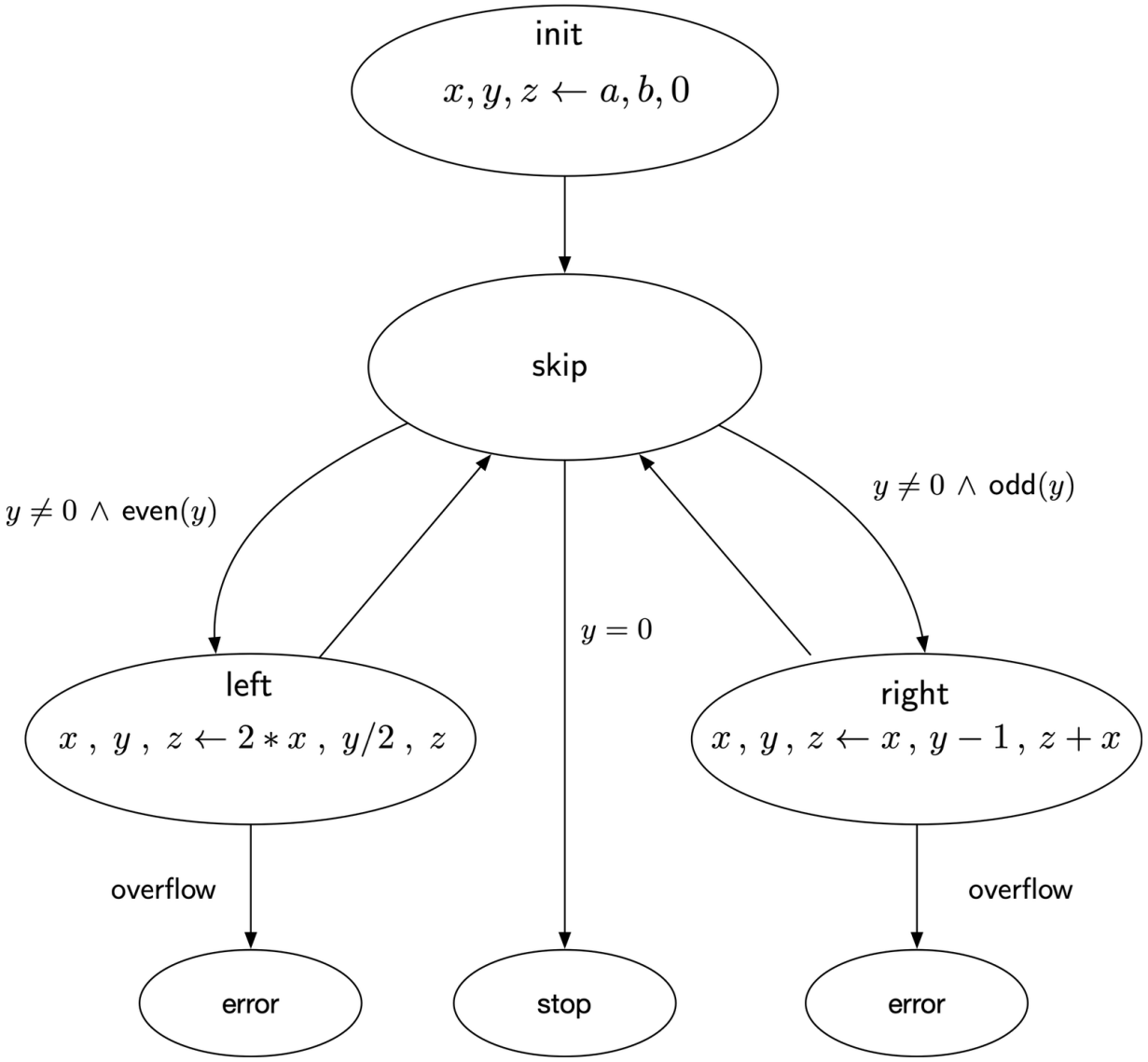
Lógica Computacional: Multiplicação de Inteiros

Um programa imperativo como ilustrado no exemplo seguinte

```
assert a >= 0 and b >= 0
x , y, z = a , b , 0
while not y == 0:
  if even(y):
      x , y , z = x << 1 , y >> 1 , z
  else:
      x , y , z = x , y - 1, z + x
# no final deve ser  z == a*b
```

pode ser descrito por um modelo do tipo *Control Flow Automaton* (CFA) como a seguir se indica



Algoritmo de Multiplicação

O programa implementa a multiplicação de dois inteiros a, b , fornecidos como "input", e com precisão limitada a n bits (fornecido como parâmetro do programa). Por outro lado o diagrama descreve a mesma funcionalidade através de um grafo orientado que é interpretado da forma seguinte:

- Os *nodos* do grafo representam **ações** que atuam sobre os "inputs" do nodo e produzem um "output" executando as operações indicadas. A cada *nodo* está associado um identificador único. Existe um nodo sem antecedentes.
- Os *ramos* do grafo representam **ligações** que transferem o "output" de um nodo para o "input" do nodo seguinte. Esta transferência é condicionada pela satisfação de uma **condição,** associada ao ramo, calculada como o valor que a ligação transfere. Adicionalmente
  - Numa ligação a ausência de condição é equivalente a `True`.
  - Se mais do que uma ligação têm origem no mesmo nodo e uma dá azo

a um situação de erro, então essa tem precedência sobre todas as outras; se não existir qualquer precedência, então uma das ligações é selecionada não deterministicamente.

O **estado** do sistema é formado por um *identificador* do nodo e pelo *valor das variáveis* no "input" do nodo.
As **transições** do sistema são determinadas pelas condições associadas aos ramos e pelas alterações dos valores das variáveis efectuadas pelas ações.

Notas

1. A terminologia neste tipo de modelo varia consoante os autores. Outras designações para "ações" são "modos" ou "locais". As ligações ("links") também se designam por "switchs" e as condições a elas associadas também se designam por "jumps".
2. O diagrama reflete explicitamente a possibilidade de o programa produzir um erro de "overflow" (tanto em `x = x << 1` como em `z = z + x`); esta possibilidade não transparece do código.

BitVec's em Z3

O funcionamento das operações BitVec's definidas em "solvers" (como o Z3) difere do que se espera da "arithmetic logic unit" ALU de um processador.

Em BitVec de tamanho n, em ambos os casos, as diversas operações aritméticas são sempre definidas módulo 2^n. Isto significa que dois quaisquer inteiros a, b (positivos ou negativos) cuja diferença seja um múltiplo de 2^n têm a mesma representação e são considerados iguais.
Isto causa algum cuidado com as relações de ordem $\leq$ e $>$ por, por exemplo, verificar-se $-1 > 2$. Por essa razão o tipo BitVec tem um segundo conjunto de operações de ordem que actuam separadamente só sobre inteiros sem sinal.

Em Z3 as operações "shift" `<< 1` e a soma aritmética `z + x` são sempre totais: nunca assinalam um resultado de erro e dão sempre um resultado no domínio pretendido — mesmo que inesperado!

Numa ALU as operações são parciais e no "output", para além dos eventuais resultados, podem activar vários bits de sinal; nomeadamente os bits de `overflow` e de `carry`. Ver aqui uma descrição do papel de ambos os bits.

Nomeadamente o `overflow` é ativado sempre que uma operação muda o sinal dos argumentos (e.g. a soma de dois números positivos dá um resultado negativo!)

Por isso uma situação onde o ALU ative o bit de `overflow` não pode ser simplesmente retirado do resultado da mesma operação em Z3. É preciso programar adequadamente a deteção desse evento.

Para ajudar a enquadrar o uso destes dados junto o seguinte texto retirado da introdução ao Z3 da Microsoft

---

Machine Arithmetic

The following example demonstrates how to create bit-vector variables and constants. The function `BitVec('x', 16)` creates a bit-vector variable in Z3 named `x` with `16` bits. For convenience, integer constants can be used to create bit-vector expressions in Z3Py. The function `BitVecVal(10, 32)` creates a bit-vector of size `32` containing the value `10`.

```
x = BitVec('x', 16)
y = BitVec('y', 16)

print (x + 2)
# Internal representation
print ((x + 2).sexpr())
# -1 is equal to 65535 for 16-bit integers
print (simplify(x + y - 1))

# Creating bit-vector constants
a = BitVecVal(-1, 16)
b = BitVecVal(65535, 16)
print (simplify(a == b))
a = BitVecVal(-1, 32)
b = BitVecVal(65535, 32)

# -1 is not equal to 65535 for 32-bit integers
```

```
print (simplify(a == b))
```

In contrast to programming languages, such as C, C++, C#, Java, there is no distinction between signed and unsigned bit-vectors as numbers. Instead, Z3 provides special signed versions of arithmetical operations where it makes a difference whether the bit-vector is treated as signed or unsigned. In Z3Py, the operators `<`, `<=`, `>`, `>=`, `/`, `%` and `>>`;correspond to the signed versions. The corresponding unsigned operators are `ULT`, `ULE`, `UGT`, `UGE`, `UDiv`, `URem` and `LShR`.

```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)
solve(x + y == 2, x > 0, y > 0)

# Bit-wise operators
# & bit-wise and
# | bit-wise or
# ~ bit-wise not
solve(x & y == ~y)
solve(x < 0)

# using unsigned version of <
solve(ULT(x, 0))
```

The operator `>>` is the arithmetic shift right, and `<<` is the shift left. The logical shift right is the operator `LShR`.

```
# Create to bit-vectors of size 32
x, y = BitVecs('x y', 32)

solve(x >> 2 == 3)
solve(x << 2 == 3)
solve(x << 2 == 24)
```