Relazione Progetto Compilatori 20/21

Gruppo: Nicola Levorato 1241689, Marco Alessio 1242412

1. Progettazione del linguaggio

Per lo svolgimento del progetto è necessario ideare e definire un linguaggio di programmazione e sviluppare un compilatore che traduca programmi per tale linguaggio in codice 3AC. È richiesto che esso supporti:

- Istruzioni di dichiarazione di variabili, esclusivamente di tipo `int`.
- Istruzioni di assegnazione di variabili.
- Espressioni aritmetiche e logiche, contenenti sia letterali che variabili.
- Istruzioni `if`, `if-else` e `while`.
- Istruzione di stampa a console del valore di una variabile.

Per definire la sintassi del linguaggio, abbiamo scelto di prendere come riferimento il linguaggio C, introducendovi però alcune modifiche:

- Nonostante sia possibile dichiarare solamente variabili di tipo `int`, abbiamo deciso di considerare esplicitamente espressioni di tipo booleano, in modo similare a Java. Una espressione booleana può essere ottenuta usando le parole chiave `true` e `false` oppure un operatore di confronto tra due espressioni intere, e possono essere combinate tra loro usando operatori logici. Le espressioni booleane non possono essere assegnate ad una variabile intera, vengono impiegate nelle condizioni delle istruzioni `if` e `while`, e possono essere argomento della funzione `print()`.
- Esistono due parole chiave `true` e `false` per rappresentare rispettivamente i due valori booleani vero e falso.
- Gli operatori logici `&&`, `||` e `!` sono rappresentati rispettivamente dai token `and`, `or` e `not`.
- Per stampare in output variabili, espressioni intere e booleane, esiste la funzione `print()` invece di usare una funzione di libreria come printf().
- È possibile usare un singolo operatore di assegnazione o di assegnamento con operazione in una istruzione. Ciò comporta che istruzioni C valide come `x = y = 0;` oppure `x += y *= 5;` non sono consentite nel nostro linguaggio e devono essere separate in più istruzioni equivalenti.

2. Implementazione del compilatore

Per implementare l'analizzatore lessicale del nostro compilatore abbiamo usato Flex, mentre per l'analizzatore sintattico Bison.

Tra i token riconosciuti dall'analizzatore lessicale, oltre a quelli "standard", abbiamo incluso l'operatore di resto della divisione intera (`%`), gli operatori di assegnazione con operazione (`+=`, `-=`, `%=`, …), gli operatori di incremento e decremento unario (`++` e `--`), gli operatori di shift (`<<` e `>>`) e l'operatore di complemento unario (`~`). Inoltre, vengono riconosciuti ed ignorati i commenti sia a linea singola (`//` …) che multilinea (`/*` … `*/`).

Nell'implementazione dell'analizzatore sintattico, il tipo associato a tutti i token e a tutti i simboli non terminali è il medesimo, la *struct* "address". In questo modo la loro gestione in Bison si semplifica notevolmente, potendo assegnare un valore solamente a quei campi di interesse per lo specifico token o simbolo non terminale della grammatica ed ignorando i rimanenti. La *struct* "address" è così definita:

- addr: contiene il nome della variabile o della variabile temporanea in cui è memorizzato il valore del token/simbolo considerato.
- type: contiene il nome del tipo di dato considerato. Viene usato esclusivamente nell'istruzione di dichiarazione di variabili e può assumere esclusivamente il valore `int` nel nostro progetto.
- next_label: contiene il nome di una etichetta che punta alla prima istruzione che segue l'istruzione `if` o `while` corrente.
- true_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire qualora la condizione di una istruzione `if` o `while` sia vera.
- false_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire qualora la condizione di una istruzione `if` sia falsa.
- begin_label: contiene il nome di una etichetta che punta alle istruzioni da eseguire per valutare la condizione di una istruzione `while`.

Per implementare la *symbol table* abbiamo sfruttato l'implementazione *open-source* di una *hash table* in C di Troy D. Hadson (file "uthash.h") [1]. Per non complicare eccessivamente il nostro progetto, abbiamo deciso di non considerare il ciclo di vita delle variabili; in questo modo una variabile sarà sempre visibile dal punto in cui viene dichiarata fino al termine del programma.

Per gestire la dichiarazione delle variabili e l'assegnazione del tipo corrispondente abbiamo utilizzato la tecnica dello *stack* descritta in classe, anche se l'unico tipo disponibile nel nostro linguaggio è il tipo intero. Questo metodo permette di semplificare la gestione di più tipi di dato, qualora si volesse estendere il linguaggio e renderlo più complesso.

I campi *label* next_label, true_label, false_label e begin_label vengono generati solamente per le istruzioni che ne fanno uso, come le istruzioni `if` e `while`, mentre vengono ignorati per tutte le altre istruzioni. In questo modo si evita di creare etichette inutilmente quando non sono necessarie. La loro gestione avviene tramite il passaggio di attributi ereditati: vengono creati tramite un non

terminale fittizio e la loro stampa avviene tramite delle *mid-rule action*, sfruttando inoltre la tecnica dello *stack* vista a lezione.

L'istruzione `if` è stata così implementata nel nostro compilatore:

1. Viene riconosciuto il token `if`.
2. Vengono generate le etichette true_label e false_label tramite una produzione fittizia.
3. Viene riconosciuto il token `(`, la condizione booleana e la parentesi chiusa `)`.
4. Tramite una *mid-rule action* viene generato il codice 3AC per:
    a. Saltare a true_label se la condizione risulta essere vera.
    b. Saltare a false_label (verrà eseguito il salto solo se la condizione risulta essere falsa).
    c. Viene stampata l'etichetta true_label.
5. Viene riconosciuto il blocco di istruzioni da eseguire per il ramo true dell'if.
6. Se viene riconosciuto il token `else`:
    a. Viene generato l'etichetta next_label.
    b. Viene generato il salto a next_label al fine del blocco di istruzioni del ramo true.
    c. Viene stampata l'etichetta false_label.
    d. Viene riconosciuto il blocco di istruzioni da eseguire nel ramo false dell'if.

    Altrimenti:
    e. L'etichetta false_label viene assegnata al campo next_label.
7. Viene stampata l'etichetta next_label.

Abbiamo implementato l'istruzione `while` in questo modo:

1. Viene riconosciuto il token `while`.
2. Vengono generate le etichette begin_label, true_label e next_label tramite una produzione fittizia.
3. Viene stampata l'etichetta begin_label, tramite una mid-rule action.
4. Viene riconosciuto il token `(`, la condizione booleana e la parentesi chiusa `)`.
5. Tramite una *mid-rule action* viene generato il codice 3AC per:
    a. Saltare a true_label se la condizione risulta essere vera.
    b. Saltare a next_label (verrà eseguito il salto solo se la condizione risulta essere falsa).
    c. Viene stampata l'etichetta true_label.
6. Viene riconosciuto il blocco di istruzioni da eseguire nel corpo del while.
7. Viene generato il salto a begin_label alla fine del corpo del while, per poter rivalutare la condizione del ciclo.
8. Viene stampata l'etichetta next_label.

Nell'implementazione delle istruzioni `if` e `while` vengono creati dei salti incondizionati aggiuntivi non necessari (come il salto a true_label nell'istruzione `while`, il salto ad uno dei due rami di un'istruzione `if` o nel caso in cui siano presenti istruzioni innestate come ad esempio un `if` all'interno di un `while`), tuttavia, nonostante ciò, il codice 3AC ottenuto risulta essere corretto.

Nell'implementazione del compilatore, abbiamo creato la macro "PROJECT_LOGGING" per abilitare o disabilitare la stampa in *output* di stringhe di *log*, al fine di *debuggare* più facilmente il codice del progetto.

Per compilare ed eseguire il progetto abbiamo usato i seguenti comandi:

```
1. bison -d compiler.y
2. flex scanner.fl
3. gcc compiler.tab.c lex.yy.c
4. a.exe < input_i.txt
```

Il primo comando restituisce in output un warning su un conflitto shift/reduce. Questo è dovuto all'ambiguità del dangling-else che è risolta correttamente da Bison andando a eseguire lo shift e quindi impilando il token else nel caso in cui venga riconosciuto.

3. Principali difficoltà riscontrate

Le maggiori difficoltà riscontrate durante lo sviluppo del compilatore sono state:

- La gestione degli attributi ereditati: inizialmente abbiamo provato ad implementarli usando la SDD presente nelle slide viste a lezione; tuttavia, abbiamo riscontrato difficoltà riguardo la gestione dello *stack*.
- La gestione dell'ambiguità dovuta al dangling-else: a causa del fatto che le due istruzioni `if` e `if-else` condividono la stessa parte iniziale, qualora si usino *mid-rule action* differenti per implementare gli attributi ereditati si verificano dei conflitti *reduce*/*reduce*, poiché i token sono gli stessi ma Bison non sa che azione eseguire. Per risolvere questo problema abbiamo usato un trucco: la prima parte dell'`if` è la stessa per entrambi i casi, quindi next_label coincide con false_label, dopo aver processato il token `else` devo assegnare a next_label una nuova label. Questo è possibile grazie al fatto che nell'`if-else` la label next_label viene stampata solamente dopo aver processato il token `else`.

4. Bibliografia

[1] https://github.com/troydhanson/uthash