



POLITECNICO DI BARI

Corso di Laurea Magistrale in Ingegneria Informatica

A.A. 2010/2011

“ A2C ”

An ADA-like parser/scanner and C translator

Tema d'anno per l'esame di

Linguaggi Formali e Compilatori

Docente:

Chiar.mo Prof. Giacomo Piscitelli

Studente:

Ruggiero Campese

Sommario

Introduzione	3
1 - Compilazione del sorgente ed uso del Compilatore	4
2 - L'analizzatore sintattico	7
2.1 - Grammatica del Linguaggio ADA-like	8
3 - L'analizzatore lessicale	14
4 - L'analizzatore semantico	16
4.1 – Le tabelle dei simboli	16
4.2 – Strutture di supporto	19
5 – La gestione degli errori	23
6 – Il generatore di codice C	24
7 – Caso applicativo di Test	26
Allegati	28
Allegato A: parser.y	28
Allegato B: scanner.l	29
Allegato C: varST.h	30
Allegato D: proST.h	31
Allegato E: typST.h	32

Introduzione

Oggetto di questa relazione è la realizzazione del *Front-End* di un Compilatore dedicato all'analisi ed alla traduzione di un linguaggio ADA-like, ovvero un linguaggio di programmazione formato con i costrutti tipici in notazione ADA ma implementato su una restrizione della sua grammatica.

ADA, sebbene originariamente adoperato per scopi militari, è un linguaggio di programmazione '*general-purpose*' che include gran parte dei principi dell'ingegneria del software, come la programmazione modulare, la programmazione orientata agli oggetti, la programmazione concorrente e la programmazione distribuita.

Ha tra le sue peculiarità quello di possedere paradigmi di programmazione estremamente efficienti e sicuri, che rendono ADA la scelta prediletta nella progettazione di sistemi software '*mission and safety critical*'; a testimonianza di tale robustezza ADA è l'unico caso di linguaggio di programmazione in cui i compilatori devono essere sottoposti ad un processo di certificazione secondo lo standard internazionale *ISO/IEC 18009 -Ada: Conformity Assessment of a Language Processor*. Coadiuvare i vantaggi offerti dal linguaggio ADA ed allo stesso tempo la qualità del C di essere il linguaggio '*machine-independent*' per antonomasia grazie alla diffusione dei suoi compilatori su qualsiasi architettura e Sistema Operativo, è la motivazione di base che ha portato alla scelta di questo tema d'anno.

In questo progetto si è costruito il primo stadio di compilazione di un Compilatore a due passi, dipendente esclusivamente dal linguaggio sorgente ADA-like e avulso dalla macchina target su cui il programma dovrà girare, dal momento che il compilatore traduce il sorgente ADA-like in linguaggio Ansi-C.

Il progetto del Compilatore si è articolato nello sviluppo di vari moduli:

- un analizzatore lessicale (Scanner);
- un analizzatore sintattico (Parser);
- un analizzatore semantico (Semantic Checker);
- delle Tabelle dei Simboli (Symbol Tables);
- un Gestore degli errori (Error Handler);
- un Traduttore di alto livello che genera codice C.

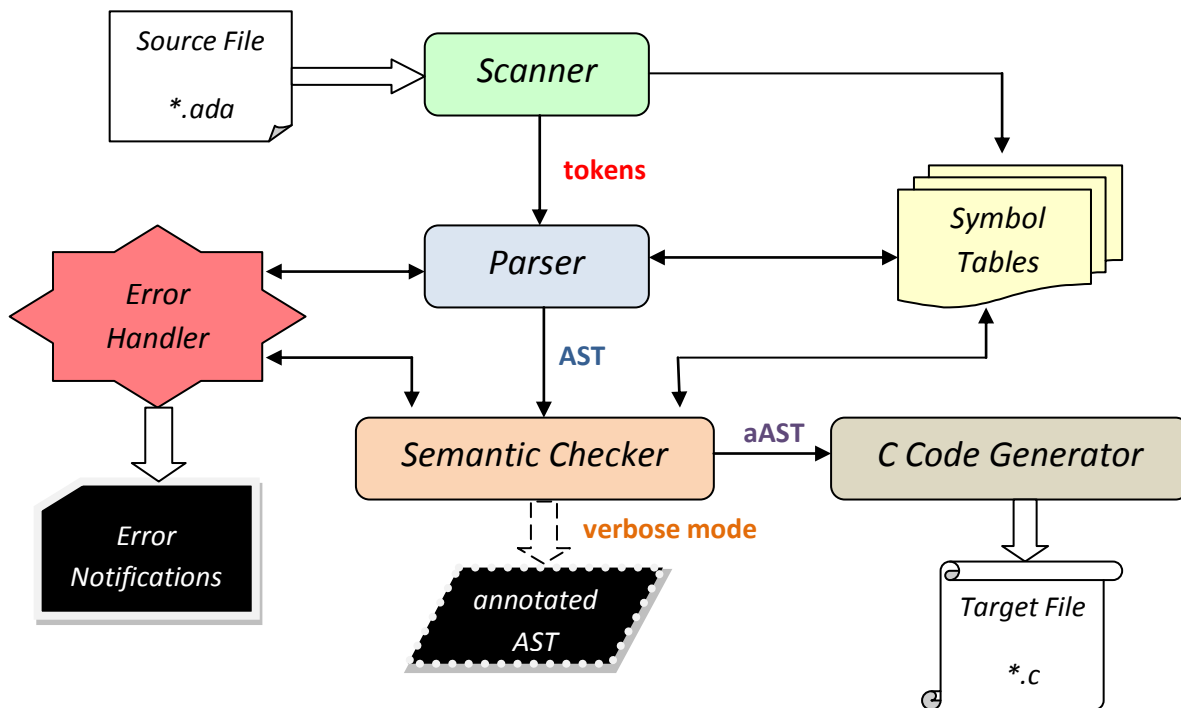


Fig.1: schema strutturale del Compilatore

Lo progettazione e l'implementazione del compilatore sono avvenuti interamente in ambiente *Linux - Ubuntu 10.10 Maverick Meerkat*, distribuzione in cui sono inclusi i tools di sviluppo *Flex* e *Bison* di cui ci si è avvalsi per la realizzazione rispettivamente dello scanner e del parser.

1 - Compilazione del sorgente e uso del compilatore

La compilazione del sorgente e l'uso del compilatore sono stati testati con Sistema Operativo Linux, per cui è garantita la funzionalità del software in ambienti Unix-like forniti dei tools *Bison* e *Flex* e del compilatore *GCC* (GNU Compiler Collection).

Nella cartella i file principali sono:

- *scanner.l*, file che contiene le specifiche per lo scanner generator *Flex*;
- *parser.y*, file che contiene le specifiche per il parser generator;
- *script*, un piccolo script per generare automaticamente l'eseguibile;

- *Adaprova.ada*, un file contenente codice ADA con le istruzioni dei principali paradigmi di programmazione usati per testare la funzionalità del Compilatore.

Ci sono due modi per compilare ed ottenere l'eseguibile finale. Il primo è procedere per passi generando e compilando i singoli file uno per volta ed eseguendo quindi il link dei file-oggetto creati; l'altro consiste nell'eseguire lo script esistente che contiene una sequenza preimpostata di tutti i comandi precedenti necessari alla creazione dell'eseguibile. Chiaramente se si vuole specificare particolari flag, opzioni di compilazione o scegliere il file eseguibile finale, è consigliabile la prima modalità.

```
bison -d parser.y
gcc -c parser.tab.c
flex scanner.l
gcc -c lex.yy.c
gcc -o A2C parser.tab.o lex.yy.o -lm
```

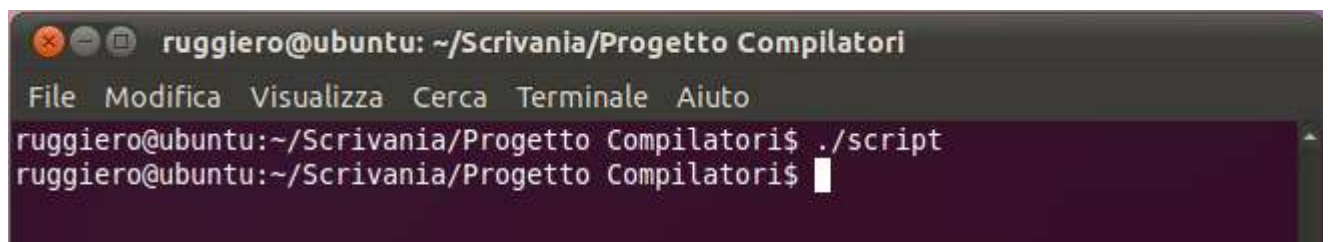


Fig 2: script ed esempio d'uso

In sintesi occorre innanzitutto generare il parser tramite il comando

```
bison -d parser.y
```

dove *parser.y* è il file che contiene le direttive specificate dalla grammatica del linguaggio; viene usata l'opzione *-d* per istruire bison della generazione di un file header contenente i codici dei token associati alla grammatica, che sarà incluso e utilizzato dallo scanner. Bison genera quindi il file *parser.tab.c* contenente il codice C che è in grado di eseguire l'analisi sintattica. Si compila il file,

```
gcc -c parser.tab.c
```

e si istruisce il compilatore con l'opzione *-c* a creare il relativo file oggetto.

Possiamo ora occuparci della generazione dello scanner con il comando

```
flex scanner.l
```

e relativa compilazione del file *lex.yy.c* generato da Flex e creazione del file oggetto,

```
gcc -c lex.yy.c
```

A questo punto sono stati ottenuti i file oggetto che devono essere sottoposti ad operazione di *linking* per la generazione del file eseguibile finale

```
gcc -o A2C parser.tab.o lex.yy.o -lm
```

che nello script è chiamato *A2C*; l'opzione *-lm* indica al compilatore di cercare le funzioni matematiche specificate nel codice.

Ottenuto il file eseguibile è pertanto possibile usare il compilatore da linea di comando inserendo il nome del file eseguibile e il percorso del sorgente ADA-like da tradurre. Il compilatore mostrerà a video gli eventuali errori di compilazione e genererà il relativo file di traduzione in codice C, salvandolo nella cartella di lavoro.

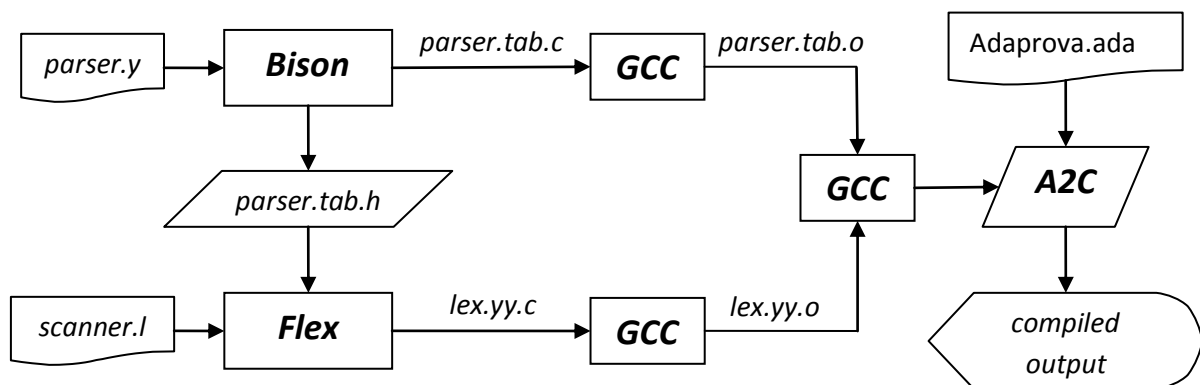
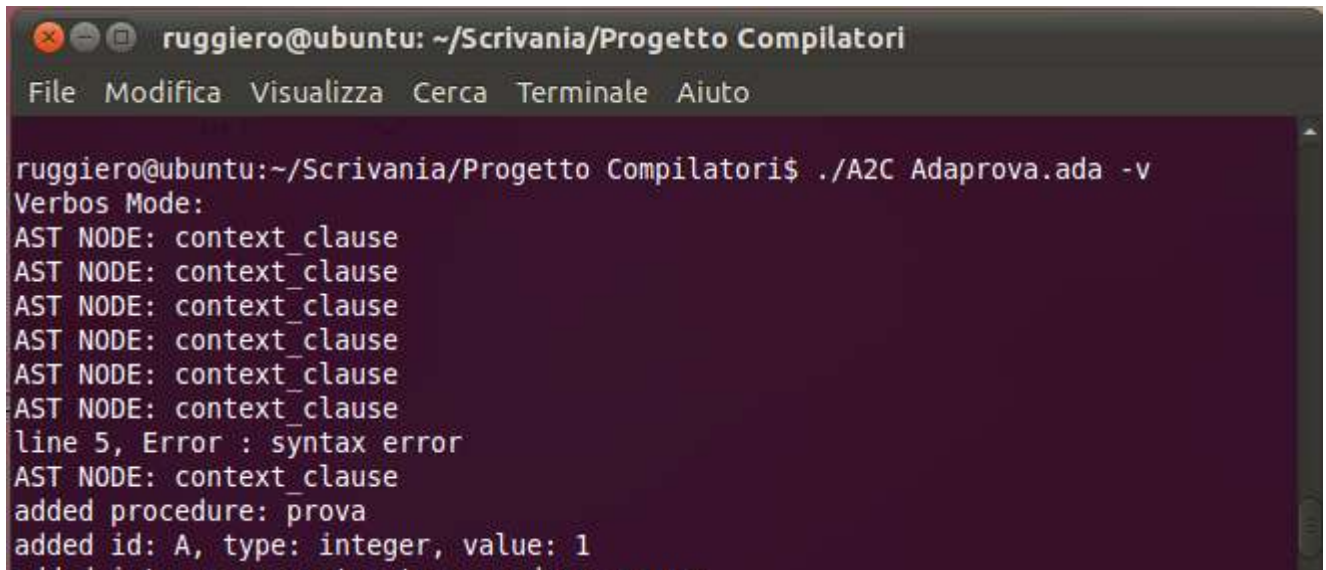


Fig.3: diagramma a blocchi della generazione del Compilatore

E' possibile anche utilizzare il Compilatore in modalità '*verbose*', specificando l'opzione *-v*. In questa modalità oltre al classico output il Compilatore mostrerà a video alcuni dei simboli non terminali più significativi che ha riconosciuto nella

costruzione dell'albero sintattico insieme all'arricchimento di attributi semantici che si hanno con le operazioni sulle *Symbol Tables*.



```
ruggiero@ubuntu: ~/Scrivania/Progetto Compilatori
File Modifica Visualizza Cerca Terminale Aiuto

ruggiero@ubuntu:~/Scrivania/Progetto Compilatori$ ./A2C Adaprova.ada -v
Verbos Mode:
AST NODE: context_clause
AST NODE: context_clause
AST NODE: context_clause
AST NODE: context_clause
AST NODE: context_clause
AST NODE: context_clause
line 5, Error : syntax error
AST NODE: context_clause
added procedure: prova
added id: A, type: integer, value: 1
```

Fig. 4: esempio uso A2C

2 - L'analizzatore sintattico

L'analizzatore sintattico, il Parser, rappresenta il modulo centrale, il *core* di un Compilatore. L'obiettivo di un parser è quello di stabilire la correttezza sintattica della sequenza di token fornitigli in input ovvero se il sorgente da tradurre rispetti la grammatica con cui è stato definito il linguaggio di programmazione.

In effetti, *"il progetto della grammatica costituisce la fase iniziale del progetto del compilatore"*.

Un Parser considera gli elementi lessicali come atomi indivisibili, definiti *token*, che vengono generati dall'analizzatore lessicale ed inviati al parser stesso. I token rappresentano i costituenti di livello più basso che formano le parti essenziali del programma, i costituenti di alto livello dichiarativi ed esecutivi.

La strutturazione di questi costituenti è delegata al parser che li identifica, ne controlla la correttezza e costruisce il cosiddetto *Syntax Tree*, il modello che struttura il sorgente secondo le regole di produzione specificate.

Esistono diverse tipologie di rappresentazione di una sintassi ma quelle che vengono universalmente adottate come input degli attuali parser sono le sintassi libere, anche dette sintassi non-contestuali; una grammatica context-free consente di implementare algoritmi di analisi sintattica che presentano una complessità lineare.

2.1 - Grammatica del linguaggio ADA-like

La descrizione della grammatica *Context-Free* associata al parser è solitamente rappresentata con regole di produzione nella *Backus Naur Form*. Viene riportata di seguito la grammatica BNF del linguaggio ADA-like a cui si è fatto riferimento per descrivere le regole di produzione nel file *parser.y*.

```
compilation_unit ::= context_clause subprogram_body
context_clause ::= {context_item}
context_item ::= with_clause | use_clause
with_clause ::= "with" library_unit_name { "," library_unit_name } ";"
use_clause ::= "use" library_unit_name { "," library_unit_name } ";"
library_item ::= package_declaration
subprogram_body ::= subprogram_specification "is"
                    declarative_part
                    "begin"
                    sequence_of_statements
                    "end" [designator] ";"
subprogram_specification ::= "procedure" procedure_name parameter_profile
                           | "function" procedure_name parameter_profile "return" type
parameter_profile ::= [ "(" parameter_specification
                      { ";" parameter_specification } ")" ]
parameter_specification ::= parameter_name_list ":" mode type
                        [ ":@" default_expression ]
mode ::= [ "in" ] | "out" | "in" "out"
```



```

parameter_name_list ::= identifier { "," identifier }

procedure_name ::= identifier

declarative_part ::= { declarative_item }

declarative_item ::= record_declaration | array_declaration
                    | object_declaration | subprogram_body

record_declaration ::= "type" identifier "is" "record"
                      component_item
                      "end" "record" ";"

component_item ::= { object_declaration }

array_declaration ::= identifier "is" "array" range "of" type ";"

range ::= numeral ".." numeral

object_declaration ::= identifier_list : type [":=" expression] ";"

sequence_of_statements ::= { statement }

statement ::= assignment_statement ";" | if_statement ";" |
              case_statement ";" | loop_statement ";" | return_statement ";" |
              procedure_call_statement ";" | io_statement ";"

assignment_statement ::= identifier ":=" assign_expression

if_statement ::=
    "if" condition "then"
        sequence_of_statements
    {"elsif" condition "then"
        sequence_of_statements}
    ["else"
        sequence_of_statements]
    "end if;"

case_statement ::=
    "case" expression "is"
        case_statement_alternative
        { case_statement_alternative }
    "end case;"

case_statement_alternative ::=
    "when" discrete_choice_list "=>" sequence_of_statements

discrete_choice_list ::= discrete_choice { | discrete_choice }

discrete_choice ::= expression | discrete_range | "others"

loop_statement ::=
    [loop_statement_identifier:]
    [iteration_scheme] loop
        sequence_of_statements
    end loop [loop_identifier];

```

```

iteration_scheme ::= while condition
                  | for loop_parameter_specification

loop_parameter_specification ::=
    defining_identifier in [reverse] discrete_subtype_definition

return_statement ::= return [relation];

procedure_call_statement ::= procedure_prefix actual_parameter_part;

actual_parameter_part ::= term {,term}

relation ::= simple_expression "=" simple_expression
           | simple_expression "/" simple_expression
           | simple_expression ">" simple_expression
           | simple_expression ">=" simple_expression
           | simple_expression "<" simple_expression
           | simple_expression "<=" simple_expression

simple_expression ::= term + term | term "-" term | term "*" term
                  | term "/" term | term

term ::= numeric_literal | decimal_literal | identifier | "'" letter "'"
      | procedure_call_statement | "(" simple_expression ")"

identifier ::= letter { [ "_" ] letter_or_digit }
letter_or_digit ::= letter | digit

numeric_literal ::= decimal_literal | based_literal
decimal_literal ::= numeral [ "." numeral ] [ exponent ]
numeral ::= digit { digit | "_" }

```

La grammatica presa in considerazione pur essendo una restrizione di quella originale del linguaggio ADA, preserva il teorema sulla descrizione degli algoritmi di *Bohm-Jacopini* e risulta essere allo stesso tempo molto espressiva grazie ai vari costrutti che si è deciso di includere.

In sintesi, la grammatica presenta come assioma

```

compilation_unit ::= context_clause subprogram_body

```

ogni programma deve essere ricondotto ad una unità di compilazione formata a sua volta da una clausola iniziale opzionale, dove si possono includere (*use*) e impostare (*with*) i vari package utili alla stesura del programma ADA, e il corpo del programma.

```

subprogram_body ::= subprogram_specification "is"
                    declarative_part
                    "begin"
                    sequence_of_statements
                    "end" [designator] ";"

subprogram_specification ::= "procedure" procedure_name
parameter_profile
    | "function" procedure_name parameter_profile "return" type

```

Il programma ha una struttura base composta da un'intestazione, una parte dichiarativa e una parte esecutiva che inizia con la parola chiave "*begin*". La sintassi consente di specificare come programma una procedura semplice o una funzione con valore di ritorno; entrambe possono avere dei parametri formali. Nella parte dichiarativa vanno inserite tutte le variabili e strutture che si intende adoperare nell'esecuzione del programma, si ricorda infatti che ADA è un linguaggio a tipizzazione forte, aspetto che sarà ovviamente anche oggetto di controllo semantico.

```

declarative_item ::= record_declaration | array_declaration
                  | object_declaration | subprogram_body

```

Nella parte dichiarativa possiamo inserire la dichiarazione di una nuova struttura record aggiungendo un nuovo tipo, la dichiarazione di un array statico, la dichiarazione di identificatori e anche di funzioni interne al *subprogram*.

La parte esecutiva del *subprogram* è formata da una sequenza di istruzioni, di *statement*; gli statement consentiti dalla sintassi sono:

```

statement ::= assignment_statement ";" | if_statement ";"
           | case_statement ";" | loop_statement ";" |
           return_statement ";" | procedure_call_statement ";" |
           io_statement ";"

```

Sono contemplati dalla grammatica l'istruzione di assegnazione di un valore ad un identificatore, la struttura condizionale dell'*if*, la struttura di selezione del *case*, la

struttura di iterazione del *loop*, l'istruzione di ritorno valore da inserire all'interno di una funzione, l'istruzione di chiamata di funzione, l'istruzione di input/output.

Per quanto riguarda le espressioni è stata effettuata la scelta progettuale di associare a due diversi non-terminali, *relation* e *simple_expression*, le produzioni che hanno come valore finale rispettivamente un dato di tipo *boolean* o il risultato di operazioni aritmetiche. Tutti i tipi di espressioni vengono ovviamente costruiti a partire dai terminali associati al non-terminale *term*, che può rappresentare un numero intero, un numero float, un carattere, un identificatore o un valore di ritorno determinato da una chiamata di procedura.

```
term ::= numeric_literal | decimal_literal | identifier | "'"
letter "'" | procedure_call_statement | "(" simple_expression ")"
```

La grammatica BNF è stata riscritta in formato GNU-Bison esplicitando le regole di produzione e le corrispondenti azioni.

Il Parser è stato creato tramite il *parser generator GNU-Bison v2.5*.

Bison legge la specifica di una grammatica context-free, notifica dei *warning* in corrispondenza di eventuali ambiguità o conflitti insiti nella sintassi e genera un parser che può essere scritto in C, C++ o Java, che acquisisce lo stream di token in input e qualora questi siano conformi alle regole sintattiche specificate, costruisce l'Abstract Syntax Tree. In A2C il Parser è scritto in linguaggio C.

Il Parser generato da GNU-Bison funziona come un'*automa a stati finiti di tipo push-down(PDA)* con metodo di *analisi bottom-up* ottimizzato per grammatiche *LARL(1)*.

In breve il meccanismo di Parsing creato è modellizzato secondo un'*automa a pila*; l'*automa a pila* è un riconoscitore deterministico a stati finiti dotato di una testina d'ingresso che legge i simboli del nastro d'ingresso, di un'unità di controllo che computa le transizioni di stato e di una testina di memoria che esegue operazione di lettura e scrittura (*Push, Pop*) su una memoria ausiliaria illimitata organizzata secondo struttura dati a *stack*.

La tecnica di riconoscimento è di tipo bottom-up, ovvero il Parse-Tree viene costruito a partire dalle foglie per poi risalire sino all'assioma.

Il Parser

- processa l'input viene da sinistra verso destra un simbolo alla volta,
- lavora in modalità '*shift-reduce parsing*' ossia adopera una riduzione 'right-most-derivation' dove ad ogni passo si cerca di sostituire il non-terminale più a destra,
- usa un solo simbolo di *look-ahead* per individuare gli *handle* (maniglie) con cui verrà deciso il parsing.

Il file di input per il tool Bison, quello dove è specificata la grammatica, è *parser.y* ; il file è inserito in coda alla presente relazione come allegato. Il file è suddiviso formalmente in diverse sezioni identificate da opportuni separatori:

```
%{  
  Prologue  
%}  
  
  Bison declarations  
  
%%  
  Grammar rules  
%%  
  Epilogue
```

Fig.5: struttura del file parser.y

In particolare nella sezione '*Prologue*' troviamo l'inclusione di tutte le librerie e dei vari *headers* che si sono rivelati necessari all'implementazione.

Vengono quindi dichiarati molti dei flag e delle variabili di lavoro che vengono diffusamente usati nel seguito del programma.

Sono inoltre dichiarate anche le procedure che occorrono per aggiornare le varie *Symbol Tables* e le varie funzioni necessarie ad un'adeguata analisi semantica.

Nella sezione '*Bison declarations*' è ridefinita la variabile *union yylval* che serve per associare un tipo di dato al token che viene trasmesso dallo scanner al parser; nel presente progetto sono stati dichiarati un tipo int, float, char* e char. Il tipo char* è

definito per associare ai token la chiave con cui sono identificati nella *Symbol Table*. La sezione continua quindi con la definizione di tutti i simboli terminali (in maiuscolo) e di quelli non-terminali (in minuscolo); per quegli elementi sintattici in cui era rilevante associare un valore al relativo token, la dichiarazione è preceduta dalle parentesi angolari con il tipo della variabile *yyval* scelto. Specifici tag *left* sono usati alternativamente al tag *token* per specificare l'associativa delle operazioni.

Proseguendo con la sezione '*Grammar rules and actions*' arriviamo al corpo del Bison-file dove sono specificate le regole di produzione e le eventuali azioni in C racchiuse tra parentesi graffe. Le regole di produzione sono state opportunamente e soprattutto, accuratamente dedotte dalla grammatica in forma BNF di riferimento per il linguaggio ADA-like. Rilevante notare come si è imposta una costante ricorsione sinistra per tutte le regole eccetto per il caso del non-terminale *identifier_list*, in cui si è resa necessaria una ricorsione destra per impilare correttamente la lista degli identificatori che saranno utili nelle successive azioni semantiche.

Ultima è la sezione '*Epilogue*' in cui è stata dichiarata la funzione main principale del compilatore in cui si controlla l'eventuale modalità *verbose* e ovviamente si lancia la funzione standard di parsing *yyparse()*. Vengono infine inserite le implementazioni di tutte le funzioni semantiche, le procedure di utilità delle *Symbol Tables* e le routine di gestione degli errori precedentemente indicate nella sezione di Prologo.

3 - L'analizzatore lessicale

L'analizzatore lessicale, anche detto Scanner o Tokenizer, ha il compito di scandire uno stream di caratteri di input e verificare una eventuale decomposizione in lessemi (*token*) opportunamente specificati nel lessico impostato. Se il riconoscimento fallisce, lo Scanner trasmette in uscita invariato lo stream di input appena esaminato. Nel progetto del Compilatore lo Scanner rappresenta una sorta di modulo di pre-processing per il Parser; esso infatti seleziona, raggruppa e trasmette gli elementi lessicali che sono necessari all'analisi sintattica quando gli vengono richiesti dal Parser stesso.

Gli elementi lessicali dello scanner possono essere delle parole chiave, esplicitate con un insieme di caratteri fisso, o delle classi lessicali, stringhe identificate da opportune espressioni regolari. Eventualmente lo Scanner può trasmettere in associazione al generico elemento lessicale individuato, anche il valore semantico che lo caratterizza. Ciò avviene usando la variabile *union yylval* che è condivisa dallo *scanner* e dal *parser*.

Esistono diverse modalità implementative per la realizzazione di uno Scanner; in A2C lo Scanner è stato generato tramite uno *Scanner Generator*, Flex 2.5.35 (Fast Lexical Analyzer). Per generare lo Scanner Flex ha bisogno di un file di input che in questo progetto è *scanner.l*, allegato alla relazione.

Flex genera uno scanner basato sul funzionamento classico di un'automa a stati finiti che definisce l'approccio riconoscitivo di un linguaggio tramite la specifica di opportune espressioni regolari. Nel file *scanner.l* sono indicate le espressioni regolari utili al riconoscimento dei *token* del linguaggio ADA-like.

Il file è strutturato secondo convenzione

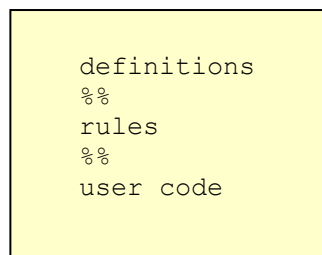


Fig.6: struttura del file scanner.l

Nella sezione '*definitions*' si includono le librerie e, quando usato in combine con Bison, si include anche *parser.tab.h*, l'*header* contenente i codice dei *token* del *Parser*.

E' dichiarata la variabile esterna *verb* che è impostata dal Parser per l'opzione *verbose* del Compilatore.

Vengono dopo specificate due opzioni; una rende l'analisi lessicale *case-insensitive*, lo Scanner non farà differenza tra caratteri minuscoli o maiuscoli rispettando la regola con cui è definito per l'appunto ADA. L'altra istruisce lo Scanner al conteggio delle righe che esamina, attività che si rivelerà utile per la gestione degli errori.

Sono dichiarate quindi le classi lessicali adoperate per poi arrivare alla sezione *'rules'*, dove sono presenti le regole di analisi lessicale vere e proprie. La sezione *'user code'* è vuota dal momento che la funzione di *scanning yylex()* è automaticamente richiamata dal Parser quando si renderà necessario acquisire token.

Da notare come la lunghezza degli elementi lessicali da riconoscere sia al minimo di due caratteri; ciò è dovuto al fatto che è possibile specificare in Bison nelle regole sintattiche particolari singoli caratteri delimitatori inseriti tra apici.

4 - L'analizzatore semantico

L'uso di una particolare sotto-classe di grammatica *context-free* consente di avere un computazione tempo-lineare in fase di compilazione ma allo stesso tempo presenta forti limitazioni nell'espressione di condizioni sul linguaggio. E' manifesta quindi la necessità di sovra visionare la composizione sintattica da opportuni controlli semantici; questi basano la loro attività sul significato che viene attribuito agli elementi sintattici. In effetti il Parser creato genera insitamente nell'AST gli attributi semantici dei nodi non-terminali e terminali, tramite l'aggiornamento della variabile *yylval* definito dallo Scanner, la presenza di *Symbol Table* e il supporto fornito da alcuni *headers* aggiunti al programma che mettono a disposizione varie utility.

4.1 - Le tabelle dei simboli

Le tabelle dei Simboli (*Symbol Table*) sono strutture dati ausiliarie al Compilatore che tengono traccia della semantica degli elementi sintattici individuati durante la fase di *Parsing*. Nel progetto le tabelle sono state implementate usando delle *Hash Tables*, strutture dati che associano ogni record presente in tabella ad una chiave univoca; sulla chiave primaria viene applicata una funzione di hashing $h(k)$ che determina idealmente la posizione univoca in cui memorizzare la nuova entry, ovvero l'indice dell'elemento in un array associativo dove il valore deve essere

memorizzato. Con questo approccio si ottiene il grande vantaggio di rendere praticamente istantaneo il tempo di accesso e ricerca di ogni record.

A2C è codificato in linguaggio C per cui dato che le *hash tables* non sono previste nello standard, è sorta l'esigenza di implementarle. Le strutture adoperate nel presente progetto hanno fatto uso del package '*uthash-1.9.3*'.

Uthash, scritta da Troy Hanson, è effettivamente composto da un header file da includere nel sorgente dove utilizzare l'Hash Table implementata in C; supporta operazioni tempo-costanti di aggiunta, ricerca e rimozione su strutture dati C.

Qualsiasi struttura o record in cui si sia definito un unico membro chiave di qualsiasi tipo esso sia, può essere gestito come Hash Table semplicemente aggiungendo il membro *UT_hash_handle* alla struttura; l'API mette quindi a disposizione le varie *macros* con cui effettuare le operazioni.

```
#include "uthash.h"

struct my_struct {
    int id;           /* we'll use this field as the key */
    char name[10];
    UT_hash_handle hh; /* makes this structure hashable */
};

struct my_struct *users = NULL;

void add_user(struct my_struct *s) {
    HASH_ADD_INT( users, id, s );
}

struct my_struct *find_user(int user_id) {
    struct my_struct *s;

    HASH_FIND_INT( users, &user_id, s );
    return s;
}

void delete_user(struct my_struct *user) {
    HASH_DEL( users, user);
}
```

Fig.7: esempio uso Udash handle

In A2C si è optato per la scelta progettuale di avere diverse *Symbol Tables* separate, ciascuna delle quali dedicate alla trattazione di un particolare tipo di dati. Le tabelle implementate sono:

1. varST.h, per memorizzare gli identificatori, il tipo, il valore, la dimensione delle variabili e dei vettori;

```
typedef struct {
    char name [30];
    char type [10];
    int l1;
    int l2;
    typeVal value;
    UT_hash_handle hh;
} var_hash_struct;
```

Fig.8: struttura varST

2. proST.h, per tener traccia delle procedure dichiarate, del numero di parametri formali associato, dei parametri formali stessi, dell'eventuale tipo del valore di ritorno, del valore di ritorno stesso;

```
typedef struct {
    char name[30] ;
    int numPar;
    stackStrNodePtr parFor;
    char rtnVal[10];
    typeVal value;
    UT_hash_handle hh;
} pro_hash_struct;
```

Fig.9: struttura proST

3. typST.h, con cui si aggiornano i tipi dichiarati da nuove strutture record e dei relativi componenti.

```
typedef struct {
    char name [30];
    char type [30];
    UT_hash_handle hh;
} typ_hash_struct;
```

Fig.10: struttura typST

Questo approccio ha consentito di ottimizzare l'allocazione di memoria destinando ciascun elemento nell'appropriata struttura invece di avere un'unica tabella che comprendesse tutti gli attributi. Per ogni Tabella sono definite tutte le operazioni primarie di inserimento, ricerca e cancellazione istanza.

4.2 - Strutture dati di supporto

Nell'ambito dell'analisi semantica si è rivelato altresì necessario utilizzare altre strutture dati per supportare le varie azioni di controllo.

Le utility aggiuntive adoperate sono state incluse in due librerie aggiunte nel file *parser.y*, sono *stackStr.h* e *stackHvar.h*:

```
#include "stackStr.h"
#include "stackHvar.h"
```

Come si evince dal nome dei due headers le strutture dati incluse sono stack. In ogni libreria sono definiti la nuova struttura e le classiche operazioni di *push* e *pop*.

In particolare la prima libreria è stata creata come strumento di supporto alla memorizzazione di liste di identificatori. Quando il Parser identifica liste di identificatori, come nella specifica di parametri di funzione o nella dichiarazione multipla di variabili, vi è la necessità di immagazzinare queste sequenze per effettuare valutazioni semantiche postume; ad esempio il controllo del tipo in caso di assegnazione valore in fase dichiarativa è una di queste verifiche.

```
procedure prova(A, B : in integer; Result : out integer) is
    R, G : float := 2.2;
```

Fig.11: esempio uso stackStr

L'altra libreria è invece uno stack in cui vengono impilati i puntatori delle *Symbol Table* degli identificatori. Con questa struttura si vuole realizzare il controllo semantico del livello di visibilità di una variabile.

Quando, infatti, viene dichiarata una variabile questa è visibile soltanto all'interno della funzione in cui è stata dichiarata. Non è possibile usare una variabile se questa è stata dichiarata in un contesto diverso.

L'azione semantica prevede quindi il cambio, con opportune operazioni di push e di pop, di *Symbol Table* ogni qualvolta si passa ad una nuova funzione o procedura, creando un adeguato spazio di indirizzamento dedicato.

Definendo quindi l'analisi semantica si parla in maniera più appropriata di azioni semantiche. L'approccio preferito in A2C è quello *syntax-directed translation*: le regole per l'analisi semantica del linguaggio possono essere codificate nelle azioni associate alle regole di produzione del Parser per descrivere come computare i valori attribuiti al nodo od al corrispondente nodo-padre.

Spesso la il confine tra errore sintattico ed errore semantico non è ben definito. Si può per esempio precludere all'utente la possibilità di inserire termini semanticamente incoerenti definendo una grammatica che consenta l'inserimento di un particolare set di simboli terminali corretti.

Nel progetto sono stati tenuti in considerazione alcuni controlli semantici generali dei linguaggi di programmazione (es. divisione per zero) in aggiunta a quelli tipici di un linguaggio ADA-like (es. tipizzazione forte).

Nella descrizione delle azioni semantiche sono state definite delle procedure di supporto:

```
void insert_var(char* id, char *type, typeValPtr temp, int assign);
```

- inserisce un identificatore nella *Symbol Table* delle variabili comunicando il nome, il tipo, il flag di assegnazione e l'eventuale valore di assegnazione. Contestualmente viene controllato che il nome adoperato non sia già in uso per un altro identificatore;

```
int context_var_check(char*);
```

- realizza il controllo sulla tipizzazione forte degli identificatori adoperati nel programma, ovvero controlla se la variabile che si sta cercando di usare è stata correttamente dichiarata ed inizializzata in precedenza;

```
void ass_val(char *id);
```

- assegna un valore in fase dichiarativa di una variabile in base al suo tipo;

```
void update_val(char *id1, char *id2);
```

- aggiorna il valore dell'identificatore *id1* con quello dell'identificatore *id2* nella *Symbol Table* in base al tipo degli operatori (segue sempre un controllo di *type_checking*);

```
void insert_pro(char* proc);
```

- in seguito alla specifica di una nuova procedura o funzione inserisce nella *Symbol Table* delle procedure una nuova voce con chiave *proc*;

```
void insert_par_pro(char *parType);
```

- aggiunge la lista dei tipi di parametri formali che sono stati dichiarati nell'ambito di una procedura e inserisce le nuove istanze tra le variabili;

```
void par_check();
```

- controlla la corrispondenza tra parametri attuali e formali in concomitanza di una chiamata di funzione;

```
int context_pro_check(char* id, int numPar);
```

- realizza il controllo sull'uso delle funzioni e delle procedure verificando che l'identificatore usato risulti una procedura correttamente dichiarata in precedenza e che il numero di parametri attuali corrisponda a quello dei parametri formali;

```
int type_checking(char* id1, char* id2);
```

- realizza il controllo sul tipo degli operatori di un'operazione in quanto ADA non permette di mettere in relazione due tipi di operatori differenti;

```
int context_return_check(char* expr);
```

- controlla che il valore di ritorno che viene specificato nel corpo di una funzione sia dello stesso tipo di quello specificato nella dichiarazione;

```
int record_com_check(char* name, char* com);
```

- controlla che nell'uso di una variabile di tipo record venga specificato un componente coerente con la dichiarazione del tipo record;

```
void insert_typ_com(char *comTyp, char *id);
```

- inserisce la dichiarazione di un nuovo componente all'interno della dichiarazione del nuovo tipo di record *id*;

```
void insert_arr(char* id, char *type, int l1, int l2);
```

- inserisce un nuovo identificatore associato ad un vettore, il relativo tipo e i limiti di dimensionalità specificati;

```
int array_bound_check(char* id, int ind);
```

- controlla che qualora si faccia riferimento all'elemento di un vettore l'indice *ind* specificato sia coerente con i limiti associati al vettore *id*.

5 - La gestione degli errori

Un buon compilatore non dovrebbe terminare la sua esecuzione al manifestarsi del primo errore; il Parser deve essere in grado di analizzare il file sorgente sino alla fine.

L'insorgenza di errori durante la compilazione può essere di vario tipo: lessicale, sintattico, semantico e logico.

In tutti i casi possibili di *error detection* il programma passa il controllo alla routine di gestione *yyerror(char cont*)* che si occupa della diagnosi del problema verificatosi trasmettendo a video il messaggio di errore.

Grazie alla definizione di una grammatica LR il compilatore gode della proprietà '*viable prefix*' che consente di rilevare un errore non appena questo si manifesta.

Per quanto riguarda gli errori sintattici la strategia di *error recovery* scelta è quella del *Panic Mode*: in corrispondenza di un errore sintattico il parser cerca di riprendere la sua analisi dai cosiddetti *token sincronizzanti*, che sono solitamente delle parole chiave che indicano al Parser fin dove scartare il codice dal punto di rilevazione dell'errore. E' un metodo molto veloce ma purtroppo non si ha una conoscenza deterministica della quantità di input che verrà scartato.

GNU - Bison supporta questa tipologia di *error recovery* tramite l'uso del token *error*. E' un simbolo terminale predefinito speciale, riservato per l'*error handling*, che viene generato automaticamente da Bison in corrispondenza di un errore sintattico; inserendolo nelle produzioni in cui si vuole effettuare il controllo sugli errori il Parser continua la sua analisi a partire dal token sincronizzante specificato. Altresì Bison mette a disposizione una macro predefinita, *yyerrok*, che gestisce in automatico la chiamata alla routine di errore.

```
declarative_item : record_declaration
                  | array_declaration
                  | object_declaration
                  | subprogram_body
                  | error ';' { yyerrok; }
```

Fig.12: esempio uso token error

Per gli errori sintattici si è comunque in presenza di una regola sintattica riconosciuta, per cui la continuità dell'analisi sintattica del Parser è preservata.

A seconda dei controlli semantici che individuano gli errori viene inviato alla routine di gestione l'adeguato messaggio da mostrare all'utente. L'analisi sintattica, come detto, proseguirà ma il fallimento di un'azione semantica causerà con tutta probabilità un significato inadeguato del resto del codice con conseguenti rischi di rilevazione e segnalazione di errori aggiuntivi. Infatti durante i test a cui è stato sottoposto il Compilatore in questi casi si è talvolta manifestato anche il messaggio di *'Errore di Segmentazione'*; tra l'altro gli errori inficiano anche la corretta creazione dell'oggetto *'*.c'*. La filosofia di riferimento è stata comunque quella di far continuare l'analisi a prescindere dal manifestarsi di errori.

6 - Il generatore di codice C

Il front-End di un Compilatore comprende come ultima fase dell'attività di Analisi, propedeutica a quella di Sintesi, la generazione di un codice oggetto (*target language*). La rappresentazione intermedia può variare a seconda delle finalità del Compilatore: annotated AST, notazione post-fissa, three-address code, codice per macchine virtuali come la JVM o Prolog, un altro linguaggio di alto livello sono tutte possibili soluzioni.

In caso di traduzione tra linguaggi di programmazione di alto livello si genera un particolare tipo di Compilatore detto *'source-to-source compiler'* o *'transcompiler'*. Questi compilatori eseguono un'operazione che in gergo informatico viene detta di *'Porting'* in cui la compilazione consente di *'portare'* il sorgente inizialmente sviluppato per una determinata piattaforma ad un codice che può girare su una piattaforma obiettivo ovviamente diversa da quella iniziale.

A2C è un tipo di Compilatore che appartiene alla categoria appena descritta, traducendo sorgente ADA-like in codice C, compilabile per qualsiasi tipo di architettura di calcolo.

Per adempiere a questo obiettivo è stato creato un modulo ad - hoc dedicato alla traduzione che si individua nel header file *CodGen.h*. Nella libreria sono state implementate diverse funzioni che creano una corrispondenza tra i costrutti del linguaggio ADA-like e quello C; quasi sempre infatti i due linguaggi differiscono nella forma delle tipiche istruzioni come i cicli, le strutture condizionali, le funzioni di input/output.

La generazione del codice tradotto viene ad assimilarsi a quella delle azioni semantiche; infatti seguendo lo stesso principio le istruzioni di traduzione vengono inserite nelle azioni delle regole di produzione del Parser. In tal modo la generazione del file oggetto in C avviene durante la costruzione dello stesso AST.

```
if_statement : IF {gen_if();} relation {gen_fi();} THEN
```

Fig.13: esempio azioni di traduzione

Come convenzione le funzioni di traduzione sono state rese riconoscibili assegnando a tutte l'omonimo prefisso '*gen*'. Inoltre il Generatore assume come nome del file di destinazione C quello dell'identificatore della prima procedura o funzione dichiarata nel file sorgente ADA-like.

7 - Caso applicativo di Test

Per testare la funzionalità del Compilatore è stato creato un file *Adaprova.ada* in cui sono stati inseriti i principali costrutti del linguaggio ADA-like supportati e alcuni degli errori tipici sia sintattici che semantici.

Viene quindi riportato l'output a terminale del Compilatore.

```
-- file Adaprova.ada
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO, Ada.Float_Text_IO; use Ada.Integer_Text_IO, Ada.Float_Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;

with stdio.h;

procedure prova(A, B : in integer := 1; Result : out integer := false) is

    X : integer := "prova";
    N, B : Integer := 3;
    R, G : float := 2.2;
    Z : character := 's';
    Table is array(1 .. 100) of Integer; -- 100 Integers.

    function Square(X: in integer) return integer is
        C : boolean := true;
    begin -- this is the beginning of Square
        N := 1;
```

Fig.14: file Adaprova.ada

```
ruggiero@buntu: ~/ ./ A2C Adaprova.ada
line 5, Error : syntax error
line 7, Error : wrong assignment
line 9, Error : syntax error
line 9, Error : syntax error
line 10, Error : B identifier is already defined
line 18, Error : N is an undeclared identifier
line 19, Error : return type inconsistent
line 32, Error : G orno is not a type
line 39, Error : syntax error
line 41, Error : the type of assignment variables is inconsistent
line 41, Error : the type of assignment variables is inconsistent
line 43, Error : syntax error
line 45, Error : expected type and expression are incompatible
```

Fig.15: output della compilazione del file Adaprova.ada

Elenco degli errori riportati:

linea 5, inclusione di una libreria inesistente;

linea 7, inizializzazione del parametro di una procedura con un valore non coerente;

linea 9, assegnazione incoerente di una variabile di tipo diverso;

linea 10, tentativo di dichiarazione di una variabile già istanziata;

linea 18, uso di un identificatore dichiarato in un'altra procedura;

linea 19, valore di ritorno incoerente con il tipo dichiarato nella funzione;

linea 32, assegnazione tipo inesistente;

linea 39, argomento di operazione I/O errata;

linea 41, espressione con operandi di tipo diverso;

linea 43, statement sintatticamente errato;

linea 45, operazione tra tipi diversi di operandi non consentita;

linea 49, divisione per zero;

linea 59, statement sintatticamente errato;

linea 62, associazione tra operandi di tipo diverso non consentita;

linea 80, usato un indice array inesistente;

linea 83, usato un indice array inesistente;

linea 85, errore nella chiamata di procedura, identificatore errato;

linea 87, usato un tipo diverso di parametro nella chiamata di funzione;

linea 89, errore nella chiamata di procedura, identificatore errato;

linea 91, chiamata di funzione con numero di parametri attuali errato;

linea 95, uso di un identificatore non dichiarato;

linea 96, usato un indice array inesistente;

linea 96, uso di un identificatore non dichiarato.

Allegati

Allegato A: *parser.y*

```
/* *****  
/*  
/* ===== A2C : an Ada-like parser/scanner  
/*  
/* _____ Politecnico di Bari: Formal Languages and  
/*  
/* *****  
/*  
/* _____ author: Campese Ruggi  
/*  
/* *****  
  
/* Prologue */  
%  
    #include <stdio.h>  
    #include <stdlib.h>  
    #include <string.h>  
    #include <math.h>  
    #include <malloc.h>  
    #define true 1  
    #define false 0  
  
    int verb = false;                // verbose mode flag  
  
    #include "varST.h"                // Variable Symbol Table library, VST  
    #include "stackStr.h"            // String Stack library to store identifier  
    #include "stackHvar.h"           // HashTable pointer Stack library to switch  
    #include "proST.h"               // Procedure Symbol Table library, PST  
    #include "typST.h"               // Record Types Symbol Table library, RST  
    #include "CodGen.h"              // C code generator module  
  
    void yyerror (char const *);      // static procedure  
    int context_var_check(char*);     // check identifier  
    int context_pro_check(char*, int); // check procedure  
    void insert_var(char*, char*, typeValPtr, int); // add an entry to the VST  
    void insert_pro(char*);           // add an entry to the PST  
    void insert_par_pro(char*);       // add an entry to the RST  
    void par_check();                 // matrix check  
    int type_checking(char*, char*);  // check type  
    void ass_val(char*);              // assign value  
    void update_val(char*, char*);    // update the value  
    int context_return_check(char*);  // check function return  
    int record_com_check(char*, char*); // check record component  
    void insert_typ_com(char*, char*); // add a component to the RST  
    void insert_arr(char*, char*, int, int); // add an array to the RST  
    int array_bound_check(char*, int); // check array bounds  
    extern int vvlينو;                // line number
```

Allegato B: scanner.l

```
%  
#include <stdio.h>  
#include <string.h>  
#include "parser.tab.h"  
  
extern int verb;  
%  
  
%option case-insensitive  
%option yylineno  
  
digit [0-9]  
id [a-zA-Z][_a-zA-Z0-9]*  
character [a-zA-Z]  
  
%%  
with { return(WTH); }  
use { return(USE); }  
Ada.Text_IO.Ada.Integer_Text_IO.Ada.Float_Text_IO.Ada.Characters.Handling  
procedure { return(PROCEDURE); }  
function { return(FUNCTION); }  
return { return(RETURN); }  
is { return(IS); }  
begin { return(BEGIN); }  
end { return(END); }  
integer { yylval.sval = "integer";  
          return(INTEGER); }  
boolean { yylval.sval = "boolean";  
          return(BOOLEAN); }  
float { yylval.sval = "float";  
        return(FLOAT); }  
character { yylval.sval = "character";  
            return(CHARACTER); }  
": =" { return(ASSIGN); }  
{ digit }+ { yylval.ival = atoi(yytext);  
              return(NUM); }  
{ digit }+\. { digit }+ { yylval.fval = atof(yytext);  
                           return(FNUM); }  
"true" { yylval.ival = 1;  
         return(TRUE); }  
"false" { yylval.ival = 0;  
          return(FALSE); }  
"' { character }'" { yylval.cval = yytext[1];  
                     return(CHAR); }  
type { return(TYPE); }
```

Allegato C: *varSt.h*

```
// Symbol Table to store program variables

#include "ut hash. h"

typedef union {
    int i;
    float f;
    char c;
} typeVal;

typedef typeVal * typeVal Ptr;

typedef struct {
    char name [ 30];
    char type [ 10];
    int l1;
    int l2;
    typeVal value;
    UT_hash_handle hh; /* makes this structure hashable */
} var_hash_struct;

typedef var_hash_struct * var_hash_struct_ptr;

var_hash_struct_ptr var_hash_table = NULL;

/* Add a variable */
void add_var(char *name, char *type, typeVal Ptr * value, int set) {

    var_hash_struct_ptr s;
    s = malloc(sizeof(var_hash_struct));
    strcpy(s->name, name);
    strcpy(s->type, type);
    // s->type = (char*)strdup(type);
    if(verb){
        printf("added id: %s, ", s->name);
        printf("type: %s, ", s->type);
    }
    if(set){
        if ( strcmp( type, "integer" ) == 0 ){
            s->value.i = (**value).i;
            if(verb)
                printf("value: %d\n", s->value.i);
        }
        else if ( strcmp( type, "float" ) == 0 ){
            s->value.f = (**value).f;
        }
    }
}
```

Allegato D: *proST.h*

```
// Symbol Table to store program procedures

typedef struct {
    char name[30] ;
    int numPar; // formal parameters count
    stackStrNodePtr parFor; // formal parameters structure
    char rtnVal[10]; // type of the returned value
    typeVal value; // returned value
    UT_hash_handle hh; // makes this structure hashable
} pro_hash_struct;

typedef pro_hash_struct* pro_hash_struct_ptr;

pro_hash_struct_ptr pro_hash_table = NULL;

// Add a procedure
void add_pro(char *name) {

    pro_hash_struct_ptr s;
    s = malloc(sizeof(pro_hash_struct));
    strcpy(s->name, name);
    s->parFor = NULL;
    if(verb)
        printf("added procedure: %s\n", s->name);

    HASH_ADD_STR( pro_hash_table, name, s ); /* name is the string-keyed field */
}

// Find a procedure
pro_hash_struct_ptr find_pro(char* name) {

    pro_hash_struct_ptr s;
    HASH_FIND_STR( pro_hash_table, name, s ); /* s: output pointer */
    return s;
}

void update_type(char* proc, char* type){
    pro_hash_struct_ptr s = find_pro(proc);
    strcpy(s->rtnVal, type);
    if(verb)
        printf("the type of %s return value is %s\n", s->name, s->rtnVal);
}

void add_par_pro(char *name, char *type) {
```

Allegato E: *typST.h*

```
// Symbol Table to store program types and records

typedef struct {
    char name [ 30];
    char type [ 30];
    UT_hash_handle hh;          /* makes this structure hashable */
} typ_hash_struct;

typedef typ_hash_struct* typ_hash_struct_ptr;

typ_hash_struct_ptr typ_hash_table = NULL;

// Add a type
void add_typ(char *name) {

    typ_hash_struct_ptr s;
    s = malloc(sizeof(typ_hash_struct));
    strcpy(s->name, name);
    if(verbose)
        printf("added record: %s\n", s->name);

    HASH_ADD_STR( typ_hash_table, name, s ); /* type is the string-keyed field */
}

// Find a type
typ_hash_struct_ptr find_typ(char* type) {

    typ_hash_struct_ptr s;
    HASH_FIND_STR( typ_hash_table, type, s ); /* s: output pointer */
    return s;
}

void add_typ_com(char* name, char* comp, char* type) {

    char com[ 50];

    if(find_typ(name)){
        sprintf(com, "%s.%s", name, comp);
        typ_hash_struct_ptr s;
        s = malloc(sizeof(typ_hash_struct));
        strcpy(s->name, com);
        strcpy(s->type, type);
        if(verbose)
            printf("added record component: %s type: %s\n", s->name, s->type);
    }
}
```