CS302 Languages and Compiler Design II

Lecture 6

Intermediate Languages

Why have one?

- Simplify compilation task by dividing into stages.
- Ease porting to new source or target languages.
- Ease implementation of code transformations.

Must bridge source and object code styles.

Source code is primarily **hierarchical.**

- expressions
- structured control statements
- (perhaps) local scoping

Object code is primarily **linear.**

- explicit intermediate values
- explicit labels and jumps
- flat name space

Source Code -> ?? -> Object Code

Intermediate language is a compromise

- maintain some hierarchy for ease of generating I.L.
- linearize somewhat for ease of generating object code.
- many possibilities.

Linear machine-like code (as in Aho, et al. text).

- expressions are linearized, with intermediate results in temporaries.
- use explicit labels & jumps
- flat address space
- for PCAT project, we'll use enriched SPARC assembler code.

I.R. trees (e.g., Appel, "Modern Compiler Implementation")

- expressions remain in tree form.
- use explicit labels & jumps
- locally-scoped temporaries.

Specifying 3-address code

General form: three operands, one operator

```
X := Y op Z
```

Typical operators:

```
A := B
A := B op C
goto L
if0 A goto L
A := addr B
A := *B
```

Operands: named variables, temporaries, labels.

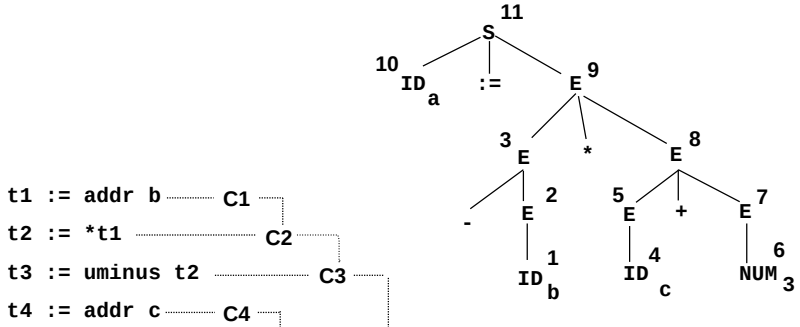
Assume an abstract instruction-generation function:

```
gen(result,operator,arg1,arg2)
```

- can produce strings, "quads", or whatever.

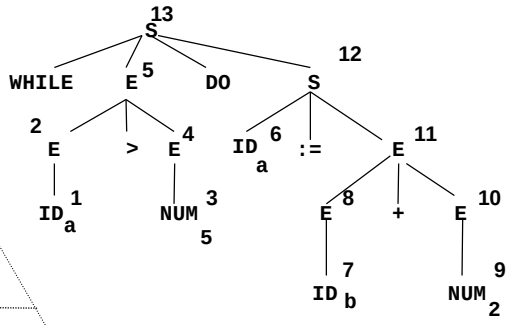
Quite ad-hoc: we'll add new instructions when we need to.

Example: 3-Addr Code for `a := -b * (c + 3)`




```
t1 := addr b ..... C1
t2 := *t1 ..... C2
t3 := uminus t2 ..... C3
t4 := addr c ..... C4
t5 := *t4 ..... C5
t6 := const 3 ..... C6 C7
t8 := t5 + t6 ..... C8
t9 := t3 * t8 ..... C9
t10 := addr a ..... C10
*t10 := t9 ..... C11
```


Example: 3-Addr Code for `while a > 5 do a := b + 2`




```
L1:
t1 := addr a ..... C1
t2 := *t1 ..... C2
t3 := const 5 ..... C3 C4

t5 := t2 > t3 ..... C5
if0 t5 goto L2 .....
t7 := addr b ..... C7
t8 := *t7 ..... C8
t9 := const 2 ..... C9 C10

t11 := t8 + t9 ..... C11
t6 := addr a ..... C6
*t6 := t11 ..... C12
goto L1 ..... C13
L2: .....
```

Syntax-directed translation of Expressions

`E := V`	E.place = newtemp(); E.code = V.code @ [gen(E.place,*,V.place,_)]
`E := N`	E.place = N.place; E.code = N.code
`E := E '+' E`	E.place = newtemp(); E.code = E1.code @ E2.code @ [gen(E.place,+,E1.place,E2.place)]
`E := '-' E`	E.place = newtemp(); E.code = E1.code @ [gen(E.place,uminus,E1.place)]
`E := E '>' E`	E.place = newtemp(); E.code = E1.code @ E2.code @ [gen(E.place,>,E1.place,E2.place)]
`V := VAR`	V.place = newtemp(); V.code = [gen(V.place,addr,VAR.var,_)]
`N := NUM`	N.place = newtemp(); N.code = [gen(N.place,const,NUM.num,_)]

Contrast Syntax-directed Evaluation of Expressions

`E := V`	E.val = *V.loc
`E := N`	E.val = N.val
`E := E '+' E`	E.val = E1.val + E2.val
`E := '-' E`	E.val = - E1.val
`E := E '>' E`	E.val = (E1.val > E2.val) ? 1 : 0
`V := VAR`	V.loc = lookup(VAR.var)
`N := NUM`	N.val = NUM.num

Syntax-directed translation of Statements

```
S := V ':=' E          S.code = E.code @ V.code @
                                [gen(*V.place,:=,E.place,_)]

S := WHILE E DO S1     S.code =
                        let begin=newlabel()
                            end = newlabel()
                        in [gen(begin,:,_,_)] @
                           E.code @
                           [gen(end,if0,E.place,_)] @
                           S1.code @
                           [gen(begin,goto,_,_),
                            gen(end,:,_,_)]

S := S1 ';' S2         S.code = S1.code @ S2.code
```

generates:

```
L1: if0 E goto L2
        S1
        goto L1
L2:
```


Contrast: Syntax-directed Evaluation of Statements

```
S := V ':=' E          s () {update(V.loc, E.val);
                            }

S := WHILE E DO S1     s () {while (E.val <> 0)
                                  S1();
                            }

S := S1 ';' S2         s () {s1();
                             s2();
                            }
```