Semestrální práce z KIV-FJP

Jméno: Jakub Šlechta (A23N0026P)
Strávený čas: 7-8 dní (viz commit history na gitu)
E-mail: slechta@students.zcu.cz

Cílem mé práce bylo vytvořit repliku jazyka C a vytvořit pro ni překladač. Proto pouze uvedu, jaké odlišnosti má "můj" jazyk oproti jazyku C (viz níže).

Popis řešení

Na vytvoření překladače jsem použil nástroje bison a flex. U nástroje bison jsem využil možnosti reentrantního parseru s podporou pro lokalizaci chyb (ukázka překladu a kompilace souboru se syntaktickou chybou):

```
$ ./build.sh && ./build/compiler.exe data/syntax_error_case.fjp
Flex version required: 2.6.x^
Bison version required: 3.0.x^
Windows build
Build finished
Run with: ./build/compiler.exe <source_file> [-o <output_file>]
Press enter to continue
data/syntax_error_case.fjp:3:10: error: syntax error, unexpected IDENTIFIER,
expecting ';' or ','
error: syntax parsing failed due to unrecognized token
```

Pří syntaktické analýze jsem si sestavil vlastní abstraktní syntaktický strom (AST), nad kterým jsem prováděl zbytek kompilace. Místo atributované gramatiky jsem si tedy zvolil rekurzivní procházení AST stromu, čímž jsem oddělil gramatiku a sémantiku (příklad vytváření AST uzlů pro deklaraci):

```
single_declaration:
    type identifier                 {$$ = new CDeclaration_Node($1, $2,
false);}
    | CONST type identifier         {$$ = new CDeclaration_Node($2, $3, true);}
    | type identifier '=' expression {$$ = new CDeclaration_Node($1, $2, false,
$4);}
    | CONST type identifier '=' expression {$$ = new CDeclaration_Node($2, $3,
true, $5);}
    ;
```

Víceprůchodová kompilace

Překladač je implementován v jazyce C++.

Každý AST uzel dědí od abstraktní třídy `CStatement_Node`:

```
class CStatement_Node
{
    public:
        virtual void Update_Break_Statements(unsigned int address)
        {
            //
        };

        virtual void Update_Continue_Statements(unsigned int address)
        {
            //
        };

        virtual void Update_Return_Type(EData_Type return_type)
        {
            //
        };

        virtual void Compile() = 0;
        virtual ~CStatement_Node() = default;
};
```

Finální metodou je `Compile`, která má zajistit vygenerování do PL/0 pro daný uzel.

Kořenový uzle využívá třídu `CBlock_Node`, která ma za úkol provolat `Compile` metody všech statementů v něm definovaných:

```
void Compile() override
{
    std::cout << "CBlock_Node::Compile()" << std::endl;

    for (statement_list_t::iterator it = mStatement_List->begin(); it !=
mStatement_List->end(); ++it)
    {
        (*it)->Compile();
    }
};
```

a méně abstraktní příklad pro uzel popisující `if-else` statement, kde se už generují instrukce pro PL/0:

```
void Compile() override
{
    std::cout << "CIf_Node::Compile()" << std::endl;

    // LIT(boolean) value onto stack
    mCondition_Node->Compile();

    // save JMC instruction address
    unsigned int jmc_instruction_address = sCode_Length;

    // JMC(ADDR_ELSE_BODY) now 0, ADDR_ELSE_BODY will be set later
    // (if no else statement, ADDR_ELSE_BODY is ADDR_AFTER)
    emit_JMC(0);

    // IF BODY instructions
    branch_compile(mIf_Statement_Node);

    if (mElse_Statement_Node)
    {
        unsigned int jmp_instruction_address = sCode_Length;
                // if else branch provided, we need to jump over it
        // i.e. genereate JMP instruction at the end of IF BODY
        // JMP(ADDR_AFTER) now 0, ADDR_AFTER will be set later
        emit_JMP(0);

        // sCode_Length is ADDR_ELSE_BODY
        unsigned int else_body_address = sCode_Length;

        branch_compile(mElse_Statement_Node);

        modify_param_2(jmc_instruction_address, else_body_address);

        // sCode_Length is ADDR_AFTER
        modify_param_2(jmp_instruction_address, sCode_Length);
    }
    else
    {
        // sCode_Length is ADDR_AFTER
        modify_param_2(jmc_instruction_address, sCode_Length);
    }
};
```

V práci jsem se snažil co nejvíce využívat OOP vlastnosti jazyka C++.

Testování

V kořenovém adresáři odevzdané práce je složka data, ve které jsou připravené *test-cases*.

Např.:

Testování nekonečných smyček

```
int main()
{
    for(;;);
    return 0;
}

// should never end
main();
```

Testování autoamtické konverze

```
int main()
{
    return 3.4;
}

// should push 3 on top of the stack
main();
```

Testovní chyb

```
int main()
{
    return; // missing expression error
}
```

```
// whole program
continue; // outside of branch error
```

Vzorové programy pro překladač

Hanojské věže

```
void print_rod_movement(char disk, char from_rod, char to_rod)
{
    write(NEW_LINE); // new line
    write(' ');
    write('M');
    write('o');
    write('v');
    write('e');
```

```
    write(' ');

    //...
}

void towerOfHanoi(int n, char from_rod, char to_rod, char aux_rod)
{
    if (n == 1)
    {
        print_rod_movement('1', from_rod, to_rod);
        return;
    }
    towerOfHanoi(n-1, from_rod, aux_rod, to_rod);
    print_rod_movement(n + '0', from_rod, to_rod);
    towerOfHanoi(n-1, aux_rod, to_rod, from_rod);
}

int main()
{
    int n;
    // read is not an expression (has syntax as function, but is a keyword...)
    // so for example, you can't do this: n = read(n) - 0;
    read(n);
    n = n - '0';
    if (n < 1)
    {
        write('E');
        write('r');
        write('r');
        write('o');
        write('r');
        write('!');
        return 1;
    }
    towerOfHanoi(n, 'A', 'C', 'B');
    return 0;
}

main();
```

Faktoriál

```
int factorial(int n)
{
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
```

```
}

// should return 120
factorial(5);
```

Komplexní příklad

```
int _switch(int a)
{
    // int to char conversion
    // note: break statements not allowed, they are automatically inserted
    switch (a)
    {
        case '0':
            return 0;
        case '1':
            return 1;
        case '2':
            return 2;
        case '3':
            return 3;
        default:
            return 4;
    }
}

bool _if_else(const char a)
{
    int val = _switch(a);
    if (val < 3) return true;
    else if (val == 3) return false;
    else if (val > 3) return true;
    else return true; // unreachable
}

void _while(char a)
{
    while (_if_else(a)) // fn called as an expression
    {
        a = a + 1;
    }
    write(a); // should print 3

    // void -> return not needed
}

int main()
{
```

```
    for (char a = -5; a < 5; a = a + 1)
    {
        _while(a); // should be executed once and print '3'
        return 0; // we can return even if it's not the last statement
    }
    return -1;
}

main(); // called as an expression, but return value is ignored
```

Implementované funkce jazyka

Základní povinné (10b)

- definice celočíselných proměnných
- definice celočíselných konstant
- přiřazení
- základní aritmetiku a logiku (+, -, *, /, AND, OR, negace a závorky, operátory pro porovnání čísel)
- cyklus (libovolný)
- jednoduchou podmínku (if bez else)
- definice podprogramu (procedura, funkce, metoda) a jeho volání
  10b

Jednoduchá rozšíření (1 bod)

- další typ cyklu - `for`, `do .. while`, `while` (2x 1b)
- else větev
- datový typ boolean a logické operace s ním
- datový typ real (s celočíselnými instrukcemi)
- rozvětvená podmínka (switch, case)
- násobné přiřazení (`a = b = c = d = 3;`)
- příkazy pro vstup a výstup (`read` , `write`)
  8b

Složitější rozšíření (2 body)

- parametry předávané hodnotou
- návratová hodnota podprogramu
  4b

Další rozšíření (? body)

- datový typ `void`

- datový typ `char`
- možnost použít `return` statement kdekoliv v těle funkce
- klíčové slovo `break`
- klíčové slovo `continue`
- multi-deklarace
- možnost použít statement i mimo blok
  - `if (cond) {statement}` vs `if (cond) statement`
- možnost použít funkci jako výraz, ale i jako kdyby měla datový typ void (tzn. jen samotné volání)
  - `int fn(){...}` -> `int a = fn()` vs `fn()` (pouze volání a nevyzvednutí vrácené hodnoty)
- unární plus a mínus
- použití stejných názvů identifikátorů, pokud jsou v odlišném scopu (např. v if-else větvy, funkci, cyklus, ...)
  (?)b

Celkově: 10 + 8 + 4 + (?) = 22b

Odlišnosti oproti C

- `switch` statement má pro každý `case` statement implicitní `break` - při explicitním použití se vyhodí chyba.
- další neimplementované funkcionality jazka C (ukazatele, struktury, pole, string literály, ...)

Struktura projektu

Projekt je realizován jako C++ knihovna.

```
.
├── src/
│   ├── ast/
│   │   ├── nodes/
│   │   │   ├── assignment_node.h
│   │   │   ├── block_node.h
│   │   │   └── ...
│   │   └── ast_nodes.h
│   ├── rules/
│   │   ├── grammar.y
│   │   └── lex.l
│   └── utils
├── data/
│   ├── complex.fjp
```

```
│   ├── complex.pl0
│   ├── echo.fjp
│   └── ...
├── .gitignore
├── build.sh
├── compiler.exe
├── doc.pdf
└── README.md
```

`nodes` - jednotlivé uzly AST stromu
`ast_nodes.h` - hromadný include
`grammar.y` - vstupní soubor pro nástroj bison
`lex.l` - vstupní soubor pro nástroj flex
`utils` - definice struktur, funkcí pro logování, údržbu tabulky symbolů, definice pro PL/0, ...
`data` - *use-cases* vytovřeného jazyka (uvnitř jsou 3 přeložené programy do rozšířené PL/0 - soboury jsou zakončené koncovkou `pl0`)
`build.sh` - skript pro překlad
`compiler.exe` - přeložený program spustitelný v prostředí Windows
`doc.pdf` - elektronická forma této dokumentace

Repozitář

GitHub - JamesAri/fjp