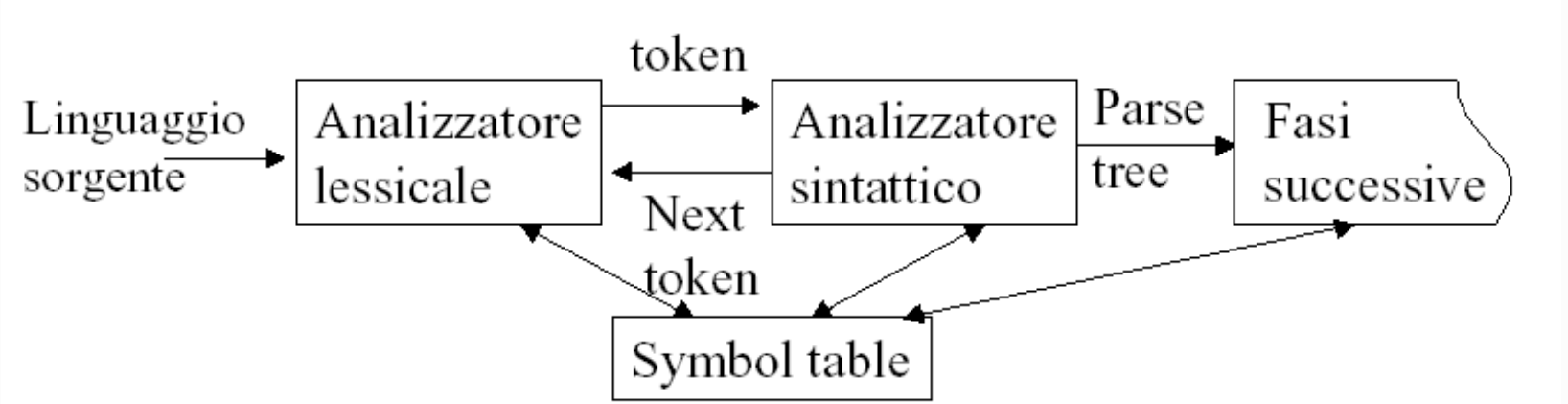
```
%{
#include <stdio.h>
%}
%%
[ \t\n]+                    printf("white space, length %i\n",yyleng);

"*"                         printf("times\n");
"/"                         printf("div\n");
"+"                         printf("plus\n");
"-"                         printf("minus\n");
"("                         printf("left-par\n");
")"                         printf("right-par\n");

0|([1-9][0-9]*)             printf("integer %s\n",yytext);
[a-zA-Z_][a-zA-Z0-9_]*      printf("identifier %s\n",yytext);
.                           printf("illegal char %s\n",yytext);
%%
main(){
      yylex();
}
yywrap(){       return(1) ;       /* serve per la terminazione dell' input */
}
```

Analisi sintattica

Analisi sintattica: i parser



- data una sequenza **s** di token, l'analizzatore sintattico verifica se la sequenza appartiene al linguaggio generato dalla grammatica *G. A* questo scopo, l'analizzatore sintattico cerca di costruire l'albero sintattico di *s*
  - in caso positivo, restituisce in uscita l'albero sintattico per la sequenza di input nella grammatica *G*
  - in caso negativo, restituisce un errore (errore sintattico)

Es: if (a == b) è corretto
if (a = b) è un errore di sintassi

Analisi sintattica

- Strumento essenziale di questa fase sono le **grammatiche context-free** (tipo 2)

- Due tipi di analisi sintattica:
  - analisi **top-down**: l'albero sintattico della stringa di input viene generato a partire dalla radice (assioma), scendendo fino alle foglie (simboli terminali della stringa di input)
  - analisi **bottom-up**: l'albero sintattico della stringa di input viene generato a partire dalle foglie (simboli terminali della stringa di input), risalendo fino alla radice (assioma)

Analisi sintattica

- **Caratteristiche/problematiche**
  - effettuare il riconoscimento di una stringa rispetto ad una grammatica è un'operazione intrinsecamente *non deterministica*
  - questo comporta in generale tempi esponenziali per risolvere il problema del riconoscimento, e quindi dell'analisi sintattica
  - tuttavia, l'analizzatore sintattico deve essere molto *efficiente*

Analisi sintattica

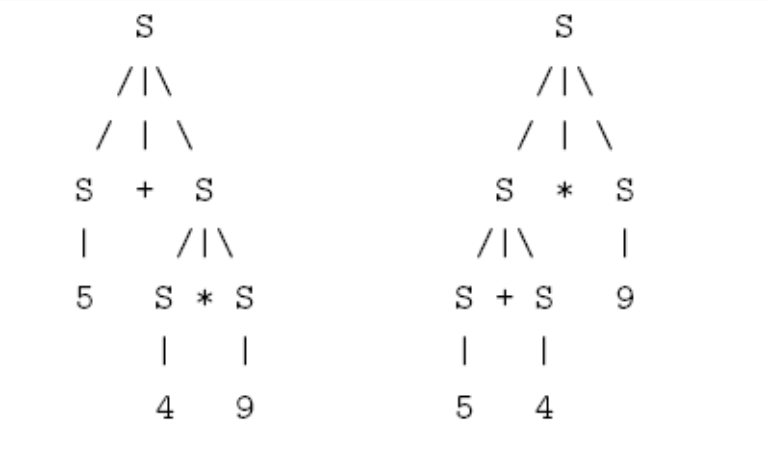
- In genere si richiede all'analizzatore sintattico di

1. girare in tempi lineari rispetto alle dimensioni della stringa di input
2. costruire l'albero sintattico leggendo pochi simboli della stringa di input per volta (tipicamente un solo simbolo della stringa per volta, detto simbolo di *lookahead*)

- a questo scopo, è necessario che la grammatica (in particolare, le regole di produzione) abbia delle caratteristiche che la rendono adatta all'analisi sintattica

Analisi sintattica

- **Grammatiche ambigue:**
  - Una grammatica *G* si dice *ambigua* se esiste una stringa che ammette due o più alberi sintattici distinti
    - Esempio: grammatica *G* per espressioni aritmetiche:
    - S→S S+S|S-S|S*S|S/S|(0|1|2|3|4|5|6|7|8|9)
    - la stringa *s* = 5 + 4 * 9 ammette due alberi sintattici distinti



Analisi sintattica

- Una grammatica ambigua non è adatta all'analisi sintattica (e alla traduzione automatica)

- Una grammatica è adatta all'analisi sintattica se per tale grammatica è possibile costruire un algoritmo (analizzatore sintattico) deterministico

  - Pertanto la forma delle regole di *G* deve essere tale da evitare il **non determinismo** (cioè l'analizzatore deve ***sempre sapere*** quale regola usare per espandere la radice dell'albero sintattico) nella **derivazione** (analisi top-down) **o** nella **riduzione** (analisi bottom-up) per la costruzione dell'albero sintattico

Analisi sintattica: **top-down**

- Generazione dell'albero sintattico dall'alto verso il basso (dalla radice alle foglie)
- ordine di espansione dei nodi da sinistra verso destra, corrispondente alla ***derivazione canonica sinistra*** della stringa di input

Analisi sintattica: top-down

- **derivazione canonica sinistra** = derivazione in cui ad ogni passo si espande il simbolo non terminale più a sinistra della forma di frase corrente

  - esempio: grammatica *G* con regole di produzione

    $F=\{S \rightarrow AB \mid cSc,\ A \rightarrow a,\ B \rightarrow bB \mid b\}$

  - derivazione canonica sinistra della stringa *cabbc*:

    $S \Rightarrow cSc \Rightarrow cABc \Rightarrow caBc \Rightarrow cabBc \Rightarrow cabbc$

(al terzo passo di derivazione si espande il simbolo non terminale A invece del non terminale B)

Analisi sintattica: top-down

- ***Problema del non determinismo***:
  - *come scegliere la regola per espandere un nodo non terminale in modo efficiente?*

Analisi sintattica top -down

- **Grammatiche LL(1)** (**L**eft to right, performing **L**eftmost derivation)
  - grammatica LL(*k*) = grammatica che ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando *k* simboli di lookahead e leggendo la stringa da sinistra verso destra
  - grammatica LL(1) = grammatica che ammette un riconoscitore deterministico che è in grado di costruire la derivazione canonica sinistra della stringa di input usando 1 simbolo di lookahead e leggendo la stringa da sinistra verso destra

Analisi sintattica:top-down

Osservazioni

- Non esistono tecniche sistematiche per ottenere grammatiche LL(1).
- Esistono linguaggi deterministici non ottenibili da grammatiche LL(1)
- Esistono condizioni sufficienti perché una grammatica non sia LL(1):
  - presenza di ricorsione sinistra
  - presenza di prefissi comuni

Analisi sintattica:top-down

- ricorsione sinistra = esiste un X ϵ VN tale che esiste una derivazione X =>Xα (con α ϵ VN)
- presenza di prefissi comuni = esistono due regole distinte A →aα , A →aβ (con a ϵ VT , α, β ϵ VN)
- entrambe queste caratteristiche possono essere eliminate, attraverso trasformazioni equivalenti di G, cioè trasformazioni delle regole di G che lasciano invariato il linguaggio L(G)

Analisi sintattica:top-down

- Presenza di prefissi comuni
  - ***Rimedio: fattorizzazione a sinistra***
  - Quando non è chiaro quale produzione usare per espandere un non terminale A guardando il primo simbolo, si possono riscrivere le produzioni in modo da posticipare la scelta introducendo un non terminale supplementare

- Es $A \rightarrow a\alpha \mid a\beta$ *fattorizzando* $A \rightarrow aA' \quad A' \rightarrow \alpha \mid \beta$
- <stmt>→ if <expr> then <stmt>
  | if <expr> then <stmt> else <stmt>

Fattorizzando <smt>→ if <expr> then <stmt><S>
<S>→ε | else <stmt>

Analisi sintattica:top-down

- L'analisi discendente non è applicabile se c'è ricorsione
  - ***Rimedio: elimazione ricorsione sinistra***
  - Per eliminare al ricorsione sinistra si procede come segue
    1. Si individuano le produzioni (e i non terminali) con ricorsione a sinistra
    2. Ogni produzione con ricorsione a sinistra si trasforma in ricorsione a destra introducendo nuovi simboli non terminali
  - Date le produzioni ricorsive di un non terminale A

E→E + num A|num

Si sostituiscono con

E→num E'

E'→+E|ϵ

Analisi sintattica: top-down

Esempio: grammatica per espressioni aritmetiche

grammatica iniziale:

$$E \rightarrow E + E \mid E * E \mid (\ E \ ) \mid n$$

1) trasformazione per eliminare l'ambiguità (si assegna precedenza all'operatore prodotto rispetto alla somma):

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (\ E \ ) \mid n$$

2) trasformazione per eliminare la ricorsione sinistra diretta:

$$E \rightarrow TE', \quad E' \rightarrow +TE' \mid \epsilon,$$
$$T \rightarrow FT', \quad T' \rightarrow *FT' \mid \epsilon,$$
$$F \rightarrow (\ E \ ) \mid n$$

Analisi sintattica: tabella di parsing

- L'analisi top-down per grammatiche LL(1) è basata sulla costruzione di una tabella, detta ***tabella di parsing*** *LL(1)*

- la **tabella di parsing** LL(1) è un array bidimensionale:
  - una riga per ogni simbolo non terminale
  - una colonna per ogni simbolo terminale
  - ogni cella della tabella (elemento dell'array) contiene una o più regole di produzione della grammatica
  - la presenza della produzione $A \rightarrow \alpha$ nella cella in posizione (A, a) significa che la regola $A \rightarrow \alpha$ può essere usata per espandere il simbolo non terminale A quando il simbolo di lookahead (cioè il simbolo corrente in input) è a

Analisi sintattica tabella di parsing

- **Esempio**

Data la seguente grammatica G per espressioni aritmetiche

$\{ E \to TE',\ \ E' \to +TE' \mid \epsilon,\ \ T \to FT'\ \ T' \to *FT' \mid \epsilon\ \ F \to (E) \mid n \}$

la tabella LL(1) per G è la seguente:

	n	$+$	$*$	$($	$)$	\$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to n$			$F \to (E)$		

Analisi sintattica: tabella di parsing

Altre parti del parser sono:

- **Input buffer** : Stringa di cui effettuare l'analisi sintattica. Si assume che termini con un simboli di fine stringa $.
- **Stack :** Può contenere i simboli terminali e non del linguaggio. Alla fine della pila c'è il simbolo speciale di fine stringa $. Inizialmente lo stack contiene il simbolo speciale $ e l'assioma S. Quando lo stack è vuoto il parsing è completato.
- **Output :** Una produzione rappresenta un passo della sequenza di derivazioni (left-most derivation) della stringa in un input buffer

Analisi sintattica tabella di parsing

- Il parser LL1 per la stringa di input ***n+n***

dove n è indicato id

stack	**input**	**output**
$E	id+id$	$E \rightarrow TE'$
$E'T	id+id$	$T \rightarrow FT'$
$E'T'F	id+id$	$F \rightarrow id$
$ E'T'id	id+id$	
$ E'T'	+id$	$T' \rightarrow \varepsilon$
$ E'	+id$	$E' \rightarrow +TE'$
$ E'T+	+id$	
$ E'T	id$	$T \rightarrow FT'$
$ E'T'F	id$	$F \rightarrow id$
$ E'T'id	id$	
$ E'T'	$	$T' \rightarrow \varepsilon$
$ E'	$	$E' \rightarrow \varepsilon$
$	$	accept

Data la seguente grammatica G per espressioni aritmetiche

$\{ E \rightarrow TE',\ E' \rightarrow +TE' \mid \epsilon,\ T \rightarrow FT'\ T' \rightarrow *FT' \mid \epsilon\ F \rightarrow (E) \mid n \}$

la tabella LL(1) per G è la seguente:

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

Analisi sintattica: **bottom-up**

- **analisi sintattica bottom-up** = l'albero sintattico della stringa di input viene generato a partire dalle foglie (simboli terminali della stringa di input), risalendo fino alla radice (assioma)
- in particolare, l'albero sintattico viene generato effettuando la ***riduzione canonica destra*** della stringa di input

Analisi sintattica: bottom-up

- Costruzione dell'albero di analisi dalle foglie risalendo (*riducendolo)* alla radice

- Tecnica ***shift-reduce****:* si scorre la stringa da sinistra a destra, ed ogni volta che una parte della stringa input coincide con la parte destra di una produzione viene sostituita col non terminale a sinistra.

$S \rightarrow aABc$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$abbcde$

$aAbcde$

$aAde$

$aABe$

S

Nell'esempio:

1. A->b
2. A->Abc
3. B->d
4. S→aABe

Analisi sintattica: bottom-up

- **Analisi ascendente con stack (automa)**
  - Inizialmente lo stack è vuoto e sull'input c'è la stringa da analizzare
  - **Shift** L'automa scandisce l'input da sinistra a destra mettendoli sullo stack.
  - **Reduce** Quando trova nello stack una parte "riducibile" (*handle*) la sostituisce col non terminale associato
  - L'automa si arresta quando non può più "*shiftare*" ne ridurre. Si ha:
    - Riconoscimento: tutta la stringa analizzata e sullo stack c'è l'assioma
    - Errore: tutti gli altri casi
- **L'automa è deterministico se non c'è mai più di una parte riducibile sullo stack**.

Analisi sintattica bottom up

esempio: si consideri la grammatica precedente

S→aABe

A→Abc|b

B→d

Simbolo di fine dell'input, anche se assente deve essere valutato

la frase *abbcde$* viene valutata come segue

- nello stato iniziale la pila è vuota
- l'automa esegue azioni, che oltre ad " **accept**" sono di due ulteriori tipi:
  - **shift**: un simbolo terminale è spostato dalla stringa di input sulla pila
  - **reduce**: una stringa α sulla pila è ridotta al non-terminale A, secondo la regola A →α ("**reduce** A →α")

Analisi sintattica bottom up

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

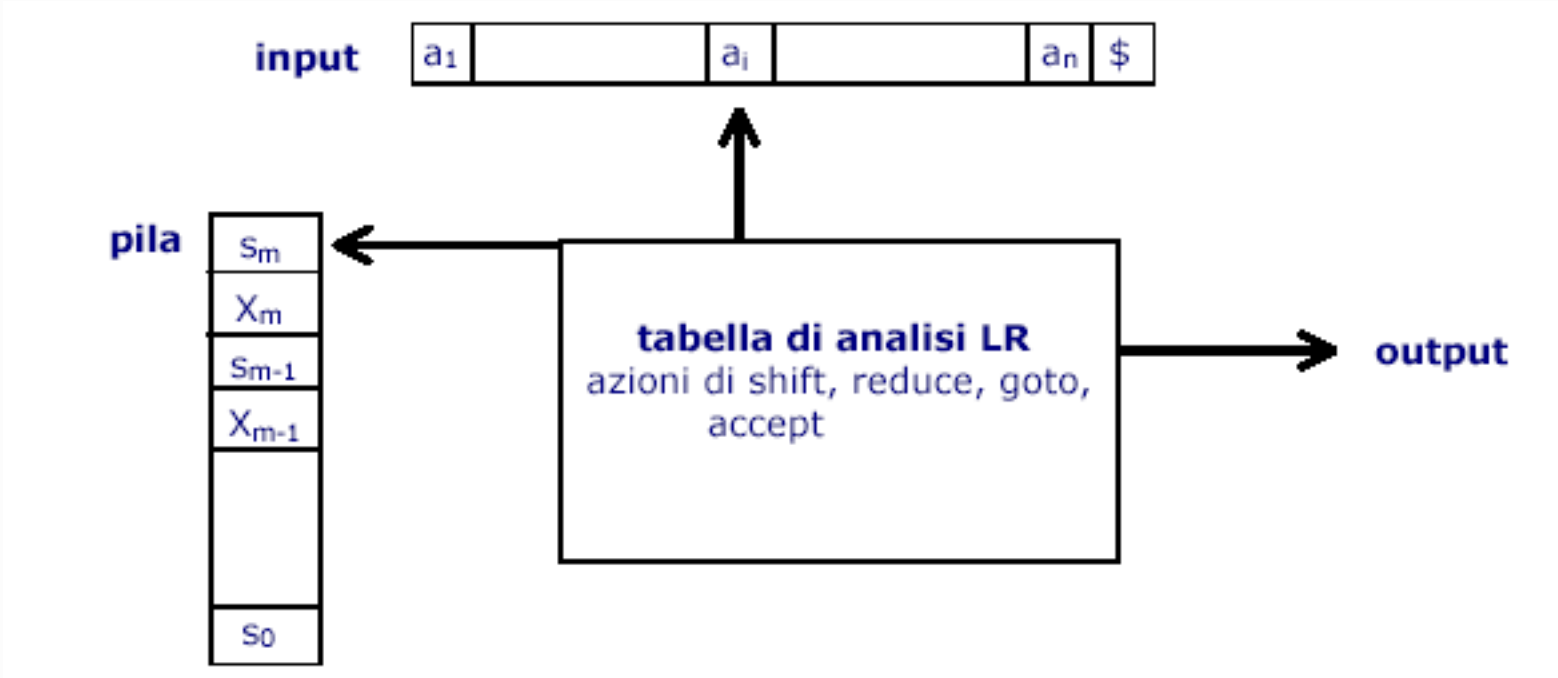
- Commenti
  - la sequenza di passi di sopra corrisponde alla derivazione rightmost:
  - S→aABe→aAde→aAbcde→abbcde

Analisi sintattica bottom up

- il parser bottom-up sposta sulla pila un numero di simboli di input finchè non è in grado di determinare l'azione da fare (vedi passi 2-3 e 5-6)
- il parser bottom-up può richiedere di guardare a fondo nella pila per individuare il tipo di azione (vedi passi 6 e 10)
- in particolare, per determinare le azioni ai passi 3 e 6 occorre ***conoscere lo stato della pila***, cioè ciò che si trova nella pila
- conoscere lo stato della pila non è problematico come l'input lookahead: ***si può memorizzare mediante stati***
  - per determinare la prossima azione da eseguire, lo stato della pila può non essere sufficiente: in questi casi occorre leggere dalla stringa di input un certo numero di simboli (**LR(k)** ha un lookahead di **k** simboli di input)

Analisi sintattica bottom up

- il ***calcolo*** è fatto tramite una tabella di analisi stati/simboli
- graficamente, l'automa può essere disegnato così:



Analisi sintattica :bottom-up

- Grammatiche LR(k):
  - grammatica LR(k) = grammatica che ammette un riconoscitore deterministico in grado di costruire la derivazione canonica destra della stringa di input usando k simbolo di lookahead e leggendo la stringa da sinistra verso destra
- Grammatiche LR(1):
  - grammatica LR(1) = 1 simbolo di lookahead

Analisi sintattica

- ***Confronto tra analisi LR(1) e analisi LL(1):***
  - l'analisi bottom-up è in generale più potente dell'analisi top-down, nel senso che la classe delle grammatiche LR(1) è più ampia della classe delle grammatiche LL(1)
  - questo è dovuto al fatto che la strategia bottom-up sfrutta in modo maggiore l'informazione costituita dalla stringa di input (simboli di lookahead), perché la riduzione parte dai simboli terminali (foglie dell'albero sintattico) invece che dall'assioma della grammatica

Analisi sintattica: **i parser LR(k)***

- L'analisi LR è usata nei linguaggi di programmazione.
- Infatti:

1. Esistono tool automatici per generare parser di questo tipo
2. La classe delle grammatiche LR include quelle LL
3. Riconoscimento immediato degli errori

L" indica che l'input è analizzato **da sinistra a destra**
R" indica che è prodotta una derivazione **rightmost**
(riscriviamo sempre il non terminale più a destra)
k" indica che sono usati **k simboli di lookahead**
Generalmente k=1

* *Donald Knuth – 1965, lo stesso che nel 1978 creò il TeX*

Analisi sintattica: *i parser LARL(1)**

- **Look-Ahead LR parser**
- Sono una versione semplificata e più efficienti dei parser LR canonici

- **Yacc** e **GNU bison** sono generatori di parser di tipo LARL(1)

- sono dei programmi che generano dei parser (programmi scritti anche C, C++ o java) che leggono una sequenza di token e decidono se questa sequenza di token è conforme alla sintassi specificata dalla grammatica)

* Frank DeRemer, 1969

Analisi sintattica: **yacc bison**

- Il file da dare in input al generatore di parser è del tipo :

  dichiarazioni
  %%
  regole
  %%
  procedure ausiliarie

Oss. : YACC sta per
Yet Another Compiler Compiler)

Analisi sintattica: Yacc -bison

Parte dichiarazioni

- Definizione dei token del linguaggio

%token DIGIT

- Informazioni di precedenza e di associatività

%left ‘+’ ‘-’

%left ‘*’ ‘/’

%rigth UMINUS

%nonassoc ‘<‘

- Simbolo di start (solo uno)

%start <S>

- Il codice ‘c’ va racchiuso

tra “%{ … }%”

Ordine di precedenza dal basso verso
l’altro. ‘+’,’-’ sono associativi a sinistra, ed hanno lo stesso livello di precedenza. L’ordine di precedenza è
UMINUS Þ * / Þ + -. Gli operatori non associativi non possono essere combinati a due a due.

Analisi sintattica: Yacc -bison

Parte regole

- Struttura:

<non_terminale> **:** *produzione* 1 {*azione semantica* 1} **|**
produzione 2 {*azione semantica* 2} **|**
…
produzione n {*azione semantica n*}
;

- *Produzione* = rappresenta la modalità costruttiva del simbolo non terminale, attraverso la sequenza di uno più simboli sia terminali che non terminali che concorrono a costruirlo.
- *Azione semantica* = ad ogni regola può essere associata un'azione che verrà eseguita ogni volta che la regola venga riconosciuta

Analisi sintattica: Yacc -bison

Parte regole

- Struttura:

<non_terminale> **:** *produzione* 1 {*azione semantica* 1} **|**

produzione 2 {*azione semantica* 2} **|**

...

produzione n {*azione semantica n*}

;

- *azione semantica* = Le azioni sono un blocco istruzioni “c”; Le azioni possono scambiare dei valori con il parser tramite delle pseudo-variabili introdotte dai simboli ($$, $1, $2, ...)
- La pseudo-variabile $$ è associata al lato sinistro della produzione mentre le pseudo variabili $n sono associate al non terminale di posizione n nella parte destra della produzione.

Analisi sintattica: Yacc -bison

Parte regole

- Struttura:

A : B {$$ = 1} C {$1 = 2; $2 = 12}

E : E ‘+’ DIGIT { $$ = $1 + $3;}

- $$ valore associato al non terminale a sinistra; $i si riferisce al valore termine i-mo della produzione (terminale o non terminale). Nella azione semantica generalmente si calcola il valore di $$ in funzione dei $i.
- I terminali vanno racchiusi tra apici, se non specificati come TOKEN. L’analizzatore lessicale si fa carico di riconoscere i TOKEN.

Analisi sintattica: Yacc -bison

Procedure ausiliarie

- La parte procedure ausiliarie contiene generalmente la funzione main e altre routine di supporto. Il parser è racchiuso nella funzione yyparse(), che restituisce 0 in assenza di errori, 1 altrimenti.
  - L'analizzatore lessicale può essere creato con LEX/flex:
    - *incluso nel file nella parte procedure*:

#include "lex.yy.c"

    - *effettuando il link a parte*

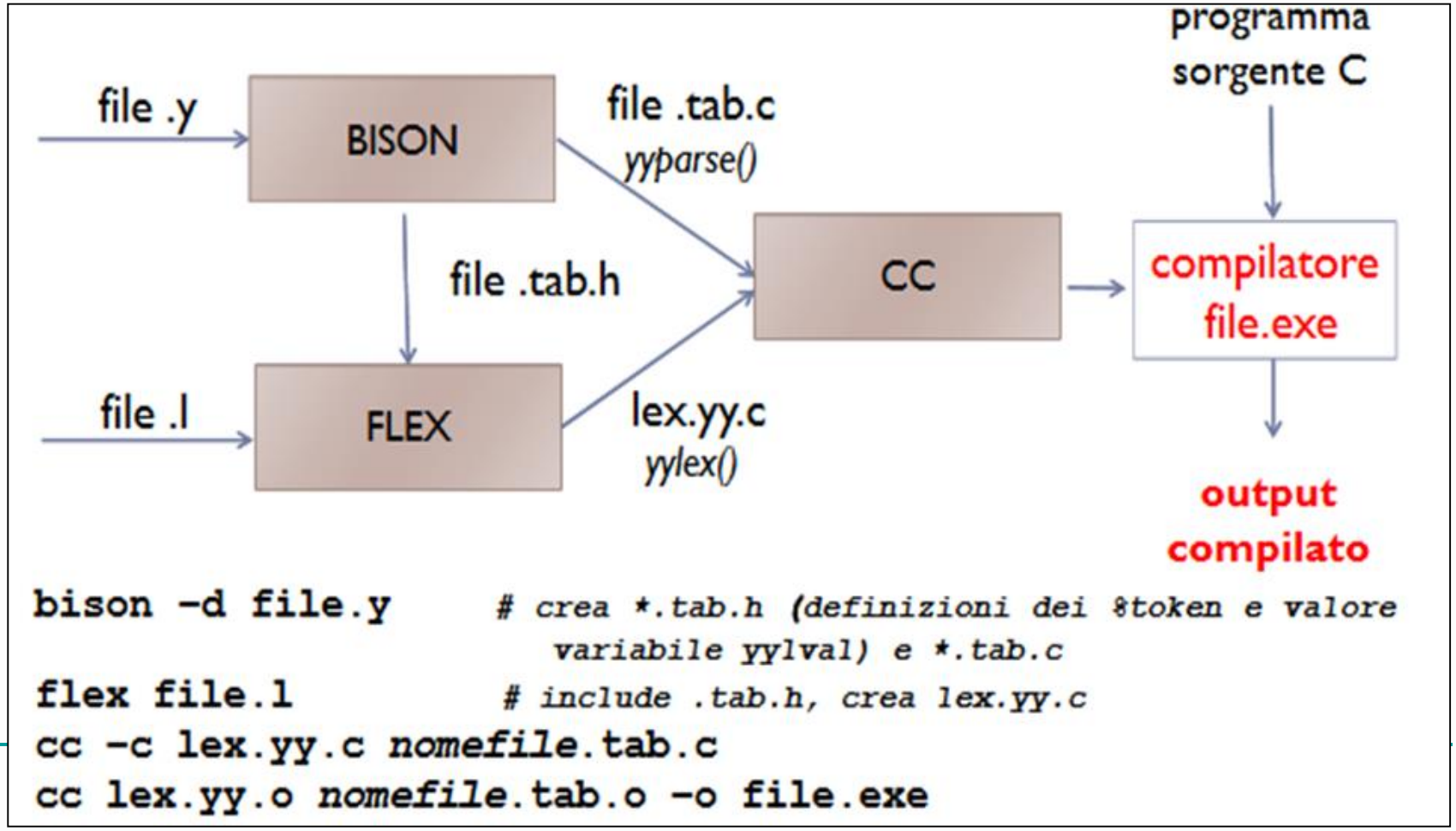
lex lexical.l

yacc syntax.y

cc y.tab.c -ly -ll

Lex /Yacc (Flex/Bison)

- Homepage di http://flex.sourceforge.net/
- Homepage di http://www.gnu.org/software/bison/




```
bison -d file.y      # crea *.tab.h (definizioni dei %token e valore
                       variabile yylval) e *.tab.c
flex file.l          # include .tab.h, crea lex.yy.c
cc -c lex.yy.c nomefile.tab.c
cc lex.yy.o nomefile.tab.o -o file.exe
```

Analisi sintattica:Yacc

```
%{
#include <ctype.h>
%}
%token DIGIT
%%
line : expr '\n' { printf("%g\n", $2); }
       ;
expr : expr '+' term { $$ = $1 + $3; }
       | term
        ;
term : term '*' factor { $$ = $1 * $3; }
        | factor
        ;
factor : '(' expr ') '{ $$ = $2; }
          | DIGIT
          ;
%%
#include “lex.yy.c”
```

Regole yacc

```
%%
int yylex(void)
{
int c;
while ((c = getchar()) == ' ');
if (isdigit(c)) {
yyval=c-’0’;
return DIGIT;
}
return c;
}
```

Parte lex

Regole

E→E+T

T →T*F

***F →E* | DIGIT**

Parser esempio aritmetica in Python

- https://github.com/tomamic/fondinfo/blob/master/examples/i2_infix_eval.py
- https://github.com/tomamic/fondinfo/blob/master/examples/i2_infix_to_prefix.py

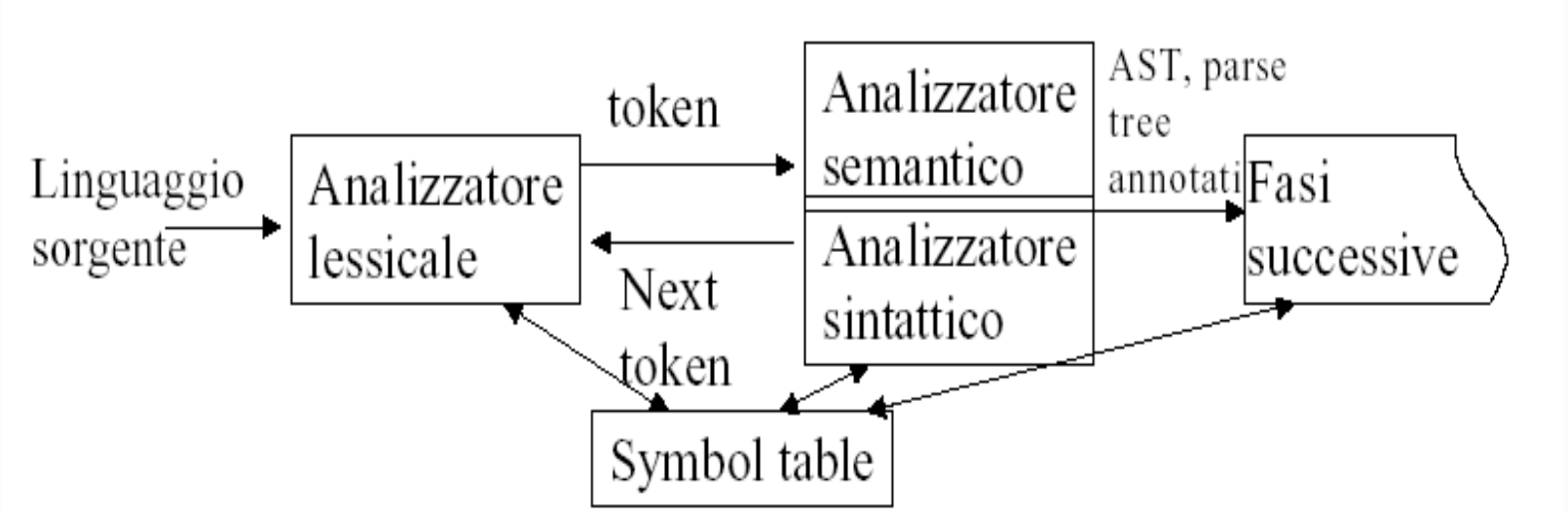
Analisi semantica

Analisi semantica: scopi

Per poter essere usati nelle fasi successive della compilazione, le derivazioni (alberi sintattici) devono conservare informazioni di natura semantica sui costrutti della grammatica.

- Il punto di partenza sono le produzioni della grammatica:
  - *ai simboli di una produzione vengono associate le informazioni semantiche sotto forma di attributi*.
- Le strutture semantiche si definiscono attraverso schemi sintattici.
  - *Con gli schemi di traduzione si può creare opportunamente l'albero sintattico di una frase.*
- ***L'analisi semantica integra l'analisi sintattica o traduce alberi sintattici in strutture utili alle elaborazioni successive***

Analisi semantica



AST
Abstract Syntax Tree

Analisi semantica introduce

- Una verifica della correttezza nell'impiego degli identificatori e dei costrutti del linguaggio (a compile-time):
  - identificatori dichiarati prima dell'uso ed in modo univoco
  - compatibilità operandi
  - chiamate congruenti con le dichiarazioni (visibilità)
- segnalazione degli eventuali errori semantici non appena possibile per evitare sia di riportare lo stesso errore sia di segnalare errori in cascata

Analisi Semantica controlli

Dato un programma sintatticamente corretto

=>Si fanno controlli statici del programma per esser certi della sua correttezza

-cioè che il programma possa produrre un valore

I controlli sono "context sensitive"

-es: una variabile occorre solo se è dichiarata

Controlli sui tipi

Un operatore o funzione sia invocato su argomenti di tipo opportuno

Controlli matematici

Divisioni per 0

Controlli sul flusso

Verifica se è legale l'occorrenza di un break

Controlli di unicità

Variabile dichiara una sola volta all'interno della suo scope