UNIVERSITÀ DEGLI STUDI DI SALERNO

Corso di Laurea Magistrale in Informatica

Progetto di corso

Documentazione NewLang

Team member:
Carmine D'ANGELO

Team member:
Matteo DELLA ROCCA

COMPILATORI - ANNO ACCADEMICO 2022/2023

Indice

Capitolo 1

Analisi lessicale

Questa fase viene eseguita sfruttando un analizzatore lessicale (Lexer) scritto in flex e compilato con Jflex (uno strumento di supporto open source per generare Lexer in Java).
È composto da un singolo file sorgente .flex contenente tutta la logica per generare una classe Lexer, che è anche fondamentale per la fase successiva.
Il Lexer risultante dalla compilazione del file .flex presenta tutte le funzionalità di un analizzatore lessicale. In questo capitolo saranno esposte le specifiche lessicali del linguaggio con le rispettive implementazioni in flex.

1.1 Specifiche lessicali

1.1.1 Spazi bianchi e caratteri di escape

Gli spazi bianchi o caratteri di escape non sono stati tokenizzati, ma ignorati.

```
LineTerminator =  \r|\n|\r\n
InputCharacter =    [^\r\n] \\
Whitespace     =    {LineTerminator} | [ \t\f] \\
{Whitespace}   {} //Ignora spazi bianchi o escape
```

1.1.2 Commenti

I commenti non sono stati tokenizzati, ma ignorati. Esistono due tipi di commenti:

- Commento tradizionale
- Commento su singola linea

```
TraditionalComment  = "|*" [^*] ~"|" | "|*"+ "|" \\
LineComment     = "||" {InputCharacter} LineTerminator}? \\
Comment = {TraditionalComment} | {LineComment}
{Comment} {} //Ignora commenti
```

1.1.3 Letterali

Per letterali si intendono insiemi di caratteri alfanumerici; in genere utilizzati per la costruzione degli identificatori e, nel contesto numerico, per i numeri. Se ne distinguono di due tipi:

- Letterali con underscore;
- Letterali senza underscore;

```
letter = [a-zA-Z]
letter_$ = [$\_a-zA-Z]
zero = [0]
digit = [0-9]
digit1 = [1-9]
digits = {digit}+
```

1.1.4 Identificatori

Per identificatori si intendono insiemi di letterali che danno un nome alle variabili di un linguaggio.

```
Identifier = ({letter_$}|({letter_$}+{digit}*)*)
```

1.1.5 Numeri

Per numeri si intendono insiemi di letterali numerici. Se ne distinguono di due tipi:

- **Numeri interi positivi**, dove per interi si intendono i numeri senza parte decimale e senza segno;
- **Numeri reali**, dove per reali si intendono numeri con parte decimale.
  Nota: *per delimitare la parte intera da quella decimale è stato utilizzato il carattere '.'*

```
FloatNumber = (({digit1}+{digit}* | {zero}))(\.{digits})?
IntegerNumber = (({digit1}+{digit}* | {zero}))
INTEGER_CONST = {IntegerNumber}
REAL_CONST = {FloatNumber}
```

1.1.6 Stringa letterale

Implementazione in Jflex della stringa letterale. Token corrispondente: STRING_CONST.

```
%state STRING_STATE
<STRING_STATE> {
    \"              {yybegin(YYINITIAL); return symbol("STRING_CONST",
        STRING_CONST, string.toString(), string.length()); }
    [^\n\r\"\\]+    { string.append( yytext() ); }
    \\t             { string.append('\t'); }
    \\n             { string.append('\n'); }
    \\r             { string.append('\r'); }
    \\\"            { string.append('\"'); }
    \\              { string.append('\\'); }
    <<EOF>>         {yybegin(YYINITIAL); error("La stringa non e' stata
        correttamente chiusa!! ;)");}
}
```

1.1.7 Parole chiave

Sono previste le seguenti parole chiave:

- MAIN = "start:"
  La parola chiave "start:" permette di etichettare una funzione come principale, ovvero eseguita all'inizio del programma.

- IF = "if"
  La parola chiave "if" viene utilizzata per dichiarare il costrutto condizionale if.

- THEN ="then"
  La parola chiave "then" viene utilizzata in combo con l'if per etichettare l'inizio del corpo da seguire se la condizione dell'if è verificata.

- ELSE = "else"
  La parola chiave "then" viene utilizzata in combo con l'if per etichettare l'inizio del corpo da seguire se la condizione dell'if non è verificata.

- WHILE = "while"
  La parole chiave "while" viene utilizzata per far eseguire al programma dei cicli una volta stabilita una condizione.

- LOOP = "loop"
  La parole chiave "loop" viene utilizzata in combo coi costrutti "while" e "for" per etichettare l'inizio del corpo da eseguire ciclicamente in base alla condizione.

- TO = "to"
  La parole chiave "to" viene utilizzata in combo col costrutto "for" per definire la condizione la cui veridicità comporterà la prosecuzione del ciclo.

- FOR = "for"
  La parole chiave "for" viene utilizzata per far eseguire al programma dei cicli una volta stabilito un valore iniziale ed un valore finale.

- DEF= "def"
  La parole chiave "def" viene utilizzata per dichiarare funzioni.

- OUT ="out"
  La parole chiave "out" viene utilizzata per etichettare un parametro della funzione che sfrutta il passaggio per riferimento.
- RETURN = "return"
  La parole chiave "return" viene utilizzata per ritornare un valore.
- TRUE = "true"
  La parole chiave "true" viene utilizzata per indicare un letterale booleano che ha valore *true*;
- FALSE = "false"
  La parole chiave "false" viene utilizzata per indicare un letterale booleano che ha valore *false*;

Tipi primitivi

- INTEGER = "integer"
  La parole chiave "integer" viene utilizzata per indicare il tipo intero;
- FLOAT = "float"
  La parole chiave "float" viene utilizzata per indicare il tipo floating point;
- VAR = "var"
  La parole chiave "var" viene utilizzata per indicare un tipo la quale determinazione viene effettuata in automatico dal compilatore;
- BOOL = "boolean"
  La parole chiave "boolean" viene utilizzata per indicare il tipo boolean;
- STRING = "string"
  La parole chiave "string" viene utilizzata per indicare il tipo string;
- CHAR = "char"
  La parole chiave "char" viene utilizzata per indicare il tipo char;
- VOID = "void"
  La parole chiave "void" viene utilizzata per indicare il tipo void;

1.1.8 Operatori

Operatori Input/Output

- READ = "<- -"
  Utilizzato per leggere input da tastiera;
- WRITE = "- ->"
  Utilizzato per stampare a schermo, senza ritorno a capo;
- WRITELN = "- ->!"
  Utilizzato per stampare a schermo, con ritorno a capo;

Operatori relazionali

- LT = "<"
- GT = ">"
- LQ = "<="

- GQ = ">="
- NE = "!=" o NE = "<>"
- EQ = "="
- ASSIGN = "< <"
  Utilizzato per assegnare un valore ad una variabile.

Operatori aritmetici

- PLUS = "+"
- MINUS = "-"
- TIMES = "*"
- DIV = "/"
- POW = "∧"

Operatori su stringhe

- STR_CONCAT = "&"
  Utilizzato per concatenare stringhe.

Operatori logici

- AND = "and"
- OR = "or"
- NOT = "not"

1.1.9 Separatori

- COLON = ":"
- SEMI = ";"
- COMMA = ","
- PIPE = "|"
- LPAR = "("
- RPAR = ")"
- LBRAC = "{"
- RBRAC = "}"

1.1.10 Gestione errori lessicali

Per tutto ciò che non è presente nelle specifiche lessicali, l'analizzatore si comporterà lanciando un eccezione di tipo *LexicalError*, descrivendo il carattere non valido e la locazione dell'errore.

Capitolo 2

Analisi sintattica

Questa fase viene eseguita sfruttando un parser (Parser) scritto in cup e compilato con JavaCup (uno strumento di supporto open source per generare Parser in Java). Questa parte è composta da un singolo file sorgente .cup contenente tutta la logica per generare una classe Parser, che è anche fondamentale per la fase successiva. Il Parser risultante dalla compilazione del file .cup presenta tutte le funzionalità di un parser, inoltre sfruttando le funzionalità di cup e l'utilizzo degli attributi è stato costruito l'albero sintattico. In questo capitolo saranno esposte le specifiche sintattiche del linguaggio (riportando il contenuto del file .cup) e le modifiche effettuate.

2.1 Specifiche sintattiche

2.1.1 Grammatica originale

```
Program -> DeclList MainFunDecl DeclList
;
DeclList -> VarDecl DeclList | FunDecl DeclList | /* empty */
;
MainFunDecl -> MAIN FunDecl
;
VarDecl ::= Type IdInitList SEMI
                      | VAR IdInitObblList SEMI
;
Type ::= INTEGER | BOOL | FLOAT | STRING | CHAR
;
IdInitList ::= ID
                | IdInitList COMMA ID
                | ID ASSIGN Expr
                | IdInitList COMMA ID ASSIGN Expr
;
IdInitObblList ::= ID ASSIGN Const
                | IdInitObblList COMMA ID ASSIGN Const
;
Const ::= INTEGER_CONST | REAL_CONST | TRUE | FALSE | STRING_CONST | CHAR_CONST
;
FunDecl ::= DEF ID LPAR ParamDeclList RPAR COLON TypeOrVoid Body
;
```

```
Body ::= LBRAC VarDeclList StatList RBRAC
;
ParamDeclList ::= /*empty */
                | NonEmptyParamDeclList
;
NonEmptyParamDeclList ::= ParDecl
                    | NonEmptyParamDeclList PIPE ParDecl
;
ParDecl ::= Type IdList
        | OUT Type IdList
;
TypeOrVoid ::= Type | VOID
;
VarDeclList ::= /* empty */
| VardDecl VarDeclList
;
StatList ::= Stat | Stat StatList
;
Stat ::= IfStat | ForStat
        | ReadStat SEMI | WriteStat SEMI
        | AssignStat SEMI | WhileStat
        | FunCall SEMI | RETURN Expr SEMI
        | RETURN SEMI | /* empty */
;
IfStat ::= IF Expr THEN Body Else
;
Else ::= /* empty */ | ELSE Body
;
WhileStat ::= WHILE Expr LOOP Body
;
ForStat ::= FOR ID ASSIGN INTEGER_CONST TO INTEGER_CONST LOOP Body
;
ReadStat ::= IdList READ STRING_CONST
        |  IdList READ
;
IdList ::= ID | IdList COMMA ID
;
WriteStat ::=  LPAR ExprList RPAR WRITE
            |  LPAR ExprList RPAR WRITELN
;
AssignStat ::=  IdList ASSIGN ExprList
;
FunCall ::= ID LPAR ExprList RPAR
        | ID LPAR RPAR
;
ExprList ::= Expr
        | Expr COMMA ExprList
;
Expr ::= TRUE | FALSE
        | INTEGER_CONST | REAL_CONST
        | STRING_CONST | CHAR_CONST
        | ID | FunCall | Expr  PLUS Expr
        | Expr  MINUS Expr
        | Expr  TIMES Expr
        | Expr  DIV Expr
        | Expr  AND Expr
        | Expr POW Expr
        | Expr STR_CONCAT Expr
        | Expr  OR Expr
        | Expr  GT Expr | Expr  GE Expr | Expr  LT Expr
        | Expr  LE Expr | Expr  EQ Expr | Expr  NE Expr
        | MINUS Expr | NOT Expr | LPAR Expr RPAR
;
```

2.1.2 Conflitti nella grammatica originale

La grammatica originale soffriva di due tipi di conflitti.

- Shift/Reduce
- Reduce/Reduce

Shift/reduce

Per risolvere i conflitti shift/reduce che si verificavano nelle produzioni del non terminale Expr, non abbiamo modificato realmente la grammatica, ma sono state utilizzate le precedenze:

```
precedence right NOT;
precedence right ASSIGN;
precedence left OR, AND;
precedence nonassoc LT, GT, LE, GE, EQ, NE;
precedence left  STR_CONCAT;
precedence left PLUS, MINUS;
precedence left TIMES, DIV, POW;
precedence left MINUS;
```

Reduce/reduce

La risoluzione dei conflitti reduce/reduce ha apportato delle modifiche alla grammatica:

- Nel non terminale Stat è stata eliminata la produzione che andava a vuoto, questo ha portato ad avere nel non terminale Body un'altra produzione senza StatList;
- Nel non terminale VarDecList è stata eliminata la produzione che andava a vuoto, questo ha portato da avere in VarDecList una produzione che va solo a VarDecl, è stata aggiunta anche una produzione al non terminale Body in cui non è presente VarDecList;
- Visto che sia StatList e VarDecList possono scomparire nel non terminale Body è stata aggiunta una produzione in cui non sono presenti;
- Il non terminale Else è stato eliminato e la sua produzione è stata portata nel non terminale IfStat;
- A FunDecl è stata aggiunta una produzione senza ParamDeclList visto che quest'ultima poteva scomparire.

Modifiche per ottimizzare la costruzione dell'albero

- La produzione NonEmptyParamDeclList è stata eliminata e le sue produzioni sono state portate in ParamDecList;
- In Expr la produzione con FunCall è stata eliminata e sostituita direttamente con le produzioni del non terminale FunCall.

2.1.3 Grammatica modificata

```
Program ::= DeclList MainFunDecl DeclList
;
DeclList ::=   /* empty */
        | VarDecl:vd DeclList
        | FunDecl:fd DeclList
;
MainFunDecl ::= MAIN FunDecl
;
VarDecl ::= Type IdInitList SEMI
    | VAR IdInitObblList SEMI
;
Type ::=  INTEGER
    | BOOL
    | FLOAT
    | STRING
    | CHAR
;
IdInitList ::= ID
                | IdInitList:idl COMMA ID
                | ID ASSIGN Expr
                | IdInitList COMMA ID ASSIGN Expr
;

IdInitObblList ::= ID ASSIGN Const
                     | IdInitObblList COMMA ID ASSIGN Const
;
Const ::= INTEGER_CONST
            | REAL_CONST
            | TRUE
            | FALSE
            | STRING_CONST
            | CHAR_CONST
;

    FunDecl ::= DEF ID LPAR ParamDeclList RPAR COLON
    TypeOrVoid Body
               |  DEF ID LPAR RPAR COLON TypeOrVoid Body
;
 Body ::= LBRAC VarDeclList StatList RBRAC
          | LBRAC VarDeclList RBRAC
          | LBRAC StatList RBRAC
          | LBRAC RBRAC
;
ParamDeclList ::= ParDecl
             | ParamDeclList PIPE ParDecl
;
ParDecl ::= Type IdList
        | OUT Type IdList
;
TypeOrVoid ::= Type
        | VOID
;
VarDeclList ::= VarDecl VarDeclList
    | VarDecl
;
StatList ::= Stat
    | Stat StatList
;
```

```
Stat ::=  IfStat
            | ForStat| ReadStat SEMI
            | WriteStat SEMI | AssignStat SEMI
            | WhileStat | FunCall SEMI
            | RETURN Expr SEMI
            | RETURN SEMI
;
IfStat ::= IF Expr THEN Body
    | IF Expr THEN Body ELSE Body
;
WhileStat ::= WHILE Expr LOOP Body
;
ForStat ::= FOR ID ASSIGN INTEGER_CONST TO INTEGER_CONST LOOP Body
 ;
ReadStat ::= IdList READ STRING_CONST
        | IdList READ
;
IdList ::= ID
        | IdList COMMA ID
;
WriteStat ::= LPAR ExprList RPAR WRITE
        | LPAR ExprList RPAR WRITELN
;
AssignStat ::=  IdList ASSIGN ExprList
;
FunCall ::= ID LPAR ExprList RPAR
              | ID LPAR RPAR
;
ExprList ::= Expr
        | Expr COMMA ExprList
;
Expr ::= TRUE
        | FALSE
        | INTEGER_CONST
        | REAL_CONST
        | STRING_CONST
        | CHAR_CONST
        | ID
        | ID
        | ID LPAR RPAR
        | Expr  PLUS Expr
        | Expr  MINUS Expr
        | Expr  TIMES Expr
        | Expr  DIV Expr
        | Expr  AND Expr
        | Expr POW Expr
        | Expr STR_CONCAT Expr
        | Expr  OR Expr
        | Expr  GT Expr
        | Expr  GE Expr | Expr  LT Expr
        | Expr  LE Expr | Expr  EQ Expr
        | Expr  NE Expr | MINUS Expr
        | NOT Expr | LPAR Expr RPAR
;
```

2.1.4 Gestione errori sintattici

Per tutto ciò che non è presente nelle specifiche sintattiche, il parser si comporterà lanciando un eccezione di tipo *SyntaxError* e localizzando tramite riga e colonna l'errore.

2.2 Costruzione albero sintattico

La Grammatica modificata è stata poi arricchita con una serie di azioni, una per ogni produzione. Generalmente, queste azioni istanziano un oggetto di tipo Node relativo a ciascun elemento che appare nella parte destra della produzione stessa.

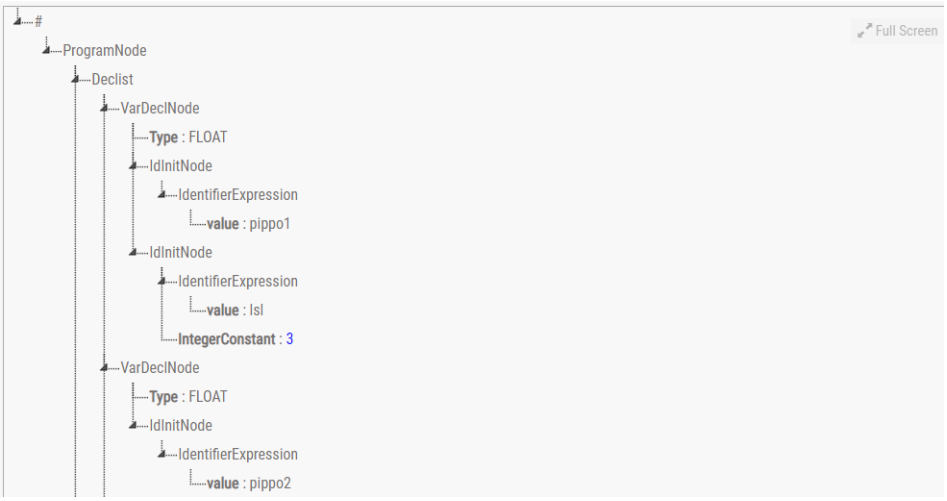
```
Program ::= DeclList:dl1 MainFunDecl:m DeclList:dl2
{:
              RESULT = new ProgramNode(dl1,m, dl2);
:}
;
...
IdInitList ::=   ID:i
{:
    List<IdInitNode> idl = new LinkedList<IdInitNode>();
    idl.add( new IdInitNode(new IdentifierExprNode(i,ixleft,ixright)));
    RESULT = idl;
:}
...
```

Man mano che il Parser elabora un dato file sorgente, viene generato il corrispondente (e unico) albero sintattico ricorsivamente. Con il completamento del processo, viene conservato un puntatore alla radice (ProgramNode) dell'albero di sintassi restituito, utile per la fase successiva o utile per altre eventuali azioni sull'albero.

Un modo per visualizzare l'albero sintattico

Una funzionalità aggiuntiva post-costruzione albero implementata è quella di visualizzare l'albero tramite XML. Il pattern Visitor si adatta perfettamente all'albero costruito ed una volta che il parser ha finito il suo lavoro, è possibile tramite una visita stampare tutti i nodi dell'albero in un file .xml sulla specifica istanza dell'albero sintattico. Una volta fatto ciò è possibille visualizzare il file utilizzando un qualsiasi Web Browser oppure consigliamo di incollare i tags all'interno del file sul seguente sito.

Figura 2.1: Un esempio di visualizzazione dell'albero sul sito consigliato...



Capitolo 3

Analisi semantica

La correttezza semantica del programma dato in input avviene attraverso due visite dell'AST generato dal parser:

1. Una visita per lo scoping (approfondita nella sezione 3.1);
2. Una visita per la correttezza semantica (approfondita nella sezione 3.2 e 3.3).

3.1 Scoping

Lo scoping del nostro linguaggio segue le seguenti regole:

1. Ogni volta che si accede in uno dei costrutti, che saranno di seguito elencati, si crea un nuovo scope, quindi è possibile sovrascrivere variabili dichiarate in scope precedenti (**shadowing)**.
   I blocchi che creano un nuovo scope sono i seguenti:
   - Scope globale;
   - Funzioni;
   - Costrutti: if, else, while, for.
2. È possibile utilizzare variabili che saranno dichiarate successivamente nello scope corrente

```
start: def main(){
    integer x << y;  // corretto: e' possibile
    integer y << 2;

    if(z == 3) then { // errore: non e' possibile
        integer z = 3;
    }
}
```

3. Durante questa fase viene anche inferito il tipo delle variabili dichiarate come var, assegnado il tipo della costante utilizzata:

   var x « 5 verrà assegnato integer, e alla fine nel codice c otterremo: int x = 5

4. È possibile scrivere assegnamenti ciclici nella dichiarazione di variabile, questi saranno risolti con l'inserimento di valori di deafult del C:

   - **integer** : valore di default 0;
   - **float** : valore di default 0.000000;
   - **char** : valore di default carattere vuoto ";
   - **string** : valore di default (null);
   - **boolean** : valore di default 0;

```
start: def main(){
    integer x << y;  // corretto: e' possibile
    integer y << x;
}
```

3.2 Regole d'inferenza

Di seguito verranno elencate le regole d'inferenza implementate per controllare la correttezza semantica. Alcune informazioni per leggere correttamente le regole semantiche:

- id: indica un identificatore;
- e: indica un'espressione, può essere sia un identificatore che un valore costante;
- $\wedge$ oppure uno spazio vuoto: è l'and logico;
- ———: significa allora;
- x: T oppure x : τ: x ha tipo T;
- $\vdash$: è possibile provare che;
- O oppure Γ: indica il type environmen.

3.2.1 Dichiarazioni

Identificatore

$$\frac{\Gamma(id)=\tau}{\Gamma \vdash id:\tau}$$

Costanti

$$\Gamma \vdash \text{int\_const} : \text{integer}$$

$$\Gamma \vdash \text{float\_const} : \text{float}$$

$$\Gamma \vdash \text{char\_const} : \text{char}$$

$$\Gamma \vdash \text{string\_const} : \text{string}$$

$$\Gamma \vdash \text{true} : \text{boolean}$$

$$\Gamma \vdash \text{false} : \text{boolean}$$

Costante var

$$\frac{\Gamma \vdash id : notype \quad \Gamma \ e{:}\tau \vdash \textbf{var}\ id << e}{\Gamma \vdash id : \tau}$$

3.2.2 Statement

Lista di istruzioni

$$\frac{\Gamma \vdash stmt_1 : notype \quad \Gamma \vdash stmt_2 : notype}{\Gamma \vdash stmt_1 ; stmt_2 : notype}$$

Blocco dichiarazione-istruzione

$$\frac{\Gamma[id \rightarrow \tau] \vdash stmt : notype}{\Gamma \vdash \tau ;\ stmt : notype}$$

Assegnazione di identificatori

$$\frac{\Gamma \vdash (id_i) : \tau^{i \in 1,\ldots,n} \quad \Gamma \vdash e_i : \tau_i^{i \in 1,\ldots,n}}{\Gamma \vdash id_1,\ldots,id_n << e_1,\ldots,e_n : notype}$$

Read statement

$$\frac{\Gamma \vdash (id_i) : \tau_i^{i \in 1,..,n} \quad s : string\_const}{\Gamma \vdash id_1,\ldots,id_n <-- s : notype}$$

$$\frac{\Gamma \vdash (id_i) : \tau_i^{i \in 1,..,n}}{\Gamma \vdash id_1,\ldots,id_n <-- : notype}$$

Writeln statement

- $$\frac{\Gamma \vdash e_i : \tau_i^{i \in 1,..,n}}{\Gamma \vdash e_1,\ldots,e_n --> ! : notype}$$

Write statement

- $$\frac{\Gamma \vdash e_i : \tau_i^{i \in 1,..,n}}{\Gamma \vdash e_1,\ldots,e_n --> : notype}$$

While statement

$$\frac{\Gamma \vdash e : boolean \quad \Gamma \vdash body : notype}{\Gamma \vdash \textbf{while}\ e\ \textbf{loop}\ body : notype}$$

For statement

$$\frac{\Gamma \vdash e_1 : int\_const \quad \Gamma \vdash e_2 : int\_const \quad \Gamma[id \rightarrow integer] \vdash body : notype}{\Gamma \vdash \textbf{for}\ id << e_1\ \textbf{to}\ e_2\ \textbf{loop}\ body : notype}$$

If-then statement

$$\frac{\Gamma \vdash e : boolean \quad \Gamma \vdash body : notype}{\Gamma \vdash \textbf{if}\ e\ \textbf{then}\ body : notype}$$

3.2.3 Funzioni

Tipo di ritorno di una funzione

Indichiamo con τ_r il tipo del valore di ritorno della funzione, mentre utilizziamo returntype() per effettuare il check sul tipo dei valori di ritorno (vedere tabella 3.3).

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n \rightarrow \tau_r \quad \Gamma \vdash e_1 : \tau_i^{i \in 1,..,n} \quad returntype(\tau_r, e_i)^{i \in 1,..,n} = \tau}{\Gamma \vdash return\ \ e_i : \tau}$$

$$\frac{\Gamma \vdash f : \rightarrow \tau_r \quad \Gamma \vdash e_1 : \tau_i^{i \in 1,..,n} \quad returntype(\tau_r, e_i)^{i \in 1,..,n} = \tau}{\Gamma \vdash return\ \ e_i : \tau}$$

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n \rightarrow notype}{\Gamma \vdash return; : notype}$$

Chiamata di funzione con o senza tipo di ritorno e senza il controllo del parametro out

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i^{i \in 1,..,n}}{\Gamma \vdash f(e_1,...,e_n) : \tau}$$

$$\frac{\Gamma \vdash f : \rightarrow \tau}{\Gamma \vdash f() : \tau}$$

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n \rightarrow notype \quad \Gamma \vdash e_i : \tau_i^{i \in 1,..,n}}{\Gamma \vdash f(e_1,...,e_n) : notype}$$

$$\frac{\Gamma \vdash f : \rightarrow notype}{\Gamma \vdash f() : notype}$$

Chiamata di funzione con o senza tipo di ritorno e con il controllo del parametro out

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n | out\ \tau_{n+1} \times ... \times \tau_m \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i^{i \in 1,..,n} \quad \Gamma \vdash id_i : \tau_i^{i \in n+1,..,m}}{\Gamma \vdash f(e_1,...,e_n,id_{n+1},...,id_m) : \tau}$$

$$\frac{\Gamma \vdash f : out\ \tau_{n+1} \times ... \times \tau_m \rightarrow \tau \quad \Gamma \vdash id_i : \tau_i^{i \in n+1,..,m}}{\Gamma \vdash f(id_{n+1},...,id_m) : \tau}$$

$$\frac{\Gamma \vdash f : \tau_1 \times ... \times \tau_n | out\ \tau_{n+1} \times ... \times \tau_m \rightarrow notype \quad \Gamma \vdash e_i : \tau_i^{i \in 1,..,n} \quad \Gamma \vdash id_i : \tau_i^{i \in n+1,..,m}}{\Gamma \vdash f(e_1,...,e_n,id_{n+1},...,id_m) : notype}$$

$$\frac{\Gamma \vdash f : out\ \tau_{n+1} \times ... \times \tau_m \rightarrow notype \quad \Gamma \vdash id_i : \tau_i^{i \in n+1,..,m}}{\Gamma \vdash f(id_{n+1},...,id_m) : notype}$$

3.2.4 Operazioni matematiche

Espressioni

$$\frac{(e{:}\tau) \in \Gamma}{\Gamma \vdash e : \tau}$$

Operatori unari (vedi tabella 3.1):

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad optype1(op_1,\tau_1) = \tau}{\Gamma \vdash op_1\ e_1 : \tau}$$

Operatori binari (vedi tabella 3.2):

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad optype2(op_2,\tau_1,\tau_2) = \tau}{\Gamma \vdash e_1\ op_2\ e_2 : \tau}$$

op1	**operando**	**risultato**
MINUS	integer	integer
MINUS	float	float
NOT	boolean	boolean

Tabella 3.1: Tabella per optype1

Definiamo l'insieme AllType = {integer, float, boolean, char, string }

op1	**operando 1**	**operando 2**	**risultato**
PLUS, MINUS, TIMES, DIV, POW	integer	integer	integer
PLUS, MINUS, TIMES, DIV, POW	integer	float	float
PLUS, MINUS, TIMES, DIV, POW	float	integer	float
PLUS, MINUS, TIMES, DIV, POW	float	float	float
PLUS, MINUS, TIMES, DIV	char	char	char
PLUS, MINUS, TIMES, DIV	char	integer	char
PLUS, MINUS, TIMES, DIV	integer	char	char
STR_CONCAT	AllType	AllType	string
AND, OR	boolean	boolean	boolean
GT, GE, LT, LE, NE, EQ	integer	integer	boolean
GT, GE, LT, LE, NE, EQ	float	integer	boolean
GT, GE, LT, LE, NE, EQ	integer	float	boolean
GT, GE, LT, LE, NE, EQ	float	float	boolean
GT, GE, LT, LE, NE, EQ	char	char	boolean
GT, GE, LT, LE, NE, EQ	string	string	boolean

Tabella 3.2: Tabella per optype2

Funtype	ReturnType	risultato
integer	integer	integer
integer	float	float
float	integer	float
float	float	float
string	string	string
boolean	boolean	boolean
char	char	char

Tabella 3.3: Tabella per returntype

op	operando	operando	risultato
ASSIGN	integer	integer	integer
ASSIGN	integer	float	float
ASSIGN	float	integer	float
ASSIGN	float	float	float
ASSIGN	string	string	string
ASSIGN	boolean	boolean	boolean
ASSIGN	char	char	char

Tabella 3.4: Tabella tipi per assegnazione

3.3 Gestione errori semantici

Gli errori durante la fase di analisi semantica sono stati gestiti tramite l'uso delle eccezioni indicando il tipo di errore e la locazione tramite riga e colonna. Di seguito verranno mostrate le tipologie di errori.

3.3.1 Errori di scoping

Dichiarazione multipla del main

Nel caso in cui una funzione non etichettata con start: assume il nome main.

```
1) def main(): void{}
2)
3) start: def casa(integer x):integer{
4) }
```

exception.MultipleMainDeclaration: Errore (riga:1, colonna: 5)
-> Non è possibile chiamare una funzione 'main'!

Dichiarazione multipla di una funzione

Non è possibile dichiarare due o più volte una funzione con lo stesso nome:

```
1) def casa(): void{}
2)
3) start: def casa(integer x):integer{
4) }
```

exception.MultipleFunctionDeclaration: Errore (riga:3, colonna: 12)
-> Funzione casa già dichiarata precedentemente! :P

Dichiarazione multipla di una variabile

Non è possibile dichiarare due o più volte una variabile con lo stesso nome:

```
1) start: def main():integer{
2)     integer x;
3)     integer x;
4) }
```

exception.MultipleVariableDeclaration: Errore (riga: 3, colonna: 13)
->Variabile x già dichiarata precedentemente! :P

3.3.2 Errori di semantica

Assegnazione tra tipi errati

Si verifica quando si cerca di assegnare ad una varibile un tipo diverso da quello dichiarato:

```
1) start: def main():integer{
2)     integer x << "casa";
3) }
```

exception.TypeMismatch: Errore (riga: 2 , colonna: 23)
-> Assegnazione tra tipi incompatibili (integer e string)

Argomenti mancanti nell'assegnazione

Si verifica quando, durante l'assegnamento, non corrisponde il numero di variabili e il numero di valori da assegnare:

```
1) start: def main():integer{
2)      integer x, y;
3)
4)      x, y << 4;
5) }
```

exception.MissingAssignArguments: Errore (riga : 4, colonna :13)
->Assegnazione tra 2 identificatore/i e 1 espressione/i

Tipo non corretto nella condizione dei costrutti if e while

```
1) start: def main():integer{
2)      integer x;
3)
4)      if(x) then{}
5) }
```

exception.TypeMismatch: Errore (riga: 4, colonna: 8)
-> Espressione inserita di tipo integer, l'if si aspetta un tipo boolean!!

```
1) start: def main():integer{
2)      integer x;
3)
4)      while(x) loop{}
5) }
```

exception.TypeMismatch: Errore (riga: 4, colonna: 11)
-> Espressione inserita di tipo integer, il while si aspetta un tipo boolean!!

Funzione non dichiarata

Si verifica quando si tenta di utilizzare una funzione non dichiarata precedentemente.

```
1) start: def main():integer{
2)      integer x;
3)
4)      somma()
5) }
```

exception.FunctionNotDeclared: Errore (riga: 4, colonna: 9)
-> Funzione somma non dichiarata

Parametri della chiamata a funzione mancanti

Si verifica quando si utilizza una funzione senza inserire tutti i parametri.

```
1) def somma(integer x) : void {}
2)
3) start: def main():integer{
4)      integer x;
5)
6)      somma();
7) }
```

exception.MissingParametersFunction: Errore (riga :6, colonna :9)
->Parametri della funzione mancanti

Mancanza del valore di ritorno

Si verifica quando si definisce una funzione non void il cui corpo principale non presenta un return.

```
1) def somma(integer x) : integer {
2)
3) }
4)  start: def main():integer{
5)
6)
7) }
```

exception.MissinReturnInBodyFunction:
-> Manca il return nella funzione somma

Tipo dei parametri della chiamata a funzione errato

Si verifica quando si chiama una funzione con il tipo dei parametri errato.

```
1) def somma(integer x) : void {}
2)
3) start: def main():integer{
4)     integer x;
5)
6)     somma("casa");
7) }
```

exception.TypeMismatch: Errore (riga :6 colonna :9)
->Tipo dei parametri della funzione somma non corrisponde

Passaggio di una costante per riferimento

Si verifica quando si passa per riferimento una costante.

```
1) def somma(out integer x) : void {}
2)
3) start: def main():integer{
4)     integer x;
5)
6)     somma(3);
7) }
```

exception.TypeMismatch: Errore (riga : 6, colonna :11)
-> Non si può assegnare una costante ad un variabile di tipo out

Variabile non dichiarata

Si verifica quando si utilizza una variabile non dichiarata precedentemente.

```
1) def somma(out integer x) : void {}
2)
3) start: def main():integer{
4)     integer x;
5)
6)     somma(y);
7) }
```

exception.VariableNotDeclared: Errore (riga: 6, colonna: 11)
->Variabile y non dichiarata precedentemente!

Capitolo 4

Traduzione verso il linguaggio C

In questo capitolo verranno mostrare le principali differenze fra NewLang e C, e di come queste abbiano trovato un'implementazione.

4.1 Differenze principali

Tipo booleano

In C non esiste il tipo booleano mentre in NewLang quest'ultimo esiste ed assume come valori 'true' e 'false'. Quest'ultimi sono stati tradotti come 1 e 0.

Tipo string

In C non esiste il tipo string mentre in NewLang quest'ultimo esiste. Per implementare il tipo string si è scelto di utilizzare un puntatore di caratteri : char * nome_variabile.
Attenzione: questa implementazione delle stringhe ha un grande problema: non possono essere deallocate. Si può pensare ad un'implementazione di un garbage collector, ma per questa prima versione del progetto non è stata considerata la gestione ottimizzata della memoria.

Tipo var

In NewLang esiste il tipo var, un tipo che ha la particolarità di assumere il valore della costante assegnata. Nel linguaggio C non esiste questo tipo. Nella sua traduzione sarà scritto il tipo della costante:

In NewLang abbiamo:

```
var x << 5;
```

In C:

```
int x = 5;
```

Variabili globali

Sia in C che in NewLang esistono le variabili globali, la differenza è che in C non è possibile assegnare ad una variabile globale un valore che non sia costante:

```
int y = 5;
int x = y; //errore
```

Questo invece è possibile in NewLang, infatti nella traduzione in C abbiamo che le variabili globali sono prima solo dichiarate, mentre la loro inizializzazione è demandata ad una funzione 'int initialize_global()', che sarà invocata come prima istruzione all'interno del main.

In NewLang abbiamo:

```
int x << 5;
```

In C:

```
int x;
int initialize_global(){x = 5;}
int main(){
    initialize_global();
}
```

Dichiarazioni funzioni sia prima che dopo il 'main'

È possibile dichiarare una funzione sia prima che dopo il main. Per implementare questa funzionalità in C, vengono scritti i prototipi delle funzioni:

```
void sottrai(float a,float b,float *result);

void main(){
    float result;
    sottrai(4,3,&result);
}

void sottrai(float a,float b,float *result){
    *result = a - b;
}
```

Return obbligatorio

In NewLang è obbligatorio inserire in una funzione non void un return nel corpo principale:

```
start: def main () : integer {
     if(true) {
         ("Vero") -->!;
     }
     return 1;
}
```

Dichiarazioni di variabili prima del loro utilizzo

In NewLang è possibile dichiarare nello scope corrente una variabile prima del suo utilizzo. Nella traduzione in C questo si è tradotto in dichiarare inizialmente ogni singola variabile, successivamente vengono inizializzate per prima le costanti; risolvendo così il problema delle assegnazioni cicliche facendo si che sia C, come *side effect*, ad assegnare dei valori di default al loro uso.

In NewLang abbiamo:

```
start: def main() : void {
    integer y << x;
    integer x << 5;
    integer z << a;
    integer a << z;
}
```

In C:

```
int main(){
    int y;
    int x;
    int z;
    int a;
    x << 5;
    y << x;
    z << a;
    a << x;
}
```

Si può non chiamare main la funzione etichettata con start:

In NewLang è possibile non chiamare *main* la funzione etichettata con start:. In questo specifico caso viene creato un main *finto* che invoca la funzione.

In NewLang abbiamo:

```
start: def x() : void {}
```

In C:

```
void x(){}
int main(){
    x();
}
```

Capitolo 5

NewLang2C

NewLang2C è lo script bash che rappresenta l'eseguibile del nostro compilatore. Per utilizzare NewLang2C bisogna installare i seguenti applicativi:

- JAVA versione 19.0.1;
- Scaricare il JAR file NewLang presente al seguente link: NewLang.jar
- Compilatore gcc;

Una volta installato tutto l'occorrente si può invocare lo script come segue:

./NewLang2C nome_file [-com_opz1 -com_opz2 -com_opz3 -com_opz4]

I parametri opzionali (com_opz) utilizzabili sono i seguenti:

- -help: mostra una guida su come utilizzare il compilatore;
- -sp: stampa lo scoping del codice analizzato (lettura dal basso verso l'alto);
- -sc: salva il file C prodotto;
- -xml: salva il file xml prodotto.
- per ogni altro parametro opzionale restituisce un messaggio: comando non trovato.

Lo script bash per eseguire il compilatore è presente al seguente link: NewLang2C.