Il linguaggio C

Tipi, variabili e costanti

Tipi elementari

- un tipo è caratterizzato dall'insieme dei valori rappresentati e dalle operazioni che ci si possono applicare
- Tipi elementari:
  `char`, `int`, `float`, `double`, `void`
- `<type>::= char | [unsigned] [short | long] int | float | [long] double | void`

- non esistono variabili di tipo void
- non esistono tipi booleani
  - in c++ c'è `bool`
- `<limits.h>` e `<float.h>` riportano i valori limite dei tipi

Variabili

- Sono una locazione di memoria che contiene un valore di un tipo
  - il valore può cambiare, il tipo no
- Le variabili devono essere dichiarate prima dell'uso:
  `<declaration>::=<type><decl>;`
  `<decl>::=<identifier>|...`

- Una variabile dichiarata può essere referenziata attraverso il suo identificatore:
  `<var>::= <identifier> | (<identifier>) | ...`
- La semantica del riferimento a `<var>` consiste nell'individuare la variabile denotata dal riferimento

Costanti

- Sono valori non modificati nel corso dell'elaborazione
- Hanno un tipo dichiarato implicitamente o esplicitamente, es:

  12.
  unsigned 12

Esercizi

- Costruire gli alberi sintattici di:
  - int a;
  - unsigned long int b;

Operatori ed espressioni

- I termini elementari delle espressioni sono variabili e costanti
- <expr>::= <const> | <var> | ++<var> | --<var> | <var>++ | <var>-- | ...

- <expr>::= ... | <expr1> <op2> <expr2> | <op1> <expr2> | ...
- <op2>::= + | - | * | / | % | < | <= | == | != | >= | > | && | || | & | | | << | >>
- <op1>::= ! | ~

- L'assegnamento del risultato di un'espressione è un operatore:
  `<expr>::= ... | <var>=<expr1>`
  `| <var><op>=<expr2> | ...`
- Operatori per raccogliere, concatenare e sequenzializzare:
  `<expr>::= ... | (<expr1>) |`
  `<expr1>,<expr2> | <expr1>?`
  `<expr2>:<expr3>|...`

```
<expr>::= ... | (<cast-type>)<expr2> |
(<cast-type><expr2>) | ...
<cast-type>::= <type> | ...
```

- La sintassi di cast-type verrà estesa aggiungendo i puntatori

Puntatori

- I puntatori sono variabili i cui valori sono indirizzi di locazioni in cui sono memorizzate altre variabili
    - architettura a 32 bit: $2^{32}-1$ indirizzi, ma non si usa unsigned int come tipo (tipo=valori + operazioni)
    - hanno senso somme e sottrazioni, non altre operazioni
    - soprattutto si definisce l'operazione per referenziare la variabile identificata dal valore del puntatore

Dichiarazione puntatori

- si usa il modificatore * prima del nome della variabile, es.:

```
int *pA;  /* pA è puntatore a
int */

int **ppA; /* ppA è puntatore
a puntatore a int */
```

Sintassi puntatori

- Sintassi di `<decl>` che tiene conto dei puntatori:

```
<decl>::= <identifier> |
*<decl> | ...
```

Operatore dereferenziazione

- * applicato in forma prefissa ad un'espressione che restituisce un valore di tipo puntatore, es.:

```
int *pA;
....
....
*pA = 12; /* assegna 12 alla
variabile puntata da pA */
```

Sintassi dereferenziazione

- `<var>::= <identifier> | *<expr> | ...`
- Quindi `*pA = ...` è legale

Operatore di indirizzo

- L'operatore `&` è all'incirca l'inverso di `*`
- `<expr> ::= ... | &<var> | ...`
  - Γ(&<var>) restituisce il valore dell'indirizzo della variabile referenziata da `<var>`. Non ha side effects.
    - `&A = ...` non è legale (compara la sintassi con quella di `*A`)!

Puntatori a void

- Si possono dichiarare puntatori di tipo void, es.:

```
void *ptr;
int n;
ptr = &n;
```

- il compilatore sa quanti byte servono per rappresentare un indirizzo: posso usare &...

- Per dereferenziare un puntatore void devo fare un cast
- come farebbe il compilatore a sapere quanti byte leggere...

```
void *ptr;
int m,n;
ptr = &n;
m=*((int *)ptr);
```

Sintassi cast di puntatore

- `<expr>::= ... | (<cast-type>) <expr1> | ...`
- `<cast-type> ::= <type> | <cast-type> *`