Preface, Disclaimer, and all that...

These lecture notes are **always** under construction, and will be stable **only** in the limit.

They are a snapshot of what I think the course **should** be like at any given moment, not what it **was** like when you attended a particular lecture.

Certain slides, lectures, or chapters may even be **missing** or **outdated** if I did not have the time to transcribe them. In those cases, I sincerely hope you took notes... :-)

I am interested in improving these lecture notes, so please email me about any errors, omissions, or untoward words you find.

Table of Contents

CS 152: Compiler Design

Welcome!

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

September 24, 2004

Today's Lecture

The Three Questions. . .

What is this course about?

Study compilers (and interpreters) to

- finally understand how the tools you use all the time work
- learn a bunch of generally useful programming techniques
- better appreciate the costs of certain language features
- hone your hacking skills, hopefully quite a bit... :-)

Learn in three ways:

- Lecture: theoretical background, algorithms and data structures, software engineering considerations
- Lab: additional material, often more applied; discussion of compiler implementation
- Assignments: hack your own compiler and interpreter for a small imperative language

Does this course work?

I am the **worst** person to ask, but some apparently think so:

- "... your course in compilers is the most educational and enjoyable course I've ever taken at UCR. I really enjoyed the hands-on approach you gave us ... I have a sufficient background ... to continue doing work on compilers, which is more that I can say for any other course I've taken at UCR."
  — "A" student, Winter 2004
- "Total ~~nutcase~~ when it comes to work load, but still definitely the best CS teacher at UCR by far. Sure he gives you alot of work, but you actually learn something about the subject at hand when you take one of his classes, but lazy students beware."
  — Anonymous, Spring 2004

You can find negative opinions on your own, just ask around...

How much work is this course?

Rule of thumb:

- 1 unit ≈ 3–4 hours / week
- 4 units ≈ **12–16** hours / week

Hours break down as follows:

- Lectures: 3 hours / week
- Labs: 3 hours / week
- Assignments: 6–10 hours / week

Some hints (for less work):

- You can choose C++ or Python to write your compiler.
- Experience shows that Python reduces workload by ≈ 50%.
- You can share test cases (but **not** code) on the mailing list.
- Old exams are online, and I like my questions, so. . . :-)

Administrivia

People

Instructor: Peter H. Fröhlich `http://www.cs.ucr.edu/~phf/`

- PhD, UC Irvine: component-oriented programming languages
- Research interests: programming languages, software engineering, compilers and interpreters, networking (peer-to-peer), social implications (patents, privacy)
- Teaching experience: algorithms and data structures, software construction, software engineering, compilers and interpreters, programming languages

Assistant: Mikiko Matsunaga `http://www.cs.ucr.edu/~matsunam/`

- Interests: Graphics, was a grader for **this** course before

Assistant: Vi Pham `http://www.cs.ucr.edu/~vpham/`

- Interests: Databases, took a similar course at Pomona before

Style

Feel free to interrupt and ask questions

- right when you don't get something is the best moment to ask
- I am more than happy to do more/different examples, just ask

I am a person, not a teaching machine

- I make mistakes, please point them out when you notice any
- I have opinions, and sometimes they come out in a rant
- feel free to disagree, "free speech" applies to you as well

I like democracy, the more direct the better

- feel free to start/sign petitions for whatever you want changed
- I can't guarantee that I'll accept everything, but give it a shot

Applies to lecture! Ask your TA how he or she likes to run labs...

Evaluation

Final grade determined on the “usual” 60/70/80/90 scale

- possibly with some slight “fudging” at the end

4 Exams (in class, 40% of grade, graded by instructor)

- two quizzes (10% = 5% each)
- midterm (15%, builds on quiz)
- comprehensive (15%, builds on quizzes, midterm)
- **no final exam**

8 Assignments (home & lab, 60% of grade, graded by assistants)

- mostly hacking of course, namely your own compiler
- also a short weekly log for tracking your progress
- final assignment involves writing a short reaction paper

Miscellaneous

Read the **policies** `http://www.cs.ucr.edu/~phf/`

- cheating will be taken seriously, so please don't even try
- no late homework, no make-up exams, no extra-credit stuff

Sign up for the **mailing list** `cs152@lists.cs.ucr.edu`

- important announcements (changes, extensions, etc.)
- ongoing discussion of compiler implementation

Attend lab **every** week

- material from lab will be on exams, you have been warned
- lab is your chance to discuss the compiler with others, use it
- your TA will help you in any way he or she can, just ask

Important Warning

We interrupt this lecture. . .

You must have good programming skills to succeed!

If you do **not** have those skills

- you will probably spend a **lot** more time on assignments
- you should attend **all** labs and ask lots of questions
- you should attend office hours **frequently** and do the same

Also, you might want to consider

- taking CS 100 (or CS 180) before taking CS 152
- taking only less “intense” courses while in CS 152

Literature

Compilers

- Niklaus Wirth:
  Compiler Construction.
  Addison-Wesley, 1996.
- Andrew Appel, Jens Palsberg:
  Modern Compiler Implementation in Java (or ML or C).
  Cambridge University Press, 2003 (and earlier).
- Keith Cooper, Linda Torczon:
  Engineering A Compiler.
  Morgan Kaufmann, 2004.

- Wirth's text is close "in spirit" to this course, recommended.
  - Best of all, it's available as a PDF file on the course website.
- Appel's and Cooper's texts are **way** more comprehensive...
  - Recommended if you **still** like compilers after the course. :-)

Programming

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides:
  Design Patterns.
  Addison-Wesley, 1995.
- Andrew Hunt, David Thomas:
  The Pragmatic Programmer.
  Addison-Wesley, 1999.
- Brian Kernighan, Rob Pike:
  The Practice of Programming.
  Addison-Wesley, 1999.

- Gamma discusses lots of useful object-oriented techniques.
  - Not just for compilers, it's a classic you **really** should read.
- Hunt and Kernighan discuss software development techniques.
  - Not just for compilers, but you'll hack a lot in this course.

Compilers and Interpreters

What is a Compiler?

A compiler is a machine that **translates** sentences

- from a **source** language (e.g. C++, Pascal, ML)
- to a **target** language (e.g. C, MIPS, x86)

Usually translation from **higher** to **lower** level

- e.g. C++ (classes) ⇒ C (structs) ⇒ x86 (addresses)
- e.g. ML (functions) ⇒ MIPS (addresses, jumps)

However, there are also "decompilers" that go the other way

- e.g. Java byte code ⇒ Java source code
- e.g. MIPS object file ⇒ MIPS assembly source

What is an Interpreter?

An interpreter is a machine that **executes** sentences

- either a "real" hardware machine (e.g. MIPS, x86 processors)
- or a "virtual" software machine (e.g. SPIM, Java VM)

The sentence being executed is usually called a **program**

- a running program reads input and writes output Doh!
- in particular, the program could be a compiler :-)

Essence of Computer Science: sentence = program = machine

- it's all the same to us, which is a curse and a blessing
- actually the genetics guys are taking it a step further. . .

Qualities

Compilers and interpreters should be...

1. Correct: the **meaning** of sentences must be preserved
   - given `a[7] := 10` a compiler can't spit out `a[6] := 1`
   - given `a[7] := 10` an interpreter can't format your drive
2. Robust: wrong input is the **common** case
   - compilers and interpreters can't just crash on wrong input
   - they need to diagnose all kinds of errors safely and reliably
3. Efficient: resource usage should be **minimal** in two ways
   - the process of compilation or interpretation itself is efficient
   - the generated code is efficient when interpreted (for compilers)
4. Usable: integrate with environment, accurate feedback
   - work well with other tools (editors, linkers, debuggers, ...)
   - descriptive error messages, relating accurately to source

A Little History

The Big Bang

Compilers and Interpreters: The Big Picture

- used to be (pretty) hard, but is (relatively) easy today
- shaped by computer architecture and programming languages

First (widely adopted) compiler: FORTRAN (1957) Backus

- 18 staff-years, a “huge” project for the time
- arithmetic expressions, arrays, non-recursive functions, static typing

First (widely adopted) interpreter: LISP (1959) McCarthy

- about 4 staff-years, the `EVAL` “accident”
- symbolic expressions, lists, recursive and higher-order functions, dynamic typing

Imperative Highlights

Algol (1960) Naur

- first "committee" language, Backus and McCarthy influential
- concise language definition, Backus-Naur form (BNF)
- block structure (static scoping), recursion

Pascal (1970) Wirth

- Wirth's "protest" against the Algol-68 monster (in part)
- modern type concept, lots of improvements on Algol
- recursive descent parsing, virtual "p-code" machine

Recent History: C++, C#, Java, ML, Oberon, . . . "thousands"

- modules, classes and objects, components, . . .
- exceptions, templates, overloading, . . .
- concurrent, parallel, distributed, . . .

History References

R. L. Wexelblat (Ed.):
History of Programming Languages.
Academic Press, 1978.

T. J. Bergin, R. G. Gibson (Eds.):
History of Programming Languages.
ACM Press, 1996.

- If you are a history buff like me, these two are essential reading
  - Volume 1: Algol, COBOL, FORTRAN, LISP, Simula, ...
  - Volume 2: Ada, C, CLU, Pascal, Prolog, Smalltalk, ...
- Two cool websites on the history of programming languages
  - `http://hopl.murdoch.edu.au/`
  - `http://www.levenez.com/lang/`

CS 152: Compiler Design

Compiler Architecture

Department of Computer Science & Engineering
University of California, Riverside

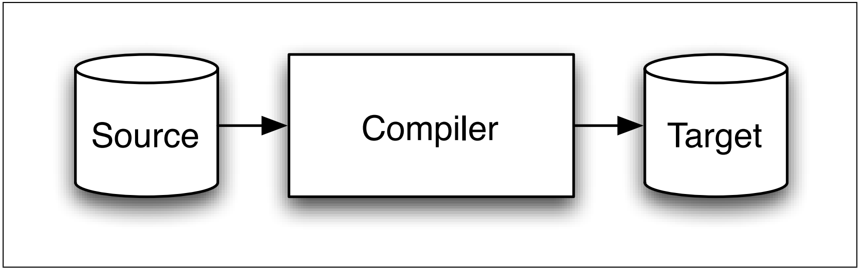
Peter H. Fröhlich
phf@cs.ucr.edu

September 27, 2004

Today's Lecture

Introduction

Why Functional Decomposition?



Compiler translates source to target language:

- `i := i + 1` ⇒ `add #1, -24(a5)`

Simple enough, right? But consider this:

- `i := j + 1` ⇒

```
move -28(a5), d0
add #1, d0
move d0, -24(a5)
```

Why Functional Decomposition?

Two **very** similar constructs yield **very** different code!

- for `i := i + 1` we can
  - load `i`, add `1`, and store `i` directly
- for `i := j + 1` we must
  - load `j` into the register `d0`
  - add `1` to the register `d0`
  - store the register `d0` to `i`

Implementing the translation is **complex**!

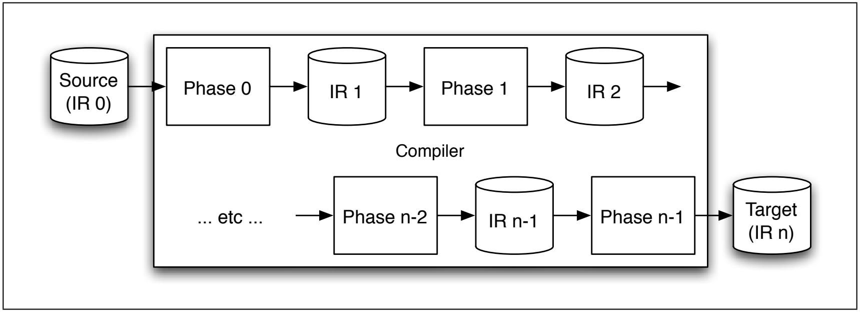
- real source languages have **many** different constructs
- real target languages have **many** different instructions

Decompose into **subtasks**, don't try to do everything at once!

- each task is comparatively simple, probably more reliable
- each task can be tested independently, maybe even reused

Architecture

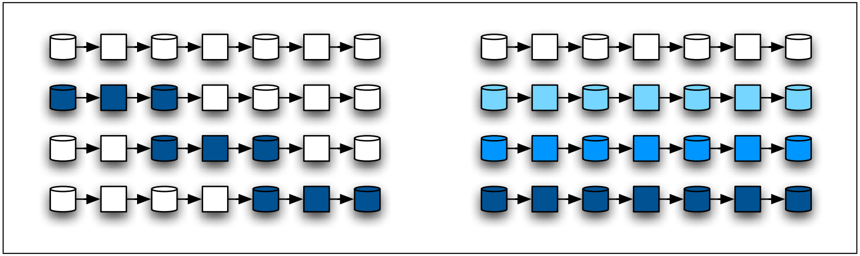
Static Architecture: What happens where?



The standard **pipeline** architecture:

- a “series” of tasks, usually called **phases** for compilers
  - replace one “big” compiler by many “small” ones
- connected by **intermediate representation** (IR) of sentence
  - appropriate data structures for the task at hand

Dynamic Architecture: What happens when?



Multi **pass** (left) versus single **pass** (right):

- Multi: each phase finishes before the next phase starts
  - **multi** pass compiler ⇔ **one** phase active concurrently
- Single: all phases interleaved until translation finishes
  - **single** pass compiler ⇔ **all** phases active concurrently
- Of course there are also compilers that mix the two...

Remarks on Architecture

Note the difference between **phase** and **pass**:

- **phase** is a static notion, a certain unit of functionality
- **pass** is a dynamic notion, a certain pattern of execution

Lots of **variations** on the "basic" pipeline are possible:

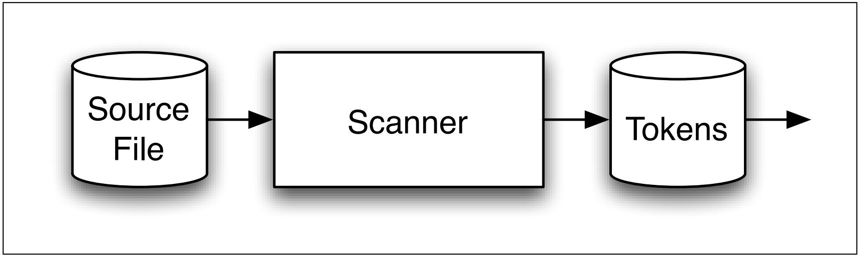
- two or more intermediate representations between phases
- hierarchical decomposition: split a phase into it's own pipeline

Intermediate representations can be on **disk** or in **heap**:

- typically in heap for single pass compilers
  - faster on average, but require more heap memory
  - if heap runs out, swapping can easily ruin that gain. . .
- typically on disk for multi pass compilers
  - slower on average, but require less heap memory
  - historically, the compiler itself would often not fit. . .

Typical Phases

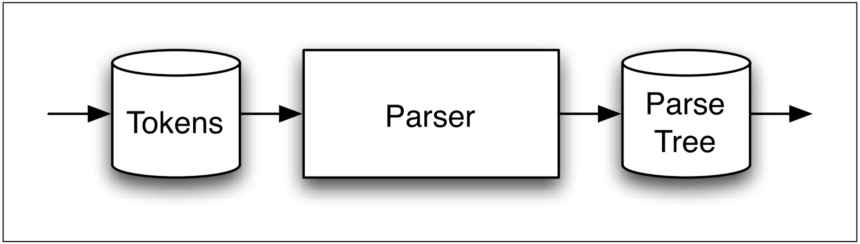
Lexical Analysis



Translate from **individual** characters into **larger** tokens

- $\boxed{\texttt{n}}\boxed{\texttt{2}}\sqcup\boxed{\texttt{:}}\boxed{\texttt{=}}\sqcup\boxed{\texttt{c}}\boxed{\texttt{n}}\boxed{\texttt{t}}\sqcup\boxed{\texttt{+}}\sqcup\boxed{\texttt{1}}\boxed{\texttt{2}} \Rightarrow \boxed{\text{n2}}\boxed{:=}\boxed{\text{cnt}}\boxed{+}\boxed{12}$
- distinguishes keywords, operators, identifiers, numbers, ...
- filters comments, whitespace, ... and tracks positions

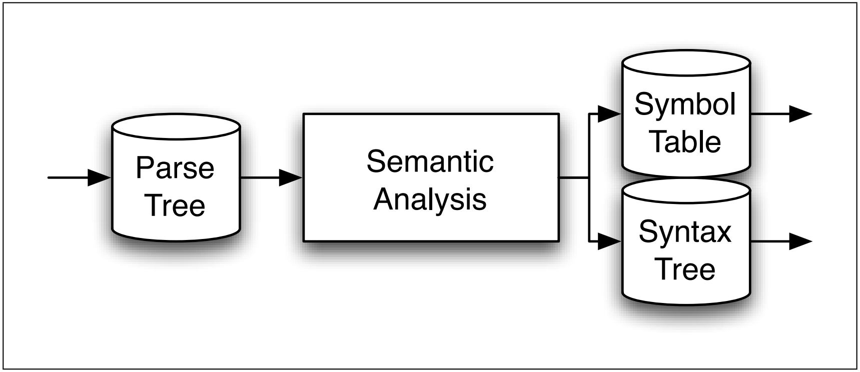
Syntactic Analysis



Translate from **linear** tokens to **hierarchical** trees

- $\boxed{\text{n2}}\boxed{:=}\boxed{\text{cnt}}\boxed{+}\boxed{12}\boxed{-}\boxed{\text{b}} \Rightarrow \boxed{\boxed{\text{n2}}:=\boxed{\boxed{\boxed{\text{cnt}}+\boxed{12}}-\boxed{\text{b}}}}$
- distinguish **precedence** and **associativity** of operators
- recognize **more** structure that is not explicit in the source

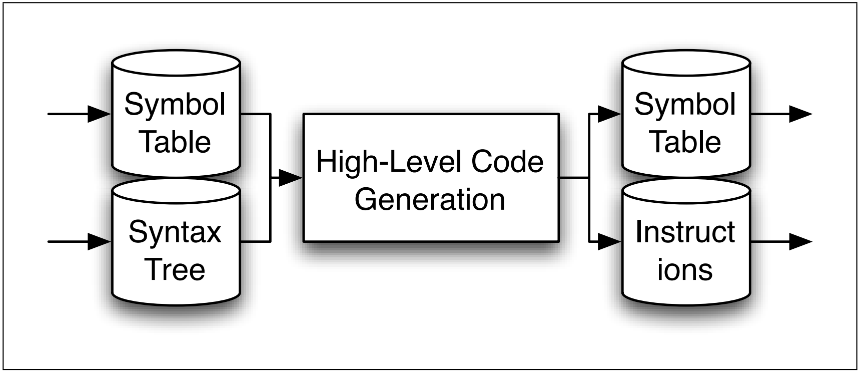
Semantic Analysis



A **fuzzy** phase with a variety of tasks. . .

- collect information on **declarations** in symbol table
- simplify the parse tree and connect **uses** with **declarations**
- perform **type inference** and **type checking**

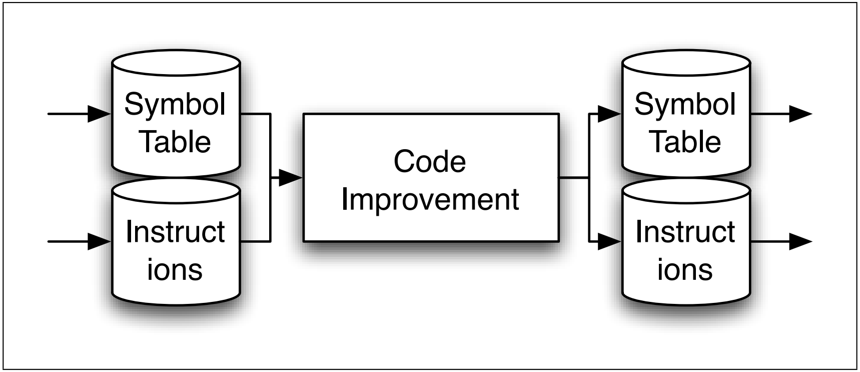
Code Generation (High-Level or Intermediate)



Translate from **checked** IR into **virtual** instructions

- some could be **real** instructions already, depends on context
- probably no **registers** yet, generic “placeholders” instead

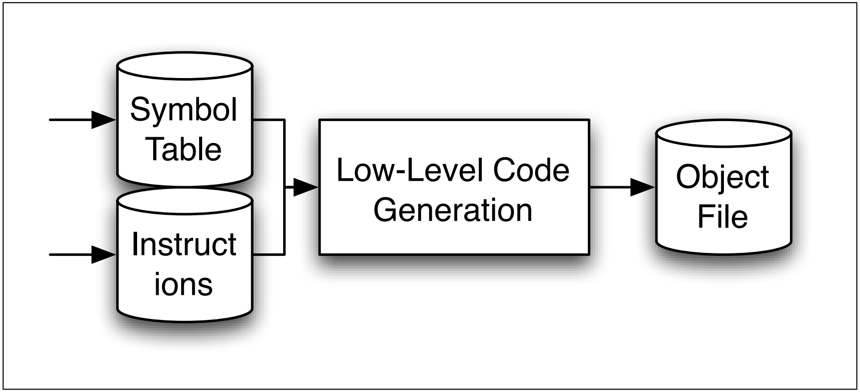
Code Improvement (Optimization)



Translate from **instructions** into **instructions**

- but **improve** instruction sequence according to certain criteria
- often **speed** as criterion, sometimes **size** or **energy** though

Code Generation (Low-Level or Object)



Translate from **virtual** instructions into **real** instructions

- typically includes register allocation and instruction scheduling
- also determines memory layout, object format depends on OS

Remarks on Phases

Again, lots of **variations** are possible:

- some compilers do not build a parse tree explicitly
- some compilers perform semantic analysis while parsing
- some compilers do not include any code improvement phase

The concrete architecture is **engineered** as needed:

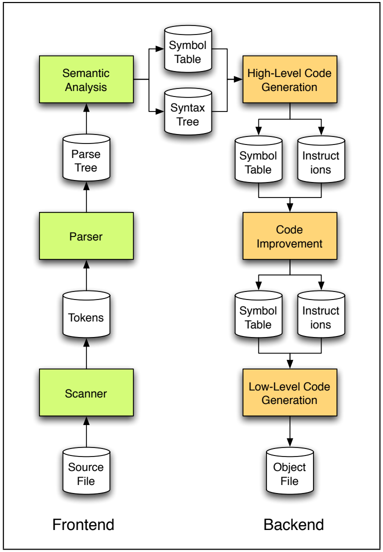
- "throwaway prototype" versus "GNU Compiler Collection"
  - simplicity of implementation helps getting things done quickly
  - however, it limits how well you can cope with evolution later
- capabilities of the language the compiler itself is written in
  - certain languages invite you to take shortcuts more often
  - others encourage (and enforce) modular architectural design

Sometimes non-technical concerns play a role:

- Got 5 developers? Don't be surprised if their compiler has 5 phases as well... :-)

Frontend and Backend

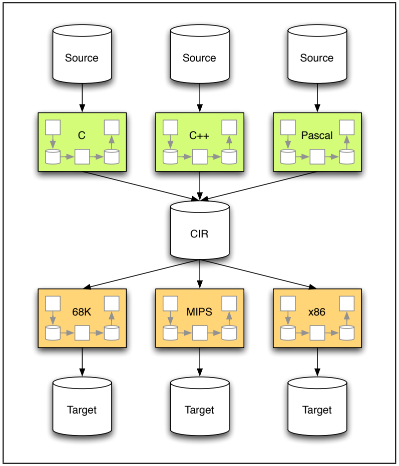
Analysis versus Synthesis



Phases fall in two major areas:

- Analysis (green-ish)
  - "understand" the source sentence
  - mostly dependent on source language
  - also known as "frontend" of compiler
- Synthesis (orange-ish)
  - "formulate" the target sentence
  - mostly dependent on target language
  - also known as "backend" of compiler

Extensible Compilers

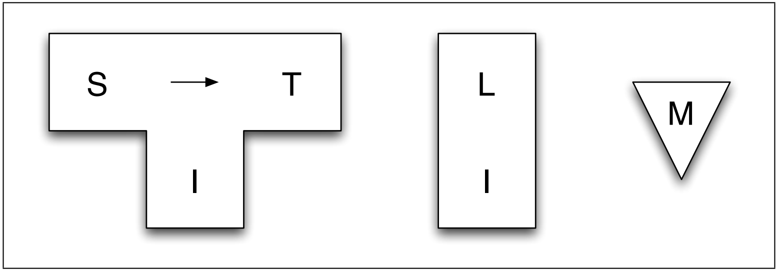


Utilize a **common** intermediate representation

- languages must be reasonably similar, otherwise IR too big
  - UNCOL: Universal Computer-Oriented Language (1958)
- write n frontends $+$ m backends to get $n \times m$ compilers
  - scales better, more flexible, but also more complicated

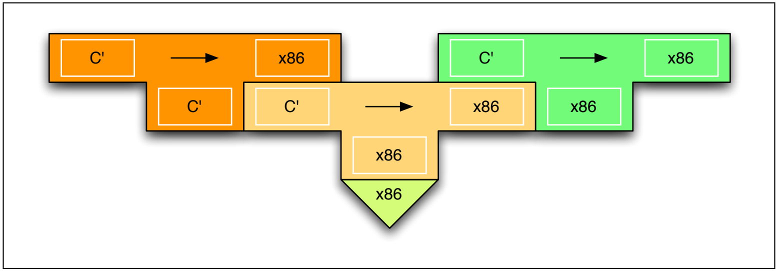
Bootstrapping and Porting

Graphical Notation (T-Diagrams)



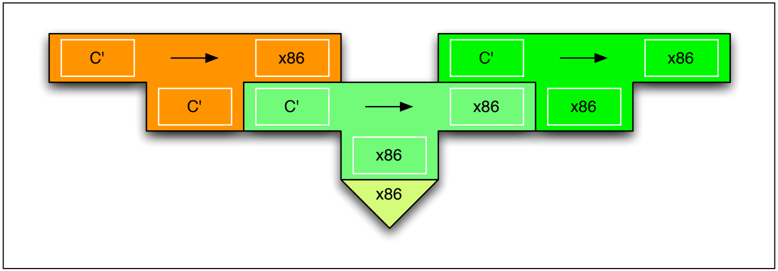
- Compiler
  - translating from source language S to target language T
  - written in implementation language I
- Interpreter
  - executing programs written in language L
  - written in implementation language I
- Machine
  - executing programs written in machine language M
  - implemented in hardware, bottom of the game. . .

Bootstrap: C Compiler (Part I)



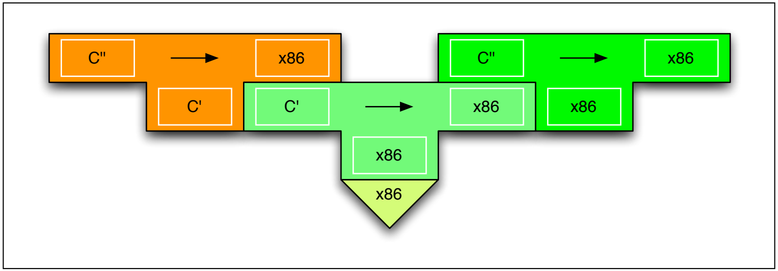
1. Write a compiler for a **small subset** C' of C
    - written in x86 assembly, "by hand" of course
2. Write (again) a compiler for **small subset** C' of C
    - written in the C' subset our bootstrap compiler understands
3. Compile new compiler with existing bootstrap compiler
    - Now we have a C' compiler written in C' and executable on x86

Bootstrap: C Compiler (Part 2)



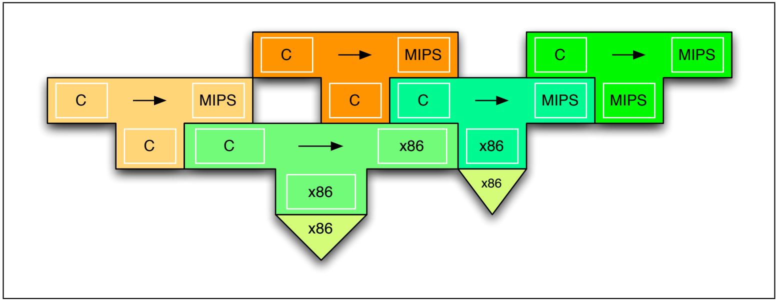
1. Recompile the C' compiler with **itself**
   - if not an identical copy, there is at least one bug somewhere
   - fix bugs and repeat until identical copy is obtained...
2. Now we can **get rid of** the bootstrap compiler
   - in principle anyway, of course we keep it around "just in case"

Bootstrap: C Compiler (Parts 3, 4, 5, . . .)



1. Extend existing compiler to a **bigger** subset C” of C
   - written in C’ still, can’t use C” extensions yet
2. Compiling yields an executable compiler from C” to x86
   - recompile again, until all bugs are gone and copies are identical
   - now we can use C” features to implement the next version
3. Keep extending and recompiling as appropriate until **full** C
   - And if you don’t go crazy, you get a nice compiler. . .

Porting: From x86 to MIPS



1. Replace x86 backend with MIPS backend in existing compiler
   - another reason to use frontend/backend architecture
2. Compiling yields compiler from C to MIPS running on x86
   - **cross compiler**: runs on A but translates to B
3. Recompile **identical** source with new compiler
   - yields a compiler from C to MIPS running on MIPS
   - transfer to MIPS machine and debug as usual...

CS 152: Compiler Design

Formal Languages

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

September 29, 2004

Today's Lecture

3. Formal Languages

What are Languages?

- The language L consists of all sentences
  - with the subject “Peter” or “Joe”
  - with the verb “sleeps” or “works”
- The language L is this set of four sentences
  - $L = \{$“Peter sleeps”, “Joe works”, “Peter works”, “Joe sleeps”$\}$
- But our description above is “vague”
  - natural language is **very** ambigious
  - programming languages can't afford that

Formal Languages

- A language is defined formally by
  - a set T of terminal symbols (not substitutable)
  - a set NT of non-terminal symbols (substitutable)
  - a set P of productions (substitutions)
  - a start symbol $S \in NT$
- For example, $L = (TL, NTL, PL, SL)$:
  - $TL = \{$ "Peter", "Joe", "sleeps", "works" $\}$
  - $NTL = \{$ Sentence, Subject, Verb $\}$
  - $PL = \{$ Sentence $\rightarrow$ Subject Verb, Subject $\rightarrow$ "Peter", Subject $\rightarrow$ "Joe", Verb $\rightarrow$ "sleeps", Verb $\rightarrow$ "works" $\}$
  - $SL =$ Sentence

Grammars

- The set P of productions is also called a grammar
  - generate sentence: go “forward” from S
  - recognize sentence: go “backward” from T
- We can generate the sentence “Peter works”
  - Sentence $\Rightarrow$ Subject Verb $\Rightarrow$ “Peter” Verb $\Rightarrow$ “Peter” “works”
- Or recognize the sentence “Joe works”
  - “Joe” “works” $\Leftarrow$ Subject “works” $\Leftarrow$ Subject Verb $\Leftarrow$ Sentence
- The sentence “sleeps Joe” is not in L
  - “sleeps” “Joe” $\Leftarrow$ Verb “Joe” $\Leftarrow$ Verb Subject $\Leftarrow$???

Syntax Trees

- Sequence of substitutions leads to a parse tree
  - “Joe” “sleeps”
  - (Subject (“Joe”)) (Verb (“sleeps”))
  - (Sentence (Subject (“Joe”)) (Verb (“sleeps”)))
  - also derivation tree or (concrete) syntax tree
- Abstract syntax trees: Forget substitutions!
  - (Sentence (“Joe”) (“sleeps”))
  - only retain the “essential” structure
  - not how we got there

Optional And Repetitive Phrases

- Empty right-hand side to make things optional
  - Example: P = {A $\rightarrow$ “x” B “z”, B $\rightarrow$ “y”, B $\rightarrow$ }
  - Generates: xyz, xz
  - Note: Often ϵ denotes “empty”
- Recursive right-hand side to make things repeat
  - Example: P = {A $\rightarrow$ “(” A “)”, A $\rightarrow$ “x”}
  - Generates: x, (x), ((x)), (((x))), . . .
  - Note: Language generated by P is infinite
- “All useful languages are infinite!”

Notations for Grammars

- Canonical Backus-Naur-Form
  - Productions have the form $\alpha \rightarrow \beta$
- (Plain) Backus-Naur-Form
  - Allows $\alpha \rightarrow \beta|\gamma$ instead of $\alpha \rightarrow \beta$, $\alpha \rightarrow \gamma$
- **Extended** Backus-Naur-Form (in EBNF)

```
Grammar = Production {Production}.
Production = identifier "=" Expression ".".
Expression = Term {"|" Term}.
Term = Factor {Factor}.
Factor = string | identifier
  | "(" Expression ")" | "[" Expression "]"
  | "{" Expression "}".
string = """ character {character} """.
identifier = letter {letter | digit }.
```

- Notes: $|$ = choice, () = precedence, [] = zero or one (optional), {} = zero or many

Syntax Trees Again

- A simple expression grammar

```
E = T {"+" T}.
T = F {"*" F}.
F = "0" | "1" | "2" | "3".
```

- Play compiler, recognize `1 + 2 * 3`
  - "1" "+" "2" "*" "3"
  - (F "1") "+" (F "2") "*" (F "3")
  - (T (F "1")) "+" (T (F "2") "*" (F "3"))
  - (E (T (F "1")) "+" (T (F "2") "*" (F "3")))
- Abstract syntax tree for `1 + 2 * 3`
  - (+ (1) (* (2) (3)))
  - Can be interpreted by traversing the tree

Ambiguity, Precedence (1)

- Another expression grammar

```
E = E "+" E | E "*" E | "0" | "1" | "2" | "3".
```

- Play compiler, recognize `1 + 2 * 3`
  - “1” “+” “2” “*” “3”
  - (E “1”) “+” (E “2”) “*” (E “3”)
  - Alternative 1
    - (E (E “1”) “+” (E “2”)) “*” (E “3”)
    - (E (E (E “1”) “+” (E “2”)) “*” (E “3”))
  - Alternative 2
    - (E “1”) “+” (E (E “2”) “*” (E “3”))
    - (E (E “1”) “+” (E (E “2”) “*” (E “3”)))

Ambiguity, Precedence (2)

- A grammar is **ambigious** if
  - one sentence can lead to multiple parse trees
- In this example: **precedence** of operators
  - does + bind stronger than * or vice versa?
- The earlier grammar used
  - two separate productions for
  - two levels of precendence
- The design of the **grammar** matters!
  - two grammars producing the same language might not lead to the same parse trees

Classification of Languages (1)

- Assume a set of productions $\alpha \rightarrow \beta$
- Class 3: **Regular languages**
  - Condition: $\alpha =$ NT, $\beta =$ T | T NT
  - Example: A $\rightarrow$ “b” | “b” A
  - Finite automata (finite state machines)
- Class 2: **Context-free languages**
  - Condition: $\alpha =$ NT, $\beta \neq \epsilon$
  - Example: A $\rightarrow$ “b” | “(” A “)”
  - Push-down automata

Classification of Languages (2)

- Class 1: Context-sensitive languages
  - Condition: $|\alpha| \leq |\beta|$
  - Example: “a” A $\longrightarrow$ “a” “b” “c”
  - Linear-bounded automata
- Class 0: Unrestricted languages
  - Condition: None!
  - Example: “a” A “b” $\longrightarrow$ “c” “d”
  - Turing machines
- Compilers only use regular and context-free!

CS 152: Compiler Design

Introduction to Lexical Analysis

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today's Lecture

Introduction

What is Lexical Analysis?

- Lexical analysis: Characters into Tokens
  - Characters $\longrightarrow \boxed{\text{Scanner}} \longrightarrow$ Tokens
    - but that's not the whole story
  - $\boxed{\text{i}}\boxed{\ }\boxed{:}\boxed{=}\boxed{\ }\boxed{\text{i}}\boxed{\ }\boxed{+}\boxed{\ }\boxed{1} \longrightarrow \boxed{\text{Scanner}} \longrightarrow \boxed{\text{i}}\boxed{:=}\boxed{\text{i}}\boxed{+}\boxed{1}$
    - also filter whitespace (e.g. blank, tab)
  - $\boxed{(}\boxed{*}\boxed{\ }\boxed{\text{x}}\boxed{\text{z}}\boxed{\ }\boxed{*}\boxed{)} \longrightarrow \boxed{\text{Scanner}} \longrightarrow$
    - also filter comments
- Why do we separate this from syntax analysis?
  - Simplify the grammar for parsing (whitespace).
  - Simpler techniques suffice for lexical analysis.
  - Isolate later phases from representation (e.g. new representations for existing tokens).

Implementation Concerns

- Lexical analysis can be expensive
  - accesses source file, possibly from disk
  - accesses every character at least once
  - **production** compilers use efficient techniques
    - to buffer the input stream
    - to touch characters "as little as possible"
- Interface to the parser
  - return token objects vs. return small integers
  - return one token vs. return list of tokens
- Source text: file name vs. file handle vs. string

Regular Languages and Grammars (1)

- How do we **recognize** the **structure** of a sequence of characters? $\Rightarrow$ Formal languages!
- Regular languages (class 3)
  - suffice for **most** lexical analysis tasks (e.g. not for nested comments)
  - Productions $\alpha \rightarrow \beta$: $\alpha =$ NT, $\beta =$ T | T NT
  - Or: **Single non-recursive EBNF production**
- Examples
  - `identifier = letter {letter | digit}.`
  - `number = digit {digit}.`
  - `while = "W" "H" "I" "L" "E".`

Regular Languages and Grammars (2)

- Is this grammar regular? Try to transform it...

```
E = T {"+" T}.
T = F {"*" F}.
F = "0" | "1".
```

$\Rightarrow$

```
E = T {"+" T}.
T = ("0" | "1") {"*" ("0" | "1")}.
```

$\Rightarrow$

```
E = ("0" | "1") {"*" ("0" | "1")}
    {"+" ("0" | "1") {"*" ("0" | "1")}}.
```

- Yes, the grammar is regular! If it were not, the transformation process would never end...

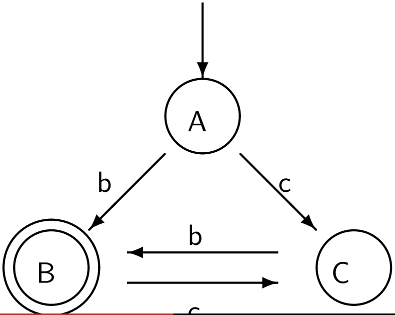
Finite Automata (Finite State Machines)

What is a Finite Automaton?

- A finite automaton is defined formally by
  - a set Q of symbols
  - a set S of states
  - a set $T \subseteq S \times Q \times S$ of transitions
  - a set $F \subseteq S$ of final states
  - a start state $R \in S$
- For example:
  - $Q = \{$ b, c $\}$, $S = \{$ A, B, C $\}$
  - $T = \{$ A $\longrightarrow$b$\longrightarrow$ B, A $\longrightarrow$c$\longrightarrow$ C, B $\longrightarrow$c$\longrightarrow$ C, C $\longrightarrow$b$\longrightarrow$ B $\}$
  - $F = \{$ B $\}$, $R =$ A

Drawing Finite Automata

- Things get easier if we use some graphics
  - each state becomes a circle
  - each transition becomes a labeled arrow
  - final states are marked with another circle
  - the start state has a “blank” arrow coming in
- For example:



Language Accepted by Finite Automata

- A finite automaton
  1. Starts in the start state (surprise!)
  2. Looks at the next input symbol
     - if symbol triggers transition: consume symbol, change state, goto 2; else: goto 3
  3. If the current state is a final state, accept the input, otherwise fail
- For example:
  - accepts: "b", "bcb", "bcbcbcbcbcb", . . .
  - rejects: empty input, "bc", "cbcbc", . . .
  - `L = ["c"] "b" {"c" "b"}.`
  - special: "cbb", "cbx" $\Rightarrow$ accepts "cb" part!

Some Notes on Finite Automata

- A finite automaton
  - recognizes regular (class 3) languages
  - regular grammar $\Leftrightarrow$ finite automaton
- Can be implemented by
  - a transition table: index by current state and symbol, resulting element is next state
  - nested if/switch instructions, current state is the position in the program
- Abreviations for drawing transitions
  - unlabeled: taken for “all other” symbols
  - combined: one arrow for many symbols

Implementation

Work from EBNF Productions (1)

- `identifier = letter {letter|digit}.`

```
read( ch )
if is_letter( ch ):
  read( ch )
  while is_letter( ch ) or is_digit( ch ):
    read( ch )
else:
  error()
```

- Construct k leads to a program fragment $p(k)$

k	$p(k)$
"x"	if ch == "x": read(ch) else: error()
(e)	$p(e)$
$[e]$	if ch in first(e): $p(e)$
$\{e\}$	while ch in first(e): $p(e)$
$f_0 f_1 \ldots$	$p(f_0)p(f_1)\ldots$
$t_0 \| t_1 \| \ldots$	if ch in first(t_0): $p(t_0)$ elif ch in first(t_1): $p(t_1)$

Work from EBNF Productions (2)

- What is first(e)?
    - all starting symbols of e, all valid characters e can begin with
    - first(identifier) = { "a",..., "z", "A",..., "Z" }
    - if |first(e)| is small compare characters, otherwise write a function (e.g. `is_letter`)...
- Required grammar properties:
    - grammar must be unambigious (deterministic)
    - choices: all first sets must be disjunct (i.e. their intersection must be empty)
- Of course, you also have to "assemble" the symbol, in case of identifiers, numbers, strings, ...

Dealing with Keywords

- `while = "W" "H" "I" "L" "E".`

  ```
  read( ch )
  if ch == "W":
    read( ch )
    if ch == "H":
      read( ch )
        ...
    else:
      error()
  else:
    error()
  ```
- This is **way** too complicated!
  - write a function to check whether a string is a keyword
  - after recognizing an identifier call that function
  - return either the correct keyword token or the identifier token with the value of the "assembled" string

Representing Tokens

- Various possibilities, for example as a class:

  ```
  public class Token {
    public int kind;
    public String str; // for identifiers
    public int val; // for numbers
    public int start_pos, end_pos;
  }
  ```

- Create an instance when one token is recognized
  - set `kind` according to the kind of token, e.g. 1 for identifiers, 2 for numbers, 3 for the keyword `PROGRAM`, 4 for the keyword `WHILE`, etc.
  - define **constants** for these values so the parser can access the kind field easily
  - set `str` and `val` to the recognized content of identifier and number tokens

Scanner Skeleton

- The scanner can be structured like this (at least for Simple):

```
...
while not end-of-file:
  read( ch )
  if is_whitespace( ch ):
    read( ch )
  elif is_letter( ch ):
    must be an identifier or a keyword
    process according to earlier slide
    return either keyword or identifier token
  elif is_digit( ch ):
    must be a number
    process and assemble as string
    convert string to integer
    return number token with integer value
  elif ch == "(":
    read( ch )
    return token for open parenthesis
```

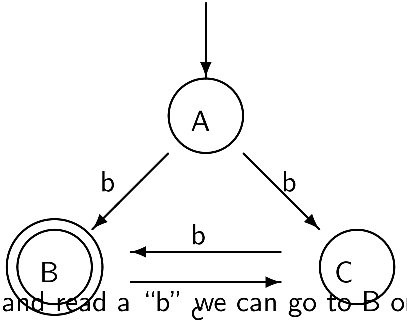
More on Lexical Analysis

Regular Expressions

- Regular expressions (REs) are
  - a different notation for regular grammars
  - used heavily in text processing languages
- One possible definition:
  - the empty string ϵ is a RE
  - terminal symbols are REs
  - if α and β are REs
    - $\alpha\beta$ is a RE (concatenation)
    - $(\alpha|\beta)$ is a RE (choice)
    - $(\alpha)^+$ is a RE (one or more)
    - $(\alpha)^*$ is a RE (zero or more)

Non-Deterministic Finite Automata

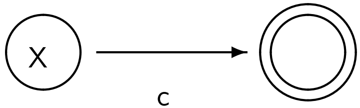
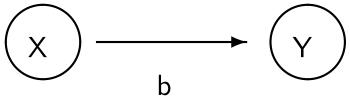
- A finite automaton is **non-deterministic** if
  - there exists a state from which the **same** input symbol leads to two or more **different** states
- For example:



- If we are in A and read a "b" we can go to B or C
- Every NDFA can be converted to a DFA
  - makes scanner generators easier to write
  - see Appel or Aho for the algorithm

Automata ⇔ Grammars (1)

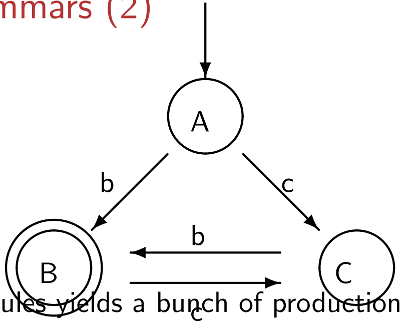
- Deterministic finite automata ⇔ regular grammars?
- Two simple rules:

	
X → “c”	X → “b” Y

- How to apply these rules:
  - go through states, apply rules for each transition **out** of the state we are considering
  - first rule: only transitions to a **final** state
  - second rule: all transitions, including the ones we applied the first rule to

Automata ⇔ Grammars (2)

- For example:



- Applying the rules yields a bunch of productions:

  $A \rightarrow$ "b", $A \rightarrow$ "b" B, $A \rightarrow$ "c" C
  $B \rightarrow$ "c" C
  $C \rightarrow$ "b", $C \rightarrow$ "b" B

- That doesn't show structure well enough, simplify:

  $A \rightarrow$ "b" $|$ "b" B $|$ "c" C
  $B \rightarrow$ "c" C
  $C \rightarrow$ "b" $|$ "b" B

Automata ⇔ Grammars (3)

- Still not great, so substitute C into A and B:

$$A \rightarrow \text{“b”} \mid \text{“b”}\ B \mid \text{“c”}\ (\ \text{“b”} \mid \text{“b”}\ B\ )$$
$$B \rightarrow \text{“c”}\ (\ \text{“b”} \mid \text{“b”}\ B\ )$$

- Simplify again, by removing parenthesis:

$$A \rightarrow \text{“b”} \mid \text{“b”}\ B \mid \text{“c”}\ \text{“b”} \mid \text{“c”}\ \text{“b”}\ B$$
$$B \rightarrow \text{“c”}\ \text{“b”} \mid \text{“c”}\ \text{“b”}\ B$$

- Convert B production to EBNF-like notation:

$$A \rightarrow \text{“b”} \mid \text{“b”}\ B \mid \text{“c”}\ \text{“b”} \mid \text{“c”}\ \text{“b”}\ B$$
$$B \rightarrow \text{“c”}\ \text{“b”}\ \{\ \text{“c”}\ \text{“b”}\ \}$$

- Substitute B into A:

$$A \rightarrow \text{“b”} \mid \text{“b”}\ (\text{“c”}\ \text{“b”}\ \{\ \text{“c”}\ \text{“b”}\ \}) \mid \text{“c”}\ \text{“b”} \mid$$
$$\text{“c”}\ \text{“b”}\ (\text{“c”}\ \text{“b”}\ \{\ \text{“c”}\ \text{“b”}\ \})$$

Automata $\Leftrightarrow$ Grammars (4)

- Drop the parenthesis:

  $A \rightarrow$ "b" | "b" "c" "b" { "c" "b" } | "c" "b" | "c" "b" "c" "b" { "c" "b" }

- Eliminate redundant prefixes (first two terms):

  $A \rightarrow$ "b" { "c" "b" } | "c" "b" | "c" "b" "c" "b" { "c" "b" }

- Eliminate redundant prefixes (last two terms):

  $A \rightarrow$ "b" { "c" "b" } | "c" "b" { "c" "b" }

- Combine the alternatives:

  $A \rightarrow$ ["c"] "b" { "c" "b" }

Automata ⇔ Grammars (5)

- Lessons to learn from this:
  - We can play with grammars like we play with equations in algebra.
  - Rewritten in “proper” EBNF we have exactly what we “guesstimated” in the last lecture.
  - The process can be **very** involved; if you need a little challenge, make C into a final state as well and try the same thing!
  - To get to a result, you can freely invent notation, you just have to use your notation consistently.
- Of course, we can also go from EBNF to an automaton, provided the language is regular. . .

Generating Scanners

CS 152: Compiler Design

Introduction to Syntactic Analysis

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today's Lecture

Introduction

What is Syntactic Analysis? (1)

- Syntax analysis: Tokens into Tree + Symbol Table
  - Tokens $\rightarrow \boxed{\text{Parser}} \rightarrow$ Tree + Symbol Table
    - but that's not the whole story
  - $\boxed{\text{IF}}\boxed{\text{17}}\boxed{\text{THEN}} \rightarrow \boxed{\text{Parser}} \rightarrow$?
    - also error handling, "17" is not a condition (at least not in our language, Simple)
- There is more than one way of course...
  - Less passes $\Rightarrow$ Symbol Table + Code
  - More passes $\Rightarrow$ Tree, Symbol Table later
  - For simple languages we can do semantic analysis (e.g. type checking) in the parser as well

What is Syntactic Analysis? (2)

- Syntactic analysis
  - recognizes (and checks!) the **structure** of the input
  - produces **some** form of intermediate representation
- In most languages we can clearly distinguish
  - parts of the parser that build the symbol table (declarations in the syntax)
  - parts of the parser that build the tree (instructions and expressions in the syntax)
- Therefore building those two structures in the parser seems to be a good idea. . .

What is Syntactic Analysis? (3)

- Why do we separate this from lexical analysis?
  - Many of the same reasons explained earlier
  - We also need more powerful algorithms
    - i.e. a different kind of automaton
- Why do we separate this from semantic analysis and code generation?
  - Transformations that simplify both (e.g. rewrite WHILE into IF/REPEAT, fold constants)
  - Flexibility to exchange the "back end" (e.g. different target machines, interpreter)
  - In general, this is again just "good software engineering" as learned in other classes

Context-Free Languages

What is a Context-Free Language?

- How do we **recognize** the **structure** of a sequence of tokens?
  $\Rightarrow$ Formal languages!
- Context-free languages (class 2)
  - suffice for syntax analysis tasks (but not for semantic constraints, e.g. type checking)
  - Productions $\alpha \rightarrow \beta$: $\alpha = \mathsf{NT}$, $\beta \neq \epsilon$
- Examples:
  - `Expr = ident | "(" Expr ")".`

  -
    ```
    Inst = Assign | While | If.
    While = "WHILE" Cond "DO" Inst "END"
    ```

Context-Free versus Regular Languages

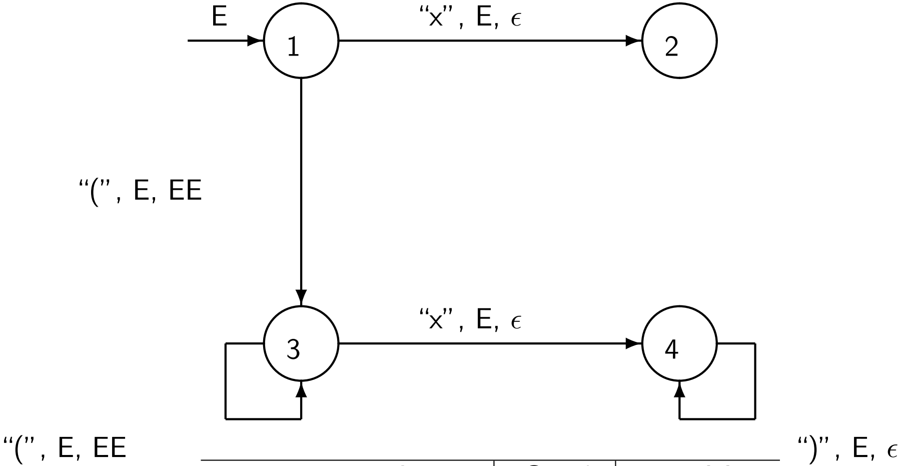
- Regular languages can not express “recursion in the middle”
  - a finite automaton has at most n states
  - but we can always take $n + 1$ recursions
- Context-free languages
  - are recognized by **pushdown automata**
    - which are similar to finite automata
    - but also have an **infinite stack**
- Context-free languages can express “recursion in the middle”
  - occurs frequently in programming languages

Deterministic Push-Down Automata (1)

- Deterministic push-down automata
  - start with a symbol on the stack
  - have (deterministic) transitions that
    - fire on input symbol and top of stack
    - push new symbols on the stack
  - accept if stack empty **and** input exhausted
    - no final states
- Context-free languages $\Leftrightarrow$ non-deterministic PDAs
  - deterministic PDAs are less powerful
  - but easier to play with (i.e. can be implemented)

Deterministic Push-Down Automata (2)

- For example: `E = "x" | "(" E ")".`



Input	Stack	Transition
$\boxed{“(”}$ “(” “x” “)” “)”	$\boxed{\text{E}}$	1 → 3

Fundamentals of Parsing

Overview

- The parser tries to recognize a sentence
  - which comes down to constructing a parse tree
  - but it's not as smart as we are (hopefully :-)
- Most parsing algorithms
  - go from **left to right** over the input
  - look at the **current** token and the **next** one
  - have to make **local** choices
- Two basic approaches:
  - top-down: start symbol $\Rightarrow$ what to derive next
  - bottom-up: terminal symbols $\Rightarrow$ what to reduce next

Top-Down Parsing (1)

- Grammar: E = “x” | “[” E “]”. Input: “[” “[” “x” “]” “]”
- Top-down parse:

Goal	Parse Tree	Choices
E_0	(E ?)	“x”, $\boxed{\text{“[”}}$
E_0	(E “[” ?)	$\boxed{\text{E}}$
E_1	(E “[” (E ?) ?)	“x”, $\boxed{\text{“[”}}$
E_1	(E “[” (E “[” ?) ?)	$\boxed{\text{E}}$
E_2	(E “[” (E “[” (E ?) ?) ?)	$\boxed{\text{“x”}}$, “[”
E_1	(E “[” (E “[” (E “x”) ?) ?)	$\boxed{\text{“]”}}$
E_0	(E “[” (E “[” (E “x”) “]”) ?)	$\boxed{\text{“]”}}$
E	(E “[” (E “[” (E “x”) “]”) “]”)	none, accepted

- In each step: Match token choices to the actual input or set new goal if no token choices.
- Parse tree “grows” from the root to the leaves

Top-Down Parsing (2)

- Another expression grammar:

```
E = T {"+" T}.
T = F {"*" F}.
F = id | num | "(" E ")".
```

- Top-down for the sentence `1 * ( i + 3 )`:

Goal	Parse Tree	Choices
E_0	(E ?)	$\boxed{\text{T}}$
T_0	(E (T ?) ?)	$\boxed{\text{F}}$
F_0	(E (T (F ?) ?) ?)	id, $\boxed{\text{num}}$, “(”
T_0	(E (T (F “1”) ?) ?)	ϵ, $\boxed{\text{“*”}}$
T_0	(E (T (F “1”) “*” ?) ?)	$\boxed{\text{F}}$
F_1	(E (T (F “1”) “*” (F ?) ?) ?)	id, num, $\boxed{\text{“(”}}$
F_1	(E (T (F “1”) “*” (F “(” ?) ?) ?)	$\boxed{\text{E}}$
E_1	(E (T (F “1”) “*” (F “(” (E ?) ?) ?) ?)	$\boxed{\text{T}}$
...	...	...

- (E (T (F “1”) “*” (F “(” (E (T (F “i”)) “+” (T (F “3”))) “)”)))

Bottom-Up Parsing (1)

- Grammar: E = “x” | “[” E “]”. Input: “[” “[” “x” “]” “]”
- Bottom-up parse:

Input / Parse Tree	**Stack**	**Action**
$\boxed{\text{“[”}}$ “[” “x” “]” “]”		shift “[”
“[” $\boxed{\text{“[”}}$ “x” “]” “]”	“[”	shift “[”
“[” “[” $\boxed{\text{“x”}}$ “]” “]”	“[” “[”	shift “x”
“[” “[” “x” $\boxed{\text{“]”}}$ “]”	“[” “[” $\boxed{\text{“x”}}$	reduce E
“[” “[” (E “x”) $\boxed{\text{“]”}}$ “]”	“[” “[” E	shift “]”
“[” “[” (E “x”) “]” $\boxed{\text{“]”}}$	“[” $\boxed{\text{“[” E “]”}}$	reduce E
“[” (E “[” (E “x”) “]”) $\boxed{\text{“]”}}$	“[” E	shift “]”
“[” (E “[” (E “x”) “]”) “]”	$\boxed{\text{“[” E “]”}}$	reduce E
(E “[” (E “[” (E “x”) “]”) “]”)	E	done, accepted

- In each step: Reduce on stack if possible, otherwise shift next input symbol.

Bottom-Up Parsing (2)

- Expression grammar again:[1]

```
E = T {"+" T}.
T = F {"*" F}.
F = id | num | "(" E ")".
```

- Bottom-up for the sentence `1 * ( i + 3 )`:

Input / Parse Tree	Stack	A
$\boxed{\text{“1”}}$ “*” “(” “i” “+” “3” “)”		s
“1” $\boxed{\text{“*”}}$ “(” “i” “+” “3” “)”	$\boxed{\text{“1”}}$	re
(F “1”) $\boxed{\text{“*”}}$ “(” “i” “+” “3” “)”	$\boxed{\text{F}}$	**s**
(F “1”) “*” $\boxed{\text{“(”}}$ “i” “+” “3” “)”	F “*”	s
(F “1”) “*” “(” $\boxed{\text{“i”}}$ “+” “3” “)”	F “*” “(”	s
(F “1”) “*” “(” “i” $\boxed{\text{“+”}}$ “3” “)”	F “*” “(” $\boxed{\text{“i”}}$	re
(F “1”) “*” “(” (F “i”) $\boxed{\text{“+”}}$ “3” “)”	F “*” “(” $\boxed{\text{F}}$	re
(F “1”) “*” “(” (T (F “i”)) $\boxed{\text{“+”}}$ “3” “)”	F “*” “(” $\boxed{\text{T}}$	**s**
(F “1”) “*” “(” (T (F “i”)) “+” $\boxed{\text{“3”}}$ “)”	F “*” “(” T “+”	s
(F “1”) “*” “(” (T (F “i”)) “+” “3” $\boxed{\text{“)”}}$	F “*” “(” T “+” $\boxed{\text{“3”}}$	re

Derivations and Reductions (1)

- Terminology:[2]
  - A non-terminal symbol **derives to** a string
  - A string **reduces to** a non-terminal symbol
- Notation: Assume E $\longrightarrow$ E "-" E, E $\longrightarrow$ "x", E $\longrightarrow$ "y"
  - E $\Rightarrow$ E "-" E $\Rightarrow$ "x" "-" E $\Rightarrow$ "x" "-" "y" (direct)
    E $\overset{*}{\Rightarrow}$ "x" "-" "y" (indirect)
  - "x" "-" "y" $\Leftarrow$ E "-" "y" $\Leftarrow$ E "-" E $\Leftarrow$ E (direct)
    "x" "-" "y" $\overset{*}{\Leftarrow}$ E (indirect)
- It is undefined
  - which non-terminal is derived in each step
  - which string is reduced in each step

[2] A string, here, is a sequence of terminals or non-terminals.

Derivations and Reductions (2)

- Canonical derivations (reductions)
  - define the order of derivations (reductions)
- Derivations:
  - Left-canonical: derive left-most non-terminal
  - Right-canonical: derive right-most non-terminal
- Reductions:
  - Left-canonical: reduce left-most string
  - Right-canonical: reduce right-most string
- Left-most derivations $\Leftrightarrow$ Right-most reductions (and vice versa)

Derivations and Reductions (3)

- Example: $E \rightarrow E$ "-" E, $E \rightarrow$ "x", $E \rightarrow$ "y"
  - Left-most derivation:
    $E \Rightarrow E$ "-" $E \Rightarrow$ "x" "-" $E \Rightarrow$ "x" "-" "y"
  - Right-most derivation:
    $E \Rightarrow E$ "-" $E \Rightarrow E$ "-" "y" $\Rightarrow$ "x" "-" "y"
  - Left-most reduction:
    "x" "-" "y" $\Leftarrow E$ "-" "y" $\Leftarrow E$ "-" $E \Leftarrow E$
  - Right-most reduction:
    "x" "-" "y" $\Leftarrow$ "x" "-" $E \Leftarrow E$ "-" $E \Leftarrow E$
- Relation to Parsing:
  - Top-down parser $\Leftrightarrow$ left-most derivations
  - Bottom-up parser $\Leftrightarrow$ left-most reductions

Recursive Productions (1)

- Classification:
  - left recursion: A $\longrightarrow$ A “x” | “y” yxxx...
  - right recursion: A $\longrightarrow$ “x” A | “y” ...xxxy
  - central rec.: A $\longrightarrow$ “x” A “x” | “y” xx...y...xx
  - indirect recursion: A $\longrightarrow$ “x” B | “z”, B $\longrightarrow$ A “y”
- Top-down parsers and left recursion:
  - top-down parsers perform left-most derivations
  - if a production is left-recursive, this never ends
  - (A ?) $\Rightarrow$ (A (A ?) ?) $\Rightarrow$ (A (A (A ?) ?) ?) $\Rightarrow$ ...

Recursive Productions (2)

- Eliminating left recursion:
  - transform into right recursion
    - from E $\rightarrow$ E “+” T, E $\rightarrow$ T
    - into E $\rightarrow$ T E’, E’ $\rightarrow$ “+” T E’, E’ $\rightarrow \epsilon$
  - transform into iteration
    - from E $\rightarrow$ E “+” T, E $\rightarrow$ T
    - into E = T {“+” T}.
  - observe the how {“+” T} is exactly E’
- Why not E $\rightarrow$ T “+” E, E $\rightarrow$ T instead?
  - the parser can’t choose the “right” E!

Recursive Descent Parsing

What is Recursive Descent Parsing? (1)

- Recursive descent parsing (predictive parsing)
  - easy-to-implement top-down parsing technique
  - non-terminal productions become procedures
- While = "WHILE" Condition "DO" Instructions "END".

```
def While():
  match( "WHILE" )
  Condition()
  match( "DO" )
  Instructions()
  match( "END" )

def match( kind ):
  if kind == current.kind:
    next() // updates current
  else:
    raise "Expected " + kind + " not " + current
```

- PDA states = position in the procedure
- PDA stack = procedure activation stack

What is Recursive Descent Parsing? (2)

- Recursive descent parsing only works
  - if the parser is **always** able to choose an alternative by looking only **one** token ahead
- We need to ensure this
  - by analyzing the grammar of the language
  - by transforming it if necessary (and possible)
- Consider: E $\rightarrow$ T "+" E, E $\rightarrow$ T, T $\rightarrow$ "x"
  - Which E production when we look at "x"?
    - choose longer one: can't parse "x" alone
    - choose shorter one: can't parse "x+x+..."

First and Follow Sets (1)

- For a string[3] α, first(α) is defined as
  - the set of all tokens that prefix **any** sentence derivable from α
- For a non-terminal X, follow(X) is defined as
  - the set of all tokens that can immediately follow X in **any** derivation
- Example: A $\rightarrow$ “0” | “1”

α	first(α)	follow(α)
“0”	“0”	undefined
“1”	“1”	undefined
A	“0”, “1”	

[3]Remember that “string” means “sequence of terminals or non-terminals”

First and Follow Sets (2)

- Example: A $\rightarrow$ B | A B, B $\rightarrow$ "0" | "1"

α	first(α)	follow(α)
"0"	"0"	undefined
"1"	"1"	undefined
A	"0", "1"	"0", "1"
B	"0", "1"	

- Example: Z $\rightarrow$ X Y Z, Z $\rightarrow$ "d", Y $\rightarrow$ "c", Y $\rightarrow$ ϵ, X $\rightarrow$ "a", X $\rightarrow$ Y

α	first(α)	follow(α)
"a"	"a"	undefined
"c"	"c"	undefined
"d"	"d"	undefined
X	"a", "c"	"a", "c", "d"
Y	"c"	"a", "c", "d"
Z	"a", "c", "d"	

First and Follow Sets (3)

- Example: S → A | B, A → "c" A "+" "b" | "a", B → "c" B "+" "a" | "b"

α	first(α)	follow(α)
S	"a", "b", "c"	
A	"a", "c"	"+"
B	"b", "c"	"+"

- Example: E = T {("+"|"-") T}. T = F {("*"|"/") F}. F = "x" | "(" E ")".

α	first(α)	follow(α)
E	"x", "("	")"
T	"x", "("	"+", "-", ")"
F	"x", "("	"*", "/", "+", "-", ")"

First and Follow Sets (4)

- Example: E $\rightarrow$ T E', E' $\rightarrow$ "+" T E' | "-" T E' | ϵ, T $\rightarrow$ F T', T' $\rightarrow$ "*" F T' | "/" F T' | ϵ, F $\rightarrow$ "x" | "(" E ")"

α	first(α)	follow(α)
E	"x", "("	")"
E'	"+", "-"	")"
T	"x", "("	"+", "-", ")"
T'	"*", "/"	"+", "-", ")"
F	"x", "("	"*", "/", "+", "-", ")"

- Notes:
  - Last two: Same language, different grammar
  - Algorithms for first/follow: see Aho or Appel

LL(1) Grammars (1)

- A grammar is an LL(1) grammar if
  - all productions conform to the LL(1) conditions
- LL(1) conditions:
  1. For each production $A \rightarrow \sigma_1 | \sigma_2 | \ldots | \sigma_n$:
     $$\text{first}(\sigma_i) \cap \text{first}(\sigma_j) = \emptyset, \forall i \neq j$$
  2. If non-terminal X can derive the empty string:
     $$\text{first}(X) \cap \text{follow}(X) = \emptyset$$
- Notes:
  - condition 1 means we always know which alternative to choose
  - condition 2 means we always know whether something optional is there or not

LL(1) Grammars (2)

- Example: S → A | B, A → "c" A "+" "b" | "a", B → "c" B "+" "a" | "b"

α	first(α)	follow(α)
S	"a", "b", "c"	
A	"a", "c"	"+"
B	"b", "c"	"+"

first(A) $\cap$ first(B) = { "c" } $\Rightarrow$ not LL(1)!

- Example: E = T {("+" | "-") T}. T = F {("*" | "/") F}. F = "x" | "(" E ")".

α	first(α)	follow(α)
E	"x", "("	")"
T	"x", "("	"+", "-", ")"
F	"x", "("	"*", "/", "+", "-", ")"

first({("+" | "-") T}) $\cap$ follow({("+" | "-") T}) = $\emptyset$,
first({("*" | "/") F}) $\cap$ follow({("*" | "/") F}) = $\emptyset$
$\Rightarrow$ LL(1)!

First and Follow Sets (1)*

- first(α): defined for a **string**[4] α
    - set of terminal symbols that α starts with
    - t $\in$ first(α) if **any derivation** contains

$$\ldots \Rightarrow \alpha \Rightarrow \text{“t”}\ \beta \Rightarrow \ldots$$

- Example: A $\rightarrow$ B “x”, B $\rightarrow$ “a” | “b” | ϵ
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{B}}\ \text{“x”}} \Rightarrow \boxed{\text{“a”}}\ \text{“x”}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{B}}\ \text{“x”}} \Rightarrow \boxed{\text{“b”}}\ \text{“x”}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{B}}\ \text{“x”}} \Rightarrow \epsilon\ \boxed{\text{“x”}}$
- first(“a” “x”) = {“a”}, first(“b” “x”) = {“b”}, first(ϵ “x”) = {“x”}, first(B) = {“a”, “b”, “x”}, first(B “x”) = {“a”, “b”, “x”}, first(A) = {“a”, “b”, “x”}

[4] terminal and non-terminal symbols

First and Follow Sets (2)*

- follow(X): defined for a **non-terminal** symbol X
  - set of terminal symbols that follow X
  - t $\in$ follow(X) if **any derivation** contains

$$\ldots \Rightarrow X \text{ ``t''} \Rightarrow \ldots$$

- Example: A $\rightarrow$ B "x", B $\rightarrow$ "a" | "b" | ϵ
  - $\text{A} \Rightarrow \text{B} \boxed{\text{``x''}} \Rightarrow \text{``a''}\ \text{``x''}$
  - $\text{A} \Rightarrow \text{B} \boxed{\text{``x''}} \Rightarrow \text{``b''}\ \text{``x''}$
  - $\text{A} \Rightarrow \text{B} \boxed{\text{``x''}} \Rightarrow \epsilon\ \text{``x''}$
- follow(B) = {"x"}, follow(A) = {}

First and Follow Sets (3)*

- Example: $\text{A} \rightarrow \text{“y”} \mid \text{A “x”}$
  - first(α):
    - $\boxed{\text{A}} \Rightarrow \boxed{\text{“y”}}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{A}}\ \text{“x”}} \Rightarrow \boxed{\text{“y”}}\ \text{“x”}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{A}}\ \text{“x”}} \Rightarrow \boxed{\boxed{\boxed{\text{A}}\ \text{“x”}}\ \text{“x”}} \Rightarrow \ldots$

      $\Rightarrow \boxed{\text{“y”}}\ \text{“x”} \ldots \text{“x”}$
    - therefore e.g. first(A) $=$ {“y”}
  - follow(X):
    - $\text{A} \Rightarrow \text{“y”}$
    - $\text{A} \Rightarrow \text{A}\ \boxed{\text{“x”}} \Rightarrow \text{“y”}\ \text{“x”}$
    - $\text{A} \Rightarrow \text{A}\ \boxed{\text{“x”}} \Rightarrow \text{A}\ \boxed{\text{“x”}}\ \text{“x”} \Rightarrow \ldots$

      $\Rightarrow \text{A}\ \boxed{\text{“x”}} \ldots \text{“x”} \Rightarrow \text{“y”}\ \text{“x”} \ldots \text{“x”}$
    - therefore follow(A) $=$ {“x”}

First and Follow Sets (4)*

- Example: A $\longrightarrow$ “y” | “x” A
  - first(α):
    - $\boxed{\text{A}} \Rightarrow \boxed{\text{“y”}}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{“x”}}\ \boxed{\text{A}}} \Rightarrow \boxed{\text{“x”}}\ \boxed{\text{“y”}}$
    - $\boxed{\text{A}} \Rightarrow \boxed{\boxed{\text{“x”}}\ \boxed{\text{A}}} \Rightarrow \boxed{\boxed{\text{“x”}}\ \boxed{\boxed{\text{“x”}}\ \boxed{\text{A}}}} \Rightarrow \dots$
      $\Rightarrow \boxed{\text{“x”}}\ \boxed{\text{“x”}} \dots \boxed{\text{“y”}}$
    - therefore e.g. first(A) = {“x”, “y”}
  - follow(X):
    - A $\Rightarrow$ “y”
    - A $\Rightarrow$ “x” A $\Rightarrow$ “x” “y”
    - A $\Rightarrow$ “x” A $\Rightarrow$ “x” “x” A $\Rightarrow \dots$
      $\Rightarrow$ “x” “x” $\dots$ “x” A $\Rightarrow$ “x” “x” $\dots$ “x” “y”
    - therefore follow(A) = {}

Implementation

Working from EBNF Grammars (1)

- The basic idea:
  - left side of production ⇒ procedure
  - A = ... ⇒ def A(): ...
  - right side of production ⇒ call
  - ... = ... A .... ⇒ ... A() ...
- Example: A = B C D. B = ... C = ... D = ...

```
def A():
  B()
  C()
  D()
def B(): ...
def C(): ...
def D(): ...
```

- If B, C, and D are successful, we recognized A.

Working from EBNF Grammars (2)

- Optional Parts: A = . . . [B] . . . .

```
def A():
  ...
  if current.kind in first(B):
    B()
  ...
```

- Repetitive Parts: A = . . . { B } . . . .

```
def A():
  ...
  while current.kind in first(B):
    B()
  ...
```

- Use comparison for small first sets, a separate procedure for large ones. . .
- Lexical analysis used the same rules, but there was no recursion!

Working from EBNF Grammars (3)

- Each non-terminal (production)
  - becomes a procedure that parses (sub-) sentences of that part of the grammar
  - (indirectly) recursive productions become (indirectly) recursive procedure calls
- While = “WHILE” Condition “DO” Instructions “END”.

```
def While():
  match( "WHILE" )
  Condition()
  match( "DO" )
  Instructions() // possible recursion
  match( "END" )
```

Working from EBNF Grammars (4)

- Condition = Expression (“=” | “#” | “<” | “>” |“<=” | “>=”) Expression.

```
def Condition():
  Expression()
  if current.kind in ["=","#","<",">","<=",">="]:
    next() // updates current
  else:
    raise "Comparison expected not " + current
  Expression()
```

or (with a smart match procedure)

```
def Condition():
  Expression()
  match( ["=","#","<",">","<=",">="] )
  Expression()
```

however that can be problematic later when we actually want to know **which** operator it was...

Working from EBNF Grammars (5)

- Requesting Tokens from the Scanner

  ```
  def next():
    last = current
    current = Scanner.next()
  ```

  It can be useful to remember the previous token for semantic analysis. . .

- Matching Terminal Symbols

  ```
  def match( kind ):
    if current.kind == kind:
      next()
    else:
      error( kind + " expected, not " + current )
  ```

  Read next token if everything's fine, else give an error

Working from EBNF Grammars (6)

- Assignment = identifier Selector ":=" Expression.

```
def Assignment():
  match( "identifier" )
  Selector()
  match( ":=" )
  Expression()
```

- If = "IF" Condition "THEN" Instructions ["ELSE" Instructions] "END".

```
def If():
  match( "IF" )
  Condition()
  match( "THEN" )
  Instructions()
  if current.kind == "ELSE":
    match( "ELSE" ) // redundant, can use next
    Instructions()
  match( "END" )
```

Working from EBNF Grammars (7)

- Instructions = Instruction {";" Instruction}.

```
def Instructions():
  Instruction()
  while current.kind == ";":
    next() // look ma, no match!
    Instruction()
```

- Factor = identifier | number | "(" Expression ")".

```
def Factor():
  if current.kind == "identifier":
    next()
  elif current.kind == "number":
    next()
  elif current.kind == "(":
    next()
    Expression()
    match( ")" )
  else:
    error( "Factor expected, not " + current )
```

Tricks of the Trade (1)

- Using an “end-of-file” token
  - We want to avoid checking for the end of the input all the time
  - If the scanner returns an “end-of-file” token, we can do that
- S = "PROGRAM" X "END". ⇒ S = "PROGRAM" X "END" "eof".

```
def S():
  match( "PROGRAM" )
  X()
  match( "END" )
  match( "eof" )
```

- If “eof” turns up within X, an error message will be generated by match()

Tricks of the Trade (2)

- Simplify the parser by tweaking the grammar. . .
- Decls = { ConstDecl | TypeDecl | VarDecl }.
  - first(ConstDecl) = “CONST”, first(TypeDecl) = “TYPE”, first(VarDecl) = “VAR”
  - We would need to check for these in Decls() **and** in ConstDecl(), TypeDecl(), and VarDecl()
- Decls = { “CONST” ConstDecl | “TYPE” TypeDecl | “VAR” VarDecl }.
  - Move “CONST”, “VAR”, “TYPE” into Decls() and only match them there
- However we need to be careful not to change the language[5]

[5]And you still need to produce the right output for the assignment

Handling Syntax Errors

Handling Syntax Errors (1)

- Task: Find as many errors as possible within one compilation, but. . .
  - Avoid “cascading” errors that only occur because of a previous one
  - Don’t ever crash, for a parser “wrong” input is the common case
  - Don’t slow down error free parsing, don’t inflate the parser code
- There is no “perfect” solution, error handling is full of heuristics
  - We need to make assumptions what the programmer “intended” to do, which is bound to fail in some cases

Handling Syntax Errors (2)

- Suppressing cascading errors
  - Once an error is detected, the parser might be “out of sync” with the input
  - Then more “errors” would occur, that are not really there
- A simple heuristic:
  - Keep track of how many tokens were processed since the last error message
  - As long as we only moved *n* tokens, we don’t generate new error messages
  - Obviously, the choice of *n* is critical, and there must be a way for the parser to get “in sync” with the input again

Handling Syntax Errors (3)

- Weak and strong symbols
  - Weak symbols are **often** left out, for example the semicolon separating instructions
  - Strong symbols are **almost never** left out, for example the "CONST" keyword
- If we detect that a **weak** symbol is missing
  - we first generate an error message, but then pretend it was there
  - Examples: ";" between instructions, ")" in factors, use of "=" for assignment

Handling Syntax Errors (4)

- If we detect **any** other syntax error
  - we first generate an error message, and then try to find a strong symbol to synchronize
  - Examples: “CONST”, “VAR”, “TYPE” for declarations, “IF”, “WHILE”, “REPEAT” for instructions
- Using modified grammars
  - Decls = [“CONST” {ConstDecl}] [“VAR” {VarDecl}].
  - Constants must be declared before variables, but we could process them either way...
  - Use: Decls = {“CONST” {ConstDecl} | “VAR” {VarDecl} }. but generate error messages

Handling Syntax Errors (5)

- Obviously, if the input is “wrong enough” none of these approaches will work...
- For your Simple compiler, you will use “panic mode” for error handling
  - As soon as the parser finds **one** error, it throws an exception (incl. message) and aborts!
- Advantages:
  - simple and cheap to implement
  - sufficient for our purposes
- Problems:
  - not appropriate for **production** compilers

Parsing Algorithms and Language Classes

Parsing Algs. and Language Classes (1)

- Depending on the algorithm used for parsing
  - we can only recognize a subset of context-free languages
  - we need to transform the grammar, or change it altogether (language design)
- Algorithms for **all** context-free languages exist
  - but their worst-case time complexity for n input symbols is $O(n^3)$
  - although recent results indicate that in practice they are often "almost" linear

Parsing Algs. and Language Classes (2)

- LL(k) Parsers and Languages
  - parse the input from **left** to right
  - proceed with **left**-most derivations
  - make decisions based on the next **k** tokens
- LR(k) Parsers and Languages
  - parse the input from **left** to right
  - proceed with **right**-most derivations
  - make decisions based on the next **k** tokens
  - SLR(k), LALR(k) are common subclasses
- See Appel or Aho for a more detailed discussion

Visualization

The Observer Pattern

Generating Parsers

CS 152: Compiler Design

Introduction to Semantic Analysis

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today’s Lecture

Introduction

What is Semantic Analysis? (1)

- A variety of things, not easily classified!
  - Books give definitions the **authors** find useful.
  - Here things begin to get **really** fuzzy...
- Enforce **context conditions** the parser could not:
  - e.g. identifiers need to be declared before use
    - ...VAR $\boxed{\text{i}}$: INTEGER; ... $\boxed{\text{k}}$:= 10 ...
  - e.g. distinguish constants, variables, types, ...
    - ...CONST $\boxed{\text{A}}$ = 10; VAR a: $\boxed{\text{A}}$; ...
  - e.g. distinguish between different types
    - ...VAR i: $\boxed{\text{INTEGER}}$; ... $\boxed{\text{i[14]}}$:= 20 ...

What is Semantic Analysis? (2)

- **Evaluate** the program as far as possible:
  - e.g. fold constant expressions
    - . . . a := 10*3+2; . . . ⇒ . . . a := 32; . . .
  - e.g. eliminate "dead" code
    - . . . IF 0 > 10 THEN . . . END . . . ⇒ . . .
  - e.g. perform advanced error checks
    - . . . VAR a: ARRAY $\boxed{8}$ OF INTEGER; . . .
      . . . a[$\boxed{12}$] := 10 . . .
- But **not** further than sensible:
  - e.g. don't do code generation yet (portability!)
  - e.g. the last two are easier to perform later in a more general fashion

What is Semantic Analysis? (3)

- Build **intermediate representation** for the next phase:
  - e.g. abstract syntax trees
    - $\ldots$a := a + 1; $\ldots \Rightarrow \ldots$(:= a (+ a 1)) $\ldots$
  - e.g. abstract machine languages
    - $\ldots$a := a + 1; $\ldots \Rightarrow \ldots$add a, 1, a $\ldots$
- Our (arbitrary) choices for Simple:
  - enforce context conditions
  - fold constants
  - build abstract syntax tree

What is Semantic Analysis? (4)

- Syntax analysis reconsidered:[6]
  - Our previous idea of syntax analysis:
    - Tokens $\rightarrow \boxed{\text{Parser}} \rightarrow$ AST + ST
  - A refined view (only conceptual!):
    - Tokens $\rightarrow \boxed{\text{Parser}} \rightarrow$ CST
    - CST $\rightarrow \boxed{\text{Sem. Analysis}} \rightarrow$ AST + ST
- A recursive descent parser
  - builds CST **implicitly** as procedure calls
  - semantic actions **inside** parsing procedures

[6] CST = concrete syntax tree (parse tree), AST = abstract syntax tree, ST = symbol table

Attribute Grammars

Attribute Grammars (1)

- Start with a context-free **grammar**
  Expr = Term { "+" Term }.
  Term = Factor { "*" Factor }.
  Factor = number | "(" Expr ")".
- Add **attributes** to terminals and non-terminals
  Expr<↑val> = Term<↑val> { "+" Term<↑val1> }.
  Term<↑val> = Factor<↑val> { "*" Factor<↑val1> }.
  Factor<↑val> = number<↑val> | "(" Expr<↑val> ")".
- Two kinds of attributes:
  - <↑x> synthesized (output attribute)
    - "flows" from a symbol into its context
  - <↓x> inherited (input attribute)
    - "flows" from the context into a symbol

Attribute Grammars (2)

- Add **semantic actions**, e.g. as Java code

```
Expr<↑val>                              int val, val1;
= Term<↑val>
{ "+" Term<↑val1>                       val = val + val1;
}.
Term<↑val>                              int val, val1;
= Factor<↑val>
{ "*" Factor<↑val1>                     val = val * val1;
}.
Factor<↑val>                                   int val;
= number                                val = token.value;
| "(" Expr<↑val> ")".
```

- What do we do now?
  - build the parse tree according to the grammar
  - propagate attributes up along the parse tree

Attribute Grammars (3)

- An attributed production like this...

  Expr<↑val> int val, val1;
  = Term<↑val>
  { "+" Term<↑val1> val = val + val1;
  }.

- ...becomes a procedure like this:

```
def Expr():
  val = Term()
  while current.kind == "+":
    next()
    val1 = Term()
    val = val + val1
  return val
```

- Existing parser code gets **augmented** with semantic actions from the attribute grammar!
- Synthesized attributes ⇔ Output parameters
  Inherited attributes ⇔ Input parameters

Symbol Tables

What is a Symbol Table?

- A symbol table keeps track of **declarations**
  - associates the **name** of a declared identifier
  - with a **value** representing its meaning
- Example: . . . CONST a = 26*3+1; . . .
  - name: “a” ⇒ value: 79
- Example: . . . CONST b = 2*a; . . .
  - name: “b” ⇒ value: 158
- We must be able to
  - **insert** a new (name, value) pair
  - **find** the value for a given name

Data Structures for Symbol Tables

- Many data structures support **insert** and **find**
  - How many entries? More insert or more find? Overhead of data structure itself...
- Linear lists
  - insert fast, find slow, keeps declaration order
- Search trees
  - insert okay, find okay, loses declaration order
- Hash tables
  - insert okay, find okay, loses declaration order
- Linear lists are usually good enough for “small” symbol tables (all we ever need)...

Handling Nested Scopes (1)

- Scopes in programming languages
  - define the **visibility** of a declaration
  - in **different** parts of the source text
- In many languages, scopes can be **nested**
  - certain declarations introduce **new** scopes
    - procedures, methods, classes, modules, . . .
  - **inside** already existing scopes
    - methods in classes, classes in classes, . . .
- Since symbol tables store declarations
  - they must handle relevant scope rules correctly

Handling Nested Scopes (2)

- Common case: Block-structured languages
  - declarations in scope X are **not** visible outside
  - declarations made outside X **are** visible in X
    - provided they are not **shadowed** by a new declaration in X
  - common exception: “dot-notation” for fields of records, members of classes, . . .
- Block-structured symbol tables
  - separate table for each scope, each table knows its **outer** scope
  - insert modifies **current** table, find searches current table first, outer if needed

Handling Nested Scopes (3)

- Universe scope
  - a “virtual” scope outside the “largest” scope in a programming language
  - contains **predeclared** identifiers, e.g. types INTEGER, REAL, functions ABS(), ORD(), ...
  - handling these as keywords introduces more special cases than putting them in the universe
- Universe in Simple
  - contains the type INTEGER, nothing else
  - set up by the parser before parsing begins

Handling Declarations (1)

- What are the **kinds** of declarations?
  - Constants, can only be read
  - Variables, can be read and written
  - Types, can be used in variable declarations
  - Procedures, can be called, assigned, . . .
  - . . .
- What do we store for constants?
  - Value (obviously!)
  - Type (if there are multiple)
  - Address (depending on the back end)
  - . . .

Handling Declarations (2)

- What do we store for variables?
  - Type (if there are multiple)
  - Address (depending on the back end)
  - ...
- What do we store for types?
  - Kind (basic vs. structured types)
  - Size (depending on the back end)
  - Structure
    - Arrays: Type of elements, length (index range)
    - Records: Fields (a new symbol table!)
    - ...

Handling Declarations (3)

- Note on types:
  - Basic types are usually **singleton** objects, i.e. the compiler only creates **one** instance and puts it into the universe
- What do we store for procedures?
  - Address (depending on the back end)
  - Signature
    - Parameters (a new symbol table)
    - Return type
  - Local declarations (new symbol table)
  - . . .

Handling Declarations (4)

- Note on procedures and types:
  - If procedures can be assigned to variables and passed as parameters, procedures become type constructors as well
- Various additional information
  - storage classes in C or Java (e.g. register, volatile)
  - estimated usage patterns (for optimizations)
  - ...
- The symbol table is a **central** data structure
  - shared by front end and back end, both may add information

Handling Declarations (5)

- Implementation as class hierarchy
  - base class Entry for all symbol table entries
  - derived classes for kinds (Constant, Variable, Type, Procedure, . . .)
  - derived classes for basic types (Integer, . . .), structured types (Array, . . .)
- Implementation as a single type/class
  - use a **kind** field to distinguish, slots for all possible things we need to store
- Compromise (see Wirth):
  - one class for Variables, Constants, Procedures
  - one class for Types

Visualization

CS 152: Compiler Design

Introduction to Semantic Analysis II

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today's Lecture

Abstract Grammars

What is an Abstract Grammar? (1)

- Concrete Grammars
  - describe the **syntactic** structure of a source language **text**
  - are filled with "punctuation symbols" to **guide** the parser (e.g. to avoid ambiguities)
  - depend on parsing **algorithms** (e.g. no left recursion for top-down parsers)
- Abstract Grammars
  - describe the **essential** structure of a language
  - **disregard** parsing, error handling, and all that
  - provide a better basis for defining **semantics**
- Concrete Syntax Tree vs. Abstract Syntax Tree

What is an Abstract Grammar? (2)

- Concrete grammar for conditional instructions
  - If = "IF" Condition "THEN" Instructions ["ELSE" Instructions] "END".
  - "IF" guides parser to recognize a conditional
  - "THEN", "ELSE" indicate start of instructions
  - "END" avoids the "dangling else" problem
- What is there to **know** about a conditional?
  - evaluates **condition** to a boolean result
  - executes these **instructions** for **true**
  - executes those **instructions** for **false**

What is an Abstract Grammar? (3)

- Abstract grammar for conditional instructions
  - If = Condition Instructions$_{\textbf{true}}$ [Instructions$_{\textbf{false}}$].
  - only the **essential** constituents remain
  - lexical and syntactic details **disappear**
- Concrete grammars ⇒ Abstract grammar
  - If = “if” “(” Condition “)” “{” Instructions “}” [“else” “{” Instructions “}”].
  - If = “if” Condition “:” Instructions [“else” “:” Instructions].
  - If = Condition “?” Instructions [“:” Instructions].
  - C-like, Python-esque, and insane concrete grammars have the **same** abstract grammar!

What is an Abstract Grammar? (4)

- Example: Repetitive instructions
  While = "WHILE" Condition "DO" Instructions "END".
  $\Rightarrow$
  While = Condition Instructions.
- Example: Type declarations
  TypeDecl = "TYPE" { identifier "=" Type }.
  $\Rightarrow$
  TypeDecl = { **Identifier** Type }.
  $\Leftrightarrow$
  TypeDecl = ϵ | **Identifier** Type TypeDecl.
- Example: Write instruction
  Write = "WRITE" Expression.
  $\Rightarrow$
  Write = Expression.

What is an Abstract Grammar? (5)

- Example: Expressions
  Expression = Term {"+" Term}.
  Term = Factor {"*" Factor}.
  Factor = identifier | number | "(" Expression ")".
  $\Rightarrow$
  Expression = Expression Operator Expression.
  Expression = **Identifier** | **Number**.
  Operator = **Plus** | **Times**.
- Concrete grammar is designed for **parsing**
  - avoid ambiguity, model precedence, model associativity, simple parsing algorithms, ...
- Abstract grammar is designed for **representation**
  - once done with parsing, a lot of simplifications can be made

Abstract Syntax Trees

What is an Abstract Syntax Tree? (1)

- Abstract syntax trees (ASTs)
    - **common** form of intermediate representation
    - retains **structural** properties of source language
    - described by **abstract grammars**
    - easily **produced** during parsing
    - easily **traversed** by backends
- For Simple:
    - Declarations $\Rightarrow$ Symbol Table (ST)
    - Instructions, Expressions $\Rightarrow$ AST
    - ST + AST = **validated** representation of input (provided context conditions were enforced)

What is an Abstract Syntax Tree? (2)

- Example: Expression
  “1” “+” “2” “*” “3” **Source**
  $\Rightarrow$
  (E (T (F “1”)) “+” (T (F “2”) “*” (F “3”))) **CST**
  $\Rightarrow$
  (+ (“1”) (* (“2”) (“3”)) **AST**
- Seems convoluted? No!
  - source text is **linear**, without any structure
  - concrete grammars **impose** a structure
    - which can be “hard” to find, many steps
  - abstract grammars **retain** the structure
    - but forget the “ugly” details of how we got it
- In Simple: CST is implicit in parsing procedures!

What is an Abstract Syntax Tree? (3)

- Example: If Instruction
  “IF” “x” “=” “3” “THEN” “x” “:=” “0” “END” **Source**
  $\Rightarrow$
  (If “IF” **CST**
  (Condition (E (T (F “x”))) “=” (E (T (F “3”))))
  “THEN”
  (Instructions (Instruction
  (Assignment “x” “:=” (E (T (F “0”))))
  ))
  “END”)
  $\Rightarrow$
  (If (= (“x”) (“3”)) (:= (“x”) (“3”)) ()) **AST**
- Would you rather remember the CST or the AST?
  - AST has **all** the information without **any** of the complexity!

Representing Abstract Syntax Trees (1)

- Work from the abstract grammar ("specification")
  - kinds of nodes ⇔ left-hand side
  - contents of nodes ⇔ right-hand side
- Example: Instructions

  Instructions = Instruction | Instructions Instruction

  ⇒ (mechanical) ⇒

```
class Instructions {
  Instructions next;
  Instruction inst;
}
```

  ⇒ (with some thinking) ⇒

```
class Instruction {
  Instruction next;
}
```

Representing Abstract Syntax Trees (2)

- An object-oriented design for Instructions:
  - abstract base class `Instruction`
  - derived conrete classes for `Assignment`, `If`, `While`, ... with corresponding members
- Example: If Instructions
  If = Condition Instructions$_{\textbf{true}}$ [Instructions$_{\textbf{false}}$].
  $\Rightarrow$

```
class If extends Instruction {
  Condition cond;
  Instruction true;
  Instruction false; // can be null
}
```

Representing Abstract Syntax Trees (3)

- Design tradeoffs:
  - A single class and a “kind” field
    - prohibits type checking (**of** the compiler)
    - needs extra documentation for “generic” fields
  - Too many classes (one for “+”, one for “*”, . . .)
    - doesn’t really gain anything anymore
    - increases complexity of the data structure
- Traversal of ASTs:
  - decoupling of backend specifics from ASTs
    - can’t put `eval()` or `code()` methods there
  - external traversal easier for class hierarchy
    - the Visitor design pattern

Representing Abstract Syntax Trees (4)

- An object-oriented design for Expressions:
  - abstract base class `Expression`
  - derived conrete classes for `Number`, `Location`, `Binary`, . . .
- For Simple we don't need `Unary`
  - we can use the binary "0 - x" for "-x"
  - works for integers, not for floating point
- Locations model "anonymous variables" (memory)
  - for example arrays: what kind of node is "i[10]"?
  - Location = **Variable** | Index.
    Index = Location Expression.

Generating Abstract Syntax Trees (1)

- Inspired by **attribute grammars**
  - parsing procedures return **subtrees** of the AST
  - the **synthesized** attribute of each non-terminal
- Similar to the **symbol table**
  - parser procedures “collect” all the information
  - all information there? $\Rightarrow$ create the subtree
- Example: Instructions (pseudo-code)

```
def Instructions():
  first = last = Instruction() // store subtree
  while current.kind == ";":
    next = Instruction() // store subtree
    last.next = next // link instructions
    last = next
  return first // return subtree
```

Generating Abstract Syntax Trees (2)

- Notes:
  - although `Instructions()` returns a **list**, we still call it an abstract syntax **tree**
  - `Instruction()` parses whatever instruction there is in the input
  - and returns the appropriate tree, e.g. for an assignment
- Example: While Instructions (pseudo-code)

```
def While():
  match( "WHILE" )
  cond = Condition() // store subtree
  match( "DO" )
  inst = Instructions() // store subtree
  match( "END" )
  return AST.While( cond, inst ) // return subtree
```

Generating Abstract Syntax Trees (3)

- Sometimes **inherited** attributes are needed
  - Factor = identifier Selector | … .
    Selector = { "[" Expression "]" }.
  - Selector: **which** identifier to index from?
  - Factor passes it to Selector as a parameter
- Example: (pseudo-code, no error handling)

```
def Factor():
  if current.kind == "identifier":
    name = current.value
    next()
    value = table.find( name )
    if value.kind == "VAR":
      id = AST.Location( value )
    elif ...
    result = Selector( id )
  elif ...
```

Generating Abstract Syntax Trees (4)

- Example: (pseudo-code, no error handling)

```
def Selector( id ):
  top = id
  while current.kind == "[":
    next()
    expr = Expression()
    match( "]" )
    top = AST.Index( top, expr )
  return top
```

- For `i[10][20][30]` this yields

  (Index
  (Index
  (Index (Loc “i”) (Num “10”))
  (Num “20”))
  (Num “30”))

- Checking context conditions gets a little hairy
  - each expression node should get a type field
  - helps decide if another “[]” pair is okay

Simple AST Transformations

What are AST Transformations? (1)*

- ASTs can be designed in various ways. . .
  - very close to the source language
  - very close to the target language
  - somewhere in between
- Example: Consider `a[2] := 3`
  - Close to source
    (Assign (Index (Var a) (Num 2)) (Num 3))
  - Closer to target
    (Store (+ (Address a) (* (Lit 2) (Size a[])))
    (Lit 3))
  - Really close to target
    (move (add (lea a) (mul (load #2) (load #4))
    (load #3))

What are AST Transformations? (2)*

- We don't have to model **every** source construct
  - store value instead of constant expression
  - store common subexpressions only once
  - transform control-flow instructions
- Good:
  - can simplify the back end
  - can preserve memory during compilation
- Bad:
  - transformation costs compilation time
  - some information easier to derive in backend

Constant Folding (1)*

- If the subtrees of a binary operation are constant
  - the result of the operation is constant as well
- Example: Consider 5+3*4
  - Simple AST: (+ (5) (* 3 4))
  - Folded AST: (17)
- We can either
  - build the simple AST and then transform
  - or build the folded AST right away

Constant Folding (2)*

- Where binary operation nodes are created
  - check if subtrees are constant
  - evaluate directly if they are
- Example: Pseudo-code for parsing procedure

```
def Expression():
  t1 = Term()
  while current.kind in ["+","-"]:
    op = current.kind
    next()
    t2 = Term()
    if constant(t1) and constant(t2):
      if op == "+":
        t1 = Number(t1.val+t2.val)
      else:
        t1 = Number(t1.val-t2.val)
    else:
      t1 = Binary(op, t1, t2)
    return t1
```

Common Subexpressions (1)*

- If the subtrees of a binary operation
  - have **identical** structure
  - we could store them only once
- Example: Consider `(a+b)*(a+b)`[7]
  - Simple AST: (* (+1 a b) (+2 a b))
  - Better "AST": (* (+1 a b) (+1 a b))
- Results in a DAG: Directed Acyclic Graph
  - common subexpressions are identified
  - need to be evaluated only once
  - except when there are side-effects!

[7]Since I can't draw the tree or dag, I give the nodes numbers...

Common Subexpressions (2)*

- Where binary nodes are created
  - check for identical node created before
  - return that node instead of a new one[8]
- Example: Pseudo-code for parsing procedure

```
def Expression():
  t1 = Term()
  while current.kind in ["+","-"]:
    op = current.kind
    next()
    t2 = Term()
    t1 = Binary(op, t1, t2)
    // return t1 again if not found
    t1 = archive.find(t1)
    return t1
```

- Common subexpression elimination is usually done in the backend since it needs detailed analysis...

[8]The design pattern **Factory** would be useful here...

Control Flow Transformations (1)*

- Example: Transforming FOR into WHILE

  ```
  FOR i := 1 TO 10 DO x[i] := i END
  ```

  $\Rightarrow$

  ```
  l := 1; u := 10;
  WHILE l <= u do
    x[l] := l
    l := l + 1
  END
  ```

- While parsing the FOR we
  - collect all the necessary parts, but generate subtrees for the WHILE
- Simplifies backend, but we lose information, e.g. that a FOR is "properly bounded"

Control Flow Transformations (2)*

- Example: Transforming `WHILE` into `IF`/`REPEAT`

```
WHILE condition DO instructions END
```

$\Rightarrow$

```
IF condition THEN
  REPEAT
    instructions
  UNTIL NOT condition END
END
```

- Simplifies backend, but we blow up the AST
  - like for most other things, the tradeoff can be difficult to make

Visualization

CS 152: Compiler Design

Introduction to Interpreters

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today's Lecture

Introduction

What is an Interpreter?

- Compiler vs. Interpreter
  - There is no clear distinction!
    - a processor “interprets” machine language
    - a compiler “interprets” parts of a program (e.g. when folding constants)
    - Just-in-time (JIT) compilers, adaptive compilers, . . .
  - Where do the results come from?
    - compilers generate “object files” that need a separate “interpreter” to produce results
    - interpreters process some intermediate representation and produce results

Interpreting Simple (1)

- Symbol Table (ST) and Abstract Syntax Tree (AST)
  - provide a **complete** representation of a Simple program
    - everything we need to execute a program
  - are already **checked** syntactically and semantically
    - exception: array indices out of array bounds
- Strategy: Traverse AST and “simulate” program.
  - traverse in post-order and from left to right
  - simulate what a stack machine would do

Interpreting Simple (2)

- Constant expressions:
  - constant node: push the value on our stack
  - binary node: pop two values, perform operation, push result
- Example: 1+2*3 ⇒ (+ 1 (* 2 3))
  start traversal at "+" with empty stack
  visit "1" ⇒ push "1" on stack
  visit "*"
  visit "2" ⇒ push "2" on stack
  visit "3" ⇒ push "3" on stack
  back at "*" ⇒ push(pop()*pop())
  back at "+" ⇒ push(pop()+pop())
  end traversal with "7" on stack

Interpreting Simple (3)

- The interpreter could look like this (pseudo-code):

```
stack = []
...
def Interpret( ast ):
  if isinstance( ast, Number ):
    stack.push( ast.value )
  elif isinstance( ast, Binary ):
    Interpret( ast.left )
    Interpret( ast.right )
    if ast.operator == "+":
      stack.push( stack.pop() + stack.pop() )
    elif ast.operator == "*":
      stack.push( stack.pop() * stack.pop() )
    elif...
    ...
  elif ...
  ...
  elif isinstance( ast, Write ):
```

Environments

Interpreting Simple (4)

- Handling variables:
  - we need to simulate the "data memory"
  - to enable access to the **current** value
  - to enable **assigning** new values
- Environments:
  - similar to the symbol table (ST)
  - but maps names to "**boxes**" (variables, memory, storage) which store values
  - while ST maps names to **meanings**
  - can be generated from ST

Interpreting Simple (5)

- For Simple:
  - only **one** environment necessary
  - universe does not have variables in it
  - but we need “scopes” for record types
- Languages with nested scopes:
  - create new environments at run-time
  - whenever a scope is entered dynamically
- Example: a := b + c ⇒
  (:= (Var a) (+ (Var b) (Var c)))
  lookup a and push “box” (“lvalue”)
  lookup “boxes” for b and c, push values (“rvalue”)
  add values for b and c, push result
  store result in “box” for a

Interpreting Simple (6)

- The interpreter could look like this (pseudo-code):

```
stack = []
environment = {}
...
def Interpret( ast ):
  ...
  elif isinstance( ast, Variable ):
    location = environment[ast.name]
    stack.push( location )
  elif isinstance( ast, Assign ):
    Interpret( ast.location )
    Interpret( ast.expression )
    val = stack.pop()
    loc = stack.pop()
    loc.set( val )
  elif ...
  ...
```

- Binary (earlier slide) needs to be modified to handle locations

Representing Current Values (1)

- Symbol Table (ST) vs. Environment (ENV)
  - ST: maps **names** to **meanings**
  - ENV: maps **names** to **boxes** containing current **values**
- For STs, we represented meanings (compile-time)
  - by instances of some class hierarchy
  - e.g. Constant, Variable, Type, ArrayType, ...
- For ENVs, we need to represent values (run-time)
  - probably again by instances of certain classes

Representing Current Values (2)

- What kinds of values do we have at run-time?
  - simple Variables ⇒ IntegerBox
  - array Variables ⇒ ArrayBox
  - record Variables ⇒ RecordBox
- What do we store in IntegerBoxes?
  - Just an integer representing the current value!
- What operations do we need on IntegerBoxes?
  - we need to **get** the current value
  - we need to **set** a new value[9]

[9] We already performed the checks to ensure that a program never assigns to a constant...

Representing Current Values (3)

- What do we store for ArrayBox?
  - for ARRAY 10 OF INTEGER
    - we need 10 IntegerBox instances
  - for ARRAY 20 OF ARRAY 10 OF INTEGER
    - we need 20 ArrayBox instances
    - with 10 IntegerBox instances each
  - Each ArrayBox instance stores a list of “other boxes”!
- What operations do we need on ArrayBox?
  - we need to **index** the element “box” at a certain position within the array
- Similar reasoning leads to RecordBox design...

Representing Current Values (4)

- Resulting class hierarchy for boxes:

```
class Box:
  pass

class IntegerBox( Box ):
  def __init__( self, value=0 ):
    self.__value = value
  def get( self ):
    return self.__value
  def set( self, value ):
    self.__value = value

class ArrayBox( Box ):
  def __init__( self, list ):
    self.__length = len( list )
    self.__boxes = list
  def index( self, offset ):
    if 0 <= offset < self.__length:
```

Building Environments (1)

- Base case: For each variable of type INTEGER
  - generate a new IntegerBox initialized to 0
  - insert into environment under appropriate name
- Inductive case: For each variable of type ARRAY x OF y
  - generate a new ArrayBox referring to x Boxes representing y
  - insert into environment under appropriate name
  - similar for RECORD variables...

Building Environments (2)

- Recursive procedure for variables:

```
def make_box( type ):
  if isinstance( type, IntegerType ):
    // base case
    return IntegerBox()
  elif isinstance( type, ArrayType ):
    // inductive case
    list = []
    for i in range( 0, type.length() ):
      list.append( make_box( type.type ) )
    return ArrayBox( list )
  else: // isinstance( type, RecordType )
    ...
```

- Can be done using a visitor as well. . .

More on Interpreting Simple

Interpreting Conditions

- Conditions occur in IF and REPEAT instructions
  - consist of a comparison operator
  - and two expressions to be compared
- Strategy for Conditions:
  - evaluate the left and right expressions
    - values are the two top stack elements
  - compare the values according to the condition
    - yields a truth value in our implementation language
  - encode the truth-value and push it on the stack
    - e.g. 0 for false, 1 for true

Interpreting Repeat (1)

- For REPEAT instructions we simulate control flow
  - interpret body (always at least once)
  - if condition is false, interpret body **again**
  - if condition is true, interpret instruction **following** REPEAT
- Strategy for REPEAT:

```
do {
  "interpret body, do whatever
    needs doing";
  "evaluate condition,
    yields 0 or 1 on stack";
} while "0 on top of stack";
"interpret next instruction";
```

Interpreting Repeat (2)

- The interpreter could look like this (pseudo-code):

```
stack = []
environment = {}
...
def Interpret( ast ):
  ...
  elif isinstance( ast, Repeat ):
    flag = 0
    while not flag:
      Interpret( ast.body )
      Interpret( ast.condition )
      flag = stack.pop()
  elif ...
  ...
```

- IF instructions work almost exactly like this as well!

Famous Last Words

- Several things were not discussed in detail (would spoil the fun), but here are some hints:
  - Handling assignments between arrays and records
    - All values need to be copied "by hand", you can't just assign a pointer around...
  - Handling indexing for arrays
    - Evaluate the index expression and then access the ArrayValue object, pushing whatever result on the stack...
  - Interpreting a sequence of instructions
    - You have to follow the "next" pointer for instructions, either using a "big loop" in the Interpret() procedure, or by recursion...

CS 152: Compiler Design

Introduction to Code Generation

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

October 1, 2004

Today's Lecture

Introduction

What is Code Generation? (1)

- The “big” picture again:
  - Source Text $\rightarrow \boxed{\text{Compiler}} \rightarrow$ Target Code
  - $i := i + 1 \rightarrow \boxed{\text{Compiler}} \rightarrow$ addi #1, -24(a5)
- Code Generation: The “last” remaining step. . .
  - from the intermediate representation (IR)
  - to a sequence of machine instructions
- The “little” picture for the remaining lectures:
  - IR (= ST+AST) $\rightarrow \boxed{\text{Generator}} \rightarrow$ Instructions
  - (:= i (+ i 1)) $\rightarrow \boxed{\text{Generator}} \rightarrow$ addi #1, -24(a5)

What is Code Generation? (2)

- The most important criteria for code generation:
  - preserve the “meaning” of the source program (correctness)
  - use target machine resources in an “optimal” way (efficiency)
- Further considerations:
  - code generation itself must be efficient, otherwise nobody uses the compiler
  - obvious mapping from source constructs to target instructions, simplifies debugging
  - modularize code generator itself to ease porting, maybe use a code generator generator

What is Code Generation? (3)

- Two basic tasks for code generation:
  - **storage allocation:** Where do things go?
  - **instruction selection:** How do we do things?
  - both dependent on specific target machine
- Storage allocation:
  - determine where variables, constants, etc. go
  - which "address" in memory or which register
- Instruction selection:
  - determine which target instruction to use
  - also which addressing mode

What is Code Generation? (4)

- Example: Consider i := i + 1 again
  - Where in the store is the value of "i"?
    - . . . , $\boxed{\text{-24(a5)}}$
    - at offset -24 relative to register a5
  - What do we have to do?
    - $\boxed{\text{add}}$ . . . , -24(a5)
    - the "+" operator yields an "add" instruction
  - Where is value "1", what do we have to do?
    - $\boxed{\text{addi \#1}}$, -24(a5)
    - we add a constant, so we need to choose the "right kind" of add

What is Code Generation? (5)

- Instruction Selection ⇔ Storage Allocation
  - the two tasks are **related**
  - **both** depend on the target machine
- Target machine characteristics (see Hennesey, Patterson: Computer Architecture):
  - available instructions and their restrictions
  - available memory areas
  - available registers
  - available addressing modes
  - relative size/speed of instructions
  - . . .

Code Patterns

Code Patterns (1)

- Before we can generate code we need to
  - study the target machine in detail
  - develop conventions for using it
- Strategy: Identify code patterns!
  - Postulate assumptions for storage allocation
  - Examine source language constructs
  - Translate source constructs to target patterns
  - Iterate until satisfied!
- Code patterns should be general
  - to allow combination when translating compound source constructs

Code Patterns (2)

- Example: CISC architecture, e.g. Motorola 68K
  - two sets of 8 registers, one for data, one for addresses
  - lots of addressing modes, e.g. indirect, indexed, immediate
- Example: Decisions for storage layout (simplified)
  - put temporaries into registers, variables into memory, constants into immediate instructions
  - address global variables relative to register a5
  - address local variables relative to register a7
- Now we can develop code patterns for source language constructs. . .

Code Patterns (3)

- Example: Variables as rvalues

  a ⇒ move offset_a(a5), dx
- Example: Expressions involving variables

```
a + b  ⇒  move   offset_a(a5),   dx
          move   offset_b(a5),   dy
          add    dx,             dy
```

- **But:** We could do **better** on a CISC!

```
a + b  ⇒  move   offset_a(a5),   dx
          add    offset_b(a5),   dx
```

- Change the storage allocation:
  - temporaries into registers **only when needed**
  - of course this is more complex to implement...

Code Patterns (4)

- Example: Expressions involving constants as well

```
a + b + 2  ⇒  move    offset_a(a5),   dx
              add     offset_b(a5),   dx
              addi    #2,             dx
```

- Example: Simple assignments between variables

```
a := b  ⇒  move    offset_b(a5),   of
```

- Example: Combine the patterns

```
c := a + b  ⇒  move    offset_a(a5),   dx
               add     offset_b(a5),   dx
               move    dx,             offset_c(a5)
```

- **But:** We could do **better** on a CISC!
  - . . . this can go on for a long time until we have a satisfactory collection of patterns. . .

Code Patterns (5)

- Example: WHILE condition DO instructions END
  - evaluate condition, execute instructions, as long as condition is true
- Example: Pattern for code(while) =
  1. code(condition)
  2. if false goto $\boxed{?}$
  3. code(instructions)
  4. goto 1
  5. . . . things after while. . .

  The target of “goto ?” is **patched** to $\boxed{5}$ when that address becomes known. . .

Code Patterns (6)

- Example: A concrete WHILE translated...

```
...
WHILE a < 10 DO
  a := a + 1
END
...
```

$\Rightarrow$

```
$0FFE   ...
$1000   cmpi #10, offset_a(a5)
$1002   bge $xxxx
$1004   addi #1, offset_a(a5)
$1006   jmp $0000
$1008   ...
```

Then we patch address $1008 into the instruction at address $1002...

Tree Matching

- General code generation from AST (see Appel):
  - think of code patterns as “tiles” we can “move” over the tree
  - when we find a “tile” that “matches” some part of the tree, we select the instructions for it
  - the goal is to have enough “tiles” to “cover” the whole tree (not necessarily in a unique way)
- This idea is useful
  - to think about the code generation process
  - as an approach for code generator generators
- We will only “match” one node at a time
  - much simpler, good enough for our purposes

Target Machine Restrictions

- The Simple language definition did not mention
  - maximum values for constants and variables
  - maximum size of all variables in a program
- On the target machine, certain restrictions exist
  - maximum word-length and instruction set
  - maximum available memory for variables
- To keep the frontend "pure," the backend has to
  - traverse the intermediate representation
  - generate error messages for violations

Storage Allocation

Storage Allocation (1)

- Compute the memory addresses of
  - whatever we need an address for at runtime!
  - e.g. variables, constants, procedures, . . .
- Depends on source language **and** target machine
  - No constants? No need for their addresses either. . .
  - Immediate mode? No need for addresses for constants. . .
- Strategy: Traverse symbol table and add addresses
  - this works great for relative addresses
  - which are all we need

Storage Allocation (2)

- Absolute Addressing:
  - compute a final physical address for everything
  - not useful if there is other things already
  - operating system, other programs, etc.
- Relative Addressing:
  - compute offsets from a base address obtained at runtime
  - allocate memory from operating system, put pointer in register
  - for procedures and local variables: allocate on the stack

Storage Allocation (3)

- Activation Records:
  - recursive procedures, nested procedures, etc.
  - each call allocates memory on stack for locals
  - access to static and dynamic scope
    - dynamic link: points to start of previously active scope
    - static link: points to start of statically surrounding scope
- For Simple we don't need any of this, but you might want to review it in the book anyway. . .

Storage Allocation (4)

- Example: Sequential allocation, 16-bit words, byte addressed store

```
VAR a: INTEGER;
    b: ARRAY 10 OF INTEGER;
    c: INTEGER;
```

$\Rightarrow$

Offset	Content
0	a
2	b[0]
4	b[1]
6	b[2]
8	b[3]
10	b[4]
12	b[5]
14	b[6]
16	b[7]

Storage Allocation (5)

- Example: Sequential allocation, 16-bit words, byte addressed store

  ```
  VAR a: ARRAY 3 OF ARRAY 2 OF INTEGER;
  ```

  $\Rightarrow$

Offset	Content
0	a[0,0]
2	a[1,0]
4	a[2,0]
6	a[0,1]
8	a[1,1]
10	a[2,1]

or

Offset	Content
0	a[0,0]
2	a[0,1]
4	a[1,0]
6	a[1,1]
8	a[2,0]
10	a[2,1]

- row-major vs. column-major allocation (generalizes to more dimensions as well)
- For Simple it is natural to choose the “inner” array to be allocated continuously. . .

Data Alignment (1)

- Processor restrictions on data access
  - for example 32-bit RISC, byte-addressed store
  - accessing a 16-bit word is only possible at even addresses
  - accessing a 32-bit long is only possible at addresses divisible by 4
- Storage allocation
  - needs to account for these restrictions
  - either by "padding" (really simple)
  - or by reordering (almost as simple)

Data Alignment (2)

- Example: char = 1 byte, int = 2 byte, long = 4 byte

```
VAR a: CHAR; b: INTEGER; c: LONG; d: CHAR;
```

$\Rightarrow$

	0	1	2	3
0	a	b	b	c
4	c	c	c	d

both b and c not accessible

$\Rightarrow$

	0	1	2	3
0	a		b	b
4	c	c	c	c
8	d			

padding

$\Rightarrow$

	0	1	2	3
0	c	c	c	c
4	b	b	a	d

reordering

The VMICS Processor

What is VMICS?

- The VMICS processor ("vee-mix")
  - is a simple 16-bit stack architecture
  - is our target for compiling Simple
- The VMICS package (from course web site)
  - includes more detailed documentation
    - on the architecture itself
    - on the instruction set
    - on the object file format
  - includes a VMICS simulator written in Python
  - includes several example programs

Basic Architecture (1)

- Three distinct memory areas
  - 64kB (= 65536 bytes) data memory
  - 64kB (= 65536 bytes) code memory
  - stack memory, unspecified size
- Stack architecture
  - no registers, all temporaries on the stack
  - instructions push and pop values as needed
- Basic operation
  - load object file into code memory
  - start execution at address 0
  - last instruction must be “halt” else error

Basic Architecture (2)

- Storage allocation
  - variables are allocated in data memory
  - data memory is byte-addressed
  - required size depending on type:
    - INTEGER ⇒ 2 bytes
    - ARRAY x OF y ⇒ x * size(y) bytes
    - RECORD ... END ⇒ sum of fields
- Machine restrictions
  - constants (declared, literal) must fit in 16 bit
  - variables (declared) must fit in 64 kB
- Data alignment is not necessary...

Some Instructions (1)

- Instruction formats
  - A: $\boxed{\text{code}}$ 1 byte
    1 byte instruction, no value
  - B: $\boxed{\text{code}}\boxed{\text{value}}\boxed{\text{value}}$ 3 byte
    1 byte instruction, 2 bytes for 16-bit value
- Note: Instruction formats for CISC and RISC
  - CISC: many different formats, length from 1 to over 20 bytes not uncommon
  - RISC: few different formats, length mostly 4 bytes, easier to decode
  - Therefore RISC code generation is usually more regular, with less special cases to consider.

Some Instructions (2)

- Categories of instructions:
  - stack manipulation, input/output, memory
  - arithmetic, conditional and unconditional jumps

Name	**Code**	**Value**	**Description**
halt	\$00	—	stop execution successfully
push	\$10	value	push value on stack
dup	\$12	—	duplicate top of stack
out	\$20	—	print pop()
in	\$21	—	push(input)
add	\$30	—	push(pop() + pop())
sub	\$31	—	push(pop() - pop())
mul	\$32	—	push(pop() * pop())
load	\$40	—	push(memory[pop()])
store	\$41	—	memory[pop()] := pop()
jump	\$50	address	pc := address
jneg	\$51	address	if pop() $<$ 0 then pc := address
jeql	\$52	address	if pop() = 0 then pc := address
jpos	\$53	address	if pop() $>$ 0 then pc := address

Object File Format

- Object file is a binary file containing
  - size information for data and code memory
  - the actual instructions (code) for a program
- Can be specified in a variant of EBNF:

```
File = Header Code Trailer .
Header = $FF version:1 dataSize:2 codeSize:2 .
Code = $FE {Instruction} .
Instruction = operation:1 [value:2].
Trailer = $F0 .
```

  Literals “$xx” mean that this byte has to occur in the file. The form “x:y” means that y bytes are used to store the information “x”.
- For details see the VMICS package!

Code Generation Examples

Simple Assignments

- Consider the following source program:

```
PROGRAM X;
  VAR a: INTEGER;
BEGIN
  a := 47
END X.
```

- Storage allocation puts “a” at address 0, occupying bytes 0 and 1, for a data size of 2 bytes.
- What is the code we need for this?

0: push 47	push **value** 47
3: push 0	push **address** of “a”
6: store	put 47 into “a”
7: halt	stop execution

- Code size 8 bytes, code memory bytes 0 to 7.

Expressions (1)

- Consider the following source program:

```
PROGRAM X;
  VAR a, b: INTEGER;
BEGIN
  b := 10; a := b + 3; WRITE a
END X.
```

- Storage allocation puts “a” at address 0 and “b” at address 2, for a data size of 4 bytes.
- Notes on the code (see next slide):
  - b := 10 $\Rightarrow$ instructions @ 0–6
  - b + 3 $\Rightarrow$ instructions @ 7–14
  - a := ... $\Rightarrow$ instructions @ 15–18
  - WRITE a $\Rightarrow$ instructions @ 19–23

Expressions (2)

- What is the code we need for this?

0: push 10	push **value** 10
3: push 2	push **address** of "b"
6: store	put 10 into "b"
7: push 2	push **address** of "b"
10: load	push **value** of "b"
11: push 3	push **value** 3
14: add	add the two, result on stack
15: push 0	push **address** of "a"
18: store	put result into "a"
19: push 0	push **address** of "a"
22: load	push **value** of "a"
23: out	print top of stack
24: halt	stop execution

- Code size 25 bytes, code memory bytes 0 to 24.

Organization of the Code Generator

Introduction

- The code generator transforms
  - our intermediate representation (ST+AST)
  - into an object file for VMICS
- The basic approach is similar to the interpreter
  - we traverse the AST in a certain order
  - we emit VMICS instructions whenever we can
- Two questions to answer:
  - how do we represent VMICS instructions
  - how do we traverse the AST

Arrays

- Use an array of bytes (see Wirth)
  - to represent the sequence of instructions
  - append new instructions as we generate them
- Maintain a program counter
  - to keep track of where we are in the array
  - to perform backpatching (needed for some jumps)
- Tradeoff:
  - very close to the eventual object file
  - pretty "low-level" approach

Lists (1)

- Model VMICS instructions with a simple class

```
class Instruction {
  int code; // the operation code
            // e.g. 0x30 for "add"
  int value; // an optional value
             // e.g. address for "jump"
  Instruction next; // linear list
}
```

- Maintain global instruction list and program counter

```
def emit( code, value=None ):
  inst = Instruction( code, value )
  list.append( inst )
  if value == None:
    pc = pc + 1
  else:
    pc = pc + 3
  return inst
```

Lists (2)

- Points to note:
  - provides a little more structure than arrays, but still pretty "low-level" compared to ST and AST
  - we return the instruction generated for backpatching, see later examples
  - we maintain the program counter in form of target machine addresses for backpatching
  - using a class for instructions allows for easy printing, useful for debugging
- Want more structure?
  - insert pointers to other instruction lists instead of addresses
  - **not** recommended for Simple, it'll cost you too much time...

Code Generation Procedures

Introduction

- Traverse the AST using
  - a recursive function with a big if instruction
  - the Visitor pattern (if you know how to)
- Basic structure:

```
def code( ast ):
  if isinstance( ast, Number ):
    // emit code for that case
  elif isinstance( ast, Binary ):
    // emit code for that case
  elif ...
  ...
  else:
    raise "Oops, illegal node!"
```

Has to be recursive, e.g. for binary operations...

Numbers and Variables

- Conventions for the resulting code
  - Numbers: push integer value on the stack
  - Variables: push address on the stack
    - We don't know if we need the actual value (rvalue) or address (lvalue)
    - If we push the address, we can still "load" the value later
- Relevant cases:

```
def code( ast ):
  ...
  elif isinstance( ast, Number ):
    emit( PUSH, ast.value )
  elif isinstance( ast, Variable ):
    emit( PUSH, ast.object.address )
  elif ...
  ...
```

Binary Operations

- Conventions for the resulting code
  - Generate code for the left and right side
    - For locations "load" the actual values
  - Generate code to perform the operation
- Relevant cases:

```
def code( ast ):
  ...
  elif isinstance( ast, Binary ):
    code( ast.left )
    if isinstance( ast.left, Location ):
      emit( LOAD )
    code( ast.right )
    if isinstance( ast.right, Location ):
      emit( LOAD )
    if ast.operator == "+":
      emit( ADD )
    elif ast.operator == "*":
      emit( MUL )
    elif ...
    ...
  elif ...
```

Conditions (1)

- Note: This is **different** from the discussion on the previous slide set!
- We need to generate code for conditions
  - in a uniform manner
  - regardless whether they are part of IF or WHILE
- We need to map
  - six kinds of conditions:
    $l = r$, $l \# r$, $l < r$, $l > r$, $l \leq r$, $l \geq r$
  - to four kinds of instructions:
    sub, jneg, jeql, jpos
  - nothing else in VMICS that could be used!

Conditions (2)

- Conventions for conditions
  - code generated for a condition puts TRUE (i.e. 1) or FALSE (i.e. 0) on the stack
  - this is what code generated for IF and WHILE relies on
- Examine conditions and available instructions:
  - l $=$ r $\Rightarrow$ is TRUE if l-r $= 0$
  - l $\#$ r $\Rightarrow$ is TRUE if l-r $\neq 0$
  - l $<$ r $\Rightarrow$ is TRUE if l-r < 0
  - l $\geq$ r $\Rightarrow$ is TRUE if l-r $\not< 0$
  - l $>$ r $\Rightarrow$ is TRUE if l-r > 0
  - l $\leq$ r $\Rightarrow$ is TRUE if l-r $\not> 0$

Conditions (3)

- Common tasks to generate code for conditions
  - generate code to evaluate left and right side
  - if these are locations, “load” actual value
  - subtract right value from left value
- Tasks for the particular comparison
  - generate code that pushes TRUE or FALSE
  - depending on the difference just calculated

Conditions (4)

- Basic structure for common tasks:

```
def code( ast ):
  ...
  elif isinstance( ast, Condition ):
    // code for left and right sides
    code( ast.left )
    if isinstance( ast.left, Location ):
      emit( LOAD )
    code( ast.right )
    if isinstance( ast.right, Location ):
      emit( LOAD )
    // calculate the difference
    emit( SUB )
    // handle the actual comparison
    if ast.comparison == "=":
      // emit code for that case
    elif ast.operator == "#":
      // emit code for that case
```

Conditions (5)

- Deriving the code pattern for equal (“=”):
  - if difference is 0 ⇒ push 1
  - if difference is not 0 ⇒ push 0
- Therefore we need something like this:

```
   ...instructions before condition...
   ...instructions for left and right...
   sub
   jeql x
   push 0
   jump y
x: push 1
y: ...instructions following condition...
```

  For not equal (“#”) the “pushes” are “inverted”.
- This code sequence will do what we need, but how do we generate it?

Conditions (6)

- Code generation for equal (“=”):

```
def code( ast ):
  ...
  elif isinstance( ast, Condition ):
    ...see earlier slide...
    if ast.comparison == "=":
      // remember the "true" jump for fixup
      true = emit( JEQL, 0 )
      // push FALSE
      emit( PUSH, 0 )
      // remember the "false" jump for fixup
      false = emit( JUMP, 0 )
      // fixup the "true" jump
      true.value = pc
      // push TRUE
      emit( PUSH, 1 )
      // fixup the "false" jump
      false.value = pc
```

Conditions (7)

- Code generation for not equal (“#”):

```
def code( ast ):
  ...
  elif isinstance( ast, Condition ):
    ...see earlier slide...
    elif ast.comparison == "#":
      // remember the "false" jump for fixup
      false = emit( JEQL, 0 )
      // push TRUE
      emit( PUSH, 1 )
      // remember the "true" jump for fixup
      true = emit( JUMP, 0 )
      // fixup the "false" jump
      false.value = pc
      // push FALSE
      emit( PUSH, 0 )
      // fixup the "true" jump
      true.value = pc
```

Conditions (8)

- This is why returning the instruction object from emit() makes sense!
  - for a "forward" jump we have to fix the address as soon as we know it
- The remaining conditions can be handled the same way, just with "jneg" and "jpos" instead of "jeql".
  - some optimization is possible, but I just programmed the six cases out completely...
- Now that we are done with conditions, both IF and WHILE become really easy...

While Loops

- For a WHILE we have
    - a forward jump to skip the body
    - a backward jump for the loop itself

```
def code( ast ):
  ...
  elif isinstance( ast, While ):
    // remember address where loop starts
    top = pc
    // emit code for the condition
    code( ast.condition )
    // remember "false" jump for fixup
    false = emit( JEQL, 0 )
    // emit code for the body
    code( ast.body )
    // jump back to the top of the loop
    emit( JUMP, top )
    // fix the "false" jump
    false.value = pc
  elif ...
```

Assignments (1)

- For assignments we have three cases
  - INTEGER value to INTEGER variable
  - ARRAY value to ARRAY variable
  - RECORD value to RECORD variable
- First case is easy to handle:

```
def code( ast ):
  ...
  elif isinstance( ast, Assign ):
    if isinstance( ast.location.type, IntType ):
      // integer assignment
      code( ast.expression )
      if isinstance( ast.expression, Location ):
        emit( LOAD )
      code( ast.location )
      emit( STORE )
    else:
      // array assignment
      ...
```

Assignments (2)

- For assignments between ARRAYs (or RECORDs)
  - all array elements from the source array
  - must be assigned to
  - all array elements in the destination array
- We "dream" of a VMICS instruction that
  - copies a block of *n* words
  - from a source address *s*
  - to a destination address *d*
- But VMICS does **not** have such an instruction!
  - And you are **not** allowed to add it!

Assignments (3)

- A simple solution:
  - we know the address of both arrays, we know the length of both arrays
  - we could just emit "enough" push/load/push/store sequences to make the copy
- The **big** problem:
  - it takes 8 bytes of code to copy 2 bytes of data
  - for **one** assignment between **two 16 KB** arrays we already run out of code memory!
- A better solution:
  - generate a **loop** for an array assignment
  - the problem then is **how** to do that. . .

Assignments (4)

- The loop we need looks like this:

```
i = 0
while i < n:
  memory[d+i] = memory[s+i]
  i = i + 2
```

- It is **not** easy to translate this into VMICS code!
  - We have to maintain 3 values on the stack, only 2 of which are accessible at any given moment.
- A possible trick to use:
  - Always allocate an additional INTEGER variable (e.g. at address 0) for "i".
  - That is a "bad hack" and costs 2 bytes that should be available to the user, but it is certainly simpler...

Indexing

- Indexing (and field selection) will not be discussed here, you are on your own!
- However, here are some hints:
  - we know the address and size of each array
  - to index an one-dimensional array
    - generate code to evaluate the index
    - generate code to check for the array bounds
    - multiply the index with the size of a word
    - add that to the base address of the array
  - to index multi-dimensional arrays
    - you also have to take care of the size of the “outer” array. . .

CS 152: Compiler Design

Summary: The Simple Compiler

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

November 19, 2004

Today's Lecture

Introduction

Language

Architecture

Extensions

Outlook

CS 152: Compiler Design

Flow Graphs and Basic Blocks

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

November 22, 2004

Today's Lecture

Introduction

Straightline Code

Basic Blocks

Extended Basic Blocks

Traces

Outlook

CS 152: Compiler Design

Static Single Assignment Form

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

November 24, 2004

Today's Lecture

Introduction

Values versus Variables

Straightline Code

Structured Code and Phi Functions

Outlook

CS 152: Compiler Design

Graph Coloring Register Allocation

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

November 26, 2004

Today's Lecture

Introduction

Register Pressure

Register Allocation

Outlook

CS 152: Compiler Design

Instruction Scheduling

Department of Computer Science & Engineering
University of California, Riverside

Peter H. Fröhlich
phf@cs.ucr.edu

November 29, 2004

Today's Lecture

Introduction

List Scheduling

Outlook