[peːlaŋː]

plang 1.10

Användarmanual

Philip Arvidsson
Högskolan Borås (ASYAR14h)

INNEHÅLL

1. VAD ÄR SPRÅKET P?

Språket P är ett väldigt simpelt programspråk, dock Turing-komplett. Vi kan använda det till att räkna ut allt som går att räkna ut, rent teoretiskt. Det finns bara några få enkla operationer i språket. Alla operationer utförs sekventiellt, så det är viktigt att hålla reda på olika variablers värden för att kunna nyttja språket väl. Att språket är Turing-komplett innebär att alla kalkylerbara problem går att lösa med det, så länge rätt algoritm kan hittas. Språket används på följande vis:

Alla program börjar med nyckelordet **PROGRAM**, följt av namnen på input-variablerna (i det här fallet X1 och X2):

```
PROGRAM (X1, X2)
```

Ett program får ha som minst en input-variabel, men kan också ha ett godtyckligt antal input-variabler. Input-variablerna initialiseras av användaren som kör programmet.

Programmet slutar alltid med nyckelordet **RESULT**, följt av output-variabelns namn:

```
RESULT (X7)
```

Den specificerade output-variabeln är det resultat som programmet kommer att ge när det exekverat färdigt. Inuti program kan vi använda oss av tre olika typer av tilldelningssatser:

```
X1 := 0
```

Tilldelningssatsen ovan sätter X1 till värdet noll. Språket P tillåter egentligen inte tilldelning av andra värden än noll. Detta är väldigt lätt att komma runt genom att skriva en tilldelning med noll följt av ett antal **SUCC**-operationer. För enkelhetens

skull tillåter däremot plang tilldelning av godtyckliga värden, vilket innebär att plang accepterar ex. `X1 := 42`. Såhär fungerar **SUCC**:

```
X2 := SUCC(X3)
```

Tilldelningen ovan betyder 'variabeln X2 ska tilldelas värdet i variabeln X3 plus ett', dvs efteråt kommer X2 vara detsamma som X3+1. Självklart kan vi byta ut X3 mot X2 ovan, och få X2 = X2+1, dvs att vi ökar värdet på X2 med ett. **PRED** fungerar omvänt:

```
X5 := PRED(X5)
```

PRED innebär att vi minskar med ett. Satsen ovan innebär alltså $X5 = X5\text{-}1$, dvs att vi minskar X5 med ett. Om X5 redan är noll så är resultatet av **PRED**(X5) också noll, dvs **PRED** kan aldrig resultera i negativa tal. Programspråket P stödjer endast naturliga tal.

Vi kan även använda oss av while-loopar för att köra samma sekvens flera gånger. En while-loop har alltid en loop-variabel som testas mot noll. När variabeln når noll så upphör loopen. Beakta följande loop:

```
WHILE X4 != 0 DO
      # ...
END
```

Syntaxen innebär att allting som står mellan **DO** och **END** kommer upprepas tills dess att X4 blir noll. Så länge X4 inte är noll så kommer det inuti loopen att upprepas om och om igen. Inuti loopen är det därför viktigt att vi justerar värdet på (i det här fallet) X4, annars kommer loopen aldrig ta slut. Plang genererar en varning vid syntaxkontroll om loop-variabeln inte tilldelas något inuti loopen.

Därutöver stöds även kommentarer i plang. En kommentar påbörjas med en fyrkant (#). Alla tecken upp till nästa rad ignoreras då av tokeniseraren:

```
# Detta är en kommentar.
```

Ett enkelt program för att addera två naturliga tal kan exempelvis se ut såhär:

```
PROGRAM (X1, X2)
      WHILE X2 != 0 DO     # Så länge X2 inte är noll...
            X1 := SUCC(X1) # ... så ökar vi X1 med ett …
            X2 := PRED(X2) # ... och minskar X2 med ett.
      END
RESULT (X1)                # Resultatet finns i X1.
```

Genom att öka X1 och minska X2 tills dess att X2 är noll så "för vi över" värdet i X2 till X1, varefter X1 kommer innehålla sitt ursprungliga värde plus X2, dvs X1 + X2.

Titta i mappen examples för fler exempelprogram skrivna i P. Granska deras källkod och kompilera dem för att lära dig språket mer ingående!

BNF-notationen (se Backus-Naur-form under fördjupningsmaterial) för programspråket P ser ut på följande vis:

```
<program>           ::= PROGRAM (<variabel-lista>)
                            <sekvens>
                        RESULT (<variabel>)
<variabel-lista>    ::= <variabel>
                        | <variabel>, <variabel-lista>
<variabel>          ::= X<naturligt-tal>
<naturligt-tal>     ::= <siffra> | <siffra><naturligt-tal>
<siffra>            ::= 0|1|2|3|4|5|6|7|8|9
<sats>              ::= <tilldelnings-sats> | <while-sats>
<tilldelnings-sats> ::= <variabel> := <uttryck>
<uttryck>           ::= 0 | PRED(<variabel>) | SUCC(<variabel>)
<while-sats>        ::= WHILE <villkor> DO <sekvens> END
<villkor>           ::= <variabel> != 0
<sekvens>           ::= <sats> | <sats><kommentar>
                        | <sats><kommentar> <sekvens>
<kommentar>         ::= #<tecken-sekvens><rad-bryt>
<tecken-sekvens>    ::= <tecken> | <tecken><tecken-sekvens>
<tecken>            ::= godtyckligt tecken
<rad-bryt>          ::= '\n'
```

Genom att granska BNF-notationen kan vi avgöra vad som är giltig syntax i språket P. Notera att en variabel utgörs av bokstaven X följt av ett naturligt tal. Då det är de naturliga talen som skiljer variablerna åt, fungerar namnet mer som ett index än ett

egentligt variabelnamn. Detta innebär att X1, X01, X001 och så vidare alla refererar samma variabel. Nedan följer några olika sekvenser som kan vara användbara vid utveckling av P-program och dessutom visar att språket är långt mer mångsidigt än vid första anblick:

IF-SATSER

P har inga inbyggda språkelement för if-satser, men det går att använda while-satser listigt för att konstruera if-satser. Nedan följer några exempel, först i språket C, sedan av sina motsvarigheter i P.

```
if (X1 > 0) { /* Detta exekveras bara om X1 är större
                 än noll. */ }

X101 := SUCC(X1)
X101 := PRED(X101)
WHILE X101 != 0 DO
      X101 := 0
      # Detta exekveras bara om X1 är större än noll.
END
```

```
if (X1 == 0) { /* ... */ }

X101 := SUCC(X1)
X101 := PRED(X101)
X102 := SUCC(X101)
WHILE X101 != 0 DO
      X101 := 0
      X102 := 0
END
WHILE X102 != 0 DO
      X102 := 0
      # ...
END
```

```
if (X1 > X2) { /* ... */ }

X101 := SUCC(X1)
X102 := SUCC(X2)
X101 := PRED(X101)
```

```
X102 := PRED(X102)
WHILE X102 != 0 DO
      X101 := PRED(X101)
      X102 := PRED(X102)
END
WHILE X101 != 0 DO
      X101 := 0
      # ...
END
```

FOR-LOOP

En for-loop kan konstrueras enkelt med hjälp av en vanlig while-loop och if-sats:

```
for (X101 = X1; X101 <= X2; X101++) { /* ... */ }

X101 := SUCC(X1)
WHILE X101 != 0 DO
      X101 := PRED(X101)
      # ...
      X101 := SUCC(X101)
      X101 := SUCC(X101)
      X102 := SUCC(X101)
      X103 := SUCC(X2)
      X102 := PRED(X102)
      X103 := PRED(X103)
      WHILE X103 != 0 DO
            X102 := PRED(X102)
            X103 := PRED(X103)
      END
      WHILE X102 != 0 DO
            X102 := 0
            X101 := 0
      END
END
```

2. VAD ÄR PLANG?

Programmet plang kompilerar och kör programkod skriven i programspråket P. Genom att läsa in programkoden, dela upp den i s.k. tokens, analysera syntaxen genom att kontrollera att alla tokens ligger i acceptabel följd enligt språkets syntaxregler och generera ett s.k. abstrakt syntax-träd, så kan koden köras i en virtuell maskin. Vi kan dessutom generera assembly-kod utifrån ett AST, för att därefter kompilera P-programmet till en körbar exe-fil.

Processen från källkod till körbart AST eller kompilering ser ut på följande vis:

INLÄSNING

Källkoden laddas in från en textfil till en sträng i minnet. Vid det här steget görs inga felkontroller av källkodens innehåll, utan den läses endast in i sin befintliga form till plang.

LEXIKAL ANALYS / TOKENISERING

Strängen läses av tecken för tecken, varpå s.k. tokens genereras. Dessa tokens representerar olika språkliga element. Exempelvis blir nyckelord (så som **PRED** eller **SUCC**) enskilda tokens. På samma vis blir även := (tilldelning) en egen token, såväl som parenteser, heltal och variabler.

Vid tokenisering kastas "onödiga" delar av källkoden bort. Dessa innefattar radbrytningar, mellanslag och kommentarer. De behövs inte tas i beaktning eftersom de inte påverkar programmets logiska sammasättning, utan endast finns där för att underlätta för programmeraren.

Beakta följande programkod:

```
PROGRAM (X1)
        X1 := 0
RESULT (X1)
```

Om vi genomför lexikal analys (med programspråket P i åtanke) på koden ovan får vi följande lista av tokens:

1.	**PROGRAM**	*nyckelord*
2.	(	*vänster-parentes*
3.	X1	*variabel*
4.	)	*höger-parentes*
5.	X1	*variabel*
6.	:=	*tilldelning*
7.	0	*naturligt tal*
8.	**RESULT**	*nyckelord*
9.	(	*vänster-parentes*
10.	X1	*variabel*
11.	)	*höger-parentes*
12.		*filslut*

SYNTAXKONTROLL

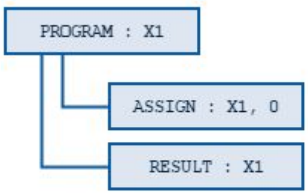
När vi har en lista med tokens måste vi verifiera att de ligger i rätt ordning. Till exempel vet vi att efter nyckelordet **PROGRAM** måste alltid en vänsterparentes komma. Genom att beakta programspråksreglerna för P kan vi avgöra om källkoden innehåller ett giltigt program eller inte.

Exempelprogrammet ovan råkar vara giltigt, men skulle vi exempelvis stöta på en variabel direkt efter **PROGRAM**, ett heltal direkt efter en variabel (utan en tilldelnings-token mellan), eller nyckelordet **WHILE** direkt efter := (tilldelning), så kan vi säga ifrån och anmärka på att programmet inte är giltigt, dvs att det inte är körbart. I det läget genereras ett kompileringsfel med tillhörande meddelande som

informerar om var och hur felet uppstått, så att en programmerare kan åtgärda det. Om programmet händelsevis är giltigt så går vi vidare till nästa steg och genererar ett abstrakt syntax-träd (AST).

ABSTRAKT SYNTAX-TRÄD

Trots att genereringen av AST (efter föregående steg) inte är särskilt komplex kan den vara svår att greppa. I själva verket handlar det om att generera noder för de olika nyckelorden och deras relationer. `PROGRAM`-noden är alltid rot-noden i trädet eftersom nyckelordet alltid ligger längst upp i ett P-program. En `WHILE`-nod har sitt innehåll som barn-noder. Programmet ovan skulle exempelvis resultera i att följande syntax-träd genererades:



Detta AST går att skicka in till en virtuell maskin som exekverar programmet, alternativt till en kodgenerator som kan kompilera trädet till assembly-kod.

KÖRNING / VIRTUELL MASKIN

Den virtuella maskinen kör programmet, en nod åt gången. Först börjar den med root-noden, sedan körs alla barn-noder, och så vidare. För varje nod så utför den virtuella maskinen den operation som noden representerar. Exempelvis kan den, när den påträffar en nod av typen `ASSIGN`, läsa ut vilken variabel det gäller och vilket

värde som variabeln ska tilldelas, för att därefter genomföra operationen. Resultatet blir att källkoden i första steget exekveras som ett program!

KOMPILERING

Istället för att köra programmet i den virtuella maskinen, kan plang generera assembly-kod och sedan assemblera koden till en körbar exe-fil. Plang genererar då en särskild sekvens instruktioner (assembly x86) för varje nod, för att sedan skicka vidare koden till fasm (flat assembler). Fasm genererar sedan en exe-fil som motsvarar programmet skrivet i programspråket P.

Kompilerade exe-filer exekverar långt mycket snabbare än vad den virtuella maskinen klarar av. Detta eftersom de körs direkt i datorns processor, medan den virtuella maskinen är som ett slags virtuell processor, som körs i den riktiga processorn.

3. BRUKSANVISNING

Använd plang genom att starta plang.exe. Programmet kan startas på flera olika vis beroende på som ska utföras.

METOD 1

Dra en fil innehållandes P-programkod till plang.exe med muspekaren. Plang startar då och laddar in källkodsfilen, varefter du ombeds ange inputvärdena till programmet som ska köras. Därefter körs programmet i plangs virtuella maskin, och resultatet presenteras i fönstret.

METOD 2

Dubbelklicka på plang.exe för att starta programmet. Du ombeds nu ange namnet på den fil du vill köra i den virtuella maskinen. Utöver detta fungerar allt på precis samma sätt som i metod ett.

METOD 3

Plang startas bäst från kommandoraden, då det ger störst möjlighet att konfigurera körningen. Plang stödjer då fem olika kommandon; `-asm`, `-compile`, `-printast`, `-runvm` och `-syncheck`. Kommandot `-asm` genererar assemblykod från P-kod, `-compile` kompilerar en källkodsfil till en körbar exe-fil, `-printast` genererar ett AST och skriver ut det i fönstret, `-runvm` kör den specificerade källkodsfilen (utan att kompilera den till exe-fil) i plangs virtuella maskin och `-syncheck` kontrollerar syntaxen hos en källkodsfil.

Utöver detta stödjer plang även debug-läge vid körning av den virtuella maskinen, samt möjlighet att stänga av optimeringar vid kompilering. För att

kompilera en fil anger vi ex. `plang -compile examples/multiply.p`. Detta kompilerar filen examples/multiply.p till en körbar exe-fil (examples/multiply.exe). Vill vi kompilera utan att optimera den genererade assembly-koden anger vi även `-no-opt`. `-no-opt` går även att använda med kommandot `-asm`.

```
plang -compile examples/multiply.p -no-opt
```

Kommandot ovan kompilerar filen examples/multiply.p till examples/multiply.exe utan att optimera den genererade assembly-koden.

Vill vi generera assembly-kod utan att kompilera den till en körbar exe-fil används istället kommandot `-asm`:

```
plang -asm examples/multiply.p
```

Kommandot ovan genererar optimerad assembly-x86-kod för den specificerade filen, utan kompilering till exe.

Med kommandot `-printast` skriver vi ut det abstrakta syntax-trädet för en P-programkodsfil. Kommandots syfte är i själva verket bara att ge en överblick av hur plang fungerar när det läser in och tokeniserar källkodsfiler.

```
plang -printast examples/multiply.p
```

Använd kommandot ovan för att skriva ut det abstrakta syntax-trädet för den specificerade filen.

För att köra en P-fil direkt i plangs virtuella maskin, utan att kompilera källkoden till en exe-fil, används kommandot `-runvm`. Specificerar vi dessutom `-debug` i slutet körs programmet i debug-läge, vilket underlättar felsökning av algoritmer skrivna i P.

```
plang -runvm examples/multiply.p -debug
```

Kommandot ovan laddar in och kör den specificerade filen i plangs virtuella maskin, dessutom i debug-läge.

Om vi bara vill kontrollera syntaxen hos en källkodsfil för att se om där finns några fel eller varningar, så använder vi kommandot `-syncheck`.

```
plang -syncheck examples/errors.p
```

Kommandot ovan laddar in källkodsfilen och kontrollerar om syntaxen innehåller några fel.

4. TEKNISKT

TOKENISERING

Tokeniseringsprocessen, eller den lexikala analysen, är i själva verket en väldigt enkel process. En sträng kan delas upp i olika tokens (symboler). Uppdelningen görs godtyckligt men i enlighet med språkets syntaxregler. Processen går till på följande vis:

1. Läs in ett tecken från källkoden.
2. Är tecknet kolon? I sådana fall kontroller vi vad nästa tecken är. Om det är ett likhetstecken så läser vi in det tecknet också. Vi har nu samlat på oss :=, vilket är en tilldelningstoken. Vi lägger till denna token i token-listan och går tillbaka till 1.
3. Är tecknet ett utropstecken? Då kontrollerar vi nästa tecken. Om det är ett likhetstecken så läser vi in det tecknet, och lägger till tokens för ekivalenstest (!=) i tokenlistan.
4. Om det är en bokstav eller siffra så lägger vi till tecknet i en sträng. Sedan läser vi in nästa tecken. Vi gör detta om och om igen, tills vi stöter på ett tecken som inte är en bokstav eller siffra. Därefter ser vi vad vi samlat på oss i strängen. Kanske är det "WHILE", "PRED" eller "RESULT", vilka ju är nyckelord. Kanske är det ett X följt av siffror, varpå vi vet att det är en variabel. Kanske är det enbart siffror i strängen, i vilket fall det ju är ett heltal. Vi markerar tokentypen och lägger till den i listan. Sedan tillbaka till steg 1.
5. Om källkoden är slut, så avslutar vi den lexikala analysen.

Genom att utföra processen ovan på hela strängen får vi en lista med tokens. Titta gärna i source/tokenizer.c för att se hur detta är implementerat i Plang. Funktionen `Tok_Tokenize()` står för det mesta av processen som beskrivs ovan.

ASSEMBLY X86

Då P är ett enkelt språk så är assemblykoden relativt enkel att generera. Det fungerar på ungefär samma sätt som exekvering i den virtuella maskinen, men istället för att direkt utföra operationerna så genereras assembly-x86-kod för varje nod. Variablerna är sparade som en vektor i minnet, så det handlar helt enkelt om att läsa och skriva till de olika variabelplatserna i vektorn. En tilldelningsoperation handlar exempelvis om att generera kod för att skriva det givna värdet till den givna variabelns plats i minnet:

```
# P-kod
X3 := 1337

; Assembly x86
MOV EBX, _Vars+3*4
MOV [EBX], DWORD 1337
```

Vi börjar med att se till så EBX pekar till rätt variabeladress. Grundadressen (för `X0`) är `_Vars`, 3 är den variabel vi vill tilldela (`X3`). Vi multiplicerar 3:an med 4, eftersom varje variabel är 4 bytes "bred". Därefter skriver vi tilldelningsvärdet till den adress dit EBX pekar. Sedan är tilldelningen genomförd. En SUCC-operation ser ut på följande vis:

```
# P-kod
X7 := SUCC(X2)

; Assembly x86
MOV EBX, _Vars+2*4
MOV EAX, [EBX]     ; Läs in värdet från X2
INC EAX            ; Öka med ett.
MOV EBX, _Vars+7*4
MOV [EBX], EAX     ; Skriv värdet till X7
```

PRED-operationen är något mer komplicerad eftersom värdet inte får gå under noll:

```
# P-kod
X4 := PRED(X6)
```

```
      ; Assembly x86
      MOV EBX, _Vars+6*4
      MOV EAX, [EBX]     ; Läs in värdet från X6
      DEC EAX            ; Minska med ett.
      JNS NotNeg         ; Om värdet inte är negativt, hoppa till
                         ; NotNeg.
      XOR EAX, EAX       ; Värdet var negativt, vi nollar det.
NotNeg:
      MOV EBX, _Vars+4*4
      MOV [EBX], EAX     ; Skriv värdet till X4.
```

Koden ovan kan skilja sig lite från den genererade koden beroende på optimeringar etc. Titta i source/asm.c för att förstå detta mer ingående!

5. FÖRDJUPNINGSMATERIAL

ABSTRAKTA SYNTAX-TRÄD (AST)

1. http://en.wikipedia.org/wiki/Abstract_syntax_tree

BACKUS-NAUR-FORM

2. http://en.wikipedia.org/wiki/Backus–Naur_Form

FILHANTERING I C

3. http://en.wikipedia.org/wiki/C_file_input/output

KOMPILERING

4. http://en.wikipedia.org/wiki/Compiler

LEXIKAL ANALYS/TOKENISERING

5. http://en.wikipedia.org/wiki/Lexical_analysis
6. http://en.wikipedia.org/wiki/Tokenization_(lexical_analysis)

TURING-FULLSTÄNDIGHET

7. http://en.wikipedia.org/wiki/Turing_completeness

VIRTUELLA MASKINER

8. http://en.wikipedia.org/wiki/Virtual_machine

X86 ASSEMBLY

9. http://en.wikipedia.org/wiki/X86_assembly_language
10. http://flatassembler.net/

6. KÄNDA BUGGAR

1. I kompilerade exefiler går det inte trycka enter för att ange inputvärden i de små inputrutorna vid start. Istället får man skriva in önskat inputvärde (heltal) och sedan stänga fönstret med hjälp av kryssrutan. Inputvärdet registreras ändå korrekt.

7. ÖVRIGT

Frågor, idéer, önskemål, tips och fel-/buggrapporter skickas per mail till philip@philiparvidsson.com.

Plang är fritt att modifiera, redistribuera m.m. Se license.txt innehållandes Simplified BSD License för mer information.

Särskilt tack till Mattias Eriksson för buggtestning, moraliskt stöd och de mest avancerade exempelprogrammen!

- Philip Arvidsson, 19:e februari 2015