Vysoké učení technické v Brně

Fakulta informačních technologií

Semestrální práce z předmětu Teorie programovacích jazyků

Roman Lukáš

2003-2004

1 Jazykové procesory (Language Processors)

1.1 Překladače a kompilátory

Překladač = program, který překládá zdrojový program (napsaný ve zdrojovém jazyce) na ekvivalentní cílový program (napsaný v cílovým jazyce).

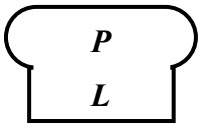
Kompilátor = program, který překládá program napsaný ve vyšším programovacím jazyce na ekvivalentní program napsaný v nižším programovacím jazyce.

Assembler = program, který překládá program napsaný ve strojovém jazyce (asembleru) na ekvivalentní strojový kód.

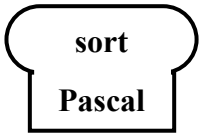
Dekompilátor = program, který překládá program napsaný v nižším programovacím jazyce na ekvivalentní program napsaný ve vyšším programovacím jazyce.

Disassembler = program, který překládá strojový kód na ekvivalentní program napsaný ve strojovém jazyce (asembleru).

Každý program může být vyjádřen v nějakém programovacím jazyce nebo přímo ve strojovém kódu, podle jehož instrukcí může již určitý procesor vykonávat jistou činnost. Pro znázornění, že nějaký program „*P*“ je implementován v jistém programovacím jazyce „*L*“ (nebo v jistém strojovém kódu), používáme tzv. náhrobní diagram (ve tvaru náhrobního kamene), který vypadá následovně:



Příklad: Program „sort“, který je implementován v programovacím jazyce Pascal znázorníme jako:

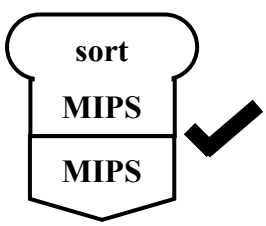


Pro znázornění, že na jistý procesor umí provést instrukce programu napsaného ve strojovém kódu *M*, budeme používat následující diagram:

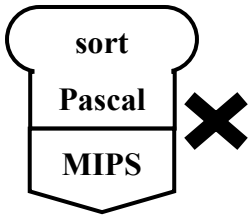
Příklad: Pokud jistý procesor umí provést instrukce program napsaného ve strojovém kódu MIPS, můžeme tuto skutečnost znázornit diagramem:

Aby mohl být jistý program proveden na daném procesoru, je potřeba, aby byl tento program implementován ve stejném strojovém kódu, jako je ten strojový kód, jehož instrukce umí procesor provést. To budeme znázorňovat postavením výše uvedených dvou diagramů na sebe.

Příklad:



Obr. a)

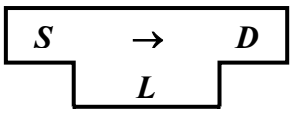


Obr. b)

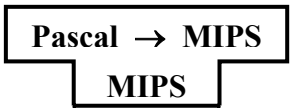
Obrázek Obr. a) znázorňuje situaci, kdy má být program „sort“, implementovaný ve strojovém kódu MIPS, proveden na procesoru, který umí provádět instrukce napsané ve strojovém kódu MIPS. Tato kombinace je zcela legální a program „sort“ tedy na tomto procesoru může být vykonán.

Obrázek Obr. b) znázorňuje situaci, kdy má být program „sort“, implementovaný v programovacím jazyce Pascal, proveden na procesoru, který umí provádět instrukce napsané ve strojovém kódu MIPS (a nic jiného). Tato kombinace je chybná a tento program „sort“ tedy na tomto procesoru nemůže být přímo vykonán.

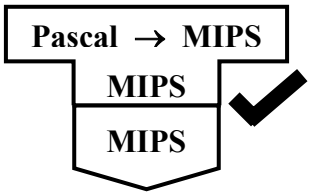
Jak již bylo zmíněno dříve, pomocí překladače lze přeložit zdrojový program a ekvivalentní cílový program. Nechť je překladač implementován v jistém programovacím jazyce „*L*“ (nebo v jistém strojovém kódu) a překládá programy napsané v jazyce „*S*“ na ekvivalentní programy napsané v jazyce „*D*“. Tuto skutečnost budeme znázorňovat následujícím diagramem:



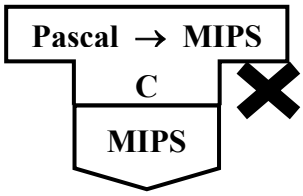
Příklad: Pokud překladač implementovaný ve strojovém kódu MIPS překládá programy napsané v programovacím jazyce Pascal na ekvivalentní programy napsané ve strojovém kódu MIPS, můžeme tuto skutečnost znázornit diagramem:



Pokud má být daný překladač použit na daném procesoru, je potřeba, byl implementován ve stejném strojovém kódu, jako je ten strojový kód, jehož instrukce umí procesor provést. To budeme značit následovně:



Obr. a)

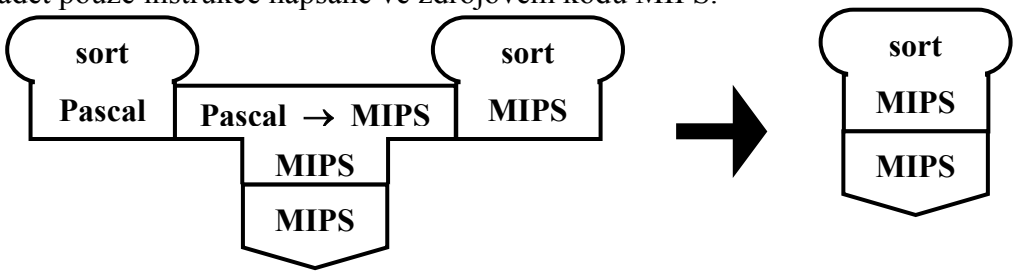


Obr. b)

Obrázek Obr. a) znázorňuje situaci, kdy má být překladač, implementovaný ve strojovém kódu MIPS, proveden na procesoru, který umí provádět instrukce napsané ve strojovém kódu MIPS. Tato kombinace je zcela legální. Naproti tomu obrázek Obr. b) znázorňuje situaci, kdy nemůže být daný překladač použit na daném procesoru.

Nyní se budeme zabývat samotným překladem již konkrétního programu. Grafické znázornění přeložení jednoho konkrétního zdrojového programu na ekvivalentní cílový program pomocí překladače, který tuto činnost může na daném procesoru provést, ukážeme v následujícím příkladě.

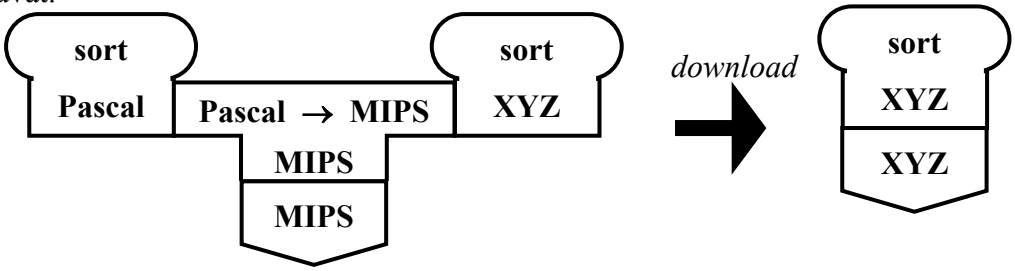
Příklad: Pomocí diagramů znázorněme přeložení programu sort napsaného v jazyce Pascal na ekvivalentní program napsaný ve strojovém kódu MIPS. Samotný překladač je implementován ve strojovém kódu MIPS, přičemž daný procesor umí provádět pouze instrukce napsané ve zdrojovém kódu MIPS.



Tento překlad je tedy možné na daném procesoru provést. Samotný program „sort“ napsaný v programovacím jazyce Pascal však nemůže být přímo proveden, ale přeložený program „sort“ do strojového kódu MIPS již proveden být může (viz. 2. část obrázku)

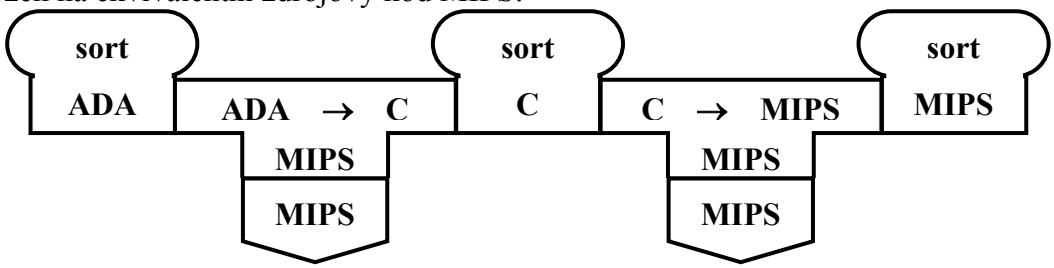
Křížový překladač = překladač, který běží na jednom procesoru, ale generuje kód pro jiný procesor.

Následující obrázek ilustruje přeložení programu sort na ekvivalentní zdrojový kód XYZ, který ovšem může běžet pouze na jiném procesoru, který umí tyto instrukce vykonávat.



Dvojstupňový překladač = překladač, který je složen ze dvou dílčích překladačů. První dílčí překladač přeloží zdrojový program na ekvivalentní meziprogram, který je zpracován druhým dílčím překladačem.

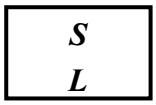
Následující obrázek ilustruje dvojstupňový překladač, který nejprve přeloží program „sort“ v programovacím jazyce ADA na ekvivalentní program v jazyce C a ten je pak přeložen na ekvivalentní zdrojový kód MIPS:



1.2 Interprety

Interpret = program, který je schopen přímo vykonat program napsaný ve zdrojovém jazyce.

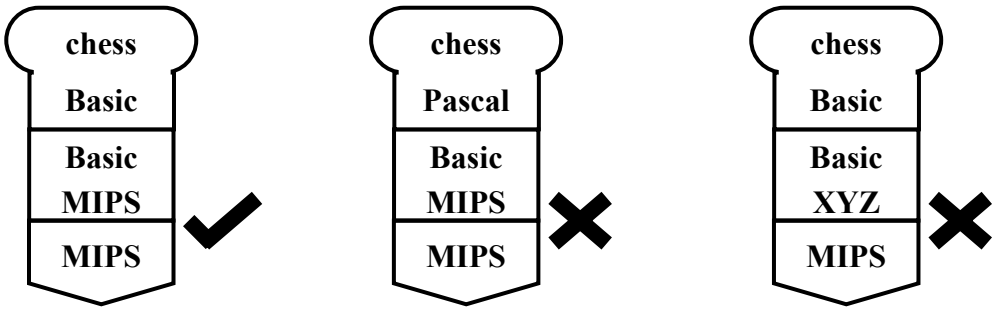
Nechť interpret je implementován v jistém programovacím jazyce „*L*“ (nebo v jistém strojovém kódu) a umí vykonat programy napsané v jazyce „*S*“. Tuto skutečnost budeme znázorňovat následujícím diagramem:



Aby mohl být jistý program proveden pomocí daného interpretu, musí být splněny následující 2 podmínky:

1) Zdrojový program musí být napsán v takovém programovacím jazyce, který je schopen interpret vykonat
2) Interpret musí být implementován v takovém strojovém kódu, jehož instrukce umí daný procesor vykonávat

Příklad:

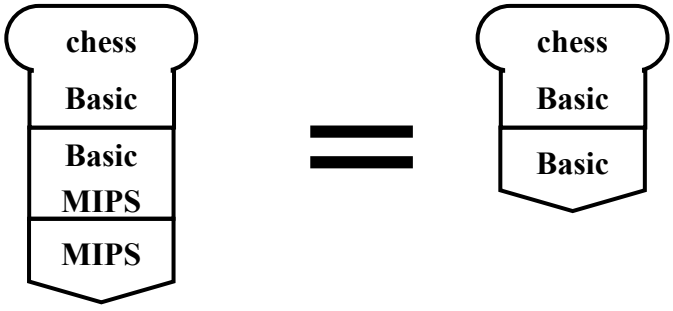


Obr. a) Obr. b) Obr. c)

Pouze obrázek Obr. a) ilustruje případ, kdy program „chess“ implementovaný v jazyce Basic může být proveden. Program „chess“ na obrázku Obr. b) nemůže být proveden, neboť program „chess“ je implementován v jiném programovacím jazyce, než je schopen interpret provést. Obr. c) ilustruje případ, kdy interpret nemůže být spuštěn na daném procesoru, tedy ani zde program „chess“ nemůže být proveden.

1.3 Kompilátor versus interpret

Interpret, který umí vykonat programy napsané v jazyce „*S*“ v jistém smyslu simuluje skutečnost, že daný počítač umí přímo provádět instrukce v jazyce „*S*“:



Hlavní nevýhoda překladače: Při každé drobné změně zdrojového programu je nutné opět zdrojový program přeložit na cílový program, který může být proveden. To pro dlouhé programy může trvat velmi dlouho.

Hlavní nevýhoda interpretu: Interpret musí při každém provedení zdrojového programu převést tyto instrukce na ekvivalentní instrukce strojového kódu, které mohou být provedeny. To může až 100x zpomalit činnost provedení programů.

Obě tyto nevýhody lze omezit kompromisem mezi kompilátorem a interpretem-tzv. interpretovaným kompilátorem.

1.4 Interpretovaný kompilátor

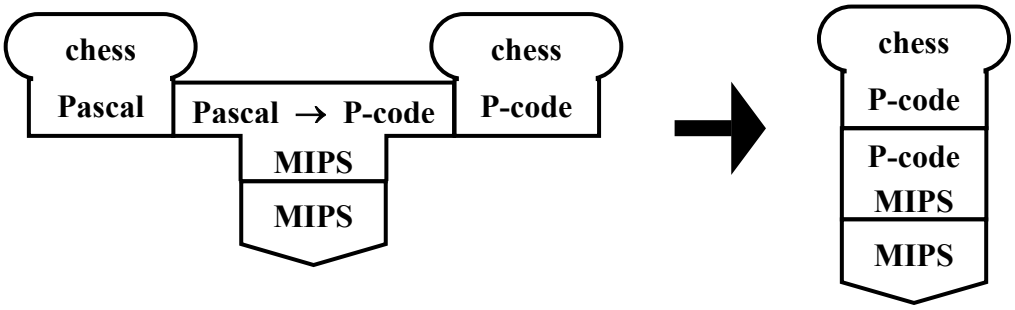
Jak již bylo zmíněno dříve, použití kompilátoru i interpretu má své nevýhody. Interpretovaný kompilátor do jisté míry tyto nevýhody odstraňuje. Pracuje následovně:

1) Zdrojový program je přeložen na ekvivalentní cílový program, který obsahuje instrukce ve „vhodném“ formátu.
2) Pomocí interpretu je cílový program proveden.

Poznámka: „vhodný“ formát instrukcí je takový, do kterého lze snadno převést instrukce zdrojového programu a navíc musí být tyto instrukce rychle převeditelné na instrukce strojového kódu, aby činnost interpretu mohla probíhat rychle.

Následující obrázek ilustruje přeložení programu „chess“ do tzv. P-code, který je dále interpretem pro P-code proveden.

P-code je jazyk, který je velmi blízký Pascalu. Obsahuje instrukce, které korespondují s Pascalovskými příkazy a navíc lze velmi rychle přeložit do strojového kódu.

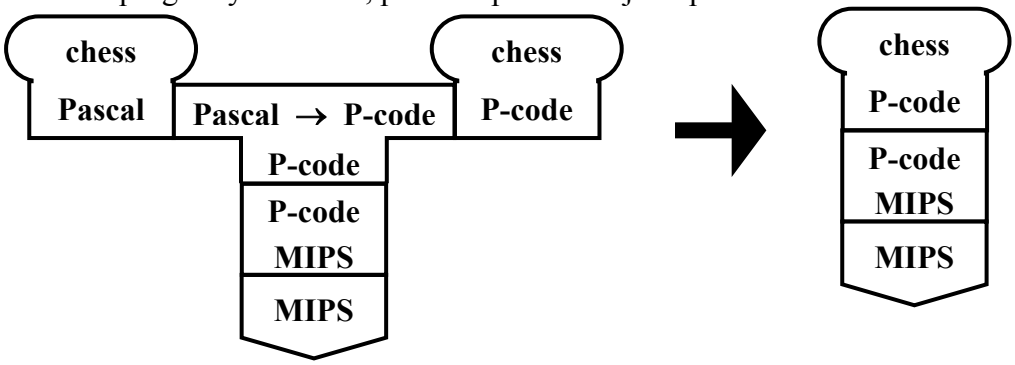


1.5 Přenositelnost překladačů

Přenositelnost programů je měřena v procentech. Její hodnoty se pohybují v rozsahu 0% až 100%. Tato veličina určuje, jak moc je potřeba program modifikovat, aby mohl být vykonán na jiném procesoru. 0% přenositelnost je pro programy psané ve strojovém jazyce. Tyto programy je potřeba pro jiný procesoru úplně předělat. Daleko větší přenositelnost je pro programy psané ve vyšším programovacím jazyce. V ideálním případě je pak přenositelnost 100%.

Problémem je však zajištění přenositelnosti překladačů. Pro vyšší přenositelnost překladače se tedy nabízí možnost nepsat překladač přímo na úrovni strojového jazyku, ale na nějaké vyšší úrovni. Pak ale musí být přeložen jiným překladačem, což tedy původní problém neřeší. Druhá možnost je, aby překladač vykonávat svoji činnost pomocí interpretu, což zase velice zpomalí jeho činnost.

Následující obrázek ilustruje použití překladače, který překládá programy v Pascalu na ekvivalentní programy v P-code, přičemž překladač je implementován v P-code:

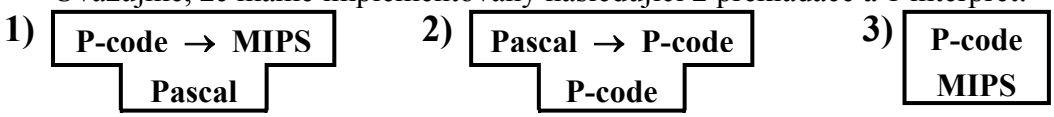


Poznámka: Pro přenos celé soustavy na jiný procesor stačí pouze modifikovat interpret pro P-code.

1.6 Samopřekládání přenositelných překladačů

V minulé kapitole byl ukázán přístup, jak lze vytvořit přenositelný překladač. Velkým problémem byla nutnost stálého užití interpretu, který značně zpomaloval činnost překladače. Nyní si ukážeme složitější postup zvaný samopřekládání přenositelného překladače, který tento problém odstraní. Metodu ukážeme na příkladě. Předpokládejme, že máme vytvořit přenositelný překladač, který překládá programy napsané v programovacím jazyce Pascal na ekvivalentní strojový kód MIPS.

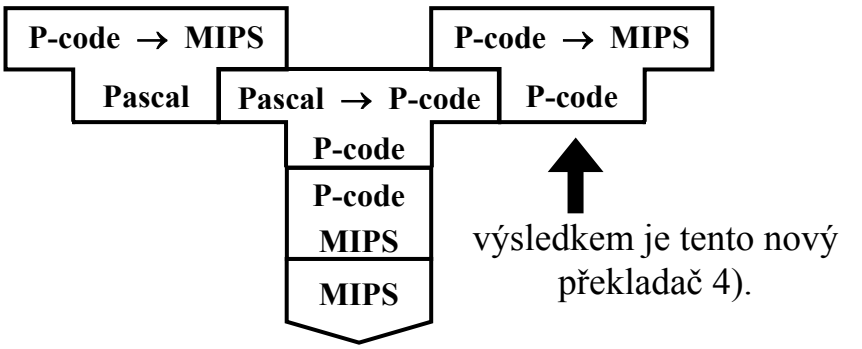
Uvažujme, že máme implementovány následující 2 překladače a 1 interpret:



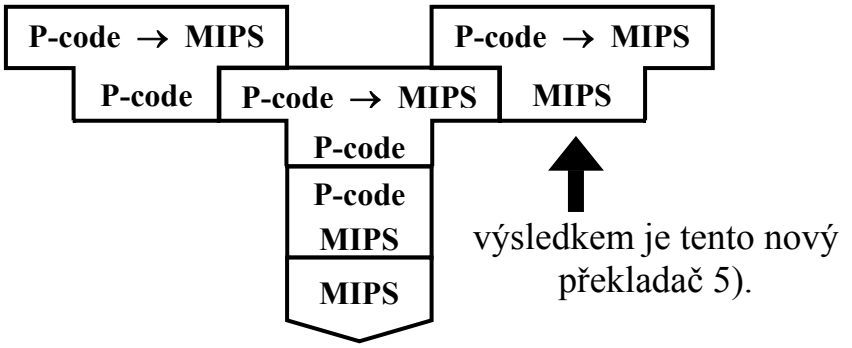
Tyto dva překladače mají jistě velkou přenositelnost. Malou přenositelnost má pouze interpret pro P-code, který lze ale snadno předělat pro jiní procesor (stejně tomu bylo i v minulé kapitole).

Kompilace překladače pak může probíhat v následujících fázích:

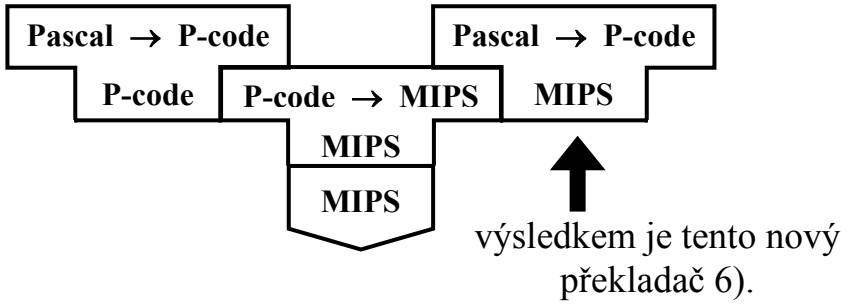
1. Pomocí překladače 2) a interpretu 3) přeložíme překladač 1) na ekvivalentní překladač implementovaný v P-code. Označme jej jako překladač 4):



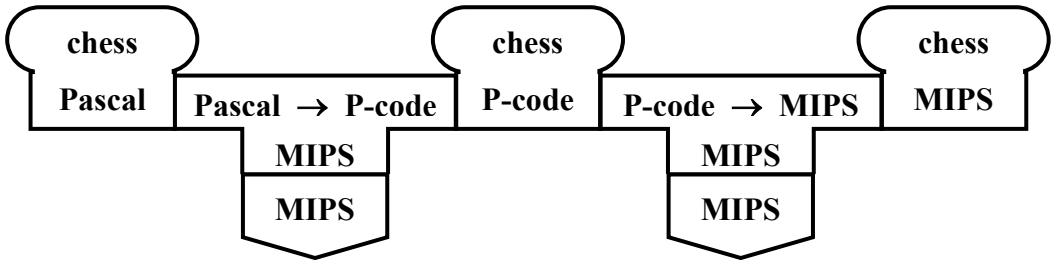
2. Nyní se s překladačem 4) provede tzv. samopřeložení za pomocí interpretu 3). (tj. pomocí tohoto překladače přeložíme sami sebe). Tím vznikne nový překladač, který je implementován ve strojovém kódu MIPS. Označme jej 5).



3. Pomocí překladače 5) přeložíme překladač 4) na ekvivalentní překladač implementovaný ve strojovém kódu MIPS. Označme jej 6):



Nyní máme vytvořen dvojstupňový překladač, který může například program „chess“ přeložit následujícím způsobem:



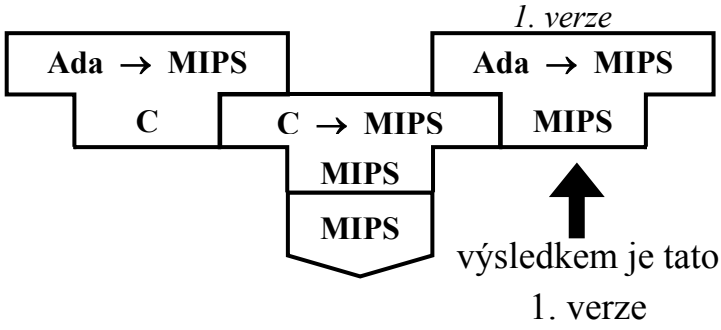
Tato metoda je velice zdlouhavá na vytvoření požadovaného překladače, avšak překlad potom probíhá přímo bez použití jakéhokoliv interpretu. Programy pak běží mnohem rychleji.

1.7 Úplné samo-překládání překladačů

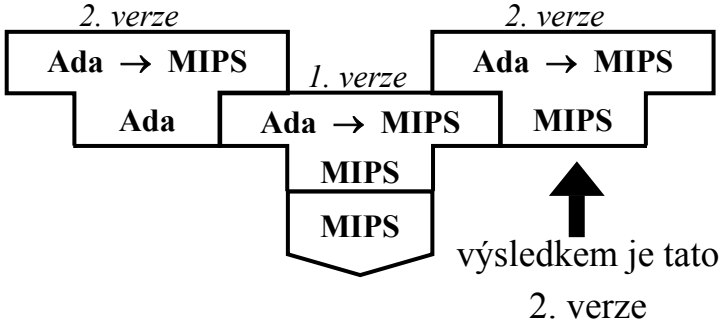
V minulé kapitole bylo ukázáno, jak lze udělat přenositelný překladač. Hlavní nevýhodou této metody je, že při drobné úpravě překladače je potřeba provést jeho kompilaci ve všech uvedených fází, což může být časově náročné.

Toto lze vyřešit následující metodou která má název úplné samo-překládání. Metodu ukážeme na příkladě. Uvažujme, že chceme vytvořit překladač, který překládá programy napsané v programovacím jazyce ADA na strojový kód MIPS. Vývoj překladače pak probíhá v několika fázích:

1. V první fází napíšeme tento překladač např. v jazyce C a necháme jej zkompilovat kompilátorem pro jazyk C. Výsledkem je 1. verze překladače, který bude v dalších fázích dále vyvíjen:



2. Abychom odstranili závislost na kompilátoru jazyka C, budeme implementovat 2. verzi překladače v samotném jazyce Ada. Tuto verzi potom zkompilujeme pomocí 1. verze překladače získané z první fáze:



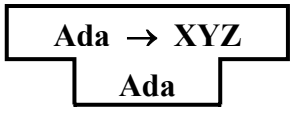
3. Následující verze již vznikají tak, že se pouze upravuje překladač, který je implementován v jazyce Ada. Po této úpravě je překladač zkompilován překladačem předchozí verze. Tato fáze může podle potřeby iteračně proběhnout několikrát.

1.8 Poloviční samopřekládání překladačů

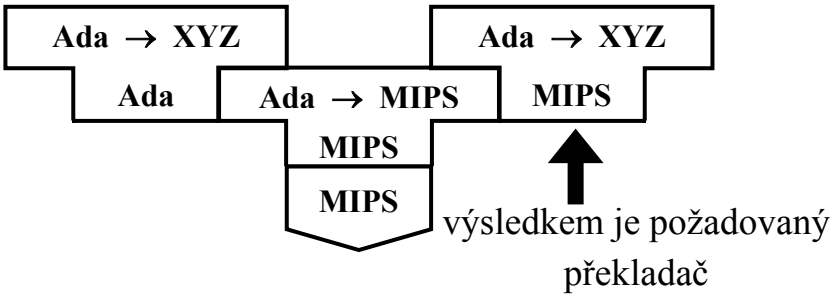
Předpokládejme, že jsme například metodou popsanou v minulé kapitole vytvořili kompilátor pro jazyk Ada. Máme tedy následující překladač:

Dále nechť je požadováno, aby například vytvořený kompilátor překládal programy na instrukce nějakého procesoru XYZ (viz. křížový překladač). Toho můžeme jednoduše dosáhnout tzv. polovičním samopřekládáním překladačů, které probíhá v následujících fázích:

1. V první fázi vytvoříme následující nový překladač:



2. Tento nově vytvořený překladač zkompilujeme již vytvořeným překladačem:



2 Syntaktická analýza

2.1 Popis syntaxe programovacího jazyka

Bezkontextová gramatika

K popisu syntaxe programovacího jazyka lze použít bezkontextovou gramatiku. Bezkontextová gramatika obsahuje následující 4 elementy:

- Konečnou množinu terminálních symbolů (terminálů). Tyto symboly jsou základní stavební prvky programovacího jazyka. Typické terminály jsou **:=**, **if** a +.
- Konečnou množinu neterminálních symbolů (neterminálů). Těmito symboly jsou označeny jisté abstraktní třídy fráze programovacího jazyka, které jsou dále rozgenerovány. Typické netermilály jsou **Program**, **Command**, **Expression** a **Declaration**.
- Startovací symbol, který je jeden z neterminálů, reprezentující frázi celého programu. Typický startovací symbol je neterminál **Program**.
- Konečnou množina přepisovacích pravidel. Přepisovací pravidla popisují, jak lze jednotlivé fráze dále rozgenerovat.

BNF (Backus-Naurova Forma)

Bezkontextovou gramatiku můžeme popsat pomocí BNF. V BNF mají přepisovací pravidla tvar:

$$N ::= \alpha,$$

kde N je neterminál a α je řetězec terminálních a neterminálních symbolů. Pravidlo lze použít tak, že již ve vygenerované frázi můžeme výskyt neterminálu N nahradit řetězcem α. Různá přepisovací pravidla postupně aplikujeme na vznikající řetězec, dokud nedostaneme řetězec obsahující pouze terminální symboly. Daná Gramatika v BNF pak generuje všechny řetězce složené pouze z terminálních symbolů, které lze tímto způsobem vygenerovat ze startovacího symbolu.

EBNF (Extended BNF)

EBNF je rozšíření BNF tak, aby mohly být programy popsány efektivněji. EBNF používá k popisu regulární výrazy, proto je dále uvedena tabulka, která je definuje:

Regulární výrazy:

Regulární výraz	Tento regulární výraz popisuje
ε	prázdný řetězec
A	pouze řetězec obsahující jeden symbol *a*
EF	všechny řetězce, které vzniknou konkatenací řetězce, který popisuje RV *E* s řetězcem, který popisuje RV *F*
E | *F*	Všechny řetězce, které popisuje RV *E* nebo RV *F*
E^*	Všechny řetězce, které vzniknou zřetězením 0 nebo více řetězců, které generuje RV *E*
(*E*)	Všechny řetězce, které generuje RV *E*

EBNF je kombinace BNF a regulárních výrazů následovně: Každé EBNF přepisovací pravidlo je ve tvaru:

$$N ::= E,$$

kde N je neterminál a E je regulární výraz nad množinou terminálních a neterminálních symbolů. Pravidlo lze použít tak, že již ve vygenerované frázi můžeme výskyt neterminálu N nahradit libovolným řetězcem obsahující terminální a neterminální symboly, který popisuje regulární výraz E.

Příklad:

Expression ::= primary-Expression (Operator primary-Expression)*

primary-Expression ::= identifier | **(** Expression **)**

Identifier ::= **a** | **b** | **c** | **d** | **e**

Operator ::= **+** | **–** | ***** | **/**

Tato gramatika v EBNF generuje aritmetické výrazy s proměnnými a, b, c, d, e. Například: a+b, a*(b+c)/d, …

2.2 Implementace syntaktické analýzy – rekurzivní sestup

Předpokládejme, že daný programovací jazyk máme popsaný pomocí bezkontextové gramatiky v EBNF. Nejjednodušší implementace provedení syntaktické analýzy je tzv. rekurzivní sestup.

Předpokládejme, že pro provedení syntaktické analýzy pomocí rekurzivního sestupu máme pro zajištěno:

- Proměnná **`CurrentToken`** je globální (lze ji tedy využít ve všech dalších procedurách) a vždy obsahuje ze zdrojového programu aktuální token.
- Dále uvažujme proceduru: **`procedure accept(t: tToken)`** – procedura prověří, zda má aktuální token (tj. proměnná **`CurrentToken`**) stejnou hodnotu, jako má parametr procedury. Pokud ano, může probíhat dále syntaktická analýza, přičemž do proměnné **`CurrentToken`** je načten následující token, jinak je hlášena syntaktická chyba.

Nyní můžeme naimplementovat procedury provádějící syntaktickou analýzu, které jsou již pro různé gramatiky odlišné. Každý neterminál gramatiky je reprezentován právě jedou korespondující procedurou. Úkolem každé z těchto procedur pokrýt část programu z korespondujícího nonterminálu. Toho je docíleno rekurzivním voláním těchto procedur. Implementace těchto procedur pro konkrétní gramatiku v EBNF bude popsána později. Hlavní tělo programu obsahuje spuštění té procedury, která koresponduje se startovacím symbolem.

Definice množiny starters:

Nechť E je regulární výraz nad terminálními a neterminálními symboly jisté gramatiky G. Potom pro regulární výraz E množina *starters*, značena *starters*[E], obsahuje právě ty terminální symboly, kterými mohou začínat řetězce generované z E. Přesnou rekurzivní definici množiny *starters*[E] ukazuje následující tabulka:

***starters*[*E*] je definováno:**	**Podmínky:**
starters[ε] = {}	-
starters[*a*] = {*a*}	*a* je terminální symbol
starters[*N*] = *starters*[*E*]	*N* je neterminální symbol, *N* ::= *E* je přepisovací pravidlo
starters[*EF*] = *starters*[*E*]	*E* negeneruje ε
starters[*EF*] = *starters*[*E*] ∪ *starters*[*F*]	*E* generuje ε
starters[*E*|*F*] = *starters*[*E*] ∪ *starters*[*F*]	-
starters[E^*] = *starters*[*E*]	-
starters[(*E*)] = *starters*[*E*]	-

Nyní si ukážeme, jak lze pro libovolný neterminál *N* implementovat korespondující proceduru *ParseN*, jak již bylo zmíněno výše. Předpokládejme, že k neterminálu *N* koresponduje pouze přepisovací pravidlo *N* ::= *E*. Pokud k neterminálu *N* koresponduje více přepisovacích pravidel *N* ::= E_1, *N* ::= E_2, …, *N* ::= E_k, můžeme jej sjednotit do jednoho pravidla tvaru *N* ::= *E*, kde $E = E_1|E_2|...|E_k$. Proceduru *ParseN* potom můžeme implementovat následovně:

```
procedure ParseN;
begin
   Parse E
end;
```

kde *Parse E* je dále rozepsáno podle následujících rekurzivních pravidel:

Parse E	**Implementace „*Parse E*“**
E = ε	prázdný příkaz
E = *a*, kde *a* je terminál	`Accept(a)`
E = *A*, kde *A* je neterminál	`ParseA`

E = E_1E_2	**begin**
	  parse E_1;
	  parse E_2
	end

E = $E_1\|E_2$	**if CurrentToken in** *starters*[E_1] **then**
	  parse E_1
	else
	if CurrentToken in *starters*[E_2] **then**
	  parse E_2
	else
	`write('syntaktická chyba');`

E= E_1^*	**while CurrentToken in** *starters*[E_1] **do** ***parse*** E_1

Poznámka: Pokud je v tabulce na místě implementace uvedeno *parseX*, znamená to, že kód této části příkazu má být vygenerován rekurzivně podle pravidel tabulky.

Příklad:

Vygenerujme kód provádějící syntaktickou analýzu pro nonterminál *Command*, kterému odpovídá přepisovací pravidlo:

Command ::= single-Command (; single-Commannd)*

Generování kódu pro toto přepisovací pravidlo postupně ukážeme v jednotlivých fázích:

Základ vygenerované procedury:

```
procedure ParseCommand;
begin
   parse single-Command (; single-Commannd)*
end;
```

- ***parse*** *single-Command (; single-Commannd)** je dále rozgenerováno na:

```
ParseSingleCommand;
```
parse *(; single-Commannd)**

- ***parse*** *(; single-Commannd)** je dále rozgenerováno na:

```
while CurrentToken in [;] do
```
  parse *(; single-Commannd)*

- ***parse*** *(; single-Commannd)* je již finálně rozgenerováno na:

```
begin
  accept(';');
  ParseSingleCommand;
end
```

Výsledná implementace kódu pro dané pravidlo je:

```
procedure ParseCommand;
begin
  ParseSingleCommand;
  while CurrentToken in [';'] do
  begin
    accept(';');
    ParseSingleCommand;
  end
end;
```

Poznámka: `ParseSingleCommand` zavolá proceduru tohoto názvu, která byla implementována analogicky.

Může tato metoda zhavarovat?

Příklad 1: Vygenerujme kód provádějící syntaktickou analýzu pro nonterminál *SingleCommand*, kterému odpovídá přepisovací pravidlo:

SingleCommand ::= Identifier := Expression | Identifier (Expresion)

Výsledný vygenerovaný kód:

```
procedure ParseSingleCommand;
begin
  if CurrentToken in ['Identifier'] then
  begin
    accept('Identifier');
    accept(':=');
    ParseExpression;
  end
  else
  if CurrentToken in ['Identifier'] then
  begin
    accept('Identifier');
    accept('(');
    ParseExpression;
    accept(')');
  end
  …
end;
```

Příklad 2: Vygenerujme kód provádějící syntaktickou analýzu pro nonterminál *Command*, kterému odpovídá přepisovací pravidlo:

Command ::= single-Command | Command; single-Command

Výsledný vygenerovaný kód:

```
procedure ParseCommand;
begin
  if CurrentToken in //starters [single-Command] then
    ParseSingleCommand;
  else
  if CurrentToken in //starters [single-Command] then
  begin
    ParseCommand;
    accept(';');
    ParseSingleCommand;
  end
  …
end;
```

Oba Kódy jsou zřejmě chybné, neboť nikdy se nemůže provést druhý příkaz „if“

Jak jsme si ukázali v předcházejících dvou příkladech, metoda může za určitých podmínek zhavarovat. Aby byla metoda použitelná, musí pro všechna přepisovací pravidla $N ::= E$ platit:

1. pokud $E = E_1|E_2$, potom $starters[E_1] \cap starters[E_2] = \varnothing$
2. pokud $E = E_1E_2$, přičemž E_1 generuje ε, potom $starters[E_1] \cap starters[E_2] = \varnothing$

Gramatiky splňující tyto podmínky se potom nazývají LL(1).

V některých případech ale můžeme vhodně upravit pravidla tak, aby výše uvedené podmínky splňovaly. Jedná se především o dvě následující metody:

Metoda faktorizace

Jedná se o převod pravidla tvaru $N ::= FE_1G| FE_2G$ na tvar $N ::= F(E_1|E_2)G$.

Příklad: Vygenerujme kód provádějící syntaktickou analýzu pro nonterminál *SingleCommand*, kterému odpovídá přepisovací pravidlo:

SingleCommand ::= Identifier := Expression | Identifier (Expresion)

Nejdříve musíme upravit pravidlo na tvar:

SingleCommand ::= Identifier(*:= Expression | (Expresion)*)

A pro takto upravené pravidlo již můžeme vygenerovat správný kód:

```
procedure ParseSingleCommand;
begin
  accept('Identifier');
  if CurrentToken in [':='] then
  begin
    accept(':=');
    ParseExpression;
  end
  else
  if CurrentToken in ['('] then
  begin
    accept('(');
    ParseExpression;
    accept(')');
  end
  else
    write('syntaktická chyba');
end;
```

Metoda odstranění levé rekurze

Jedná se o převod pravidla tvaru $N ::= E \mid NF$ na tvar $N ::= E(F)^*$.

Příklad: Vygenerujme kód provádějící syntaktickou analýzu pro nonterminál *Command*, kterému odpovídá přepisovací pravidlo:

Command ::= single-Command | Command; single-Command

Nejdříve musíme upravit pravidlo na tvar:

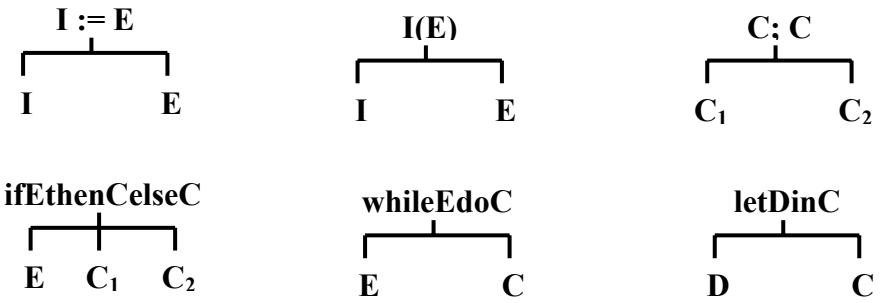
Command ::= single-Command (*; single-Command*)*

A pro takto upravené pravidlo již můžeme vygenerovat správný kód:

```
procedure ParseCommand;
begin
  ParseSingleCommand;
  while CurrentToken in [';'] do
  begin
    accept(';');
    ParseSingleCommand;
  end
end;
```

2.3 Vytvoření abstraktního syntaktického stromu

V minulé kapitole bylo pomocí rekurzivního sestupu rozhodnuto, zda daná gramatika generuje zdrojový program, či nikoliv. Později překladač ale bude muset generovat cílový kód, je tedy potřeba „vhodným způsobem“ reprezentovat zdrojový program, aby mohl být efektivně přeložen. Vhodnou reprezentací je např. abstraktní syntaktický strom. Jedná se obecně o *n*-nární strom, který explicitně reprezentuje strukturu zdrojového programu. Příklady elementů abstraktního syntaktického stromu:



Abstraktní syntaktický strom lze vytvářet již v průběhu syntaktické analýzy. Je pouze potřeba vhodně modifikovat procedury pro rekurzivní sestup, které současně kontrolují správnou syntax programu a současně vytváří abstraktní syntaktický strom.

Předpokládejme, že máme implementovány následující funkce pro vytvoření obecně *n*-nárního stromu:

```
function leafAST(tag: ASTTag; attr: TokenAtribute):AST;
```

- funkce vytvoří listový-uzel pro terminál typu tag s atributem attr.

```
function nullaryAST(tag: ASTTag):AST;
```

- funkce vytvoří uzel typu tag neobsahující žádným podstrom.

```
function unaryAST(tag: ASTTag; child1: AST):AST;
```

- funkce vytvoří uzel typu tag s jedním podstromem child1.

```
function binaryAST(tag: ASTTag; child1: AST;
                                child2: AST;):AST;
```

- funkce vytvoří uzel typu tag s dvěma podstromy child1, child2.

```
function ternaryAST(tag: ASTTag; child1: AST;
                                 child2: AST;
                                 child3: AST;):AST;
```

- funkce vytvoří uzel typu tag s třemi podstromy child1, child2, child3.

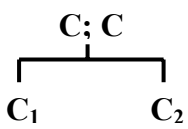
…

Procedury pro rekurzivní sestup upravíme tak, že jim přidáme parametr, který bude reprezentovat právě vytvářený uzel. Dále do těla procedury přidáme takovou sémantickou akci, která vygeneruje podstrom, který koresponduje tomuto pravidlu.

Příklad: Uvažujme následující proceduru:

```
procedure ParseCommand;
begin
  ParseSingleCommand;
  while CurrentToken=';' do
  begin
    accept(';');
    ParseSingleCommand;
  end
end;
```

Nyní zmodifikujme proceduru tak, aby v průběhu syntaktické analýzy vytvořila následující část stromu:



Zmodifikovaná procedura bude vypadat:

```
procedure ParseCommand(var comAST: AST);
var c1, c2: AST
begin
  ParseSingleCommand(c1AST);
  while CurrentToken=';' do
  begin
    accept(';');
    ParseSingleCommand(c2AST);
    c1 := binaryAST('C;C', c1, c2);
  end
  comAST := c1;
end;
```

Procedura zavolá rekurzivně vytvoření abstraktního syntaktického stromu pomocí dalších procedur (nebo i pomocí sebe sama). Výsledné získané stromy s kořeny c1 a c2 pak spojí pomocí podle výše uvedeného obrázku v jeden, jehož kořen je obsažen v proměnné comAST**.** Analogicky bychom upravili ostatní procedury.

3 Kontextová analýza

Kontextová analýza provádí kontextovou kontrolu správnosti programu, která nemohla být provedena v rámci syntaktické analýzy. Jedná se především o následující 2 kontroly:

Identifikace = Provedení kontroly, zda všechny proměnné byly deklarované. Dále je vytvořena vazba právě mezi deklarací proměnné a jejím výskem v programu.

Typová kontrola = Provedení kontroly, zda jsou skutečné typy výrazů stejné s typem, který je očekáván.

3.1 Identifikace

Vytvoření vazby mezi deklarací proměnné a jejím výskem v programu se většinou provádí pomocí tabulky identifikátorů. Implementace takové tabulky je různě složitá. Záleží především na tom, jak je ve zdrojovém jazyce navržena bloková struktura. Rozlišují se především následující 3 typy blokových struktur:

- Monolitická bloková struktura
- Plochá bloková struktura
- Vkládaná bloková struktura

Monolitická bloková struktura

Tato struktura je nejjednodušší, proto je také snadná implementace tabulky identifikátorů. V této struktuře je programem pouze jeden velký blok. Všechny proměnné jsou tedy jen globální.

Typická pravidla pro monolitickou blokovou strukturu:

- Žádný identifikátor nemůže být deklarován vícekrát než jednou.
- Pro každý výskyt identifikátoru *I* musí existovat korespondující deklarace identifikátoru *I*.

Příklad části programu v monolitické blokové struktuře:

```
program
« integer b = 10
¬ integer n
- char c
   …
  n = n * b
   …
  write c
   …
end
```

Odpovídající tabulka symbolů:

Identifikátor (id)	Atributy (attr)
b	«
n	¬
c	-

Poznámka: tabulka identifikátorů obsahuje 2 části. V první části je uložen název identifikátoru a v druhé části atributy. Zde jsou uloženy potřebné informace o identifikátoru (např. jeho typ).

Implementace tabulky pro monolitickou strukturu:

Pro práci s tabulkou jsou obvykle implementovány následující procedury:

```
procedure startIdentifikation;
```

- Procedura vytvoří prázdnou tabulku.

```
procedure enter(id: Token;attr: Attribute);
```

- Procedura vloží do tabulky nový identifikátor id s atributem attr.

```
procedure retrieve(id: Token; var found: Boolean
                               var attr: Attribute);
```

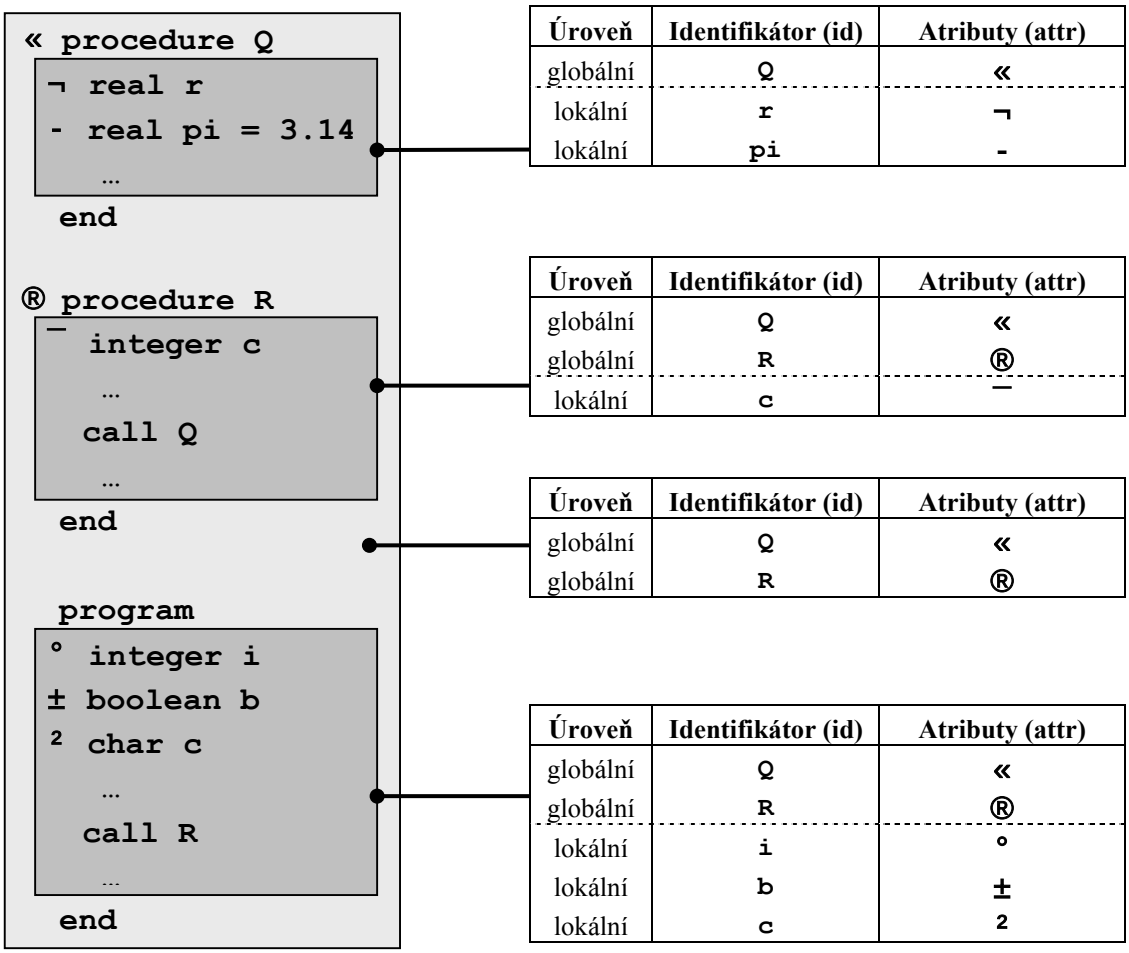
- Procedura vyhledá v tabulce identifikátor id. Pokud je nalezen, found = true a proměnná attr obsahuje atributy daného identifikátoru.

Plochá bloková struktura

Tato struktura obsahuje 2 úrovně zanoření bloku:

- Nějaké deklarace mohou být na globální úrovni. Tyto identifikátory mohou být použity kdekoliv v programu.
- Jiné deklarace mohou být na lokální úrovni. Tyto identifikátory mohou být pouze použity v rámci lokálního bloku.

Příklad části programu v ploché blokové struktuře:



Úroveň	Identifikátor (id)	Atributy (attr)
globální	Q	«
lokální	r	¬
lokální	pi	-

Úroveň	Identifikátor (id)	Atributy (attr)
globální	Q	«
globální	R	®
lokální	c	¯

Úroveň	Identifikátor (id)	Atributy (attr)
globální	Q	«
globální	R	®

Úroveň	Identifikátor (id)	Atributy (attr)
globální	Q	«
globální	R	®
lokální	i	°
lokální	b	±
lokální	c	²

Poznámka: Vedle programu je znázorněno, jak se postupně modifikuje tabulka identifikátorů.

Implementace tabulky pro plochou strukturu:

Pro práci s tabulkou jsou obvykle implementovány stejné procedury jako u monolitické blokové struktury. Navíc ale musíme uchovávat pro jednotlivé proměnné informaci o tom, zda jsou lokální či globální. To může být například vyřešeno přidáním následujícími procedurami:

`procedure openScope;`

- Procedura zařídí, že všechny nové identifikátory budou vkládány jako lokální. Procedura je volána tehdy, když v programu přecházíme do lokálního bloku.

`procedure closeScope;`

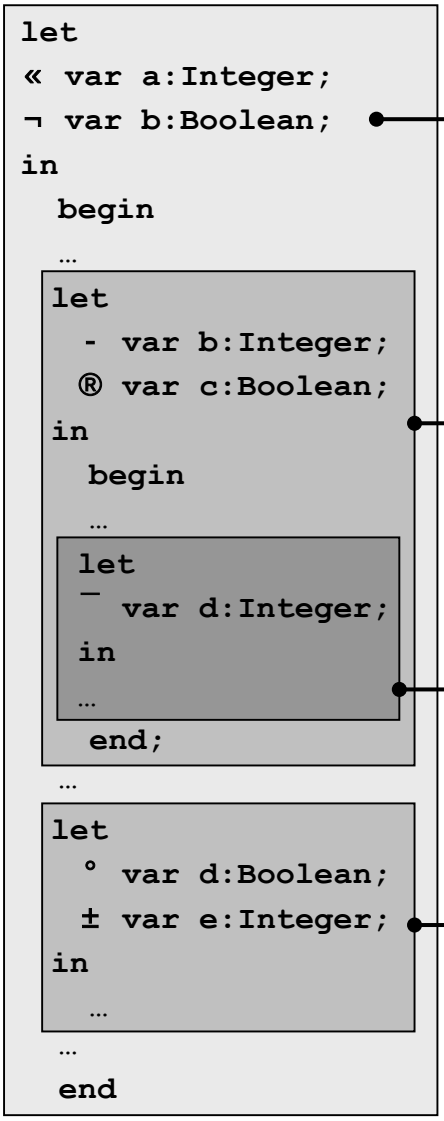
- Procedura zařídí, že všechny nové identifikátory budou vkládány jako globální a navíc jsou z tabulky odstraněny všechny identifikátory lokální. Procedura je volána tehdy, když v programu opouštíme lokální blok.

Poznámka: Tabulka musí být prohledávána zdola nahoru, aby napřed byly nalezeny lokální identifikátory a v případě neúspěchu mohly být prohledány globální identifikátory.

Vkládaná bloková struktura

Tato struktura je zobecněná plochá bloková struktura. Může totiž obsahovat libovolný počet úrovní zanořených bloků.

Příklad části programu ve vkládané blokové struktuře:



Úroveň	Identifikátor (id)	Atributy (attr)
1	a	«
1	b	¬

Úroveň	Identifikátor (id)	Atributy (attr)
1	a	«
1	b	¬
2	b	-
2	c	®

Úroveň	Identifikátor (id)	Atributy (attr)
1	a	«
1	b	¬
2	b	-
2	c	®
3	d	¯

Úroveň	Identifikátor (id)	Atributy (attr)
1	a	«
1	b	¬
2	d	°
2	e	±

Poznámka: Vedle programu je znázorněno, jak se postupně modifikuje tabulka identifikátorů.

Implementace tabulky pro vkládanou strukturu:

Pro práci s tabulkou jsou obvykle implementovány stejné procedury jako u ploché blokové struktury. Rozdíl je ale v činnosti procedur openScope a closeScope. Procedury pouze nenastavují, zda se proměnné budou deklarovat jako globální nebo lokální, ale musí si obecně pamatovat úroveň aktuálního bloku. Při zavolání procedury openScope se toto pořadí zvýší o jedničku, při zavolání procedury closeScope se toto pořadí sníží o jedničku a odstraní se všechny proměnné úrovně, která byla aktuální doposud.

3.2 Kontrola typů

Další kontrola na úrovni kontextové analýzy je kontrola typů. Zde je například kontrolováno, zda jsou prováděny jisté operace se správnými typy, zda jsou výrazy daného typu přiřazovány proměnným stejného typu a podobně.

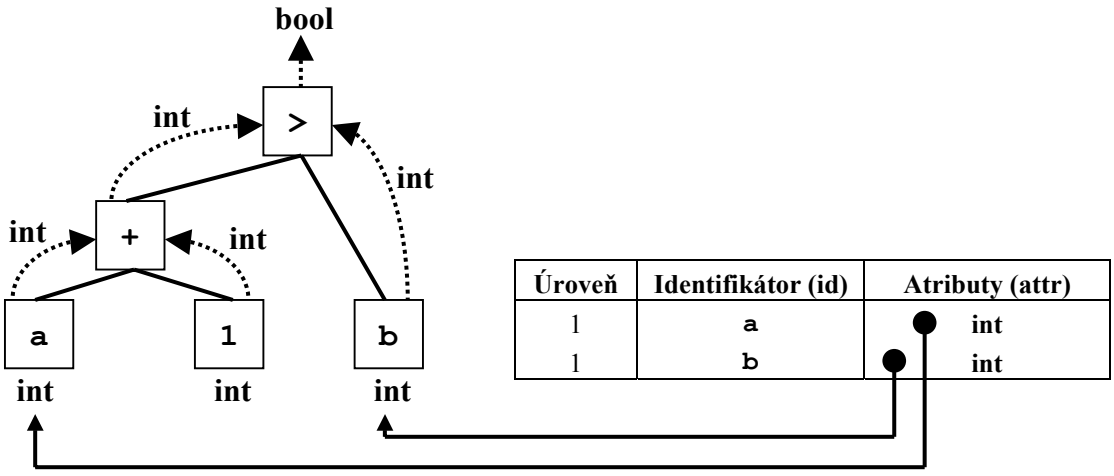
Kontrola typů pro výrazy

Kontrola typů pro výrazy se většinou provádí směrem zdola nahoru:

- Typy listových uzlů jsou známy. Jedná se buď o literály, jejichž typ je znám nebo o výskyt identifikátoru (proměnná, konstanta, …), jehož typ lze zjistit z tabulky identifikátorů.
- Uvažujme výraz tvaru $\oplus E$, kde $\oplus$ je unární operátor typu: $T_1 \rightarrow T_2$. Typová kontrola proběhne tak, že je zkontrolováno, zda E je typu T_1. Pokud ano, výrazu $\oplus E$ je přiřazen typ T_2 a typová kontrola může pokračovat, jinak je hlášena typová chyba.
- Uvažujme výraz tvaru $E_1 \otimes E_2$, kde $\otimes$ je binární operátor typu: $T_1 \times T_2 \rightarrow T_3$. Typová kontrola proběhne tak, že je zkontrolováno, zda E_1 je typu T_1 a E_2 je typu T_2. Pokud ano, výrazu $E_1 \otimes E_2$ je přiřazen typ T_3 a typová kontrola může pokračovat, jinak je hlášena typová chyba.
- Rekurzivně podle výše uvedených pravidel může proběhnout kontrola celého výrazu.

Příklad:

Nechť operátor + je typu: int × int → int, operátor > je typu: int × int → bool. Následující obrázek ilustruje, jak probíhá kontrola typů ve výrazu: $a + 1 > b$:



4 Reference

Tato semestrální práce je výtah z knihy:

- David A. Watt: Programing Language Processors, 1993