Lecture 7

Compilers and interpreters

Iztok Savnik, FAMNIT

April, 2023.

Slides

Jan Karabaš, Lecture, Compilers and interpreters, Programming II, FAMNIT, 2015.

Torben Aegidius Mogensen, Basics of Compiler design, University of Copenhagen, 2010.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, Compilers, 2nd Edition, Addison Wesley, 2007 (Red Dragon Book).

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Bits of history

- 30-40’s: one-purpose computers, hard-wired programs and data (ZuSe, ENIAC);
- 1952: G. Hopper, A-0 compiler, rather linker and loader than compiler;
- 1952: A. Glennie, Mark 1, autocode, first compiler in the modern sense;
- 1957: J. Backus, Fortran, first ‘real’ compiler, BNF, code better than the same written in assembly code;
- 1958: Bauer et al., Algol 58, first ‘modern’ compiler;
- 60’s: IBM’s crosscompilers for IBM 7xx architecture (e.g. from UNIVAC)
- 1962: Hart & Levin: LISP, first self-hosting compiler;

Bits of history

- 1968-1972: UNIX, C, lex, yacc; also make, sh, grep;
- 70's: Pascal, Niklaus Wirth, Zürich, CDC Pascal
- 70's: SmallTalk, first just-in-time compilers;
- 80's: Erlang, C++, Ocaml, Modula, Ada, Perl, Objective-C;
- 90’s: Haskell, Python, Ruby, Java;
- 00’s: C#, Scala, F#, Go, mobile architectures, functional;
- 10’s: TypeScript, Julia, Raku, Swift; data, functional, object-oriented.

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Why compilers? Why interpreters?

- Source code:
  - Program is mostly represented by an ordinary text file, which cannot be directly executed by target machine.
- Executable code:
  - The form of the program which can be directly executed by a computer.
- Usual targets of compilers/interpreters
  - machine code, in fact a form of intermediate code;
  - bytecode;
  - ‘runnable’ file;
  - other programming language;
  - cross compiling.

Targets

- Machine code:
  - bound to the particular architecture and the host operating system, C, C++ (gcc), Fortran (gfortran), Pascal (fpc), OCaml (ocamlc),. . .
- Bytecode:
  - prepared for the running in the environment of the specific virtual machine, hence (mostly) machine and operating system independent, Java (javac), Python (python), Erlang (erl), C# (.NET), but also C and C++ (LLVM or .NET). . .
- Runnable source:
  - the source code is directly executed, Unix shells (bash, csh, zsh), BASIC (many, but not Visual Basic), Web (tangle);

Targets

- Programming languages:
  - Fortran to C (f2c), Python to C (cpython);
- Cross-compilers:
  - applications for mobile phones and other gadgets are compiled in this way, ObjC for iOS, C++ for Android, Windows Phone, NVida CUDA, C for OpenCL. . .

Structure and action of a compiler

The job of a compiler usually consists of

i. Front-end:

Analysis

- Line reconstruction,
- Lexical analysis,
- Preprocessing,
- Syntax analysis,
- Semantic analysis;

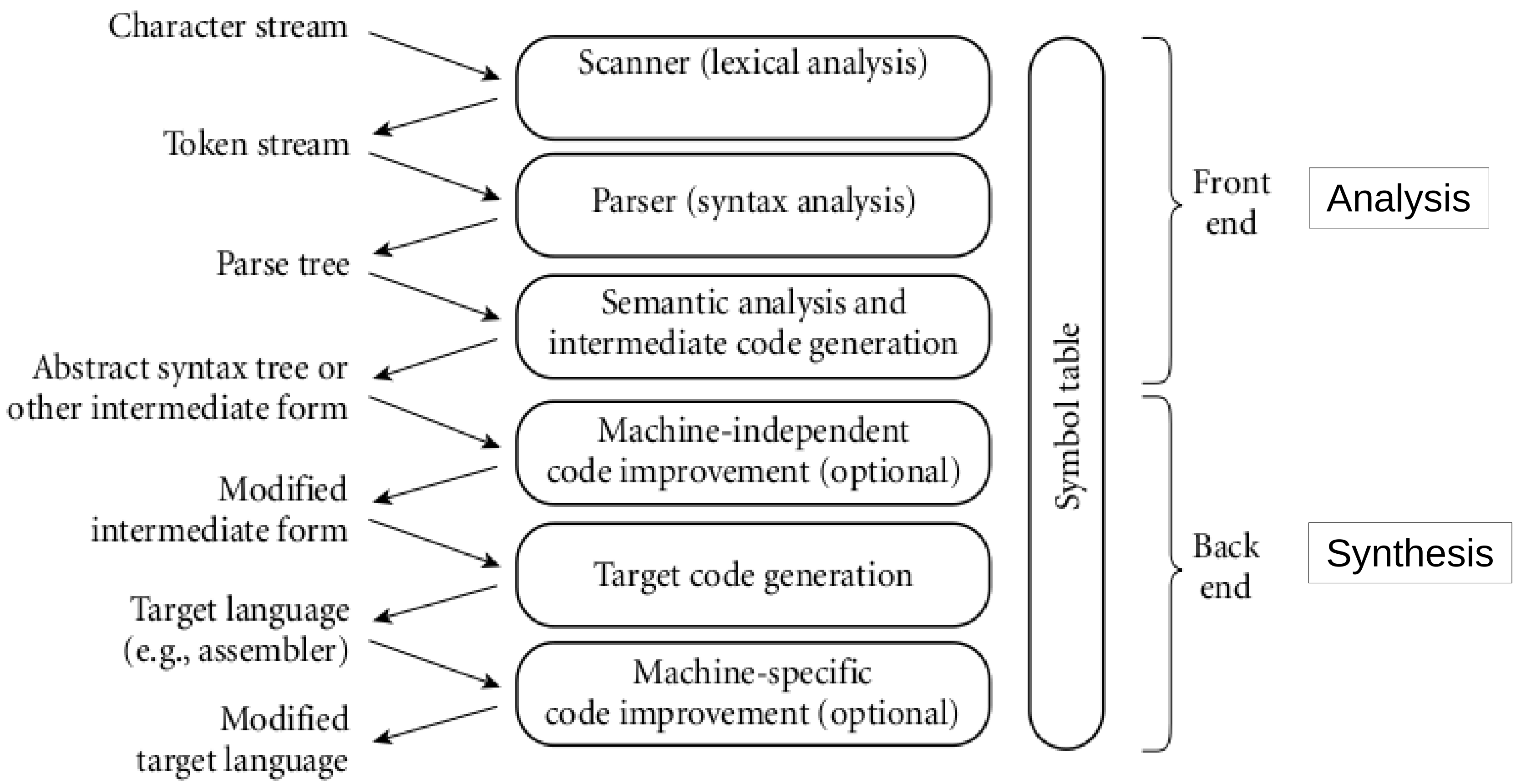
ii. Back-end:

Synthesis

- Intermediate optimisation
- Flow (data and execution) analysis,
- Code generation + target-dependent optimisation;

iii. Linking (optional)

Phases of compilation



Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Line reconstruction

- UNICODE characters are substituted by escape-sequences, e.g. č → \x10D;
- tabulators are substituted by delimiters (usually spaces);
- end-of-line characters are substituted by delimiters;
- string literals

  are merged;
- empty lines

  are removed.

Example (in C)

```
#include<stdio.h> int
main(
void)
{
                printf("Hello"
" world"
)
;
    return 0;}
```

Lexical analysis

- Normalised text form of program is divided into the list of tokens, the lexical units of the languaguage.
- The usual types of tokens are:
  – literals: 12345, 0xAB, "hello world", ’c’, true. . .
  – keywords: if, else, for, function, return. . .
  – identifiers: var1, a$, MyClass,. . .
  – type names: int, char*, ’a, void. . .
  – operators: =, <-, +, :=, ==, >=,. . .
  – parenthesis, braces, and brackets;
  – indentations (old Fortran, Python);
- The sequence of tokens is produced, the delimiters are removed.

Lexical analysis: Rationale

- Lexical analyser reads the source by characters and if a token is recognised, it is appended to the sequence of tokens;
- Very tedious and complicated, regular expressions are used instead;
- Regular expressions are kind of abstract computers (DFAs), which recognise prescribed patterns in the text;
- Regular expressions are ‘hungry’, they match the longest possible text;
- Lexical analysers are constructed using specialised compilers - lexers (lex, flex);
- Lexers used to produce the source code of the lexical analyser in a high-level programming language (C, Java, OCaml, Pascal, Python).

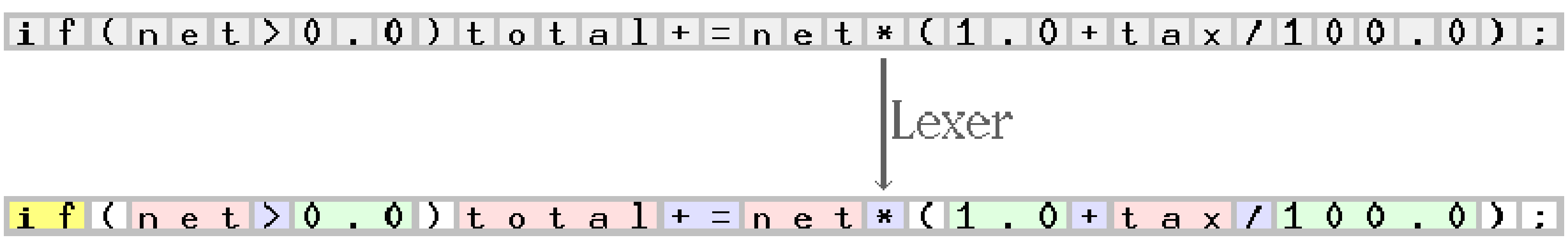
Lexical analysis: Examples

Regular expressions

```
     <id>  ::=  [_|__]?[a-zA-Z][_a-zA-Z0-9]*
 <number>  ::=  [0-9]+ | 0x[0-9a-fA-F]+
    <kwd>  ::=   if | else | for | while | return
  <const>  ::=  <number> | <float> | <character>| \
                 <boolean> | <string>
```

Regular expressions are ‘hungry’, The RE 0x[0-9a-fA-F]+ recognise in string 0x109AB a hexadecimal constant instead of <const><id>.

Recognising tokens



Preprocessing

```
#ifndef SET_H
#define SET_H
#define max(a,b)  ((a) > (b) ? (a) : (b))
```

- Comment blocks and comment lines are thrown away;
- Many languages use preprocessor macros - special language for text processing;
- Macros are expanded at this moment;
- Conditional compilation is evaluated and omitted blocks of tokens are removed;
- Auxiliary files (header files, in C, C++) are placed instead of corresponding macros (#include);
- The lexical analyser is run again.

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Grammars of programming languages

Programming languages have simple syntax which can be described in terms of context-free grammars.

Context-free grammar is a mathematical structure of the form

$$G = (V, T, R, S)$$

where

- V is the set of non-terminals (variables),
- T is the set of terminals (symbols),
- R is the set of production rules,
- S is the starting non-terminal.

.

Grammars of programming languages

Example:

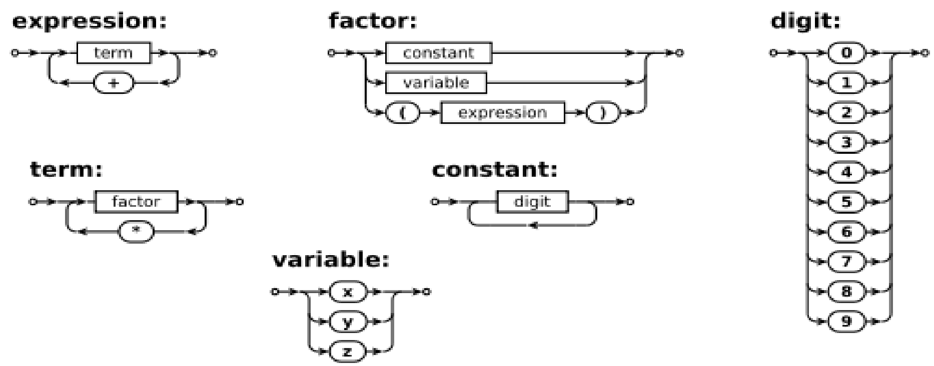
Language of binary words consisting of equal number of ‘0’s and ‘1’ can be expressed by grammar

$$G = (\{S\}, \{0, 1\}, R, S)$$, where R is the set of rules

$$S \rightarrow 01 \mid 10 \mid 0S1 \mid 1S0$$

Backus-Naur form

A convenient way of expressing (context-free) grammars of programming languages.



Example:

The simple languag

```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
```

Syntax analysis - parsing

- Parser is the algorithm which takes the ordered list of tokens and recognise which rules of the grammar form input list of tokens
- The (concrete) syntax tree of the input is created
  - The top (base, root) vertex corresponds to the starting non-terminal (expression)
  - Nodes correspond to the rules, and leaves correspond to terminals (tokens)
- Every expression in the source yields a syntax tree

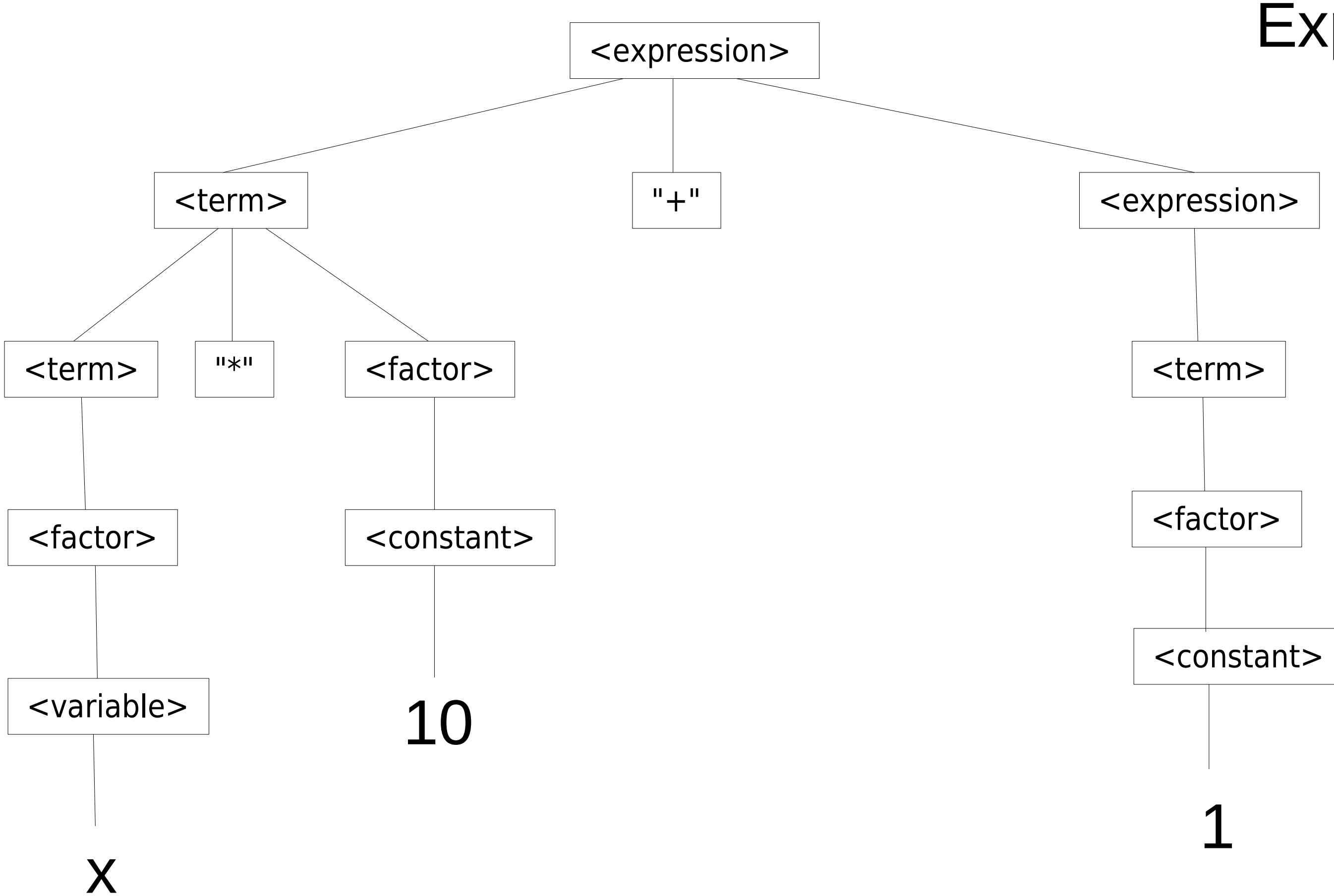
Syntax analysis - parsing

- The process of construction of a syntax tree is ambiguous, since CFGs are ambiguous. We have several methods to get rid of ambiguities:
  - orderting of rules in the grammar (precedence);
  - look-ahead buffer for tokens (LL-parsers, LL(1));
  - read from left, parse from right (LR, LALR parsers)

Example: Syntax tree

<expression> ::= <term> | <term> "+" <expression>
<term> ::= <factor> | <term> "*" <factor>
<factor> ::= <constant> | <variable> | "(" <expression> ")"
<variable> ::= "x" | "y" | "z"
<constant> ::= <digit> | <digit> <constant>

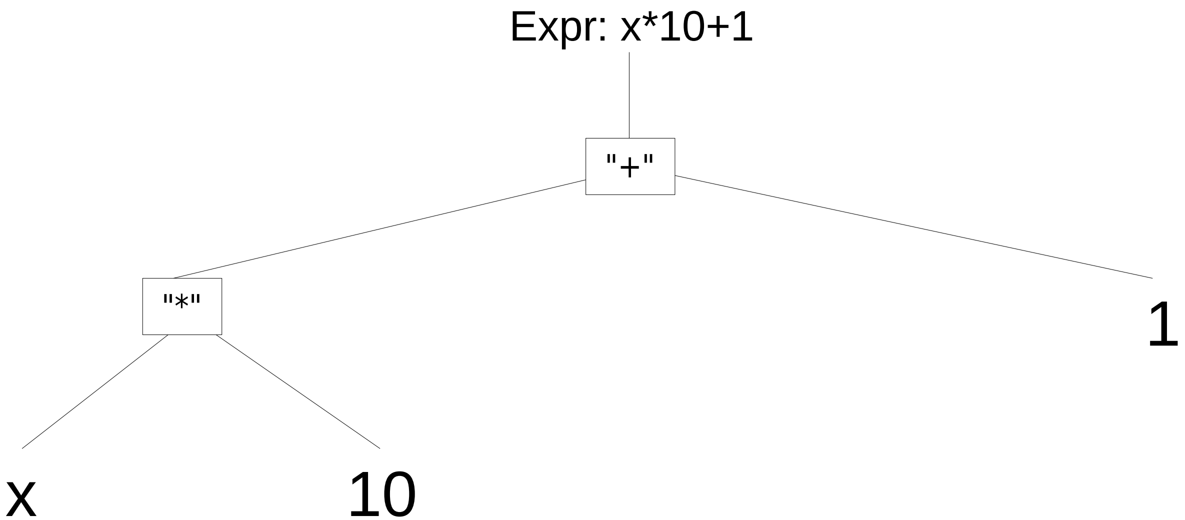
Expression: x*10+1



Abstract syntax tree (AST)

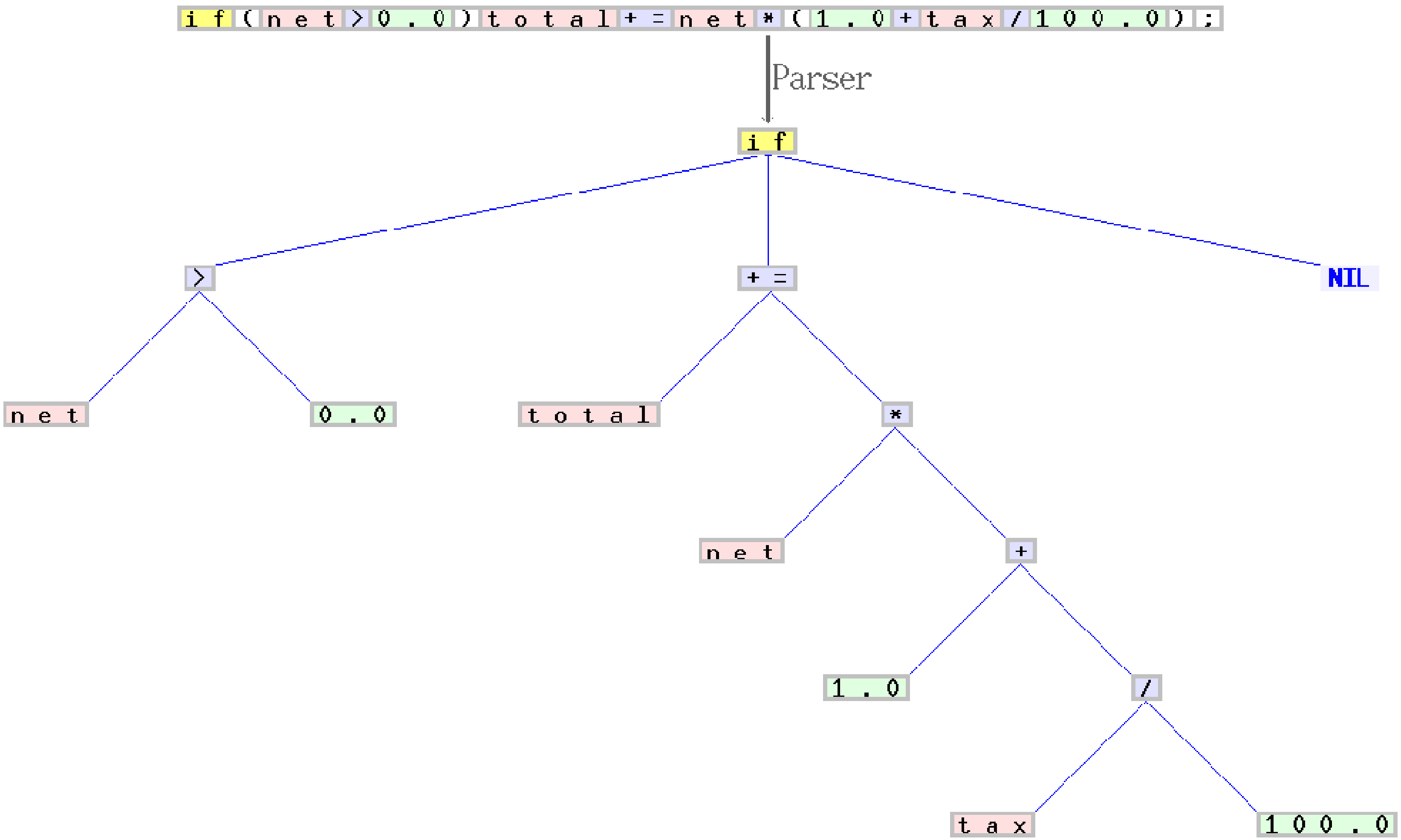
- AST keeps the essence of the structure but omits the irrelevant details
  - Each node corresponds to one or more nodes in the (concrete) syntax tree
- Additional data can be attached to the nodes
  - For type checking
  - For analyses that can not be implemented in syntax analaysis or type checking
  - For the syntax directed translation

Example: Abstract syntax tree (1)




```
<expression> ::= <term> | <term> "+" <expression>
<term>       ::= <factor> | <term> "*" <factor>
<factor>     ::= <constant> | <variable> | "(" <expression> ")"
<variable>   ::= "x" | "y" | "z"
<constant>   ::= <digit> | <digit> <constant>
```

Example: Abstract syntax tree (2)



Symbol tables

- Symbol table holds information about source-program symbols
  - Names of variables, types, functions, etc.
  - Data about symbols are collected incrementally by the analysis phases of a compiler
    - Used by the synthesis phases to generate the target code
  - For each identifier we store the following data
    - Character string of identifier, its type, its position in storage, …
- A symbol table is defined for each scope
  - We can have different symbols with the same name in different scopes
  - Scopes are linked hierarchicaly

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Type checking

- Types as abstract representation of values
  - Description of the structure and the domain of variables
- Type checking is a semantic analysis of a program
  - Aspect that is <u>not covered</u> by grammar rules
  - Verifies that the types of the operation arguments agree with the type of the operation
    - First, the types of arguments are derived
    - Then the rules are used to check the compatibility
- Type derivation
  - Types of expressions are derived from the types of the sub-expressions
    - AST is usually used

Type checking

  - Types in polimorphic type systems are computed as the results of a system of equations
- Type derivation and type checking is covered in more detail in the lecture on Types
  - Type equality, conversions, coerction
  - Type derivations
  - Rule-based type checking
  - Type derivations in languages with polymorphism
- Here we will inspect an example of the type checking rules for expressions

Type checking

- Example
  - Type checking the language of Expressions

$Check_{Exp}(Exp, vtable, ftable) = \texttt{case}\ Exp\ \texttt{of}$	
num	$\texttt{int}$
id	$t = lookup(vtable, getname(\textbf{id}))$ $if\ \ t = unbound$ $then\ \ \textbf{error}();\ \texttt{int}$ $else\ \ t$
$Exp_1 + Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ \ t_1 = \texttt{int}\ and\ t_2 = \texttt{int}$ $then\ \ \texttt{int}$ $else\ \ \textbf{error}();\ \texttt{int}$
$Exp_1 = Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $if\ \ t_1 = t_2$ $then\ \ \texttt{bool}$ $else\ \ \textbf{error}();\ \texttt{bool}$
$\texttt{if}\ Exp_1$ $\texttt{then}\ Exp_2$ $\texttt{else}\ Exp_3$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $t_2 = Check_{Exp}(Exp_2, vtable, ftable)$ $t_3 = Check_{Exp}(Exp_3, vtable, ftable)$ $if\ \ t_1 = \texttt{bool}\ and\ t_2 = t_3$ $then\ \ t_2$ $else\ \ \textbf{error}();\ t_2$
id ($Exps$)	$t = lookup(ftable, getname(\textbf{id}))$ $if\ \ t = unbound$ $then\ \ \textbf{error}();\ \texttt{int}$ $else$ $\quad ((t_1, \ldots, t_n) \rightarrow t_0) = t$ $\quad [t'_1, \ldots, t'_m] = Check_{Exps}(Exps, vtable, ftable)$ $\quad if\ \ m = n\ and\ \ t_1 = t'_1, \ldots, t_n = t'_n$ $\quad then\ \ t_0$ $\quad else\ \ \textbf{error}();\ t_0$
$\texttt{let}\ \textbf{id} = Exp_1$ $\texttt{in}\ Exp_2$	$t_1 = Check_{Exp}(Exp_1, vtable, ftable)$ $vtable' = bind(vtable, getname(\textbf{id}), t_1)$ $Check_{Exp}(Exp_2, vtable', ftable)$

Semantic analysis

- Semantic analyser enforces a variety of rules
  - These rules are not captured by parsing (using CFG) or type checking.
- Static and dynamic semantic rules
  - Static are checked in compile time
  - Code is generated for dynamic checks

Semantic analysis

- Typical static rules covered by semantic analyser
  - Every identifier is declared before it is used
  - Labels on the arms of a switch statement are distinct constants
  - Some languages allow coerecion
    - Parameters are automatically converted to the expected type
  - Any function with a non-void return type returns a value explicitly

Semantic analysis

- Examples of rules enforced at run time include the following
  - Variables are never used in an expression unless they have been given a value.
  - Pointers are never dereferenced unless they refer to a valid object.
  - Array subscript expressions lie within the bounds of the array.
  - Arithmetic operations do not overflow.

Pre-intermediate code

- Every node of the abstract syntax tree (a rule of CFG) is represented by a ‘template code’
  - The parsing tree is transformed to the linear sequence of instructions (codes);
  - The templates are filled by actual variables and values;
  - The codes for all expression in the program are merged into the resulting sequence of instructions.
- Pre-intermediate code
  - The form of pre-intermediate code can be
    - Sequence of instructions
    - Annotated abstract syntax tree

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Intermediate-Code Generation

- Eventually, the program will have to be expressed as code for the given concrete machine
- Many compilers use a medium-level language.
  - A stepping-stone language is called an intermediate language (abbr. IL)
- Advantages:
  - Structuring the compiler into smaller jobs
  - Several high-level languages can be compiled to IL
  - IL can be compiled to different target architectures
  - IL can be interpreted
    - By program implemented on the target architecture

Intermediate-Code Generation

- To compile many different languages (N) to many target architectures (M)
  - Direct translation: N*M
  - Intermediate language: N+M
- Interpreter for an intermediate language
  - Use a PL implemented on many architectures
  - The same interpreter can be used for different architectures
  - Single intermediate code can be used for all machines
  - Intermediate form may be more compact than machine code

Intermediate-Code Generation

- The disadvantage is speed
  - Interpreting will (in most cases) be a lot slower than the machine code
- Java virtual machine is a great success
- Translating the intermediate code to machine code during execution of the program
  - Just-in-time (JIT) compilation
  - Often used for executing the intermediate code for Java
- We will focus mainly on using the intermediate code for traditional compilation
  - Translated to machine code by a back-end of compiler

Choosing an intermediate language

- Conflicting goals:
  - Easy to translate from (different) high-level languages to the intermediate language
  - Easy to translate from the intermediate language to a wide range of different target architectures
  - The intermediate format should be suitable for optimisations
- The level of the language
  - High-level intermediate language, more burden on the back-ends
  - Low-level intermediate language, more burden on the front-ends

Choosing an intermediate language

- Intermediate language “granularity”
  - Should an operation in the intermediate language correspond to a large or small amount of work?
- Complex operations
  - Often used for interpreters (performance)
  - Mapped to a sequence of machine instructions
- Very simple operations
  - Sequence of operations mapped to one machine instruction

Intermediate language

- Fairly low-level fine-grained intermediate language

$$
\begin{array}{lcl}
\mathit{Program} & \rightarrow & [\,\mathit{Instructions}\,] \\
\\
\mathit{Instructions} & \rightarrow & \mathit{Instruction} \\
\mathit{Instructions} & \rightarrow & \mathit{Instruction}\ ,\ \mathit{Instructions} \\
\\
\mathit{Instruction} & \rightarrow & \texttt{LABEL}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{unop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \textbf{id}\ \textbf{binop}\ \mathit{Atom} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := M[\mathit{Atom}] \\
\mathit{Instruction} & \rightarrow & M[\mathit{Atom}] := \textbf{id} \\
\mathit{Instruction} & \rightarrow & \texttt{GOTO}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \texttt{IF}\ \textbf{id}\ \textbf{relop}\ \mathit{Atom}\ \texttt{THEN}\ \textbf{labelid}\ \texttt{ELSE}\ \textbf{labelid} \\
\mathit{Instruction} & \rightarrow & \textbf{id} := \texttt{CALL}\ \textbf{functionid}(\mathit{Args}) \\
\\
\mathit{Atom} & \rightarrow & \textbf{id} \\
\mathit{Atom} & \rightarrow & \textbf{num} \\
\\
\mathit{Args} & \rightarrow & \textbf{id} \\
\mathit{Args} & \rightarrow & \textbf{id}\ ,\ \mathit{Args}
\end{array}
$$

Syntax directed translation

- Generate code using translation functions for each syntactic category
  - Syntactical category is defined by a grammar rule (non-terminal)
  - The parameters of a translation function hold information about the context (e.g., symbol table)
  - Additional attributes are defined for nodes of AST.
- Code generated locally, for a given category
  - Translation is not optimal in regards to the related categories
  - Optimization (presented later) eliminates unnecessary variables, itd.

Translating expressions

$$
\begin{array}{lcl}
Exp & \rightarrow & \textbf{num} \\
Exp & \rightarrow & \textbf{id} \\
Exp & \rightarrow & \textbf{unop}\ Exp \\
Exp & \rightarrow & Exp\ \textbf{binop}\ Exp \\
Exp & \rightarrow & \textbf{id}(Exps) \\
\\
Exps & \rightarrow & Exp \\
Exps & \rightarrow & Exp\ ,\ Exps
\end{array}
$$

$Trans_{Exp}(Exp, vtable, ftable, place) =$ case Exp of	
num	$v = getvalue(\textbf{num})$ $[place := v]$
id	$x = lookup(vtable, getname(\textbf{id}))$ $[place := x]$
unop Exp_1	$place_1 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $op = transop(getopname(\textbf{unop}))$ $code_1 +\!\!+ [place := op\ place_1]$
Exp_1 **binop** Exp_2	$place_1 = newvar()$ $place_2 = newvar()$ $code_1 = Trans_{Exp}(Exp_1, vtable, ftable, place_1)$ $code_2 = Trans_{Exp}(Exp_2, vtable, ftable, place_2)$ $op = transop(getopname(\textbf{binop}))$ $code_1 +\!\!+ code_2 +\!\!+ [place := place_1\ op\ place_2]$
id$(Exps)$	$(code_1, [a_1, \ldots, a_n])$ $= Trans_{Exps}(Exps, vtable, ftable)$ $fname = lookup(ftable, getname(\textbf{id}))$ $code_1 +\!\!+ [place := \texttt{CALL}\ fname(a_1, \ldots, a_n)]$

$Trans_{Exps}(Exps, vtable, ftable) =$ case $Exps$ of	
Exp	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_1, [place])$
$Exp\ ,\ Exps$	$place = newvar()$ $code_1 = Trans_{Exp}(Exp, vtable, ftable, place)$ $(code_2, args) = Trans_{Exps}(Exps, vtable, ftable)$ $code_3 = code_1 +\!\!+ code_2$ $args_1 = place :: args$ $(code_3, args_1)$

Translating statements

$$
\begin{array}{lcl}
Stat & \rightarrow & Stat\ ;\ Stat \\
Stat & \rightarrow & \mathbf{id} := Exp \\
Stat & \rightarrow & \texttt{if}\ Cond\ \texttt{then}\ Stat \\
Stat & \rightarrow & \texttt{if}\ Cond\ \texttt{then}\ Stat\ \texttt{else}\ Stat \\
Stat & \rightarrow & \texttt{while}\ Cond\ \texttt{do}\ Stat \\
Stat & \rightarrow & \texttt{repeat}\ Stat\ \texttt{until}\ Cond \\
 & & \\
Cond & \rightarrow & Exp\ \mathbf{relop}\ Exp
\end{array}
$$

$Trans_{Stat}(Stat, vtable, ftable) =$ case $Stat$ of	
$Stat_1\ ;\ Stat_2$	$code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 +\!\!+ code_2$
$\mathbf{id} := Exp$	$place = lookup(vtable, getname(\mathbf{id}))$ $Trans_{Exp}(Exp, vtable, ftable, place)$
if $Cond$ then $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_1 +\!\!+ [\text{LABEL } label_1] +\!\!+ code_2$ $\quad +\!\!+ [\text{LABEL } label_2]$
if $Cond$ then $Stat_1$ else $Stat_2$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_1, label_2, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_3 = Trans_{Stat}(Stat_2, vtable, ftable)$ $code_1 +\!\!+ [\text{LABEL } label_1] +\!\!+ code_2$ $\quad +\!\!+ [\text{GOTO } label_3,\ \text{LABEL } label_2]$ $\quad +\!\!+ code_3 +\!\!+ [\text{LABEL } label_3]$
while $Cond$ do $Stat_1$	$label_1 = newlabel()$ $label_2 = newlabel()$ $label_3 = newlabel()$ $code_1 = Trans_{Cond}(Cond, label_2, label_3, vtable, ftable)$ $code_2 = Trans_{Stat}(Stat_1, vtable, ftable)$ $[\text{LABEL } label_1] +\!\!+ code_1$ $\quad +\!\!+ [\text{LABEL } label_2] +\!\!+ code_2$ $\quad +\!\!+ [\text{GOTO } label_1,\ \text{LABEL } label_3]$
repeat $Stat_1$ until $Cond$	$label_1 = newlabel()$ $label_2 = newlabel()$ $code_1 = Trans_{Stat}(Stat_1, vtable, ftable)$ $code_2 = Trans_{Cond}(Cond, label_2, label_1, vtable, ftable)$ $[\text{LABEL } label_1] +\!\!+ code_1$ $\quad +\!\!+ code_2 +\!\!+ [\text{LABEL } label_2]$

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Machine-Code Generation

- Intermediate language we used is quite low-level
  - Similar to the machine code you can find on modern RISC processors
  - We will use RISC MIPS (RISC V)
- Often we have one-to-one mapping from IL to the RISC instruction set
  - Complex RISC (MIPS) instructions map to IL patterns
  - And vice versa
- Problems involved in translation
  - Differences in the instruction sets
  - Register allocation
  - Function call sequences

Machine-Code Generation

- Differences between RISC processor operations and IL operations
  1) IF-THEN-ELSE instruction has two target labels
     - Conditional jump instruction has only one target label
  2) Any constant can be operand to an instruction
     - RISC processors allow only small constants as operands
  3) There are some complex operations in (RISC) MIPS and ARM processors
  4) We used an unbounded number of variables
     - There is a bounded number of registers
  5) We have used a complex CALL instruction

Conditional jumps

- Conditional jumps come in many forms on different machines
  - Relational comparison between registers (if-then-else)
  - Conditional jump instructions specify only one target address
    - if c a1; jp a2
  - Often followed by one of target addresses (see rule for IF in IL)
    - Generator checks what follows
  - Code must be generated for the condition and stored
    - In gen.purpose register (MIPS, Alpha) + In special register (IA-64, PowerPC), + In flags (Sparc, IA-32) ...

IF c THEN l_t ELSE l_f

```
branch_if_c   l_t
jump          l_f
```

```
branch_if_not_c   l_f
```

Constants

- IL allows arbitrary constants as operands to binary or unary operators
  - Not so in MIPS, ARM, …
  - More machine instructions needed for a single comparison
- Code generator must check if constant matches with some machine-code instruction
  - If it does, the code generator generates a single machine-code instruction
  - If not,
    1) sequence of instructions builds the constant in a register,
    2) an instruction uses this register in place of the constant

Complex instructions

- Most instructions in our IL are atomic
  - Using RISC MIPS, ARM (in mobile phones)
  - Each instruction corresponds to a single operation
- Complex RISC operations (MIPS, ARM)
  - Mapped to a sequence of IL instuctions
- Example:

$$t_2 := t_1 + 116$$
$$t_3 := M[t_2]$$

```
lw r3, 116(r1)
```

Pattern/
replacement
pairs for
a subset of
the MIPS
instruction set

$t := r_s + k$, $r_t := M[t^{last}]$		lw	$r_t,\ k(r_s)$
$r_t := M[r_s]$		lw	$r_t,\ 0(r_s)$
$r_t := M[k]$		lw	$r_t,\ k(\mathtt{R0})$
$t := r_s + k$, $M[t^{last}] := r_t$		sw	$r_t,\ k(r_s)$
$M[r_s] := r_t$		sw	$r_t,\ 0(r_s)$
$M[k] := r_t$		sw	$r_t,\ k(\mathtt{R0})$
$r_d := r_s + r_t$		add	r_d, r_s, r_t
$r_d := r_t$		add	$r_d, \mathtt{R0}, r_t$
$r_d := r_s + k$		addi	r_d, r_s, k
$r_d := k$		addi	$r_d, \mathtt{R0}, k$
GOTO $label$		j	$label$
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$	 $label_f$:	beq	$r_s, r_t, label_t$
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$	 $label_t$:	bne	$r_s, r_t, label_f$
IF $r_s = r_t$ THEN $label_t$ ELSE $label_f$		beq j	$r_s, r_t, label_t$ $label_f$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_f$	 $label_f$:	slt bne	r_d, r_s, r_t $r_d, \mathtt{R0},\ label_t$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$, LABEL $label_t$	 $label_t$:	slt beq	r_d, r_s, r_t $r_d, \mathtt{R0},\ label_f$
IF $r_s < r_t$ THEN $label_t$ ELSE $label_f$		slt bne j	r_d, r_s, r_t $r_d, \mathtt{R0}, label_t$ $label_f$
LABEL $label$	$label$:		

Register allocation

- When generating code in IL we used as many variables as we found convenient
  - Processors do not have an unlimited number of registers
- We need register allocation to handle this conflict
  - Map a large num of vars into a small num of registers
  - Letting several variables share a single register
  - Sometimes, not enough registers in the processor
- This is called spilling
  - Some variables must be temporarily stored in memory

Register allocation

- When can two variables share a register?
- Liveness of a variable
  - Value it contains might conceivably be used in future
    - Formal definition (through the changes of states)
- Register allocation by graph colouring
  - Interference graph of variables
    - Two variables are linked if they interfere
    - Two nodes that share an edge have different register numbers
    - Register numbers must not be higher than the num of available registers
    - Otherwise, some variables are stored to the memory
  - NP complete problem

Function calls

- Function call was not considered before
  - When translating the function...
- The call stack
  - When function is called all live variables are stored (from registers) to the memory
    - Stack is used as the temporary storage
  - Now registers are free to be used in a callee
  - Stack is used also for the activation records of callees
    - Variables and parameters of the callee
    - Control data such as return address, scope info, etc.
  - Activation records are detailed in lecture on Memory management

Function calls

- Issues handeled by a function call
  - Prologues, epilogues and call-sequences
    - Handling registers, live vars, parameters, activation records
    - What happens when a function call is issued?
    - And when function returns?
  - Who saves registers? Caller or callee?
    - More complex when caller saves live variables, and callee saves the variables it needs.
  - Accessing global variables
    - Handling scope by relating activation records
  - Using registers to pass parameters
  - Interaction with the register allocator

Outline

- History
- Architecture of compiler
- Lexical analysis
- Grammars & parser
- Semantic analysis
- Intermediate language
- Code generation
- Code optimization

Analysis and optimisation

- Recognising specific patterns in a program
- Replacing recog.patterns by smaller and/or faster patterns
  - Replacing sequences of instructions by other sequences
  - Can be applied to intermediate lang. or machine code
- Peephole optimisation
  - We look at the code through a small hole
  - We only see short sequences of instructions
  - However, non-local properties require looking at an arbitrarily large context!

Peephole optimizations

- Eliminating redundant loads and stores

```
LD a, R0
ST R0, a
```

- Eliminating unreachable code

```
    if debug == 1 goto L1
    goto L2
L1: print debugging information
L2:
```

- Flow-of-control optimizations

```
    goto L1
    ...
L1: goto L2
```

- Algebraic simplification and reduction in strength
  - Eliminate three-address statements
  - Replace expensive operations by cheaper ones (x^2=x*x)

```
x = x + 0
x = x * 1
```

Data-flow analyses

- Attempt to discover how information flows through a program
  - Recognising patterns in a given context
  - E.g., liveness analysis of variables at each instruction
    - We have **in** and **out** sets (of vars) for each instructions
- Context data needed for the analyses
  - Represents one aspect of program computation
  - Different types of analyis requires different types and treatments of data-flows

Data-flow analyses

- Data-flow analysis is used for optimisation
  - Replacing one sequence by another sequence of instructions
  - E.g., analysis of variable liveness can improve register allocation
- Backward and forward analysis
  - Liveness analysis is a backward analysis
  - Data-flow goes back: from the use to the assignement of variables
- Some examples of data-flow analyses will be presented in this section

Examples of data-flow analyses

- Dead code elimination
  - False branches of conditionals
- Constant propagation
  - It is cheaper to allocate a fixed block of memory for constants at the beginning
- Common sub-expressions
  - Identified and computed only once (references to values are set)
- Jump-to-jump elimination
  - Sequences of jumps replaced by the direct jump

Examples of data-flow analyses

- Indexes to arrays are checked only once
  - Unnecessary checks are eliminated.
- Loop transformation
  - Memory pre-fetching
  - Interchange: exchange inner loops with outer loops. Improve locality of reference
  - Vectorisation: attempts to run as many of the loop iterations as possible at the same time
  - Reversal: Enables other optimization by reversing the loop
  - Code hoisting: removing loop-invariant code

Examples of data-flow analyses

- Function calls
  - Inline expansion: inline code is ‘cheaper’ than function call
  - Tail-call optimisation: transform tail recursion to iteration
  - Specialization: handle specific calls by removing unnecessary code
- Automatic parallelisation
  - Processors have several cores, independent code segments can be executed at once.