
 **CSE 582 – Compilers**


Intermediate Representations
Hal Perkins
Autumn 2002

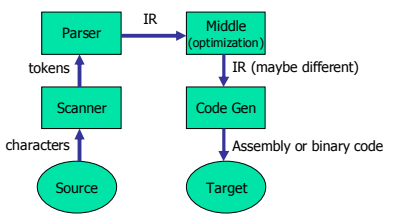
10/22/2002 © 2002 Hal Perkins & UW CSE G-1

 **Agenda**

- Parser Semantic Actions
- Intermediate Representations
 - Abstract Syntax Trees (ASTs)
 - Linear Representations
 - & more


10/22/2002 © 2002 Hal Perkins & UW CSE G-2

 **Compiler Structure (review)**



```
graph TD; Source((Source)) -- characters --> Scanner[Scanner]; Scanner -- tokens --> Parser[Parser]; Parser -- IR --> Middle["Middle (optimization)"]; Middle -- "IR (maybe different)" --> CodeGen[Code Gen]; CodeGen -- "Assembly or binary code" --> Target((Target));
```


10/22/2002 © 2002 Hal Perkins & UW CSE G-3



What's a Parser to Do?

- n Idea: at significant points in the parse perform a *semantic action*
 - n Typically when a production is reduced (LR) or at a convenient point in the parse (LL)
- n Typical semantic actions
 - n Build (and return) a representation of the parsed chunk of the input (compiler)
 - n Perform some sort of computation and return result (interpreter)


10/22/2002 © 2002 Hal Perkins & UW CSE G-4



Intermediate Representations

- n In most compilers, the parser builds an *intermediate representation* of the program
- n Rest of the compiler transforms the IR for efficiency and eventually translates it to final code
 - n May transform initial IR to a different IR at some point

10/22/2002 © 2002 Hal Perkins & UW CSE G-5



IR Design

- n Decisions affect speed and efficiency of the rest of the compiler
- n Desirable properties
 - n Easy to generate
 - n Easy to manipulate
 - n Expressive
 - n Appropriate level of abstraction
- n Different tradeoffs depending on compiler goals
 - n Not necessarily the same in different parts of the same compiler

10/22/2002 © 2002 Hal Perkins & UW CSE G-6

Types of IRs

- Three major categories
 - Structural
 - Linear
 - Hybrid
- Some basic examples now; more when we get to later phases of the compiler

10/22/2002 © 2002 Hal Perkins & UW CSE G-7

Levels of Abstraction

- Key design decision: how much detail to expose
 - Affects possibility and profitability of various optimizations
 - Structural IRs are typically fairly high-level
 - Linear IRs are typically low-level
 - But this isn't necessarily true

10/22/2002 © 2002 Hal Perkins & UW CSE G-8

Example: Array Reference

A[i,j]

```

loadI 1 => r1
sub rj,r1 => r2
loadI 10 => r3
mult r2,r3 => r4
sub ri,r1 => r5
add r4,r5 => r6
loadI @A => r7
add r7,r6 => r8
load r8 => r9

```

10/22/2002 © 2002 Hal Perkins & UW CSE G-9



Structural IRs

- n Typically reflect source (or other higher-level) language structure
- n Tend to be large
- n Examples: abstract syntax trees (ASTs), DAGs
- n Particularly useful for source-to-source transformations

10/22/2002

© 2002 Hal Perkins & UW CSE

G-10



Concrete Syntax Trees

- n Full grammar needed to guide parser, but contains many extraneous details
 - n Chain productions
 - n Rules that control precedence and associativity

10/22/2002

© 2002 Hal Perkins & UW CSE

G-11




Syntax Tree Example

- n Concrete syntax for $x=2*(n+m);$

10/22/2002

© 2002 Hal Perkins & UW CSE


G-12



Abstract Syntax Trees

- n Want only essential structural information
 - n Omit extraneous junk
- n Can be represented explicitly as a tree or in a linear form
 - n Example: LISP/Scheme S-expressions are essentially ASTs


10/22/2002 © 2002 Hal Perkins & UW CSE G-13



AST Example

- n AST for $x=2*(n+m);$

10/22/2002 © 2002 Hal Perkins & UW CSE G-14



Linear IRs

- n Pseudo-code for an abstract machine
- n Level of abstraction varies
- n Simple, compact data structures
- n Examples: stack machine code, three-address code

10/22/2002 © 2002 Hal Perkins & UW CSE G-15

Stack Machine Code

- Originally used for stack-based computers (famous example: B5000)
- Now used for Java, C# (MSIL)
- Advantages
 - Compact; mostly 0-address opcodes
 - Easy to generate
 - Simple to translate to naïve machine code
 - But need to do better in production compilers

10/22/2002 © 2002 Hal Perkins & UW CSE G-16

Stack Code Example

- Hypothetical code for $x=2*(n+m)$;

```
pushaddr x
pushconst 2
pushval n
pushval m
add
mult
store
```


10/22/2002 © 2002 Hal Perkins & UW CSE G-17

Three-Address code

- Many different representations
- General form: $x \leftarrow y(op)z$
 - One operator
 - Maximum of three names
- Example: $x=2*(n+m)$; becomes

```
t1 ← n + m
t2 ← 2 * t1
x ← t2
```


10/22/2002 © 2002 Hal Perkins & UW CSE G-18



Three Address Code (cont)

- Advantages
 - Resembles code for actual machines
 - Explicitly names intermediate results
 - Compact
 - Often easy to rearrange
- Various representations
 - Quadruples, triples, SSA
 - Much more later...


10/22/2002 © 2002 Hal Perkins & UW CSE G-19



Hybrid IRs

- Combination of structural and linear
- Level of abstraction varies
- Example: control-flow graph

10/22/2002 © 2002 Hal Perkins & UW CSE G-20



What to Use?

- Common choice: all(!)
 - AST or other structural representation built by parser and used in early stages of the compiler
 - Closer to source code
 - Good for semantic analysis
 - Facilitates some higher-level optimizations
 - Flatten to linear IR for later stages of compiler
 - Closer to machine code
 - Exposes machine-related optimizations
 - Hybrid forms in optimization phases

10/22/2002 © 2002 Hal Perkins & UW CSE G-21



Coming Attractions

- n Representing ASTs
- n Working with ASTs
 - n Where do the algorithms go?
 - n Is it really object-oriented?
 - n Visitor pattern
- n Then: semantic analysis, type checking, and symbol tables

10/22/2002

© 2002 Hal Perkins & UW CSE

G-22
