Compilatori, A.A. 2020-2021, secondo semestre

Relazione Finale (Progetto: Compilatore 3AC)

Luca Badin 1242060

Indice

Sintassi e lexer

Il linguaggio, che ho ribattezzato "Rebus"[1], ha la seguente sintassi:

- Statements terminati da semicolon (";")
- Un solo tipo permesso (intero con segno)
- Ciclo while del tipo

```
while (<boolean condition>) { ... }
```

- Statement if a una branch del tipo

```
if (<boolean condition>) { ... }
```

- Statement if con else branch del tipo

```
if (<boolean condition>) { ... }
else { ... }
```

- Dichiarazione di variabile, anche con assegnazione ad espressione aritmetica

```
int a, b = 0, c;
```

- Assegnazione di una variabile al risultato di un'espressione aritmetica
- Istruzione di stampa a console di espressioni aritmetiche

```
print <arithmetic expression>;
```

- Quattro operatori aritmetici (+,-,*,/) e unary minus
- Sei operatori relazionali (==, !=, <, <=, >, >=)
- Tre operatori booleani (&&, ||, !)

Con espressione aritmetica, si intendono le produzioni

$$E \rightarrow E\ op\ E \mid -E \mid (E) \mid \textbf{num} \mid \textbf{id}$$

dove $op = \{+, -, *, /\}$. Con espressione booleana, si intendono le produzioni

$$B \rightarrow B\ \&\&\ B \mid B||B \mid !B \mid B \mid E\ rel\ E$$

dove $rel = \{==, !=, >, <, >=, <=\}$.

La singola routine che compone il programma non richiede signature né parentesi graffe.

[1] "Rebus" era il nome di un linguaggio di programmazione interpretato che provai a scrivere per divertimento personale (senza nessuna conoscenza di linguaggi formali) mentre ero alle superiori tra il 2013 e il 2015. Il codice incompleto dell'ultima versione sopravvive su GitHub: https://github.com/badinl/ReBUS

Implementazione del compilatore e parser

Gestione delle variabili

Le variabili dichiarate sono salvate in una symbol table con i seguenti campi:

`char * identifier`	Letterale che identifica la variabile, così come compare nel codice sorgente
`char * type`	Tipo della variabile
`int address`	Indirizzo univoco che identifica la variabile nella symbol table e nel codice 3AC generato

Non è possibile né assegnare a variabili non dichiarate, né in altro modo accedere a variabili non dichiarate, e neppure ridichiarare variabili già dichiarate. Tutti questi casi portano a un errore terminale.

Una volta dichiarate, alle nuove variabili viene associato un indirizzo intero semipositivo. Questo intero viene utilizzato nel three-address code generato per riferirsi alla variabile, nel formato `s<n>`. La scelta di non permettere ai nomi di variabile scelti dall'utente di rimanere nel codice finale è per evitare che, ad esempio, una variabile venga chiamata con un formato utilizzato dalle variabili temporanee (`t<n>`).

La dichiarazione di variabile può anche essere fatta con simultanea assegnazione. In questo caso, dal punto di vista del three-address code generato:

- il valore di inizializzazione è assegnato a una variabile temporanea;
- la nuova variabile viene dichiarata;
- il valore della variabile temporanea è assegnato alla nuova variabile.

Le variabili temporanee NON vengono memorizzate in alcuna struttura. Nel caso volessimo implementare più tipi, sarebbe probabilmente necessario mantenere una tabella simile alla symbol table (o direttamente aggiungere le variabili temporanee alla symbol table) per eseguire type checking statico.

Backpatching

La soluzione scelta per ovviare al problema degli attributi ereditati in Bison è il backpatching, in quanto giudicato più "elegante" da implementare. Il backpatching è stato implementato tramite:

- i nonterminali di tipo `backpatch`, che propagano verso l'altro strutture di tipo `bp_payload` contenenti la lista di espressioni true e false (truelist e falselist), per i branch condizionati usati da loop statements ed espressioni booleane, e la nextlist, usata dagli statements per stampare le etichette di uscita;
- funzioni `bp_new()` and `bp_merge()` per la manipolazione di liste di backpatch;
- la funzione `ia_backpatch()`, che effettua il backpatching delle righe di codice nell'IA indicate da una backpatch list in input, con un valore di riga indicato.

Necessaria al backpatching è la presenza di un instruction array, che è stato implementato con una semplice single linked list di stringhe. Il compilatore utilizza la funzione `ia_emit()` per produrre nuove linee di codice, `ia_generate()` per produrre il listato finale, e `ia_count()` per ottenere il numero di righe presenti (e pertanto il numero di riga della prossima istruzione three-address code, da inserire nelle nextlist).

Si segnala inoltre che le righe di three-address code prodotte dal compilatore sono zero-indexed; questo perché con il backpatching non esistono label, o meglio, le label sono gli indici dell'instruction array, che è zero-indexed. Peraltro, sembra realistico che si possano sommare gli indirizzi relativi a cui puntano le istruzioni di jump a un ipotetico indirizzo assoluto a cui è ospitata la prima istruzione.

Espressioni booleane: short-circuit code

Una delle scelte progettuali è stata di implementare le espressioni booleane utilizzando short-circuit code, dal momento che è un metodo che ben si integra con la tecnica del backpatching.

La grammatica e azioni semantiche implementate sono essenzialmente quelle discusse durante il corso, con l'eccezione dell'assenza dei simboli terminali true e false, che sono stati ritenuti superflui.

Conflitti

Un problema sorto al momento dell'implementazione pratica del backpatching è che le due regole

$$S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1, \qquad S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$$

generano un conflitto reduce/reduce in Bison. Quando incontra uno statement if-else, il compilatore utilizza sempre la prima produzione per poi segnalare errore sintattico quando incontra il token else.

Entrambi i nonterminali M ed N sono ε-produzioni che servono solamente come "aree di parcheggio". La soluzione trovata è stata di modificare il comportamento di N in modo che sia una produzione $N \rightarrow \textbf{else}$. Pertanto:

$$S \rightarrow \textbf{if}\ (\ B\ )\ M\ S_1, \qquad S \rightarrow \textbf{if}\ (\ B\ )\ M_1\ S_1\ N\ M_2\ S_2, \qquad N \rightarrow \textbf{else}$$

In questo modo, Bison segnala un conflitto shift/reduce, che viene sempre risolto a favore dello shift. Siccome questo è il comportamento che noi ci attendiamo, il warning può essere ignorato senza dover aggiungere regole di precedenza esplicite.

Test e risultati

Ci sono tre opzioni di debug integrate nel compilatore, immediatamente sotto gli header in `parser.y` (è necessario ricompilare per abilitarle o disabilitarle):

- `trace`, che stampa informazioni sulle riduzioni effettuate dal parser;
- `debug`, che stampa informazioni sulle azioni semantiche;
- `print_line_num`, che stampa i numeri di riga nel listato in output.

backpatch.rebus

Il programma contiene l'esempio di backpatching in un'espressione booleana visto nel set di slide "06 - Intermediate Code (part III - control flow)":

$$x < 100 \,||\, x > 200 \,\&\&\, x \,!= y$$

Il file sorgente contiene il seguente input:

```
int x, y;
if (x < 100 || x > 200 && x != y) print x;
else print y;
```

Il compilatore produce il seguente output:

```
0:    int s0
1:    int s1
2:    if s0 < 100 goto 8
3:    goto 4
4:    if s0 > 200 goto 6
5:    goto 10
6:    if s0 != s1 goto 8
7:    goto 10
8:    print s0
9:    goto 11
10:   print s1
```

fibonacci.rebus

Il programma stampa i casi base e le prime 10 iterazioni successive della sequenza di Fibonacci, come esempio di ciclo while, dichiarazioni con assegnazioni e operazioni.

Il file sorgente contiene il seguente input:

```
int a = 0, b = 1, c;
print a;
print b;
int i = 0;
while(i < 10) {
    c = b + a;
    a = b;
    b = c;
    print c;
}
```

Il compilatore produce il seguente output:

```
0:    t0 = 0
1:    int s0
2:    s0 = t0
3:    t1 = 1
4:    int s1
5:    s1 = t1
6:    int s2
7:    print s0
8:    print s1
9:    t2 = 0
10:   int s3
11:   s3 = t2
12:   if s3 < 10 goto 14
13:   goto 20
14:   t3 = s1 + s0
15:   s2 = t3
16:   s0 = s1
17:   s1 = s2
18:   print s2
19:   goto 12
```

nested.rebus

Il programma testa il funzionamento di un while inserito in un if-else, con la presenza di altri tipi di statement (dichiarazione con assegnazione, operazioni booleane, etc).

Il file sorgente contiene il seguente input:

```
int a = 3, b = 2;
print a;
if (a > 2 && !(a == b)) {
    print b;
    int c = 0;
    while (c <= 2) {
        int d = c;
        print d;
    }
    a = a + 1;
} else {
    print b;
}
print a;
```

Il compilatore produce il seguente output:

```
0:    t0 = 3
1:    int s0
2:    s0 = t0
3:    t1 = 2
4:    int s1
5:    s1 = t1
6:    print s0
7:    if s0 > 2 goto 9
```

```
8:    goto 25
9:    if s0 == s1 goto 25
10:   goto 11
11:   print s1
12:   t2 = 0
13:   int s2
14:   s2 = t2
15:   if s2 <= 2 goto 17
16:   goto 22
17:   t3 = s2
18:   int s3
19:   s3 = t3
20:   print s3
21:   goto 15
22:   t4 = s0 + 1
23:   s0 = t4
24:   goto 26
25:   print s1
26:   print s0
```