Mälardalen University TEN1 DVA230 2016
School of innovation, design and engineering

Compiler theory

Thursday, August 18, 14:10 - 18:30
Teacher: Daniel Hedin, 021-107052 (15:00 - 17:00)
Allowed aids: Any books, lecture notes and written notes

The exam consists of 44 points distributed over 13 questions. Answers must be given in English or Swedish and should be clearly justified.
Grades:

3	22p
4	32p
5	38p
Max	44p

- Explain all solutions. A correctly explained solution with minor mistakes may render full points.
- Write clearly. Unreadable solutions will not be graded.
- Start each question on a new page and only write on one side of the page.
- Write down any assumptions you make.
- NOTE: each question can contain more than one part that needs an answer. Read carefully, and make sure you answer everything!

Lexical analysis - regular expressions (5p)

1) Write a regular expression that can handle one line string literals with escapes. Examples of valid string literals are `"string"`, and `"\""`, while `"str"ing"` and `"\\""` are not valid string literals. (3p)

Suggested solution: A string begins with a quotation mark `\"` and can contain any quoted character `\.` or any character that is not a quote or the beginning of an escape `[^"\]` and ends with a quotation mark `\"`.

Put together the result is `\" ( \. | [^\"\])* \"`

---

2) Is the following language regular or not? Justify your answer. (2p)

$$L ::= a \mid b \mid (L)$$

Suggested solution: No it is not - it matches parenthesis and we know that regular language cannot match parenthesis. Formally, a regular language is left-linear or right-linear, i.e., the single non-terminal on the right in the production rule should occur left-most or right-most. The above grammar is linear, but not left-linear or right-linear.

Grammars (8p)

3) What is an LL(1) grammar and why does it matter to recursive descent parser? (3p)

Suggested solution: An LL(1) grammar is on a form where each production rule can be uniquely selected by looking at the next token. For recursive descent parsers, in particular, this means that no backtracking will be necessary.

---

4) Is it possible to create a context-free grammar that ensures that a function is always called with the right number of arguments, i.e., where

```
void f(int a) { ... }

f(1);
f(2);
```

is syntactically correct, but where

```
void f(int a) { ... }

f(1);
f(1,2);
```

is not? Justify your answer. (2p)

Suggested solution: While context-free grammars are able to match parentheses, i.e., define languages like $a^n b^n$, they are unable to count in general. In fact, already $a^n b^n c^n$ is beyond the capabilities of a context-free grammar. In order to create a grammar that that works as suggested, the number of arguments would have to be counted and matched against the function definition. From $a^n b^n$ and the inability to handle $a^n b^n c^n$ we can see that it would work for the definition and *a single* function call, but for *two or more* function calls it would not be possible.

---

5) Let X range over variables. Rewrite the following grammar to be unambiguous without modifying the generated language. (3p)

$$E \ ::= \ \textit{define } X \textit{ in } E \mid E,E \mid X$$

Suggested solution: There are two sources of ambiguity. The first is in E,E where sequences longer than two can be parsed into several different trees. The second is in the interaction between *define X in E* and E,E where *define X in E*$,E$ could put E,E as part of the *define* or on the top level. We resolve the first by making sequences right associative, and the second by disallowing sequences inside *define*. The result is as follows

$$\begin{aligned} E \ &::= \ F \mid F,E \\ F \ &::= \ \textit{define } X \textit{ in } F \mid X \end{aligned}$$

It can be shown that the language generated by the new grammar is identical to the language generated by the old grammar.

Derivation trees and abstract syntax (6p)

6) When is a grammar unambiguous? (2p)

Suggested solution: A grammar is unambiguous when no string corresponds to two different (or more!) derivation trees.

---

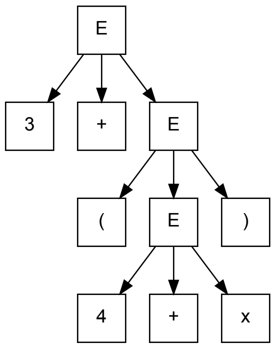
7) Give the derivation tree and the abstract syntax tree for `3 + ( 4 + x )` given the following grammar.

$$E \quad \rightarrow \quad VAR \mid NUM \mid E + E \mid (\; E \;)$$

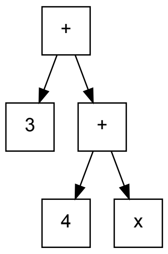
What is the difference between the two? (4p)

Suggested solution:

The derivation tree contains all information about the derivation, which includes syntax and syntactic categories that have been introduced only to guide the parser. The abstract syntax tree abstracts away from such syntactic details.



derivation tree



abstract syntax tree

Parsing (8p)

8) Write pseudo code for a recursive descent parser for the following language. (6p)

$$\begin{array}{lcl} E & \rightarrow & T\,U \\ T & \rightarrow & Ta \mid \lambda \\ U & \rightarrow & a \mid a\,b \end{array}$$

Suggested solution: The grammar is not LL(1) and there are two ways. Either use back tracking or rewrite to LL(1). To avoid having to write pseudo code for a back tracking recursive descent parser we rewrite the grammar to LL(1).

There are three LL(1) conflicts in the grammar. First, T is left recursive which causes a first/follow conflict for a and T, since T can be followed by a and is nullable. We solve this by making T right recursive. Second, there is another first/follow conflict for T and a. T can be followed by a since U can begin with a and T is nullable. Third there is a first/first conflict in U. We resolve the last two conflicts by letting T consume the a (which becomes mandatory, since U always match an a). The resulting grammar produces the same language and is on LL(1) form.

$$\begin{array}{lcl} S & \rightarrow & T\,U \\ T & \rightarrow & aT \mid a \\ V & \rightarrow & aV \mid \lambda \\ U & \rightarrow & b \mid \lambda \end{array}$$

The recursive descent parser follows the recursive structure of the grammar.

```
parseS = parseT; parseU
parseT = match(a); parseV
parseV = if next(a) then match(a); parseV; end
parseU = if next(b) then match(b); end
```

---

9) Can all context-free grammars be parsed with an SLR parser? Justify your answer. (2p)

Suggested solution: No, there are unambiguous context free grammars that cause SLR conflicts. Resolving such conflicts may prevent the parsing of strings that should be in the language.

Type Checking (8p) Consider a simple language of assignments and the corresponding type language

$$\begin{array}{lcl} s & \rightarrow & x = e \mid \tau\, x \mid s_1 ; s_2 \\ e & \rightarrow & n \mid b \mid x \\ \tau & \rightarrow & \mathit{int} \mid \mathit{bool} \end{array}$$

where x denotes variable names (identifiers), n denotes integers and b denotes booleans. Given the following pseudo code for a type system for the language

```
check(tenv, x = e):
  if (x not defined in tenv) error;
  t1 = tenv[x]; t2 = check e; if (t1 != t2) error

check(tenv, s1 ; s2):
  check(tenv, s1); check(tenv, s2)

check(tenv, t x):
  if (x defined in tenv) error; tenv[x] = t

check(tenv, n): return int
check(tenv, b): return bool
check(tenv, x):
  if (x not defined in tenv) error;
  return tenv[x]
```

10) Consider the following extension with code blocks

$$\begin{array}{lcl} s & \rightarrow & x = e \mid \tau\, x \mid s_1 ; s_2 \mid \{s\} \\ e & \rightarrow & n \mid b \mid x \\ \tau & \rightarrow & \mathit{int} \mid \mathit{bool} \end{array}$$

Modify the pseudo code to handle this extension in such a way that, e.g.,

```
int x; { bool x; x = true }; x = 5
```

and

```
int x; { x = 5 }
```

are type correct, but

```
{ bool y }; y = true
```

is not. (6p)

Suggested solution:
Rules are numbered to make solution of 11 more readable.

```
check(tenvs, x = e): [rule 1]
  t1 = lookup(x, tenvs);
  t2 = check e;
  if (t1 != t2) error

check(tenvs, s1 ; s2): [rule 2]
  check(tenvs, s1); check(tenvs, s2)

check(tenvs, { s }) : [rule 3]
  check({ } : tenvs, s)

check(tenv : tenvs, t x): [rule 4]
  if (x defined in tenv) error; tenv[x] = t

check(tenvs, n): return int [rule 5]
check(tenvs, b): return bool [rule 6]
check(tenvs, x): return lookup(x, tenv) [rule 7]

lookup(x, tenv)          = if x defined in tenv
                             then tenv[x]
                             else error

lookup(x, tenv : tenvs) = if x defined in tenv
                             then tenv[x]
                             else lookup(x, tevs)
```

---

11) Show that the following program is type correct. (2p)

```
int x; { bool x; x = true }; x = 5
```

Suggested solution: The solution below illustrates (it's unreadable!) why we use natural deduction style inferences and not this kind of inference.

```
check({}, int x; { bool x; x = true }; x = 5 ) = [rule 2]
  check({}, int x;);
  check({}, { bool x; x = true }; x = 5 ) = [rule 4]

  check({x : int}, { bool x; x = true }; x = 5 ) = [rule 2]
    check({x : int}, { bool x; x = true });
    check({x : int}, x = 5) = [rule 3]

    check({} : {x : int}, bool x; x = true);
    check({x : int}, x = 5) = [rule 2]

      check({} : {x : int}, bool x);
      check({} : {x : int}, x = true);
    check({x : int}, x = 5) = [rule 4]

      check({x : bool} : {x : int}, x = true);
    check({x : int}, x = 5) = [rule 1]

    check({x : int}, x = 5) [rule 1]. DONE
```

Code generation. (9p)

12) Consider the following small language with integer functions of one argument.

$$\begin{array}{lcl} s & \rightarrow & x \;=\; e \mid \textit{int } x \mid s_1 ; s_2 \mid \textit{function } f(x) \textit{ begin } s \textit{ end} \mid \textit{return } e \\ e & \rightarrow & n \mid x \mid e + e \mid f(e) \end{array}$$

Write pseudo code for a code generator that takes a program in the above language and produces Trac-42 stack code. (6p)

Suggested solution: We assume that programs are type correct. Further, we assume that `emit` gives each instruction an address and that the `f` in `emit(BSR f)` is linked in a postpass. Since function calls occur in the expressions we assume that a type correct program always returns.

Code generation for statements can be defined as follows. Note that the environment contains bindings for the argument and return offset, which must be reflected in the offset computation of variable declaration (first should be at offset -1).

```
gen(env, x = e) =
  emit(LVAL env[x](FP));
  gen(env, e);
  emit(ASSINT),

gen(env, int x) =
  emit(DECL 1),
  env[x] = -(size(env)-1)

gen(env, s1; s2) =
  gen(env, s1);
  gen(env, s2)

gen(env, function f(x) s) =
  emit(LINK);
  gen({x : 2, return : 3}, s)

gen(env, return e) =
  emit(LVAL 3(FP));
  gen(env, e);
  emit(ASSINT);
  emit(UNLINK);
  emit(RTS)
```

Code generation for expressions can be defined as follows.

```
gen(env, n) = emit(PUSHINT n)
gen(env, x) = emit(RVALINT env[x](FP))

gen(env, e1 + e2) =
  gen(env, e1);
  gen(env, e2);
  emit(ADD)

gen(env, f(e)) =
  emit(DECL 1);
  gen(env, e);
  emit(BSR f);
  emit(POP 1);
```

Key parts to get right for full points are 1) assignments, 2) variable declarations, 3) functions, 4) returns and 5) function calls.

---

13) Generate Trac-42 code for the following program. (3p)

```
int x;
function f(x) begin
  return x + x
end;
x = f(f(2))
```

Suggested solution:

Note that the generator assumes that the initial environment is of size 2. Thus, the top level code has to be generated in some dummy environment of size 2. Below is the code after linking the calls to f.

```
-- int x;
0:  DECL 1
-- f
1:  LINK
-- return x + x
2:  LVAL 3(FP)
3:    RVALINT 2(FP)
4:    RVALINT 2(FP)
5:    ADD
6:  ASSINT
7:  UNLINK
8:  RTS
-- x = f(f(2))
9:  LVAL -1(FP)
-- f(f(2))
10:   DECL 1
-- f(2)
11:     DECL 1
12:     PUSHINT 2
13:     BSR 1
14:     POP
15:   BSR 1
16:   POP
17: ASSINT
```
