Compilers

J.M. Dana
Jose.Dana@cern.ch

DANGER!

Theoretical content approaching!

Based on last year presentation by Lori Pollock

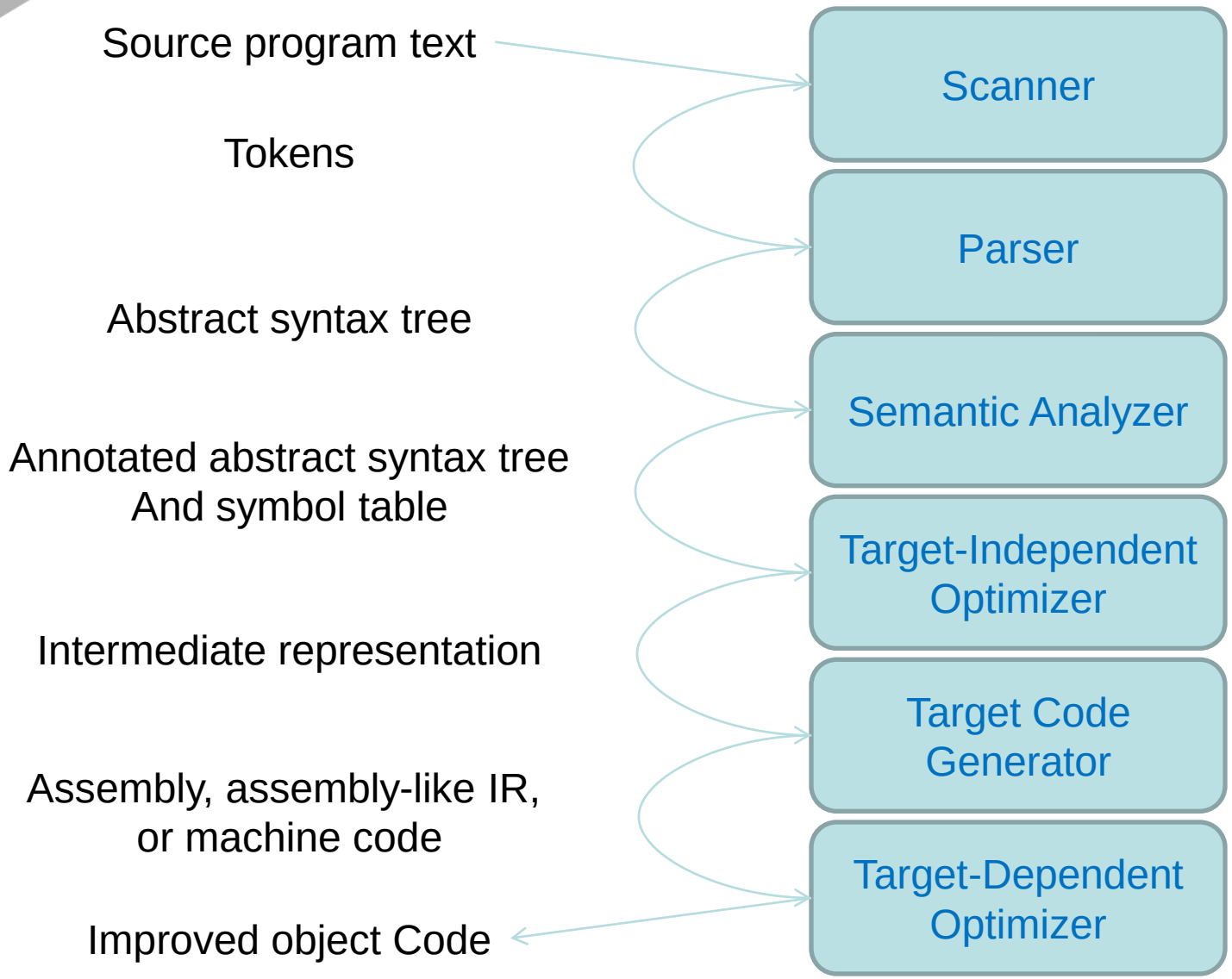
Why are compilers important?

- Why should I care about compilers? The compiler is just a tool…
  - The compiler is NOT just a tool
  - It will “explain” to the computer what you are trying to do…
- Knowledge of the compilation process can help programmers write better code
- Good to know what the compiler can do for you (and what it can’t)

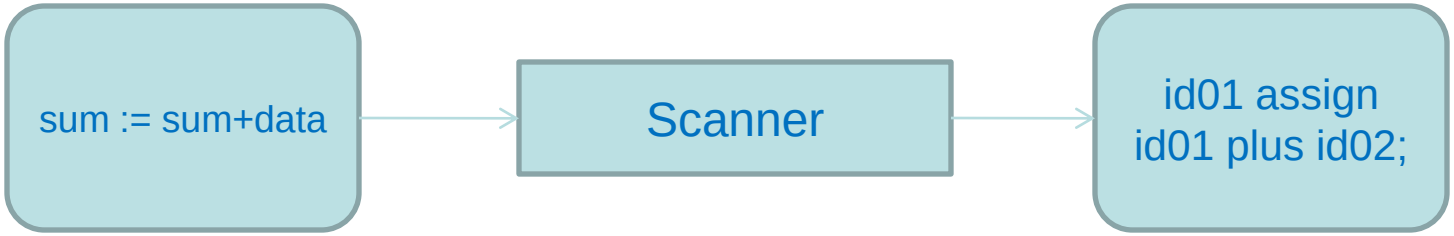
```
const vs. #define
```

What is a compiler?

Basic functional overview

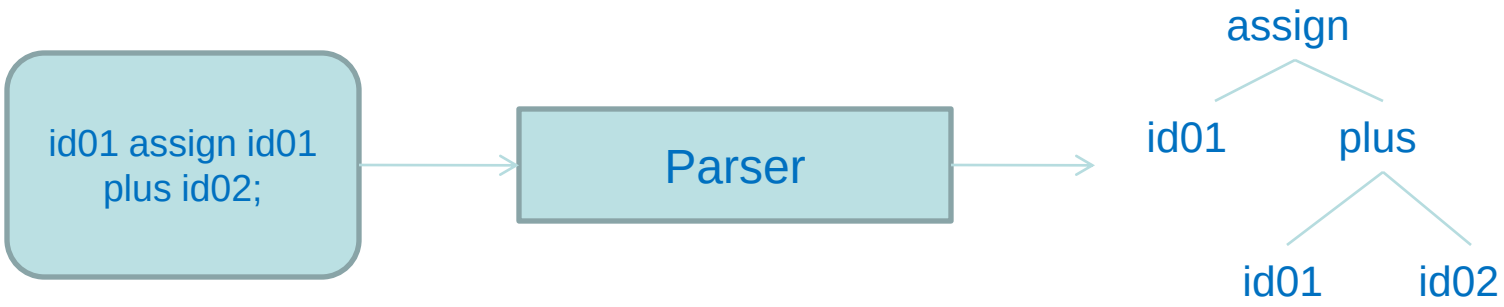


Scanning or Lexical Analysis



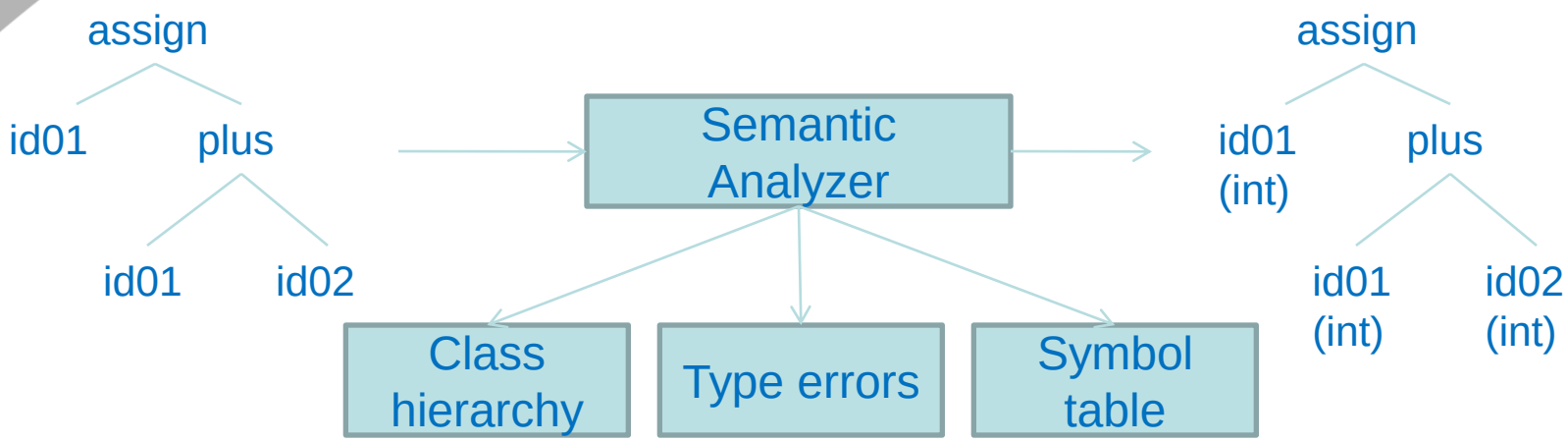
- Partition of the text into tokens (smallest meaningful unit)
- Remove comments, white spaces, etc.
- Track line numbers
- The scanner is basically a recognizer of a regular language

Parsing or Syntactic Analysis



- Build Abstract Syntax Tree (AST)
- The parser is a recognizer of a context-free language

Semantic Analysis



- Symbol Table creation (debugging)
- Class inheritance hierarchy
- Type checking
- Static semantic checking (def before use)

Target-independent optimization

- Local
  - Remove dead code (result never used)
  - Remove redundant expression evaluations
  - Propagate and evaluate constants
- Global
  - Data flow analysis over a control flow graph
  - Code transformation if safe and profitable
- Interprocedural
  - Interprocedural analysis over a call graph
  - Interprocedural constant propagation
  - Inlining and global optimization

Examples local optimization (1)

- Elimination of redundant loads and stores

```
r2 ← r1 + 5
i ← r2
r3 ← i
r3 ← r3 x 3
```

```
r2 ← r1 + 5
i ← r2
r3 ← r2 x 3
```

- Constant folding

```
r2 ← 3 x 2
```

```
r2 ← 6
```

Examples local optimization (2)

- Constant propagation

```
r2 ← 4
r3 ← r1 + r2
r2 ← …
```

```
r3 ← r1 + 4
r2 ← …
```

- Common subexpression elimination

```
r2 ← r1 x 5
r2 ← r2 + r3
r3 ← r1 x 5
```

```
r4 ← r1 x 5
r2 ← r4 + r3
r3 ← r4
```

Interprocedural optimization

```
method B() {call A(b,10)}
method C() {call A(c,10)}
```

```
method A(int x, int y) {
    …
    z=y+5
    if(z<10) {
        return  0
    } else {
        return 1
    }
}
```

```
method A(int x, int y){
     return 1
 }
```

What affects the optimizer's ability?

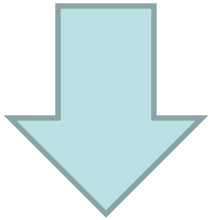
- Pointers: lack of knowledge of which location is being referenced
- Calling functions through function pointers
- Aliasing
- Polymorphism
- Branching
- …

Optimizing for cache (1)

- Goals:
  - Optimize for spatial locality: prefetching of data in same cache line
  - Optimize for temporal locality: Re-use data which has been brought into cache as much as possible
- The programmer can help:
  - Good use of data structures
  - Memory alignment when neccesary

Optimizing for cache (2)

```
for(i=0;i<N;i++) {
    imageA[i]=loadimg(fileA[i]);
    imageB[i]=loadimg(fileB[i]);
    a1[i]=funcA(imageA[i]);
    a2[i]=funcB(imageA[i]);
    b1[i]=funcA(imageB[i]);
    b2[i]=funcB(imageB[i]);
}
```



```
for(i=0;i<N;i++) {
    imageA[i]=loadimg(fileA[i]);
    a1[i]=funcA(imageA[i]);
    a2[i]=funcB(imageA[i]);
    imageB[i]=loadimg(fileB[i]);
    b1[i]=funcA(imageB[i]);
    b2[i]=funcB(imageB[i]);
}
```

Avoid RAW dependencies

```
a=b+c
d=b+a
e=b+d
```

```
a=b+c
d=2*b+c
e=3*b+c
```

Loop unrolling

```
for(i=0;i<100;i++) {
    a[i]=a[i]+b[i];
    c[i]=a[i]*2;
}
```

```
for(i=0;i<99;i+=2) {
    a[i]=a[i]+b[i];
    c[i]=a[i]*2;
    a[i+1]=a[i+1]+b[i+1];
    c[i+1]=a[i+1]*2;
}
```

Function inlining

```
inline int max (int a, int b) {
    if (a > b)
        return a;
    else
        return b;
}

a=max(x,y);
```

```
a = (x > y ? x : y);
```

Memory alignment

```
matrix=(unsigned char**)malloc(height*sizeof(unsigned char*));

for(i=0;i<height;i++)
    matrix[i]=(unsigned char*)malloc(width*sizeof(unsigned char));
```

```
matrix=(unsigned char**)malloc(height*sizeof(unsigned char*));
matrix[0]=(unsigned char*)malloc(height*width*sizeof(unsigned char));

for(i=1;i<height;i++)
    matrix[i]=matrix[i-1]+width;
```

Data structure alignment

```
struct MixedData {
    char Data1;
    short Data2;
    int Data3;
    char Data4;
};
```

```
struct MixedData {
    char Data1;
    char Padding0[1];
    short Data2;
    int Data3;
    char Data4;
    char Padding1[3];
};
```

Target-dependent optimization

- SIMD instructions (MMX, SSE, SSE2, etc.)
- 32 vs. 64 bits
- Specific registers
- Predicated operations
- In-order vs. out-of-order executions
- … and any architecture specific property

Profile-guided optimization

- Compilers are far from being perfect
- PGO analyzes your software and chooses the “best” optimization techniques for it
- gcc (from 4.1)
  - -fprofile-generate + execution + -fprofile-use
- icc
  - -prof-gen + execution + -prof-use

Q & A