# Intermediate Code Generation

**Martin Sulzmann**

# Overview

## Purpose

- Machine independent.
- Facilitates retargeting and optimization.

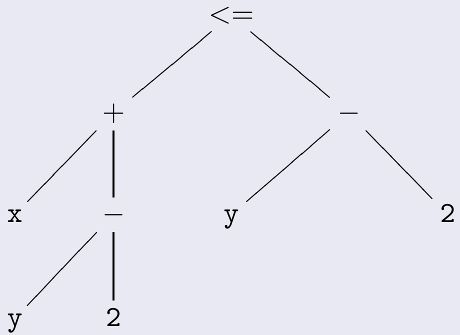
## Things to talk about

- Intermediate representation (AST vs DAG, three-address code)
- Translating
  - expressions,
  - control flow,
  - declarations, and
  - statements.

*Not specific to Mini-Go*

# AST Variants

## Example: `x+ (y-2) <= (y-2)`

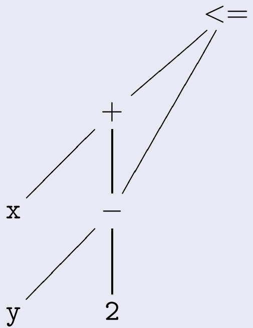


## Observation

Observe common subexpressions `y-2`.

# Directed Acyclic Graph (DAG)

**Example: `x+ (y-2) <= (y-2)`**



**Purpose**

- No repetition of patterns.
- More compact.
- Efficient compilation.

How to construct such a DAG for Mini-Go?

# Three-Address Code

## Concept

- Linearized representation of AST.
- Explicit names (addresses) for internal nodes.
- Ingredients: Labels, addresses and instructions.
    - Labels are connected to program points.
    - Addresses refer to program variables, constants and temporary variables (generated during compilation).
- 'Flat' expressions: At most one operator on the right hand side of an instruction.

# Three-Address Code Instructions

## Instructions

| | | |
|---|---|---|
| 1.) | Assignment statement | `x = y op z` |
| 2.) | Unary assignment | `x = op y` |
| 3.) | Copy statement | `x = y` |
| 4.) | Unconditional jump | `goto L` |
| 5.) | Conditional jump | `if x rel y goto L` |
| 6.) | Procedure call | |
| | - parameter setup (push) | `param x` |
| | - call name, arity | `call p,n` |
| | - return | `return y` |
| | - retrieve (pop) | `x = get` |
| 7.) | Address/pointer asg. | `x := &y` |
| | | `x:= *y` |
| | | `*x := y` |

# Translation to Three-Address Code

## Approach

- Syntax-directed where we employ semantic rules (AGs).
- To each expression $E$ attach two S-attributes:
  - $E.place$ is an address holding value of $E$,
  - $E.code$ is code to evaluate $E$.
- We will need to create
  - temporaries to hold values of internal expressions, and
  - labels for use in the generated code.

## Assumptions

- $newtemp()$ generates a new temp address,
- $newlabel()$ generates a new label, and
- $gen(x :=' y + z)$ generates the three address code.
- $nil$ = empty code (like skip).

# Translating Expressions (1)

## Syntax-Directed Translation

$$
\begin{array}{llll}
S & \rightarrow & id \ := \ E & \{S.code = E.code \| gen(id.place{=}E.place)\} \\
 & & & \\
E & \rightarrow & E_1 + E_2 & \{E.place{=}newtemp(); \\
 & & & E.code{=}E_1.code \| E_2.code \| \\
 & & & \qquad gen(E.place{=}E_1.place{+}E_2.place)\} \\
 & & & \\
E & \rightarrow & E_1 * E_2 & \{E.place{=}newtemp(); \\
 & & & E.code = E_1.code \| E_2.code \| \\
 & & & \qquad gen(E.place{=}E_1.place{*}E_2.place)\}
\end{array}
$$

where $\|$ denotes “concatenation” of code.

# Translating Expressions (2)

## Syntax-Directed Translation

$$\begin{array}{lcll}
E & \rightarrow & -E_1 & \{E.\mathit{place}{=}\mathit{newtemp}(); \\
 & & & E.\mathit{code} = E_1.\mathit{code} \| \mathit{gen}(E.\mathit{place}{=}\ {-E_1}.\mathit{place})\} \\
 & & & \\
E & \rightarrow & (E_1) & \{E.\mathit{place}{=}E_1.\mathit{place}; E.\mathit{code} = E_1.\mathit{code}\} \\
 & & & \\
E & \rightarrow & \mathit{id} & \{E.\mathit{place}{=}\mathit{id}.\mathit{place}; E.\mathit{code} = \mathit{nil}\}
\end{array}$$

# Translation of Control Flow (1)

## New Attributes

- $S.begin$ label at the beginning, and
- $S.after$ label at the end.

## Syntax-Directed Translation

$$
\begin{array}{lcll}
S & \rightarrow & \mathit{repeat}\ S_1\ \mathit{whilenot}\ E & \{S.begin = newlabel(); S.after = newlabel(); \\
 & & & S.code = gen(S.begin :) \| S_1.code \| E.code \| \\
 & & & \quad gen(\mathit{if}\ \ E.place\ \texttt{goto}\ S.begin) \| gen(S.after :)\}
\end{array}
$$

| |
|---|
| $S.begin$ : |
| $\quad S_1.code$ |
| $\quad E.code$ |
| $\quad \mathit{if}\ \ E.place\ \mathit{goto}\ S.begin$ |
| $S.after$ : |

# Translation of Control Flow (2)

### Exercise

- if-then-else
- while

# Translating Procedure Calls

## Syntax-Directed Translation

$$
\begin{array}{lcll}
S & \rightarrow & id := f(E_1, ..., E_n) & \{E_1.code \quad \| \\
 & & & ... \quad \| \\
 & & & E_n.code \quad \| \\
 & & & \texttt{param } E_1.place \quad \| \\
 & & & ... \quad \| \\
 & & & \texttt{param } E_n.place \quad \| \\
 & & & \texttt{call f, n} \quad \| \\
 & & & id.place\texttt{= get}\}
\end{array}
$$

- Call-by value semantics.
- Parameters are pushed onto call stack.
- We retrieve (get) the return value by popping the top-most value on call stack.

# Translating Assignments

## Syntax-Directed Translation

$$\begin{array}{lllll} S & \rightarrow & id := E & & p = lookup(id.name); \\ & & & & if\ p \neq nil\ then\ emit(p = E.place)\ else\ error \end{array}$$

## Things to consider

- *lookup*(*id.name*): returns storage position of *id*.
- Nested scope! Make sure we access the 'right' *id*.
- To avoid conflicts, we could introduce distinct names by renaming local variables.

# Translating Boolean Expressions

### Things to consider

- Representation of Boolean values? We simply use integers (like in C).
- Short-circuit evaluation!