|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Documento di manutenzione  Compilatore-YASPL   |  |  | | --- | --- | | Riferimento |  | | Versione | 1.2 | |

RevisionHistory

|  |  |  |  |
| --- | --- | --- | --- |
| **Data** | **Versione** | **Descrizione** | **Autori** |
| 12/08/2019 | 1.0 | Prima stesura | Mazzotta Fabio |
| 22/08/2019 | 1.1 | Aggiornamento tool manutenzione | Mazzotta Fabio |
| 02/09/2019 | 1.2 | Aggiunta parte relativa al testing | Mazzotta Fabio |

Sommario

[RevisionHistory 2](#_Toc17816932)

[1. Descrizione del progetto. 4](#_Toc17816933)

[1.1 Analizzatore lessicale. 4](#_Toc17816934)

[1.2 Analizzatore sintattico. 5](#_Toc17816936)

[1.3 Analizzatore semantico. 7](#_Toc17816937)

[1.4 Generazione del codice. 8](#_Toc17816938)

[2. Il tool di manutenzione 8](#_Toc17816939)

[3. Ulteriori correzioni 9](#_Toc17816940)

[4. Testing 9](#_Toc17816941)

1. Descrizione del progetto.

Il progetto software tratta lo sviluppo di un compilatore che traduce il linguaggio YASPL 2.0 in linguaggio C. Sono state implementate le seguenti componenti:

* Analizzatore lessicale,
* Analizzatore sintattico,
* Analizzatore semantico e
* Generatore di codice.

Di seguito vengono descritte nei dettagli le varie componenti.

1.1 Analizzatore lessicale.

L’analizzatore lessicale è stato implementato con il tool JFlex e riconosce i seguenti token:

*HEAD* ***head***

*START* ***start***

*SEMI* ***;***

*INT* ***int***

*BOOL* ***bool***

*DOUBLE* ***double***

*NAME* [a-zA-Z0-9\_]\*

*COMMA* ***,***

*DEF* ***def***

*LPAR* ***(***

*RPAR* ***)***

*COLON* ***:***

*LGPAR* ***{***

*RGPAR* ***}***

*READ* ***<-***

*WRITE* ***->***

*PLUS* ***+***

*MINUS* ***-***

*TIMES* ***\****

*DIV* ***/***

*INT\_CONST* [0-9]\*

*DOUBLE\_CONST* [0-9.]\*

*STRING\_CONST pattern per stringhe*

*TRUE* ***true***

*FALSE* ***false***

*ASSIGN* ***=***

*IF* ***if***

*THEN* ***then***

*WHILE* ***while***

*DO* ***do***

*ELSE* ***else***

*GT* ***>***

*GE* ***>=***

*LT* ***<***

*LE* ***<=***

*EQ* ***==***

*NOT* ***not***

*AND* ***&&***

*OR* ***||***

*UMINUS* ***-***

*COMMENTIINLINEA* **//**

*COMMENTISUPIURIGA* ***/\* \*/***

*CONTENUTOCOMMENTO*

*TAB*

*NEWLINE*

*WHITESPACE*

L’analizzatore lessicale usa le espressioni regolari per riconoscere i token all’interno del codice sorgente. La componente software è la classe ***Lexer*** che è presente nel pacchetto ***cup.example.***

1.2 Analizzatore sintattico.

L’analizzatore sintattico è stato prodotto con il tool JavaCUP e si occupa di verificare che una sequenza di token appartenga alla grammatica del compilatore. La grammatica del compilatore è la seguente:

***Programma → HEAD Decls START Statements***

***Decls → Var\_decl Decls***

***| Def\_decl Decls***

***| ε***

***Statements → Stat Statements***

***| Stat***

***Var\_decl → Type Vars SEMI***

***Type → INT | BOOL | DOUBLE***

***Vars → NAME COMMA Vars***

***| NAME***

***Types → Type COMMA Types***

***| Type***

***Def\_decl → DEF NAME LPAR Var\_decls RPAR COLON Par\_decls Body***

***Var\_decls → Var\_decl Var\_decls***

***| ε***

***Par\_decls → Var\_decl Par\_decls***

***| Var\_decl***

***Body → LGPAR Var\_decls Statements RGPAR SEMI***

***Comp\_stat→ LGPAR Statements RGPAR***

***Stat → Comp\_stat***

***| Simple\_stat***

***Simple\_stat→ Vars READ Types***

***| Out\_values WRITE SEMI***

***| NAME ASSIGN Expr SEMI***

***| NAME LPAR Exprs COLON Vars RPAR SEMI***

***| IF LPAR Bool\_expr RPAR THEN Comp\_stat ELSE Comp\_stat SEMI***

***| IF LPAR Bool\_expr RPAR THEN Comp\_stat SEMI***

***| WHILE LPAR Bool\_expr RPAR DO Comp\_stat SEMI***

***Out\_values → Expr COMMA Out\_values***

***| STRING\_CONST COMMA Out\_values***

***| Expr***

***| STRING\_CONST***

***Exprs → Expr COMMA Exprs***

***| Expr***

***Expr → INT\_CONST***

***| DOUBLE\_CONST***

***| Expr Arith\_op Expr***

***| NAME***

***| UMINUS Expr***

***| LPAR Expr RPAR***

***Arith\_op → PLUS | MINUS | TIMES | DIV***

***Bool\_expr → TRUE***

***| FALSE***

***| Bool\_expr Bool\_op Bool\_expr***

***| NOT Bool\_expr***

***| Expr Rel\_op Expr***

***| LPAR Bool\_Expr RPAR***

***Bool\_op → AND | OR***

***Rel\_op → GT | GE | LT| LE | EQ***

L’analizzatore sintattico è stato progettato in maniera tale che ogni produzione crea un nodo dell’albero sintattico. L’abstract syntax tree viene stampato su un file XML. Le componenti che implementano l’AST sono presenti nel pacchetto ***yaspl2.*** Le sue componenti sono:

* Programma
* Par\_decl
* Body
* Bool\_expr
* Expr
* Var\_decl
* Stat
* Def\_decl

Ogni classe, rappresenta un non-terminale ed ha delle classi interne che indicano i vari nodi dell’AST delle varie produzioni. La componente software che implementa l’analizzatore sintattico è la classe ***Parser*** presente nel pacchetto ***cup.example***. Le classi implementano l’interfaccia ***AzioniCompilatore*** che contiene il metodo ***drawComponent()*** che disegna il nodo sul file XML. Alla fine viene stampato l’abstract syntrax tree.

1.3 Analizzatore semantico.

L’analizzatore semantico si occupa di gestire lo scope delle variabili e il type checking. Le azioni che esegue nello specifico sono:

* Controllo dei tipi delle variabili
* Controllo sul tipo dei parametri della chiamata a funzione con la sua definizione
* Controllo del numero dei parametri della chiamata a funzione con la sua definizione
* Controllo sull’esistenza della funzione
* Controllo sul tipo di output della funzione assegnato alla variabile
* Controllo sulla quantità di dati restituito dalla funzione.
* Controllo sui tipi delle variabili nella funzione Read
* Controllo sui tipi delle variabili nella funzione Write
* Controllo dello scoping all’interno del codice sorgente yaspl

Le componenti sono le stesse per l’analizzatore sintattico e le classi implementano il metodo ***startScoping()***,definito nell’interfaccia ***AzioniCompilatore***,che implementa le operazionieffettuate dall’analizzatore semantico. Ogni qualvolta viene violato un controllo viene lanciata un eccezione che segnala il problema. Inoltre viene disegnato un abstract syntax tree con le informazioni ricavate dall’analisi semantica in cui vengono rappresentati i tipi di dati dei nodi. Per conoscere i tipi di dati dei vari nodi è stata aggiunta l’interfaccia ***OttieniTipo*** all’interno del pacchetto ***analizzatoresemantico*** che contiene il metodo ***getType()*** equesto metodo viene implementato dalle classi ***Bool\_expr, Expr, Par\_decl*** in quanto sono le classi che gestiscono i dati del sorgente e i loro tipi. Per gestire lo scope delle variabili è stata creata la classe ***Env*** presente all’interno del pacchetto ***analizzatoresemantico.***

1.4 Generazione del codice.

Ogni componente del sistema calcola la corrispondente istruzione in linguaggio C. Ogni componente implementa il metodo ***scriviCodice()*** dell’interfaccia ***AzioniCompilatori*** in cui scrive il codice in linguaggio C su un file apposito. Per avviare il processo di compilazione bisogna eseguire il metodo ***main()*** all’interno della classe ***Driver*** presente nel pacchetto ***cup.example.***

2. Il tool di manutenzione

Il compilatore ha subito un processo di manutenzione ed esegue l’analisi del flusso dati. Consente di rilevare anomalie sull’utilizzo delle variabili all’interno del codice sorgente. I valori calcolati sulle variabili sono:

* **Definizione (d):** assegna un valore alla variabile x (istruzioni di assegnamento e istruzioni di input),
* **Uso (u):** usa il valore della variabile in un’istruzione di output, in una espressione per il calcolo di un’altra variabile, o in un predicato
* **Annullamento (a):** al termine dell’esecuzione dell’istruzione il valore assegnato alla variabile non è più significativo.

La definizione di una variabile, così come un annullamento, cancella l’effetto di una precedente definizione della stessa variabile, ovvero ad essa è associato il nuovo valore derivante dalla nuova definizione (o il valore nullo). Per stabilire eventuali anomalie si seguono due regole:

1. L’uso di una variabile x deve essere sempre preceduto in ogni sequenza da una definizione della variabile x, senza annullamenti intermedi. Infatti un uso non preceduto da una definizione può corrispondere al potenziale uso di un valore non determinato.
2. Una definizione di una variabile x deve essere seguita da un uso della variabile x, prima di un’altra definizione o di un annullamento di x. Se una definizione non è seguita da un uso corrisponde all’assegnamento di un valore non utilizzato e quindi potenzialmente inutile.

Le espressioni calcolate delle variabili vengono usate per costruire l’algoritmo di **reaching definition.** L’algoritmo opera nel seguente modo:verifica in quali istruzioni una definizione di una variabile x raggiunge senza essere “uccisa” da una nuova definizione. Per implementare ciò è stato aggiunto il metodo ***controlFlowDati()*** nell’interfaccia ***AzioniCompilatori*** ed ogni componente del pachetto ***yaspl2*** calcola il corrispondente valore del flusso dati. Per tenere traccia dei valori delle varie variabili all’interno di ogni componente è stata creata la classe ***TracciaDati*** presente nel pacchetto ***toolmanutenzione.*** Infine i risultati dell’algoritmo vengono scritti in un file usando il metodo ***eseguiReachingDefinition()*** della classe ***TracciaDati***.

3. Ulteriori correzioni

Durante il processo di manutenzione sono stati corretti i seguenti bug:

* Riconoscimento dell’istruzione unaria per rappresentare i numeri negativi in quanto precedentemente non venivano riconosciute a causa delle precedenze sui segni non inseriti nel parser.
* Miglioramenti sulla gestione delle operazioni delle priorità delle parentesi in quanto venivano calcolate in modo scorretto
* Riparazione del riconoscimento semantico del nome delle funzioni, poiché era possibile inserire il nome delle funzioni e delle variabili con lo stesso identificativo.

4. Testing

Per verificare se il tool di manutenzione funziona correttamente e soddisfa tutti i requisiti stabiliti è stata effettuata l’attività di testing. Per il testing è stato realizzato il control flow graph (CFG) dove ogni nodo rappresenta un istruzione del codice sorgente ed ogni arco rappresenta il passaggio del flusso che avviene da un nodo all’altro. Il grafo riporta le informazioni sui valori calcolati e le rispettive variabili. In questo modo è possibile verificare se i valori calcolati dall’algoritmo di reaching definition coincidono con quelli del grafo. Se coincidono i risultati calcolati dall’algoritmo di reaching definition sono corretti, altrimenti sono errati. Inoltre il CFG viene usato per verificare come vengono eseguiti i vari statements***.*** Il grafo è stato creato inserendo il metodo ***drawNode()*** all’interno dell’interfaccia ***AzioniCompilatore*** che viene poi implementato da ogni componente presente nel pacchetto ***yaspl2.*** Ogni componente calcola il nodo corrispondente ed infine il grafo viene rappresentato su un file XML.