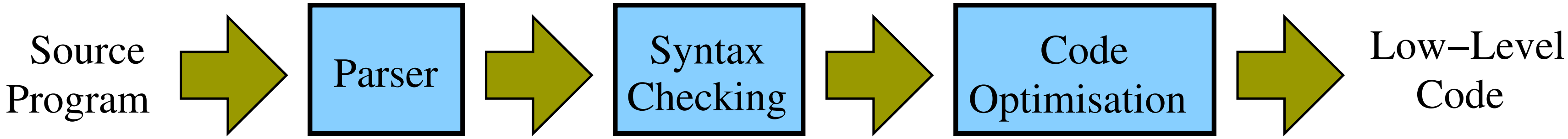
SWEN430 - Compiler Engineering

Lecture 2 - Compiler Architecture and the WHILE Language

Linsday Groves and David Pearce

School of Engineering and Computer Science
Victoria University of Wellington

What is a Compiler?



Compilers translate **source programs** into **low-level code**

- recognise program structure
- check for certain errors (e.g. syntax errors, type errors)
- optimise the program where possible
- generate "low-level" code (VM code or machine code)

Compiler Architecture

- The usual structure of a compiler is:
  - Front end: Read program, check for errors, create intermediate form (e.g. AST)
  - Back end: Translate AST to machine/vm code and optimise;
    or just execute (interpreter);
    or compile while executing (JIT)
- This gives a "2^+ pass" compiler — front and back ends may each make several passes over the program (e.g. see JKit slide lecture 1).
- Intermediate form(s) may be written to a file or passed as a data structure, or front and back ends may run as coroutines.
- Compiled form usually written to a file which is then loaded, along with run-time system. May just write to memory and execute.
- A "one pass compiler" does everything in one pass!
  (Ex: What problems does that pose?)

Front end – more closely

The front end has several steps, mirroring the language definition:

- Scanner (Lexer, Tokeniser):
  - Read input as a sequence characters/lines
  - Output a sequence of tokens/lexemes/symbols
  - Report lexical errors (illegal symbols)
  - Strip out comments and white space
  - Valid tokens are defined using *regular expressions*.

    E.g.: *arithOp* $= + \mid - \mid * \mid /$
    relOp $=$ < | > | = | <= | >= | !=
    ident $=$ *letter* (*letter*|*digit*)*
    number $=$ *digit digit** ["." *digit**]

Front end – more closely

- Parser:
  - Read input as sequence of tokens
  - Build a parse tree (perhaps implicitly) with the input as its fringe
  - Report syntax errors (illegal combinations of symbols)
  - Build symbol table containing identifiers declared in the program
  - Valid programs are defined using a *context free grammar (BNF)*.

    E.g.:

    $$
    \begin{array}{ll}
    \textit{prog} & :== \textit{declPart stmtPart} \\
    \textit{declPart} & :== \textit{decl}^* \\
    \textit{decl} & :== \textit{ident ident}\text{''};\text{''} \\
    \textit{stmtPart} & :== \textit{stmt}^* \\
    \textit{stmt} & :== \text{''}\textit{skip}\text{''} \mid \textit{ident} = \textit{exp} \mid \text{''}\textit{if}\text{''}\ \textit{exp}\ \text{''}\textit{then}\text{''}\ \textit{stmt}\ [\text{''}\textit{else}\text{''}\ \textit{stmt}] \mid \ldots
    \end{array}
    $$

Front end – more closely

- Context checking, type checking and static analysis:
  - Check for context conditions such as variables used and methods called must be declared
  - Check for type correctness: methods and operators must be applied to arguments of the correct type
  - Check for properties such as definite assignment and dead code
  - Context/type constriants can be defined formally, but need more complex formalisms that grammars, so they are usually defined informally.

Designing the interfaces: Tokens

We have important choices to make in designing the set of tokens to use.

- We might treat all operators as distinct tokens, or all as one token type, or as sets of tokens that are used in the same way (arithOp, relOp, ...).
  Making a good choice here helps to simplify the grammar, and hence the parser.

Designing the interfaces: AST

We have important choices to make in designing the abstract syntax tree and symbol table representations.

- Again, making good choices here helps to simplify the parser and the checking and translation that is done later.

The WHILE Language — overview

test.while

```
type Point is {int x, int y}

Point move(Point p, int dx, int dy) {
  return {x: p.x + dx, y: p.y + dy};
}
```

- A **simple** imperative language
- **Statements**: for, while, if, switch, ...
- **Expressions**: binary, unary, invocation, ...
- **Types**: bool, int, strings, arrays, records, ...

Primitive Types

```
bool f2() { return false; }

char f3() { return 'X'; }

int f4(int x) { return x + 1; }

string f6() { return "Hello World"; }
```

- **bool** — **true** or **false**
- **char** — ASCII characters (not unicode, for simplicity)
- **int** — 32bit signed integers (identical to Java **int**)
- **string** — sequence of **chars**

Record Types

```
{int x, int y} Point(int x, int y) {
  return {x: x, y: y};    // record construction
}

int getX({int x, int y} p) {
  return p.x;             // field access
}

{int x,int y} setX({int x, int y} p, int v) {
  p.x = v;                // field assignment
  return p;
}
```

- Similar to `structs` in C and objects in Java and/or JavaScript
- Support **width** and **depth** subtyping (more on this later)

Array Types

```
int[] trim(int[] xs, int n) {
  int[] rs = [0; n]
  int i = 0;
  while(i < |rs|) {
    rs[i] = xs[i];
    i = i + 1;
  }
  return rs;
}
```

- Similar to **arrays** in C and Java, but have **value** semantics
- Support *array access* (`xs[i]`) and *array length* (`|xs|`)
- Support *array initialisers* (`[1,2,3]`) and *generators* (`[0; n]`)

Type Declarations

```
type Point is {int x, int y}

type Line is {Point start, Point end}
```

- Can declare **new types** via `type`
- Types are **structural** and cannot be **recursive**
- Types can refer to types declared **earlier** in source file
- Should regard such declarations as **macros**
- E.g. `{{int x,int y} start, {int x,int y} end}`

Statements

```
string toString(int c) {
  switch(c) {
    case 1:
        return "ONE";
    case 2:
        return "TWO";
    case 3:
        return "THREE";
    default:
      return "";
  }
}
```

- Support **`if`**, **`while`**, **`for`**, **`return`**, **`switch`**, **`print`**
- Syntax is roughly same as for Java

Expressions

- **Constants:** `1`, `2345`, **`true`**, **`false`**, *`'c'`*, *`"hello"`*
- **Comparators**: `==`, `!=`, `<`, `<=`, `>=`, `>`
- **Arithmetic**: `+`, `-`, `*`, `/`, `%`
- **Logical**: `!`, `&&`, `||`
- **Arrays**: `[1,2,3]`, `[e; n]`, `xs[i]`, `|xs|`
- **Records**: `{x: 1, y: 2}`, `r.f`
- **Invocations**: `f(1,2,3)`

Value Semantics

```
int[] inc(int[] xs) {
  for(int i=0;i!=|xs|;i=i+1) { xs[i] = xs[i] + 1; }
  return xs;
}

void f() {
  int[] xs = [1,2,3];
  int[] ys = inc(xs);
  print xs;            // prints [1,2,3]
  print ys;            // prints [2,3,4]
}
```

- All data types have **value semantics**
- They are **passed by value**, and updates to them **do not affect** other variables

Definite Assignment

```
int f() {
 int x;
 return x+1;              // error
}
```

```
int f(int y) {
 int x;
 if(y == 1) { x = 1; }
 return x;                // error
}
```

- Every variable must be **defined** before it is used!
- Simple (conservative) analysis used to check this (see JLS §16)

Unreachable Code

```
int f() {
  return 1;
  return 2; // error
}
```

```
int f(int y) {
 if(y == 1) { return 1; }
 else { return 2; }
 return 3; // error
}
```

- Code which is **unreachable** is not permitted (see JLS §14.21)
- Simple (conservative) analysis used to check this.

Type Checking

```
int f(real x) { return x; } // error
```

```
void f(bool x) { int y = x; } // error
```

```
real f(int x, bool y) { return x + y; } // error
```

```
type Point is {int x, int y}

bool f(Point p) { return p.x; } // error
```

- The process of checking a program is **well-typed**

Subtyping

Definition (Structural Subtyping)

We write $\mathtt{T}_1 \leq \mathtt{T}_2$ to indicates $\mathtt{T}_1$ is a *subtype* of $\mathtt{T}_2$. If $[\![\mathtt{T}]\!]$ is the set of all values represented by $\mathtt{T}$, then $\mathtt{T}_1 \leq \mathtt{T}_2 \iff [\![\mathtt{T}_1]\!] \subseteq [\![\mathtt{T}_2]\!]$

- **Void** is bottom (e.g. $\mathbf{void} \leq \mathbf{int}$)
- **Covariant** array subtyping (e.g. $\mathbf{void}\mathtt{[]} \leq \mathbf{int}\mathtt{[]}$)
- **Width** subtyping of records (e.g. $\{\mathbf{int}\ \mathtt{x}, \mathbf{int}\ \mathtt{y}\} \leq \{\mathbf{int}\ \mathtt{x}\}$)
- **Depth** subtyping of records (e.g. $\{\mathbf{void}\mathtt{[]}\ \mathtt{x}\} \leq \{\mathbf{int}\mathtt{[]}\ \mathtt{x}\}$)

Note: *Covariant array subtyping is safe in* WHILE *because arrays have value semantics, unlike Java where it is unsafe.*

(No) Function Overloading

```
int f(int x) {
  return 42;
}

int f(int[] xs) { // error
  return 42;
}
```

- Java supports **function overloading**, but WHILE does not
- This eliminates problem of determining which **function called**
- Could be added, but we'd need to use **name mangling**